Oracle® Fusion Applications

Developer's Guide for Oracle Enterprise Scheduler

11*g* Release 1 (11.1.1.5)

E10142-01

August 2011

ORACLE®

Oracle Fusion Applications Developer's Guide for Oracle Enterprise Scheduler 11*g* Release 1 (11.1.1.5)

E10142-01

Primary Author: Thomas Van Raalte

Contributors: Kirk Bittler, Weifeng Bao, Shelly Butcher, David Craft, Diane Davison, Carlos Fuentes, Charles Hall, Vaibhav Lole, Solomon Nelson, Shengsong Ni, Rachna Shukla, Steven Traut, Venkat Vengala, Aaron Weisberg

Contents

4 Using the Metadata Service

5 Using Parameters and System Properties

6 Creating and Using PL/SQL Jobs

7 Creating and Using Process Jobs

8 Defining and Using Schedules

9 Working with Extensions to Oracle Enterprise Scheduler

10 Using the Oracle Enterprise Scheduler Web Service

11 Defining and Using Job Sets

12 Defining and Using a Job Incompatibility

13 Using the Runtime Service

14 Using Subrequests

15 Working with Asynchronous Java Jobs

16 Oracle Enterprise Scheduler Security

17 Managing Business and System Errors

Preface

Oracle Enterprise Scheduler provides the ability to run different job types, including: Java, PL/SQL, and binary scripts, distributed across the nodes in an Oracle WebLogic Server cluster. Oracle Enterprise Scheduler runs these jobs securely, with high availability and scalability, with load balancing and provides monitoring and management through Oracle Enterprise Manager Fusion Applications Control.

Audience

This document is intended for Oracle applications developers and assumes familiarity with Java and SQL.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these web sites.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see the following documents in the Oracle 11g Fusion Middleware documentation set:

- *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*
- *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*
- *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*
- *Oracle Fusion Middleware Application Security Guide*
- *Oracle Fusion Applications Administrator's Guide*

  The following chapters in this guide describe Oracle Enterprise Scheduler administrative functions:

  – "Managing Oracle Enterprise Scheduler Service and Jobs"
  – "Troubleshooting Oracle Enterprise Scheduler"
  – "High Availability for Oracle Enterprise Scheduler"

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction to Oracle Enterprise Scheduler

This chapter includes the following sections:

1.1 About Oracle Enterprise Scheduler

Enterprise applications require the ability to respond to many real-time transactions requested by online users or web services. However, they also require the ability to offload larger transactions to run at a future time or automate the running of application maintenance work based on a defined schedule.

Oracle Enterprise Scheduler provides the ability to run different job types, including: Java, PL/SQL, and binary scripts, distributed across the nodes in an Oracle WebLogic Server cluster. Oracle Enterprise Scheduler runs these jobs securely, with high availability and scalability, with load balancing and provides monitoring and management through Fusion Applications Control.

Oracle Enterprise Scheduler provides scheduling services for the following purposes:

- To distribute job request processing across a grid of application servers,
- To run Java, PL/SQL and binary process jobs,
- To group job requests into job sets,
- To schedule job requests based on recurrence expressions,
- To administer job requests with Fusion Applications Control.

Oracle Enterprise Scheduler provides the critical requirements in a service-oriented environment to automate processes that must recur on a scheduled basis and to defer heavy processing to specific time windows. Oracle Enterprise Scheduler lets you:

- Support sophisticated scheduling and workload management,
- Automate the running of administrative jobs,
- Schedule the creation and distribution of reports,
- Schedule a future time for a step in a business flow for business process management.

Oracle Enterprise Scheduler provides features to manage the complete life cycle of a job definition: development, distribution, scheduling, and monitoring. Using Oracle JDeveloper, application developers can easily create job requests in their development

environment. Application administrators and other users can specify when and where they want their job requests to run. Users and administrators can monitor how the job ran and access the end results of those jobs.

Customers that implement large systems typically have to manage a large number of diverse machines to handle the workload of their users. Oracle Enterprise Scheduler provides the ability to control how work is distributed to individual machines or groups of machines.

1.2 Oracle Enterprise Scheduler Overview for Application Developers

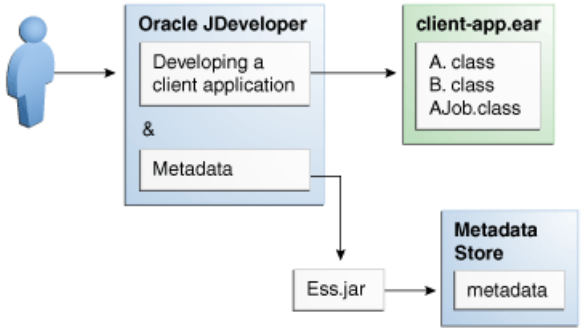
Oracle Enterprise Scheduler is primarily a Java EE application that provides time and schedule based callbacks to other applications to run their jobs. Oracle Enterprise Scheduler compares with the Calendar application you might use in your phone or the Oracle Calendar, where you create events and meetings with details about time and recurrence; the application sends an alarm or notification at the right time for the particular event. Similarly, Oracle Enterprise Scheduler applications define jobs and specify when those jobs need to be executed, and Oracle Enterprise Scheduler gives these applications a callback when that time or when a particular event arrives. This is a simplified model of how a particular application can interact with an instance of Oracle Enterprise Scheduler. Oracle Enterprise Scheduler does not execute the jobs itself, it gives a callback to the application and the application actually executes the job request. This implies that Oracle Enterprise Scheduler is not aware of the details of the job request, all the job request details are owned and consumed by the application. An application that submits requests to run a job is called a *client application*.

For development purposes, both Oracle Enterprise Scheduler and the Oracle Enterprise Scheduler client application are deployed on the same Oracle WebLogic Server. Oracle Enterprise Scheduler does not provide a direct interface for the end user. The end user interacts with the client application which decides what the interface should be and how it is experienced; the client application interacts with Oracle Enterprise Scheduler to setup a job request and to specify when the job request is scheduled to be executed, and eventually gets a callback from Oracle Enterprise Scheduler when the time or event arrives.

1.2.1 Introduction to Working with Oracle Enterprise Scheduler at Design-time

At design time an application developer uses Oracle JDeveloper to create a Java EE application that contains the Oracle Enterprise Scheduler executable class and Oracle Enterprise Scheduler specific metadata for this executable. The Oracle Enterprise Scheduler metadata consists of job definitions, including the executable class and parameters, and schedules. Schedules capture the times when a job request can be sent for execution. Schedules are defined independent of job requests and get associated with job requests at runtime when the job request is submitted for execution. Figure 1–1 shows the design time view of an Oracle Enterprise Scheduler application.

Figure 1–1 Oracle Enterprise Scheduler Design Time Integration



In Figure 1–1, although the metadata is written to the MDS store through Oracle Enterprise Scheduler APIs, the client application owns the metadata and the metadata does not belong to the Oracle Enterprise Scheduler application. This metadata together with the job implementation is packaged in an OAR, including the EAR for the application and the MAR containing the metadata; this is deployed in the runtime environment.

The following types of Oracle Enterprise Scheduler Metadata can be created at design time. The metadata is called *Oracle Enterprise Scheduler Metadata* because it is used for scheduling and not because it is owned by Oracle Enterprise Scheduler, it is actually owned by and packaged with the client application.

- **Job type**: This is a basic definition of what a job would be comprised of and defines the following:
  a. The type of job to be run, such as Java, PL/SQL, C, binary script, and so on.
  b. The Java executable class if the job is of Java type, or the PL/SQL function if the job is of PL/SQL type, or the script if the job is of Script type.
  c. Parameters definitions for the job and their data type, and default values.
- **Job definition**: A job definition, or job, is the smallest unit of work which gets performed in context of the client application. It is defined by an underlying job type and any parameters additional to the ones defined in the job type.
- **Job set**: A job set is a sequential or parallel set of job steps, where a job step can be a single job or another job set. A job set and each of its job set steps can have additional parameters, the value for which will be provided when the job or job set is submitted as a job request.
- **Schedule**: A job schedule is a predefined time or a recurrence for a period of time or indefinite. Schedules are defined independent of jobs but are associated with one or more jobs at run time when a job request is submitted.
- **Incompatibility**: An Incompatibility lets you specify job definitions and job sets that cannot run at the same time.

1.2.2 Introduction to Working with Oracle Enterprise Scheduler at Runtime

At run time an application user associates a Schedule with the job to be submitted and provides values for the job parameters. This information is then submitted as a job request. Once Oracle Enterprise Scheduler receives a job request it determines the right time to execute the job request, and at that time sends a message to the owning client

application. The client application then executes the job based on the job metadata and run time values for the parameters.

Figure 1–2 Oracle Enterprise Scheduler Runtime Integration

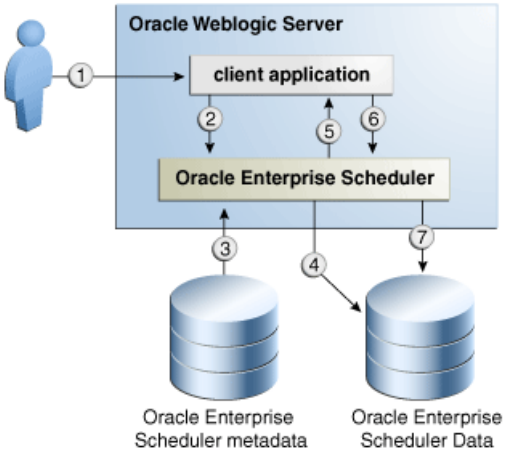


Figure 1–2 shows the sequence involved with running an application using Oracle Enterprise Scheduler, and the following steps:

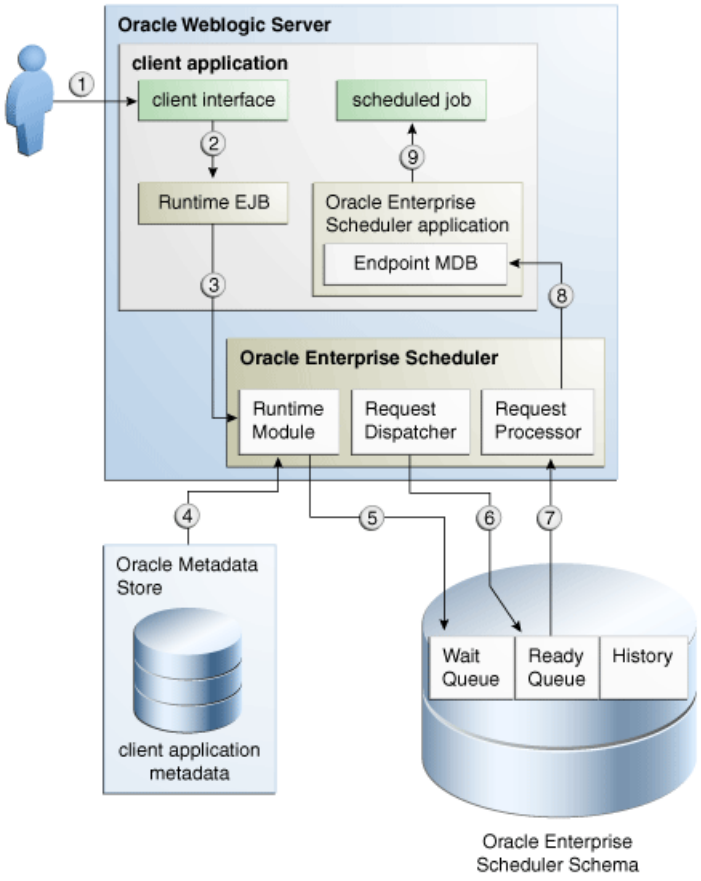
1. User submits a request using a client application.
2. Client application sends the request to Oracle Enterprise Scheduler.
3. Oracle Enterprise Scheduler reads the metadata for the request.
4. Oracle Enterprise Scheduler puts the request in a wait queue in Oracle Enterprise Scheduler data store, along with the metadata.
5. At the appropriate time, according to the request specifics, Oracle Enterprise Scheduler sends a message to the client application with all the request parameters and metadata captured at the time of submission.
6. Client application performs the jobs and returns a status.
7. Oracle Enterprise Scheduler updates the history with the job request status.

1.2.3 Oracle Enterprise Scheduler Job Requests

Figure 1–3 shows the important Oracle Enterprise Scheduler components, including the following:

- The scheduler component itself, including the runtime module, request dispatcher and request processor.
- The client application, including the run time EJB and end point Message-Driven-Bean (MDB) which it calls and the job it requests to execute.
- Oracle Metadata Store and the client application metadata.
- Oracle Enterprise Scheduler schema, including the wait and ready queues and job history.

Figure 1–3 Oracle Enterprise Scheduler Runtime Details



As shown in Figure 1–3, a client application is composed and runs as follows:

1. A user interacts with the client application, submitting a job request.
2. The client application specifies the two EJBs and the Endpoint MDB in its ejb-jar.xml. These beans are then instantiated in the client application context.
3. The beans in the application context contact the underlying Oracle Enterprise Scheduler modules. The run time EJB sends the job request to the underlying run time module in Oracle Enterprise Scheduler.
4. The run time module accesses the client application metadata from Oracle MDS.
5. The run time module persists the request along with its metadata and schedule in the wait queue in the Oracle Enterprise Scheduler schema.
6. The Oracle Enterprise Scheduler request dispatcher determines the correct time to run the job request based on its corresponding schedule. At this time, the request dispatcher moves the request to a ready queue in Oracle Enterprise Scheduler schema.
7. The Oracle Enterprise Scheduler request processor continues picking up job requests to be processed from the ready queue.
8. The request processor sends a message to the application using the endpoint MDB.
9. Oracle Enterprise Scheduler executes the scheduled job.

In most cases or at least in the simplified case, this application will be the same as the application which submitted the request.

1.2.4 Overview of Integration Steps

Once you have installed a basic Oracle WebLogic Server instance, take the following steps to setup Oracle Enterprise Scheduler.

1. Configure Oracle Enterprise Scheduler.
2. Develop your client application which has your job definitions and other required metadata.
3. Deploy your client application.
4. Invoke your client application to submit job request, which in turn calls Oracle Enterprise Scheduler.
5. Invoke your client application to check the status of job request, or other history, which in turn calls Oracle Enterprise Scheduler. Alternatively, use Fusion Applications Control to check the status of a given job request.

1.3 Fixed-Rate Scheduling with Oracle Enterprise Scheduler

Oracle Enterprise Scheduler supports *fixed-rate scheduling* where instances of a repeating job requests are executed at a constant rate starting from the initial scheduled execution time. Each job request runs as near to the absolute time of the schedule as possible. Oracle Enterprise Scheduler ensures that only one job request in a repeating request is running at any one time. If a job request runs beyond the scheduled execution time of the next job request, the next job request becomes late and is dispatched immediately upon completion of the previous job request.

When a job request is dispatched, the next request is placed in the wait queue. The execution time for the next job request is the next time in the schedule that is no earlier than the current time. Oracle Enterprise Scheduler skips time slots that are in the past.

If the desired behavior is to run all instances of the repeating request regardless of when they are run and regardless of the requested or recurrence end date, the request must set the system property `EXECUTE_PAST`.

Oracle Enterprise Scheduler does not support *fixed-delay scheduling*. Using fixed-delay scheduling, each request is executed a fixed delay period after the previous request completes. This means that when one request is late, all subsequent requests will be late as well. In contrast, fixed-rate scheduling tries to get things back on schedule after a late request.

2

Verifying the Oracle Enterprise Scheduler Installation

This chapter describes how to ensure that Oracle Enterprise Scheduler has been correctly installed.

This chapter includes the following sections:

2.1 Introduction to Verifying the Oracle Enterprise Scheduler Installation

The Oracle Enterprise Scheduler health check enables verifying the Oracle Enterprise Scheduler installation using a web browser. The health check web page submits a simple scheduled job so as to verify that Oracle Enterprise Scheduler works as it should.

2.2 How to Verify the Oracle Enterprise Scheduler Installation Using a Browser

Access the Java health check servlet in a web browser. Access to the health check page is available only to users with administrator privileges.

To verify the Oracle Enterprise Scheduler installation:

1. In a web browser, enter the following URL:

   `http://<hostName>:<port>/EssHealthCheck/checkHealth.jsp`

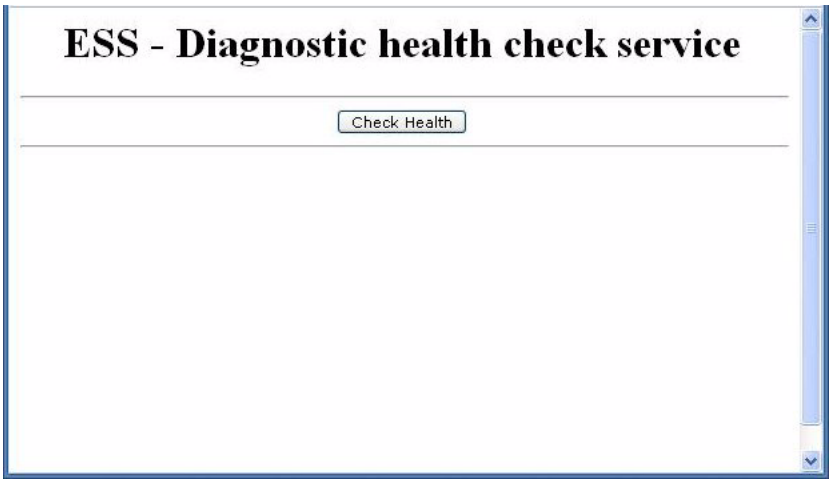
   where `hostName` is the server to which Oracle Enterprise Scheduler is installed and `port` is the port number.

   To verify an Oracle Enterprise Scheduler cluster, use the following URL:

   `http://<hostName>:<port>/EssHealthCheck/diagnoseHealth.jsp`

The Oracle Enterprise Scheduler Diagnostic Health Check page displays, as shown in Figure 2–1.

Figure 2–1 Diagnostic Health Check Page



2. Login to the diagnostic servlet using an Oracle WebLogic Server administrator username and password.
3. Click the **Check Health** button to verify the installation.

2.3 How to Programmatically Verify the Oracle Enterprise Scheduler Installation

Programmatically access the health check servlet from your application. Access to the health check page is available only to users with administrator privileges.

To programmatically verify the Oracle Enterprise Scheduler installation:

1. Access the following URL:

   `http://<hostName>:<port>/EssHealthCheck/checkHealth`
   where `hostName` is the server to which Oracle Enterprise Scheduler is installed and `port` is the port number.
2. Use the HTTP response codes to gauge the health of the Oracle Enterprise Scheduler installation, as shown in Table 2–1.

Table 2–1 HTTP Response Codes

Response Code	Oracle Enterprise Scheduler Status Code	Comments
200(OK)	Oracle Enterprise Scheduler is up and running.	The test job has been submitted and has succeeded within the default duration.
202(ACCEPTED)	Oracle Enterprise Scheduler is up and running but a delay in processing has occurred. A value of 202 (SC_ACCEPTED) indicates to the client that the request is being acted upon but processing is not yet complete.	The test job has been submitted but has failed to complete within the default duration.
500 (INTERNAL_SERVER_ ERROR)	The Oracle Enterprise Scheduler installation has errors.	An error has occurred during the submission or execution of the job.

2.4 What Happens When You Verify the Oracle Enterprise Scheduler Installation

The health check mechanism consists of an ESSHealthcheck servlet that extends HttpServlet. The metadata and packaging dependencies are the same as that of the web service approach.

Metadata services are used to retrieve metadata objects such as job type and job definition. The required metadata files are EssHealthcheckJobType.xml and EssHealthcheckJobDefinition.xml. These are packaged as ess-app-meta.mar, which must itself be packaged with the file eas-app.ear. The servlet, archived as ess-health-check.war, accesses the runtime metadata in order to schedule the job.

> **Note:** Make sure to properly configure the file adf-config.xml so as to register all metadata with the repository.

Example 2–1 illustrates the structure of the files ess-app.ear, ess-ejb.jar, and ess-app-meta.mar.

Example 2–1 The Structure of the Health Check Files

```
ESS-APP.EAR
|  |
|  |__APP-INF/classes/META-INF/ESSWebService.wsdl
|__ess-ejb.jar
|__ess-mbeans.war
|__ess-ws.war
|__ess-ra.rar |
|__ess-health-check.war
    |__WEB-INF
         |__web.xml
         |__weblogic.xml
         |__classes/oracle/ess/healthcheck/view/EssHealthcheckServlet.class
         |__classes/oracle/ess/healthcheck/view/EssConsoleServlet.class
         |__classes/oracle/ess/healthcheck/view/EssClusterHealthcheckServlet.class
         |__checkHealth.jsp
         |__diagnoseHealth.jsp
         |__essVersion.jsp
```

```
ESS-EJB.JAR
Along with the existing set of files,
oracle/ess/healthcheck/core/EssHealthcheckJob.class is added to the ess-ejb.jar.

ESS-APP-META.MAR

oracle/as/ess/essapp/internal/WorkAssignment/ESSInternalWA.xml
oracle/as/ess/essapp/internal/Workshift/ESSInternalWS.xml
oracle/as/ess/essapp/healthcheck/Jobs/EssHealthcheckJobDefn.xml
oracle/as/ess/essapp/batchdelete/Jobs/BatchDeleteJob.xml
oracle/as/ess/essapp/healthcheck/JobType/EssHealthcheckJobType.xml
oracle/as/ess/essapp/batchdelete/JobType/BatchDeleteJobType.xml
```

The health check servlet schedules a trivial job with Oracle Enterprise Scheduler as part of an HTTP request. After a few seconds, the servlet calls `RuntimeServiceBean.getRequestState()` to check the status of the job and constructs a response message within the servlet code. The servlet then returns a response indicating the success or failure of the job.

2.5 What Happens at Runtime: How the Oracle Enterprise Scheduler Installation is Verified

The servlet waits for the job to either reach a terminal state, or run for 10 seconds, whichever occurs first.

- If the job reaches a terminal state in less than 10 seconds, the job results in a state of success.
- If the job's terminal state does not change within 10 seconds, the job results in a state of success. However, the job is listed as not having been executed. This is because the system may be overloaded such that executing the job may take some time.
- If any problems occur when submitting or executing the job, the job results in a state of failure.

When checking the health of a single node or cluster, the processor specific to the server where the health check is submitted processes the health check request. This is achieved through a system property called `SYS_requestedProcessor`. For more information about system properties, see the table in the section "Creating or Editing a Job Set" in the chapter "Managing Oracle Enterprise Scheduler Service and Jobs" in *Oracle Fusion Applications Administrator's Guide*.

3

Use Case Oracle Enterprise Scheduler Sample Application

This chapter describes how to create and run an application that uses Oracle Enterprise Scheduler to run job requests and demonstrates how to work with Oracle JDeveloper to create an application using Oracle Enterprise Scheduler. It then shows a variation on the sample application using two split applications — a job submission application, a *submitter*, and a job execution application, *a hosting application.*

This chapter includes the following sections:

3.1 Introduction to the Scheduler Sample Application

The scheduler sample application includes a complete application that you build with Oracle JDeveloper using Oracle Enterprise Scheduler APIs. Oracle Enterprise Scheduler lets you run different types of job requests, including: Java classes, PL/SQL procedures, and process type jobs. To create an application that schedules job requests you need to do the following:

- Create the Java classes, PL/SQL procedures, or executable processes that specify the routine you want to schedule and run with Oracle Enterprise Scheduler.
- Specify Oracle Enterprise Scheduler metadata and the characteristics for job requests.
- Define the Java application that uses Oracle Enterprise Scheduler APIs to specify and submit job requests.

- Assemble and deploy the Java application that uses Oracle Enterprise Scheduler APIs.
- Run the Java application that uses Oracle Enterprise Scheduler APIs.

> **Note:** The instructions in this chapter assume that you are using a new Oracle JDeveloper that you install without previously saved projects or other saved Oracle JDeveloper state. If you have previously used Oracle JDeveloper, some of the instructions may not match the exact steps shown in this chapter, or you may be able to shorten procedures or perform the same action in fewer steps. In some cases Oracle JDeveloper does not show certain dialogs based on your past use of Oracle JDeveloper.

When you use Oracle Enterprise Scheduler the application Metadata is stored with MDS. To use MDS you need to have access to a database with MDS user and schema configured.

3.2 Creating the Application and Projects for Scheduler Sample Application

Using Oracle JDeveloper you create an application and the projects within the application contain the code and support files for the application. To create the scheduler sample application you need to do the following:

- Create an application in Oracle JDeveloper.
- Create a project in Oracle JDeveloper.
- Create the application code that uses the Oracle Enterprise Scheduler APIs. For the scheduler sample application you create the EssDemo servlet in the EssDemoApp application.

3.2.1 How to Create the EssDemoApp Application

To work with Oracle Enterprise Scheduler, you first create an application and a project in Oracle JDeveloper.

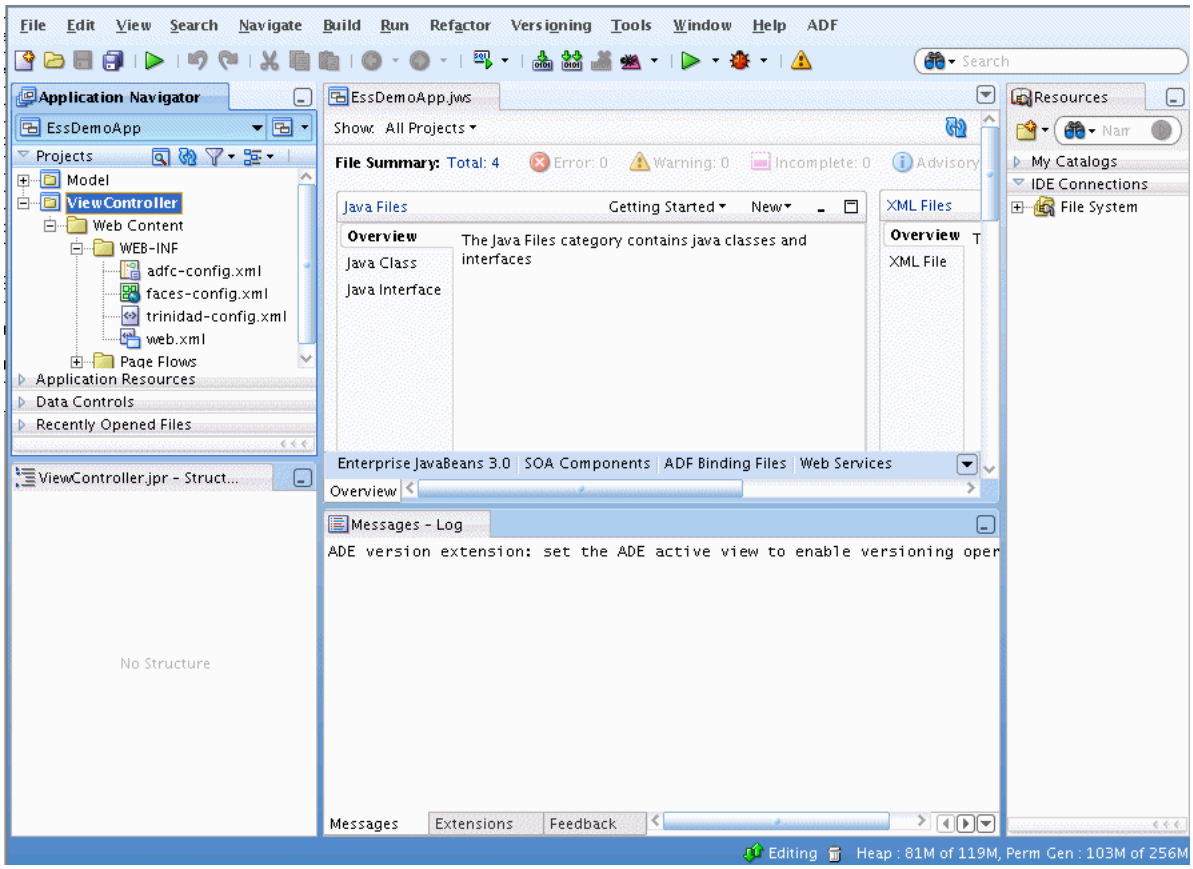
To create the EssDemo application:

1. In the Application Navigator, select **New Application...**.
2. In the Name your application window enter the name and location for the new application.
   a. In the **Application Name** field, enter an application name. For this sample application, enter `EssDemoApp`.
   b. In the **Directory** field, accept the default.
   c. Enter an application package prefix or accept the default, no prefix.

      The prefix, followed by a period, applies to objects created in the initial project of an application.
   d. In the **Application Template** area, select **Fusion Web Application (ADF)**.
   e. Click **Next**.
   f. Click **Finish**.

3. This displays the File Summary page, as shown in Figure 3–1.

Figure 3–1 Sample Application File Summary Page



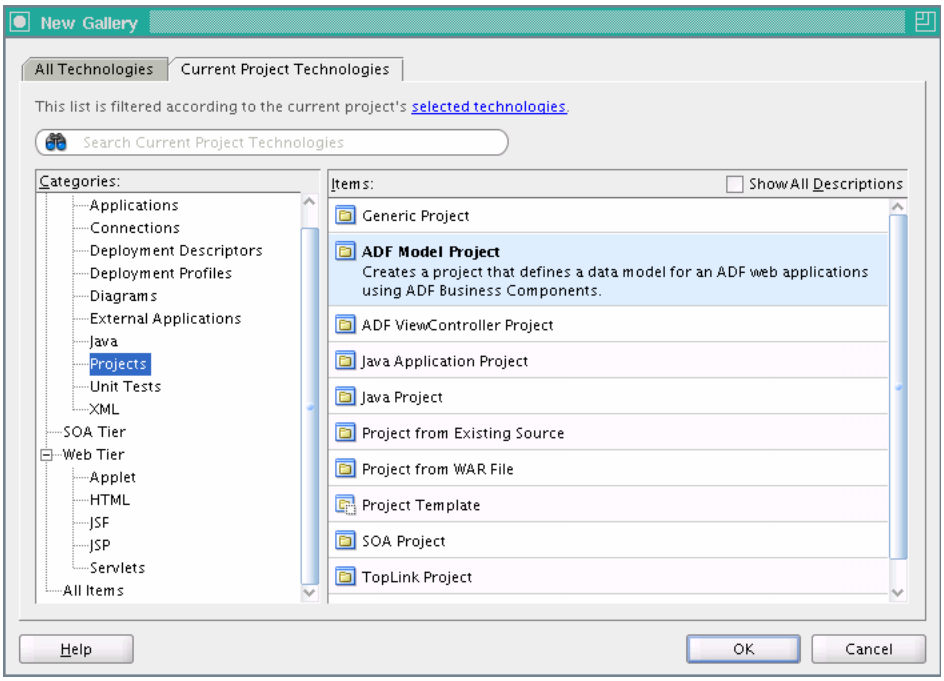
3.2.2 How to Create a Project in the Scheduler Sample Application

When you create an application using the Fusion Web Application (ADF) template, Oracle JDeveloper adds two projects in the application named **Model** and **ViewController** (Oracle ADF is based on the MVC design pattern that includes these areas). To organize an Oracle Enterprise Scheduler application you add another project and use this project to add the Oracle Enterprise Scheduler metadata and the Oracle Enterprise Scheduler implementation for the Java classes that you want to run with Oracle Enterprise Scheduler.

To create a scheduler project:

1. From the Application Menu for the **EssDemoApp** application select **New Project...**.
2. In the New Gallery, under **Categories** expand **General** and select **Projects**.
3. In the **Items** area select **ADF Model Project**, as shown in Figure 3–2.

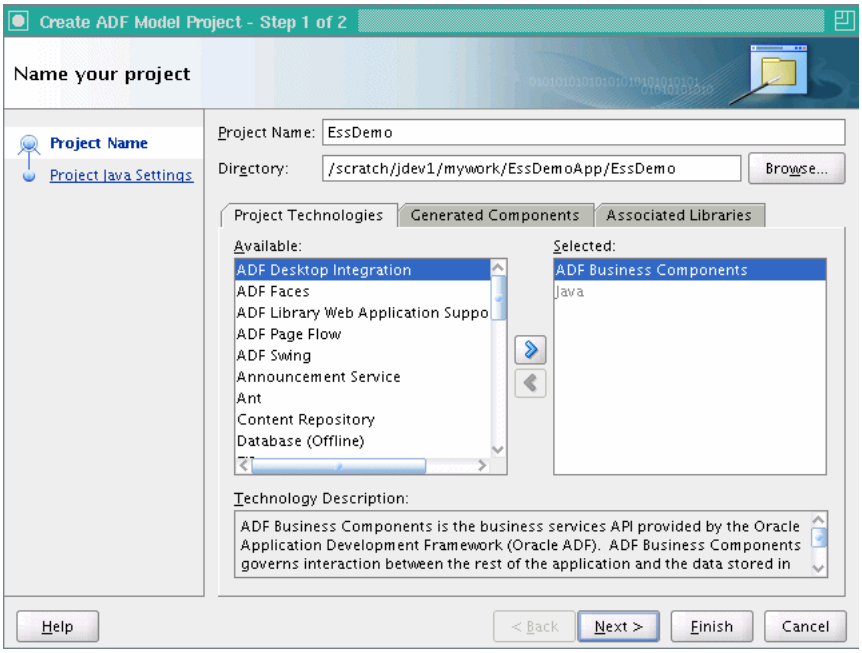
Figure 3–2 Adding an Empty Project for Sample Application



4. Click **OK**.

5. On the Name Your Project page enter a project name. For example, enter `EssDemo` as the project name, as shown in Figure 3–3.

Figure 3–3 Adding the EssDemo Project to the Sample Application

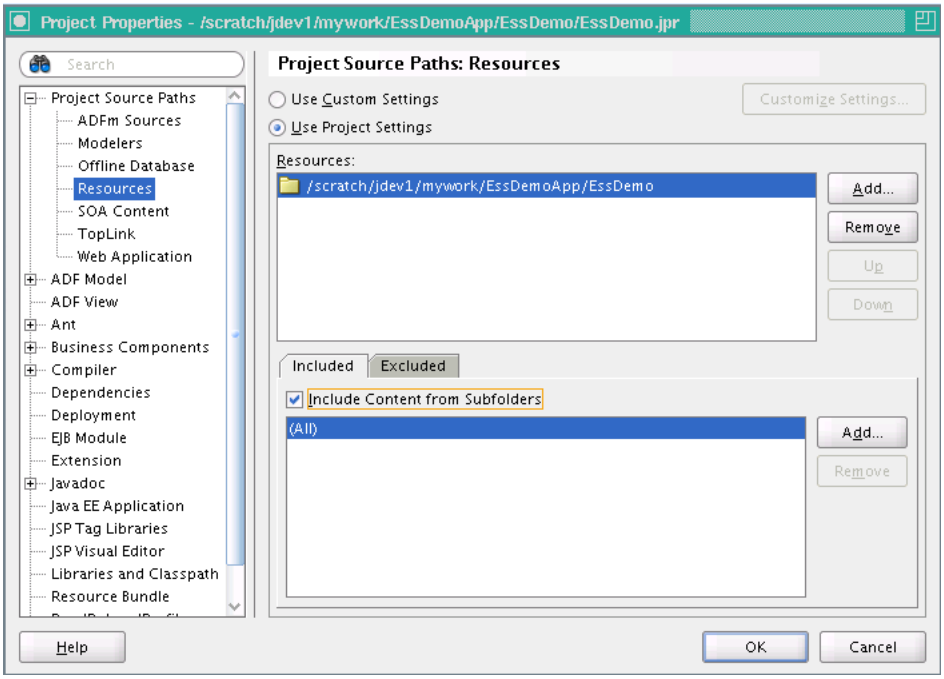


6. Click **Finish**.

Configure Oracle JDeveloper resource options for project:

1. In the Application Navigator, select the **EssDemo** project.
2. Right-click and from the dropdown list select **Project Properties...**.
3. In the Project Properties window, in the navigator expand **Project Source Paths** and select **Resources**.
4. Select the **Included** tab and then select the **Include Content From Subfolders** checkbox, as shown in Figure 3–4.
5. Click **OK**.

Figure 3–4 Updating Project Resources for Sample Project



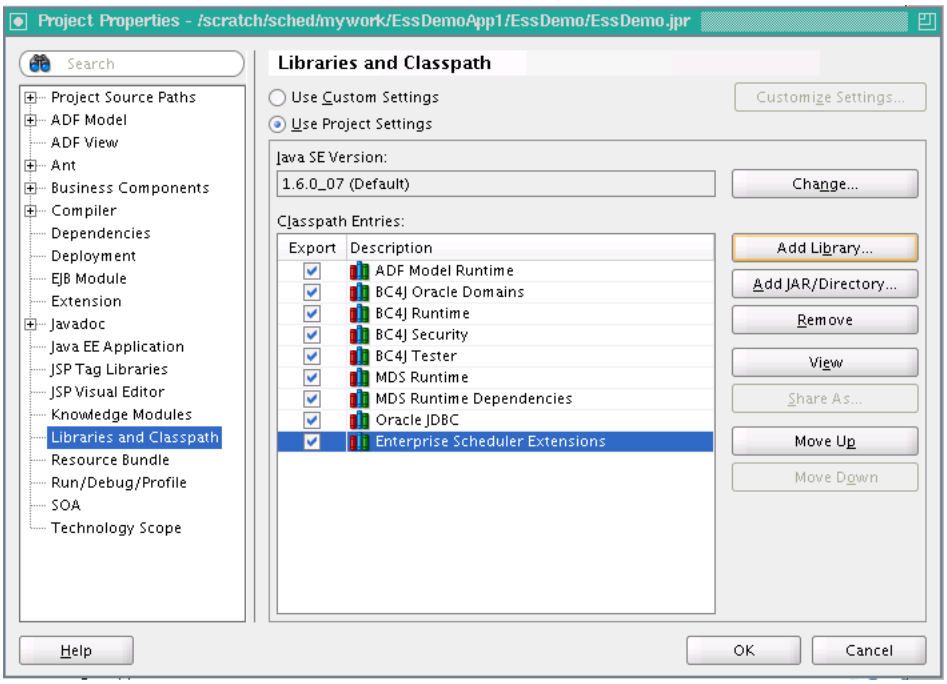
3.2.3 How to Set Project Properties for Enterprise Scheduler

You need to add the Oracle Enterprise Scheduler extensions to the project before you use the Oracle Enterprise Scheduler APIs.

To allow Oracle JDeveloper to use Oracle Enterprise Scheduler extensions:

1. In Oracle JDeveloper, in the Application Navigator select the **EssDemo** project.
2. Right-click and from the dropdown list select **Project Properties...**.
3. In the Project Properties navigator, select **Libraries and Classpath**.
4. In the Libraries and Classpath area, click **Add Library...**.
5. In the Add Library dialog, in the **Libraries** area select **Enterprise Scheduler Extensions**.
6. In the Add Library dialog click **OK**. This adds the appropriate libraries, as shown in Figure 3–5.

Figure 3–5 Adding Oracle Enterprise Scheduler Extensions to Project



7. Click **OK** to dismiss the Project Properties dialog.

3.3 Creating a Java Implementation Class for the Sample Application

To define an application that runs a Java class under control of Oracle Enterprise Scheduler you need to create the Java class that implements the Oracle Enterprise Scheduler `Executable` interface. The `Executable` interface specifies the contract that allows you to use Oracle Enterprise Scheduler to invoke a Java class.

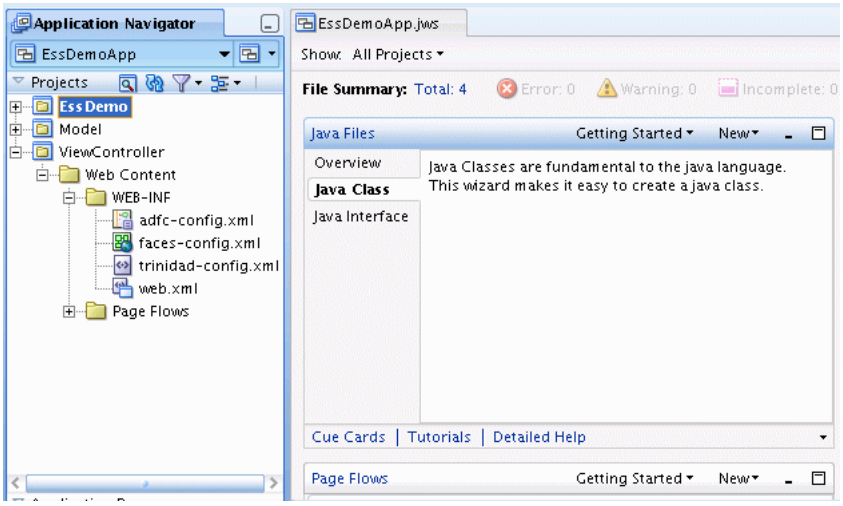
3.3.1 How to Create a Java Class Using the Executable Interface

A Java class that implements the `Executable` interface must provide an empty `execute()` method.

To create a Java class that implements the executable interface:

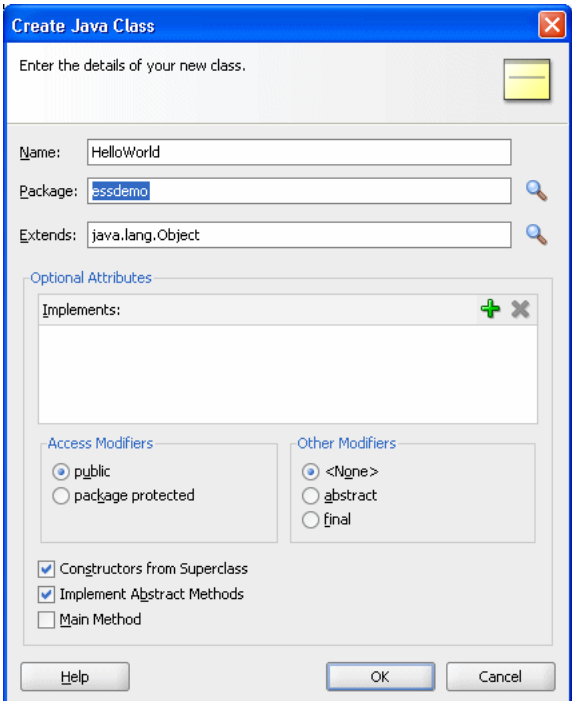
1. In the Application Navigator, select the **EssDemo** project.
2. In the Overview area, select the **Java Class** navigation tab as shown in Figure 3–6.

Figure 3–6 Add a Java Class to the EssDemo Project



3. In the Overview area in the Java Files area, select **New** and from the dropdown list select **Java Class**.
4. In the Select a Project dialog, select the **EssDemo.jpr** project.
5. Click **OK**. This displays the Create Java Class dialog.
6. In the Create Java Class dialog, in the **Name** field, enter `HelloWorld`.
7. In the Create Java Class window, in the **Package** field, enter `essdemo`.
8. In other fields accept the defaults as shown in Figure 3–7.

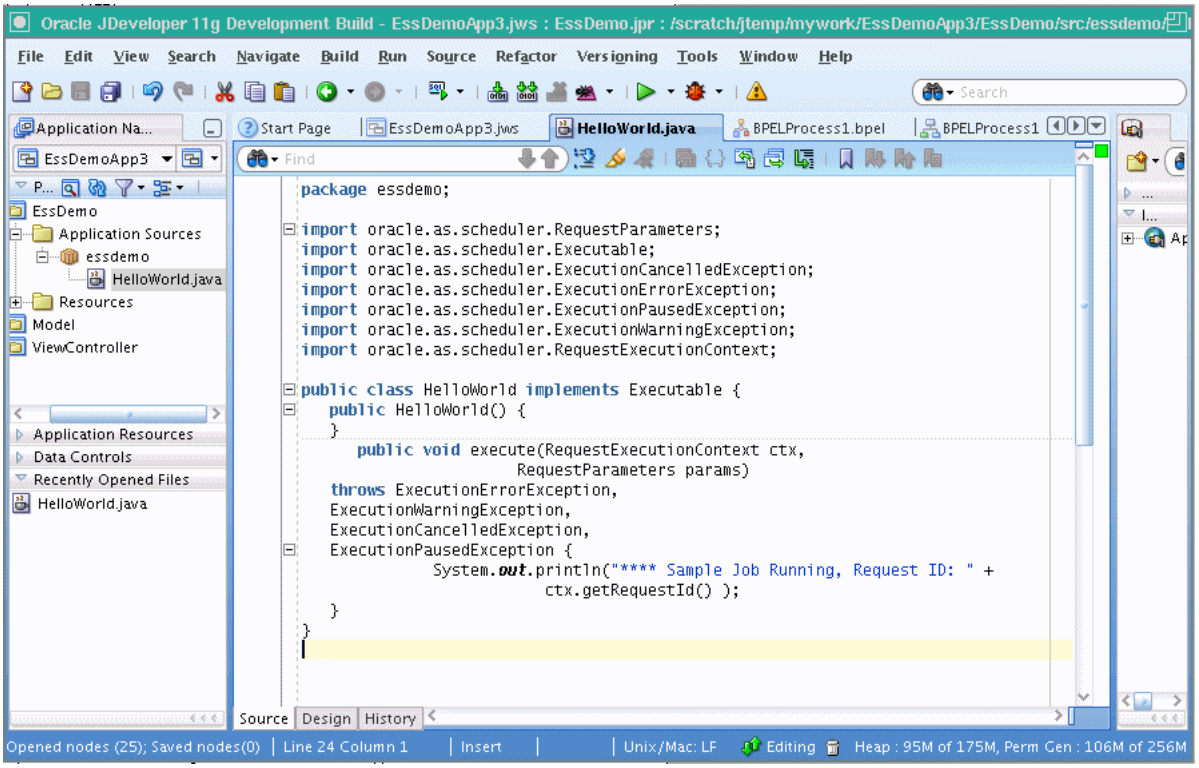
Figure 3–7 Adding a Java Implementation Class to the Sample Application



9. Click **OK**.

10. Replace the generated contents of the `HelloWorld.java` file with the contents of the `HelloWorld.java` supplied with the scheduler sample, as shown in Example 3–1. This code is also shown in Figure 3–8.

Figure 3–8 Java Class That Implements Executable for Sample Application



Example 3–1 shows `HelloWorld()`, the Java class that implements the interface `oracle.as.scheduler.Executable`.

Example 3–1 Oracle Enterprise Scheduler HelloWorld Java Class

```
package essdemo;

import oracle.as.scheduler.RequestParameters;
import oracle.as.scheduler.Executable;
import oracle.as.scheduler.ExecutionCancelledException;
import oracle.as.scheduler.ExecutionErrorException;
import oracle.as.scheduler.ExecutionPausedException;
import oracle.as.scheduler.ExecutionWarningException;
import oracle.as.scheduler.RequestExecutionContext;

public class HelloWorld implements Executable {
    public HelloWorld() {
    }

    public void execute(RequestExecutionContext ctx, RequestParameters params)
        throws ExecutionErrorException,
               ExecutionWarningException,
               ExecutionCancelledException,
               ExecutionPausedException
    {
        System.out.println("**** Sample Job Running, Request ID: " +
                           ctx.getRequestId());
```

```
    }
}
```

3.3.2 What Happens When You Create a Java Class That Implements the Executable Interface

You need to create a Java class to use Oracle Enterprise Scheduler. The Oracle Enterprise Scheduler `Executable` interface provides a hook for using the Java class that you supply with Oracle Enterprise Scheduler. A Java class that implements the `Executable` interface can be submitted to Oracle Enterprise Scheduler for execution.

3.3.3 What You Need to Know About the Executable Interface

When you create a class that implements the `Executable` interface you should follow certain practices to make sure that your code performs correctly. These practices allow you to handle Oracle Enterprise Scheduler exceptions.

> **Note:** Every time a job request executes, Oracle Enterprise Scheduler calls the `execute()` method. All of the business logic associated with a job request should be implemented through this method. Thus, the Java implementation should not rely on instance or static member variables for maintaining state. The Java implementation can use static variables but their use is not recommended to manage state.

In Example 3–1, note the following:

- The routine should throw the `ExecutionErrorException` to signal to the Oracle Enterprise Scheduler runtime that an unrecoverable error occurred during execution. For example, you can wrap your exception generated during execution with this exception. Upon this exception, Oracle Enterprise Scheduler transitions the request to the `ERROR` state.
- The routine should throw the `ExecutionWarningException` when the implementation detects a failure condition that it needs to communicate to Oracle Enterprise Scheduler. Upon this exception, Oracle Enterprise Scheduler transitions the request to the `WARNING` state.
- The routine should throw the `ExecutionCancelledException` when the implementation detects a condition for request cancellation that it needs to communicate to Oracle Enterprise Scheduler. Upon this exception, Oracle Enterprise Scheduler transitions the request to the `CANCELLED` state.
- The routine should throw the `ExecutionPausedException` to indicate that the class implementing the Executable interface should pause for the completion of a subrequest. Upon this exception, Oracle Enterprise Scheduler transitions the request to the `PAUSED` state.

3.4 Adding Application Code to Submit Oracle Enterprise Scheduler Job Requests

In an Oracle Enterprise Scheduler application you use the Oracle Enterprise Scheduler APIs to submit job requests from any component in the application. The `EssDemoApp` sample application provides a Java servlet for a servlet based user interface for submitting job requests (using Oracle Enterprise Scheduler).

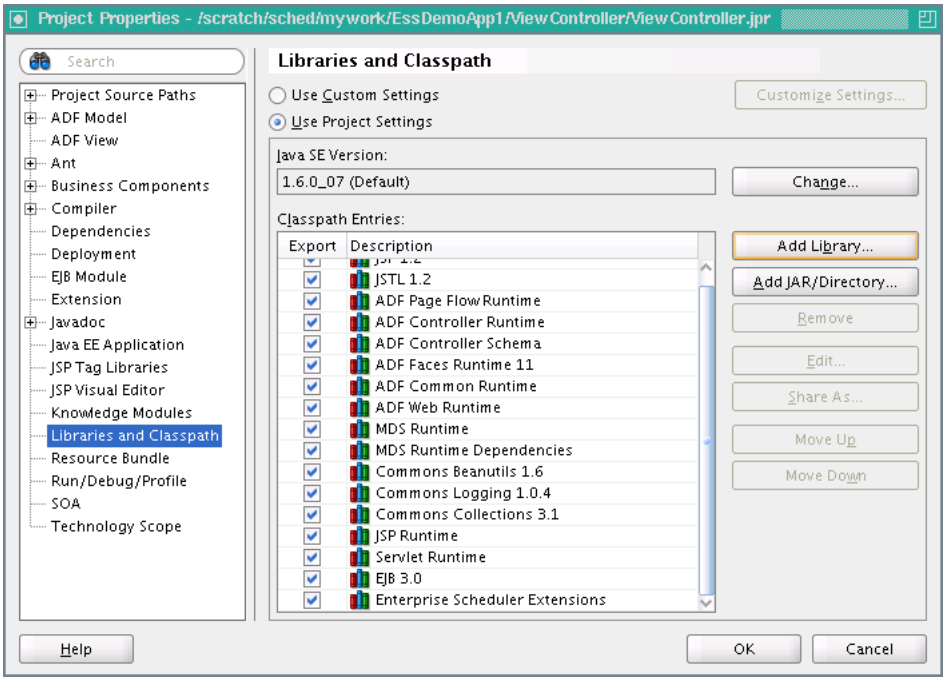
3.4.1 How to Add Required Libraries to Project

You need to add the EJB3.0 libraries and the Oracle Enterprise Scheduler extensions to the ViewController project before you use the Oracle Enterprise Scheduler APIs in a servlet.

To add Oracle JDeveloper EJB3.0 and Enterprise Scheduler libraries:

1. In the Application Navigator select the **ViewController** project.
2. Right-click and from the dropdown list select **Project Properties...**.
3. In the Project Properties navigator, select **Libraries and Classpath**.
4. In the Libraries and Classpath area, click **Add Library...**.
5. In the Add Library dialog select **Enterprise Scheduler Extensions**.
6. In the Add Library dialog also select **EJB 3.0**.
7. Click **OK**. This action adds the libraries as shown in Figure 3–9.

Figure 3–9 Adding Enterprise Scheduler Extensions to ViewController Project



8. Click **OK** to dismiss the Project Properties dialog.

3.4.2 How to Create the EssDemo Servlet

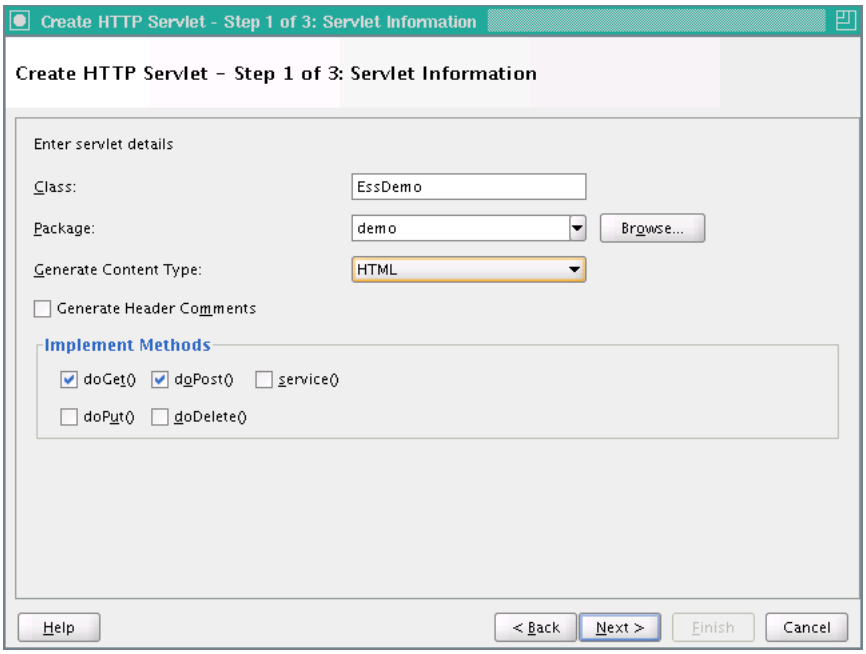
Using MVC design pattern you create the EssDemo servlet in the ViewController project.

To create the sample servlet:

1. In Application Navigator select the **ViewController** project.
2. Click the **New...** icon to open the New Gallery.
3. In the New Gallery, in the **Categories** area expand **Web Tier** and select **Servlets**.

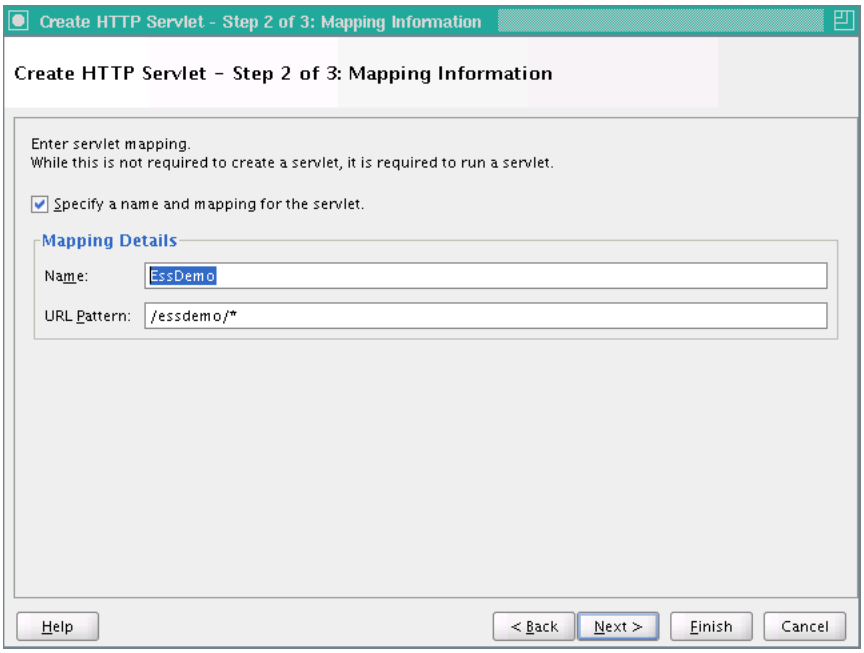
4. In the New Gallery, in the **Items** area select **HTTP Servlet**.
5. Click **OK**. This starts the Create HTTP Servlet Wizard.
6. On the create HTTP Servlet Page - Welcome, click **Next**.
7. On the Create HTTP Servlet - Step 1 of 3: Servlet Information page, enter the class name in the **Class** field. For this example in the **Class** field, enter **EssDemo**.
8. Enter the package name in the **Package** field. For this example, in the **Package** field, enter **demo**.
9. In the **Generate Content Type** field, from the dropdown list select **HTML**.
10. In the **Implement Methods** area, select the **doGet()** and **doPost()** checkboxes, as shown in Figure 3–10.

Figure 3–10 Using the Create HTTP Servlet Wizard to Create the Sample Servlet



11. Click **Next**.
12. In the Create HTTP Servlet - Step 2 of 3: Mapping Information dialog, in the **Name** field, enter: `EssDemo`
13. In the Create HTTP Servlet - Step 2 of 3: Mapping Information dialog, in the **URL Pattern** field, enter: `/essdemo/*`, as shown in Figure 3–11.

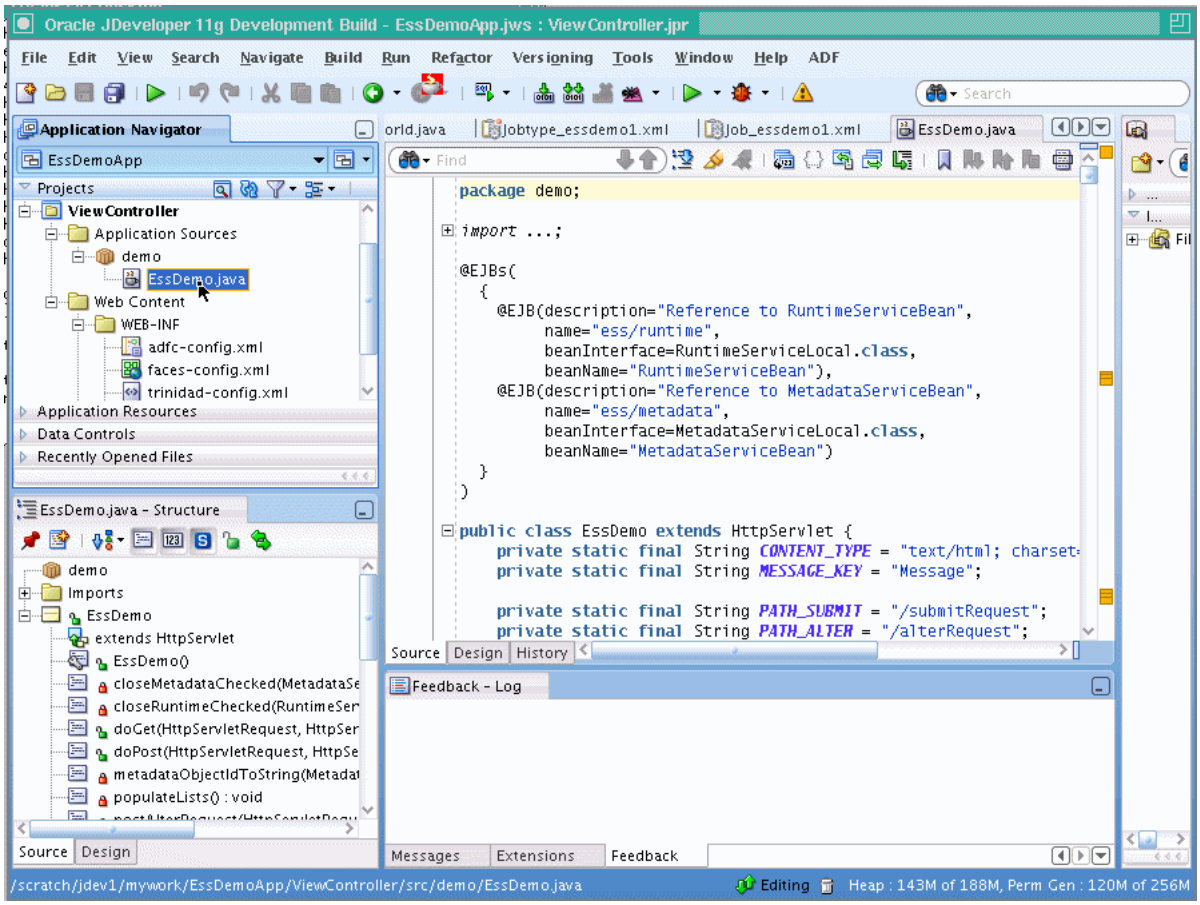
Figure 3–11 Using the Create HTTP Servlet Wizard: Step 2 of 3 Dialog



14. Click **Finish**.

15. The supplied EssDemo application includes the completed servlet. You need to copy the source code into your project. To do this, in Oracle JDeveloper replace the contents of the servlet with the contents of the file `EssDemo.java` supplied with the sample application, as shown in Figure 3–12. The `EssDemo.java` sample code includes several hundred lines, so it is not included in this text in an example.

Figure 3–12 Adding the Sample Servlet to the ViewController Project



3.5 Creating Metadata for Scheduler Sample Application

To use the Oracle Enterprise Scheduler sample application to submit a job request you need to create metadata that defines a job request, including the following:

- A job type: this specifies an execution type and defines a common set of parameters for a job request.
- A job definition: this is the basic unit of work that defines a job request in Oracle Enterprise Scheduler.

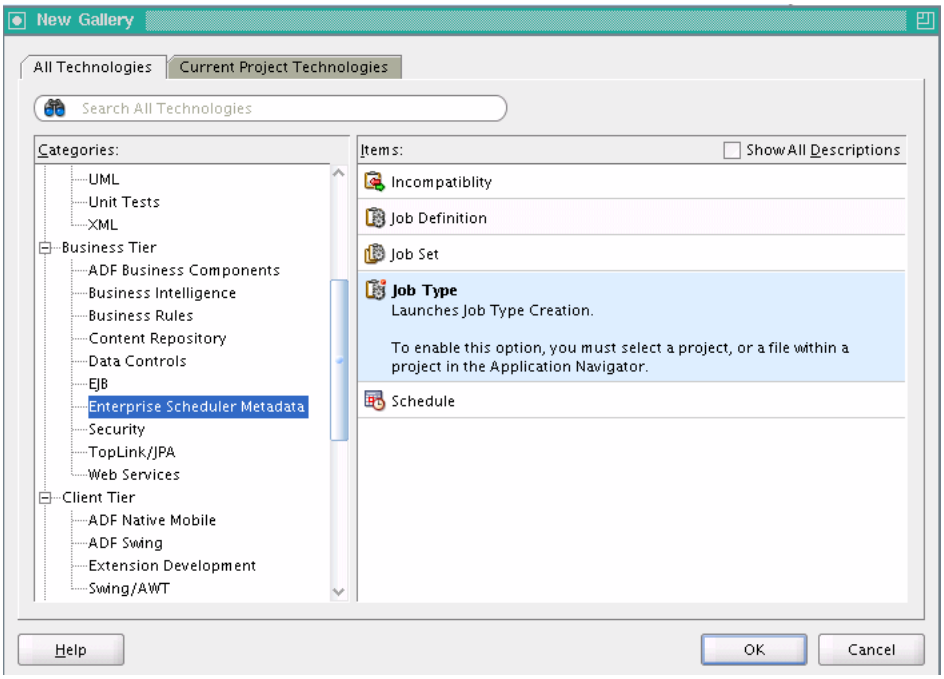
3.5.1 How to Create a Job Type for Java

An Oracle Enterprise Scheduler job type specifies an execution type and defines a common set of parameters for a job request.

To create a job type:

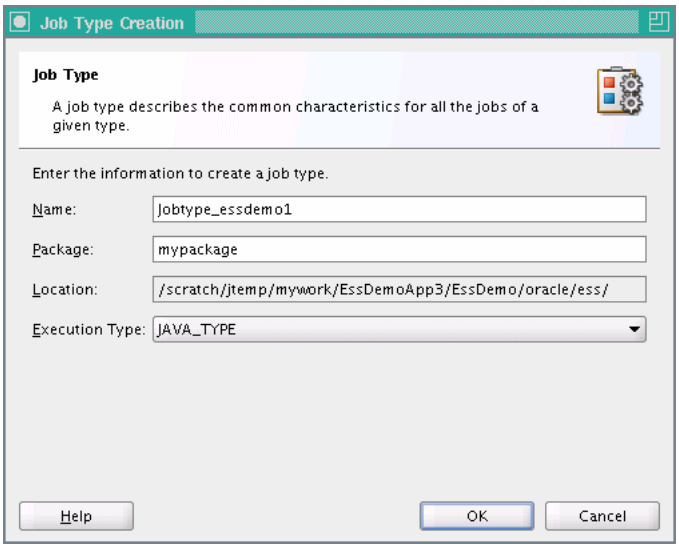
1. In the Application Navigator, select the **EssDemo** project.
2. Press **Ctrl-N**. This displays the New Gallery.
3. In the New Gallery, select the **All Technologies** tab.
4. In the New Gallery, in the **Categories** area expand **Business Tier** and select **Enterprise Scheduler Metadata**.
5. In the New Gallery, in the **Items** area select **Job Type** as shown in Figure 3–13.

Figure 3–13 Adding Job Type Metadata to the Sample Application



6. Click **OK**. This displays the Create Job Type dialog.
7. In the Create Job Type dialog, specify the following:
   a. In the **Name** field, enter a name for the job type. For this example, enter the name: `Jobtype_essdemo1`.
   b. In the **Package** field, enter a package name. For example, enter `mypackage`.
   c. In the **Execution Type** field, from the dropdown list select **JAVA_TYPE** as shown in Figure 3–14.

Figure 3–14 Creating a Job Type with the Job Type Creation Wizard



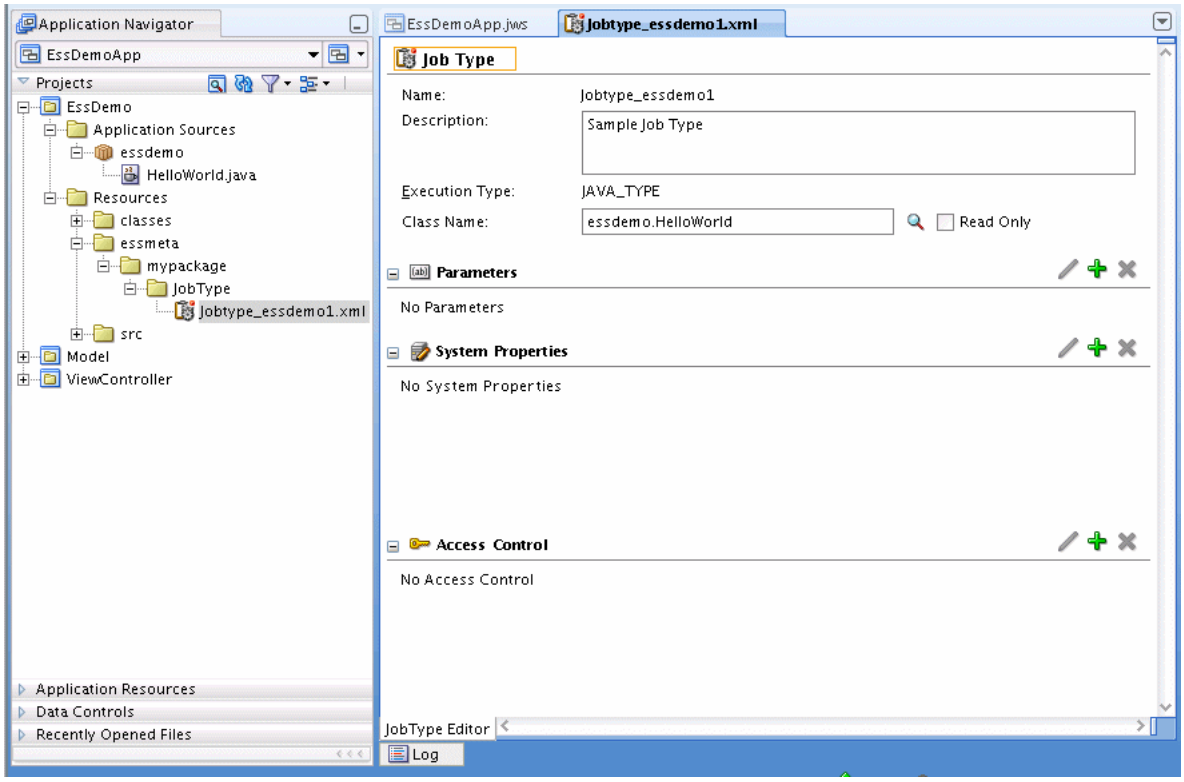
   d. Click **OK**. This creates the `Jobtype_essdemo1.xml` file and Oracle JDeveloper displays the Job Type page.

8. In the Job Type page, in the **Description** field enter a description for the job type. For this example enter: `Sample Java Job Type`.
9. In the **Class Name** field, click the **Browse** icon.
10. Click the **Hierarchy** tab and then navigate to select the appropriate class. For this sample application, select **essdemo.HelloWorld**. Click **OK**.

    The Job Type page displays, as shown in Figure 3–15.

    Tip: You can add the job class at either the job type level or the job definition level.

Figure 3–15 Adding Sample Job Type Metadata



3.5.2 How to Create a Job Definition for Java

To use a Java class with Oracle Enterprise Scheduler you need to create a job definition. A job definition is the basic unit of work that defines a job request in Oracle Enterprise Scheduler.

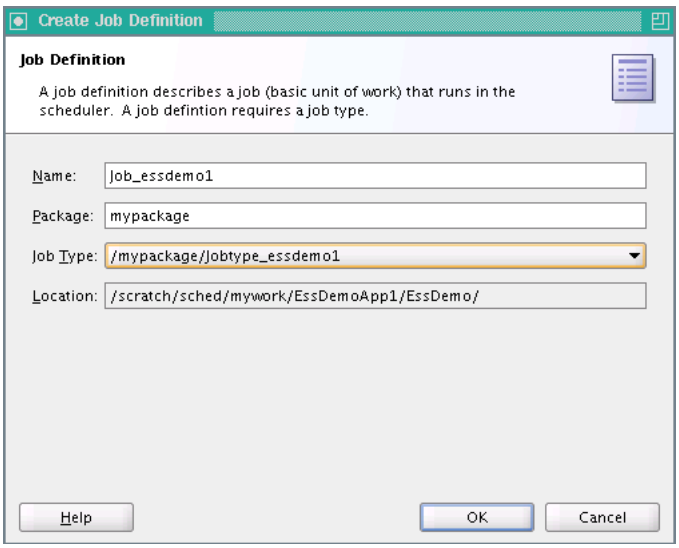
When you create a job definition you specify a name, select a job type, and specify system properties.

To create a job definition:

1. In the Application Navigator, select the **EssDemo** project.
2. Press **Ctrl-N**. This displays the New Gallery.
3. In the New Gallery in the **Categories** area expand **Business Tier** and select **Enterprise Scheduler Metadata**.
4. In the New Gallery in the **Items** area select **Job Definition**.

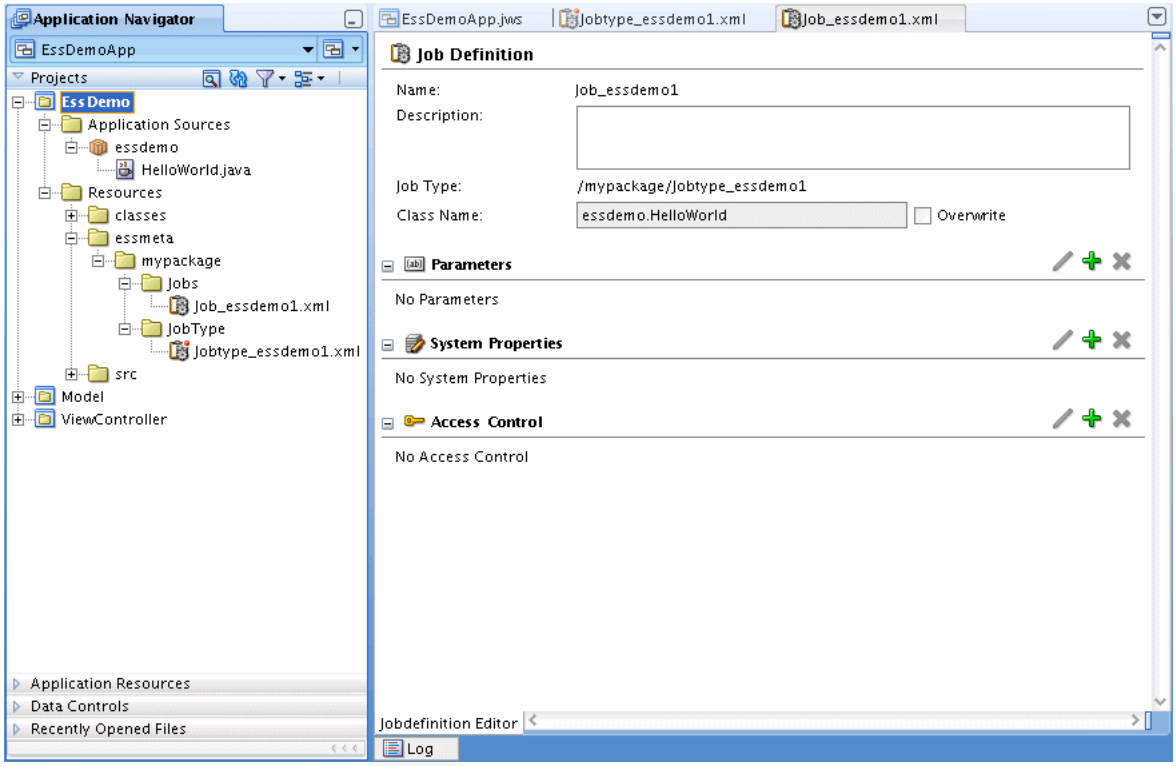
5. Click **OK**. Oracle JDeveloper displays the Create Job Definition dialog.
6. Use the Create Job Definition dialog to specify the following:
   a. Enter a name for the job definition or accept the default name. For example, for the scheduler sample application, enter **Job_essdemo1**.
   b. In the **Package** field, enter a package name. For example, enter `mypackage`.
   c. In the **JobType** field, from the dropdown list select a value. For example for the scheduler sample application select the job type you previously created, **Jobtype_essdemo1**, as shown in Figure 3–16.

Figure 3–16 Using the Job Definition Creation Dialog



   d. Click **OK**. This creates the job definition `Job_essdemo1.xml` and the jobs folder in mypackage and shows the Job Definition page, as shown in Figure 3–17.

Figure 3–17 Job Definition Page for Sample Application



e. In the System Properties field, click the add button and create a system property called `EffectiveApplication`. Set its value to that used in Section 3.6.1, "How to Assemble the EJB Jar Files for Scheduler Sample Application."

3.6 Assembling the Scheduler Sample Application

After you create the scheduler sample application you use Oracle JDeveloper to assemble the application.

To assemble the application you do the following:

- Create the EJB Jar files
- Create the application MAR File
- Create the application EAR file
- Update WAR File options

3.6.1 How to Assemble the EJB Jar Files for Scheduler Sample Application

The sample application needs to contain the required EJB descriptors. You need to create the `ejb-jar.xml` and `weblogic-ejb-jar.xml` files and include these files with any Java implementation class that you create.

Oracle Enterprise Scheduler requires an application to assemble and provide an EJB JAR so that Oracle Enterprise Scheduler can find its entry point in the application while running job requests on behalf of the application. This EJB jar should have its required EJB descriptors in `ejb-jar.xml` and `weblogic-ejb-jar`, as well as any Java class implementations that are going to be submitted to Oracle Enterprise Scheduler.

The descriptor files `ejb-jar.xml` and `weblogic-ejb-jar` must contain descriptions for the Oracle Enterprise Scheduler EJBs and should not be modified.

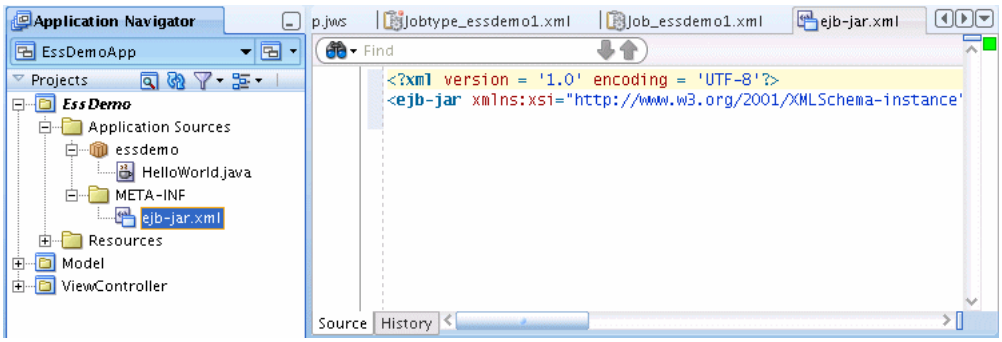
To prepare for the assembly of the scheduler sample application, do the following to add the EJB jar files:

- Create the `ejb-jar.xml` file: this provides the description for the Oracle Enterprise Scheduler EJBs and associated resources. The context of Oracle Enterprise Scheduler request submission, processing, metadata, and runtime data for an application is specified as the name of an Oracle Enterprise Scheduler client application using the deployment name. You can also specify the context using the `applicationName` property, as shown in Example 3–4.
- Create the `weblogic-ejb-jar.xml` file: this provides the Oracle WebLogic Server specific descriptions for the Oracle Enterprise Scheduler EJBs and associated resources.
- Create the EJB JAR archive: this includes descriptors for the Java Job implementations.

To create the ejb-jar.xml file in the Java implementation project:

1. In Application Navigator select the **EssDemo** project.
2. Click the **New...** icon.
3. In the New Gallery, in the navigator expand **General** and select **Deployment Descriptors**.
4. In the New Gallery in the **Items** area select **Java EE Deployment Descriptor**.
5. Click **OK**.
6. In the Select Descriptor page select **ejb-jar.xml**.
7. Click **Next**.
8. In the Select Version page select **3.0**.
9. Click **Finish**.
10. This creates `ejb-jar.xml` file and the `META-INF` directory in the EssDemo project, as shown in Figure 3–18.

Figure 3–18 Adding the ejb-jar.xml File to the Sample Application



11. Replace the entire contents of the `ejb-jar.xml` file that you just created with a copy of the scheduler `ejb-jar.xml` supplied with the scheduler sample application. This sample `ejb-jar.xml` file is shown in Example 3–2.

Example 3–2 EJB Contents to Copy to ejb-jar.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                             http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
         version="3.0">
  <display-name>ESS</display-name>
  <enterprise-beans>
    <message-driven>
      <ejb-name>ESSAppEndpoint</ejb-name>
      <ejb-class>oracle.as.scheduler.ejb.EssAppEndpointBean</ejb-class>
    </message-driven>

    <session>
      <description>Async Request Bean</description>
      <ejb-name>AsyncRequestBean</ejb-name>
      <ejb-class>oracle.as.scheduler.ejb.AsyncRequestBean</ejb-class>
    </session>

    <session>
      <description>Runtime Session Bean</description>
      <ejb-name>RuntimeServiceBean</ejb-name>
      <ejb-class>oracle.as.scheduler.ejb.RuntimeServiceBean</ejb-class>
    </session>

    <session>
      <description>Metadata Session Bean</description>
      <ejb-name>MetadataServiceBean</ejb-name>
      <ejb-class>oracle.as.scheduler.ejb.MetadataServiceBean</ejb-class>
    </session>
  </enterprise-beans>
```

To create the weblogic-ejb-jar.xml file in the Java implementation project:

1. In Application Navigator select the **EssDemo** project.
2. Click **New...** icon.
3. Under **Categories** expand **General** and select **Deployment Descriptors**.
4. In the **Items** area select **Weblogic Deployment Descriptor**.
5. Click **OK**.
6. In the Select Descriptor dialog, select **weblogic-ejb-jar.xml**.
7. Click **Next**.
8. Click **Next**.
9. Click **Finish**. This creates `weblogic-ejb-jar.xml` file.
10. Replace the entire contents of the `weblogic-ejb-jar.xml` file with the sample `weblogic-ejb-jar.xml` supplied with the scheduler sample application. This file is shown in Example 3–3.

Example 3–3 EJB Descriptor Contents to Copy to weblogic-ejb-jar.xml File

```
<?xml version="1.0" encoding="US-ASCII" ?>
<weblogic-ejb-jar xmlns="http://www.bea.com/ns/weblogic/10.0"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="http://www.bea.com/ns/weblogic/10.0
```

```
http://www.bea.com/ns/weblogic/10.0/weblogic-ejb-jar.xsd">
   <weblogic-enterprise-bean>
      <ejb-name>ESSAppEndpoint</ejb-name>
      <message-driven-descriptor>
         <resource-adapter-jndi-name>ess/ra</resource-adapter-jndi-name>
      </message-driven-descriptor>
      <dispatch-policy>ESSRAWM</dispatch-policy>
   </weblogic-enterprise-bean>

   <run-as-role-assignment>
      <role-name>essSystemRole</role-name>
      <run-as-principal-name>weblogic</run-as-principal-name>
   </run-as-role-assignment>
</weblogic-ejb-jar>
```

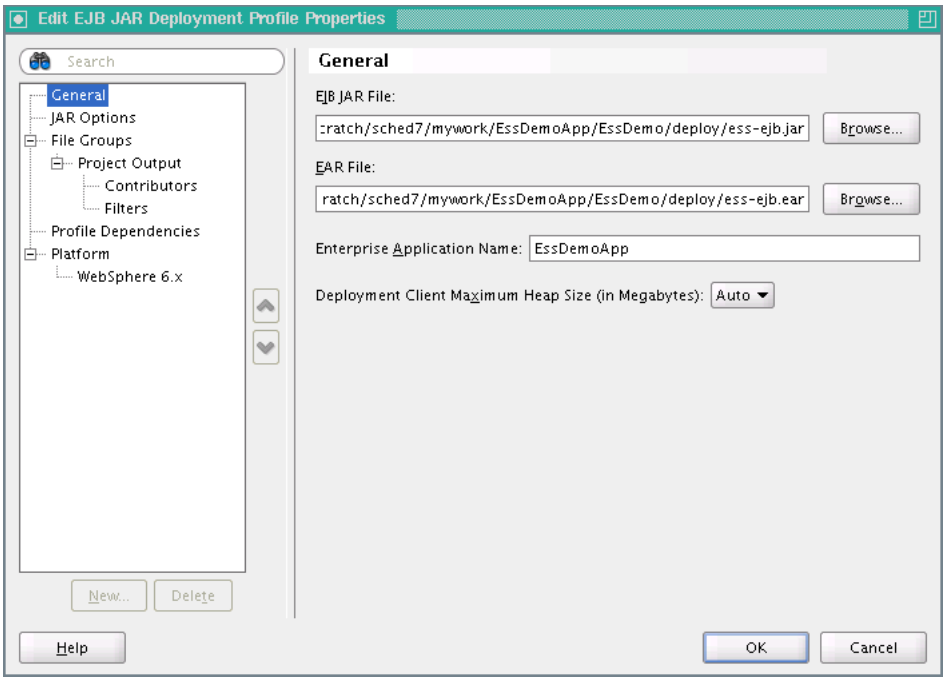
To create the EJB JAR archive:

1. In Application Navigator select the **EssDemo** project.
2. Right-click and from the dropdown list, select **Make EssDemo.jpr**. In the Messages Log you should see a successful compilation message, for example:

   ```
   [3:40:22 PM] Successful compilation: 0 errors, 0 warnings.
   ```

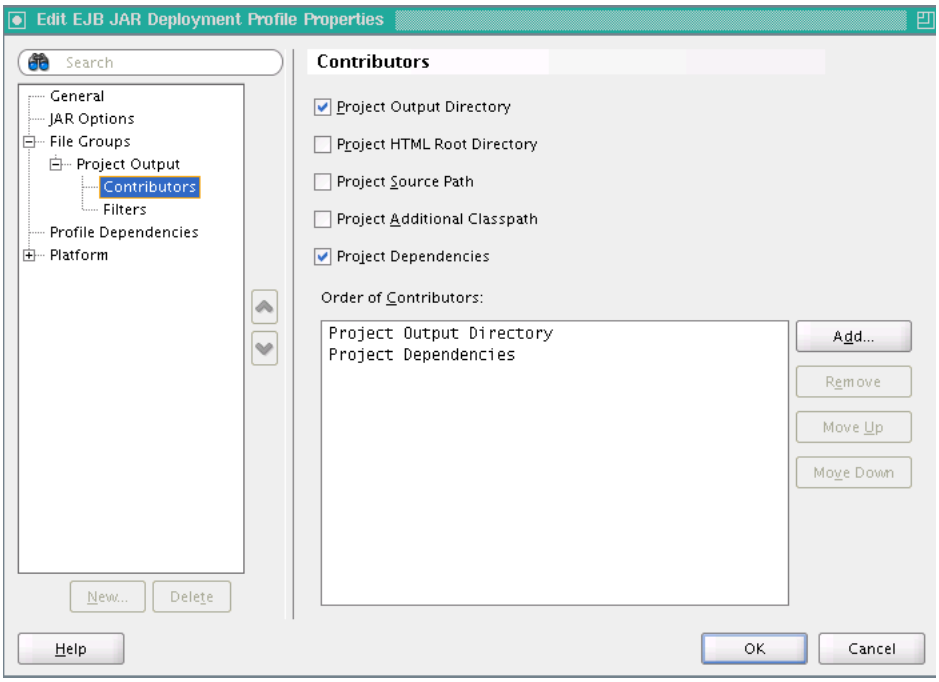
3. In Application Navigator select the **EssDemo** project.
4. Select the **New...** icon.
5. In the New Gallery, in the **Categories** area expand **General** and select **Deployment Profiles**.
6. In the New Gallery, in the **Items** area select **EJB JAR File**.
7. Click **OK**. This displays the Create Deployment Profile - EJB JAR File dialog.
8. In the Create Deployment Profile - EJB JAR File dialog, in the **Deployment Profile Name** field enter `ess-ejb`.
9. Click **OK**. This displays the Edit EJB JAR Deployment Profile Properties dialog.
10. In the Edit EJB JAR Deployment Profile Properties dialog, in the **Enterprise Application Name** field enter `EssDemoApp`, as shown in Figure 3–19.

Figure 3–19 EJB JAR Deployment Profile for Sample Application



11. In the EJB JAR Deployment Profile Properties dialog, in the Navigator expand **File Groups** and expand **Project Output**, and select **Contributors**.
12. In the Contributors area select **Project Output Directory** and **Project Dependencies** as shown in Figure 3–20.

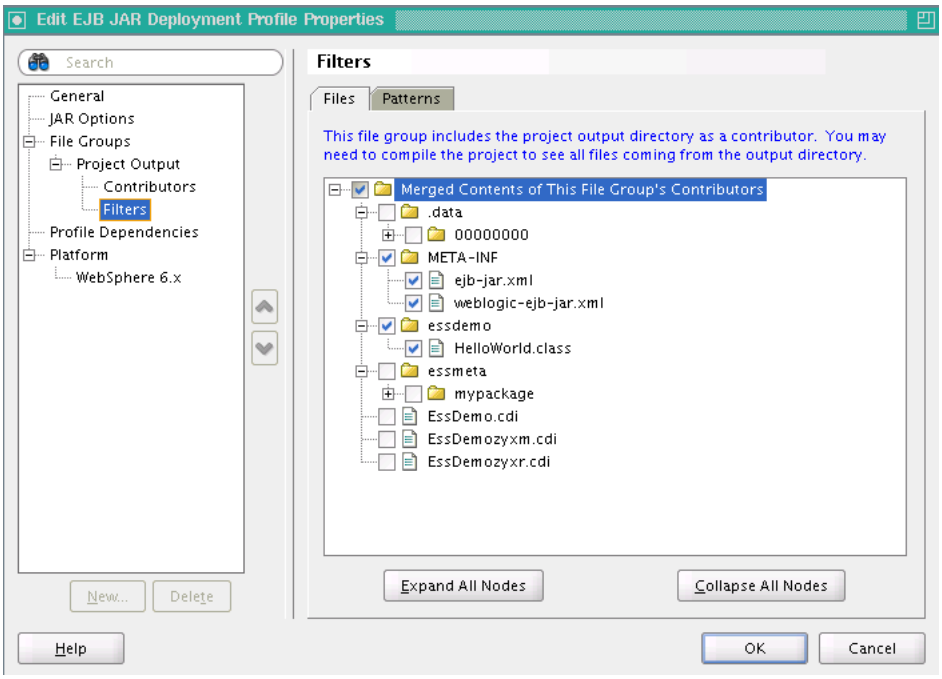
Figure 3–20 Selecting EJB Contributors for the EJB JAR Deployment



13. In the EJB JAR Deployment Properties dialog, in the Navigator expand **File Groups** and **Project Output**, and select **Filters**.

14. Select the **META-INF** folder and the **essdemo** folder as shown in Figure 3–21.

Figure 3–21 EJB JAR Deployment Profile File Groups Filters

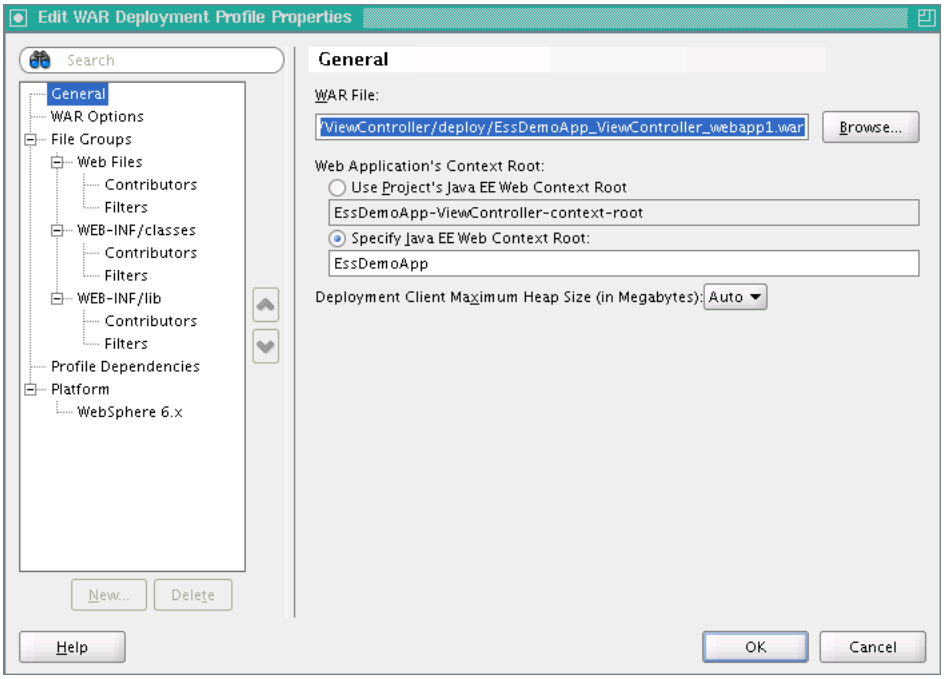


15. On the EJB JAR Deployment Profile Properties page, click **OK**.
16. On the Project Properties page, click **OK**.

To update WAR archive options:

1. In the Application Navigator, select the ViewController project.
2. Right-click and select **Project Properties...**.
3. In the Navigator, select **Deployment**.
4. In the Deployment page, in the **Deployment Profiles** area select the WAR File.
5. Click **Edit...**. This displays the Edit WAR Deployment Profile Properties dialog.
6. In the Edit War Deployment Profile Properties dialog, select **General** and configure the General page as follows, as shown in Figure 3–22:
   a. Set the **WAR File**: *path_to_mywork* /mywork/EssDemoApp/ViewController/deploy/EssDemoApp_ViewController_webapp1.war
   b. In the **Web Application Context Root** area, select **Specify Java EE Web Context Root**:
   c. In the **Specify Java EE Web Context Root**: text entry area, enter `EssDemoApp`.
   d. In the **Deployment Client Maximum Heap Size (in Megabytes)**: dropdown list select **Auto**

Figure 3–22 WAR Deployment Configuration Options



7. In the Edit WAR Deployment Profile Properties dialog, click **OK**.

   Oracle JDeveloper updates the deployment profile.

8. In the Project Properties dialog, click **OK**.

9. An application either uses the deployment name as the default value for its application name or you can set the application name using the property `applicationName` in the `ejb-jar.xml`. The default application name is the deployment name if the `applicationName` is not specified.

   To set the `applicationName` edit the `ejb-jar.xml` file to set the value of the `<activation-config-property>` named `applicationName`, as shown in Example 3–4.

Example 3–4 Setting applicationName in ejb-jar.xml

```
<enterprise-beans>
  <message-driven>
    <ejb-name>ESSAppEndpoint</ejb-name>
    <ejb-class>oracle.as.scheduler.ejb.EssAppEndpointBean</ejb-class>
    <activation-config>
      <activation-config-property>
        <activation-config-property-name>
          applicationName
        </activation-config-property-name>
        <activation-config-property-value>
          MY_APPLICATION_NAME
        </activation-config-property-value>
      </activation-config-property>
    </activation-config>
  </message-driven>
<enterprise-beans>
```

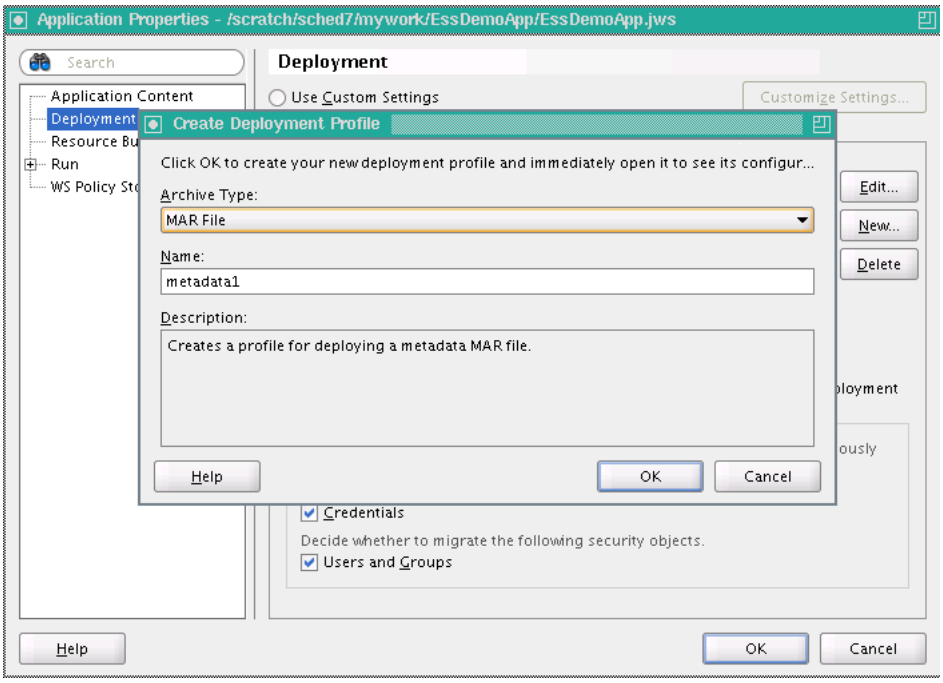
3.6.2 How to Assemble the MAR File for User Metadata

The sample application needs to contain the required MAR profile.

To create the MAR file:

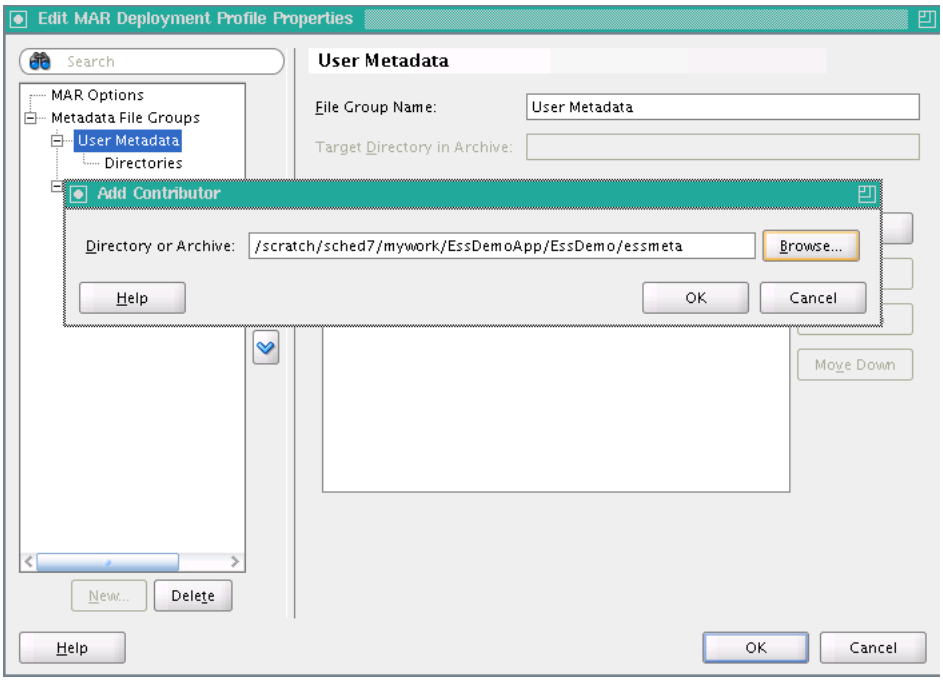
1. Open the EssDemoApp application and from the Application Menu select **Application Properties...**
2. In the Application Properties dialog, in the navigator select **Deployment**.
3. Select and delete the default deployment profile.
4. Click **New...**. This displays the Create Deployment Profile page.
5. In the **Archive Type** field, from the dropdown list select **MAR File** as shown in Figure 3–23.

Figure 3–23 Create Deployment Profile Page for New MAR



6. In the Create Deployment Profile dialog, in the **Name** field enter a name, for example enter **essMAR**.
7. In the Create Deployment Profile dialog, click **OK**.
8. On the Edit MAR Deployment Profile dialog, in the navigator expand **Metadata File Groups** and select **User Metadata**.
9. Click **Add...**. This displays the Add Contributor dialog.
10. On the Add Contributor dialog click **Browse** to add the essmeta metadata that contains the namespace for the Jobs and JobTypes directory, as shown in Figure 3–24. Note, you select the path that you need to include in the Add Contributor dialog by double-clicking the **essmeta** directory.

Figure 3–24 Adding User Metadata to MAR Profile



11. On the Add Contributor dialog, click **OK**.
12. In the navigator expand **Metadata File Groups** and **User Metadata** and select **Directories**.
13. Select the `mypackage` directory. This selects all the appropriate information for Oracle Enterprise Scheduler application user metadata for the application.

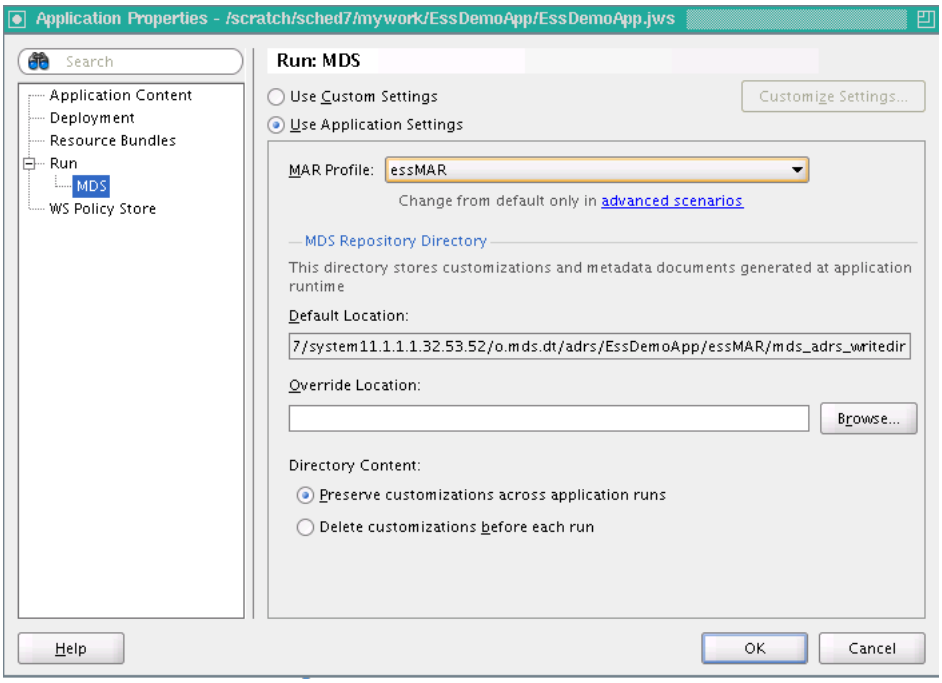
    Select the bottom most directory in the tree. This is the directory from which the namespace is created. For example, when selecting `oracle`, the namespace is `oracle`. When selecting the `product` directory, the namespace is `oracle/apps/product`. For example, to create the namespace `oracle/apps/product/component/ess`, click the `ess` directory.

    The folder you select in this dialog determines the top level namespace in `adf-config.xml`. For more information, see Section 3.6.3, "How to Assemble the EAR File for Scheduler Sample Application." This namespace should be the same as the package defined in job and job type definition. For more information, see Section 3.5, "Creating Metadata for Scheduler Sample Application."

    Note: If your namespace is too generic, then your Oracle ADF application might fail. Make sure to use proper package structure and map only the required namespaces.

14. On the Edit MAR Deployment Profile Properties page, click **OK**.
15. On the Application Properties page, in the navigator expand **Run** and select **MDS**.
16. Select the MAR profile you just created, **essMAR**, as shown in Figure 3–25.
17. Click **OK**.

Figure 3–25 Setting Application Properties Run MDS MAR Profile



3.6.3 How to Assemble the EAR File for Scheduler Sample Application

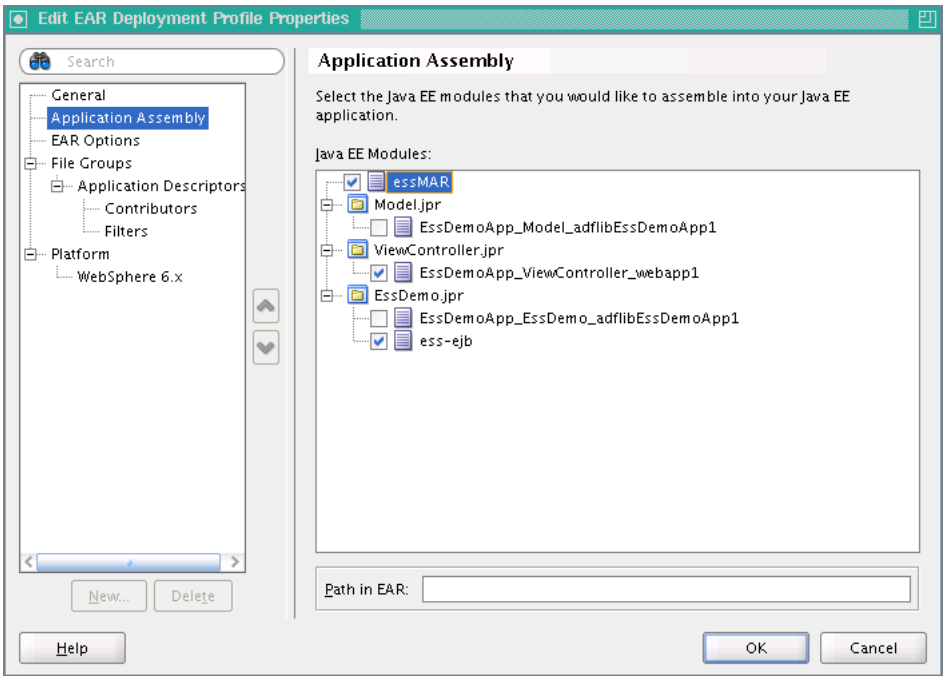
You need to prepare an EAR file that assembles the scheduler sample application. The EAR archive consists of the following:

- EJB JAR including the Oracle Enterprise Scheduler Java job implementation.
- WAR archive with the EssDemo servlet.

To create the EAR file for the application:

1. In the Application Navigator, select the EssDemoApp application.
2. From the Application Menu, select **Application Properties...**.
3. In the Application Properties Navigator, select **Deployment**.
4. Click **New...** to create a new deployment descriptor.
5. In the **Archive Type** dropdown list, select **EAR File**.
6. In the Create Deployment Profile dialog in the **Name** field enter the application name. For the scheduler application, enter `EssDemoApp`.
7. Click **OK**.
8. In the Edit EAR Deployment Profile Properties dialog, in the navigator select **Application Assembly**.
9. In the Application Assembly page in the **Java EE Modules** area select the appropriate checkboxes, including the following: **essMAR**, the WEB module in the **ViewController** project and the EJB module, **ess-ejb**, in the **EssDemo** project as shown in Figure 3–26.

Figure 3–26 Setting Application Assembly Options for EAR File



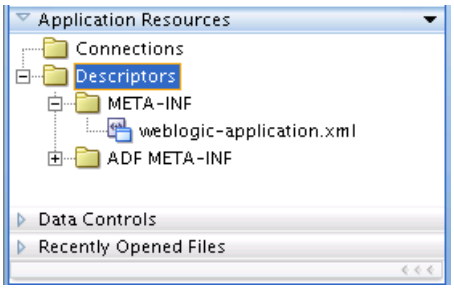
10. Click **OK**.
11. On the Application Properties page, click **OK**.

3.6.4 Add oracle.ess Library Weblogic Application Descriptor

You need to update the `weblogic-application.xml` file to include the `oracle.ess` library.

1. In the Application Navigator expand **Application Resources**.
2. In the navigator expand **Descriptors** and expand **META-INF**, as shown in Figure 3–27.

Figure 3–27 Viewing weblogic-application.xml in Application Resources



3. Double-click to open the `weblogic-application.xml` file.
4. Add the following to the `weblogic-application.xml` file. Example 3–5 shows a complete `weblogic-application.xml` file, including this `<library-ref>` element.

