

Oracle® Fusion Middleware

Developer's Guide for Oracle Enterprise Scheduler

11g Release 1 (11.1.1.6.3)

E24713-05

August 2012

Documentation for developers that describes how to use Oracle Enterprise Scheduler to develop jobs that execute Java, PL/SQL, and binary process code to schedule and offload enterprise application work.

Oracle Fusion Middleware Developer's Guide for Oracle Enterprise Scheduler 11g Release 1 (11.1.1.6.3)

E24713-05

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

Primary Author: Steve Traut, 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

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface	xv
What's New in This Guide for Release 11.1.1.6.x	xvii
1 Introduction to Oracle Enterprise Scheduler	
1.1 About Oracle Enterprise Scheduler.....	1-1
1.2 Oracle Enterprise Scheduler Overview for Application Developers	1-2
1.2.1 Introduction to Working with Oracle Enterprise Scheduler at Design-Time	1-2
1.2.2 Introduction to Working with Oracle Enterprise Scheduler at Runtime	1-3
1.2.3 Oracle Enterprise Scheduler Job Requests	1-4
1.2.4 Overview of Integration Steps	1-6
1.3 Fixed-Rate Scheduling with Oracle Enterprise Scheduler	1-6
2 Verifying the Oracle Enterprise Scheduler Installation	
2.1 Introduction to Verifying the Oracle Enterprise Scheduler Installation.....	2-1
2.2 How to Verify the Oracle Enterprise Scheduler Installation Using a Browser.....	2-1
2.3 How to Programmatically Verify the Oracle Enterprise Scheduler Installation	2-2
2.4 What Happens When You Verify the Oracle Enterprise Scheduler Installation.....	2-3
2.5 What Happens at Runtime: How the Oracle Enterprise Scheduler Installation is Verified....	2-4
3 Using Ant to Generate a Hosting Application	
3.1 Introduction to Generating a Hosting Application with Ant	3-1
3.1.1 Prerequisites for Using the Ant Build Files.....	3-2
3.2 Ant Targets for Creating and Deploying a Hosting Application	3-2
3.3 Creating a Hosting Application and Project Workspace with Ant	3-3
3.4 Creating a Java Job as a Shared Library with Ant.....	3-5
3.5 Packaging a Java Job as a Shared Library with Ant.....	3-8
3.6 Deploying a Shared Library with Ant	3-8
3.7 Packaging a Hosting Application with Ant	3-8
3.8 Deploying a Hosting Application with Ant.....	3-9
3.9 Configuring the Generated Ant Targets.....	3-9

4 Using Oracle JDeveloper to Generate an Oracle Enterprise Scheduler Application

4.1	How to Start JDeveloper to Support Building Oracle Enterprise Scheduler Applications.....	4-1
4.2	Building a Combined Oracle Enterprise Scheduler Application	4-2
4.2.1	Creating the Application and Projects for EssDemoApp Application	4-3
4.2.1.1	How to Create the EssDemoApp Application and Host Project.....	4-3
4.2.1.2	How to Create the Client Project.....	4-4
4.2.2	Creating Metadata and an Implementation Class for the EssDemoApp Application	4-4
4.2.2.1	How to Create Metadata for the EssDemoApp Application	4-5
4.2.3	Adding Application Code to Submit Job Requests	4-7
4.2.3.1	How to Add Application Code to Submit Job Requests.....	4-7
4.2.4	Setting Oracle Enterprise Scheduler Properties	4-9
4.2.4.1	How to Set Oracle Enterprise Scheduler Properties for the Application	4-10
4.2.5	Assembling the EssDemoApp Application	4-10
4.2.5.1	How to Create the EJB-JAR Deployment Profile for the EssDemoApp	4-10
4.2.5.2	How To Update the WAR Archive Options.....	4-11
4.2.5.3	How to Update the EAR Options.....	4-12
4.2.6	Deploying and Running the EssDemoApp Application	4-12
4.2.6.1	How to Deploy the EssDemoApp Application.....	4-12
4.2.6.2	How to Run the EssDemoApp Sample Application	4-13
4.2.6.3	How to Purge Jobs in the EssDemoApp Sample Application	4-14
4.3	Building Split Submitting and Hosting Applications.....	4-14
4.3.1	How to Create the Back-End Hosting Application for EssDemoApp	4-15
4.3.1.1	Creating the Back-End Hosting Application.....	4-15
4.3.1.2	Configuring Security for the Back-End Hosting Application.....	4-16
4.3.1.3	Defining Metadata for the Back-End Hosting Application.....	4-16
4.3.1.4	Creating a Java Implementation Class in the Back-End Hosting Application .	4-18
4.3.1.5	Setting Oracle Enterprise Scheduler Properties	4-18
4.3.1.6	Assembling the Back-End Hosting Application for Oracle Enterprise Scheduler.....	4-19
4.3.1.6.1	How to Assemble the EJB JAR File for the Back-End Hosting Application	4-19
4.3.1.6.2	How to Assemble the MAR and EAR Files for the Back-End Hosting Application	4-20
4.3.1.7	Deploying the Back-End Hosting Application.....	4-20
4.3.2	How to Create the Front-End Submitter Application for Oracle Enterprise Scheduler ...	4-21
4.3.2.1	Creating the Front-End Submitter Application	4-21
4.3.2.2	Creating the SuperWeb Project.....	4-21
4.3.2.3	Configuring Security for the Front-End Submitter Application	4-22
4.3.2.4	Creating the HTTP Servlet for the Front-End Submitter Application.....	4-22
4.3.2.5	Editing the web.xml File for the Front-End Submitter Application	4-37
4.3.2.6	Editing the weblogic-application.xml file for the Front-End Submitter Application.	4-37
4.3.2.7	Editing the adf-config file for the Front-End Submitter Application	4-38

4.3.2.8	Assembling the Front-End Submitter Application for Oracle Enterprise Scheduler .	
	4-39	
4.3.2.8.1	How to Assemble the EJB JAR File for the Front-End Submitter Application.....	
	4-39	
4.3.2.8.2	How to Assemble the WAR File for the Front-End Submitter Application	
	4-39	
4.3.2.8.3	How to Assemble the MAR and EAR Files for the Front-End Hosting	
	Application 4-40	
4.3.2.9	Deploying the Front-End Submitter Application	4-40
4.3.2.10	Running the Split Application	4-40

5 Use Case Oracle Enterprise Scheduler Sample Application (Deprecated)

5.1	Introduction to the Oracle Enterprise Scheduler Sample Application	5-1
5.2	Creating the Application and Projects for the Sample Application	5-2
5.2.1	How to Create the EssDemoApp Application	5-2
5.2.2	How to Create a Project in the Sample Application	5-3
5.2.3	How to Set Project Properties for Oracle Enterprise Scheduler.....	5-6
5.3	Creating a Java Implementation Class for the Sample Application.....	5-7
5.3.1	How to Create a Java Class Using the Executable Interface.....	5-7
5.3.2	What Happens When You Create a Java Class That Implements the Executable	
	Interface 5-10	
5.3.3	What You Need to Know About the Executable Interface	5-10
5.4	Adding Application Code to Submit Oracle Enterprise Scheduler Job Requests	5-10
5.4.1	How to Add Required Libraries to Project	5-11
5.4.2	How to Create the EssDemo Servlet	5-11
5.5	Creating Metadata for Oracle Enterprise Scheduler Sample Application.....	5-14
5.5.1	How to Create a Job Type for Java	5-14
5.5.2	How to Create a Job Definition for Java	5-16
5.6	Assembling the Oracle Enterprise Scheduler Sample Application	5-18
5.6.1	How to Assemble the EJB Jar Files for Oracle Enterprise Scheduler Sample Application	
	5-18	
5.6.2	How to Assemble the MAR File for User Metadata	5-25
5.6.3	How to Assemble the EAR File for Oracle Enterprise Scheduler Sample Application.....	
	5-27	
5.6.4	Add oracle.ess Library Weblogic Application Descriptor.....	5-28
5.7	Deploying and Running the Oracle Enterprise Scheduler Sample Application	5-29
5.7.1	How to Deploy the EssDemoApp Application.....	5-29
5.7.2	How to Run the Oracle Enterprise Scheduler Sample Application	5-31
5.7.3	How to Purge Jobs in the Oracle Enterprise Scheduler Sample Application	5-32
5.8	Troubleshooting the Oracle Enterprise Scheduler Sample Application.....	5-33
5.8.1	How to Create the Oracle Enterprise Scheduler Database Schema	5-34
5.8.2	How to Drop the Oracle Enterprise Scheduler Runtime Schema.....	5-34
5.9	Using Submitting and Hosting Split Applications	5-35
5.9.1	How to Create the Back-End Hosting Application for Oracle Enterprise Scheduler	
	5-35	
5.9.1.1	Creating the Back-End Hosting Application.....	5-36
5.9.1.2	Configuring Security for the Back-End Hosting Application.....	5-36
5.9.1.3	Defining the Deployment Descriptors for the Back-End Hosting Application	5-37

5.9.1.4	Creating a Java Implementation Class in the Back-End Hosting Application .	5-41
5.9.1.5	Creating Metadata for the Back-End Hosting Application	5-42
5.9.1.6	Assembling the Back-End Hosting Application for Oracle Enterprise Scheduler.....	5-44
5.9.1.6.1	How to Assemble the EJB JAR File for the Back-End Hosting Application	5-44
5.9.1.6.2	How to Assemble the MAR and EAR Files for the Back-End Hosting Application	5-44
5.9.1.7	Deploying the Back-End Hosting Application.....	5-45
5.9.2	How to Create the Front-End Submitter Application for Oracle Enterprise Scheduler ...	5-45
5.9.2.1	Creating the Front-End Submitter Application	5-46
5.9.2.2	Configuring the ejb-jar.xml File for the Front-End Submitter Application	5-46
5.9.2.3	Creating the SuperWeb Project.....	5-48
5.9.2.4	Configuring Security for the Front-End Submitter Application	5-48
5.9.2.5	Creating the HTTP Servlet for the Front-End Submitter Application.....	5-49
5.9.2.6	Editing the web.xml File for the Front-End Submitter Application	5-63
5.9.2.7	Editing the weblogic-application.xml file for the Front-End Submitter Application.	5-63
5.9.2.8	Editing the adf-config file for the Front-End Submitter Application	5-65
5.9.2.9	Assembling the Front-End Submitter Application for Oracle Enterprise Scheduler .	5-65
5.9.2.9.1	How to Assemble the EJB JAR File for the Front-End Submitter Application.....	5-66
5.9.2.9.2	How to Assemble the WAR File for the Front-End Submitter Application	5-66
5.9.2.9.3	How to Assemble the MAR and EAR Files for the Front-End Hosting Application	5-66
5.9.2.10	Deploying the Back-End Hosting Application.....	5-67

6 Using the Metadata Service

6.1	Introduction to Using the Metadata Service	6-1
6.1.1	Introduction to Metadata Service Namespaces.....	6-2
6.1.2	Introduction to Metadata Service Operations	6-2
6.1.3	Introduction to Metadata Service Transactions	6-3
6.2	Accessing the Metadata Service.....	6-3
6.2.1	How to Access the Metadata Service with a Stateless Session EJB	6-3
6.3	Accessing the Metadata Service with Oracle JDeveloper	6-4
6.4	Querying Metadata Using the Metadata Service	6-4
6.4.1	How to Create a Filter	6-4
6.4.2	How to Query Metadata Objects	6-5

7 Using Parameters and System Properties

7.1	Introduction to Using Parameters and System Properties	7-1
7.1.1	What You Need to Know About Application Defined Property and System Property Naming	7-1
7.1.2	What You Need to Know About Parameter Conflict Resolution and Parameter Materialization	7-2

7.1.2.1	What You Need to Know About Job Definition Parameter Materialization	7-2
7.1.2.2	What You Need to Know About Job Set Level Parameter Materialization	7-3
7.2	Using Parameters with the Metadata Service	7-4
7.2.1	How to Use Parameters and System Properties in Metadata Objects	7-5
7.3	Using Parameters with the Runtime Service	7-6
7.3.1	How to Use Parameters with the Runtime Service.....	7-6
7.3.2	How to Use Parameters with a Step ID for Job Set Steps	7-7
7.4	Using System Properties	7-8

8 Creating and Using PL/SQL Jobs

8.1	Introduction to Using PL/SQL Stored Procedure Job Definitions	8-1
8.2	Creating a PL/SQL Stored Procedure for Oracle Enterprise Scheduler.....	8-2
8.2.1	How to Define a PL/SQL Stored Procedure with the Correct Signature.....	8-2
8.2.2	Handling Runtime Exceptions in an Oracle Enterprise Scheduler PL/SQL Stored Procedure 8-3	
8.2.3	How to Access Job Request Information In PL/SQL Stored Procedures.....	8-4
8.2.4	What You Need to Know When You Define a PL/SQL Stored Procedure	8-4
8.3	Performing Oracle Database Tasks for PL/SQL Stored Procedures.....	8-4
8.3.1	How to Grant PL/SQL Stored Procedure Permissions.....	8-5
8.3.2	What You Need to Know About Granting PL/SQL Stored Procedure Permissions	8-5
8.4	Creating and Storing Job Definitions for PL/SQL Job Types.....	8-6
8.4.1	How to Create a PL/SQL Job Type.....	8-6
8.4.2	How to Create and Store a Job Definition for PL/SQL Job Type.....	8-7
8.4.3	Using a PL/SQL Stored Procedure with an Oracle Enterprise Scheduler Application ...	8-7

9 Creating and Using Process Jobs

9.1	Introduction to Creating Process Job Definitions.....	9-1
9.2	Creating and Storing Job Definitions for Process Job Types	9-1
9.2.1	How to Create and Store a Process Job Type.....	9-2
9.2.2	How to Create and Store a Process Type Job Definition.....	9-4
9.3	Using a Perl Agent Handler for Process Jobs	9-5

10 Defining and Using Schedules

10.1	Introduction to Schedules	10-1
10.2	Defining a Recurrence	10-1
10.2.1	How to Define a Recurrence with a Recurrence Fields Helper	10-2
10.2.2	How to Define a Recurrence with an iCalendar RFC 2445 Specification	10-4
10.2.3	What You Need to Know When You Use a Recurrence Fields Helper	10-4
10.2.4	What You Need to Know When You Use an iCalendar Expression.....	10-6
10.3	Defining an Explicit Date.....	10-6
10.3.1	How to Define an Explicit Date	10-6
10.3.2	What You Need to Know About Explicit Dates.....	10-6
10.4	Defining and Storing Exclusions	10-7
10.4.1	How to Define an Exclusion.....	10-7
10.4.2	How to Create an Exclusions Definition	10-7

10.5	Defining and Storing Schedules.....	10-8
10.5.1	How to Define and Store a Schedule	10-8
10.5.2	What Happens When You Define and Store a Schedule	10-8
10.5.3	What You Need to Know About Handling Time Zones with Schedules.....	10-9
10.6	Identifying Job Requests That Use a Particular Schedule.....	10-9
10.7	Updating and Deleting Schedules.....	10-9

11 Using the Oracle Enterprise Scheduler Web Service

11.1	Introduction to the Oracle Enterprise Scheduler Web Service.....	11-1
11.2	Developing and Using ESSWebservice Applications.....	11-3
11.2.1	How to Develop and Use an ESSWebservice Java EE Application.....	11-3
11.2.2	How to Develop and Use an ESSWebservice SOA Application with BPEL.....	11-4
11.2.3	Setting Web Service Addressing Headers for getCompletionStatus() Operation...	11-4
11.2.4	Limitations for ESSWebservice.....	11-4
11.2.5	ESSWebservice Implementation.....	11-5
11.3	ESSWebservice WSDL File	11-5
11.4	Use Case Using Oracle Enterprise Scheduler ESSWebservice from a BPEL Process....	11-5
11.5	Creating the ESSWebService Application and a SOA Project.....	11-5
11.5.1	How to Create the ESSWebService Application and Project	11-5
11.6	Creating the ESSWebService Reference.....	11-6
11.6.1	How to Add the ESSWebService Partner Link.....	11-6
11.7	Adding the BPEL Process to Call the ESSWebService.....	11-9
11.7.1	How to Add a BPEL Process to Call the ESSWebService	11-9
11.7.2	Copy Types Into BPEL Process Schema	11-11
11.7.3	How to Invoke the ESSWebService submitRequest Operation	11-13
11.7.4	Assign Required Input Parameters for Request Submission	11-15
11.7.5	Invoke the getCompletionStatus Operation	11-21
11.7.6	Assign Input to the getCompletionStatus Operation	11-22
11.7.7	Receive the Job Completion Status.....	11-25
11.7.8	Return Result to Client.....	11-27
11.8	Using Additional ESSWebService Operations	11-30
11.8.1	How to Invoke the ESSWebService submitRecurringRequest Operation	11-31
11.8.2	How to Invoke the ESSWebService setSubmitArgs Operation	11-34
11.8.3	How to Invoke the ESSWebService addPPActions Operation	11-37
11.8.4	How to Invoke the ESSWebService setStepsArgs Operation	11-41
11.9	Securing the Oracle Enterprise Scheduler Web Service.....	11-46
11.9.1	How to Secure the Oracle Enterprise Scheduler Web Service	11-46
11.9.2	What Happens When You Secure the Oracle Enterprise Scheduler Web Service	11-48
11.10	Deploying and Testing the Project	11-48
11.10.1	How to Test the Web Service	11-48

12 Defining and Using Job Sets

12.1	Introduction to Defining and Using Job Sets.....	12-1
12.2	Defining Job Sets	12-2
12.2.1	How to Define a Job Set	12-2
12.2.2	How to Define Serial Job Set Steps.....	12-4
12.2.3	How to Define Parallel Job Set Steps	12-6

12.2.4	What Happens When You Define a Job Set.....	12-7
12.2.5	What You Need to Know About Serial Job Sets.....	12-7
12.2.6	What You Need to Know About Job Set Application Defined Properties and System Properties	12-8
12.2.7	What Happens at Runtime for Job Set State Priorities and State Transitions.....	12-8
12.3	Cross Application Job Sets.....	12-10
12.3.1	Overview of Cross Application Job Sets	12-11
12.3.2	Requirements for Cross Application Job Sets.....	12-11
12.4	Using Input and Output Forwarding	12-12
12.4.1	Supporting Input and Output Forwarding in Job Sets	12-12

13 Defining and Using a Job Incompatibility

13.1	Introduction to Using a Job Incompatibility	13-1
13.1.1	Job Self Incompatibility.....	13-2
13.2	Defining Incompatibility with Oracle JDeveloper	13-2
13.2.1	How to Define a Global Incompatibility	13-2
13.2.2	How to Define a Domain Incompatibility.....	13-4
13.3	What Happens at Runtime to Handle Job Incompatibility	13-6
13.3.1	What Happens to Subrequests with an Incompatible Parent Request	13-6
13.3.2	What Happens to the Scope of Request Incompatibility	13-6

14 Using the Runtime Service

14.1	Introduction to the Runtime Service.....	14-1
14.2	Accessing the Runtime Service	14-1
14.2.1	How to Access the Runtime Service and Obtain a Runtime Service Handle	14-2
14.3	Submitting Job Requests	14-2
14.3.1	How to Submit a Request to the Runtime Service.....	14-3
14.3.2	What You Should Know About Default System Properties When You Submit a Request	14-3
14.3.3	What You Should Know About Metadata When You Submit a Request	14-4
14.4	Managing Job Requests.....	14-4
14.4.1	How to Get Job Request Information with getRequestDetail	14-4
14.4.2	How to Change Job Request State.....	14-5
14.4.3	How to Update Job Request Priority and Job Request Parameters.....	14-6
14.5	Querying Job Requests.....	14-7
14.6	Submitting Ad Hoc Job Requests	14-9
14.6.1	How to Create an Ad Hoc Request.....	14-10
14.6.2	What Happens When You Create an Ad Hoc Request.....	14-11
14.6.3	What You Need to Know About Ad Hoc Requests.....	14-11
14.7	Implementing Pre-Process and Post-Process Handlers	14-11
14.7.1	Implementing a Pre-Process Handler	14-12
14.7.1.1	Implementing the PreProcessHandler Interface	14-12
14.7.2	Implementing a Post-Process Handler	14-12
14.7.2.1	Implementing the PostProcessHandler Interface	14-13

15 Using Subrequests

15.1	Introduction to Using Subrequests.....	15-1
15.2	Sample Subrequest.....	15-2
15.3	Creating and Managing Subrequests.....	15-3
15.3.1	How to Submit Subrequests.....	15-3
15.3.2	How to Cancel Subrequests	15-3
15.3.3	How to Hold Subrequests	15-4
15.3.4	How to Delete Subrequests	15-4
15.3.5	How to Submit Multiple Subrequests	15-4
15.3.6	How to Manage Paused Subrequests	15-4
15.3.6.1	Indicating Paused Status	15-4
15.3.6.2	Storing the Paused State for a Parent Request	15-4
15.3.7	How Subrequests Are Processed.....	15-5
15.3.8	How to Identify Subrequests	15-6
15.3.9	How to Manage Subrequests and Incompatibility	15-6
15.4	Creating a Java Procedure that Submits a Subrequest	15-6
15.5	Creating a PL/SQL Procedure that Submits a Subrequest.....	15-9

16 Working with Asynchronous Java Jobs

16.1	Introduction to Working with Asynchronous Java Jobs.....	16-1
16.2	Creating an Asynchronous Java Job.....	16-1
16.2.1	Implementing the Asynchronous Java Job Asynchronous Interface	16-2
16.2.2	Asynchronous Java Job execute() Method	16-2
16.2.3	Invoking a Remote Job from an Asynchronous Java Job.....	16-2
16.2.4	Calling Back to Oracle Enterprise Scheduler with Status Updates	16-3
16.2.5	Updating the Asynchronous Java Job.....	16-3
16.2.6	Notifying Oracle Enterprise Scheduler When an Asynchronous Job Completes ...	16-3
16.2.6.1	Using the Web Service to Notify When an Asynchronous Job Completes.....	16-4
16.2.6.2	Using EJB to Notify When an Asynchronous Job Completes.....	16-4
16.2.7	Asynchronous Java Job AsyncCancellable Interface	16-4
16.2.8	Sample Asynchronous Java Job Invoking a BPEL Process Through Event Delivery Network 16-5	
16.2.8.1	Sample BPEL Process Design Time with Oracle Enterprise Scheduler.....	16-7
16.3	A Use Case Illustrating the Implementation of a BPEL Process as an Asynchronous Job	16-10
16.3.1	Introduction to the Recommended Design Pattern.....	16-11
16.3.2	Potential Approaches	16-11
16.3.3	Use Case Summary.....	16-11
16.4	How to Implement BPEL with an Asynchronous Job.....	16-12
16.4.1	Use Case: Add Oracle JDeveloper Libraries	16-12
16.4.2	Use Case: Create the Asynchronous Job Definition	16-13
16.4.3	Use Case: Design the Event Payload Schema and Event Definition Files.....	16-14
16.4.4	Programmatically Raise a Business Event from the Asynchronous Job Methods	16-15
16.4.5	Design the SOA Composite with Mediator and BPEL	16-17
16.4.6	Add Fault Handling and Correlated onMessage Branch for Error and Cancel Job	16-18
16.4.6.1	Create Correlation Set and Define Initiate Activity.....	16-20

16.4.6.2	Create the onMessage Branch with Use of Correlation Set.....	16-21
16.4.6.3	Create the Fault Branch	16-22
16.4.6.4	Populate the onMessage and Fault Branch.....	16-23
16.4.7	Validating the Deployment	16-24
16.4.8	Troubleshooting the Use Case	16-26
16.5	Handling Time Outs and Recovery for Asynchronous Jobs	16-26
16.5.1	Asynchronous Request Time Outs.....	16-26
16.5.1.1	Setting the Time Out Value.....	16-27
16.5.1.2	Discovering the Asynchronous Job Requests that Have Timed Out.....	16-27
16.5.1.3	Completing Asynchronous Requests without a Time Out	16-27
16.5.1.4	What Happens When an Asynchronous Job Request Times Out	16-27
16.5.2	Handling Asynchronous Jobs Marked for Manual Recovery.....	16-28
16.5.3	Using RecoverRequest to Manually Recover a Job Request.....	16-28
16.6	Oracle Enterprise Scheduler Interfaces and Classes	16-29

17 Creating Job Request Logs and Output

17.1	Creating Request Logs	17-1
17.1.1	System Properties	17-1
17.1.2	APIs for Handling Request Logs.....	17-2
17.1.3	Log Header	17-2
17.1.4	Creating Request Logs from a Java Job	17-2
17.1.4.1	APIs for Java Job Logging	17-3
17.1.4.2	Example.....	17-3
17.1.5	Creating Request Logs from a PL/SQL Job.....	17-4
17.1.5.1	ESS_JOB Package Support for Creating Logs.....	17-4
17.1.5.2	PL/SQL Request Logging Example.....	17-5
17.1.6	Creating Request Logs from a Process Job.....	17-6
17.2	Creating Request Output	17-6
17.2.1	Using the Request File Directory	17-6
17.2.1.1	Common Request File Directory Behavior	17-7
17.2.1.2	Shared Request File Directory Behavior	17-7
17.2.1.2.1	Error Handling When a Shared Request File Directory is Used	17-7
17.2.1.3	Local Request File Directory Behavior	17-7
17.2.1.3.1	Error Handling When a Local Request File Directory is Used.....	17-8
17.2.2	System Properties	17-8
17.2.3	APIs for Handling Request Output	17-9
17.2.4	Creating Request Output from a Java Job	17-10
17.2.4.1	APIs for Handling Request Output from a Java Job	17-10
17.2.4.2	Example.....	17-13
17.2.5	Creating Request Output from a PL/SQL Job	17-14
17.2.5.1	PL/SQL Package Support for Creating Output.....	17-14
17.2.5.2	PL/SQL Output Creation Examples.....	17-16
17.2.6	Creating Request Output from a Process Job	17-19

18 Oracle Enterprise Scheduler Security

18.1	Introduction to Oracle Enterprise Scheduler Security.....	18-1
------	---	------

18.1.1	Oracle Enterprise Scheduler Metadata Access Control	18-1
18.1.2	Oracle Enterprise Scheduler Job Execution Security	18-2
18.2	Configuring Metadata Security for Oracle Enterprise Scheduler	18-2
18.2.1	How to Enable Application Security with Oracle ADF Security Wizard.....	18-3
18.2.2	How to Define Principals for Security	18-3
18.2.3	How to Create Grants with Oracle Enterprise Scheduler Metadata Pages.....	18-4
18.2.4	How to Create Grants with Oracle ADF Security Wizard.....	18-5
18.2.5	About MetadataPermission APIs	18-7
18.2.6	What Happens When You Configure Metadata Security.....	18-7
18.3	Configuring Web Service Security for Oracle Enterprise Scheduler.....	18-8
18.4	Configuring PL/SQL Job Security for Oracle Enterprise Scheduler	18-8
18.5	Elevating Privileges for Oracle Enterprise Scheduler Jobs	18-8
18.6	Configuring a Single Policy Stripe in Oracle Enterprise Scheduler	18-8
18.6.1	How to Configure a Single Policy Stripe in Oracle Enterprise Scheduler.....	18-9
18.6.2	What Happens When You Configure a Single Policy Stripe	18-10
18.6.3	What Happens at Runtime	18-10

List of Tables

3-1	Ant Targets in the Included Build File.....	3-2
3-2	Ant Targets in the Generated Build File.....	3-3
3-3	Information Needed by the Ant Target.....	3-3
3-4	Information Needed by the Ant Target.....	3-6
3-5	Build Properties for Customizing Ant Builds.....	3-9
4-1	EJB Resources for the Front-End Submitter Application.....	4-37
5-1	EJB Resources for the Front-End Submitter Application.....	5-63
6-1	Filter Comparison Operators.....	6-4
7-1	Parameter Precedence Levels.....	7-2
7-2	ParameterInfo Parameter Properties.....	7-4
8-1	Terminal States for PL/SQL Stored Procedure Results.....	8-3
10-1	Recurrence Field Helper Patterns.....	10-2
11-2	Submit Request Web Service Arguments for BPEL Assign Activity Mapping.....	11-15
11-3	Submit Recurring Request Web Service Arguments for BPEL Assign Activity Mapping.....	11-33
14-2	Runtime Service Get Request Methods.....	14-5
14-3	Runtime Service Job Request State Methods.....	14-5
14-4	Runtime Service Update Methods.....	14-7
17-1	RuntimeService Methods for Handling Request Logs.....	17-2
17-2	ContentFactory Methods for Creating Request Logs.....	17-3
17-3	RequestLogger Methods for Creating Request Logs.....	17-3
17-4	ESS_JOB Functions and Procedures for Request Logging.....	17-5
17-5	System Properties for Creating Request Output.....	17-9
17-6	RuntimeService Methods for Handling Request Output.....	17-9
17-7	ContentFactory Methods for Java Request Output.....	17-10
17-8	RequestOutput Methods for Java Request Output.....	17-10
17-9	OutputContentHelper Methods for Java Request Output.....	17-11
17-10	CommitSemantics Enum Members to Express Commit Semantics.....	17-12
17-11	ESS_JOB Procedures and Functions for Request Output.....	17-14
18-1	Sample Permission Grants for Security Using Oracle ADF.....	18-7
18-2	Grant Actions for Metadata Security.....	18-7

Preface

This document describes how to develop jobs and other extensions of Oracle Enterprise Scheduler.

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 Middleware 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>.

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 Middleware Administrator's Guide for Oracle Enterprise Scheduler*

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.
<i>italic</i>	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.

What's New in This Guide for Release 11.1.1.6.x

For Release 11.1.1.6.x, this guide has been updated in several ways. The following table lists the sections that have been added or changed. If a feature was not available in the first release of 11.1.1.6.x, the last columns denote which documentation release contains the update.

For a list of known issues (release notes), see the "Known Issues for Oracle SOA Products and Oracle AIA Foundation Pack" at <http://www.oracle.com/technetwork/middleware/docs/soa-aiafp-knownissuesindex-364630.html>.

Sections	Changes Made	11.1.1.6.0	11.1.1.6.1	11.1.1.6.2	11.1.1.6.3
All Chapters					
All Sections	Created this book from content originally included with Oracle Fusion Applications.	X			
Chapter 3 Using Ant to Generate a Hosting Application					
Chapter 3, "Using Ant to Generate a Hosting Application"	Added chapter describe how to use new Ant targets to generate a hosting application and supporting components.	X			
Chapter 4 Using Oracle JDeveloper to Generate an Oracle Enterprise Scheduler Application					
Chapter 4, "Using Oracle JDeveloper to Generate an Oracle Enterprise Scheduler Application"	Added chapter describing how to use new Oracle JDeveloper user interface create an application.	X			
Chapter 5 Use Case Oracle Enterprise Scheduler Sample Application					
Chapter 5, "Use Case Oracle Enterprise Scheduler Sample Application (Deprecated)"	Deprecated content in this chapter. New development should use the content in Chapter 4, "Using Oracle JDeveloper to Generate an Oracle Enterprise Scheduler Application"	X			

Introduction to Oracle Enterprise Scheduler

This chapter introduces Oracle Enterprise Scheduler as a service for developing jobs that offload work such as executing Java, PL/SQL, and binary process code.

- [Section 1.1, "About Oracle Enterprise Scheduler"](#)
- [Section 1.2, "Oracle Enterprise Scheduler Overview for Application Developers"](#)
- [Section 1.3, "Fixed-Rate Scheduling with Oracle Enterprise Scheduler"](#)

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 Middleware 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 Middleware 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

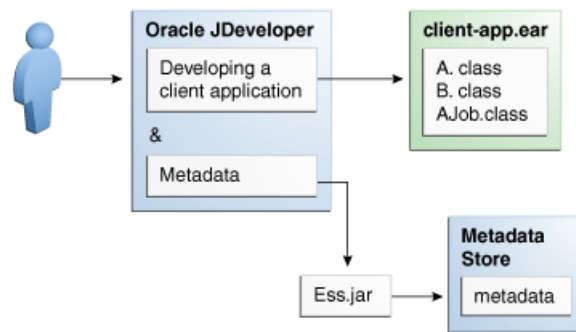
Oracle Enterprise Scheduler is primarily a Java EE application that provides time- and schedule-based callbacks to other applications to run their jobs. Oracle Enterprise Scheduler compares with the Calendar application you might use in your phone or the Oracle Calendar, where you create events and meetings 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. The Fusion Middleware Control can provide an interface for interacting with Oracle Enterprise Scheduler. Typically, however, you will provide a client application with which the end user can set up 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.

You can create the following types of metadata at design time.

- **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, 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.

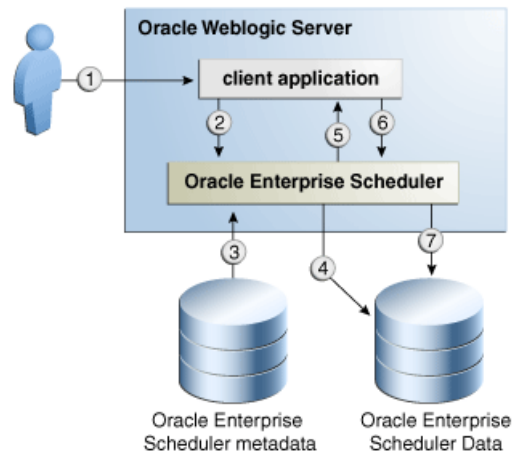
Figure 1–2 Oracle Enterprise Scheduler Runtime Integration

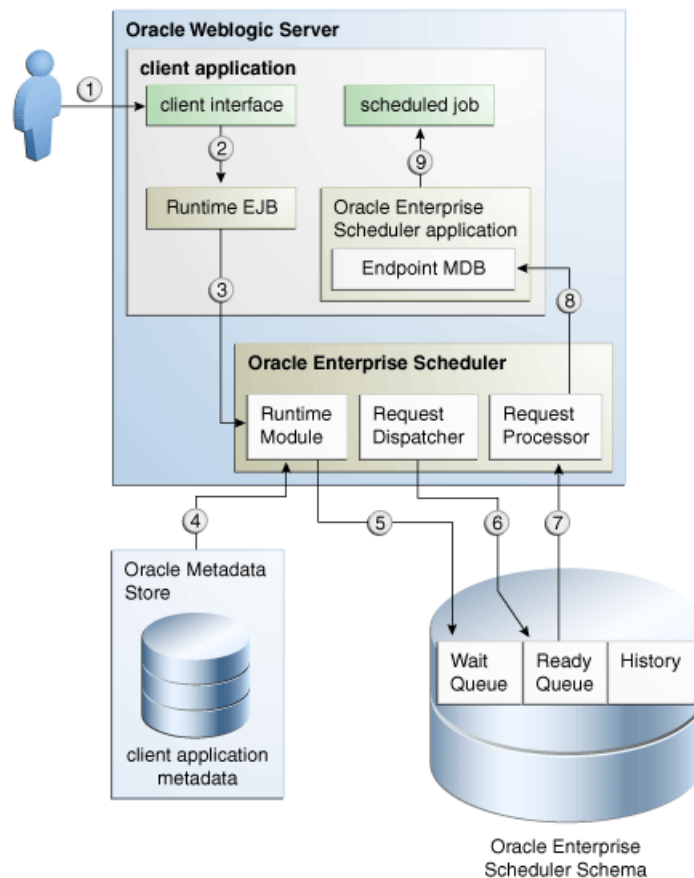
Figure 1–2 shows the sequence involved with running an application using Oracle Enterprise Scheduler, and the following steps:

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 set up 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 Middleware 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.

Verifying the Oracle Enterprise Scheduler Installation

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

- [Section 2.1, "Introduction to Verifying the Oracle Enterprise Scheduler Installation"](#)
- [Section 2.2, "How to Verify the Oracle Enterprise Scheduler Installation Using a Browser"](#)
- [Section 2.3, "How to Programmatically Verify the Oracle Enterprise Scheduler Installation"](#)
- [Section 2.4, "What Happens When You Verify the Oracle Enterprise Scheduler Installation"](#)
- [Section 2.5, "What Happens at Runtime: How the Oracle Enterprise Scheduler Installation is Verified"](#)

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
```

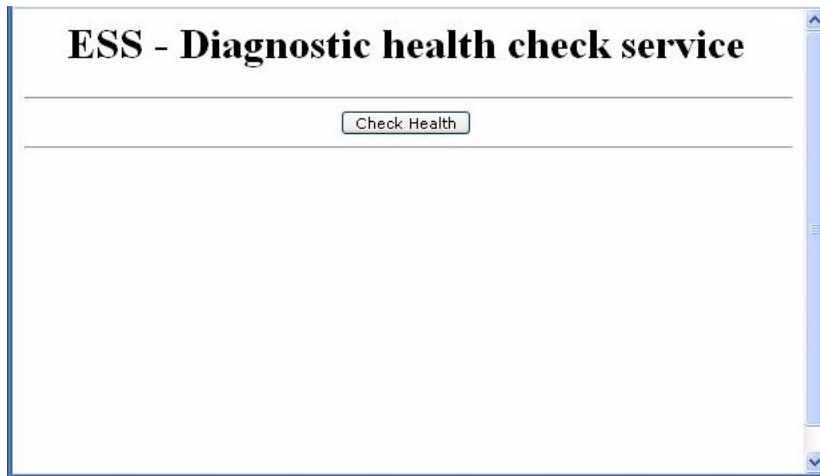
where `hostName` is the server to which Oracle Enterprise Scheduler is installed and `port` is the port number.

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. Log in 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 the Work of Oracle Enterprise Scheduler Jobs" in *Oracle Fusion Middleware Administrator's Guide for Oracle Enterprise Scheduler*.

Using Ant to Generate a Hosting Application

This chapter describes how you can use Ant targets from a build.xml file included with Oracle Enterprise Scheduler to create a hosting application for use with Java jobs.

Using these targets, you can create the application artifacts in an Oracle JDeveloper workspace, create a template for a Java job implementation, and package and deploy both the application and the Java job (as a shared library).

Note that the Ant targets described here do not create a client user interface with which users can interact with the job. To perform client tasks, you can use Fusion Middleware Control or develop a client user interface with Oracle JDeveloper.

When you have created and deployed your application and shared library, you can use JDeveloper or Enterprise Manager to associate metadata with the deployed outputs.

This chapter includes the following sections:

- [Section 3.1, "Introduction to Generating a Hosting Application with Ant"](#)
- [Section 3.2, "Ant Targets for Creating and Deploying a Hosting Application"](#)
- [Section 3.3, "Creating a Hosting Application and Project Workspace with Ant"](#)
- [Section 3.4, "Creating a Java Job as a Shared Library with Ant"](#)
- [Section 3.5, "Packaging a Java Job as a Shared Library with Ant"](#)
- [Section 3.6, "Deploying a Shared Library with Ant"](#)
- [Section 3.7, "Packaging a Hosting Application with Ant"](#)
- [Section 3.8, "Deploying a Hosting Application with Ant"](#)
- [Section 3.9, "Configuring the Generated Ant Targets"](#)

3.1 Introduction to Generating a Hosting Application with Ant

Oracle Enterprise Scheduler includes an Ant build file through which you can generate the basic artifacts you'll need to get a hosting application running, along with a Java job you can deploy to be executed by the application.

You use the included Ant build file to generate a hosting application. When you do, you also generate another Ant build file that contains targets you can use to generate artifacts for a Java job, as well as to build and deploy the generated components.

When you have created and deployed your application and shared library, you can use JDeveloper or Enterprise Manager to associate metadata with the deployed outputs.

You can also use a generated build.properties file to customize the work Ant does by setting values for variables a target uses when it runs.

The steps described in this chapter include the following you can do with Ant.

1. Create a hosting application that can execute jobs. Use the create-user-home in the included build.xml file.
2. Create a JDeveloper project workspace through which you can edit application artifacts with the IDE. This is done when you create the hosting application.
3. Create an Ant build file with targets for building and deploying parts of the application.
4. Create a Java job template to which you can add business logic. Use the create-new-job-def target in the generated build.xml file.
5. Package the implemented Java job as a shared library. Use the package_essjob_library target in the generated build.xml file.
6. Deploy the shared library to the hosting application. Use the deploy_essjob_library target in the generated build.xml file.
7. Package the hosting application. Use the package_hosting_app target in the generated build.xml file.
8. Deploy the hosting application. Use the deploy_hosting_app target in the generated build.xml file.

3.1.1 Prerequisites for Using the Ant Build Files

Before you get started with the provided and generated build files, make sure you're set up with the following prerequisites:

- You must have Ant installed and set up, with the ANT_HOME variable set properly and the PATH pointing to ant's bin directory.
- You must install and set up Oracle JDeveloper. Your PATH variable must contain the Oracle JDeveloper bin directory so that the jdev command can be executed from the command prompt.

3.2 Ant Targets for Creating and Deploying a Hosting Application

Oracle Enterprise Scheduler includes an Ant build file to get you started toward deploying a hosting application that can execute jobs. However, you're actually using two build files to finish the job: one that is included with Oracle Enterprise Scheduler and another that is generated by a target in the included build file. The following tables list and describe the targets that are included by default in the two files.

By default, the included build.xml file is located in the Oracle Enterprise Scheduler extensibility_scripts directory. For example, in an Oracle JDeveloper installation, you'll find them in MW_HOME/jdeveloper/extensibility_scripts/build.xml; with installations of products that include Oracle Enterprise Scheduler, you'll probably find them in an ORACLE_HOME/extensibility_scripts directory.

Table 3–1 Ant Targets in the Included Build File

Ant Target	Description
create-user-home	Default target to create a user home.
help-create-user-home	Help on creating a user home.

When you run the create-user-home target from the included build.xml file, one of the target's actions is to create another build.xml file. That file contains the following targets that you can use to create, build and deploy artifacts for your application.

Table 3–2 Ant Targets in the Generated Build File

Ant Target	Description
build_ears	Package the job shared library and the hosting application.
create-new-job-def	Create Java job as a shared library.
deploy	Package and deploy the job library and hosting application.
deploy_essjob_library	Deploy the Java job shared library.
deploy_hosting_app	Deploy the hosting application.
deploy_job_logic	Package and deploy the job shared library.
package_essjob_library	Package the Java job as a shared library.
package_hosting_app	Package the hosting application.

3.3 Creating a Hosting Application and Project Workspace with Ant

You can create a hosting application by running the create-user-home Ant target in the build.xml file included with Oracle Enterprise Scheduler.

After the script completes successfully, you'll have the artifacts for a hosting application that you can package and deploy. The artifacts will be generated within a JDeveloper-compatible workspace in the target directory you specified. The created workspace will have a build.xml that you can use to build, package and deploy the hosting application and the generated Java job as a shared library.

As the target runs, you'll be prompted to enter details that guide the target's work. These details include the environment for which the target's work is intended (such as to run with a particular application), the new application's name and target directory, and so on.

Before you get started, you should have in hand the following information for which you'll be prompted by the Ant target:

Table 3–3 Information Needed by the Ant Target

Input Prompt	Description
Which template should be used	<p>Possible values are "Fusion" and "Standalone". If you're developing for use with Oracle Fusion Applications, enter Fusion here.</p> <p>If you're not developing for use with Oracle Fusion Applications, enter "Standalone."</p> <p>There are significant differences between the Oracle Fusion Applications and standalone contexts. For example, in the Oracle Fusion Applications context, the target generates a slightly different hosting application, as well as a client application.</p>
Middleware Home directory path	The Middleware Home directory that was created when Oracle Enterprise Scheduler was installed (probably with another product that embeds it). The locations of supporting libraries will be found as relative to this directory.
Hosting application name	The name you want the new hosting application to have.

Table 3–3 (Cont.) Information Needed by the Ant Target

Input Prompt	Description
Hosting application JPS stripe ID	A stripe is a security construct that defines the subset of values in the policy store that the application intends to use. At run time, it determines which set of policies are applicable for the application. The application name is often used.
Shared library name for job business logic	The name for the shared library into which the generated Java job source code should be placed.
Empty directory where the application will be created	The directory where you want the generated files to go. This will be the location of the JDeveloper workspace, where artifacts such as the build.xml file you'll use later will be created.

To create a hosting application with Ant

- To get started, open a console window and change directory to where the included build.xml is located. By default, this is the Oracle Enterprise Scheduler extensibility_scripts directory. For example, in MW_HOME/jdeveloper/extensibility_scripts/build.xml.

Run the target with a command such as the following. You can omit the target name because it is the default target in the build file.

```
ant
```

If you want to use the target name, you can do so with the following command.