```
<library-ref>
  <library-name>oracle.ess</library-name>
</library-ref>
```

Example 3–5 Contents of Sample weblogic-application.xml File with oracle.ess

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<weblogic-application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.bea.com/ns/weblogic/weblogic-application
http://www.bea.com/ns/weblogic/weblogic-application/1.0/weblogic-application.xsd"
xmlns="http://www.bea.com/ns/weblogic/weblogic-application">
  <listener>
    <listener-class>oracle.mds.lcm.weblogic.WLLifecycleListener</listener-class>
  </listener>
  <listener>

<listener-class>oracle.adf.share.weblogic.listeners.ADFApplicationLifecycleListene
r</listener-class>
  </listener>
  <library-ref>
    <library-name>adf.oracle.domain</library-name>
    <implementation-version>11.1.1.1.0</implementation-version>
  </library-ref>

  <library-ref>
    <library-name>oracle.ess</library-name>
  </library-ref>

</weblogic-application>
```

3.7 Deploying and Running the Scheduler Sample Application

After you complete the steps to build and assemble the scheduler sample application you need to deploy the application to Oracle WebLogic Server. After you successfully deploy an application you can run the application. For the scheduler sample application you use a browser to run the EssDemo servlet to submit job requests to Oracle Enterprise Scheduler running on Oracle WebLogic Server.

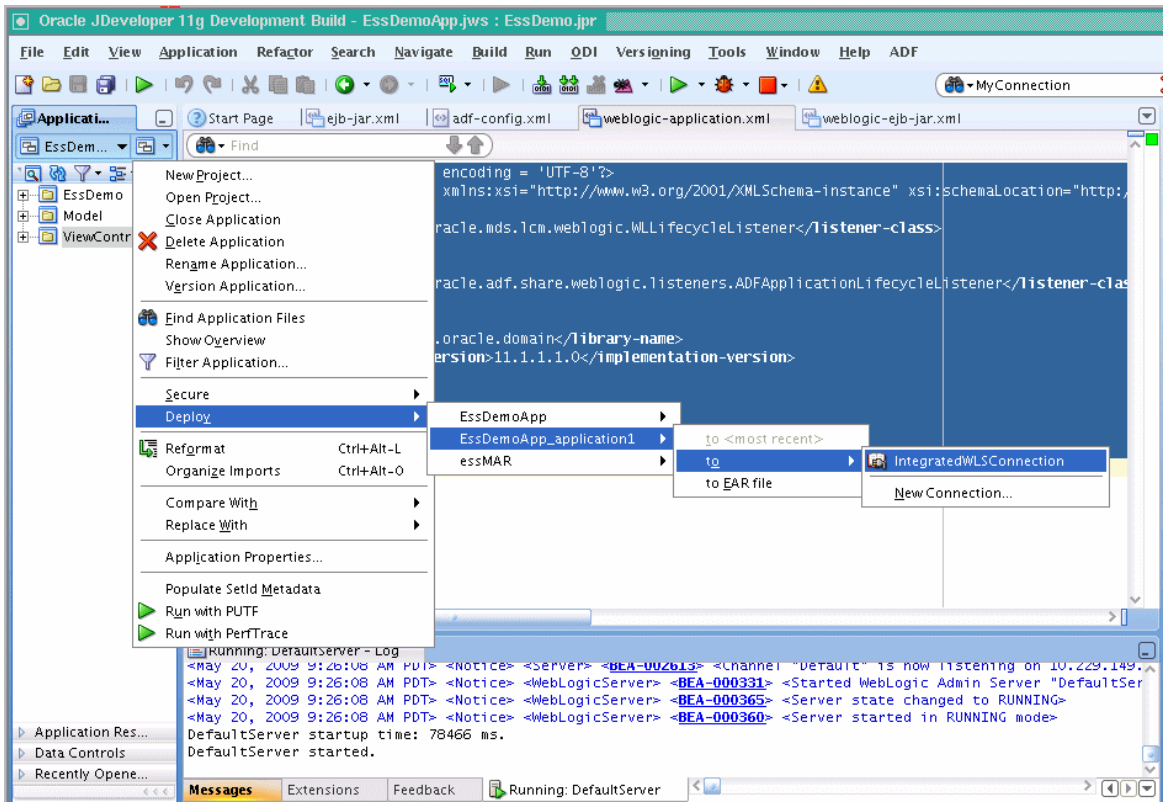
3.7.1 How to Deploy the EssDemoApp Application

To deploy the EssDemoApp application you need a properly configured and running Oracle WebLogic Server, and you need an active metadata server. When you deploy the application Oracle JDeveloper brings up the Deployment Configuration page. Select your repository from the dropdown list and Enter a partition name (the partition name defaults to application name).

To deploy the EssDemoApp application:

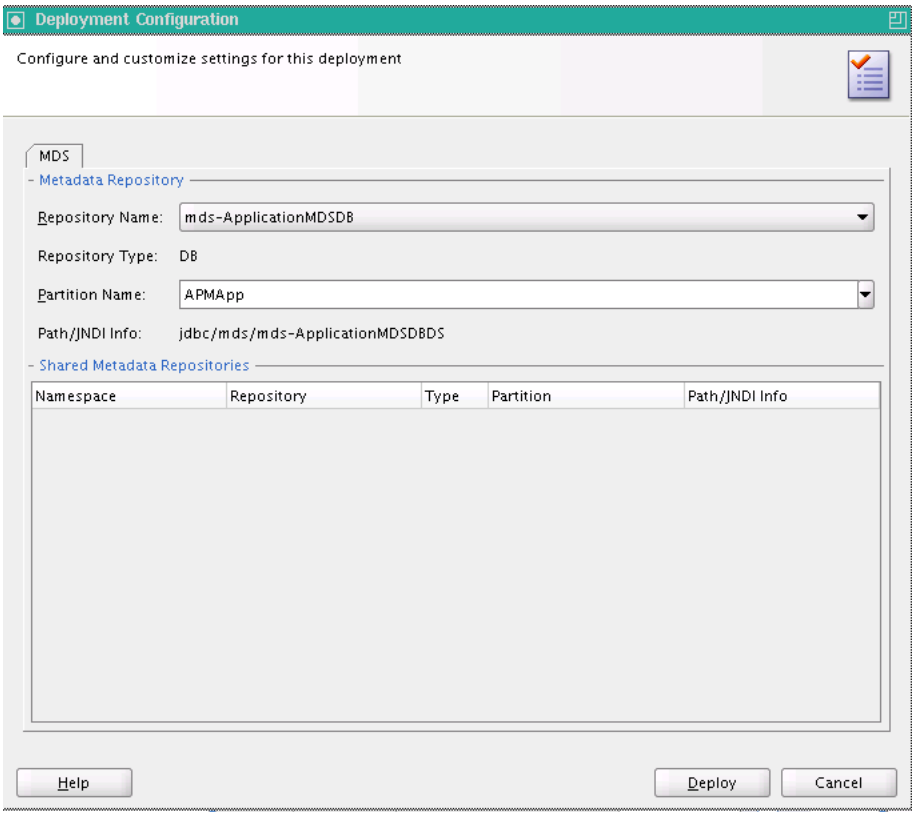
1. Check the Run Manager to make sure the Oracle WebLogic Server is up and running. If the Oracle WebLogic Server is not running, start the server. To start the server, from the **Run** menu click **Start Server Instance**.
2. In the Application Navigator, select the **EssDemoApp** application.
3. In the Application Navigator from the Application Menu select **Deploy > EssDemoApp** > **to** > **IntegratedWLSConnection**, as shown in Figure 3–28.

Figure 3–28 Deploying the EssDemoApp Application



4. Oracle JDeveloper shows the Deployment Configuration page, as shown in Figure 3–29. Select the appropriate options for your Metadata Repository.

Figure 3–29 Deployment Configuration Page with Metadata Repository Options



5. Click **Deploy**.
6. Verify the deployment using the Deployment Log.

3.7.2 How to Run the Scheduler Sample Application

To run the scheduler sample application you access the EssDemo servlet in a browser.

To access the EssDemo servlet:

1. Enter the following URL in a browser:

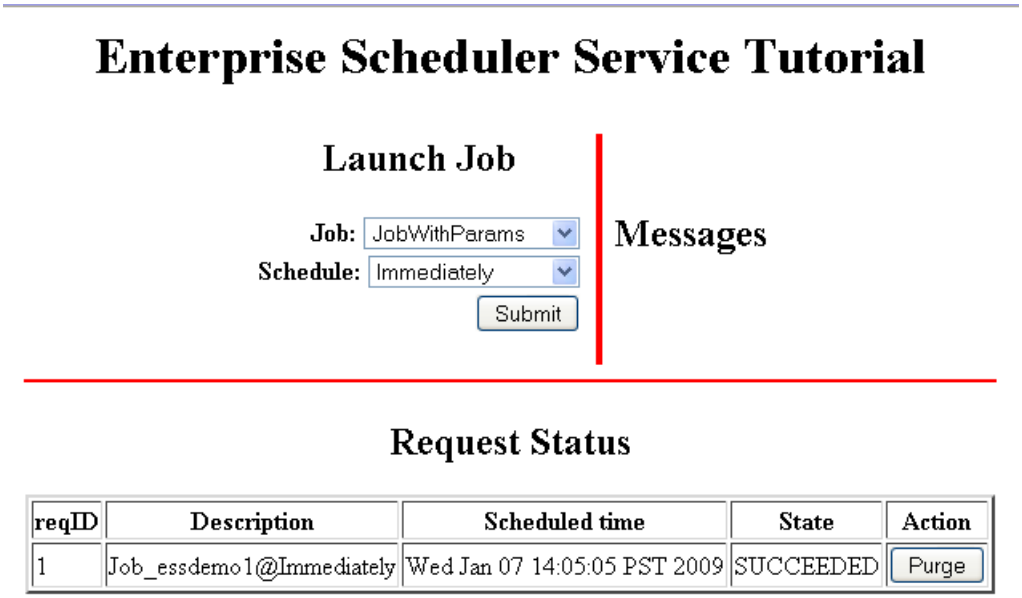
   http://*host*:*http-port*/*context-root*/essdemo

   For example,

   `http://myserver.us.oracle.com:7101/EssDemoApp/essdemo`

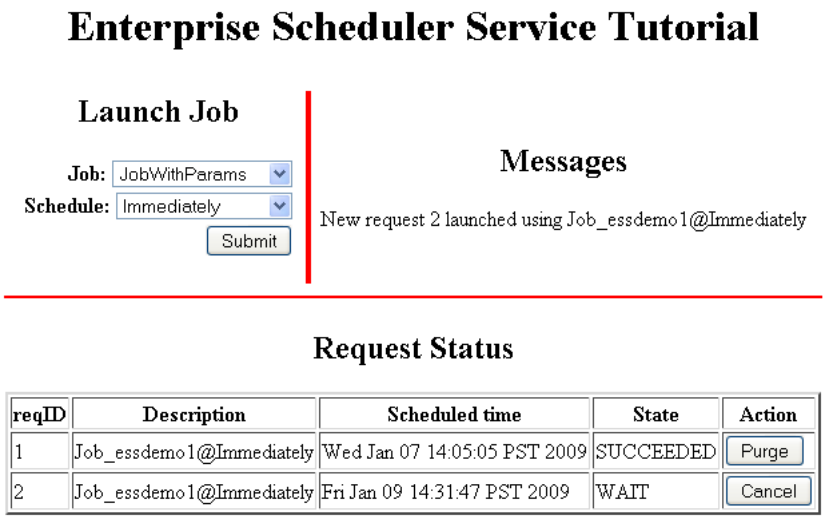
   This shows the EssDemo servlet, as shown in Figure 3–30.

Figure 3–30 Running EssDemo Servlet for Oracle Enterprise Scheduler Sample Application



2. Select a job definition from the **Job** drop-down menu.
3. Select a value from the **Schedule** drop-down menu.
4. Click **Submit**.
5. Refresh the browser to see the progress of the job in the Request Status area, as shown in Figure 3–31.

Figure 3–31 Running EssDemo Servlet with Request Status for Submitted Requests



3.7.3 How to Purge Jobs in the Scheduler Sample Application

Using the scheduler sample application and the EssDemo servlet you can remove completed jobs from the Request Status list.

To remove completed jobs:

1. Click **Purge** to purge a request.
2. Click **Cancel** to cancel a request that is either `RUNNING` or `WAITING`.

3.8 Troubleshooting the Oracle Enterprise Scheduler Sample Application

This section covers common problems and solutions for these problems.

1. **Problem**: `sqlplus: Command not found.`

   Solution: Run the Oracle Database commands in an environment that includes Oracle Database.

2. **Problem**: `SP2-0310: unable to open file "createuser_ess_oracle.sql"`

   Solution: Change to the `/rcu/integration/ess/sql` directory before running sqlplus scripts.

3. **Problem**:

   ```
   404 Not Found
   Resource /EssDemoApp-ViewController-context-root/essdemo not found on this
   server
   ```

   Solution: This and similar problems can be due to not using a URL that matches the root URL that you specify when set the context-root on the URL to access the application. To use a context-root that matches the deployed application, use the value that you specified.

   To check and set the context-root value in the WAR archive:

   a. Select the **ViewController** project.
   b. Right-click and from the dropdown list select **Project Properties**.
   c. In the navigator, select **Deployment**.
   d. In the Deployment Profiles area, select **essdemoapp** and click **Edit**.
   e. Choose the desired context-root, this forms the context-root on the URL to access the application.
   f. In the General area, select **Specify Java EE Web Context Root**.
   g. For the **Java EE Web Context Root:** text entry area, enter `EssDemoApp`.
   h. In the WAR Deployment Profile Properties window, click **OK**.
   i. In the Project Properties window, click **OK**.

4. **Problem:** `Unresolved application library references, defined in weblogic-application.xml: [Extension-Name: oracle.ess, exact-match: false]..`

   Deployment fails with errors. For example:

   ```
   09:30:59 AM] Building...
   [09:31:00 AM] Deploying 2 profiles...
   [09:31:01 AM] Wrote Web Application Module to
   /scratch/sched7/mywork/EssDemoApp/ViewController/deploy/EssDemoApp_
   ViewController_webapp1.war
   [09:31:01 AM] removed bundleresolver.jar from APP-INF because it cannot be part
   of an EJB deployment[09:31:01 AM] Wrote Enterprise Application Module to
   /scratch/sched7/mywork/EssDemoApp/deploy/EssDemoApp_application1.ear
   [09:31:02 AM] Deploying Application...
   ```

```
[09:31:04 AM] [Deployer:149193]Deployment of application 'EssDemoApp_
application1' has failed on 'DefaultServer'
[09:31:04 AM] [Deployer:149034]An exception occurred for task
[Deployer:149026]deploy application EssDemoApp_application1 on DefaultServer.:
[J2EE:160149]Error while processing library references. Unresolved application
library references, defined in weblogic-application.xml: [Extension-Name:
oracle.ess, exact-match: false]..
[09:31:05 AM] Weblogic Server Exception:
weblogic.management.DeploymentException: [J2EE:160149]Error while processing
library references. Unresolved application library references, defined in
weblogic-application.xml: [Extension-Name: oracle.ess, exact-match: false].
[09:31:05 AM]   See server logs or server console for more details.
[09:31:05 AM] weblogic.management.DeploymentException: [J2EE:160149]Error while
processing library references. Unresolved application library references,
defined in weblogic-application.xml: [Extension-Name: oracle.ess, exact-match:
false].
[09:31:05 AM] ####  Deployment incomplete.  ####
[09:31:05 AM] Deployment Failed
```

Solution: This deployment error can be seen when the application is correct, but the Oracle WebLogic Server configuration is not correct. The configuration includes the step, 3.1.4, "Create WLS domain". This configuration step is required.

3.8.1 How to Create the Oracle Enterprise Scheduler Database Schema

You need to create the Oracle Enterprise Scheduler Oracle Database schema. Oracle Enterprise Scheduler uses this schema to maintain information about job requests.

> **Note:** In the Oracle Fusion Applications environment, this step is not required. In this environment the database is installed with the Oracle Enterprise Scheduler schema pre-configured. Thus, in this environment you can skip this step.

In order to create the Oracle Enterprise Scheduler database schema, you need to install Oracle JDeveloper for use with Oracle Enterprise Scheduler. For more information, see the *Oracle Fusion Applications Installation Guide*.

3.8.2 How to Drop the Oracle Enterprise Scheduler Runtime Schema

If you have been running with previous version of the Oracle Enterprise Scheduler runtime schema, or if for any reason you need to drop the schema, you can do this using the `dropschema_ess_oracle.sql` script.

Use these steps only to drop the Oracle Enterprise Scheduler runtime schema. These steps clean up certain database objects and then drop the schema user. Note that simply dropping the Oracle Enterprise Scheduler schema is not sufficient to correctly drop and remove an existing schema.

> **Note:** For a first time installation you do not need to perform these steps. Only use these steps if you need to drop the database schema due to a previous installation error or to clean up your database after a previous use of Oracle Enterprise Scheduler.

To drop the database schema:

1. Terminate any container that is using Oracle Enterprise Scheduler schema.

2. Change to the `ess/sql` directory with the following command:

```
% cd  JDEV_install_dir/rcu/integration/ess/sql
```

3. Do the following, when connected as SYS or as SYSDBA. In the text, *ess_schema* represents Oracle Enterprise Scheduler schema being removed:

```
@dropschema_ess_oracle.sql ess_schema
alter session set current_schema=sys;
drop user ess_schema cascade;
```

Example in which *ess_schema* is oraess:

```
> @dropschema_ess_oracle.sql oraess
> alter session set current_schema=sys;
> drop user oraess cascade;
> exit
```

3.9 Using Submitting and Hosting Split Applications

When you build and deploy Oracle Enterprise Scheduler applications, you can use two split applications — a job submission application, a *submitter*, and a job execution application, *a hosting application*. Using this design you need to configure and deploy each application with options that support such a split configuration. In addition, some Oracle Enterprise Scheduler deployments use a separate Oracle WebLogic Server for the hosting and the submitting applications; for this deployment option the submitting application and the hosting application are deployed to separate Oracle WebLogic Servers. When the submitter application and the hosting application for Oracle Enterprise Scheduler run on separate Oracle WebLogic Servers, you need to configure the Oracle WebLogic Server for the hosting application so that the submitting application can find the hosting application.

To build the sample split applications, you do the following:

1. Build a backend hosting application that includes the code to be scheduled and run.
2. Build a frontend submitter application initiates the job requests.

3.9.1 How to Create the Backend Hosting Application for Scheduler

Using Oracle JDeveloper you create the backend application. To create the scheduler backend sample application you do the following:

- Create a backend application and project.
- Configure security.
- Define the deployment descriptors.
- Create the Java class that implements the Oracle Enterprise Scheduler executable interface.
- Create the Oracle Enterprise Scheduler metadata to describe the job
- Assemble the application.
- Deploy the application.

3.9.1.1 Creating the Backend Hosting Application

To work with Oracle Enterprise Scheduler with a split application you use Oracle JDeveloper to create the backend application and project, and to add Oracle Enterprise Scheduler extensions to the project.

To create the backend hosting application:

1. From JDeveloper choose **File > New** from the main menu.
2. In the New Gallery, expand **General**, select **Applications** and then **Generic Application**, and click **OK**.
3. In the Name your application page of the Create Generic Application wizard, set the **Application Name** field to `EssDemoApp`.
4. In the Name your project page, set the **Project Name** to `SuperEss`.

   This project is where you will create and save the Oracle Enterprise Scheduler metadata.
5. Add the **EJB** technology to the project.
6. In the Project Java Settings page, change the default package to `oracle.apss.ess.howto`.
7. In the Configure EJB Settings page, select **Generate ejb-jar.xml in this project** and click **Finish**.
8. In the Application Navigator, right-click the **SuperEss** project and select **Project Properties**.
9. In the Project Properties dialog, expand **Project Source Paths** and click the **Resources** navigation tab.
10. Select **Include Content from Subfolders**.
11. Click the **Libraries and Classpath** navigation tab.
12. Click **Add Library**, select **Enterprise Scheduler Extensions**, and click **OK**.

3.9.1.2 Configuring Security for the Backend Hosting Application

You need to create a user that is assigned to the EssDempAppRole role.

To configure security for the backend hosting application:

1. Select **Application** > **Secure** > **Configure ADF Security** from the main menu.
2. In the ADF Security page of the Configure ADF Security wizard, select **ADF Authentication**.
3. In the Authentication Type page, accept the default values as this application will not have a web module to secure.
4. Click **Finish**.

   A file named `jps-config.xml` is generated. You can find this file in the Application Resources panel by expanding **Descriptors**, and expanding **META-INF**. This file contains a security context or security stripe named after the application.
5. Select **Application** > **Secure** > **Users** from the main menu.

   A file named `jps-config.xml` is generated.

6. In the overview editor for the `jps-config.xml` file, click the **Add** icon in the **Users** list.
7. Set the name to `EssDemoAppUser` and set the password to `welcome1`.
8. Click the **Application Roles** navigation tab.
9. Click the **Add** icon in the **Roles** list and choose **Add New Role**.
10. Set the name to `EssDemoAppRole`.
11. Click the **Add** icon in the Mappings tab and choose **Add User**.
12. Select `EssDemoAppUser` and click **OK**.

3.9.1.3 Defining the Deployment Descriptors for the Backend Hosting Application

The sample application needs to contain the required EJB descriptors. You need to create the `ejb-jar.xml` and `weblogic-ejb-jar.xml` files and include these files with any Java implementation class that you create.

Oracle Enterprise Scheduler requires an application to assemble and provide an EJB JAR so that Oracle Enterprise Scheduler can find its entry point in the application while running job requests on behalf of the application. This EJB jar should have its required EJB descriptors in `ejb-jar.xml` and `weblogic-ejb-jar`, as well as any Java class implementations that are going to be submitted to Oracle Enterprise Scheduler. The descriptor files `ejb-jar.xml` and `weblogic-ejb-jar` must contain descriptions for the Oracle Enterprise Scheduler EJBs.

The Oracle Enterprise Scheduler backend application is deployed to Oracle WebLogic Server. You need to create a deployment profile in Oracle JDeveloper to deploy the `EssDemoApp` application.

The `EssDemoApp` application is a standalone application that contains an Oracle Enterprise Scheduler Java job and includes the required Oracle Enterprise Scheduler metadata, an Oracle Enterprise Scheduler message-driven bean (MDB), and the EJB descriptors for the application. This application does not perform Oracle Enterprise Scheduler submit API; in this hosting application the submission occurs in the frontend submitter application. In the hosting application, EssDemoApp, the `weblogic-ejb-jar.xml` exposes the EJB remote interface through JNDI (using the EJB remote interface allows for the job submission to occur in the frontend application).

You also need to create the `weblogic-application.xml` file to include the `oracle.ess` library, to add an Oracle Enterprise Scheduler listener, and to indicate which stripe to use to upload the `jazn-data.xml` policy.

To define the deployment descriptors for the backend hosting application:

1. In the Application Navigator, expand **SuperEss**, expand **Application Sources**, expand **META-INF**, and double-click **ejb-jar.xml**.
2. Replace the contents of the file with the XML shown in Example 3–6

Example 3–6 Contents to Copy to ejb-jar.xml for a Backend Hosting Application

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                             http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
         version="3.0">
  <display-name>ESS</display-name>
```

```
  <enterprise-beans>
    <message-driven>
      <ejb-name>ESSAppEndpoint</ejb-name>
      <ejb-class>oracle.as.scheduler.ejb.EssAppEndpointBean</ejb-class>
      <activation-config>
        <activation-config-property>
          <!-- The "applicationName" property specifies the logical name used
             - by Oracle Enterprise Scheduler to identify this application.
             - This name is independent of the application name used when
             - deploying the application to the container.  This decoupling
             - allows applications to safely hardcode the logical application
             - name in source code without having to worry about the more
             - frequently changed deployment name.
             -
             - Note: The name given here must also be specified in the
             - SYS_effectiveApplication property of each job definition and
             - job set of this application.
            -->

<activation-config-property-name>applicationName</activation-config-property-name>

<activation-config-property-value>EssDemoApp</activation-config-property-value>
        </activation-config-property>
        <activation-config-property>
          <!-- The "applicationStripe" property specifies which JPS security
             - stripe or "security context" Oracle Enterprise Scheduler should
             - use to perform security checks.
             -
             - The value here must be the same as the "injection-target-name"
             - value used by the "oracle.security.jps.ee.ejb.JpsInterceptor"
             - interceptor descriptor below.
             -
             - Note: When creating jps-config.xml through JDev, it will create
             - default security context using the JDev workspace name.  In
             - order to simplify things, we will use the JDev workspace name
             - as our value.  Otherwise, you will have to rename the security
             - context created by JDev or create your own.
            -->
          <activation-config-property-name>applicationStripe
          </activation-config-property-name>
          <activation-config-property-value>EssDemoApp
          </activation-config-property-value>
        </activation-config-property>
      </activation-config>
    </message-driven>

    <!-- The AsyncBean allows asynchronous Java jobs to notify
       - Oracle Enterprise Scheduler of its status through Java EE EJB APIs.
       - It is recommended to use the WebService callback pattern
       - instead of the EJB callbacks wherever possible.
      -->
    <session>
      <description>Async Request Bean</description>
      <ejb-name>AsyncRequestBean</ejb-name>
      <ejb-class>oracle.as.scheduler.ejb.AsyncRequestBean</ejb-class>
    </session>

    <!-- The Runtime Service allows users to interact with an Executable.
       - Operations include submitting, cancelling, querying, etc.
      -->
```

```
    <session>
      <description>Runtime Session Bean</description>
      <ejb-name>RuntimeServiceBean</ejb-name>
      <ejb-class>oracle.as.scheduler.ejb.RuntimeServiceBean</ejb-class>
    </session>

    <!-- The Metadata Service allows user to interact with
       - Oracle Enterprise Scheduler, metadata including job definitions,
       - job sets, job types, schedules, and so on. Operations include reading,
       - writing, querying, copying, deleting, and so on.
      -->
    <session>
      <description>Metadata Session Bean</description>
      <ejb-name>MetadataServiceBean</ejb-name>
      <ejb-class>oracle.as.scheduler.ejb.MetadataServiceBean</ejb-class>
    </session>

  </enterprise-beans>

  <!--
     - The JPS interceptor is used by JPS (Java Platform Security) in order to
     - perform security checks.  The "stripe name" is usually associated with
     - the application name but some groups split their security permissions
     - between Oracle ADF grants and Oracle Enterprise Scheduler grants, creating
     - two stripes.
     - For example, the Oracle ADF grants would live in the "MyApp" stripe while
     - the Oracle Enterprise Scheduler grants would live in the "MyAppEss".
     -
     - Note: For this example, we will use only 1 stripe.
     -
     - Note: When creating jps-config.xml through JDev, it will create
     - default security context using the JDev workspace name.  In
     - order to simplify things, we will use the JDev workspace name
     - as our value.  Otherwise, you will have to rename the security
     - context created by JDev or create your own.
    -->
  <interceptors>
    <interceptor>

<interceptor-class>oracle.security.jps.ee.ejb.JpsInterceptor</interceptor-class>
      <env-entry>
        <env-entry-name>application.name</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>EssDemoApp</env-entry-value>
        <injection-target>

<injection-target-class>oracle.security.jps.ee.ejb.JpsInterceptor</injection-targe
t-class>
          <injection-target-name>application_name</injection-target-name>
        </injection-target>
      </env-entry>
    </interceptor>
  </interceptors>
</ejb-jar>
```

3. In Application Navigator, right-click the **SuperEss** project and select **New**.
4. In the New Gallery, expand **General**, select **Deployment Descriptors** and then **Weblogic Deployment Descriptor**, and click **OK**.
5. In the Select Descriptor page select **weblogic-ejb-jar.xml**.

6. Click **Next**, click **Next** again, and click **Finish**.
7. In the source editor, replace the contents of the weblogic-`ejb-jar.xml` file that you just created with the XML shown in Example 3–7.

   This XML associates the MDB in the `ejb-jar.xml` file with the Oracle Enterprise Scheduler Resource Adapter. Without this XML, the application would not know what to talk to.

Example 3–7 Contents to Copy to weblogic-ejb-jar.xml for a Backend Hosting Application

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<weblogic-ejb-jar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.bea.com/ns/weblogic/weblogic-ejb-jar

http://www.bea.com/ns/weblogic/weblogic-ejb-jar/1.0/weblogic-ejb-jar.xsd"
                  xmlns="http://www.bea.com/ns/weblogic/weblogic-ejb-jar">

   <weblogic-enterprise-bean>
      <ejb-name>ESSAppEndpoint</ejb-name>
      <message-driven-descriptor>
         <resource-adapter-jndi-name>ess/ra</resource-adapter-jndi-name>
      </message-driven-descriptor>
      <dispatch-policy>ESSRAWM</dispatch-policy>
   </weblogic-enterprise-bean>

</weblogic-ejb-jar>
```

8. In Application Navigator, right-click the **SuperEss** project and select **New**.
9. In the New Gallery, expand **General**, select **Deployment Descriptors** and then **Weblogic Deployment Descriptor**, and click **OK**.
10. In the Select Descriptor page select **weblogic-application.xml**.
11. Click **Next**, click **Next** again, and click **Finish**.
12. In the source editor, replace the contents of the `weblogic-application.xml` file that you just created with the XML shown in Example 3–8.

Example 3–8 Contents to Copy to weblogic-application.xml for a Backend Hosting Application

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<weblogic-application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.bea.com/ns/weblogic/weblogic-application

http://www.bea.com/ns/weblogic/weblogic-application/1.0/weblogic-application.xsd"
                     xmlns="http://www.bea.com/ns/weblogic/weblogic-application">

    <!-- The following application parameter tells JPS which stripe it should
       - use to upload the jazn-data.xml policy.  If this parameter is not
       - specified, it will use the Java EE deployment name plus the version
       - number (e.g. EssDemoApp#V2.0).
      -->
    <application-param>
        <param-name>jps.policystore.applicationid</param-name>
        <param-value>EssDemoApp</param-value>
    </application-param>
```

```
    <!-- This listener allows JPS to configure itself and upload the
       - jazn-data.xml policy to the appropriate stripe
      -->
    <listener>

<listener-class>oracle.security.jps.wls.listeners.JpsApplicationLifecycleListener<
/listener-class>
    </listener>

    <!-- This listener allows MDS to configure itself and upload any metadata
       - as defined by the MAR profile and adf-config.xml
      -->
    <listener>

<listener-class>oracle.mds.lcm.weblogic.WLLifecycleListener</listener-class>
    </listener>

    <!-- This listener allows Oracle Enterprise Scheduler to configure itself
      -->
    <listener>

<listener-class>oracle.as.scheduler.platform.wls.deploy.ESSApplicationLifecycleLis
tener</listener-class>
    </listener>

    <!-- This shared library contains all the Oracle Enterprise Scheduler classes
      -->
    <library-ref>
        <library-name>oracle.ess</library-name>
    </library-ref>
</weblogic-application>
```

3.9.1.4 Creating a Java Implementation Class in the Backend Hosting Application

To define an application that runs a Java class under control of Oracle Enterprise Scheduler you need to create the Java class that implements the Oracle Enterprise Scheduler `Executable` interface. The `Executable` interface specifies the contract that allows you to use Oracle Enterprise Scheduler to invoke a Java class.

A Java class that implements the `Executable` interface must provide an empty `execute()` method.

To create a Java class that implements the executable Interface:

1. In the Application Navigator, right-click the **SuperEss** project and choose **New**.
2. In the New Gallery, expand **General**, select **Java** and then **Java Class**, and click **OK**.
3. In the Create Java Class dialog, set the name to `HelloWorldJob`.
4. Set the package to `oracle.apps.ess.howto`.
5. Click the **Add** icon, add the `oracle.as.scheduler.Executable` interface, and click **OK**.
6. In other fields accept the defaults.
7. Click **OK**.

8. In the source editor, replace the generated contents of the `HelloWorldJob.java` file with the code shown in Example 3–9.

Example 3–9 Oracle Enterprise Scheduler HelloWorldJob Java Class

```
package oracle.apps.ess.howto;

import java.util.logging.Logger;

import oracle.as.scheduler.Executable;
import oracle.as.scheduler.ExecutionCancelledException;
import oracle.as.scheduler.ExecutionErrorException;
import oracle.as.scheduler.ExecutionPausedException;
import oracle.as.scheduler.ExecutionWarningException;
import oracle.as.scheduler.RequestExecutionContext;
import oracle.as.scheduler.RequestParameters;

public class HelloWorldJob implements Executable {
    public HelloWorldJob() {
        super();
    }

    public void execute(RequestExecutionContext requestExecutionContext,
                        RequestParameters requestParameters)
        throws ExecutionErrorException, ExecutionWarningException,
               ExecutionCancelledException, ExecutionPausedException
    {
        printBanner(requestExecutionContext, requestParameters);
    }

    protected void printBanner(RequestExecutionContext requestExecutionContext,
                               RequestParameters requestParameters)
    {
        StringBuilder sb = new StringBuilder(1000);
        sb.append("\n==================================");
        sb.append("\n= EssDemoApp request is now running");
        long myRequestId = requestExecutionContext.getRequestId();
        sb.append("\n= Request Id = " + myRequestId);
        sb.append("\n= Request Properties:");

        for (String paramKey : requestParameters.getNames()) {
            Object paramValue = requestParameters.getValue(paramKey);
            sb.append("\n=\t(" + paramKey + ", " + paramValue + ")");
        }
        sb.append("\n=");
        sb.append("\n==================================");

        Logger logger = Logger.getLogger("oracle.apps.ess.howto");
        logger.info(sb.toString());
    }
}
```

3.9.1.5 Creating Metadata for the Backend Hosting Application

To use the Oracle Enterprise Scheduler split application to submit a job request you need to create metadata that defines a job request, including the following:

- A job type: this specifies an execution type and defines a common set of parameters for a job request.

- A job definition: this is the basic unit of work that defines a job request in Oracle Enterprise Scheduler.

> **Note:** For Oracle Fusion Applications use cases, use the prepackaged Oracle Enterprise Scheduler job types instead of creating your own. For demonstration purposes, you will create your own job type.

To create metadata for the backend hosting application:

1. In the Application Navigator, right-click the **SuperEss** project and choose **New**.
2. In the New Gallery, select the **All Technologies** tab.
3. Expand **Business Tier**, select **Enterprise Scheduler Metadata** and then **Job Type**, and click **OK**.
4. In the Create Job Type dialog, specify the following:
   a. In the **Name** field, enter `HelloWorldJobType`.
   b. In the **Package** field, enter `/oracle/apps/ess/howto/`.
   c. Select **JAVA_TYPE** from the **Execution Type** dropdown list.
   d. Click **OK**. This creates the `HelloWorldJobType.xml` file and Oracle JDeveloper displays the file in the editor.
5. In the editor window, set the description to `HelloWorld Example`.
6. Set the class name to `oracle.apps.ess.howto.HelloWorldJob`.
7. In the Application Navigator, right-click the **SuperEss** project and choose **New**.
8. Expand **Business Tier**, select **Enterprise Scheduler Metadata** and then **Job Definition**, and click **OK**.
9. In the Create Job Definition dialog, specify the following:
   a. Set the name to `HelloWorldJobDef`.
   b. Set the package to `/oracle/apps/ess/howto/`.
   c. Set the job type to `/oracle/apps/ess/howto/HelloWorldJobType`.
   d. Click **OK**. This creates the `HelloWorldJobDef.xml` file and Oracle JDeveloper displays the file in the editor.
10. In the editor window, set the description to `HelloWorld Example`.
11. Click the **Add** icon in the **System Properties** section.
12. In the Add System Property dialog, select **SYS_effectiveApplication** from the **Name** dropdown list.
13. Set the initial value to `EssDemoApp` and click **OK**.
14. Click the **Add** icon in the **Access Control** section.
15. In the Add Access Control dialog, ensure that **EssDemoApp** role is selected in the **Role** dropdown list.

    This is the role that you created in Section 3.9.1.2, "Configuring Security for the Backend Hosting Application."
16. Select **Read** and select **Execute**.
17. Click OK.

3.9.1.6 Assembling the Backend Hosting Application for Oracle Enterprise Scheduler

After you create the backend sample application you use Oracle JDeveloper to assemble the application.

To assemble the backend application you do the following:

- Create the EJB Java Archive
- Create the application MAR and EAR files

3.9.1.6.1 How to Assemble the EJB JAR File for the Backend Hosting Application The EJB Java archive file includes descriptors for the Java job implementations.

To assemble the EJB JAR file for the backend hosting application:

1. In Application Navigator, right-click the **SuperEss** project and select **Rebuild SuperEss.jpr**.

   In the Messages Log you should see a successful compilation message, for example:

   ```
   [3:40:22 PM] Successful compilation: 0 errors, 0 warnings.
   ```

2. In Application Navigator, right-click the **SuperEss** project and choose **New**.
3. In the New Gallery, expand **General**, select **Deployment Profiles** and then **EJB JAR File**, and click **OK**.
4. In the Create Deployment Profile dialog, set the **Deployment Profile Name** to `JAR_SuperEssEjbJar`.
5. Optionally, in the Edit EJB JAR Deployment Profile Properties dialog, expand **File Groups**, expand **Project Output**, and select **Filters** and clear the **essmeta** checkbox.

   Clearing this checkbox prevents the JAR file from being cluttered with unnecessary XML files and reduces the overall memory footprint.
6. On the EJB JAR Deployment Profile Properties dialog, click **OK**.
7. On the Project Properties dialog, click **OK**.

3.9.1.6.2 How to Assemble the MAR and EAR Files for the Backend Hosting Application The sample application needs to contain the MAR profile and the EAR file that assembles the scheduler backend application.

To create the MAR and EAR files for the backend hosting application:

1. From the main menu, choose **Application Menu > Application Properties...**
2. In the Application Properties dialog, click the **Deployment** navigation tab and click **New**.
3. In the Create Deployment Profile dialog, select **MAR File** from the **Archive Type** dropdown list.
4. In the **Name** field, enter `MAR_EssDemoAppMar` and click **OK**.
5. In the Edit MAR Deployment Profile dialog, expand **Metadata File Groups** and click **User Metadata**.
6. Click **Add**.
7. In the Add Contributor dialog add the `essmeta` directory.

For example, if your work space is at `/tmp/EssDemoApp`, then the directory to add is `/tmp/EssDemoApp/SuperEss/essmeta`.

8. On the Add Contributor dialog, click **OK**.
9. In the navigator expand **Metadata File Groups** and **User Metadata** and select **Directories**.
10. Expand the directories and select the deepest directory of the package name, which is the `howto` directory.

    The directory that you select forms the MDS namespace. In order to avoid conflicts, you must select the most specific namespace.
11. Click **OK**.
12. In the Deployment page of the Application Properties dialog, click **New**.
13. In the Create Deployment Profile dialog, select **EAR File** from the **Archive Type** dropdown list.
14. In the **Name** field, enter `EAR_EssDemoAppEar` and click **OK**.
15. In the Edit EAR Deployment Profile dialog, click the **General** navigation tab and enter `EssDemoApp` in the **Application Name** field.
16. Click the **Application Assembly** navigation tab, then select `MAR_ESSDemoAppMar` and select `JAR_SuperEssEjbJar`.
17. Click **OK**.
18. In the Application Properties dialog, click **OK**.

3.9.1.7 Deploying the Backend Hosting Application

After assembling the application, you can deploy it to the server.

To deploy the backend hosting application:

1. From the main menu, choose **Application > Deploy > EAR_EssDemoAppEar...**
2. Set up and deploy the application to a container.
3. When the Deployment Configuration dialog appears, make a note of the default values, but do not change them.

3.9.2 How to Create the Frontend Submitter Application for Oracle Enterprise Scheduler

In an Oracle Enterprise Scheduler split application you use the Oracle Enterprise Scheduler APIs to submit job requests from a frontend application. The `EssDemoAppUI` application provides a Java servlet for a servlet based user interface for submitting job requests (using Oracle Enterprise Scheduler).

To create the frontend submitter sample application you do the following:

- Create a frontend application and project.
- Configure the `ejb-jar.xml` file.
- Create the web project
- Configure security.
- Create the HTTP servlet.
- Edit the `web.xml` file.

- Edit the `weblogic-application.xml` file.
- Edit the `adf-config` file.
- Assemble the application.
- Deploy the application.

3.9.2.1 Creating the Frontend Submitter Application

You use JDeveloper to build the frontend submitter application using similar steps as you used for the backend hosting application.

To create the frontend submitter application:

1. Complete the steps in Section 3.9.1.1, "Creating the Backend Hosting Application" but this time use `ESSDemoAppUI` as the name of the application.
2. In the Application Navigator, right-click the **SuperEss** project and choose **New**.
3. In the New Gallery, select **General**, select **Folder**, and click **OK**.
4. Set the folder name to `essmeta` and click **OK**.

3.9.2.2 Configuring the ejb-jar.xml File for the Frontend Submitter Application

You need to add entries to the ejb-jar.xml file to enable asynchronous Java jobs to notify the Oracle Enterprise Scheduler of its status and to enable users to interact with executable operations, such as submitting operations, and with Oracle Enterprise Scheduler metadata, such as job definitions. You also need to indicate which stripe to use.

To define the deployment descriptors for the frontend submitter application:

1. In the Application Navigator, expand **SuperEss**, expand **Application Sources**, expand **META-INF**, and double-click **ejb-jar.xml**.
2. Replace the contents of the file with the XML shown in Example 3–10

Example 3–10 Contents to Copy to ejb-jar.xml for a Frontend Submitter Application

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
         version="3.0">
  <display-name>ESS</display-name>

  <enterprise-beans>

    <!-- Note that the UI application does NOT have a message driven bean.
       - This is because the UI application does not run any jobs.  The UI
       - application does have the other EJBs.
      -->

    <!-- The AsyncBean allows asynchronous Java jobs to notify
       - Oracle Enterprise Scheduler of its status through Java EE EJB APIs.
       - It is recommended to instead use the WebService callback pattern
       - instead of the EJB callbacks wherever possible.
      -->
    <session>
      <description>Async Request Bean</description>
```

```
      <ejb-name>AsyncRequestBean</ejb-name>
      <ejb-class>oracle.as.scheduler.ejb.AsyncRequestBean</ejb-class>
    </session>

    <!-- The Runtime Service allows users to interact with an Executable.
       - Operations include submitting, cancelling, querying, etc.
      -->
    <session>
      <description>Runtime Session Bean</description>
      <ejb-name>RuntimeServiceBean</ejb-name>
      <ejb-class>oracle.as.scheduler.ejb.RuntimeServiceBean</ejb-class>
    </session>

    <!-- The Metadata Service allows users to interact with
       - Oracle Enterprise Scheduler, metadata, including job definitions,
       - job sets, job types, schedules, and so on.
       - Operations include reading, writing, querying, copying, deleting,
       - and so on.
      -->
    <session>
      <description>Metadata Session Bean</description>
      <ejb-name>MetadataServiceBean</ejb-name>
      <ejb-class>oracle.as.scheduler.ejb.MetadataServiceBean</ejb-class>
    </session>

  </enterprise-beans>

  <!--
     - The JPS interceptor is used by JPS (Java Platform Security) in order to
     - perform security checks.  The "stripe name" is usually associated with
     - the application name but some groups split their security permissions
     - between Oracle ADF grants and Oracle Enterprise Scheduler grants, thereby
     - creating two stripes. For example, the Oracle ADF grants would live
     - in the "MyApp" stripe while the Oracle Enterprise Scheduler
     - grants would live in the "MyAppEss".
     -
     - Note: For this example, we will use only 1 stripe.
     -
     - Note: When creating jps-config.xml through JDev, it will create
     - default security context using the JDev workspace name.  In
     - order to simplify things, we will use the JDev workspace name
     - as our value.  Otherwise, you will have to rename the security
     - context created by JDev or create your own.
    -->
  <interceptors>
    <interceptor>

<interceptor-class>oracle.security.jps.ee.ejb.JpsInterceptor</interceptor-class>
      <env-entry>
        <env-entry-name>application.name</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>EssDemoApp</env-entry-value>
        <injection-target>

<injection-target-class>oracle.security.jps.ee.ejb.JpsInterceptor</injection-targe
t-class>
          <injection-target-name>application_name</injection-target-name>
        </injection-target>
      </env-entry>
    </interceptor>
```

```
  </interceptors>
</ejb-jar>
```

3.9.2.3 Creating the SuperWeb Project

You need to create a web project for the servlet.

To create the SuperWeb project:

1. Right-click the SuperEss **project** and choose **New**.
2. In the New Gallery, expand **General**, select **Projects** and then **Generic Project**, and click **OK**.
3. In the Name your application page of the Create Generic Application wizard, set the **Application Name** field to `SuperWeb`.
4. In the Name your project page, set the **Project Name** to `SuperEss`.
5. Add the **JSP and Servlets** technology to the project.
6. In the Project Java Settings page, change the default package to `oracle.apss.ess.howto` and click **Finish**.
7. In the Application Navigator, right-click the **SuperWeb** project and choose **Project Properties**.
8. Click the **Libraries and Classpath** navigation tab.
9. Click **Add Library**, select **ADF Web Runtime** and **Enterprise Scheduler Extensions**, and click **OK**.

3.9.2.4 Configuring Security for the Frontend Submitter Application

You need to configure security for the application. You do not have to create any users or roles as the EssDemoAppUI application will simply share the users and roles created by the EssDemoApp application.

To configure security for the frontend submitter application:

1. Select **Application** > **Secure** > **Configure ADF Security** from the main menu.
2. In the ADF Security page of the Configure ADF Security wizard, select **ADF Authentication**.
3. In the Authentication Type page, select **SuperWeb.jpr** from the **Web Project** dropdown list.
4. Select **HTTP Basic Authentication**.
5. Click **Finish**.

   A file named `jps-config.xml` is generated. You can find this file in the Application Resources panel by expanding **Descriptors**, and expanding **META-INF**.

3.9.2.5 Creating the HTTP Servlet for the Frontend Submitter Application

Normally, more complex user interfaces that are built on heavy weight frameworks such as Oracle Application Development Framework are employed, but for the sake of simplicity, you use a basic HTTP servlet for the submitter application.

To create the HTTP Servlet for the frontend submitter application:

1. Right-click the SuperEss **project** and choose **New**.
2. In the New Gallery, expand **Web Tier**, select **Servlets** and then **HTTP Servlet**, and click **OK**.
3. In the Web Application page of the Web Application wizard, select **Servlet 2.5\JSP 2.1 (Java EE 1.5)**.
4. In the Create HTTP Servlet - Step 1 of 3: Servlet Information page, enter `EssDemoAppServlet` in the **Class** field.
5. Enter `oracle.apps.ess.howto` in the **Package** field and click **Next**.
6. Click **Finish**.
7. In the source editor, replace the contents of ESSDemoAppServlet.java with the code in Example 3–11.

Example 3–11 HTTP Servlet Code for the Frontend Submitter Application

```
package oracle.apps.ess.howto;

import java.io.IOException;
import java.io.PrintWriter;
import java.io.StringWriter;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Enumeration;
import java.util.HashSet;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;
import java.util.Map;
import java.util.Set;
import java.util.SortedSet;
import java.util.TreeSet;
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.regex.Pattern;

import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import oracle.as.scheduler.MetadataObjectId;
import oracle.as.scheduler.MetadataObjectId.MetadataObjectType;
import oracle.as.scheduler.MetadataService;
import oracle.as.scheduler.MetadataService.QueryField;
import oracle.as.scheduler.MetadataServiceHandle;
import oracle.as.scheduler.RequestDetail;
import oracle.as.scheduler.RequestParameters;
import oracle.as.scheduler.RuntimeService;
import oracle.as.scheduler.RuntimeServiceHandle;
import oracle.as.scheduler.State;
import oracle.as.scheduler.core.JndiUtil;
```

```
public class EssDemoAppServlet extends HttpServlet {
    @SuppressWarnings("compatibility:4685800289380934682")
    private static final long serialVersionUID = 1L;

    private static final String CONTENT_TYPE = "text/html; charset=UTF-8";
    private static final String MESSAGE_KEY = "Message";
    private static final String PATH_SUBMIT = "/submitRequest";
    private static final String PATH_ALTER = "/alterRequest";
    private static final String MDO_SEP = ";";
    private static final String ACTION_CANCEL = "Cancel";
    private static final String ESS_UNAVAIL_MSG =
        "<p>Enterprise Scheduler Service is currently unavailable. Cause: %s</p>";

    private enum PseudoScheduleChoices {
        Immediately(0),
        InTenSeconds(10),
        InTenMinutes(10 * 60);

        @SuppressWarnings("compatibility:-5637079380819677366")
        private static final long serialVersionUID = 1L;

        private int m_seconds;

        private PseudoScheduleChoices(int seconds) {
            m_seconds = seconds;
        }

        public int getSeconds() {
            return m_seconds;
        }
    }




    public EssDemoAppServlet() throws ServletException {
        super();
    }

    @Override
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType(CONTENT_TYPE);

        HttpSession session = request.getSession(true);
        String lastMessage = String.valueOf(session.getAttribute(MESSAGE_KEY));

        if ("null".equals(lastMessage)) {
            lastMessage = "";
        }

        try {
            RuntimeLists runtimeLists = getRuntimeLists();
```

```
            MetadataLists metadataLists = getMetadataLists();
            renderResponse(metadataLists, runtimeLists,
                           request, response, lastMessage);
        } catch (ServletException se) {
            throw se;
        } catch (Exception e) {
            throw new ServletException(e);
        }
    }

    @Override
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType(CONTENT_TYPE);
        request.setCharacterEncoding("UTF-8");

        HttpSession session = request.getSession(true);
        String pathInfo = request.getPathInfo();

        // Clear the message on every post request
        StringBuilder message = new StringBuilder("");

        try {
          // Select each handler based on the form action
          if ("".equals(pathInfo)) {
            // No processing
          } else if (PATH_SUBMIT.equals(pathInfo)) {
            postSubmitRequest(request, message);
          } else if (PATH_ALTER.equals(pathInfo)) {
            postAlterRequest(request, message);
          } else {
            message.append(String.format("<p>No handler for pathInfo=%s</p>",
                                         pathInfo));
          }
        }
        catch (ServletException se) {
            Throwable t = se.getCause();
            String cause = (t == null) ? se.toString() : t.toString();
            message.append (String.format(ESS_UNAVAIL_MSG, cause));
        }

        // Storing the messages in the session allows them to persist
        // through the redirect and across refreshes.
        session.setAttribute(MESSAGE_KEY, message.toString());

        // render the page by redirecting to doGet(); this intentionally
        // strips the actions and post data from the request.
        response.sendRedirect(request.getContextPath() +
                              request.getServletPath());
    }

    /**
     * Handle the job submission form.
     * @param request
     * @param message
     * @throws ServletException
     */
    private void postSubmitRequest(HttpServletRequest request,
```

```
                                    StringBuilder message)
        throws ServletException
    {
        String jobDefName = request.getParameter("job");
        String scheduleDefName = request.getParameter("schedule");

        // Various required args for submission
        Calendar start = Calendar.getInstance();
        start.add(Calendar.SECOND, 2);


        // Launch the job based on form contents
        if (jobDefName == null || scheduleDefName == null) {
            message.append("Both a job name and a schedule name must be
specified\n");
        } else {
            PseudoScheduleChoices pseudoSchedule = null;

            // See if schedule given is actually a pseudo schedule
            try {
                pseudoSchedule = PseudoScheduleChoices.valueOf(scheduleDefName);
            } catch (IllegalArgumentException e) {
                // The string is not a valid member of the enum
                pseudoSchedule = null;
            }


            MetadataObjectId scheduleDefId = null;
            String scheduleDefNamePart = null;
            MetadataObjectId jobDefId = stringToMetadataObjectId(jobDefName);

            // Don't look up schedules that aren't real
            if (pseudoSchedule != null) {
                scheduleDefNamePart = scheduleDefName;
                start.add(Calendar.SECOND, pseudoSchedule.getSeconds());
            } else {
                scheduleDefId = stringToMetadataObjectId(scheduleDefName);
                scheduleDefNamePart = scheduleDefId.getNamePart();
            }

            String jobDefNamePart = jobDefId.getNamePart();
            String requestDesc = jobDefNamePart + "@" + scheduleDefNamePart;


            Logger logger = getLogger();
            long requestId = submitRequest(pseudoSchedule, requestDesc,
                                           jobDefId, scheduleDefId, start,
logger);

            // Populate the message block based on results
            message.append(String.format("<p>New request %d launched using
%s</p>",
                                         requestId, requestDesc));
        }
    }

    private Long submitRequest(final PseudoScheduleChoices pseudoSchedule,
                               final String requestDesc,
                               final MetadataObjectId jobDefId,
                               final MetadataObjectId scheduleDefId,
```

```
                                final Calendar start,
                                final Logger logger)
            throws ServletException
        {
            RuntimeServicePayload<Long> myPayload = new RuntimeServicePayload<Long>()
    {
                @Override
                Long execute(RuntimeService service,
                             RuntimeServiceHandle handle,
                             Logger logger)
                    throws Exception
                {
                    RequestParameters params = new RequestParameters();
                    return (null != pseudoSchedule)
                        ? service.submitRequest(handle, requestDesc, jobDefId,
                                                start, params)
                        : service.submitRequest(handle, requestDesc, jobDefId,
                                                scheduleDefId, null,
                                                start, null, params);
                }
            };
            try {
                return performOperation(myPayload, logger);
            } catch (Exception e) {
                throw new ServletException("Error submitting request using job: " +
                                           jobDefId + " and schedule: " +
                                           scheduleDefId, e);
            }
        }

        /**
         * Handle the "Cancel" and "Purge" actions from the form enclosing
         * the Request Status table.
         * @param request
         * @param message
         * @throws ServletException
         */
        private void postAlterRequest(HttpServletRequest request,
                                      StringBuilder message)
            throws ServletException
        {
            String cancelID = null;

            /*
             * there are a few assumptions going on here...
             * the HTTP button being used to transmit the action and
             * request is backwards from its normal usage (eg. the name
             * should be invariable, and the value variable).  Because we
             * want to display either "Purge" or "Cancel" on the button, and
             * transmit the reqId with it, we are reversing the map entry
             * to get the key (which in this case will be the reqID), and
             * match it to the value (Purge or Cancel).
             * Assumptions are that there will be only one entry in the map
             * per request (one purge or cancel).  Also, that the datatypes
             * for the key and value willl be those documented for
             * ServletRequest (<K,V> = <String, String[]>).
             */
            Map requestMap = request.getParameterMap();
            Iterator mapIter = requestMap.entrySet().iterator();
            while (mapIter.hasNext()) {
```

```
            Map.Entry entry = (Map.Entry)mapIter.next();
            String key = (String)entry.getKey();
            String[] values = (String[])entry.getValue();
            if (ACTION_CANCEL.equals(values[0])) {
                cancelID = key;
            }
        }

        if (cancelID != null) {
            try {
                final String cancelId2 = cancelID;
                RuntimeServicePayload<Void> myPayload = new
RuntimeServicePayload<Void>() {
                    @Override
                    Void execute(RuntimeService service,
                                 RuntimeServiceHandle handle,
                                 Logger logger)
                        throws Exception
                    {
                        service.cancelRequest(handle, Long.valueOf(cancelId2));
                        return null;
                    }
                };

                Logger logger = getLogger();
                performOperation(myPayload, logger);
                message.append
                    (String.format("<p>Cancelled request %s</p>", cancelID));
            } catch (Exception e) {
                throw new ServletException
                    ("Error canceling or purging request", e);
            }
        } else {
            message.append("<p>No purge or cancel action specified</p>");
        }
    }

    private String metadataObjectIdToString(MetadataObjectId mdoID)
        throws ServletException {

        String mdoString =
            mdoID.getType().value() + MDO_SEP + mdoID.getPackagePart() +
            MDO_SEP + mdoID.getNamePart();

        return mdoString;
    }

    private MetadataObjectId stringToMetadataObjectId(String mdoString)
        throws ServletException {
        String[] mdoStringParts = mdoString.split(Pattern.quote(MDO_SEP));
        if (mdoStringParts.length != 3) {
            throw new ServletException(String.format("Unexpected number of
components %d found " +
                                                     "when converting %s to
MetadataObjectID",
                                                     mdoStringParts.length,
                                                     mdoString));
        }

        MetadataObjectType mdType =
```

```
            MetadataObjectType.getMOType(mdoStringParts[0]);
        String mdPackage = mdoStringParts[1];
        String mdName = mdoStringParts[2];

        MetadataObjectId mdoID =
            MetadataObjectId.createMetadataObjectId(mdType, mdPackage, mdName);
        return mdoID;
    }

    /**
     * this changes the format used in this class for job definitions to the one
     * which will be used in the runtime query.
     * @param strMetadataObject
     * @return string representing object in runtime store
     * @throws ServletException
     */
    private String fixMetadataString(String strMetadataObject)
        throws ServletException {
        String fslash = "/";
        String[] mdoStringParts =
            strMetadataObject.split(Pattern.quote(MDO_SEP));
        if (mdoStringParts.length != 3) {
            throw new ServletException(String.format("Unexpected number of
components %d found " +
                                                     "when converting %s to
MetadataObjectID",
                                                     mdoStringParts.length,
                                                     strMetadataObject));
        }
        String[] trimStringParts = new String[mdoStringParts.length];
        for (int i = 0; i < mdoStringParts.length; i++) {
            String mdoStringPart = mdoStringParts[i];
            trimStringParts[i] = mdoStringPart.replaceAll(fslash, " ").trim();
        }

        MetadataObjectType mdType =
            MetadataObjectType.getMOType(trimStringParts[0]);
        String mdPackage = fslash + trimStringParts[1];
        String mdName = trimStringParts[2];
        MetadataObjectId metadataObjId =
            MetadataObjectId.createMetadataObjectId(mdType, mdPackage, mdName);
        return metadataObjId.toString();
    }

    private Set<String> getSetFromMetadataEnum(Enumeration<MetadataObjectId>
enumMetadata)
        throws ServletException {
        Set<String> stringSet = new HashSet<String>();

        while (enumMetadata.hasMoreElements()) {
            MetadataObjectId objId = enumMetadata.nextElement();
            String strNamePart = objId.getNamePart();
            stringSet.add(strNamePart);
        }
        return stringSet;
    }

    //************************************************************************
    //
    //   HTML Rendering Methods
```

```
    //
    //*********************************************************************

    /**
     * Rendering code for the page displayed.
     * In a real application this would be done using JSP, but this approach
     * keeps everything in one file to make the example easier to follow.
     * @param response The response object from the main request.
     * @param message Text that will appear in the message panel, may contain HTML
     * @throws IOException
     */
    private void renderResponse(MetadataLists ml,
                                RuntimeLists rl,
                                HttpServletRequest request,
                                HttpServletResponse response,
                                String message)
        throws IOException, ServletException
    {
        response.setContentType(CONTENT_TYPE);
        PrintWriter out = response.getWriter();

        String urlBase = request.getContextPath() + request.getServletPath();

        // Indents maintained for clarity
        out.println("<html>");
        out.println("<head><title>EssDemo</title></head>");
        out.println("<body>");
        out.println("<table align=\"center\"><tbody>");
        out.println("  <tr><td align=\"center\"><h1>Enterprise Scheduler Service
Tutorial</h1></td></tr>");
        out.println("  <tr><td align=\"center\"><table cellspacing=6><tr>");

        // Job launch form
        out.println("    <td align=\"center\">");
        out.println("      <h2>Launch Job</h2>");
        renderLaunchJobForm(ml, out, urlBase);
        out.println("    </td>");

        out.println("    <td align=\"center\" bgcolor=\"blue\" width=\"2\"/>");

        out.println("  </tr></table></td></tr>");

        out.println("  <tr><td bgcolor=\"red\"/></tr>");

        // Message panel
        out.println("      <tr><td align=\"center\"><h3>Messages</h3></td></tr>");
        out.println("      <tr><td>");
        out.println(message);
        out.println("    </td></tr>");

        out.println("  <tr><td bgcolor=\"red\"/></tr>");

        // Request status
        out.println("  <tr><td align=\"center\">");
        out.println("    <form name=\"attrs\" action=\"" + urlBase +
                    PATH_ALTER + "\" method=\"post\">");
        out.println("    <h2>Request Status</h2>");
        out.println("    <table border=2><tbody>");
        out.println("          <th>reqID</th>");
        out.println("          <th>Description</th>");
```

```
        out.println("          <th>Scheduled time</th>");
        out.println("          <th>State</th>");
        out.println("          <th>Action</th>");

        renderStatusTable(out, rl.requestDetails);

        out.println("    </tbody></table>");
        out.println("    </form>");
        out.println("  </td></tr>");
        out.println("</tbody></table>");
        out.println("</body></html>");
        out.close();
    }

    private void renderLaunchJobForm(MetadataLists ml, PrintWriter out, String
urlBase)
        throws ServletException {
        out.println("      <form name=\"attrs\" action=\"" + urlBase +
                    PATH_SUBMIT + "\" method=\"post\">");
        out.println("        <table><tbody>");
        out.println("          <tr><td align=\"right\">");
        out.println("            <b>Job:</b>");
        out.println("            <select name=\"job\">");

        renderMetadataChoices(out, ml.jobDefList, false);
        renderMetadataChoices(out, ml.jobSetList, false);

        out.println("            </select>");
        out.println("          </td></tr>");
        out.println("          <tr><td align=\"right\">");
        out.println("            <b>Schedule:</b>");
        out.println("            <select name=\"schedule\">");

        renderPseudoScheduleChoices(out);
        renderMetadataChoices(out, ml.scheduleList, false);

        out.println("            </select>");
        out.println("          </td></tr>");
        out.println("          <tr><td align=\"center\">");
        out.println("            <input name=\"submit\" value=\"Submit\"
type=\"submit\">");
        out.println("          </td></tr>");
        out.println("        </tbody></table>");
        out.println("      </form>");
    }

    /**
     *
     * @param out - printwriter
     * @param jobChoices --  metadata to be displayed
     * @param bBlankFirst -- blank first (so that this param is not required)
     * @throws ServletException
     */
    private void renderMetadataChoices(PrintWriter out,
                                       Enumeration<MetadataObjectId> jobChoices,
                                       boolean bBlankFirst)
        throws ServletException
    {
        if (jobChoices == null)
            return;
```

```
    boolean bFirst = true;
    while (jobChoices.hasMoreElements()) {
        MetadataObjectId job = jobChoices.nextElement();
        String strJob = metadataObjectIdToString(job);
        String strNamePart = job.getNamePart();
        if (strNamePart.compareTo("BatchPurgeJob") == 0) {
            continue;
        } else {
            if (bFirst && bBlankFirst) {
                out.printf("<option value=\"%s\">%s</option>", "", "");
                bFirst = false;
            }
            out.printf("<option value=\"%s\">%s</option>", strJob,
                       strNamePart);
        }
    }
}

/**
 * helper method for rendering choices based on strings, adding an empty
 * string to the beginning of the list
 * @param out
 * @param choices
 */
private void renderStringChoices(PrintWriter out, Set<String> choices) {
    if (choices == null)
        return;

    choices.add("");
    SortedSet<String> sorted = new TreeSet<String>(choices);
    Iterator choiceIter = sorted.iterator();
    while (choiceIter.hasNext()) {
        String choice = (String)choiceIter.next();

        out.printf("<option value=\"%s\">%s</option>", choice, choice);
    }
}

private void renderPseudoScheduleChoices(PrintWriter out) {
    for (PseudoScheduleChoices c : PseudoScheduleChoices.values()) {
        out.printf("<option value=\"%s\">%s</option>", c, c);
    }
}

private void renderStatusTable
    (PrintWriter out, List<RequestDetail> reqDetails)
{
    if (reqDetails == null) {
        return;
    }

    for (RequestDetail reqDetail : reqDetails) {
        State state = reqDetail.getState();

        Calendar scheduledTime = reqDetail.getScheduledTime();
        String scheduledTimeString = null;

        if (scheduledTime == null) {
            scheduledTimeString = "null scheduled time";
```

```
            } else {
                scheduledTimeString = String.valueOf(scheduledTime.getTime());
            }

            final String actionButton;
            if (!state.isTerminal()) {
                String action = ACTION_CANCEL;
                String reqId = String.valueOf(reqDetail.getRequestId());
                actionButton = String.format
                    ("<button type=submit value=%s name=\"%s\">%s</button>",
                     action, reqId, action);
            } else {
                actionButton = " ";
            }


out.printf("<tr><td>%d</td><td>%s</td><td>%s</td><td>%s</td><td>%s</td></tr>\n",
                    reqDetail.getRequestId(), reqDetail.getDescription(),
                    scheduledTimeString, state, actionButton);
        }
    }


    private MetadataService getMetadataService() throws Exception {
        return JndiUtil.getMetadataServiceEJB();
    }

    private RuntimeService getRuntimeService() throws Exception {
        return JndiUtil.getRuntimeServiceEJB();
    }

    private abstract class Payload<SERVICE, HANDLE, RETURN> {
        abstract SERVICE getService() throws Exception;
        abstract HANDLE getHandle(SERVICE service) throws Exception;
        abstract void closeHandle(SERVICE service,
                                  HANDLE handle,
                                  boolean abort)
            throws Exception;
        abstract RETURN execute(SERVICE service, HANDLE handle, Logger logger)
            throws Exception;
    }

    private abstract class MetadataServicePayload<T>
        extends Payload<MetadataService, MetadataServiceHandle, T>
    {
        @Override
        MetadataService getService() throws Exception {
            return getMetadataService();
        }

        @Override
        MetadataServiceHandle getHandle(MetadataService service)
            throws Exception
        {
            return service.open();
        }

        @Override
        void closeHandle(MetadataService service,
                         MetadataServiceHandle handle,
```

```
                    boolean abort)
        throws Exception
    {
        service.close(handle, abort);
    }
}

private abstract class RuntimeServicePayload<T>
    extends Payload<RuntimeService, RuntimeServiceHandle, T>
{
    @Override
    RuntimeService getService() throws Exception {
        return getRuntimeService();
    }

    @Override
    RuntimeServiceHandle getHandle(RuntimeService service)
        throws Exception
    {
        return service.open();
    }

    @Override
    void closeHandle(RuntimeService service,
                     RuntimeServiceHandle handle,
                     boolean abort)
        throws Exception
    {
        service.close(handle, abort);
    }
}

private <S, H, R> R performOperation
        (Payload<S, H, R> payload, Logger logger)
    throws Exception
{
    S service = payload.getService();
    H handle = payload.getHandle(service);

    Exception origException = null;
    try {
        return payload.execute(service, handle, logger);
    } catch (Exception e2) {
        origException = e2;
        throw e2;
    } finally {
        if (null != handle) {
            try {
                boolean abort = (null != origException);
                payload.closeHandle(service, handle, abort);
            } catch (Exception e2) {
                if (null != origException) {
                    logger.log(Level.WARNING, "An error occurred while " +
                        "closing handle, however, a previous failure was " +
                        "detected.  The following error will be logged " +
                        "but not reported: " + stackTraceToString(e2));
                }
            }
        }
    }
```

```
}

private final String stackTraceToString(Exception e) {
    StringWriter sw = new StringWriter();
    PrintWriter pw = new PrintWriter(sw);
    e.printStackTrace(pw);
    pw.flush();
    pw.close();
    return sw.toString();
}

private Logger getLogger() {
    return Logger.getLogger(this.getClass().getName());
}

private class MetadataLists {
    private final Enumeration<MetadataObjectId> jobDefList;
    private final Enumeration<MetadataObjectId> jobSetList;
    private final Enumeration<MetadataObjectId> scheduleList;
    private final Enumeration<MetadataObjectId> jobTypeList;

    private MetadataLists(Enumeration<MetadataObjectId> jobDefList,
                          Enumeration<MetadataObjectId> jobSetList,
                          Enumeration<MetadataObjectId> scheduleList,
                          Enumeration<MetadataObjectId> jobTypeList)
    {
        this.jobDefList   = jobDefList;
        this.jobSetList   = jobSetList;
        this.scheduleList = scheduleList;
        this.jobTypeList  = jobTypeList;
    }
}

private class RuntimeLists {
    private final List<RequestDetail> requestDetails;
    private final Set<String> applicationChoices;
    private final Set<String> stateChoices;
    private final Set<MetadataObjectId> jobDefMDOChoices;

    private RuntimeLists(List<RequestDetail> requestDetails,
                         Set<String> applicationChoices,
                         Set<String> stateChoices,
                         Set<MetadataObjectId> jobDefMDOChoices)
    {
        super();
        this.requestDetails = requestDetails;
        this.applicationChoices = applicationChoices;
        this.stateChoices = stateChoices;
        this.jobDefMDOChoices = jobDefMDOChoices;
    }
}

/**
 * Retrieve lists of jobs, schedules, and status for use by the renderer
 * @throws ServletException
 */
private MetadataLists getMetadataLists() throws Exception {
    Logger logger = getLogger();

    MetadataServicePayload<MetadataLists> myPayload =
```

```
            new MetadataServicePayload<MetadataLists>()
        {
            @Override
            MetadataLists execute(MetadataService service,
                                  MetadataServiceHandle handle,
                                  Logger logger)
                throws Exception
            {
                Enumeration<MetadataObjectId> jobDefs =
                    service.queryJobDefinitions(handle, null, QueryField.NAME,
true);
                Enumeration<MetadataObjectId> jobSets =
                    service.queryJobSets(handle, null, QueryField.NAME, true);
                Enumeration<MetadataObjectId> schedules =
                    service.querySchedules(handle, null, QueryField.NAME, true);
                Enumeration<MetadataObjectId> jobTypes =
                    service.queryJobTypes(handle, null, QueryField.NAME, true);

                return new MetadataLists(jobDefs, jobSets, schedules, jobTypes);
            }
        };
        MetadataLists ml = performOperation(myPayload, logger);
        return ml;
    }

    private RuntimeLists getRuntimeLists() throws Exception {
        Logger logger = getLogger();

        RuntimeServicePayload<List<RequestDetail>> myPayload2 =
            new RuntimeServicePayload<List<RequestDetail>>()
        {
            @Override
            List<RequestDetail> execute(RuntimeService service,
                                        RuntimeServiceHandle handle,
                                        Logger logger)
                throws Exception
            {
                List<RequestDetail> reqDetails =
                    new ArrayList<RequestDetail>(10);
                Enumeration requestIds = service.queryRequests
                    (handle, null, RuntimeService.QueryField.REQUESTID, true);

                while (requestIds.hasMoreElements()) {
                    Long reqId = (Long)requestIds.nextElement();
                    RequestDetail detail = service.getRequestDetail(handle,
reqId);
                    reqDetails.add(detail);
                }

                return reqDetails;
            }
        };
        List<RequestDetail> reqDetails = performOperation(myPayload2, logger);
        RuntimeLists rl = getRuntimeLists(reqDetails);
        return rl;
    }

    private RuntimeLists getRuntimeLists(List<RequestDetail> reqDetails) {
        Set<String> applicationSet = new HashSet<String>(10);
        Set<String> stateSet = new HashSet<String>(10);
```

```
        Set<MetadataObjectId> jobDefMOSet = new HashSet<MetadataObjectId>(10);

        if (reqDetails != null) {
            ListIterator detailIter = reqDetails.listIterator();
            while (detailIter.hasNext()) {
                RequestDetail detail = (RequestDetail)detailIter.next();
                applicationSet.add(detail.getDeployedApplication());
                State state = detail.getState();
                if (state.isTerminal())
                    stateSet.add(state.name());
                jobDefMOSet.add(detail.getJobDefn());
            }
        }

        RuntimeLists rl = new RuntimeLists
            (reqDetails, applicationSet, stateSet, jobDefMOSet);
        return rl;
    }

}
```

3.9.2.6 Editing the web.xml File for the Frontend Submitter Application

You need to edit the `web.xml` file to and Oracle Enterprise Scheduler metadata and runtime EJB references.

To edit the web.xml file for the frontend submitter application:

1. In the Application Navigator, expand **SuperWeb**, expand **Web Content**, expand **WEB-INF** and double-click **web.xml**.
2. In the overview editor, click the **References** navigation tab and expand the **EJB References** section.
3. Add two EJB resources with the information shown in Table 3–1.

Table 3–1 EJB Resources for the Frontend Submitter Application

EJB Name	Interface Type	EJB Type	Local/Remote Interface
`ess/metadata`	Local	Session	`oracle.as.scheduler.MetadataServiceLocal`
`ess/runtime`	Local	Session	`oracle.as.scheduler.RuntimeServiceLocal`

4. Click the **Servlets** navigation tab and click the **Servlet Mappings** tab.
5. Change the `/essdemoappservlet` URL pattern to `/essdemoappservlet/*`.

3.9.2.7 Editing the weblogic-application.xml file for the Frontend Submitter Application

You need to create and edit the weblogic-application.xml file.

To edit the weblogic-application.xml file for the frontend submitter application:

1. In Application Navigator, right-click the **SuperEss** project and select **New**.
2. In the New Gallery, expand **General**, select **Deployment Descriptors** and then **Weblogic Deployment Descriptor**, and click **OK**.
3. In the Select Descriptor page select **weblogic-application.xml**.
4. Click **Next**, click **Next** again, and click **Finish**.

5. In the source editor, replace the contents of the `weblogic-application.xml` file that you just created with the XML shown in Example 3–12.