```
ant create-user-home
```

In the following example of Ant console output, note that the prompts begin with the word "[input]". For each prompt, type the value you want to use, then press Enter.

After you've entered the information needed, the target creates the directories and files you requested, copying needed files into your new workspace and setting up some of the configuration for the new hosting application.

Example 3–1 Console Output for the create-user-home Target

```
[extensibility_scripts]$ ant
Buildfile: build.xml
-init:
create-user-home:
[input] Enter which template should be used (source_template) (default=Fusion)
      [input]      ([Fusion], Standalone)
Standalone
      [input] Enter Middleware Home Directory path (fmw_home_dir) (default=) []
/scratch/fmwtools/mw_home
      [input] Enter hosting application name (hosting_application_name)
(default=MyAppEss) [MyAppEss]
NewDemoApp
      [input] Enter hosting application JPS stripe id (hosting_application_stripe_
id) (default=MyAppEss) [MyAppEss]
NewDemoApp
      [input] Enter the shared library name for the job business logic (jobdef_
library_name) (default=MyJobsLibrary) [MyJobsLibrary]
```



```

NewDemoAppJobsLib
  [input] Enter an empty directory where the applications will be created (user_
home)
/scratch/WLServers/MW_HOME/standalone_apps/NewDemoApp
  [echo]
  [echo]
  [mkdir] Created dir: /scratch/WLServers/MW_HOME/standalone_apps/NewDemoApp
[propertyfile] Creating new property file: /scratch/WLServers/MW_HOME/standalone_
apps/NewDemoApp/template.properties
  [copy] Copying 12 files to /scratch/WLServers/MW_HOME/standalone_
apps/NewDemoApp
  [copy] Copied 25 empty directories to 8 empty directories under
/scratch/WLServers/MW_HOME/standalone_apps/NewDemoApp
  [copy] Copying 1 file to /scratch/WLServers/MW_HOME/standalone_
apps/NewDemoApp/ant/config
  [copy] Copying 1 file to /scratch/WLServers/MW_HOME/standalone_
apps/NewDemoApp
  [copy] Copying 15 files to /scratch/WLServers/MW_HOME/standalone_
apps/NewDemoApp
  [move] Moving 1 file to /scratch/WLServers/MW_HOME/standalone_
apps/NewDemoApp/Template_Hosting
  [echo]
  [echo] =====
  [echo]
  [echo] A new workspace has been created at: /scratch/WLServers/MW_
HOME/standalone_apps/NewDemoApp
  [echo] This workspace can be opened and modified using JDeveloper
  [echo] To deploy the applications, run the following command:
  [echo]     ant -f /scratch/WLServers/MW_HOME/standalone_
apps/NewDemoApp/ant/build-ess.xml deploy
  [echo] To create new jobs from predefined templates, run the following
command:
  [echo]     ant -f /scratch/WLServers/MW_HOME/standalone_
apps/NewDemoApp/build.xml create-new-job-def

BUILD SUCCESSFUL
Total time: 1 minute 27 seconds

```

3.4 Creating a Java Job as a Shared Library with Ant

You can create a Java job class template by running the `create-new-job-def` Ant target that's in the build file generated when you created a new hosting application. (For more information, see [Section 3.3, "Creating a Hosting Application and Project Workspace with Ant"](#) for more information.)

The Java class you create here is a template to which you can add logic that implements your Java job. A Java job executes Java code. With the Java job implemented, you can add metadata that comprises some of the specifics for the job.

Note: Currently, you can create only synchronous Java job templates with this Ant target.

As the target runs, you'll be prompted to enter details that guide the target's work. Before you get started, you should have in hand the following information for which you'll be prompted by the Ant target:

Table 3–4 Information Needed by the Ant Target

Input Prompt	Description
Number of job definition template to create	A number corresponding to the type of Java job implementation you're creating. Currently, only synchronous Java jobs can be created this way, so the only supported value is "1".
Java package name for job definition	The package name for the Java job you're creating.
Java class name for job definition	The class name for the Java job you're creating.

To create a Java job class template with Ant:

1. To get started, in a console window change directory to the directory you specified as the location to create the application. The build.xml file should be there. Use the following command to run the target:

```
ant create-new-job-def
```

In the following example of Ant console output, you can see where the prompts occur. After you've entered that information, the target creates the file you requested, copying needed files into your new workspace and setting up some of the configuration for the new hosting application.

Example 3–2 Console Output of the create-new-job-def Target

```
[extensibility_scripts]$ ant -f /scratch/WLServers/MW_HOME/standalone_
apps/NewDemoApp/build.xml create-new-job-def

Buildfile: /scratch/WLServers/MW_HOME/standalone_apps/NewDemoApp/build.xml

-init:

create-new-job-def:
    [echo] Available Job Definition Templates:
    [echo]     1) Simple Synchronous Java Job
    [input] Enter number of job definition template to create (job_template_to_
create)
    1
    [echo] Calling default target on /scratch/miscFiles/ExtnDemo/extensibility_
scripts/Standalone/Template_JobLibrary/simple_synchronous_job/build.xml

-init:

create-job-definition:
    [input] Enter Java package name for Job Definition (jobdef_package_name)
(default=oracle.apps.ess.custom) [oracle.apps.ess.custom]
oracle.apps.ess.custom
    [input] Enter Java class name for Job Definition (jobdef_class_name)
(default=MySynchronousJavaJob) [MySynchronousJavaJob]
NewDemoHelloWorld
    [copy] Copying 1 file to /scratch/WLServers/MW_HOME/standalone_
apps/NewDemoApp/NewDemoApp/EssSharedLibrary/src
    [copy] Copying 1 file to /scratch/WLServers/MW_HOME/standalone_
apps/NewDemoApp/NewDemoApp/EssSharedLibrary/src/oracle/apps/ess/custom

BUILD SUCCESSFUL
Total time: 34 seconds
```

2. Having created the class template for the Java job, you can add code that implements the job's logic. The template is located in project in the JDeveloper workspace you created when you created the hosting application in [Section 3.3, "Creating a Hosting Application and Project Workspace with Ant"](#). The file's directory path is shown in the Ant console output. You can use the editor you prefer for editing Java code, such as JDeveloper or a simple text editor.

Open the Java file and add code to implement the `execute()` method.

[Example 3-3](#) shows what the generated code will look like. You would replace the simple implementation of the `oracle.as.scheduler.Executable` interface's `execute()` method with code that does your Java job's work.

Example 3-3 Oracle Enterprise Scheduler HelloWorld Java Class

```
package oracle.apps.ess.custom;

import java.io.StringWriter;
import java.security.AccessControlContext;
import java.security.AccessController;
import javax.security.auth.Subject;

import oracle.as.scheduler.RequestParameters;
import oracle.as.scheduler.job.BaseSynchronousJavaJob;
import oracle.as.scheduler.request.ContentType;
import oracle.security.jps.util.SubjectUtil;

public class NewDemoHelloWorld extends BaseSynchronousJavaJob {

    public NewDemoHelloWorld() {
        super();
    }

    protected void execute() throws Exception
    {
        long requestId = getRequestExecutionContext().getRequestId();
        RequestParameters params = getRequestParameters();
        AccessControlContext accContext = AccessController.getContext();
        Subject subject = Subject.getSubject(accContext);
        String username = SubjectUtil.getUserName(subject);
        /*
         * Write contents to request log
         */
        StringWriter strWriter = new StringWriter();
        strWriter.write("Simple ESS Java job execution LOG");
        strWriter.write("ESS Job requestId: " + requestId);
        strWriter.write("Username: " + username);
        writeToRequestLog(requestId, strWriter.toString());

        /*
         * Write Text contents to request output
         */
        strWriter = new StringWriter();
        strWriter.write("Simple ESS Java job execution Text Out");
        strWriter.write("ESS Job requestId: " + requestId);
        strWriter.write("Username: " + username);
        writeToRequestOutput(requestId, strWriter.toString(), ContentType.Text);
    }
}
```

3.5 Packaging a Java Job as a Shared Library with Ant

You can package a Java job implementation by running the `package_essjob_library` Ant target.

Note: The build file containing this target is generated when you create a new hosting application. (For more information, see [Section 3.3, "Creating a Hosting Application and Project Workspace with Ant"](#).)

The `package_essjob_library` target compiles and JARs the job code. The target simply runs to completion, requiring no user input.

To package a Java job class implementation with Ant:

- In a console window change directory to the directory you specified as the location to create the hosting application. Use the following command to run the target:

```
ant package_essjob_library
```

3.6 Deploying a Shared Library with Ant

You can deploy a Java job shared library by running the `deploy_essjob_library` Ant target.

Note: The build file containing this target is generated when you create a new hosting application. (For more information, see [Section 3.3, "Creating a Hosting Application and Project Workspace with Ant"](#).)

The `deploy_essjob_library` target deploys the job library. The target simply runs to completion, requiring no user input.

To deploy a Java job shared library with Ant:

- In a console window change directory to the directory you specified as the location to create the hosting application. Use the following command to run the target:

```
ant deploy_essjob_library
```

3.7 Packaging a Hosting Application with Ant

You can package a hosting application by running the `package_hosting_app` Ant target.

Note: The build file containing this target is generated when you create a new hosting application. (For more information, see [Section 3.3, "Creating a Hosting Application and Project Workspace with Ant"](#).)

The `package_hosting_app` target packages the hosting app created with the `create-user-home` target (for more information, see [Section 3.3, "Creating a Hosting](#)

[Application and Project Workspace with Ant](#)"). The target simply runs to completion, requiring no user input.

To package a hosting application with Ant:

- In a console window change directory to the directory you specified as the location to create the hosting application. Use the following command to run the target:

```
ant package_hosting_app
```

3.8 Deploying a Hosting Application with Ant

You can deploy a hosting application by running the `deploy_hosting_app` Ant target.

Note: The build file containing this target is generated when you create a new hosting application. (For more information, see [Section 3.3, "Creating a Hosting Application and Project Workspace with Ant"](#).)

The `deploy_hosting_app` target deploys the hosting app created with the `create-user-home` target (for more information, see [Section 3.3, "Creating a Hosting Application and Project Workspace with Ant"](#)). This target simply runs to completion, requiring no user input.

To deploy a hosting application with Ant:

- In a console window change directory to the directory you specified as the location to create the hosting application. Use the following command to run the target:

```
ant deploy_hosting_app
```

3.9 Configuring the Generated Ant Targets

The file `<user_home>/ant/config/ess-build.properties` contains various parameters to specify information used by the Ant scripts during build, packaging and deployment. The `<user_home>` is the directory specified to contain the application workspace in step 1 above.

Before deployment of archives, the weblogic server based details has to be changed appropriate to the user's environment.

Use the build properties described in `<table>` to customize the Ant targets with configuration values of your own.

Table 3–5 Build Properties for Customizing Ant Builds

Build Property	Description
<code>customEss.hostapp.earprofile</code>	
<code>customEss.hostapp.jarfile</code>	
<code>customEss.hostapp.jarprofile</code>	
<code>customEss.hostapp.jprproject</code>	
<code>customEss.hostapp.jwsfile</code>	
<code>customEss.hostapp.mds.jdbc</code>	

Table 3–5 (Cont.) Build Properties for Customizing Ant Builds

Build Property	Description
customEss.hostapp.mds.partition	
customEss.hostapp.name	The name to be used for the generated hosting application.
customEss.hostapp.workspace	
customEss.project.dir	The directory location for the generated JDeveloper project.
customEss.shared.library.name	The name to be given to the generated shared library.
ess.script.base.dir	
fmw.home	
jdev.home	
oracle.common	
weblogic.admin.password	The WebLogic Server admin password.
weblogic.admin.user	The WebLogic Server admin username.
weblogic.server.host	
weblogic.server.port	
weblogic.server.ssl.port	
weblogic.t3.url	

Example 3–4 Contents of an `ess-build.properties` File for Configuring Ant Builds

```
# ESS build properties
ess.script.base.dir=${user_home}

fmw.home=${fmw_home}
jdev.home=${fmw.home}/jdeveloper
oracle.common=${fmw.home}/oracle_common

# ===== ESS JDev project details =====
customEss.project.dir=${ess.script.base.dir}

customEss.hostapp.workspace=${hosting_application_name}
customEss.hostapp.jwsfile=${hosting_application_name}
customEss.hostapp.earprofile=${hosting_application_name}
customEss.hostapp.jprproject=EssSharedLibrary
customEss.hostapp.jarprofile=EssSharedLibrary
customEss.hostapp.jarfile=${jobdef_library_name}

customEss.shared.library.name=${jobdef_library_name}

customEss.hostapp.mds.partition=${hosting_application_name}
customEss.hostapp.mds.jdbc=mds-ESS_MDS_DS
customEss.hostapp.name=${hosting_application_name}

# ===== Weblogic Server details =====
MW_HOME=${fmw.home}
ORACLE_HOME=${jdev.home}
MW_ORA_HOME=${jdev.home}
COMMON_COMPONENTS_HOME=${oracle.common}
WEBLOGIC_HOME=${fmw.home}/wlserver_10.3
weblogic.server.host=adc2170657.us.oracle.com
WEBLOGIC_HOME=${fmw.home}/wlserver_10.3
```

```
weblogic.server.host=adc2170657.us.oracle.com
weblogic.server.port=7001
weblogic.server.ssl.port=7002
weblogic.admin.user=weblogic
weblogic.admin.password=welcome1
weblogic.t3.url=t3://${weblogic.server.host}:${weblogic.server.port}
```

Using Oracle JDeveloper to Generate an Oracle Enterprise Scheduler Application

This chapter is a tutorial that 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.

The chapter 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:

- [Section 4.1, "How to Start JDeveloper to Support Building Oracle Enterprise Scheduler Applications"](#)
- [Section 4.2, "Building a Combined Oracle Enterprise Scheduler Application"](#)
- [Section 4.3, "Building Split Submitting and Hosting Applications"](#)

4.1 How to Start JDeveloper to Support Building Oracle Enterprise Scheduler Applications

Some aspects of developing Oracle Enterprise Scheduler applications with Oracle JDeveloper require that you set the Middleware Home environment variable to the installation location of Oracle JDeveloper itself. Before you begin using Oracle JDeveloper to develop Oracle Enterprise Scheduler applications, be sure to set this variable.

To set an environment for building Oracle Enterprise Scheduler applications:

1. Open a command prompt.
2. Change directory to the installed location of Oracle JDeveloper. For example, on Windows you might do the following:

```
>cd c:\Oracle\Middleware\jdeveloper
```

3. Set MW_HOME to the location of Oracle JDeveloper. For example:

```
>set MW_HOME=c:\Oracle\Middleware
```

4. Start Oracle JDeveloper.

```
>jdeveloper
```

4.2 Building a Combined Oracle Enterprise Scheduler Application

The EssDemoApp sample application you build in this tutorial includes a complete application that you build with Oracle JDeveloper using Oracle Enterprise Scheduler APIs.

In this example, you'll create a hosting application and a simple Java job implementation. Though the example here is simple, its job class implements the `Executable` interface from which a more complex Java job might call out to other code as part of its work.

To create an application that schedules job requests you do the following:

- Create the Java class that specifies the logic 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. The application consists of two projects: one for hosting jobs and another for submitting them.
- 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 installed 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.

You will also need a WebLogic Server instance to which Oracle Enterprise Scheduler is deployed in standalone mode. You should have access to a database with the Oracle Enterprise Scheduler schema installed.

This section includes the following subsections:

- [Section 4.2.1, "Creating the Application and Projects for EssDemoApp Application"](#)
- [Section 4.2.2, "Creating Metadata and an Implementation Class for the EssDemoApp Application"](#)
- [Section 4.2.3, "Adding Application Code to Submit Job Requests"](#)
- [Section 4.2.4, "Setting Oracle Enterprise Scheduler Properties"](#)
- [Section 4.2.5, "Assembling the EssDemoApp Application"](#)
- [Section 4.2.6, "Deploying and Running the EssDemoApp Application"](#)

4.2.1 Creating the Application and Projects for EssDemoApp Application

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

- Create an application in Oracle JDeveloper.
- Create projects in Oracle JDeveloper. You will create two projects -- one in which to develop "Hello World"-style Java job and another in which to develop a client that submits requests with the job.

4.2.1.1 How to Create the EssDemoApp Application and Host Project

To work with Oracle Enterprise Scheduler, you first create an application in Oracle JDeveloper. You'll also create a hosting application to support job execution.

To create the EssDemoApp application and hosting project:

1. Start Oracle JDeveloper as described in [Section 4.1](#).
2. In the Select Role dialog, select the **Default Role**, then click **OK**.
3. Click the Application menu, then click **New**.
4. 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 **Generic Application**.
 - e. Click **Next**.
5. In the Name your project window, enter the name for the host project you're creating and select supporting technologies.
 - a. In the **Project Name** field, enter a name for your hosting project. For this sample application, enter `EssHost`.
 - b. On the **Project Technologies** tab, under **Available**, double-click **ESS Host Support** and **ESS Job Support** so that they are both listed under **Selected** on the right side of the dialog box.
 - c. Click **Next**.
6. In the Configure Java settings window, in the Default Package field, enter `oracle.esshost`.
Click **Next**.
7. In the Configure EJB settings window, select the following:
 - Under EJB Version, select the **Enterprise JavaBeans 3.0** option button.
 - Under EJB Version 3.0, select the **Generate ejb-jar.xml in this project** check box.
 Click **Next**.

8. In the Configure ESS Host Support settings window, in the **Application Id** field, enter `EssDemoApp`.

Click **Finish**.

This displays the `EssDemoApp` Overview page. You can use sections of this page to get information about aspects of the application you're creating, as well as to manage its development progress. For now, though, you'll move on to creating project artifacts to support creating jobs.

4.2.1.2 How to Create the Client Project

In the preceding step, you created a project in which to develop the application to host your jobs. In this section, you'll use Oracle JDeveloper to create another project in the `EssDemoApp` application. This second project will provide support for client interaction with the hosting application.

To create the client project:

1. Click the File menu, then click **New**.
2. In the New Gallery, under Categories, expand General, then click **Projects**.
3. Under Items, click **Generic Project**, then click **OK**.
4. In the Name your project window, enter the name for the client project you're creating and select supporting technologies.
 - a. In the **Project Name** field, enter a name for your client project. For this sample application, enter `EssClient`.
 - b. On the **Project Technologies** tab, under **Available**, double-click the following items so that they are listed under **Selected** on the right side of the dialog box:

ESS Client Support

HTML

JSF

JSP and Servlets

XML

- c. Click **Next**.
5. In the Configure Java settings window, in the **Default Package** field, enter `oracle.essclient`.
Click **Next**.
 6. In the Configure EJB settings window, select the following:
 - Under EJB Version, select the **Enterprise JavaBeans 3.0** option button.Click **Next**.
 7. In the Configure ESS Client Support settings window, in the **Application Id** field, ensure the `EssDemoApp` is displayed there.
Click **Finish**.

4.2.2 Creating Metadata and an Implementation Class for the `EssDemoApp` Application

For a Java job, which is what you'll be adding here, an implementation class implements the logic of your job -- the code that does job's actual work. The class implements the `oracle.as.scheduler.Executable` interface. The interface's `execute` method provides a place where you can add the job's logic. Though the code in this

example is very simple, the execute method can also serve as a starting place for processing that continues into code to which the Java job has access.

As with other job types, including PL/SQL and process jobs, a Java job's work is guided by metadata. This metadata forms a job-specific context that can include Oracle Enterprise Scheduler-defined system properties, properties you create, and control of who has access to the metadata. For example, metadata might be a way for you to collect and pass instance data to downstream code.

To use the EssDemoApp sample application to submit a job request, you need to create:

- Metadata in the form of a job definition that is the basic unit of work that defines a job request in Oracle Enterprise Scheduler.
- A Java job implementation class.

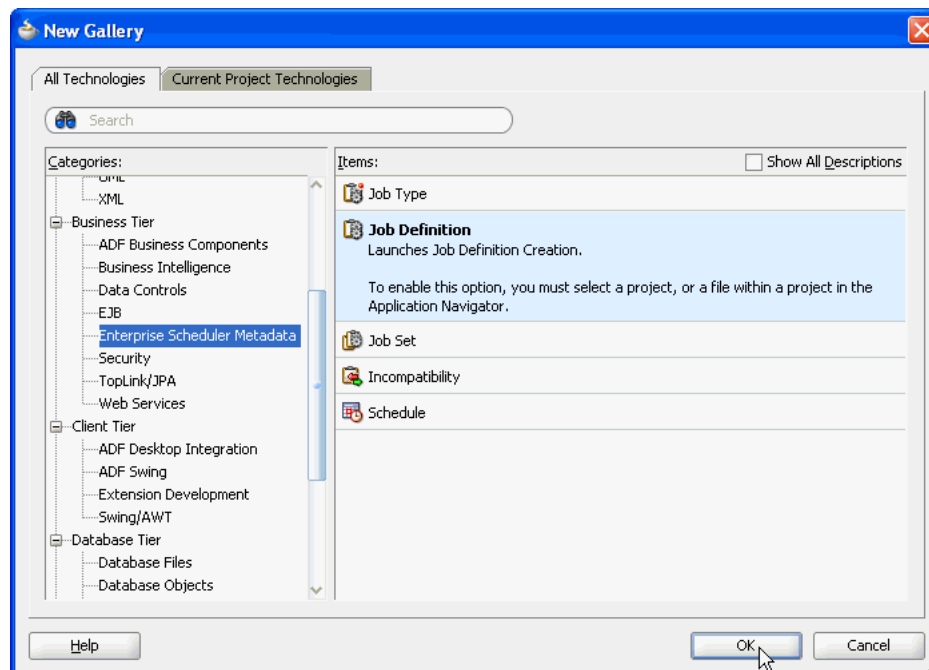
4.2.2.1 How to Create Metadata for the EssDemoApp Application

In this section, you use Oracle JDeveloper to create job definition metadata and a simple implementation class for a Java job.

To create metadata for the application:

1. In the Application Navigator, select the **EssHost** project.
2. Press **Ctrl-N**. This displays the New Gallery.
3. In the New Gallery, select the **All Technologies** tab.
4. In the **Categories** area expand **Business Tier** and select **Enterprise Scheduler Metadata**.
5. In the **Items** area, select **Job Definition** as shown in [Figure 4-1](#).

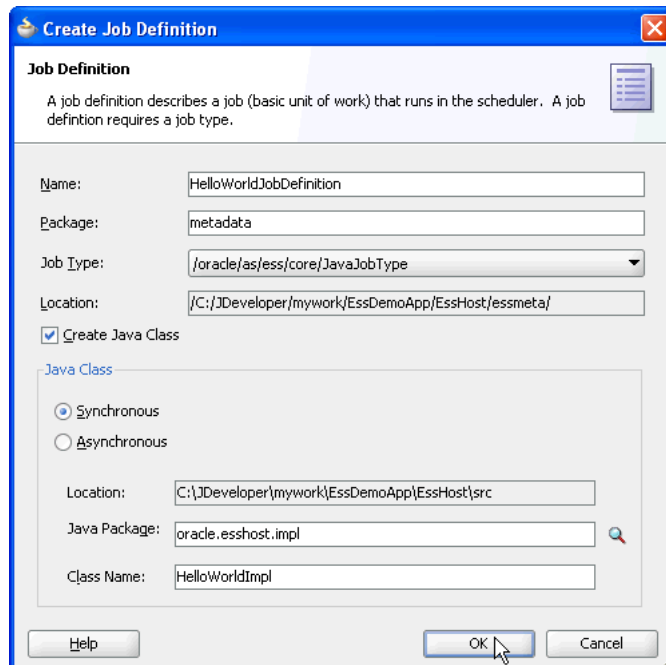
Figure 4-1 Adding Job Type Metadata to the Sample Application



6. Click **OK**. This displays the Create Job Definition dialog.

7. In the Create Job Definition dialog, specify the following:
 - a. In the **Name** field, enter a name for the job definition. For this example, enter the name: `HelloWorldJobDefinition`.
 - b. In the **Package** field, enter a package name. For this example, enter `/oracle/esshost/metadata`.
 Note that you should use slashes, rather than dots, to delimit names in metadata package names. A metadata package ending in ".metadata" will not be visible in Oracle JDeveloper.
 - c. In the **Job Type** field, from the dropdown list select `/oracle/as/ess/core/JavaJobType`.
 If job types are not listed in the dropdown, ensure that you started Oracle JDeveloper as described in [Section 4.1](#).
 - d. Ensure that the **Create Java Class** check box and the **Synchronous** option button are selected.
 By selecting the Create Java Class check box, you're asking that a Java class for your Java job be created, saving you the trouble of creating one later. Selecting the Synchronous option specifies that this will be a synchronous Java job.
 - e. Under **Java Class**, specify details for the Java class you're creating. In the **Java Package** field, enter its package name -- here, enter `oracle.esshost.impl`. In the **Class Name** field, enter a name for the class -- here, enter `HelloWorldImpl` as shown in [Figure 4-2](#)

Figure 4-2 Creating a Job Definition with the Job Definition Creation Wizard



- f. Click **OK**.
 This creates the Java class you requested, along with the `HelloWorldJobDefinition.xml` file. Oracle JDeveloper displays XML file's contents in the Job Definition page.

On the Job Definition page, you can edit job definition metadata, including properties that specify parameters for the job, access to this metadata, and a resource bundle to use for localization.

8. In the Job Definition page, in the **Description** field enter a description for the job type. For this example enter: Sample Java Job Definition.

Leave the rest of the metadata unchanged.

9. In the Application Navigator, locate the class you created by expanding the items in the projects panel to EssHost > Application Sources > oracle.esshost.impl > HelloWorldImpl.java.
10. Open HelloWorldImpl.java in the source editor.
11. In the source editor, add simple code to implement the execute method. The execute method is where execution for a Java job begins. Your HelloWorldImpl class should look something like [Example 4–1](#).

Example 4–1 HelloWorldImpl with Execute Method Implemented

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

    public void cancel()
    {
    }
}
```

12. Save and close HelloWorldImpl.java.

4.2.3 Adding Application Code to Submit Job Requests

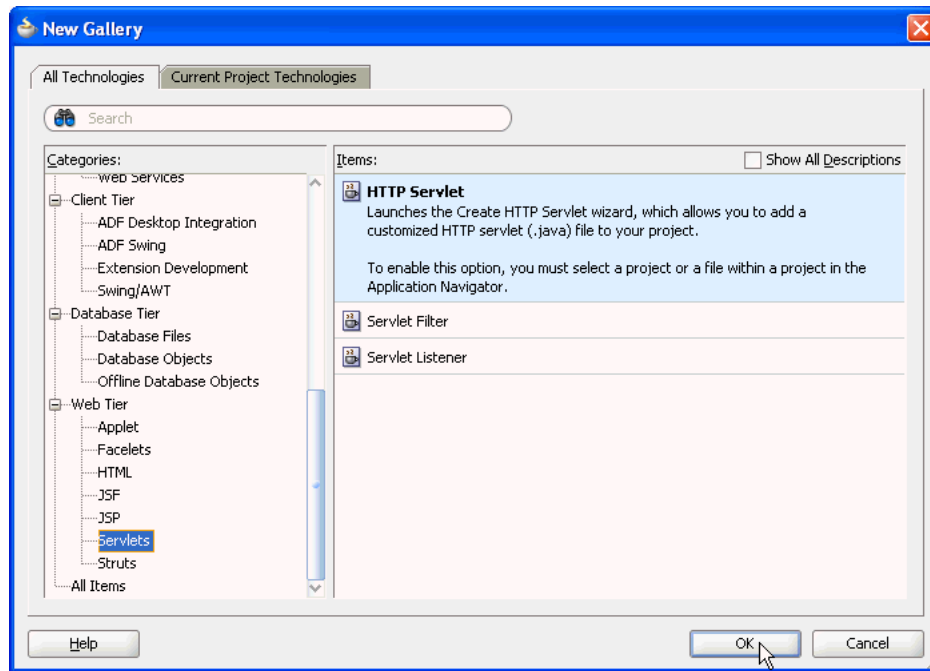
In an Oracle Enterprise Scheduler application you use the Oracle Enterprise Scheduler APIs to submit job requests from any component in the application. The EssDemoApp sample application provides a Java servlet for a servlet-based user interface for submitting job requests (using Oracle Enterprise Scheduler).

4.2.3.1 How to Add Application Code to Submit Job Requests

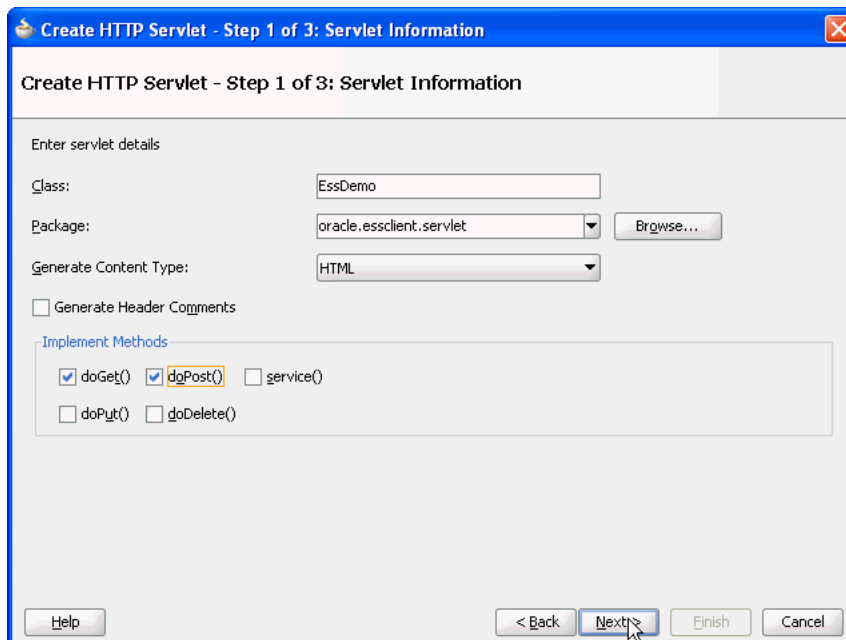
In this section, you'll create a servlet for receiving job submission requests.

To add a servlet to support job request submissions:

1. In the Application Navigator, select the **EssClient** project.
2. Press **Ctrl-N**. This displays the New Gallery.
3. In the New Gallery, in the **Categories** area, expand **Web Tier** and select **Servlets**.
4. In the **Items** area, select **HTTP Servlet** as shown in [Figure 4–3](#).

Figure 4–3 Adding Job Type Metadata to the Sample Application

5. Click **OK**. This displays the Create HTTP Servlet wizard.
6. In the Welcome page, click **Next**.
7. In the Create HTTP Servlet - Step 1 of 3: Servlet Information page, specify the following:
 - a. In the **Class** field, enter a name for the servlet class. For this example, enter the name: `EssDemo`.
 - b. In the **Package** field, enter a package name. For this example, enter `oracle.essclient.servlet`.
 - c. In the **Generate Content Type** field, from the dropdown list ensure the **HTML** is selected.
 - d. In the **Implement Methods** area, select the **doGet()** and **doPost()** check boxes, as shown in [Figure 4–4](#).

Figure 4–4 Creating a Servlet -- Step 1 of 3

- e. Click **Next**.
8. In the Create HTTP Servlet - Step 2 of 3: Mapping Information page, specify the following:
 - a. In the **Name** field, enter a name for the servlet. For this example, enter the name: EssDemo.
 - b. In the **URL Pattern** field, enter a URL for servlet mapping. For this example, enter /essdemo/*.
 - c. Click **Finish**.

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.

4.2.4 Setting Oracle Enterprise Scheduler Properties

With Oracle Enterprise Scheduler properties, you set values for settings used in the ejb-jar.xml file associated with the application. These properties include the following:

- **Logical Application Name**
Specifies the logical name used to identify this application. Separate from the application name used when deploying the application to the container, this value lets you safely hard code the logical application name in source code.
- **Application Policy Stripe**
Specifies which JPS security stripe (or "security context") should be used to perform security checks.
- **JPS Interceptor Application Name**

Specifies the application stripe name used at run time to determine which set of security policies are applicable.

4.2.4.1 How to Set Oracle Enterprise Scheduler Properties for the Application

In this section, you'll set default values for Oracle Enterprise Scheduler properties.

To set values for Oracle Enterprise Scheduler properties:

1. In the Application Navigator, right-click the EssHost project, then click **Enterprise Scheduler Properties**.
2. In the Enterprise Scheduler Properties dialog, enter EssDemoApp for all three of the fields provided: **Logical Application Name**, **Application Policy Stripe**, and **JPS Interceptor Application Name**.
3. Click **OK**.

4.2.5 Assembling the EssDemoApp Application

After you create the 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.

4.2.5.1 How to Create the EJB-JAR Deployment Profile for the EssDemoApp

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.xml`, 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.xml` must contain descriptions for the Oracle Enterprise Scheduler EJBs and should not be modified.

To create the EJB-JAR deployment profile:

1. In the Application Navigator, in the Projects panel, right-click the **EssHost** project, then click **Project Properties**.
2. In the Project Properties window, in the navigator, click **Deployment**.
3. Under Deployment Profiles, delete all profiles listed in the window, then click **New**.
4. In the Create Deployment Profile dialog, from the **Archive Type** dropdown, select **EJB JAR file**.
5. In the **Name** field, enter a name for the EJB. For this example, enter `ess-ejb`.
6. Click **OK**.

7. In the Edit EJB JAR Deployment Profile Properties dialog, in the navigator on the left, click **General**.
8. In the General window, in the **Enterprise Application Name** field, enter `EssDemoApp`.
9. In the navigator, expand to File Groups > Project Output > Contributors.
10. In the Contributors window, select the following check boxes:
 - Project Output Directory
 - Project Source Path
 - Project Additional Classpath
 - Project Dependencies
11. In the navigator, expand to File Groups > Project Output > Filters.
12. In the Filters window, on the Files tab, ensure that the following folders are selected:
 - META-INF (and its contents)
 - oracle (and its contents)
13. Click **OK**.
14. In the Project Properties dialog, click **OK**.

4.2.5.2 How To Update the WAR Archive Options

In this section, you specify information that Oracle JDeveloper can use to generate a WAR file.

To update the WAR archive options:

1. In the Application Navigator, in the Projects panel, right-click the **EssClient** project, then click **Project Properties**.
2. In the Project Properties window, in the navigator, click **Deployment**.
3. In the Deployment window, click **New**.
4. In the Create Deployment Profile dialog, from the **Archive** Type dropdown, select **WAR file**.
5. In the **Name** field enter `WAR_EssDemoApp`.
6. Click **OK**.
7. In the Edit WAR Deployment Profile Properties dialog, in the navigator on the left, click **General**.
8. In the General window, select the **Specify Java EE Web Context Root** option. In the field beneath the option, enter `EssDemo`.
9. In the navigator, expand to File Groups > Web Files > Contributors.
10. In the Contributors window, select the following check boxes:
 - Project Output Directory
 - Project HTML Root Directory
 - Project Source Path
11. In the navigator, expand to File Groups > Web Files > Filters.

12. In the Filters window, on the Files tab, ensure that the following folders are selected:

- oracle (and its contents)
- WEB-INF (and its contents)

Click **OK**.

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

4.2.5.3 How to Update the EAR Options

In this section, you'll prepare an EAR file that assembles the EssDemoApp 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 update the EAR options:

1. Click the **Application** menu, then click **Application Properties**.
2. In the Application Properties dialog, in the navigation pane, click **Deployment**.
3. In the Deployment window, click **New**.
4. In the Create Deployment Profile dialog, in the **Name** field, enter EAR_EssDemoApp as the deployment profile's name.
Click **OK**.
5. In the Edit EAR Deployment Profile Properties dialog, in the navigation pane on the left, click **Application Assembly**.
6. In the Application Assembly window, under Java EE Modules, ensure that all item check boxes are selected.
7. Click **OK**.
8. In the Application Properties dialog, click **OK**.

4.2.6 Deploying and Running the EssDemoApp Application

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

4.2.6.1 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 > EAR_ EssDemoApp > to > IntegratedWLSConnection**.
4. Oracle JDeveloper shows the Deployment Configuration page. Select the appropriate options for your Metadata Repository.
5. Click **Deploy**.
6. Verify the deployment using the Deployment Log.

4.2.6.2 How to Run the EssDemoApp Sample Application

To run the EssDemoApp 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 4-5](#).

Figure 4-5 Running EssDemo Servlet for Oracle Enterprise Scheduler Sample Application

Enterprise Scheduler Service Tutorial

Launch Job

Job:
 Schedule:

Messages

Request Status

reqID	Description	Scheduled time	State	Action
1	Job_essdemo1@Immediately	Wed Jan 07 14:05:05 PST 2009	SUCCEEDED	<input type="button" value="Purge"/>

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 4-6](#).

Figure 4–6 Running EssDemo Servlet with Request Status for Submitted Requests

Enterprise Scheduler Service Tutorial

Launch Job

Job:

Schedule:

Messages

New request 2 launched using Job_essdemo1@Immediately

Request Status

reqID	Description	Scheduled time	State	Action
1	Job_essdemo1@Immediately	Wed Jan 07 14:05:05 PST 2009	SUCCEEDED	<input type="button" value="Purge"/>
2	Job_essdemo1@Immediately	Fri Jan 09 14:31:47 PST 2009	WAIT	<input type="button" value="Cancel"/>

4.2.6.3 How to Purge Jobs in the EssDemoApp Sample Application

Using the EssDemoApp 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`.

4.3 Building Split Submitting and Hosting 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.

Note: This section creates a new application. If you have created EssDemoApp with the sections beginning with [Section 4.2.1](#), note that this section creates a project of the same name. You'll need to choose a different location for the application or delete the previous application in order to use the EssDemoApp application name in this section.

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

1. Build a back-end hosting application that includes the code to be scheduled and run.
2. Build a front-end submitter application initiates the job requests.

This section includes the following subsections:

- [Section 4.3.1, "How to Create the Back-End Hosting Application for EssDemoApp"](#)
- [Section 4.3.2, "How to Create the Front-End Submitter Application for Oracle Enterprise Scheduler"](#)

4.3.1 How to Create the Back-End Hosting Application for EssDemoApp

Using Oracle JDeveloper you create the back-end application. To create the EssDemoApp back-end sample application you do the following:

- Create a back-end 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.

4.3.1.1 Creating the Back-End Hosting Application

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

To create the back-end 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. Click **Next**.
5. In the Name your project window, enter the name for the host project you're creating and select supporting technologies. This project is where you will create and save the Oracle Enterprise Scheduler metadata
 - a. In the **Project Name** field, enter a name for your hosting project. For this sample application, enter `SuperEss`.
 - b. On the **Project Technologies** tab, under **Available**, double-click **ESS Host Support** and **ESS Job Support** so that both are listed under **Selected** on the right side of the dialog box.

Click **Next**.

6. In the Configure Java Settings page, change the default package to `oracle.apss.ess.howto`, then click **Next**.
7. In the Configure EJB Settings page, select **Generate ejb-jar.xml in this project** and click **Next**.
8. In the Configure ESS Host Support settings page, in the **Application Id** field, enter `EssDemoApp`.

9. Click **Finish**.

4.3.1.2 Configuring Security for the Back-End Hosting Application

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

To configure security for the back-end 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**, then click **Next**.
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 `jazn-data.xml` is generated.

6. In the overview editor for the `jazn-data.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**.

4.3.1.3 Defining Metadata for the Back-End Hosting Application

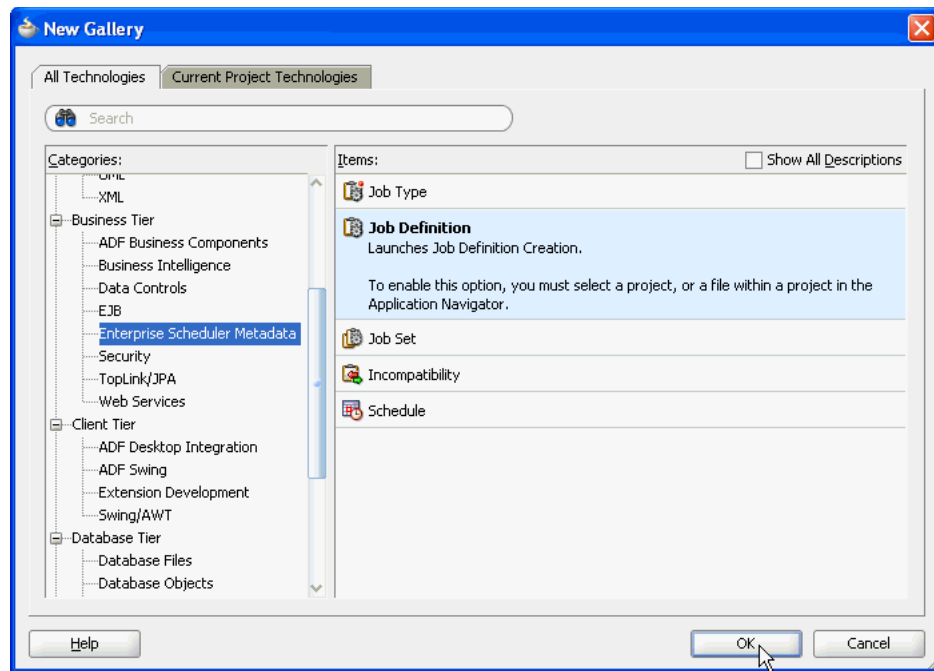
To use the Oracle Enterprise Scheduler split application to submit a job request you need to create metadata that defines a job request, including the following:

- 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.

To create metadata for the back-end hosting application:

1. In the Application Navigator, select the **SuperEss** project.
2. Press **Ctrl-N**. This displays the New Gallery.
3. In the New Gallery, select the **All Technologies** tab.
4. In the **Categories** area expand **Business Tier** and select **Enterprise Scheduler Metadata**.
5. In the **Items** area, select **Job Definition** as shown in [Figure 4-7](#).

Figure 4–7 Adding Job Type Metadata to the Sample Application



6. Click **OK**. This displays the Create Job Definition dialog.
7. In the Create Job Definition dialog, specify the following:
 - a. In the **Name** field, enter a name for the job definition. For this example, enter the name: `HelloWorldJobDef`.
 - b. In the **Package** field, enter a package name. For this example, enter `oracle/apps/ess/howto/metadata`.
 - c. In the **Job Type** field, from the dropdown list select **/oracle/as/ess/core/JavaJobType**.

If job types are not listed in the dropdown, ensure that you started Oracle JDeveloper as described in [Section 4.1](#).

- d. Ensure that the **Create Java Class** check box and the **Synchronous** option button are selected.

By selecting the Create Java Class check box, you're asking that a Java class for your Java job be created, saving you the trouble of creating one later. Selecting the Synchronous option specifies that this will be a synchronous Java job.

- e. Under **Java Class**, specify details for the Java class you're creating. In the **Java Package** field, enter its package name -- here, enter `oracle.apps.ess.howto`. In the **Class Name** field, enter a name for the class -- here, enter `HelloWorldJob`.
- f. Click **OK**.

This creates the Java class you requested, along with the `HelloWorldJobDefinition.xml` file. Oracle JDeveloper displays XML file's contents in the Job Definition page.

On the Job Definition page, you can edit job definition metadata, including properties that specify parameters for the job, access to this metadata, and a resource bundle to use for localization.

8. In the HelloWorldJobDef.xml Job Definition page, in the **Description** field, enter HelloWorld Example.
9. In the System Properties section, click the **Add** button.
10. In the Add System Property dialog, from the **Name** dropdown, select **SYS_effectiveApplication**.
11. In the **Initial Value** field, enter EssDemoApp.
12. Click **OK**.
13. In the Access Control section, click the **Add** button.
14. In the Add Access Control dialog, from the **Role** dropdown, ensure that **EssDemoAppRole** is selected. This is the role that you created during [Section 4.3.1.2](#).
15. Select the following actions: **Read** and **Execute**.
16. Click **OK**.

4.3.1.4 Creating a Java Implementation Class in the Back-End 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.

To implement the execute method:

1. In the Application Navigator, locate the class you created by expanding the items in the projects panel to SuperEss > Application Sources > oracle.apps.ess > howto > HelloWorldJob.java.
2. Open HelloWorldJob.java in the source editor.
3. In the source editor, add the following code to implement the `execute` method. The `execute` method is where execution for a Java job begins. The code inside your method should look something like [Example 4-2](#).

Example 4-2 HelloWorldJob Execute Method Code

```
StringBuilder sb = new StringBuilder(1000);
sb.append("\n=====");
sb.append("\n= EssDemoApp request is now running");
long myRequestId = ctx.getRequestId();
sb.append("\n= Request Id = " + myRequestId);
sb.append("\n= Request Properties:");
for (String paramKey : params.getNames()) {
    Object paramValue = params.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());
```

4.3.1.5 Setting Oracle Enterprise Scheduler Properties

With Oracle Enterprise Scheduler properties, you set values for settings used in the `ejb-jar.xml` file associated with the application. These properties include the following:

- **Logical Application Name**

Specifies the logical name used to identify this application. Separate from the application name used when deploying the application to the container, this value lets you safely hardcode the logical application name in source code.

- **Application Policy Stripe**

Specifies which JPS security stripe (or "security context") should be used to perform security checks.

- **JPS Interceptor Application Name**

Specifies the application stripe name used at run time to determine which set of security policies are applicable.

To set values for Oracle Enterprise Scheduler properties:

1. In the Application Navigator, right-click the EssHost project, then click **Enterprise Scheduler Properties**.
2. In the Enterprise Scheduler Properties dialog, enter `EssDemoApp` for all three of the fields provided: **Logical Application Name**, **Application Policy Stripe**, and **JPS Interceptor Application Name**.
3. Click **OK**.

4.3.1.6 Assembling the Back-End Hosting Application for Oracle Enterprise Scheduler

After you create the back-end sample application you use Oracle JDeveloper to assemble the application.

To assemble the back-end application you do the following:

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

4.3.1.6.1 How to Assemble the EJB JAR File for the Back-End Hosting Application The EJB Java archive file includes descriptors for the Java job implementations.

To create the EJB-JAR deployment profile:

1. In the Application Navigator, in the Projects panel, right-click the **SuperEss** project, then click **Project Properties**.
2. In the Project Properties window, in the navigator, click **Deployment**.
3. Under Deployment Profiles, delete all profiles listed in the window, then click **New**.
4. In the Create Deployment Profile dialog, from the **Archive Type** dropdown, select **EJB JAR file**.
5. In the **Name** field, enter a name for the EJB. For this example, enter `JAR_SuperEssEjbJar`.
6. Click **OK**.
7. In the Edit EJB JAR Deployment Profile Properties dialog, in the navigator, expand to File Groups > Project Output > Contributors.
8. In the Contributors window, select the following check boxes:
 - Project Output Directory
 - Project Source Path

- Project Additional Classpath
 - Project Dependencies
9. In the navigator, expand to File Groups > Project Output > Filters.
 10. In the Filters window, on the Files tab, ensure that the following folders are selected:
 - META-INF (and its contents)
 - oracle (and its contents)
 11. Click **OK**.
 12. In the Project Properties dialog, click **OK**.

4.3.1.6.2 How to Assemble the MAR and EAR Files for the Back-End Hosting Application In this section, you'll prepare an EAR file that assembles the EssDemoApp 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 update the EAR options:

1. Click the **Application** menu, then click **Application Properties**.
2. In the Application Properties dialog, in the navigation pane, click **Deployment**.
3. Select the default MAR file profile, then click **Edit**.
4. In the Edit MAR Deployment Profile Properties dialog, in the navigation pane, expand to Metadata File Groups > User Metadata > Directories and select **Directories**.
5. In the Directories window, select the oracle.apps.ess.howto check box, then click **OK**.
6. In the Application Properties dialog, on the Deployment window, click **New**.
7. In the Create Deployment Profile dialog, from the Archive Type dropdown, select **EAR File**.
8. In the **Name** field, enter EAR_EssDemoAppEar.
Click **OK**.
9. In the Edit EAR Deployment Profile Properties dialog, in the navigation pane, select **General**.
10. In the General window, in the **Application Name** field, enter EssDemoApp.
11. In the navigation pane, select **Application Assembly**.
12. In the Application Assembly window, ensure that all check boxes are selected.
13. Click **OK**.
14. In the Application Properties dialog, click **OK**.

4.3.1.7 Deploying the Back-End Hosting Application

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

To deploy the back-end 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.

4.3.2 How to Create the Front-End 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 front-end 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 front-end submitter sample application you do the following:

- Create a front-end 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.

4.3.2.1 Creating the Front-End Submitter Application

You use JDeveloper to build the front-end submitter application using similar steps as you used for the back-end hosting application.

To create the front-end submitter application:

1. Complete the steps in [Section 4.3.1.1, "Creating the Back-End Hosting Application"](#) but this time use `ESSDemoAppUI` as the name of the application. When you configure ESS host support settings, in the **Application Id** field, be sure to enter `EssDemoApp`.

4.3.2.2 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 project window, enter the name for the host project you're creating and select supporting technologies. This project is where you will create and save the Oracle Enterprise Scheduler metadata

- a. In the **Project Name** field, enter a name for your hosting project. For this sample application, enter `SuperWeb`.
- b. On the **Project Technologies** tab, under **Available**, double-click **ESS Client Support, JSP and Servlets**, and **ADF Library Web Application Support** so that both are listed under **Selected** on the right side of the dialog box.

Click **Next**.

4. In the Project Java Settings page, change the default package to `oracle.apss.ess.howto` and click **Finish**.

4.3.2.3 Configuring Security for the Front-End 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 front-end 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**.

4.3.2.4 Creating the HTTP Servlet for the Front-End 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 front-end 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.apss.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 4-3](#).

Example 4-3 HTTP Servlet Code for the Front-End Submitter Application

```
package oracle.apss.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 will 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 " +
MetadataObjectID",
                                                "when converting %s to
                                                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;
    }

    /**
     * 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>Oracle Enterprise Scheduler
Tutorial</h1></td></tr>");
out.println("  <tr><td align=\"center\"><table cellpadding=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 = "&nbsp;";
        }

        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;
}
}

```

4.3.2.5 Editing the web.xml File for the Front-End Submitter Application

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

To edit the web.xml file for the front-end 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 4-1](#).

Table 4-1 EJB Resources for the Front-End 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/*`.

4.3.2.6 Editing the weblogic-application.xml file for the Front-End Submitter Application

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

To edit the weblogic-application.xml file for the front-end 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 4-4](#).

Example 4-4 Contents to Copy to weblogic-application.xml for a Front-End 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>

```

4.3.2.7 Editing the adf-config file for the Front-End 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 front-end 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 4-5](#).

Example 4-5 Contents to Copy to adf-config.xml for a Front-End 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>

```

4.3.2.8 Assembling the Front-End Submitter Application for Oracle Enterprise Scheduler

After you create the front-end sample application you use Oracle JDeveloper to assemble the application.

To assemble the back-end application you do the following:

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

4.3.2.8.1 How to Assemble the EJB JAR File for the Front-End Submitter Application The EJB Java archive file includes descriptors for the Java job implementations.

To assemble the EJB JAR File for the front-end 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**.

4.3.2.8.2 How to Assemble the WAR File for the Front-End Submitter Application You need to create a web archive file for the web application.

To assemble the WAR file for the front-end 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**.

4.3.2.8.3 How to Assemble the MAR and EAR Files for the Front-End Hosting Application The sample application needs to contain the MAR profile and the EAR file that assembles the `EssDemoApp` back-end application.

To create the MAR and EAR files for the front-end 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**.

4.3.2.9 Deploying the Front-End Submitter Application

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

To deploy the front-end submitter application:

1. From the main menu, choose **Application > Deploy > EAR_EssDemoUIEar...**
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**.

4.3.2.10 Running the Split Application

To run the split application:

1. Enter the following URL in a browser:

`http://host:http-port:/ESSDemoAppUI/essdemoappservlet`

For example,

`http://myserver.us.oracle.com:7101/EssDemoAppUI/essdemoappservlet`

2. Log in as `EssDemoAppUser` with the password `welcome1`.
3. Follow the same steps as in the combined application.

Use Case Oracle Enterprise Scheduler Sample Application (Deprecated)

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.

Note: This chapter includes a tutorial that uses an older release of Oracle JDeveloper. This content is deprecated. For development with a current version of Oracle JDeveloper, see [Chapter 4, "Using Oracle JDeveloper to Generate an Oracle Enterprise Scheduler Application"](#).

The chapter 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:

- [Section 5.1, "Introduction to the Oracle Enterprise Scheduler Sample Application"](#)
- [Section 5.2, "Creating the Application and Projects for the Sample Application"](#)
- [Section 5.3, "Creating a Java Implementation Class for the Sample Application"](#)
- [Section 5.4, "Adding Application Code to Submit Oracle Enterprise Scheduler Job Requests"](#)
- [Section 5.5, "Creating Metadata for Oracle Enterprise Scheduler Sample Application"](#)
- [Section 5.6, "Assembling the Oracle Enterprise Scheduler Sample Application"](#)
- [Section 5.7, "Deploying and Running the Oracle Enterprise Scheduler Sample Application"](#)
- [Section 5.8, "Troubleshooting the Oracle Enterprise Scheduler Sample Application"](#)
- [Section 5.9, "Using Submitting and Hosting Split Applications"](#)

5.1 Introduction to the Oracle Enterprise Scheduler Sample Application

The Oracle Enterprise 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.

5.2 Creating the Application and Projects for the 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 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 sample application you create the EssDemo servlet in the EssDemoApp application.

Note: This chapter includes a tutorial that uses an older release of Oracle JDeveloper. This content is deprecated. For development with a current version of Oracle JDeveloper, see [Chapter 4, "Using Oracle JDeveloper to Generate an Oracle Enterprise Scheduler Application"](#).

5.2.1 How to Create the EssDemoApp Application

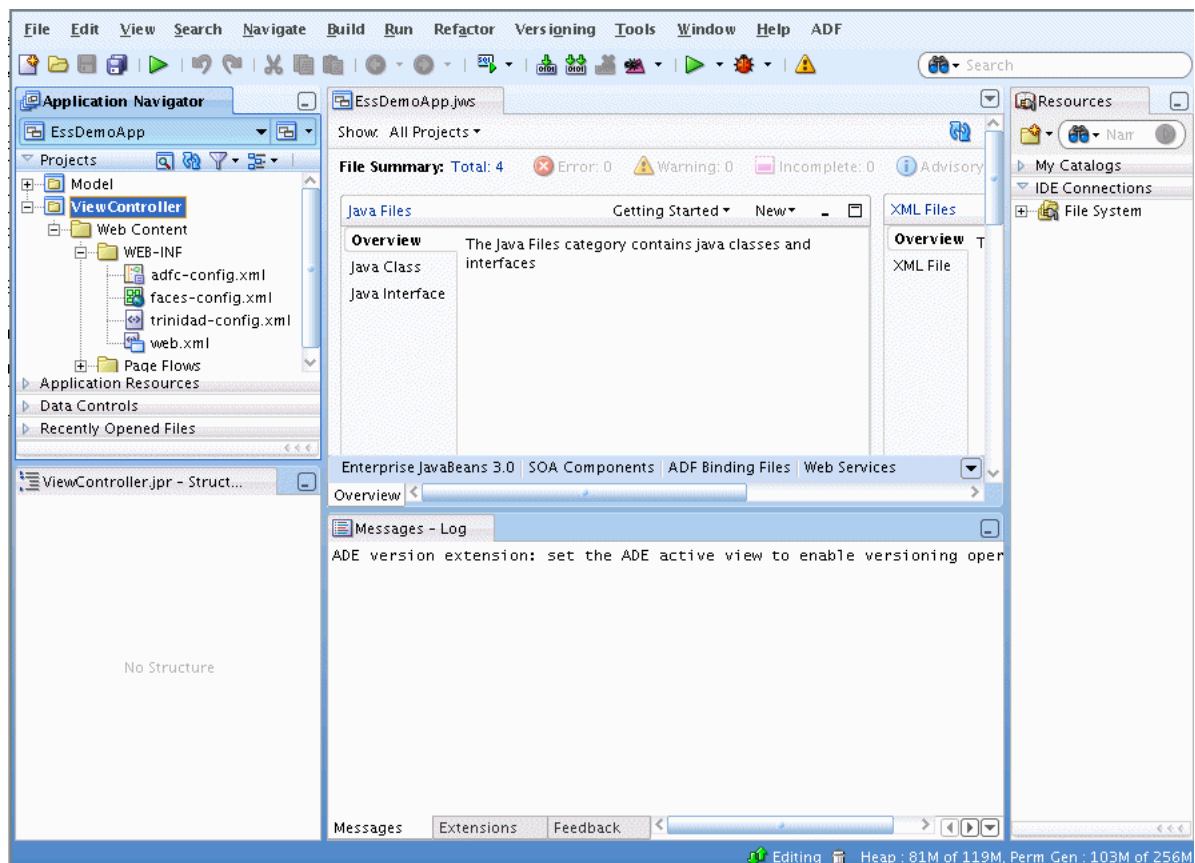
To work with Oracle Enterprise Scheduler, you first create an application and a project in Oracle JDeveloper.

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 5-1](#).

Figure 5-1 Sample Application File Summary Page



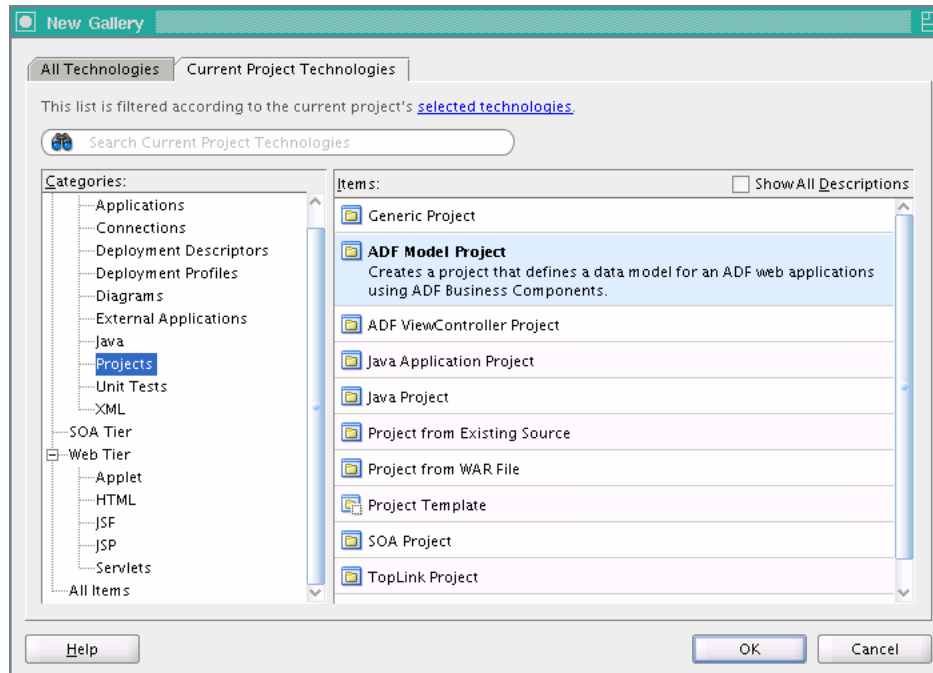
5.2.2 How to Create a Project in the Sample Application

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.

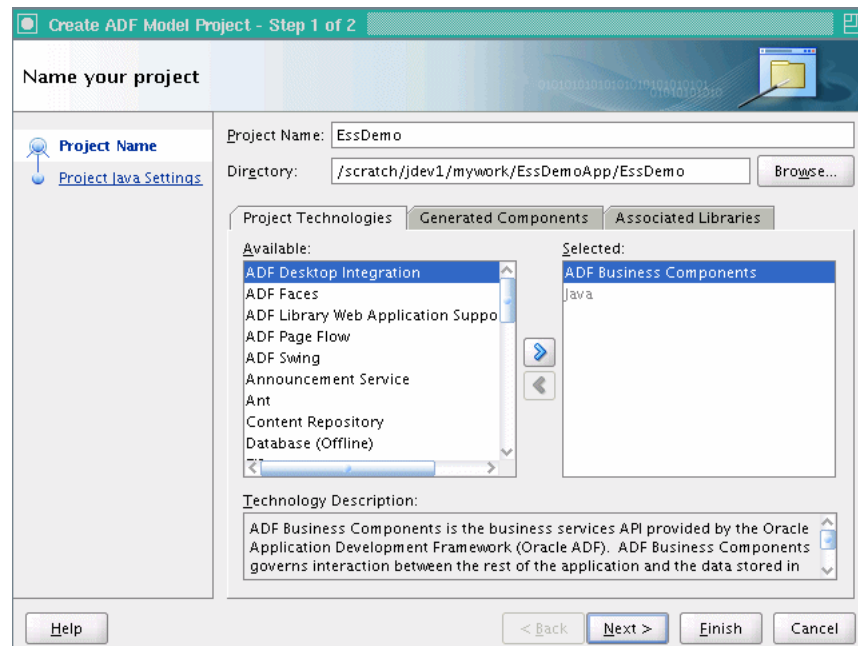
To create a project:

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 5-2](#).

Figure 5-2 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 5-3](#).

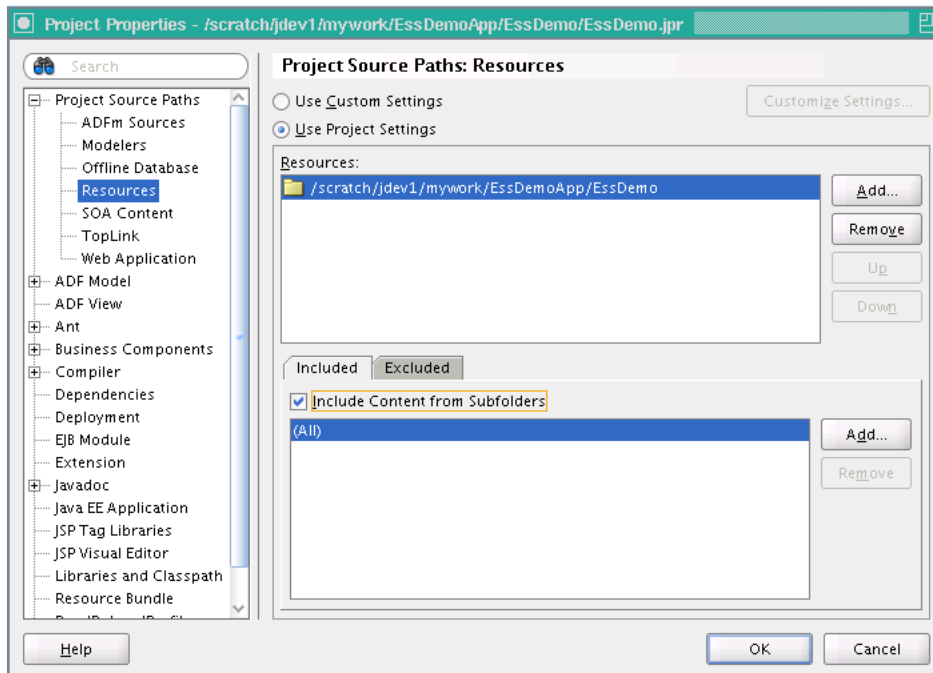
Figure 5–3 Adding the EssDemo Project to the Sample Application

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** check box, as shown in [Figure 5–4](#).
5. Click **OK**.

Figure 5–4 Updating Project Resources for Sample Project

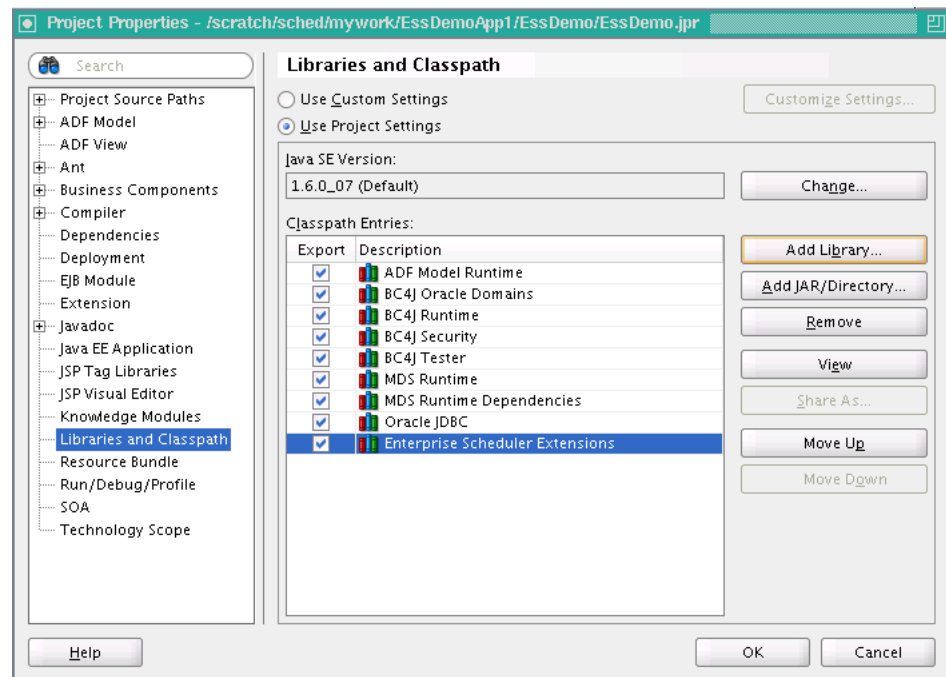


5.2.3 How to Set Project Properties for Oracle Enterprise Scheduler

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 5–5](#).

Figure 5–5 Adding Oracle Enterprise Scheduler Extensions to Project

7. Click OK to dismiss the Project Properties dialog.

5.3 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.

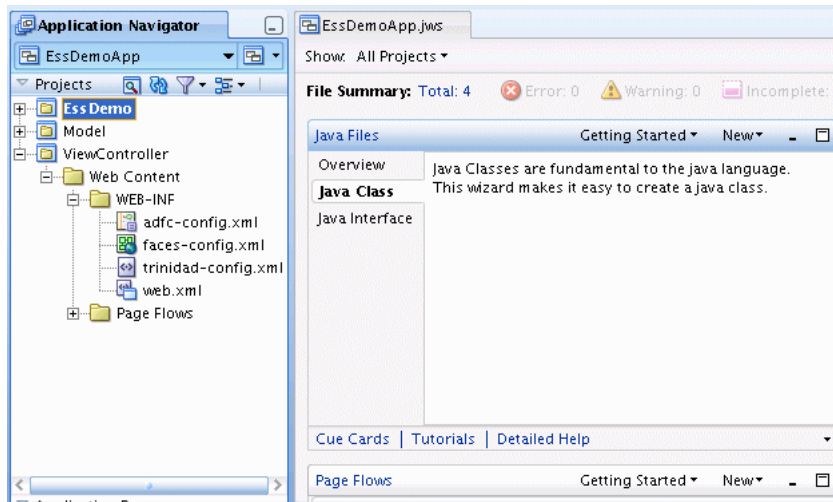
5.3.1 How to Create a Java Class Using the Executable Interface

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

To create a Java class that implements the executable interface:

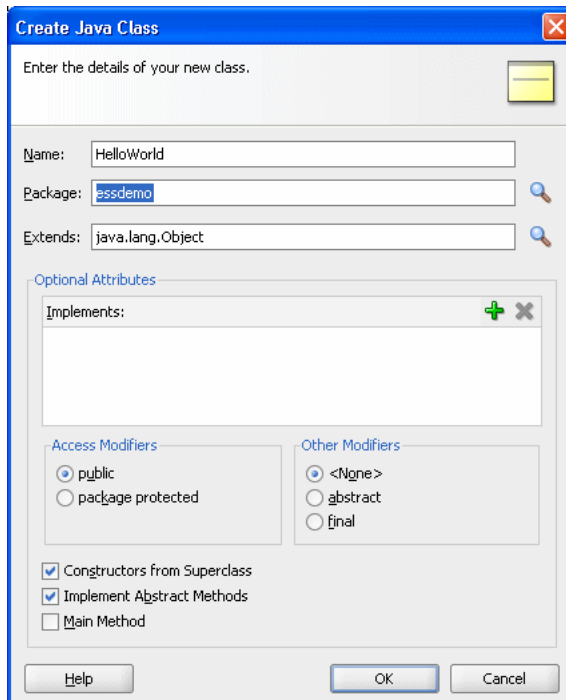
1. In the Application Navigator, select the `EssDemo` project.
2. In the Overview area, select the **Java Class** navigation tab as shown in [Figure 5–6](#).

Figure 5–6 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 5–7](#).

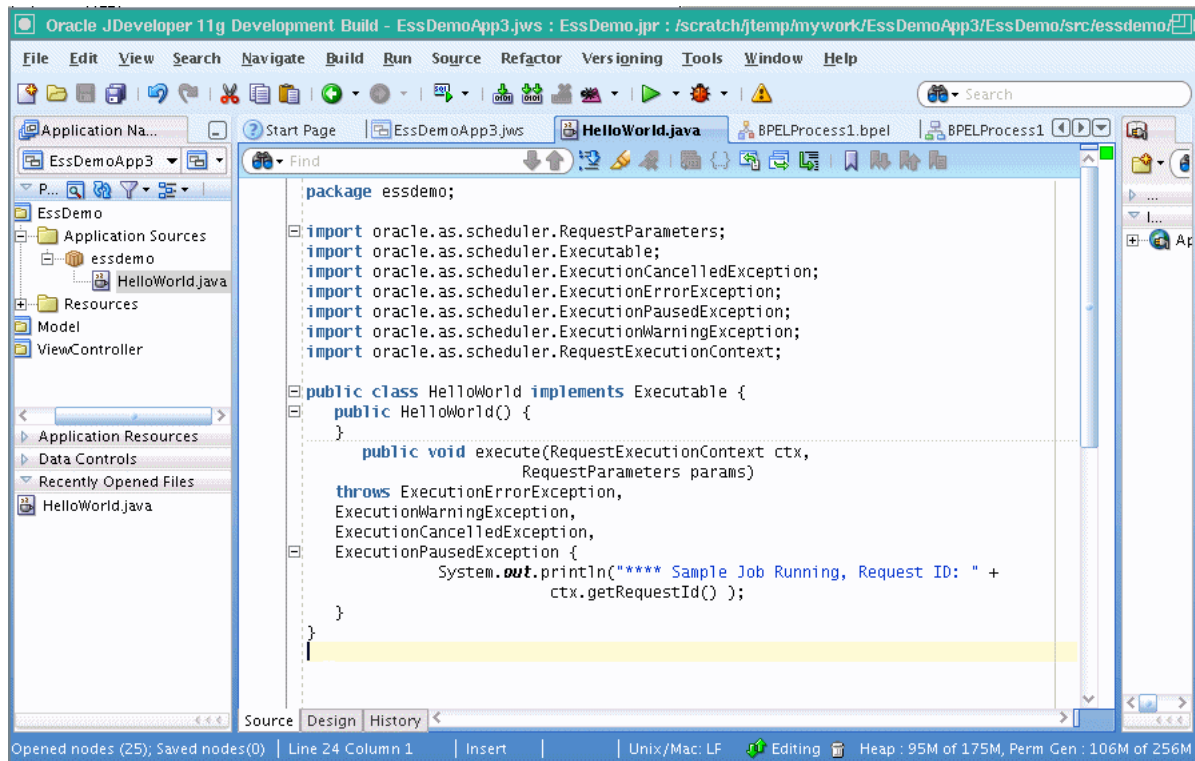
Figure 5–7 Adding a Java Implementation Class to the Sample Application



9. Click **OK**.

10. Replace the generated contents of the HelloWorld.java file with the contents of the HelloWorld.java supplied with the sample, as shown in [Example 5-1](#). This code is also shown in [Figure 5-8](#).

Figure 5-8 Java Class That Implements Executable for Sample Application



[Example 5-1](#) shows `HelloWorld()`, the Java class that implements the interface `oracle.as.scheduler.Executable`.

Example 5-1 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());
    }
}

```

```
    }  
}
```

5.3.2 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.

5.3.3 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 5-1](#), 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.

5.4 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).

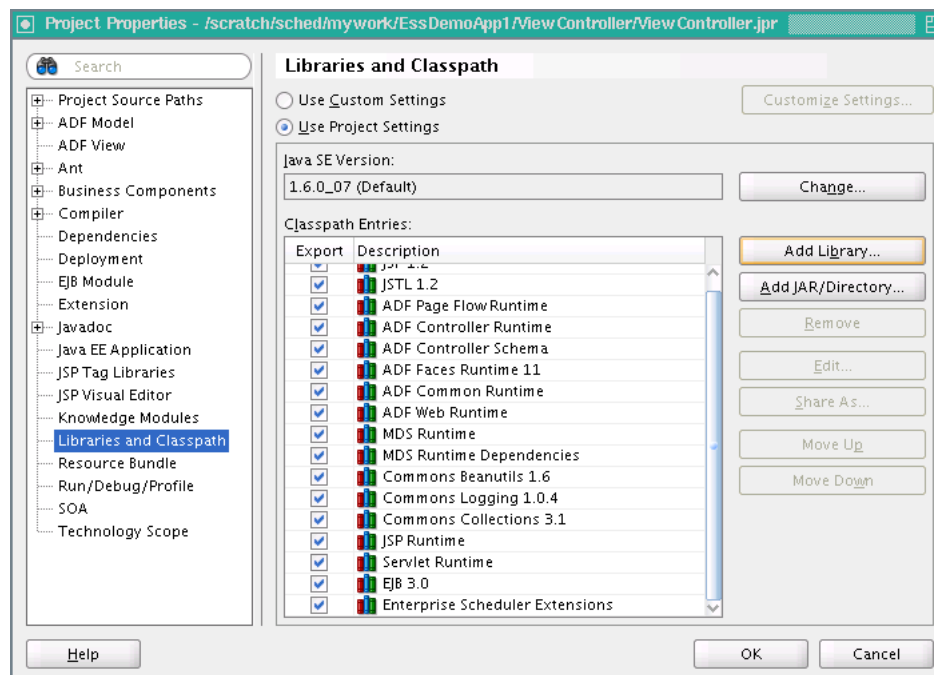
5.4.1 How to Add Required Libraries to Project

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 Oracle 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 5–9](#).

Figure 5–9 Adding Oracle Enterprise Scheduler Extensions to ViewController Project



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

5.4.2 How to Create the EssDemo Servlet

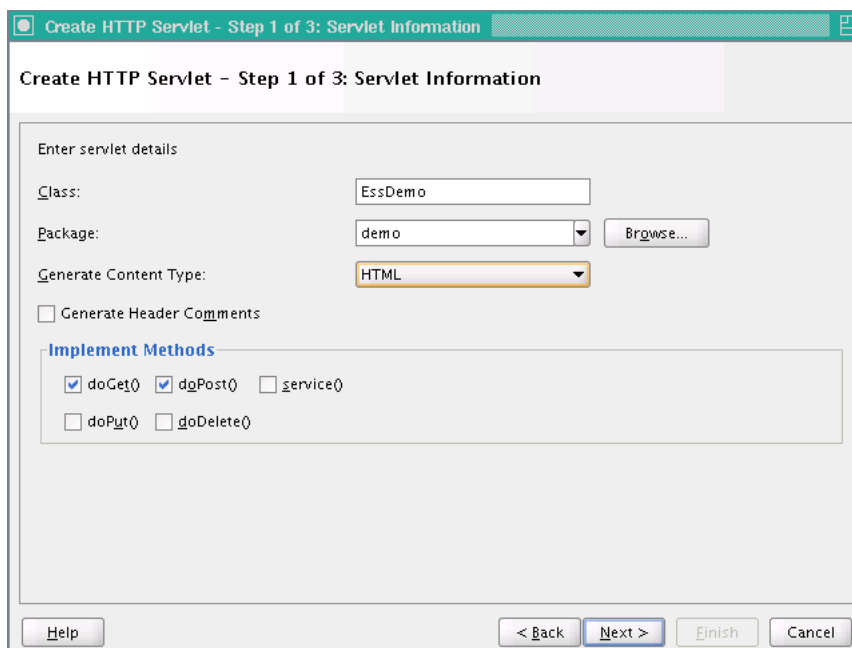
Using MVC design pattern you create the EssDemo servlet in the ViewController project.

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()** check boxes, as shown in [Figure 5-10](#).

Figure 5-10 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 5-11](#).

Figure 5–11 Using the Create HTTP Servlet Wizard: Step 2 of 3 Dialog

Create HTTP Servlet - Step 2 of 3: Mapping Information

Create HTTP Servlet - Step 2 of 3: Mapping Information

Enter servlet mapping.
While this is not required to create a servlet, it is required to run a servlet.

Specify a name and mapping for the servlet.

Mapping Details

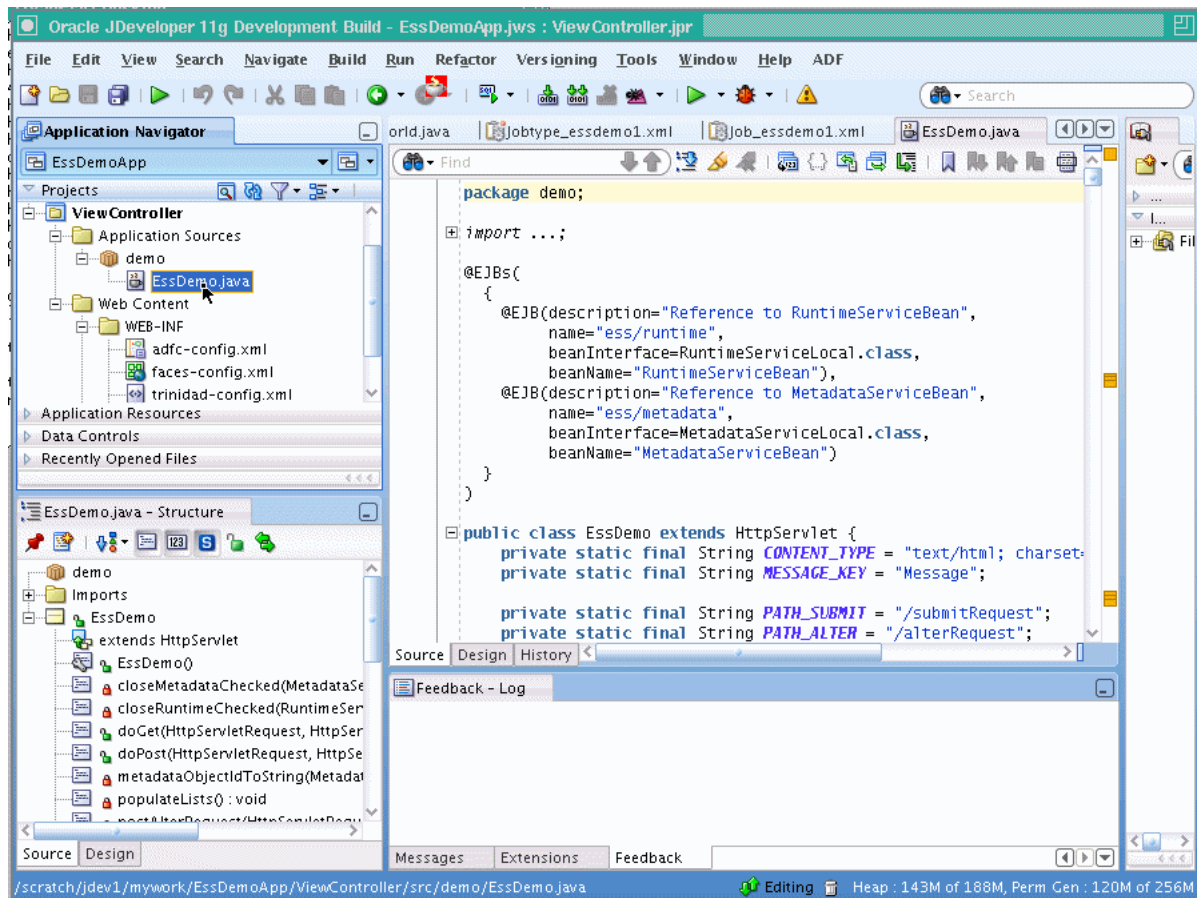
Name:

URL_Pattern:

Help < Back Next > Finish Cancel

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 5–12](#). The `EssDemo.java` sample code includes several hundred lines, so it is not included in this text in an example.

Figure 5–12 Adding the Sample Servlet to the ViewController Project

5.5 Creating Metadata for Oracle Enterprise 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.

5.5.1 How to Create a Job Type for Java

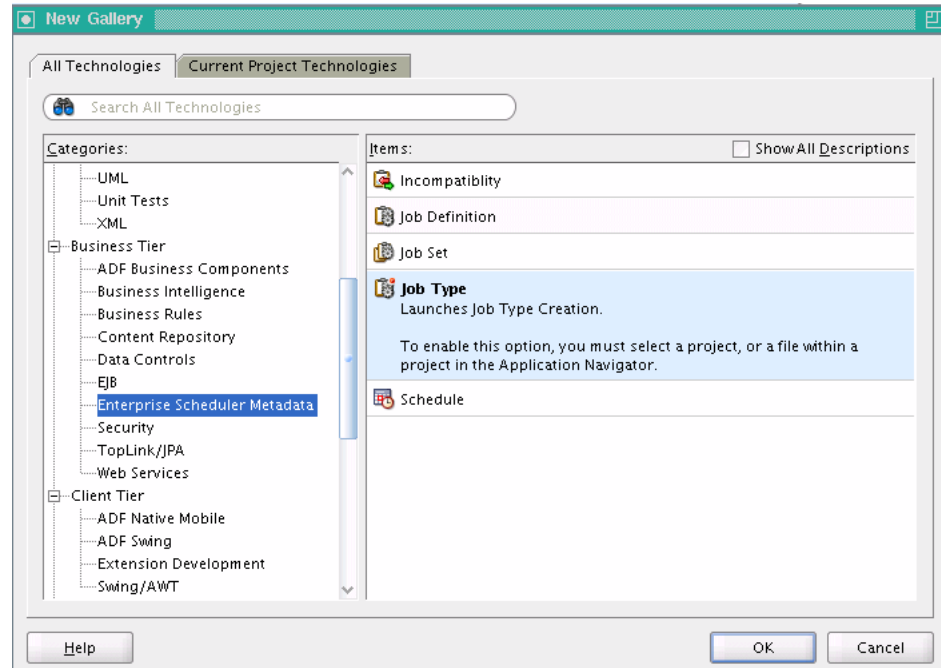
An Oracle Enterprise Scheduler job type specifies an execution type and defines a common set of parameters for a job request.

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**.

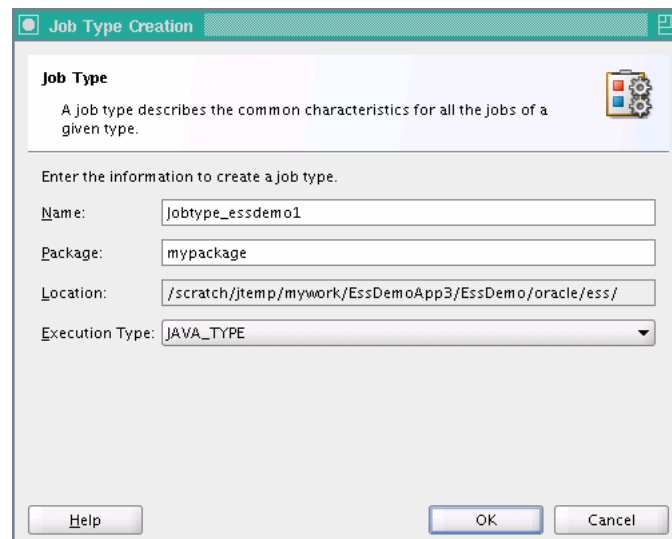
5. In the New Gallery, in the **Items** area select **Job Type** as shown in [Figure 5–13](#).

Figure 5–13 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 5–14](#).

Figure 5–14 Creating a Job Type with the Job Type Creation Wizard

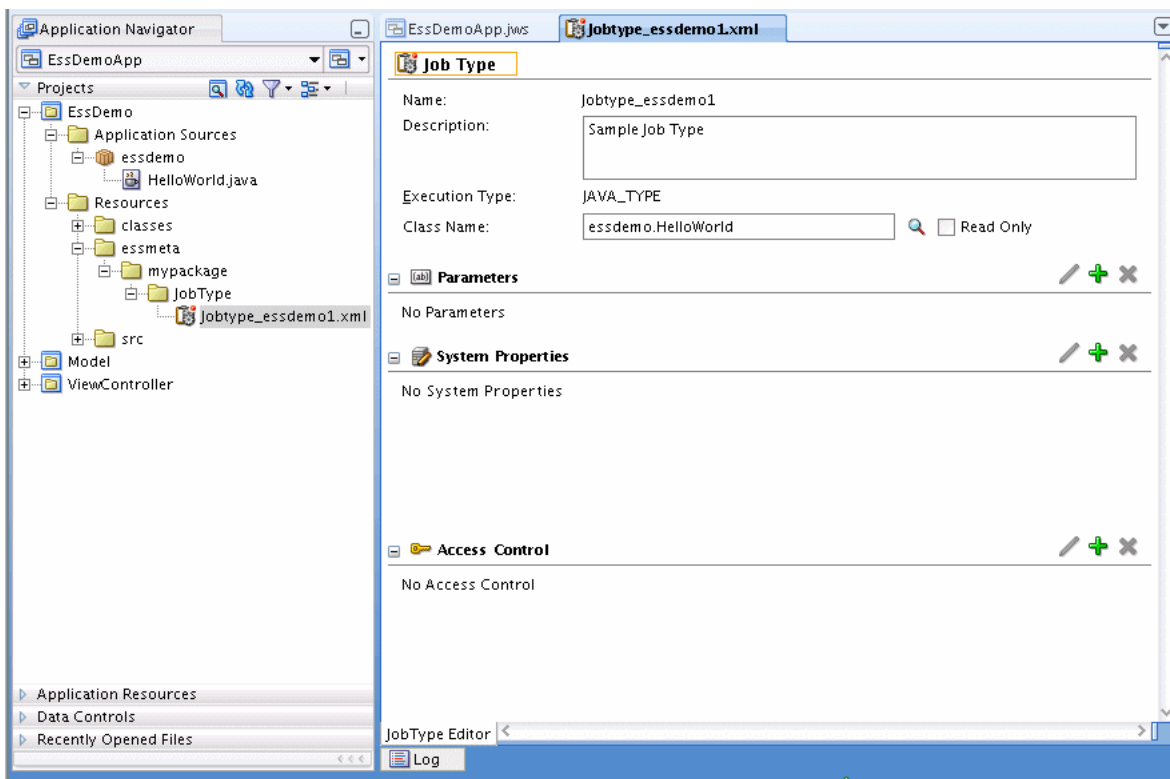


- d. Click **OK**. This creates the `Jobtype_essdemo1.xml` file and Oracle JDeveloper displays the Job Type page.
8. In the Job Type page, in the **Description** field enter a description for the job type. For this example enter: `Sample Java Job Type`.
9. In the **Class Name** field, click the **Browse** 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 5–15](#).

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

Figure 5–15 Adding Sample Job Type Metadata



5.5.2 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 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 sample application select the job type you previously created, **Jobtype_essdemo1**, as shown in [Figure 5-16](#).

Figure 5-16 Using the Job Definition Creation Dialog

Create Job Definition

Job Definition

A job definition describes a job (basic unit of work) that runs in the scheduler. A job definition requires a job type.

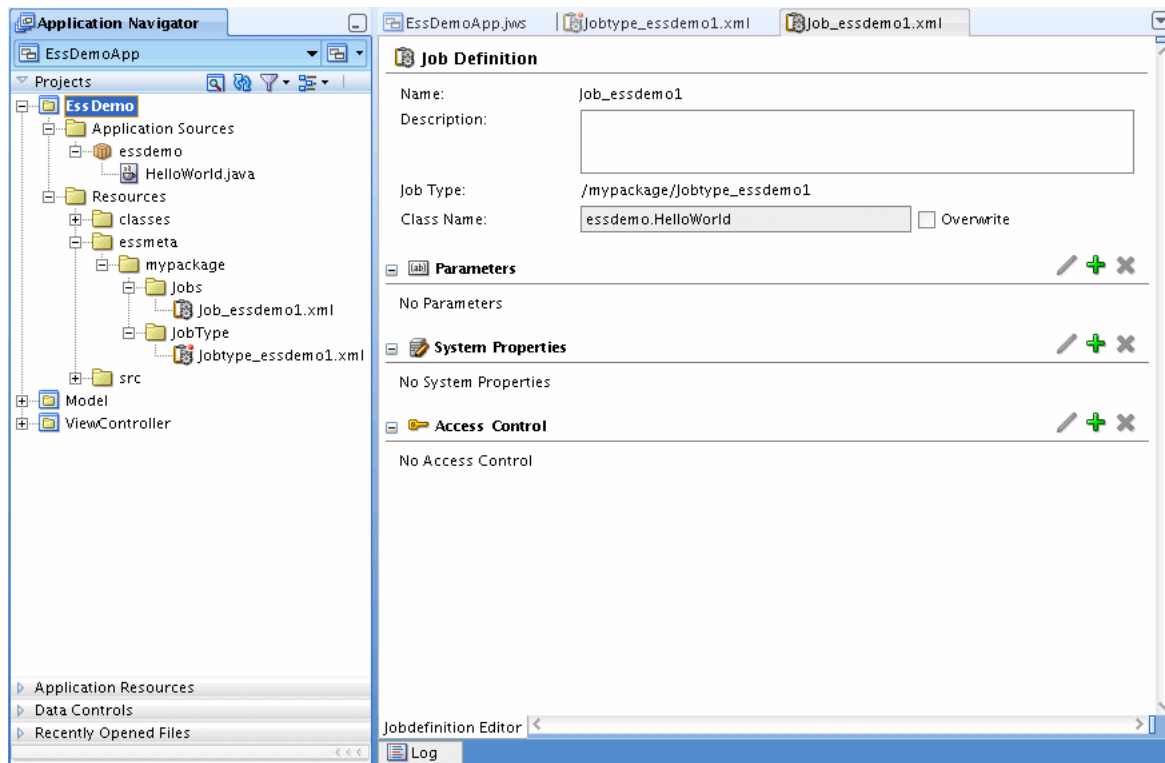
Name:

Package:

Job Type:

Location:

- 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 5-17](#).

Figure 5–17 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 5.6.1, "How to Assemble the EJB Jar Files for Oracle Enterprise Scheduler Sample Application."](#)

5.6 Assembling the Oracle Enterprise Scheduler Sample Application

After you create the 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

5.6.1 How to Assemble the EJB Jar Files for Oracle Enterprise 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.xml`, 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.

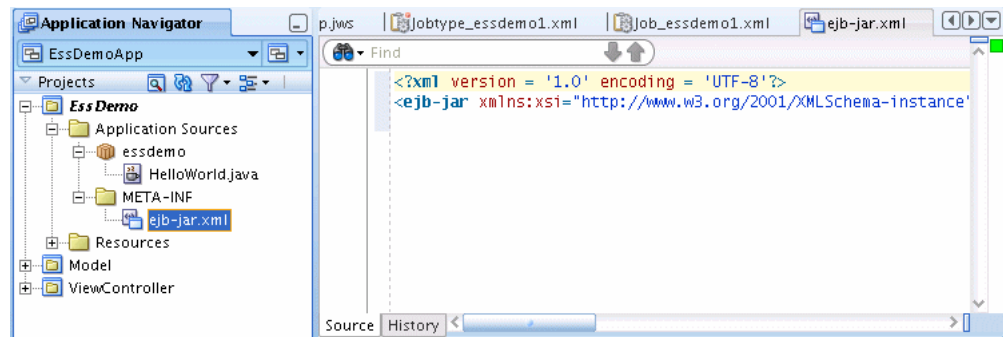
To prepare for the assembly of the sample application, do the following to add the EJB jar files:

- 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 5-4](#).
- 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 5-18](#).

Figure 5-18 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 `ejb-jar.xml` supplied with the sample application. This sample `ejb-jar.xml` file is shown in [Example 5-2](#).

Example 5-2 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 sample application. This file is shown in [Example 5-3](#).

Example 5-3 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>

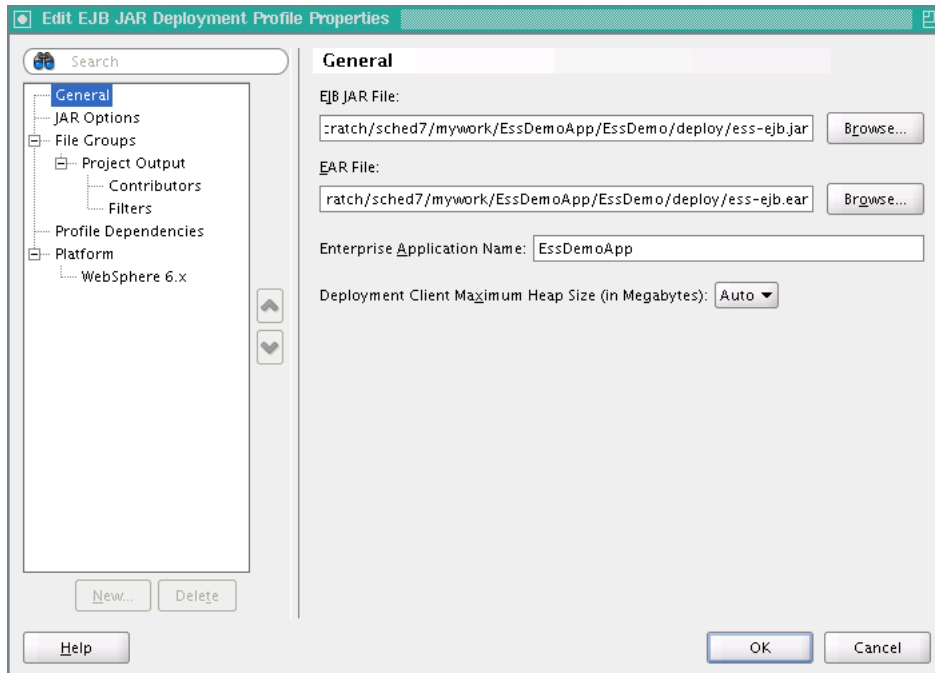
  <run-as-role-assignment>
    <role-name>essSystemRole</role-name>
    <run-as-principal-name>weblogic</run-as-principal-name>
  </run-as-role-assignment>
</weblogic-ejb-jar>
```

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:

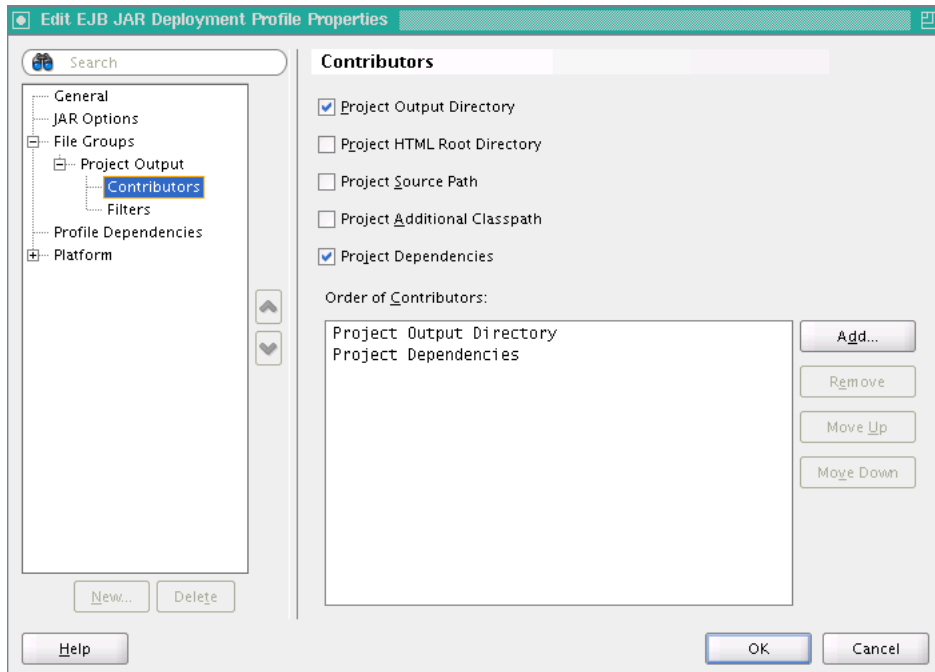
```
[3:40:22 PM] Successful compilation: 0 errors, 0 warnings.
```
3. In Application Navigator select the **EssDemo** project.
4. Select the **New...** icon.
5. In the New Gallery, in the **Categories** area expand **General** and select **Deployment Profiles**.
6. In the New Gallery, in the **Items** 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 5-19](#).