Example 3–12 Contents to Copy to weblogic-application.xml for a Frontend Submitter Application

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<weblogic-application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.bea.com/ns/weblogic/weblogic-application

http://www.bea.com/ns/weblogic/weblogic-application/1.0/weblogic-application.xsd"
                     xmlns="http://www.bea.com/ns/weblogic/weblogic-application">

    <!-- The following application parameter tells JPS which stripe it should
       - use to upload the jazn-data.xml policy.  If this parameter is not
       - specified, it will use the Java EE deployment name plus the version
       - number (e.g. EssDemoApp#V2.0).
      -->
    <application-param>
        <param-name>jps.policystore.applicationid</param-name>
        <param-value>EssDemoAppUI</param-value>
    </application-param>

    <!-- This listener allows JPS to configure itself and upload the
       - jazn-data.xml policy to the appropriate stripe
      -->
    <listener>

<listener-class>oracle.security.jps.wls.listeners.JpsApplicationLifecycleListener<
/listener-class>
    </listener>

    <!-- This listener allows MDS to configure itself and upload any metadata
       - as defined by the MAR profile and adf-config.xml
      -->
    <listener>

<listener-class>oracle.mds.lcm.weblogic.WLLifecycleListener</listener-class>
    </listener>

    <!-- This listener allows Oracle Enterprise Scheduler to configure itself
      -->
    <listener>

<listener-class>oracle.as.scheduler.platform.wls.deploy.ESSApplicationLifecycleLis
tener</listener-class>
    </listener>

    <!-- This shared library contains all the Oracle Enterprise Scheduler classes
      -->
    <library-ref>
        <library-name>oracle.ess.client</library-name>
    </library-ref>
    <library-ref>
        <library-name>adf.oracle.domain</library-name>
    </library-ref>
</weblogic-application>
```

3.9.2.8 Editing the adf-config file for the Frontend Submitter Application

You need to edit the adf-config.xml file to tell the application to share the metadata that was created in the hosting application.

To edit the adf-config.xml file for the frontend submitter application:

1. From the Application Resources panel, expand **Descriptors**, expand **ADF META-INF**, and double-click **adf-config.xml**.
2. In the source editor, replace the contents of the `adf-config.xml` file with the XML shown in Example 3–13.

Example 3–13 Contents to Copy to adf-config.xml for a Frontend Submitter Application

```
<?xml version="1.0" encoding="UTF-8" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config">
  <adf-security-child xmlns="http://xmlns.oracle.com/adf/security/config">
    <JaasSecurityContext
initialContextFactoryClass="oracle.adf.share.security.JAASInitialContextFactory"

jaasProviderClass="oracle.adf.share.security.providers.jps.JpsSecurityContext"
                        authorizationEnforce="false"
                        authenticationRequire="true"/>
  </adf-security-child>
  <adf-mds-config xmlns="http://xmlns.oracle.com/adf/mds/config">
    <mds-config xmlns="http://xmlns.oracle.com/mds/config" version="11.1.1.000">
      <persistence-config>
        <metadata-namespaces>
          <namespace metadata-store-usage="ess_shared_metadata"
path="/oracle/apps/ess/howto"/>
        </metadata-namespaces>
        <metadata-store-usages>
          <metadata-store-usage default-cust-store="false" deploy-target="false"
id="ess_shared_metadata"/>
        </metadata-store-usages>
      </persistence-config>
    </mds-config>
  </adf-mds-config>
</adf-config>
```

3.9.2.9 Assembling the Frontend Submitter Application for Oracle Enterprise Scheduler

After you create the frontend sample application you use Oracle JDeveloper to assemble the application.

To assemble the backend application you do the following:

- Create the EJB Java Archive
- Create the WAR file
- Create the application MAR and EAR files

3.9.2.9.1 How to Assemble the EJB JAR File for the Frontend Submitter Application

The EJB Java archive file includes descriptors for the Java job implementations.

To assemble the EJB JAR File for the frontend submitter application:

1. In Application Navigator, right-click the **SuperEss** project and choose **New**.

2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **EJB JAR File**, and click **OK**.
3. In the Create Deployment Profile dialog, set the **Deployment Profile Name** to `JAR_SuperEssEjbJar`.
4. On the Edit EJB JAR Deployment Profile Properties dialog, click **OK**.
5. On the Project Properties dialog, click **OK**.

3.9.2.9.2 How to Assemble the WAR File for the Frontend Submitter Application

You need to create a web archive file for the web application.

To assemble the WAR file for the frontend submitter application

1. In Application Navigator, right-click the **SuperWeb** project and choose **New**.
2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **WAR File**, and click **OK**.
3. In the Create Deployment Profile dialog, set the **Deployment Profile Name** to `WAR_SuperWebWar`.
4. On the Edit WAR Deployment Profile Properties dialog, click the **General** navigation tab, select **Specify Java EE Web Context Root**, and enter `ESSDemoApp`.
5. Click **OK**.
6. On the Project Properties dialog, click **OK**.

3.9.2.9.3 How to Assemble the MAR and EAR Files for the Frontend Hosting Application

The sample application needs to contain the MAR profile and the EAR file that assembles the scheduler backend application.

To create the MAR and EAR files for the frontend submitter application:

1. From the main menu, choose **Application Menu > Application Properties...**
2. In the Application Properties dialog, click the **Deployment** navigation tab and click **New**.
3. In the Create Deployment Profile dialog, select **MAR File** from the **Archive Type** dropdown list.
4. In the **Name** field, enter `MAR_EssDemoAppUIMar` and click **OK**.
5. Click **OK**.
6. In the Deployment page of the Application Properties dialog, click **New**.
7. In the Create Deployment Profile dialog, select **EAR File** from the **Archive Type** dropdown list.
8. In the **Name** field, enter `EAR_EssDemoAppUIEar` and click **OK**.
9. In the Edit EAR Deployment Profile dialog, click the **General** navigation tab and enter `EssDemoAppUI` in the **Application Name** field.
10. Click the **Application Assembly** navigation tab, then select `MAR_ESSDemoAppUIMar` and select `JAR_SuperEssEjbJar`.
11. Click **OK**.
12. In the Application Properties dialog, click **OK**.

3.9.2.10 Deploying the Backend Hosting Application

After assembling the application, you can deploy it to the server.

To deploy the backend hosting application:

1. From the main menu, choose **Application > Deploy > EAR_EssDemoAppEar...**
2. Set up and deploy the application to a container.
3. On the Deployment Configuration dialog, there should be two entries in the **Shared Metadata Repositories** panel. Find the shared repository mapped to the `/oracle/apps/ess/howto namespace`. Change its partition to the partition used when deploying `EssDemoApp`. If you used the default value, this should be `EssDemoApp_V2.0`.
4. Click **OK**.

4

Using the Metadata Service

This chapter describes how to use the Oracle Enterprise Scheduler Metadata Service. The Metadata Service allows you to save schedules, job definitions, and other Oracle Enterprise Scheduler metadata to a repository. You can also use the Metadata Service query methods to list objects stored in metadata.

This chapter includes the following sections:

For information about how to create job definitions, see the following chapters: Chapter 3, "Use Case Oracle Enterprise Scheduler Sample Application", Chapter 6, "Creating and Using PL/SQL Jobs", and Chapter 7, "Creating and Using Process Jobs".

4.1 Introduction to Using the Metadata Service

Oracle Enterprise Scheduler provides the Metadata Service and exposes it to your application program as a Stateless Session Enterprise Java Bean (EJB). The Metadata Service allows you to save Oracle Enterprise Scheduler application level metadata objects. The Metadata Service uses Oracle Metadata Services (MDS) to save metadata objects to a repository (the repository can be either database based or file based). The Metadata Service allows you to reuse application-level metadata across multiple job request submissions.

Oracle Enterprise Scheduler metadata objects include the following:

- Application Level Metadata: You use the Metadata Service to store job type, job definition, job set, and other application-level metadata object definitions for job requests.
- Default (global) Oracle Enterprise Scheduler Metadata: The global Oracle Enterprise Scheduler metadata includes administrative objects such as schedules, workshifts and work assignments. Oracle Enterprise Scheduler provides `MetadataServiceMXBean` and the `MetadataServiceMXBeanProxy` to access and store default administrative objects

> **Note:** Oracle Enterprise Scheduler Schedule objects are used both in application level metadata and in global metadata.

Access to application level metadata objects is exposed only with the `MetadataService` interface. The `MetadataService` is exposed as a stateless session EJB. External clients must access the service only through the corresponding EJB. Clients should not interact with the internal API layer directly. When an application client uses the metadata service through the stateless session EJB, all the methods in this interface accept a reference to a MetadataServiceHandle argument, which stores state across multiple calls, for example when multiple methods are to be called within a user transaction. The `MBeanProxy` interface does not require a handler.

In an Oracle Enterprise Scheduler application you do not need to access or manipulate the MetadataServiceHandle. The application just needs to hold on to the reference created by the open method and pass it in methods being called. Finally the handle must explicitly be closed by calling the close method. Only upon calling the close method, any changes made using a given handle are committed (or aborted).

Metadata object names must be unique within the scope of a given package or namespace. Within a given package, two metadata objects with the same name, and of the same type cannot be created.

4.1.1 Introduction to Metadata Service Namespaces

Each Oracle WebLogic Server domain generally includes one metadata repository. A metadata repository is divided into a number of partitions, where each partition is independent and isolated from the others in the repository.

Each application can choose which partition to use. Two applications can also choose to share a partition.

Within a partition, you can organize the data in any way. Usually, the data is organized hierarchically like the file system of an operating system. Where a file system uses folders or directories, the Metadata Service uses namespaces or package names which form a unique name used to locate a file. In the context of Oracle Fusion Applications, all data related to Oracle Enterprise Scheduler must be stored in a partition called `globalEss` with the namespace `/oracle/apps/ess`.

Each Fusion Applications product family—SCM, HCM, CRM, and so on—has a separate namespace under `/oracle/apps/ess`, such as `/oracle/apps/ess/scm`, `/oracle/apps/ess/hcm`, `/oracle/apps/ess/crm`, and so on.

For all other Oracle Enterprise Scheduler applications, the application name and an optional package name containing the application level metadata displays under the namespace `/oracle/apps/ess`. For example, the metadata repository for an application named `application1` can be divided into packages with the names `dev`, `test`, and `production`.

The metadata repository for this application has the following structure:

```
/oracle/apps/ess/application1/dev/metadata
/oracle/apps/ess/application1/test/metadata
/oracle/apps/ess/application1/production/metadata
```

Each Metadata Service method that creates a metadata object takes a required `packageName` argument that specifies the package part of the directory structure.

4.1.2 Introduction to Metadata Service Operations

After you access an Oracle Enterprise Scheduler metadata repository you can perform different types of Metadata Service operations, including:

- Add, Update, Delete: These operations have transactional characteristics.

- Copy: These operations have transactional characteristics.
- Query: These operations have read-only characteristics and let you list metadata objects in the metadata repository.
- Get: These operations have either read-only or transactional characteristics, depending on the value of the `forUpdate` flag.

4.1.3 Introduction to Metadata Service Transactions

Because clients access the Metadata Service through a Stateless Session EJB, each method uses a reference to a `MetadataServiceHandle` argument; this argument stores state for Metadata Service operations. The Metadata Service `open()` method begins each Oracle Enterprise Scheduler metadata repository user transaction. In an Oracle Enterprise Scheduler application client you obtain a `MetadataServiceHandle` reference with the `open()` method and you pass the reference to subsequent Metadata Service methods. The `MetadataServiceHandle` reference provides a connection to the metadata repository for the calling application.

In a client application that uses the Metadata Service you must explicitly close a Metadata Service transaction by calling `close()`. This ends the transaction and causes the transaction to be committed or rolled back (undone). The `close()` not only controls the transactional behavior within the Metadata Service, but it also allows Oracle Enterprise Scheduler to release certain resources. Thus, the `close()` is also required for Metadata Service read-only `query()` and get() operations.

Note: The Metadata Service does not support JTA global transactions, but you can still make Metadata Service calls in the boundary of your transactions. While you can make Metadata Service calls in bean/container managed transactions, the calls will not be part of your transaction.

4.2 Accessing the Metadata Service

There are several ways to access the Metadata Service, including:

- Stateless Session EJB access: Use this type of access with Oracle Enterprise Scheduler user applications.
- MBean access: This access is intended for use by administrative applications that perform administrative functions using the `oracle.as.scheduler.management` APIs.
- MBean proxy access: This access is intended for use by administrative applications that perform administrative functions using the `oracle.as.scheduler.management` APIs. Use the MBean proxy if the administrative client is remote to the Oracle Enterprise Scheduler.

4.2.1 How to Access the Metadata Service with a Stateless Session EJB

User applications use a Stateless Session EJB to access the Metadata Service for application level metadata operations. Using JNDI you can lookup the Metadata Service associated with an Oracle Enterprise Scheduler application.

Example 4–1 shows the JNDI lookup for the Oracle Enterprise Scheduler Metadata Service that allows you to use application level metadata. Note that the `getMetadataServiceEJB()` method looks up the metadata service using the name

"ess/metadata". By convention, Oracle Enterprise Scheduler applications use "ess/metadata" for the EJB reference to the `MetadataServiceBean`.

Example 4–1 JNDI Lookup for Stateless Session EJB Access to Metadata Service

```
// Demonstration on how to lookup metadata service from a Java EE application
// JNDI lookup on the metadata service EJB

import oracle.as.scheduler.core.JndiUtil;

MetadataService ms = JndiUtil.getMetadataServiceEJB();
```

4.3 Accessing the Metadata Service with Oracle JDeveloper

Using Oracle JDeveloper at design time you can create, view, and update application level metadata objects.

4.4 Querying Metadata Using the Metadata Service

The Metadata Service query methods let you view objects in the metadata repository. You can query job types with the `queryJobTypes()` method, query job definitions with `queryJobDefinitions()` method, and likewise you can query other metadata objects using the corresponding `MetadataService` query method.

Associated with a query you can use a filter to restrict the output to obtain only items of interest (in a manner similar to using a SQL `WHERE` clause).

4.4.1 How to Create a Filter

A filter specifies a comparison or a criteria for a query. You create a filter by creating a comparison that includes a `field` argument (`String`), a `comparator`, and an associated `value` (`Object`). In a filter, you can use the filter methods to combine comparisons to form filter expressions.

Table 4–1 lists the comparison operators (`comparator` argument).

Table 4–1 Filter Comparison Operators

Comparison Operator	Description
`CONTAINS`	Field contains the specified value
`ENDS_WITH`	Field ends with the specified value
`EQUALS`	Field equals the specified value
`GREATER_THAN`	Field is greater than the specified value
`GREATER_THAN_EQUALS`	Field is greater than or equal to the specified value
`LESS_THAN`	Field is less than the specified value
`LESS_THAN_EQUALS`	Field is less than or equal to the specified value
`NOT_CONTAINS`	Field does not contain the specified value
`NOT_EQUALS`	Field does not equal the specified value
`STARTS_WITH`	Field starts with the specified value

Example 4–2 shows code that creates a new filter.

Example 4–2 Creating a Filter with a Filter Comparator for a Query

```
Filter filter =
  new Filter(MetadataService.QueryField.PACKAGE.fieldName(),
  Filter.Comparator.NOT_EQUALS, null);
```

Table 4–2 MetadataService Query Fields

Query Field	Description
`MetadataService.QueryField.PACKAGE`	The name of the package.
`MetadataService.QueryField.NAME`	The job definition name.
`MetadataService.QueryField.JOBTYPE`	The job type associated with the job definition.
`MetadataService.QueryField.EXECUTIONTYPE`	The type of job execution, synchronous or asynchronous.
`MetadataService.QueryField.EXECUTIONMODE`	The mode of job set execution, parallel or serial.
`MetadataService.QueryField.FIRSTSTEP`	The first step in a job set.
`MetadataService.QueryField.ACTIVE`	Indicates whether a work assignment is active.
`MetadataService.QueryField.PRODUCT`	Indicates the name of the product with which the job is associated.
`MetadataService.QueryField.EFFECTIVEAPPLICATION`	The name of the hosting application wherein this job should run.

4.4.2 How to Query Metadata Objects

A `MetadataService` query returns an enumeration list of `MetadataObjectIDs` of the form:

```
java.util.Enumeration<MetadataObjectId>
```

Example 4–3 shows a sample routine that queries for a list of job types in the metadata.

Example 4–3 Using Metadata Service Query Methods

```
Enumeration<MetadataObjectId> qryResults
           = m_service.queryJobTypes(handle, filter, null, false);
```

Example 4–3, shows the following important steps for using the `queryJobTypes()` method:

- You need to supply a reference to a metadata repository by obtaining an instance of `MetadataServiceHandle`.
- You need to create a filter for the query. The filter contains the fields, comparators, and values to search for.
- You determine the field to sort by in the query using the `orderBy` argument, or you set the `orderBy` argument to null to indicate that no specific ordering is applied.
- You set the `ascending` argument for the query. When ordering is applied setting the `ascending` argument to `true` indicates ascending order or `false` indicates descending order for the result list.

5

Using Parameters and System Properties

Using Oracle Enterprise Scheduler you can define parameters in the metadata service and you can define new parameters or provide new values for existing parameters in the runtime service when you submit a job request. A given parameter may represent a value for an Oracle Enterprise Scheduler system property or a value for an application-specific parameter.

This chapter includes the following sections:

5.1 Introduction to Using Parameters and System Properties

You can define Oracle Enterprise Scheduler parameters as follows:

- In metadata associated with a job definition, a job type, or a job set.
- In the request parameters when a job request is submitted. A request parameter can override a parameter specified in metadata or can specify a value for a parameter not previously defined in the metadata associated with a job request (subject to certain constraints). You can also add new parameters or update parameter values (subject to certain constraints) after a job request has been submitted.

Oracle Enterprise Scheduler System Properties are parameters with names that Oracle Enterprise Scheduler reserves. For some system properties Oracle Enterprise Scheduler also defines the values or provides a default value if you do not specify a value. For more information on the Oracle Enterprise Scheduler system properties, see Section 5.4, "Using System Properties".

5.1.1 What You Need to Know About Parameter and System Property Naming

Oracle Enterprise Scheduler parameters and system properties are case sensitive. For example the parameter name `USER_PARA` and `user_para` represent different parameters in Oracle Enterprise Scheduler.

When you use parameters note that Oracle Enterprise Scheduler reserves the names starting with "SYS_" (case-insensitive) for Oracle Enterprise Scheduler defined system properties. Thus, you should not use application-level parameters with names that start with "SYS_" (case-insensitive).

5.1.2 What You Need to Know About Parameter Conflict Resolution and Parameter Materialization

When submitting a job request, Oracle Enterprise Scheduler combines parameters specified in the job metadata with any submission parameters to form the runtime request parameters. The runtime parameters are saved to the database runtime store and used for subsequent processing of the request. The metadata parameters are obtained from the job definition, job type, and if applicable, the job set as they are defined in the metadata repository at the time of submission. Any subsequent changes to the metadata is normally not seen or used as the request is processed. Oracle Enterprise Scheduler resolves parameter conflicts for parameters with the same name associated with the job metadata or the submit parameters.

A parameter conflict can occur in the following cases:

- A parameter is defined repeatedly with different values. For example if the `SystemProperty.PRIORITY` property is set with different values in the job type and in the job definition associated with a request.
- A parameter is defined repeatedly and at least one definition is specified as read-only with the `ParameterInfo readonly` flag set to `true`.

To resolve conflicts with parameters Oracle Enterprise Scheduler uses one of the following conflict resolution models and the parameter value inheritance hierarchy shown in Table 5–1:

- *Last definition wins*: used when the same parameter is defined repeatedly with the `readonly` flag set to `false` in all cases. In the *last definition wins* model, conflicts are resolved according to the precedence rules where the highest level wins (last definition). For example a property specified at the job request level wins over the same property specified at the job definition level.
- *First read-only definition wins*: used when the same parameter is defined repeatedly and at least one definition is read-only (the `ParameterInfo readonly` flag is set to `true`.) In the *first read-only definition wins* model, parameter conflicts are resolved according to the precedence rules shown in Table 5–1, lowest level wins. For example a readonly parameter specified at the job type definition level wins over the same property specified at the job definition level, read-only or not.

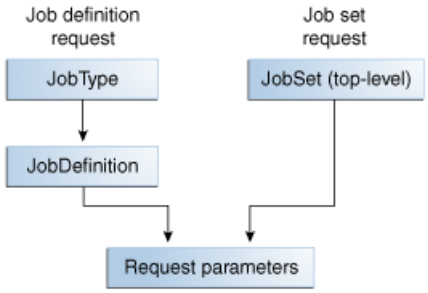
Table 5–1 Parameter Precedence Levels

Object	Level
`JobType`	1 - Lowest Level
`JobDefinition`	2
job set step	3
`job set`	4
Job request (via `RequestParameters` passed to `submitRequest()`)	5 - Highest Level

5.1.2.1 What You Need to Know About Job Definition Parameter Materialization

Figure 5–1 illustrates the order of precedence taken by parameters defined in various components.

Figure 5–1 Parameter Precedence



In the case of a job request, the parameters defined by the job type take first precedence, followed by the parameters defined in the job definition. The parameters submitted with the job request take final precedence. In the case of a job set request, the parameters defined in the job set take first precedence, followed by the parameters defined by the job request run as a child of the job set.

5.1.2.2 What You Need to Know About Job Set Level Parameter Materialization

When the job set step parameters are materialized, if the job set defines any of the following system properties as read-only, and those properties are defined in the definition of the topmost job set, that is the job set of the absolute parent, the job set values will override the values set at the job set step level. This causes every definition, job definition or job set definition, that runs in the context of a specific job set to run with the same values.

PRIORITY

REQUEST_EXPIRATION

RETRIES, only if the step definition value is > 0

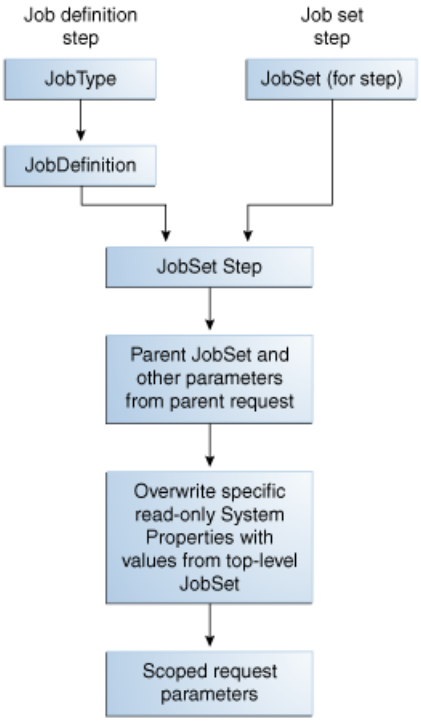
There is an exception for `RETRIES` because a value of 0 may mean that the job is not capable of being restarted. So if a step is defined with `RETRIES = 0`, it is not overridden, but if the step has `RETRIES > 0`, it will be overridden with the job set value.

Properties for a job set step request are materialized during the processing of a job set when the step is reached. Properties for a job step request are materialized in the following order.

1. Job type and job definition (if the step is a job definition) or job set (if the step is a job set).
2. Job set step.
3. Parent request properties and system properties (parent is step's parent job set).
4. Scoped request properties.

Figure 5–2 illustrates the parameter precedence for job set steps.

Figure 5–2 Parameter Precedence for Job Set Steps



When job sets include steps that are job sets, this is a nested job set. For a nested job set, the precedence shown in Table 5–1 applies. When a nested job set is reached, Oracle Enterprise Scheduler applies the parameters of the parent request and the parameters of the parent request follow the same precedence. The effect is that parameters of the parent request, job set and job set step are inherited by nested job sets.

5.2 Using Parameters with the Metadata Service

Oracle Enterprise Scheduler metadata includes parameters that you can associate with a metadata object. The parameters can include both application level parameters and system properties for a given definition (metadata object). An instance of the `ParameterList` class declares the parameters for a given job definition, job type or job set. To set parameters for a given job definition, job type, or job set definition, you can use a `ParameterList` object with the `setParameters()` method for the metadata object or you can use the constructor and supply a `ParameterList`. To supply parameter information in a parameter list, each `ParameterList` object includes `ParameterInfo` objects that represent parameters, such that each parameter is defined with properties as shown in Table 5–2.

Table 5–2 ParameterInfo Parameter Properties

Parameter Property	Description
Name	Specifies the parameter name.
Value	Specifies the parameter value.

Table 5–2 (Cont.) ParameterInfo Parameter Properties

Parameter Property	Description
Readonly	This boolean flag can be set for each parameter. This flag indicates whether the parameter is read-only. When `true`, subsequent objects in the parameter precedence hierarchy, such as request submission parameter, cannot change the parameter value. Typically a read-only parameter will have a default value that cannot be changed by subsequent objects. Note that the value of a read-only parameter can be changed in the object itself where this parameter is defined. For example if this parameter is defined in a job type as a read-only parameter, its value can be changed in the job type definition itself, but a job definition that uses the job type or a request submission parameter cannot override the value, subject to the conflict resolution rules specified for parameter values. For more information, see Section 5.1.2, "What You Need to Know About Parameter Conflict Resolution and Parameter Materialization".
Legacy	A boolean that specifies that a parameter should be visible when used in a GUI.
DataType	Values can only be one of the supported types, including: Boolean, Integer, Long, String, and `DATETIME` that represents a date as a `java.util.Calendar` object.

You can set parameters at different levels appropriate to parameter precedence rules for a job request. For example, you can set parameters that apply for a job type, a job definition, a job set, a job set step, or a request submission parameter. For information about the precedence rules, see Section 5.1.2, "What You Need to Know About Parameter Conflict Resolution and Parameter Materialization".

5.2.1 How to Use Parameters and System Properties in Metadata Objects

Example 5–1 shows code that uses a `ParameterList` to set parameter and system property values in a metadata object.

Example 5–1 Adding Parameters and System Properties in a Metadata Object

```
String name = "JobDescription_name";
MetadataObjectId jobtype;
.
.
.
JobDefinition jd = new JobDefinition(name, jobtype);
ParameterList parlist = new ParameterList();
parlist.add(SystemProperty.APPLICATION, "METADATA_UNITTEST_APP", false);
parlist.add(SystemProperty.PRODUCT, "METADATA_UNITTEST_PROD", false);
parlist.add(SystemProperty.CLASS_NAME, "oracle.as.scheduler.myself", false);
parlist.add(SystemProperty.RETRIES, "2", false);
parlist.add(SystemProperty.REQUEST_EXPIRATION, "60", false);
parlist.add("MyProp", "Value", false);
parlist.add("MyReadOnlyProp", "readyOnlyValue", true);
jd.setParameters(parlist);
```

Example 5–1, shows the following important steps for using parameters with a metadata object:

- You need a reference to a metadata service handle to create the metadata object where you want to add parameters.

- You need to use the `ParameterList` `add()` method to add parameter information.
- You can use a `SystemProperty` as the name for a parameter to specify a value for a system property.
- You can specify an application specific property by using a name that you define with the parameter information in a `ParameterList`.
- You need to use a metadata object `setParameters()` method to apply the parameters specified in the `ParameterList` to the metadata object. In this case, use the job definition `setParameters()` method.

5.3 Using Parameters with the Runtime Service

You can specify parameters when a job request is submitted by supplying a `RequestParameters` object with `submitRequest()`. A request parameter can override a parameter specified in metadata or can specify a value for a parameter not previously defined in the metadata associated with a job request (subject to certain constraints). You can also use the runtime service `setRequestParameter()` method to set or modify request parameters (subject to certain constraints) after the request has been submitted.

The `submitRequest()` method will validate each request parameter against its definition in the metadata, if one exists. Such validations include checking the data type of the parameter against the data type specified in the metadata, checking the read-only constraint for the parameter, and so on. If a given request parameter does not exist in the corresponding metadata, the data type for the parameter is determined by doing an instanceof on the parameter value. The data type of a request parameter value must be one of the supported types specified by `ParameterInfo.DataType`.

If the value of a request parameter is null and the property has not been assigned in the metadata, it defaults to the `STRING` data type when calling `submitRequest()`. Oracle Enterprise Scheduler assigns a null value to the parameter. As such, a parameter need not be assigned in the metadata.

The `RuntimeService` `setRequestParameter()` method, which is similar to `updateRequestParameter()`, allows a previously undefined request parameter to be set by a job during execution.

5.3.1 How to Use Parameters with the Runtime Service

When you submit a job request you set a parameter in a `RequestParameters` object. This parameter may represent an Oracle Enterprise Scheduler system property or an application-specific parameter. The `RequestParameters` parameter value may be used to override a parameter specified in metadata, or to specify the value for a parameter not previously defined in metadata associated with the job request.

Example 5–2 shows code using a `RequestParameters` object with the `add()` method to set a system property value.

Example 5–2 Using the PRIORITY System Property with Request Parameters

```
import oracle.as.scheduler.RequestParameters;
import oracle.as.scheduler.MetadataObjectId;
import oracle.as.scheduler.RuntimeService;
import oracle.as.scheduler.RuntimeServiceHandle;
import oracle.as.scheduler.SystemProperty;

RuntimeService runtime;
```

```
RuntimeServiceHandle rs_handle;
MetadataObjectId jobSetId;
int startsIn;
long requestID = 0L;

RequestParameters req_par = new RequestParameters();

req_par.add(SystemProperty.PRIORITY, new Integer(7));

Calendar start = Calendar.getInstance();
start.add(Calendar.SECOND, startsIn);

requestID =
     runtime.submitRequest(rs_handle,"My job set", jobSetId, start, req_par);
.
.
.
```

The example assumes that there is a user-created `runtimeServiceHandle` named `rs_handle`.

5.3.2 How to Use Parameters with a Step ID for Job Set Steps

The `RequestParameters` object is a container for all the parameters for a request. Some of the `RequestParameters` methods take a step ID as an argument. Such methods allow you to specify parameters for a job set at request submission, where parameters can be specified for, or scoped to, individual steps associated with a job set request. For such methods, the step ID argument identifies the step within the job set to which the given parameter applies. For non-job set requests, the step ID does not apply, but you can use the parameter as required by your application requirements.

When a step ID is specified in a `RequestParameters` method such as `add()`, you need to specify the step ID using the following format:

```
id1.id2.id3...
```

where the fully qualified step ID identifies the unique step, node, in the job set hierarchy (tree).

Parameters without a step ID in a job set request are treated as global parameters and they apply to each step of the job set request. The step ID argument for `RequestParameters` provides the capability to support shared parameters, where the parameter can apply to both a job set and either a job definition or a job type.

Oracle Enterprise Scheduler prepends the step ID to the name in the form of `stepId:name` to generate the unique identifier, with a colon as a separator.

Example 5–3 shows code using a `RequestParameters` object with a step ID specified with the `add()` method to set a system property value for a step in a job set.

Example 5–3 Using the CLASS_NAME System Property with Job Set Request Parameters

```
import oracle.as.scheduler.RequestParameters;
import oracle.as.scheduler.MetadataObjectId;
import oracle.as.scheduler.RuntimeService;
import oracle.as.scheduler.RuntimeServiceHandle;
import oracle.as.scheduler.SystemProperty;

RuntimeService runtime;
RuntimeServiceHandle rs_handle;
```

```
MetadataObjectId jobSetId;
int startsIn;
long requestID = 0L;

RequestParameters req_par = new RequestParameters();

req_par.add(SystemProperty.PRIORITY, "stepId-1", new Integer(8));
req_par.add(SystemProperty.PRIORITY, "stepId-2.stepId-1", new Integer(6));

Calendar start = Calendar.getInstance();
start.add(Calendar.SECOND, startsIn);

requestID =
     runtime.submitRequest(rs_handle,"My job set", jobSetId, start, req_par);
.
.
.
```

The example assumes that there is a user-created `runtimeServiceHandle` named `rs_handle`.

5.4 Using System Properties

Oracle Enterprise Scheduler represents parameter names that are known to and used by the system in the `SystemProperty` class. You can specify system properties as parameter names in the application metadata and using request parameters when a request is submitted. Oracle Enterprise Scheduler sets certain system properties when a request is submitted or at some point in the life cycle of a request.

Table 5–3 lists the available system properties, as defined in `oracle.as.scheduler.SystemProperty`. Most system properties are common to all job types while some system properties are specific to a particular job type, as indicated in the descriptions in Table 5–3.

When you use parameters, note that Oracle Enterprise Scheduler reserves the parameter names starting with "SYS_" (case-insensitive) for Oracle Enterprise Scheduler defined properties.

Table 5–3 System Properties

Name	Description
`ALLOW_MULT_PENDING`	Specifies whether multiple pending requests for the same job definition is allowed. This property has no meaning for a job set step. Type: `BOOLEAN`
`APPLICATION`	Specifies the logical name of the Java EE application used for request processing. This property is automatically set by Oracle Enterprise Scheduler during request submission. Type: `STRING`
`ASYNC_REQUEST_TIMEOUT`	Specifies the time, in minutes, that the processor waits for an asynchronous request after it has begun execution. Following this period, the request is considered to have timed out. Type: `LONG`
`BIZ_ERROR_EXIT_CODE`	Specifies the process exit code for a process job request that denotes an execution business error. If this property is not specified, the system treats a process exit code of 4 as an execution business error. This property is optional for a process job type. It is not used for other job types. Type: `STRING`

Table 5–3 (Cont.) System Properties

Name	Description
CLASS_NAME	Specifies the Java executable for a Java job request. This should be the name of a Java class that implements the `oracle.as.scheduler.Executable` interface. This property is required for a Java job type. It is not used for other job types. Type: `STRING`
CMDLINE	Specifies the command line used to invoke an external program for a Process job request. This property is required for a Process job type. It is not used for other job types. Type: `STRING`
EFFECTIVE_ APPLICATION	Specifies the logical name of the Java EE application that will be the effective application used to process the request. A job definition, job type, or a job set step can be associated with a different application by defining the `EFFECTIVE_APPLICATION` system property. This property can only be specified via metadata and cannot be specified as a submission parameter. Type: `STRING`
ENVIRONMENT_ VARIABLES	Specifies the environment variables to be set for the spawned process of a Process job request.The property value should be a comma separated list of name value pairs (name=value) representing the environment variables to be set. This property is optional for a process job type. It is not used for other job types. Type: `STRING`
EXECUTE_PAST	Specifies whether instances of a repeating request with an execution time in the past should be generated. Instances are never generated before the requested start time nor after the requested end time. To cause past instances to be generated, you must set this property to TRUE and specify the requested start time as the initial time from which instances should be generated. Note that a null requested start time defaults to the current time. Valid values for this property are: ■ `TRUE`: All instances specified by a schedule are generated regardless of the time of generation. ■ `FALSE`: Instances with a scheduled execution time in the past (that is, before the time of generation) will not be generated. If this property is not specified, the system defaults to `TRUE`. Type: `BOOLEAN`
EXTERNAL_ID	Specifies an identifier for an external portion of an asynchronous Java job. For example, an asynchronous Java job usually invokes some remote process and then returns control to Oracle Enterprise Scheduler. This property can be used to identify the remote process. This property should be set by the job implementation of asynchronous Java jobs when the identifier is known. It is never set by Oracle Enterprise Scheduler. Type: `STRING`
GROUP_NAME	Specifies the name of the Oracle Enterprise Scheduler isolation group to which this request is bound. This property is automatically set by Oracle Enterprise Scheduler during request submission. Type: `STRING`
INPUT_LIST	Specifies input to a request. The input to a serial job set is forwarded as input to the first step only. The input to a parallel job set is forwarded as input to all the parallel steps. Oracle Enterprise Scheduler imposes no format on the value of this property. Type: `STRING`
LISTENER	Specifies the event listener class associated with the request. This should be the name of a Java class that implements the `oracle.as.scheduler.EventListener` interface. Type: `STRING`

Table 5–3 (Cont.) System Properties

Name	Description
LOCALE	Specifies the locale associated with the request. Type: STRING
OUTPUT_LIST	Specifies output from a request. The output of a serial job set is the OUTPUT_LIST of the last step. The output of a parallel job set is the concatenation of the OUTPUT_LIST of all the steps, in no guaranteed order, with oracle.as.scheduler.SystemProperty.OUTPUT_LIST_DELIMITER as a separator. Type: STRING
POST_PROCESS	Specifies the post-process callout handler class. This should be the name of a Java class that implements the oracle.as.scheduler.PostProcessHandler interface. Type: STRING
PRE_PROCESS	Specifies the pre-process callout handler class. This should be the name of a Java class that implements the oracle.as.scheduler.PreProcessHandler interface. Type: STRING
PRIORITY	Specifies the request processing priority. The priority interval is [0..9] with 0 as the lowest priority and 9 as the highest. Default: If this property is not specified, the system default value used is 4. Type: INTEGER
PROCEDURE_NAME	Specifies the name of the PL/SQL stored procedure to be called for a SQL job request. The stored procedure should be specified using schema.name format. The property is required for a SQL job type. It is not used for other job types. Type: STRING
PRODUCT	Specifies the product within the application that submitted the request. Type: STRING
REDIRECTED_OUTPUT_FILE	Specifies the file where standard output and error streams are redirected for a Process job request. This represents the full path of the log file where the standard output and error streams are redirected for the spawned process when the request is executed. This property is optional for a Process job type. It is not used for other job types. Type: STRING
REPROCESS_DELAY	Specifies the callout handler processing delay time. This represents the time, in minutes, to delay request processing when a delay is requested by a callback handler. Default: If this property is not specified, the system default used is 5. Type: INTEGER
REQUEST_CATEGORY	Specifies an application-specific label for a request. The label, defined by an application or system administrator, allows administrators to group job requests according to their own specific needs. Type: STRING
REQUEST_EXPIRATION	Specifies the expiration time for a request. This represents the time, in minutes, that a request will expire after its scheduled execution time. A expiration value of zero (0) means that the request never expires. If this property is not specified, the system default value used is 0. Request expiration only applies to requests that are waiting to run. If a request waits longer than the specified expiration period, it does not run. After a request starts running the request expiration no longer applies. Type: INTEGER

Table 5–3 (Cont.) System Properties

Name	Description
REQUESTED_PROCESSOR	Specifies the request processor node on which the request should be processed. This allows processor affinity to be specified for a request. If this property is not specified, the request can run on any available request processor node. In general, this property should not be specified. If this property is specified for a request, the request processor's work assignments `oracle.as.scheduler.WorkAssignment` (specialization) must allow the execution of such requests, otherwise the request will never be executed. If the specified node is not running, the request will remain in `READY` state and will not be executed until the node is restarted. Type: `STRING`
RETRIES	Specifies the retry limit for a failed request. If request execution fails, the request will retried up to the number of times specified by this property until the request succeeds. If retry limit is zero (0), a failed request will not be retried. Default: If this property is not specified, the system default used is 0. Type: `INTEGER`
RUNAS_APPLICATIONID	Specifies the `runAs` identifier that should be used to execute the request. Normally, a request runs as the submitting user. However, if this property is set in the metadata of the job associated with the request, then the request executes under the user identified by this property. This property can only be specified via metadata and cannot be specified as a submission parameter. Type: `STRING`
SELECT_STATE	Specifies whether the result state of a job set step affects the eventual state of its parent job set. In order for the state of a job set step to be considered when determining the state of the job set, the `SELECT_STATE` must be set to `true`. If `SELECT_STATE` is not specified on a job set step, the state of the step will be included in the determination of the state of the job set. Type: `BOOLEAN`
SQL_JOB_CLASS	Specifies an Oracle Enterprise Scheduler job class to be assigned to the Oracle Enterprise Scheduler job used to execute a SQL job request. This property need not be specified unless the job used for a job request is associated with a particular Oracle Database resource consumer group or has affinity to a database service. If this property is not specified, a default Oracle Enterprise Scheduler job class is used for the job that executes the SQL request. That job class is associated with the default resource consumer group. It belongs to the default service, such that it has no service affinity and, in an Oracle RAC environment, any one of the database instances within the cluster might run the job. No additional privilege or grant is required for an Oracle Enterprise Scheduler SQL job request to use that default job class. This property is optional for a SQL job type. It is not used for other job types. Type: `STRING`
SUBMITTING_ APPLICATION	Specifies the logical name of the Java EE application for the submitted (absolute parent) request. This property is automatically set by Oracle Enterprise Scheduler during request submission. Type: `STRING`
SUCCESS_EXIT_CODE	Specifies the process exit code for a Process job request that denotes an execution success. If this property is not specified the system treats a process exit code of 0 as execution success. This property is optional for a Process job type. It is not used for other job types. Type: `STRING`

Table 5–3 (Cont.) System Properties

Name	Description
USER_FILE_DIR	Specifies a base directory in the file system where files, such as input and output files, may be stored for use by the request executable. Oracle Enterprise Scheduler supports a configuration parameter that specifies a file directory where requests may store files. At request submission, a USER_FILE_DIR property is automatically added for the request if the configuration parameter is currently set and USER_FILE_DIR property was not specified for the request. If the property is added, it will be initialized to the value of the configuration parameter. The property will not be added if the configuration parameter is not set at the time of request submission. Type: STRING
USER_NAME	Specifies the name of the user used to execute the request. Normally this is the submitting user unless the RUNAS_APPLICATIONID property was set in the job metadata. This property is automatically set by Oracle Enterprise Scheduler during request submission. Type: STRING
WARNING_EXIT_CODE	Specifies the process exit code for a Process job request that denotes an execution warning. If this property is not specified, the system treats a process exit code of 3 as execution warning. This property is optional for a Process job type. It is not used for other job types. Type: STRING
WORK_DIR_ROOT	Specifies the working directory for the spawned process of a Process job request. This property is optional for a Process job type. It is not used for other job types. Type: STRING

6

Creating and Using PL/SQL Jobs

This chapter describes how to create PL/SQL stored procedures for use with Oracle Enterprise Scheduler, and describes Oracle Database tasks that you need to perform to use PL/SQL stored procedures with Oracle Enterprise Scheduler. After you create a PL/SQL procedure and define a job definition, you can use the Oracle Enterprise Scheduler runtime service to submit a job request for a PL/SQL procedure.

This chapter includes the following sections:

For information about how to use the Runtime Service, see Chapter 13, "Using the Runtime Service".

6.1 Introduction to Using PL/SQL Stored Procedure Job Definitions

Oracle Enterprise Scheduler lets you run job requests of different types, including: Java classes, PL/SQL stored procedures, and process requests that run as a forked process. To use Oracle Enterprise Scheduler with PL/SQL stored procedures you need to do the following:

- Create or obtain the PL/SQL stored procedure that you want to use with Oracle Enterprise Scheduler.
- Load the PL/SQL stored procedure in the Oracle Database and grant the required permissions and perform other required DBA tasks.
- Use Oracle JDeveloper to create job type and job definition objects and store these objects with the Oracle Enterprise Scheduler application metadata.
- Use Oracle JDeveloper to create an application with Oracle Enterprise Scheduler APIs that runs and submits a PL/SQL stored procedure.

Finally, after you create an application that uses the Oracle Enterprise Scheduler APIs you use Oracle JDeveloper to deploy and run the application.

At runtime, after you submit a job request you can monitor and manage the job request. For more information, see Chapter 13, "Using the Runtime Service".

Oracle Enterprise Scheduler uses an asynchronous execution model for PL/SQL stored procedure job requests. This means that Oracle Enterprise Scheduler does not directly call the PL/SQL stored procedure, but instead uses Oracle Enterprise Scheduler (part of the Oracle Database). When a PL/SQL stored procedure job request is ready to

execute, Oracle Enterprise Scheduler creates an immediate, run-once Oracle Enterprise Scheduler job. This Oracle Enterprise Scheduler job is owned by the Oracle Enterprise Scheduler runtime schema user associated with the container instance that executes the application that specifies the PL/SQL stored procedure. Finally, when the Oracle Enterprise Scheduler job runs, the PL/SQL stored procedure is called using dynamic SQL. After the PL/SQL stored procedure completes, either by a successful return or by raising an exception, the Oracle Enterprise Scheduler job completes.

6.2 Creating a PL/SQL Stored Procedure for Oracle Enterprise Scheduler

When you want to use a PL/SQL stored procedure with Oracle Enterprise Scheduler, the PL/SQL procedure must have certain characteristics to work with an Oracle Enterprise Scheduler application and a DBA must assure that certain Oracle Database permissions are assigned to the PL/SQL stored procedure.

Creating a PL/SQL stored procedure involves the following steps:

- Define the PL/SQL stored procedure that has the correct signature for use with Oracle Enterprise Scheduler
- Perform the required DBA tasks to make the PL/SQL stored procedure available to Oracle Enterprise Scheduler

6.2.1 How to Define a PL/SQL Stored Procedure with the Correct Signature

Note: For more information about defining a PL/SQL stored procedure job in the context of Oracle Fusion Applications, see Section 9.6, "Implementing a PL/SQL Scheduled Job."

The PL/SQL stored procedure that you call from Oracle Enterprise Scheduler must have a specific signature and include specific procedure parameters, as follows:

```
PROCEDURE my_proc(request_handle IN VARCHAR2);
```

The `request_handle` parameter is an opaque value representing an execution context for the Oracle Enterprise Scheduler request being executed.

Example 6–1 shows a sample `HELLO_WORLD` stored procedure for use with Oracle Enterprise Scheduler.

Example 6–1 HELLO_WORLD PL/SQL Stored Procedure

```
create or replace procedure HELLO_WORLD( request_handle in varchar2 )
as
    v_request_id  number := null;
    v_prop_name  varchar2(500) := null;
    v_prop_int  integer := null;
begin
    -- Get the Oracle Enterprise Scheduler request ID being executed.
    begin
        v_request_id := ess_runtime.get_request_id(request_handle);
    exception
        when others then
            raise_application_error(-20000,
                'Failed to get request id for request handle ' ||
                request_handle || '. [' || SQLERRM || ']');
    end;
```

```
    -- Retrieve value of an existing request property.
    begin
        v_prop_name := 'mytestIntProp';
        v_prop_int := ess_runtime.get_reqprop_int(v_request_id, v_prop_name);
    exception
        when others then
            rollback;
            raise_application_error(-20001,
                'Failed to get request property ' || v_prop_name ||
                ' for Oracle Enterprise Scheduler request ID ' || v_request_id ||
                '. [' || SQLERRM || ']' );
    end;

    -- Update an existing request property with a new value.
    -- This procedure is responsible for commit/rollback of the update operation.
    begin
        v_prop_name := 'myJobdefProp';
        ess_runtime.update_reqprop_varchar2(v_request_id, v_prop_name,
                                            'myUpdatedalue');
        commit;
    exception
        when others then
            rollback;
            raise_application_error(-20002,
                'Failed to update request property ' || v_prop_name ||
                ' for Oracle Enterprise Scheduler request ID ' || v_request_id ||
                '. [' || SQLERRM || ']' );
    end;
end helloworld;
/
```

6.2.2 Handling Runtime Exceptions in an Oracle Enterprise Scheduler PL/SQL Stored Procedure

Oracle Enterprise Scheduler uses an asynchronous execution model for PL/SQL stored procedure job types. Oracle Enterprise Scheduler does not directly call the PL/SQL stored procedure, but instead uses the Oracle Enterprise Scheduler in the Oracle Database. When a PL/SQL stored procedure request is ready to execute, Oracle Enterprise Scheduler creates an immediate, run-once Oracle Enterprise Scheduler job that is owned by the Oracle Enterprise Scheduler runtime schema user associated with the container instance executing that executes the application associated with the PL/SQL stored procedure. The PL/SQL stored procedure is called using dynamic SQL when the Oracle Enterprise Scheduler job runs. After the PL/SQL stored procedure completes, either by a successful return or by raising an exception, the Oracle Enterprise Scheduler job completes.

In the PL/SQL stored procedure, you can handle exceptions and other issues by raising a `RAISE_APPLICATION_ERROR` exception. The `RAISE_APPLICATION_ERROR` requires that the error code from the PL/SQL stored procedure range from -20000 to -20999. The PL/SQL stored procedure can use `RAISE_APPLICATION_ERROR` if it needs to raise an exception. `RAISE_APPLICATION_ERROR` requires that the error code range from -20000 to -20999.

Table 6–1 indicates the Oracle Enterprise Scheduler state based on the result of the PL/SQL stored procedure.

Table 6–1 Terminal States for PL/SQL Stored Procedure Results

Final State	Description
SUCCEEDED	If the PL/SQL stored procedure returns normally, without raising an exception, the request state transitions to the SUCCEEDED state, bearing any subsequent errors completing the request.
WARNING	If the PL/SQL stored procedure returns with an exception, the request state is based on the SQL error code of the exception. The request transitions to the WARNING terminal state if the SQL error code ranges from -20900 to -20919.
ERROR	If the PL/SQL stored procedure returns with an exception, the request state is based on the SQL error code of the exception. The request transitions to the ERROR terminal state for any error code outside the range of -20900 to -20919 (error codes within this range indicate a WARNING). Return codes in the range -20920 to -20929 result in an ERROR state with a BUSINESS error type, where the request is not subject to automatic retries.

6.2.3 How to Access Job Request Information In PL/SQL Stored Procedures

Oracle Enterprise Scheduler provides a PL/SQL package, ESS_RUNTIME to perform certain operations that you may need when you are working in a PL/SQL stored procedure. You can use these procedures perform job request operations and to obtain job request information for an Oracle Enterprise Scheduler runtime schema. For example, you can use these runtime procedure to submit requests and retrieve and update request information associated with an Oracle Enterprise Scheduler job request.

The following sample code shows use of an ESS_RUNTIME procedure:

```
v_request_id := ess_runtime.get_request_id(request_handle);
```

This request obtains the request ID associated with a job request.

Certain procedures in the ESS_RUNTIME package require a request handle parameter and provide information on an executing request (these should only be called from the PL/SQL stored procedure that is executing the PL/SQL stored procedure request). You can call some procedures in the ESS_RUNTIME package from outside of the context of an executing request; these procedures may include a request id parameter.

6.2.4 What You Need to Know When You Define a PL/SQL Stored Procedure

You need to know the following when you create an use a PL/SQL stored procedure with Oracle Enterprise Scheduler:

- It is not required that the PL/SQL stored procedure exist when the Oracle Enterprise Scheduler request is submitted, but the PL/SQL stored procedure must exist and be callable by the Oracle Enterprise Scheduler runtime schema user when the request is ready to run.
- The PL/SQL stored procedure must exist on the same database as the Oracle Enterprise Scheduler Runtime schema.

6.3 Performing Oracle Database Tasks for PL/SQL Stored Procedures

After you create the PL/SQL stored procedure that you want to use with Oracle Enterprise Scheduler a DBA needs to load the PL/SQL stored procedure in the Oracle Database and grant the required permissions.

6.3.1 How to Grant PL/SQL Stored Procedure Permissions

Before the DBA grants permissions, the DBA must determine the Oracle Database and the Oracle Enterprise Scheduler run time schema that is associated with the deployed Java EE application that is going to submit the Oracle Enterprise Scheduler PL/SQL stored procedure request.

Use the following definitions when you grant PL/SQL stored procedure permissions:

`ess_schema`: specifies the Oracle Enterprise Scheduler runtime schema associated with the Java EE application.

`user_schema`: specifies the name of the application user schema.

`PROC_NAME`: specifies the name of the PL/SQL stored procedure associated with the Oracle Enterprise Scheduler job request.

To grant Oracle Database permissions:

1. In the Oracle Database grant execute on the `ESS_RUNTIME` package to the application user schema. For example:

   ```
   GRANT EXECUTE ON ess_schema.ess_runtime to user_schema;
   ```

2. In the Oracle Database, create a private synonym for the `ESS_RUNTIME` package. This is a convenience step that allows the PL/SQL stored procedure to reference the `ESS_RUNTIME` as simply `ESS_RUNTIME` rather than using the full *schema_name*.`ESS_RUNIME`. For example:

   ```
   create or replace synonym user_schema.ess_runtime for ess_schema.ess_runtime;
   ```

3. In the Oracle Database, grant execute on the PL/SQL stored procedure to the Oracle Enterprise Scheduler runtime schema user.

   ```
   GRANT EXECUTE ON user_schema.proc_name to ess_schema;
   ```

   For example, if the Oracle Enterprise Scheduler runtime schema is `TEST_ORAESS`, the application user schema is `HOWTO`, and the PL/SQL procedure is named `HELLO_WORLD`, the DBA operations needed would be:

   ```
   GRANT EXECUTE ON test_oraess.ess_runtime to howto;
   create or replace synonym howto.ess_runtime for test_oraess.ess_runtime;
   GRANT EXECUTE ON howto.hello_world to test_oraess;
   ```

6.3.2 What You Need to Know About Granting PL/SQL Stored Procedure Permissions

The first two steps shown for DBA tasks for granting permissions on the `ESS_RUNTIME` package are only required if the `ESS_RUNTIME` package is referenced by a PL/SQL procedure. These two steps are not required if the `ESS_RUNTIME` package is never used from that application user schema. The third step shown is always required since it allows Oracle Enterprise Scheduler to call the user defined PL/SQL stored procedure.

All PL/SQL stored procedures in a given application user schema that are used for Oracle Enterprise Scheduler PL/SQL stored procedure jobs should always be associated with the same (single) Oracle Enterprise Scheduler Runtime schema. While

this is not technically required, it greatly simplifies the DBA setup and does not require the PL/SQL stored procedure to explicitly specify the Oracle Enterprise Scheduler Runtime schema if the procedure references the ESS_RUNTIME.

6.4 Creating and Storing Job Definitions for PL/SQL Job Types

To use PL/SQL stored procedures with Oracle Enterprise Scheduler you need to locate the Metadata Service and create a job definition. You create a job definition by specifying a name and a job type. When you create a job definition you also need to set certain system properties. You can then store the job definition and other associated objects using the Metadata Service.

For information about how to use the Metadata Service, see Chapter 4, "Using the Metadata Service".

You can use Oracle Enterprise Scheduler system properties to specify certain attributes for the Oracle Enterprise Scheduler job that calls the PL/SQL stored procedure.

These SystemProperty properties apply specifically to SQL job types; PROCEDURE_NAME, SQL_JOB_CLASS.

The PROCEDURE_NAME system property specifies the name of the PL/SQL stored procedure to be executed. The stored procedure name should have a schema.name format. This property must be specified for either the job type or job definition.

The SQL_JOB_CLASS system property specifies an Oracle Enterprise Scheduler job class to be assigned to the Oracle Enterprise Scheduler job used to execute an SQL job request. This property does not need to be specified unless the Oracle Enterprise Scheduler job used for a request should be associated with a particular Oracle Database resource consumer group or have affinity to a database service.

Oracle Enterprise Scheduler uses an Oracle Enterprise Scheduler job to execute the PL/SQL stored procedure for a SQL job request. An Oracle Enterprise Scheduler job class can be associated with the Scheduler job when that job needs to have affinity to a database service or is to be associated with an Oracle Database resource consumer group. The Oracle Enterprise Scheduler job owner must have EXECUTE privilege on the Oracle Enterprise Scheduler job class in order to successfully create a Scheduler job using that job class.

If the SQL_JOB_CLASS system property is not specified, a default Oracle Enterprise Scheduler job class is used for the Oracle Enterprise Scheduler job. The default job class is associated with the default resource consumer group. It will belong to the default service, which means it will have no service affinity and, in an Oracle RAC environment any one of the database instances within the cluster might run the Scheduler job. No additional privilege grant is needed for an Oracle Enterprise Scheduler SQL request to use that default job class.

6.4.1 How to Create a PL/SQL Job Type

An Oracle Enterprise Scheduler JobType object specifies an execution type and defines a common set of properties for a job request. A job type can be defined and then shared among one or more job definitions. Oracle Enterprise Scheduler supports three execution types:

- JAVA_TYPE: for job definitions that are implemented in Java and run in the container.
- SQL_TYPE: for job definitions that run as PL/SQL stored procedures in a database server.

- PROCESS_TYPE: for job definitions that are binaries and scripts that run as separate processes.

When you specify the JobType you can also specify properties that define the characteristics associated with the JobType. Table 6–2 describes the SystemProperties that are appropriate for a PL/SQL stored procedure job type.

Table 6–2 Oracle Enterprise Scheduler System Properties for a PL/SQL Stored Procedure Job Type

System Property	Description
PROCEDURE_NAME	Specifies the name of the stored procedure to run as part of PL/SQL job execution. For a SQL_TYPE application, this is a required property.
SQL_JOB_CLASS	Specifies an Oracle Enterprise Scheduler job class to be assigned to the Oracle Enterprise Scheduler job used to execute an SQL job request. This is an optional property for a SQL_TYPE job type.

When you create and store a PL/SQL job type, you do the following:

- Use the JobType constructor and supply a String name and a JobType.ExecutionType.SQL_TYPE argument.
- Set the appropriate properties for the new JobType.
- Obtain the metadata pointer, as shown in Section 4.2, "Accessing the Metadata Service". Use the Metadata Service addJobType() method to store the JobType in metadata.
- Use a MedatdataObjectId that uniquely identifies metadata objects in the metadata repository, and, using a unique identifier the MetadataObjectID contains the fully qualified name for a metadata object.

See Section 6.4.3, "Using a PL/SQL Stored Procedure with an Oracle Enterprise Scheduler Application" for sample code.

6.4.2 How to Create and Store a Job Definition for PL/SQL Job Type

To use PL/SQL with Oracle Enterprise Scheduler, you need to create and store a job definition. A job definition is the basic unit of work that defines a job request in Oracle Enterprise Scheduler. Each job definition belongs to one and only one job type.

> **Note:** Once you create a job definition with a job type, you cannot change the type or the job definition name. To change the type or the job definition name, you need to create a new job definition.

Section 6.4.3, "Using a PL/SQL Stored Procedure with an Oracle Enterprise Scheduler Application" shows how to create a job definition using the job definition constructor and the job type.

6.4.3 Using a PL/SQL Stored Procedure with an Oracle Enterprise Scheduler Application

Example 6–2 shows sample code in which job type and job definition application metadata are created for a SQL job type.

Example 6–2 Oracle Enterprise Scheduler Program Using PL/SQL Stored Procedure

```
import oracle.as.scheduler.JobType;
import oracle.as.scheduler.JobDefinition;
import oracle.as.scheduler.MetadataService;
import oracle.as.scheduler.MetadataServiceHandle;
import oracle.as.scheduler.MetadataObjectId;
import oracle.as.scheduler.ParameterInfo;
import oracle.as.scheduler.ParameterInfo.DataType;
import oracle.as.scheduler.ParameterList;

void createDefinition( )
{
   MetadataService metadata = ...
   MetadataServiceHandle mshandle = null;

   try
   {
      ParameterInfo pinfo;
      ParameterList plist;

      mshandle = metadata.open();

      // Define and add a PL/SQL job type for the application metadata.
      String jobTypeName = "PLSQLJobDefType";
      JobType jobType = null;
      MetadataObjectId jobTypeId = null;

      jobType = new JobType(jobTypeName, JobType.ExecutionType.SQL_TYPE);

      plist = new ParameterList();
      pinfo = SystemProperty.getSysPropInfo(SystemProperty.PROCEDURE_NAME);
      plist.add(info.getName(), pinfo.getDataType(), "HOWTO.HELLO_WORLD", false);
      pinfo = SystemProperty.getSysPropInfo(SystemProperty.PRODUCT);
      plist.add(pinfo.getName(), pinfo.getDataType(), "HOW_TO_PROD", false);
      jobType.setParameters(plist);

      jobTypeId = metadata.addJobType(mshandle, jobType, "HOW_TO_PROD");

      // Define and add a job definition for the application metadata.
      String jobDefName = "PLSQLJobDef";
      JobDefinition jobDef = null;
      MetadataObjectId jobDefId = null;

      jobDef = new JobDefinition(jobDefName, jobTypeId);
      jobDef.setDescription("Demo PLSQL Job Definition " + jobDefName);

      plist = new ParameterList();
      plist.add("myJobdefProp", DataType.STRING, "myJobdefVal", false);
      jobDef.setParameters(plist);

      jobDefId = metadata.addJobDefinition(mshandle, jobDef, "HOW_TO_PROD");
   }
   catch (Exception e)
   {
      [...]
   }
   finally
   {
      // always close metadata service handle in finally block
      if (null != mshandle)
```

```
        {
           metadata.close(mshandle);
           mshandle = null;
        }
     }
  }
```

7

Creating and Using Process Jobs

This chapter describes how to create process jobs with an Oracle Enterprise Scheduler job definition. After you create a script or binary command that you want to run as a process job in a forked process, you define a job definition and then use the Oracle Enterprise Scheduler runtime service to submit a job request that runs as a spawned job.

This chapter includes the following sections:

- Section 7.1, "Introduction to Creating Process Job Definitions"
- Section 7.2, "Creating and Storing Job Definitions for Process Job Types"

For information about how to use the Runtime Service, see Chapter 13, "Using the Runtime Service".

7.1 Introduction to Creating Process Job Definitions

Oracle Enterprise Scheduler lets you run job requests of different types, including: Java classes, PL/SQL stored procedures, or process jobs that run as spawned jobs. To use Oracle Enterprise Scheduler to run process type jobs you need to specify certain metadata to define the characteristics of the process type job that you want to run. You may also want to specify properties of the job request, such as the schedule for when it runs.

Specifying a process type job request with Oracle Enterprise Scheduler is a three step process:

1. You create or obtain the script or binary command that you want to run with Oracle Enterprise Scheduler. We do not cover this step because we assume that you have previously created the script or command for the spawned process.
2. Using the Oracle Enterprise Scheduler APIs in your application, you create job type and job definition objects and store these objects to the metadata repository.
3. Using the Oracle Enterprise Scheduler APIs you submit a job request. For information about how to submit a request, see Chapter 13, "Using the Runtime Service".

After you create an application that uses the Oracle Enterprise Scheduler APIs, you need to package and deploy the application.

At runtime, after you submit a job request you can monitor and manage the job request. For more information on monitoring and managing job requests, see Chapter 13, "Using the Runtime Service".

7.2 Creating and Storing Job Definitions for Process Job Types

To use process type jobs with Oracle Enterprise Scheduler, you need to locate the Metadata Service and create a job definition. You create a job definition by specifying a name and a job type. When you create a job definition you also need to set certain system properties. You can store the job definition in the metadata repository using the Metadata Service.

For information about how to use the Metadata Service, see Chapter 4, "Using the Metadata Service".

7.2.1 How to Create and Store a Process Job Type

An Oracle Enterprise Scheduler `JobType` object specifies an execution type and defines a common set of properties for a job request. A job type can be defined and then shared among one or more job definitions. Oracle Enterprise Scheduler supports three execution types:

- `JAVA_TYPE`: for job definitions that are implemented in Java and run in the container.
- `SQL_TYPE`: for job definitions that run as PL/SQL stored procedures in a database server.
- `PROCESS_TYPE`: for job definitions that are binaries and scripts that run as separate processes under the control of the host operating system.

When you specify the `JobType` you can also specify `SystemProperties` that define the characteristics associated with the `JobType`. Table 7–1 describes the properties that specify how the request should be processed if the request results in spawning a process for a process job type.

Table 7–1 System Properties for Process Type Jobs

System Property	Description
`BIZ_ERROR_EXIT_CODE`	Specifies the process exit code for a process job request that denotes an execution business error. If this property is not specified, the system treats a process exit code of 4 as an execution business error.
`CMDLINE`	Command line required for invoking an external program.
`ENVIRONMENT_VARIABLES`	A comma separated list of name value pairs (name=value) representing the environment variables to be set for Spawned processes.
`REDIRECTED_OUTPUT_FILE`	Specifies the file where standard output and error streams are redirected for a process job request.
`REQUESTED_PROCESSOR`	The Oracle WebLogic Server node on which a spawned job is executed.
`SUCCESS_EXIT_CODE`	The process exit code for a process job request that denotes a successful execution. If this property is not specified, the system treats a process exit code of 0 as a successful completion.
`WARNING_EXIT_CODE`	The process exit code for a Spawned job that denotes a successful execution. If this property is not specified, the system treats a process exit code of 3 as a warning exit.
`WORK_DIR_ROOT`	The working directory for a Spawned process.

For more information about system properties, see Chapter 5, "Using Parameters and System Properties."

Example 7–1 shows a sample job type definition with a `PROCESS_TYPE`.

Example 7–1 Creating an Oracle Scheduler JobType and Setting JobType Properties

```
import oracle.as.scheduler.ConcurrentUpdateException;
import oracle.as.scheduler.JobType;
import oracle.as.scheduler.JobDefinition;
import oracle.as.scheduler.MetadataService;
import oracle.as.scheduler.MetadataServiceHandle;
import oracle.as.scheduler.MetadataObjectId;
import oracle.as.scheduler.MetadataServiceException;
import oracle.as.scheduler.ParameterInfo;
import oracle.as.scheduler.ParameterInfo.DataType;
import oracle.as.scheduler.ParameterList;
import oracle.as.scheduler.SystemProperty;
import oracle.as.scheduler.ValidationException;

    void createDefinition( )
        throws MetadataServiceException,ConcurrentUpdateException,
               ValidationException
    {
        MetadataService metadata = ...
        MetadataServiceHandle mshandle = null;

        try
        {
            ParameterInfo pinfo;
            ParameterList plist;

            mshandle = metadata.open();

            // Define and add a PL/SQL job type for the application metadata.
            String jobTypeName = "ProcessJobDefType";
            JobType jobType = null;
            MetadataObjectId jobTypeId = null;

            jobType = new JobType(jobTypeName, JobType.ExecutionType.
                                  PROCESS_TYPE);

            plist = new ParameterList();
            pinfo = SystemProperty.getSysPropInfo(SystemProperty.CMDLINE);
            plist.add(pinfo.getName(), pinfo.getDataType(), "/bin/myprogram
                      arg1 arg2", false);
            pinfo = SystemProperty.getSysPropInfo(SystemProperty.
                                                  ENVIRONMENT_VARIABLES);
            plist.add(pinfo.getName(), pinfo.getDataType(),
                                       "LD_LIBRARY_PATH=/usr/lib", false);
            pinfo = SystemProperty.getSysPropInfo(SystemProperty.PRODUCT);
            plist.add(pinfo.getName(), pinfo.getDataType(), "HOW_TO_PROD", false);
            jobType.setParameters(plist);

            jobTypeId = metadata.addJobType(mshandle, jobType, "HOW_TO_PROD");

            // Define and add a job definition for the application metadata.
            String jobDefName = "ProcessJobDef";
            JobDefinition jobDef = null;
            MetadataObjectId jobDefId = null;

            jobDef = new JobDefinition(jobDefName, jobTypeId);
            jobDef.setDescription("Demo Process Type Job Definition " +
                                  jobDefName);

            plist = new ParameterList();
```

```
            plist.add("myJobdefProp", DataType.STRING, "myJobdefVal", false);

            pinfo = SystemProperty.getSysPropInfo(SystemProperty.
                                                  REDIRECTED_OUTPUT_FILE);
            plist.add(pinfo.getName(), pinfo.getDataType(), "/tmp/" + jobDefName
                      + ".out", false);

            jobDef.setParameters(plist);

            jobDefId = metadata.addJobDefinition(mshandle, jobDef, "HOW_TO_PROD");
        }
        catch (Exception e)
        {
          [...]
        }
        finally
        {
            // Close metadata service handle in finally block.
            if (null != mshandle)
            {
                metadata.close(mshandle);
                mshandle = null;
            }
        }
    }
```

As shown in Example 7–1, when you create and store a process job type, you do the following:

- Use the `JobType` constructor and supply a `String` name and a `JobType.ExecutionType.PROCESS_TYPE` argument.
- Obtain the metadata pointer, as shown in Section 4.2, "Accessing the Metadata Service". Use the Metadata Service `addJobType()` method to store the `JobType` in metadata.
- The `MedatdataObjectId`, returned by addJobType(), uniquely identifies metadata objects in the metadata repository using a unique identifier.

7.2.2 How to Create and Store a Process Type Job Definition

To use process type jobs, you need to create and store a job definition.

> **Note:** Once you create a job definition with a job type, you cannot change the type or the job definition name. To change the job type or the job definition name, you need to create a new job definition.

Example 7–1 shows how to create a job definition using the job definition constructor and the job type. Table 7–1 describes some of the system properties that are associated with the job definition.

As shown in Example 7–1, when you create and store a job definition you do the following:

- Use the `JobDefinition` constructor and supply a `String` name and a `MetadataObjectID` that points to a job type stored in the metadata.
- Set the appropriate properties for the new job definition.

- Obtain the metadata pointer, as shown in Section 4.2, "Accessing the Metadata Service". Then, use the Metadata Service `addJobDefinition()` method to store the job definition in the metadata repository and to return a `MetadataObjectID`.

7.3 Using a Perl Agent Handler for Process Jobs

Oracle Enterprise Scheduler requires a Perl agent to manage individual process jobs. The Perl agent is responsible for validating, spawning, monitoring and controlling process job execution, as well as returning the exit status of process jobs to Oracle Enterprise Scheduler. The Perl agent also monitors Oracle Enterprise Scheduler availability and handles job cancellation requests. In the event of abnormal job termination (or job cancellation requests), the Perl agent terminates the spawned process (along with its children) and exits. It detects the operating system type and uses appropriate system calls to invoke, manage and terminate process jobs.

The Oracle Enterprise Scheduler Perl agent can generate its log under the `/tmp` folder. This must be enabled by setting the Oracle Enterprise Scheduler log level to `FINE`, `FINER` or `FINEST` and ensuring read and write access to the `/tmp` folder. One log file is generated for each process job invocation. The log file lists the process job invocation log, including a list of environment variables, the command line and redirected output file specified for the process job, process ID and exit code for the process job or errors detected while spawning the process.

Oracle Enterprise Scheduler Perl agent requires Oracle Perl version 5.10 or later.

8 Defining and Using Schedules

This chapter describes how to define schedules that can be associated with a job definition. Using a schedule you can specify when a job request runs. You can also define and use schedules to specify administrative actions such as workshifts that specify time-based controls for processing with Oracle Enterprise Scheduler.

This chapter includes the following sections:

8.1 Introduction to Schedules

Using Oracle Enterprise Scheduler you can create a schedule to determine when a job request runs or use a schedule for other purposes, such as determining when a work assignment becomes active. A schedule can contain a list of explicit dates, such as July 14, 2008. A schedule can also include expressions that represent a set of recurring dates (or times and dates).

Using Oracle Enterprise Scheduler you create a schedule with one or more of the following:

- Explicit Date: Defines a date for use in a schedule or exclusion.
- Recurrence: Contains an expression that represents a pattern for a recurring date and time. For example, you can specify a recurrence representing a regular period such as Mondays at 10:00AM.
- Exclusion: Contains a list of dates to exclude or dates and times to exclude from a schedule. For example, you can create an exclusion that contains a list of holidays to exclude from a schedule.

8.2 Defining a Recurrence

A recurrence is an expression that represents a recurring date and time. You specify a recurrence using an Oracle Enterprise Scheduler `Recurrence` object. You use a

`Recurrence` object when you create a schedule or with an exclusion to specify a list of dates.

The `Recurrence` constructor allows you to create a recurrence as follows:

- Using the fields defined in the `RecurrenceFields` class, such as `DAY_OF_MONTH`.
- Using a recurrence expression compliant with the iCalendar (RFC 2445) specification. For information about using iCalendar RFC 2245 expressions see,

  http://www.ietf.org/rfc/rfc2445.txt

> **Note:** When you create a recurrence you can only use one of these two mechanisms for each recurrence instance.

A recurrence can also include the following (these are not required):

- Start date: The starting time and date for the recurrence pattern.
- End date: The ending time and date for the recurrence pattern.
- Count: The count for the recurrence pattern. The count indicates the maximum number of occurrences the object generates. For example, if you specify a recurrence representing a regular period such as Mondays at 10:00AM, and a count of 4, then the recurrence includes only four Mondays.

The start date, end date, and count attributes are valid for either a `RecurrenceFields` helper based instance or an iCalendar based instance of a recurrence.

You can validate a recurrence using the recurrence `validate()` method that checks if an instance of a `Recurrence` object represents a well defined and complete recurrence pattern. A `Recurrence` instance is considered complete if it has the minimum required fields that can generate occurrences of dates or dates and times.

8.2.1 How to Define a Recurrence with a Recurrence Fields Helper

You can create a recurrence using a recurrence fields helper. The `RecurrenceFields` helper class provides a user-friendly way to specify a recurrence pattern. Table 8–1 shows the recurrence fields helper classes available to specify a recurrence pattern.

Table 8–1 Recurrence Field Helper Patterns

Recurrence Field	Description
`DAY_OF_MONTH`	Defines the day of a month
`DAY_OF_WEEK`	Enumeration of the day of a week
`FREQUENCY`	Defines the repeat frequency of a Recurrence. Choices are: ▪ `DAILY`: Indicates every day repetition ▪ `HOURLY`: Indicates every hour repetition ▪ `MINUTELY`: Indicates every minute repetition ▪ `MONTHLY`: Indicates every month repetition ▪ `SECONDLY`: Indicates every second repetition ▪ `WEEKLY`: Indicates every week repetition ▪ `YEARLY`: Indicates every year repetition
`MONTH_OF_YEAR`	Defines the months of the year
`TIME_OF_DAY`	Defines the time of the day

Table 8–1 (Cont.) Recurrence Field Helper Patterns

Recurrence Field	Description
WEEK_OF_MONTH	Enumerations for the week of a month
YEAR	Encapsulate the value of a year

Example 8–1 shows a sample recurrence created using the RecurrenceFields helper class with a weekly frequency (every Monday at 10:00 a.m.) using no start or end date.

Example 8–1 Defining a Recurrence with Weekly Frequency

```
Recurrence recur1 =
      new Recurrence(RecurrenceFields.FREQUENCY.WEEKLY, 1, null, null);
recur1.addDayOfWeek(RecurrenceFields.DAY_OF_WEEK.MONDAY);
recur1.setRecurTime(RecurrenceFields.TIME_OF_DAY.valueOf(10, 0, 0));
recur1.validate();
```

In Example 8–1, note the following:

- The schedule becomes active as specified with the start time supplied at runtime by Oracle Enterprise Scheduler when a job request that uses the schedule is submitted.
- The interval parameter 1 specifies that this recurrence generates occurrences every week. You calculate this value by multiplying the frequency with the interval.

Example 8–2 shows a sample recurrence for every 4 hours with no start or end date. The recurrence was created using the RecurrenceFields helper class with an hourly frequency, an interval multiplier of 4, a null start date, and a null end date.

Example 8–2 Defining a Recurrence with Four Hourly Frequency

```
Recurrence recur2 =
    new Recurrence(RecurrenceFields.FREQUENCY.HOURLY, 4, null, null);
    recur2.validate();
```

In Example 8–2, note the following:

- The schedule becomes active as specified with the start time supplied at runtime by Oracle Enterprise Scheduler when a job request that uses the schedule is submitted.
- The interval parameter 4 specifies that this recurrence generates occurrences every 4 hours. You calculate this value by multiplying the frequency with the interval.

Example 8–3 shows a sample recurrence created using the RecurrenceFields helper class and a monthly frequency.

Example 8–3 Defining a Recurrence with Monthly Frequency

```
Recurrence recur3 =
   new Recurrence(RecurrenceFields.FREQUENCY.MONTHLY, 1, null, null);
recur3.addWeekOfMonth(RecurrenceFields.WEEK_OF_MONTH.SECOND);
recur3.addDayOfWeek(RecurrenceFields.DAY_OF_WEEK.TUESDAY);
recur3.setRecurTime(RecurrenceFields.TIME_OF_DAY.valueOf(11, 00, 00));
recur3.validate();
```

Example 8–3 specifies a recurrence with the following characteristics:

- Includes an interval parameter with the value 1 specifies that this recurrence generates occurrences every month.
- Includes a specification for the week of month, indicating the second week.
- Includes a specification for the day of week, Tuesday.
- Includes the specification for the time of day, with the value 11:00.

Example 8–4 shows a sample recurrence created using the `RecurrenceFields` helper class and a monthly frequency specified with a start date and time.

Example 8–4 Defining a Recurrence with Start Date and Time Specified

```
Calendar cal = Calendar.getInstance();
cal.set(Calendar.YEAR, 2007);
cal.set(Calendar.MONTH, Calendar.JULY);
cal.set(Calendar.DAY_OF_MONTH, 1);
cal.set(Calendar.HOUR, 9);
cal.set(Calendar.MINUTE, 0);
cal.set(Calendar.SECOND, 0);
Recurrence recur4 = new Recurrence(RecurrenceFields.FREQUENCY.WEEKLY,
                                   1,
                                   cal,
                                   null);
recur4.validate();
```

Example 8–4 defines a recurrence with the following characteristics:

- The end date is specified as null meaning no end date.
- Using this recurrence, the start date is specified with the `Calendar` instance `cal`, and its value is set with the `set()` method calls.

8.2.2 How to Define a Recurrence with an iCalendar RFC 2445 Specification

You can specify a recurrence pattern using the `Recurrence` constructor with a `String` containing an iCalendar (RFC 2445) specification.

For information about using iCalendar RFC 2245 expressions see the following link:

`http://www.ietf.org/rfc/rfc2445.txt`

Example 8–5 shows a sample recurrence created using an iCalendar expression.

Example 8–5 Defining a Recurrence with an iCalendar String Expression

```
Recurrence recur5 = new Recurrence("FREQ=YEARLY;INTERVAL=1;BYMONTH=5;BYDAY=2MO;");
recur5.validate();
```

Note: The following are **not** supported through iCalendar expressions:

`COUNT`, `UNTIL`, `BYSETPOS`, `WKST`

You can still directly specify a count on the `Recurrence` object using the `setCount` method.

8.2.3 What You Need to Know When You Use a Recurrence Fields Helper

When you define a recurrence with a `RecurrenceFields` helper, note the following:

- Providing a frequency with one of the `RecurrenceFields.FREQUENCY` constants is always mandatory when you define a recurrence pattern using the `RecurrenceFields` helper classes (for more information on frequency, see Table 8–1).
- The frequency interval supplied with the recurrence constructor is an integer that acts as a multiplier for the supplied frequency. For example if the frequency is `RecurrenceFields.FREQUENCY.HOURLY` and the interval is 8, then the combination represents every 8 hours.
- Providing either a start or end date is optional. But if a start or end date is specified, it is guaranteed that the object will not generate any occurrences before the start date or after the end date (and if specified, any associated start time or end time).
- In general if both start date and recurrence fields are used, then the recurrence fields always take precedence. This qualification means the following:
  - If a start date is specified with just the frequency fields from the `RecurrenceFields` then the start date defines the occurrences with the frequency field, starting with the first occurrence on the start date itself. For example if a start date is specified as 01-MAY-2007:09:00:00 with a `RecurrenceFields.FREQUENCY` of `WEEKLY` without using other recurrence fields, the occurrences happen once every week starting on 01-MAY-2007:09:00:00 (and including 08-MAY-2007:09:00:00, 15-MAY-2007:09:00:00, and so on).

    Thus, providing a start date along with a specification of frequency fields provides a quick way of defining a recurrence pattern.
  - If the start date or end date is specified together with additional recurrence fields, the recurrence fields take precedence, and the start date or end date only act as absolute boundary points. For example, with a start date of 01-MAY-2007:09:00:00 and a frequency of `WEEKLY` if the additional recurrence field `DAY_OF_WEEK` is used with a value of `WEDNESDAY` the occurrence happens on every Wednesday starting with the first Wednesday that comes after 01-MAY-2007. Because 01-MAY-2007 is a Tuesday, the first occurrence happens on 02-MAY-2007:09:00:00 and not on 01-MAY-2007:09:00:00.

    In this case, with the start date of 01-MAY-2007:09:00:00, if the `TIME_OF_DAY` is also specified as 11:00:00, all the occurrences happen at 11:00:00 overriding the 09:00:00 time from the starting date specification.
- When just a frequency is supplied and a recurrence does not include either a start date, start time, or a `TIME_OF_DAY` field, the occurrences happen based on a timestamp that Oracle Enterprise Scheduler supplies at runtime (typically this timestamp is provided during request submission).

  For example, when a recurrence indicates a 2 hour recurrence then the time of the job request submission determines the start time for the occurrences. Thus, in such cases the occurrences for a job request are each 2 hours apart, but when multiple job requests are submitted, the start times will be different and are set at the request submission time for the job requests.
- When the start date is not used, recurrence fields can be included such that a recurrence pattern is completely defined. For example, specifying a `MONTH_OF_YEAR` alone does not define a recurrence pattern when a start date is not also present. Without a start date the number of minimum recurrence fields required to define a pattern depends upon the value of the frequency used. For example with frequency of `WEEKLY`, only `DAY_OF_WEEK` and `TIME_OF_DAY` are sufficient to define which day the weekly occurrences should happen. With a frequency of `YEARLY`,

MONTH_OF_YEAR, DAY_OF_MONTH (or the WEEK_OF_MONTH and DAY_OF_WEEK) and the TIME_OF_DAY are sufficient to define the recurrence pattern.

- You can supply multiple values for recurrence fields, except for the frequency field. However, at runtime Oracle Enterprise Scheduler skips invalid combinations silently. For example with MONTH_OF_YEAR specified as January and ending in June, and with DAY_OF_MONTH as 30, the recurrence skips an invalid day, that is day 30 for February.

8.2.4 What You Need to Know When You Use an iCalendar Expression

When you define a recurrence with an iCalendar expression, note the following:

- When the recurrence does not include either a start date or time and the iCalendar expression does not specify a time of day, the occurrences happen based on a timestamp that Oracle Enterprise Scheduler supplies at runtime (typically this timestamp is provided during request submission).

  For example a recurrence can indicate a 2 hour recurrence, and the start date and time of the job request submission determines the exact start time for the occurrences. Note that in such cases, when the start time is not specified, occurrences for different job requests can happen at different times, based on the submission time, but the individual occurrences will be 2 hours apart.

- Providing either a start date with setStartDate() or an end date with setEndDate() is optional. But if a start or end date is specified, it is guaranteed that the object will not generate any occurrences before the start date or after the end date (and if specified, any associated start time or end time).

8.3 Defining an Explicit Date

An explicit date defines a date and time for use in a schedule or an exclusion. You construct an ExplicitDate using appropriate fields from the RecurrenceFields class.

8.3.1 How to Define an Explicit Date

Example 8–6 shows an explicit date definition.

Example 8–6 Defining an Explicit Date

```
ExplicitDate date = new ExplicitDate(RecurrenceFields.YEAR.valueOf(2007),
                              RecurrenceFields.MONTH_OF_YEAR.AUGUST,
                              RecurrenceFields.DAY_OF_MONTH.valueOf(17));
```

In Example 8–6 a RecurrenceFields helper defines a date in the constructor and the value does not include a time of day. You can optionally use setTime to set the time associated with an explicit date.

8.3.2 What You Need to Know About Explicit Dates

The ExplicitDate class provides the ability to define a partial date, when compared with java.util.Calendar where the time part is not specified. Also all other java.util.Calendar fields such as TimeZone are not defined with an ExplicitDate. When the time part is not specified in an ExplicitDate, Oracle Enterprise Scheduler computes the time appropriately. For example, consider a schedule that indicates every Monday after June 1, 2007, and adds an explicit date for the 17th of August 2007 (a Friday). In this example, the 17th of August 2007 is a partial date since it does not include a time.

8.4 Defining and Storing Exclusions

Using an Oracle Enterprise Scheduler exclusion you can represent dates that need to be excluded from a schedule. For example, you can use an exclusion to create a list of holidays to skip in a schedule.

8.4.1 How to Define an Exclusion

You represent an individual exclusion with an `Exclusion` object. You can define the dates to exclude in an exclusion using either an `ExplicitDate` or with a `Recurrence` object.

Example 8–7 shows how to create an `Exclusion` instance using a recurrence.

Example 8–7 Defining Explicit Dates and an Exclusion

```
Recurrence recur = new Recurrence(RecurrenceFields.FREQUENCY.YEARLY, 1);
recur.addMonth(RecurrenceFields.MONTH_OF_YEAR.JULY);
recur.addDayOfMonth(RecurrenceFields.DAY_OF_MONTH.valueOf(4));
Exclusion e = new Exclusion("Independence Day", recur);
```

Example 8–7 defines an individual exclusion. For information about creating a list of Exclusions, see Section 8.4.2, "How to Create an Exclusions Definition".

8.4.2 How to Create an Exclusions Definition

To create a list of exclusions and persist the exclusion dates you do the following:

1. Create a list of exclusions.
2. Define an `ExclusionsDefinition` object using the list of exclusions.
3. Use the Metadata Service `addExclusionDefinition()` method to persist the `ExclusionsDefinition`.

Finally, when you want to associate an `ExclusionsDefinition` with a schedule, you use the schedule `addExclusion()` method.

Example 8–8 shows how to create an `ExclusionDefinition` and store the definition to the metadata repository.

Example 8–8 Creating and Storing a List of Exclusions in an ExlusionDefinition

```
Collection<Exclusion> exclusions = new ArrayList<Exclusion>();
Exclusion e = new Exclusion("Independence Day", recur);
exclusions.add(e);
ExclusionsDefinition exDef1 =
    new ExclusionsDefinition("OrclHolidays1", "Annual Holidays", exclusions);
MetadataObjectId exId1 =
    m_service.addExclusionDefinition(handle,
                                     exDef1,
                                     "METADATA_UNITTEST_PROD");
```

Note in Example 8–8 that the `ExclusionsDefinition` constructor needs three arguments.

8.5 Defining and Storing Schedules

Using Oracle Enterprise Scheduler you can create a schedule to determine when a job request runs or use the schedule for other purposes (such as determining when a work assignment becomes active). A schedule contains a list of explicit dates, such as June 13, 2007 or a set of expressions that represent a recurring date or date and time. A schedule can also specify specific exclusion and inclusion dates.

You create a schedule using the following:

- Explicit Dates: Define a date for use in a schedule or exclusion. For more information, see Section 8.3, "Defining an Explicit Date"
- Recurrences: Contain an expression that represents a pattern for a recurring date and time. For example, you can specify a recurrence representing a regular period such as Mondays at 10:00AM. For more information, see Section 8.2, "Defining a Recurrence"
- Exclusions: Contain a list of dates to exclude or dates and times to exclude from a schedule. For example, you can create an exclusion that contains a list of holidays to exclude from a schedule. For more information, see Section 8.4, "Defining and Storing Exclusions"

8.5.1 How to Define and Store a Schedule

To define a schedule:

1. Create a schedule by defining an Oracle Enterprise Scheduler `Schedule` object and using the schedule constructor to create a new schedule.
2. Obtain a metadata service reference, `m_metadataService`, and open a metadata session in a `try` block with `MetadataServiceHandle`.

   ```
   MetadataObjectId scheduleId = m_service.addScheduleDefinition(handle, s1, "HOW_
   TO_PROD");
   ```

3. Define the date, recurrences and exclusions.
4. Store the schedule using `addScheduleDefinition`.
5. Close the session with a `finally` block.

8.5.2 What Happens When You Define and Store a Schedule

Example 8–9 shows a sample schedule definition using a recurrence with the `RecurrenceFields` helper class for a weekly schedule, specified to run on Mondays at 10:00AM.

The schedule uses the `addInclusionDate()` method to add an explicit date to the occurrences in the schedule, and the `addExclusionDate()` method to explicitly exclude the date of May 15 from schedule occurrences.

Example 8–9 Creating a Schedule Recurrence with RecurrenceFields Helpers

```
Recurrence recur = new Recurrence(RecurrenceFields.FREQUENCY.WEEKLY);
 recur.addDayOfWeek(RecurrenceFields.DAY_OF_WEEK.MONDAY);
 recur.setRecurTime(RecurrenceFields.TIME_OF_DAY.valueOf(10, 0, 0));

 ExplicitDate july10 = new ExplicitDate(RecurrenceFields.YEAR.valueOf(2008),
                                        RecurrenceFields.MONTH_OF_YEAR.JULY
                                        RecurrenceFields.DAY_OF_MONTH.valueOf(10));
```

```
ExplicitDate may15 = new ExplicitDate(RecurrenceFields.YEAR.valueOf(2008),
                                      RecurrenceFields.MONTH_OF_YEAR.MAY,
                                      RecurrenceFields.DAY_OF_MONTH.valueOf(15));

Schedule schedule = new Schedule("everyMonday", "Weekly Schedule", recur);
schedule.addInclusionDate(july10);
schedule.addExclusionDate(may15);
```

Example 8–10 shows sample code used to store a schedule. The method `addScheduleDefinition()` is used to store the schedule within a `try` block, followed by a `finally` block that includes error handling.

Example 8–10 Storing a Schedule

```
MetadataServiceHandle handle = null;
boolean abort = true;
try
   {
      handle = m_service.open();
      m_service.addScheduleDefinition(handle, schedule, "HOW_TO_PROD");
      abort = false;
   }
finally
   {
      if (handle != null)
      {
         m_service.close(handle, abort);
      }
   }
```

8.5.3 What You Need to Know About Handling Time Zones with Schedules

You can use a `java.util.TimeZone` object to set the time zone for a schedule. Use the Schedule `setTimeZone()` method to set or clear the `TimeZone` for a Schedule. The Schedule method `getTimeZone()` returns a `java.util.TimeZone` value if the Schedule object has as `TimeZone` set.

8.6 Identifying Job Requests That Use a Particular Schedule

You can use Fusion Applications Control to search for job requests that use a particular schedule.

For more information about searching for job requests that use a certain schedule, see the section "Searching for Oracle Enterprise Scheduler Job Requests" in the chapter "Managing Oracle Enterprise Scheduler Service and Jobs" in *Oracle Fusion Applications Administrator's Guide*.

8.7 Updating and Deleting Schedules

You can use Fusion Applications Control to edit and delete schedules.

For information about editing and deleting schedules, see the section "Managing Schedules" in the chapter "Managing Oracle Enterprise Scheduler Service and Jobs" in *Oracle Fusion Applications Administrator's Guide*.

9

Working with Extensions to Oracle Enterprise Scheduler

This chapter explains how to use extensions to Oracle Enterprise Scheduler to manage job request submissions.

9.1 Introduction to Oracle Enterprise Scheduler Extensions

Oracle Enterprise Scheduler provides the ability to run different job types, including: Java, PL/SQL and spawned jobs. Jobs can run on demand, or scheduled to run in the future.

Oracle Enterprise Scheduler provides scheduling services for the following purposes:

- Distributing job request processing across a grid of application servers
- Running Java, PL/SQL and process or spawned jobs
- Processing multiple jobs concurrently
- Running the same job in different languages

Using Oracle JDeveloper, application developers can easily create and implement jobs. While implemented in JDeveloper, Oracle Enterprise Scheduler runs the jobs. A number of APIs are provided to interface between jobs executed within applications developed in JDeveloper and Oracle Enterprise Scheduler Service.

The Oracle JDeveloper extensions to Oracle Enterprise Scheduler enable the following:

- Running scheduled Oracle BI Publisher, spawned, Java, PL/SQL, Perl, SQL*Plus, SQLoader and C jobs
- Running the same job in multiple locales, time zones, currencies, and so on.
- Creating log and output files for jobs, as well as acting upon those files, such as enabling notifications.
- Creating Oracle ADF task flows to schedule jobs and job sets, as well as monitor job requests.

Before you begin:

Install Oracle Enterprise Scheduler Service to the WLS server. For more information, see the chapter "Setting Up Your Development Environment" in *Oracle Fusion Applications Developer's Guide*.

9.2 Standards and Guidelines

The following standards and guidelines apply to working with extensions to Oracle Enterprise Scheduler Service:

- Always use the pre-configured job types provided when defining metadata for job definitions.

9.3 Creating and Implementing a Scheduled Job in JDeveloper

Submitting job requests from an Oracle Fusion application requires developing the following components:

- A job definition, created in JDeveloper
- The job itself, implemented in Java, PL/SQL, SQL*Loader, SQL*Plus, Perl, C or host scripts
- A user interface enabling end-users to submit job requests and/or additional properties for the job

A wizard enables easily defining a new job within the context of an Oracle Fusion application. The job can be any one of the following types: Java, PL/SQL, SQL*Loader, SQL*Plus, Perl, C or host scripts.

9.3.1 How to Create and Implement a Scheduled Job in JDeveloper

Creating and implementing a scheduled job in JDeveloper involves creating a package or class from which to call the job, as well as defining a job definition. The job must then be deployed and tested, and a job request submission interface defined.

To create and implement a scheduled job in JDeveloper:

1. Create a package, class, or job, and include the minimum required methods or functions.
   - Define the job request
   - Define any sub-requests, if required.
2. If a job requires parameters to be filled in by end users using an Oracle ADF user interface, define a standard ADF Business Components view object with validation.

   For example, if a job requires information regarding duration, date, and time, create an ADF Business Components view object with the properties `duration`, `date`, and `time`.
3. Create a job definition in JDeveloper using the wizard.

   If using an ADF Business Components view object to collect additional values at runtime from end users, specify the name of the view object as a property of the job definition.
4. Deploy the job.
5. Test the job.
6. Create the end user job request submission interface.

   For more information about creating the end user job request submission interface, see Section 9.14, "Creating an Oracle ADF User Interface for Submitting Job Requests".

9.3.2 What Happens at Runtime: How a Scheduled Job Is Created and Implemented in JDeveloper

An Oracle ADF interface is provided to enable application end-users to submit job requests from an Oracle Fusion application. The Oracle ADF interface is easily integrated into an Oracle Fusion application. Once a job request is submitted through the interface, Oracle Enterprise Scheduler Service runs the job as scheduled.

9.4 Creating a Job Definition

In order to submit a job request, you must first create a job definition.

9.4.1 How to Create a Job Definition

A job definition and job type are required to submit a job request.

- Job Definition: This is the basic unit of work that defines a job Request in Oracle Enterprise Scheduler.

- Job Type: This specifies an execution type and defines a common set of properties for a job request.

The extensions to Oracle Enterprise Scheduler Service provide the following execution types:

- **JavaType**: for job definitions that are implemented in Java and run in the container.
- **SQLType**: for job definitions that run as PL/SQL stored procedures in a database server.
- **CJobType**: for job definitions that are implemented in C and run in the container.
- **PerlJobType**: for job definitions that are implemented in Perl and run in the container.
- **SqlLdrJobType**: for job definitions that are implemented in SQL*Loader and run in the container.
- **SqlPlusJobType**: for job definitions that are implemented in SQL*Plus and run in the container.
- **BIPJobType**: for job definitions that are executed as Oracle BI Publisher (*BIP*) reports. Oracle BI Publisher jobs require configuring the parameter `reportID`.

  For more information about defining a Business Intelligence Publisher job, see the *Business Intelligence Publisher Administrator's and Developer's Guide* and the *Business Intelligence Publisher Report Designer's Guide*.
- **HostJobType**: for job definitions that run as host scripts executed from the command line.

Before you begin:

If your job definition requires additional properties to be filled in by end users at submission time, you'll need to create a view object that defines these properties. The view object must be associated with the job definition you create. The view object is later associated with the user interface you create to allow end users to submit job requests along with the properties at submission time.

For more information about defining properties to be filled in at runtime by end users, see Section 9.14, "Creating an Oracle ADF User Interface for Submitting Job Requests."

To create a new job definition in Oracle JDeveloper:

1. In Oracle JDeveloper, create an Oracle Fusion web application by clicking the Application Menu icon on the Application Navigator, selecting **New Project** > **Projects > Generic Project** and clicking **OK**.
2. Right-click the project and select Properties. In the Resources tab, add the directory `$MW_HOME/jdeveloper/integration/ess/extJobTypes`.
3. If your job includes any properties to be filled in by end users using an Oracle ADF user interface at runtime, create an ADF Business Components view object with validation and the parameters to be filled in by end users.
   a. Right-click the Model project and select **Properties**. In the Resource Bundle section, configure one bundle per file and select resource bundle type **Xliff Resource Bundle**.
   b. Define attributes for the view objects sequentially, `ATTRIBUTE1`, `ATTRIBUTE2`, and so on, with an attribute for each required parameter. Use ADF Business Components attribute control hints to specify required prompt, validation,

and formatting for each parameter. For more information, see the chapter "Creating a Business Domain Layer Using Entity Objects" in *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

c. Add the property `parametersVO` to your job definition and specify the fully qualified path of the view object as the value of `parametersVO`. For example, set `parametersVO` to `oracle.my.package.TestVO`. A maximum of 100 attributes can be used for `parametersVO`. The attributes should be named incrementally, for example `ATTRIBUTE1`, `ATTRIBUTE2`, and so on.

d. Define the following required properties:

- `jobDefinitionName`: The short name of the job.
- `jobDefinitionApplication`: The short name of the application running the job.
- `jobPackageName`: The name of the package running the job.

Additional properties can be defined as shown in Table 9–1.

Table 9–1 Additional Job Definition Properties

Property	Description
`completionText`	An optional string value that can be used to communicate details of the final state of the job. This property value is displayed in the UI used to monitor job request submissions in the details section of the job request. It can be useful for displaying a short explanation as to why a request ended in an error or warning state.
`CustomDatacontrol`	The name of the data control for the application to which the parameter task flow is bound. Following is an example. `<parameter name="CustomDatacontrol" data-type="string">ExtParameterAM</parameter>` Use this property when adding a custom task flow to an Oracle ADF user interface used to submit job requests at run time. For more information, see Section 9.14.2, "How to Add a Custom Task Flow to an Oracle ADF User Interface for Submitting Job Requests."
`defaultOutputExtension`	The suffix of the output file. Possible values are `txt`, `xml`, `pdf`, `html`.
`enableTimeStatistics`	A boolean parameter that enables or disables the accumulation of time statistics (Y or N).
`enableTrace`	A numerical value that indicates the level of tracing control for the job. Possible values are as follows: ■ **1**: Database trace ■ **5**: Database trace with bind ■ **9**: Database trace with wait ■ **13**: Database trace with bind and wait ■ **16**: PL/SQL profile ■ **17**: Database trace and PL/SQL profile ■ **21**: Database trace with bind and PL/SQL profile ■ **25**: Database trace with wait and PL/SQL profile ■ **29**: Database trace with bind, wait and PL/SQL profile
`executionLanguage`	Stores the preferred language in which the job request should run.

Table 9–1 (Cont.) Additional Job Definition Properties

Property	Description
executionNumchar	The numeric characters used in the preferred language in which the job runs, as defined by executionLanguage.
executionTerritory	The territory of the preferred language in which the job runs, as defined by executionLanguage.
EXT_ PortletContainerWebModule	Specifies the name of the web module for the Oracle Enterprise Scheduler UI application to use as a portlet when submitting a job request. The Oracle Enterprise Scheduler central UI looks up the producer from the topology based on the registered producer application name derived from EXT_ PortletContainerWebModule.
incrementProc	Enables a PL/SQL procedure evaluated at runtime which calculates the next set of date parameter values for a recurring request. Enter the name of the PL/SQL procedure. The procedure expects one argument—a number signifying the change in milliseconds between the start dates of the first and current requests.

Table 9–1 (Cont.) Additional Job Definition Properties

Property	Description
incrementProcArgs	A list of comma-separated date arguments to be incremented. The incrementProc property is used to increment these values. Alternatively, a default value is used if the property incrementProc is not defined. Enter a list of argument numbers to identify which job arguments are to be incremented (for example, "1, 2, 5"). In the example shown here, an incrementProc procedure calculates the next set of date parameter values for a recurring request. The procedure expects one argument: a number signifying the change in milliseconds between the start dates of the first and current requests.