Figure 5–19 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 5–20](#).

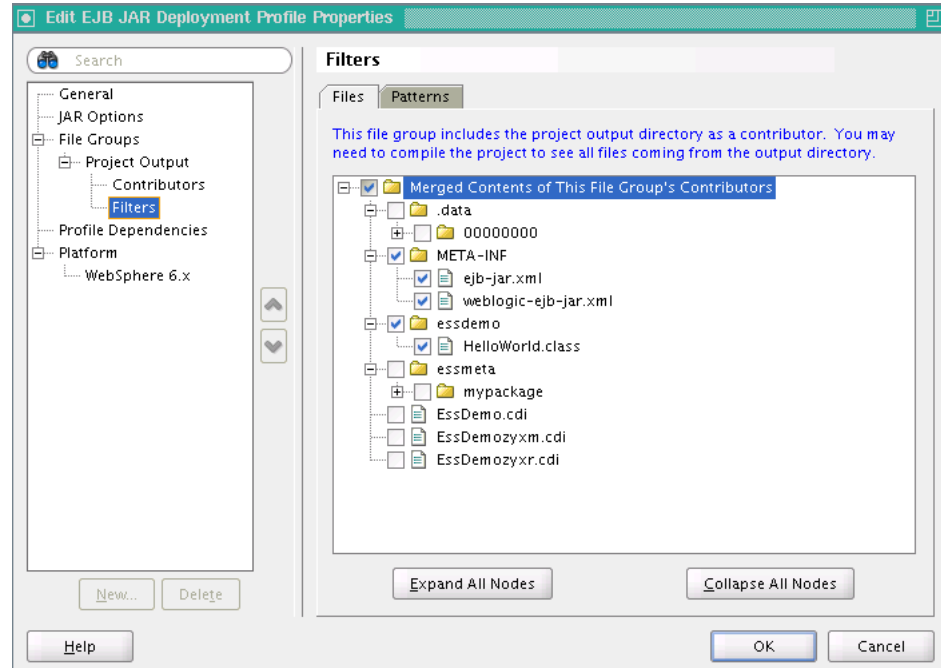
Figure 5–20 Selecting EJB Contributors for the EJB JAR Deployment



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 5–21](#).

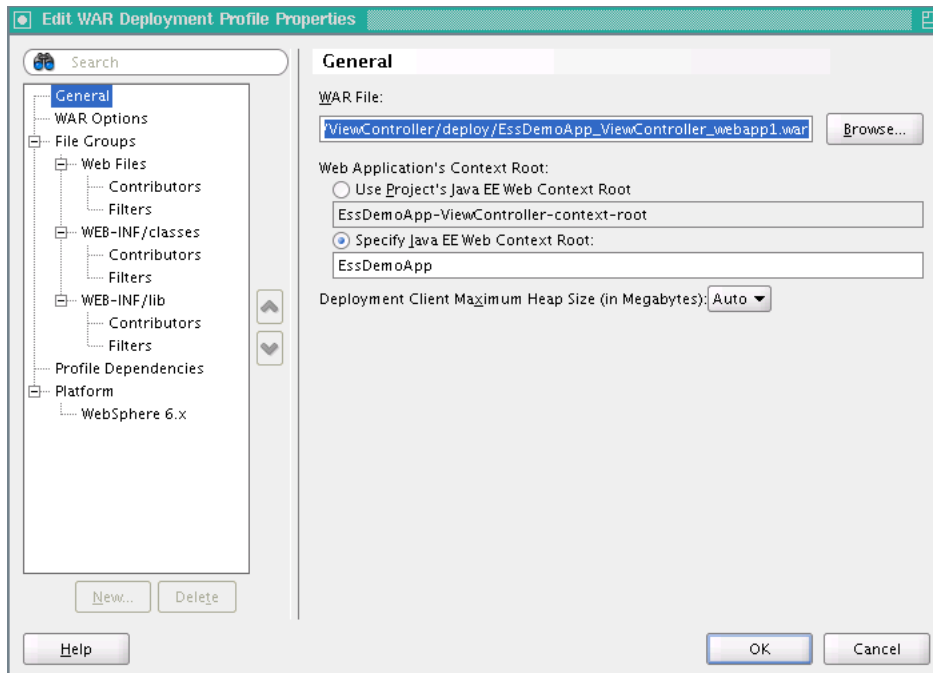
Figure 5–21 EJB JAR Deployment Profile File Groups Filters



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 5–22](#):
 - 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 5–22 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 5–4](#).

Example 5–4 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>
```

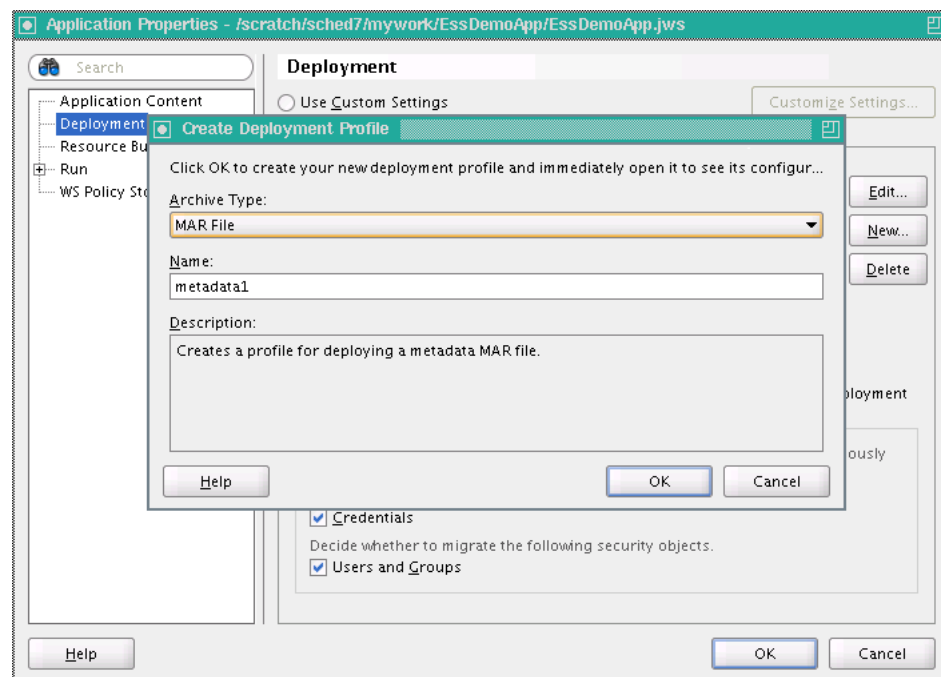
5.6.2 How to Assemble the MAR File for User Metadata

The sample application needs to contain the required MAR profile.

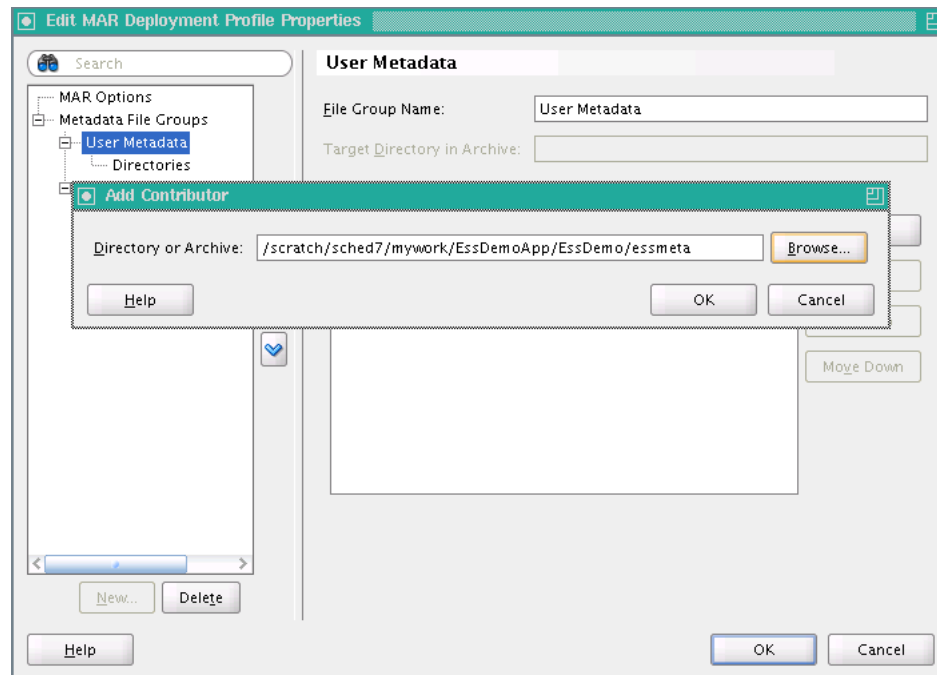
To create the MAR file:

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 5–23](#).

Figure 5–23 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 5–24](#). Note, you select the path that you need to include in the Add Contributor dialog by double-clicking the **essmeta** directory.

Figure 5–24 Adding User Metadata to MAR Profile

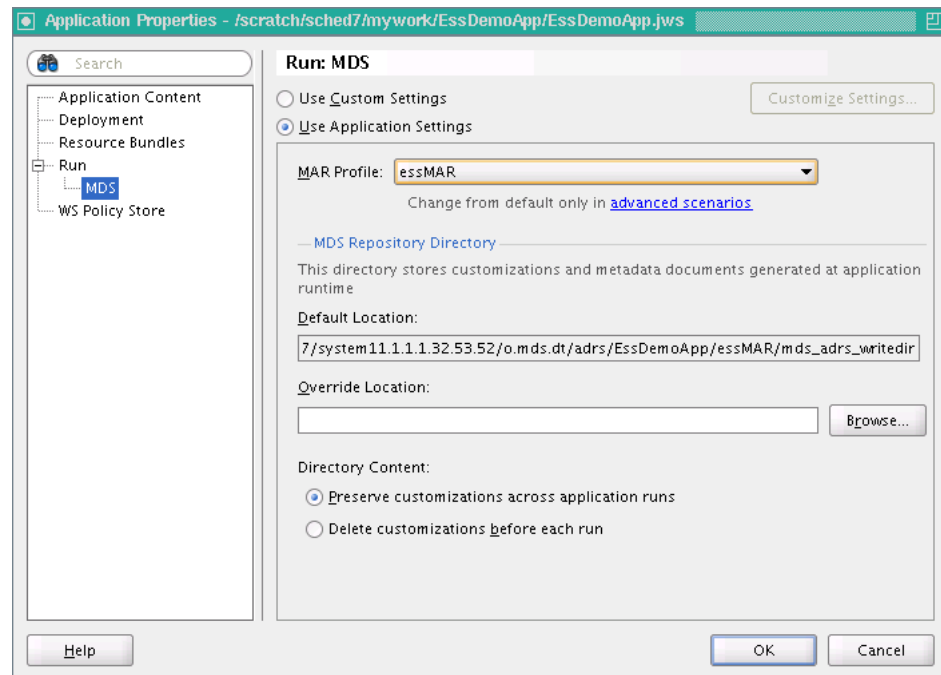
11. On the Add Contributor dialog, click **OK**.
12. In the navigator expand **Metadata File Groups** and **User Metadata** and select **Directories**.
13. Select the `mypackage` directory. This selects all the appropriate information for Oracle Enterprise Scheduler application user metadata for the 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 5.6.3, "How to Assemble the EAR File for Oracle Enterprise Scheduler Sample Application."](#) This namespace should be the same as the package defined in job and job type definition. For more information, see [Section 5.5, "Creating Metadata for Oracle Enterprise 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 5–25](#).
17. Click **OK**.

Figure 5–25 Setting Application Properties Run MDS MAR Profile

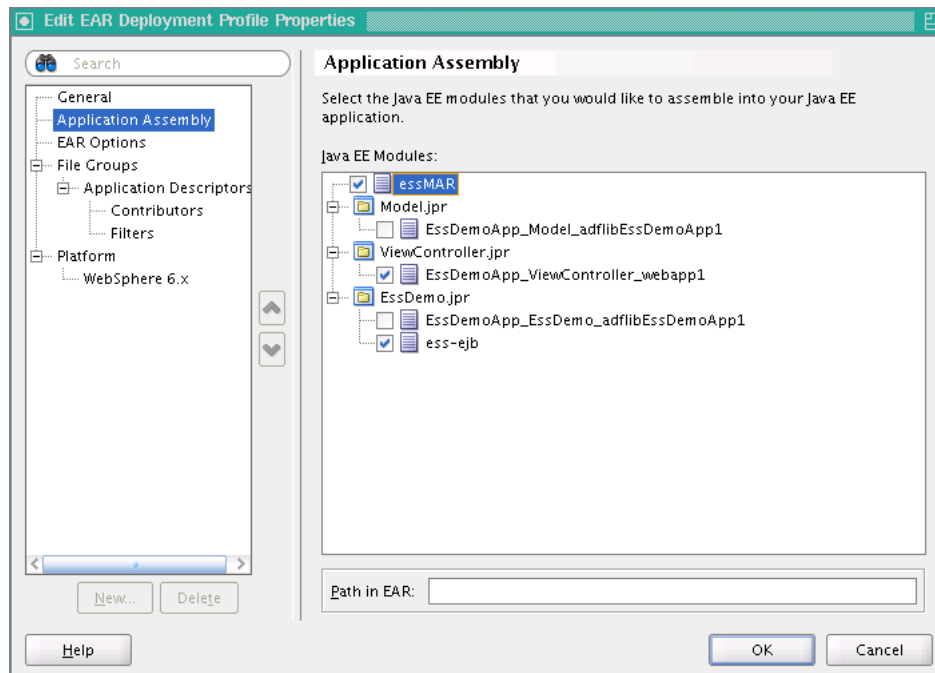
5.6.3 How to Assemble the EAR File for Oracle Enterprise Scheduler Sample Application

You need to prepare an EAR file that assembles the 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 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 check boxes, 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 5–26](#).

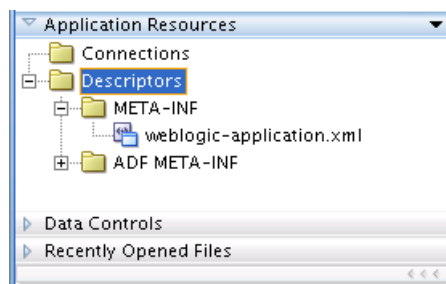
Figure 5–26 Setting Application Assembly Options for EAR File

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

5.6.4 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 5–27](#).

Figure 5–27 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 5–5](#) shows a complete `weblogic-application.xml` file, including this `<library-ref>` element.

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

Example 5-5 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>

```

5.7 Deploying and Running the Oracle Enterprise Scheduler Sample Application

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

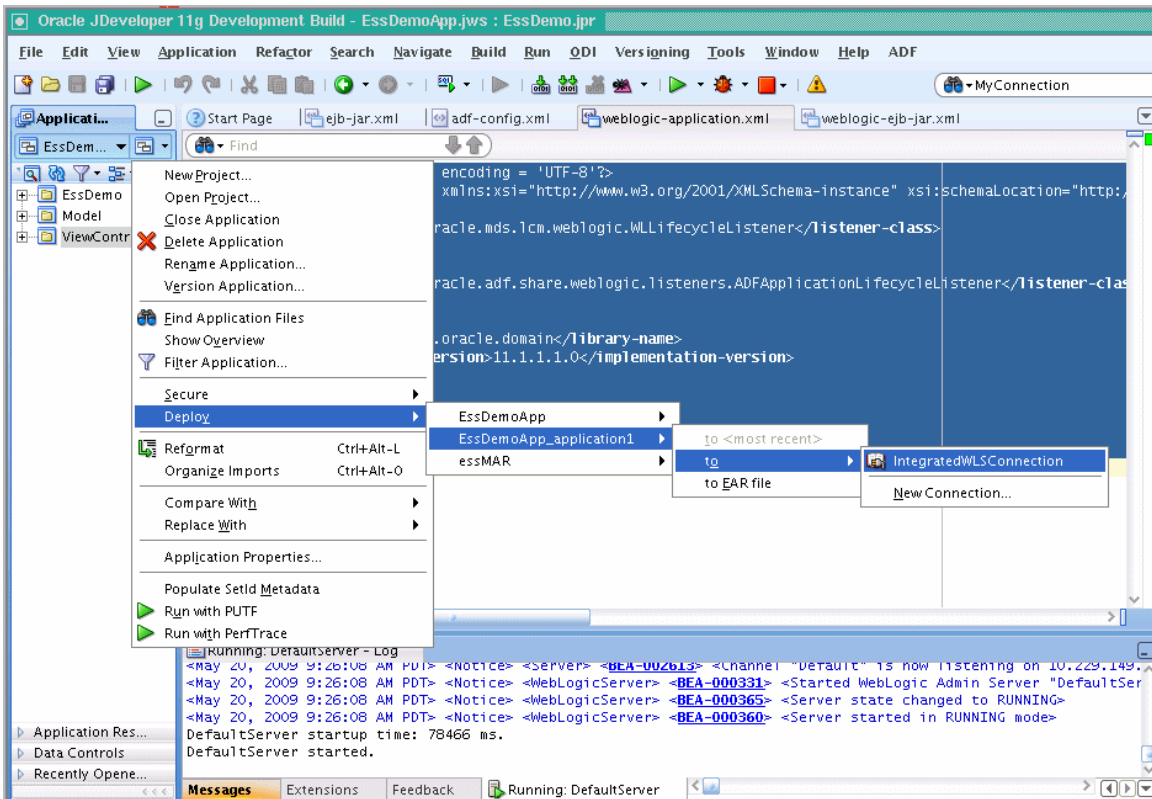
5.7.1 How to Deploy the EssDemoApp Application

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 5-28](#).

Figure 5–28 Deploying the EssDemoApp Application



4. Oracle JDeveloper shows the Deployment Configuration page, as shown in Figure 5–29. Select the appropriate options for your Metadata Repository.

Figure 5–29 Deployment Configuration Page with Metadata Repository Options

Deployment Configuration

Configure and customize settings for this deployment

MDS

- Metadata Repository -

Repository Name: mds-ApplicationMDSDB

Repository Type: DB

Partition Name: APMApp

Path/JNDI Info: jdbc/mds/mds-ApplicationMDSDBDS

- Shared Metadata Repositories -

Namespace	Repository	Type	Partition	Path/JNDI Info

Help Deploy Cancel

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

5.7.2 How to Run the Oracle Enterprise Scheduler Sample Application

To run the 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 5–30](#).

Figure 5–30 Running EssDemo Servlet for Oracle Enterprise Scheduler Sample Application

Enterprise Scheduler Service Tutorial

Launch Job

Job:

Schedule:

Messages

Request Status

reqID	Description	Scheduled time	State	Action
1	Job_essdemo1@Immediately	Wed Jan 07 14:05:05 PST 2009	SUCCEEDED	<input type="button" value="Purge"/>

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 5–31](#).

Figure 5–31 Running EssDemo Servlet with Request Status for Submitted Requests

Enterprise Scheduler Service Tutorial

Launch Job

Job:

Schedule:

Messages

New request 2 launched using Job_essdemo1@Immediately

Request Status

reqID	Description	Scheduled time	State	Action
1	Job_essdemo1@Immediately	Wed Jan 07 14:05:05 PST 2009	SUCCEEDED	<input type="button" value="Purge"/>
2	Job_essdemo1@Immediately	Fri Jan 09 14:31:47 PST 2009	WAIT	<input type="button" value="Cancel"/>

5.7.3 How to Purge Jobs in the Oracle Enterprise Scheduler Sample Application

Using the 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`.

5.8 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.

5.8.1 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*.

5.8.2 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
```

5.9 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.

Note: This chapter includes a tutorial that uses an older release of Oracle JDeveloper. This content is deprecated. For development with a current version of Oracle JDeveloper, see [Chapter 4, "Using Oracle JDeveloper to Generate an Oracle Enterprise Scheduler Application"](#).

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

1. Build a back-end hosting application that includes the code to be scheduled and run.
2. Build a front-end submitter application initiates the job requests.

5.9.1 How to Create the Back-End Hosting Application for Oracle Enterprise Scheduler

Using Oracle JDeveloper you create the back-end application. To create the back-end sample application you do the following:

- Create a back-end 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.

5.9.1.1 Creating the Back-End Hosting Application

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

To create the back-end 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**.

5.9.1.2 Configuring Security for the Back-End Hosting Application

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

To configure security for the back-end 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**.

5.9.1.3 Defining the Deployment Descriptors for the Back-End 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.xml`, 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.xml` must contain descriptions for the Oracle Enterprise Scheduler EJBs.

The Oracle Enterprise Scheduler back-end 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 front-end 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 front-end 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 back-end 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 5-6](#)

Example 5-6 Contents to Copy to `ejb-jar.xml` for a back-end 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-target-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 5-7](#).

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 5-7 Contents to Copy to `weblogic-ejb-jar.xml` for a Back-End 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 5-8](#).

Example 5-8 Contents to Copy to `weblogic-application.xml` for a Back-End 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>

```

5.9.1.4 Creating a Java Implementation Class in the Back-End 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 5–9](#).

Example 5–9 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());
    }
}
```

5.9.1.5 Creating Metadata for the Back-End 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 back-end 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 5.9.1.2, "Configuring Security for the Back-End Hosting Application."](#)

16. Select **Read** and select **Execute**.

17. Click **OK**.

5.9.1.6 Assembling the Back-End Hosting Application for Oracle Enterprise Scheduler

After you create the back-end sample application you use Oracle JDeveloper to assemble the application.

To assemble the back-end application you do the following:

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

5.9.1.6.1 How to Assemble the EJB JAR File for the Back-End Hosting Application The EJB Java archive file includes descriptors for the Java job implementations.

To assemble the EJB JAR file for the back-end 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** check box.

Clearing this check box 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**.

5.9.1.6.2 How to Assemble the MAR and EAR Files for the Back-End Hosting Application The sample application needs to contain the MAR profile and the EAR file that assembles the back-end application.

To create the MAR and EAR files for the back-end 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**.

5.9.1.7 Deploying the Back-End Hosting Application

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

To deploy the back-end 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.

5.9.2 How to Create the Front-End 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 front-end 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 front-end submitter sample application you do the following:

- Create a front-end 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.

5.9.2.1 Creating the Front-End Submitter Application

You use JDeveloper to build the front-end submitter application using similar steps as you used for the back-end hosting application.

To create the front-end submitter application:

1. Complete the steps in [Section 5.9.1.1, "Creating the Back-End 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**.

5.9.2.2 Configuring the ejb-jar.xml File for the Front-End 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 front-end 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 5–10](#)

Example 5–10 Contents to Copy to `ejb-jar.xml` for a Front-End 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-target-class>
    <injection-target-name>application_name</injection-target-name>
</injection-target>
</env-entry>
</interceptor>
</interceptors>
</ejb-jar>
```

5.9.2.3 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**.

5.9.2.4 Configuring Security for the Front-End 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 front-end 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**.

5.9.2.5 Creating the HTTP Servlet for the Front-End 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 front-end 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 5-11](#).

Example 5-11 HTTP Servlet Code for the Front-End 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 will 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>Oracle Enterprise Scheduler
Tutorial</h1></td></tr>");
        out.println("  <tr><td align=\"center\"><table cellpadding=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 = "&nbsp;";
        }

        out.printf("<tr><td>%d</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;
    }
}

```

5.9.2.6 Editing the web.xml File for the Front-End Submitter Application

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

To edit the web.xml file for the front-end 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 5-1](#).

Table 5-1 EJB Resources for the Front-End 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/*`.

5.9.2.7 Editing the weblogic-application.xml file for the Front-End Submitter Application

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

To edit the weblogic-application.xml file for the front-end 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 5–12](#).

Example 5–12 Contents to Copy to `weblogic-application.xml` for a Front-End 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>

```

5.9.2.8 Editing the adf-config file for the Front-End 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 front-end 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 5–13](#).

Example 5–13 Contents to Copy to adf-config.xml for a Front-End 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>

```

5.9.2.9 Assembling the Front-End Submitter Application for Oracle Enterprise Scheduler

After you create the front-end sample application you use Oracle JDeveloper to assemble the application.

To assemble the back-end application you do the following:

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

5.9.2.9.1 How to Assemble the EJB JAR File for the Front-End Submitter Application The EJB Java archive file includes descriptors for the Java job implementations.

To assemble the EJB JAR File for the front-end 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**.

5.9.2.9.2 How to Assemble the WAR File for the Front-End Submitter Application You need to create a web archive file for the web application.

To assemble the WAR file for the front-end 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**.

5.9.2.9.3 How to Assemble the MAR and EAR Files for the Front-End Hosting Application The sample application needs to contain the MAR profile and the EAR file that assembles the back-end application.

To create the MAR and EAR files for the front-end 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**.

5.9.2.10 Deploying the Back-End Hosting Application

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

To deploy the back-end 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**.

Using the Metadata Service

This chapter describes how to use the Oracle Enterprise Scheduler Metadata Service 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:

- [Section 6.1, "Introduction to Using the Metadata Service"](#)
- [Section 6.2, "Accessing the Metadata Service"](#)
- [Section 6.3, "Accessing the Metadata Service with Oracle JDeveloper"](#)
- [Section 6.4, "Querying Metadata Using the Metadata Service"](#)

For information about how to create job definitions, see the following chapters: [Chapter 5, "Use Case Oracle Enterprise Scheduler Sample Application \(Deprecated\)"](#), [Chapter 8, "Creating and Using PL/SQL Jobs"](#), and [Chapter 9, "Creating and Using Process Jobs"](#).

6.1 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 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 will any changes made using a given handle be 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.

6.1.1 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.

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.

6.1.2 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.

6.1.3 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 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.

6.2 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 applications that perform administrative functions using the `oracle.as.scheduler.management` APIs.
- **MBean proxy access:** This access is intended for use by 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.

6.2.1 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 6-1](#) 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 6-1 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();
```

6.3 Accessing the Metadata Service with Oracle JDeveloper

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

6.4 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).

6.4.1 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 6–1](#) lists the comparison operators (comparator argument).

Table 6–1 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 6–2](#) shows code that creates a new filter.

Example 6–2 Creating a Filter with a Filter Comparator for a Query

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

Table 6–2 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.

6.4.2 How to Query Metadata Objects

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

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

[Example 6–3](#) shows a sample routine that queries for a list of job types in the metadata.

Example 6–3 Using Metadata Service Query Methods

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

[Example 6–3](#), 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.

Using Parameters and System Properties

This chapter describes how you can define parameters and values in the Oracle Enterprise Scheduler metadata and runtime services 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 defined property.

This chapter includes the following sections:

- [Section 7.1, "Introduction to Using Parameters and System Properties"](#)
- [Section 7.2, "Using Parameters with the Metadata Service"](#)
- [Section 7.3, "Using Parameters with the Runtime Service"](#)
- [Section 7.4, "Using System Properties"](#)

7.1 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 7.4, "Using System Properties"](#).

7.1.1 What You Need to Know About Application Defined Property and System Property Naming

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

When you use application defined properties, 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 defined properties with names that start with "SYS_" (case-insensitive).

7.1.2 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 7-1](#):

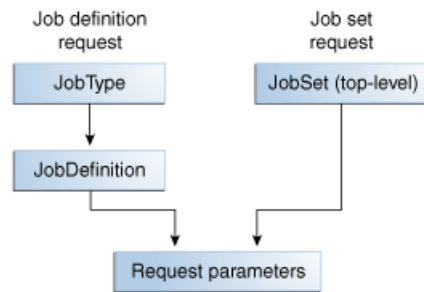
- *Last definition wins*: used when the same parameter is defined repeatedly with the `readonly` flag set to `false` in all cases. In the *last definition wins* model, conflicts are resolved according to the precedence rules where the highest level wins (last definition). For example a property specified 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 7-1](#), 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 7-1 Parameter Precedence Levels

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

7.1.2.1 What You Need to Know About Job Definition Parameter Materialization

[Figure 7-1](#) illustrates the order of precedence taken by parameters defined in various components.

Figure 7-1 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.

7.1.2.2 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

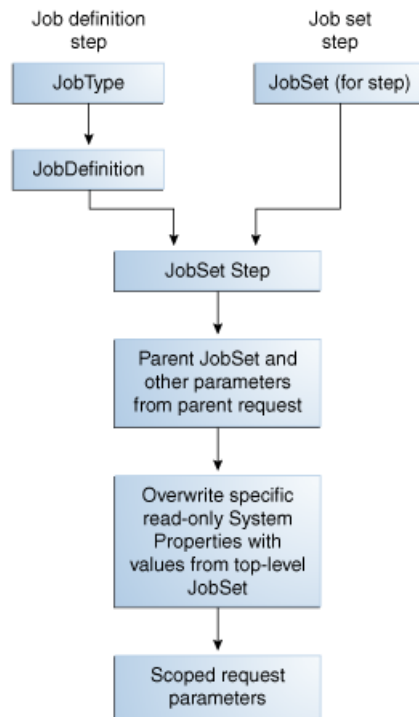
RETRIES, only if the step definition value is > 0

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 7-2 illustrates the parameter precedence for job set steps.

Figure 7-2 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 7-1](#) 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.

7.2 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 defined properties 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 7-2](#).

Table 7-2 ParameterInfo Parameter Properties

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

Table 7–2 (Cont.) ParameterInfo Parameter Properties

Parameter Property	Description
ReadOnly	<p>This boolean flag can be set for each parameter. This flag indicates whether the parameter is read-only.</p> <p>When <code>true</code>, 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.</p> <p>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 7.1.2, "What You Need to Know About Parameter Conflict Resolution and Parameter Materialization".</p>
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 <code>java.util.Calendar</code> 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 7.1.2, "What You Need to Know About Parameter Conflict Resolution and Parameter Materialization"](#).

7.2.1 How to Use Parameters and System Properties in Metadata Objects

[Example 7–1](#) shows code that uses a `ParameterList` to set parameter and system property values in a metadata object.

Example 7–1 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", "readOnlyValue", true);
jd.setParameters(parlist);
```

[Example 7–1](#), 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 defined 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.

7.3 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.

7.3.1 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 defined property. 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 7-2](#) shows code using a `RequestParameters` object with the `add()` method to set a system property value.

Example 7-2 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`.

7.3.2 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 7-3](#) 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 7-3 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`.

7.4 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 7–3](#) 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 7–3](#).

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 7–3 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

Table 7–3 (Cont.) System Properties

Name	Description
BIZ_ERROR_EXIT_CODE	<p>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.</p> <p>This property is optional for a process job type. It is not used for other job types.</p> <p>Type: <code>STRING</code></p>
CLASS_NAME	<p>Specifies the Java executable for a Java job request. This should be the name of a Java class that implements the <code>oracle.as.scheduler.Executable</code> interface. This property is required for a Java job type. It is not used for other job types.</p> <p>Type: <code>STRING</code></p>
CMDLINE	<p>Specifies the command line used to invoke an external program for a Process job request.</p> <p>This property is required for a Process job type. It is not used for other job types.</p> <p>Type: <code>STRING</code></p>
EFFECTIVE_APPLICATION	<p>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 <code>EFFECTIVE_APPLICATION</code> system property. This property can only be specified via metadata and cannot be specified as a submission parameter.</p> <p>Type: <code>STRING</code></p>
ENVIRONMENT_VARIABLES	<p>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.</p> <p>This property is optional for a process job type. It is not used for other job types.</p> <p>Type: <code>STRING</code></p>
EXECUTE_PAST	<p>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 <code>TRUE</code> 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.</p> <p>Valid values for this property are:</p> <ul style="list-style-type: none"> ■ <code>TRUE</code>: All instances specified by a schedule are generated regardless of the time of generation. ■ <code>FALSE</code>: Instances with a scheduled execution time in the past (that is, before the time of generation) will not be generated. <p>If this property is not specified, the system defaults to <code>TRUE</code>.</p> <p>Type: <code>BOOLEAN</code></p>
EXTERNAL_ID	<p>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.</p> <p>Type: <code>STRING</code></p>
GROUP_NAME	<p>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.</p> <p>Type: <code>STRING</code></p>

Table 7–3 (Cont.) System Properties

Name	Description
INPUT_LIST	<p>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.</p> <p>Oracle Enterprise Scheduler imposes no format on the value of this property.</p> <p>Type: <code>STRING</code></p>
LISTENER	<p>Specifies the event listener class associated with the request. This should be the name of a Java class that implements the <code>oracle.as.scheduler.EventListener</code> interface.</p> <p>Type: <code>STRING</code></p>
LOCALE	<p>Specifies the locale associated with the request.</p> <p>Type: <code>STRING</code></p>
OUTPUT_LIST	<p>Specifies output from a request.</p> <p>The output of a serial job set is the <code>OUTPUT_LIST</code> of the last step. The output of a parallel job set is the concatenation of the <code>OUTPUT_LIST</code> of all the steps, in no guaranteed order, with <code>oracle.as.scheduler.SystemProperty.OUTPUT_LIST_DELIMITER</code> as a separator.</p> <p>Type: <code>STRING</code></p>
POST_PROCESS	<p>Specifies the post-process callout handler class. This should be the name of a Java class that implements the <code>oracle.as.scheduler.PostProcessHandler</code> interface.</p> <p>Type: <code>STRING</code></p>
PRE_PROCESS	<p>Specifies the pre-process callout handler class. This should be the name of a Java class that implements the <code>oracle.as.scheduler.PreProcessHandler</code> interface.</p> <p>Type: <code>STRING</code></p>
PRIORITY	<p>Specifies the request processing priority. The priority interval is [0..9] with 0 as the lowest priority and 9 as the highest.</p> <p>Default: If this property is not specified, the system default value used is 4.</p> <p>Type: <code>INTEGER</code></p>
PROCEDURE_NAME	<p>Specifies the name of the PL/SQL stored procedure to be called for a SQL job request. The stored procedure should be specified using <code>schema.name</code> format.</p> <p>The property is required for a SQL job type. It is not used for other job types.</p> <p>Type: <code>STRING</code></p>
PRODUCT	<p>Specifies the product within the application that submitted the request.</p> <p>Type: <code>STRING</code></p>
REDIRECTED_OUTPUT_FILE	<p>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.</p> <p>This property is optional for a Process job type. It is not used for other job types.</p> <p>Type: <code>STRING</code></p>
REPROCESS_DELAY	<p>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.</p> <p>Default: If this property is not specified, the system default used is 5.</p> <p>Type: <code>INTEGER</code></p>
REQUEST_CATEGORY	<p>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.</p> <p>Type: <code>STRING</code></p>

Table 7–3 (Cont.) System Properties

Name	Description
REQUEST_EXPIRATION	<p>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.</p> <p>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.</p> <p>Type: INTEGER</p>
REQUESTED_PROCESSOR	<p>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.</p> <p>If this property is specified for a request, the request processor's work assignments <code>oracle.as.scheduler.WorkAssignment</code> (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 <code>READY</code> state and will not be executed until the node is restarted.</p> <p>Type: STRING</p>
RETRIES	<p>Specifies the retry limit for a failed request. If request execution fails, the request will be 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.</p> <p>Default: If this property is not specified, the system default used is 0.</p> <p>Type: INTEGER</p>
RUNAS_APPLICATIONID	<p>Specifies the <code>runAs</code> 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.</p> <p>Type: STRING</p>
SELECT_STATE	<p>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 <code>SELECT_STATE</code> must be set to <code>true</code>. If <code>SELECT_STATE</code> 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.</p> <p>Type: BOOLEAN</p>
SQL_JOB_CLASS	<p>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.</p> <p>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.</p> <p>This property is optional for a SQL job type. It is not used for other job types.</p> <p>Type: STRING</p>
SUBMITTING_APPLICATION	<p>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.</p> <p>Type: STRING</p>

Table 7–3 (Cont.) System Properties

Name	Description
SUCCESS_EXIT_CODE	<p>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.</p> <p>This property is optional for a Process job type. It is not used for other job types.</p> <p>Type: <i>STRING</i></p>
USER_FILE_DIR	<p>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.</p> <p>Oracle Enterprise Scheduler supports a configuration parameter that specifies a file directory where requests may store files. At request submission, a <i>USER_FILE_DIR</i> property is automatically added for the request if the configuration parameter is currently set and <i>USER_FILE_DIR</i> 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.</p> <p>Type: <i>STRING</i></p>
USER_NAME	<p>Specifies the name of the user used to execute the request. Normally this is the submitting user unless the <i>RUNAS_APPLICATIONID</i> property was set in the job metadata. This property is automatically set by Oracle Enterprise Scheduler during request submission.</p> <p>Type: <i>STRING</i></p>
WARNING_EXIT_CODE	<p>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.</p> <p>This property is optional for a Process job type. It is not used for other job types.</p> <p>Type: <i>STRING</i></p>
WORK_DIR_ROOT	<p>Specifies the working directory for the spawned process of a Process job request.</p> <p>This property is optional for a Process job type. It is not used for other job types.</p> <p>Type: <i>STRING</i></p>

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:

- [Section 8.1, "Introduction to Using PL/SQL Stored Procedure Job Definitions"](#)
- [Section 8.2, "Creating a PL/SQL Stored Procedure for Oracle Enterprise Scheduler"](#)
- [Section 8.3, "Performing Oracle Database Tasks for PL/SQL Stored Procedures"](#)
- [Section 8.4, "Creating and Storing Job Definitions for PL/SQL Job Types"](#)

For information about how to use the Runtime Service, see [Chapter 14, "Using the Runtime Service"](#).

8.1 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 14, "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.

8.2 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

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

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 8-1](#) shows a sample `HELLO_WORLD` stored procedure for use with Oracle Enterprise Scheduler.

Example 8-1 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);
    end;
```

```

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;
/

```

8.2.2 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 8–1](#) indicates the Oracle Enterprise Scheduler state based on the result of the PL/SQL stored procedure.

Table 8–1 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 <code>SUCCEEDED</code> state, bearing any subsequent errors completing the request.

Table 8–1 (Cont.) Terminal States for PL/SQL Stored Procedure Results

Final State	Description
WARNING	<p>If the PL/SQL stored procedure returns with an exception, the request state is based on the SQL error code of the exception.</p> <p>The request transitions to the <code>WARNING</code> terminal state if the SQL error code ranges from -20900 to -20919.</p>
ERROR	<p>If the PL/SQL stored procedure returns with an exception, the request state is based on the SQL error code of the exception.</p> <p>The request transitions to the <code>ERROR</code> terminal state for any error code outside the range of -20900 to -20919 (error codes within this range indicate a <code>WARNING</code>).</p> <p>Return codes in the range -20920 to -20929 result in an <code>ERROR</code> state with a <code>BUSINESS</code> error type, where the request is not subject to automatic retries.</p>

8.2.3 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.

8.2.4 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.

8.3 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.

8.3.1 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_RUNTIME`. 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;
```

8.3.2 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`.

8.4 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 6, "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 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 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 job. No additional privilege grant is needed for an Oracle Enterprise Scheduler SQL request to use that default job class.

8.4.1 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 8-2](#) describes the SystemProperties that are appropriate for a PL/SQL stored procedure job type.

Table 8–2 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 6.2, "Accessing the Metadata Service"](#). Use the Metadata Service `addJobType()` method to store the `JobType` in metadata.
- Use a `MetadataObjectId` 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 8.4.3, "Using a PL/SQL Stored Procedure with an Oracle Enterprise Scheduler Application"](#) for sample code.

8.4.2 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 8.4.3, "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.

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

[Example 8–2](#) shows sample code in which job type and job definition application metadata are created for a SQL job type.

Example 8–2 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;
        }
    }
}

```

Creating and Using Process Jobs

This chapter describes how to use Oracle Enterprise Scheduler to create process jobs, which run a script or binary command in a forked process.

This chapter includes the following sections:

- [Section 9.1, "Introduction to Creating Process Job Definitions"](#)
- [Section 9.2, "Creating and Storing Job Definitions for Process Job Types"](#)

For information about how to use the Runtime Service, see [Chapter 14, "Using the Runtime Service"](#).

9.1 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 14, "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 14, "Using the Runtime Service"](#).

9.2 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 6, "Using the Metadata Service"](#).

9.2.1 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 9–1](#) 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 9–1 System Properties for Process Type Jobs

System Property	Description
<code>BIZ_ERROR_EXIT_CODE</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.
<code>CMDLINE</code>	Command line required for invoking an external program.
<code>ENVIRONMENT_VARIABLES</code>	A comma-separated list of name/value pairs (name=value) representing the environment variables to be set for spawned processes.
<code>REDIRECTED_OUTPUT_FILE</code>	Specifies the file where standard output and error streams are redirected for a process job request.
<code>REQUESTED_PROCESSOR</code>	The Oracle WebLogic Server node on which a spawned job is executed.
<code>SUCCESS_EXIT_CODE</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.
<code>WARNING_EXIT_CODE</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.
<code>WORK_DIR_ROOT</code>	The working directory for a spawned process.

For more information about system properties, see [Chapter 7, "Using Parameters and System Properties."](#)

[Example 9–1](#) shows a sample job type definition with a `PROCESS_TYPE`.

Example 9–1 Creating an Oracle Enterprise 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 jobId = 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);

        jobId = 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, jobId);
        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 9–1](#), 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 6.2, "Accessing the Metadata Service"](#). Use the Metadata Service `addJobType()` method to store the `JobType` in metadata.
- The `MetadataObjectId`, returned by `addJobType()`, uniquely identifies metadata objects in the metadata repository using a unique identifier.

9.2.2 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 9–1](#) shows how to create a job definition using the job definition constructor and the job type. [Table 9–1](#) describes some of the system properties that are associated with the job definition.

As shown in [Example 9–1](#), 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 6.2, "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`.

9.3 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.

Defining and Using Schedules

This chapter describes how to define schedules that you can associate with a Oracle Enterprise Scheduler job definition, specifying when a job request runs and including administrative actions such as workshifts that specify time-based controls for processing with Oracle Enterprise Scheduler.

This chapter includes the following sections:

- [Section 10.1, "Introduction to Schedules"](#)
- [Section 10.2, "Defining a Recurrence"](#)
- [Section 10.3, "Defining an Explicit Date"](#)
- [Section 10.4, "Defining and Storing Exclusions"](#)
- [Section 10.5, "Defining and Storing Schedules"](#)
- [Section 10.6, "Identifying Job Requests That Use a Particular Schedule"](#)
- [Section 10.7, "Updating and Deleting Schedules"](#)

10.1 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, 2012. 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.