```
    -- incr_test - Sample PL/SQL incrementProc procedure
    -- This procedure gets the list of arguments to be incremented
    -- using the incrementProcArgs property and increments each
    -- argument by the delta provided. This behavior is identical
    -- to the default behavior if no incrementProc is set for the
    -- job.
procedure incr_test(   delta IN number ) is
   request_id number;
   incrProcArgs varchar2(200);
   curr_arg_n varchar2(100);
   curr_arg_v varchar2(2000);
   del_pos number := 0;
   prev_pos number := 1;
   old_date date;
   new_date date;
   delta_days number;
   begin
      request_id := FND_JOB.REQUEST_ID;
      delta_days := delta / (1000*60*60*24);

      -- incrProcArgs must be defined for this procedure to be
      -- called.
      incrProcArgs := ESS_RUNTIME.GET_REQPROP_VARCHAR(request_id,
                      FND_JOB.INCR_PROC_ARGS_P) || ',';

      LOOP
     del_pos := INSTR(incrProcArgs, ',', prev_pos);
     EXIT WHEN del_pos = 0;

     curr_arg_n := FND_JOB.SUBMIT_ARG_PREF_P || SUBSTR(incrProcArgs,
                   prev_pos, del_pos-prev_pos);

     curr_arg_v := ESS_RUNTIME.GET_REQPROP_VARCHAR(request_id,
                                                   curr_arg_n);

     old_date := FND_DATE.CANONICAL_TO_DATE(curr_arg_v);
     new_date := old_date + delta_days;

     ESS_RUNTIME.UPDATE_REQPROP_VARCHAR(request_id, curr_arg_n,
                                        FND_DATE.DATE_TO_
                                        CANONICAL(new_date));

     prev_pos := del_pos+1;
      END LOOP;
   end incr_test;
```

Table 9–1 (Cont.) Additional Job Definition Properties

Property	Description
`logLevel`	The level at which events are logged (between 0 and 4). Each job type has a `logLevel` of 1 by default. This optional value is used to override the job type `logLevel` in the job definition. For more information about log levels, see the *Enterprise Scheduler Service Developer's Guide*.
`optimizerMode`	This flag enables setting the database optimizer mode for the job. Optimizer mode is useful for fine-tuning performance.
`parametersVO`	The ADF Business Components view object you define for additional properties to be entered at runtime by end users using an Oracle ADF user interface.
`ParameterTaskflow`	Enter the name of the task flow as a parameter. The name of the `taskflow.xml` file must be the same as the `taskflowId`. Following is an example. `<parameter name="ParameterTaskflow" data-type="string">/WEB-INF/oracle/apps/prod/project/ParamTestTaskFlow.xml#ParamTestTaskFlow</parameter>` Use this property when adding a custom task flow to an Oracle ADF user interface used to submit job requests at run time. For more information, see Section 9.14.2, "How to Add a Custom Task Flow to an Oracle ADF User Interface for Submitting Job Requests."
`reportID`	The BIP report value specified in the Oracle BI Publisher repository. Required parameter for Oracle BI Publisher jobs only.
`rollbackSegment`	Enables setting a database rollback segment for the job, which will be used until the first commit. When implementing the rollback segment, use `FND_JOB.AF_COMMIT` and `FND_JOB.AF_ROLLBACK` to commit and rollback.
`srsFlag`	A boolean parameter (Y or N) that controls whether the job displays in the job request submission user interface (see Section 9.14, "Creating an Oracle ADF User Interface for Submitting Job Requests").
`SYS_runasApplicationID`	Enables elevating access privileges for completing a scheduled job. For more information about elevating access privileges for the completion of a particular job, see Section 9.13, "Elevating Access Privileges for a Scheduled Job."

4. Create a new job. From the New Gallery, select **SOA Tier** > **Enterprise Scheduler Metadata** and click **Job Definition**.
5. In the Job Definition Name & Location page in the Job Definition Creation Wizard, do the following:
   - **Name**: Enter a name for the job.
   - **JobType**: Select the job type from the drop-down list.

   Click **Finish**. The new job definition displays.
6. Edit the following properties in the job definition as required for the selected job type:
   - **JavaJobType**: Uncheck the read-only checkbox next to `className` and set its value to the value of the business logic class.
   - **PlsqlJobType**: Uncheck the read-only checkbox next to `procedureName` and set its value to the name of the procedure (such as `myprocedure.proc`). Create a new parameter named `numberOfArgs`. Set `numberOfArgs` to the number of job submission arguments, excluding `errbuf` and `retcode`.
   - **CJobType**: Add the parameter `executableName` and set its value to the name of the C job to be executed. The executable file identified by the

executableName parameter must exist in the directory $APPLICATIONS_BASE/$APPLBIN.

- **PerlJobType**: Add the parameter executableName and set its value to the name of the Perl script.
- **SqlLdrJobType**: Add the parameter executableName and set its value to the name of the control file to be executed (located under PRODUCT_TOP/$APPLBIN). Add SQL*Loader options such (such as direct=yes) as a sqlldr.directoption parameter in the job definition.
- **SqlPlusJobType**: Add the parameter executableName and set its value to the name of the SQL*Plus job script to be executed (located under PRODUCT_TOP/$APPLSQL).
- **HostJobType**: Add the parameter executableName and set its value to the name of the host script job to be executed. The executable file identified by the executableName parameter must exist in the directory PRODUCT_TOP/$APPLBIN.

Note: Make sure the $APPLBIN and $APPLSQL variables are configured in the environment.properties file. The $APPLBIN and $APPLSQL variables point to the location of executable files under PRODUCT_TOP. These variables enable the extensions to Oracle Enterprise Scheduler Service to locate the jobs to be run. Typically, these variables are set in a pre-existing environment properties file in the system.

9.4.2 How to Define File Groups for a Job

A file group is a collection of output files such as text files, XML files, and so on. File groups enable categorizing files together for a specific purpose, such as file groups for human resources or financial reports.

File groups are used for post-processing jobs such as Business Intelligence Publisher jobs. Using post-processing actions, the results of a job can be saved as an HTML file, for example, or printed. File groups specify the type of post-processing action to be taken for a given job.

There are two types of file groups: output and layout. Post-processing layout actions create additional output files using the job request output files. For example, an XML job output file can be processed as an HTML or PDF file.

Post-processing output actions act upon job request output files by printing, faxing, or e-mailing the files, for example. Output post-processing actions can be taken on job request output files, as well as files created by layout post-processing actions. For example, a job request output XML file can be converted to a PDF file using layout post-processing actions, and then e-mailed using output post-processing actions.

For more information about defining a Business Intelligence Publisher job, see the *Business Intelligence Publisher Administrator's and Developer's Guide* and the *Business Intelligence Publisher Report Designer's Guide*.

To define file group properties:

1. In the job definition for which you want to define post-processing, define a file group.
   a. Name the property *Program.FMG.*

b. For the value of the property, enter a list of comma-separated file management groups, where each file group is prefixed by an L or O to indicate a layout or output file group, respectively. A sample file group property is shown in Example 9–1:

Example 9–1 File Group Property Sample Value

```
Program.FMG = L.MYXML, O.ALL, O.PDF
```

Three file groups are listed in this example.

2. In the job definition, create a property containing a regular expression used to filter the files in the output work directory of the job request. Any output files that match the filter will be part of the relevant file group.

Example regular expressions are shown in Example 9–2, Example 9–3 and Example 9–4.

Example 9–2 File Group Regular Expression Filtering for All Files with the Suffix XML

```
MYXML = '.*.\xml$'
```

Example 9–3 File Group Regular Expression Filtering for All Files

```
ALL = '.*$'
```

Example 9–4 File Group Regular Expression Filtering for All Files with the Suffix PDF

```
PDF = '.*.\pdf$'
```

An example of file group properties in a job definition is shown in Example 9–5.

Example 9–5 File Group Properties with File Group Regular Expression Filtering

```
Program.FMG = L.MYXML, O.ALL, O.PDF
MYXML = '.*.\xml$'   ALL = '.*$'   PDF = '.*.\pdf$'
```

These properties specify the use of the Business Intelligence Publisher post-processing action on the `MYXML` file group, followed by the print post-processing action on either `ALL` or `PDF` file groups.

3. Optionally, rename the file group and store it in Oracle Metadata Store so that it displays in a more user-friendly way in the scheduled job request submission UI.

9.4.3 What Happens When You Create a Job Definition

The job definition is written to an XML file called `<job name>.xml`.

9.4.4 What Happens at Runtime: How Job Definitions Are Created

The Fusion application passes the job definition file to Oracle Enterprise Scheduler Service, which runs the job defined in the file.

9.5 Configuring a Spawned Job Environment

Configuring a spawned job involves creating an environment file and configuring an Oracle wallet.

9.5.1 How to Create an Environment File for Spawned Jobs

Spawned jobs require an `environment.properties` file to provide the correct environment for execution. The `environment.properties` file should be located in the `config/fmwconfig` directory under the domain.

Additional environment variables may be added to the same directory in a similar file called `env.custom.properties`. Variables defined in this file take precedence over those in the `environment.properties` file.

Similarly, server-specific environment variables may be set in the server config directory in files called `environment.properties` and `env.custom.properties`.

Before you begin:

The following variables are used to identify the correct interpreters for various spawned job types:

- `AFSQLPLUS`: The executable for SQL*Plus scripts.
- `AFSQLLDR`: The executable for SQL*Loader uploads.
- `AFPERL`: The Perl interpreter.
- `ATGPF_TOP`: The `TOP` directory for ATGPF files, needed to locate key files for SQL*Plus and Perl jobs.

The following environment properties are available to all spawned jobs:

- `REQUESTID`: The request ID of the current job request.
- `WORK_DIR_ROOT`: The directory on the local file system where the request can perform file operations.
- `OUTPUT_WORK_DIR`: The directory to which the job writes all output files.
- `LOG_WORK_DIR`: The directory to which the job writes all log files.
- `INPUT_WORK_DIR`: The directory to which input files are saved before the job is spawned.
- `OUTFILE_NAME`: The default name for the job output file.
- `LOGFILE_NAME`: The name of the log file for the job.
- `USER_NAME`: The name of the user submitting the job. The job runs in the context of this user.
- `REQUEST_HANDLE`: The Oracle Enterprise Scheduler request handle for the current request.

The environment variables must point to the client `ORACLE_HOME` and environment so that spawned jobs can connect to the database.

Note: Make sure the variables you define in the `environment.properties` file do not include any trailing spaces. Follow the guidelines required by `java.util.properties`.

Make sure to restart the server after editing the `environment.properties` file.

To create an environment file for spawned jobs:

1. Use a text editor to create an `environment.properties` file for the spawned job.

2. Set the following environment variables in the `environment.properties` file:
   - `LD_LIBRARY_PATH`
   - `ORACLE_HOME`
   - `PATH`: The full path of the spawned job. In Windows environments, the `PATH` must include all directories that are normally part of `LD_LIBRARY_PATH`.
   - `TNS_ADMIN`: The directory which stores files related to the database connection (such as `tnsnames.ora`, `sqlnet.ora`).
   - `TWO_TASK`: The TNS name identifying the database to which spawned jobs should connect. In Windows environments, the environment variable is `LOCAL`.
3. Configure the following variables, which are required to locate spawned jobs:
   - `APPLBIN`: C executables and SQL*Loader control files must reside in the `$APPLBIN` directory under the product `TOP`.
   - `APPL_TOP`: Set this property to the top level directory where the `bin` directory of C executables resides.
   - `APPLSQL`: SQL*Plus scripts must reside in the `$APPLSQL` directory under the product `TOP`. This means that the product `TOP` should be accessible to the environment.
   - `ATGPF_TOP`: This variable is required for SQL*Plus jobs. This should point to where the wrapper script is available.
4. Save the `environment.properties` file and restart the server.

9.5.2 How to Configure an Oracle Wallet for Spawned Jobs

Use the `TNS_ADMIN` and `ORACLE_HOME` variables specified in the `environment.properties` file created in Section 9.5.1.

A configured Oracle wallet enables spawned jobs to connect to the database at the command line. A provisioned Fusion applications environment will have this wallet pre-configured.

To configure an Oracle wallet for the spawned job:

1. At the prompt, enter the following commands as shown in Example 9–6.

Example 9–6 Creating a Wallet

```
cd $TNS_ADMIN
mkdir wallet
mkstore -wrl ./wallet -create
```

2. When prompted, choose a password for the wallet.
3. At the prompt, enter the following command as shown in Example 9–7.

Example 9–7 Creating Wallet Credentials

```
mkstore -wrl ./wallet -createCredential <$TWO_TASK> fusion_runtime <fusion_
runtime_password password>
```

where `TWO_TASK` is the variable in the `environment.properties` file and <*fusion password*> is the password for the `fusion` username.

This command creates permissions for accessing the wallet.

4. When prompted, enter the wallet password created earlier.
5. In a text editor, create a file called `sqlnet.ora` that includes the lines shown in Example 9–8.

Example 9–8 Create a File Called sqlnet.ora

```
SQLNET.WALLET_OVERRIDE = TRUE
WALLET_LOCATION =
  (SOURCE =
    (METHOD = FILE)
    (METHOD_DATA =
  (DIRECTORY = <$TNS_ADMIN>/wallet)
  )
 )
```

6. In a text editor, create a file called `tnsnames.ora` that includes the lines shown in Example 9–9.

Example 9–9 Create a File Called tnsnames.ora

```
dbname =
  (DESCRIPTION =
    (ADDRESS =
       (PROTOCOL = TCP)
       (HOST = host.us.oracle.com)
       (PORT = 1521)
     )
    (CONNECT_DATA = (SID-sidname))
  )
```

7. Execute the following commands as shown in Example 9–10.

Example 9–10 Set Directory and File Permissions

```
chmod 755 wallet
chmod 744 wallet/cwallet.sso
```

The first command enables anyone to read and execute files in the directory, while reserving write access to the directory creator.

The second command enables only the file owner to read, write and execute the file, while anyone can read the file.

8. Test the wallet by connecting to it. Execute the following command as shown in Example 9–11.

Example 9–11 Connect to the Wallet

```
sqlplus /@<$TWO_TASK>
```

9.5.3 What Happens When You Configure a Spawned Job Environment

A configured Oracle wallet enables spawned jobs to connect to the database at the command line.

9.6 Implementing a PL/SQL Scheduled Job

Implementing a PL/SQL scheduled job requires creating a job definition and creating a PL/SQL package.

9.6.1 Standards and Guidelines for Implementing a PL/SQL Scheduled Job

Be sure to run sub-requests through Oracle Enterprise Scheduler Service using theOracle Enterprise Scheduler APIs to access Oracle Enterprise Scheduler.

A PL/SQL stored procedure scheduler job should have a signature with the first two arguments being `errbuf` and `retcode`. The remaining arguments are used as required for defining job parameters. All arguments have a data type of `varchar2`.

9.6.2 How to Define Metadata for a PL/SQL Scheduled Job

Create a job definition as described in Section 9.4, "Creating a Job Definition."

PL/SQL jobs require setting an additional property `numberOfArgs` in the job definition. This property identifies the number of job submission arguments (not including the required arguments `errbuf` and `retcode`.)

9.6.3 How to Implement a PL/SQL Scheduled Job

Oracle Enterprise Scheduler Service provides runtime PL/SQL APIs for implementing PL/SQL jobs and running the jobs using Oracle Enterprise Scheduler. A view object is defined and associated with the job definition for the job.

When create a PL/SQL job, use the `fusion` database user. For information about granting access privileges to database users in the context of Oracle Fusion Applications, see the "Security" section in *Oracle Fusion Applications Developer's Guide*.

Before you begin:

For more information about implementing a PL/SQL stored procedure scheduled job see Chapter 6, "Creating and Using PL/SQL Jobs."

To implement a PL/SQL scheduled job:

1. Create a PL/SQL package, including at minimum the required `errbuf` and `retcode` arguments.
2. Deploy the package to a database.
3. Test the package.

9.6.4 What Happens When You Implement a PL/SQL Job

The sample PL/SQL job shown in Example 9–12 provides a signature of a PL/SQL procedure run as a job. The first two arguments to the PL/SQL procedure, `errbuf` and `retcode`, are required. The remaining arguments are properties filled in by end users and passed to Oracle Enterprise Scheduler when the job is submitted.

The example shown in Example 9–12 illustrates a sample PL/SQL job that uses the PL/SQL API.

Example 9–12 Running a Job Using the PL/SQL API

```
procedure fusion_plsql_sample(
-- The first two arguments are required: errbuf and retcode
--
```

```
                                errbuf    out NOCOPY varchar2,
                                retcode   out NOCOPY varchar2,

-- The errbuf is logged when a job request ends in a warning or error state to
-- provide a quick indication as to why the job request ended in an error or
-- warning state.
-- Job submission arguments, as collected from the view object associated with the
-- job as configured in the job definition. The view object is used to present a
-- user interface to end users, allowing them to enter the properties listed in
-- the following lines of code.
-- interface. These values are submitted by the end user.
--
                                run_mode  in  varchar2 default 'BASIC',
                                duration  in  varchar2 default '0',
                                p_num     in  varchar2 default NULL,
                                p_date    in  varchar2 default NULL,
                                p_varchar in  varchar2 default NULL) is

  begin
       -- Write log file content using FND_FILE API
       FND_FILE.PUT_LINE(FND_FILE.LOG, "About to run the sample program");

       -- Implement the business logic of the job here.
       --
       FND_FILE.PUT_LINE(FND_FILE.OUT, " RUN MODE : " || run_mode);
       FND_FILE.PUT_LINE(FND_FILE.OUT, "DURATION: " || duration);
       FND_FILE.PUT_LINE(FND_FILE.OUT, "P_NUM: " || p_num);
       FND_FILE.PUT_LINE(FND_FILE.OUT, "P_DATE: " || p_date);
       FND_FILE.PUT_LINE(FND_FILE.OUT, "P_VARCHAR: " p_varchar);

       -- Retrieve the job completion status which is returned to Oracle
       -- Enterprise Scheduler.
       errbuf := fnd_message.get("FND", "COMPLETED NORMAL");
       retcode := 0;
 end;
```

The sample shown in Example 9–13 illustrates a PL/SQL job with a sub-request submission. The `no_requests` argument identifies the number of sub-requests that must be submitted.

Example 9–13 Submitting a Sub-request Using the PL/SQL Runtime API

```
procedure fusion_plsql_subreq_sample(
                                errbuf    out NOCOPY varchar2,
                                retcode   out NOCOPY varchar2,
                                no_requests  in  varchar2 default '5',
                                ) is
       req_cnt number := 0;
       sub_reqid number;
       submitted_requests varchar2(100);
       request_prop_table_t jobProp;
  begin
       -- Write log file content using FND_FILE API
       FND_FILE.PUT_LINE(FND_FILE.LOG, "About to run the sample program with
sub-request functionality");

       -- Requesting the PAUSED_STATE property set by job identifies request as
       -- having started for the first time or restarting after being paused.
       if ( ess_runtime.get_reqprop_varchar(fnd_job.job_request_id, 'PAUSED_
STATE') ) is null )  -- first time start
```

```
       then
          -- Implement the business logic of the job here.
          FND_FILE.PUT_LINE(FND_FILE.OUT, " About to submit sub-requests : " ||
no_requests);

          -- Loop through all the sub-requests.
          for req_cnt 1..no_requests loop
            -- Retrieve the request handle and submit the subrequest.
            sub_reqid := ess_runtime.submit_subrequest(request_handle => fnd_
job.request_handle,
                                        definition_name => 'sampleJob',
                                        definition_package => 'samplePkg',
                                        props => jobProp);
            submitted_requests := sub_reqid || ',';
          end loop;

          -- Pause the parent request.
          ess_runtime.update_reqprop_varchar(fnd_job.request_id, 'STATE', ess_
job.PAUSED_STATE);

          -- Update the parent request with the state of the sub-request, enabling
          -- the job to retrieve the status during restart.
          ess_runtime.update_reqprop_int(fnd_job.request_id, 'PAUSED_STATE',
submitted_requests);

       else
          -- Restart the request, retrieve job completion status and return the
          -- status to Oracle Enterprise Scheduler Service.
          errbuf := fnd_message.get("FND", "COMPLETED NORMAL");
          retcode := 0;
       end if;
 end;
```

9.6.5 What Happens at Runtime: How a PL/SQL Job is Implemented

Oracle Enterprise Scheduler Service calls routines to initialize the context of the PL/SQL job, including PL/SQL global values, local values (such as language and territory), and request-specific values such as request ID and request handle.

The view object associated with the job definition displays a user interface so that end users may fill in values for each property. The Oracle Fusion web application calls Oracle Enterprise Scheduler using the provided APIs and submits the job request. Oracle Enterprise Scheduler runs the job, which calls the context routines and then runs the job logic. The job ends with a `retcode` value of 0, 1, 2 or 3, representing `SUCCESS`, `WARNING`, `FAILURE` or `BUSINESS ERROR`, respectively. The Oracle Fusion web application can retrieve the result from Oracle Enterprise Scheduler and display it in the user interface.

9.7 Implementing a SQL*Plus Scheduled Job

Implementing a SQL*Plus scheduled job involves writing a SQL*Plus script and configuring an environment file for the job.

9.7.1 Standards and Guidelines for Implementing a SQL*Plus Scheduled Job

Be sure to run sub-requests through Oracle Enterprise Scheduler Service using the Oracle Enterprise Scheduler APIs to access Oracle Enterprise Scheduler.

9.7.2 How to Implement a SQL*Plus Job

Implementing a SQL*Plus stored procedures job involves writing the SQL*Plus script, storing the script and configuring a spawned job environment.

To implement a SQL*Plus job:

1. Write the SQL*Plus job as a SQL*Plus script. Include the `FND_JOB.set_sqlplus_status` call so as to report the final job status.

   Include the following in the SQL*Plus scheduled job:

   - `FND_JOB.set_sqlplus_status`: Call to report the final job status. Statuses include:
     - `FND_JOB.SUCCESS_V`: Success.
     - `FND_JOB.WARNING_V`: Warning.
     - `FND_JOB.FAILURE_V`: Failure.
     - `FND_JOB.BIZERR_V`: Business Error.
   - `FND_FILE` routines: Can be used for producing log data and output files.
   - `FND_JOB` API for request values: API calls are initialized for SQL*Plus jobs.

   Note: SQL*Plus jobs must **not** exit.

2. Store the script under `PRODUCT_TOP/$APPLSQL`.
3. Configure the spawned job environment as described in Section 9.5, "Configuring a Spawned Job Environment". Be sure to configure the `ATGPF_TOP` value in the `environment.properties` file for spawned jobs.
4. Run and test the job.

9.7.3 How to Use the SQL*Plus Runtime API

Oracle Enterprise Scheduler Service provides runtime SQL*Plus APIs for implementing SQL*Plus jobs and running the jobs using Oracle Enterprise Scheduler.

This sample SQL*Plus job provides a signature of a SQL*Plus procedure run as a job. Any necessary arguments are properties filled in by end users and passed to Oracle Enterprise Scheduler when the job is submitted. A view object is defined and associated with the job definition for the job. The view object is then used to display a user interface so that end users may fill in values for each property. Finally, the sample prints to an output file.

9.7.4 What Happens When You Implement a SQL*Plus Job

Example 9–14 shows a sample SQL*Plus scheduled job, which is executed by a wrapper script.

Example 9–14 Implementing a SQL*Plus Scheduled Job

```
SET VERIFY OFF
SET linesize 132

WHENEVER SQLERROR EXIT FAILURE ROLLBACK;
WHENEVER OSERROR EXIT FAILURE ROLLBACK;
REM dbdrv: none
```

```
/* ---------------------------------------------------------------------*/

DECLARE
errbuf        varchar2(240) := NULL;
retval        boolean;
run_mode      varchar2(200)  := '&1';

BEGIN
        DBMS_OUTPUT.PUT_LINE(run_mode);

        update dual set dummy = 'Q';

    FND_FILE.PUT_LINE(FND_FILE.LOG, 'Parameter 1 = ' || nvl(run_mode,'NULL'));

/*  print out test message to log file and output file  */
/*  by making direct call to FND_FILE.PUT_LINE          */
/*  from sql script.                                    */

    FND_FILE.PUT_LINE(FND_FILE.LOG,   '
                        ');
    FND_FILE.PUT_LINE(FND_FILE.LOG,   '----------------------------------------
----------------------');
    FND_FILE.PUT_LINE(FND_FILE.LOG,   'Printing a message to the LOG FILE
                        ');
    FND_FILE.PUT_LINE(FND_FILE.LOG,   '----------------------------------------
----------------------');
    FND_FILE.PUT_LINE(FND_FILE.LOG,   'SUCCESS!
                        ');
    FND_FILE.PUT_LINE(FND_FILE.LOG,   '
                        ');
    FND_FILE.PUT_LINE(FND_FILE.OUTPUT,'----------------------------------------
----------------------');
    FND_FILE.PUT_LINE(FND_FILE.OUTPUT,'Printing a message to the OUTPUT FILE
                        ');
    FND_FILE.PUT_LINE(FND_FILE.OUTPUT,'----------------------------------------
----------------------');
    FND_FILE.PUT_LINE(FND_FILE.OUTPUT,'SUCCESS!
                        ');
    FND_FILE.PUT_LINE(FND_FILE.OUTPUT,'
                        ');

retval :=  FND_JOB.SET_SQLPLUS_STATUS(FND_JOB.SUCCESS_V);

END;
/
COMMIT;
-- EXIT; Fusion Applications  SQL*Plus Jobs must not exit.
```

9.7.5 What Happens at Runtime: How a SQL*Plus Job Is Implemented

Oracle Enterprise Scheduler Service calls routines in a wrapper script to initialize the context of the SQL*Plus job, including global values, local values (such as language and territory), and request-specific values such as request ID and request handle. The wrapper script introduces the prologue of commands shown in Example 9–15.

Example 9–15 SQL*Plus wrapper script

```
SET TERM OFF
```

```
SET PAUSE OFF
SET HEADING OFF
SET FEEDBACK OFF
SET VERIFY OFF
SET ECHO OFF
SET ESCAPE ON

WHENEVER SQLERROR EXIT FAILURE
```

The Fusion application calls Oracle Enterprise Scheduler using the provided APIs. Oracle Enterprise Scheduler runs the job, and the final job status—`SUCCESS`, `WARNING`, `BUSINESS ERROR` or `FAILURE`—is communicated to Oracle Enterprise Scheduler. The Oracle Fusion web application can retrieve the result from Oracle Enterprise Scheduler and display it in the user interface.

9.8 Implementing a SQL*Loader Scheduled Job

Implementing a SQL*Loader scheduled job involves creating a SQL*Loader control file and configuring a spawned job environment.

9.8.1 How to Implement a SQL*Loader Scheduled Job

Before you begin:

Keep in mind that the control file and data file must conform to the following SQL*Loader standards:

- Place control files in the `$APPLBIN` directory under the product TOP.
- Make sure that the control file's name is the same as the `executableName` parameter in the job definition.
- Ensure that the data file's location is the first submit argument to the job.
- Add SQL*Loader options such as `direct=yes`, if needed, as the `sqlldr.directoption` parameter in the job definition.

To implement a SQL*Loader scheduled job:

1. Create a SQL*Loader control file (.ctl).
2. Enter the full path of the data file as the first submit argument to the job.
3. Store the control file under `PRODUCT_TOP/$APPLBIN`.
4. Configure the spawned job environment as described in Section 9.5, "Configuring a Spawned Job Environment."
5. Test the file.

9.8.2 What Happens When You Implement a SQL*Loader Scheduled Job

A sample SQL*Loader scheduled job is shown in Example 9–16.

Example 9–16 Sample SQL*Loader scheduled job

This sample control file will upload data from the data file into the fnd_applcp_test table, into the columns listed here (id1, id2, ..., mesg). See the SQL*Loader documentation for more information on writing control files.

```
OPTIONS (silent=(header,feedback,discards))
```

```
LOAD DATA
INFILE *
INTO TABLE fnd_applcp_test
APPEND
FIELDS TERMINATED BY ','
(id1,
 id2,
 id3,
 func CHAR(30),
 time SYSDATE,
 action CHAR(30),
 mesg CHAR(240))
```

9.9 Implementing a Perl Scheduled Job

Implementing a Perl scheduled job involves creating a job definition, enabling the Perl job to connect to a database and configuring a spawned job environment.

9.9.1 How to Implement a Perl Scheduled Job

Before you begin:

For more information about creating a Perl scheduled job see Chapter 6, "Creating and Using PL/SQL Jobs."

To implement a Perl scheduled job:

1. Place the Perl job under the directory `PRODUCT_TOP/$APPLBIN`.
2. Create a job definition for the Perl job, setting the `executableName` parameter to the name of the Perl script. The following functions can be used in the Perl script:
   - `writeln()`: Write a message to the log file.
   - `timestamp()`: Write a timestamped message.
3. To enable the Perl job to connect to a database, use `/@$TWO_TASK` as a connection string without specifying a username or password.
4. Configure the spawned job environment as described in Section 9.5, "Configuring a Spawned Job Environment". The context provides values for the following:
   - `reqid`: The request ID.
   - `outfile`: The full path to the output file.
   - `logfile`: The full path to the log file.
   - `username`: The name of the user submitting the job request.
   - `log`: The log object.
5. Implement an exit code for the job, with values of 0, 2 or 3 representing the following states: success, warning and business error. All other values represent an errored state.
6. Test the job.

9.9.2 What Happens When You Implement a Perl Scheduled Job

Example 9–17 shows a sample scheduled Perl job which does the following:

1. Checks for basic or full mode.
2. Prints arguments.
3. Gets the scheduled job request context object.
4. Retrieves contextual information about the scheduled job request, which is stored in the context object.
5. Writes the request to the log file.
6. Prints information as required.

Example 9–17 Perl Scheduled Job

```
# dbdrv: none

use strict;

(my $VERSION) = q$Revision: 120.1 $ =~ /(\d+(\.\d+)*)/;

print_header("Begin Perl testing script (version $VERSION)");

# check first argument for BASIC or FULL mode
# if not FULL mode, exit successfully without doing anything
if (! $ARGV[0] || uc($ARGV[0]) ne "FULL") {
    exit(0);
}

# -- If argument #1 was passed, use it as a sleep time
if ($ARGV[1]) {

    if ($ARGV[1] =~ /\D/) {
      print "** Argument #1 is not a valid number, unable to sleep!\n\n";
        } else {
      printf("Sleeping for %d seconds...\n", $ARGV[1]);
      sleep($ARGV[1]);
        }
}

# -- Arguments
print_header("Arguments");
my $i = 1;
foreach (@ARGV) {
  print "Argument #", $i++, ": $_\n";
}

# -- Get the request context object
my $context = get_context();

# -- Use this object to retrieve context information about this request

print_header("Context Information");
printf "Request id \t= %d\n", $context->reqid();
printf "User name \t= %d\n", $context->username();
printf "Logfile \t= %s\n", $context->logfile();
printf "Outfile \t= %s\n", $context->outfile();

# -- Writing to the request log file
print_header("Writing to log file");

# -- retrieve a Logfile object from the context
```

```
my $log = $context->log();
$log->writeln("This message should appear in the request logfile");
$log->timestamp("This is a timestamped message to the request logfile");

print "Wrote two messages to the request logfile\n";

# -- Print out some useful information

print_header("Environment");
foreach (sort keys %ENV) {
    print "$_=$ENV{$_}\n";
}

print_header("Perl Information");
print "PROCESS ID = $$\n";
print "REAL USER ID = $<\n";
print "EFF USER ID = $>\n";
print "SCRIPT NAME = $0\n";
print "PERL VERSION = $]\n";
print "OS NAME = $^O\n";
print "EXE NAME = $^X\n";
print "WARNINGS ON = $^W\n";

print "\n\@INC path:\n";
foreach  (@INC) {
    print "$_\n";
}

print "\nAll loaded perl modules:\n";
foreach (sort keys %INC) {
    print "$_ => $INC{$_}\n";
}

# -- Exiting the script
# -- The exit status of the script will be used as the request exit status.
# -- A zero exit status is reported as state of success.
# -- An exit status of 2 is reported as a warning state.
# -- An exit status of 3 is reported as a business error state.
# -- Any other exit status is reported as an error state.

print_header("Exiting script with status 0. (Normal completion)");
exit(0);

sub print_header {

  my $msg = shift;
  print "\n\n", "-" x 40, "\n", $msg, "\n", "-" x 40, "\n";

}
```

9.10 Implementing a C Scheduled Job

The main steps required to implement a C scheduled job are as follows:

- Creating a job definition
- Configuring a spawned job environment
- Implementing and testing a C scheduled job

9.10.1 How to Define Metadata for a C Scheduled Job

Create a job definition as described in Section 9.4, "Creating a Job Definition".

9.10.2 How to Implement a C Scheduled Job

To implement a C scheduled job:

1. In a separate function or file rather than in `main`, implement your required business logic.

   Include the following header files:

   - `afcp.h`: This is the header file for Oracle Enterprise Scheduler.
   - `afstd.h` and `afstr.h`: These are Fusion Application header files.

2. Call `afpend` in the business logic function.
3. In the `main` function, call `afprcp`, passing to it a pointer to the business logic function.

   The business logic function is called by `afprcp`, taking the arguments `argc`, `argv`, and `reqinfo`.

4. Save the executable job file to the `$APPLICATIONS_BASE/$APPLBIN` directory.
5. Configure the spawned job environment, as described in Section 9.5, "Configuring a Spawned Job Environment".

   Be sure to set both the `TOP` and `APPLBIN` variables for your application in the `environment.properties` file.

9.10.3 Scheduled C Job API

Several C functions are available for use in developing Fusion applications, while several others are not. Table 9–2 and Table 9–3 list the available and unavailable functions.

Table 9–2 C Functions Available for Developing Fusion Applications

Function	Description
`afprcp`	Run C program. The recommended API for writing a C program. The main .oc file should call this function to run the program logic. It initializes the context and calls the program. `int afprcp (uword argc, text **argv, afsqlopt *options, afpfcn *function);`
`afpend`	End C program. All programs must call this to signal the completion of the program. The program should pass completion status, and message if necessary. Indicate completion status with the following constants: ■ `FDP_SUCCESS`: Success ■ `FDP_WARNING`: Warning ■ `FDP_ERROR`: System Error ■ `FDP_BIZERR`: Business Error `boolean afpend (text *outcome, dvoid *handle, text *compmesg);`
`fdpfrs`	Find request status. For a given request, retrieve the status. The following are possible request states: ■ `ESS_WAIT_STATE` ■ `ESS_READY_STATE` ■ `ESS_RUNNING_STATE` ■ `ESS_COMPLETED_STATE` ■ `ESS_BLOCKED_STATE` ■ `ESS_HOLD_STATE` ■ `ESS_CANCELLING_STATE` ■ `ESS_EXPIRED_STATE` ■ `ESS_CANCELLED_STATE` ■ `ESS_ERROR_STATE` ■ `ESS_WARNING_STATE` ■ `ESS_SUCCEEDED_STATE` ■ `ESS_PAUSED_STATE` ■ `ESS_PENDING_VALID_STATE` ■ `ESS_VALID_FAILED_STATE` ■ `ESS_SCHEDULE_ENDED_STATE` ■ `ESS_FINISHED_STATE` ■ `ESS_ERROR_AUTO_RETRY_STATE` ■ `ESS_MANUAL_RECOVERY_STATE` `afreqstate fdpfrs (text *request_id, text *errbuf);`

Table 9–2 (Cont.) C Functions Available for Developing Fusion Applications

Function	Description
fdpgret	Get the error type of a specific job request ID. The following are possible error types: ■ ESS_UNDEFINED_ERROR_TYPE ■ ESS_SYSTEM_ERROR_TYPE ■ ESS_BUSINESS_ERROR_TYPE ■ ESS_TIMEOUT_ERROR_TYPE ■ ESS_MIXED_NON_BUSINESS_ERROR_TYPE ■ ESS_MIXED_BUSINESS_ERROR_TYPE `afreqstate fdpgret (text *request_id, text *status, text *errbuf);`
fdpgrs	Get request status. For a given request, retrieve the current status and completion text. `afreqstate fdpgrs (text *request_id, text *status, text *errbuf);`
fdplck	Lock table. Locks the desired table with the specified lock mode and NOWAIT.
fdpscp	Legacy API for concurrent programs. All new concurrent programs should use afprcp. `boolean fdpscp (sword *argc, text **argv[], text args_type, text *errbuf);`
fdpwrt	Routines for creating log/output files and writing to files. These are routines concurrent programs should use for writing to all log and output files.

Table 9–3 C Functions Not Available for Developing Fusion Applications

Function	Description
fdpgoi	Get Oracle data group.
fdpgpn	Get program name.
fdpgrc	Get request count.
fdpimp	Run the import utility.
fdpldr	Run SQL*Loader.
fdpperl	Run Perl concurrent program.
fdprep	Run report.
fdprpt	Run Sql*Rpt program.
fdprsg	Submit concurrent program. Use the afpsub routines instead.
fdpscr	Get resource security group.
fdpsql	Run SQL*Plus concurrent program.
fdpstp	Run stored procedure.

9.10.4 How to Test a C Scheduled Job

When developing a C job, it is possible to test the job by running it from a command line interface.

Running a C job from the command line involves the following main steps:

- Invoking the job
- Obtaining a database connection and setting the runtime context by passing special arguments.
- Passing any program-specific parameters at the command line.

To run a C job from the command line:

- Use the syntax shown in Example 9–18 to run a C job from the command line for testing purposes.

Example 9–18 Syntax for Running a C Job from the Command Line

```
%program <heavyweight user connection string> <lightweight username> <flag> <job
parameters> ...
```

where

`<heavyweight user connection string>` is the `username/password@TWO_TASK` pair used to connect to the database

`<lightweight username>` is the name of the lightweight user submitting the job. This value is used to set the user context in the database connection.

`<flag>` must be set to 'L' for lightweight user.

An example illustrating running a C job from the command line is shown in Example 9–19.

Example 9–19 Running a C Job from the Command Line for Testing Purposes

```
program username/password@my_db MYUSER L <parameter1> <parameter2> ....
```

9.10.5 What Happens When You Implement a C Scheduled Job

The sample C job shown in Example 9–20 uses `afprcp` to initialize and obtain a database connection. It uses both Pro*C and `afupi`.

Example 9–20 Using the C Runtime API

```
#ifndef AFSTD
#include <afstd.h>
#endif

#ifndef AFSTR
#include <afstr.h>
#endif

#ifndef AFCP
#include <afcp.h>
#endif

#ifndef SQLCA
#include <sqlca.h>
#endif

#ifndef AFUPI
#include <afupi.h>
#endif
```

```
#ifndef FDS
#include <fds.h>
#endif

boolean testupi()
{
  text *sqltext;
  text buffer[ERRLEN];
  text os_user[31];
  text session_user[31];
  text db_name[31];

  aucursor  *use_curs;
  word      errcode;

  os_user[0] = session_user[0] = db_name[0] = (text)'\0';

  sqltext = (text*) "SELECT sys_context('USERENV','DB_NAME',30), sys_context('US
ERENV','SESSION_USER',30), sys_context('USERENV','OS_USER',30) from dual";

  use_curs = NULLCURSOR;
  use_curs = afuopen (NULLHOST, NULLCURSOR, (dvoid *)
                      sqltext,
                      UPISTRING);
  if (use_curs == NULLCURSOR) {goto upierror;}

  afudefine(use_curs, 1, AFUSTRING, (dvoid *)db_name, 31);
  afudefine(use_curs, 2, AFUSTRING, (dvoid *)session_user, 31);
  afudefine(use_curs, 3, AFUSTRING, (dvoid *)os_user, 31);

  if (!afuexec (use_curs, (uword)1, (uword)1, CSTATHOLD|CSTATEXACT) ||
      (errcode = afuerror (NULLHOST, (text *) NULL, 0)) != ORA_NORMAL) {
    goto upierror;
  }

  DISCARD afurelease (use_curs);

  DISCARD sprintf((char *)buffer, "%s as %s@%s", os_user,
                  session_user, db_name);

  DISCARD fdpwrt(AFWRT_OUT | AFWRT_NEWLINE, buffer);


  return TRUE;

 upierror:
  if (use_curs != NULLCURSOR)
    DISCARD afurelease (use_curs);
  DISCARD fdpwrt(AFWRT_LOG | AFWRT_NEWLINE, "Error in testupi");
  return FALSE;
}

void testrpc()
{
  text buffer[256];


  EXEC SQL BEGIN DECLARE SECTION;

  VARCHAR os_user[31];
```

```
  VARCHAR session_user[31];
  VARCHAR db_name[31];

  EXEC SQL END DECLARE SECTION;

  buffer[0] = os_user.arr[0] = session_user.arr[0] = db_name.arr[0] = '\0';

  EXEC SQL SELECT sys_context('USERENV','DB_NAME',30),
    sys_context('USERENV','SESSION_USER',30),
    sys_context('USERENV','OS_USER',30)
    INTO :db_name, :session_user, :os_user
    from dual;

  nullterm(os_user);
  nullterm(session_user);
  nullterm(db_name);

  DISCARD sprintf((char *)buffer, "%s as %s@%s", os_user.arr,
                  session_user.arr, db_name.arr);

  DISCARD fdpwrt(AFWRT_OUT | AFWRT_NEWLINE, buffer);
}

sword cptest(argc, argv, reqinfo)
/* ARGSUSED */
sword argc;
text *argv[];
dvoid *reqinfo;
{
  ub2 i;
  text errbuf[ERRLEN+1];

 /* Write to the log file */
  DISCARD fdpwrt(AFWRT_LOG | AFWRT_NEWLINE, (text *)"Test Success");
 /* Write to the out file */
  DISCARD fdpwrt(AFWRT_OUT | AFWRT_NEWLINE, (text *)"Test Args:");
 /* Loop through argv and write to the out file. */
  for ( i=0; i<argc; i++)
    DISCARD fdpwrt(AFWRT_OUT | AFWRT_NEWLINE, argv[i]);
  /* Call the Fusion Applications function afpoget to return the value of a */
  /* profile option called SITENAME and write the results to the error buffer. */
  DISCARD afpoget((text *)"SITENAME", errbuf);
  /* Write the value to the output file. */
  DISCARD fdpwrt(AFWRT_OUT | AFWRT_NEWLINE, errbuf);
  /* Connect to the database and run a SELECT against the database. Creates a */
  /* string and writes the returned data to the output file. Uses prc APIs. */
  testrpc();
  /* Open a cursor for the SELECT statement, defines variables to collect data */
  /* upon running statement, and executes SELECt. Creates a string which it */
  /* writes to the output file. Uses afupi APIs. */
  testupi();
  /* Writes the string "Test Completed." to the output file. */
  DISCARD fdpwrt(AFWRT_OUT | AFWRT_NEWLINE, (text *)"Test Completed.");
  /* Call afpend to identify the exit status, which in this case is successful. */
  /* Other possible values are FDP_WARNING, FDP_ERROR and FDP_BIZERR. The
  /* reqinfo originally passed to cptest is passed here. Optionally, additional */
  /* text can be passed here, for example explaining the outcome of the exit */
  /* status. */
  return((sword)afpend(FDP_SUCCESS, reqinfo, (text *)NULL));
};
```

```
int main(/*_ int argc, text *argv[] _*/);
int main(argc, argv)
  int argc;
  text *argv[];
{

    /* Run cptest and return an exit value to Oracle ESS. */
    return(afprcp((uword)argc, (text **)argv,
    (afsqlopt *)NULL, (afpfcn *)cptest));
}
```

9.10.6 What Happens at Runtime: How a C Scheduled Job Is Implemented

When Oracle Enterprise Scheduler Service runs a C job, `afprcp()` runs first to initialize the context and obtain the database connection. The function afprcp() then calls the function containing the program logic. Oracle Enterprise Scheduler runs the job, and the result of the job is returned to Oracle Enterprise Scheduler. The Fusion Application can retrieve the result from Oracle Enterprise Scheduler and display it in the user interface.

Note: Wallet configuration is required for the client `ORACLE_HOME` to obtain the database connection. The operating system environment in which the job runs (including the location of the client `ORACLE_HOME`, which is also required) is set in the `environment.properties` file. The `environment.properties` file must be configured and placed in the `config/fmwconfig` directory under the domain.

You can add your own environment variables by creating an `env.custom.properties` file in the same directory. Variables you define in this file take precedence over those in the `environment.properties` file.

Similarly, you can set server-specific environment variables with `environment.properties` and `env.custom.properties` files in the server config directory.

9.11 Implementing a Host Script Scheduled Job

Arguments submitted for a host script job request are passed to the script at the command line. Host scripts may access the standard environment variables to get `REQUESTID`, `LOG_WORK_DIRECTORY`, `OUTPUT_WORK_DIRECTORY`, and so on. Script output is redirected to the request log file by default.

Use the following steps when implementing a host script job:

- Complete the steps for configuring a spawned job as described in Section 9.5, "Configuring a Spawned Job Environment".
- Create one script file each for Unix and Windows platforms. Name each script file the same as `executableName` parameter in the job definition. For example, if your `executableName` is "myscript", the script files would be called myscript.sh (on Unix platforms) and myscript.cmd (on Windows).
- Put host scripts in the `$APPLBIN` directory under the product TOP.
- The script should exit with one of the following exit codes (anything else is considered a SYSTEM ERROR):

- 0 for SUCCESS
- 2 for WARNING
- 3 for BUSINESS ERROR

9.12 Implementing a Java Scheduled Job

For more information about implementing Java Scheduler jobs, see Chapter 3, "Use Case Oracle Enterprise Scheduler Sample Application."

9.12.1 How to Define Metadata for a Scheduled Java Job

Create a job definition as described in Section 9.4, "Creating a Job Definition".

9.12.2 How to Use the Java Runtime API

For information about the Java runtime API, see the *Oracle Fusion Applications Java API Reference for Oracle Enterprise Scheduler Service*.

You can access the Oracle Fusion Middleware Extensions for Applications `Message` and `Profile` objects directly, using those APIs which handle the service accessing themselves.

9.12.3 How to Cancel a Scheduled Java Job

You can cancel a scheduled Java job by implementing the `Cancellable` interface.

The `Cancellable` implementation in Example 9–21 checks as logic progresses to see if the job has been canceled. If it has, the code cleans up after itself before exiting.

Example 9–21 Handling a Job Cancellation Request

```
import oracle.as.scheduler.Cancellable;
import oracle.as.scheduler.Executable;
import oracle.as.scheduler.ExecutionCancelledException;
import oracle.as.scheduler.ExecutionErrorException;
import oracle.as.scheduler.ExecutionPausedException;
import oracle.as.scheduler.ExecutionWarningException;
import oracle.as.scheduler.RequestExecutionContext;
import oracle.as.scheduler.RequestParameters;

public class MyExecutable
    implements Executable, Cancellable
{
    private volatile boolean m_cancel = false;

    public void execute( RequestExecutionContext reqCtx,
                         RequestParameters reqParams )
        throws ExecutionErrorException, ExecutionWarningException,
               ExecutionPausedException, ExecutionCancelledException
    {
        // Do some work and check if this request has been canceled.
        // ... work ...
        checkCancel(reqCtx);

        // Do more work and check if this request has been canceled.
        // ... work ...
        checkCancel(reqCtx);
        // Finish work.
```

```
            // ... work ...
        }

        // Set flag that the app logic should check periodically to
        // determine if this request has been canceled.
        public void cancel()
        {
            m_cancel = true;
        }

        // Check if request has been canceled. If not, do nothing.
        // Otherwise, do any clean up work that may be needed for
        // this request and end by throwing an ExecutionCancelledException.
        private void checkCancel(RequestExecutionContext reqCtx )
            throws ExecutionCancelledException
        {
            if (m_cancel)
            {
                // Do work any clean up work that may be needed
                // prior to ending this executable.
                // ... clean up work ...
                String msg = "Request " + reqCtx.getRequestId() +
                             " was cancelled.";
                throw new ExecutionCancelledException(msg);
            }
        }
    }
```

9.12.4 What Happens at Runtime: How a Java Scheduled Job Is Implemented

Oracle Enterprise Scheduler Service initializes the context of the job. The Fusion application calls Oracle Enterprise Scheduler Service using the provided APIs. Oracle Enterprise Scheduler runs the job, and a result of success or failure is returned to Oracle Enterprise Scheduler. The Fusion Application can retrieve the result from Oracle Enterprise Scheduler and display it in the user interface.

9.13 Elevating Access Privileges for a Scheduled Job

Oracle Enterprise Scheduler executes jobs in the user context of the job submitter at the scheduled time. Some scheduled jobs require access privileges that are different from those of the submitting user. However, information regarding the submitter of the scheduled job must be retrievable for auditing purposes.

In Oracle Enterprise Scheduler, it is prohibited to run a job in the context of a user other than the submitting user with `runAs`. Doing so would be considered a security breach. Using an application identity enables running a job with different access privileges from those allotted to the submitting user.

Application identity is a SOA and JPS concept that addresses the requirement for escalated privileges in completing an action. The application installer creates an application identity in Oracle Identity Management Repository.

For more information, see the following chapters in the *Oracle Fusion Applications Developer's Guide*:

- "Implementing Oracle Fusion Data Security"
- "Implementing Application User Sessions"
- "Implementing Function Security"

9.13.1 How to Elevate Access Privileges for a Scheduled Job

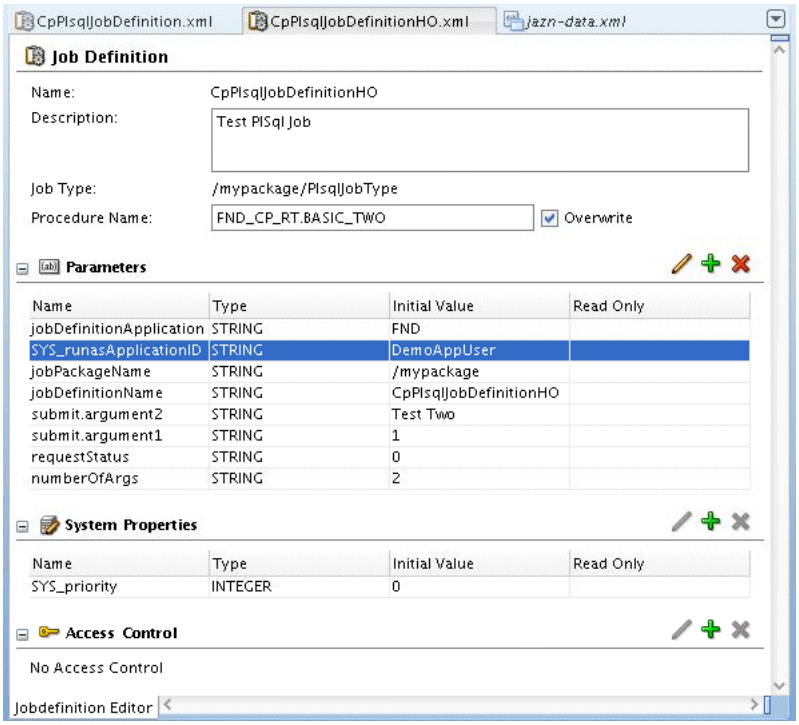
The Oracle Enterprise Scheduler job system property `SYS_runasApplicationID` enables elevating access privileges for completing a scheduled job.

To elevate access privileges for a scheduled job:

1. Create a job definition, as described in Section 9.4, "Creating a Job Definition."
2. Under the Parameters section, add a parameter called `SYS_runasApplicationID`.
3. In the text field for the `SYS_runasApplicationID`, enter the application ID under which you want to run the job, as shown in Figure 9–1.

   Make sure the input string is a valid `ApplicationID` that exists when the job executes.

Figure 9–1 Defining the runAs User for the Job



You can retrieve the executing user by running either of the methods shown in Example 9–22 and Example 9–23.

Example 9–22 Retrieving the Executing User with getRunAsUser()

```
requestDetail.getRunAsUser()
```

Example 9–23 Retrieving the Executing User with getRequestParameter()

```
String sysPropUserName =
        (String) runtime.getRequestParameter(h, reqid, SystemProperty.USER_NAME);
```

Given a request ID, you can retrieve the submitting and executing users of a job request.

To retrieve the submitting and executing users of a job request in Oracle Enterprise Scheduler RuntimeService EJB:

- Example 9–24 shows a code snippet for retrieving the submitting and executing users of a job request using the Oracle Enterprise Scheduler `RuntimeService` EJB.

Example 9–24 Retrieving the Submitting and Executing Users of a Job Request Using the RuntimeService EJB

```
// Lookup runtimeService

RequestDetail requestDetail = runtimeService.getRequestDetail(h, reqid);
String runAsUser = requestDetail.getRunAsUser();
String submitter = requestDetail.getSubmitter();
```

To retrieve the submitting and executing users of a job request from within an Oracle Fusion application:

- Example 9–25 shows a code snippet for retrieving the submitting and executing users of a job request from within an Oracle Fusion application.

Example 9–25 Retrieving the Submitting and Executing Users of a Job Request from an Oracle Fusion Application

```
import oracle.apps.fnd.applcore.common.ApplSessionUtil;
// The elevated privilege user name.
ApplSessionUtil.getUserName()
// The submitting user.
ApplSessionUtil.getHistoryOverrideUserName()
```

9.13.2 How Access Privileges Are Elevated for a Scheduled Job

When a job request schedule executes, Oracle Enterprise Scheduler:

1. Validates the submitter's execution privileges on the job metadata.
2. Retrieves the application identity information from the job metadata. If the job metadata does not specify an application identity for the job, Oracle Enterprise Scheduler executes the job in the context of the job submitter.

   - **Java job**: An FND session is established as the user with elevated privileges.

     The executing user is taken from the current subject as viewed from the job logic.

     Note: Oracle Enterprise Scheduler does not directly support invoking a web service or composite. If your job logic invokes a web service or composite, you must write the client code logic in your job, establish a connection and propagate the job submitter information as a payload for auditing purposes. For an asynchronous web service call, the job must wait for a response.

   - **Spawned C job**: An application user session is established as the executing user. The submitter information is an attribute of the application user session.

     The spawned job executes as the operating system user who starts Oracle WebLogic Server.

- **PL/SQL job**: An FND session is established as the executing user. The submitter information is attribute of the FND session.

  The job runs in the context of the FND session in the RDBMS job scheduler.

3. Executes the job logic.

9.13.3 What Happens When Access Privileges Are Elevated for a Scheduled Job

Oracle Enterprise Scheduler validates the user's execution privileges on the job metadata. If so, the user context is captured and stored in the Oracle Enterprise Scheduler database as the submitting user, and the request is placed in the queue.

9.14 Creating an Oracle ADF User Interface for Submitting Job Requests

When implemented as part of an Oracle Fusion application, the Oracle ADF user interface enables end users to submit job requests.

9.14.1 How to Create an Oracle ADF User Interface for Submitting Job Requests

The Oracle ADF UI enables end users to submit job requests. End users can enter complex data types for the arguments of descriptive and key flexfields. The Parameters tab in the Oracle ADF UI interface allows end users to enter parameters to be used when submitting the job request.

Flexfields display in a separate task flow region. This region is a child task flow of the parent task flow displayed in the Parameters tab.

Note: Make sure to define customization layers and authorize runtime customizations to the `adf-config.xml` file as described in the chapter "Creating Customizable Applications" in *Oracle Fusion Applications Developer's Guide*.

To create a user interface for submitting job requests:

1. Create a new Oracle Fusion web application by clicking **New Application** in the Application Navigator and selecting **Fusion Web Application (ADF)** from the Application Templates drop-down list.

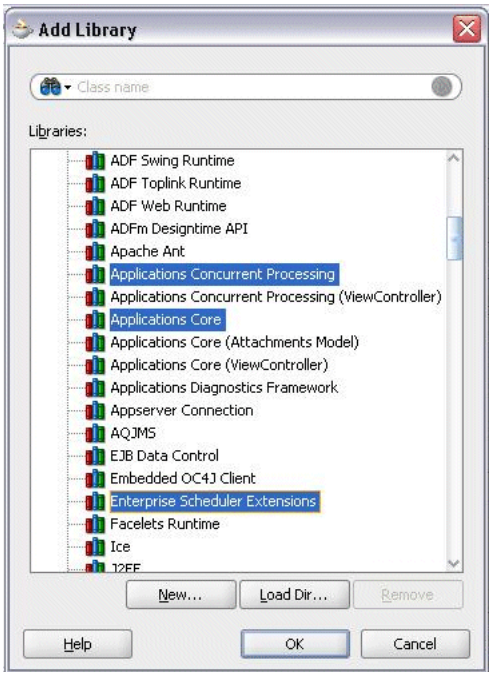
   Model and ViewController projects are created within the application.

2. Right-click the Model project and select **Project Properties** > **Libraries and Classpath** > **Add Library**.

3. From the list, select the following libraries, as shown in Figure 9–2:

   - Applications Core
   - Applications Concurrent Processing
   - Enterprise Scheduler Extensions

Figure 9–2 Adding the Libraries to the Model Project

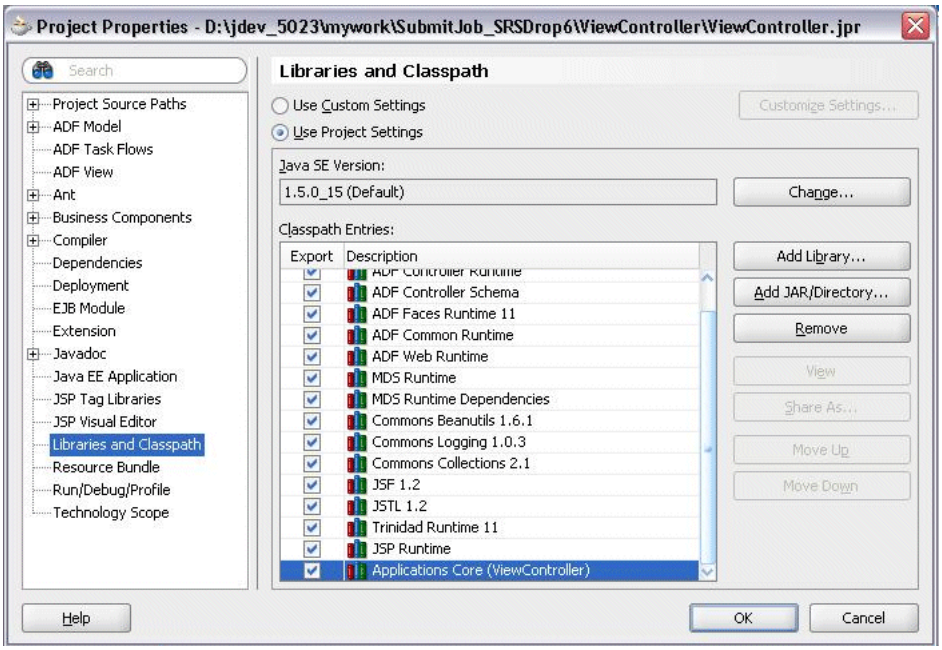


Click **OK** to close the window and add the libraries.

4. Right-click the View Controller project and select **Project Properties** > **Libraries and Classpath** > **Add Library**.

   Add the library Applications Core (ViewController), as shown in Figure 9–3.

Figure 9–3 Adding the Library to the View Controller Project



5. In the **Project Properties** dialog, in the left pane, click **Business Components**.

6. The Initialize Business Components Project window displays. Click the Edit icon to create a database connection for the project.

   Fill in the database connection details as follows:

   - **Connection Exists in**: Application Resources
   - **Connection Type**: Oracle (JDBC)
   - **Username/Password**: Fill in the relevant username and password for the database.
   - **Driver**: thin
   - **Host Name**: Enter the host name of the database server.
   - **JDBC port**: Enter the port number of the database.
   - **SID**: The unique Oracle system ID for the database.

   Click **OK**.

7. In the file `weblogic.xml`, import `oracle.applcp.view`.
8. In the file `weblogic-application.xml`, import the following libraries:

   - `oracle.applcore.attachments (for ESS-UCM)`
   - `oracle.applcp.model`
   - `oracle.applcp.runtime`
   - `oracle.ess`
   - `oracle.sdp.client (for notification)`
   - `oracle.ucm.ridc.app-lib (for ESS-UCM)`
   - `oracle.webcenter.framework (for ESS-UCM)`
   - `oracle.xdo.runtime`
   - `oracle.xdo.service.client`
   - `oracle.xdo.webapp`

   The libraries `oracle.applcp.model` and `oracle.applcp.view` are deployed as part of the installation while running the `config.sh` wizard.

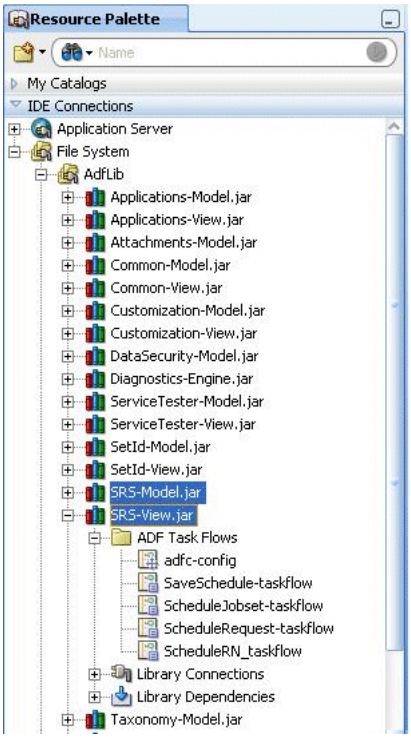
9. Create a new JSPX page for the ViewController project by right-clicking ViewController and selecting **New** > **Web Tier** >**JSF** > **JSF JSP Page**.
10. Create a new File System connection. In the Resource Palette, right-click **File System**, select **New File System Connection**, and do the following:

    a. Provide a connection name and directory path for the Oracle ADF Library files (`<jdev_install>/jdev/oaext/adflib`).

    b. Click **Test Connection** and click **OK** once the connection is successful.

11. Expand the contents of the `SRS-View.jar` file to display the list of available task flows that can be used in the application, as shown in Figure 9–4.

Figure 9–4 Displaying the List of Available Task Flows



12. To include the job request submission page in the application, select the `ScheduleRequest-taskflow` from the Resource Palette and drop it onto the JSF page in the area where you want to create a call to the taskflow. Create the taskflow call as a link or button.

For example, to invoke the job request submission page from within a dialog box in the application, do the following:

a. From the Component Palette, drag and drop a Link onto the form in the JSPX page.

b. In the Property Inspector, configure the behavior of the link to `showpopup`.

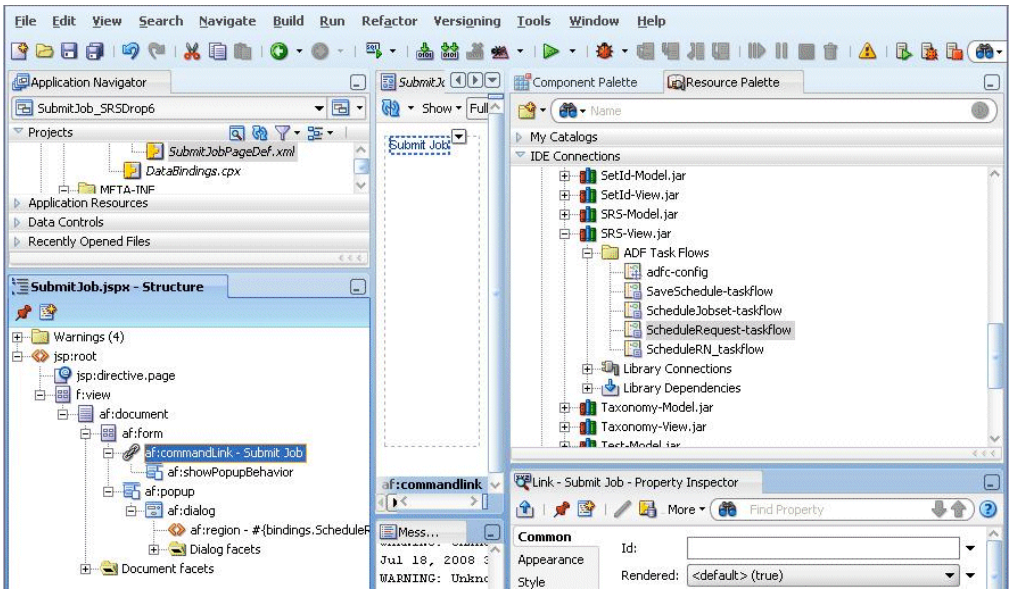
c. From the Component Palette, drag and drop a Popup component with a dialog component onto the form.

d. To enable submitting a job request, drag and drop ScheduleRequest-taskflow onto the dialog component as a dynamic region.

To enable submitting a job set request, drag and drop `ScheduleJobset-taskflow` onto the dialog component.

Figure 9–5 displays the task flows in the Resource Palette.

Figure 9–5 Including the Job Request Submission Page in the Application



e. From the context menu, select **Create a Dynamic Region**.

13. When prompted, add the required library to the ViewController project by clicking **Add Library**. Save the JSF page.

14. Edit the task flow binding. Define the following parameters for the task flow, as shown in Figure 9–6.

a. `jobdefinitionname`: Enter the name of the job definition to be submitted. This is not the name that displays. This is the job definition defined in Section 9.4, "Creating a Job Definition". Required.

b. `jobdefinitionpackagename`: Enter the package name under which the job definition metadata is stored. This should be the namespace path appended to the package name, for example `/oracle/ess/Scheduler`. The namespace path typically begins with a forward slash ("/"), but should have no forward slash at the end. Required.

c. `centralui`: When setting this parameter to `true`, then the task flow UI does not display the header section containing the name, description and basic Oracle BI Publisher actions (such as e-mail, print and notify). This parameter must be a boolean value. Optional.

d. `pageTitle`: When passed, the task flow will render this passed String value as the page title. The `pageTitle` value is currently configured to be truncated at 30 characters. Optional.

e. `requireRootOutcome`: If `true` is passed as the value, then the task flow will generate root-outcome when the user clicks the Submit or Cancel buttons. By default, the task flow generates parent-outcome. Optional.

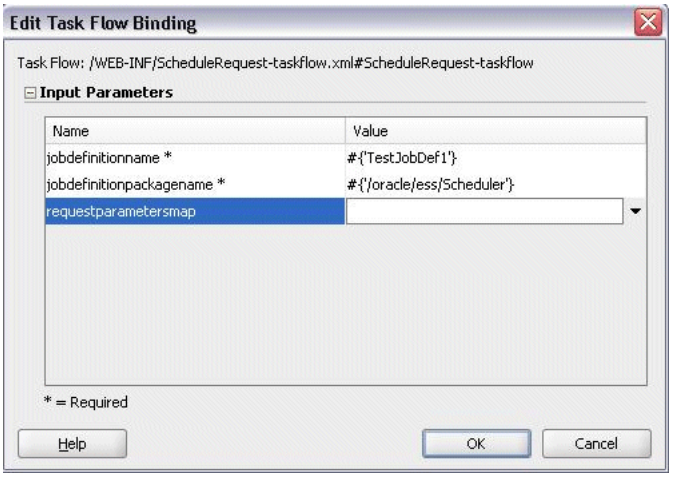
f. `requestparametersmap`: Enter the name of the map object variable that contains the parameters required for the job request submission. If this parameter is filled in, the Parameters tab in the request scheduling submission page will not prompt end users to enter parameters for executing the request. The map can be passed to the task flow as a parameter. Typically, this parameter takes the data type `java.util.Map` in which keys are parameter names and values are parameter values. For example, if you will be using a

`paramsMap` object in the `pageFlowScope`, you might enter a `requestparametersmap` value of `#{pageFlowScope.paramsMap}`. Optional.

In the page that holds the SRS task flow region, set the following property for the popup that launches the SRS window: `contentDelivery = immediate`.

In the page definition file of the page that contains the task flow region, set the following property for the task flow: **Pagedef** > **executables** > **taskflow** > `Refresh=IfNeeded`.

Figure 9–6 Defining Parameters for the Task Flow



15. If you are using a map to pass parameters to the taskflow (`requestparametersmap`), create a new taskflow parameter, such as the `paramsMap` object in the `pageFlowScope` of a pageflow.

    These values can be accessed in the job executable, for example from the `RequestParameters` object in the case of a Java job. Example 9–26 illustrates passing the values stored in the `RequestParameters` object to a Java job. This code is used in the class that implements the `oracle.as.scheduler.Executable` interface.

Example 9–26 Passing Values in a Map Object to a Java Job

```
public void execute(RequestExecutionContext ctx,RequestParameters props)
    throws ExecutionErrorException, ExecutionWarningException,
        ExecutionCancelledException,ExecutionPausedException
{
    String pageTitle = (String) props.getValue("pageTitle");
    // Retrieve other parameters.
    // ...
}
```

Note: When using a `requestparametersmap`, make sure to set the following properties for the popup within which the task flow is launched.

- Set Content Delivery to **Immediate**.
- In the page definition XML file for the page that contains the region, select **PageDef** > **Executables** > **taskflow** > **set Refresh = ifNeeded**.

---

16. If the job is defined with properties that must be filled in by end users, the user interface allows end users to fill in these properties prior to submitting the job request. For example, if the job requires a start and end time, end users can fill in the desired start and end times in the space provided by the user interface.

    The properties that are filled in by end users are associated with a view object, which in turn is associated with the job definition itself. When the job runs, Oracle Enterprise Scheduler Service accesses the view object to retrieve the values of the properties.

    If using a view object to pass parameters to the job definition, do the following:

    a. Create a view object called TestVO using a query such as the one shown in Example 9–27.

Example 9–27 Creating a View Object Using a Query

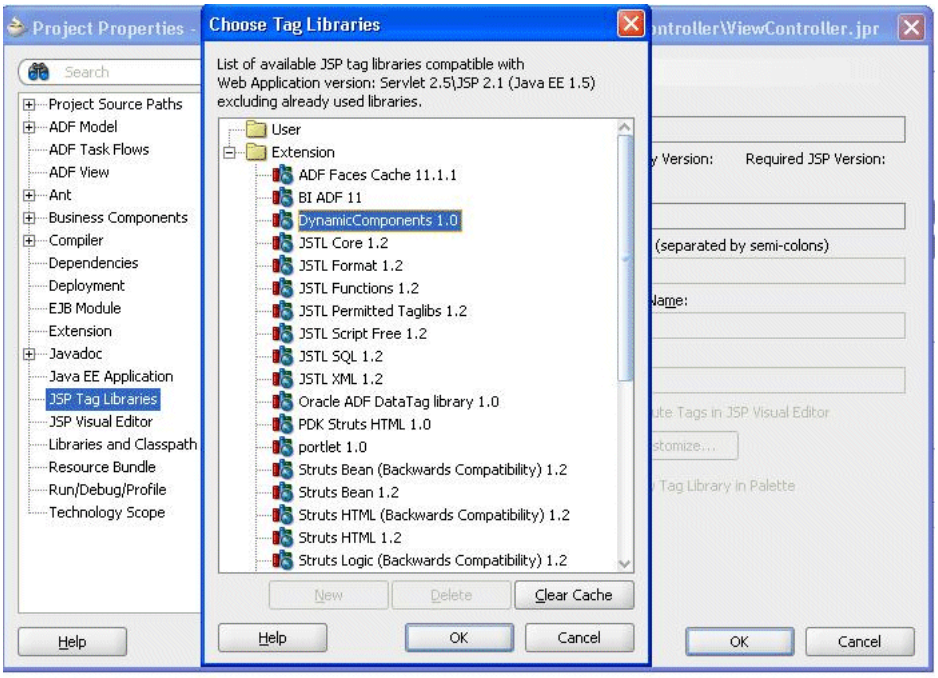
```
select null as Attribute1, null as Attribute2 from dual"
```

    b. Specify control UI hints, for example set the display label for Attribute1 to **Run Mode** and for Attribute2 to **Duration**.

       As a result, the parameters tab in the job request submission UI renders with the input fields Run Mode and Duration.

    c. In order to render the Parameters tab in the job request submission UI, add the DynamicComponents 1.0 library as follows. Right-click ViewController and select **Project Properties** > **JSP Tag Libraries** > **Add**. In the Choose Tag Libraries window, select the library **DynamicComponents 1.0** and click **OK**. Figure 9–7 displays the Choose Tag Libraries window.

Figure 9–7 Adding the Library DynamicComponents 1.0



17. In the JSF application you created, create another project called Scheduler. Select **File > New**, and choose **General** > **Empty Project**. This project will be used to create Enterprise Scheduler Service metadata and job implementations.
18. In the Scheduler project, add the Enterprise Scheduler Extensions library to the classpath. Right-click the Scheduler project and select **Project Properties** > **Libraries and Classpath** > **Add Library** > Enterprise Scheduler Extensions.
19. Deploy the libraries `oracle.xdo.runtime` and `oracle.xdo.webapp` to the Oracle Enterprise Scheduler UI managed server. These libraries are located in the directory `$MW_HOME/jdeveloper/xdo`, where `MW_HOME` is the Oracle Fusion Middleware home directory.
20. Deploy the application.

9.14.2 How to Add a Custom Task Flow to an Oracle ADF User Interface for Submitting Job Requests

You can add a custom task flow to an Oracle ADF user interface used to submit job requests at run time.

To add a custom task flow to an Oracle ADF user interface for submitting job requests:

1. Create a task flow and bind it to your Oracle ADF user interface for submitting a job request created in Section 9.14.1, "How to Create an Oracle ADF User Interface for Submitting Job Requests."

   For more information about creating task flows and binding them to an Oracle ADF user interface, see the following chapters in *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*:

   - "Getting Started with ADF Task Flows"
   - "Working with Task Flow Activities"

- "Using ADF Task Flows as Regions"

2. Create an ADF Business Components view object for each UI field. Name the view objects that are bound to UI fields `ParameterVO1`, `ParameterVO2`, and so on.

   Name the attributes of the view objects as follows: `ATTRIBUTE1`, `ATTRIBUTE2`, and so on.

   For more information about creating an ADF Business Components view object, see the chapters "Defining SQL Queries Using View Objects" and "Advanced View Object Techniques" in *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

3. Include the view objects in the relevant application module. Even if their names are different, make sure the view object instance names are `ParameterVO1`, `ParameterVO2`, `ParameterVO3`, and so on.

4. In the job definition, make sure to define the properties `CustomDataControl` and `ParameterTaskflow` For more information, see Section 9.4.1, "How to Create a Job Definition."

   For more information about passing parameters to the Oracle ADF task flow, see the chapter "Using Parameters in Task Flows" in Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework.

5. Optionally, include the method `preSubmit()` in the application module. Oracle Enterprise Scheduler invokes this method before retrieving the parameter values for the submission request.

   Your implementation of the `preSubmit()` method (which returns a boolean value) could include validation code in the custom task flow. If the validation fails, your code can throw an exception with the proper internationalized error message.

   If this validation fails while submitting the request, the error message is displayed to the user and the submission doesn't go through.

9.14.3 How to Enable Support for Context-Sensitive Parameters in an Oracle ADF User Interface for Submitting Job Requests

After integrating your application with the Oracle ADF UI for submitting job requests, enable context-sensitive parameter support in the UI.

The request submission UI will render the context-sensitive parameters first so that the end user will specify the context-sensitive parameter values. Context is set in the database based on these parameters. After setting the context, it renders the rest of the parameters based on context set at database layer. When the job runs, the actual business logic will run after setting the context based on the context-sensitive parameter values inside the database.

Follow this procedure to enable context-sensitive parameter support in the UI.

To enable support for context sensitive parameters in an Oracle ADF user interface for submitting job requests:

1. Follow the instructions described in Section 9.14.1.

2. Create a native ADF Business Components view object with attributes `CTXATTRIBUTE1`, `CTXATTRIBUTE2`, and so on, with a maximum of 100 attributes.

   For example, create a view object with the query `Select null as CTXATTRIBUTE1, CTXATTRIBUTE2, CTXATTRIBUTE3 from dual`. Include required UI hints such as display label, tool tip, and so on.

3. Create a PL/SQL procedure or function in order to set the context.
4. Specify the parameters shown in Example 9–28 and Example 9–29 in the job definition metadata.
   - `contextParametersVO`: Enter the fully qualified name of the view object that holds the context sensitive parameters.

Example 9–28 contextParametersVO

```
<parameter name="contextParametersVO" data-type="string">_
oracle.apps.mypkg.TestCtxVO</parameter>_
```

   - `setContextAPI`: PL/SQL API to set the context, along with the package name. The `_myPkg1.mySetCtx` procedure receives arguments based on attributes in the `contextParametersVO`.

Example 9–29 setContextAPI

```
<parameter name="setContextAPI"  data-type="string">_myPkg1.mySetCtx</parameter>_
```

9.14.4 How to Save and Schedule a Job Request Using an Oracle ADF UI

Saving and scheduling a job request using an Oracle ADF UI involves using the Enterprise Scheduler Extensions library in conjunction with a JSF application that includes a task flow in which a job is scheduled and saved.

To schedule a job request using an Oracle ADF UI:

1. Follow the instructions in Section 9.14.1, "How to Create an Oracle ADF User Interface for Submitting Job Requests" up to step 9.

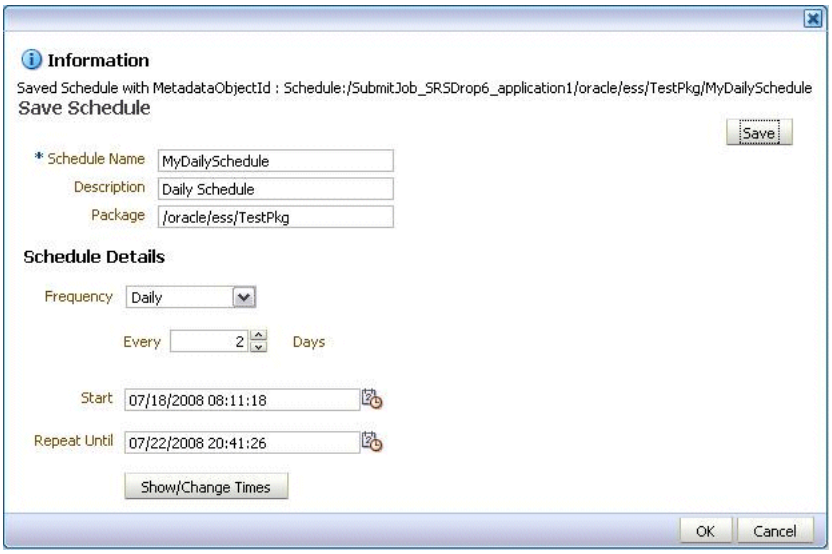
> **Note:** If the custom parameters task flow has no transactions of its own, it must set the data-control-scope to "isolated". This ensures that multiple ParameterVOs using the same application module get their independent application module instance.

2. Drag and drop `SaveSchedule-taskflow` onto the dialog. No input parameters are required.
3. When prompted, add the required library to the ViewController project by clicking **Add Library**. Save the JSF page.
4. In the JSF application you created, create another project called Scheduler. Select **File > New**, and choose **General** > **Empty Project**. This project will be used to create Enterprise Scheduler Service metadata and job implementations.
5. In the Scheduler project, add the Enterprise Scheduler Extensions library to the classpath. Right-click the Scheduler project and select **Project Properties** > **Libraries and Classpath** > **Add Library** > Enterprise Scheduler Extensions.
6. Deploy the application as described in the *Oracle Enterprise Scheduler Developer's Guide*.
7. Launch the application using the following URL:

   ```
   http://<machine>:<http-port>/<context-root>/faces/<page>
   ```

8. Enter a schedule name, description and package name with the namespace appended, as shown in Figure 9–8.

Figure 9–8 Saving a Job Submission Schedule



9. Save the schedule.

   A message displays indicating the metadata object ID of the saved schedule. This ID can be used for further job or job set request submissions

9.14.5 How to Submit a Job Using a Saved Schedule in an Oracle ADF UI

Submitting a saved job request schedule using an Oracle ADF UI involves using the Enterprise Scheduler Extensions library in conjunction with a JSF application that includes a task flow in which a saved job schedule can be submitted.

To submit a job using a saved schedule in an Oracle ADF UI:

1. Follow the instructions in Section 9.14.1, "How to Create an Oracle ADF User Interface for Submitting Job Requests".
2. Deploy the application. Launch the page using the following URL:

   ```
   http://<machine>:<http-port>/<context-root>/faces/<page>
   ```

3. Click the Schedule tab. In the Run option field, select the **Use a Schedule** radio button.
4. From the Frequency drop-down list, select **Use a Saved Schedule**.
5. Enter the namespace and package names for the schedule along with the name of the schedule.
6. To view the list of scheduled jobs, click **Get Details**. Click **Submit** to submit the saved job request.

9.14.6 How to Notify Users or Groups of the Status of Executed Jobs

The Oracle ADF user interface for submitting job requests provides the ability to notify users of the status of submitted jobs (via the Notification tab of the user interface). For example, users can request a notification to be sent to the originator of the job request.

A notification includes two components: the user or group to whom the notification is to be delivered, and the completion status of the job that triggers the notification. For example, notifications can be sent upon the successful completion of a job, or when a job completes in an error or warning state.

To notify users or groups of the status of executed jobs:

1. Configure Oracle User Messaging Service. For more information, see the chapter "Configuring Oracle User Messaging Service" in *Oracle Fusion Middleware Administrator's Guide for Oracle SOA Suite and Oracle Business Process Management Suite*.
2. Deploy the drivers required for Oracle User Messaging Service. You can do so using Oracle WebLogic Server Scripting Tool. For more information, see the chapter "Managing Oracle User Messaging Service" in Oracle Fusion Middleware Administrator's Guide for Oracle SOA Suite and Oracle Business Process Management Suite.
3. In the Oracle Enterprise Scheduler `connections.xml` file, specify the URL of the notification service. An example is shown in Example 9–30. While you cannot edit this file, you can browse Oracle ADF connection information using MBeans. For more information on configuring application properties, see the chapter "Monitoring and Configuring ADF Applications" in *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

Example 9–30 Specify the URL of the Notification Service

```
<References>
-
    <Reference name="EssConnection1"
        className="oracle.as.scheduler.config.ca.EssConnection">
        <Factory className="oracle.as.scheduler.config.ca.EssConnectionFactory"/>
-
        <RefAddresses>
-
            <StringRefAddr addrType="NotificationServiceURL">
                <Contents>http://localhost:8001</Contents>
            </StringRefAddr>
-
            <StringRefAddr addrType="RequestFileDirectory">
                <Contents>/tmp/ess/requestFileDirectory</Contents>
            </StringRefAddr>
-
            <StringRefAddr addrType="SAMLTokenPolicyURI">
                <Contents/>
            </StringRefAddr>
-
            <StringRefAddr addrType="FilePersistenceMode">
                <Contents>file</Contents>
            </StringRefAddr>
        </RefAddresses>
    </Reference>
</References>
```

4. Follow the instructions described in Section 9.14.1, "How to Create an Oracle ADF User Interface for Submitting Job Requests."
5. Create a native ADF Business Components view object with attributes representing the following properties:

- **Recipient Type**: Specify whether the notification recipient is a user or a group of users. This should be defined as a radio button. Values are `User` or `Group`.
- **Recipient ID**: Specify the User- or GroupID, depending on the recipient type. Create an LOV that provides a list of users or groups for the current submitting user. This LOV is dependent on the selected recipient type.
- **On Success**: Notify the recipient upon successful completion of the job.
- **On Warning**: Notify the recipient in the event of a job that ends with a warning.
- **On Error**: Notify the recipient in the event that a job completes in an error state.

Note: If using the post-processing action infrastructure to display the notification view object, it is not necessary to define status options in the view object (On Success, On Warning, On Error). Status data collection is built into the post-processing action infrastructure.

---

6. Launch the application using the following URL:

```
http://<machine>:<http-port>/<context-root>/faces/<page>
```

9.14.7 What Happens When You Create an Oracle ADF User Interface for Submitting Job Requests

The Oracle ADF interface is integrated with the Fusion application, and the application is tested and deployed. End users access the Oracle ADF user interface, fill in optional job properties, and click a button to submit the job request.

9.14.8 What Happens at Runtime: How an Oracle ADF User Interface for Submitting Job Requests Is Created

The application receives the submitted job request and calls Oracle Enterprise Scheduler Service to run the job. The Fusion application accesses the values of the properties entered by end users through the view object in which these properties were defined at design time. The job returns a result of success or failure, and the result passes from the Fusion application to Oracle Enterprise Scheduler.

Custom Task Flow

A job that includes properties to be filled in by end users through an Oracle ADF user interface at runtime includes ADF Business Components view objects with validation and the parameters to be filled in by end users. These parameters are submitted at runtime in the order in which they have been defined, meaning the first custom parameter to be defined is submitted first. The custom parameters must be named as follows:

`ParameterVO1.ATTRIBUTE1`, `ParameterVO1.ATTRIBUTE2`, `ParameterVO2.ATTRIBUTE1`, `ParameterVO3.ATTRIBUTE1`, and so on.

If the job definition includes `ContextParametersVO`, `ParameterTaskflow` and `parametersVO`, these properties render in that order at run time.

Context-Sensitive Parameters

When launching the SRS UI to submit a job or job set request with context-sensitive parameters, `contextParametersVO` initially renders in the Parameters tab of the Oracle ADF user interface.

The end-user can then enter values for the context-sensitive parameters. Clicking **Next** invokes `setConextAPI` by passing the context parameters. The context is set at the database level and the remaining `parametersVO` job parameters are rendered.

When the context-sensitive parameters are modified, end-users must click **Next** in order to set the context with the new values.

Notifications

When the final status of the job is determined, Oracle Enterprise Scheduler delivers the notifications to the relevant users or groups using the User Messaging Service. Groups receive notifications via e-mailed, whereas users receive notifications based on their messages preferences.

The notification view object defined at design time populates the input box in the submission request user interface at run time.

9.15 Submitting Job Requests Using the Request Submission API

You can submit, cancel and otherwise manage job requests using the request submission API.

For information about using the request submission API, see Section 13, "Using the Runtime Service."

9.16 Defining Oracle Business Intelligence Publisher Post-Processing Actions for a Scheduled Job

Oracle Business Intelligence Publisher enables generating reports from a variety of data sources, such as Oracle Database, web services, RSS feeds, files, and so on. BI Publisher provides a number of delivery options for generated reports, including print, fax, and e-mail.

In order to create an Oracle BI Publisher report, an Oracle BI Publisher report definition is required. Oracle BI Publisher report definitions consist of a data model that specifies the type of data source (database, web service, and so on) and a template for output formatting.

With report definitions in place, options for reporting are available to end users in the Output tab of the Oracle ADF user interface. The Output tab provides options through which an end user can define templates for reports. They can specify layout templates, document formats (such as PDF, RTF, and more), report destinations (email addresses, fax numbers, or printer addresses), and so on. When the user submits a request, this information is stored in the Oracle Enterprise Scheduler schema. The post-processor then invokes the Oracle BI Publisher service and passes the saved data to it.

Extensions to Oracle Enterprise Scheduler provide the ability to run Oracle BI Publisher reports as batch jobs. The Oracle Enterprise Scheduler post-processing infrastructure enables applying Oracle BI Publisher formatting templates to XML data and delivering the formatted reports by printing, faxing, and so on.

For more information about defining post-processing actions for scheduled jobs, see "Creating a Business Domain Layer Using Entity Objects" in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

9.16.1 How to Define Oracle BI Publisher Post-Processing for a Scheduled Job

Defining post-processing for a scheduled job involves the following:

- Define the post-processing action.
- Create a Java class for the post-processing action. The Java class uses the parameters collected by the Oracle Enterprise Scheduler UI and calls Oracle BI Publisher APIs as required.
- Create a native ADF Business Components view object to save parameters for post-processing, such as template name, output format, locale, and so on.

Before you begin:

1. Follow the instructions for setting up Oracle BI Publisher reporting as described in the Oracle BI Publisher documentation.

   Use the following file to set up reporting and seed your database with the relevant Oracle BI Publisher data:

Example 9–31 Location of the File for Setting Up Oracle BI Publisher Reporting and Seeding the Database

```
$BEAHOME/jdeveloper/jdev/oaext/adflib/PPActions.jar
```

2. Create an Oracle BI Publisher job definition, following the instructions in the Oracle BI Publisher documentation.
3. Define File Management Group (*FMG*) properties for the Oracle BI Publisher job definition as described in Section 9.4.2, "How to Define File Groups for a Job."

To create an Oracle BI Publisher post-processing action:

1. In the table `APPLCP_PP_ACTIONS`, define the post-processing action to be executed for the job.

   The columns to be seeded in the `APPLCP_PP_ACTIONS` table are as follows:

   - `Action_SN`: Define a short name for the action, used when post-processing actions are submitted programatically. For example, `OBFUSC8`.
   - `Action Name`: Enter a name for the action to be displayed in the user interface. This name is stored separately for translation purposes.
   - `Class`: Enter the name of the Java class that defines the logic for the post-processing action. For example, `oracle.apps.shh.obfuscate.PPobfuscate`.
   - `VO_Def_Name`: Enter the name of the view object used to collect the arguments for the post-processing action. For example, `oracle.apps.shh.obfuscate.PPobfuscateVO`.
   - `Type`: Enter the category of the post-processing action to be taken. Enter one of the following categories of post-processing actions:
     - `L`: Indicates a Layout post-processing action. Layout actions change the output of the job, and produce new output.
     - `O`: Indicates an Output post-processing action. Output actions act on the output created by the job and its layout actions, performing delivery, publishing, printing, and so on.

  - F: Indicates a Final post-processing action. Final Actions take no input. Final post-processing actions execute using the final status of the job after all Layout and Output actions have executed.

- On_Success: Indicate whether the post-processing action runs following a successful job. Enter Y or N.
- On_Warning: Indicate whether the post-processing action runs following a job that ends in a warning. Enter Y or N.
- On_Failure: Indicate whether the post-processing action runs following a failed job. Enter Y or N.
- SEQ_NUM: Enter a number to sequentially order the post-processing actions. Only registered post-processing actions of the same type can be sequentially ordered. This value determines both the order in which the tabs corresponding to the actions appear in the user interface, and the order in which the actions run.

Each action can also specify request parameters used by the post-processing action view object. These parameters must be set in the job definition for any job using this action. The parameter names are stored in the APPLCP_PP_ACTION_PARAMS table. The values of these parameters are accessible from the parameter view object at the time of job request submission. Post-processing actions can access all request parameters at runtime using the request ID.

2. Define a Java class for the post-processing action, implementing the interface oracle.apps.fnd.applcp.request.postprocess.PostProcess. Use the methods required by the interface as described in Table 9–4.

Table 9–4 Methods Required When Implementing the Interface oracle.apps.fnd.applcp.request.postprocess.PostProcess

Method	Description
`PostProcessState invokePostProcess(long requestID, String ppArguments[], ArrayList files);`	Receives the requestID, the ppArguments[] array of arguments collected from the view object (or submitted programmatically), and the files array list which identifies the files on which the action is to be taken. It is possible to specify the location of the output file.
`ArrayList getOutputFileList();`	Returns an array of the output files created by the post-processing action.

Additional methods used by the invokePostProcess method are shown in Table 9–5.

Table 9–5 Oracle BI Publisher Client API oracle.xdo.service.client.ReportService Used by the invokePostProcess method

Method	Description
`runReport()`	Enables the post-processing action to pass to the Business Intelligence Publisher the job's XML output along with the template ID and format (all collected during job request submission).

Additional methods used by the ReportRequest object are shown in Table 9–6.

Table 9–6 Oracle BI Publisher Client API oracle.xdo.service.client.types.ReportRequest Used by the ReportRequest Object

Method	Description
`setAttributeFormat()`	Set the format for the Oracle BI Publisher report request.
`setAttributeLocale()`	Set the locale data for the Oracle BI Publisher report request.
`setAttributeTemplate()`	Set the template for the Oracle BI Publisher report request.
`setXMLData()`	Set the XML data for the Oracle BI Publisher report request.

An example of a Java class that defines a post-processing action is shown in Example 9–32:

Example 9–32 A Java Class that Defines a Post-Processing Action

```
package oracle.apps.shh.Obfuscate;

import oracle.apps.fnd.applcp.request.postprocess.PostProcess;
import oracle.apps.fnd.applcp.util.ESSContext;
import oracle.apps.fnd.applcp.util.PostProcessState;
import oracle.as.scheduler.*;

public class PPobfuscate implements PostProcess {

  ArrayList myOutputFiles;

  ArrayList getOutputFileList()
  {
    return myOutputFiles;
  }

  PostProcessState invokePostProcess(long requestID, String ppArguments[],
    ArrayList files)
  {

    RuntimeService rService = null;
    RuntimeServiceHandle rHandle = null;
    try {
      // Accessing Runtime Details for a given requestID
      RequestDetail rDetail = null;
      RequestParameters rParam = null;
      String obfuscationSeed = ppArguments[0];
      String codedFileName = ppArguments[1];
      String myNewFile;
      String outDir = null;

      rService = ESSContext.getRuntimeService();
      if (rService != null) rHandle = rService.open();
      if (rHandle != null)  rDetail = getRequestDetails(rHandle, requestID);
      if (rDetail != null)  rParam  = rDetail.getParameters();
      if (rParam != null)   outDir  = rParam.getValue("outputWorkDirectory");
      if (outDir == null)
      {
        // Didn't get our details for some reason, usually an exception
        // would have been thrown by now. We handle this case to be robust.
```

```
        // log the ERROR to ODL
        return PostProcessState.ERROR;
      }
      // Check files
      if (files[0] == null)
      {
        // no files - PostProcessing should never call us in this state
        // in case it does - log Error to ODL
        return PostProcessState.ERROR;
      }
      // This example expects a single file
      myNewFile = outputDir + System.getProperty("file.separator") +
        codedFileName;
      Obfuscate.performObfuscation( files[0], obfuscationSeed, myNewFile );
      myOutputFiles[0] = myNewFile;

      // In case we're called on multiple files
      for ( i = 1; files[i] != null; i++ )
      {
        // appending a counter to the filename to be unique
        myNewFile = outputDir + System.getProperty("file.separator") +
        codedFileName + i ;
        Obfuscate.performObfuscation( files[i], obfuscationSeed, myNewFile );
        myOutputFiles[i] = myNewFile;
      }

      // Return our success
      return PostProcessState.SUCCESS;

    } catch (RuntimeServiceException rse)
    {
      // log RuntimeServiceException to ODL
      return PostProcessState.ERROR;
    } catch (Exception e)
    {
      // log Exception to ODL
      return PostProcessState.ERROR;
    } finally {
      if (rHandle != null)
        rService.close(rHandle);
    }
  }
} // end class
```

3. Create a native ADF Business Components view object to collect the parameters to be used in the post-processing action. Follow the procedure described in Section 9.4, "Creating a Job Definition." Define any view object attributes sequentially.

   If the view object requires access to action-specific values from the job definition, specify the required job definition parameters in the action definition. The submission UI automatically retrieves the values from the job definition metadata and sets them as Applications Core Session attributes that may be retrieved using the `ApplSession` standard API.

9.16.2 How to Define Oracle BI Publisher Post-Processing Actions for a Scheduled PL/SQL Job

Example 9–33 shows a PL/SQL job that includes Oracle BI Publisher post-processing actions. The PL/SQL job calls ess_runtime.add_pp_action so as to generate a layout for the data from the post-processing action. This example formats the XML generated by the job as a PDF file.

Example 9–33 Defining a Scheduled PL/SQL Job with Oracle BI Publisher Post-Processing Actions

```
declare
l_reqid   number;
l_props   ess_runtime.request_prop_table_t;
begin
.
 ess_runtime.add_pp_action (
   props              => l_props,               -- IN OUT
request_prop_table_t,
   action_order       => 1,                     -- order in which this post
processing action will execute.
   action_name        => 'BIPDocGen',           -- Action for Document
Generation (layout)
   on_success         => 'Y',                   -- Should this be called on
success,
   on_warning         => 'N',                   -- Should this be called on
warning,
   on_error           => 'N',                   -- Should this be called on
error,
   file_mgmt_group    => 'XML',                 -- File types this action
will process. It has to be defined in Job Defintion,
   step_path          => NULL,                  -- IN varchar2 default NULL,
   argument1          => 'XLABIPTEST_RTF',      -- Template name needed for
Documnet Generation action,
   argument2          => 'pdf'                  -- What type of layout file
will be generated by Document Generation action,
 );
.
  l_reqid :=
          ess_runtime.submit_request_adhoc_sched
             (application => 'SSEssWls',               -- Application
Application
              definition_type => 'JOB',
              definition_name => 'BIPTestJob',         -- Job definition
              definition_package => '/mypackage',      -- Job definition package
              props => l_props);
commit;
dbms_output.put_line('request_id = :'||l_reqid);
end;
```

9.16.3 What Happens When You Define Oracle BI Publisher Post-Processing Actions for a Scheduled Job

Depending on the FMG property set for the job definition, the relevant post-processing action is selected for the job.

The ppArguments array stores the values collected from the view object attributes. The array is passed to the invokePostProcess method which executes in the Java class that defines the post-processing action.

9.16.4 What Happens at Runtime: How Oracle BI Publisher Post-Processing Actions are Defined for a Scheduled Job

At runtime, the user interface uses the view object to collect the arguments for executing the post-processing action as defined in the table APPLCP_PP_ACTIONS. These arguments also instruct the user interface as to how to invoke the action logic.

The post-processing action accesses the XML output file from the job request, and passes the XML output to Oracle BI Publisher. The post-processing action creates a report request containing the XML data.

The post-processing action displays in the submission Oracle ADF UI. The UI enables adding a post-processing action for the scheduled job, selecting arguments for the action using the view object and selecting output options for the action. The user interface also displays the name of the file management group with which the output files are associated.

9.16.5 Invoking Post-Processing Actions Programmatically

You can invoke post-processing actions programmatically from a client using a Java or web service API. Both APIs require the same set of parameter values described in table Table 9–7.

For Java clients, call the addPPAction method of oracle.as.scheduler.cp.SubmissionUtil. The method takes the values needed to invoke the action and throws IllegalArgumentException if the number of arguments exceeds 10. Here's the method's declaration:

```
public static void addPPAction (RequestParameters params,
        int actionOrder,
        String actionName,
        String description,
        boolean onSuccess,
        boolean onWarning,
        boolean onError,
        String fileMgmtGroup,
        String[] arguments)
    throws IllegalArgumentException
```

For web service clients, you invoke the method using a proxy, as in Example 9–34. For more on the web service, see Chapter 10, "Using the Oracle Enterprise Scheduler Web Service.".

Example 9–34 Adding Post-Processing Actions for a Request

```
ESSWebService proxy = createProxy("addPPActions");

PostProcessAction ppAction = new PostProcessAction();
ppAction.setActionOrder(1);
ppAction.setActionName("BIPDocGen");
ppAction.setOnSuccess(true);
ppAction.setOnWarning(false);
ppAction.setOnError(false);
ppAction.getArguments().add("argument1");
ppAction.getArguments().add("argument2");
```

```
List<PostProcessAction> ppActionList =   new ArrayList<PostProcessAction>();
ppActionList.add(ppAction);

RequestParameters reqParams = new RequestParameters();
reqParams = proxy.addPPActions(reqParams, ppActionList);
```

Table 9–7 Parameters for Adding a Post-Processing Action

Parameter	Description
`params`	A `RequestParameters` object into which this method adds parameters.
`actionOrder`	The ordinal location of this action in the sequence of actions to be performed within the action domain. BIP processes requests starting with action order index 1.
`actionName`	The name of the action to perform. The following lists acceptable values for this parameter, along with the acceptable values you can use in the `arguments` parameter of this method. ■ `BIPDocGen`: for applying Oracle Business Intelligence Publisher templates. Acceptable argument parameter values are: ■ `argument1`: maps to report parameter TEMPLATE, the template name. ■ `argument2`: maps to report parameter OUTPUT_FORMAT, the output format for BIP document generation, for example, "pdf" or "html". ■ `argument3`: maps to report parameter LOCALE, the locale to be used while generating output. ■ `BIPPrintService`: for specifying the print action. Acceptable argument parameter values are: ■ `argument1`: maps to printerName ■ `argument2`: maps to numberOfCopies ■ `argument3`: maps to side ■ `argument4`: maps to tray ■ `argument5`: maps to pagesRange ■ `argument6`: maps to orientation ■ `BIPDeliveryEmail`: for specifying the email action. Acceptable argument parameter values are: ■ `argument1`: maps to emailServerName ■ `argument2`: maps to from ■ `argument3`: maps to to ■ `argument4`: maps to cc ■ `argument5`: maps to bcc ■ `argument6`: maps to replyTo ■ `argument7`: maps to subject ■ `argument8`: maps to messageBody ■ `BIPDeliveryFax`: for specifying the fax action. Acceptable argument parameter values are: ■ `argument1`: maps to faxServerName ■ `argument2`: maps to faxNumber
`description`	Description of this post processor action.

Table 9–7 (Cont.) Parameters for Adding a Post-Processing Action

Parameter	Description
`onSuccess`	Determines whether this action should be performed on successful completion of the job.
`onWarning`	Determines whether this action should be performed when the job or step has completed with a warning.
`onError`	Determines whether this action should be performed when the job or step has completed with an error.
`fileMgmtGroup`	Name of the File Management Group.For BIP applying template this will be 'XML'; defined in job definition Program.FMG property with value 'L.XML'.
`arguments`	A list of arguments for the post processor action. See the `actionName` parameter for values you can use for the `arguments` parameter.

9.17 Monitoring Scheduled Job Requests Using an Oracle ADF UI

It is possible to view previously submitted jobs by integrating the Monitoring Processes taskflow into an application.

For information about enabling tracing for jobs, see "Developing Diagnostic Tests" in *Oracle Fusion Applications Developer's Guide*. For more information about tracing Oracle Enterprise Scheduler jobs, see the section "Tracing Oracle Enterprise Scheduler Jobs" in the chapter "Managing Oracle Enterprise Scheduler Service and Jobs" in the *Oracle Fusion Applications Administrator's Guide*.

9.17.1 How to Monitor Scheduled Job Requests

The main steps involved in monitoring scheduled job requests using an Oracle ADF UI are as follows:

- Configure Oracle Enterprise Scheduler in JDeveloper
- Create and initialize an Oracle Fusion web application
- Create a UI Shell page and drop the Monitor Processes task flow onto it

> **Note:** Fields such as submission date, ready time, scheduled date, process start, name, type, definition, and so on, are not set unless the job request or sub-request is successfully validated.

To monitor scheduled job requests using an Oracle ADF UI:

1. Follow the instructions in Section 9.14.1, "How to Create an Oracle ADF User Interface for Submitting Job Requests" up to and including step 5.
2. Under the ViewController project, right-click Web Content and create a new JSF page called Consumer.jspx. Be sure to select the following options:
   - UIShell (template)
   - Create as XML Document
3. Create a new JSF page fragment. This page initializes the project.
4. Open adfc-config.xml and drag Consumer.jspx onto adfc-config.xml.
5. Right-click adfc-config.xml and select **Create ADF Menu**.

   The Create ADF Menu Model window displays.

6. Rename the default file root_menu.xml to something else.
7. Open the XML file created in the previous step. Look for an itemNode element as follows:

   ```
   <itemNode id="itemNode_JSF/JSPX page name">
   ```

   For example, the Consumer.jspx page has the following itemNode value:

   ```
   <itemNode id="itemNode_Consumer">
   ```

8. In the Structure window, right-click the root itemNode and select **Insert inside itemNode-itemNode_*JSF/JSPX page name*** > **itemNode**.
9. In Common Properties, enter the following values:
   - **id**: MonitorNode
   - **focusViewId**: /Consumer
10. In Advanced Properties, enter Monitor Processes in the label field.
11. Right-click the itemNode you just added and select **Go to Properties**.
12. In the Property Inspector, select **Advanced** and do the following:
    - Select the **dynamicMain** task type.
    - In the `taskFlowId` field, enter the following:

      ```
      /WEB-INF/oracle/apps/fnd/applcp/monitor/ui/flow/MonitorProcessesMainAreaFlow.xml#MonitorProcessesMainAreaFlow
      ```

    - Enter a string for the `pageTitle` parameter, which will become the title for the monitoring page. If this parameter is not specified, then the page title will be shown as "Manage Scheduled Processes".
13. Repeat steps 8-12 to create a second itemNode element with the following properties:
    - **id**: __Launcher_itemNode__FndTaskList
    - **focusViewId**: /Launcher
    - **label**: #{applcoreBundle.TASKS}
    - **Task Type**: defaultRegional
    - **taskFlowId**: `/WEB-INF/oracle/apps/fnd/applcore/patterns/uishell/ui/publicFlow/TasksList.xml#TasksList`
14. Right-click adfc-config.xml and select **Link ADF Menu to Navigator**.
15. Configure Oracle JDeveloper Integrated Oracle WebLogic Server for development with Oracle Enterprise Scheduler extensions.
16. Deploy and test the application.

9.17.2 How to Embed a Table of Search Results as a Region on a Page

You can embed a table of job request search results as a region on a page. A number of task flow parameters can be used to further specify the job requests returned by the search.

To embed a search results table as a region:

1. Add the **Applications Concurrent Processing (View Controller)** library to the ViewController project.

   For more information about adding this library to the project, see Section 9.3.1.

2. In the Resource Palette, select **File System** > **Applications Core** > **MonitorProcesses-View.jar** > **ADF Task Flows**.

3. Drag and drop onto the page as a region the **SearchResultsFlow** task flow.

   The task flow accepts the following parameters:

   - `processId`: The request ID number uniquely identifying the process.
   - `processName`: The name of the process, which corresponds to the name of the job definition.
   - `processNameList`: Fetches the job requests of multiple process names using a list which contains the relevant job names.

     When specifying the task flow parameter `processName`, this parameter takes precedence over the task flow parameter `processNameList`. The requests returned are for the single process name specified by the `processName` parameter only.
   - `scheduledDays`: Queries requests for the last *n* days. If this parameter is not specified in a work area task flow, job requests from the last three days are displayed. If the value of this parameter is greater than three days, then the parameter value will be taken as three and only the last three days of job requests display.
   - `status`: The status of the request. This filter narrows down the result set to display only the requests with the selected status in the filter.

     If the `status` input parameter is not specified, then the results table shows all requests with all statuses (by default, `All` is selected in the status filter list).

     If the `status` input parameter is specified, then the results table show only the requests of the given status. The selected status is chosen as the default in the status filter list.
   - `isEmbedResults`: A boolean value that indicates whether search results are embedded in the task flow. True or false.

     Set to `true` in order to embed table results.
   - Time Range Filter: This filter is used to narrow down the result set to show only the requests for last `n` hours. This filter lists the following values in a combobox: (1) Last 1 Hour, (2) Last 12 Hours, (3) Last 24 Hours, (4) Last 48 Hours and (5) Last 72 Hours.

     The default selected item displays based on the value assigned or given to the task flow parameter `scheduledDays`.

     A `scheduledDays` value of 1 means the time range filter list displays only the first three items.

     A `scheduledDays` value of 2 means the time range filter list displays only the first four items.

     If the value of `scheduledDays` is 1, then by default, the time range combobox displays **Last 24 Hours**.

If the value of scheduledDays is 3 or more, then by default, the time range combobox displays **Last 72 Hours**.

- pageTitle: When passed, the task flow will render this passed String value as the page title. Optional.
- requireRootOutcome: If true is passed as the value, then the task flow will generate root-outcome when the user clicks on the Submit or Cancel buttons. By default the task flow generates parent-outcome.

Specifying more than one of these parameters causes the search to run using the AND conjunction.

9.17.3 How to Log Scheduled Job Requests in an Oracle ADF UI

You can enable Oracle Diagnostic Logging in an Oracle ADF UI used to monitor scheduled job requests. When enabling logging, the UI displays a View Log button.

The View Log functionality in the monitoring UI applies only to scheduled requests with a persistenceMode property set to file. Hence, the View Log button in the scheduled request submission monitoring UI displays only when viewing requests with persistenceMode property set to file.

The only other valid value for the persisteceMode is content. The View Log button is hidden for all requests with a persistenceMode property value of content. If the persistenceMode property is not specified for a given request, then the monitoring UI defaults to a persistenceMode value of file, and displays the View Log button when viewing relevant requests.

To log scheduled job requests:

1. Open the server's logging.xml file.
2. In the logging.xml file, enter the required logging level for oracle.apps.fnd.applcp.srs, for example: INFO, FINE, FINER or FINEST.

   Example 9–35 shows a snippet of a logging.xml file with Oracle Diagnostic Logging configured.

Example 9–35 Enabling Logging in the logging.xml File

```
<logger name='oracle.apps.fnd.applcp.srs' level='FINEST'
    useParentHandlers='false'>
   <handler name='odl-handler'/>
</logger>
```

3. Save the logging.xml file and restart the server.

9.17.4 How to Troubleshoot an Oracle ADF UI Used to Monitor Scheduled Job Requests

Some useful tips for troubleshooting the Oracle ADF UI used to monitor scheduled job requests.

- **Displaying a readable name**. When defining metadata, use the display-name attribute to configure the name to be displayed in the Oracle ADF UI. The monitoring UI will display the value defined for the display-name attribute. If this attribute is not defined, the UI displays the value of the metadata-name attribute assigned to the metadata.
- **Displaying multiple links in the task flow UI that each display a pop-up window with a different job definition**. The recommended approach is to create

a single page fragment that contains the scheduled request submission task flow within an Oracle ADF region. This page is re-used by each link to display a different job definition in the scheduled request submission UI. For each link, be sure to pass the relevant parameters such as the job definition name, package name, and so on. This approach ensures that the UI session creates and uses a single instance of the task flow.

- **Displaying the correct name given the metadata name and display name attributes**. By default, the display name takes precedence and displays in the UI. If the display name is not defined, then the UI displays the job or job set name.
- **Resolving name conflicts between a job metadata parameter name and a request parameter with the same name**. Oracle Enterprise Scheduler uses the following rules to resolve parameter name conflicts.
  - **The last definition takes precedence**. When the same parameter is defined repeatedly with the read-only flag set to false in all cases, the last parameter definition takes precedence. For example, a property specified at the job request level takes precedence over the same property specified at the job definition level.
  - **The first read-only definition takes precedence**. When the same parameter is defined repeatedly and at least one definition is read-only (that is, the `ParameterInfo` read-only flag is set to true), the first read-only definition takes precedence. For example a read-only parameter specified at the job type definition level takes precedence over a property with the same name specified at the job definition level, regardless of whether or not it is read-only.
- **Resolving name conflicts between the job or job set metadata name and display name attributes**. By default, the display name takes precedence over the metadata name. If the display name is not defined, then the UI defaults to displaying the job or job set name.
- **Understanding the state of a job request**. There are 20 possible states for a job request, each with a corresponding number value. These are shown in Table 9–8.

Table 9–8 Job Request States

Job State Number	Job Request State	Description
-1	`UNKNOWN`	The state of the job request is unknown.
1	`WAIT`	The job request is awaiting dispatch.
2	`READY`	The job request has been dispatched and is awaiting processing.
3	`RUNNING`	The job request is being processed.
4	`COMPLETED`	The job request has completed and post-processing has commenced.
5	`BLOCKED`	The job request is blocked by one or more incompatible job requests.
6	`HOLD`	The job request has been explicitly held.
7	`CANCELLING`	The job request has been cancelled and is awaiting acknowledgement.
8	`EXPIRED`	The job request expired before it could be processed.
9	`CANCELLED`	The job request was cancelled.
10	`ERROR`	The job request has run and resulted in an error.
11	`WARNING`	The job request has run and resulted in a warning.

Table 9–8 (Cont.) Job Request States

Job State Number	Job Request State	Description
12	SUCCEEDED	The job request has run and completed successfully.
13	PAUSED	The job request paused for sub-request completion.
14	PENDING_VALIDATION	The job request has been submitted but has not been validated.
15	VALIDATION_FAILED	The job request has been submitted, but validation has failed.
16	SCHEDULE_ENDED	The schedule for the job request has ended, or the job request expiration time specified at submission has been reached.
17	FINISHED	The job request, and all child job requests, have finished.
18	ERROR_AUTO_RETRY	The job request has run, resulted in an error, and is eligible for automatic retry.
19	ERROR_MANUAL_ RECOVERY	The job request requires manual intervention in order to be retried or transition to a terminal state.

- **Fixing an Oracle BI Publisher report that does not generate, even though the Oracle Enterprise Scheduler schema `REQUEST_PROPERTY` table contains all the relevant post-processing parameters**. Verify that the post-processing parameters begin with index value of 1. If a set of parameters begins with an index value of 0 (such as `pp.0.action`), then the Oracle BI Publisher report will not generate. Oracle BI Publisher expects parameters to begin with an index value of 1. In the case of a job set with multiple Oracle BI Publisher jobs, verify that all the individual step post-processing actions begin with an index value of 1.
- **Fixing a scheduled request submission UI that does not display, and throws a partial page rendering error in the browser indicating that the `drTaskflowId` is invalid**. This error may occur as a result of any of the following.
  - The object `oracle.as.scheduler.JobDefinition` may be unavailable to the scheduled request submission UI, which attempts to query the object using the `MetadataService` API.
  - The job definition name or the job definition package name is incorrect when passed as task flow parameters. Ensure that the package name does not end with a trailing forward slash.
  - The metadata permissions are not properly configured for the user who is currently logged in. The `JobDefinition` object, being stored in Oracle Metadata Repository, requires adequate metadata permissions in order to read and modify the `JobDefinition` metadata. Ensure that the Oracle Metadata Repository to which you are referring contains the job definition name in the proper package hierarchy.

9.18 Using a Task Flow Template for Submitting Scheduled Requests through an Oracle ADF UI

The Oracle ADF UI used to submit scheduled requests supports basic and advanced modes. Switching between modes requires page navigation between two view activities.

In some cases, you may want to use a custom parameter task flow for the UI in the context of an Oracle Fusion web application. One such use case is when you require a method call activity as the default activity of a custom bounded task flow so as to initialize the parameters view object and flex filters defined in that task flow.

When using page navigation between two view activities and custom bounded task flows with a default method call activity, switching between basic and advanced modes might re-initialize the related view objects and entity objects. If this happens, any data entered in basic mode is lost when changing to advanced mode.

The task flow template enables switching between basic and advanced modes in the scheduled request submission Oracle ADF UI without losing data.

9.18.1 How to Use a Task Flow Template for Submitting Scheduled Requests through an Oracle ADF UI

A bundled task flow template is provided, containing the components required to enable switching between basic and advanced modes in the Oracle ADF UI. The task flow template adds a router activity and an input parameter to the custom bounded task flow. Configure the router activity as the default activity.

You need only extend the task flow template as needed and implement the activity IDs defined in the task flow template.

Example 9–36 shows a sample implementation of the task flow template.

Example 9–36 Task Flow Template

```
<?xml version="1.0" encoding="UTF-8" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2">
  <task-flow-template id="srs-custom-task-flow-template">
    <default-activity id="defActivity">defaultRouter</default-activity>
    <input-parameter-definition id="param1">
      <description id="paramDescription">Parameter to decide on initialization.</description>
      <name id="paramName">shouldInitialize</name>
      <value id="paramID">#{pageFlowScope.shouldInitialize}</value>
      <class id="paramType">boolean</class>
      <required/>
    </input-parameter-definition>

    <router id="defaultRouter">
      <case id="routerCaseID">
        <expression id="routerExprID">#{pageFlowScope.shouldInitialize}</expression>
        <outcome id="outcomeID">initializeTaskflow</outcome>
      </case>
      <default-outcome id="defOutcomeID">skip</default-outcome>
    </router>

    <control-flow-rule id="ctrlFlwRulID">
      <from-activity-id id="FrmAc1">defaultRouter</from-activity-id>
      <control-flow-case id="CtrlCase1">
        <from-outcome id="FrmAct3">initializeTaskflow</from-outcome>
        <to-activity-id id="ToAct1">initActivity</to-activity-id>
      </control-flow-case>
      <control-flow-case id="CtrlCase2">
        <from-outcome id="FrmAct2">skip</from-outcome>
        <to-activity-id id="ToAct2">defaultView</to-activity-id>
        </control-flow-case>
     </control-flow-rule>
    <use-page-fragments/>
  </task-flow-template>
</adfc-config>
```

The task flow template defines the following:

- A default-activity,
- An input parameter of boolean type,
- A router activity,
- A control-flow-rule containing two cases.

9.18.2 How to Extend the Task Flow Template for Submitting Scheduled Requests through an Oracle ADF UI

If you need to create your own custom bounded task flow UI for the parameters section of the scheduled request submission UI, you will need to extend this template.

To extend the task flow template for the Oracle ADF UI used to submit scheduled requests:

1. When creating a new task flow, extend the task flow by selecting **Use a template**. (For more information, see the chapter "Creating ADF Task Flows in *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.") Alternatively, add the lines of code shown in Example 9–37 to the task flow XML file.

Example 9–37 Extending a Task Flow

```
<template-reference>
     <document id="doc1">/WEB-INF/srs-custom-task-flow-template.xml</document>
     <id id="temid">srs-custom-task-flow-template</id>
</template-reference>
```

> **Note:** Make sure that your bounded task flow does not define any default activity.

2. Implement the activity IDs defined in the template, which are invoked by the router activity in the template.
   - `initActivity`: The ID of the method call activity.
   - `defaultView`: The ID of the default view activity.

   To do this, to the task flow drag and drop the `createInsert` method from the VO used in the defaultView. This creates a pagedef file and adds the binding details in DateBinding.cpx.
3. Define a control flow rule to navigate from `initActivity` to `defaultView`. This navigation depends on the outcome of `initActivity`, as well as individual use cases.

   Example 9–38 shows a sample implementation of a control flow rule.

Example 9–38 Implementing a Control Flow Rule

```
<control-flow-rule>
     <from-activity-id>initActivity</from-activity-id>
     <control-flow-case>
          <from-outcome>outcome_of_init_activity</from-outcome>
          <to-activity-id>defaultView</to-activity-id>
     </control-flow-case>
</control-flow-rule>
```

9.18.3 What Happens When you Use a Task Flow Template for Submitting Scheduled Requests through an Oracle ADF UI

Based on the value of the input parameter, the router invokes the method call activity or skips it, and invokes the view activity directly. The Oracle ADF UI must pass the correct parameter values to the task flow while switching modes.

9.18.4 What Happens at Runtime: How a Task Flow Template Is Used to Submit Scheduled Requests through an Oracle ADF UI

When loading the initial page in basic mode, the method call activity is invoked. While loading the page in the advanced mode, the custom bounded task flow directly invokes the view activity. This ensures that the user entered data persists in the view objects across modes.

If the custom task flow UI does not render correctly, check whether transactional properties have been set in the custom task flow, such as `requires-transaction`, and so on.

Remove transactional properties from the task flow definition and set the data control scope to shared.

As the parent scheduled request submission UI task flow already has a transaction, Oracle ADF will commit all called task flow transactions as long as the data controls are shared.

Note: When using the UI to schedule a job to run for a year, for example, a maximum of 300 occurrences display when clicking **Customize Times**.

9.19 Securing Oracle ADF UIs

When creating Oracle ADF UIs for scheduled jobs, you can secure the individual task flows involved using a security policy.

The task flows you can secure are as follows.

Scheduling Job Requests UI

- `/WEB-INF/ScheduleRequest-taskflow.xml`
  - `/WEB-INF/srs-test-task-flow.xml#srs-test-task-flow`
  - `/WEB-INF/LayoutRN-taskflow.xml#LayoutRN-taskflow`
  - `/WEB-INF/NotifyRN-taskflow.xml#NotifyRN-taskflow`
  - `/WEB-INF/ScheduleRN_taskflow.xml#ScheduleRN_taskflow`

Monitoring Job Requests UI

- `/WEB-INF/oracle/apps/fnd/applcp/monitor/ui/flow/MonitorProcessesMainAreaFlow.xml#MonitorProcessesMainAreaFlow`
  - `/WEB-INF/oracle/apps/fnd/applcp/monitor/ui/flow/EmptyFlow.xml`

9.20 Integrating Scheduled Job Logging with Fusion Applications

Oracle Enterprise Scheduler is fully integrated with Oracle Fusion Applications logging. The logger captures Oracle Enterprise Scheduler-specific attributes when invoking logging from within the context of a running job request. You can set the values to these Oracle Enterprise Scheduler attributes within the context of defining a job.

Jobs can generate a log file on the file system that can be viewed with the Monitoring UI.

In a typically configured Oracle Enterprise Scheduler hosting application, log and output files are stored in an Oracle Universal Content Management content repository rather than on the file system. These files are available to end users through a page you provide for monitoring scheduled job requests. For more on request monitoring, see Section 9.17, "Monitoring Scheduled Job Requests Using an Oracle ADF UI."

9.21 Logging Scheduled Jobs

Log messages written using the request log file APIs are written to the request log file and Oracle Fusion Applications logging at a severity level of `FINE` (only if logging is enabled at a level of `FINE` or lower).

9.21.1 Using the Request Log

> **Note:** Do not use the Request Log for debug and internal error reporting. For Oracle Enterprise Scheduler jobs, the "Request Log" is equivalent to the end-user UI for online applications. When writing Oracle Enterprise Scheduler job code, you should ideally log only translatable end user-oriented messages to the Request Log. You should not use the Request Log for debug messages or internal error messages that are oriented to system administrators and/or Oracle Support. Please keep in mind that the audience for debug messages and detailed internal error messages is typically system administrators and Oracle Support, not the end user.
>
> Therefore, debug and detailed internal error messages should be logged to `FND_LOG` only.

For Oracle Enterprise Scheduler jobs, the request log is equivalent to the end user interface for web applications. When developing an Oracle Enterprise Scheduler job, make sure to log to the request log only translatable end-user oriented messages.

For example, if an end user inputs a bad parameter to the Oracle Enterprise Scheduler job, a translated error message logged to the request log is displayed to the end user. The end user can then take the relevant corrective action.

Example 9–39 shows how to set log messages using the request log.

Example 9–39 Setting Log Messages Using the Request Log

```
-- Seeded message to be displayed to the end user.
FND_MESSAGE.SET_NAME('FND', 'INVALID_PARAMETER');
-- Runtime parameter information
FND_MESSAGE.SET_TOKEN('PARAM_NAME', pName);
FND_MESSAGE.SET_TOKEN('PARAM_VALUE', pValue);
-- The following is useful for auto-logging errors.
```

```
FND_MESSAGE.SET_MODULE('fnd.plsql.mypackage.myfuntionA');
fnd_file.put_line( FND_FILE.LOG, FND_MESSAGE.GET );
```

If the Oracle Enterprise Scheduler job fails due to an internal software error, log the detailed failure message to `FND_LOG` for the system administrator or support. You can also log a high-level generic message to the request log so as to inform end users of the error. An example of a generic error message intended for end users: "Your request could not be completed due to an internal error."

9.21.2 Using the Output File

Note: Do not use the output file for debugging and internal error reporting.

The output file is a formally formatted file generated by an Oracle Enterprise Scheduler job. An output file can be sent to a printer or viewed in a UI window. Example 9–40 shows an invoice sent to an output file.

Example 9–40 Invoice Output File

```
fnd_file.put_line( FND_FILE.OUTPUT, '******** XYZ Invoice ********' );
```

9.21.3 Debugging and Error Logging

Debug and error logging should be done using the Diagnostic Logging APIs only. The Oracle Enterprise Scheduler Request Log should not be used for system administrator or Oracle support-oriented debug and error logging purposes. The Request Log is for the end users and it should only contain messages that are clear and easy for end users to understand. When an error occurs in an Oracle Enterprise Scheduler job, an appropriate high-level (and, ideally, translated) message should be used to report the error to the end user through the Request Log. The details of the error and any debug messages should be logged with Diagnostic Logging APIs.

Common PL/SQL, Java, or C code that could be invoked by both Oracle Enterprise Scheduler jobs and interactive application code should only use Diagnostic Logging APIs. If needed, the wrapper Oracle Enterprise Scheduler job should perform appropriate batching and logging to the Request Log for progress reporting purposes.

For more information, see the chapter "Managing Log Files and Diagnostic Data" in *Oracle Fusion Middleware Administrator's Guide*.

Using Logging in a Java Application

In Java jobs, use `AppsLog` for debugging and error logging. You can retrieve an `AppsLog` instance from the `CpContext` object, by calling `getLog()`.

Example 9–41 shows the use of logging in a Java application.

Example 9–41 Logging in Java Using AppsLog

```
public boolean authenticate(AppsContext ctx, String user, String passwd)
      throws SQLException, NoSuchUserException {
    AppsLog alog = (AppsLog) ctx.getLog();
    if(alog.isEnabled(Log.PROCEDURE)) /* To avoid String Concat if not enabled */
    alog.write("fnd.security.LoginManager.authenticate.begin",
                "User=" + user, Log.PROCEDURE);
```

```
  /* Never log plain-text security sensitive parameters like passwd! */
  try {
    validUser = checkinDB(user, passwd);
  } catch(NoSuchUserException nsue) {
      if(alog.isEnabled(Log.EXCEPTION))
      alog.write("fnd.security.LoginManager.authenticate",nsue, Log.EXCEPTION);
    throw nsue; // Allow the caller to Handle it appropriately
  } catch(SQLException sqle) {
   if(alog.isEnabled(Log.UNEXPECTED)) {
      alog.write("fnd.security.LoginManager.authenticate", sqle,
      Log.UNEXPECTED);
      Message Msg = new Message("FND", "LOGIN_ERROR"); /* System Alert */
      Msg.setToken("ERRNO", sqle.getErrorCode(), false);
      Msg.setToken("REASON", sqle.getMessage(), false);
     /* Message Dictionary messages should be logged using write(..Message..),
      * and never using write(..String..) */
      alog.write("fnd.security.LoginManager.authenticate", Msg, Log.UNEXPECTED);
   }
   throw sqle; // Allow the caller to handle it appropriately
  } // End of catch(SQLException sqle)
  if(alog.isEnabled(Log.PROCEDURE)) /* To avoid String Concat if not enabled */
     alog.write("fnd.security.LoginManager.authenticate.end",
                "validUser=" + validUser, Log.PROCEDURE);
  return success;
 }
```

> **Note:** Example 9–41 uses an active `WebAppsContext`. Do not attempt to log messages using an inactive or freed `WebAppsContext`, as this can cause connection leaks.

Using Logging in a PL/SQL Application

PL/SQL APIs are part of the `FND_LOG` package. These APIs require invoking relevant application user session initialization APIs—such as `FND_GLOBAL.INITIALIZE()`— in order to set up user session properties in the database session.

These application user session properties, including `UserId`, `RespId`, `AppId`, `SessionId`, are needed for the log APIs. Typically, Applications Core invokes these session initialization APIs.

Log plain text messages with `FND_LOG.STRING()`. Log translatable message dictionary messages with `FND_LOG.MESSAGE()`. `FND_LOG.MESSAGE()` logs messages in encoded, but not translated, format, and allows the Log Viewer UI to handle translating messages based on the language preferences of the system administrator viewing the messages.

For details regarding the `FND_LOG` API, run `$fnd/patch/115/sql/AFUTLOGB.pls` at the prompt.

Example 9–42 PL/SQL Logging Syntax

```
PACKAGE FND_LOG IS
   LEVEL_UNEXPECTED CONSTANT NUMBER  := 6;
   LEVEL_ERROR      CONSTANT NUMBER  := 5;
   LEVEL_EXCEPTION  CONSTANT NUMBER  := 4;
   LEVEL_EVENT      CONSTANT NUMBER  := 3;
   LEVEL_PROCEDURE  CONSTANT NUMBER  := 2;
   LEVEL_STATEMENT  CONSTANT NUMBER  := 1;

  /*
```

```
   **  Writes the message to the log file for the specified
   **  level and module
   **  if logging is enabled for this level and module
   */
   PROCEDURE STRING(LOG_LEVEL IN NUMBER,
                    MODULE    IN VARCHAR2,
                    MESSAGE   IN VARCHAR2);

   /*
   **  Writes a message to the log file if this level and module
   **  are enabled.
   **  The message gets set previously with FND_MESSAGE.SET_NAME,
   **  SET_TOKEN, etc.
   **  The message is popped off the message dictionary stack,
   **  if POP_MESSAGE is TRUE.
   **  Pass FALSE for POP_MESSAGE if the message will also be
   **  displayed to the user later.
   **  Example usage:
   **  FND_MESSAGE.SET_NAME(...);    -- Set message
   **  FND_MESSAGE.SET_TOKEN(...);   -- Set token in message
   **  FND_LOG.MESSAGE(..., FALSE);  -- Log message
   **  FND_MESSAGE.RAISE_ERROR;      -- Display message
   */
   PROCEDURE MESSAGE(LOG_LEVEL   IN NUMBER,
                     MODULE      IN VARCHAR2,
                     POP_MESSAGE IN BOOLEAN DEFAULT NULL);

   /*
   ** Tests whether logging is enabled for this level and module,
   ** to avoid the performance penalty of building long debug
   ** message strings unnecessarily.
   */
   FUNCTION TEST(LOG_LEVEL IN NUMBER, MODULE IN VARCHAR2)
RETURN BOOLEAN;
```

Example 9–43 shows how to log a message in PL/SQL after the AOL session has been initiliazed.

Example 9–43 Logging a Message in PL/SQL After the AOL Session Has Been Initialized

```
begin

  /* Call a routine that logs messages. */
  /* For performance purposes, check whether logging is enabled. */
  if( FND_LOG.LEVEL_PROCEDURE >= FND_LOG.G_CURRENT_RUNTIME_LEVEL ) then
    FND_LOG.STRING(FND_LOG.LEVEL_PROCEDURE,
        'fnd.plsql.MYSTUFF.FUNCTIONA.begin', 'Hello, world!' );
  end if;
/
```

The global variable `FND_LOG.G_CURRENT_RUNTIME_LEVEL` allows callers to avoid a function call for messages at a lower level than the current configured level. If logging is disabled, the current runtime level is set to a large number such as 9999 so that it is sufficient to simply log messages with levels greater than or equal to this number. This global variable is automatically populated by the `FND_LOG_REPOSITORY` package during session and context initialization.

Example 9–44 shows sample code that illustrates the use of the global variable `FND_LOG.G_CURRENT_RUNTIME_LEVEL`.

Example 9–44 Logging a Message in PL/SQL Using FND_LOG.G_CURRENT_RUNTIME_LEVEL

```
if( FND_LOG.LEVEL_STATEMENT >= FND_LOG.G_CURRENT_RUNTIME_LEVEL ) then
      dbg_msg := create_lengthy_debug_message(...);
      FND_LOG.STRING(FND_LOG.LEVEL_STATEMENT
           'fnd.form.ABCDEFGH.PACKAGEA.FUNCTIONB.firstlabel', dbg_msg);
end if;
```

> **Note:** For PL/SQL in a forms client, use the same APIs. Use `FND_LOG.TEST()` to check whether logging is enabled.

Example 9–45 shows logging message dictionary messages.

Example 9–45 Logging Message Dictionary Messages

```
if( FND_LOG.LEVEL_UNEXPECTED >=
            FND_LOG.G_CURRENT_RUNTIME_LEVEL) then
        FND_MESSAGE.SET_NAME('FND', 'LOGIN_ERROR'); -- Seeded Message
        -- Runtime Information
        FND_MESSAGE.SET_TOKEN('ERRNO', sqlcode);
        FND_MESSAGE.SET_TOKEN('REASON', sqlerrm);
        FND_LOG.MESSAGE(FND_LOG.LEVEL_UNEXPECTED,
                        'fnd.plsql.Login.validate', TRUE);
end if;
```

Using Logging in C

Example 9–46 illustrates the use of logging in a C application.

Example 9–46 Logging in C

```
#define  AFLOG_UNEXPECTED  6
#define  AFLOG_ERROR       5
#define  AFLOG_EXCEPTION   4
#define  AFLOG_EVENT       3
#define  AFLOG_PROCEDURE   2
#define  AFLOG_STATEMENT   1

/*
** Writes a message to the log file if this level and module is
** enabled
*/
void aflogstr(/*_ sb4 level, text *module, text* message _*/);

/*
** Writes a message to the log file if this level and module is
** enabled.
** If pop_message=TRUE, the message is popped off the message
** Dictionary stack where it was set with afdstring() afdtoken(),
** etc. The stack is not cleared (so messages below will still be
** there in any case).
*/
void aflogmsg(/*_ sb4 level, text *module, boolean pop_message _*/);

/*
** Tests whether logging is enabled for this level and module, to
** avoid the performance penalty of building long debug message
```

```
** strings
*/
boolean aflogtest(/*_ sb4 level, text *module _*/);

/*
** Internal
** This routine initializes the logging system from the profiles.
** It will also set up the current session and username in its state */
void afloginit();
```

10

Using the Oracle Enterprise Scheduler Web Service

Oracle Enterprise Scheduler provides a rich set of functionality for enterprise level scheduling. This functionality includes support for the Oracle Enterprise Scheduler web service (`ESSWebservice`) to access a subset of the Oracle Enterprise Scheduler runtime functionality.

This chapter includes the following sections:

10.1 Introduction to the Oracle Enterprise Scheduler Web Service

Oracle Enterprise Scheduler provides a rich set of functionality for enterprise level scheduling. This functionality includes support for the following operations:

- Creating and managing Oracle Enterprise Scheduler metadata
- Submitting and managing Oracle Enterprise Scheduler job requests
- Configuring and managing Oracle Enterprise Scheduler

Client applications can use the Oracle Enterprise Scheduler web service (ESSWebservice) to access a subset of the Oracle Enterprise Scheduler runtime functionality. The ESSWebservice is provided primarily to support SOA integration, for example invoking Oracle Enterprise Scheduler from a BPEL process. However, any client that needs a web service to interact with Oracle Enterprise Scheduler can use ESSWebservice. ESSWebservice exposes job scheduling and management functionality for request submission and request management.

ESSWebservice is deployed within the Oracle Enterprise Scheduler application, where the application is a Java EE application within the Oracle Enterprise Scheduler runtime framework. Thus, the ESSWebservice is available on every node where Oracle Enterprise Scheduler is installed and deployed.

The ESSWebservice is a synchronous web service, such that all the operations invoked are synchronous operations. Since internally, the job execution model in Oracle Enterprise Scheduler is asynchronous, the APIs themselves do not need to be asynchronous. However, Oracle Enterprise Scheduler web service also provides the capability to retrieve the job completion events asynchronously (in a manner similar to implementing the Oracle Enterprise Scheduler EventListener contract in the core API layer).

The ESSWebservice WSDL describes the complete functionality for the ESSWebservice. Table 10–1 summarizes the operations available with ESSWebservice.

Table 10–1 Summary of Operations Available with ESSWebservice

Operation	Communication Type	Description
`addPPAction`	Synchronous	Adds a post-processing action to a step in a job set request. This method is called prior to submitting the request. The method provides support for action previously supported by add_printer, add_notification, add_layout in concurrent processing. The parameters to these legacy routines are passed as arguments to addPPAction in the order in which they were declared in the original routine. For more information, see Section 10.8, "Using Additional ESSWebService Operations"
`addPPActions`	Synchronous	Similar to `addPPAction`, except that you can package multiple actions in your request.
`cancelRequest`	Synchronous	Cancels the processing of a request that is not in a terminal state.
`deleteRequest`	Synchronous	Marks a request in a terminal state for deletion. This does not physically remove any data, although the request will no longer be accessible by most methods. For parent requests, this operation will cascade to all children.
`getCompletionStatus`	Asynchronous	Registers for an asynchronous status update when the request completes. A one-way operation with a separate asynchronous response.
`getRequestDetail`	Synchronous	Gets the runtime details of the specified request.
`getRequestState`	Synchronous	Retrieves the current state of the specified request.
`holdRequest`	Synchronous	Withholds further processing of a request that is in `WAIT` or `READY` state. For parent requests, this operation will cascade to all eligible child requests.
`releaseRequest`	Synchronous	Releases a request from the `HOLD` state. For parent requests, this operation will cascade to all eligible child requests.
`setAsyncRequestStatus`	Synchronous	Sets the status of an asynchronous java job.
`setNLSOptions`	Synchronous	Sets NLS environment options for a request.
`setStepsArgs`	Synchronous	Marshals arguments in the previous concurrent processing style into a Oracle Enterprise Scheduler properties for a step in a job set request. This operation is invoked prior to submitting a request. For more information, see Section 10.8, "Using Additional ESSWebService Operations".

Table 10–1 (Cont.) Summary of Operations Available with ESSWebservice

Operation	Communication Type	Description
`setSubmitArgs`	Synchronous	Marshals arguments in the previous concurrent processing style into Oracle Enterprise Scheduler properties.This operation is invoked prior to submitting the request. The key of each argument is ARGUMENT_PREFIX#, where # is the ordinal value of the argument. For example ARGUMENT_PREFIX1="firstArg" and ARGUMENT_PREFIX2="secondArg". For more information, see Section 10.8, "Using Additional ESSWebService Operations".
`submitRecurringRequest`	Synchronous	Submits a new recurring job request (a request with a schedule). For more information, see Section 10.8, "Using Additional ESSWebService Operations".
`submitRequest`	Synchronous	Submits a new job request. For more information, see Section 10.4, "Use Case Using Scheduler ESSWebservice from a BPEL Process"

10.2 Developing and Using ESSWebservice Applications

Oracle Enterprise Scheduler executes a job request, for example a Java type job request, in the context of the application that submitted the job. Typically, for development purposes, Oracle Enterprise Scheduler and client applications co-exist locally on any given node which allows Oracle Enterprise Scheduler to execute the job in the context of the target application. For the purposes of production, the client application and Oracle Enterprise Scheduler often reside on different servers.

A Java EE application that uses Oracle Enterprise Scheduler contains all the Oracle Enterprise Scheduler artifacts including the following:

- Metadata, including a job type, a job definition, a schedule, and any other required metadata such as a job set.
- Job implementation classes (for Java jobs).
- A Required Oracle Enterprise Scheduler endpoint description (an MDB description in ejb-jar.xml).

Any clients interacting with Oracle Enterprise Scheduler using ESSWebservice need to provide such a Java EE application, such that Oracle Enterprise Scheduler can run jobs in the context of the correct target application. All such web service clients must know the name of the corresponding Java EE hosting application and should pass it to Oracle Enterprise Scheduler using the web service call wherever required (where this is required is defined in the WSDL).

The details for developing this hosting application are described in Chapter 3, "Use Case Oracle Enterprise Scheduler Sample Application." Such an application is a regular Oracle Enterprise Scheduler client application, but the job request submission and other Oracle Enterprise Scheduler interactions may be skipped, as these calls are generated through the ESSWebservice.

10.2.1 How to Develop and Use an ESSWebservice Java EE Application

When the Oracle Enterprise Scheduler functionality is accessed using the ESSWebservice web service, a corresponding hosting Java EE application needs to be available to Oracle Enterprise Scheduler. Even though clients can interact with Oracle Enterprise Scheduler remotely using the Oracle Enterprise Scheduler web service, the associated Java EE application must still be co-located with Oracle Enterprise Scheduler. This allows Oracle Enterprise Scheduler to execute job requests in the correct application context. Therefore ESSWebservice clients still need to develop, package and deploy a corresponding Java EE application that contains all the required Oracle Enterprise Scheduler artifacts. For information about developing an Oracle

Enterprise Scheduler application, see Chapter 3, "Use Case Oracle Enterprise Scheduler Sample Application."

10.2.2 How to Develop and Use an ESSWebservice SOA Application with BPEL

For SOA clients all the SOA components such as a BPEL processor are deployed as a SOA composite. A SOA composite is not a Java EE application. The composite is executed using the SOA fabric runtime framework (within soa-infra).

For SOA components, create a separate Java EE hosting application that acts as the proxy between the composite and Oracle Enterprise Scheduler. This hosting application can either be created in a one-to-one association with one Oracle Enterprise Scheduler application for each composite deployed, or multiple composites can share a single Java EE hosting application. The Java EE hosting application contains all the desired Oracle Enterprise Scheduler artifacts.

10.2.3 Setting Web Service Addressing Headers for getCompletionStatus() Operation

As shown in the ESSWebservice WSDL, if clients want to be notified asynchronously on job completion they can invoke the `getCompletionStatus()` operation. Upon job completion, Oracle Enterprise Scheduler will invoke the callback operation `onJobCompletion()` following ws-addressing where ESSWebservice captures the caller's address in the incoming call. Clients should be capable of receiving the callback at any arbitrary time in future. Such a callback depends entirely upon the time required to complete the job. This is similar to the Oracle Enterprise Scheduler functionality for invoking a client's listener (that implements Oracle Enterprise Scheduler EventListener contract) upon job completion.

When you use `getCompletionStatus()` clients must include certain required web service addressing headers (in particular the `wsa:MessageID` and `wsa:ReplyTo` headers). This allows the Oracle Enterprise Scheduler runtime to asynchronously notify the job completion status be sent to the correct `ReplyTo` address. When you use `getCompletionStatus()` from a BPEL process the SOA runtime automatically adds the required headers. When using `getCompletionStatus()` programatically on the client side, using the web service proxies, then the web service client must set these addressing headers.

10.2.4 Limitations for ESSWebservice

ESSWebservice does not support the following Oracle Enterprise Scheduler features:

- Ad hoc Request Submission: ESSWebservice does not support ad hoc job request submission (ad hoc request submission is available using the EJB APIs). Therefore any job that is submitted using the ESSWebservice must have its corresponding definition, including a job type and job definition along with the schedule definitions created as metadata objects in the associated proxy application. The web service operation can then refer to such metadata objects using their identifier arguments as specified in the WSDL.
- Query API: ESSWebservice does not expose the query APIs. Web service clients do not need to obtain the query information for Oracle Enterprise Scheduler requests. ESSWebservice web service clients do not provide generic monitoring and managing functionality that would require the use of query APIs.

10.2.5 ESSWebservice Implementation

The Oracle Enterprise Scheduler functionality is exposed as web service using an interface (SEI) annotated with the JAX-WS annotations. The implementation of this (SEI) web service invokes the common Oracle Enterprise Scheduler implementation layer. The ESSWebservice is exposed in Document/Literal/Wrapped mode for maximum interoperability.

Some of the data types used in ESSWebservice are not suitable to be used in web service directly. Such data types cannot be readily converted into corresponding XML representation. Therefore the Oracle Enterprise Scheduler web service layer defines wrapper classes around these data types that are exposed in the ESSWebservice, and visible in the WSDL. Otherwise in general, the web service layer reuses the existing data types where possible.

10.3 ESSWebservice WSDL File

When Oracle Enterprise Scheduler is installed and running, you can obtain the WSDL definition file from the web services page at the following type of URL:

`http://`*`host`*`:`*`port`*`/ess/esswebservice?WSDL`

For example,

`http://system1:7001/ess/esswebservice?WSDL`

10.4 Use Case Using Scheduler ESSWebservice from a BPEL Process

The following sections show use of ESSWebService from a BPEL process; in the BPEL process you use ESSWebService to submit a job request. The use case demonstrates one path for using Oracle Enterprise Scheduler for BPEL and SOA users. Experienced SOA users and designers may have other ideas for how work with Oracle Enterprise Scheduler using the web service. To submit an Oracle Enterprise Scheduler job request from a BPEL process, you need to deploy an application that provides the required Oracle Enterprise Scheduler artifacts. For this use case you can deploy the `EssDemoApp` described in Chapter 3, "Use Case Oracle Enterprise Scheduler Sample Application."

10.5 Creating the ESSWebService Application and a SOA Project

Using Oracle JDeveloper you create an application and the projects within the application that contain the code and support files for the application. To create the ESSWebService sample application, you do the following:

- Create an application and an SOA project in Oracle JDeveloper
- Configure the SOA project in Oracle JDeveloper

10.5.1 How to Create the ESSWebService Application and Project

To work with Oracle Enterprise Scheduler you first create an application and an SOA project in Oracle JDeveloper.

To create EssWebApplication:

1. Click the **New...** icon.
2. In the New Gallery, in the navigator, expand **General** and select **Applications**.
3. In the **Items** area select **SOA Application**.

4. Click **OK**.
5. Use the Name your application window to enter the name and location for the new application and to specify the application template.
   a. In the **Application Name** field, enter an application name. For this sample application, enter `EssWebApplication`.
   b. In the **Directory** field, accept the default.
   c. Enter an application package prefix or accept the default, no prefix.

      The prefix, followed by a period, applies to objects created in the initial project of an application.
   d. Click **Next**.
6. In the Name your project dialog select SOA project options.
   a. In the **Project Name** field, enter a project name or accept the default, `Project1`.
   b. On the Project Technologies tab, the **Selected** shuttle should show **SOA**.
   c. Click **Finish**. This creates the EssWebApplication that contains an SOA project.

10.6 Creating the ESSWebService Reference

In the SOA composite application you need to add the ESSWebservice reference to make the web service available for a partner link in the SOA composite application.

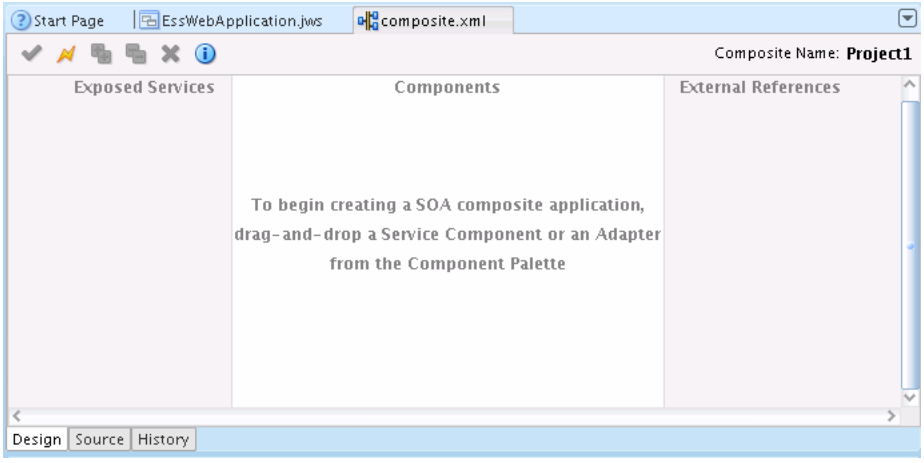
10.6.1 How to Add the ESSWebService Partner Link

You need to add the ESSWebService partner link to the SOA composite application.

To add the Oracle Enterprise Scheduler web service as a partner link:

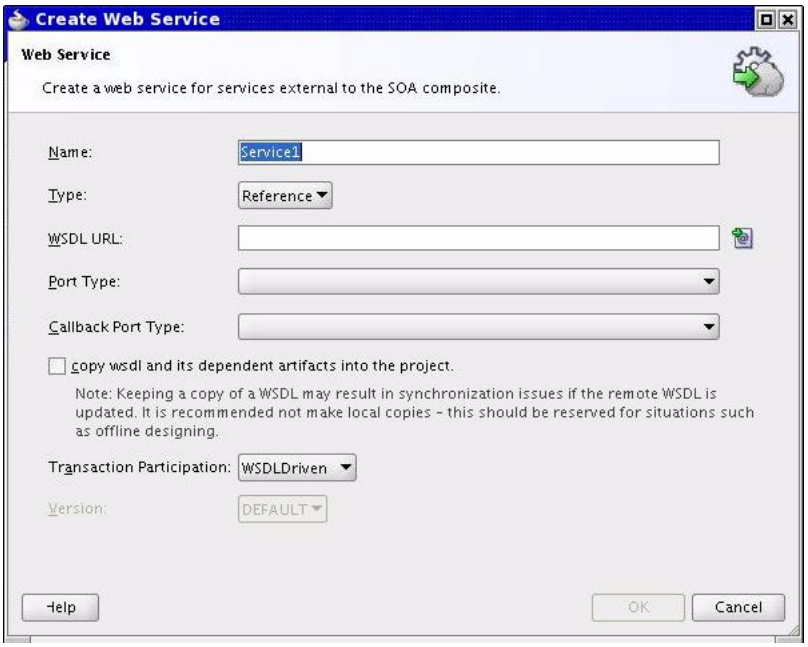
1. In the Application Navigator open the ESSWebApplication and expand **Project1** and then expand **SOA Content**.
2. In the Application Navigator select **composite.xml**.
3. Right-click and from the dropdown list select **Open**. This displays the composite as shown in Figure 10–1.

Figure 10–1 EssWebService Application composite.xml



4. In the Component Palette from the SOA dropdown list, in the Service Adapters area select **Web Service**.
5. Drag-and-drop the web service icon to the External References lane in composite.xml. This displays the Create Web Service window, as shown in Figure 10–2.

Figure 10–2 Create Web Service Dialog

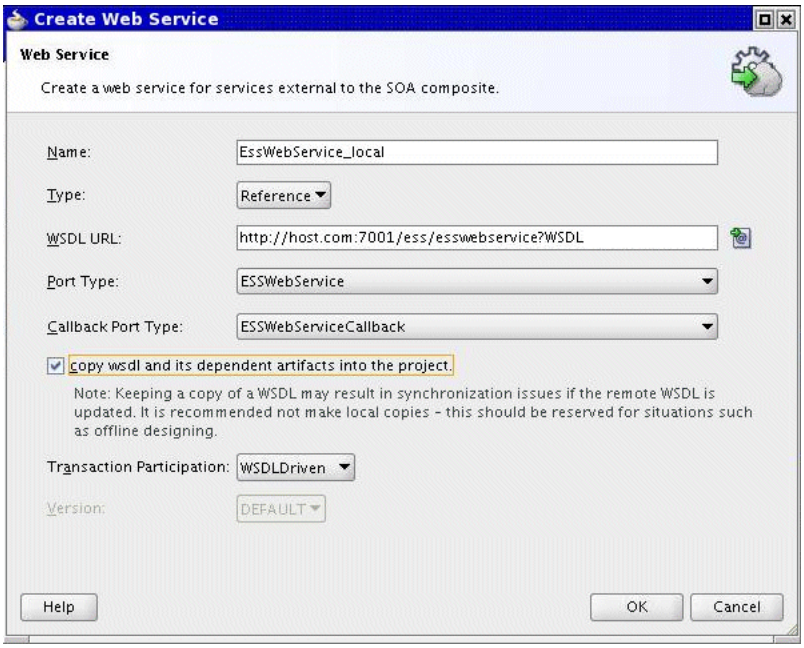


6. In the **Name** field, enter a service name, or accept the default name.
7. In the **Type** field, from the dropdown list select **Reference**.
8. In the **WSDL URL** text field enter the value for the **WSDL URL** manually, for example:

   `http://host:port/ess/esswebservice?WSDL`

9. In the SOA Resource Lookup dialog, click **OK**.
10. In the Create Web Service dialog, in the **Port Type** field, from the dropdown list select **ESSWebService**.
11. In the Create Web Service dialog, in the **Callback Port Type** select **ESSWebServiceCallback** from the dropdown list, as shown in Figure 10–3.

Figure 10–3 Create Web Service with ESSWebService WSDL



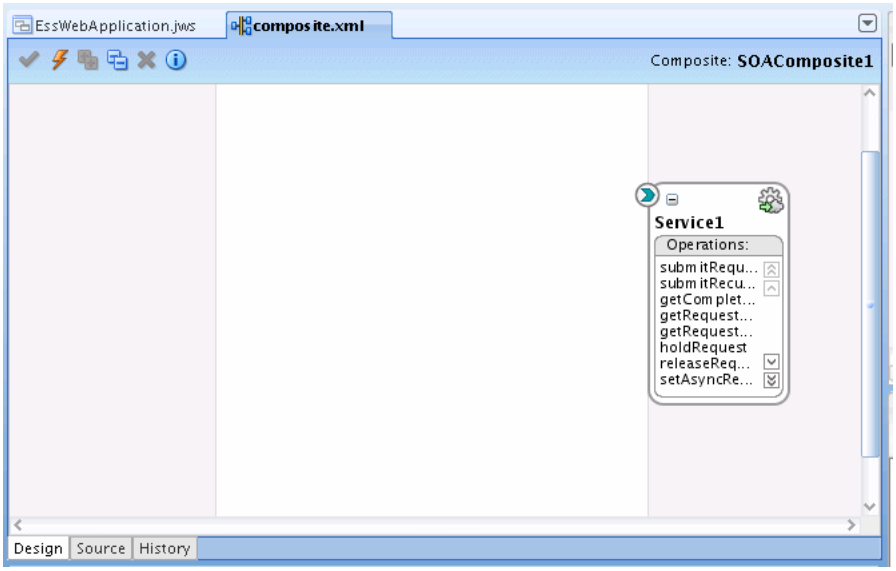
Select the checkbox **Copy WSDL and its dependent artifacts into the project**. This allows the local copy of the Oracle Enterprise Scheduler abstract WSDL and `ESSTypes.xsd` files to be moved into the SOA composite project.

Note: Keeping a local copy of a WSDL file may result in synchronization issues if the remote WSDL file is updated. Making a local copy of the remote WSDL file is therefore not recommended. However, doing so may be useful for certain scenarios such as offline designing.

---

12. Click **OK**. Now the External References lane in composite.xml displays the new web service, as shown in Figure 10–4.

Figure 10–4 Composite.xml with ESSWebService External Reference



10.7 Adding the BPEL Process to Call the ESSWebService

Now you need to add a BPEL Process to call the ESSWebService operations.

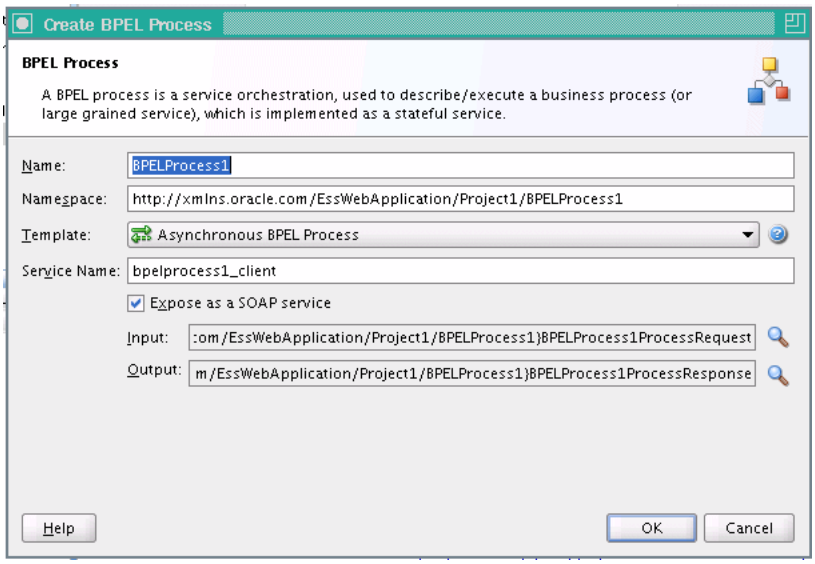
10.7.1 How to Add a BPEL Process to Call the ESSWebService

You need to add a BPEL process to use the ESSWebService.

To add a BPEL process to use the ESSWebService:

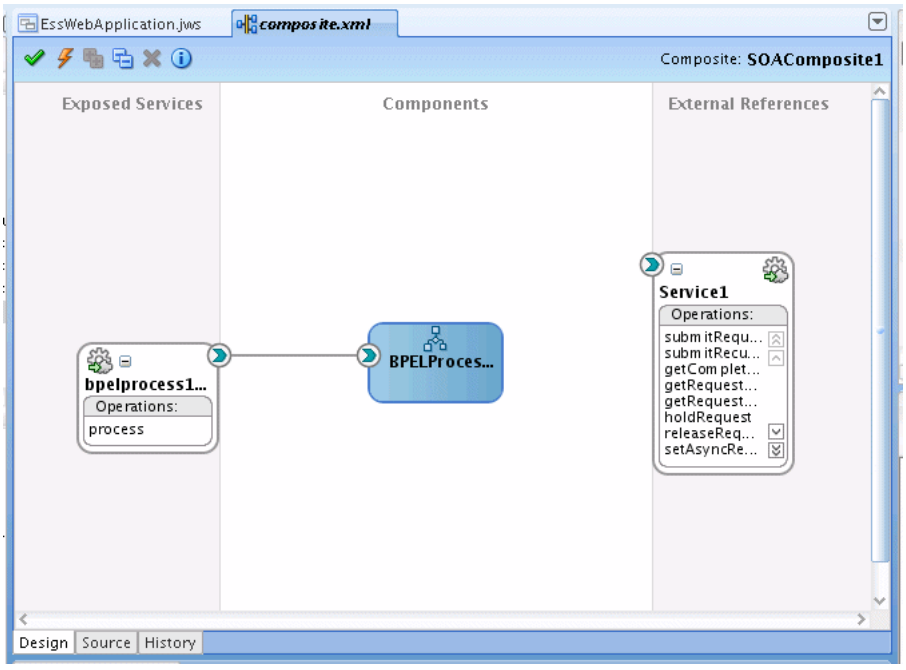
1. In the Application Navigator, in Project1 select **composite.xml**.
2. In the Component Palette, from the SOA dropdown list in the Service Components area select **BPEL Process**.
3. Drag-and-drop a BPEL process to the components swim lane. This displays the Create BPEL Process dialog, as shown in Figure 10–5.

Figure 10–5 Create BPEL Process Dialog for New BPEL Process



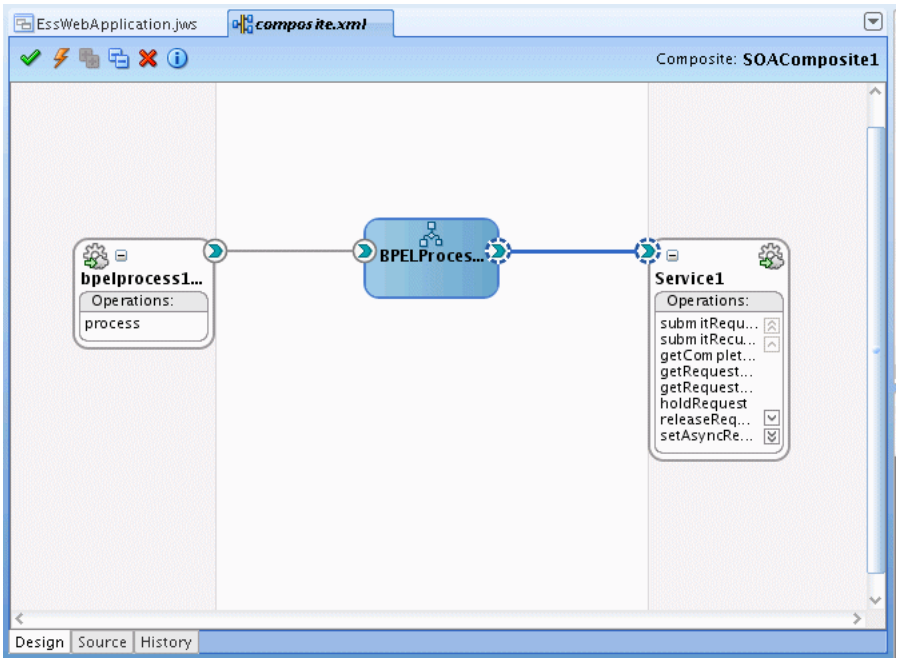
4. Click **OK**. This adds the BPEL process to composite.xml, as shown in Figure 10–6.

Figure 10–6 Adding a BPEL Process to the SOA Composite Application



5. In composite.xml, select BPELProcess1 and then select and drag the right arrow to create a reference to Service1, as shown in Figure 10–7.

Figure 10–7 Adding A Reference to the Oracle Enterprise Scheduler Web Service in composite.xml



6. Click the **Save All** icon to save the project files.

10.7.2 Copy Types Into BPEL Process Schema

You need to change the schema of the BPEL process by opening up the corresponding XSD file in the xsd folder under the project. This step is a shortcut for the demonstration purposes for this sample application. In your own application, you would use the schema types required for the ESSWebservice operations. This allows the clients of the BPEL process, for this example a simplified test case, to provide all the necessary inputs (this is required because clients are based on BPEL process schema). This step allows you to map, or assign inputs for the web service. This step is only required to correctly generate the sample application. In real scenarios the BPEL process designer is responsible for defining or supplying the input schema, and mapping this to the web service inputs.

> **Note:** The steps outlined require manual changes, depending on the BPEL process you are working with and the particular naming you are using for your BPEL process. You can find the types that are required for ESSWebService operations in the ESSWebService WSDL file. It is also possible to individually add these types to the schema.

To update the BPEL process schema:

1. In the Application Navigator, in Project1 expand the **SOA Content** folder and expand the **xsd** folder.
2. In the **xsd** folder, double-click the **BPELProcess1.xsd** file.
3. Select the **Source** tab.
4. Copy the EssWebService types so that the schema includes the contents shown in Example 10–1.

The `ESSTypes.xsd` file and other WSDL artifacts exposed by the Oracle Enterprise Scheduler web service are imported into the composite and renamed `esswebservice_XSD_<XSD file name>.xsd`.

Note: The schema shown in Example 10–1 includes the application and project name. If you change the application name or the project name for this example, you also need to update the schema `targetNamespace` and `xmlns:tns` elements to reflect the names that you use.

---

5. In the `BPELProcess1.xsd` file, refer to the artifacts created in Section 10.6, "Creating the ESSWebService Reference" that have been imported into the composite. The directory path should be relative the `BPELProcess1.xsd` file. Example 10–1 shows the composite schema file with a reference to the web service artifacts.

Example 10–1 BPEL XSD Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema attributeFormDefault="unqualified" elementFormDefault="qualified"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance"
        xmlns:ns1="http://xmlns.oracle.com/scheduler/types"
        targetNamespace="http://xmlns.oracle.com/
                         EssWebApplication/Project1/BPELProcess1"
        xmlns:tns="http://xmlns.oracle.com/
                   EssWebApplication/Project1/BPELProcess1"
        xmlns="http://www.w3.org/2001/XMLSchema">

<import namespace="http://xmlns.oracle.com/scheduler/types"
  schemaLocation="../esswebservice_XSD_ESSTypes.xsd" />

  <element name="process">
    <complexType>
      <xs:sequence>
        <xs:element name="description" type="xs:string"/>
        <xs:element name="jobDefinitionId" type="ns1:metadataObjectId"/>
        <xs:element name="requestedStartTime" type="xs:dateTime"/>
        <xs:element name="application" type="xs:string"/>
        <xs:element name="requestParameters" type="ns1:requestParameters"/>
      </xs:sequence>
    </complexType>
  </element>
  <element name="processResponse">
    <complexType>
      <sequence>
        <element name="result" type="string"/>
        <element name="requestId" type="long"/>
        <element name="state" type="ns1:state"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

6. Click the **Save** icon.

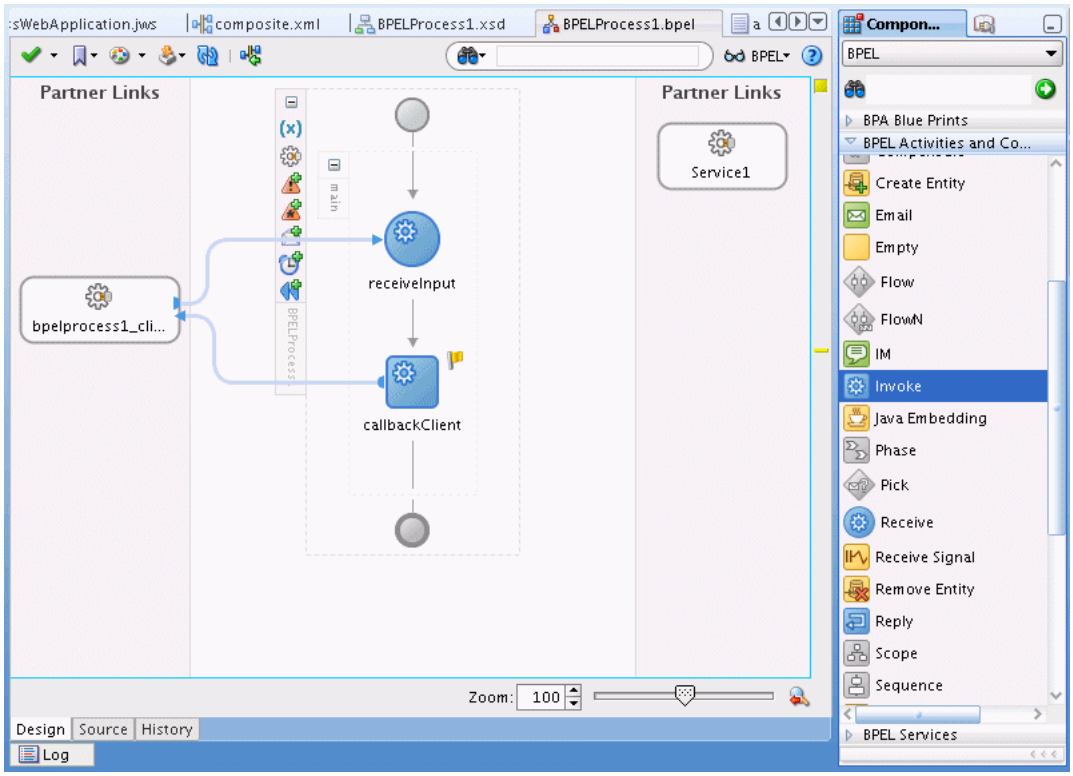
10.7.3 How to Invoke the ESSWebService submitRequest Operation

In the BPEL process you add an invoke activity to perform the Oracle Enterprise Scheduler web service `submitRequest()` operation. In this step you need to select the input and output for the Invoke Activity by associating values with the Input and Output variables.

To add the Invoke activity to submit the request using ESSWebService:

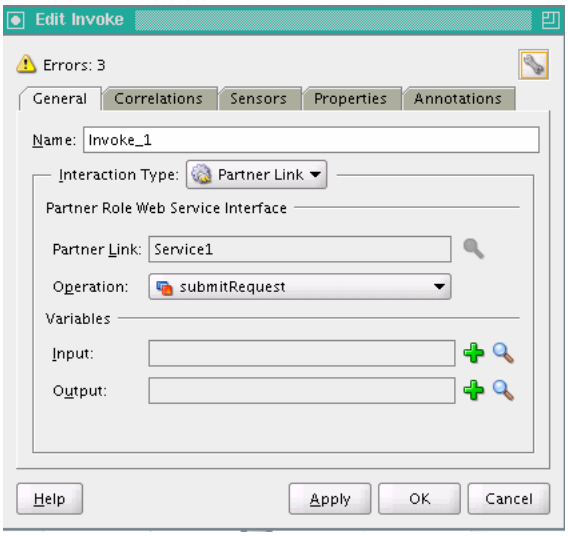
1. In the Application Navigator, in Project1 expand SOA Content and select the BPEL file. For example, select **BPELProcess1.bpel**. This displays the BPEL swim lane as shown in Figure 10–8.

Figure 10–8 BPEL Process Before Adding Invoke Activity for ESSWebService SubmitRequest



2. From the Component Palette, drag-and-drop an **Invoke** Activity and place the activity before callbackClient.
3. Link the invoke activity to the ESSWebService by selecting the right arrow and dragging it to the Partner Link Service1. This brings up the Edit Invoke dialog, as shown in Figure 10–9.

Figure 10–9 Edit Invoke Dialog for BPEL Activity



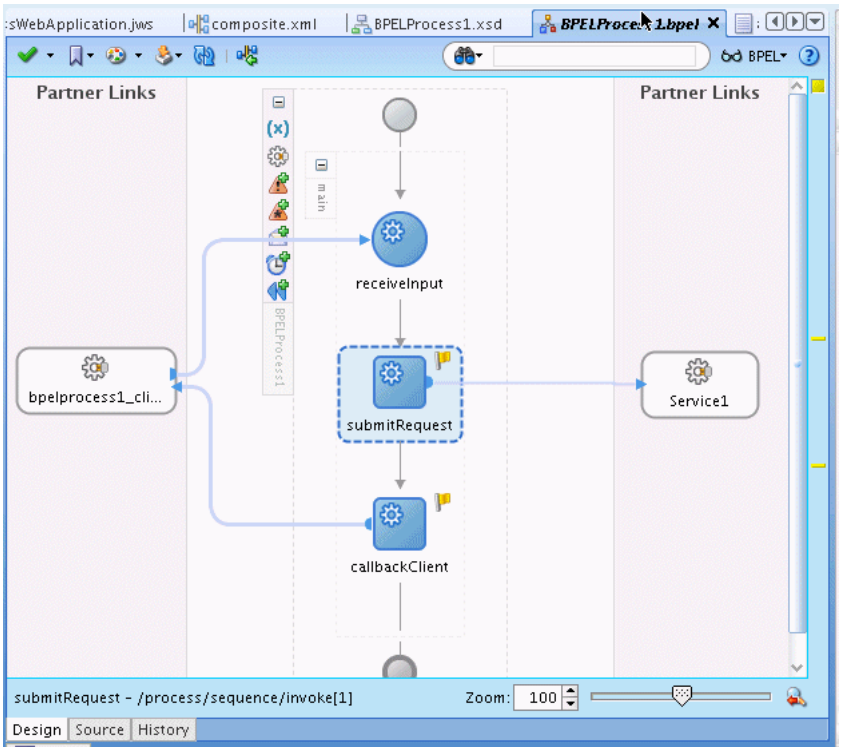
4. In the Edit Invoke dialog, in the Operation field, select **submitRequest**.
5. In the Variables field, click the **Add** icon next to the **Input** field.

   The Create Variable dialog displays. Accept the default value and click **OK**.
6. In the Edit Invoke dialog, click the **Add** icon next to the **Output** field.

   The Create Variable dialog displays. Accept the default value and click **OK**.

   The new invoke link to **Service1** displays.
7. Select the Invoke activity and double-click the name `Invoke_1` to select the text entry field. In the text entry field enter `submitRequest`, as shown in Figure 10–10.

Figure 10–10 Adding the submitRequest Invoke Activity



10.7.4 Assign Required Input Parameters for Request Submission

You add an Assign activity and then assign inputs from the BPEL process to the submitRequest Invoke activity.

> **Note:** In most cases, the input payload of the BPEL process will not directly match the input payload of the submit Request web service. Coaxing into use of CopyList will only work in the scenarios where there is a one to one mapping of the input payload to the submit Request.

For the mapping for an Assign activity with a Copy operation, the arguments correspond to the input parameters for Oracle Enterprise Scheduler submitRequest, as shown in Table 10–2. If your BPEL schema differs from the submitRequest message type, use Table 10–2 as a guide for how to populate the values manually with the Assign activity Copy operation.

Table 10–2 Submit Request Web Service Arguments for BPEL Assign Activity Mapping

Argument	Description
Description	Context for the ad hoc submission of this job, such as the 'Order Import'.
Application	The application name can be the deployment name of the hosting Oracle Enterprise Scheduler application or it can be a logical application name.

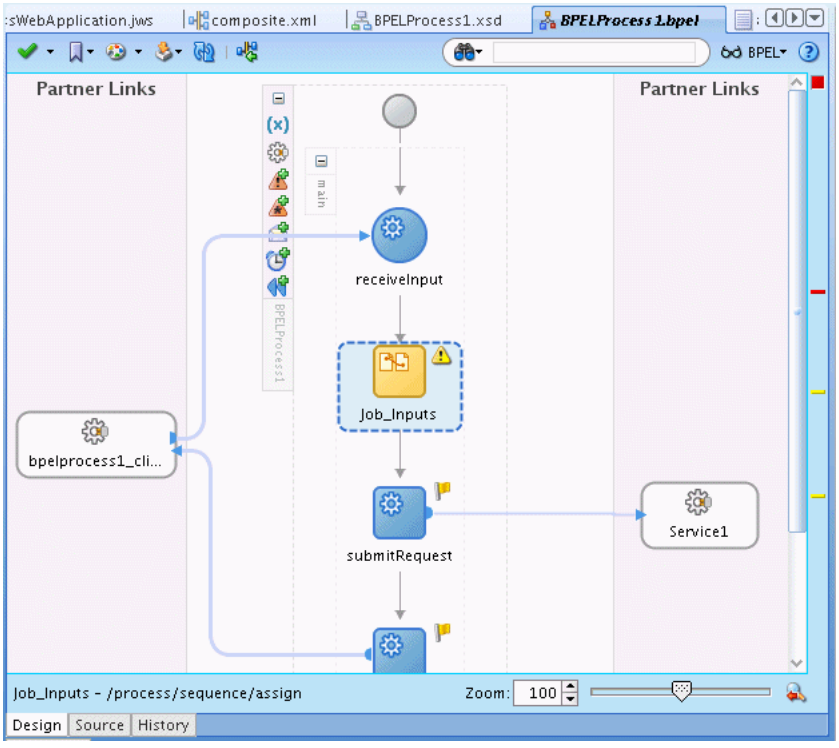
Table 10–2 (Cont.) Submit Request Web Service Arguments for BPEL Assign Activity

Argument	Description
JobDefinitionId	▪ name: The name of the Oracle Enterprise Scheduler job ▪ package: The name of the path containing the Oracle Enterprise Scheduler job ▪ type: 'JOB_DEFINITION'
parameter(s)	dataType: Value type for this parameter (STRING, INTEGER, LONG, BOOLEAN, DATETIME) name: String containing the name of the parameter defined in the Oracle Enterprise Scheduler job definition. scope: String containing the named scope for this parameter - used only for job sets. value: Element containing the parameter's value

To add an assign activity:

1. Drag-and-drop an Assign activity from the BPEL Activities area in the Component Palette to just before the Invoke Activity named **submitRequest**.
2. Select the Assign activity and double-click the name Assign_1 to enter new text. In the text entry box enter `Job_Inputs`, as shown in Figure 10–11.

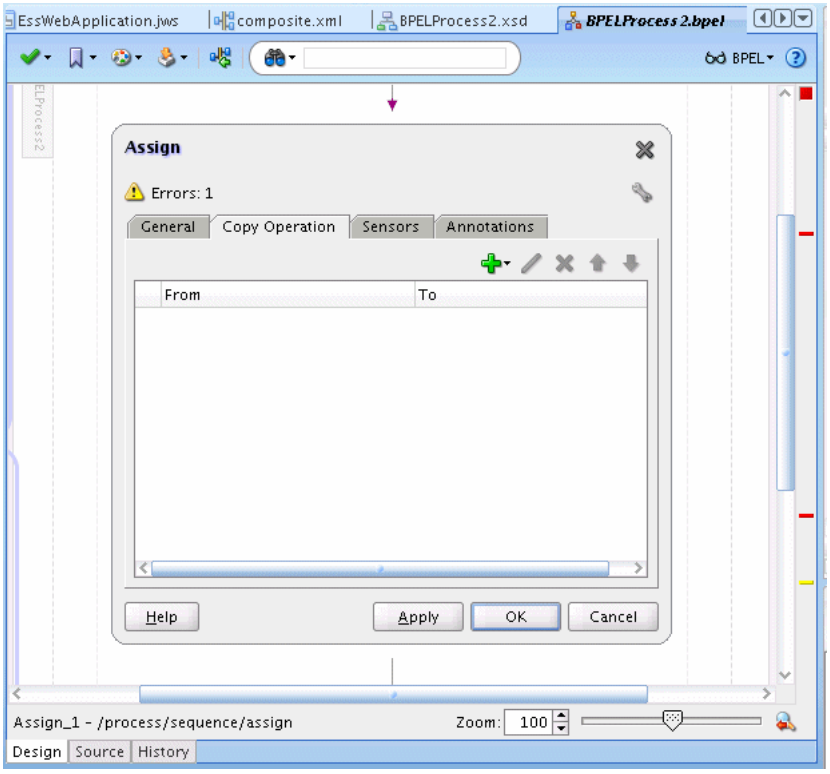
Figure 10–11 Adding an Assign Activity to BPEL



Add Copy for description JobDefinitionID requestedStartTime application:

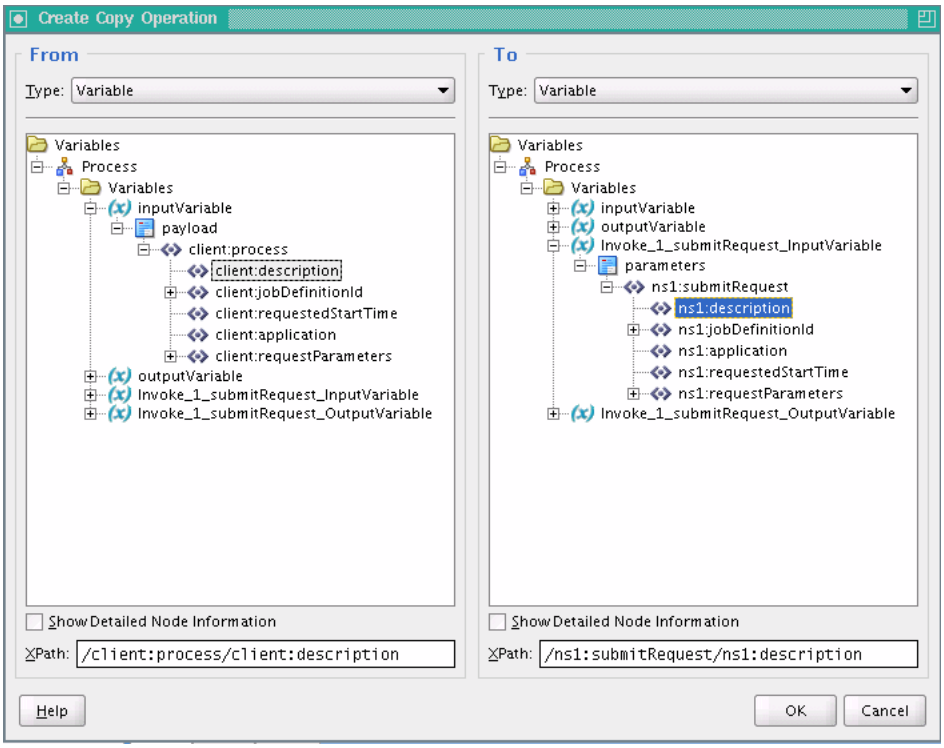
1. Double click the new Assign activity named `Job_Inputs` to show the Assign page with the Copy Operation tab, as shown in Figure 10–12.

Figure 10–12 Copy Operation for BPEL Assign Activity



2. Click the **Add** icon and from the dropdown list select **Copy Operation**, to add copy operations for variables. This displays the Create Copy Operation dialog.
3. In the Create Copy Operation dialog, expand and then navigate to select a copy operation for each input parameter (you only use a copy operation for `description`, `jobDefinitionID`, `requestedStartTime`, and `application`). This copies the input parameters to `Invoke_1_submitRequest_InputVariable` parameters for the invoke activity. Figure 10–13 shows one of these copy operations.

Figure 10–13 Copy Operation for Description Parameter for submitRequest

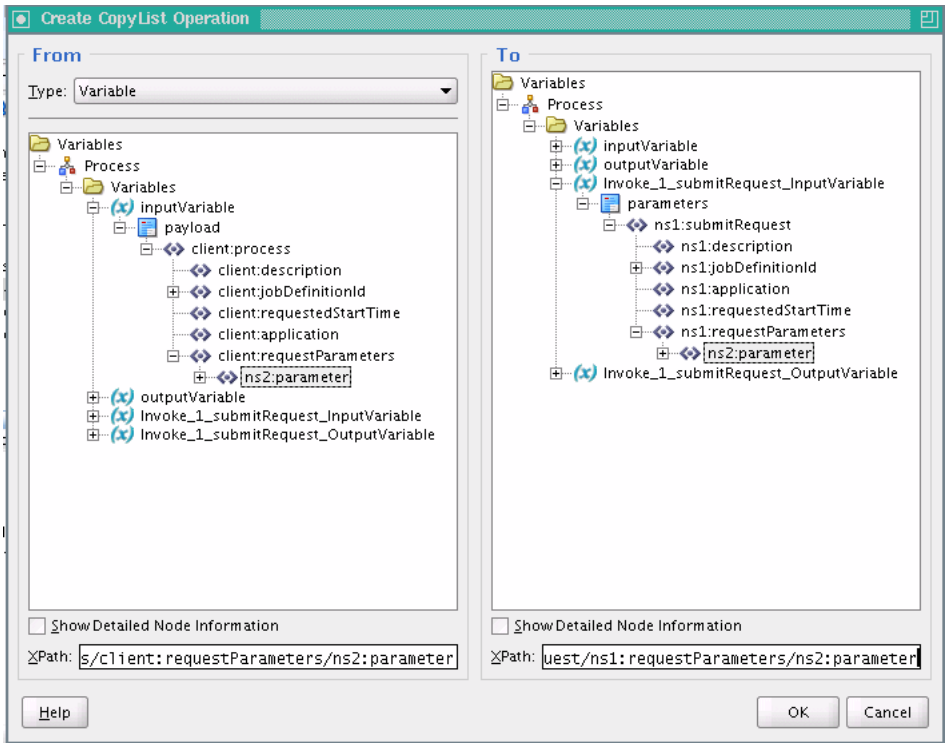


4. Click OK to add the copy operation for `description`.
5. In a similar manner, perform additional copy operations for the `jobDefintionID`, `requestedStartTime`, and `application` parameters.

To add a copy list for RequestParameters:

1. Double click the Assign activity named `Job_Inputs` to show the Assign page with the Copy Operation tab.
2. Click the **Add** icon and from the dropdown list select **CopyList Operation...**, to add CopyList operations for the requestParameters. This displays the Create CopyList Operation dialog.
3. In the Create CopyList Operation dialog, expand and then navigate to select a copylist operation for `requestParameters`. To do this you navigate and select the **parameter** element, as shown in Figure 10–14.

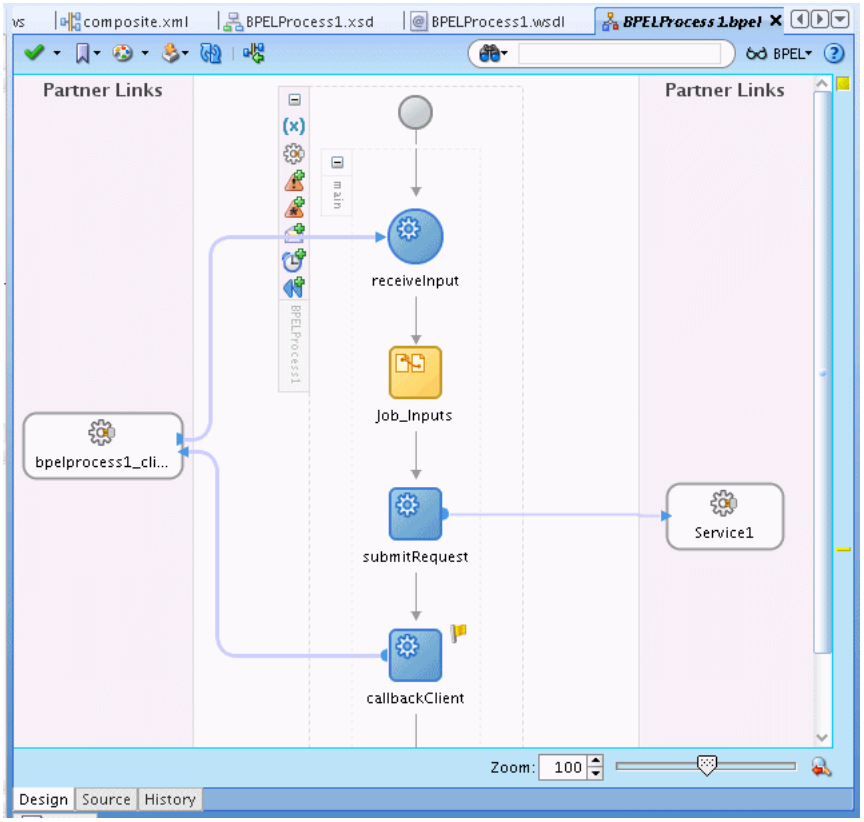
Figure 10–14 CopyList Operation for Request Parameters



4. In the Create CopyList Operation dialog, click **OK**.
5. In the Assign activity, click **OK**.

Figure 10–15 shows the BPEL Design page.

Figure 10–15 BPEL with Job_Inputs Add Activity and submitRequest Invoke

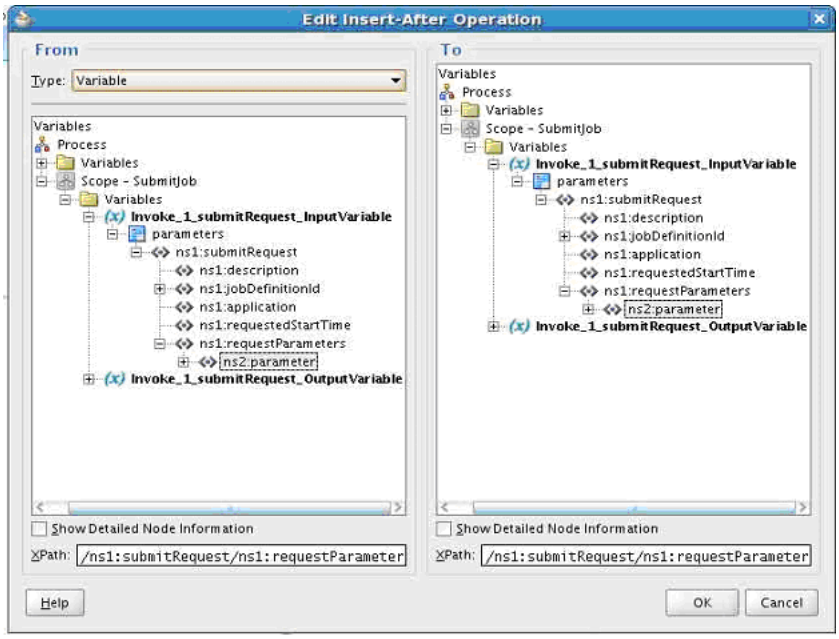


When BPEL Element Does Not Have Same Type as Oracle Enterprise Scheduler web service:

If your BPEL payload is not the same element type as that of the Oracle Enterprise Scheduler web service and you need to assign values to one or more job parameters, you can use the following approach.

1. Populate the first parameter element using copy operations, as done in previous steps.
2. Add or clone additional parameter elements using the Insert-After, as shown in Figure 10–16.

Figure 10–16 Using Insert-After to Clone Parameters



3. Populate the additional parameter elements using XPath array subscripting.
4. This action effectively copies the entire parameter element along with all sub-element values and appends it to the end of the XML array. In order to populate the values of the second job parameter, add additional copy operations and modify the XPath expressions in the bottom right of the dialog to add the appropriate array subscript [n]. where 'n' is the # of the parameter. Note that all XML arrays start with 1, not 0.

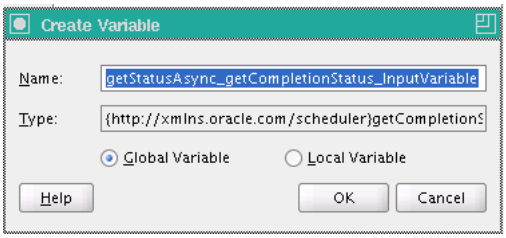
10.7.5 Invoke the getCompletionStatus Operation

Add another Invoke activity and link it to Service1 to invoke the ESSWebService getCompletionStatus operation.

To add the Invoke activity for the getCompletionStatus operation:

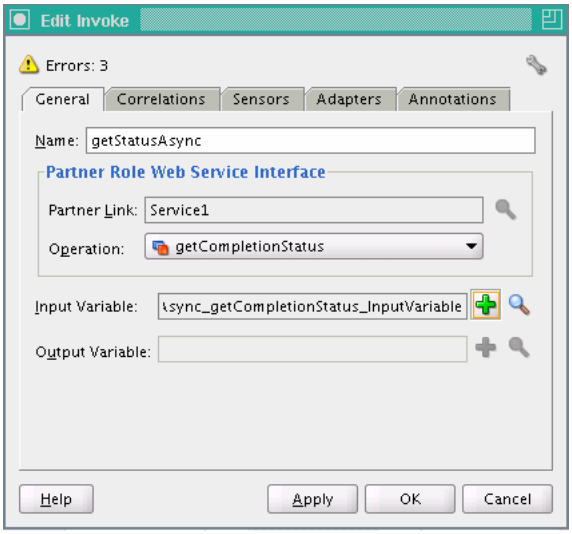
1. From the Component Palette, drag-and-drop an **Invoke** activity and drop it after submitRequest and before callbackClient.
2. In the new Invoke activity, select the text entry area with the name Invoke_1, and enter the name, `getStatusAsync`.
3. Link the invoke activity to Service1 by selecting the right arrow and dragging it to the Partner Link Service1. This displays the Edit Invoke dialog.
4. In the Edit Invoke dialog for getStatusAsync, in the **Operation** field, from the dropdown list select **getCompletionStatus**.
5. In the **Input Variable** field select the **Add** icon. This displays the create variable dialog, as shown in Figure 10–17.

Figure 10–17 Create Variable Window for getStatusAsync



6. In the Create Variable dialog, click **OK**. This displays the Edit Invoke dialog, as shown in Figure 10–18.

Figure 10–18 Edit Invoke Window for getStatusAsync



7. In the Edit Invoke dialog, click **OK**. This displays the new Invoke Activity **getStatusAsync** and the link to Service1.

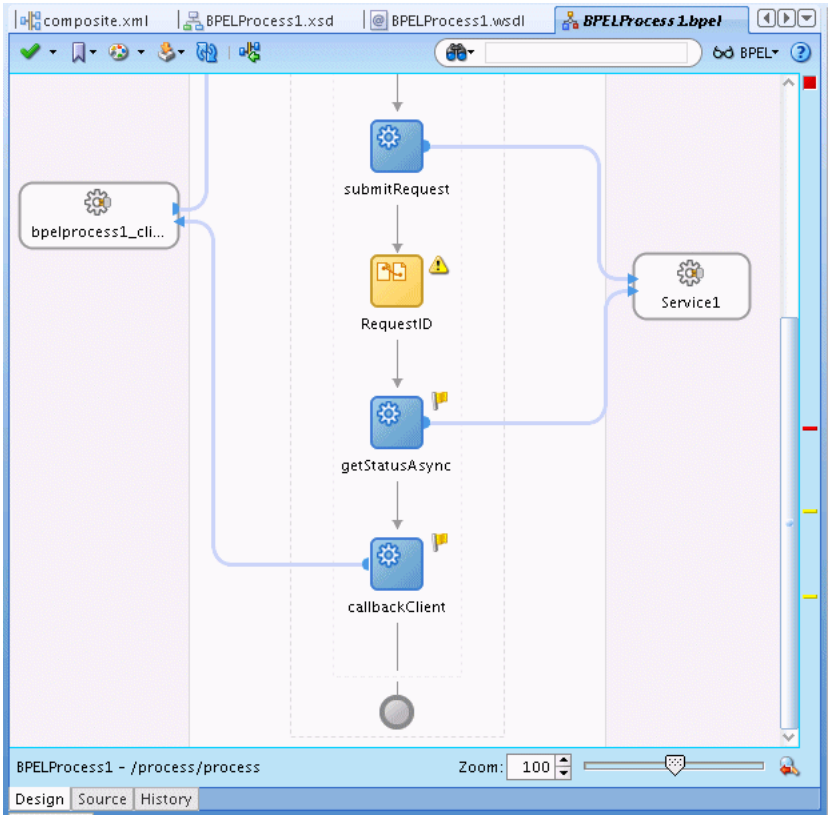
10.7.6 Assign Input to the getCompletionStatus Operation

Add a new Assign Activity after submitRequest to assign the RequestID and pass it to the getStatusAsync invoke activity.

To add the assign activity:

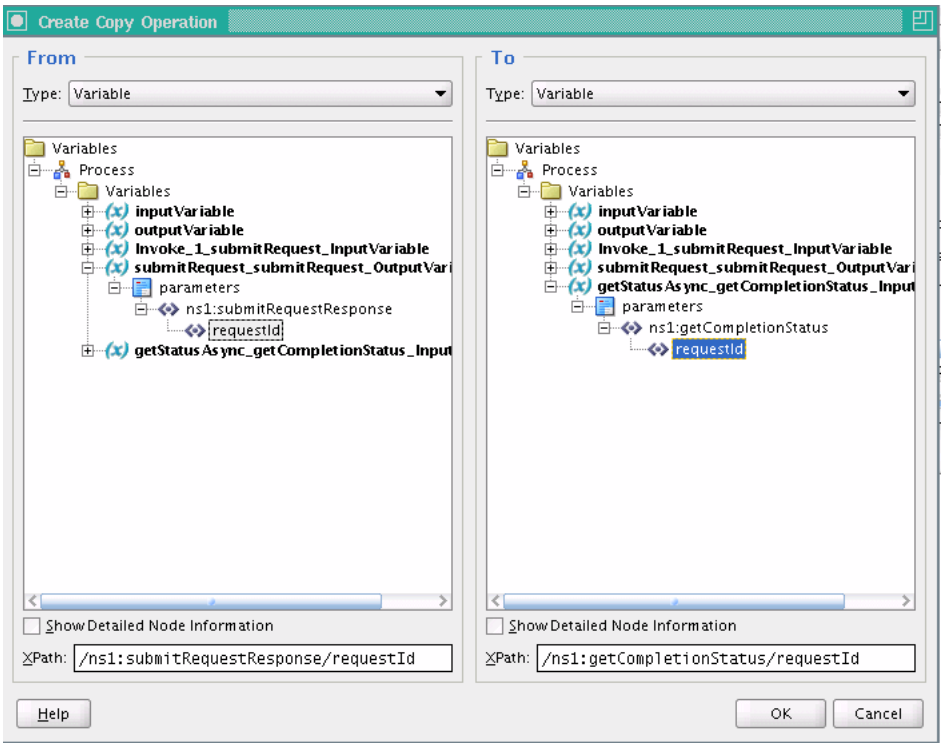
1. Drag-and-drop an Assign activity from the BPEL Activities area in the Component Palette to just after the Invoke Activity named **submitRequest** and before the Invoke Activity named **getStatusAsync**.
2. Select the Assign activity and double-click the name Assign_1 to select the text entry area. In the text entry area, enter `RequestID`. Figure 10–19 shows the Assign activity.

Figure 10–19 Adding RequestID Assign Activity



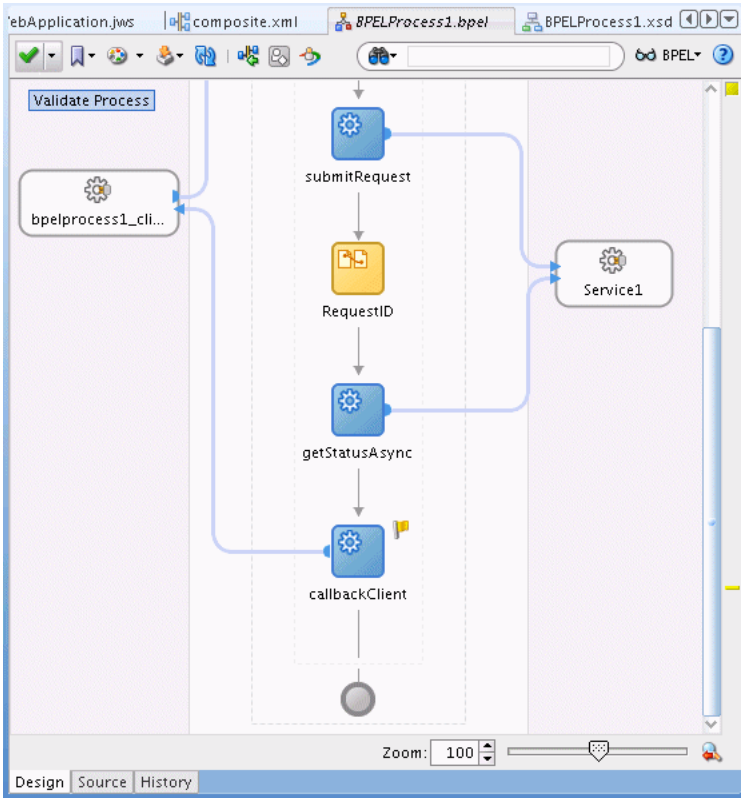
3. Double click the new Assign activity, **RequestID** to show the Assign page with the Copy Operation tab.
4. Click the **Add** icon and select **Copy Operation...** from the dropdown list.
5. In the From area expand **Invoke_1_submitRequest_OutputVariable** and select requestID. Map this in the **To** area to the `requestID` in `getStatusAsync_getCompletionStatus_InputVariable`, as shown in Figure 10–20.

Figure 10–20 Edit Copy Operation Window for Request ID Assign



6. On the Edit Copy Operation dialog, click **OK**.
7. On the Copy Operation dialog, click **OK**.
8. On the BPEL Design page, click **Validate Process**. This displays the BPEL, as shown in Figure 10–21.

Figure 10–21 BPEL with Request ID Assign Activity Added



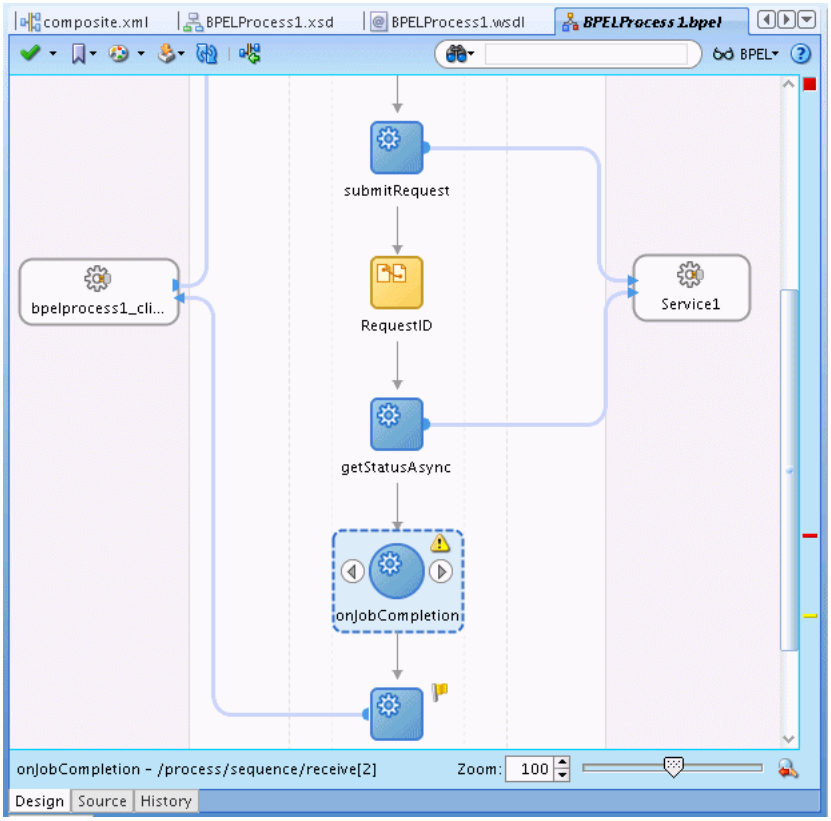
10.7.7 Receive the Job Completion Status

Add a Receive Activity and link it to the onJobCompletion ESSWEbService operation.

Add a receive activity:

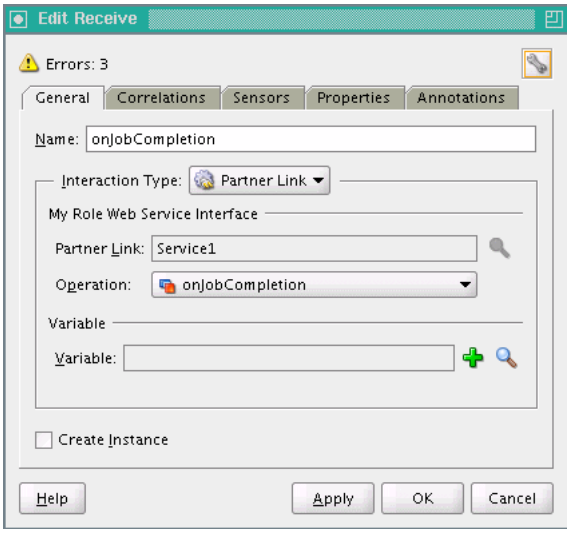
1. Drag-and-drop a Receive activity from the BPEL Activities area in the Component Palette to a position after the **getStatusAsync** Invoke activity and before the **callbackClient**.
2. Select the text entry area in the Receive Activity named `Receive_1` and enter `onJobCompletion`, as shown in Figure 10–22.

Figure 10–22 Adding Receive Activity to BPEL Process



3. Drag the right arrow from the receive activity **onJobCompletion** to Service 1. This displays the Edit Receive dialog, as shown in Figure 10–23.

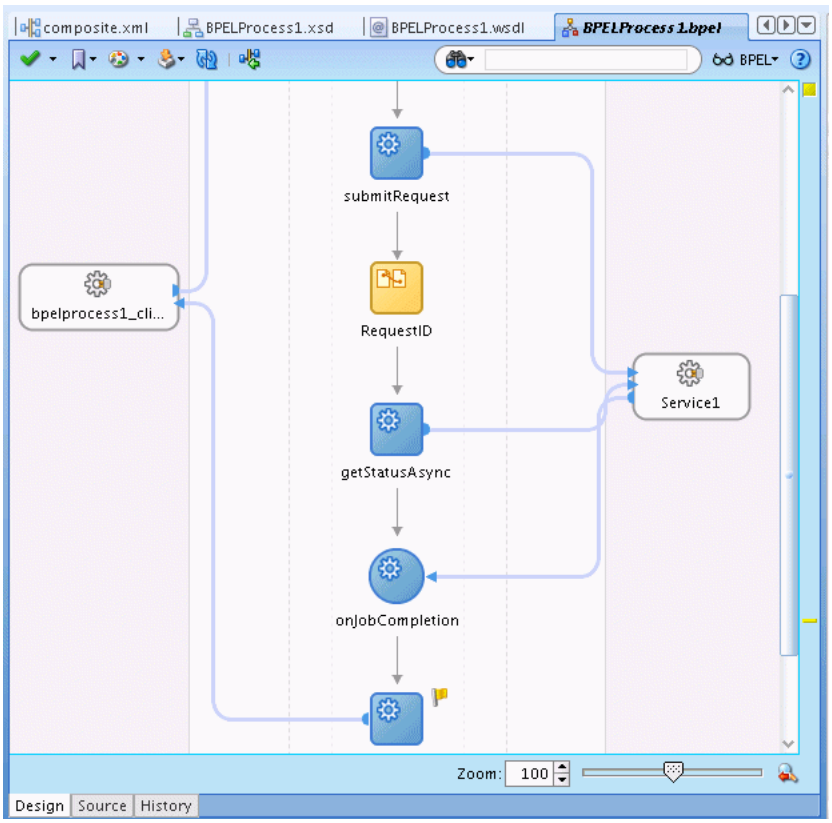
Figure 10–23 Edit Receive Window for onJobCompletion Receive Activity



4. In the Edit Receive dialog, in the **Operation** field from the dropdown list select **onJobCompletion**.
5. In the **Variable** field, click the **Add** icon. This displays the Create Variable dialog.

6. In the Create Variable dialog, click **OK**.
7. In the Edit Receive dialog, click **OK**. This adds an arrow from Service1 to the new Receive activity, **onJobCompletion** as shown in Figure 10–24.

Figure 10–24 Adding the onJobCompletion Receive Activity



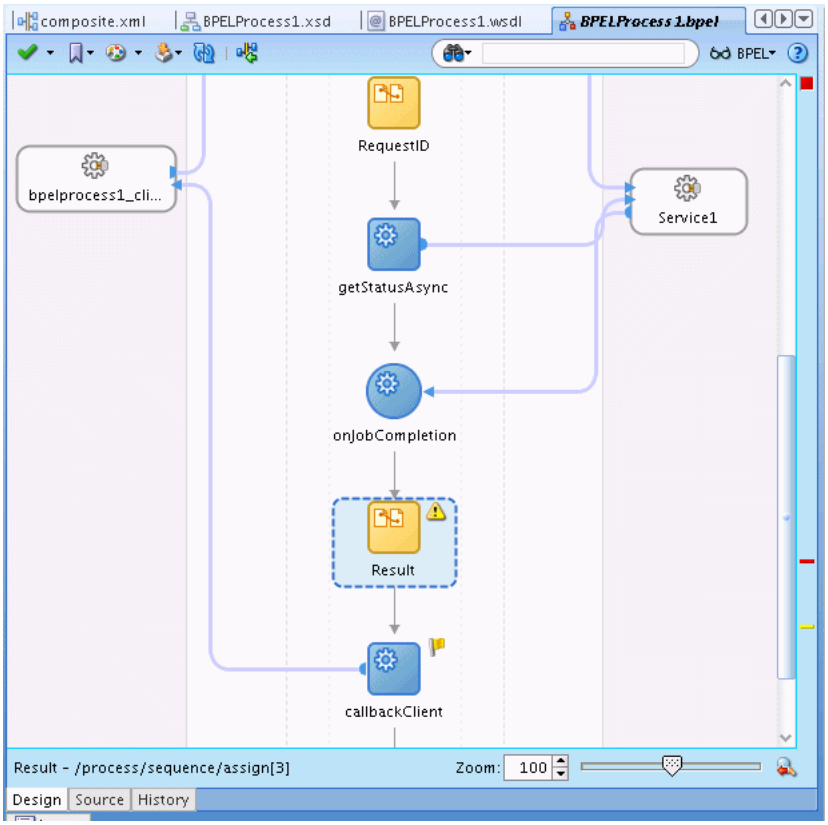
10.7.8 Return Result to Client

Add an Assign activity to copy the result output from **onJobCompletion** to the output for the client. Assign all the results from onJobCompletion to the callbackClient input variable.

To add the result assign activity:

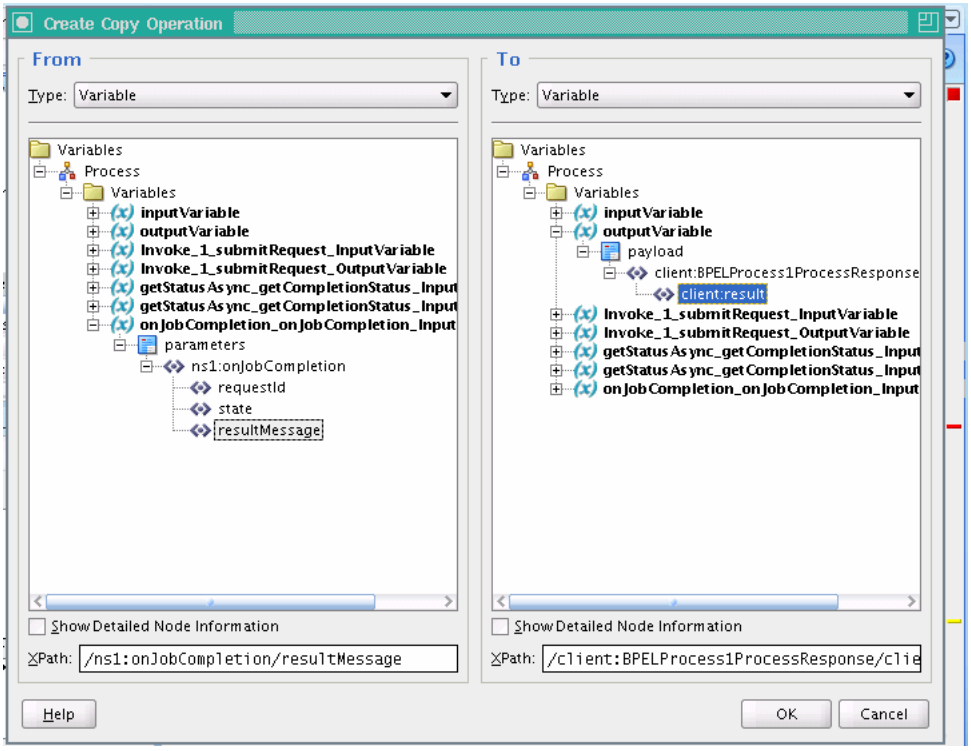
1. Drag-and-drop an Assign activity from the BPEL Activities area in the Component Palette to a position after the Receive activity **onJobCompletion** and before the callbackClient.
2. Select the Assign activity and double-click the name Assign_1 to enter new text. Enter the value `Result`, as shown in Figure 10–25.

Figure 10–25 Adding Assign Activity for Output to Client



3. Double click the new **Result** Assign activity to show the Assign page with the Copy Operation tab.
4. Click the **Add** icon and select **Copy Operation...** from the dropdown list.
5. Navigate to select the variables, for the **From** area for `onJobCompletion_onJobCompletion_InputVariable` and select **resultMessage**. In the **To** area, expand **outputVariable** and select **client:result**, shown in Figure 10–26.

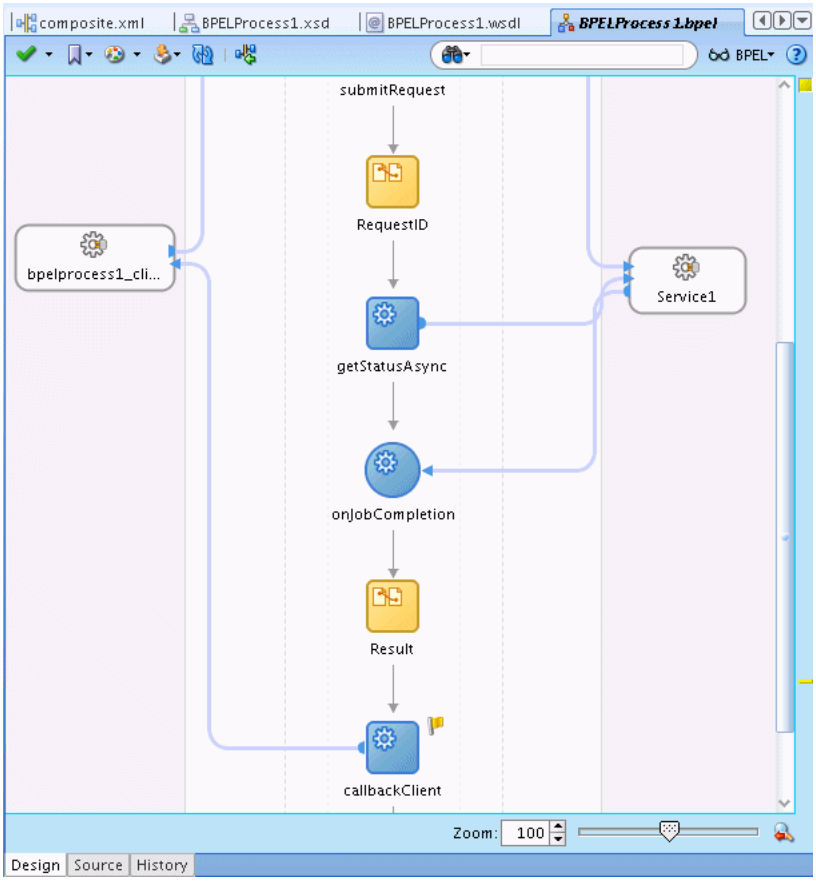
Figure 10–26 Create Copy Operation for Result



6. In the Create Copy Operation dialog, click **OK**.
7. In the Assign area, click **OK**.
8. Click **Validate Process**.

The final BPEL is shown in Figure 10–27.

Figure 10–27 Result Assign Activity with callbackClient Invoke Activity



10.8 Using Additional ESSWebService Operations

You can use other EssWebService operations, including:

- When you want to submit a request with an associated schedule, you use the `submitRecurringRequest` web service operation. For more information, see Section 10.8.1, "How to Invoke the ESSWebService submitRecurringRequest Operation."
- When you want to marshal arguments in the previous concurrent processing style into Oracle Enterprise Scheduler properties, you use the `setSubmitArgs` operation. This operation should be invoked prior to submitting a request. The key of each argument is `submit.argument#`, where `#` is the ordinal value of the argument, for example `submit.argument1="firstArg"` and `submit.argument2="secondArg"`. For more information, see Section 10.8.2, "How to Invoke the ESSWebService setSubmitArgs Operation."
- When you want to add a post-processing action to a step in a job set request, you use the `addPPAction` operation. This method is called prior to submitting the request. This operation provides support for action previously supported by `add_printer`, `add_notification` and `add_layout` in concurrent processing. The parameters to these legacy routines are passed as arguments to `addPPAction` in the order in which they were declared in the original concurrent processing routine. Section 10.8.3, "How to Invoke the ESSWebService addPPActions Operation"

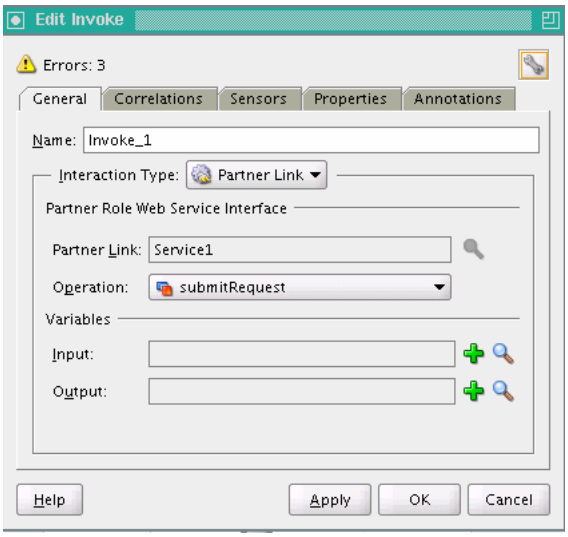
10.8.1 How to Invoke the ESSWebService submitRecurringRequest Operation

In the BPEL process you add an invoke activity to perform the Oracle Enterprise Scheduler web service `submitRecurringRequest()` operation. In this step you need to select the input and output for the Invoke Activity by associating values with the Input and Output variables. In order to submit jobs that repeat or will run at a later date that job must be submitted with an Oracle Enterprise Scheduler schedule which is constructed declaratively and stored in the metadata repository. Once the schedule has been defined, BPEL can submit jobs with that schedule through the `submitRecurringRequest()` operation.

To add the Invoke activity to submit the request using ESSWebService:

1. In the Application Navigator, in Project1 expand SOA Content and select the BPEL file. For example, select **BPELProcess1.bpel**. This displays the BPEL swim lane.
2. From the Component Palette, drag-and-drop an **Invoke** Activity and place the activity in the process. This activity populates the request submission payload and submits it to the Oracle Enterprise Scheduler web service.
3. Select the Invoke activity and double-click the name `Invoke_1` to select the text entry field. In the text entry field enter `submitRecurringRequest`.
4. Link the invoke activity to the ESSWebService by selecting the right arrow and dragging it to the Partner Link Service1. This brings up the Edit Invoke dialog, as shown in Figure 10–28.

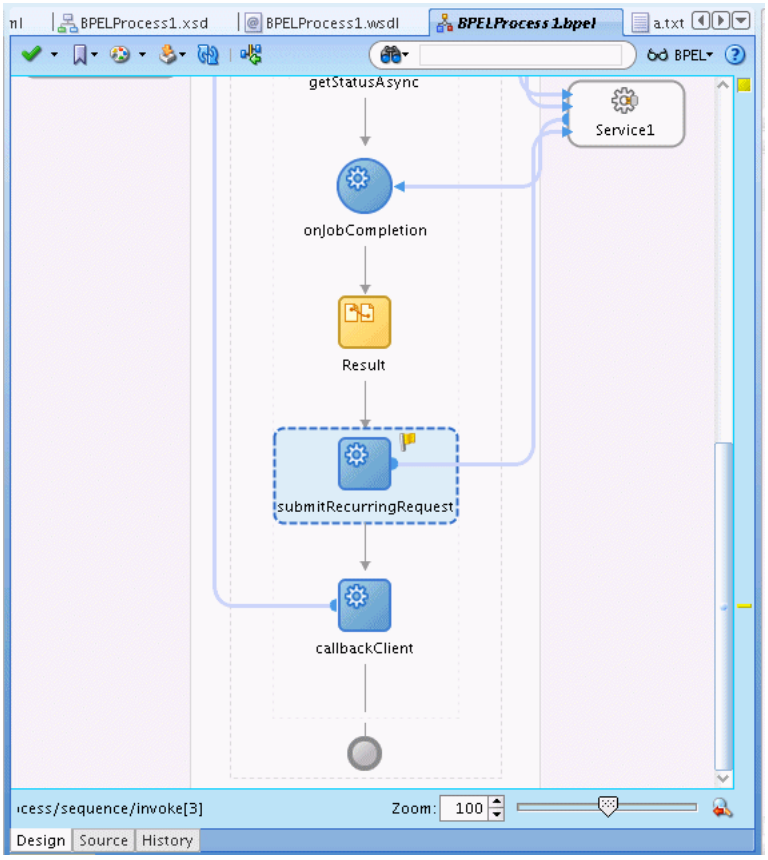
Figure 10–28 Edit Invoke Window for BPEL Activity



5. In the Edit Invoke dialog, in the Operation field, select **submitRecurringRequest**.
6. In the Edit Invoke dialog, in the **Input** field click the **Add** icon. This displays the Create Variable dialog and lets you create a scope-level variable to contain the request payload.
7. In the Create Variable dialog, click **OK**.
8. In the Edit Invoke dialog, in the **Output** field select the **Add** icon. This displays the Create Variable dialog and lets you create scope-level variable to contain the response payload.
9. In the Create Variable dialog, click **OK**.

10. In the Edit Invoke dialog, click **OK**. This displays the new invoke link to Service1, as shown in Figure 10–29.

Figure 10–29 Submitting a Request with a Schedule



To assign inputs for recurring request submission:

You add an Assign activity and then assign inputs from the BPEL process to the submitRecurringRequest Invoke activity. This allows you to populate the input variable with recurring request submission parameters.

> **Note:** In most cases, the input payload of the BPEL process will not directly match the input payload of the submit recurring request web service. Coaxing into use of CopyList will only work in the scenarios where there is a one to one mapping of the input payload to the submit Request.

For the mapping for an Assign activity with a Copy operation, the arguments correspond to the input parameters for Oracle Enterprise Scheduler submitRequest, as shown in Table 10–3. If your BPEL schema differs from the submitRequest message type, use Table 10–3 as a guide for how to populate the values manually with the Assign activity Copy operation.

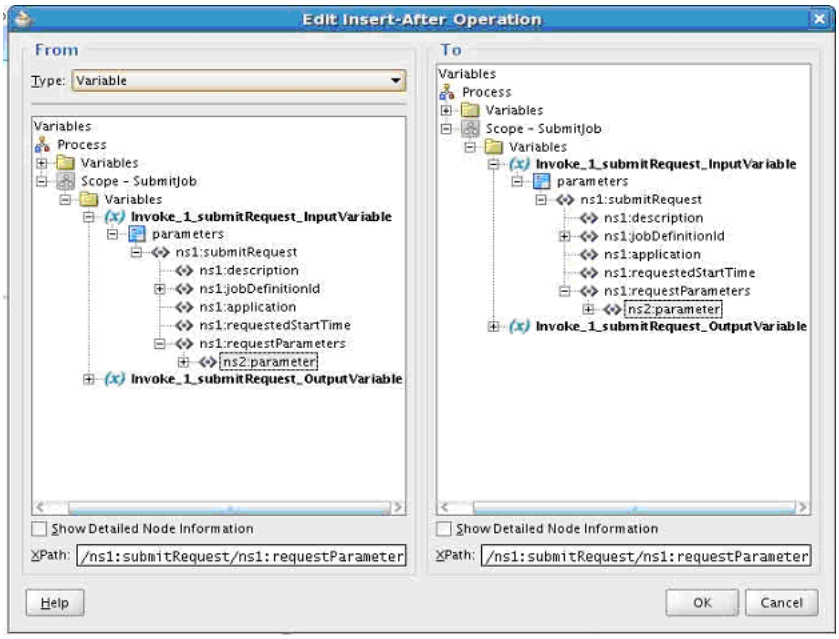
Table 10–3 Submit Recurring Request Web Service Arguments for BPEL Assign Activity Mapping

Argument	Description
Description	Context for the ad hoc submission of this job, such as the 'Order Import'.
Application	The "application" name can be the deployment name of the hosting Oracle Enterprise Scheduler application or it can be a logical application name.
JobDefinitionId	■ name: The name of the Oracle Enterprise Scheduler job ■ package: The name of the path containing the Oracle Enterprise Scheduler job ■ type: 'JOB_DEFINITION'
parameter(s)	dataType: Value type for this parameter (STRING, INTEGER, LONG, BOOLEAN, DATETIME) name: String containing the name of the parameter defined in the Oracle Enterprise Scheduler job definition. scope: String containing the named scope for this parameter - used only for job sets. value: Element containing the parameter's value
scheduleID	■ name: String containing the name of the schedule metadata file ■ packageName: String containing the name of the MDS package containing the metadata file (sans the 'Schedule' path) ■ type: 'SCHEDULE_DEFINITION''

It is possible to define multiple parameters to be passed to the Oracle Enterprise Scheduler job. When adding additional parameters to the Oracle Enterprise Scheduler service payload in BPEL, you must first add a new parameter element to the DOM using an 'Insert-After' of the original parameter element, then use array subscripting to populate that new parameter with the correct values. Repeat as needed.

First, copy and clone the existing parameter element back into the variable using the Insert-After operation. This creates a second parameter element in the XML array. For example, see Figure 10–30.

Figure 10–30 Copy with Insert-After Operation



Second, create a new Copy operation and choose the parameter elements in the To/From areas of the dialog in the same manner as when copying values for the first parameter. However, in the lower right corner, change the XPath path to include a [2] (XML Arrays start at 1 and not 0) and click **OK**. Repeat as needed for each parameter required.

10.8.2 How to Invoke the ESSWebService setSubmitArgs Operation

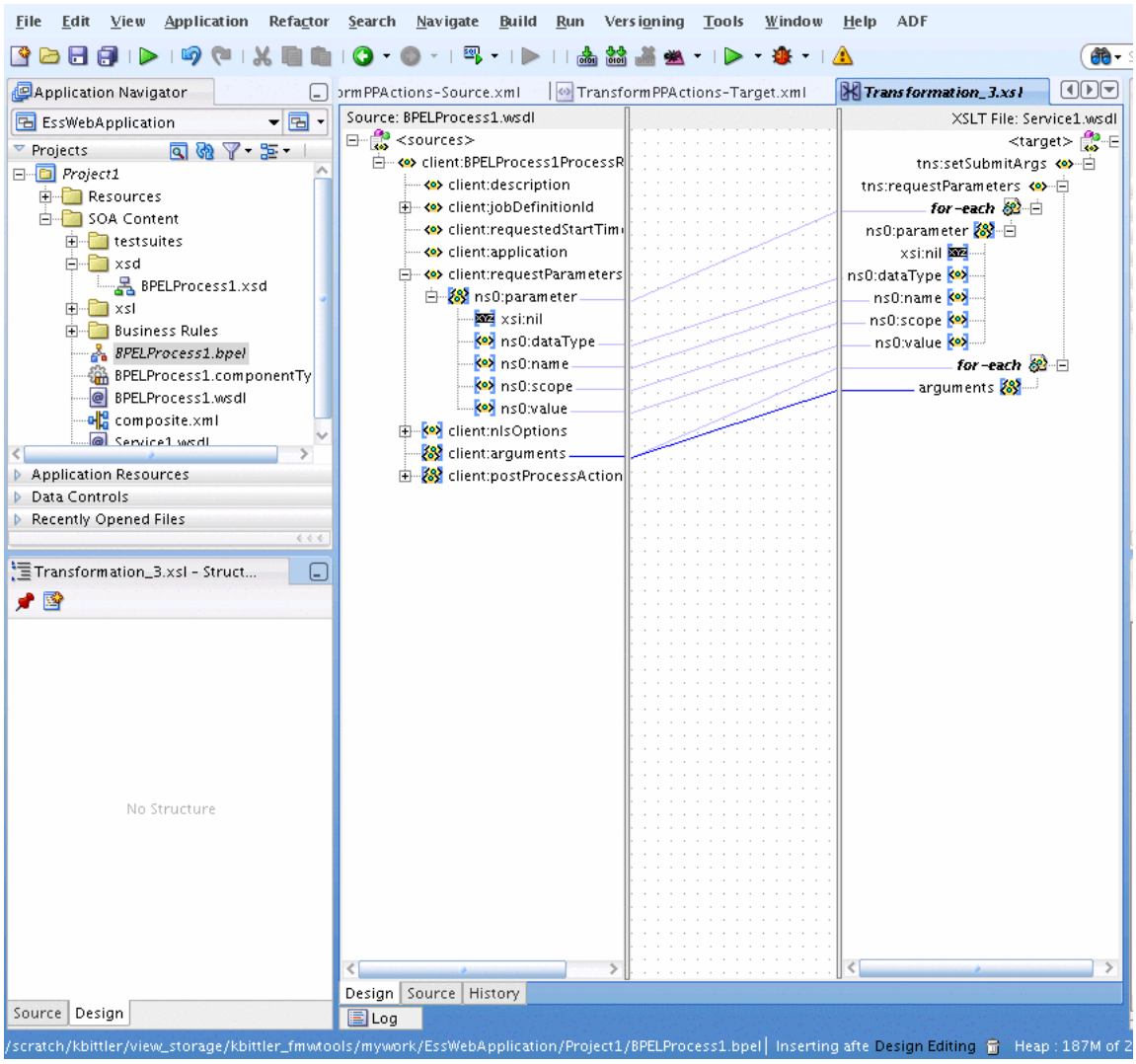
In the BPEL process you add an invoke activity to perform the Oracle Enterprise Scheduler web service `setSubmitArgs()` operation.

To add the Invoke activity to use setsubmitArgs for a request using ESSWebService:

1. In the Application Navigator, in Project1 expand SOA Content and select the BPEL file. For example, select **BPELProcess1.bpel**. This displays the BPEL swim lane.
2. From the Component Palette, drag-and-drop an **Invoke** Activity and place the activity before callbackClient.
3. Link the invoke activity to the ESSWebService by selecting the right arrow and dragging it to the Partner Link Service1. This brings up the Edit Invoke dialog.
4. In the Edit Invoke dialog, in the Operation field select **setSubmitArgs**.
5. In the Edit Invoke dialog, in the **Input** field click the **Add** icon. This displays the Create Variable dialog.
6. In the Create Variable dialog, click **OK**.
7. In the Edit Invoke dialog, in the **Output** field select the **Add** icon. This displays the Create Variable dialog.
8. In the Create Variable dialog, click **OK**.
9. In the Edit Invoke dialog, click **OK**. This displays the new invoke link to Service1.

10. Select the Invoke activity and double-click the name `Invoke_1` to select the text entry field. In the text entry field enter `setSubmitArgs`.
11. From the Component Palette, drag-and-drop a **Transform** Activity and place the activity before the **setSubmitArgs**. This transformation maps the BPEL flow input variable to the setSubmitArgs input variable.
12. Open the transformation activity. On the **Transformation** tab, in the **Source** area click the **Add** icon. This displays the Source Variable dialog.
13. In the Source Variable dialog select **inputVariable** and click **OK**.
14. In the transformation activity, on the **Transformation** tab, in the **Target Variable** field select **setSubmitArgs_setSubmitArgs_InputVariable** as the target.
15. In the transformation activity, on the **Transformation** tab, in the **Mapper File** field, click **Add** to create a new mapper file.
16. This creates a mapper file, as shown in Figure 10–31. Note that a "for-each" construct can be inserted by dragging an item from the XSLT Constructs area of the Component Palette.

Figure 10–31 Transformation for Set Submit Arguments



17. The transformation tool does not create exactly what is needed. You need to edit the XSLT source. In the source, find the following mapping.