10.2 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.

10.2.1 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 10–1](#) shows the recurrence fields helper classes available to specify a recurrence pattern.

Table 10–1 Recurrence Field Helper Patterns

Recurrence Field	Description
<code>DAY_OF_MONTH</code>	Defines the day of a month
<code>DAY_OF_WEEK</code>	Enumeration of the day of a week
<code>FREQUENCY</code>	Defines the repeat frequency of a Recurrence. Choices are: <ul style="list-style-type: none"> ■ <code>DAILY</code>: Indicates every day repetition ■ <code>HOURLY</code>: Indicates every hour repetition ■ <code>MINUTELY</code>: Indicates every minute repetition ■ <code>MONTHLY</code>: Indicates every month repetition ■ <code>SECONDLY</code>: Indicates every second repetition ■ <code>WEEKLY</code>: Indicates every week repetition ■ <code>YEARLY</code>: Indicates every year repetition
<code>MONTH_OF_YEAR</code>	Defines the months of the year
<code>TIME_OF_DAY</code>	Defines the time of the day

Table 10–1 (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 10–1](#) 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 10–1 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 10–1](#), 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 10–2](#) 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 10–2 Defining a Recurrence with Four Hourly Frequency

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

In [Example 10–2](#), 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 10–3](#) shows a sample recurrence created using the `RecurrenceFields` helper class and a monthly frequency.

Example 10–3 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 10–3](#) 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 10-4 shows a sample recurrence created using the `RecurrenceFields` helper class and a monthly frequency specified with a start date and time.

Example 10-4 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 10-4 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.

10.2.2 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 10-5 shows a sample recurrence created using an iCalendar expression.

Example 10-5 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.

10.2.3 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 10-1](#)).
- 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.

10.2.4 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).

10.3 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.

10.3.1 How to Define an Explicit Date

[Example 10–6](#) shows an explicit date definition.

Example 10–6 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 10–6](#) 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.

10.3.2 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.

10.4 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.

10.4.1 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 10-7](#) shows how to create an `Exclusion` instance using a recurrence.

Example 10-7 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 10-7](#) defines an individual exclusion. For information about creating a list of Exclusions, see [Section 10.4.2, "How to Create an Exclusions Definition"](#).

10.4.2 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 10-8](#) shows how to create an `ExclusionDefinition` and store the definition to the metadata repository.

Example 10-8 Creating and Storing a List of Exclusions in an ExclusionDefinition

```
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 10-8](#) that the `ExclusionsDefinition` constructor needs three arguments.

10.5 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 10.3, "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 10.2, "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 10.4, "Defining and Storing Exclusions"](#)

10.5.1 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.

10.5.2 What Happens When You Define and Store a Schedule

[Example 10-9](#) 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 10-9 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 10–10 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 10–10 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);
    }
}

```

10.5.3 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 a `TimeZone` set.

10.6 Identifying Job Requests That Use a Particular Schedule

You can use Fusion Middleware 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 Requests" in *Oracle Fusion Middleware Administrator's Guide for Oracle Enterprise Scheduler*.

10.7 Updating and Deleting Schedules

You can use Fusion Middleware 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 Requests" in *Oracle Fusion Middleware Administrator's Guide for Oracle Enterprise Scheduler*.

Using the Oracle Enterprise Scheduler Web Service

This chapter describes how you can use the Oracle Enterprise Scheduler web service for accessing a subset of the Oracle Enterprise Scheduler runtime functionality.

This chapter includes the following sections:

- [Section 11.1, "Introduction to the Oracle Enterprise Scheduler Web Service"](#)
- [Section 11.2, "Developing and Using ESSWebservice Applications"](#)
- [Section 11.3, "ESSWebservice WSDL File"](#)
- [Section 11.4, "Use Case Using Oracle Enterprise Scheduler ESSWebservice from a BPEL Process"](#)
- [Section 11.5, "Creating the ESSWebService Application and a SOA Project"](#)
- [Section 11.6, "Creating the ESSWebService Reference"](#)
- [Section 11.7, "Adding the BPEL Process to Call the ESSWebService"](#)
- [Section 11.8, "Using Additional ESSWebService Operations"](#)
- [Section 11.9, "Securing the Oracle Enterprise Scheduler Web Service"](#)
- [Section 11.10, "Deploying and Testing the Project"](#)

11.1 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 11-1](#) summarizes the operations available with ESSWebservice.

Table 11-1 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 <code>add_printer</code> , <code>add_notification</code> , <code>add_layout</code> in concurrent processing. The parameters to these legacy routines are passed as arguments to <code>addPPAction</code> in the order in which they were declared in the original routine. For more information, see Section 11.8, "Using Additional ESSWebService Operations"
addPPActions	Synchronous	Similar to <code>addPPAction</code> , 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 <code>WAIT</code> or <code>READY</code> state. For parent requests, this operation will cascade to all eligible child requests.
releaseRequest	Synchronous	Releases a request from the <code>HOLD</code> 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 11.8, "Using Additional ESSWebService Operations" .

Table 11–1 (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 11.8, "Using Additional ESSWebService Operations" .
submitRecurringRequest	Synchronous	Submits a new recurring job request (a request with a schedule). For more information, see Section 11.8, "Using Additional ESSWebService Operations" .
submitRequest	Synchronous	Submits a new job request. For more information, see Section 11.4, "Use Case Using Oracle Enterprise Scheduler ESSWebservice from a BPEL Process" .

11.2 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 5, "Use Case Oracle Enterprise Scheduler Sample Application \(Deprecated\)"](#). 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.

11.2.1 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 5, "Use Case Oracle Enterprise Scheduler Sample Application \(Deprecated\)."](#)

11.2.2 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.

11.2.3 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()` programmatically on the client side, using the web service proxies, then the web service client must set these addressing headers.

11.2.4 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.

11.2.5 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.

11.3 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
```

11.4 Use Case Using Oracle Enterprise 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 5, "Use Case Oracle Enterprise Scheduler Sample Application \(Deprecated\)."](#)

11.5 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

11.5.1 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.

11.6 Creating the ESSWebService Reference

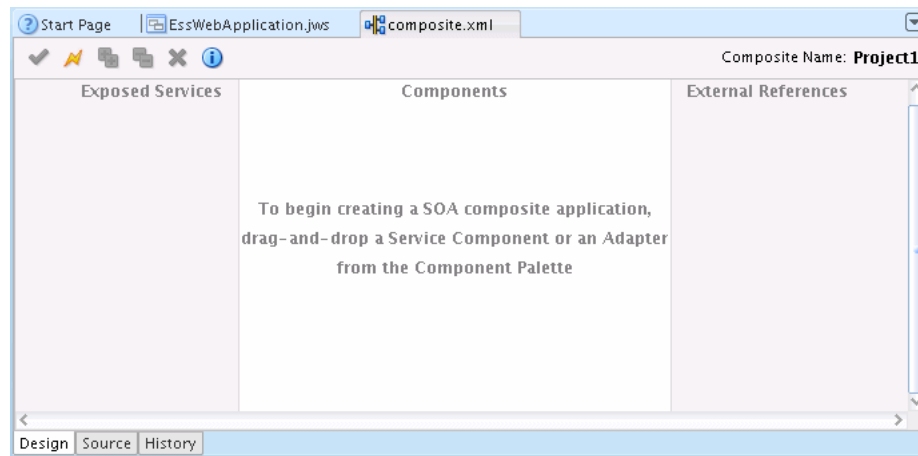
In the SOA composite application you need to add the ESSWebservice reference to make the web service available for a partner link in the SOA composite application.

11.6.1 How to Add the ESSWebService Partner Link

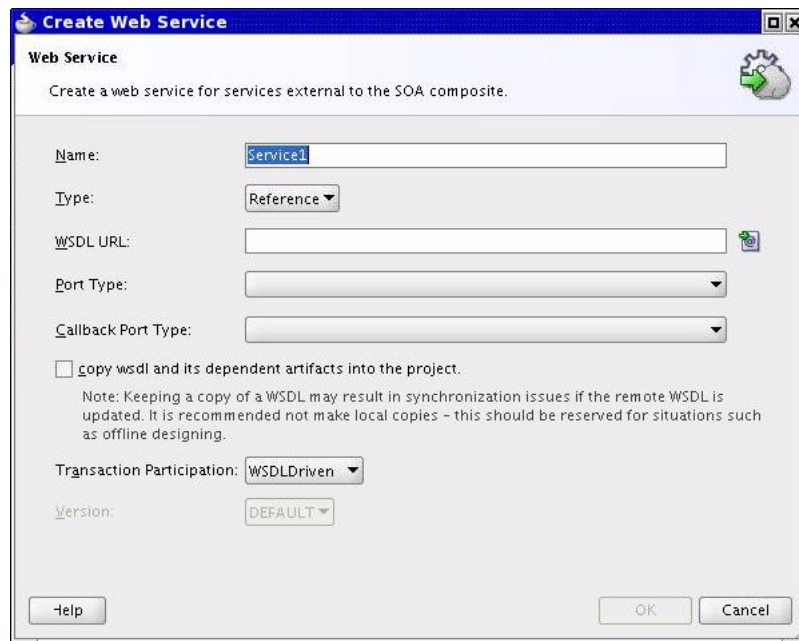
You need to add the ESSWebService partner link to the SOA composite application.

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 11-1](#).

Figure 11–1 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 11–2](#).

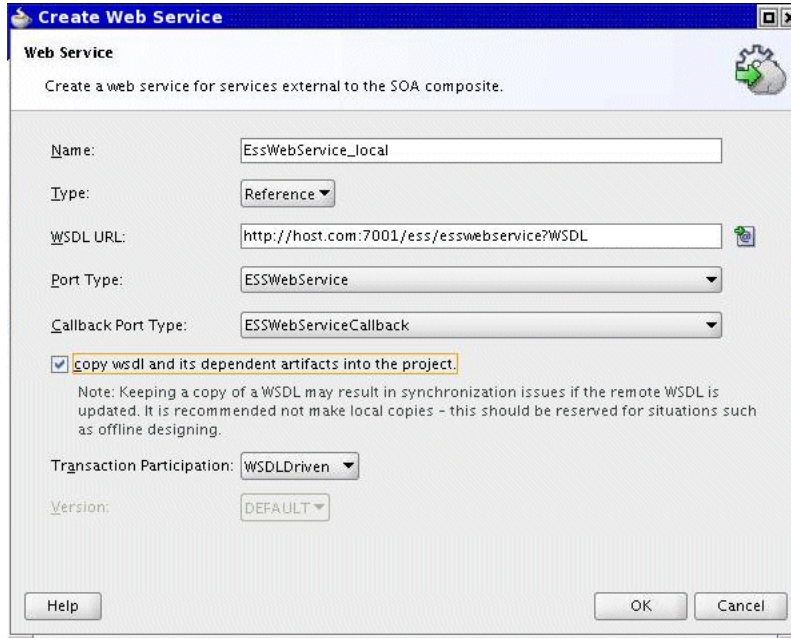
Figure 11–2 Create Web Service Dialog

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 11–3](#).

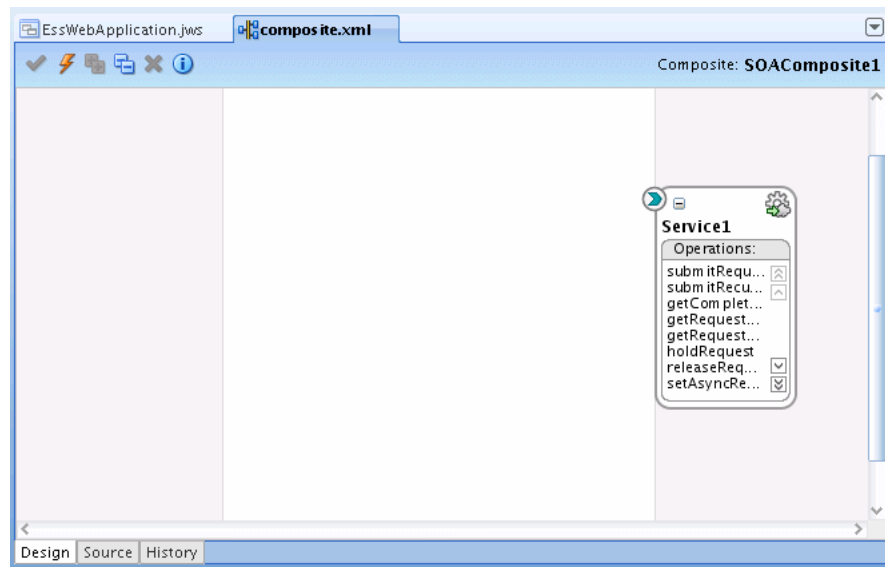
Figure 11–3 Create Web Service with ESSWebService WSDL



Select the check box **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 11–4](#).

Figure 11–4 Composite.xml with ESSWebService External Reference

11.7 Adding the BPEL Process to Call the ESSWebService

Now you need to add a BPEL Process to call the ESSWebService operations.

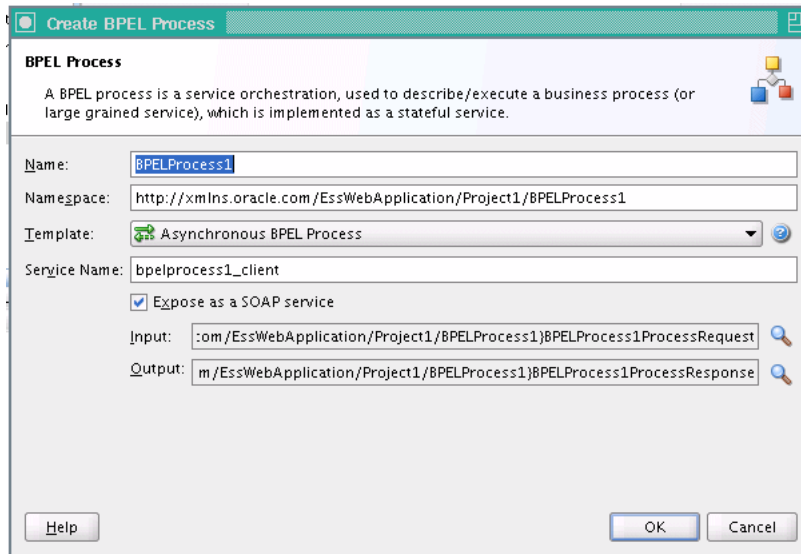
11.7.1 How to Add a BPEL Process to Call the ESSWebService

You need to add a BPEL process to use the ESSWebService.

To add a BPEL process to use the ESSWebService:

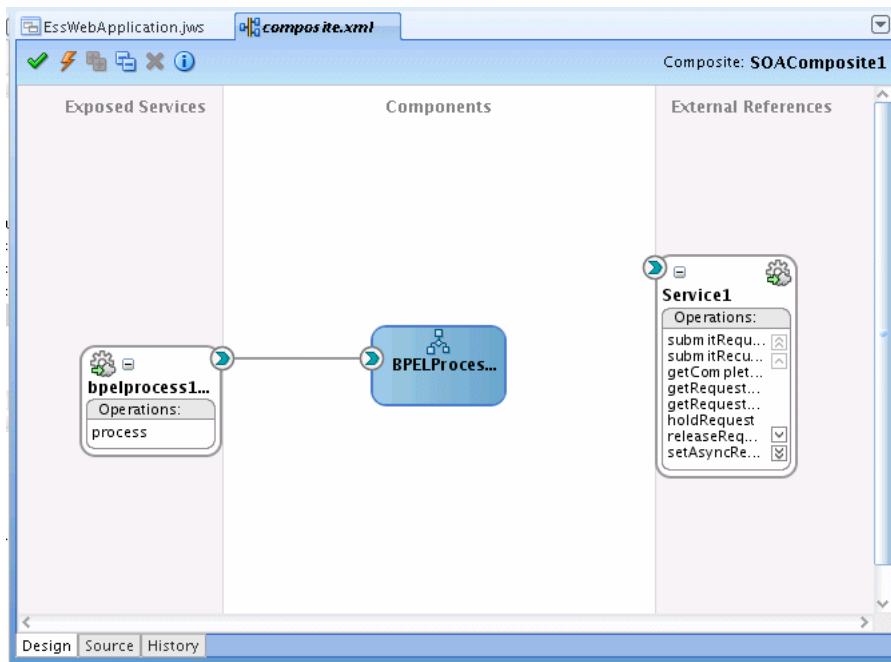
1. In the Application Navigator, in Project1 select **composite.xml**.
2. In the Component Palette, from the SOA dropdown list in the Service Components area select **BPEL Process**.
3. Drag-and-drop a BPEL process to the components swim lane. This displays the Create BPEL Process dialog, as shown in [Figure 11–5](#).

Figure 11–5 Create BPEL Process Dialog for New BPEL Process



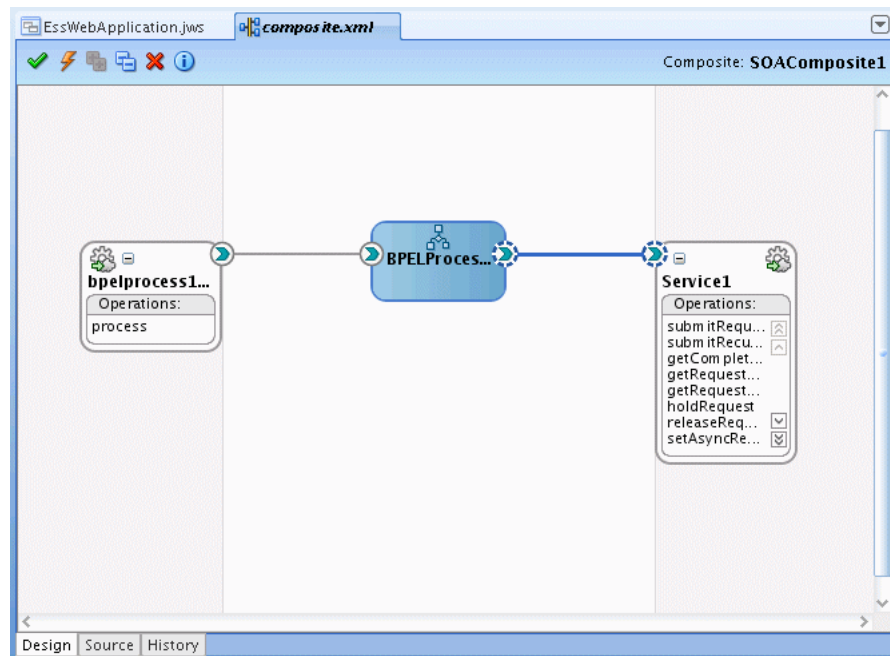
4. Click **OK**. This adds the BPEL process to composite.xml, as shown in [Figure 11–6](#).

Figure 11–6 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 11–7](#).

Figure 11–7 Adding A Reference to the Oracle Enterprise Scheduler Web Service in composite.xml



6. Click the **Save All** icon to save the project files.

11.7.2 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 11–1](#).

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 11–1](#) 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 11.6, "Creating the ESSWebService Reference"](#) that have been imported into the composite. The directory path should be relative the `BPELProcess1.xsd` file. [Example 11–1](#) shows the composite schema file with a reference to the web service artifacts.

Example 11–1 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.

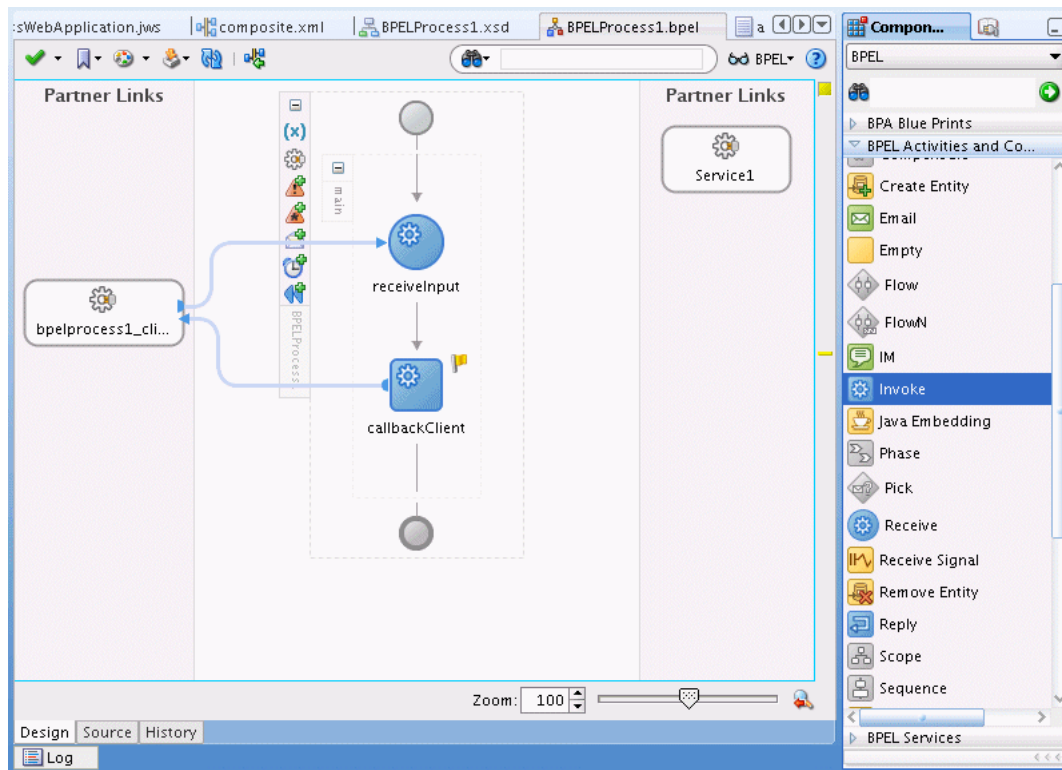
11.7.3 How to Invoke the ESSWebService submitRequest Operation

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.

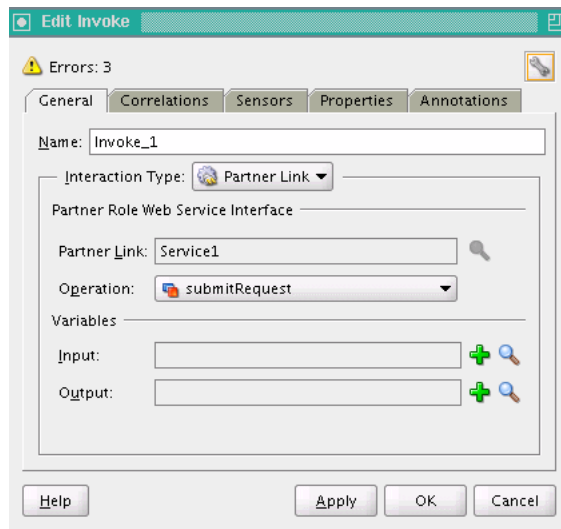
To add the Invoke activity to submit the request using ESSWebService:

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 11-8](#).

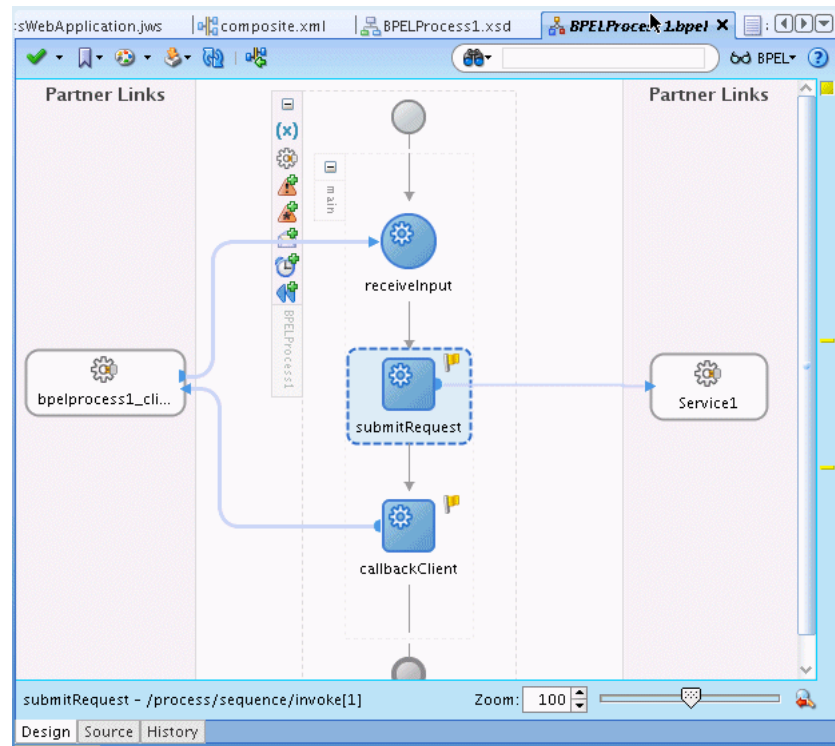
Figure 11-8 BPEL Process Before Adding Invoke Activity for ESSWebService SubmitRequest



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 11-9](#).

Figure 11–9 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 11–10](#).

Figure 11–10 Adding the submitRequest Invoke Activity

11.7.4 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 submitRequest 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 submitRequest.

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 11–2. If your BPEL schema differs from the submitRequest message type, use Table 11–2 as a guide for how to populate the values manually with the Assign activity Copy operation.

Table 11–2 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.

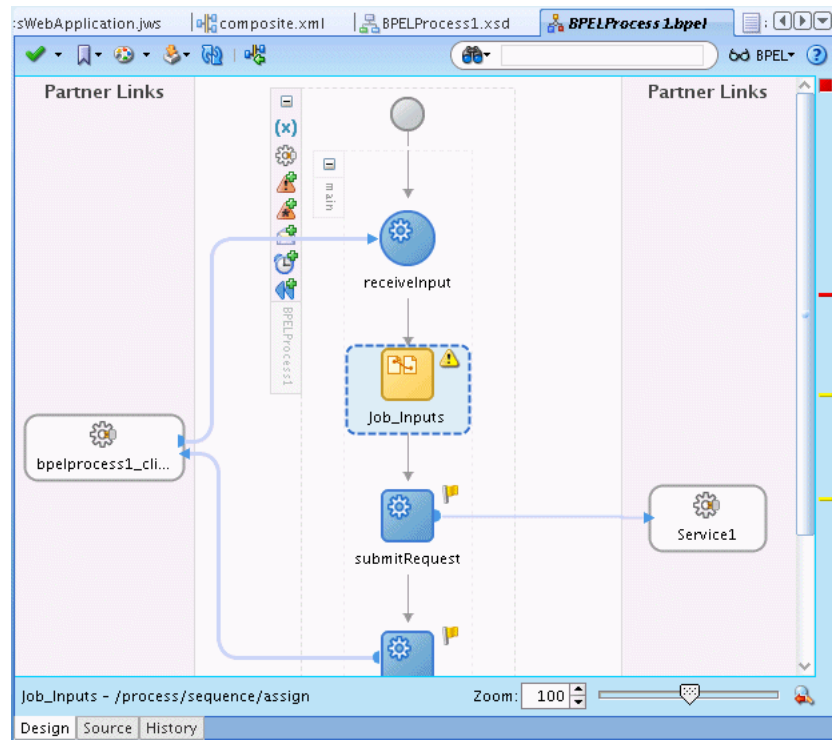
Table 11–2 (Cont.) Submit Request Web Service Arguments for BPEL Assign Activity

Argument	Description
JobDefinitionId	<ul style="list-style-type: none"> ▪ 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)	<p>dataType: Value type for this parameter (STRING, INTEGER, LONG, BOOLEAN, DATETIME)</p> <p>name: String containing the name of the parameter defined in the Oracle Enterprise Scheduler job definition.</p> <p>scope: String containing the named scope for this parameter - used only for job sets.</p> <p>value: Element containing the parameter's value</p>

To add an assign activity:

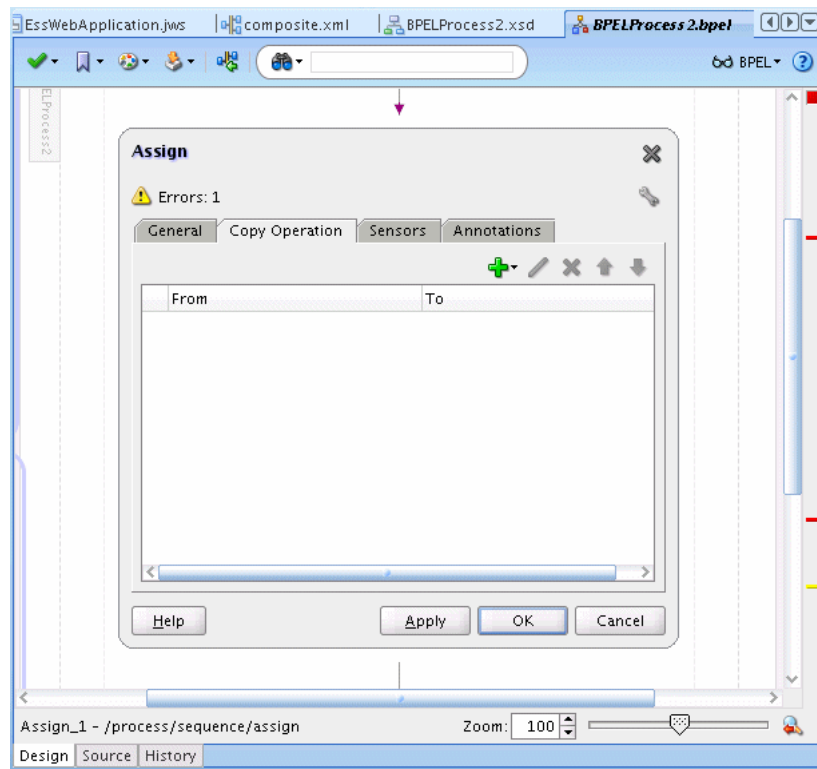
1. Drag-and-drop an Assign activity from the BPEL Activities area in the Component Palette to just before the Invoke Activity named **submitRequest**.
2. Select the Assign activity and double-click the name Assign_1 to enter new text. In the text entry box enter `Job_Inputs`, as shown in [Figure 11–11](#).

Figure 11–11 Adding an Assign Activity to BPEL

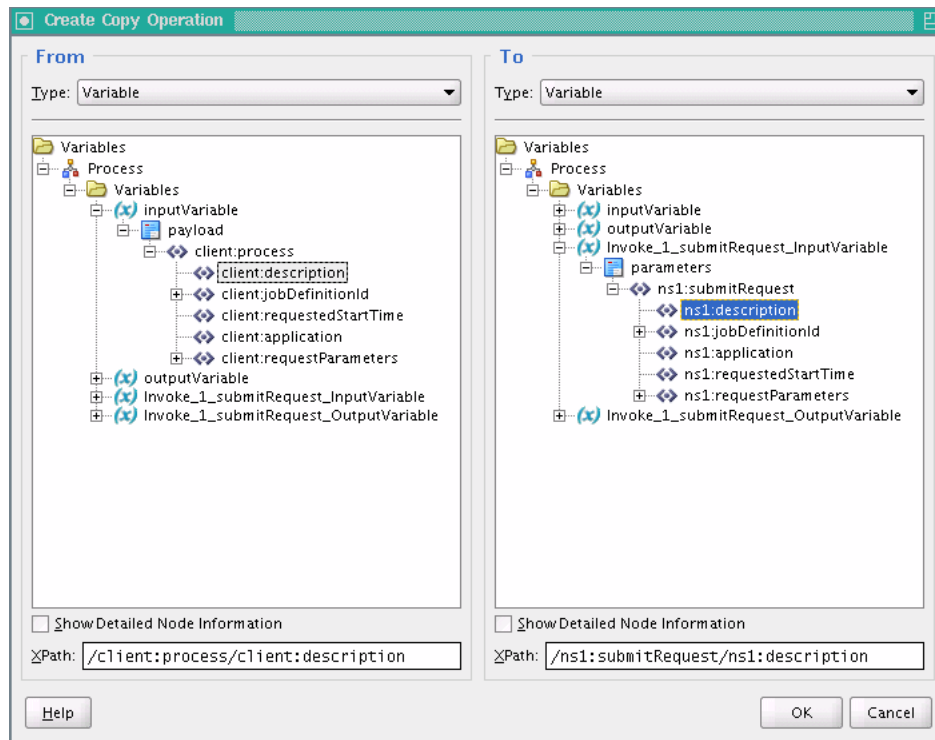


Add Copy for description JobDefinitionID requestedStartTime application:

1. Double click the new Assign activity named `Job_Inputs` to show the Assign page with the Copy Operation tab, as shown in [Figure 11–12](#).

Figure 11–12 Copy Operation for BPEL Assign Activity

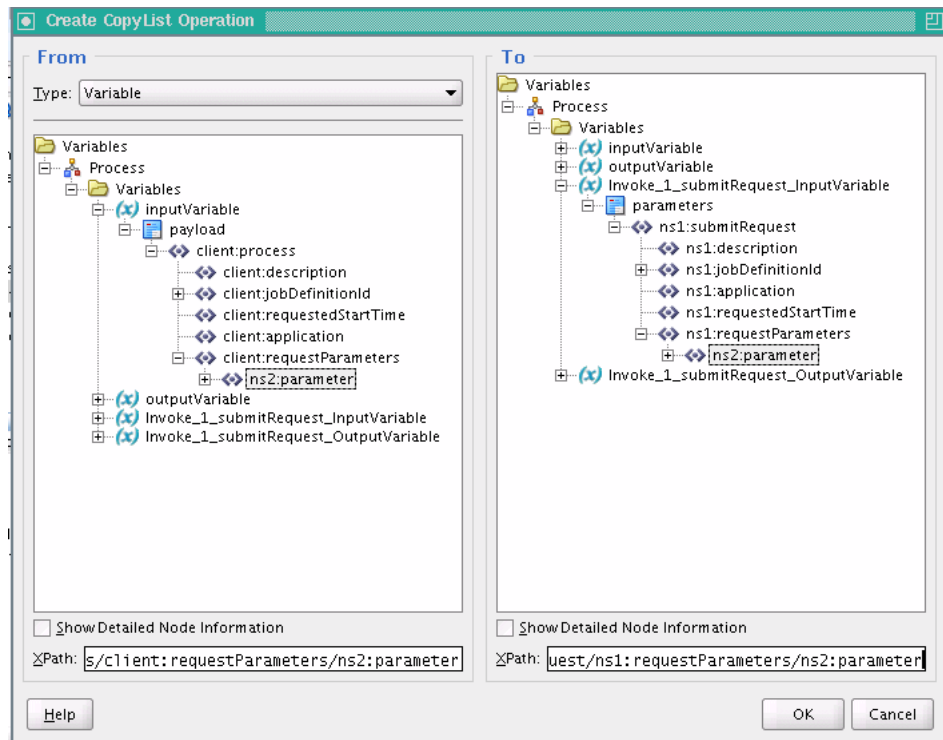
2. Click the **Add** icon and from the dropdown list select **Copy Operation**, to add copy operations for variables. This displays the Create Copy Operation dialog.
3. In the Create Copy Operation dialog, expand and then navigate to select a copy operation for each input parameter (you only use a copy operation for description, jobDefinitionID, requestedStartTime, and application). This copies the input parameters to Invoke_1_submitRequest_InputVariable parameters for the invoke activity. [Figure 11–13](#) shows one of these copy operations.

Figure 11–13 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:

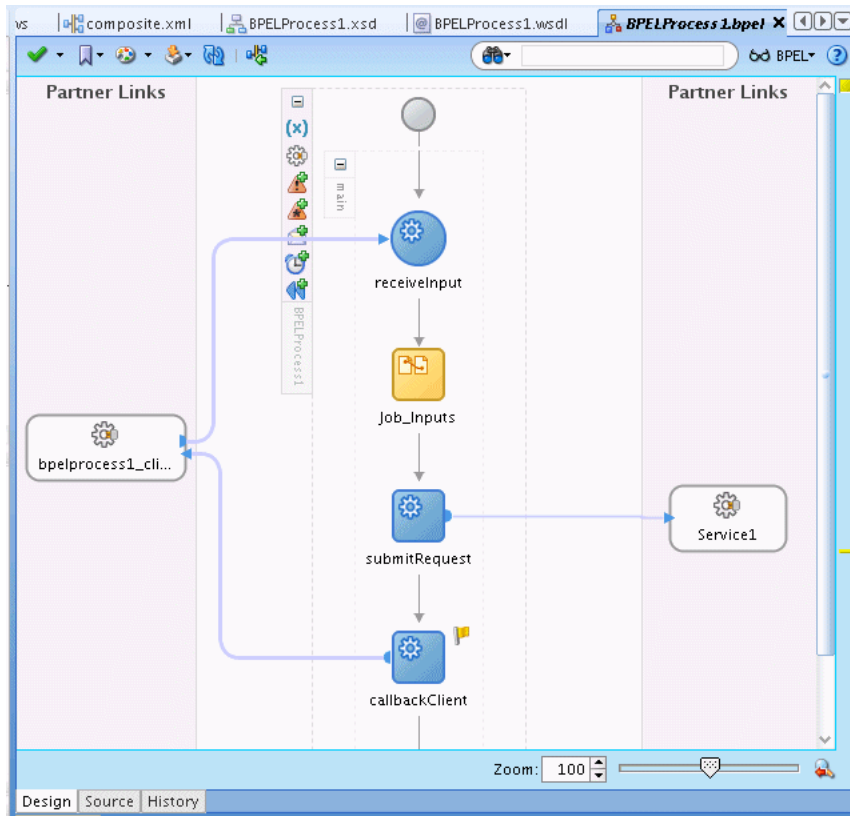
1. Double click the Assign activity named Job_Inputs to show the Assign page with the Copy Operation tab.
2. Click the **Add** icon and from the dropdown list select **CopyList Operation...**, to add CopyList operations for the requestParameters. This displays the Create CopyList Operation dialog.
3. 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 11–14](#).

Figure 11–14 CopyList Operation for Request Parameters

4. In the Create CopyList Operation dialog, click **OK**.
5. In the Assign activity, click **OK**.

Figure 11–15 shows the BPEL Design page.

Figure 11–15 BPEL with Job_Inputs Add Activity and submitRequest Invoke

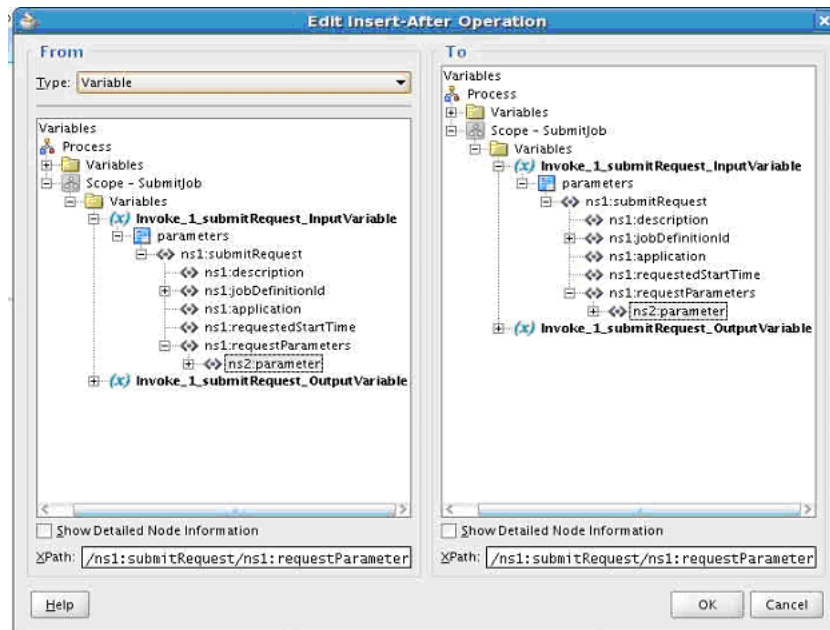


When BPEL Element Does Not Have Same Type as Oracle Enterprise Scheduler web service:

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 11–16](#).

Figure 11-16 Using Insert-After to Clone Parameters



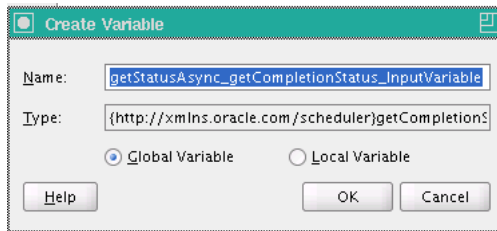
3. Populate the additional parameter elements using XPath array subscripting.
4. This action effectively copies the entire parameter element along with all sub-element values and appends it to the end of the XML array. In order to populate the values of the second job parameter, add additional copy operations 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.

11.7.5 Invoke the getCompletionStatus Operation

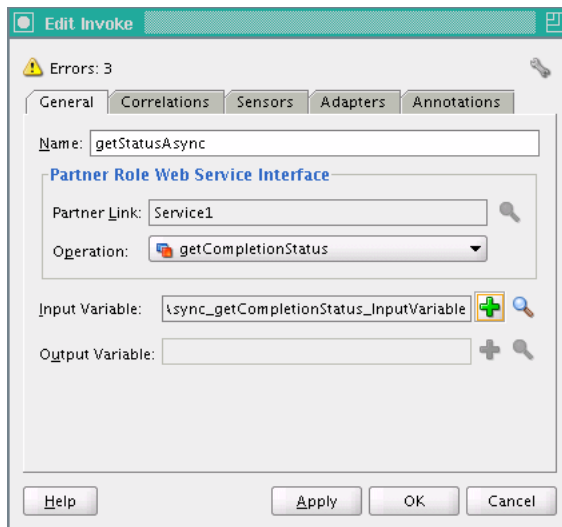
Add another Invoke activity and link it to Service1 to invoke the ESSWebService getCompletionStatus operation.

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 11-17.

Figure 11–17 Create Variable Window for getStatusAsync

6. In the Create Variable dialog, click **OK**. This displays the Edit Invoke dialog, as shown in [Figure 11–18](#).

Figure 11–18 Edit Invoke Window for getStatusAsync

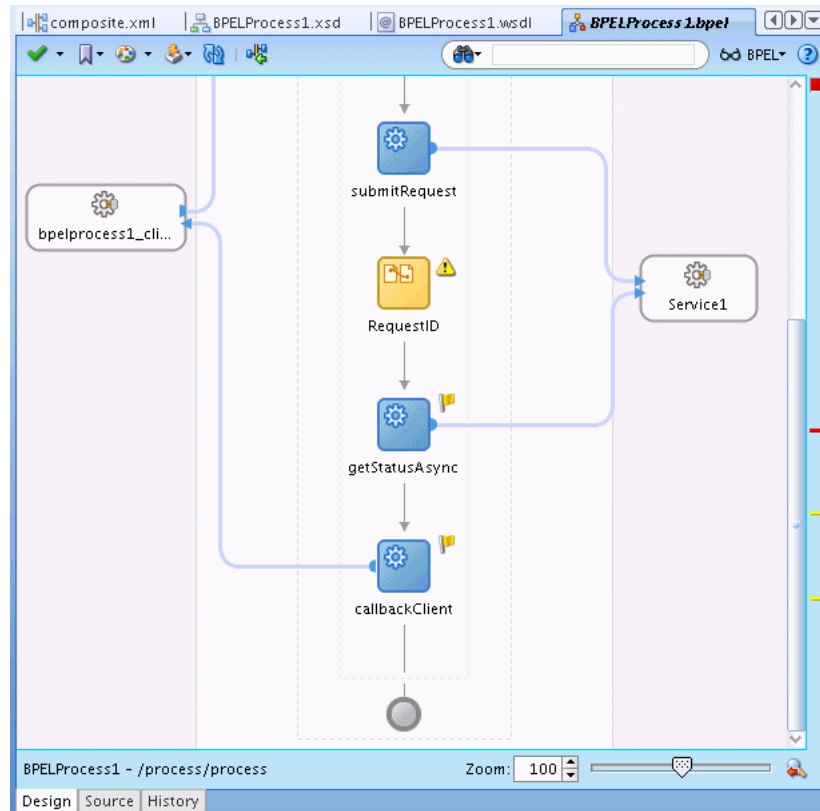
7. In the Edit Invoke dialog, click **OK**. This displays the new Invoke Activity `getStatusAsync` and the link to `Service1`.

11.7.6 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.

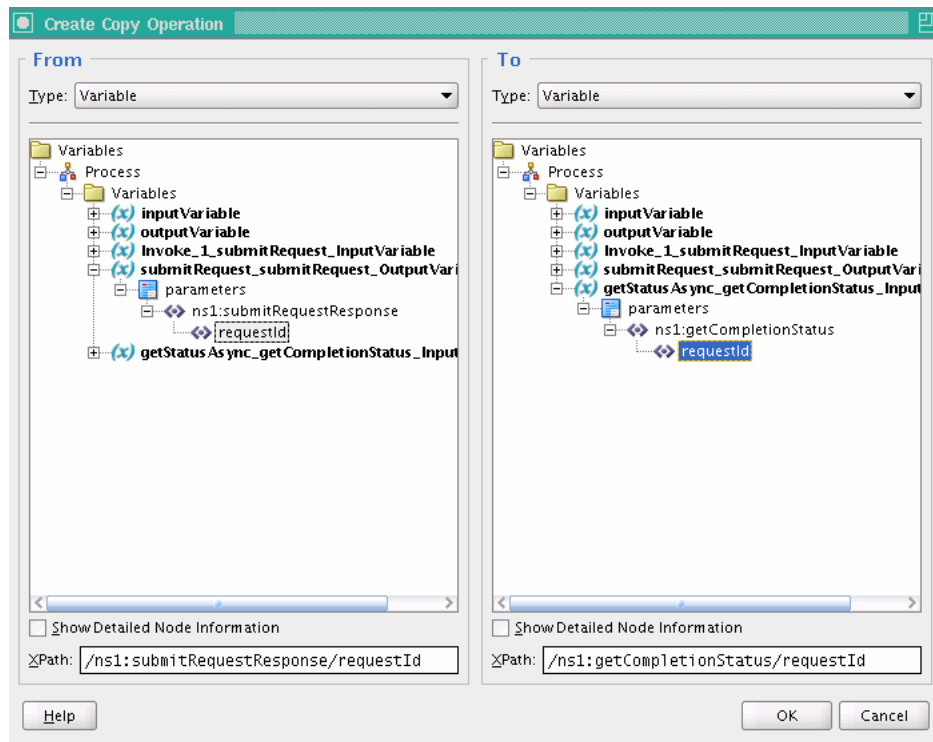
To add the assign activity:

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 11–19](#) shows the Assign activity.

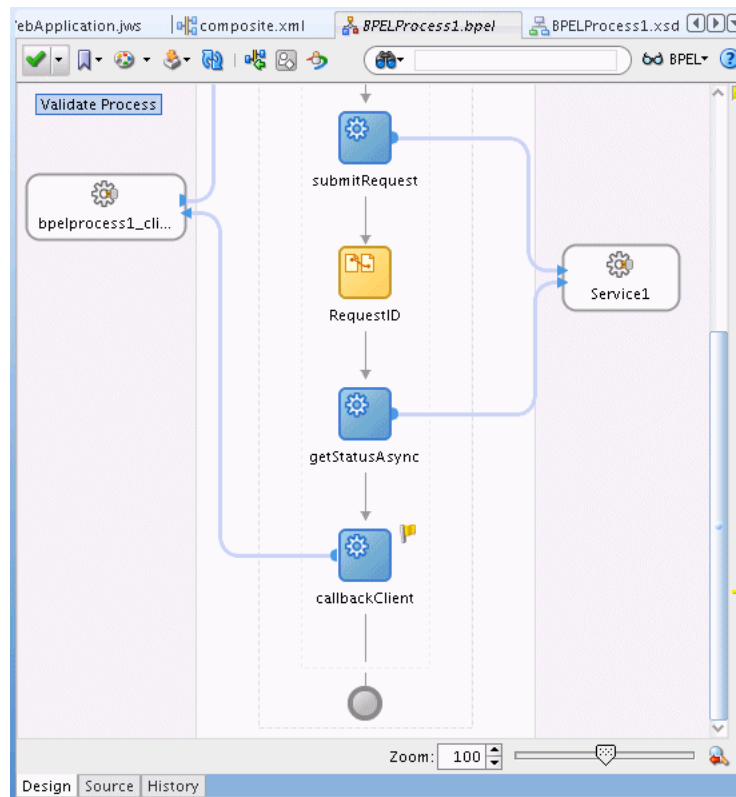
Figure 11–19 Adding RequestID Assign Activity

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 11–20](#).

Figure 11–20 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 11–21](#).

Figure 11–21 BPEL with Request ID Assign Activity Added

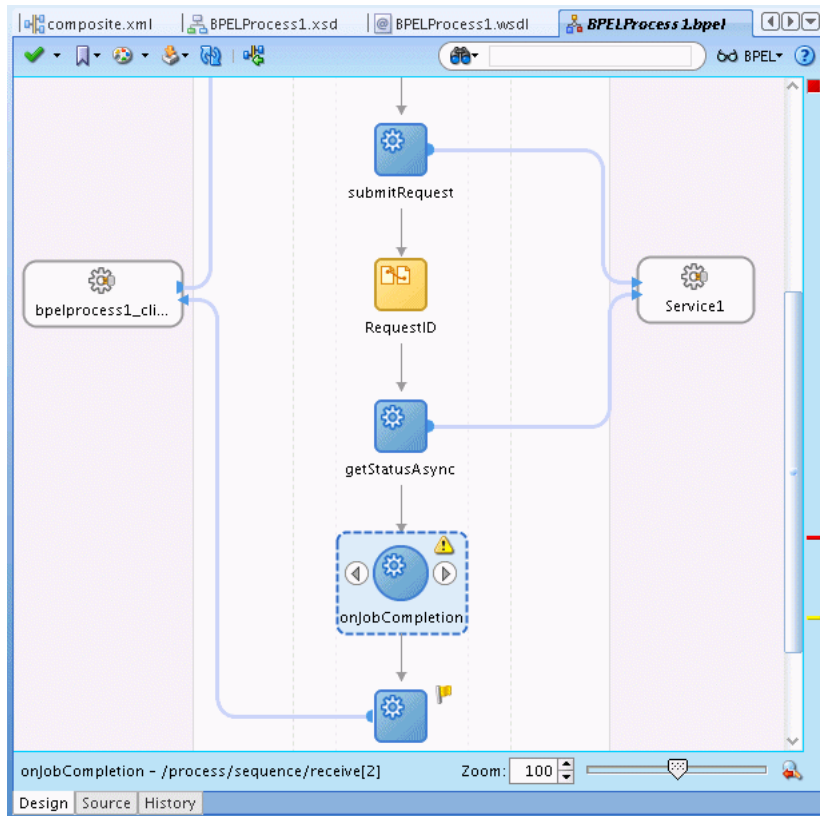
11.7.7 Receive the Job Completion Status

Add a Receive Activity and link it to the onJobCompletion ESSWebService operation.

Add a receive activity:

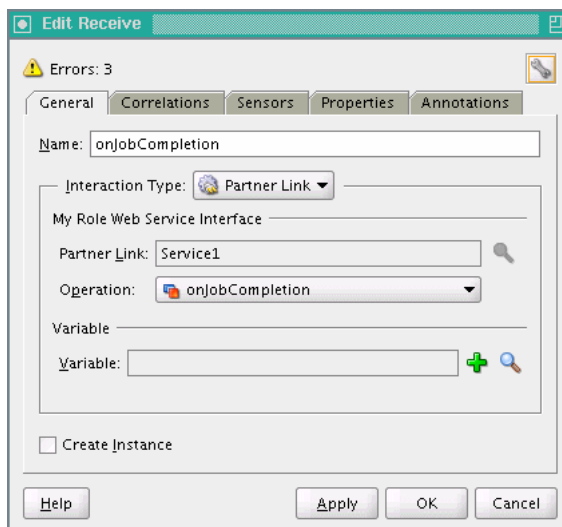
1. Drag-and-drop a Receive activity from the BPEL Activities area in the Component Palette to a position after the `getStatusAsync` Invoke activity and before the `callbackClient`.
2. Select the text entry area in the Receive Activity named `Receive_1` and enter `onJobCompletion`, as shown in [Figure 11–22](#).

Figure 11–22 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 11–23](#).

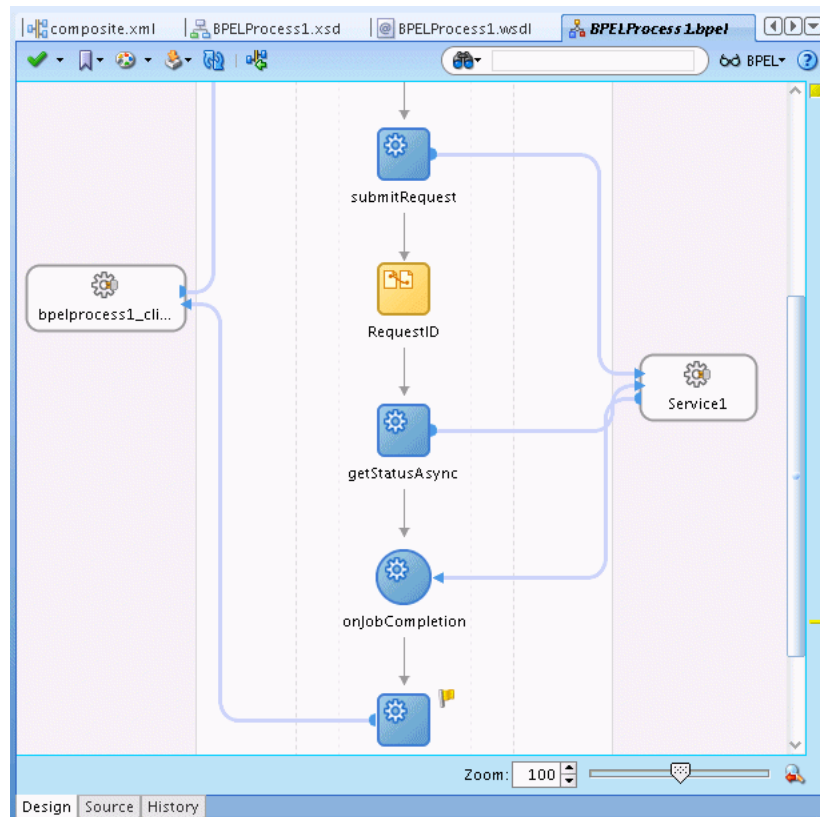
Figure 11–23 Edit Receive Window for onJobCompletion Receive Activity



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 11–24](#).

Figure 11–24 Adding the onJobCompletion Receive Activity



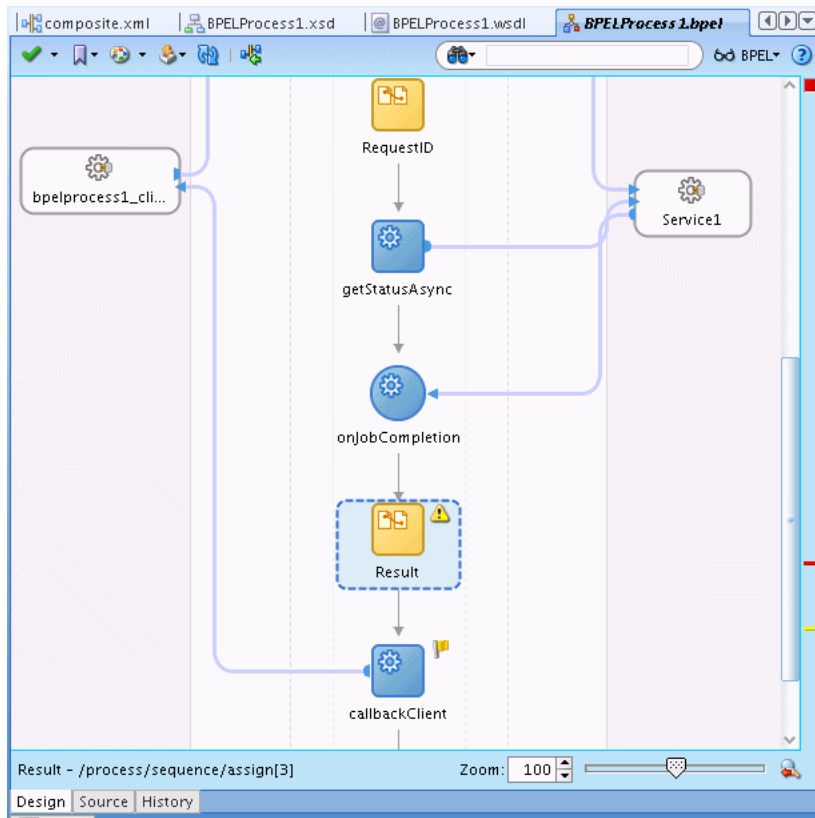
11.7.8 Return Result to Client

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.

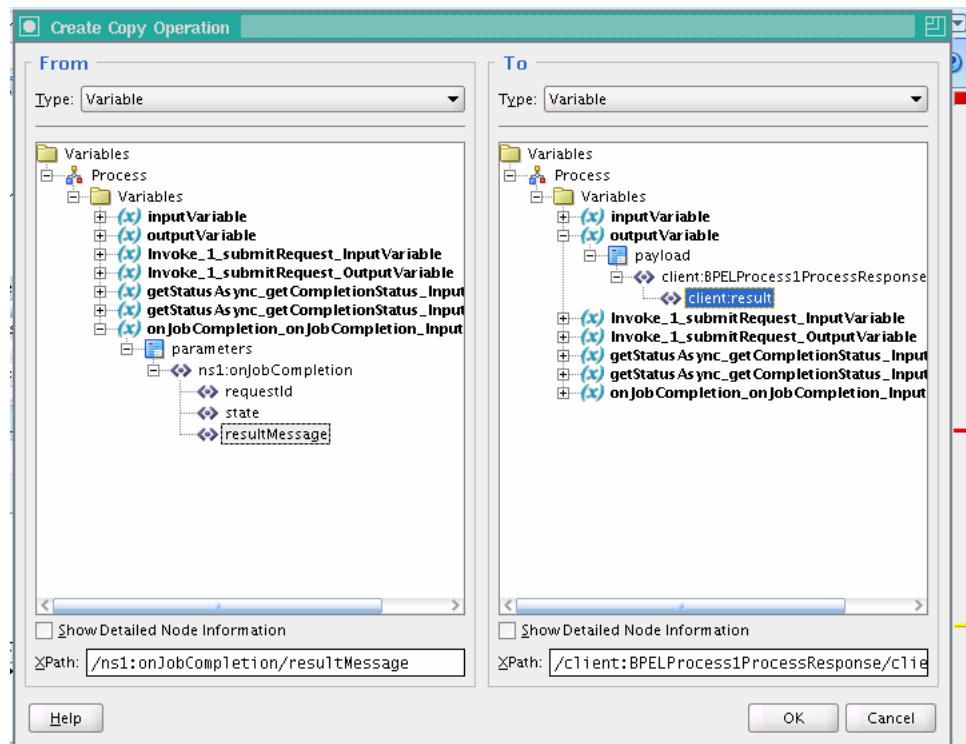
To add the result assign activity:

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 11–25](#).

Figure 11–25 Adding Assign Activity for Output to Client

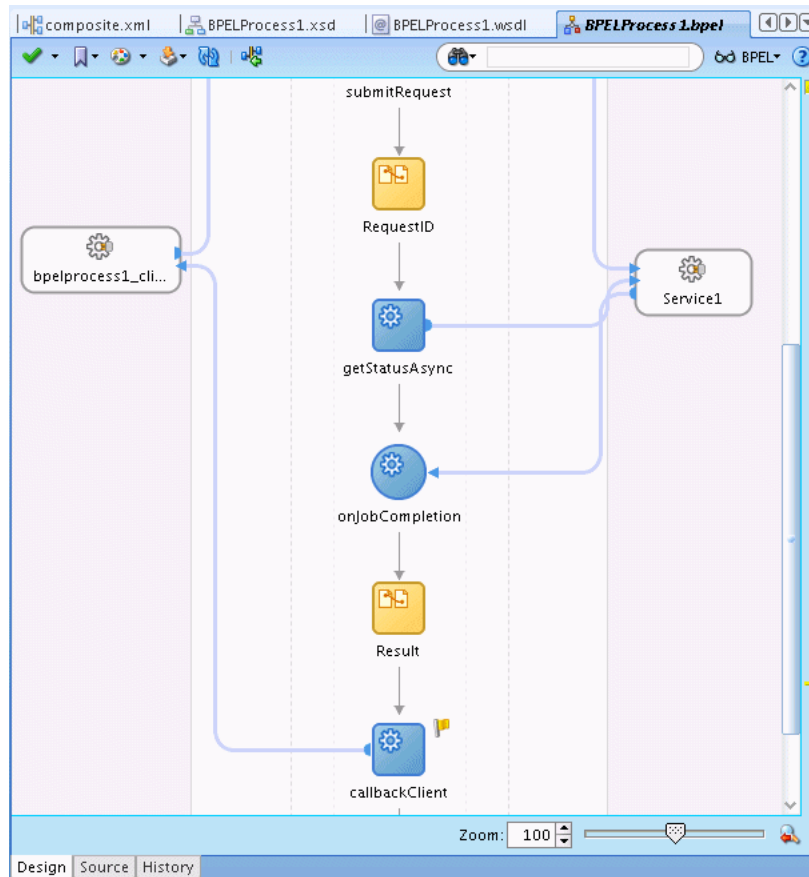


3. Double click the new **Result** Assign activity to show the Assign page with the Copy Operation tab.
4. Click the **Add** icon and select **Copy Operation...** from the dropdown list.
5. Navigate to select the variables, for the **From** area for `onJobCompletion_onJobCompletion_InputVariable` and select **resultMessage**. In the **To** area, expand **outputVariable** and select **client:result**, shown in Figure 11–26.

Figure 11–26 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 11–27](#).

Figure 11–27 Result Assign Activity with callbackClient Invoke Activity

11.8 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 11.8.1, "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 11.8.2, "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 11.8.3, "How to Invoke the ESSWebService addPPActions Operation"](#)

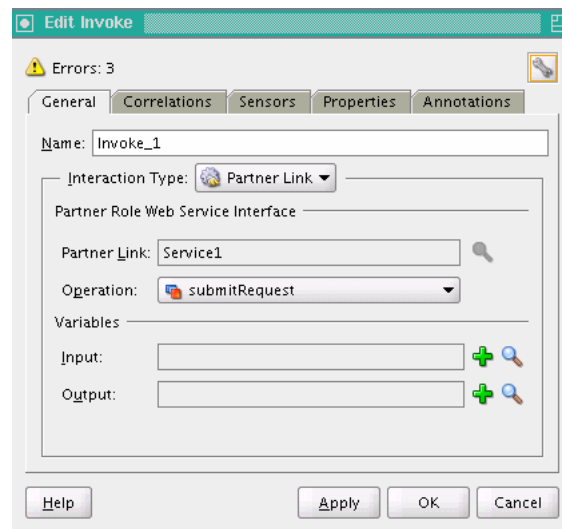
11.8.1 How to Invoke the ESSWebService submitRecurringRequest Operation

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 11-28](#).

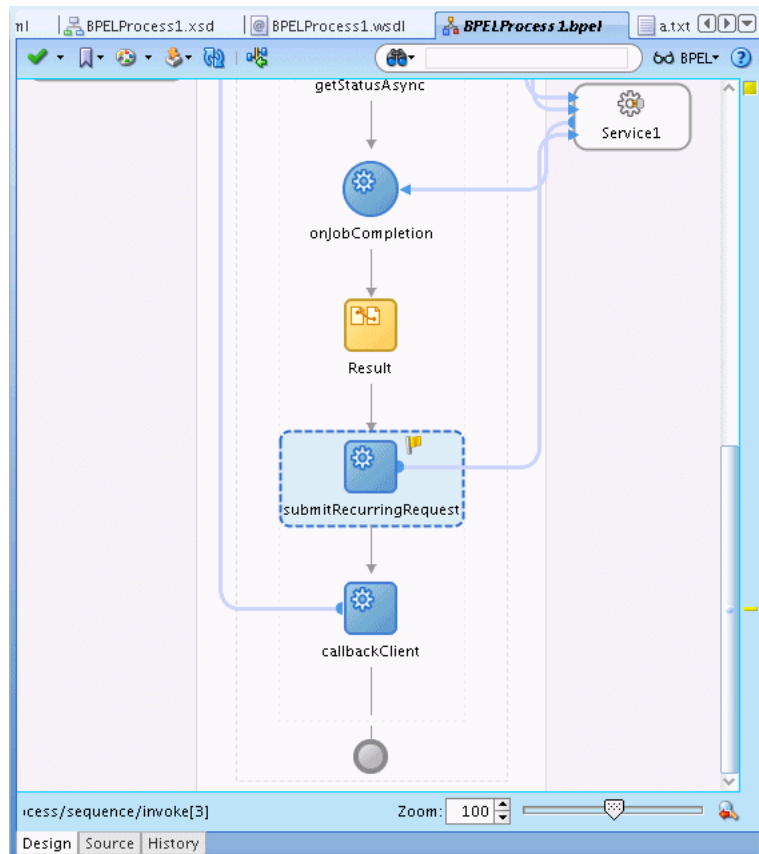
Figure 11-28 Edit Invoke Window for BPEL Activity



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 11-29](#).

Figure 11-29 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.

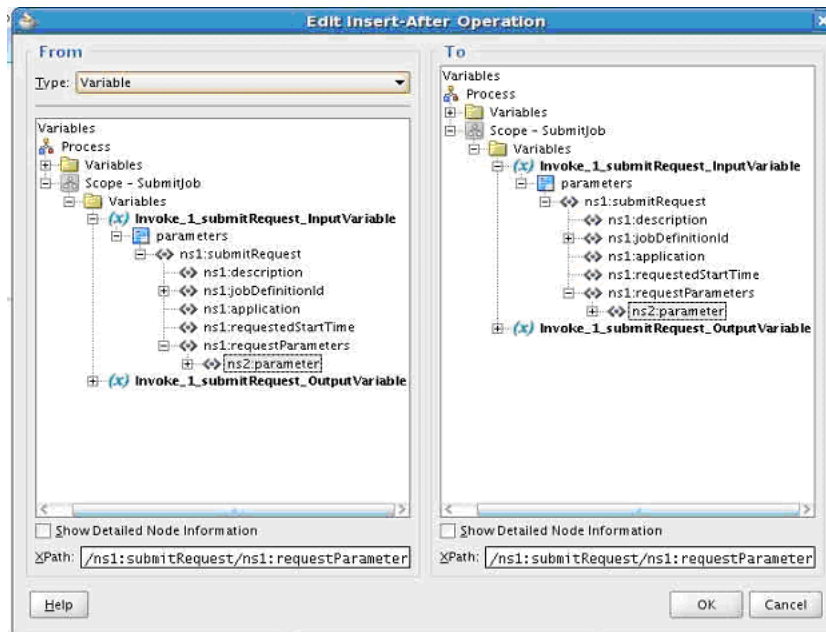
For the mapping for an Assign activity with a Copy operation, the arguments correspond to the input parameters for Oracle Enterprise Scheduler submitRequest, as shown in [Table 11-3](#). If your BPEL schema differs from the submitRequest message type, use [Table 11-3](#) as a guide for how to populate the values manually with the Assign activity Copy operation.

Table 11–3 *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	<ul style="list-style-type: none"> ▪ 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)	<p>dataType: Value type for this parameter (STRING, INTEGER, LONG, BOOLEAN, DATETIME)</p> <p>name: String containing the name of the parameter defined in the Oracle Enterprise Scheduler job definition.</p> <p>scope: String containing the named scope for this parameter - used only for job sets.</p> <p>value: Element containing the parameter's value</p>
scheduleID	<ul style="list-style-type: none"> ▪ 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 11–30](#).

Figure 11–30 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.

11.8.2 How to Invoke the ESSWebService setSubmitArgs Operation

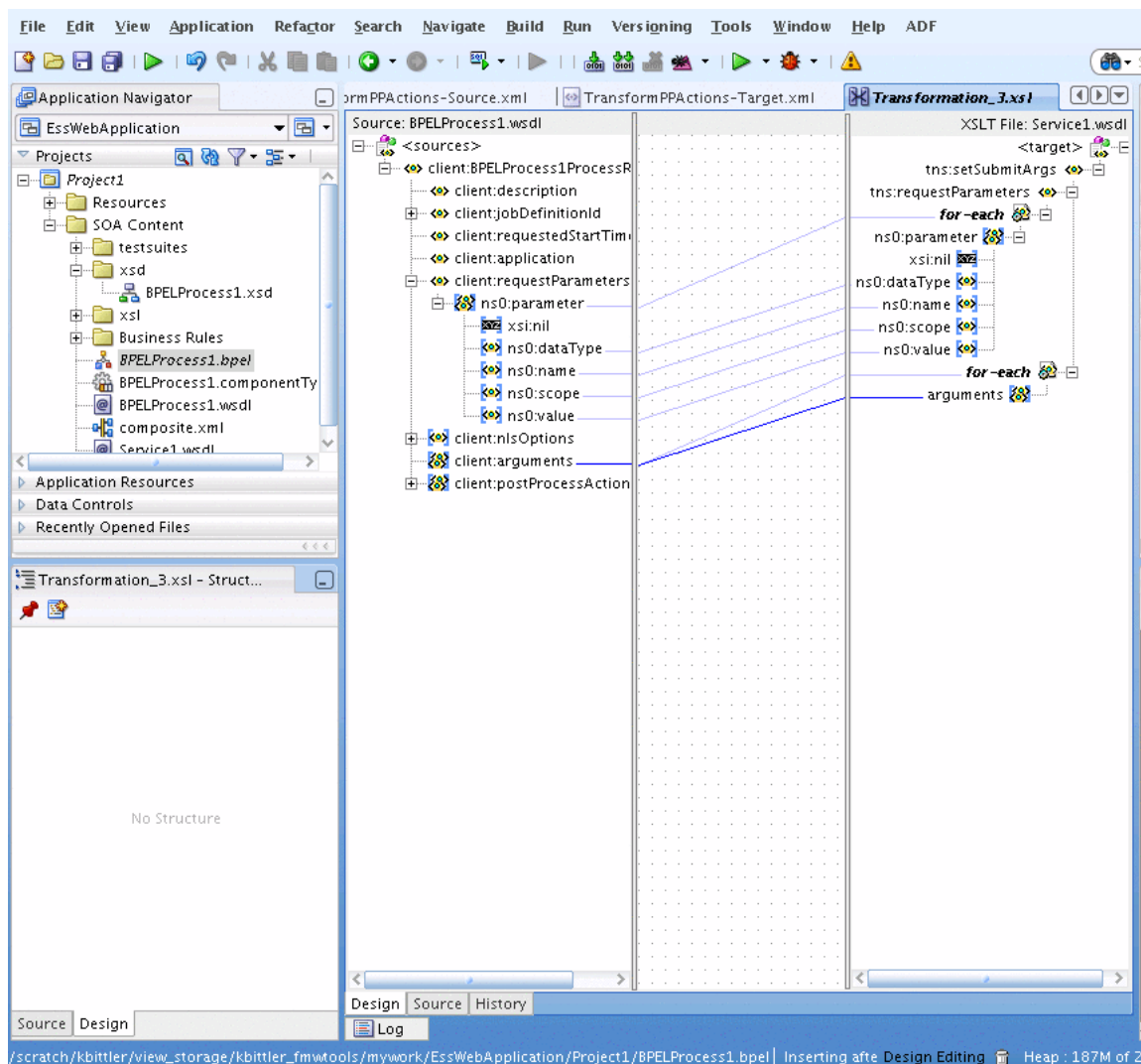
In the BPEL process you add an invoke activity to perform the Oracle Enterprise Scheduler web service `setSubmitArgs()` operation.

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 11–31](#). Note that a "for-each" construct can be inserted by dragging an item from the XSLT Constructs area of the Component Palette.

Figure 11–31 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:BPPELProcess1ProcessRequest/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:BPPELProcess1ProcessRequest/client:arguments">
  <tns:arguments>
    <xsl:value-of select="."/>
  </tns:arguments>
</xsl:for-each>
```