```
<xsl:for-each select="/client:BPELProcess1ProcessRequest/client:arguments">
    <arguments>
      <xsl:value-of select="."/>
    </arguments>
  </xsl:for-each>
```

Replace this with the following; add "tns:" as a qualifier to "arguments", resulting in the following fragment. Note that the transformation tool design tab may incorrectly complain that this is not a valid transformation:

```
<xsl:for-each select="/client:BPELProcess1ProcessRequest/client:arguments">
    <tns:arguments>
      <xsl:value-of select="."/>
    </tns:arguments>
  </xsl:for-each>
```

Example 10–2 shows the complete transformation source file.

Example 10–2 Transformation Source for Set Submit Arguments Transformation

```
<?xml version="1.0" encoding="UTF-8" ?>
<?oracle-xsl-mapper
  <!-- SPECIFICATION OF MAP SOURCES AND TARGETS, DO NOT MODIFY. -->
  <mapSources>
    <source type="WSDL">
      <schema location="../BPELProcess1.wsdl"/>
      <rootElement name="BPELProcess1ProcessRequest"
                          namespace="http://xmlns.oracle.com/EssWebApplication/
                          Project1/BPELProcess1"/>
    </source>
  </mapSources>
  <mapTargets>
    <target type="WSDL">
      <schema location="../Service1.wsdl"/>
      <rootElement name="setSubmitArgs"
                          namespace="http://xmlns.oracle.com/scheduler"/>
    </target>
  </mapTargets>
  <!-- GENERATED BY ORACLE XSL MAPPER 11.1.1.0.0(build 090113.2000.2412) AT [FRI
       FEB 06 08:27:53 PST 2009]. -->
?>
<xsl:stylesheet version="1.0"
                xmlns:xpath20="http://www.oracle.com/XSL/Transform/java
                               /oracle.tip.pc.services.functions.Xpath20"
                xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/
                            business-process/"
                xmlns:client="http://xmlns.oracle.com/EssWebApplication
                              /Project1/BPELProcess1"
               xmlns:oraext="http://www.oracle.com/XSL/Transform/java/
                             oracle.tip.pc.services.functions.ExtFunc"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:dvm="http://www.oracle.com/XSL/Transform/java/
                          oracle.tip.dvm.LookupValue"
               xmlns:hwf="http://xmlns.oracle.com/bpel/workflow/xpath"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
               xmlns:med="http://schemas.oracle.com/mediator/xpath"
               xmlns:mhdr="http://www.oracle.com/XSL/Transform/java/oracle.tip.
                           mediator.service.common.functions
```

```
                    .GetRequestHeaderExtnFunction"
                  xmlns:ids="http://xmlns.oracle.com/bpel/services/
                             IdentityService/xpath"
                  xmlns:tns="http://xmlns.oracle.com/scheduler"
                  xmlns:xdk="http://schemas.oracle.com/bpel/extension
                             /xpath/function/xdk"
                  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
                  xmlns:xref="http://www.oracle.com/XSL/Transform/java
                              /oracle.tip.xref.xpath.XRefXPathFunctions"
                  xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:ns0="http://xmlns.oracle.com/scheduler/types"
                  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:ora="http://schemas.oracle.com/xpath/extension"
                  xmlns:socket="http://www.oracle.com/XSL/Transform/
                                java/oracle.tip.adapter.socket.ProtocolTranslator"
                  xmlns:ldap="http://schemas.oracle.com/xpath/extension/ldap"
                             exclude-result-prefixes="xsi xsl client plnk
                             xsd ns0 wsdl tns soap12 soap mime xpath20 bpws oraext
                             dvm hwf med mhdr ids xdk xref ora socket ldap">
  <xsl:template match="/">
    <tns:setSubmitArgs>
      <tns:requestParameters>
        <xsl:for-each select="/client:BPELProcess1ProcessRequest/client:
                              requestParameters/ns0:parameter">
          <ns0:parameter>
            <ns0:dataType>
              <xsl:value-of select="ns0:dataType"/>
            </ns0:dataType>
            <ns0:name>
              <xsl:value-of select="ns0:name"/>
            </ns0:name>
            <ns0:scope>
              <xsl:value-of select="ns0:scope"/>
            </ns0:scope>
            <ns0:value>
              <xsl:value-of select="ns0:value"/>
            </ns0:value>
          </ns0:parameter>
        </xsl:for-each>
      </tns:requestParameters>
      <xsl:for-each select="/client:BPELProcess1ProcessRequest/client:arguments">
        <tns:arguments>
          <xsl:value-of select="."/>
        </tns:arguments>
      </xsl:for-each>
    </tns:setSubmitArgs>
  </xsl:template>
</xsl:stylesheet>
```

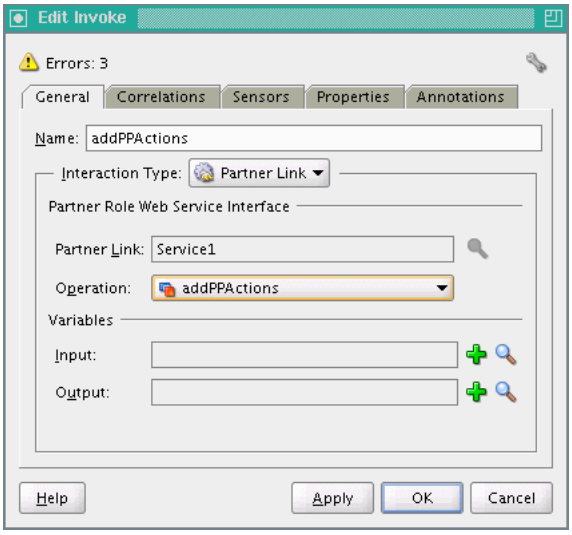
10.8.3 How to Invoke the ESSWebService addPPActions Operation

In the BPEL process you add an invoke activity to perform the Oracle Enterprise Scheduler web service `addPPActions()` operation.

To add the Invoke activity for addPPActions operation using ESSWebService:

1. In the Application Navigator, in Project1 expand SOA Content and select the BPEL file. For example, select **BPELProcess1.bpel**. This displays the BPEL swim lane.
2. From the Component Palette, drag-and-drop an **Invoke** Activity and place the activity before callbackClient.
3. Select the Invoke activity and double-click the name `Invoke_1` to select the text entry field. In the text entry field enter `addPPActions`.
4. Link the invoke activity to the ESSWebService by selecting the right arrow and dragging it to the Partner Link Service1. This brings up the Edit Invoke dialog.
5. In the Edit Invoke dialog, in the Operation field select **addPPActions**, as shown in Figure 10–32.

Figure 10–32 Adding AddPP Actions Operation



6. In the Edit Invoke dialog, in the **Input** field click the **Add** icon. This displays the Create Variable dialog.
7. In the Create Variable dialog, click **OK**.
8. In the Edit Invoke dialog, in the **Output** field select the **Add** icon. This displays the Create Variable dialog.
9. In the Create Variable dialog, click **OK**.
10. In the Edit Invoke dialog, click **OK**. This displays the new invoke link to Service1.
11. From the Component Palette, drag-and-drop a **Transform** Activity and place the activity before the **addPPActions**. This transformation maps the BPEL flow input variable to the addPPActions input variable.
12. Open the transformation activity. On the **Transformation** tab, in the **Source** area click the **Add** icon. This displays the Source Variable dialog.
13. In the Source Variable dialog select **inputVariable** and click **OK**.
14. In the transformation activity, on the **Transformation** tab in the **Target Variable** field select **addPPActions_addPPActions_InputVariable** as the target.
15. In the transformation activity, on the **Transformation** tab in the **Mapper File** field, click **Add** to create a new mapper file. This displays the XSL transformation file.

16. Create mappings as shown in Example 10–3.

    The requestParameters come from the addPPActions, overriding what is in the transformation. The remainder of the input still comes from the BPEL flow input variable. Assign `requestParametersReturn/ns2:parameter` of the addPPActions output variable to `requestParameters/ns2:parameter` of the addPPActions input variable, as in the previous examples.

Example 10–3 addPPActions Transformations

```
<?xml version="1.0" encoding="UTF-8" ?>
<?oracle-xsl-mapper
  <!-- SPECIFICATION OF MAP SOURCES AND TARGETS, DO NOT MODIFY. -->
  <mapSources>
    <source type="WSDL">
      <schema location="../BPELProcess1.wsdl"/>
      <rootElement name="BPELProcess1ProcessRequest"
         namespace="http://xmlns.oracle.com/EssWebApplication/Project1/BPELProcess1"/>
    </source>
  </mapSources>
  <mapTargets>
    <target type="WSDL">
      <schema location="../Service1.wsdl"/>
      <rootElement name="addPPActions" namespace="http://xmlns.oracle.com/scheduler"/>
    </target>
  </mapTargets>
  <!-- GENERATED BY ORACLE XSL MAPPER 11.1.1.0.0(build 090113.2000.2412) AT [FRI FEB 06 10:29:28
PST 2009]. -->
?>
<xsl:stylesheet version="1.0"
                xmlns:xpath20="http://www.oracle.com/XSL/Transform/java/
                               oracle.tip.pc.services.functions.Xpath20"
                xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
                xmlns:client="http://xmlns.oracle.com/EssWebApplication/Project1/BPELProcess1"
                xmlns:oraext="http://www.oracle.com/XSL/Transform/java/oracle.tip.pc.
                              services.functions.ExtFunc"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:dvm="http://www.oracle.com/XSL/Transform/java/oracle.tip.dvm.LookupValue"
                xmlns:hwf="http://xmlns.oracle.com/bpel/workflow/xpath"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:med="http://schemas.oracle.com/mediator/xpath"
                xmlns:mhdr="http://www.oracle.com/XSL/Transform/java/oracle.tip.
                            mediator.service.common.functions.GetRequestHeaderExtnFunction"
                xmlns:ids="http://xmlns.oracle.com/bpel/services/IdentityService/xpath"
                xmlns:tns="http://xmlns.oracle.com/scheduler"
                xmlns:xdk="http://schemas.oracle.com/bpel/extension/xpath/function/xdk"
                xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
                xmlns:xref="http://www.oracle.com/XSL/Transform/java/
                            oracle.tip.xref.xpath.XRefXPathFunctions"
                xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                xmlns:ns0="http://xmlns.oracle.com/scheduler/types"
                xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
                xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                xmlns:ora="http://schemas.oracle.com/xpath/extension"
                xmlns:socket="http://www.oracle.com/XSL/Transform/java/oracle.tip.
                              adapter.socket.ProtocolTranslator"
                xmlns:ldap="http://schemas.oracle.com/xpath/extension/ldap"
                exclude-result-prefixes="xsi xsl client plnk
```

```
                    xsd ns0 wsdl tns soap12 soap mime xpath20 bpws oraext dvm
                    hwf med mhdr ids xdk xref ora socket ldap">
  <xsl:template match="/">
    <tns:addPPActions>
      <tns:requestParameters>
        <xsl:for-each select="/client:BPELProcess1ProcessRequest/client:
                              requestParameters/ns0:parameter">
          <ns0:parameter>
            <ns0:dataType>
              <xsl:value-of select="ns0:dataType"/>
            </ns0:dataType>
            <ns0:name>
              <xsl:value-of select="ns0:name"/>
            </ns0:name>
            <ns0:scope>
              <xsl:value-of select="ns0:scope"/>
            </ns0:scope>
            <ns0:value>
              <xsl:value-of select="ns0:value"/>
            </ns0:value>
          </ns0:parameter>
        </xsl:for-each>
      </tns:requestParameters>
      <xsl:for-each select="/client:BPELProcess1ProcessRequest/client:postProcessAction">
        <tns:postProcessActions>
          <ns0:actionName>
            <xsl:value-of select="ns0:actionName"/>
          </ns0:actionName>
          <ns0:actionOrder>
            <xsl:value-of select="ns0:actionOrder"/>
          </ns0:actionOrder>
          <xsl:for-each select="ns0:arguments">
            <ns0:arguments>
              <xsl:value-of select="."/>
            </ns0:arguments>
          </xsl:for-each>
          <ns0:fileMgmtGroup>
            <xsl:value-of select="ns0:fileMgmtGroup"/>
          </ns0:fileMgmtGroup>
          <ns0:description>
            <xsl:value-of select="ns0:description"/>
          </ns0:description>
          <ns0:onError>
            <xsl:value-of select="ns0:onError"/>
          </ns0:onError>
          <ns0:onSuccess>
            <xsl:value-of select="ns0:onSuccess"/>
          </ns0:onSuccess>
          <ns0:onWarning>
            <xsl:value-of select="ns0:onWarning"/>
          </ns0:onWarning>
        </tns:postProcessActions>
      </xsl:for-each>
    </tns:addPPActions>
  </xsl:template>
</xsl:stylesheet>
```

10.8.4 How to Invoke the ESSWebService setStepsArgs Operation

In the BPEL process, you add an invoke activity to perform the Oracle Enterprise Scheduler web service `addPPActions()` operation.

As shown in Example 10–4, you can add the following to the `BPELProcess1.xsd` file to allow input for `setStepsArgs`.

Example 10–4 Enabling Input for setStepsArgs

```
<xs:element name="stepArgs" type="ns1:stepArgs"
                  minOccurs="0" maxOccurs="unbounded"/>
```

The main steps are as follows:

1. Create a transformation to map the BPEL flow input variable to the setStepsArgs input variable.

   From BPEL Activities and Components, select **Transform** and place before setStepsArgs. Open the new transformation activity. Select **inputVariable** as the source and **setStepsArgs_setStepsArgs_InputVariable** as the target. Create a new mapper file. Create the mappings as shown in the `SetStepsArgs` transformation example.

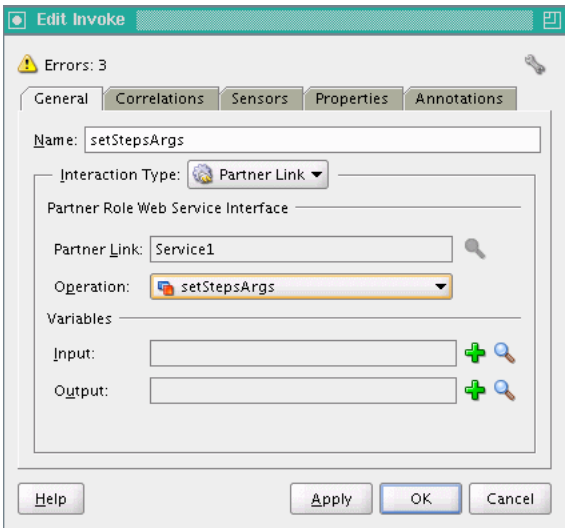
2. Create an assignment activity. In this example, you want the requestParameters to come from the previous step, addPPActions, overriding what is in the transformation. The remainder of the input still comes from the BPEL flow input variable. Assign `requestParametersReturn/ns2:parameter` of the addPPActions output variable to `requestParameters/ns2:parameter` of the setStepsArgs input variable, just as in previous examples.

   In the BPEL process you add an invoke activity to perform the Oracle Enterprise Scheduler web service `submitRecurringRequest()` operation. In this step you need to select the input and output for the Invoke Activity by associating values with the input and output variables.

To add the Invoke activity use setStepsArgs operation:

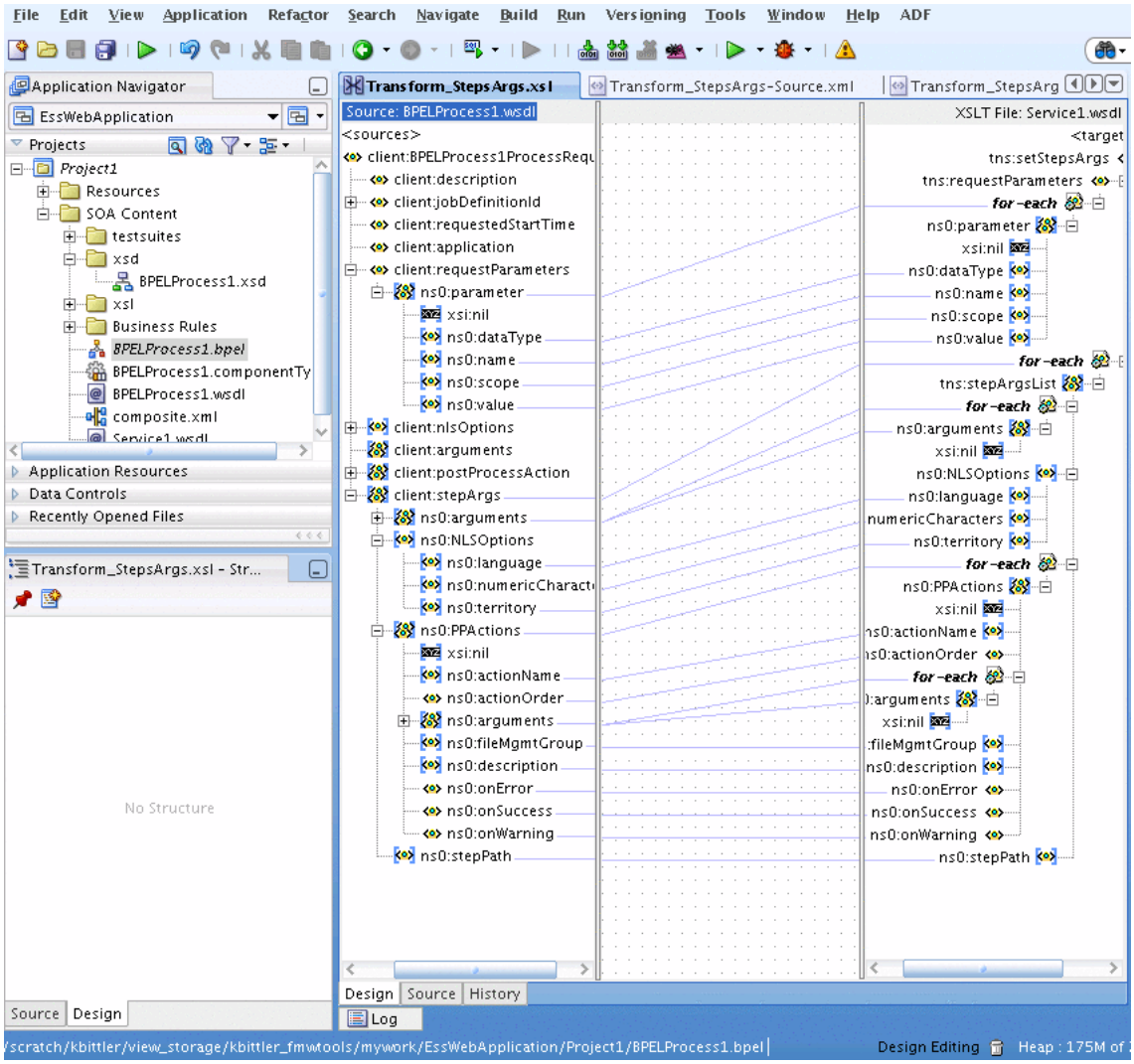
1. In the Application Navigator, in Project1 expand SOA Content and select the BPEL file. For example, select **BPELProcess1.bpel**. This displays the BPEL swim lane.
2. From the Component Palette, drag-and-drop an **Invoke** Activity and place the activity before callbackClient.
3. Select the Invoke activity and double-click the name `Invoke_1` to select the text entry field. In the text entry field enter `setStepsArgs`.
4. Link the invoke activity to the ESSWebService by selecting the right arrow and dragging it to the Partner Link Service1. This brings up the Edit Invoke dialog.
5. In the Edit Invoke dialog, in the Operation field select **setStepsArgs** as shown in Figure 10–33.

Figure 10–33 Set Step Arguments Operation



6. In the Edit Invoke dialog, in the **Input** field click the **Add** icon. This displays the Create Variable dialog.
7. In the Create Variable dialog, click **OK**.
8. In the Edit Invoke dialog, in the **Output** field select the **Add** icon. This displays the Create Variable dialog.
9. In the Create Variable dialog, click **OK**.
10. In the Edit Invoke dialog, click **OK**. This displays the new invoke link to Service1.
11. From the Component Palette, drag-and-drop a **Transform** Activity and place the activity before the **setStepsArgs**. This transformation maps the BPEL flow input variable to the setStepsArgs input variable.
12. Open the transformation activity. On the **Transformation** tab, in the **Source** area click the **Add** icon. This displays the Source Variable dialog.
13. In the Source Variable dialog select **inputVariable** and click **OK**.
14. In the transformation activity, on the **Transformation** tab in the **Target Variable** field select **setStepsArgs_setStepsArgs_InputVariable** as the target.
15. In the transformation activity, on the **Transformation** tab in the **Mapper File** field, click **Add** to create a new mapper file. This displays the XSL transformation file.
16. Create mappings as shown in Figure 10–34 using the mappings shown in Example 10–5.

Figure 10–34 Using the Transformation for Set Step Arguments Operation



17. Create an assignment activity. In this example, we want the requestParameters to come from the previous step, addPPActions, overriding what is in the transformation. There remainder of the input still comes from the BPEL flow input variable. Assign the requestParametersReturn/ns2:parameter of the addPPActions output variable to the requestParameters/ns2:parameter of the setStepsArgs input variable, just as in previous examples.

Example 10–5 Mapping Transformation for Set Steps Arguments Operation

```
<?xml version="1.0" encoding="UTF-8" ?>
<?oracle-xsl-mapper
  <!-- SPECIFICATION OF MAP SOURCES AND TARGETS, DO NOT MODIFY. -->
  <mapSources>
    <source type="WSDL">
      <schema location="../BPELProcess1.wsdl"/>
      <rootElement name="BPELProcess1ProcessRequest"namespace="http://xmlns.
           oracle.com/EssWebApplication/Project1/BPELProcess1"/>
    </source>
  </mapSources>
  <mapTargets>
    <target type="WSDL">
```

```
      <schema location="../Service1.wsdl"/>
      <rootElement name="setStepsArgs"
        namespace="http://xmlns.oracle.com/scheduler"/>
    </target>
  </mapTargets>
  <!-- GENERATED BY ORACLE XSL MAPPER 11.1.1.0.0(build 090113.2000.2412) AT [FRI
       FEB 06 10:56:22 PST 2009]. -->
?>
<xsl:stylesheet version="1.0"

xmlns:xpath20="http://www.oracle.com/XSL/Transform/java/oracle.tip.pc.services.
               functions.Xpath20"

xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/"

xmlns:client="http://xmlns.oracle.com/EssWebApplication/Project1/BPELProcess1"

xmlns:oraext="http://www.oracle.com/XSL/Transform/java/oracle.tip.pc.services.
              functions.ExtFunc"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:dvm="http://www.oracle.com/XSL/Transform/java/oracle.tip.dvm.LookupValue"
               xmlns:hwf="http://xmlns.oracle.com/bpel/workflow/xpath"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
               xmlns:med="http://schemas.oracle.com/mediator/xpath"

xmlns:mhdr="http://www.oracle.com/XSL/Transform/java/oracle.tip.mediator.service.
            common.functions.GetRequestHeaderExtnFunction"

xmlns:ids="http://xmlns.oracle.com/bpel/services/IdentityService/xpath"
          xmlns:tns="http://xmlns.oracle.com/scheduler"

xmlns:xdk="http://schemas.oracle.com/bpel/extension/xpath/function/xdk"
          xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"

xmlns:xref="http://www.oracle.com/XSL/Transform/java/oracle.tip.xref.xpath.
            XRefXPathFunctions"
               xmlns:plnk="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:ns0="http://xmlns.oracle.com/scheduler/types"
               xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
               xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
               xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
               xmlns:ora="http://schemas.oracle.com/xpath/extension"
               xmlns:socket="http://www.oracle.com/XSL/Transform/java
                            /oracle.tip.adapter.socket.ProtocolTranslator"
               xmlns:ldap="http://schemas.oracle.com/xpath/extension/ldap"
                          exclude-result-prefixes="xsi xsl client plnk xsd ns0
                          wsdl tns soap12 soap mime xpath20 bpws oraext dvm
                          hwf med mhdr ids xdk xref ora socket ldap">
  <xsl:template match="/">
    <tns:setStepsArgs>
      <tns:requestParameters>
        <xsl:for-each select="/client:BPELProcess1ProcessRequest/client:
           requestParameters/ns0:parameter">
          <ns0:parameter>
            <ns0:dataType>
              <xsl:value-of select="ns0:dataType"/>
            </ns0:dataType>
            <ns0:name>
```

```
            <xsl:value-of select="ns0:name"/>
          </ns0:name>
          <ns0:scope>
            <xsl:value-of select="ns0:scope"/>
          </ns0:scope>
          <ns0:value>
            <xsl:value-of select="ns0:value"/>
          </ns0:value>
        </ns0:parameter>
      </xsl:for-each>
    </tns:requestParameters>
    <xsl:for-each select="/client:BPELProcess1ProcessRequest/client:stepArgs">
      <tns:stepArgsList>
        <xsl:for-each select="ns0:arguments">
          <ns0:arguments>
            <xsl:value-of select="."/>
          </ns0:arguments>
        </xsl:for-each>
        <ns0:NLSOptions>
          <ns0:language>
            <xsl:value-of select="ns0:NLSOptions/ns0:language"/>
          </ns0:language>
          <ns0:numericCharacters>
            <xsl:value-of select="ns0:NLSOptions/ns0:numericCharacters"/>
          </ns0:numericCharacters>
          <ns0:territory>
            <xsl:value-of select="ns0:NLSOptions/ns0:territory"/>
          </ns0:territory>
        </ns0:NLSOptions>
        <xsl:for-each select="ns0:PPActions">
          <ns0:PPActions>
            <ns0:actionName>
              <xsl:value-of select="ns0:actionName"/>
            </ns0:actionName>
            <ns0:actionOrder>
              <xsl:value-of select="ns0:actionOrder"/>
            </ns0:actionOrder>
            <xsl:for-each select="ns0:arguments">
              <ns0:arguments>
                <xsl:value-of select="."/>
              </ns0:arguments>
            </xsl:for-each>
            <ns0:fileMgmtGroup>
              <xsl:value-of select="ns0:fileMgmtGroup"/>
            </ns0:fileMgmtGroup>
            <ns0:description>
              <xsl:value-of select="ns0:description"/>
            </ns0:description>
            <ns0:onError>
              <xsl:value-of select="ns0:onError"/>
            </ns0:onError>
            <ns0:onSuccess>
              <xsl:value-of select="ns0:onSuccess"/>
            </ns0:onSuccess>
            <ns0:onWarning>
              <xsl:value-of select="ns0:onWarning"/>
            </ns0:onWarning>
          </ns0:PPActions>
        </xsl:for-each>
        <ns0:stepPath>
```

```
            <xsl:value-of select="ns0:stepPath"/>
          </ns0:stepPath>
        </tns:stepArgsList>
      </xsl:for-each>
    </tns:setStepsArgs>
  </xsl:template>
</xsl:stylesheet>
```

10.9 Securing the Oracle Enterprise Scheduler Web Service

You can secure any of the Oracle Enterprise Scheduler web service operations using an Oracle Web Services Manager security policy.

For more information, see the "Securing and Administering WebLogic Web Services" chapter in the *Oracle Fusion Middleware Security and Administrator's Guide for Web Services*.

10.9.1 How to Secure the Oracle Enterprise Scheduler Web Service

Securing the Oracle Enterprise Scheduler web service involves attaching one security policy to the method that calls the web service, and another to the asynchronous callback to the SOA composite.

> **Note:** Oracle Fusion Applications make use of an Oracle WSM feature called global policy attachments (GPA). Using GPA, policies are not attached locally, but are specified at a global level. At runtime, components simply inherit the global policy and Oracle WSM enforces it.
>
> Unlike local policy attachments (LPA), which need to be added at every web service client and server, global policy attachment (GPA) can be attached at a domain level. This makes it easy for the system administrator to have a uniform policy for all web services across the domain.
>
> For more information about global policy attachments, see the "Securing Web Services Use Cases" chapter in the *Oracle Fusion Applications Developer's Guide*.

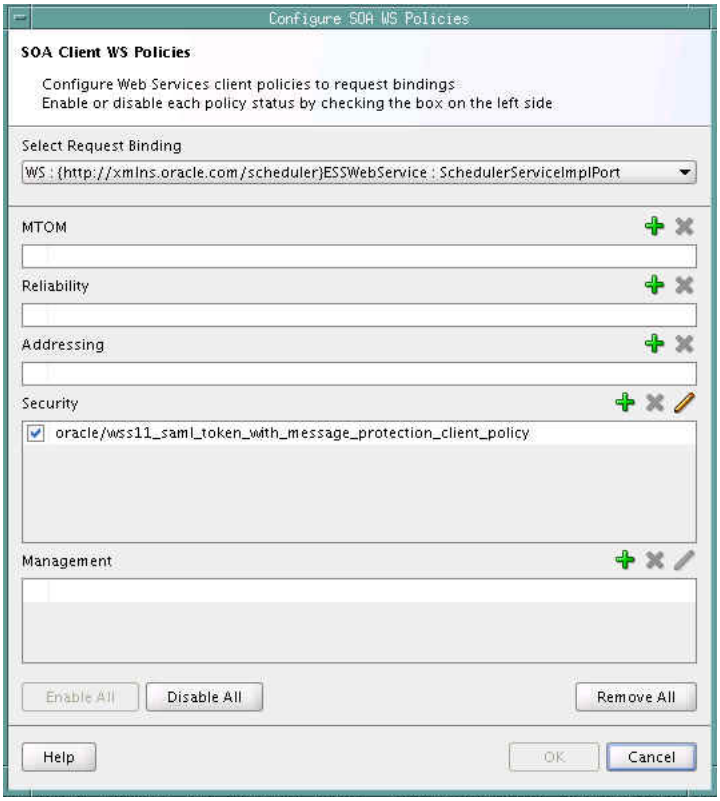
To secure the Oracle Enterprise Scheduler web service:

1. Open the SOA composite that calls the Oracle Enterprise Scheduler web service.
2. In the swim lane on the right, right-click the Oracle Enterprise Scheduler web service and select **Configure WS Policies** > **For Request**.

   The Configure SOA WS Policies window displays.
3. In the Security field, click the add button to attach a security policy to the client.

   Select the policy `oracle/wss11_saml_token_with_message_protection_client_policy` or `oracle/wss11_username_token_with_message_protection_client_policy` as shown in Figure 10–35, and click **OK**.

Figure 10–35 Client Security Policy for the Oracle Enterprise Scheduler Web Service



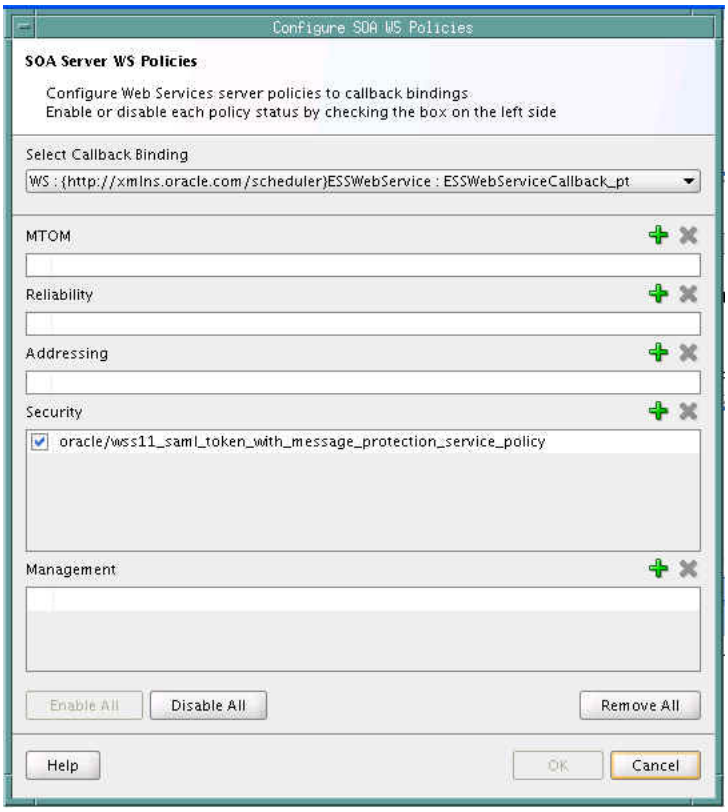
4. In the swim lane on the right, right-click the Oracle Enterprise Scheduler web service and select **Configure WS Policies** > **For Callback**.

   The Configure SOA WS Policies window displays.

5. In the Security field, click the add button to attach a security policy to the callback method.

   Select the policy `oracle/wss11_saml_token_with_message_protection_service_policy`, as shown in Figure 10–36, and click **OK**.

Figure 10–36 Callback Security Policy for the Oracle Enterprise Scheduler Web Service



6. Save your changes to the SOA composite file.

10.9.2 What Happens When You Secure the Oracle Enterprise Scheduler Web Service

The security policy `oracle/wss11_saml_token_with_message_protection_client_policy` secures the method that calls the Oracle Enterprise Scheduler web service. The security policy `wss11_saml_token_with_message_protection_service_policy` secures the asynchronous callback method that the web service uses to call back the SOA composite.

10.10 Deploying and Testing the Project

Next, you deploy the BPEL process to the Oracle WebLogic Server as described in "Deploying SOA Composite Applications" in *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*. Following deployment, you can test the web service using Oracle SOA Console.

10.10.1 How to Test the Web Service

To test the web service:

1. Open a browser and go to the SOA Console at the following URL.

   `http://<machine>:<port>/soa-console`

2. In the Applications area, select the deployed composite.
3. Click the **Test** dropdown and choose the service endpoint **Test Client**.

4. This an endpoint page where you can provide input to the BPEL process.
5. In the payload area, enter values for the job parameters.
6. Click **Invoke**.
7. Refresh the console page.
8. Click the latest instance ID to verify the progress of the BPEL file.

11

Defining and Using Job Sets

This chapter describes how to define and submit a job set. Oracle Enterprise Scheduler job sets provide for collections of job definitions that can be grouped together to run as a single unit.

This chapter includes the following sections:

11.1 Introduction to Defining and Using Job Sets

Oracle Enterprise Scheduler provides for collections of job definitions that can be grouped together to run as a single unit called a *job set*. A job set may be nested; thus a job set may contain a collection of job definitions or one or more child job sets. Each job definition or job set included within a job set is called a *job set step*.

A job set is defined as either a *serial* job set or a *parallel* job set. At runtime, Oracle Enterprise Scheduler runs parallel job set steps together, in parallel. When a serial job set runs, Oracle Enterprise Scheduler runs the steps one after another in a specific sequence. Using a serial job set Oracle Enterprise Scheduler supports conditional branching between steps based on the execution status of a previous step.

You can define a serial job set to include a parallel job set, or a parallel job set to include a serial job set. job sets that include a mix of parallel and serial job sets are called *complex job set*s. For example, when a serial job set contains a child parallel job set, the serial job set runs serially until it reaches the child parallel job set. Then, all the job definitions or job set definitions in the child parallel job set run in parallel. Upon completion of the child parallel job set the serial job set continues running its remaining steps serially. Nested parallel job sets behave the same as non-nested parallel job sets.

For every step in a job set Oracle Enterprise Scheduler supports properties that provide runtime flexibility for how a particular step affects the entire job set. These properties are defined on a per step basis. Table 11–1 shows properties that are useful for job set steps. Any property can be defined on a job set step.

Table 11–1 Job Set Step Properties

Property	Description
EFFECTIVE_APPLICATION	Specifies if the step is a job, the job will execute in the effective application. If the step is a nested job set, the jobs in the nested job set will execute in the effective application. The effective application becomes the application for the request for the step and for any child requests of the step. This property can be defined for job definitions and job types as well as job sets.
SELECT_STATE	Specifies whether the result state of a job set step should be included when determining the state of the job set. Specifies whether the execution state of the step affects the eventual state of entire job set. By default, all job set steps affect the job set state. To prevent the state of a particular job set step from affecting the state of the job set, set SELECT_STATE to false for that step. To allow the state of a job set step to affect the overall state of the job set, set SELECT_STATE to true for that step.

Oracle Enterprise Scheduler provides the capability for a job set to execute across multiple applications. A job set runs in its hosting application and by default all job set steps also run in this application. A job set step can be associated with a different application by defining the EFFECTIVE_APPLICATION system property on the step. If the step is a job definition, the job definition executes in the effective application. If the step is a nested job set definition, the job definitions or job set definitions in the nested job set execute in the effective application. The effective application becomes the application for the request for the step and for any child requests of the step. For more information, see Section 11.3, "Cross Application Job Sets".

11.2 Defining Job Sets

You can define a job set in Oracle JDeveloper by specifying the following:

- The name, package, and description for the job set
- The parameters for the job set
- The system properties for the job set
- Specifying the job set steps

The contents of a job set are specified when you define the job set steps. For example, for a serial job set you specify the name and the execution mode and then you add the job set steps to define the sequence of job definitions or child job sets that run when the job set runs.

11.2.1 How to Define a Job Set

An Oracle Enterprise Scheduler job set is defined by a name, a package, a job set execution mode, step definitions, parameters, and system properties.

To create a job set:

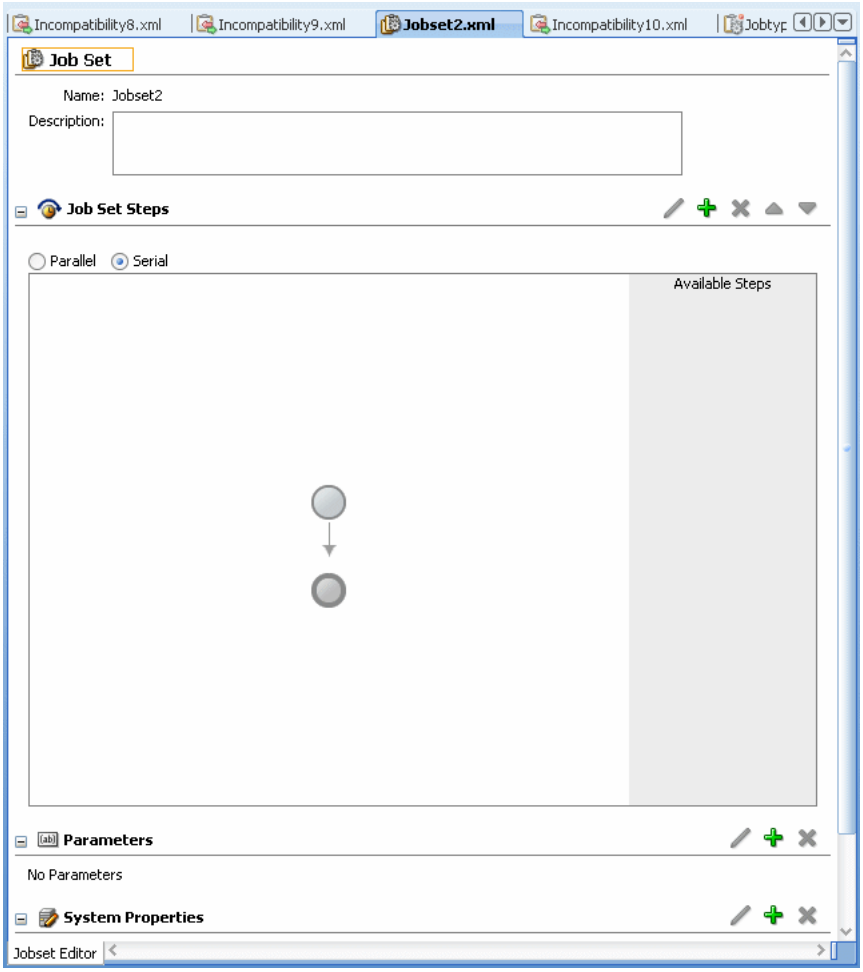
1. In Oracle JDeveloper, right-click in the project to view the New Gallery.
2. Under **Categories**, expand **Business Tier** and select **Enterprise Scheduler Metadata**.
3. Under **Items**, select **job set** and click **OK**. This displays the Create Job Set window.
4. In the Create Job Set window, specify the following:
   a. In the **Name** field, enter a name for the job set or accept the default name.

b. In the **Package** field, optionally enter a package name for the job set.

c. The **Location** field displays the full path of the directory where the job set file is stored.

d. Click **OK**. This creates the job set and displays the Job Set Definition page, as shown in Figure 11–1.

Figure 11–1 Job Set Editor with Serial Job Set



5. In the Job Set Editor pane, in the **Description** field enter a description for the job set.

6. In the Job Set Steps area, select the **Parallel** or **Serial** radio button to specify parallel or serial execution mode for the job set.

7. In the Job Set Editor pane add the job set steps. For more information on adding job set steps, see Section 11.2.2, "How to Define Serial Job Set Steps" or Section 11.2.3, "How to Define Parallel Job Set Steps".

8. In the Parameters area, click **Add** to add parameters associated with the job set. You use parameters to represent an application-specific parameter for the job set or a step specific parameter for the job set. For more information on using parameters, see Section 5.1, "Introduction to Using Parameters and System Properties". For more information, see Section 5.1.2.2, "What You Need to Know About Job Set Level Parameter Materialization".

9. In the System Properties area, click **Add** to add system properties associated with the job set. For more information on using system properties, see Section 5.4, "Using System Properties".
10. **Save** the job set.

11.2.2 How to Define Serial Job Set Steps

To define serial job set steps you select the serial execution mode and then add job set steps. Job set steps are created from the available job definitions and job sets defined in the current project. You define serial job set steps when you specify a step ID and a job definition child job set definition associated with the step. You also define links from a job set step terminal states to specify the next step. Table 11–2 lists the possible terminal states that you can specify using JDeveloper.

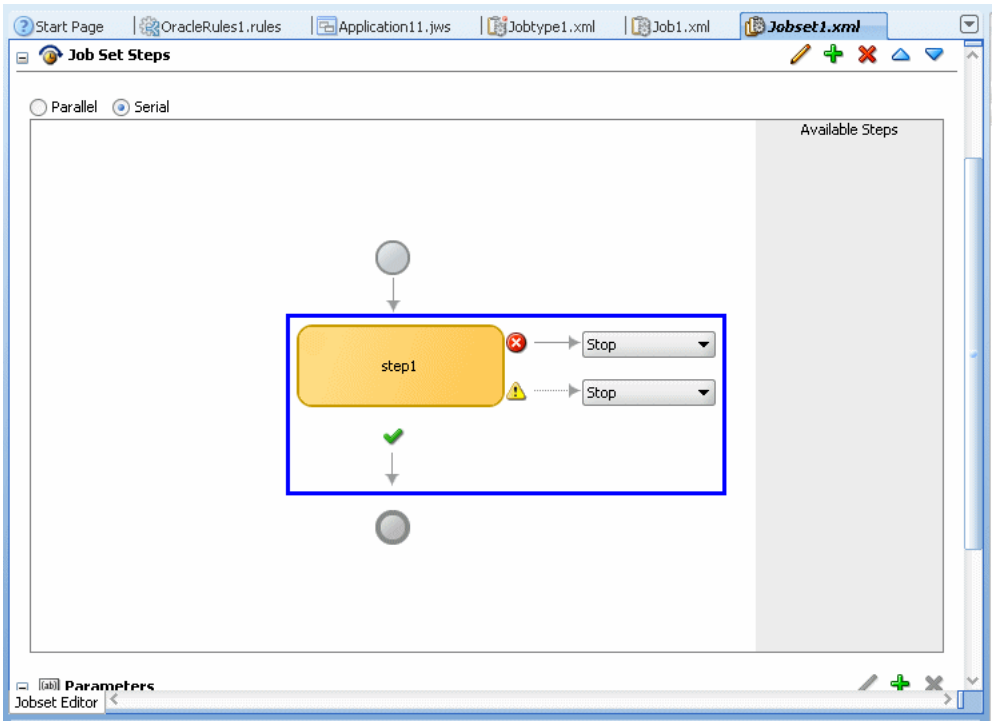
Table 11–2 Job Set Serial Execution Step Terminal States

Terminal State	Description
`SUCCEEDED`	Oracle JDeveloper indicates this state with a checkmark icon. This path represents a child step or child job set was successfully processed by the system.
`WARNING`	Oracle JDeveloper indicates this step with a warning icon. A child step or child job set resulted in a warning.
`ERROR`	Oracle JDeveloper indicates this step with an error icon. Some aspect of the request to run the child step or child job set processing resulted in an error.

To add serial job set steps:

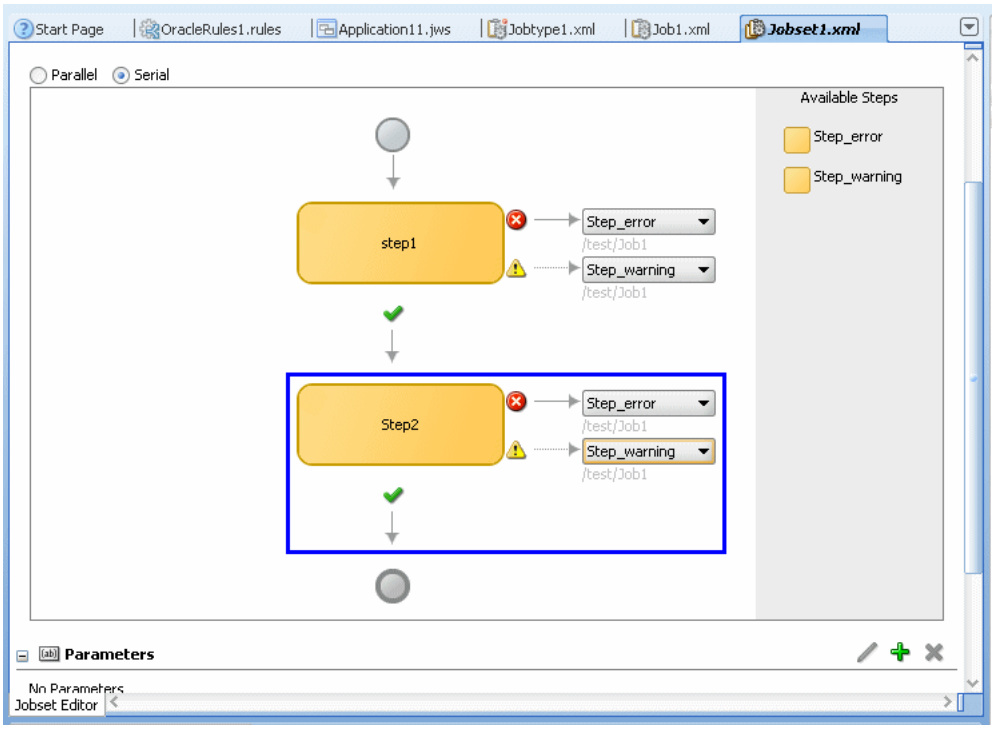
1. First, define the appropriate job definitions or job sets and define the parent job set to contain the steps.
2. In the Job Set Editor pane, in the Job Set Steps area, select **Serial** execution mode.
3. Click the **Add** icon to add a job set step. This displays the Add Step window.
4. In the Step ID field, enter the step ID. For example, enter `step1`.
5. In the Job field, from the dropdown list select a job definition or a job set to associate with the step. For example, select Job1.
6. If you need to define step level parameters, then select the Parameters tab and add job set step parameters for the step.
7. If you need to define step level system properties, then select the System Properties tab and add job set step system properties for the step.
8. Select a destination for the step. The step can be added as part of the job set by selecting **Insert into main diagram**. To make the step available for use in another step, for either error or warning states, select **Add to list of available steps**.
9. Click **OK**, this adds the job set step, as shown in Figure 11–2.

Figure 11–2 Job Set with a Step Added



10. From the dropdown list next to the error icon, select Stop or select the step for the `ERROR` terminal state for the step. For example, from the dropdown list select **Step_error** (Step_error needs to be defined).
11. From the dropdown list next to the warning icon, select Stop or select the step for the `WARNING` terminal state for the step. For example, from the dropdown list select `Step_warning` (Step_warning needs to be defined).
12. Click the **Add** icon and add additional steps as needed.
13. Click **OK**, as shown in Figure 11–3.

Figure 11–3 Job Set with Two Steps Added



11.2.3 How to Define Parallel Job Set Steps

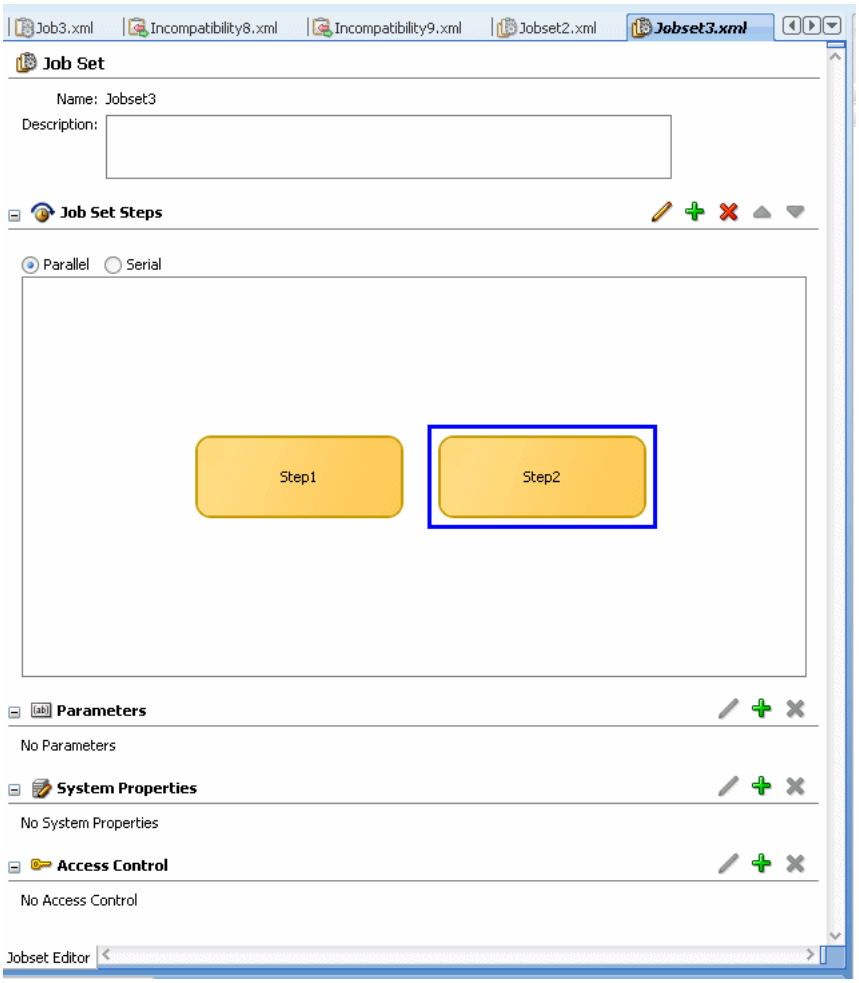
You can add parallel job set steps to a job set.

To add parallel job set steps:

1. First, define the appropriate job definitions and job set definitions and the parent job set.
2. In the Job Set Editor, select the **Parallel** execution mode.
3. Click the **Add** icon to add a job set step to the job set.

   The Add Step window displays.
4. In the Job field, select a job definition or a job set.
5. If you need to define step level parameters, then select the Parameters tab and add job set step parameters for the step.
6. If you need to define step level system properties, then select the System Properties tab and add job set step system properties for the step.
7. Click **OK**, this adds the job set step.
8. Click the **Add** icon.
9. In the Add Step dialog, select the job set or job definition to use for next job in the parallel job set.
10. Click **OK**. The job set step displays in the job set, as shown in Figure 11–4.

Figure 11–4 Adding Job Set Steps to a Parallel Job Set



11.2.4 What Happens When You Define a Job Set

When you define a job set with Oracle JDeveloper, Oracle JDeveloper creates an XML file containing elements that represent the steps that you define.

When you define a parallel job set you specify a set of job set steps that run together. A parallel job set only contains steps, and does not contain links between steps, as all the steps execute together and do not depend on each other or upon the order in which each step runs.

When you define a job set Oracle JDeveloper creates an XML document that conforms to the Oracle Enterprise Scheduler job step schema.

11.2.5 What You Need to Know About Serial Job Sets

When you define a serial job set the associated XML document includes job set steps and links. Oracle Enterprise Scheduler enforces the following limitations for serial job set definitions:

- To prevent looping within a job set, job set definitions should not contain circular execution paths. A circular execution path, or a loop, is defined at the job set level as follows: loop is a path from one job set step along the links of any number of other steps back to the same job set step. For example, in a job set with a flow from

Job_A, to Job_B, to Job_C defined, Oracle Enterprise Scheduler does not allow you to define an execution path from Job_B or Job_C back to Job_A. For example you could a create circular execution path, or a loop, if one of the links in a job set step for success, error, or warning links back to the same job set step. Thus, each job set step can link to any of the available job definitions or job sets, or they could all use the same job definition or job set as a link for the success, error and warning case. There is only a possible loop based on the path through the job set steps, as identified by the job set step ID. Oracle Enterprise Scheduler validates job sets at submission time to try to prevent job set step level looping. Also, Oracle JDeveloper does not allow you to create a job set containing a job set step level loop.

- To prevent looping within a job set, job set definitions should not contain self-referencing execution paths. For example, in a job set with Job_B defined, Oracle Enterprise Scheduler does not allow you to define an execution path from Job_B to Job_B itself if Job_B ends up with a terminal state of ERROR. However using the RETRIES property available for a job definition or a job set, you can have multiple executions up to the configured RETRIES number.
- When there is no job set link defined for a terminal state of a step, it implies that the job set should stop if the step ends with the unspecified terminal state. For example if there is no link defined for a step Job_D for the state WARNING, and if the step Job_D ends up with the state of WARNING, the job set stops execution.

Each job set step can be defined to use any of the available job definitions or job sets, and multiple steps may use the same job definition or job set.

11.2.6 What You Need to Know About Job Set Parameters and System Properties

There are cases where job set parameters or system properties may conflict with parameters or system properties set either in metadata or when a job request is submitted. For more information on how job set parameters and system properties are handled, see Section 5.2, "Using Parameters with the Metadata Service" and Section 5.3, "Using Parameters with the Runtime Service".

11.2.7 What Happens at Runtime for Job Set State Priorities and State Transitions

At runtime, the individual steps in a job set can end up with different terminal states, as indicated in Table 11–2. When a job set step is a job set, the job set step also ends with one of these terminal states. Oracle Enterprise Scheduler provides a priority hierarchy for the terminal states of job set steps. This means that when there are multiple steps in a job set, the job set terminal state is applied the terminal state of the step with the highest priority terminal state. Thus, the highest priority terminal state of the steps determines the resulting state for the entire job set.

The resulting state of a job set affects all subsequent state dependent processing within the system. A job set always follows the basic rule of transitioning to a terminal state based on the terminal states of its child requests, only after the completion of all child requests. As a rule, the job set transitions to one of the computed terminal states only after all child requests have finished and transitioned to terminal states. For example, if a given job set is actually a step within another job set, then the way in which the state of the inner job set request is computed affects the conditional execution within the outer job set.

Table 11–3 shows the possible job set terminal states with the level indicated in the Priority column.

Table 11–3 Job Set Terminal State Transitions

Terminal State	Description	Priority
ERROR	If any step in a job set finishes with the terminal state of ERROR, the entire job set is marked with the terminal state of ERROR no matter what the state of the other steps. For serial job sets, if one step goes to ERROR, subsequent steps will not execute. For parallel job sets, all steps begin at the same time, and the job set state is not determined until the job set steps reach a terminal state.	The ERROR state has the highest priority.
WARNING	If any step in a job set ends up with the terminal state of WARNING, and there is no step with the terminal state of ERROR then the job set is marked with the terminal state WARNING. When the terminal state is WARNING, post processing will begin.	Lower than ERROR
EXPIRED	The job set transitions to EXPIRED state if at least one of the child requests expires while there is no step that ends with the terminal state of ERROR or WARNING.	Lower than ERROR and WARNING
CANCELLED	Based on the actual outcome of a cancellation attempt, the job set can transition to CANCELLED if at least one child request successfully processes the cancellation attempt and transitions to CANCELLED state. The cancellation might have been requested on the entire job set or just a specific child request. Further the transition to CANCELLED follows the priorities of terminal states. Therefore the job set transitions to CANCELLED terminal state only if there is no step that ends with the state of ERROR, WARNING, or EXPIRED and there is at least one step with terminal state of CANCELLED. When a job set is cancelled, steps that have not been added or run are considered to be CANCELLED for the purpose of final state.	Lower than ERROR, WARNING, and EXPIRED
SUCCEEDED	The job set is considered as SUCCEEDED if and only if all child requests completed with the terminal state of SUCCEEDED.	The SUCCEEDED state has the lowest priority among all terminal states

Table 11–4 lists additional possible states for a job set:

Table 11–4 Possible Job Set Runtime States

State	Description
WAIT	This is the initial state of the submitted job set request. Once the job set request transitions to RUNNING state, however, all generated child requests transition directly to READY state rather than WAIT state.
READY	Job sets never go to READY state. The submitted job set request transitions from WAIT to RUNNING state. Nested job sets are generated in RUNNING state. The only job set steps that begin in READY state are steps composed of job definitions.
RUNNING	The submitted job set transitions from WAIT to RUNNING state when it begins to be processed. Nested job sets start in RUNNING state and remain in RUNNING state as long as at least one child is in a non-terminal state.

Table 11–4 (Cont.) Possible Job Set Runtime States

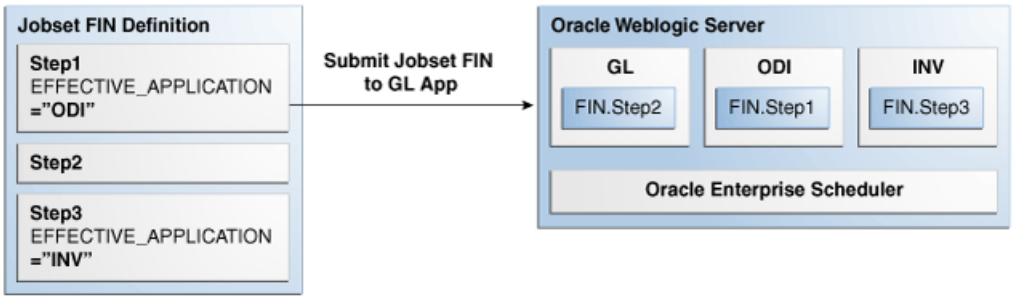
State	Description
CANCELLING	A job set transitions to CANCELLING when the user requests a cancellation for the entire job set. This can be done by calling cancelRequest() with the request ID of the parent request representing the job set. Passing the parent request ID indicates that the user wants to cancel entire job set irrespective of its current, non-terminal, state and the states of its child requests. In such cases, a cancellation will be attempted on all child requests that are still active and have not already transitioned to a terminal state. On the other hand if cancellation is attempted only on a specific child request in the job set, there won't be any state change for the parent request and only the particular child request will transition to CANCELLING if possible. If the cancel happens during post-processing, the state is set to WARNING rather than CANCELLED. If the job set finishes before the cancel is issued, the job set can have state SUCCEEDED.
COMPLETED	This state indicates that the job set or job set step has finished executing and post-processing will begin.
BLOCKED	The BLOCKED state is not a terminal state. However any request can remain in a BLOCKED state for a long period until the blocking condition is eliminated (such as incompatibility). In the case of a job set, any individual step might be BLOCKED while other steps either complete or may be running. The job set itself, however, remains in a RUNNING state. Eventually if all steps in the job set complete except the ones that are in the BLOCKED state, the job set cannot continue further until the blocking step is ready to run. When the blocked step unblocks and completes, the job set can proceed. After the steps complete, the job set eventually goes to the appropriate terminal state. For a serial job set, the job set may stop at a step that is in BLOCKED state. In such cases, all previous steps are complete and the job set cannot continue until the blocked step executes. However for a parallel job set, multiple steps can remain in BLOCKED state. Further, while some steps are blocked, other steps can still continue to run.
HOLD	The HOLD state is very similar to the BLOCKED state. Following the same rules for the BLOCKED state, a job set cannot continue running while a step is in HOLD state. A serial job set cannot continue if the current step in the execution flow is stuck at HOLD state. In the case of a parallel job set, if at least one step is stuck in HOLD state while all other steps have completed, the job set can complete when the step is no longer in HOLD state.

11.3 Cross Application Job Sets

Oracle Enterprise Scheduler provides the capability for a job or a job set to execute across multiple applications as shown in Figure 11–5:

- Job set FIN has three steps, two of which are defined to execute in different applications.
- Job set FIN is submitted to the GL application.
- Step 1 has the EFFECTIVE_APPLICATION system property set to ODI, so Step 1 executes in the ODI application.
- Step 2 does not have an effective application set, so it executes in the GL application.
- Step 3 has the EFFECTIVE_APPLICATION system property set to INV, so Step 3 executes in the INV application.

Figure 11–5 Cross Application Job Set Steps



11.3.1 Overview of Cross Application Job Sets

A job set runs in its hosting application and by default, all job set steps also run in this application. A job set step can be associated with a different application by defining the `EFFECTIVE_APPLICATION` system property on the step. If the step is a job, the job will execute in the effective application. If the step is a nested job set, the jobs in the nested job set execute in the effective application. When `EFFECTIVE_APPLICATION` is defined for a step, the request for the step and any child requests of the step are associated with the effective application, meaning the `APPLICATION` system property for those requests will be set to the effective application.

The `EFFECTIVE_APPLICATION` system property may only be defined in metadata, specifically job set, job set step, job type, and job. The property `EFFECTIVE_APPLICATION` is not supported in the request parameters. The effective application must be in the same cluster as the hosting application, or an error will result. If a submitted job set defines the effective application, that value must be the same as the hosting application, or the job set submission will be rejected.

Subrequests created by a job set step must run in the same application as the job set step. In other words, `EFFECTIVE_APPLICATION` is not supported for subrequests. If the job for a subrequest defines the effective application, that value must be the same as the application of the job submitting the subrequest, or the subrequest submission will be rejected.

For a job set that executes across multiple applications, querying for requests by application is not sufficient to retrieve all children. Oracle Enterprise Scheduler supports absolute parent id as a query field, making it possible to query for all requests in a job set regardless of the application. The absolute parent id is the request id of the job set that was submitted to the hosting application.

11.3.2 Requirements for Cross Application Job Sets

Oracle Enterprise Scheduler supports cross-application job set subject to the following requirements:

1. All applications for a given job set must be deployed in the same cluster.
2. All applications in the job set must share the same enterprise security.
3. All request metadata must be accessible from the application the job set is submitted to, referred to as the hosting application. All metadata for the request are persisted to the runtime store for the hosting application. The persisted metadata include all metadata used by the submitted job set and any nested job set.

4. Metadata for subrequests must be accessible from the application that submits the subrequest, unless the metadata used by the subrequest were already persisted to the runtime store at job set submission time.

11.4 Using Input and Output Forwarding

Oracle Enterprise Scheduler configures a USER_FILE_DIR parameter to specify the directory for all jobs to store their input and output files. This parameter is populated by the property RequestFileDirectory in the connections.xml file. When this parameter is set, Oracle Enterprise Scheduler set the system property USER_FILE_DIR for all job requests. When a job request is processed, in the pre- or post-processor or its execution the job can read, write, create, delete and manage files and sub-directories based this property. Oracle Enterprise Scheduler does not impose any structure on the user file directory nor support any file or directory operations.

The purpose of this file support is to allow job implementation to reference files relative to a configurable location so that the job implementation is not tied to a particular environment. It de-couples job implementation with file input and output from the job execution environment.

The USER_FILE_DIR property allows job requests to dynamically change the file.

11.4.1 Supporting Input and Output Forwarding in Job Sets

Sometimes a step in a job set needs input from the previous step in the job set. Oracle Enterprise Scheduler uses two system properties INPUT_LIST and OUTPUT_LIST to facilitate forwarding the output from one step to the input of the next step.

When a job produces information, such as a list of output files, that needs to be passed on to the next step in a job set, the job adds the information to the OUTPUT_LIST property. Upon completion of the job request execution, Oracle Enterprise Scheduler forwards the OUTPUT_LIST property of the request so that it becomes the INPUT_LIST property of the next step before it executes. The next step takes as its input the output of the previous step.

A job set step can be a single job or a job set, Oracle Enterprise Scheduler supports forwarding with nested job sets as well. For a serial job set, Oracle Enterprise Scheduler defines the output of the job set as the output of the last step of the job set, meaning that only the OUTPUT_LIST property of the last step is forwarded to the next step. Similarly, the input to a serial job set is forwarded only to the first step of the job set; that is, only the first step of a serial job set has the INPUT_LIST property set to the value of the OUTPUT_LIST property of the previous step.

For a parallel job set, Oracle Enterprise Scheduler specifies that the output of the job set is the concatenation of the OUTPUT_LIST property of every job in the job set, separated by a delimiter (with no order guaranteed). The input to a parallel job set is forwarded to every job in the job set, meaning that every job in the parallel job set has the same INPUT_LIST property. The system property OUTPUT_LIST_DELIMITER specifies the delimiter used when listing output files.

Suppose a job set has two jobs, each job producing its own output file, file1.txt and file2.txt. The system property OUTPUT_LIST for that job set will have the values file1.txt;file2.txt, assuming the value of OUTPUT_LIST_DELIMITER is a semi-colon. The concatenated list of output files enables the next job step in the job set to access output files generated by previous steps within the job set.

The InputFile class provides access to files as input to a job definition. There is currently no mechanism for accepting a file as an input to a job request.

Except for forwarding the value of the OUTPUT_LIST property of a step to the value of the INPUT_LIST property of the next step, Oracle Enterprise Scheduler treats the two properties like any other system properties. Oracle Enterprise Scheduler does not define the format for the value of the properties (except for the semicolon delimiter in case of parallel job set). It is the responsibility of the job to define the syntax and semantics for the properties; for example using a fully qualified name or relative path name and a comma or space as a delimiter.

12

Defining and Using a Job Incompatibility

This chapter describes how to use an Oracle Enterprise Scheduler job incompatibility. The incompatibility feature lets you specify job requests that cannot run together.

This chapter includes the following sections:

- Section 12.1, "Introduction to Using a Job Incompatibility"
- Section 12.2, "Defining Incompatibility with Oracle JDeveloper"
- Section 12.3, "What Happens at Runtime to Handle Job Incompatibility"

For information about how to create and submit job requests see the following chapters, for Java jobs, Chapter 3, "Use Case Oracle Enterprise Scheduler Sample Application", and Chapter 6, "Creating and Using PL/SQL Jobs", and Chapter 7, "Creating and Using Process Jobs". For more information on using job sets, see Chapter 11, "Defining and Using Job Sets".

> **Note:** To simplify the discussion we refer only to job definitions in this incompatibility chapter, but in all cases this discussion applies to both job definitions and job sets.

12.1 Introduction to Using a Job Incompatibility

A given incompatibility specifies either a global incompatibility or a domain, property-based, incompatibility. Oracle Enterprise Scheduler supports incompatibility between job definitions or job sets based on an incompatibility definition as represented by the `oracle.as.scheduler.Incompatibility` Java class. The `IncompatibilityType` enum specifies the valid incompatibility types.

- **Domain-Specific** (`DOMAIN`): where at most two job definitions are marked as incompatible within the scope of a resource, where the resource is identified by a system property name or a user-defined parameter name. A property name must be specified for each job definition used to define the incompatibility. Parameters specified through `parameterV0` will be submitted to the request as request properties submit.`argument1`, ... `submit.argument#`. In the incompatibility definition for the properties, specify `submit.argument1`, ... `submit.argument#`, Oracle Enterprise Scheduler ensures that requests for the incompatible jobs do not run at the same time if they have the same value for that resource.
- **Global** (`GLOBAL`): where at most two job definitions are marked as incompatible, regardless of any resource or property. Oracle Enterprise Scheduler ensures that requests for the incompatible jobs do not run at the same time.

An Oracle Enterprise Scheduler incompatibility definition specifies either a global incompatibility or a domain (property-based) incompatibility. An incompatibility

consists of at most two entities (job definition or job set) and the resource over which they need to be incompatible. A resource is not specified for a global incompatibility. Each entity can be flagged as being self-incompatible. Oracle Enterprise Scheduler does not support a mixed mode where one entity represents a domain (property-based) entity and another entity represents a global (no property) entity.

For a domain incompatibility, the resource is represented by a property name that might be different for each entity of the incompatibility. For example, if a domain incompatibility is created for two job definitions, JobA and JobB, then the resource (property) identified for each entity might have different property names in JobA and JobB. It might be called `foo` in JobA while it might be called `foo2` in JobB. Oracle Enterprise Scheduler considers a request for JobA and a request for JobB to be incompatible if they have the same value for their respective property, and those requests would not run at the same time. If the requests have a different value for their respective property, they are considered compatible and allowed to run concurrently.

An incompatibility definition specifies which job definition is incompatible with another job definition. A given job definition does not directly point to or reference any incompatibility definitions.

Oracle Enterprise Scheduler determines which, if any, incompatibility definitions reference the job definition at request submission. It also determines the resource (property) value for any domain incompatibility. That information is used throughout the processing life cycle of the request.

12.1.1 Job Self Incompatibility

A job definition or job set can be defined as self incompatible where the job definition or job set is incompatible with itself. A self-incompatibility implies that multiple job requests associated with a single job definition cannot run together. An incompatibility definition can contain a single entity if it is marked as self-incompatible. For global self-incompatibly, Oracle Enterprise Scheduler ensures that multiple requests for that particular job or job set definition are not run simultaneously. For property-based self-incompatibly, Oracle Enterprise Scheduler ensures that requests for that particular job or job set definition, and having the same value for the property, are not run at the same time.

12.2 Defining Incompatibility with Oracle JDeveloper

You can define an incompatibility in Oracle JDeveloper by specifying the following:

- The name and package for the incompatibility
- The incompatibility type
- The entity for the incompatibility and whether there is a self incompatibility
- For a domain specific incompatibility, the property associated with the incompatibility for each entity

12.2.1 How to Define a Global Incompatibility

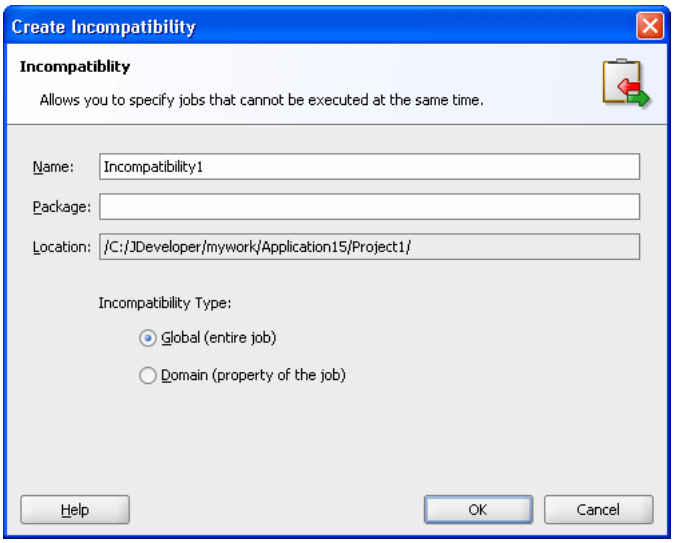
An Oracle Enterprise Scheduler global incompatibility is defined by a name, a package and entities.

To create a global incompatibility:

1. In Oracle JDeveloper, right-click in the project to view the New Gallery.

2. Under **Categories**, expand **Business Tier** and select **Enterprise Scheduler Metadata**.
3. Under **Items**, select **Incompatibility** and click **OK**. This displays the Create Incompatibility window, as shown in Figure 12–1.

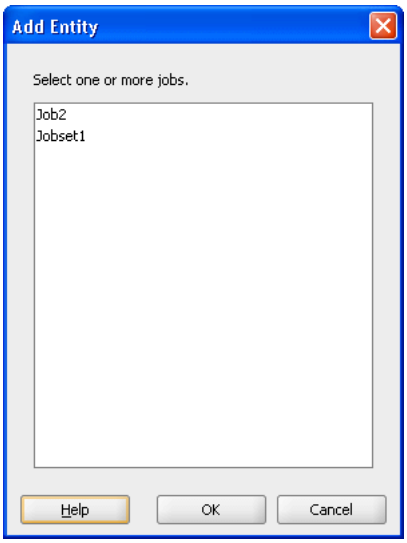
Figure 12–1 Create Incompatibility Window



4. Use the Create Incompatibility dialog to specify the following:
   a. In the **Name** field, enter a name for the incompatibility or accept the default name.
   b. In the **Package** field, enter a package name for the incompatibility.
   c. The **Location** field displays the full path of the directory where the incompatibility file is stored.
   d. In the Incompatibility Type field, select **Global**. and click **OK**.

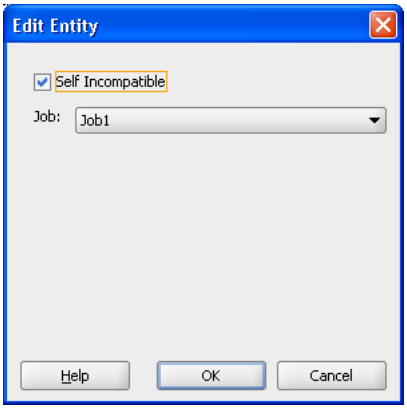
      The incompatibility is created, and the Incompatibility Definition page displays.
5. In the Incompatibility Editor pane, in the **Description** field enter a description for the incompatibility.
6. In the Incompatibility Entities area, click **Add** to add entities. This displays the Add Entity dialog, as shown in Figure 12–2.

Figure 12–2 Incompatibility Add Entity Window



7. Select one or more entities for the incompatibility and click **OK**. The Incompatibility Editor displays.
8. To specify a self incompatibility or to change the entity, double-click the entity in the Entities area. This displays the Edit Entity dialog as shown in Figure 12–3.

Figure 12–3 Edit Entity Window for Global Incompatibility



9. To specify self incompatibility, select **Self Incompatible**.
10. **Save** the incompatibility.

12.2.2 How to Define a Domain Incompatibility

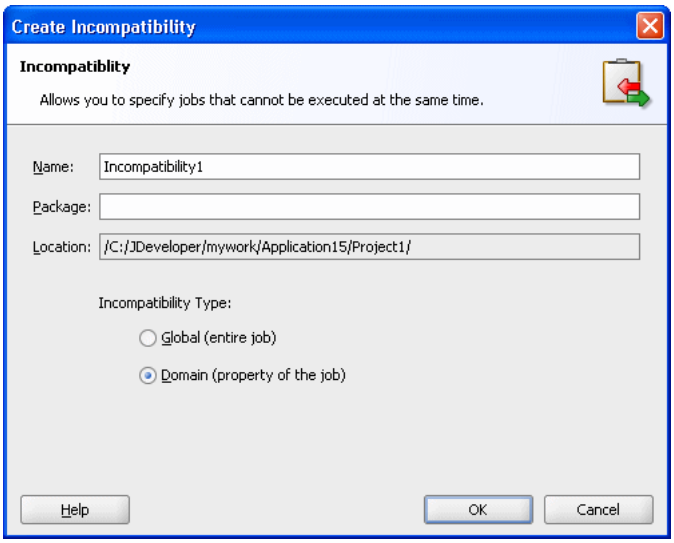
An Oracle Enterprise Scheduler domain incompatibility is defined by a name, a package, entities, and properties for each entity.

To create an incompatibility:

1. In Oracle JDeveloper, right-click in the project to view the New Gallery.
2. Under **Categories**, expand **Business Tier** and select **Enterprise Scheduler Metadata**.

3. Under **Items**, select **Incompatibility** and click **OK**. This displays the Create Incompatibility window.
4. Use the Create Incompatibility dialog to specify the following:
   a. In the **Name** field, enter a name for the incompatibility or accept the default name.
   b. In the **Package** field, optionally enter a package name for the incompatibility.
   c. The **Location** field displays the full path of the directory where the incompatibility file is stored.
   d. In the Incompatibility Type field, select **Domain**, as shown in Figure 12–4.

Figure 12–4 Create Incompatibility Window



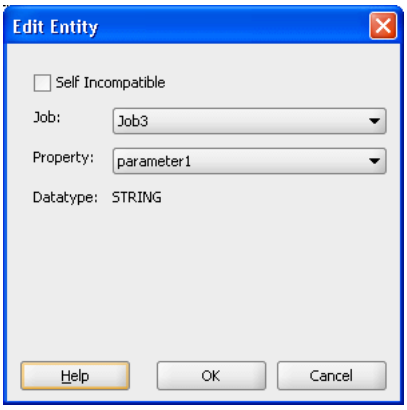
   Click **OK**. This creates the incompatibility and displays the Incompatibility Editor.
5. In the Incompatibility Editor pane, in the **Description** field enter a description for the incompatibility.
6. In the Incompatibility Entities area, click **Add**.

   The Add Entity window displays.
7. Select one or more jobs or job sets to add to the incompatibility and click **OK**.

   The Incompatibility Editor displays.
8. To specify a self incompatibility or modify an entity or its properties, under the Entities field, double-click an entity.

   The Edit Entity window displays, as shown in Figure 12–5.

Figure 12–5 Incompatibility Edit Entity Window



9. To specify self incompatibility, select **Self Incompatible**.
10. **Save** the incompatibility.

12.3 What Happens at Runtime to Handle Job Incompatibility

At runtime, Oracle Enterprise Scheduler handles incompatibility definitions according to the incompatibility type, global or domain (property-based). When a job request is submitted, Oracle Enterprise Scheduler determines which incompatibility definitions reference the job or job set definition used for the request submission. For each domain incompatibility it also determines the value of the resource, property, for that incompatibility. When the request is ready to be executed, Oracle Enterprise Scheduler checks if there are any incompatible requests already executing. If so, the request is blocked until all requests for which it is incompatible have completed.

Note: The value of the property for a domain incompatibility is determined at request submission, and originates either in the job definition or a request parameter passed during submission. If no such parameter is found, that incompatibility is ignored during subsequent request processing. The request will be compatible with any other request with regard to that incompatibility definition. This initial value as specified at request submission time is used even if it is subsequently altered.

12.3.1 What Happens to Subrequests with an Incompatible Parent Request

A request which is incompatible with another request is also incompatible with the subrequests of that request (the children). A request that has been blocked by a subrequest parent remains blocked while any subrequests execute and until the subrequest parent request is resumed and completes.

12.3.2 What Happens to the Scope of Request Incompatibility

Every validated request is assigned to an enterprise. Incompatibility is supported only among requests that are associated with the same enterprise. A request for one enterprise is never incompatible with a request for a different enterprise even if an incompatibility has been defined between the job definitions used by those requests.

13

Using the Runtime Service

This chapter describes how to use the runtime service that provides the APIs for submitting and managing job requests and for querying job request information from the job request history.

This chapter includes the following sections:

13.1 Introduction to the Runtime Service

Oracle Enterprise Scheduler lets you define and run different job types including: Java classes, PL/SQL procedures, and process job types (forked processes). To run these job types you need to submit a job definition.

You can use the runtime service to perform different types of operations, including:

- Submit: These operations let you supply a job definition to Oracle Enterprise Scheduler to create job requests
- Manage: These operations allow you to change the state of job requests and to update job requests
- Query: These operations let you find the status of job requests and report job request history

13.2 Accessing the Runtime Service

Like the metadata service, Oracle Enterprise Scheduler provides a runtime MBean proxy interface.

The runtime service `open()` method begins each Oracle Enterprise Scheduler runtime service user transaction. In an Oracle Enterprise Scheduler application client you obtain a `RuntimeServiceHandle` reference that is created by `open()` and you pass the reference to runtime service methods. The `RuntimeServiceHandle` reference provides a connection to the runtime service for the client application. In the client application you must explicitly close the runtime service by calling `close()`. This ends the transaction and causes the transaction to be committed or rolled back (undone). The

`close()` not only controls the transactional behavior within the runtime service, but it also allows Oracle Enterprise Scheduler to release the resources associated with the `RuntimeServiceHandle`.

13.2.1 How to Access the Runtime Service and Obtain a Runtime Service Handle

Oracle Enterprise Scheduler exposes the runtime service to your application program as a Stateless Session Enterprise Java Bean (EJB). You can use JNDI to locate the Oracle Enterprise Scheduler runtime service Stateless Session EJB.

Example 13–1 shows a lookup for the Oracle Enterprise Scheduler runtime service using the `RuntimeServiceLocalHome` object.

Example 13–1 JNDI Lookup to Access Oracle Scheduler Runtime Service

```
import oracle.as.scheduler.core.JndiUtil;
// Demonstration of how to lookup runtime service from a
// Java EE application component

    RuntimeService runtime = JndiUtil.getRuntimeServiceEJB();
    RuntimeServiceHandle rHandle = null;
    .
    .
    .
    try
    {
         ...
         rHandle = runtime.open();
         ...
    }
    finally
    {
        if (rHandle != null)
        {
            runtime.close(rHandle);
        }
    }
```

Note: When you access the runtime service:

- `JndiUtil.getRuntimeServiceEJB()` assumes that the RuntimeService EJB has been mapped to the local JNDI location "ess/runtime". This happens automatically in the hosted application's message-driven bean (MDB).
- The `open()` call provides a `RuntimeServiceHandle` reference. You use this reference with the methods that access the runtime service in your application program.
- When you finish using the runtime service you must call `close()` to release the resources associated with the `RuntimeServiceHandle`.

13.3 Submitting Job Requests

When you submit a job definition you create a new job request. You can submit a job request using a job definition that is persisted to a metadata repository, or you can

create a job request in an ad hoc manner where the job definition or the schedule is not stored in the metadata repository (for information about ad hoc requests, see Section 13.6, "Submitting Ad Hoc Job Requests").

13.3.1 How to Submit a Request to the Runtime Service

You create a job request by calling `submitRequest()`. Depending on your needs, you can create a job request with one of the following formats:

- Create a new job request using a job definition stored in the metadata repository, to run once at a specific time.
- Create a new job request using a job definition and a schedule, each stored in the metadata repository.

Example 13–2 shows the `submitRequest()` method that creates a new job request with a job definition that resides in the metadata repository. You can also submit an ad hoc job request where the job definition and schedule are not stored in the metadata repository. For more information, see Section 13.6, "Submitting Ad Hoc Job Requests". You can also submit a sub-request. For more information, see Chapter 14, "Using Subrequests".

Example 13–2 Creating a Job Request with submitRequest()

```
long requestID = 0L;
MetadataObjectId jobDefnId;

RequestParameters p = new RequestParameters();

p.add(SystemProperty.CLASS_NAME, "demo.jobs.Job");

Calendar start = Calendar.getInstance();
start.add(Calendar.SECOND, startsIn);

requestID =
    runtime.submitRequest(r,
        "My Java job",
        jobDefnId,
        start,
        p);
```

Note: When you submit a job request using the runtime service:

- You obtain the runtime service handle as shown in Example 13–1.
- The runtime service internally uses the metadata service to obtain job definition metadata with the supplied `MetadataObjectId`, `jobDefnId`.

13.3.2 What You Should Know About Default System Properties When You Submit a Request

When you create a job request Oracle Enterprise Scheduler resolves and stores the properties associated with the job request. Oracle Enterprise Scheduler requires that certain system properties are associated with a job request. If you do not set these required properties anywhere in the properties hierarchy when a job request is submitted, then Oracle Enterprise Scheduler provides default values.

Table 13–1 shows the runtime service field names and the corresponding system properties for the required job request properties.

Table 13–1 Runtime Service Default Value Fields and Corresponding System Properties

Value	Runtime Service Default Value Field	Corresponding System Property	Description
0	`DEFAULT_REQUEST_EXPIRATION`	`REQUEST_EXPIRATION`	The default expiration time, in minutes, for a request. The default value is 0 which means the request will never expire.
4	`DEFAULT_PRIORITY`	`PRIORITY`	The default system priority associated with a request.
5	`DEFAULT_REPROCESS_DELAY`	`REPROCESS_DELAY`	The default period, in minutes, in which processing must be postponed by a callout handler that returns `Action.DELAY`.
0	`DEFAULT_RETRIES`	`RETRIES`	The default number of times a failed request will be retried. The default value is 0 which means a failed request is not retried.

13.3.3 What You Should Know About Metadata When You Submit a Request

All Oracle Enterprise Scheduler Metadata associated with a job request is persisted in the runtime store at the time of request submission. Persisted metadata objects include job definition, job type, job set, schedule, incompatibility definitions, and exclusion definition. Metadata is stored in the context of a top level request, and each metadata object is uniquely identified by the absolute parent request id and its metadata id. Each unique metadata object is stored only once for a top-level request, even if the definition is used multiple times in the request. This ensures that every child request uses the same definition.

When a request is submitted, all known metadata for the request is persisted. For subrequests, the metadata is not know until the subrequest is submitted, so subrequest metadata is persisted when the subrequest is submitted, after first checking that the metadata object is not already persisted in the runtime store.

Metadata persisted in the runtime store is removed when the absolute parent request is deleted.

13.4 Managing Job Requests

After you submit a job request, using the `requestID` you can do the following:

- Get request information
- Change the state of the request
- Update request parameters
- Purge a request

13.4.1 How to Get Job Request Information with getRequestDetail

Using the runtime service, with a `requestID`, you can obtain information about a job request that is in the system. Table 13–2 shows the runtime service methods that allow you to obtain job request information.

Table 13–2 Runtime Service Get Request Methods

Runtime Service Method	Description
`getRequestDetail()`	Retrieves complete runtime details for the specified request
`getRequestDetailBasic()`	Retrieves basic runtime details of the specified request. The RequestDetail returned by this method includes most of the information as `getRequestDetail()`, but certain less commonly used information is omitted to improve performance.
`getRequestParameter()`	Retrieves the value of a request parameter.
`getRequests()`	Retrieves an enumeration of immediate child request identifiers associated with the specified request
`getRequestState()`	Retrieves the current state of the specified request

Example 13–3 shows code that determines if there is any immediate child request in the HOLD state.

Example 13–3 Determining Whether Any Immediate Child Job Requests Are on Hold

```
h = s_runtime.open();
try {

    s_runtime.holdRequest(h,reqid);

    Enumeration  e = s_runtime.getRequests(h, reqid);

    boolean foundHold = false;
    while (e.hasMoreElements()) {

        long childid = ((Long)e.nextElement()).longValue();
        State state = s_runtime.getRequestState(h,childid);
        if (state == State.HOLD) {
            foundHold = true;
            break;
        }
    }
```

13.4.2 How to Change Job Request State

Using the runtime service, with a `requestID`, you can change the state of a job request. Table 13–3 shows the runtime service job request state change methods. The job request management methods allow you to change the state of a request, depending on the state of the job request. For example, you cannot cancel a request with `cancelRequest()` if the request is in the `COMPLETED` state.

Table 13–3 Runtime Service Job Request State Methods

Runtime Service Method	Description
`cancelRequest()`	Cancels the processing of a request that is not in a terminal state.
`deleteRequest()`	Marks a request in a terminal state for deletion.
`holdRequest()`	Withholds further processing of a request that is in `WAIT` or `READY` state.
`releaseRequest()`	Releases a request from the `HOLD` state.

Example 13–4 shows a `submitRequest()` with methods that control the state of the job request. The `holdRequest()` holds the processing of the job request. The corresponding `releaseRequest()` releases the request. This example does not show the conditions that require the hold for the request.

Example 13–4 Runtime Service releaseRequest() Usage

```
rHandle = s_runtime.open();
try {
            s_runtime.holdRequest(rHandle,reqid);
            Enumeration  e = s_runtime.getRequests(rHandle, reqid);
            while (e.hasMoreElements()) {

                long childid = ((Long)e.nextElement()).longValue();
                State state = s_runtime.getRequestState(rHandle,childid);
                if (state == State.HOLD) {
                    foundHold = true;
                    break;
                }
            }
.
.
.
            s_runtime.releaseRequest(rHandle, reqid);
.
.
.
```

Note: Note the following in Example 13–4:

- You obtain the runtime service handle, `rHandle`, as shown in Example 13–1.
- The `holdRequest()` places the request in the `HOLD` state.
- You may do some required processing while the request is in the `HOLD` state.
- The `releaseRequest()` releases the request from the `HOLD` state.

13.4.3 How to Update Job Request Priority and Job Request Parameters

Using the runtime service you can update job request system properties or request parameters. Table 13–4 shows the runtime service methods that allow you to lock and update up a job request.

Table 13–4 Runtime Service Update Methods

Runtime Service Method	Description
`lockRequest()`	Acquires a lock for the given request. The lock is released when `close()` operation is subsequently invoked or the encompassing transaction is committed. If an application tries to invoke this operation while the lock is being held by another thread, this method will block until the lock is released. Use this method to ensure data consistency when updating request parameters or system properties.
`updateRequestParameter()`	Updates the property value of the specified request subject to the property read-only constraints.

Example 13–5 shows code that updates a job request parameter. This code would be wrapped in a try/finally block as shown in Example 13–1.

Example 13–5 Sample Runtime Service Parameter Update

```
...
  s_runtime.lockRequest(rhandle, reqid);
  s_runtime.updateRequestParameter(rhandle, reqId, paramName, "yy");
...
```

Example 13–5 shows the following:

- Obtain the runtime service handle, `rhandle`, as shown in Example 13–1.
- Acquire a lock for either the request using `lockRequest()`
- Perform the update operation with `updateRequestParameter()`
- Use `close()` to cause the transaction to be committed or rolled back (undone). The `close()` not only controls the transactional behavior within the runtime service, but it also allows Oracle Enterprise Scheduler to release the resources associated with the `RuntimeServiceHandle`.

13.5 Querying Job Requests

Using the runtime service you can query job request information. This involves the following steps:

- Query for request identifiers and limit results with a filter.
- Get request details to provide additional information for each request ID that the query returns.

There is only one query method; the runtime service `queryRequests()` method returns an enumeration of request IDs that match the query. The `queryRequests() method` includes a filter argument that contains field, comparator, and value combinations that help select query results. For more information on filters, see Section 4.4.1, "How to Create a Filter".

When you create a filter for a query, you can use any of the field names shown in Table 13–5 when querying the runtime store.

Table 13–5 Query Filter Fields For Querying the Runtime (Defined in Enum RuntimeService.QueryField)

Name	Description
`ABSPARENTID`	The absolute parent request ID of a request.
`APPLICATION`	The application name.
`ASYNCHRONOUS`	Indicates if the job is asynchronous, synchronous or unknown. The value of the field is not set until the request is processed. The field data type is `java.lang.Boolean`. The value may be `NULL` if the nature of the job has not yet been determined.
`CLASSNAME`	The name of the executable class that processed the request
`COMPLETED_TIME`	The date and time that Oracle Enterprise Scheduler finished processing the request. This field represents the time the process phase was set to COMPLETED.
`DEFINITION`	The job definition ID (Metadata Object ID).
`ELAPSEDTIME`	The amount of time, in milliseconds, that elapsed while the request was running.
`ENTERPRISE_ID`	The enterprise ID.
`ERROR_TYPE`	The request error type.

Table 13–5 (Cont.) Query Filter Fields For Querying the Runtime (Defined in Enum

Name	Description
EXTERNAL_ID	The identifier for an external portion of an Enterprise Scheduler asynchronous Java job.
INSTANCEPARENTID	The request ID of the instance parent request.
JOB_TYPE	The job type ID (Metadata Object ID).
NAME	The request description.
PARENTREQUESTID	The parent request ID.
PRIORITY	The priority of the request.
PROCESS_PHASE	The process phase of the request.
PROCESSEND	The date and time that the process ended. The PROCESSSTART is set only when a request transitions from READY to RUNNING. This implies that (PROCESSEND - PROCESSSTART) encompasses the entire span of execution: from the time the state becomes RUNNING to the time it transitions to a terminal state.
PROCESSOR	The name of the instance that processed the request.
PROCESSSTART	The date and time that the process started. The PROCESSSTART is set only when a request transitions from READY to RUNNING. This implies that (PROCESSEND - PROCESSSTART) encompasses the entire span of execution: from the time the state becomes RUNNING to the time it transitions to a terminal state.
PRODUCT	The product name.
READYWAIT_TIME	The amount of time, in milliseconds, a request has been waiting to run since it became READY.
REQUEST_CATEGORY	The request category specified for the request.
REQUESTEDEND	The requested end time.
REQUESTEDSTART	The requested start time.
REQUESTID	The request ID of a submitted request.
REQUESTTYPE	The type of request (that is, an element of RequestType)
RESULTINDEX	Controls the starting and ending index of the returned results. This field allows users to express result constraints such as "return only results 10 through 20".
RETRIED_COUNT	The retried count associated with a job. This field represents the number of times the job was retried.
SCHEDULE	The schedule ID (Metadata Object ID).
SCHEDULED	The time when the request is scheduled to be executed.
STATE	The job request state.
SUBMISSION	The submission time of the request.
SUBMITTER	The submitter of the request.
SUBMITTERGUID	The submitter GUID of the request.
TIMED_OUT	Indicates whether the job has timed out.
TYPE	The execution type of the request.
USERNAME	The name of the user who submitted the request.
WAITTIME	The amount of time, in milliseconds, a request has been waiting to run.
WORKASSIGNMENT	The name of the work assignment that was active when the request was processed.

Table 13–6 shows the runtime service method for querying job requests and Example 13–6 shows the use of this method.

Table 13–6 Runtime Service Query Methods

Runtime Query Method	Description
`queryRequests()`	Gets a summary of requests.

Example 13–6 Using queryRequest() Method

```
Filter filter =
     new Filter(RuntimeService.QueryField.DEFINITION.fieldName(),
                Filter.Comparator.EQUALS,
                m_myJavaSucJobDef.toString())
     .and(RuntimeService.QueryField.STATE.fieldName(),
          Filter.Comparator.EQUALS,
          new Integer(12) );

 //
 Enumeration requests =
     s_runtime.queryRequests(h,
                             filter,
                             RuntimeService.QueryField.REQUESTID,
                             false);
```

13.6 Submitting Ad Hoc Job Requests

To use an ad hoc request you supply request parameters, a job definition, and optionally a schedule that you create and define without saving it to a metadata repository. An ad hoc request does not require you define the details of a job request in a metadata repository. Thus, ad hoc requests support an abbreviated job request submission process that can occur without using a connection to the metadata repository.

Note: Ad hoc requests have the following limitation: job sets are not supported with ad hoc requests.

13.6.1 How to Create an Ad Hoc Request

To create an ad hoc request you use the ad hoc version of `submitRequest()`. For the job definition, instead of supplying a job definition `MetadataObjectId`, you can define the job definition object and use a system property that corresponds to the job type, as shown in Table 13–7.

Table 13–7 Ad Hoc Request Job Definition System Properties for Job Types

System Property	Description
`CLASS_NAME`	Specifies the Java class to execute (for a Java job type).
`PROCEDURE_NAME`	Specifies the PL/SQL stored procedure to execute (for an SQL job type).
`CMDLINE`	Specifies the command line used to invoke an external program for a process job request.

With one signature of the ad hoc version of `submitRequest()` you do not need to supply `MetadataObjectIds`, you can provide the `Schedule` object as an argument as

object instances directly to `submitRequest()`. Other ad hoc `submitRequest()` signatures allow you to submit a job request using a job definition from metadata and an instance for the `Schedule` object.

Example 13–7 shows sample code for an ad hoc request submission that uses a schedule.

Example 13–7 Creating Request Parameters and a Schedule for an Ad Hoc Request

```
RequestParameters p = new RequestParameters();
String propName = "testProp";
String propValue = "testValue";
p.add(propName, propValue);
p.add(SystemProperty.REQUEST_EXPIRATION, new Integer(10));
p.add(SystemProperty.LISTENER, "test.listener.TestListener");
p.add(SystemProperty.EXECUTE_PAST, "TRUE");
p.add("application", getApplication());
p.add(SystemProperty.CLASS_NAME, "test.job.HelloWorld");

Calendar start = Calendar.getInstance();
start.add(Calendar.SECOND, 5);
Calendar end = (Calendar) start.clone();
end.add(Calendar.SECOND, 5);

Recurrence recur = new Recurrence(RecurrenceFields.FREQUENCY.SECONDLY,
                                  2,
                                  start,
                                  end);

Schedule schedule = new Schedule("mySchedule",
                         "Run every 2 sec for 3 times.",
                          recur);

// adhoc submission, no metadata definitions passed
reqId = s_runtime.submitRequest(h,
                        "testAdhocJavaWithSchedule",
                        JobType.ExecutionType.JAVA_TYPE,
                        schedule,
                        null,
                        Calendar.getInstance(),
                        null,
                        p);
```

In this example, note the following ad hoc specific details for the request submission:

- The `CLASS` name is set to define the Java class that runs when Oracle Enterprise Scheduler executes the job request: `p.add(SystemProperty.CLASS_NAME, "test.job.HelloWorld");`
- The `submitRequest()` includes an argument that specifies the job type: `JobType.ExecutionType.JAVA_TYPE`.
- Specify the Java class, the procedure name, or the command line program to execute when the ad hoc Request is processed by setting one of the system properties shown in Table 13–7.
- Call the ad hoc version of `submitRequest()` specifying the type argument to correspond with the system property you set to define the request. The type you supply must be one of `JAVA_TYPE`, `SQL_TYPE`, or `PROCESS_TYPE`.

- As with any job request, set the appropriate system properties to be associated with the job request.

13.6.2 What Happens When You Create an Ad Hoc Request

The ad hoc `submitRequest()` returns the request identifier for the request. You can use this request identifier with runtime calls such as `updateRequestParameter()` or `getRequestDetail()` as you would with any other job request.

There is only one `submitRequest` signature that will create a request with an ad hoc job definition. The job definition ID, obtained from `RequestDetail.getJobDefn()`, is null in this case. Without an ad hoc job definition, a request cannot be considered ad hoc.

13.6.3 What You Need to Know About Ad Hoc Requests

If you want to define a schedule to use with an ad hoc request and you want to specify exclusion dates, you need to exclude the dates using the `addExclusionDate()` method for the schedule. For ad hoc requests, you cannot use a schedule that specifies exclusion dates using `addExclusion()` method for the schedule.

Currently, if the schedule is ad hoc, a check of `ExclusionDefinition` is skipped. Thus, if you use a schedule and use `addExclusion()` and submit an ad hoc job request, then Oracle Enterprise Scheduler does not use the `ExclusionsDefinition` IDs with the job request.

14

Using Subrequests

This chapter describes how to use Oracle Enterprise Scheduler subrequests, and includes the following sections:

14.1 Introduction to Using Subrequests

Oracle Enterprise Scheduler subrequests are useful when you want to process data in parallel. A request submitted from a running job is called a *subrequest*. You can submit multiple subrequests from a single parent request. The customary method of parallel execution in Oracle Enterprise Scheduler is the job set concept but there might be cases where the number of parallel processes may not be fixed in number. For example, when you want to allocate one request per million rows and in the last week 9.7 million rows have accumulated to process. In this case, you would allocate ten requests as opposed to 5 for a week that accumulated 4.6 million rows.

Oracle Enterprise Scheduler supports subrequest functionality so that a given running request (Job Request) can submit a subrequest and wait for the completion of such a request before it continues.

Oracle Enterprise Scheduler supports subrequests by exposing an overloaded subrequest method `submitRequest()`. An application that submits a job request can invoke this API to submit a subrequest.

The following restrictions apply to subrequests:

- A subrequest can be submitted only for onetime execution. No schedule can be specified. The subrequest is always treated as a "run now" request.
- Ad hoc subrequests are not supported. A subrequest must be submitted for an existing `JobDefinition` object in the application.
- Job sets are not supported for subrequests. A subrequest can only be submitted to a `JobDefinition` object. However, any running job (which may be part of a job set) can submit a subrequest.

These restrictions simplify the execution of subrequests and avoid any complications and delays in the execution of the submitting request itself.

There are different kinds of parent requests in Oracle Enterprise Scheduler, for the description in this chapter, a parent request refers to the request that is submitting a subrequest.

A subrequest follows the normal flow of a regular one-time request. However the processing of a subrequest starts only when the parent request pauses its execution. To indicate this, Oracle Enterprise Scheduler uses the `PAUSED` state. This state implies that the parent request is paused and waiting for the subrequest to finish.

Once a parent request submits a subrequest, that parent must return control back to Oracle Enterprise Scheduler, in the manner appropriate for its job type, indicating that it has paused execution. Oracle Enterprise Scheduler then sets the parent state to `PAUSED` and starts processing the subrequest. Once the subrequest finishes, Oracle Enterprise Scheduler places the parent request on the ready queue, where it remains `PAUSED`, until it is picked up by an appropriate request processor. The parent is then set to `RUNNING` state and re-run as a resumed request.

14.2 Sample Subrequest

Example 14–1 is a sample PL/SQL job that submits five subrequests. The subrequests are submitted one at a time. Each time a subrequest is submitted, the parent exits to a paused state, so that it does not consume any resources while waiting for the child request to complete. When the child completes the parent is restarted.

Example 14–1 PL/SQL Procedure Subrequest

```
procedure fusion_plsql_subreq_sample(
                                errbuf    out NOCOPY varchar2,
                                retcode   out NOCOPY varchar2,
                                no_requests  in  varchar2 default '5',
                                ) is
       req_cnt number := 0;
       sub_reqid number;
       submitted_requests varchar2(100);
       request_prop_table_t jobProp;
  begin
       -- Write log file content using FND_FILE API
       FND_FILE.PUT_LINE(FND_FILE.LOG, "About to run the sample program with sub-request
functionality");

       -- Requesting the PAUSED_STATE property set by job identifies request as
       -- having started for the first time or restarting after being paused.
       if ( ess_runtime.get_reqprop_varchar(fnd_job.job_request_id, 'PAUSED_STATE') ) is null )
       -- first time start
       then
          -- Implement the business logic of the job here.
          FND_FILE.PUT_LINE(FND_FILE.OUT, " About to submit sub-requests : " || no_requests);

          -- Loop through all the sub-requests.
          for req_cnt 1..no_requests loop
            -- Retrieve the request handle and submit the subrequest.
            sub_reqid := ess_runtime.submit_subrequest(request_handle => fnd_job.request_handle,
                                definition_name => 'sampleJob',
                                definition_package => 'samplePkg',
                                props => jobProp);
            submitted_requests := sub_reqid || ',';
          end loop;

          -- Pause the parent request.
```

```
         ess_runtime.update_reqprop_varchar(fnd_job.request_id, 'STATE', ess_job.PAUSED_STATE);

         -- Update the parent request with the state of the sub-request, enabling
         -- the job to retrieve the status during restart.
         ess_runtime.update_reqprop_int(fnd_job.request_id, 'PAUSED_STATE', submitted_requests);

      else
         -- Restart the request, retrieve job completion status and return the
         -- status to Oracle Enterprise Scheduler Service.
         errbuf := fnd_message.get("FND", "COMPLETED NORMAL");
         retcode := 0;
      end if;
end;
```

14.3 Creating and Managing Subrequests

14.3.1 How to Submit Subrequests

A subrequest can be submitted by calling the `submitRequest` API. The subrequest is set to `WAIT` state, but Oracle Enterprise Scheduler will not process the request while the parent request is running. A subrequest can be processed only once the parent request has paused.

14.3.2 How to Cancel Subrequests

There are two main ways a subrequest can be cancelled, either by the user cancelling the subrequest directly or as a result of the parent request being cancelled. For either method, the cancellation process of the subrequest is handled in the same manner as any other executable request. The difference lies in how Oracle Enterprise Scheduler treats the parent request once all pending subrequests have completed and reached a terminal state.

Oracle Enterprise Scheduler sets a subrequest that is in `WAIT` or `READY` state directly to `CANCELLED`. If a subrequest is currently running, then the subrequest is set to `CANCELLING` and Oracle Enterprise Scheduler then attempts to cancel the running executable in the manner appropriate for its job type. Usually, the subrequest ends up in `CANCELLED` state, but it may end in some other terminal state depending on the life cycle stage where the subrequest was at. The parent request remains in `PAUSED` or `CANCELLING` state until all subrequests have reached a terminal state.

If the user cancels a subrequest, then Oracle Enterprise Scheduler cancels that subrequest, as described previously. The parent request remains in `PAUSED` state until all subrequests are complete, at which point Oracle Enterprise Scheduler resumes or

restarts the parent request. This enables the parent request to handle the completion of the subrequest, possibly as cancelled, in an appropriate fashion. Cancellation of subrequests is thus not propagated upwards.

If the user cancels the parent request, Oracle Enterprise Scheduler sets the parent request to `CANCELLING` state, and then initiates a cancellation for all pending subrequests in the manner described previously. Once all subrequests have completed, Oracle Enterprise Scheduler sets the parent request to `CANCELLED`, and the parent request does not resume. Cancellation of a parent request is propagated down to its subrequests.

14.3.3 How to Hold Subrequests

A subrequest has the same life cycle as an ordinary request, and can be held when it is in `WAIT` or `READY` state. The parent request remains in `PAUSED` state while the subrequest is on hold.

14.3.4 How to Delete Subrequests

The delete operation will not be allowed on a subrequest, since it might lead to ambiguous data where the information about the subrequest will get lost. A subrequest is automatically purged when its parent request is purged.

14.3.5 How to Submit Multiple Subrequests

Oracle Enterprise Scheduler allows requests to submit multiple subrequests. A running request may submit more than one subrequest. All of these subrequests are processed by Oracle Enterprise Scheduler when the parent request pauses and goes to `PAUSED` state.

In case of multiple such subrequests, the parent request will be resumed only when all the subrequests finish.

Also it is possible to submit subrequests up to any depth. This creates nested subrequests. As such there are no restrictions on the depth of such subrequest submissions. This is kind of similar to stack push and pop operations.

14.3.6 How to Manage Paused Subrequests

- Section 14.3.6.1, "Indicating Paused Status"
- Section 14.3.6.2, "Storing the Paused State for a Parent Request"

14.3.6.1 Indicating Paused Status

A Java executable can submit subrequests using `RuntimeService.submitRequest`. After the subrequest has been submitted, the parent request must indicate to Oracle Enterprise Scheduler that it is pausing to allow the subrequest to be processed. This is accomplished by the parent throwing an `ExecutionPausedExcpetion` which causes the request to transition to `PAUSED` state.

Once the subrequests have completed, the parent request is runs again as a resumed request. The `RequestExecutionContext` can be used to determine if the executable is being run as a resumed request.

14.3.6.2 Storing the Paused State for a Parent Request

When a job execution pauses after submitting a subrequest, Oracle Enterprise Scheduler regards its execution as complete, for all intents and purposes, as

implementation-wise there is no notion of pausing an execution thread. Therefore, to resume such a paused job, Oracle Enterprise Scheduler must restart the job. In such cases, the job execution restarts from the beginning, whereas the desired behavior is to continue from the point at which execution was paused. This requires the job execution to store some kind of execution state that would represent the paused point. On resuming, the job can retrieve such a state and jump to the paused point to continue from there.

In general, it is incumbent on individual jobs to define an execution state that would allow it to resume in a deterministic way from each pause point throughout the business logic (jobs can have multiple pause points). In some cases, it can be as simple as storing the step number and jumping to that particular step on resuming, while in other cases it can be a huge data set that stores critical state for the business logic when it pauses. Oracle Enterprise Scheduler cannot provide a complete solution or framework to store the entire state.

Oracle Enterprise Scheduler provides a simplistic means for jobs to store their pause point in the form of a string that can be specified when the parent job pauses its execution. Upon resuming the parent job, the paused state value can be obtained by the parent to use as required.

Java jobs can specify a paused state string using a special `ExecutionPausedException` constructor. The state parameter represents the paused state string saved by Oracle Enterprise Scheduler when it sets the parent request to `PAUSED` state.

```
public ExecutionPausedException(String message, String state)
```

The resumed parent can retrieve the paused state value by calling `getPausedState()` on the `RequestExecutionContext` passed to the parent executable.

In case a single string value is not sufficient, the parent job can write any number of properties back into Oracle Enterprise Scheduler using `setRequestParameter()`, and retrieve those properties on resuming using `getRequestParameter()`.

14.3.7 How Subrequests Are Processed

When a subrequest is submitted, Oracle Enterprise Scheduler sets the request state to `WAIT` but in a deferred mode so it will not be dispatched until the parent request pauses.

The parent request of a Java job indicates that it is ready for subrequests to be processed by throwing `ExecutionPausedException`. When the Oracle Enterprise Scheduler receives such an exception, it sets the parent request state to `PAUSED`, publishes a system event message that the parent has paused, and then dispatches all waiting subrequests for that parent to the ready queue.

Subrequest execution follows the normal life cycle within Oracle Enterprise Scheduler. Once all subrequests for a given parent request are finished, the parent request can be resumed.

When a parent is ready to resume, Oracle Enterprise Scheduler places the parent request in the ready queue. The parent state remains as `PAUSED` while it is waiting to be picked up. Once Oracle Enterprise Scheduler picks up the parent request from the ready queue, the request state will be set to `RUNNING` and the request executable called as a resumed request.

If a request is paused without submitting any subrequests, it will be treated as if all subrequests had finished. That is, it will be placed in the ready queue, at `PAUSED` state, to be picked up for processing as a resumed request.

The final state of a subrequest does not influence how Oracle Enterprise Scheduler handles the parent request or the final state of the parent request once that parent executable has completed. When the parent request resumes, the parent request job logic can retrieve information about the subrequest, using this data as needed to determine subsequent actions. The final state of the parent request is based entirely on the state in which the parent request completed: succeeded, error, warning or cancelled.

14.3.8 How to Identify Subrequests

In Oracle Enterprise Scheduler, each request has a `RequestType` attribute. That attribute indicates whether the request is a singleton, part of a job set, a recurring request, a subrequest, and so on.

A subrequest has a `RequestType` of `SUB_REQUEST` or `UNVALIDATED_SUB_REQUEST`. An `UNVALIDATED_SUB_REQUEST` represents a subrequest that was submitted via the Oracle Enterprise Scheduler PL/SQL interface but has not yet been validated. The `RequestType` of the parent request is either `SINGLETON`, `RECUR_CHILD`, `JOBSET_STEP`, or `SUBREQUEST`. All other request types represent requests that can never be the parent of a subrequest.

The parent request ID attribute for a subrequest is the request that submitted the subrequest.

14.3.9 How to Manage Subrequests and Incompatibility

In general, a request acquires incompatibility locks when the request transition from `READY` to `RUNNING` state. Those locks are not released until the request finishes and is set to a terminal state; for example, `SUCCEEDED`, `ERROR`, `WARNING`, `CANCELLED`.

Incompatibility locks acquired by a subrequest parent remain in effect even while a parent request is in a `PAUSED` state. Any requests that were blocked by a subrequest parent remain blocked while the subrequests execute and until the parent request is resumed and finishes.

Subrequests follow all the rules of incompatibility. A subrequest therefore may get blocked if any incompatible requests are currently running when Oracle Enterprise Scheduler is ready to execute the subrequest. During such time windows, the parent request remains in `PAUSED` state while the subrequest transitions to `BLOCKED` state.

14.4 Creating a Java Procedure that Submits a Subrequest

This is an example of the Java class for a Java job type that submits subrequests. The procedure submits two subrequests, pausing between each one. Each subrequest uses the same `JobDefinition` but specifies a different value for the request parameter named `SubRequestData`. The `oracle.as.scheduler.Executable.execute` method of the parent request is called a total of three times for a given Oracle Enterprise Scheduler request and the following summaries the expected conditions and actions for each.

In the first call to execute method as a non-resumed request:

Entry condition:

RequestExecutionContext.isResumed() will be false

RequestExecutionContext.getPausedState() will be null

Method Action:

·Submit a subrequest with request parameter value of 'MyData1'

Throw ExecutionPausedException with pausedState of 'MyPausedState1"

Oracle Enterprise Scheduler will transition the request to PAUSED state, execute the subrequest, and then resume the request once the subrequest has completed.

First call to execute method as resumed request:

Entry condition:

RequestExecutionContext.isResumed() will be true

RequestExecutionContext.getPausedState() will be 'MyPausedState1'

Method Action:

Submit a subrequest with request parameter value of 'MyData2'

Throw ExecutionPausedException with pausedState of 'MyPausedState2"

Oracle Enterprise Scheduler will transition the request to PAUSED state, execute the subrequest, and then resume the request once the subrequest has completed.

Second call to execute method as resumed request:

Entry condition:

RequestExecutionContext.isResumed() will be true

RequestExecutionContext.getPausedState() will be 'MyPausedState2'

Method Action:

Exit normally, no exception.

Oracle Enterprise Scheduler will transition the request to SUCCEEDED state.

Example 14–2 shows a Java procedure with a subrequest.

Example 14–2 Java Procedure with Subrequest

```
// constants for the pausedState values
private final static String PAUSED_STATE_1 = "MyPausedState1";
private final static String PAUSED_STATE_2 = "MyPausedState2";

public class SubRequestSubmittor implements Executable {

    // method called by Oracle Enterprise Scheduler when the request is executed
    public void execute( RequestExecutionContext execCtx,
                         RequestParameters props )
           throws     ExecutionWarningException,
                      ExecutionErrorException,
                      ExecutionPausedException,
                      ExecutionCancelledException {

        long requestId =     execCtx.getRequestId();
        boolean isResumed =  execCtx.isResumed();
        String pausedState = execCtx.getPausedState();

        if (!isResumed) {

         // Method being called for first time, as non-resumed request.
         // Submit first subrequest.
         submitSubRequest(execCtx, "MyData1");
         throw new ExecutionPausedException("first subrequest", PAUSED_STATE_1);
```

```
    } else if (PAUSED_STATE_1.equals(pausedState)) {

      // Method being called for a resumed request.
      // Submit next subrequest.
      submitSubRequest(execCtx, "MyData2");
      throw new
            ExecutionPausedException("second subrequest", PAUSED_STATE_2);

    } else if (PAUSED_STATE_2.equals(pausedState)) {

        // Method being called for a resumed request.
        // All done, just return.

    } else {

      // Method being called for a resumed request.
      // Unknown paused state (should never happen).
      String msg = "Request " + requestId +
                   " was resumed with unexpected pause state " + pausedState;
        throw new ExecutionErrorException(msg);

    }
}

// Submit subrequest with request parameter having the given value.
private void submitSubRequest( RequestExecutionContext execCtx,
                               String paramValue )
    throws ExecutionErrorException{

    RuntimeService        rs = null;
    RuntimeServiceHandle  rh = null;

    try {
        rs = getRuntimeService();

        // Retrieve MetadataObjectId of the subrequest job definition
        String jobDef = "MySubRequestJobDef";
        MetadataObjectId jobDefId = getJobDefinition(jobDef);

         // Set value for the request parameter used by subrequest.
         RequestParameters rp = new RequestParameters();
         rp.add("SubRequestData", paramValue);

         // Submit the subrequest
         rh = rs.open();

         long subReqId = rs.submitRequest(rh,  execCtx,
                                          "subrequest submitter",
                                          jobDefId, rp);

    } catch (Exception e) {

        String msg = "Error while submitting subrequest for request " +
                     ExecCtx.getRequestId();
        throw new ExecutionErrorException(msg, e);

    } finally {

        if (null != rh) {
```

```
                try {
                    rs.close(rh);
                } catch (Exception e) {
                    String msg = "Error while submitting subrequest for request "
                                 + ExecCtx.getRequestId();
                    throw new ExecutionErrorException(msg, e);
                }
            }
        }
    }

    // Get RuntimeService.
    private RuntimeService getRuntime()
        throws ExecutionErrorException {
        // implementation not shown
    }

    // Retrieve MetadataObjectId for a given job definition name.
    private MetadataObjectId getJobDefinition( String jobDef )
        throws ExecutionErrorException {
        // implementation not shown
    }

}
```

14.5 Creating a PL/SQL Procedure that Submits a Subrequest

The `ESS_RUNTIME` PL/SQL package is used by an SQL job request to submit a subrequest. It also contains support to determine if the request procedure is being executed as a resumed request and retrieve the paused state string.

For a Java request, the parent request submits a subrequest using a `RuntimeService.submitRequest` method and then throws `ExecutionPausedException` when it is ready to be paused to allow the subrequest to execute.

For a SQL request, `ess_runtime.submit_subrequest` is used to submit the subrequest. The parent request must call `ess_runtime.mark_paused` when it is ready for the subrequest to run, commit the transaction and return successfully, without rasing an exception. The `mark_paused` method informs Oracle Enterprise Scheduler that, upon successful return from the parent request procedure, the parent request should be set to `PAUSED` and the subrequest allowed to execute. The `mark_paused` method supports an optional argument by which the paused state string can be specified.

It is important to note that subrequest will not be executed until the parent request has called `mark_paused`, commits, and returns normally, without raising an exception. If an exception is raised, Oracle Enterprise Scheduler will not set parent request to `PAUSED` state, but instead, it the parent state will be set to `ERROR` or `WARNING` depending on the SQL error code. Furthermore, the subrequests will be automatically `CANCELLED` and will not be executed.

Once the subrequest has finished, PL/SQL procedure for the parent request will be re-executed again as resumed request, similar to what occurs for a Java Executable.

For a Java executable, the `RequestExecutionContext` indicates if the request is being resumed and has the paused state string specified via the `ExecutionPausedException` thrown when the parent request paused.

For an SQL request, `ess_runtime.is_resumed` indicates whether the request procedure is being executed for a resumed request. The method `ess_runtime.get_paused_state`

returns the paused state string specified via the `ess_runtime.mark_paused` procedure when the request was paused.

This is an example of the PL/SQL stored procedure for a SQL job type that submits subrequests using the `ESS_RUNTIME` package. The procedure submits two subrequests, pausing between each one. Each subrequest uses the same `JobDefinition` but specifies a different value for the request parameter named `SubRequestData`. The PL/SQL stored procedure would be called a total of three times for a given Oracle Enterprise Scheduler request and the following summaries the expected conditions and actions for each.

First call to procedure as non-resumed request:

Entry condition:

`ess_runtime.is_resumed` will be false

`ess_runtime.get_paused_state` will be null

Procedure Action:

Submit a subrequest with request parameter value of 'MyData1'

Mark request as paused using paused state of 'MyPausedState1'

Exit normally, no exception

Oracle Enterprise Scheduler will transition the request to `PAUSED` state, execute the subrequest, and then resume the request once the subrequest has completed.

First call to procedure as resumed request:

Entry condition:

`ess_runtime.is_resumed` will be true

`ess_runtime.get_paused_state` will be 'MyPausedState1'

Procedure Action:

Submit a subrequest with request parameter value of 'MyData2'

Mark request as paused using paused state of 'MyPausedState2'

Exit normally, no exception

Oracle Enterprise Scheduler will transition the request to `PAUSED` state, execute the subrequest, and then resume the request once the subrequest has completed.

Second call to procedure as resumed request:

Entry condition:

`ess_runtime.is_resumed` will be true

`ess_runtime.get_paused_state` will be 'MyPausedState2'

Procedure Action:

Exit normally, no exception.

Oracle Enterprise Scheduler will transition the request to `SUCCEEDED` state.

Example 14–3 shows a PL/SQL procedure with a subrequest.

Example 14–3 PL/SQL Procedure with Subrequest

```
procedure fusion_plsql_subreq_sample(
```

```
                                errbuf    out NOCOPY varchar2,
                                retcode   out NOCOPY varchar2,
                                no_requests  in  varchar2 default '5',
                                ) is
     req_cnt number := 0;
     sub_reqid number;
     submitted_requests varchar2(100);
     request_prop_table_t jobProp;
begin
     -- Write log file content using FND_FILE API
     FND_FILE.PUT_LINE(FND_FILE.LOG, "About to run the sample program with
                      sub-request functionality");

     -- Requesting the PAUSED_STATE property set by job identifies request as
     -- having started for the first time or restarting after being paused.
     if ( ess_runtime.get_reqprop_varchar(fnd_job.job_request_id,
                                          'PAUSED_STATE') ) is null )
     -- first time start
     then
        -- Implement the business logic of the job here.
        FND_FILE.PUT_LINE(FND_FILE.OUT, " About to submit sub-requests : " ||
                         no_requests);

        -- Loop through all the sub-requests.
        for req_cnt 1..no_requests loop
          -- Retrieve the request handle and submit the subrequest.

      v_idx := v_idx + 1;
          v_req_props.extend;
          v_req_props(v_idx).prop_name := 'SubRequestData';
          v_req_props(v_idx).prop_datatype := ess_runtime.STRING_DATATYPE;
          v_req_props(v_idx).prop_value := 'MyData1';

          ess_runtime.set_submit_args(v_req_props, 'MyData1', 'MyData12',
                                      '1998-11-29')


          sub_reqid := ess_runtime.submit_subrequest(request_handle =>
                                      fnd_job.request_handle,
                                      definition_name => 'sampleJob',
                                      definition_package => 'samplePkg',
                                      props => jobProp);
          submitted_requests := sub_reqid || ',';
        end loop;

        -- Pause the parent request.
        ess_runtime.update_reqprop_varchar(fnd_job.request_id, 'STATE',
                                           ess_job.PAUSED_STATE);

        -- Update the parent request with the state of the sub-request, enabling
        -- the job to retrieve the status during restart.
        ess_runtime.update_reqprop_int(fnd_job.request_id, 'PAUSED_STATE',
                                       submitted_requests);

     else
        -- Restart the request, retrieve job completion status and return the
        -- status to Oracle Enterprise Scheduler Service.
        errbuf := fnd_message.get("FND", "COMPLETED NORMAL");
        retcode := 0;
     end if;
```

```
end;
```

15

Working with Asynchronous Java Jobs

This chapter describes how to invoke asynchronous Java jobs.

This chapter includes the following sections:

15.1 Introduction to Working with Asynchronous Java Jobs

Normally Oracle Enterprise Scheduler Java job requests run inside Oracle WebLogic Server in a dedicated thread; however, there are cases that require the ability to submit long running or non-container managed Java job requests.

Oracle Enterprise Scheduler supports asynchronous Java job invocation with the following features:

- From the Oracle Enterprise Scheduler user point of view there is no difference in scheduling asynchronous Java job invocation.
- From Oracle Enterprise Scheduler perspective, the asynchronous Java job invocation job request is submitted and is added to the queue, and returns immediately after running (and the job request enters the `RUNNING` state). Oracle Enterprise Scheduler continues operating until it hears back from the job at which point Oracle Enterprise Scheduler can apply post-processing or complete the job.
- Asynchronous Java jobs begin any variety of external jobs outside of Oracle Enterprise Scheduler. The external job, or the entity that manages it, must communicate the status of the job to Oracle Enterprise Scheduler.

15.2 Creating an Asynchronous Java Job

An Oracle Enterprise Scheduler asynchronous Java job consists of an Oracle Enterprise Scheduler job request and an external mechanism. The Oracle Enterprise Scheduler job request is implemented similarly to a standard Oracle Enterprise Scheduler Java job request; however, unlike a standard Oracle Enterprise Scheduler request, an asynchronous Java job request might not do any work, depending on the scenario. The

only purpose of an asynchronous Java job request is to trigger the external mechanism. The external mechanism executes the payload (monitoring a database, calculating pi, or any other long lived process), and must be separable from the thread running the Oracle Enterprise Scheduler Java job. The external mechanism can be a SOA composite (BPEL) or asynchronous Oracle ADF Business Components web service, another thread, JVM, machine, or some other mechanism. The means of communication between the external mechanism and the client application is left to the job owner. However, an important point for the asynchronous Java job is that the pointer to the physical Java object representing the asynchronous job is not stored in Oracle Enterprise Scheduler memory. This is because:

- The job can run for an indeterminate amount of time and caching this handle is a waste of resources
- Long lived jobs should be able to survive container restarts. Because this object is not cached and most likely garbage collected, the job should be stateless and its submitting application is responsible for maintaining the correlation between job requests and the external mechanisms running them. Oracle Enterprise Scheduler provides the job request ID and job request handle for this reason. This information should be persisted in order to survive restarts.

15.2.1 Implementing the Asynchronous Java Job Asynchronous Interface

An asynchronous Java job invocation must implement the `AsyncExecutable` interface.

15.2.2 Asynchronous Java Job execute() Method

The duty of an asynchronous Java jobs's `execute()` method is to set up the external mechanism in which the real work runs; this should start the external mechanism and then return. The asynchronous Java job invocation `execute()` method may not do any actual work. An exception can be thrown during the execute method to tell Oracle Enterprise Scheduler that this job had a problem during initialization and failed to run. The exception during the execute method does not tell Oracle Enterprise Scheduler that the actual work running on the external mechanism encountered a problem. It is the responsibility of the job owner to make sure any resources that may have been started or used are released, since Oracle Enterprise Scheduler does no further processing if it catches an exception. Assuming no exception is thrown, Oracle Enterprise Scheduler puts the job into the running state and then releases the handle on the job's object so that it may be garbage collected.

15.2.3 Invoking a Remote Job from an Asynchronous Java Job

An asynchronous Java job can set web service addressing headers to simplify the work of the remote job.

Correlation

The WSA `messageID` header is used to correlate the response message with the request. Oracle Enterprise Scheduler provides the method `RequestExecutionContext.getIdString`, which returns an ID to be used for the value of the WSA `messageID` header.

Reply Addressing

The WSA `ReplyTo` and `FaultTo` headers can be used to direct replies to the Oracle Enterprise Scheduler generic callback service. There is currently no Oracle Enterprise Scheduler support for obtaining these addresses.

15.2.4 Calling Back to Oracle Enterprise Scheduler with Status Updates

Oracle Enterprise Scheduler provides a web service operation for asynchronous callbacks, `setAsyncRequestStatus` (see the interface in Example 15–15). It requires typed information such as status and the status message, as well as the correlation information to be explicitly given.

Oracle Enterprise Scheduler provides another mechanism: a generic Java Required Files web service provider for asynchronous callbacks. The web service provider accepts payloads of any type, and messages are delivered as `SOAPMessage` objects. The WSA `relatesTo` header is extracted so as to correlate the message with the request. This header is populated with the WSA `messageID` header of the original request. The `Action` header is used to determine whether the response is due to the completion of the asynchronous job or a fault. If the response is due to a fault, the asynchronous job request status is provisionally set to `ERROR`. If the response is due to the successful completion of the asynchronous job, the asynchronous job request status is provisionally set to `SUCCESS`. The `SOAPMessage` body is extracted and converted to a string which is passed to the `Updatable.onEvent` method.

The web service provider address is `http://<host>:<port>/ess-async/essasynccallback`.

15.2.5 Updating the Asynchronous Java Job

Oracle Enterprise Scheduler provides the interface `oracle.as.scheduler.Updatable`, which allows the job request to receive update events initiated by the application code. When a job request is updated, Oracle Enterprise Scheduler determines whether the client class implements the `Updatable` interface. If the client class does implement the `Updatable` interface, it instantiates a new object of the job class and calls the `onEvent` method in the context of the MDB of the hosting application. This method accepts the request status as determined by the web service invocation and a string representing information in a format known to the job, for example, the `SOAPMessage` body from the Oracle Enterprise Scheduler web service. This method may log information or do some other processing. It then returns an `UpdateAction` object including a status and a status message.

The call to `onEvent` occurs in the context of the user associated with the execution of the request.

If the job does not implement the `Updatable` interface, the event is processed based on the status passed to `onEvent`, for example, the status determined from the asynchronous callback to Oracle Enterprise Scheduler.

For more information about the Updatable interface, see Example 15–12.

15.2.6 Notifying Oracle Enterprise Scheduler When an Asynchronous Job Completes

There are two ways to notify Oracle Enterprise Scheduler when an asynchronous job completes:

- Using a web service interface.
- Using an EJB interface.

15.2.6.1 Using the Web Service to Notify When an Asynchronous Job Completes

When you invoke the Oracle Enterprise Scheduler web service operation, `setAsyncRequestStatus`, this sets the asynchronous request's status and associated information. Associated with this operation, the following pieces of information are needed:

`setAsyncRequestStatus(String requestExecutionContext, AsyncStatus status, String statusMessage)`

Where:

- *requestExecutionContext* is a string that should be passed in as part of the initiating event. This parameter is derived from the Oracle Enterprise Scheduler job's `RequestExecutionContext` object.
- *status* is one of the following: `SUCCESS`, `ERROR`, `WARNING`, `PAUSE`, `CANCEL`, `BIZ_ERROR` or `UPDATE`.
- *statusMessage* is:
  - An error message if the status is `ERROR` or `BIZ_ERROR`.
  - A warning message if the status is `WARNING`.
  - A paused state if the status is `PAUSED`.
  - A customized string you define and have the job interpret accordingly if the status is `UPDATE`.
  - The value is ignored if the status is `SUCCESS` or `CANCEL`.

For more information about implementing a web service in a web application, see the chapters "Integrating Web Services Into a Fusion Web Application" in *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* and "Securing and Administering WebLogic Web Services" in *Oracle Fusion Middleware Security and Administrator's Guide for Web Services*.

15.2.6.2 Using EJB to Notify When an Asynchronous Job Completes

When an asynchronous Java jobs's `execute()` method is successful and the job request is running on the external mechanism, Oracle Enterprise Scheduler continues processing other jobs. When the job request is complete or encounters an error, it must communicate back to its submitting application. This communication channel is the responsibility of the agent and the client application owners. The submitting application then communicates the status of the job to Oracle Enterprise Scheduler through a local EJB. This EJB will also have a remote interface, so alternatively the external mechanism may invoke the remote EJB itself. The EJB sets the job status and does any appropriate post-processing. A helper class is provided which encapsulates all the EJB references. This helper only works when it is used inside the container since the helper uses dependency injection. The helper class contains methods for communicating success, errors, warnings, and cancellations.

15.2.7 Asynchronous Java Job AsyncCancellable Interface

If you want the job to be cancellable, you must also implement the `AsyncCancellable` interface. This interface differs from the normal cancellable interface in that its cancel method also provides the `RequestExecutionContext` and the `RequestParameters` for that job. The provided context and parameters should be used to determine which external mechanism is running the payload and then ask it to stop. The external mechanism (rather than the job's `AsyncCancellable.cancel()` implementation) notifies Oracle Enterprise Scheduler that the job has been cancelled.

Note: Currently, there is no way to terminate a running asynchronous Oracle ADF Business Components web service process.

15.2.8 Sample Asynchronous Java Job Invoking a BPEL Process Through Event Delivery Network

Using an asynchronous request you can invoke a BPEL process from Oracle Enterprise Scheduler. An asynchronous Oracle Enterprise Scheduler Java job is used to invoke the BPEL process. When the BPEL process completes, whether successfully, with an error or warning, or if it is canceled, the BPEL process notifies Oracle Enterprise Scheduler using a Oracle Enterprise Scheduler web service operation.

This method for invoking a BPEL process involves the following steps:

1. Create an asynchronous Oracle Enterprise Scheduler Java job.
2. Invoke a BPEL process from the Oracle Enterprise Scheduler Java job.
3. When the BPEL process is done, call back to the Oracle Enterprise Scheduler web service with the completion status. Use the web service operation method to inform Oracle Enterprise Scheduler of the request completion. For more information, see Section 15.2.6.1, "Using the Web Service to Notify When an Asynchronous Job Completes".
4. Once Oracle Enterprise Scheduler has the completion information, it will complete any required post-processing of the request (if required).

You can invoke the associated web service directly or you can publish an event telling the event mediator to start the BPEL process, as shown in Example 15–1.

Example 15–1 Job that Initiates a BPEL Process Through an Event Mediator

```
import oracle.as.scheduler.RequestParameters;
import oracle.as.scheduler.ExecutionCancelledException;
import oracle.as.scheduler.ExecutionErrorException;
import oracle.as.scheduler.ExecutionPausedException;
import oracle.as.scheduler.ExecutionWarningException;
import oracle.as.scheduler.RequestExecutionContext;

import javax.xml.namespace.QName;
import oracle.fabric.blocks.event.BusinessEventConnection;
import oracle.fabric.blocks.event.BusinessEventConnectionFactory;
import oracle.fabric.common.BusinessEvent;
import oracle.integration.platform.blocks.event.BusinessEventBuilder;
import
oracle.integration.platform.blocks.event.BusinessEventConnectionFactorySupport;
import oracle.xml.parser.v2.XMLDocument;
import org.w3c.dom.Element;


// Async imports
import oracle.as.scheduler.AsyncExecutable;
import oracle.as.scheduler.AsyncCancellable;


public class BPELJob implements AsyncExecutable, AsyncCancellable
{
    public BPELJob() {
    }

    public void execute(RequestExecutionContext ctx, RequestParameters params)
        throws ExecutionErrorException,
               ExecutionWarningException,
               ExecutionCancelledException,
               ExecutionPausedException
```

```
    {
        // Publish an event to the Event Mediator
        publishEvent(ctx.getRequestId() + "", ctx.toString(), "ESS_EVENT");
    }

    // Cancel

    public void cancel (RequestExecutionContext ctx,
                        RequestParameters requestParams) {
        publishEvent(ctx.getRequestId() + "", ctx.toString(), "CANCEL_ESS_EVENT");
        return;
    } // cancel

    // Event publishing

    private final String eventName = "ESSDemoEvent";
    private final String eventElement = "ESSDemoEventElement";
    private final String eventNamespace =
            "http://xmlns.oracle.com/apps/ta/essdemo/events/edl";
    private final String schemaNamespace =
            "http://xmlns.oracle.com/apps/ta/essdemo/events/schema";

    private XMLDocument buildEventPayload(String correlationId, String key, String
                                          eventType) {
        Element masterElem, childElem1, childElem2, childElem3;
        XMLDocument document = new XMLDocument();
        masterElem = document.createElementNS(schemaNamespace, eventElement);
        document.appendChild(masterElem);
        childElem1 = document.createElementNS(schemaNamespace, "requestId");
        childElem1.appendChild(document.createTextNode(correlationId));
        masterElem.appendChild(childElem1);
        childElem2 = document.createElementNS(schemaNamespace,
                                              "executionContext");
        childElem2.appendChild(document.createTextNode(key));
        masterElem.appendChild(childElem2);
        childElem3 = document.createElementNS(schemaNamespace, "eventType");
        childElem3.appendChild(document.createTextNode(eventType));
        masterElem.appendChild(childElem3);
        return document;
    }


    private void publishEvent(String correlationId, String key, String eventType)
{

            try {
                 // Get event connection
                 BusinessEventConnectionFactory cf =
                        BusinessEventConnectionFactorySupport.
                        findRelevantBusinessEventConnectionFactory(true);

                 if (cf != null) {
                     BusinessEventConnection conn =
                             cf.createBusinessEventConnection();

                     //  Build event
                     BusinessEventBuilder builder =
                             BusinessEventBuilder.newInstance();

                     // Specify the event name and namespace. In this prototype,
```

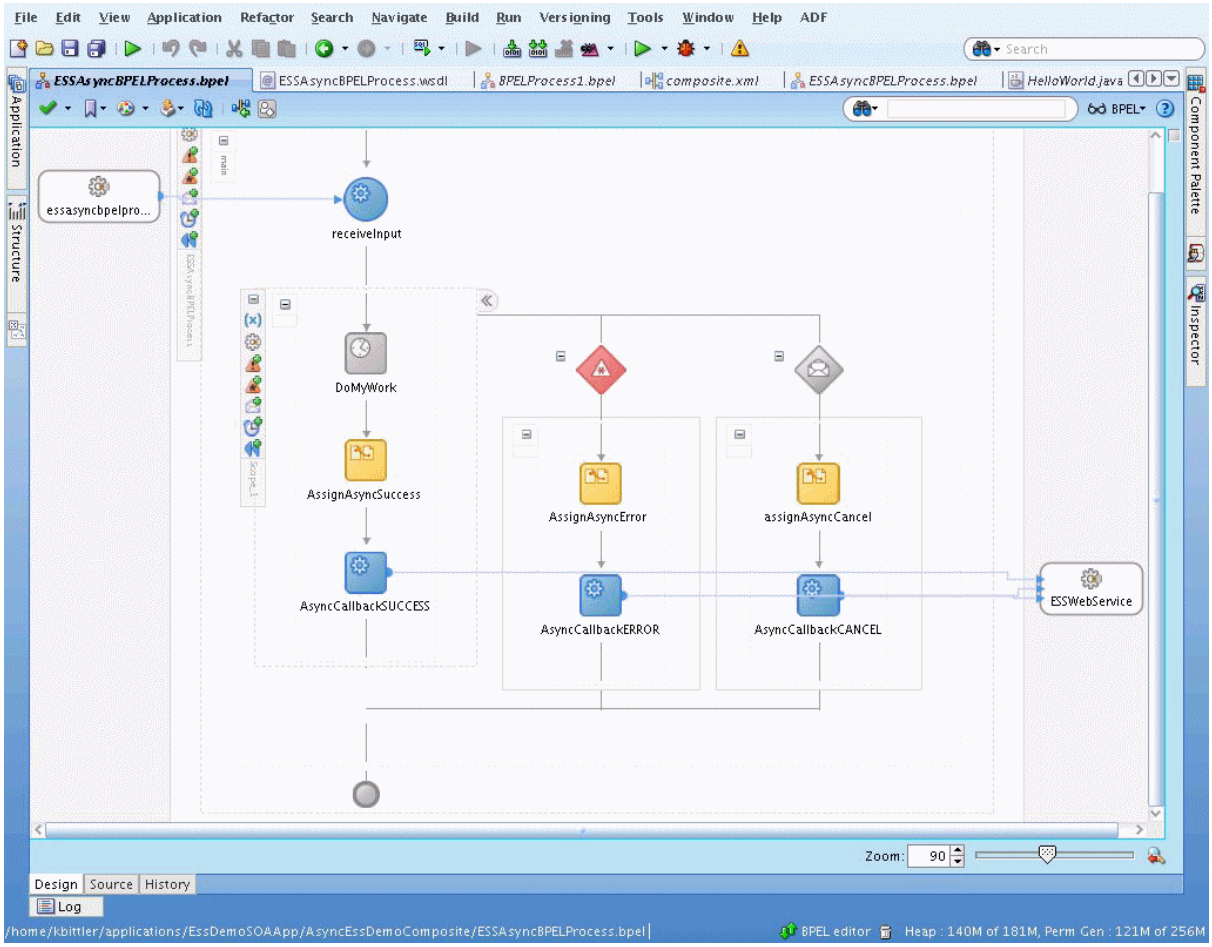
```
                    // they are constants, eventNamespace, eventName
                    builder.setEventName(new QName(eventNamespace, eventName));

                    // Specify the event payload. In this prototype, the
                    // getXMLPayload custom method constructs the payload
                    builder.setBody(buildEventPayload(correlationId, key,
                                    eventType).getDocumentElement());
                    BusinessEvent event = builder.createEvent();

                    //  Publish event
                    conn.publishEvent(event, 5);

                    // For debug only
                    System.out.println("Event was sent sucessfully");
                } else {
                    // For debug only
                    System.out.println("cf is null");
                }
            } catch (Exception exp) {
                // For debug only
                System.out.println("Failed sending event: " + exp.getMessage());
                exp.printStackTrace();
            }
        } // publishEvent
}
```

15.2.8.1 Sample BPEL Process Design Time with Oracle Enterprise Scheduler

You can use an asynchronous java job to run a BPEL process. The process initiated by an event, handled by the Event Mediator which starts the process. For an example, see Figure 15–1.

- The real work of the process is done in the DoMyWork module.
- If the work completes successfully, control will flow to AssignAsyncSuccess/AsyncCallbackSUCCESS, which invokes the Oracle Enterprise Scheduler web service callback specifying `SUCCESS` for the status and no status message.
- If the Oracle Enterprise Scheduler request is canceled, the Oracle Enterprise Scheduler job's cancel method will be called. The job object would then notify the remote job that it should be canceled. If the cancel succeeds, the remote job notifies Oracle Enterprise Scheduler using the callback mechanism, setting the status to CANCEL. In this case, control would jump to the branch on the far right.
- If a fault occurs, control will jump to the middle branch. AsyncCallbackERROR invokes the Oracle Enterprise Scheduler web service callback specifying `ERROR` for the status and an error message from the fault. AsyncCallbackCANCEL invokes the Oracle Enterprise Scheduler web service callback specifying `CANCEL` for the status and no status message.

Figure 15–1 Java Job to Call a BPEL Process and Return with Asynchronous Request



In the BPEL process, you need the web service operation values to the Oracle Enterprise Scheduler asynchronous callback, as shown in Figure 15–2, Figure 15–3, and Figure 15–4 for the `AssignAsyncError` assignment activity.

Figure 15–2 AsyncCallBackError Argument Mapping for statusMessage Element

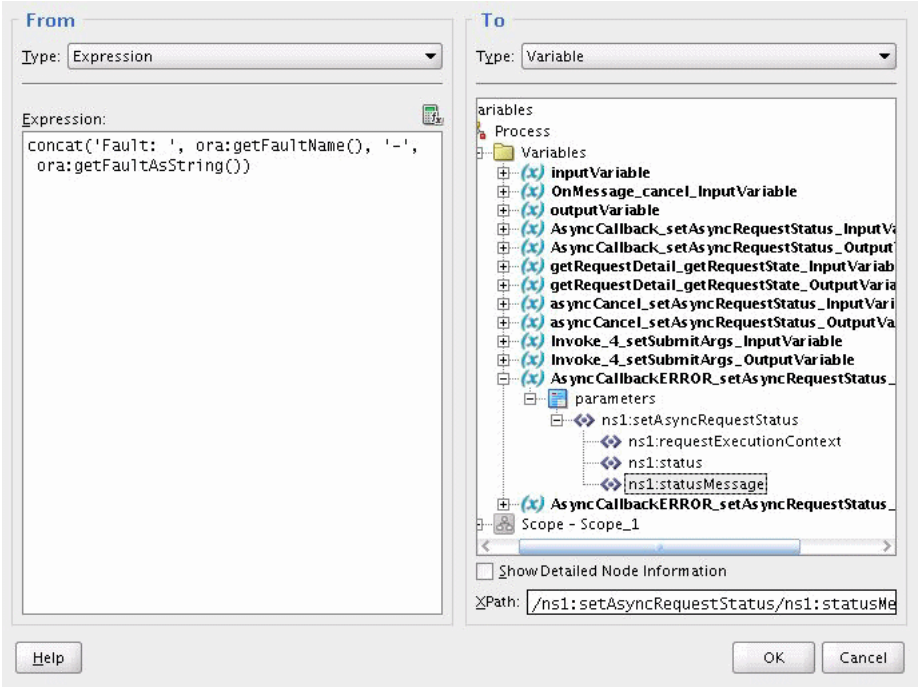


Figure 15–3 AsyncCallbackError Argument Mapping for requestExecutionContext

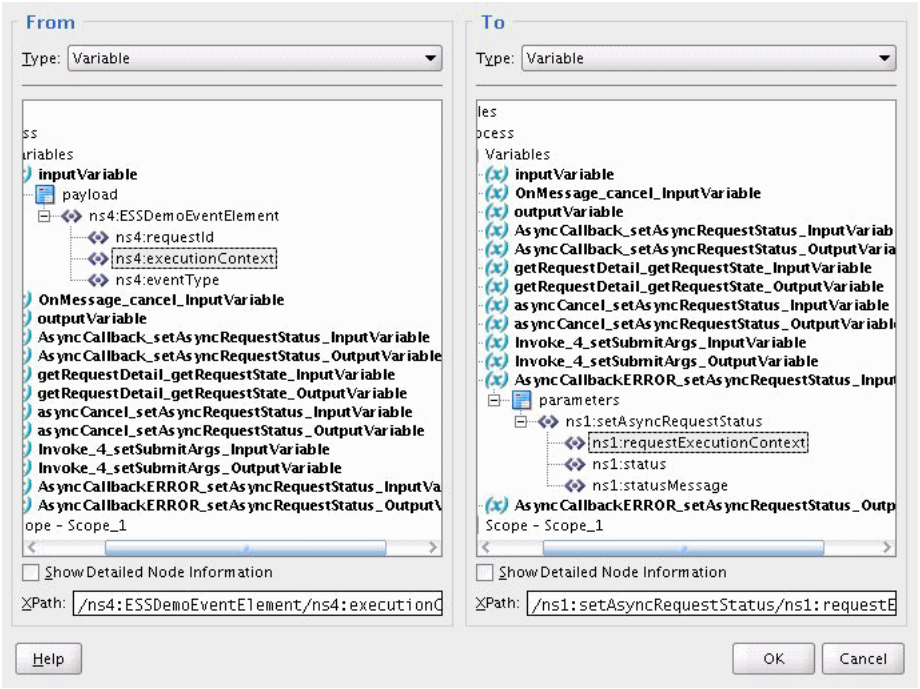
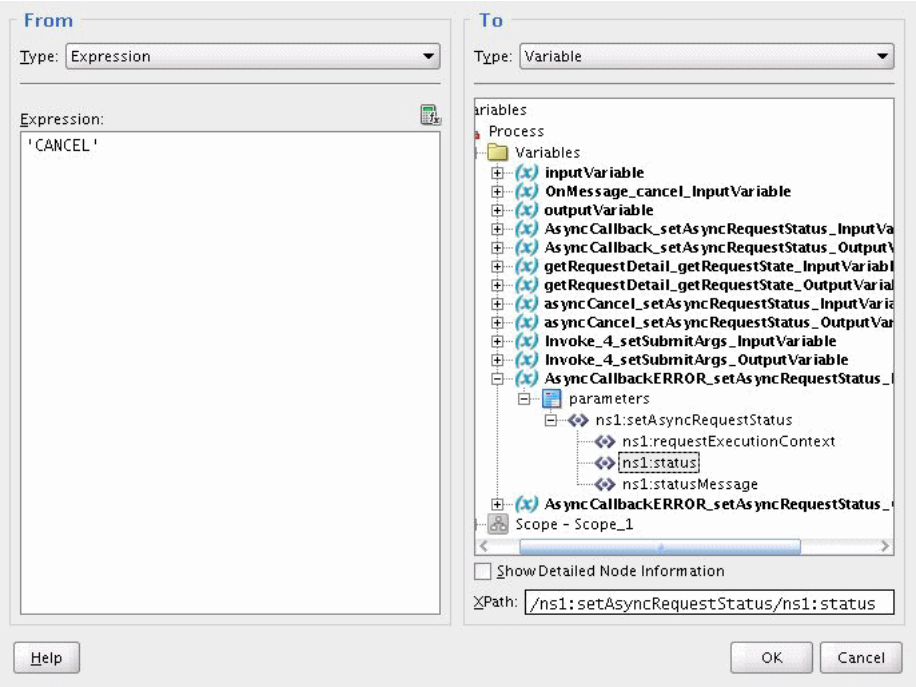


Figure 15–4 AsyncCallbackError Argument Mapping for status Element



15.3 A Use Case Illustrating the Implementation of a BPEL Process as an Asynchronous Job

Use cases for implementing a BPEL process as an asynchronous job are as follows:

- Gaining approval for a task using human workflow notifications and other SOA-specific activities.
- Notifying Oracle Enterprise Scheduler that a job has completed, while allowing other jobs to run or proceed to the next job in a set.

Design Pattern Summary

Asynchronous Oracle Enterprise Scheduler jobs are Java jobs that implement the `AsyncExecutable` interface, which is invoked by Oracle Enterprise Scheduler by implementing the `execute()` method. This method enables initiating a long running or remote task where the `execute()` method completes (such as raising a business event), while Oracle Enterprise Scheduler keeps the job in `RUNNING` status. The remote task completes and notifies Oracle Enterprise Scheduler of its completion using a status message using one of the following implementations:

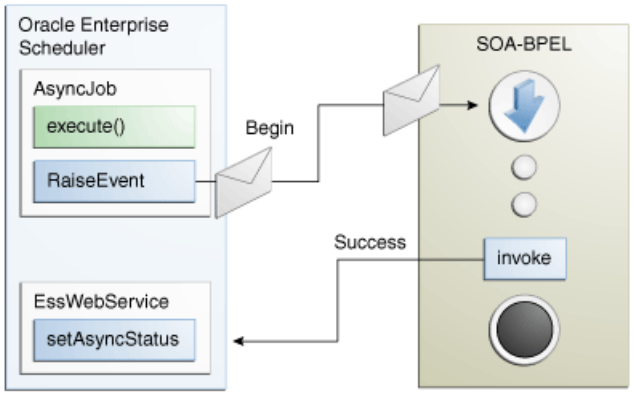
- The `RuntimeService` EJB
- The Oracle Enterprise Scheduler web service `setAsyncRequestStatus` operation.

This pattern assumes the remote task to be invoked is a BPEL process which is triggered by raising a business event in the `execute()` method of the asynchronous job. Upon termination of the process through completion, error or cancellation, the BPEL process invokes the Oracle Enterprise Scheduler web service and sets the status accordingly.

Involved Components

Oracle Enterprise Scheduler, SOA Meditator and BPEL, as shown in Figure 15–5.

Figure 15–5 BPEL Call from Oracle Enterprise Scheduler Asynchronous Job



15.3.1 Introduction to the Recommended Design Pattern

There are use cases in Oracle Fusion Applications where Oracle Enterprise Scheduler jobs need to invoke BPEL processes in a bi-directional fashion to track completion of that BPEL before moving on to other jobs. As invoking asynchronous web services from Java code (Oracle Enterprise Scheduler or Oracle ADF Business Components) in Oracle Fusion Applications is prohibited, an Oracle Enterprise Scheduler job cannot invoke an asynchronous BPEL process directly and must rely on the asynchronous job implementation type.

This approach is recommended because it leverages existing functionality in Oracle Fusion Middleware, such as events and BPEL.

15.3.2 Potential Approaches

Instead of the asynchronous Oracle Enterprise Scheduler job functionality, the following approaches are possible but not allowed:

- Invoking asynchronous web services such as Oracle ADF Business Components or BPEL via JAX-WS proxies - blocked threads and callback services are disallowed in Oracle Enterprise Scheduler.
- Raising a business event to trigger BPEL, BPEL invokes an Oracle ADF Business Components service which invokes the RuntimeService EJB to set the status, a complex and error prone procedure.

15.3.3 Use Case Summary

An Expenses system has a periodic Oracle Enterprise Scheduler job which runs to import and process expenses which requires submission of BPEL processes to leverage Human Workflow for notification and approvals. In this use case, an Oracle Enterprise Scheduler job would be responsible for importing the expenses and lines and submitting subrequests for each expense to trigger the asynchronous BPEL functionality per expense. This subrequest is implemented as an asynchronous Oracle Enterprise Scheduler job which raises a business event, completing it's Java execute() method, and staying in a running state while BPEL is initiated, submits the Human Task notification and awaits the outcome from user interaction. Once this outcome is obtained, BPEL invokes the Oracle Enterprise Scheduler web service signaling that this particular subrequest is completed.

15.4 How to Implement BPEL with an Asynchronous Job

Implementing an Oracle Enterprise Scheduler asynchronous job in BPEL requires performing the following steps:

1. Author the Oracle Enterprise Scheduler Java job to implement the `AsyncExecutable` and `AsyncCancellable` interfaces by writing `execute()` and `cancel()` methods.
2. Create the asynchronous Oracle Enterprise Scheduler job definition.
3. Design the event payload schema (XSD) and event definition (EDL) files.
4. Programmatically raise a business event from the asynchronous Oracle Enterprise Scheduler job execute() and (optionally) cancel methods.
5. Design the SOA Composite with Meditator and BPEL.
6. Add fault handling and correlated `onMessage` branch for error and cancel job status updates.

15.4.1 Use Case: Add Oracle JDeveloper Libraries

In your Oracle Enterprise Scheduler Application, be sure to add the Applications Core, and Enterprise Scheduler Service Oracle JDeveloper libraries and create a new Java class with appropriate class naming and directory structure (per standards) which will implement both the Oracle Enterprise Scheduler `AsyncExecutable` and `AsyncCancellable` interfaces. Importing both of these interfaces require you to implement the `execute()` and `cancel()` methods which Oracle Enterprise Scheduler `RuntimeService` bean invokes to initiate the desired behavior in your Oracle Enterprise Scheduler job, as shown in Example 15–2.

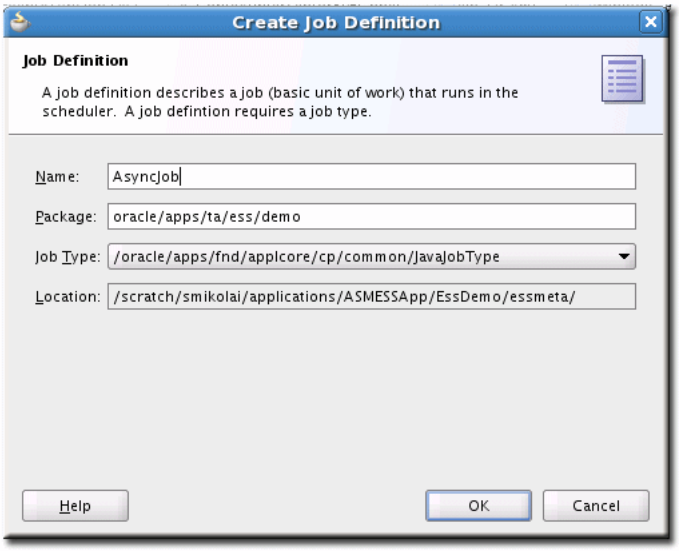
Example 15–2 Adding Oracle JDeveloper Libraries

```
public class ASMEventAsyncJob implements AsyncExecutable, AsyncCancellable {
    public ASMEventAsyncJob() {
        super();
    }

    public void execute(RequestExecutionContext ctx, RequestParameters params)
        throws ExecutionErrorException,
               ExecutionWarningException,
               ExecutionCancelledException,
               ExecutionPausedException
    {
        publishEvent(ctx.getRequestId() + "", ctx.toString(), "ESS_EVENT");
        return;

     }


    public void cancel (RequestExecutionContext ctx,
                        RequestParameters requestParams) {
        publishEvent(ctx.getRequestId() + "", ctx.toString(), "CANCEL_ESS_EVENT");
        return;
    } // cancel
```

15.4.2 Use Case: Create the Asynchronous Job Definition

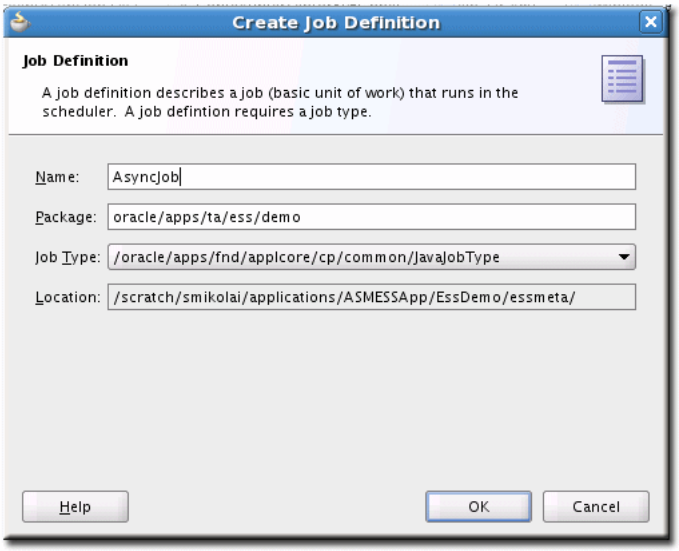
In your Oracle Enterprise Scheduler JDeveloper workspace, click "New', choose the Enterprise Scheduler Service technology group and select "Job Definition". Enter the name off your Oracle Enterprise Scheduler job definition, choose the provided "JavaJobType" and select the class build in step 1 as the overriding Java class for this job definition, as shown in Figure 15–6.

Figure 15–6 Create Job Definition



Now choose the class developed in Step 1 as the overriding Java class for this job definition, define parameters and access control as required by your use case, as shown in Figure 15–7.

Figure 15–7 Create Job Definition with Job Type Defined



15.4.3 Use Case: Design the Event Payload Schema and Event Definition Files

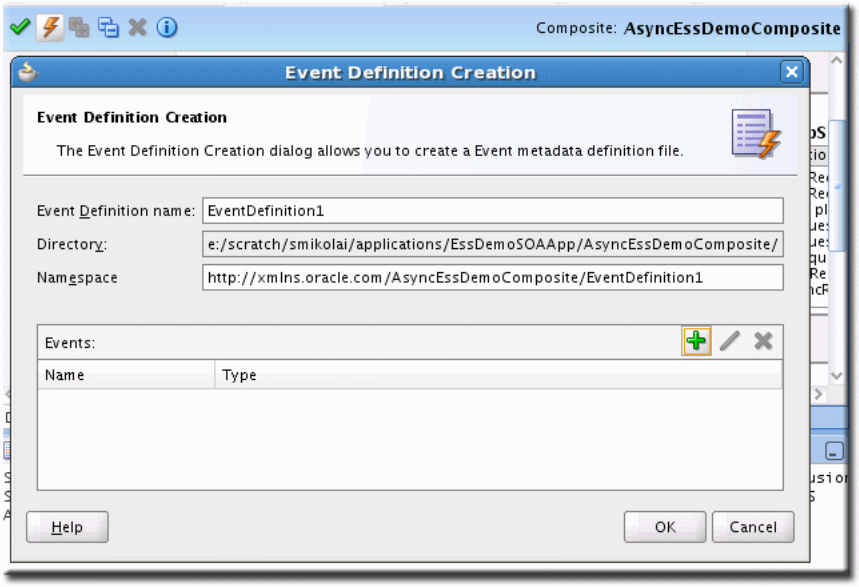
The SOA composite designer has UI features to assist in designing business event payload definitions (EDL); however your schema (.xsd) will need to be designed first. Example 15–3 shows a sample XSD file.

Example 15–3 Sample XSD File

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns="http://xmlns.oracle.com/apps/ta/essdemo/events/schema"

targetNamespace="http://xmlns.oracle.com/apps/ta/essdemo/events/schema"
     attributeFormDefault="unqualified"
            elementFormDefault="qualified">
  <xsd:element name="ESSDemoEventElement" type="ESSDemoEventElementType"/>
  <xsd:complexType name="ESSDemoEventElementType">
      <xsd:sequence>
        <xsd:element name="requestId" type="xsd:string"/>
        <xsd:element name="executionContext" type="xsd:string"/>
        <xsd:element name="eventType" type="xsd:string"/>
      </xsd:sequence>
</xsd:complexType>

</xsd:schema>
```

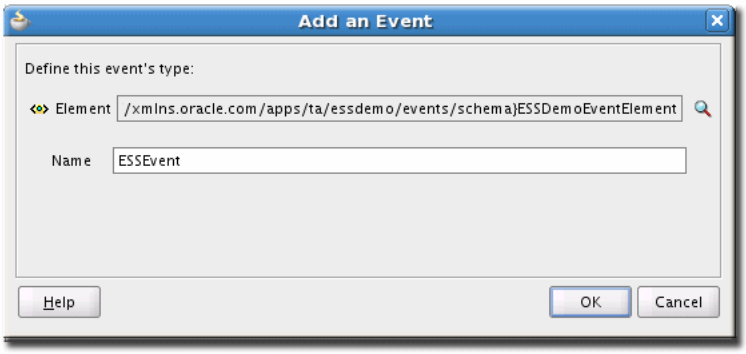
With the payload element type completed, you can either create the EDL by hand or use the event definition builder. To use the builder, open the SOA composite editor and click the lightning bolt icon at the top of the UI to open the Event Definition Creation window, as shown in Figure 15–8

Figure 15–8 Event Definition Creation



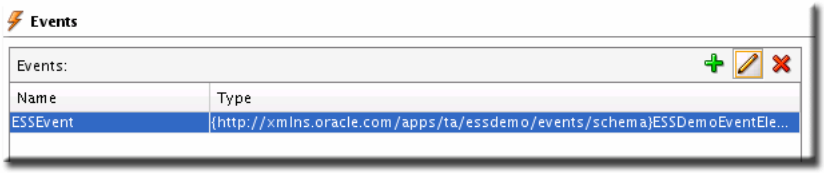
Next, assign a name and namespace per Oracle Fusion Applications naming standards and click **Add** to add a new event to this definition, as shown in Figure 15–9.