Example 11–2 shows the complete transformation source file.

Example 11–2 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="../BPPELProcess1.wsdl"/>
      <rootElement name="BPPELProcess1ProcessRequest"
        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/extension"
xmlns:socket="http://www.oracle.com/XSL/Transform/
        java/oracle.tip.adapter.socket.ProtocolTranslator"
xmlns:ldap="http://schemas.oracle.com/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>

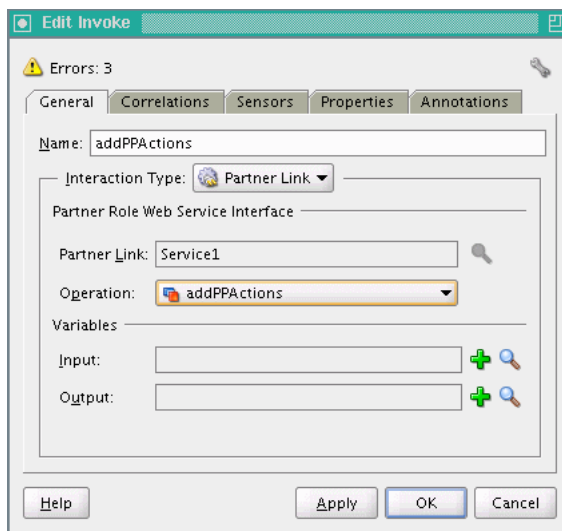
```

11.8.3 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 11–32](#).

Figure 11–32 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 11-3](#).

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 11-3 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:BPPELProcess1ProcessRequest/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:BPPELProcess1ProcessRequest/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>

```


11.8.4 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 11–4](#), you can add the following to the `BPELProcess1.xsd` file to allow input for `setStepsArgs`.

Example 11–4 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.

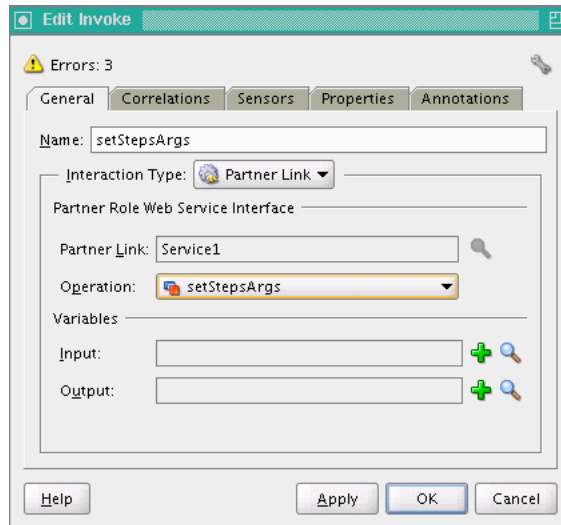
From BPEL Activities and Components, select **Transform** and place before `setStepsArgs`. Open the new transformation activity. Select **inputVariable** as the source and **setStepsArgs_setStepsArgs_InputVariable** as the target. Create a new mapper file. Create the mappings as shown in the `SetStepsArgs` 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.

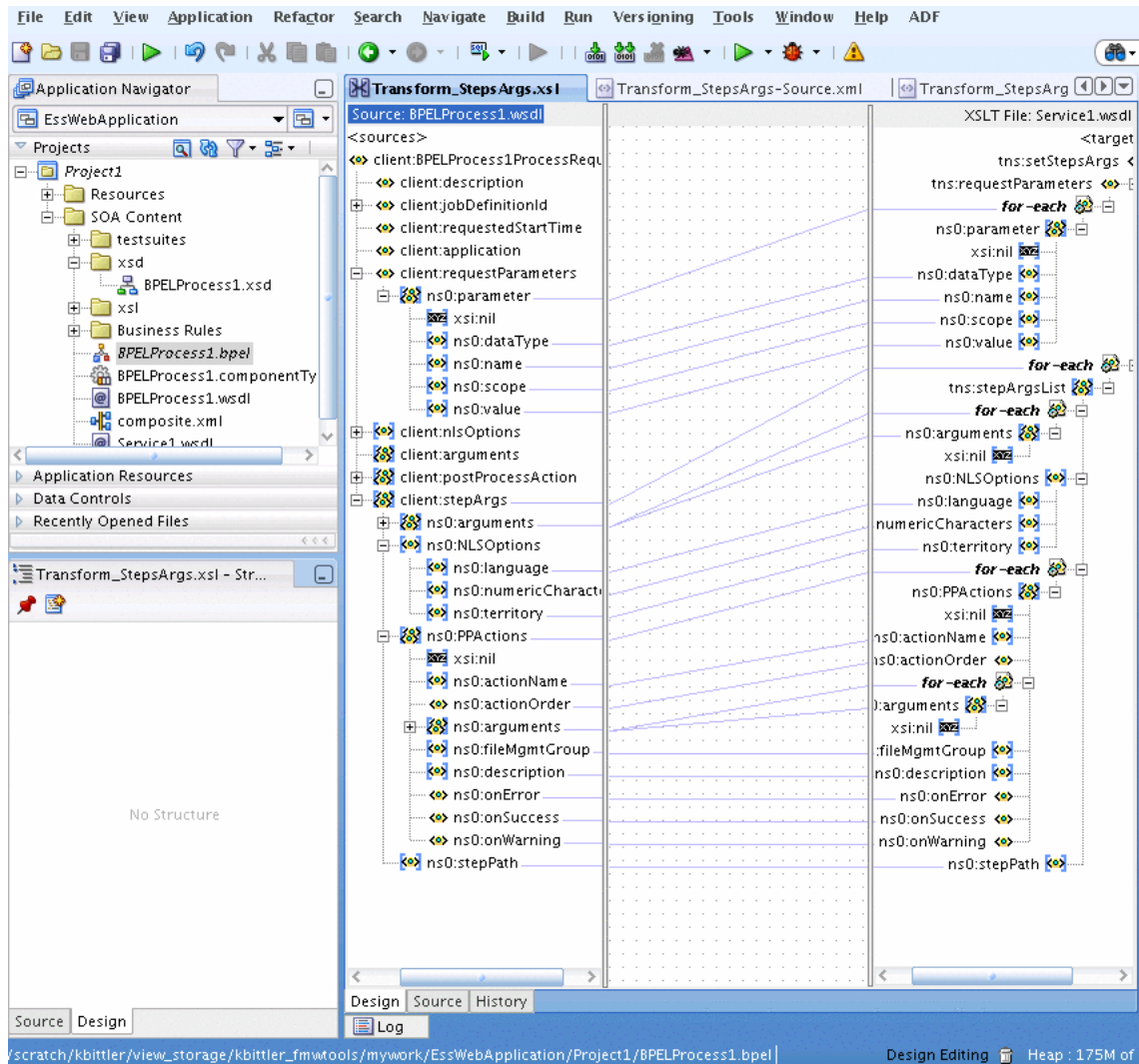
To add the Invoke activity use setStepsArgs operation:

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 11–33](#).

Figure 11–33 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 11–34](#) using the mappings shown in [Example 11–5](#).

Figure 11–34 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. The 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 11–5 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:BPPELProcess1ProcessRequest/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>
```

11.9 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 Oracle Web Services*.

11.9.1 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*.

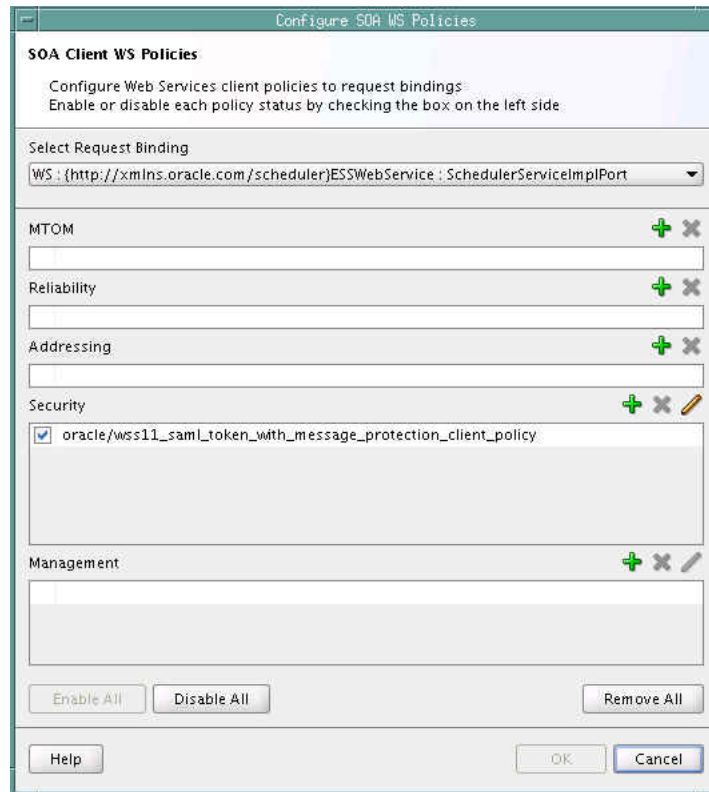
To secure the Oracle Enterprise Scheduler web service:

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 11-35](#), and click **OK**.

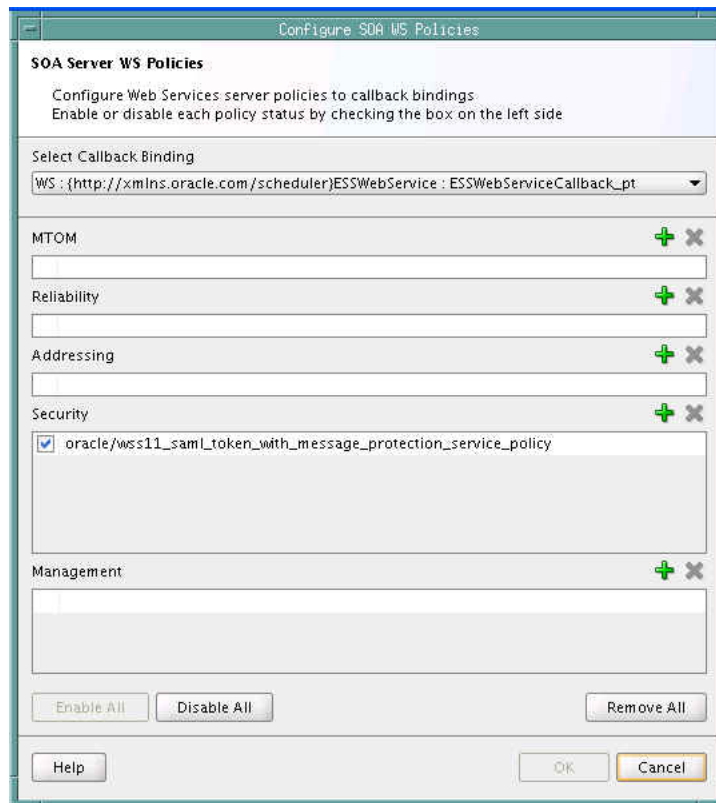
Figure 11–35 Client Security Policy for the Oracle Enterprise Scheduler Web Service

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 11–36](#), and click **OK**.

Figure 11-36 Callback Security Policy for the Oracle Enterprise Scheduler Web Service

6. Save your changes to the SOA composite file.

11.9.2 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.

11.10 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.

11.10.1 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.

Defining and Using Job Sets

This chapter describes how to define and submit an Oracle Enterprise Scheduler job set, a collection of job definitions that can be grouped together to run as a single unit.

This chapter includes the following sections:

- [Section 12.1, "Introduction to Defining and Using Job Sets"](#)
- [Section 12.2, "Defining Job Sets"](#)
- [Section 12.3, "Cross Application Job Sets"](#)
- [Section 12.4, "Using Input and Output Forwarding"](#)

12.1 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 sets*. 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 12-1](#) shows properties that are useful for job set steps. Any property can be defined on a job set step.

Table 12–1 Job Set Step Properties

Property	Description
EFFECTIVE_APPLICATION	<p>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.</p> <p>This property can be defined for job definitions and job types as well as job sets.</p>
SELECT_STATE	<p>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.</p> <p>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 <code>SELECT_STATE</code> to false for that step. To allow the state of a job set step to affect the overall state of the job set, set <code>SELECT_STATE</code> to true for that step.</p>

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 12.3, "Cross Application Job Sets"](#).

12.2 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 application defined properties 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.

12.2.1 How to Define a Job Set

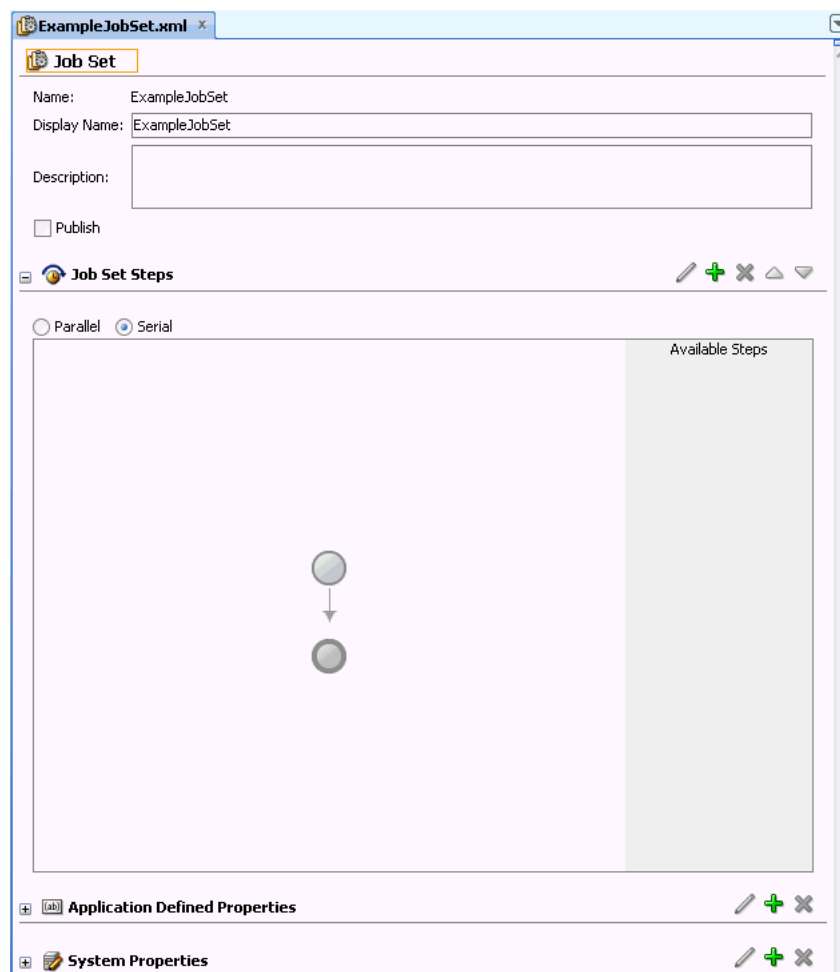
An Oracle Enterprise Scheduler job set is defined by a name, a package, a job set execution mode, step definitions, application defined properties, and system properties.

To create a job set:

1. In Oracle JDeveloper, right-click in the project to view the New Gallery.
2. In the **All Technologies** tab, 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 12-1](#).

Figure 12-1 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 12.2.2, "How to Define Serial Job Set Steps"](#) or [Section 12.2.3, "How to Define Parallel Job Set Steps"](#).
8. In the Application Defined Properties area, click **Add** to add properties associated with the job set. You use these to represent an application-specific or step-specific application defined property for the job set. For more information on using

application defined properties, see [Section 7.1, "Introduction to Using Parameters and System Properties"](#). For more information, see [Section 7.1.2.2, "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 7.4, "Using System Properties"](#).
10. In the Access Control area, click **Add** to modify the list of roles that have access to this metadata, along with their access levels. For more information on defining access, see [Section 18.2.3, "How to Create Grants with Oracle Enterprise Scheduler Metadata Pages"](#).
11. In the Localization area, enter the following information for localizing this job set:
 - Resource Bundle Base Name -- The base name for the resource bundle that specifies internationalized values.
 - Display Name Resource Key -- The resource key that specifies the display name value in the resource bundle.
 - Description Resource Key -- The resource key that specifies the description text in the resource bundle.
12. Save the job set.

12.2.2 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 12–2](#) lists the possible terminal states that you can specify using JDeveloper.

Table 12–2 Job Set Serial Execution Step Terminal States

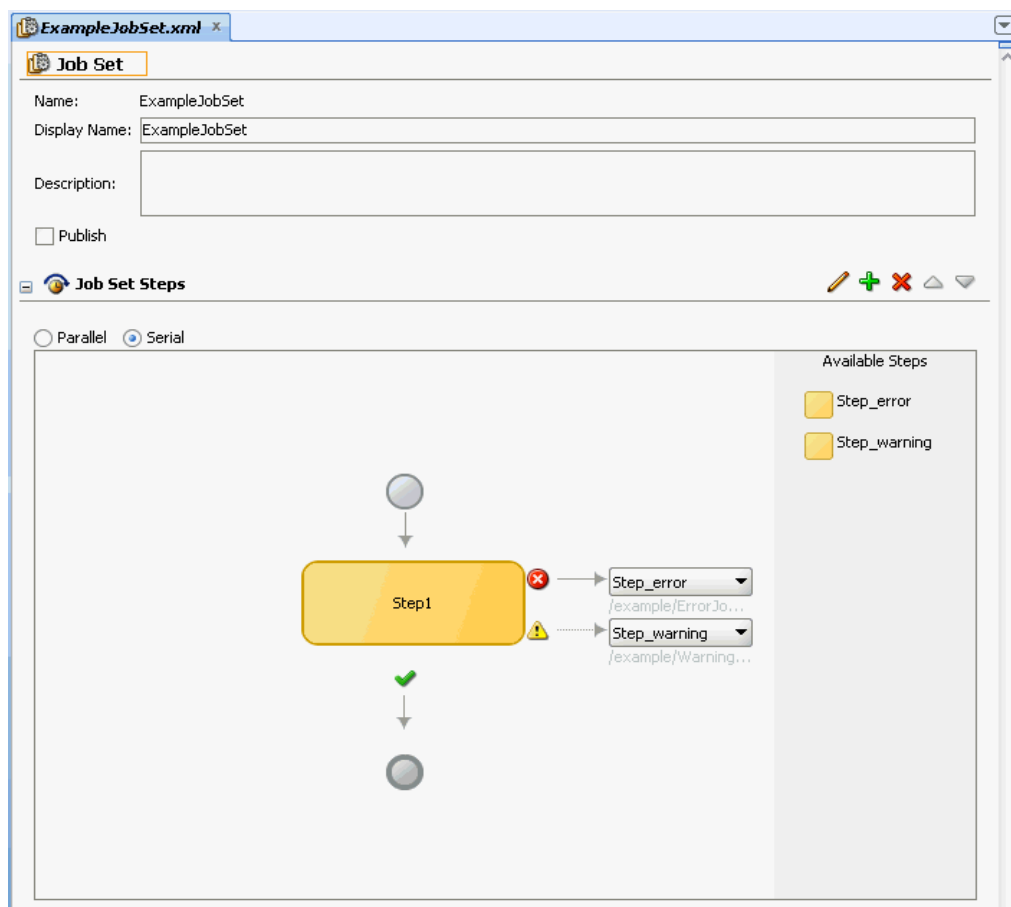
Terminal State	Description
SUCCEEDED	Oracle JDeveloper indicates this state with a checkmark icon. This path represents a child step or child job set was successfully processed by the system.
WARNING	Oracle JDeveloper indicates this step with a warning icon. A child step or child job set resulted in 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 application defined properties, then select the Application Defined Properties tab and add properties for the step.

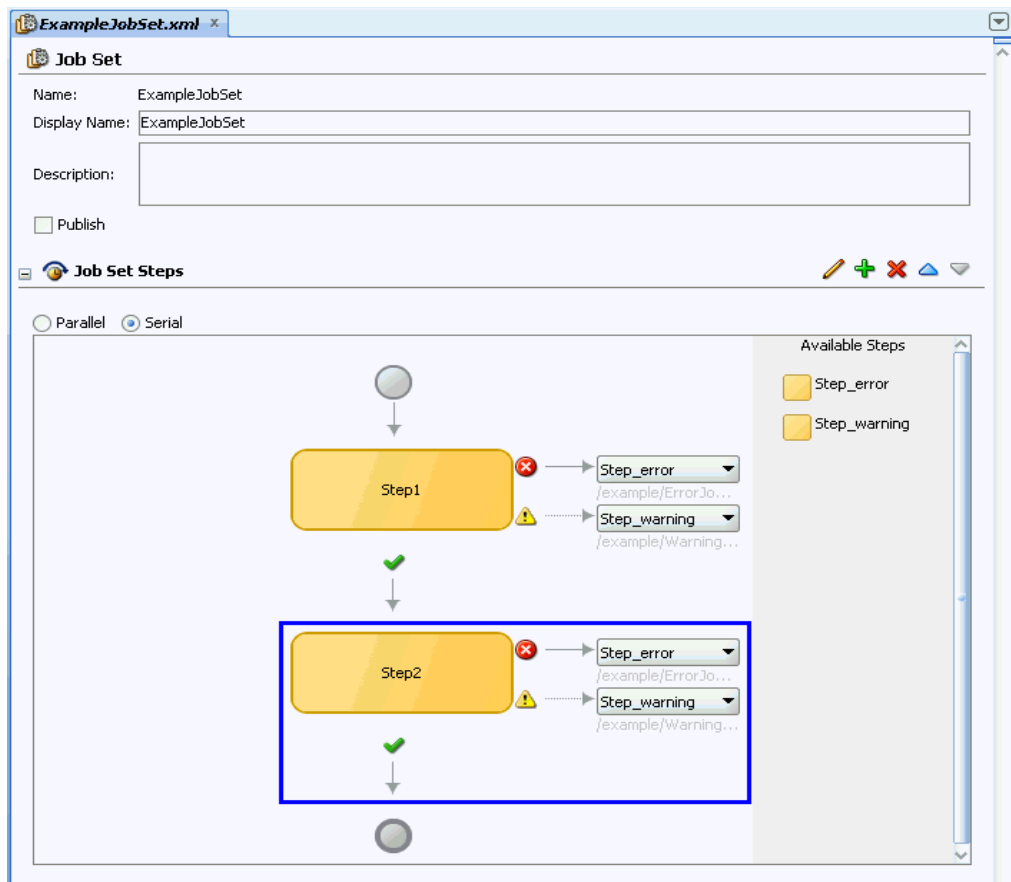
7. If you need to define step level system properties, then select the System Properties tab and add job set step system properties for the step.
8. Select a destination for the step. The step can be added as part of the job set by selecting **Insert into main diagram**. To make the step available for 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 12-2](#).

Figure 12-2 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 12-3](#).

Figure 12–3 Job Set with Two Steps Added



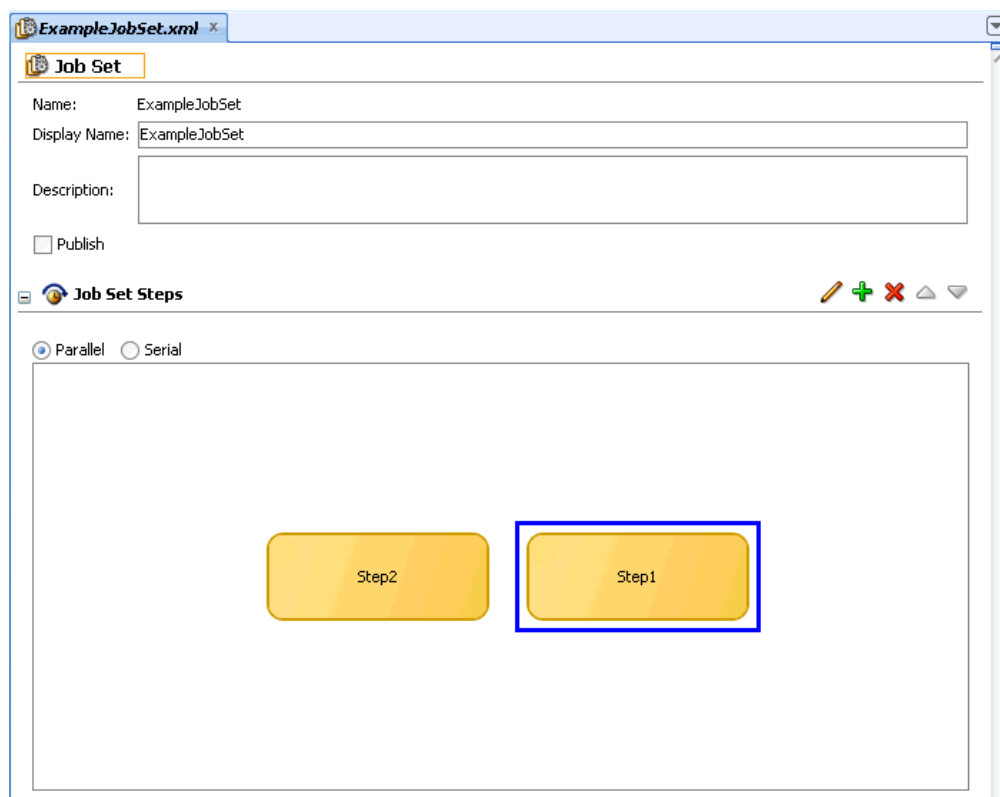
12.2.3 How to Define Parallel Job Set Steps

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 application defined properties, then select the Application Defined Properties tab and add properties 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 12–4](#).

Figure 12–4 Adding Job Set Steps to a Parallel Job Set



12.2.4 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.

12.2.5 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 create a 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.

12.2.6 What You Need to Know About Job Set Application Defined Properties and System Properties

There are cases where job set application defined properties or system properties may conflict with application defined properties or system properties set either in metadata or when a job request is submitted. For more information on how job set application defined properties and system properties are handled, see [Section 7.2, "Using Parameters with the Metadata Service"](#) and [Section 7.3, "Using Parameters with the Runtime Service"](#).

12.2.7 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 12–2](#). 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 12–3 shows the possible job set terminal states with the level indicated in the Priority column.

Table 12–3 Job Set Terminal State Transitions

Terminal State	Description	Priority
ERROR	<p>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.</p> <p>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.</p>	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	<p>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.</p> <p>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.</p> <p>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.</p>	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 12–4 lists additional possible states for a job set:

Table 12–4 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 12–4 (Cont.) Possible Job Set Runtime States

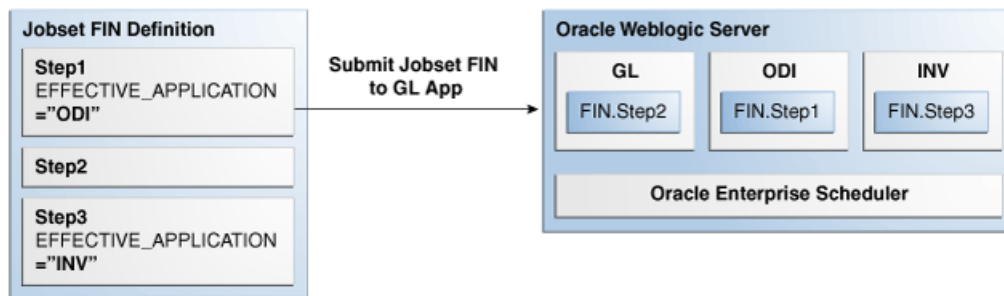
State	Description
CANCELLING	<p>A job set transitions to CANCELLING when the user requests a cancellation for the entire job set. This can be done by calling <code>cancelRequest()</code> with the request ID of the parent request representing the job set. Passing the parent request ID indicates that the user wants to cancel entire job set irrespective of its current, non-terminal, state and the states of its child requests.</p> <p>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.</p> <p>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.</p> <p>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.</p>
COMPLETED	<p>This state indicates that the job set or job set step has finished executing and post-processing will begin.</p>
BLOCKED	<p>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).</p> <p>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.</p> <p>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.</p> <p>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.</p>
HOLD	<p>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.</p>

12.3 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 12–5](#):

- 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 12–5 Cross Application Job Set Steps



12.3.1 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.

12.3.2 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.

12.4 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 `ess-config.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.

12.4.1 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.

Defining and Using a Job Incompatibility

This chapter describes how to use an Oracle Enterprise Scheduler job incompatibility, with which you can specify job requests that cannot run together.

This chapter includes the following sections:

- [Section 13.1, "Introduction to Using a Job Incompatibility"](#)
- [Section 13.2, "Defining Incompatibility with Oracle JDeveloper"](#)
- [Section 13.3, "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 5, "Use Case Oracle Enterprise Scheduler Sample Application \(Deprecated\)"](#), and [Chapter 8, "Creating and Using PL/SQL Jobs"](#), and [Chapter 9, "Creating and Using Process Jobs"](#). For more information on using job sets, see [Chapter 12, "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.

13.1 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 `parameterVO` 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.

13.1.1 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.

13.2 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

13.2.1 How to Define a Global Incompatibility

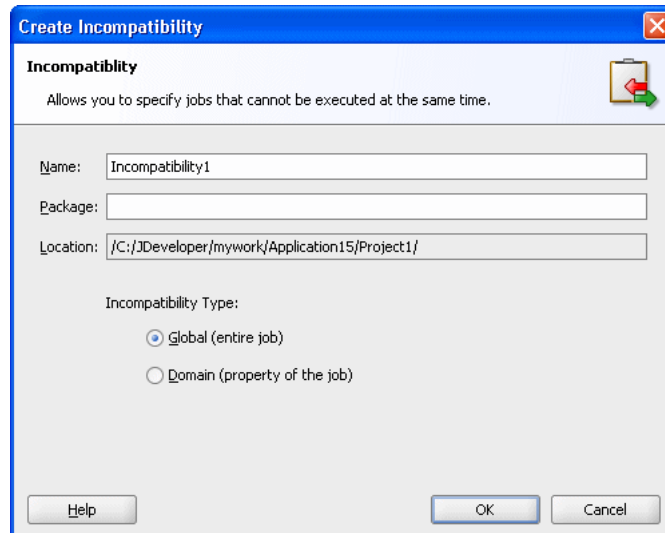
An Oracle Enterprise Scheduler global incompatibility is defined by a name, a package and entities.

To create a global incompatibility:

1. In Oracle JDeveloper, right-click in the project to view the New Gallery.

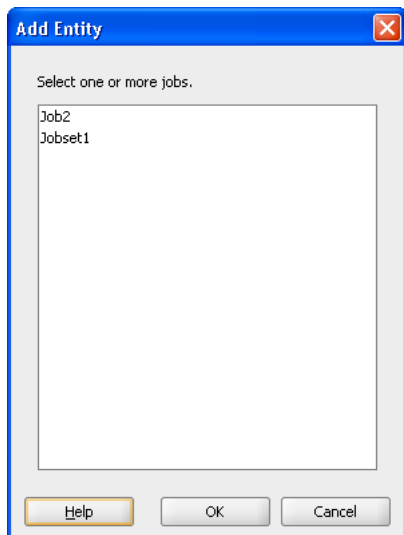
2. Under **Categories**, expand **Business Tier** and select **Enterprise Scheduler Metadata**.
3. Under **Items**, select **Incompatibility** and click **OK**. This displays the Create Incompatibility window, as shown in [Figure 13–1](#).

Figure 13–1 Create Incompatibility Window



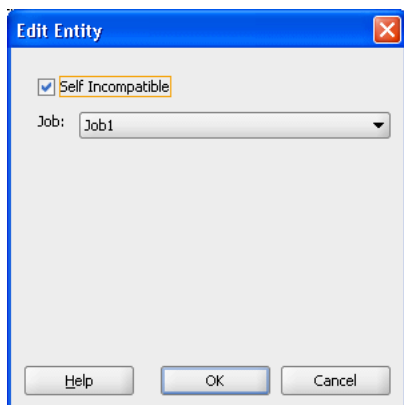
4. Use the Create Incompatibility dialog to specify the following:
 - a. In the **Name** field, enter a name for the incompatibility or accept the default name.
 - b. In the **Package** field, enter a package name for the incompatibility.
 - c. The **Location** field displays the full path of the directory 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 Entities area, click **Add** to add entities. This displays the Add Entity dialog, as shown in [Figure 13–2](#).

Figure 13–2 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 13–3](#).

Figure 13–3 Edit Entity Window for Global Incompatibility



9. To specify self incompatibility, select **Self Incompatible**.
10. Save the incompatibility.

13.2.2 How to Define a Domain Incompatibility

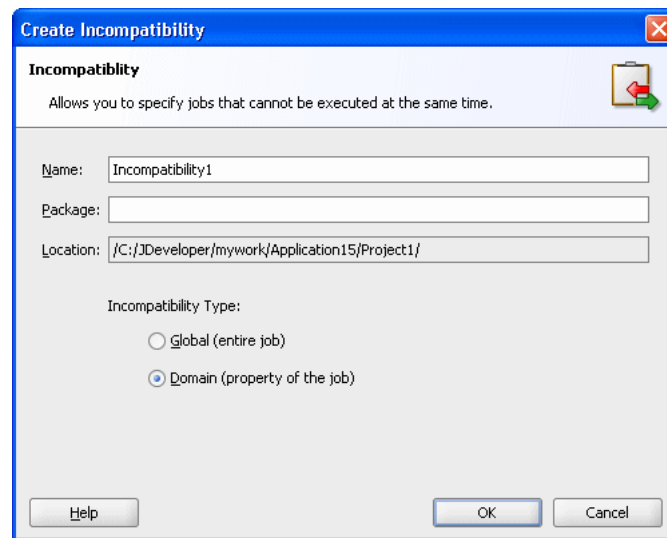
An Oracle Enterprise Scheduler domain incompatibility is defined by a name, a package, entities, and properties for each entity.

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**.

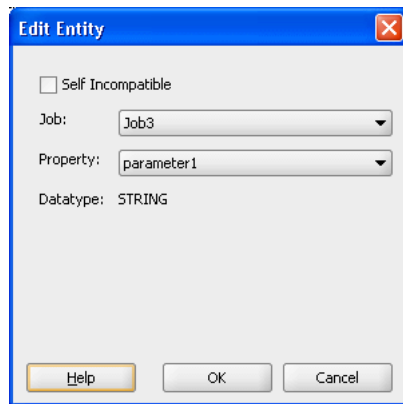
3. Under **Items**, select **Incompatibility** and click **OK**. This displays the Create Incompatibility window.
4. Use the Create Incompatibility dialog to specify the following:
 - a. In the **Name** field, enter a name for the incompatibility or accept the default name.
 - b. In the **Package** field, 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 13-4](#).

Figure 13-4 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 13-5](#).

Figure 13–5 Incompatibility Edit Entity Window

9. To specify self incompatibility, select **Self Incompatible**.
10. Save the incompatibility.

13.3 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.

13.3.1 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.

13.3.2 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.

Using the Runtime Service

This chapter describes how to use the Oracle Enterprise Scheduler runtime service APIs for submitting and managing job requests and for querying job request information from the job request history.

This chapter includes the following sections:

- [Section 14.1, "Introduction to the Runtime Service"](#)
- [Section 14.2, "Accessing the Runtime Service"](#)
- [Section 14.3, "Submitting Job Requests"](#)
- [Section 14.4, "Managing Job Requests"](#)
- [Section 14.5, "Querying Job Requests"](#)
- [Section 14.6, "Submitting Ad Hoc Job Requests"](#)

14.1 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.

14.2 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`.

14.2.1 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 14-1](#) shows a lookup for the Oracle Enterprise Scheduler runtime service using the `RuntimeServiceLocalHome` object.

Example 14-1 JNDI Lookup to Access Oracle Enterprise 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`.
-
-

14.3 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 14.6, "Submitting Ad Hoc Job Requests"](#)).

14.3.1 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 14–2](#) 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 14.6, "Submitting Ad Hoc Job Requests"](#). You can also submit a sub-request. For more information, see [Chapter 15, "Using Subrequests"](#).

Example 14–2 *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 14–1](#).
 - The runtime service internally uses the metadata service to obtain job definition metadata with the supplied `MetadataObjectId`, `jobDefnId`.
-

14.3.2 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 14–1 shows the runtime service field names and the corresponding system properties for the required job request properties.

Table 14–1 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 <code>Action.DELAY</code> .
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.

14.3.3 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 known 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.

14.4 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

14.4.1 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 14–2 shows the runtime service methods that allow you to obtain job request information.

Table 14–2 Runtime Service Get Request Methods

Runtime Service Method	Description
<code>getRequestDetail()</code>	Retrieves complete runtime details for the specified request
<code>getRequestDetailBasic()</code>	Retrieves basic runtime details of the specified request. The <code>RequestDetail</code> returned by this method includes most of the information as <code>getRequestDetail()</code> , but certain less commonly used information is omitted to improve performance.
<code>getRequestParameter()</code>	Retrieves the value of a request parameter.
<code>getRequests()</code>	Retrieves an enumeration of immediate child request identifiers associated with the specified request. This will include IDs for requests that did not complete, such as when the request transaction is rolled back or an error occurs.
<code>getRequestState()</code>	Retrieves the current state of the specified request

[Example 14–3](#) shows code that determines if there is any immediate child request in the HOLD state.

Example 14–3 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;
        }
    }
}

```

14.4.2 How to Change Job Request State

Using the runtime service, with a `requestID`, you can change the state of a job request. [Table 14–3](#) 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 14–3 Runtime Service Job Request State Methods

Runtime Service Method	Description
<code>cancelRequest()</code>	Cancels the processing of a request that is not in a terminal state.
<code>deleteRequest()</code>	Marks a request in a terminal state for deletion.

Table 14–3 (Cont.) Runtime Service Job Request State Methods

Runtime Service Method	Description
<code>holdRequest()</code>	Withholds further processing of a request that is in <code>WAIT</code> or <code>READY</code> state.
<code>releaseRequest()</code>	Releases a request from the <code>HOLD</code> state.

[Example 14–4](#) 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 14–4 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 14–4](#):

- You obtain the runtime service handle, `rHandle`, as shown in [Example 14–1](#).
 - 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.
-

14.4.3 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 14–4](#) shows the runtime service methods that allow you to lock and update up a job request.

Table 14–4 Runtime Service Update Methods

Runtime Service Method	Description
<code>lockRequest()</code>	Acquires a lock for the given request. The lock is released when <code>close()</code> 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.
<code>updateRequestParameter()</code>	Updates the property value of the specified request subject to the property read-only constraints.

[Example 14–5](#) shows code that updates a job request parameter. This code would be wrapped in a try/finally block as shown in [Example 14–1](#).

Example 14–5 Sample Runtime Service Parameter Update

```
...
s_runtime.lockRequest(rhandle, reqid);
s_runtime.updateRequestParameter(rhandle, reqId, paramName, "yy");
...
```

[Example 14–5](#) shows the following:

- Obtain the runtime service handle, `rhandle`, as shown in [Example 14–1](#).
- 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`.

14.5 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. Note that the return value will include IDs for requests that did not complete, such as when the request transaction is rolled back or an error occurs. For more information on filters, see [Section 6.4.1, "How to Create a Filter"](#).

When you create a filter for a query, you can use any of the field names shown in [Table 14–5](#) when querying the runtime store.

Table 14–5 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 <code>java.lang.Boolean</code> . The value may be <code>NULL</code> 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 <code>COMPLETED</code> .
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.
EXTERNAL_ID	The identifier for an external portion of an Oracle 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 <code>PROCESSTART</code> is set only when a request transitions from <code>READY</code> to <code>RUNNING</code> . This implies that (<code>PROCESSEND</code> - <code>PROCESSTART</code>) encompasses the entire span of execution: from the time the state becomes <code>RUNNING</code> to the time it transitions to a terminal state.
PROCESSOR	The name of the instance that processed the request.
PROCESSTART	The date and time that the process started. The <code>PROCESSTART</code> is set only when a request transitions from <code>READY</code> to <code>RUNNING</code> . This implies that (<code>PROCESSEND</code> - <code>PROCESSTART</code>) encompasses the entire span of execution: from the time the state becomes <code>RUNNING</code> 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 <code>READY</code> .
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 <code>RequestType</code>)
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).

Table 14–5 (Cont.) Query Filter Fields For Querying the Runtime (Defined in Enum

Name	Description
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 14–6 shows the runtime service method for querying job requests and Example 14–6 shows the use of this method.

Table 14–6 Runtime Service Query Methods

Runtime Query Method	Description
<code>queryRequests()</code>	Gets a summary of requests.

Example 14–6 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);

```

14.6 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.

14.6.1 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 14–7](#).

Table 14–7 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 14–7](#) shows sample code for an ad hoc request submission that uses a schedule.

Example 14–7 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 14–7](#).
- 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.

14.6.2 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.

14.6.3 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.

14.7 Implementing Pre-Process and Post-Process Handlers

Along with the core logic of your job, you can include code that executes before and after the job's main execution code. With code that executes before, known as a pre-process handler, you can do such things as set up certain conditions for the job executable. With code that executes after, known as a post-process handler, you can do such things as processing the results of the job executable, perhaps by printing reports or sending notifications.

You provide pre- and post-process handlers by implementing specific interfaces, then connecting your implementations to the service through a system property that indicates which of your classes to use.

14.7.1 Implementing a Pre-Process Handler

With a pre-process handler, your code can do things to create an environment for your job to execute. This could include creating connections to resources that your job requires, for example.

The pre-processor is instantiated and invoked at the start of request execution when the request transitions to RUNNING state. This is done each time the request is executed, including when a failed request is retried or a paused request is resumed after its sub-requests have completed.

You create a pre-process handler by implementing the `oracle.as.scheduler.PreProcessHandler` interface. With your pre-process handler class in hand, you specify that it should be used by setting the `SYS_preProcess` system property to the fully-qualified name of your handler class. You can define the property on job metadata or include it in the request submission parameters.

14.7.1.1 Implementing the PreProcessHandler Interface

Your `PreProcessHandler` implementation should do the pre-process actions your job requires, then return an `oracle.as.scheduler.HandlerAction` instance from the interface's one method, `preProcess`. (Your class may also implement the `Cancellable` interface if you want the job to support cancellation. It must also provide an empty constructor.)

The `HandlerAction` instance your `preProcess` implementation returns should give status about whether, and under what conditions, the job should proceed. When constructing the `HandlerAction` class, you pass it a `HandlerStatus` instance that indicates the status of pre-processing for the request.

Supported `HandlerStatus` values and actions are listed below. An unsupported status will cause the request to transition to an error state and be subject to retries if configured.

- `PROCEED` informs Oracle Enterprise Scheduler that request processing should commence. The request will remain in RUNNING state.
- `WARN` informs Oracle Enterprise Scheduler that request processing should commence but that a warning should be logged. The request will remain in RUNNING state.
- `CANCEL` informs Oracle Enterprise Scheduler that request pre-processing has been cancelled. The request will transition to CANCELLED state.
- `DELAY` informs Oracle Enterprise Scheduler to postpone request processing by the quantum of time specified by the `SYS_reprocessDelay` system property. The request remains in RUNNING state during the delay.
- `SYSTEM_ERROR` informs Oracle Enterprise Scheduler that the handler has experienced an error. The request will transition to an error state and is subject to retries if configured.
- `BIZ_ERROR` informs Oracle Enterprise Scheduler that the handler has experienced a business error. The request will transition to an error state not subject to retries.

14.7.2 Implementing a Post-Process Handler

With a post-process handler, your code can do things that should take place after your job has executed. This could include releasing connections to resources that your job required, for example, or generating a report based on request-specific data or status.

The post-processor is instantiated and invoked after job execution, when the request transitions to `COMPLETED` state. The post-processor is invoked only once for a request, in contrast to the pre-processor.

You create a post-process handler by implementing the `oracle.as.scheduler.PostProcessHandler` interface. With your post-process handler class in hand, you specify that it should be used by setting the `SYS_postProcess` system property to the fully-qualified name of your handler class. You can define the property on job metadata or include it in the request submission parameters.

14.7.2.1 Implementing the PostProcessHandler Interface

Your `PostProcessHandler` implementation should do the post-process actions your job requires, then return an `oracle.as.scheduler.HandlerAction` instance from the interface's one method, `postProcess`. (Your class may also implement the `Cancellable` interface if you want the job to support cancellation. It must also provide an empty constructor.)

The `HandlerAction` instance your `postProcess` implementation returns should give status about whether, and under what conditions, the job should conclude. When constructing the `HandlerAction` class, you pass it a `HandlerStatus` instance that indicates the status of post-processing for the request.

Supported `HandlerStatus` values and actions are listed below. An unsupported status will cause the request to transition to `WARNING` state.

- `PROCEED` to inform Oracle Enterprise Scheduler that request post-processing completed successfully. The request will transition to `SUCCEEDED` state or `WARNING` state depending on the status of the request prior to invoking the post-processor.
- `WARN` to inform Oracle Enterprise Scheduler that request post-processing resulted in a warning. The request will transition to `WARNING` state.
- `CANCEL` informs Oracle Enterprise Scheduler that request post-processing has been cancelled. The request will transition to `WARNING` state.
- `DELAY` to inform Oracle Enterprise Scheduler to postpone request processing by the quantum of time specified by the `SYS_reprocessDelay` system property. The request remains in `COMPLETED` state during the delay.
- `SYSTEM_ERROR` to inform Oracle Enterprise Scheduler that the handler has experienced an error. The request will transition `WARNING` state.
- `BIZ_ERROR` to inform Oracle Enterprise Scheduler that the handler has experienced a business error. The request will transition to `WARNING` state.

Using Subrequests

This chapter describes how to use Oracle Enterprise Scheduler subrequests to process data in parallel, particularly in a dynamic context, where the number of parallel requests can vary.

- [Section 15.1, "Introduction to Using Subrequests"](#)
- [Section 15.2, "Sample Subrequest"](#)
- [Section 15.3, "Creating and Managing Subrequests"](#)
- [Section 15.4, "Creating a Java Procedure that Submits a Subrequest"](#)
- [Section 15.5, "Creating a PL/SQL Procedure that Submits a Subrequest"](#)

15.1 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.

15.2 Sample Subrequest

[Example 15–1](#) 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 15–1 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.
  errbuf := fnd_message.get("FND", "COMPLETED NORMAL");
  retcode := 0;
end if;
end;
```

15.3 Creating and Managing Subrequests

- [Section 15.3.1, "How to Submit Subrequests"](#)
- [Section 15.3.2, "How to Cancel Subrequests"](#)
- [Section 15.3.3, "How to Hold Subrequests"](#)
- [Section 15.3.4, "How to Delete Subrequests"](#)
- [Section 15.3.5, "How to Submit Multiple Subrequests"](#)
- [Section 15.3.6, "How to Manage Paused Subrequests"](#)
- [Section 15.3.7, "How Subrequests Are Processed"](#)
- [Section 15.3.8, "How to Identify Subrequests"](#)
- [Section 15.3.9, "How to Manage Subrequests and Incompatibility"](#)

15.3.1 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.

15.3.2 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.

15.3.3 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.

15.3.4 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.

15.3.5 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.

15.3.6 How to Manage Paused Subrequests

- [Section 15.3.6.1, "Indicating Paused Status"](#)
- [Section 15.3.6.2, "Storing the Paused State for a Parent Request"](#)

15.3.6.1 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 `ExecutionPausedException` 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.

15.3.6.2 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()`.

15.3.7 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.

15.3.8 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.

15.3.9 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.

15.4 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 summarizes 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 15-2](#) shows a Java procedure with a subrequest.

Example 15-2 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
}
}

```

15.5 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 raising 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 summarizes 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 15-3](#) shows a PL/SQL procedure with a subrequest.

Example 15-3 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.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', 'MyData2',
'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.
errbuf := fnd_message.get("FND", "COMPLETED NORMAL");
retcode := 0;
end if;

```

```
end;
```

Working with Asynchronous Java Jobs

This chapter describes how to use Oracle Enterprise Scheduler to invoke asynchronous Java jobs to support long-running or non-container-managed jobs that invoke Java code.

This chapter includes the following sections:

- [Section 16.1, "Introduction to Working with Asynchronous Java Jobs"](#)
- [Section 16.2, "Creating an Asynchronous Java Job"](#)
- [Section 16.3, "A Use Case Illustrating the Implementation of a BPEL Process as an Asynchronous Job"](#)
- [Section 16.4, "How to Implement BPEL with an Asynchronous Job"](#)
- [Section 16.5, "Handling Time Outs and Recovery for Asynchronous Jobs"](#)
- [Section 16.6, "Oracle Enterprise Scheduler Interfaces and Classes"](#)

16.1 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.

16.2 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.

16.2.1 Implementing the Asynchronous Java Job Asynchronous Interface

An asynchronous Java job invocation must implement the `AsyncExecutable` interface.

16.2.2 Asynchronous Java Job `execute()` Method

The duty of an asynchronous Java job'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.

16.2.3 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.

16.2.4 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 16–15](#)). 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.
```

16.2.5 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 16–12](#).

16.2.6 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.

16.2.6.1 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 Oracle Web Services*.

16.2.6.2 Using EJB to Notify When an Asynchronous Job Completes

When an asynchronous Java job'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.

16.2.7 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.

16.2.8 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 16.2.6.1, "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 16–1](#).

Example 16–1 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/ed1";
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 successfully");
} 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
}

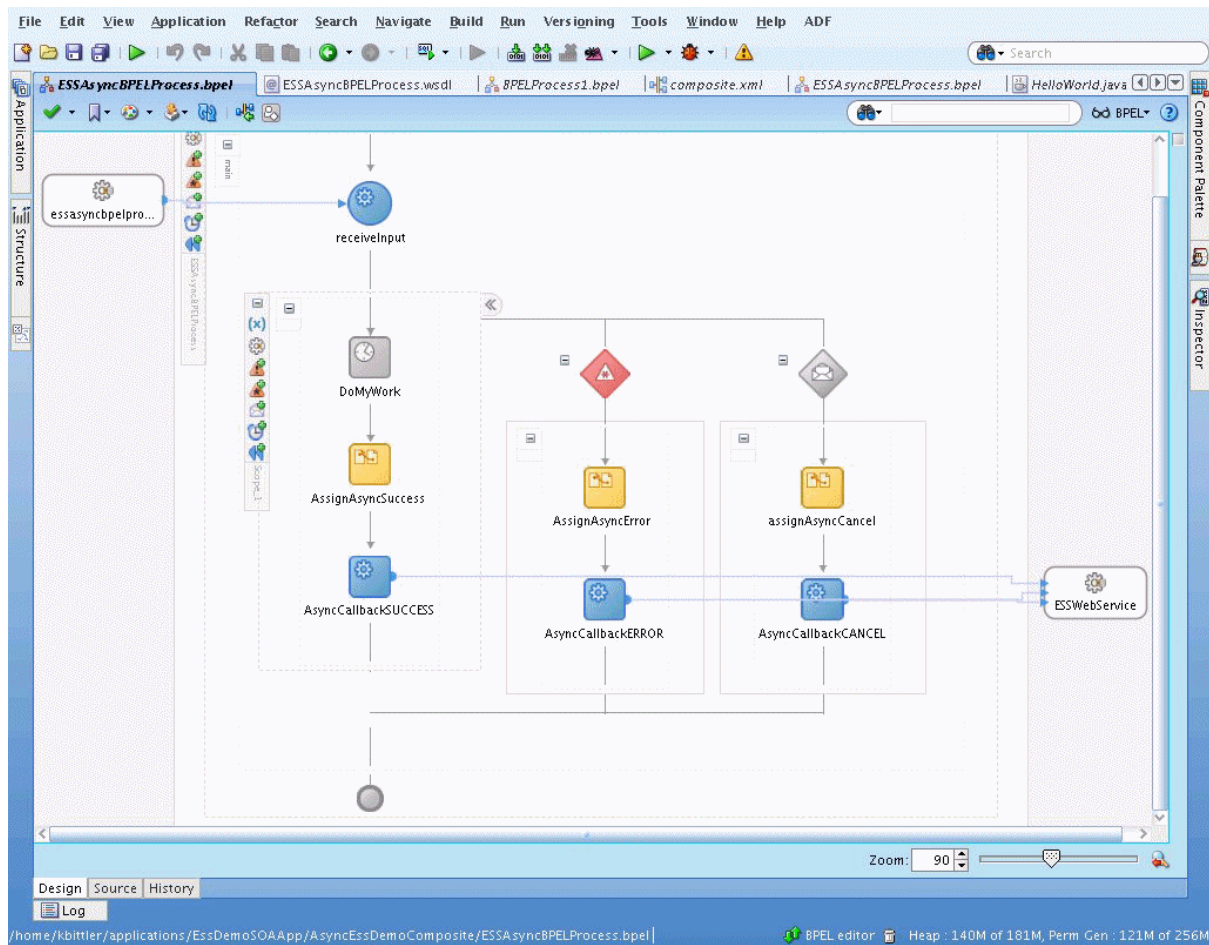
```

16.2.8.1 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 16-1](#).

- 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 16–1 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 16–2](#), [Figure 16–3](#), and [Figure 16–4](#) for the AssignAsyncError assignment activity.

Figure 16–2 AsyncCallbackError Argument Mapping for statusMessage Element

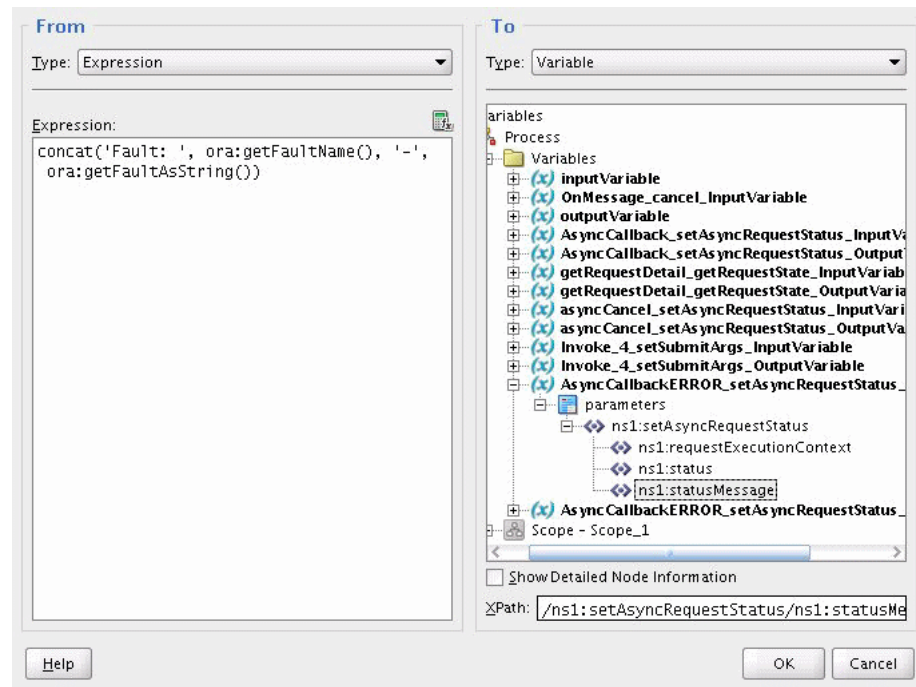


Figure 16–3 AsyncCallbackError Argument Mapping for requestExecutionContext

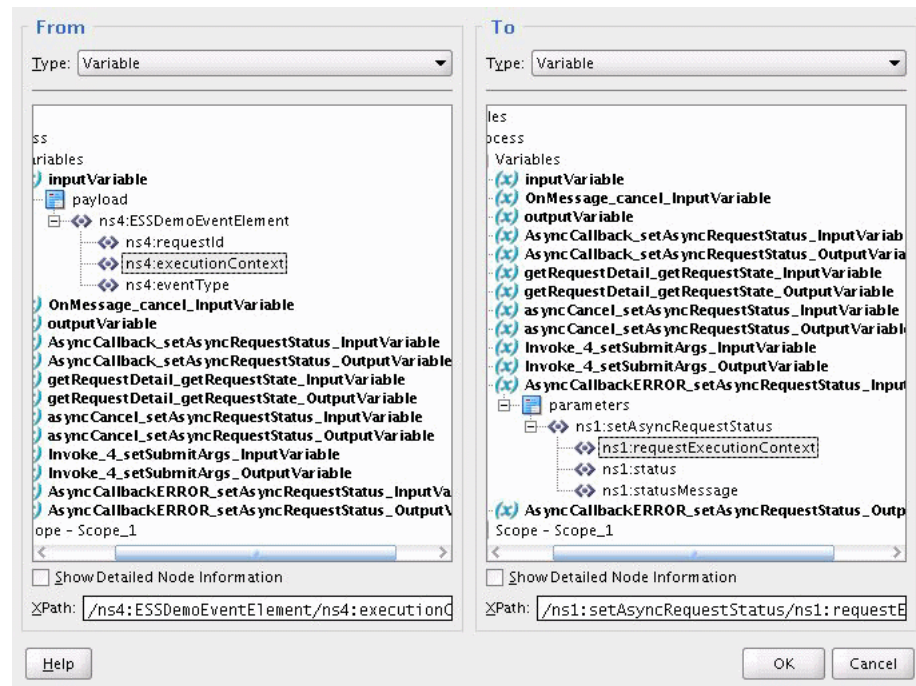
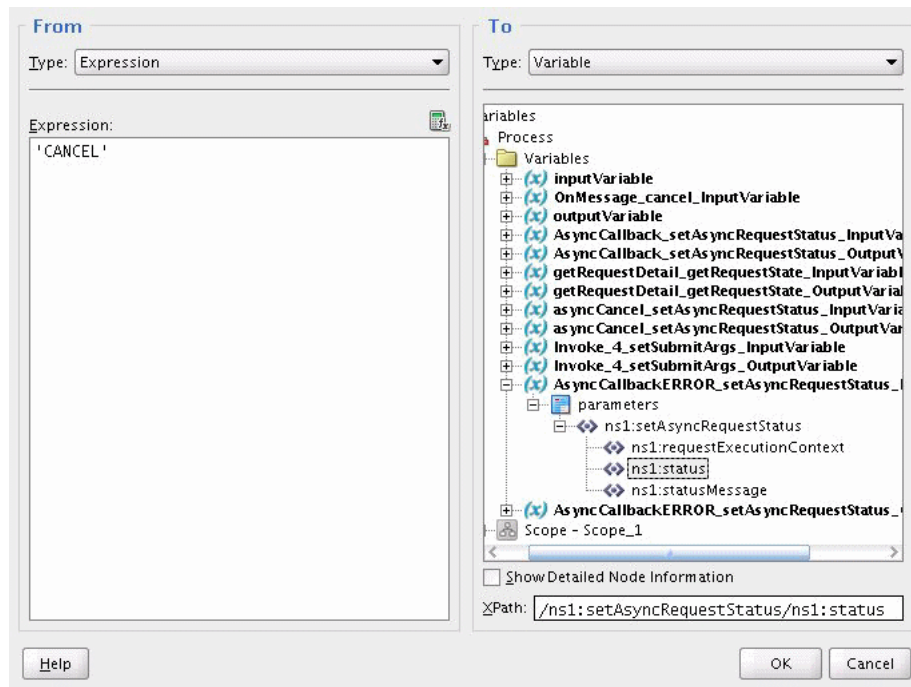


Figure 16–4 AsyncCallbackError Argument Mapping for status Element

16.3 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

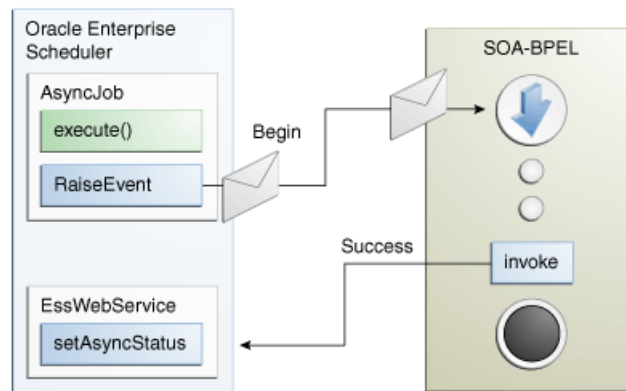
Asynchronous Oracle Enterprise Scheduler jobs are Java jobs that implement the `AsyncExecutable` interface, which is invoked by Oracle Enterprise Scheduler by implementing the `execute()` method. This method enables initiating a long running or remote task where the `execute()` method completes (such 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 Mediator and BPEL, as shown in [Figure 16–5](#).

Figure 16–5 BPEL Call from Oracle Enterprise Scheduler Asynchronous Job

16.3.1 Introduction to the Recommended Design Pattern

There are use cases 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.

16.3.2 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.

16.3.3 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 its 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.

16.4 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 Mediator and BPEL.
6. Add fault handling and correlated `onMessage` branch for error and `cancel` job status updates.

16.4.1 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 16–2](#).

Example 16–2 Adding Oracle JDeveloper Libraries

```
public class ASMEventAsyncJob implements AsyncExecutable, AsyncCancellable {
    public ASMEventAsyncJob() {
        super();
    }

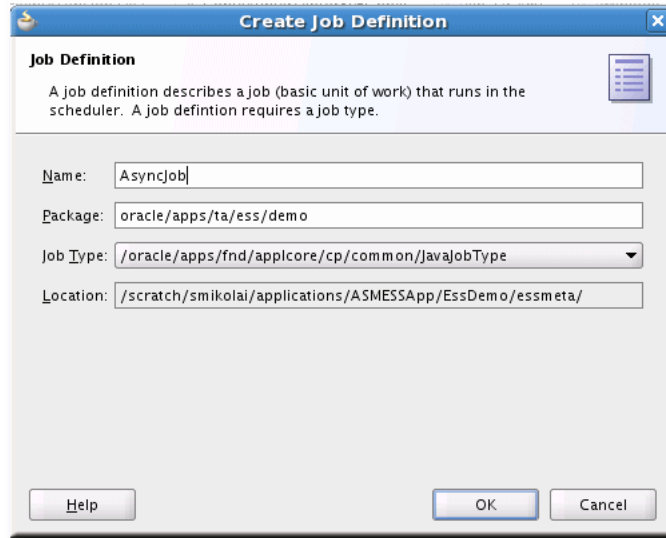
    public void execute(RequestExecutionContext ctx, RequestParameters params)
        throws ExecutionErrorException,
               ExecutionWarningException,
               ExecutionCancelledException,
               ExecutionPausedException
    {
        publishEvent(ctx.getRequestId() + "", ctx.toString(), "ESS_EVENT");
        return;
    }

    public void cancel (RequestExecutionContext ctx,
                       RequestParameters requestParams) {
        publishEvent(ctx.getRequestId() + "", ctx.toString(), "CANCEL_ESS_EVENT");
        return;
    } // cancel
}
```

16.4.2 Use Case: Create the Asynchronous Job Definition

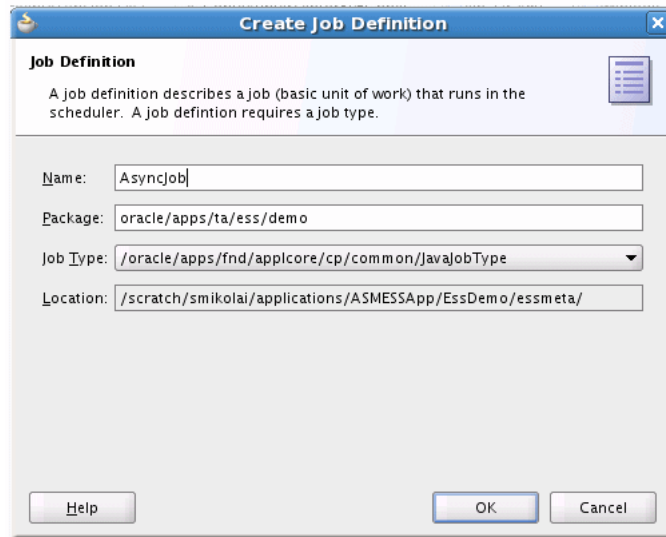
In your Oracle Enterprise Scheduler JDeveloper workspace, click "New", choose the Enterprise Scheduler Service technology group and select "Job Definition". Enter the name of your Oracle Enterprise Scheduler job definition, choose the provided "JavaJobType" and select the class build in step 1 as the overriding Java class for this job definition, as shown in [Figure 16-6](#).

Figure 16-6 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 16-7](#).

Figure 16-7 Create Job Definition with Job Type Defined



16.4.3 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 16-3](#) shows a sample XSD file.