Figure 15–9 Add an Event



Click **OK**. The event definition summary displays the completed event definition. Add more events as needed for your requirements, as shown in Figure 15–10.

Figure 15–10 Events List



Example 15–4 shows a sample of the EDL file that is created.

Example 15–4 EDL File

```
 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions xmlns="http://schemas.oracle.com/events/edl"
             targetNamespace="http://xmlns.oracle.com/
             AsyncEssDemoComposite/EventDefinition1">
    <schema-import namespace="http://xmlns.oracle.com/singleString"
                   location="xsd/singleString.xsd"/>
    <schema-import namespace="http://xmlns.oracle.com/apps/ta
                   /essdemo/events/schema"
                   location="xsd/ESSDemoEventSchema.xsd"/>
    <event-definition name="ESSEvent">
        <content xmlns:ns1="http://xmlns.oracle.com/apps/ta/essdemo/events/schema"
                 element="ns1:ESSDemoEventElement"/>
    </event-definition>
</definitions>
```

15.4.4 Programmatically Raise a Business Event from the Asynchronous Job Methods

The business event raised from the asynchronous Oracle Enterprise Scheduler job must contain the request execution context's `toString()` value in order for BPEL to indicate which job is completed/cancelled/errored. Programmatically Raising Business Events from Java is covered in the "Initiating SOA from ADF" section which contains the specifics on how to write Java code that raises business events. You will need to design an event schema (.xsd) and definition (EDL) in order to declaratively build the SOA composite which will subscribe to this raised business event. Your Java code must create this XML document from scratch and it must exactly match QName values such as element and namespace attributes in the payload structure.

Note that your execute() method is invoked when Oracle Enterprise Scheduler starts to run your job, when an end user or external entity instructs Oracle Enterprise Scheduler to cancel the running job, Oracle Enterprise Scheduler sets the job's status to 'CANCELLING" and will then invoke the cancel() method. It's recommended that both methods raise events that contain similar payload types/namespaces so correlation sets can be used and the cancel event can be sent to the in-flight BPEL process in order to have it perform alternative functionality and then invoke the Oracle Enterprise Scheduler web service to set the job status to 'CANCELLED'.

This sample places the event raising code in the Oracle Enterprise Scheduler job's class code, however, the best approach is to share the code as an Oracle ADF Library which you can then import into this project to reduce duplication of publishing code.

Sample code calling the event raising code passing in `requestID` (for the BPEL correlation set to allow in-flight cancel) and the execution context's `toString()` value:

```
publishEvent(ctx.getRequestId() + "", ctx.toString(), "ESS_EVENT");
```

Sample event raising code is shown in Example 15–5.

Example 15–5 Event Raising Code

```
    private final String eventElement = "ESSDemoEventElement";
    private final String eventNamespace =
"http://xmlns.oracle.com/apps/ta/essdemo/events/edl";
    private final String schemaNamespace =
"http://xmlns.oracle.com/apps/ta/essdemo/events/schema";

    private XMLDocument buildEventPayload(String correlationId, String key, String
            eventType) {
        Element masterElem, childElem1, childElem2, childElem3;
        XMLDocument document = new XMLDocument();
        masterElem = document.createElementNS(schemaNamespace, eventElement);
        document.appendChild(masterElem);
        childElem1 = document.createElementNS(schemaNamespace, "requestId");
        childElem1.appendChild(document.createTextNode(correlationId));
        masterElem.appendChild(childElem1);
        childElem2 = document.createElementNS(schemaNamespace,
                    "executionContext");
        childElem2.appendChild(document.createTextNode(key));
        masterElem.appendChild(childElem2);
        childElem3 = document.createElementNS(schemaNamespace, "eventType");
        childElem3.appendChild(document.createTextNode(eventType));
        masterElem.appendChild(childElem3);
        return document;
    }


    public void publishEvent(String correlationId, String key, String eventType) {
            // Determine whether we are outside of a JTA transaction
            try {
                 // Get event connection
                 BusinessEventConnectionFactory cf =
BusinessEventConnectionFactorySupport.findRelevantBusinessEventConnectionFactory
        (true);

                 if (cf != null) {
                     BusinessEventConnection conn =
                             cf.createBusinessEventConnection();

                     //  Build event
```

```
                    BusinessEventBuilder builder =
BusinessEventBuilder.newInstance();

                    // Specify the event name and namespace. In this prototype,
                    // they are constants, eventNamespace, eventName
                    builder.setEventName(new QName(eventNamespace, eventName));

                    // Specify the event payload. In this prototype, the
                    // getXMLPayload custom method constructs the payload
                    builder.setBody(buildEventPayload(correlationId, key,
                            eventType).getDocumentElement());
                    BusinessEvent event = builder.createEvent();

                    //  Publish event
                    conn.publishEvent(event, 5);

                    // For debug only
                    System.out.println("Event was sent sucessfully");
                    conn.close();
                } else {
                    // For debug only
                    System.out.println("cf is null");
                }
            } catch (Exception exp) {
                // For debug only
                System.out.println("Failed sending event: " + exp.getMessage());
                exp.printStackTrace();
            }
        } // publishEvent
}
```

15.4.5 Design the SOA Composite with Meditator and BPEL

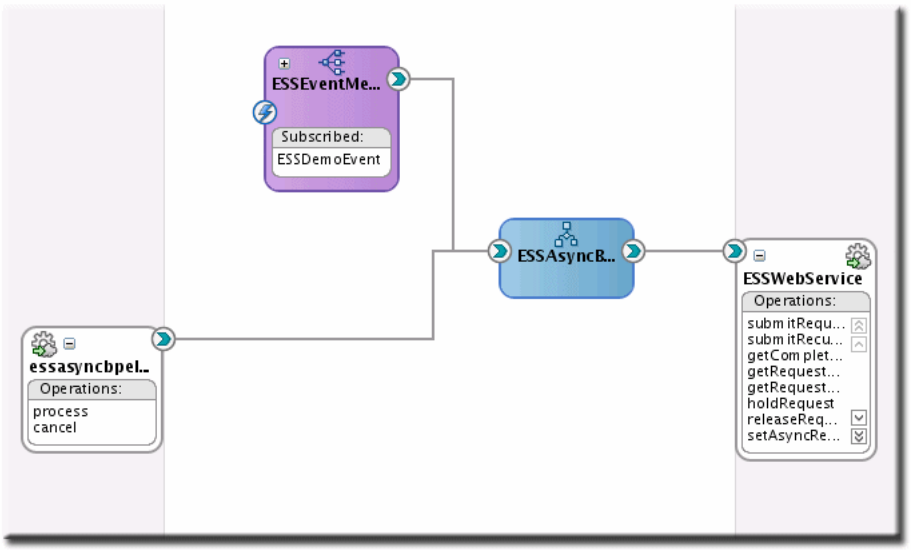
Since this use case depends on BPEL functionality it is necessary to build a SOA composite which contains a Mediator for event subscription which can then transform the payload and initiate the BPEL process.

In your SOA workspace, create a new SOA composite. To setup the composite for this pattern, add a Mediator that subscribes to your Oracle Enterprise Scheduler raised event and wire it to a BPEL process. Add a service reference to the Oracle Enterprise Scheduler web service WSDL. For example,

```
http://myhost.com:7001/ess/esswebservice?WSDL
```

Continue to build the required functionality in the BPEL process using one or more nested scopes. Bear in mind that your functionality should reside within at least one primary scope on which you can add an `onMessage` event (for in-flight cancel message receipt) and fault handler branches, as shown in Figure 15–11.

Figure 15–11 Composite with BPEL and ESSWebService



For more information about invoking the Oracle Enterprise Scheduler web service, see Chapter 10, "Using the Oracle Enterprise Scheduler Web Service." For more information about subscribing to an event, see the chapter "Cross Family Business Event Subscription Pattern" in *Oracle Fusion Applications Developer's Guide*.

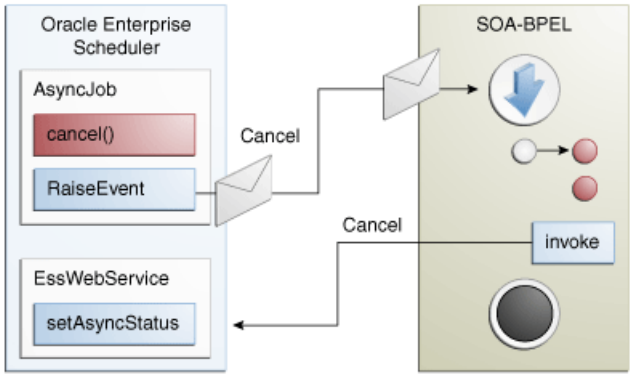
15.4.6 Add Fault Handling and Correlated onMessage Branch for Error and Cancel Job

Oracle Enterprise Scheduler does not perform any sort of heartbeat monitoring of asynchronous Oracle Enterprise Scheduler jobs after the `execute()` method's Java code has completed. Once the job is submitted it exists in a `RUNNING` state within the Oracle Enterprise Scheduler infrastructure until the remote job code, BPEL, or end user interacts with Oracle Enterprise Scheduler directly to set the status of the job. Because of this caveat, developers need to design their BPEL processes to handle, at a minimum, two types of scenarios that will most often occur in the life span of an Oracle Enterprise Scheduler job and, whenever possible, push that state information back to Oracle Enterprise Scheduler so monitoring UIs can reflect the correct state of the job to end users.

BPEL Handling Cancellation:

For example, if the end user interacts with the monitoring UI and requests that the job be cancelled Oracle Enterprise Scheduler will then update the job's status to `CANCELLING` and wait for the remote functionality to tidy up and confirm that it has cancelled, as shown in Figure 15–12.

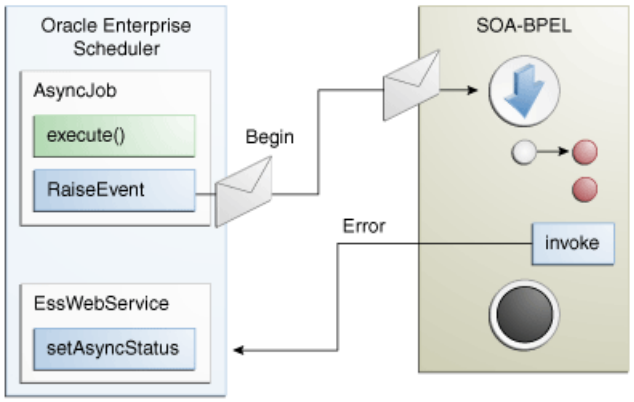
Figure 15–12 BPEL Handling Cancellation



BPEL Handling Error

Additionally, when the remote functionality encounters a failure, the responsibility to notify Oracle Enterprise Scheduler of this failure falls on the shoulders of the remote functionality (in this case, BPEL) to notify Oracle Enterprise Scheduler that the job's status is `ERROR` and provide a status message in addition to any logging that was performed. This is illustrated in Figure 15–13.

Figure 15–13 BPEL Handling Error



In order to acknowledge cancellation and arbitrate proper status back to the Oracle Enterprise Scheduler infrastructure, BPEL must be designed within a certain layout to support receipt of the incoming cancellation message and trapping of any failures such that, in either case, the Oracle Enterprise Scheduler subsystem can be updated. For this purpose, in the BPEL Process, there should be at least one scope which will contain the functionality for this asynchronous job. This will allow sufficient control for handling cancel and error states which must then be sent to the Oracle Enterprise Scheduler web service in order to update the job's status in the Oracle Enterprise Scheduler runtime.

To build the basic process flow to support these states, the following steps should be completed in order:

1. Create the correlation set and flag it for imitate on the incoming Receive activity.
2. Create the `onMessage` branch with use of correlation set created in sub-step 1.
3. Create the fault handling branch.

4. Populate the `onMessage` and fault handling branches with cleanup activities as needed and invoke the Oracle Enterprise Scheduler web service with appropriate status.

15.4.6.1 Create Correlation Set and Define Initiate Activity

In order to support receiving the cancel event while the BPEL process is in the middle of performing other activities or waiting for an asynchronous callback the process must be configured with a correlation set. A correlation set is key value that is built from one or more incoming payload attributes which are used to uniquely identify the BPEL process to the BPEL engine whereby additional service requests that contain matching sets of attributes can be routed to the process that is currently running instead of initiating a new one. While correlation is standard functionality used for asynchronous request responses, it can also be used to change the flow of execution in a BPEL process through scope-level `onMessage` branches.

To setup the correlation set, open the BPEL process in the designer, double-click the Receive activity and click the correlations tab.

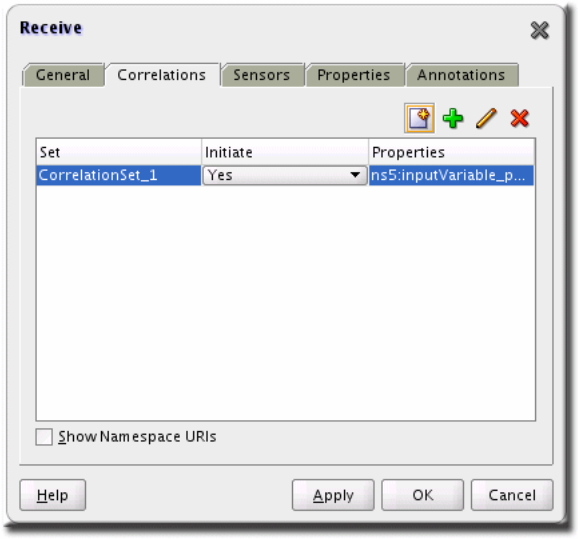
Note that coarctation sets have an "initiate" property which indicates which activity will be the starting point for this correlation set's life cycle. In this case, the start of the BPEL process will be the point at which the correlation set's life cycle should begin allowing correlated events to route to this process at any point during the process.

To create a correlation set:

- Click the "New" icon in the Correlations tab of any Receive, Invoke or `onMessage` activity and provide a name for the correlation set.
- Next, click "Add" to define one or more property attributes to use as the correlation key.
- Choose a variable attribute as the set property and click "OK".
- Repeat steps 2 and 3 as necessary to build an attribute set that will always be unique.
- Set the initiate flag on the correlation to "Yes" on the activity for which the correlation set's life cycle should begin.

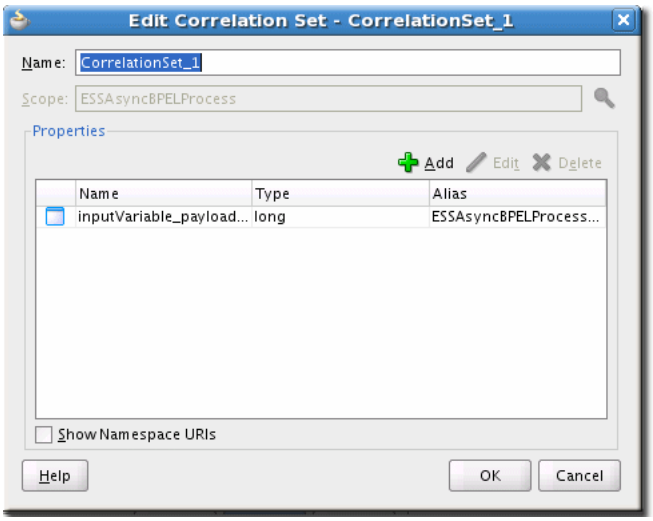
Primary (first) Receive Activity with Defined Correlation Set and "Initiate" flagged to "Yes", as shown in Figure 15–14.

Figure 15–14 Correlations for Receive Activity



CorrelationSet_1 definition with a single property defined (define more as needed to ensure unique keys are created), as shown in Figure 15–15.

Figure 15–15 Edit Correlation Set



15.4.6.2 Create the onMessage Branch with Use of Correlation Set

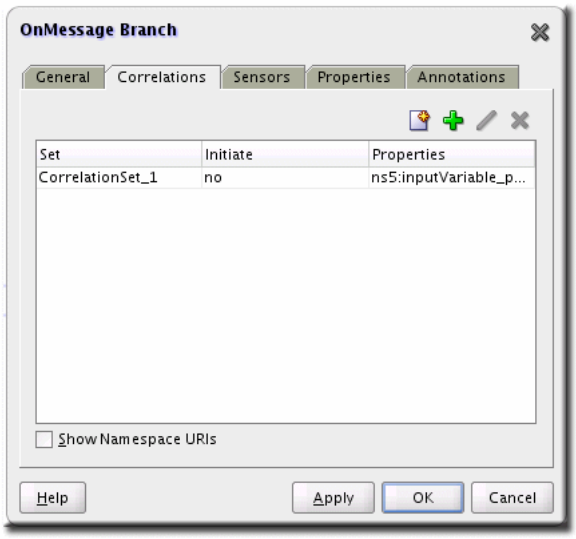
Once the correlation set has been defined and set for initiate it's now possible to create the `onMessage` branch on the scope which will contain the activities necessary to accept the incoming cancellation message, perform any compensation or cleanup and then assign the job's completion status to `CANCEL`.

> **Note:** At this point, the `onMessage` branch could contain the invoke activity or finish allowing a higher order scope to perform the invoke, reducing the overall number of necessary invoke activities in the flow.

The following steps guide you through adding the previously created correlation set to the `onMessage` branch activity, as shown in Figure 15–16.

- On the nested scope containing the process functionality, click the 'Add onMessage branch' icon which should create a new flow off to the side of the scope.
- Double-click the `onMessage` branch activity to open the activity editor.
- Choose the "Correlations" tab.
- Click the Add '+' icon and select the previously created correlation set ensuring that the initiate flag is set to 'No' and click "Ok".

Figure 15–16 BPEL OnMessage Branch



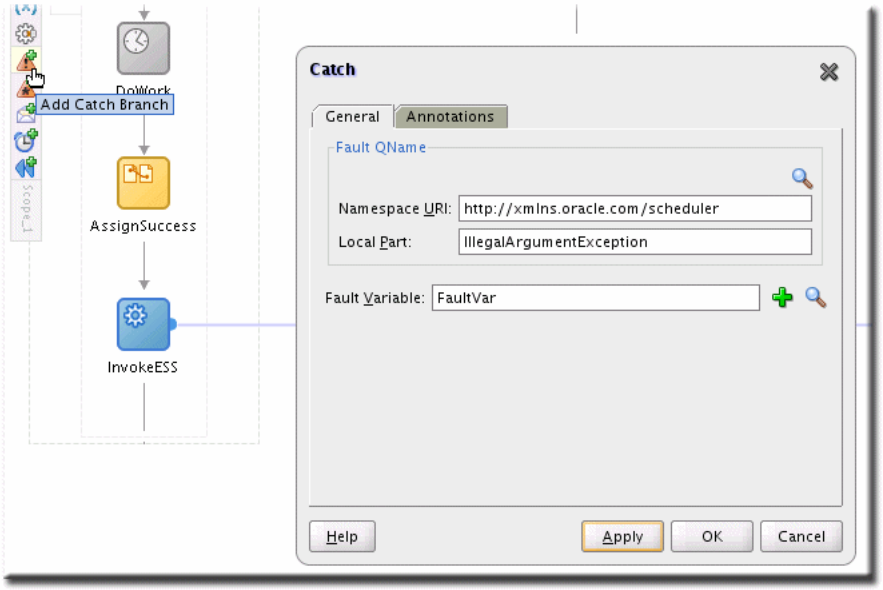
15.4.6.3 Create the Fault Branch

Through the course of performing the various activities in the nested work scope BPEL may encounter faults from business services or system functionality. In most cases, business services will define one or more WSDL-defined faults that can be thrown back to the calling process. Ordinarily, a BPEL CatchAll fault branch will trap any and all faults that are raised regardless of their type and origin but there may be cases where product teams have requirements to perform different sets of behavior in response to specific business faults. In cases where it's desirable to perform unique compensation behavior for specific business faults, the developer should create a named fault handling branch for each WSDL-defined fault. In addition to these named fault handler branches, it is still necessary to add a CatchAll fault handling branch to trap any system level or unmanaged faults that are raised from the scope.

Click the CatchFault and CatchAll scope icons to create the desired fault handling branches, then double-click the named fault handling branches and define the named fault those branches will catch.

Note the available status, as shown in Figure 15–17.

Figure 15–17 Catch Branch for BPEL Flow



15.4.6.4 Populate the onMessage and Fault Branch

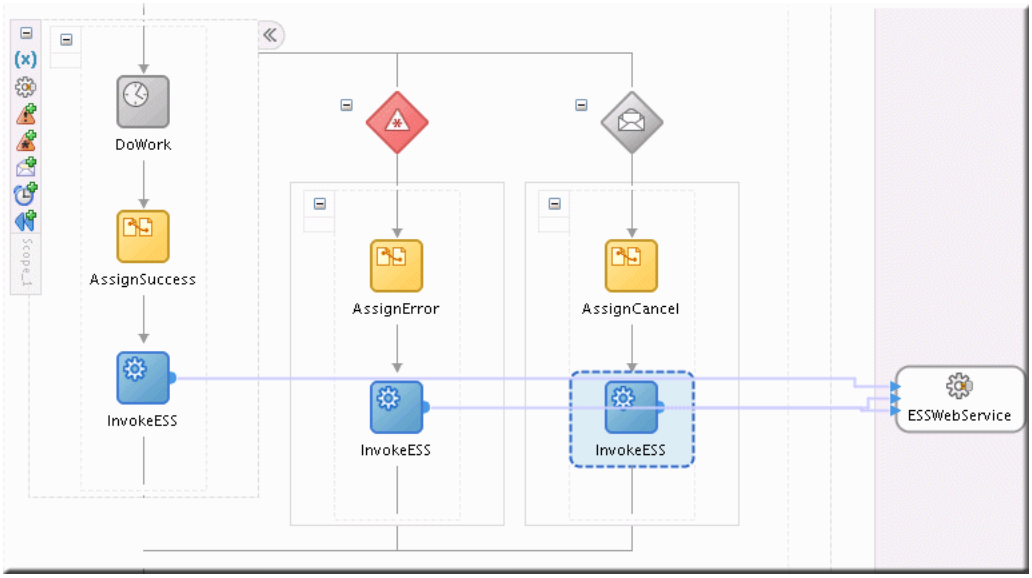
You need to populate the onMessage and Fault branch with cleanup activities as needed and invoke Oracle Enterprise Scheduler web service with appropriate status.

In the event of a fault or receipt of the cancellation message through the onMessage branch the Oracle Enterprise Scheduler infrastructure needs to be updated directly via the Oracle Enterprise Scheduler web service in order to reflect the job's status and status message properly in the monitoring UIs. As a result, each fault handling or onMessage branch should assign the correct status and status message value to the Oracle Enterprise Scheduler web service invoke variable and optionally contain the invoke activity or, by design, return to a higher order scope which is designed to be agnostic to the outcome of the job status and will perform the invoke activity on the Oracle Enterprise Scheduler web service before completing.

Additionally, drag activities into the onMessage and fault branches as needed to cleanup/log/compensate.

Example scope with onMessage and Fault handling branches is shown in Figure 15–18.

Figure 15–18 Entire BPEL Flow Sample



15.4.7 Validating the Deployment

To test that the functionality works you must perform the following sequence of steps:

1. Turn on the EDN-DB-LOG page by navigating to the following site to make sure it reads "Log is Enabled". If not, click the link for "Enable",

   `http://host:port/soa-infra/events/edn-db-log`

2. Submit your job through your own application, Fusion Applications Control the task flow user interface for submitting job requests and confirm that the status of the job is `RUNNING`.

3. Your event should immediately show up in the EDN-DB-LOG page. Check for this event payload, as shown in Example 15–6.

Example 15–6 Event Payload

```
Example:Enqueing event:
http://xmlns.oracle.com/apps/ta/essdemo/events/edl::ESSDemoEvent from J
Body: <business-event
xmlns:ns="http://xmlns.oracle.com/apps/ta/essdemo/events/edl"
xmlns="http://oracle.com/fabric/businessEvent">
<name>ns:ESSDemoEvent</name>
<id>df8e34c1-4c65-4379-b9be-2c692670ebbe</id>
<content>
<ESSDemoEventElement
xmlns="http://xmlns.oracle.com/apps/ta/essdemo/events/schema">
<requestId>3</requestId>
<executionContext>3, false, null, 6A4A16757764CD60E0402382B7703F44,
12</executionContext>
<eventType>ESS_EVENT</eventType>
</ESSDemoEventElement>
</content>
</business-event>
Subject name:
Enqueing complete
Enqueing event: http://xmlns.oracle.com/apps/ta/essdemo/events/edl::ESSDemoEvent
```

```
from J
Body: <business-event
xmlns:ns="http://xmlns.oracle.com/apps/ta/essdemo/events/edl"
xmlns="http://oracle.com/fabric/businessEvent">
<name>ns:ESSDemoEvent</name>
<id>a4104da8-5579-4434-ab8b-d31a226e3b0f</id>
<content>
<ESSDemoEventElement
xmlns="http://xmlns.oracle.com/apps/ta/essdemo/events/schema">
<requestId>4</requestId>
<executionContext>4, false, null, 6A4A2BC7E5477C60E0402382B77041C9,
12</executionContext>
<eventType>ESS_EVENT</eventType>
</ESSDemoEventElement>
</content>
</business-event>
```

4. Your subscribing mediator will have been triggered, you can check Fusion Applications Control ($DOMAIN_HOME/as.log) or soa-diagnostic logs ($DOMAIN_HOME/servers/<serverName>logs/<serverName>.log) to see any mediator activity as a result of your event, as shown in Example 15–7.

Example 15–7 Mediator Activity

```
INFO: MediatorServiceEngine received an event =
{http://xmlns.oracle.com/apps/ta/ess/demo/events/edl}ESSDemoEvent
Apr 17, 2009 1:57:26 PM oracle.tip.mediator.common.persistence.MediatorPersistor
persistCallback
INFO: No call back info set in incoming message
Apr 17, 2009 1:57:26 PM oracle.tip.mediator.common.persistence.MediatorPersistor
persistCallback
INFO: Message properties :
{id=041ecfcf-8b73-4055-b5c0-0b89af04f425, tracking.compositeInstanceId=50003,
tracking.ecid=0000I2pqzVCBLA5xrOI7SY19uEYF00004g:47979}
Apr 17, 2009 1:57:26 PM oracle.tip.mediator.dispatch.InitialMessageDispatcher
dispatch
INFO: Executing Routing Service..
Apr 17, 2009 1:57:26 PM oracle.tip.mediator.dispatch.InitialMessageDispatcher
processCases
INFO: Unfiltered case list size :1
Apr 17, 2009 1:57:26 PM oracle.tip.mediator.monitor.MediatorActivityMonitor
createMediatorCaseInstance
INFO: Creating case instance with name :ESSDemoProcess.essdemoprocess_
client.process
Apr 17, 2009 1:57:26 PM oracle.tip.mediator.dispatch.InitialMessageDispatcher
processCase
INFO: Immediate case
{ESSDemoProcess.adedemoprocess_client.process}with case id :
{5B52B4A02B9211DEAF64D3EF6E2FB21D}will be executed
Apr 17, 2009 1:57:26 PM oracle.tip.mediator.service.filter.FilterFactory
createFilterHandler
INFO: No Condition defined
```

5. Check the Oracle Enterprise Manager Fusion Middleware Control Console for an instance of your SOA composite and check for errors.

   ```
   http://host:port/em
   ```

6. If your BPEL process has not errored and is expecting a response from the human workflow notification, navigate to the worklist, login as the assigned approver and approve or reject the notification per your design requirements.
7. From here, the BPEL process should complete and invoke the Oracle Enterprise Scheduler web service to set the job's completion status and status message. Check the monitoring UI diagnostic logs for stack traces and log messages.
8. Additionally, you can check the `REQUEST_HISTORY` table in the Oracle Enterprise Scheduler schema for details on your job's state.

15.4.8 Troubleshooting the Use Case

To troubleshoot issues with the Oracle ADF UI functionality such as the monitoring and submission task flows use the server's console log, applications log and server diagnostic logs for information on what is failing and why.

To troubleshoot issues with the events functionality, such as the event not reaching the BPEL process with request execution context intact, use the EDN database log page (`http://host:post/soa-infra/events/edn-db-log`) to inspect the event payload and carefully compare it to the schema definition, even slight mismatches can cause the transformation to 'succeed' but produce an skeleton payload to BPEL which is missing any request context values. Oracle JDeveloper and third-party tools can be used to validate the schema of the event payload and debug the transformation against that payload.

To troubleshoot the mediator, BPEL SOA functionality, use the Oracle Enterprise Manager and server console or diagnostics log files for diagnostics and AppsLogger Sensor variables for logging.

For more information about troubleshooting Oracle Enterprise Scheduler at run time, see the chapter "Troubleshooting Oracle Enterprise Scheduler" in *Oracle Fusion Applications Administrator's Guide*.

15.5 Handling Time Outs and Recovery for Asynchronous Jobs

Oracle Enterprise Scheduler asynchronous Java jobs depend on the remote job to update Oracle Enterprise Scheduler with its completion status before it can finish processing the request. Due to the nature of remote communication, there may be cases where Oracle Enterprise Scheduler does not receive the remote request status because of network failures, and so on. In these cases, the request may be stuck in a non-terminal state.

Transitioning a timed out request to a terminal state is important as it:

- Frees any incompatibility locks held by that job request.
- If the job request is a job set step, allows the job set to continue.
- If the request is a subrequest, allows the parent request to resume.
- Allows the job request to be deleted or purged.

15.5.1 Asynchronous Request Time Outs

An Oracle Enterprise Scheduler system property, `SystemProperty.ASYNC_REQUEST_TIMEOUT`, enables setting job request time out values for asynchronous Java jobs. By default, the property is not enabled, such that its value is less than or equal to zero.

The property may be set in the job definition metadata or when the job request is submitted. The value represents the duration, in minutes, from the time the job request begins local execution until a terminal asynchronous job status is received from the remote job.

15.5.1.1 Setting the TIme Out Value

For a given asynchronous job request, set the system property `SystemProperty.ASYNC_REQUEST_TIMEOUT` to a value greater than 0.

15.5.1.2 Discovering the Asynchronous Job Requests that Have Timed Out

For a given request, `RequestDetail.isTimedOut` indicates the status of the time out. Requests that have timed out can be discovered using the query shown in Example 15–8.

Example 15–8 Indicating the Time Out Status

```
Filter timedOutRunningFilter = new Filter(
    RuntimeService.QueryField.TIMED_OUT.fieldName(),
    Filter.Comparator.EQUALS,
    Boolean.TRUE)
.and(
    RuntimeService.QueryField.STATE.fieldName(),
    Filter.Comparator.EQUALS,
    State.RUNNING.value());
runtimeService.queryRequests(handle, timedOutRunningFilter, null, true);
```

A similar query can be run using `REQUEST_HISTORY_VIEW`, as shown in Example 15–9.

Example 15–9 Using REQUEST_HISTORY_VIEW

```
SELECT requestId FROM request_history_view WHERE timedout='Y' AND state=3;
```

15.5.1.3 Completing Asynchronous Requests without a Time Out

In the absence of a time out value, asynchronous requests whose remote job has completed without delivering the status to Oracle Enterprise Scheduler may be completed directly using `RuntimeMXBean.completeAsyncRequest`. Because there is no time out value to flag the request as needing attention, you must carefully track requests without time outs.

For more information about managing job requests without time outs, see the chapter "Troubleshooting Oracle Enterprise Scheduler" in *Oracle Fusion Applications Administrator's Guide*.

15.5.1.4 What Happens When an Asynchronous Job Request Times Out

Oracle Enterprise Scheduler periodically checks for asynchronous job requests on which the property `SystemProperty.ASYNC_REQUEST_TIMEOUT` has been set. When the time has exceeded without a terminal status having been received, the job is flagged as timed out. Otherwise, the job state is unaffected, and remains in a `RUNNING` state. Meanwhile, Oracle Enterprise Scheduler continues to accept status updates from the remote job. The flag indicates that the status of the remote job may need to be investigated.

15.5.2 Handling Asynchronous Jobs Marked for Manual Recovery

If the remote job completed but its status was not delivered to Oracle Enterprise Scheduler, you can complete the request manually.

In some cases, the status of a job status cannot be determined automatically, such that it is unknown whether or not a job is executing, for example. If the job is executing, the job request must not transition to a terminal state. If the job does transition to a terminal state, incompatibility locks could be released, possibly causing incompatible job requests to run simultaneously.

For example:

- An asynchronous Java job encounters an error when starting a remote service, such that it is unclear that the remote service has actually been invoked. The job request must not go to an error state until it is determined whether the remote job is running. If the job might be running, the job should throw an `oracle.as.scheduler.ExecutionManualRecoveryException` to indicate to Oracle Enterprise Scheduler that the job request must transition to `ERROR_MANUAL_RECOVERY` state.
- An Oracle Enterprise Scheduler asynchronous Java job throws a `java.lang.Error` which does not indicate to Oracle Enterprise Scheduler whether the remote service has been invoked.
- A spawned job is running in a clustered environment, with the job request running on Oracle Enterprise Scheduler instance1. The Oracle Enterprise Scheduler instance1 server goes down, along with the associated Perl agent. If instance1 is not going to recover for a while, the job status is unknown. The property `State.ERROR_MANUAL_RECOVERY` is used for this type of situation. This is a non-terminal state that suspends processing on a job request until a recovery operation is manually invoked. Any incompatibility locks acquired will be retained until manual recovery completes.

For more information about handling asynchronous jobs marked for manual recovery, see the section "Handling Stuck Asynchronous Jobs Requiring Manual Recovery" in the chapter "Troubleshooting Oracle Enterprise Scheduler" in *Oracle Fusion Applications Administrator's Guide*.

15.5.3 Using RecoverRequest to Manually Recover a Job Request

If some job requests are stuck in an incomplete state, it should first be determined whether the job requests can complete by normal means. For instance, if a job request is in `RUNNING` state, it may be for an asynchronous Java job running remotely. If the remote job is unable to respond, then you must try to cancel the job request. This transitions the job request to `CANCELLING` state. If the job request does not transition to `CANCELLED` state, then it may be a candidate for recovery.

All child requests of the request to be recovered must have already completed, meaning that its process phase is `ProcessPhase.Complete`. You can retrieve the process phase by executing `RequestDetail.getProcessPhase()`.

Using `RuntimeService.queryRequests`, you can run a query to determine incomplete child requests using the filter shown in Example 15–10.

Example 15–10 Filtering for Incomplete Child Requests

```
Filter filter =
            new Filter(RuntimeService.QueryField.ABSPARENTID.fieldName(),
                       Filter.Comparator.EQUALS, requestId)
```

```
            .and(RuntimeService.QueryField.REQUESTID.fieldName(),
                 Filter.Comparator.NOT_EQUALS, requestId)
            .and(RuntimeService.QueryField.PROCESS_PHASE.fieldName(),
                 Filter.Comparator.NOT_EQUALS,
                 ProcessPhase.Complete.value());
```

If it is determined that any child requests require manual recovery, then invoke `recoverRequest` for those jobs first. If `recoverRequest` is invoked on a parent request with incomplete child requests, an exception will be thrown. The exception message will list child requests that are incomplete. Example 15–11 shows the `recoverRequest` syntax.

Example 15–11 recoverRequest

```
/**
  * Attempts to force a request to complete under certain conditions.
  * <p>
  * 1. The request must already by in a terminal state, {@code
  *    State.CANCELLING}, or {@code State.ERROR_MANUAL_RECOVER}.
  *    If a request is in another state,
  *    {@code RuntimeService.cancel} must be called first. If the
  *    request does not eventually transition to {@code State.CANCELLED},
  *    then this operation may be invoked on the request.
  * 2. All child requests of the given request must already be complete.
  * <p>
  * A <b>completed></b> request is a request in a terminal state with
  * a process phase of {@code ProcessPhase.Complete}.
  * <p>
  * Note that this operation will lock the request.
  * <p>
  * @param requestId the request identifier of the request.
  * @throws IOException if a protocol error occurred.
  * @throws InstanceNotFoundException if the request is not found
  * @throws OperationException if the given request has child requests
  *  that are not complete.
  * @throws RuntimeOperationsException if a RuntimeService subsystem failure
  * occurs.
  */
 public void recoverRequest( long requestId )
     throws IOException, InstanceNotFoundException, OperationsException,
            RuntimeOperationsException;
```

For more information about manually handling synchronous Java jobs, see the section "Handling Synchronous Java Jobs Requiring Manual Recovery" in "Troubleshooting Oracle Enterprise Scheduler" in *Oracle Fusion Applications Administrator's Guide*.

15.6 Oracle Enterprise Scheduler Interfaces and Classes

Sample code illustrating the new Oracle Enterprise Scheduler asynchronous callback interfaces and classes are shown in Example 15–12, Example 15–13, Example 15–14 and Example 15–15.

Example 15–12 Oracle Enterprise Scheduler Updatable Interface

```
public interface Updatable
{
   /**
    * Invoked by Enterprise Scheduler when a job request is updated.
    * This method must eventually return control to the caller.
```

```
    *
    * @param  context  An oracle.as.scheduler.RequestExecutionContext
    * object for this request.
    *
    * @param parameters  the request parameters associated with this request
    *
    * @param resultCode the {@code
    * oracle.as.scheduler.async.UpdateAction.ActionCode} indicating the
    * action that generated this event.
    *
    * @param messagePayload a {@code String} representing the body of this
    * event. The content and format are not known by the Enterprise Scheduling
    * Service.
    */
    public UpdateAction onEvent( RequestExecutionContext context,
RequestParameters parameters,
                            oracle.as.scheduler.async.AsyncStatus resultCode,
                            String messagePayload );
}
```

The `UpdateAction` class is returned by `Updatable.onEvent`.

Example 15–13 Oracle Enterprise Scheduler UpdateAction Class

```
package oracle.as.scheduler.async;

/**
 * Enumeration of return values from application execution callout. The
 * action returned determines how the subsequent processing of the request
 * will proceed.
 */
public class UpdateAction
{
    /**
     * Constructor. Creates an UpdateAction object from the status
     * and message components.
     *
     * @param status Indicates the status of execution of this update event.
     * This status may result in a state transition for the request.
     *
     * @param message A message that, depending on the value of {@code status},
     * may be used for various purposes.
     */
    public UpdateAction( AsyncStatus status, String message );

    public AsyncStatus getAsyncStatus( );

    public String getMessage( );
}
```

The `AsyncStatus` enum has been modified.

Example 15–14 Oracle Enterprise Scheduler AsyncStatus Enum

```
Package oracle.as.scheduler.async;

/**
* Valid values for the callback status of an asynchronous java job.
 * Returning an {@code AsyncStatus} does not guarantee that the state of the
 * request will change to the corresponding value. The new state of the request
```

```
 * will depend on the old state, the async status, the result of the
 * post-Process handler (if any), and any errors that may occur in
 * subsequent processing.
 */
public enum AsyncStatus
{
    /**
     * The asynchronous job ran successfully.
     */
    SUCCESS,

    /**
     * The asynchronous job has paused for the execution of sub-requests.
     */
    PAUSE,

    /**
     * The asynchronous job is issuing a WARNING.
     */
    WARNING,

    /**
     * The asynchronous job encountered an error.
     */
    ERROR,

    /**
     * The asynchronous job has canceled its execution. Usually this
     * originates from a {@code RuntimeService.cancel} call.
     */
    CANCEL,

    /**
     * The asynchronous job is updated. The request state is not changed
     * by this action.
     */
    UPDATE
}
    /**
     * The asynchronous job encountered a business error.
     */
    BIZ_ERROR,

    /**
     * The asynchronous job requests manual recovery to complete the request.
     */
    ERROR_MANUAL_RECOVERY;
```

Example 15–15 Existing Asynchronous Callback Web Service Operation

```
   /**
     * Set the status of an Oracle Enterprise Scheduler asynchronous java job.
     *
     * @param requestExecutionContext A java.lang.String representing
     * an oracle.as.scheduler.RequestExecutionContext object.
     * @param status
     * @param statusMessage
     *   An error message if the status is ERROR,
     *   A business error message if the status is BIZ_ERROR,
```

```
     *   A warning message if the status is WARNING,
     *   A paused state if the status is PAUSED.
     *   The value is ignored if the status is SUCCESS or CANCEL.
     *
     */
    public void setAsyncRequestStatus( String requestExecutionContext,
                                       AsyncStatus status,
                                       String statusMessage )
        throws RequestNotFoundException, RuntimeServiceException ;
```

16

Oracle Enterprise Scheduler Security

Oracle Enterprise Scheduler Security features provide access control for Oracle Enterprise Scheduler resources and application identity propagation for job execution.

16.1 Introduction to Oracle Enterprise Scheduler Security

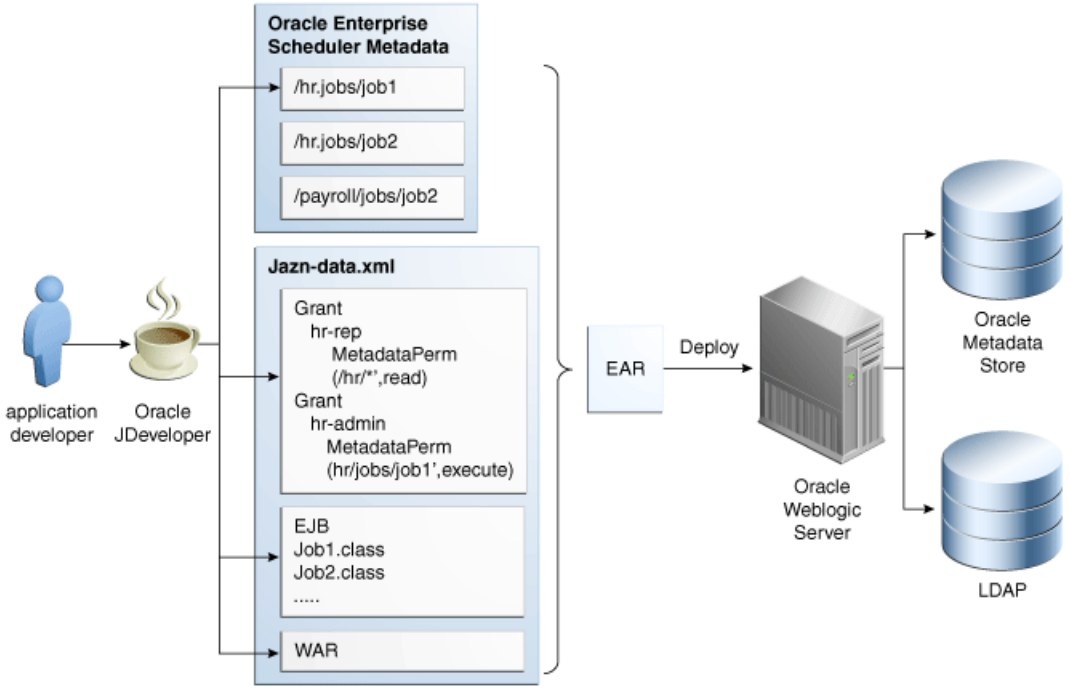
Oracle Enterprise Scheduler Security includes the following:

- Protected operations on `MetadataService`; protected by `MetadataPermission`, which enforces metadata access control. Access control on metadata objects. Only privileged user may create, delete, and update job and schedule metadata. For more information see Section 16.1.1, "Oracle Enterprise Scheduler Metadata Access Control."
- Access control for job requests, enforced by Oracle Fusion Data Security policies. For more information about using Oracle Fusion Data Security policies, see Section 16.7, "Configuring Oracle Fusion Data Security for Job Requests."
- Support for the use of an application identity. Using an application identity enables elevated privileges for completing a job that requires higher privileges than those allotted to the submitting user. For more information, see Section 16.1.2, "Oracle Enterprise Scheduler Job Execution Security."

16.1.1 Oracle Enterprise Scheduler Metadata Access Control

At design time the Metadata creator needs to decide which job functions can access which Metadata objects. This is expressed by associating each Metadata object with one or more roles and specifying one or more actions for each role. Figure 16–1 shows the metadata security summary.

Figure 16–1 Design Time Metadata Security for Oracle Enterprise Scheduler



16.1.2 Oracle Enterprise Scheduler Job Execution Security

During job submission, the user under whose permissions the job request is submitted is called the submitting user. At request execution time all user Java code including pre-processing, post-processing, Java jobs, and substitution, is run as the submitting user, retaining all roles and credentials.

If the job metadata specifies `SYS_RUNAS_APPLICAITONID`, however, the job runs under the elevated privileges of an application ID. For more information, see Section 16.5, "Elevating Privileges for Oracle Enterprise Scheduler Jobs."

16.2 Configuring Metadata Security for Oracle Enterprise Scheduler

When a user accesses Oracle Enterprise Scheduler services using the RuntimeService or `MetadataService`, the identity of the user is acquired and Oracle Enterprise Scheduler checks if the user has the required permissions to access resources (for example Metadata objects). For example, if a user named teller1 needs to call `getJobDefinition` to access a Metadata object named `caclulateFees`, Oracle Enterprise Scheduler ensures that teller1 has `READ` permission for the Metadata object `caclulateFees` before returning the object.

At design time the Metadata creator needs to decide which job functions can access which Metadata objects. This is expressed by associating each Metadata object with one or more roles and specifying one or more actions for each role.

There are two options for Metadata role assignments:

- Using Oracle JDeveloper Tools Oracle ADF Security Wizard
- Using Oracle JDeveloper Oracle Enterprise Scheduler add-in Metadata pages

Oracle JDeveloper ADF Security wizard creates the roles you use; the roles must be created before you can register roles with a metadata object.

16.2.1 How to Enable Application Security with Oracle ADF Security Wizard

These steps describe a minimal, validated security setup for an application using Oracle Enterprise Scheduler.

Follow these steps to create a working `jps-config.xml` and a partially-populated `jazn-data.xml`. Use these steps to configure servlets to work with JPS.

To enable security using the ADF Security wizard:

1. In Oracle JDeveloper, with an application open, from the Application menu select **Secure**.
2. From the dropdown list, select **Configure ADF Security**. The Configure ADF Security wizard displays.
3. In the Enable ADF Security page, select either **ADF Authentication and Authorization** or **ADF Authentication** and click **Next**.
4. In the Select authentication type page, select either **HTTP Basic Authentication** or **Form-Based Authentication** and click **Next**.
5. In the Enable automatic policy grants page, select the appropriate options from the Enable Automatic Grant area, and click **Next**.
6. In the Specify authenticated welcome page, select options as needed and click **Next**.
7. In the Summary page verify the options and click **Finish**.
8. In the Security Infrastructure Created dialog, click **OK**.

Next, to enable security and to ensure that the `jazn-data.xml` is included in the application deployment, perform the following steps after assembling the EAR file for the application. For more information, see Section 3.6.3, "How to Assemble the EAR File for Scheduler Sample Application."

Ensure the security related files are included with EAR file:

1. In Oracle JDeveloper, select **Application** > **Application Properties**.
2. In the Application Properties page, in the Navigator select **Deployment**.
3. In the Deployment Profiles area, select the EAR file Deployment descriptor. For example, for the sample application this is shown in Section 3.6.3, "How to Assemble the EAR File for Scheduler Sample Application".
4. Click **Edit**. This displays the Edit EAR Deployment Profile Properties page.
5. In the Edit EAR Deployment Profile Properties page, expand **File Groups** > **Application Descriptors** > **Filters**.
6. In the Filters area, select the **Files** tab.
7. Ensure that the files `jazn-data.xml`, `jps-config.xml`, and `weblogic-application.xml` are selected under the `META-INF` folder.
8. Click **OK** to save the descriptor.

16.2.2 How to Define Principals for Security

You need to define roles before the roles are used in Oracle Enterprise Scheduler security. There are two types of roles that may be defined:

- Enterprise roles: These are defined directly in Oracle WebLogic Server either using the Oracle WebLogic Server console, using the WLST scripts, or using the ADF Security Wizard in Oracle JDeveloper.
- Application roles: These can be defined in the `jazn-data.xml` file or using the ADF Security Wizard.

To define principals security:

1. In Oracle JDeveloper, open the application and expand Application Resources in the **Application Navigator**.
2. In the Application Resources area, expand **Descriptors** and **META-INF**.
3. In META-INF, double-click to open `jazn-data.xml`.
4. In the page showing `jazn-data.xml`, select the **Overview** tab. Note, if the **Overview** tab is not shown, try closing `jazn-data.xml` and then opening it again.
5. Click **Application Roles...**(Manage Users and Roles).
6. On the Edit JPS Identity and Policy Store page, in the navigator expand **Identity Store** and **jazn.com**.
7. In the navigator, select **Roles** and click **Add...**. This displays the Add Role dialog.
8. In the Add Role dialog, enter a name in the **Name** field.
9. Click **OK**.
10. On the Edit JPS Identity and Policy Store page, in the navigator select **Application Policy Store**. If there is a sub-element with the same name as the application, go to the next step, Otherwise, do the following:
    a. Select **Application Policy Store**.
    b. Click **New...** . This displays the Create Application Policy dialog.
    c. In the Create Application Dialog the **Display Name** field should contain the application name.
    d. Click **OK** to accept the default Display Name.
11. On the Edit JPS Identity and Policy Store page, in the navigator expand **Application Policy Store** and expand the *application name*.
12. In the navigator, select **Application Roles**. This displays the Application Roles page.
13. In the Application Roles page, click **Add**... to add roles. For correct functionality at least one enterprise role must be mapped to the application role by adding enterprise roles in the **Member Roles** tab.
14. Click **OK**.

16.2.3 How to Create Grants with Oracle Enterprise Scheduler Metadata Pages

Access to all Metadata is controlled by grants. In order to ensure access by the right identities, you need to give the correct grants. It is expected that most Metadata grants will be done using the Oracle Enterprise Scheduler Oracle JDeveloper add-in.

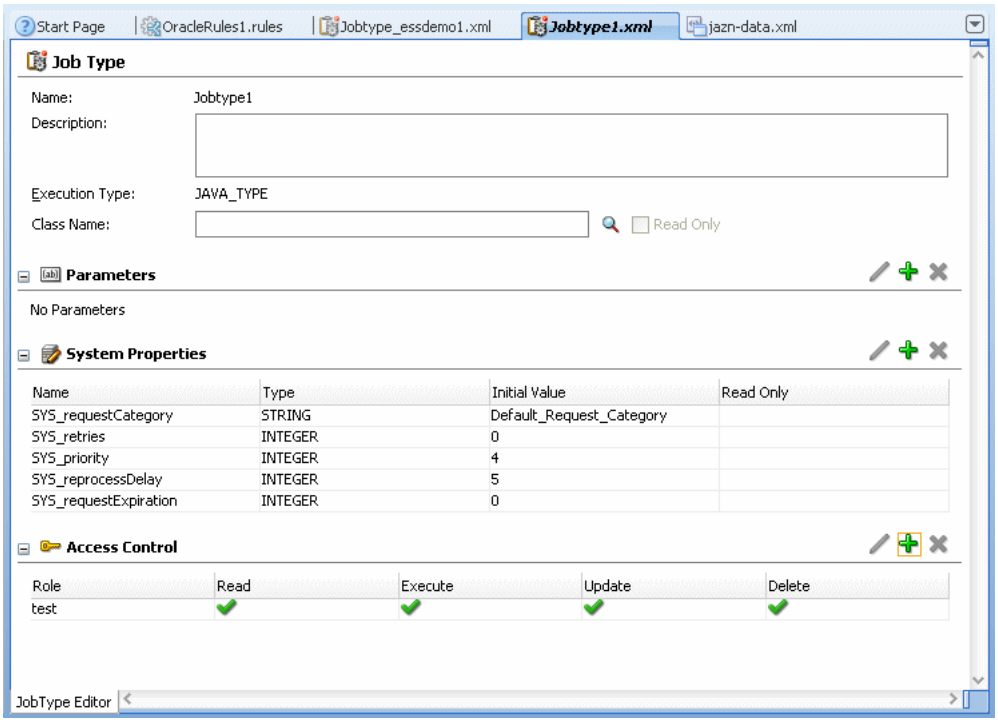
First, create any required Oracle Enterprise Scheduler Metadata in an application using **File** > **New** > **Business Tier** > **Enterprise Scheduler Metadata**. For more information on creating Metadata, see Section 3.5, "Creating Metadata for Scheduler Sample Application."

Using Oracle JDeveloper, you can add security grants to Oracle Enterprise Scheduler metadata objects.

To secure Oracle Enterprise Scheduler metadata objects:

1. Open the Editor page for any Oracle Enterprise Scheduler Metadata object.
2. In the Access Control area, click **Add** to add a new access control item.
3. In the Add Access Control dialog, select a Role from the dropdown list. This selects a role to grant access privileges.
4. Select one or more actions from the list, **Read**, **Execute**, **Update**, or **Delete**.
5. Click **OK**. This displays the updated role, as shown in Figure 16–2.
6. Repeat for as many roles as needed.

Figure 16–2 Security Roles for Oracle Enterprise Scheduler Metadata



16.2.4 How to Create Grants with Oracle ADF Security Wizard

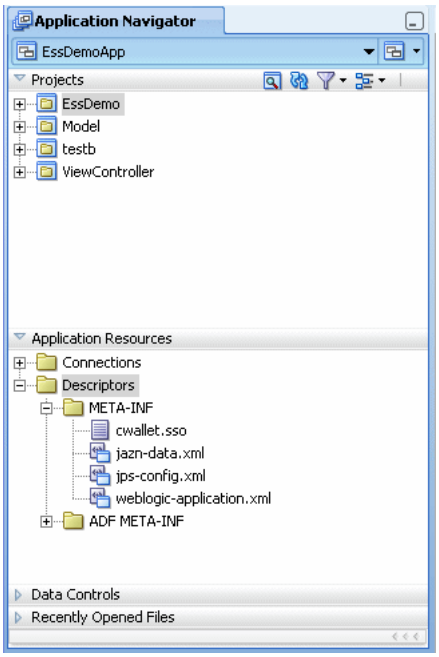
There may be occasions where you want to create grants explicitly, for example when using wildcards. These steps show how to set up grants using the ADF Security wizard.

Note that these steps assume you have already created application roles.

To specify grants with the ADF Security wizard:

1. In the Application Navigator, expand the Application Resources panel.
2. Expand **Descriptors** and **META-INF**, as shown in Figure 16–3.

Figure 16–3 Security Configuration Files Including jazn-data.xml in META-INF



3. Double-click `jazn-data.xml` to open the file. In the editor panel for `jazn-data.xml`, select the **Overview** tab, and click **Application Roles... (Manage Users and Roles)**. This displays the JPS Identity & Policy Store dialog. Note, if the **Overview** tab is not shown, try closing `jazn-data.xml` and then opening it again.
4. In the JPS Identity & Policy Store dialog, in the navigator expand **Application Policy Store**.
5. Expand *application-name*, and select **Application Roles**.
6. Click **New**.
7. Enter the display name you wish for this grant, and click **OK**.
8. Select the **Principals** tab, and click **Add...** .
9. Enter the name of the application role which will receive the grant; this should be one of the role names created. Leave the **Class** field as is.
10. Click **OK**.
11. With the new role selected in the **Principals** tab, make sure the **Type** is `role`.
12. Select the **Permissions** tab, and click **Add...**.
13. For the **Name** field, enter a full permission string or a partial string with wildcards; see Table 16–1 for examples. In the **Class** field, enter `oracle.as.scheduler.security.MetadataPermission`. Click **OK**.
14. With the new permission selected in the **Permissions** tab, enter the desired actions in the **Actions** Field.
15. Click **OK** to save the grant.

Note: If necessary, use the following workaround:

1. Right-click the `jazn-data.xml` file and select **Open**.
2. Click the Source tab.
3. Under `<jazn-policy><grant><grantee>`, remove the elements `<display-name>` and `<type>`.

Table 16–1 Sample Permission Grants for Security Using Oracle ADF

Name	Actions	Effect
`package-part.JobDefinition .MyJavaSucJobDef`	`EXECUTE`	Grants the ability to submit requests for a single Metadata item.
`mypackage.subpackage.*`	`CREATE,EXECUTE`	Grants to ability to create and execute any new Metadata items in /mypackage/subpackage
`JobDefinition.SYS_ AdHocRequest`	`CREATE,EXECUTE`	Grants ad hoc submission permission
`mypackage.*`	`CREATE,EXECUTE,DELETE`	Grants wide-open permissions

16.2.5 About MetadataPermission APIs

Grants for Metadata are part of the class `oracle.as.scheduler. security.MetadataPermission`. The name, or target of the permission is based on the package, Metadata object type, and name of the Metadata object being protected; this identifier can be retrieved from `MetdataObjectId#toPermissionString()`.

Table 16–2 lists the actions for the grants. The notation <*Type*> is a placeholder for all of the metadata object types. For example, get<*Type*>() refers to the methods `getJobDefinition()`, `getJobType()`, `getJobSet()`.

Table 16–2 Grant Actions for Metadata Security

Action	Implies	Metadata Functions
`READ`	`None`	get<*Type*>(), query<*Type*>()
`EXECUTE`	`READ`	submitRequest()
`CREATE`	`READ`	add<*Type*>()
`UPDATE`	`READ`	update<*Type*>()
`DELETE`	`READ`	delete<*Type*>()

If you are submitting ad-hoc requests, you can have full wildcard ("*") permission with both `EXECUTE` and `CREATE` actions. When submitting ad-hoc requests, that is, using `submitRequest()` without certain `MetadataObjectIds`, you can grant permissions with the full wildcard ("*") name using the `EXECUTE` and `CREATE` actions.

16.2.6 What Happens When You Configure Metadata Security

Each time a user application calls a `MetdataService` or `RuntimeService` method, Oracle Enterprise Scheduler checks the current subject for privileges on the metadata accessed by the methods. For example, submitting a request requires `EXECUTE` permissions on the job definition or job set metadata object associated with the submission. Methods that change metadata, for example calling `updateJobDefinition()`, require `UPDATE` permissions.

For all `MetadataService` methods except queries, an exception is thrown when the user tries to access a Metadata object for which the user does not have permission.

The `MetadataService` query methods have different behavior. When a user performs a query Oracle Enterprise Scheduler only returns Metadata objects that have `READ` permission. Thus a user who has no permissions on Metadata objects receives an empty list for all queries, but this user would not see an exception thrown due to lack of permissions.

The value of `SystemProperty.USER_NAME` is overwritten at submission time; the user cannot spoof an identity at submission time using `SystemProperty.USER_NAME`.

16.3 Configuring Web Service Security for Oracle Enterprise Scheduler

For information about securing the Oracle Enterprise Scheduler web service, see Section 10.9, "Securing the Oracle Enterprise Scheduler Web Service."

16.4 Configuring PL/SQL Job Security for Oracle Enterprise Scheduler

For standalone cases, implement the application user session using Java or the PL/SQL API as described in the chapter "Implementing Application User Sessions" in *Oracle Fusion Applications Developer's Guide*.

16.5 Elevating Privileges for Oracle Enterprise Scheduler Jobs

When a user accesses Oracle Enterprise Scheduler services using the `RuntimeService` or `MetadataService` interfaces, the identity of the user calling the methods is acquired. This identity is used to check whether the user has the required permissions to access certain resources such as metadata objects. For example, if user `teller1` calls the method `getJobDefinition` for metadata object `caclulateFees`, Oracle Enterprise Scheduler ensures that `teller1` has read permissions for metadata object `caclulateFees` before returning the object.

The caller identity is also used to run jobs requested by the user. For example, if user `teller1` calls the method `submitRequest()` for a Java job, the requested jobs run under `teller1` and retain all roles and credentials assigned to that user.

Oracle Enterprise Scheduler supports the use of an application identity. Using an application identity enables elevated privileges for completion of a job that requires higher privileges than those allotted to the submitting user.

For more information about enabling elevating privileges, see Section 9.13, "Elevating Access Privileges for a Scheduled Job."

16.6 Configuring a Single Policy Stripe in Oracle Enterprise Scheduler

Oracle Platform Security policy store serves as the repository for authorization policies. Authorization policies load at run time into the Java Virtual Machine, and are used to make decisions regarding authorization. Authorization policies comprise a hierarchy of application roles, the mapping of enterprise roles to application roles and permissions grants to application roles. Application roles can also be hierarchical.

Aside from authorization policies, Oracle Platform Security policy store also stores administrative constructs that help in maintaining these authorization policies, including resource catalogs (with associated resource types), permission sets and role categories. The authorization polices and administrative components are scoped to an application. This is known as an application stripe.

An application stripe is a collection of JAAS policies applicable to the application with which it is associated. Out of the box, an application stripe maps to an Oracle Java EE application. Oracle Platform Security also supports mapping multiple Java EE applications to one application stripe. The application ID string identifies the name of the application or applications.

16.6.1 How to Configure a Single Policy Stripe in Oracle Enterprise Scheduler

Oracle Enterprise Scheduler allows specifying an `applicationStripe` name and mapping it to a JPS policy context ID. You can assign multiple Oracle Enterprise Scheduler hosting applications to a single policy context.

To configure an Oracle Enterprise Scheduler hosting application to a specific applicationStripe:

1. Open the `ejb-jar.xml` file.
2. Under the `message-driven` element, add an `activation-config-properties` element with the value `applicationStripe`.
3. Under the `jpsinterceptor-class` element, configure the `JpsInterceptor`.

   Make sure to match the value of `applicationStripe` under the `<message-driven>` element with the `application.name` value under the `<interceptor>` element.

   Example 16–1 shows an `applicationStripe` configuration for the policy context `ESS_FUNCTIONAL_TEST_APP_STRIPE`.

Example 16–1 Configuring the applicationStripe and the JpsInterceptor

```
<ejb-jar>
   ....

   <enterprise-beans>
    <message-driven>
      <ejb-name>ESSAppEndpoint</ejb-name>
      <ejb-class>oracle.as.scheduler.ejb.EssAppEndpointBean</ejb-class>
      <activation-config>
        ....
        <activation-config-property>
          <activation-config-property-name>applicationStripe</activation-config-property-name>
          <activation-config-property-value>ESS_FUNCTIONAL_TESTS_APP_
            STRIPE</activation-config-property-value>
        </activation-config-property>
      </activation-config>
    </message-driven>
    .....

  </enterprise-beans>

  <interceptors>
    <interceptor>
      <interceptor-class>oracle.security.jps.ee.ejb.JpsInterceptor</interceptor-class>
      <env-entry>
        <env-entry-name>application.name</env-entry-name>
        <env-entry-type>java.lang.String</env-entry-type>
        <env-entry-value>ESS_FUNCTIONAL_TESTS_APP_STRIPE</env-entry-value>
        <injection-target>
          <injection-target-class>oracle.security.jps.ee.ejb.JpsInterceptor
          </injection-target-class>
          <injection-target-name>application_name</injection-target-name>
```

```
        </injection-target>
      </env-entry>
    </interceptor>
  </interceptors>
</ejb-jar>
```

4. If your application has a web module, configure the web module `JpsFilter` to use the same `applicationStripe` in the file `web.xml`. Example 16–2 shows a code sample.

Example 16–2 Configuring the Web Module in web.xml

```
<web-app>
  <filter>
      <filter-name>JpsFilter</filter-name>
      <filter-class>oracle.security.jps.ee.http.JpsFilter</filter-class>
      ...
      <init-param>
          <param-name>application.name</param-name>
          <param-value>ESS_FUNCTIONAL_TESTS_APP_STRIPE</param-value>
      </init-param>
  </filter>

</web-app>
```

16.6.2 What Happens When You Configure a Single Policy Stripe

At design time, an application stripe manifests as:

- An `<application>` element under the `<policystore>` element in the `jazn-data.xml` file.
- A node under the node `cn=<Weblogic.domain.name>,cn=JPSContext,cn=<root.node>`, such as `cn=ATGDemo,cn=base_domain,cn=JPSContext,cn=MY_Node`.

16.6.3 What Happens at Runtime

At run time, an application stripe manifests as an instance of the class `oracle.security.jps.service.policystore.ApplicationPolicy`.

16.7 Configuring Oracle Fusion Data Security for Job Requests

Oracle Fusion Data Security for Oracle Fusion Applications enforces security authorizations for access and modification of specific data records. Oracle Fusion Data Security integrates with Oracle Platform Security Services (OPSS) by granting actions to OPSS principals. The grant defines who (the principals) can do what (the actions) on a given resource. A grant in Oracle Fusion Data Security can use any enterprise user or enterprise group as principals. For more information about implementing Oracle Fusion Data Security, see the chapter "Implementing Oracle Fusion Data Security" in *Oracle Fusion Applications Developer's Guide*.

In the context of Oracle Enterprise Scheduler, a job request access control data security policy comprises a grant, a grantee and a set of `ESS_REQUEST` privileges for a set of job requests as follows:

- A grantee, represented by grantee ID such as a user or application role, the ID should match the user GUID or application role GUID retrieved from Oracle Fusion Middleware.
- A set of `ESS_REQUEST` privileges represented by a menu ID mapped to a set of form functions.
- A set of data represented by an `INSTANCE_SET` ID. An `INSTANCE_SET` is typically represented by a predicate which can be appended to a query to the job request data exposed to Oracle Fusion Applications (see Section 16.7.1, "Oracle Fusion Data Security Artifacts").

The job request access control data security policy can be managed using Oracle Authorization Policy Manager as are other Oracle Fusion Data Security policies. If Oracle Authorization Policy Manager is not available, you can use SQL scripts to manipulate the Oracle Fusion Data Security artifacts.

16.7.1 Oracle Fusion Data Security Artifacts

To use Oracle Enterprise Scheduler job request access control feature in the context of Oracle Fusion Applications, the Oracle Fusion Applications schema and Oracle Enterprise Scheduler schema must be located in a single database.

Oracle Enterprise Scheduler implements job request data security on top of the `request_history` and `request_property` tables. It exposes Oracle Enterprise Scheduler job request related data to the Oracle Fusion Applications schema through the following views: `request_history_view` and `request_property_view`. Two synonyms are created in the Oracle Fusion Applications schema which are linked to the Oracle Enterprise Scheduler schema.

The `request_history_view` contains all columns that correspond to `RuntimeService.QueryField`, which is used when constructing the filter for `queryRequest()` operations, as well as two other columns: `submitter` and `submitterguid`. Be sure to define your `INSTANCE_SET` based on these columns only.

Table 16–3 lists the Oracle Fusion Applications schema tables and their Oracle Enterprise Scheduler synonyms, as well as the columns used to define data security policies.

Table 16–3 Mapping Oracle Fusion Applications Schema Synonyms to Oracle Enterprise Scheduler Schema Views and Relevant Columns

Oracle Fusion Applications Schema Synonym	Link to Oracle Enterprise Scheduler Schema View	Columns
ess_request_history	request_history_view	See table for the QueryField and View Column mapping.
ess_request_property	request_property_view	create or replace view request_property_view as select requestid, name, scope, datatype, value, lobvalue, lobflag from request_property with read only;

Table 16–4 shows the mapping of RuntimeService.QueryField columns to the Oracle Enterprise Scheduler request_history_view columns.

Table 16–4 Mapping RuntimeService.QueryField Columns to request_history_view Columns

RuntimeService.QueryField Columns	Request_history_view Columns
QueryField.REQUESTID	requestid
QueryField.APPLICATION	application
QueryField.USERNAME	userName
QueryField.PRODUCT	product
QueryField.REQUEST_CATEGORY	requestCategory
QueryField.PRIORITY	priority
QueryField.NAME	name
QueryField.ABSPARENTID	absParentId
QueryField.TYPE	type
QueryField.DEFINITION	definition
QueryField.STATE	state
QueryField.SCHEDULE	schedule
QueryField.PROCESSSTART	processStart
QueryField.PROCESSEND	processEnd
QueryField.REQUESTEDSTART	requestedStart
QueryField.REQUESTEDEND	requestedEnd
QueryField.SUBMISSION	submission

Table 16–4 (Cont.) Mapping RuntimeService.QueryField Columns to request_history_view Columns

RuntimeService.QueryField Columns	Request_history_view Columns
QueryField.PARENTREQUESTID	parentRequestId
QueryField.WORKASSIGNMENT	workAssignment
QueryField.SCHEDULE	scheduled
QueryField.REQUESTTRIGGER	requesttrigger
QueryField.PROCESSOR	processor
QueryField.CLASSNAME	classname
QueryField.ELAPSEDTIME	elapsedtime
QueryField.WAITTIME	waittime
QueryField.SUBMITTER	submitter
QueryField.SUBMITTERGUID	submitterguid

Table 16–5 maps FND_MENUS to FND_FORM_FUNCTIONS as reflected in FND_MENU_ENTRIES.

Table 16–5 Mapping FND_MENUS to FND_FORM_FUNCTIONS

FND_MENUS in the Oracle Fusion Applications Schema	FND_FORM_FUNCTIONS
ESS_REQUEST_ADMIN	ESS_REQUEST_READ
	ESS_REQUEST_UPDATE
	ESS_REQUEST_HOLD
	ESS_REQUEST_CANCEL
	ESS_REQUEST_LOCK
	ESS_REQUEST_RELEASE
	ESS_REQUEST_DELETE
	ESS_REQUEST_PURGE
ESS_REQUEST_VIEW	ESS_REQUEST_READ
ESS_REQUEST_OPERATE	ESS_REQUEST_READ
	ESS_REQUEST_HOLD
	ESS_REQUEST_CANCEL
	ESS_REQUEST_LOCK
	ESS_REQUEST_RELEASE
ESS_REQUEST_OUTPUT_ADMIN	ESS_REQUEST_OUTPUT_VIEW
	ESS_REQUEST_OUTPUT_DELETE

Table 16–6 lists the required data privilege (form_function) for a user to perform an Oracle Enterprise Scheduler runtimeService operation.

Table 16–6 Data Privileges Needed to Execute runtimeService Operations

RuntimeService API Operation	Data Privilege (FND_FORM_FUNCTIONS)	Notes
open	none	
close	none	Two overloaded methods.
submitRequest	none	Five overloaded methods, which are secured by metadata security, not data security.
getRequestParameter	ESS_REQUEST_READ	
getRequestState	ESS_REQUEST_READ	
getRequests	ESS_REQUEST_READ	
getRequestDetail	ESS_REQUEST_READ	
getRequestDetailBasic	ESS_REQUEST_READ	
lockRequest	ESS_REQUEST_LOCK	
updateRequestParameter	ESS_REQUEST_UPDATE	
queryRequests	ESS_REQUEST_READ	
holdRequest	ESS_REQUEST_HOLD	
releaseRequest	ESS_REQUEST_RELEASE	
cancelRequest	ESS_REQUEST_CANCEL	
deleteRequest	ESS_REQUEST_DELETE	
purgeRequest	ESS_REQUEST_PURGE	
publishEvent	none	Not targeted to a request.
isHandleRollbackOnly	none	Not targeted to a request.
setHandleRollbackOnly	none	Not targeted to a request.
replaceSchedule	none	

Table 16–7 displays the INSTANCE_SET conditions provided by Oracle Authorization Policy Manager.

Table 16–7 INSTANCE_SET Conditions Provided by Oracle Authorization Policy Manager

INSTANCE_SET Condition	Description
REQS_SUBMITTEDBY_SESSIONUSER	Oracle Enterprise Scheduler requests that the submitter is the current session user.
REQS_RUNAS_SESSIONUSER	Oracle Enterprise Scheduler requests that the RunAs user is the current session user.
REQS_SUBREQS_BY_SUBMITTER	Oracle Enterprise Scheduler requests and subrequests are all submitted by the submitter.
REQS_ALL_OF_ONE_APP	Indicates all Oracle Enterprise Scheduler requests related to a product within a logical application. This condition takes two parameters that match the job request parameter values of SYS_application and SYS_product.
ESS_REQS_BY_NAME_VALUE_PARAM	Oracle Enterprise Scheduler job request whose RequestParameter name value pair is specified in data security grants. This condition takes two parameters that match the one job request parameter's name and value.

Table 16–8 lists the Oracle Fusion Data Security policies available for use with Oracle Enterprise Scheduler out of the box.

Table 16–8 Oracle Fusion Data Security Policies for Oracle Enterprise Scheduler

Oracle Fusion Data Security Policy	Description
ESS_REQUEST_SUBMITTER_ADMIN_SUBMITTED_REQUESTS	The submitter of the job request is permitted to view and administer the requests they submitted.
ESS_REQUEST_SUBMITTER_ADMIN_SUBMITTED_REQUESTS_SUBREQS	The submitter of the job requests and subrequests is permitted to view and administer on the requests they submitted.
ESS_REQUEST_RUNASUSER_ADMIN_EXECUTED_REQUESTS	The runAs user is permitted to view and administer the requests they execute.
ESS_REQUEST_RUNASUSER_VIEWOUTPUT_EXECUTED_REQUESTS	The runAs user is permitted to view the output of the job requests they executed.

For more information about the runAs user, or elevating access privileges, see Section 9.13, "Elevating Access Privileges for a Scheduled Job."

16.7.2 How to Apply Oracle Fusion Data Security Policies

The Oracle Fusion Data Security components described in Section 16.7.1, "Oracle Fusion Data Security Artifacts" can be applied as follows.

To apply Oracle Fusion Data Security policies:

1. Examine the policies described in Table 16–8 and determine whether you can use any of them in your application.
   - If you can use one of these policies, skip to the last step.
   - If the policies do not apply, continue on to the next step.
2. Determine whether any of the FND_MENUS listed in Table 16–5 suit the out-of-the-box Oracle Fusion Data security policy you selected for your application. If you cannot apply any of the FND_MENUS listed in Table 16–5, create your own FND_MENUS and FND_MENUS_ENTRIES as described in the chapter "Implementing Oracle Fusion Data Security" in the *Oracle Fusion Applications Developer's Guide*.
3. Determine whether you can use the INSTANCE_SET conditions in Table 16–7 and the Oracle Fusion Data Security policies in your application. If you cannot use the conditions, create your own FND_INSTANCE_SET. For more information about creating an FND_INSTANCE_SET, see the chapter "Implementing Oracle Fusion Data Security" in the *Oracle Fusion Applications Developer's Guide*.
4. Create an Oracle Fusion Data Security policy, as described in Section 16.7.3, "How to Create Functional and Data Security Policies for Oracle Enterprise Scheduler Components."

   Note: If developing an Oracle Fusion application, do not grant an Oracle Enterprise Scheduler access policy to the grantee of an authenticated-role or anonymous-role, as doing so may affect the behavior of Oracle Enterprise Scheduler or other products.

5. Test your application.

16.7.3 How to Create Functional and Data Security Policies for Oracle Enterprise Scheduler Components

You can use Oracle Authorization Policy Manager to create functional and data security policies for Oracle Enterprise Scheduler.

For more information about creating policies in Oracle Authorization Policy Manager, see the chapter "Managing Security Artifacts" in *Oracle Fusion Middleware Oracle Authorization Policy Manager Administrator's Guide (Oracle Fusion Applications Edition)*.

To create functional and data security policies for Oracle Enterprise Scheduler:

1. Create a resource.
    a. From the list of policies, expand the **fcsm** policy stripe and select **fcsm > Resource Catalog > Resources**.
    b. From the Actions menu, click **New**.
    c. Define a resource with the resource type **ESSMetadataResourceType**, as well as the name and display name of the Oracle Enterprise Scheduler component using the following syntax:
    `oracle.apps.ess.`*`applicationName`*`.`*`JobDefintitionName`*`.`*`JobName`*.
    d. Save the resource.
2. Define a resource policy.
    a. Select the resource you just created and click **Create Policy**.
    b. Add principals (grantees) by clicking the **Add** button.
    c. In the Add Principal window, search for the relevant application role or roles. Select the roles and click **Add**.
    d. In the Actions field, select the relevant actions and click **Apply**.
3. Create an authorization condition.
    a. In the Authorization Management tab, select **Global** and search for the database resource you want to use. TABLE XY lists the database resources related to Oracle Enterprise Scheduler.
    b. Select the resource and click **Edit**.
    c. Click the Conditions tab and select **Actions > New**.
    d. Enter a name, display name and SQL predicate for the condition.
4. Define a data policy.
    a. From the Actions menu, select **New Policy**.
    b. In the New Policy window, use the Role and Database Resource fields to add the relevant roles and resources.
    c. Select the role you defined. In Database Resource Details region, select the condition name you just created and choose the actions you require.

17

Managing Business and System Errors

This chapter describes how to indicate Oracle Enterprise Scheduler system and business errors as well as implement job request retries.

This chapter includes the following sections:

- Section 17.1, "Introduction to Managing Business and System Errors"
- Section 17.2, "Indicating Errors"
- Section 17.3, "Configuring Retries for a Job Request"
- Section 17.4, "Finding and Diagnosing Job Requests in Error State"

17.1 Introduction to Managing Business and System Errors

When an Oracle Enterprise Scheduler job request encounters an error during execution, Oracle Enterprise Scheduler can indicate whether the error is a business or system error.

A business error occurs when a job request must abort prematurely, but is otherwise able to exit cleanly, leaving its data in a consistent state. Examples of scenarios requiring a job to abort prematurely include a particular application setup or configuration condition, a functional conflict that requires an early exit or corrupt or inconsistent data.

A system error occurs when a job request encounters a technical error from which it cannot recover, but otherwise exits of its own volition. Alternatively, a system error occurs when the server or operating system running the job crashes. Examples of system errors include table space issues and unhandled runtime exceptions.

A job request that indicates an error is placed in the terminal state of `ERROR`. The error type field for a job request indicates whether the error is a business or system error. System errored job requests can be automatically retried if they are properly configured. Business errored job requests cannot be retried.

17.2 Indicating Errors

You can indicate business and system errors using specific error statuses or exit codes for each job type.

For more information about using exit codes, see the following sections:

- Section 5.4, "Using System Properties,"
- Section 7.2.1, "How to Create and Store a Process Job Type,"
- Section 9.7.2, "How to Implement a SQL*Plus Job,"

- Section 9.9.1, "How to Implement a Perl Scheduled Job,"
- Section 9.10.3, "Scheduled C Job API,"
- Section 9.11, "Implementing a Host Script Scheduled Job."

17.2.1 How to Indicate a Business Error

Table 17–1 shows the code used to indicate a business error for each job type. For a business error, the job request state is set to `ERROR`, the error type to `Business` and the cause to `PROCESS_ERROR`. For the Java jobs, the table lists different stages in running a job along with a business error indication for each.

Table 17–1 Indicating a Business Error

Job Type or Job Stage	Business Error Indication
`Executable.execute` (Java job)	Throw `ExecutionBizErrorException` (extends `ExecutionErrorException`).
Asynchronous Java job (initiated from AsyncJava)	Send `AsyncStatus.BIZ_ERROR`.
`Updatable.onEvent`	Return `AsyncStatus.BIZ_ERROR` in the `UpdateAction`.
`CJobType`	Return `FDP_BIZERR` using `afpend()` API.
`PlSqlJobType`	Return `retcode = '3'`.
`SqlPlusJobType`	Set `FND_JOB.BIZERR_V` using `FND_JOB.SET_SQLPLUS_STATUS` API.
`PerlJobType`	Return exit code of 3.
`HostJobType`	Return exit code of 3.

17.2.2 How to Indicate a System Error

A system error results from an unhandled exception and may also be explicitly indicated by the job, as shown in Table 17–2. For a system error, the request state is set to `ERROR` and the error type to `System`. For the Java jobs, the table lists different stages in running a job along with a system error indication for each.

Table 17–2 Indicating System Errors

Job Type or Job Stage	System Error Indication
`Executable.execute` (Java job)	Throw `ExecutionErrorException`.
Asynchronous Java job (initiated from AsyncJava)	Send `AsyncStatus.ERROR`.
`Updatable.onEvent`	Return `AsyncStatus.ERROR` in the `UpdateAction`.
`CJobType`	Return `FDP_ERROR` using `afpend()` API.
`PlSqlJobType`	Return `retcode = '2'`.
`SqlPlusJobType`	Set `FND_JOB.FAILURE_V` using `FND_JOB.SET_SQLPLUS_STATUS` API.
`PerlJobType`	Return an exit code of 1.
`HostJobType`	Return an exit code of 1.

17.3 Configuring Retries for a Job Request

Job requests that fail as a result of a system error can be retried, meaning they can be configured to automatically re-run from the pre-process stage.

Oracle Enterprise Scheduler uses an increasing delay algorithm to improve the chances that the system error will have been resolved when the request is retried. During the delay, the request is placed in `WAIT` state. On the first system error, the delay is 1 minute; on the second, 2 minutes; on the third, 5 minutes; on the fourth system error and greater, the delay is 10 minutes. For example, suppose a job request fails with a system error three times before it is successful. The job request is delayed a total of 8 minutes (1+2+5).

When a job request fails, resources such as incompatibility locks are released, and the job request goes back to the wait queue. Incompatibility locks are released only for the job request being retried and not for any parent request that is still active.

The job may have already completed some of its processing when the error occurs. On retry, the job must be able to continue its processing from the point of error., meaning it must be an idempotent job. Idempotent jobs can be configured so that the job request is automatically retried in case of a system error. An idempotent job is able to continue where it left off when it is retried.

Note: Configure retries only for idempotent jobs.

17.3.1 How to Configure Retries for a Job Request

The system property `SYS_retries` enables configuring the maximum number of times a failed job request can be retried.

To configure retries for a job request:

1. In JDeveloper, edit the job definition.
2. Using the system property `SYS_retries`, enter the number of times the job request is to be automatically retried. A value of zero indicates that the job request will not be retried. The property `SYS_retries` has a default value of zero, and can only be defined for idempotent jobs.

Note: Job requests that fail with a business error are never automatically retried. Oracle Enterprise Scheduler ignores the `SYS_retries` parameter in such cases.

For more information about configuring properties for a job request, see Chapter 9.4.1, "How to Create a Job Definition."

17.3.2 What Happens at Run Time: How a Job Request Is Retried

The behavior of retried job requests differs depending on the type of job request.

- Job set retry: Job sets cannot be retried, however, the steps of a job set can be retried provided the steps themselves are job definitions. When a job set step throws a system error, Oracle Enterprise Scheduler retries the step if the job definition associated with the step is configured for retry. When retrying a step, the incompatibility locks for the step request are released, while incompatibility locks for parent job sets continue to be held. This means that the incompatibility locks for parent job sets are held across retries of a job set step. The state of the job

set is unaffected by the state of the step until the step reaches a non-error terminal state or all retries for the step have been exhausted.

For serial job sets, all retries are completed for a step before any link is followed. If a job set step defines both `ON_SUCCESS` and `ON_ERROR` links, the `ON_ERROR` link is not followed until all retries have been exhausted and the step has reached a terminal state of `ERROR`.

- Sub-request retry: Sub-requests can be retried. When a sub-request throws a system error, Oracle Enterprise Scheduler retries the sub-request as many times as specified by the retry configuration for the sub-request. The parent request remains in `PAUSED` state until the sub-request reaches a non-error terminal state or all retries for the sub-request have been exhausted. Neither sub-request execution nor retry affects the incompatibility locks of the parent job request, meaning the parent holds its incompatibilities across sub-request retries.
- Recurring job request retry: A submitted recurring job request cannot be retried. However, each recurring instance can itself be retried. For example, suppose the job definition for a recurring request has `SYS_retries` set to 3. Each instance of the recurrence that fails with a system error can be retried up to 3 times.

17.3.3 What You Should Know about Configuring Retries for a Job Request

Following is a list of recommendations for configuring retries for a job request.

- To minimize the amount of time and effort required to recover from a job failure, it is advisable to develop most jobs as idempotent jobs (able to continue from the point of departure when retried). Thus, if the same job request executes again after it previously failed, the job code ensures that the retry is handled properly. If a job is idempotent, it can be configured to automatically retry when encountering system errors. This is especially important for long running jobs where recovery involves manually rolling back changes and restarting the job from the beginning.
- If the job is idempotent, set `SYS_retries` to a positive number so that the job can be automatically retried in case of system error.
- If the job is not idempotent, do not set `SYS_retries`. This prevents the job from being run twice with unpredictable results.
- When defining a job set, make sure the `ERROR` branch connects to a job set step that does not depend on the successful completion of the previous step.
- When developing parent and sub-requests, use the APIs described in Section 17.4, "Finding and Diagnosing Job Requests in Error State" in the parent request to determine the outcome of the sub-request. The state of the sub-request determines what to do next in the context of the parent request. The APIs enable the parent request to retrieve the state of the sub-request and determine whether any errors that have occurred in the sub-request are business or system errors.

17.4 Finding and Diagnosing Job Requests in Error State

You can use APIs to determine the following:

- The state of a job request,
- Which job requests have ended in error,
- The number of times a job request has been retried.

Alternatively, you can use Fusion Applications Control to search for job requests that have ended in error. For more information, see the section "Managing Logging for

Oracle Enterprise Scheduler" in the chapter "Managing Oracle Enterprise Scheduler Service and Jobs" in the *Oracle Fusion Applications Administrator's Guide*. You can also use an Oracle ADF UI to view logging information for Oracle Enterprise Scheduler jobs. For more information, see Section 9.17.3, "How to Log Scheduled Job Requests in an Oracle ADF UI."

17.4.1 Retrieving the State of a Job Request

Use the `RuntimeService.getRequestDetailBasic` API to retrieve the job request state. If the job request is in error state, retrieve the `ErrorType` of the job request to determine the type of terminal error that occurred. Example 17–1 shows sample code illustrating the use of the API.

Example 17–1 Retrieving the State of a Job Request

```
RequestDetail detail = runtime.getRequestDetailBasic(handle, requestId);
State state = detail.getState();

if (state == State.ERROR) {
    ErrorType errorType = detail.getErrorType();
    if (errorType == ErrorType.System) {
        // The job request had a system error.
    } else if (errorType == ErrorType.Business) {
        // The job request had a business error.
    }
}
```

For PL/SQL job requests, use the `get_error_type` API to determine the type of terminal error that has occurred. Example 17–2 shows sample code illustrating the use of the API.

Example 17–2 Retrieving the State of a PL/SQL Job Request

```
v_req_state       integer := null;
v_error_type      integer := null;

v_req_state := ess_runtime.get_request_state(v_request_id);
if v_req_state = ERROR_STATE then
    v_error_type := ess_runtime.get_error_type(v_request_id);
    if v_error_type = ETYPE_SYSTEM then
        -- The job request had a system error.
    elsif v_error_type = ETYPE_BUSINESS then
        -- The job request had a business error.
    end if;
end if;
```

17.4.2 Finding Job Requests with Business Errors

Use the `RuntimeService.queryRequests` API and include a match for the error state and `ErrorType` of business. Example 17–3 shows sample code illustrating the use of the API.

Example 17–3 Finding Job Requests with Business Errors

```
Filter filter = new Filter(
        RuntimeService.QueryField.STATE.fieldName(),
        Filter.Comparator.EQUALS,
        new Integer(State.ERROR.value()));
```

```
filter = filter.and(
        RuntimeService.QueryField.ERROR_TYPE.fieldName(),
        Filter.Comparator.EQUALS,
        new Integer(ErrorType.Business.value()));
Enumeration requests = runtime.queryRequests(handle, filter, null, false);
```

17.4.3 Determining the Number of Times a Job Request Has Been Retried

Use the `RuntimeService.getRequestDetailBasic` API to retrieve the job request retry count. The retry count is the number of times Oracle Enterprise Scheduler automatically retries the job request due to a system error. Example 17–4 shows sample code illustrating the use of the API.

Example 17–4 Determining the Number of Times a Job Request Has Been Retried

```
RequestDetail detail = runtime.getRequestDetailBasic(handle, requestId);
int retriedCount = detail.getRetriedCount();
if (retriedCount > 0) {
    // The job request has been retried the number of times indicated by
    // retriedCount.
} else {
    // The job request has not been retried.
}
```

For PL/SQL job requests, use the `get_retried_count` API to determine the number of times Oracle Enterprise Scheduler has automatically retried the job request. Example 17–5 shows sample code illustrating the use of the API.

Example 17–5 Determining the Number of Times a PL/SQL Job Request Has Been Retried

```
v_rcount      integer := null;

v_rcount := ess_runtime.get_retried_count(v_request_id);
if v_rcount > 0 then
    -- The job request has beem retried the number of times indicated by v_rcount.
else
    -- The job request has not been retried.
end if;
```