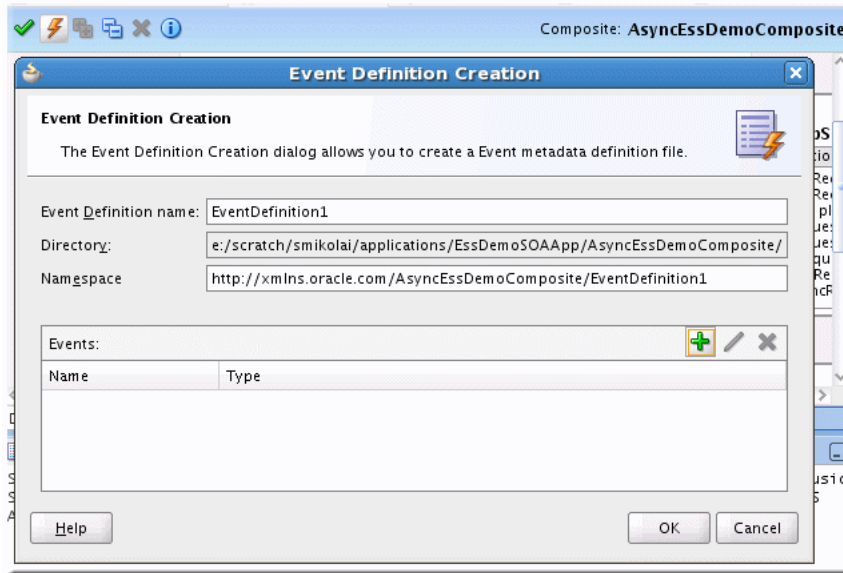
Example 16-3 Sample XSD File

```
<?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"

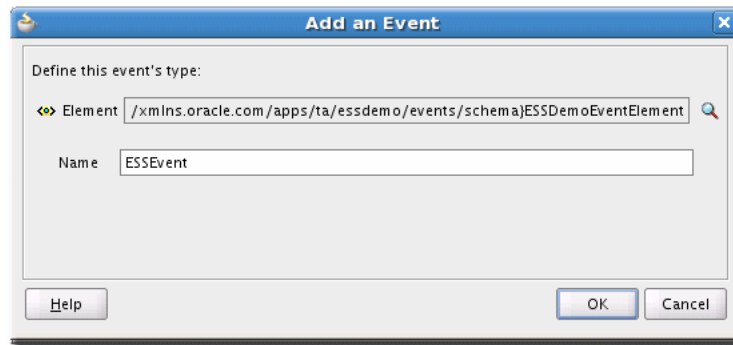
  targetNamespace="http://xmlns.oracle.com/apps/ta/essdemo/events/schema"
  attributeFormDefault="unqualified"
  elementFormDefault="qualified">
  <xsd:element name="ESSDemoEventElement" type="ESSDemoEventElementType" />
  <xsd:complexType name="ESSDemoEventElementType">
    <xsd:sequence>
      <xsd:element 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 16-8](#)

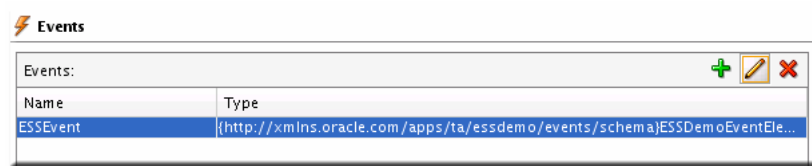
Figure 16-8 Event Definition Creation



Next, assign a name and namespace and click **Add** to add a new event to this definition, as shown in [Figure 16-9](#).

Figure 16–9 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 16–10](#).

Figure 16–10 Events List

[Example 16–4](#) shows a sample of the EDL file that is created.

Example 16–4 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>
```

16.4.4 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 16–5](#).

Example 16–5 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
}

```

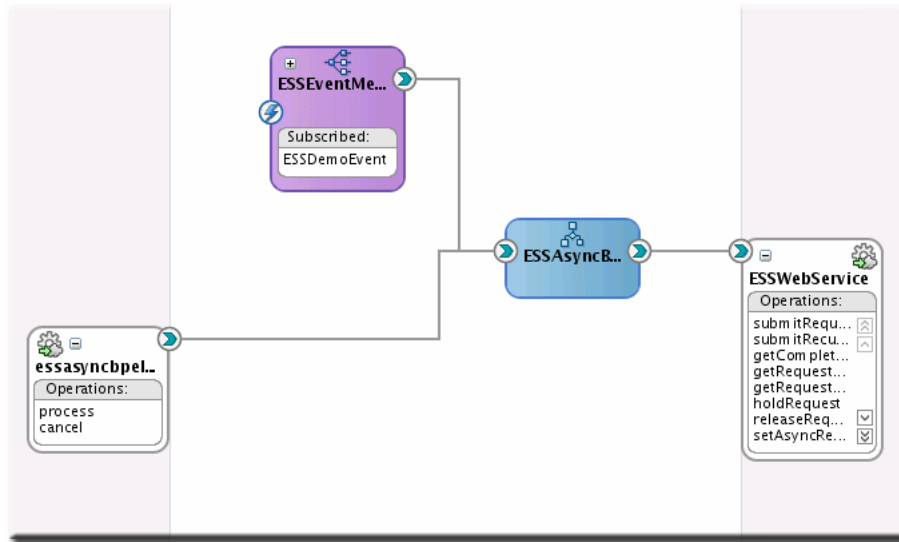
16.4.5 Design the SOA Composite with Mediator and BPEL

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 16–11](#).

Figure 16–11 Composite with BPEL and ESSWebService

For more information about invoking the Oracle Enterprise Scheduler web service, see [Chapter 11, "Using the Oracle Enterprise Scheduler Web Service."](#)

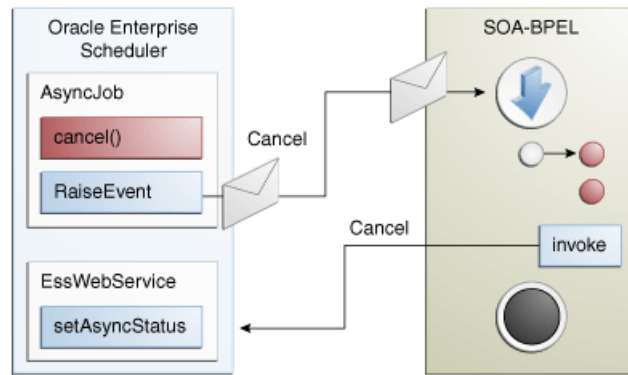
16.4.6 Add Fault Handling and Correlated onMessage Branch for Error and Cancel Job

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 16–12](#).

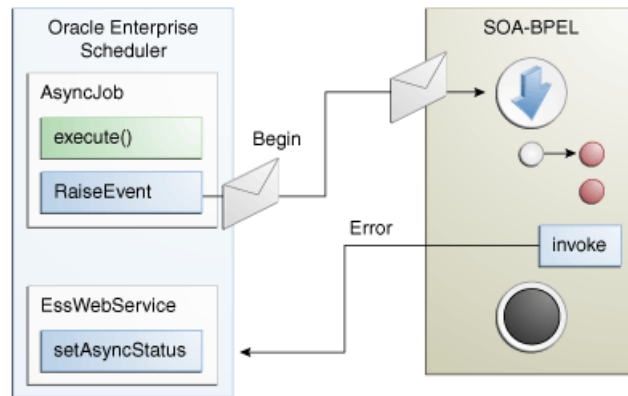
Figure 16–12 BPEL Handling Cancellation



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 16–13.

Figure 16–13 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.

16.4.6.1 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.

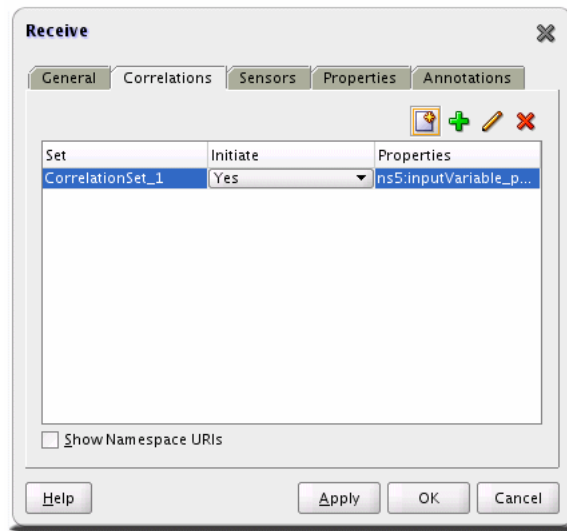
To setup the correlation set, open the BPEL process in the designer, double-click the Receive activity and click the correlations tab.

Note that correlation 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.

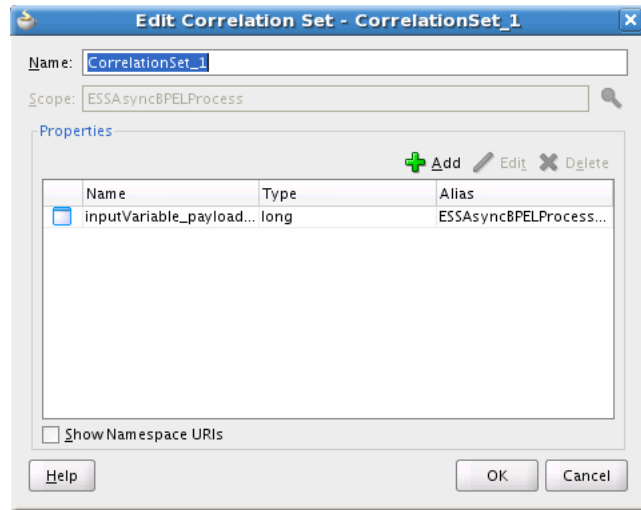
To create a correlation set:

- 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 16-14](#).

Figure 16–14 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 16–15](#).

Figure 16–15 Edit Correlation Set

16.4.6.2 Create the onMessage Branch with Use of Correlation Set

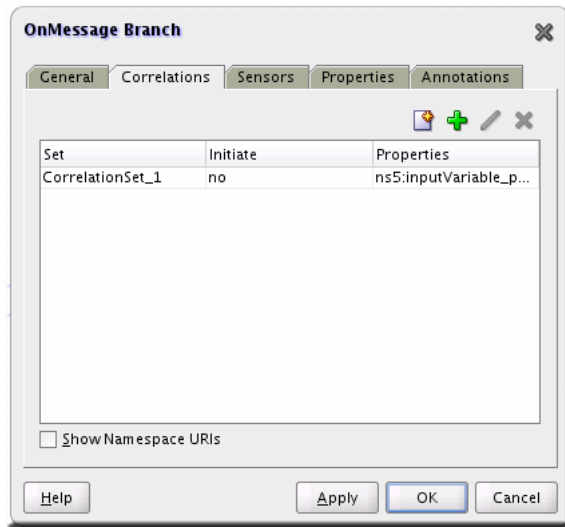
Once the correlation set has been defined and set for initiate it's now possible to create the onMessage branch on the scope which will contain the activities necessary to accept the incoming cancellation message, perform any compensation or cleanup and then assign the job's completion status to CANCEL.

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 16–16](#).

- 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 16–16 BPEL OnMessage Branch

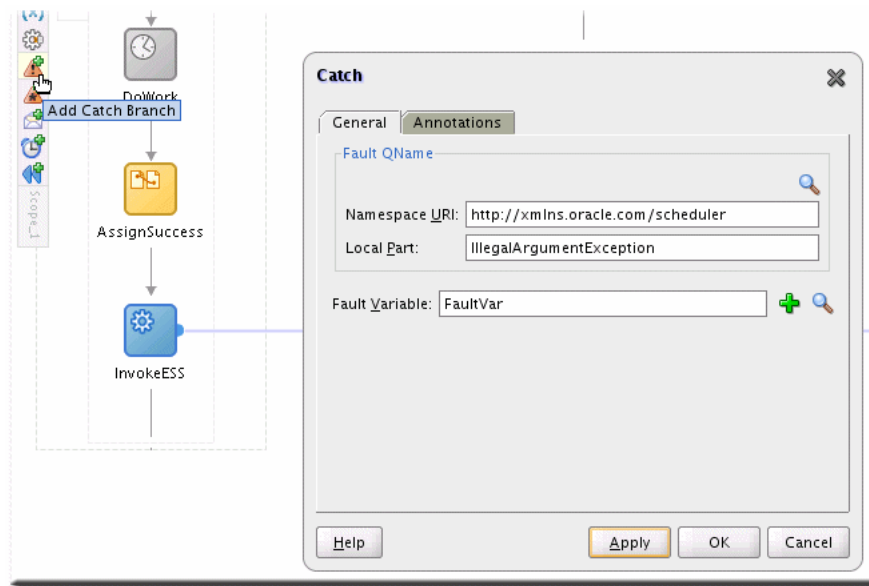


16.4.6.3 Create the Fault Branch

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 16–17](#).

Figure 16–17 Catch Branch for BPEL Flow

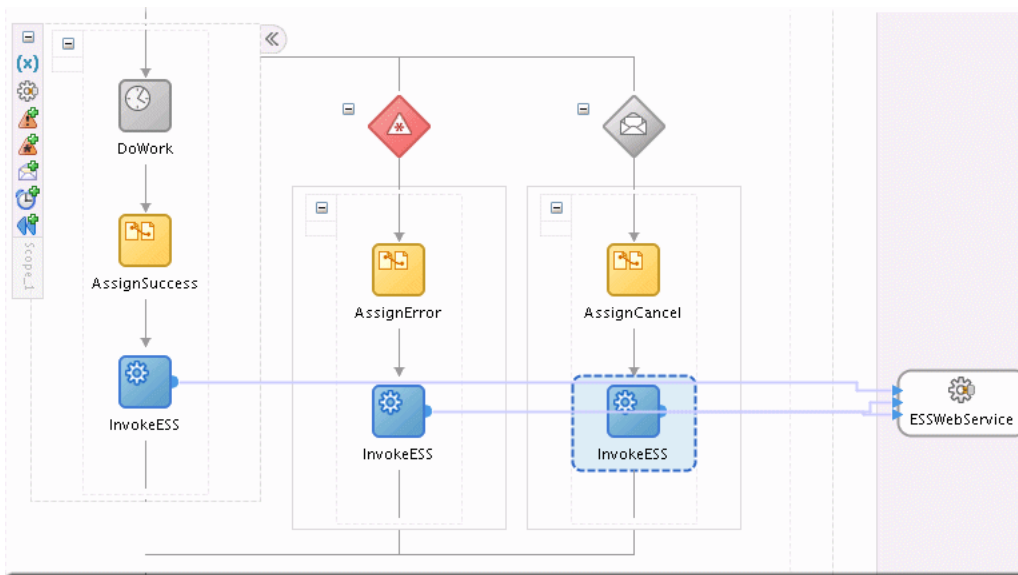
16.4.6.4 Populate the onMessage and Fault Branch

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 16–18](#).

Figure 16–18 Entire BPEL Flow Sample

16.4.7 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 Middleware 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 16–6](#).

Example 16–6 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 Middleware 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 16-7](#).

Example 16-7 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.

16.4.8 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 Middleware Administrator's Guide for Oracle Enterprise Scheduler*.

16.5 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.

16.5.1 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.

16.5.1.1 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.

16.5.1.2 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 16-8](#).

Example 16-8 *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 16-9](#).

Example 16-9 *Using REQUEST_HISTORY_VIEW*

```
SELECT requestId FROM request_history_view WHERE timedout='Y' AND state=3;
```

16.5.1.3 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 Middleware Administrator's Guide for Oracle Enterprise Scheduler*.

16.5.1.4 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.

16.5.2 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 Middleware Administrator's Guide for Oracle Enterprise Scheduler*.

16.5.3 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 16–10](#).

Example 16–10 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 16–11](#) shows the `recoverRequest` syntax.

Example 16–11 *recoverRequest*

```
/**
 * Attempts to force a request to complete under certain conditions.
 * <p>
 * 1. The request must already be 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, OperationException,
        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 Middleware Administrator's Guide for Oracle Enterprise Scheduler*.

16.6 Oracle Enterprise Scheduler Interfaces and Classes

Sample code illustrating the new Oracle Enterprise Scheduler asynchronous callback interfaces and classes are shown in [Example 16–12](#), [Example 16–13](#), [Example 16–14](#) and [Example 16–15](#).

Example 16–12 *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 Scheduler.
*/
public UpdateAction onEvent( RequestExecutionContext context,
RequestParameters parameters,
                           oracle.as.scheduler.async.AsyncStatus resultCode,
                           String messagePayload );
}
```

The `UpdateAction` class is returned by `Updatable.onEvent`.

Example 16–13 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 16–14 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 16–15 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 ;
```

Creating Job Request Logs and Output

This chapter describes how to use Oracle Enterprise Scheduler to generate job request logs and output that should be saved for later use by administrators and users.

Logs generated by job requests help administrators diagnose problems and see job-specific status. Logs are accessible through the Fusion Middleware Control. In addition, some jobs generate output as part of their work, such as a report about job-specific data that a user can review after the job has completed. Your code can create and store request log information as well as request output.

This chapter includes the following sections:

- [Section 17.1, "Creating Request Logs"](#)
- [Section 17.2, "Creating Request Output"](#)

17.1 Creating Request Logs

You can create job request logs from your job code. Oracle Enterprise Scheduler provides APIs for creating and writing logs and interacting with the content store.

Oracle Enterprise Scheduler supports a single log per request. The log will have a name of the form REQUESTID.log. The logging APIs log directly to the content store, and log content may not be rolled back.

For more about viewing job request logs with Fusion Middleware Control, see *Viewing Job Request Logs* in *Oracle Fusion Middleware Administrator's Guide for Oracle Enterprise Scheduler*.

17.1.1 System Properties

When logging for Java and PL/SQL jobs, what actually shows up in the log will be constrained by how the SYS_EXT_requestLogLevel system property has been set. The property determines the logging level supported by the Java and PL/SQL request logging APIs.

The property's value defaults to INFO. The complete set of valid values are SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST. Use Java and PL/SQL APIs to discover the request log level.

Note that because process jobs do not use the logging API, logging levels aren't supported in them.

17.1.2 APIs for Handling Request Logs

You can use methods of the `oracle.as.scheduler.RuntimeService` class to handle logs stored in the Oracle Enterprise Scheduler content store. You'll need to first get a `RuntimeServiceHandle` instance. You'll pass this instance as an argument for each of these `RuntimeService` methods.

For more on the `RuntimeServiceHandle`, see [Section 14.2.1, "How to Access the Runtime Service and Obtain a Runtime Service Handle"](#).

Table 17–1 RuntimeService Methods for Handling Request Logs

Method	Description
<code>getLogContentDetail(RuntimeServiceHandle handle, long requestId)</code>	Returns a <code>ContentDetail</code> instance with the log content detail for the request, or null if the log does not exist.
<code>openLogContent(RuntimeServiceHandle handle, long requestId)</code>	Returns a <code>ContentHandle</code> instance from opening the request log to retrieve log data for the specified request. You can use the handle to retrieve output data. The content must be closed to release the handle.
<code>getLogLines(RuntimeServiceHandle handle, ContentHandle contentHandle, int maxLines)</code>	Returns a <code>String</code> array with at most <code>maxLines</code> lines from the request log, continuing from the last call to this method. The content handle is from the previous call to <code>openLogContent</code> . This returns a <code>String</code> array of lines from the log without line terminators; if no more lines, array will be empty.
<code>getTextContent(RuntimeServiceHandle handle, ContentHandle contentHandle, int maxChars)</code>	Returns a <code>char</code> array with at most <code>maxChars</code> characters from the log or output text content.
<code>closeContent(RuntimeServiceHandle handle, ContentHandle contentHandle)</code>	Closes the previously opened log or output content and releases the handle.

17.1.3 Log Header

Your logging code will write entries to a log that begins with the following heading information. This header is prefixed to each record written to the log.

Example 17–1

```
####[TIMESTAMP] [LOGLEVEL]

Sample log lines:
####[2011-07-11T14:20:32.276-07:00] [INFO] This is a log record.
####[2011-07-11T14:20:32.282-07:00] [INFO] This is the first line of a multi-line
log record:
second line of multi-line log record.
```

17.1.4 Creating Request Logs from a Java Job

You can use the Java request logger to log during the execute and update stages of a Java or asynchronous Java job, as well as during pre-processing and post-processing for all job execution types.

The job logic must use the `ContentFactory` API to get the request logger. Oracle Enterprise Scheduler will use the current value of the `SYS_EXT_requestLogLevel` system property to constrain logging level each time the logger is retrieved.

17.1.4.1 APIs for Java Job Logging

In your Java job's logic, you can use the `oracle.as.scheduler.request.ContentFactory` class `getRequestLogger` method (see [Table 17-2](#)) to get a `RequestLogger` instance for adding log entries. Note that the request logger will not support a resource bundle.

The Java APIs available for handling logs include the following:

- Use the `ContentFactory` class to get instances of a `RequestLogger` you can use to create the log and add entries. See [Table 17-2](#).
- Use the `RequestLogger` class to write the log. See [Table 17-3](#).

The `oracle.as.scheduler.request.ContentFactory` class provides methods to get your code access to the output content framework, as well as to an instance you can use to create the output itself.

Table 17-2 *ContentFactory Methods for Creating Request Logs*

Method	Description
<code>getRequestLogger(long requestId)</code>	Returns a <code>RequestLogger</code> instance for the specified <code>requestId</code> and creates log content named <code>requestId.log</code> in Oracle Enterprise Scheduler content store.

Once you have a logger instance, you can use the methods in [Table 17-3](#) to add entries.

Table 17-3 *RequestLogger Methods for Creating Request Logs*

Method	Description
<code>log(Level level, String msg)</code>	These methods log messages at the specified levels.
<code>fine(String msg)</code>	The message is logged only if the specified logging level is equal or greater than the log level specified by the <code>SYS_EXT_requestLogLevel</code> system property. If the property isn't set, the default log level is INFO.
<code>finer(String msg)</code>	
<code>finest(String msg)</code>	
	When using the <code>log</code> method, the <code>java.util.logging.Level</code> supports the following values, in descending order.
	<ul style="list-style-type: none"> ■ SEVERE ■ WARNING ■ INFO ■ CONFIG ■ FINE ■ FINER ■ FINEST

17.1.4.2 Example

[Example 17-2](#) shows a very simple Java job example that does logging.

Example 17-2 *Java Request Logging Example*

```
import oracle.as.scheduler.request.ContentFactory;
import oracle.as.scheduler.request.RequestLogger;
import java.util.logging.Level;

class ExampleJavaLogger{

    private boolean m_loggingEnabled = false;
    private RequestLogger m_requestLogger = null;
```

```
public void execute( RequestExecutionContext ctx,
                   RequestParameters params )
{
    try
    {
        m_requestLogger = ContentFactory.getRequestLogger(ctx.getRequestId());
        m_loggingEnabled = true;
    }
    catch (Exception ex)
    {
        // failed to get request logger
    }

    log(Level.INFO, "Starting the job.");
    // ...
    log(Level.INFO, "Ending the job.");
}

private void log( Level level, String message )
{
    if (m_loggingEnabled)
    {
        m_requestLogger.log(level, message);
    }
}
}
```

17.1.5 Creating Request Logs from a PL/SQL Job

To create logs from PL/SQL, your code can use the ESS_JOB PL/SQL package to write log entries.

17.1.5.1 ESS_JOB Package Support for Creating Logs

Oracle Enterprise Scheduler provides the ESS_JOB package with functions and procedures for logging from PL/SQL code.

Table 17-4 *ESS_JOB Functions and Procedures for Request Logging*

Method	Description
procedure write_log(p_level in integer, p_text in varchar2);	<p>Writes p_text as a message to request log content for the Oracle Enterprise Scheduler request associated with the current session.</p> <p>The message is logged only if the specified logging level is equal or greater than the log level specified by the SYS_EXT_requestLogLevel system property. If the property isn't set, the default log level is LEVEL_INFO.</p> <p>Log level values correspond to those defined in java.util.logging.Level.</p> <p>Use the following values for the p_level parameter (shown in descending order):</p> <ul style="list-style-type: none"> ▪ LEVEL_SEVERE ▪ LEVEL_WARNING ▪ LEVEL_INFO ▪ LEVEL_CONFIG ▪ LEVEL_FINE ▪ LEVEL_FINER ▪ LEVEL_FINEST

17.1.5.2 PL/SQL Request Logging Example

An example of request logging by a SQL request job procedure is shown below.

Example 17-3 *PLSQL Request Logging*

```

create or replace procedure log_example_job
( request_handle in varchar2 )
as
  v_request_id number := null;
begin
  ess_job.write_log(ess_job.level_fine,
    'LOG_EXAMPLE_JOB Procedure Begin');

  -- Oracle Enterprise Scheduler request id being executed.
  begin
    v_request_id := ess_runtime.get_request_id(request_handle);
  exception
    when others then
      ess_job.write_log(ess_job.level_severe,
        'Bad request handle: '||request_handle);
      raise_application_error(-20000,
        'Failed to get request id for request handle '||request_handle,
        true);
  end;

  -- Job logic
  ess_job.write_log(ess_job.level_info,
    'Executing job logic...');

  ess_job.write_log(ess_job.level_fine,
    'LOG_EXAMPLE_JOB Procedure End');
end;
/

```

17.1.6 Creating Request Logs from a Process Job

You can write to the job request log from process job. The way this works is quite different from Java and PL/SQL jobs, where the executing code has access to an API for writing entries at a particular level. Instead, for a process job, the job's standard output and standard error are redirected to a file in the request's log work directory (a location set by Oracle Enterprise Scheduler). Oracle Enterprise Scheduler will import this file and append it to the request log in the content store.

In other words, to log from a process job, you need only write to standard output from job logic code.

The encoding used to read the log file is determined as follows. If the application `ess-config.xml` defines `LANG` in the Env properties, Oracle Enterprise Scheduler will use the encoding from that. Otherwise, Oracle Enterprise Scheduler will use the default encoding of the container.

Note that you can't log at particular levels from a process job (where the API for setting the level isn't available). So the `SYS_EXT_requestLogLevel` system property will not constrain log contents. Oracle Enterprise Scheduler will always append the contents of the log file to the request log in the content store.

17.2 Creating Request Output

You can have your job write files as output at run time. For example, your job might collect data that would be useful in a report for users. When you generate output at run time, it's available to be retrieved later through a client user interface or the Fusion Middleware Control.

The output your code creates can be imported into the Oracle Enterprise Scheduler content store, from which it can be retrieved later. You can do this either by writing to the file system in the usual way, or by using Oracle Enterprise Scheduler APIs to import directly to the store.

When you use the file system, you write to a particular directory whose location has been configured in the Oracle Enterprise Scheduler `ess-config.xml` file. Oracle Enterprise Scheduler creates a request file directory to contain files written from all requests. Your code writes to a subdirectory of this created specifically for the request. Oracle Enterprise Scheduler automatically imports all of the request's output to the content store, then deletes request-specific subdirectories.

Oracle Enterprise Scheduler provides APIs for accessing the content store and writing output from both Java and PL/SQL code.

17.2.1 Using the Request File Directory

The *request file directory* is specified in the Oracle Enterprise Scheduler `ess-config.xml` file. For each request, Oracle Enterprise Scheduler can create request-specific subdirectories of this request file directory: a working directory for temporary files and an output directory for output files that should be saved in the content store.

Your code can write temporary and output files to their respective request-specific directories at run time. Oracle Enterprise Scheduler will import to the content store files in the request's output directory. When the content is imported depends on whether the request file directory is shared or local, as described in [Section 17.2.1.1](#), [Section 17.2.1.2](#), and [Section 17.2.1.3](#).

After automatically importing all of the request's output to the content store, Oracle Enterprise Scheduler deletes the request-specific output directory and its contents.

The request file directory can be local, meaning that it is used only for work done on a single server. It can also instead be shared, in which a single directory is used for work done on multiple servers. Run time behavior differs depending on whether the directory is configured to be local or shared.

The directory is specified in the `ess-config.xml` file, as shown in [Example 17-4](#).

Example 17-4 Request File Directory Configuration with `ess-config.xml` file

```
<ess:EssConfig xmlns:ess="http://ess.oracle.com"
  xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ess:EssProperties>
    <ess:EssProperty key="RequestFileDirectory" value="/etc/outputfiles"
      immutable="true"/>
    <ess:EssProperty key="RequestFileDirectoryShared" value="false"
      immutable="true"/>
  </ess:EssProperties>
</ess:EssConfig>
```

17.2.1.1 Common Request File Directory Behavior

Oracle Enterprise Scheduler will automatically import all request output files from the output directory to the content store, and will delete the request working directory and all files in it.

Imported files always overwrite existing content of the same file name as long as the existing content was previously imported. If the existing content was created using the API, then it is considered to be distinct from the new file, and the new file will not overwrite and will be ignored. In other words, content created with the API has precedence.

Oracle Enterprise Scheduler will not import output files of zero length.

17.2.1.2 Shared Request File Directory Behavior

Any files created by the request will remain in its working and output directories until the request completes and goes to a terminal state. Any files created by a request in a shared file directory will be available to all stages of the request.

17.2.1.2.1 Error Handling When a Shared Request File Directory is Used Oracle Enterprise Scheduler creates the request work directory before the job request transitions to RUNNING state. Any error while creating the directory results in a system error for the request.

For process a job, importing the log occurs after the request transitions to a terminal state. If an error occurs while importing the log, the error is logged and the request log is left in the file system. You will need to manually import the log to the content store.

Importing output files for any job type occurs after the request transitions to a terminal state. If there is an error while importing output files, the error is logged and the output files are left in their directories on the file system. You will need to remove the output files.

17.2.1.3 Local Request File Directory Behavior

Oracle Enterprise Scheduler will create request-specific directories before any stage of the request runs. If the request file directory is local, it must be a location that is guaranteed to exist locally on every server. In this case, files created by one stage of the

request are not guaranteed to be available in the next stage because stages are independent units of work and may run on different servers.

For a local request file directory, the common behavior holds except that Oracle Enterprise Scheduler performs the actions for each stage. The reason is that each stage may execute on a different server, and it is necessary for Oracle Enterprise Scheduler to capture and clean up the files for each stage because they may not be there for the next stage.

In case a request needs access to all previously imported output files, it can set the parameter `SYS_EXT_executeAutoExport = true`. If this is set, at the beginning of the execute stage, Oracle Enterprise Scheduler will automatically export previously imported output files to the request's working output directory. This gives you an opportunity to update the file before it is imported back to the content store at the end of the execute stage. (Note that the content isn't removed from the content store when the content is exported.) Furthermore, Oracle Enterprise Scheduler provides an API for a request to selectively export previously imported output files.

17.2.1.3.1 Error Handling When a Local Request File Directory is Used When a local request file directory is used, file imports happen at the end of each stage (pre-processing, execution, update, post-processing). If an error occurs while importing logs or output files, the log and output files that failed to import will be moved to a mirror directory at `<request_file_directory>/preserve`. For example, for request 18 this would be `<request_file_directory>/preserve/18`.

For the pre-processing stage, an error creating the request directory at the beginning of the stage or importing output files at the end of the stage will result in a system error for the request.

For the post-processing stage, an error creating the request directory at the beginning of the stage or importing output files at the end of the stage will result in a warning for the request.

For the execution stage of a Java job, asynchronous Java job, and process job request, an error creating the request directory or automatically exporting previously imported output files (such as when the `SYS_EXT_executeAutoExport` system property is used) at the beginning of the stage or importing output files at the end of the stage will result in a system error for the request.

If the request is a process job, an error importing the request log will be logged and not treated as an error. The log will be left in the file system, and you may manually import it to the content store. If there is an internal error during execution of a process job, log and output files will not be imported because the job could still be running. The log and output files will be imported when the job is terminated, either automatically or manually by the user. If the job goes to `ERROR_MANUAL_RECOVERY`, it is the user's responsibility to clean up the request log and output files.

For the update stage, an error creating the request directory or importing output files is logged only.

17.2.2 System Properties

Setting the `SYS_EXT_supportOutputFiles` system property is essential to using the request file directory and automatic importing of output files.

To use the request output directory to create output files, the job must define a parameter using the system property `SYS_EXT_supportOutputFiles`. Depending on what sort of files the job wants to create, the property can be set in one of the following ways:

- Set it to "output" in order to have files written to the request output directory imported to the content store.
- Set it to "work" in order to write files to the request working directory that are not intended for import, such as temporary files.
- Set it to "none", or leave it undefined, if the job doesn't create any output or temporary files.

Table 17-5 System Properties for Creating Request Output

Method	Description
SYS_EXT_supportOutputFiles	String property indicating whether job will create files in the file system. Supported values are "work", "output", and "none". An invalid value is treated as "none".
SYS_EXT_executeAutoExport	Boolean property indicating whether previously imported output files shall be exported at the start of the execute stage. The content isn't removed from the content store when it is automatically exported.

17.2.3 APIs for Handling Request Output

You can use methods of the `oracle.as.scheduler.RuntimeService` class to handle request output stored in the Oracle Enterprise Scheduler content store. You'll need to first get a `RuntimeServiceHandle` instance. You'll pass this instance as an argument for each of these `RuntimeService` methods.

For more on the `RuntimeServiceHandle`, see [Section 14.2.1, "How to Access the Runtime Service and Obtain a Runtime Service Handle"](#).

Table 17-6 RuntimeService Methods for Handling Request Output

Method	Description
<code>getOutputContentDetail(RuntimeServiceHandle handle, long requestId, String contentName)</code>	Returns a <code>ContentDetail</code> instance for the specified output content for the specified request, or null if the content does not exist.
<code>getOutputContentDetail(RuntimeServiceHandle handle, long requestId)</code>	Returns a <code>ContentDetail List</code> instance for all output content for the request. The list will be empty if there is no output content.
<code>openOutputContent(RuntimeServiceHandle handle, long requestId, String contentName)</code>	Opens the specified request output to retrieve output data for the specified content, returning <code>ContentHandle</code> instance. You can use the handle to retrieve output data. The content must be closed to release the handle.
<code>getTextContent(RuntimeServiceHandle handle, ContentHandle contentHandle, int maxChars)</code>	Returns a char array with at most <code>maxChars</code> characters from the log or output text content.
<code>getBinaryContent(RuntimeServiceHandle handle, ContentHandle contentHandle, int maxBytes)</code>	Returns a byte array with at most <code>maxBytes</code> bytes from the binary content.
<code>closeContent(RuntimeServiceHandle handle, ContentHandle contentHandle)</code>	Closes the previously opened log or output content and releases the handle.

17.2.4 Creating Request Output from a Java Job

To create request output from Java, your job's code can use the Oracle Enterprise Scheduler API to import directly to the content store, or it can create files in the request output directory that are automatically imported to the content store.

Using the API, the job can create text or binary output content. Imported output files are always imported as binary content, meaning the bytes are uninterpreted.

17.2.4.1 APIs for Handling Request Output from a Java Job

The Java APIs available for handling request output include the following:

- Use the `ContentFactory` class to get instances of other classes you can use to create content and write output. See [Table 17-7](#).
- Use the `RequestOutput` class to write output. See [Table 17-8](#).
- Use the `OutputContentHelper` class to interact with the content store and with the request file directory. See [Table 17-9](#).

The `oracle.as.scheduler.request.ContentFactory` class provides methods to get your code access to the output content framework, as well as to an instance you can use to create the output itself.

Table 17-7 *ContentFactory Methods for Java Request Output*

Method	Description
<code>getRequestOutput(RuntimeServiceHandle rsh, long requestId, ContentType contentType, String contentName)</code>	Returns a <code>RequestOutput</code> instance with output for the specified request and creates the output content for the request. Each write will use the specified request service handle; your calling code is responsible for committing or rolling back the transaction.
<code>getOutputContentHelper(long requestId)</code>	Returns a <code>OutputContentHelper</code> instance for creating output content for requests with Standard or Extended request mode. Each operation will be performed in a separate transaction.
<code>getOutputContentHelper(long requestId, RuntimeServiceHandle rsh)</code>	Returns a <code>OutputContentHelper</code> instance for creating output content for requests with Standard or Extended request mode. Each operation will use the provided handle, and it is the caller's responsibility to commit or rollback the transaction.

The `oracle.as.scheduler.request.RequestOutput` class represents the output your code is creating. You get an instance of this class from `ContentFactory.getRequestOutput`, then use its write methods to add content to the output you're creating.

Table 17-8 *RequestOutput Methods for Java Request Output*

Method	Description
<code>writeln(String str)</code>	Appends <code>str</code> to the text output content, followed by a line feed character.
<code>write(String str)</code>	Appends <code>str</code> to the text output content.
<code>write(String str, int offset, int length)</code>	Appends <code>str</code> to the text output content.
<code>write(char[] chars)</code>	Appends <code>chars</code> to the text output content.
<code>write(char[] chars, int offset, int length)</code>	Appends <code>chars</code> to the text output content.
<code>write(byte[] bytes)</code>	Appends <code>bytes</code> to the binary output content.

Table 17–8 (Cont.) RequestOutput Methods for Java Request Output

Method	Description
<code>write(byte[] bytes, int offset, int length)</code>	Appends bytes to the binary output content.

Methods of the `oracle.as.scheduler.request.OutputContentHelper` class do the heavy lifting for output handling in Java jobs. Using these methods, your code can work with the request file directory and the content store itself.

Note that methods for importing content to the content store take a `OutputContentHelper.CommitSemantics` enum instance that you can use to specify transaction semantics during import. For more information, see [Table 17–10](#).

Table 17–9 OutputContentHelper Methods for Java Request Output

Method	Description
<code>workDirectoryExists()</code>	<p>Returns true if the request's work directory exists. Allows the job at any stage to determine if the work directory exists before it attempts to create temporary files.</p> <p>The job must define the <code>SYS_EXT_supportOutputFiles</code> system property with a value of "work" or "output" to cause Oracle Enterprise Scheduler to create the work directory.</p>
<code>outputDirectoryExists();</code>	<p>Returns true if the request's output directory exists. Allows the job at any stage, such as update, to determine if the output directory exists before it attempts to create output files.</p> <p>The job must define the <code>SYS_EXT_supportOutputFiles</code> system property with a value of "output" to cause Oracle Enterprise Scheduler to create the output directory.</p>
<code>isRequestWorkDirectoryShared();</code>	Returns true if the request file directory is shared. If it is, then any files created in the request work dir or output dir in any stage will be available to all subsequent stages of the request.
<code>getResolvedWorkDirectory();</code>	Returns a String with the request work directory as resolved to the current server. The job may create temporary files in the work directory, and Oracle Enterprise Scheduler will automatically delete the work directory at the end of request execution if the <code>RequestFileDirectory</code> is shared, or at the end of each stage (pre-processing, execution, update, post-processing) if the <code>RequestFileDirectory</code> is local.
<code>getResolvedOutputDirectory();</code>	Returns a String with the request output directory, resolved to the current server. The job may create output files in the output directory that can be automatically or manually imported to the Oracle Enterprise Scheduler content store.
<code>importOutputFiles(List<String> fileNames, CommitSemantics semantics);</code>	Returns an <code>ImportExportResult</code> instance from importing the specified files from the resolved output directory. Imported content overwrites existing content of the same name, unless the existing content was created using the API. In that case, the file will not be imported.

Table 17–9 (Cont.) OutputContentHelper Methods for Java Request Output

Method	Description
importOutputFiles(CommitSemantics semantics);	Returns an ImportExportResult instance from importing all files from the resolved output directory. Imported content overwrites existing content of the same name, unless the existing content was created using the API. In that case, the file will not be imported.
exportOutputContent(List<String> contentNames);	Returns an ImportExportResult from exporting the specified previously imported output content to files in the request output directory. The exported files will overwrite any existing files of the same names. Note that output content created using the API can not be exported.
exportOutputContent();	Returns an ImportExportResult instance from exporting all previously imported output content to files in the request output directory. The exported files will overwrite any existing files of the same names. Note that output content created using the API can not be exported.
queryOutputContent();	Returns a List of ContentDetail instances with detailed information for all existing output content in the content store. This returns information on both output content that was imported and output content created using the API.
queryOutputContent(String contentName);	Returns a ContentDetail instance with detailed information for the output content in the content store, if it exists (null if it doesn't). This returns information on both output content that was imported and output content created using the API.
outputContentExists(String contentName);	Returns true if the specified output content exists in the content store for the request. This returns information on output content that was imported and output content created using the API.
deleteOutputContent(List<String> contentNames);	Deletes the specified output content from the content store for the request. Can delete output content that was imported and output content created using the API.

Use the oracle.as.scheduler.request.OutputContentHelper.CommitSemantics enum to specify what should happen if errors occur while importing content to the content store.

Table 17–10 CommitSemantics Enum Members to Express Commit Semantics

Field	Description
StopOnFirstError	Stop the operation for all files when there is an error on a file. If the handle is internal, it will be committed.
IgnoreErrors	Attempt the operation on all files regardless of errors. If the handle is internal, it will be committed.
Transactional	Stop the operation for all files when any file has an error. This is not valid with a user-provided handle.

17.2.4.2 Example

The following example illustrates how to create an output file in the request output directory. Remember that the job must define the `SYS_EXT_supportOutputFiles` system property as "output". This example is appropriate for a Java job, an asynchronous Java job, a pre-processor, or a post-processor.

Example 17-5 Creating an Output File

```
import oracle.as.scheduler.request.ContentFactory;
import oracle.as.scheduler.request.OutputContentHelper;

class ExampleOutputCreator{

    OutputContentHelper helper = ContentFactory.getOutputContentHelper(requestId);
    String outputDir = helper.getResolvedOutputDirectory();

    File f = new File(outputDir, "myfile");
    f.createNewFile();
    if (f.exists())
    {
        // write to file
    }
}
```

The following example shows how to manually export and import output files. This would be useful if you need to create content from files during update. Be aware that you can export only files that have been imported and not files that were created using the API.

The example illustrates the scenario that a file that may have been created previously needs to be updated and imported.

Example 17-6 Manual Export and Import of Request Output

```
import oracle.as.scheduler.request.ContentFactory;
import oracle.as.scheduler.request.ImportExportResult;
import oracle.as.scheduler.request.ImportExportResult.ImportExportStatus;
import oracle.as.scheduler.request.OutputContentHelper;
import oracle.as.scheduler.request.OutputContentHelper.CommitSemantics;

class ExampleExportImport{

    OutputContentHelper helper = ContentFactory.getOutputContentHelper(requestId);

    if (!helper.outputDirectoryExists())
    {
        // error - make sure job definition defines SYS_EXT_supportOutputFiles
    }

    String outputDir = helper.getResolvedOutputDirectory();
    String fileName = "myfile.out";
    List<String> fileNamesList = new ArrayList<String>();
    fileNamesList.add(fileName);

    // Export the file if it exists; otherwise, create it.

    if (helper.outputContentExists(fileName))
    {
        ImportExportResult exportResult = exportOutputContent(fileNamesList);
        if (exportResult.getStatus() != ImportExportStatus.Success)
```

```

        {
            // handle error
        }
    }
else
{
    File f = new File(outputDir, fileName);
    f.createNewFile();
}

// ... update the file as needed ...

// Import the new or updated file.
// Updated file overwrites previous contents.

ImportExportResult importResult =
    helper.importOutputFile(fileNameList, CommitSemantics.IgnoreErrors);

if (importResult.getStatus() != ImportExportStatus.Success)
{
    // handle error
}
}

```

17.2.5 Creating Request Output from a PL/SQL Job

To create request output from PL/SQL, your code can use the ESS_JOB PL/SQL package to create output content directly in the content store.

Using functions and procedures in the package, the job can create text or binary output content.

17.2.5.1 PL/SQL Package Support for Creating Output

Table 17–11 *ESS_JOB Procedures and Functions for Request Output*

Method	Description
open_text_output_content(p_content_name in varchar2) return varchar2	Returns a handle from opening the specified output content for the request associated with the current session.
open_binary_output_content(p_content_name in varchar2) return varchar2	These are convenience functions that call open_output_content with the appropriate content type constant. See open_output_content for additional details.

Table 17–11 (Cont.) ESS_JOB Procedures and Functions for Request Output

Method	Description
<pre>open_output_content(p_content_ name in varchar2, p_content_type in integer) return varchar2;</pre>	<p>Returns an opaque handle from opening the output content <code>p_content_name</code> for the request associated with the current session.</p> <p>This creates a new output content entry if one does not already exist for the given name. If one already exists, then the specified content type must match that already established for that name.</p> <p><code>p_content_type</code> represents content type with one of the content type constants:</p> <p><code>CONTENT_TYPE_TEXT</code> (value: 1) for text content.</p> <p><code>CONTENT_TYPE_BINARY</code> (value: 2) for binary content.</p> <p>This returns an opaque handle that is passed to subsequent procedures that operate on that output content.</p> <p>The content entry is locked on successful return from this function. It may or may not be locked if this procedure fails. A commit or rollback releases the lock. The <code>write_text_content</code> or <code>write_ntext_content</code> procedures must be used to write data for text content, while the <code>write_binary_content</code> procedure must be used to write data for binary content.</p> <p>You should call <code>close_content</code> to free any resources associated with the handle returned by this method. The close should be done prior to transaction commit or rollback.</p> <p>NOTE: The content output support has DML semantics. The caller is responsible for the commit/rollback.</p>
<pre>write_ntext_content(p_content_ handle in varchar2, p_data in nvarchar2)</pre>	<p>Writes data as <code>p_data</code> to the output content associated with the given handle. These operations are supported only for <code>CONTENT_TYPE_TEXT</code> content type.</p>
<pre>write_text_content(p_content_handle in varchar2, p_data in varchar2);</pre>	<p><code>p_content_handle</code> is the output handle from a prior <code>open_output_content</code> call.</p> <p>NOTE: The content output support has DML semantics. The caller is responsible for the commit/rollback.</p>
<pre>write_binary_content(p_content_ handle in varchar2, p_data in raw);</pre>	<p>Writes binary data as <code>p_data</code> to the output content associated with the given handle. This operation is supported only for <code>CONTENT_TYPE_BINARY</code> content type.</p> <p><code>p_content_handle</code> is the output handle from a prior <code>open_output_content</code> call.</p> <p>NOTE: The content output support has DML semantics. The caller is responsible for the commit/rollback.</p>

Table 17–11 (Cont.) ESS_JOB Procedures and Functions for Request Output

Method	Description
close_content(p_content_handle in varchar2);	<p>Closes the output content handle. This releases resources associated with the given handle and it is no longer valid. Call this method before transaction commit or rollback.</p> <p>p_content_handle is the output handle from a prior open_output_content call.</p> <p>NOTE: The content output support has DML semantics. The caller is responsible for the commit/rollback. This method does not automatically perform a commit or rollback</p>
output_content_exists(p_content_name in varchar2) return boolean;	<p>Returns true if an output content entry having the specified name already exists for the request associated with the current session.</p> <p>p_content_name is the name of the output content entity.</p>
delete_output_content(p_content_name in varchar2);	<p>Deletes the specified output content entry for the request associated with the current session.</p> <p>p_content_name is the name of the output content entity.</p>

17.2.5.2 PL/SQL Output Creation Examples

[Example 17–7](#) illustrates how to write text content into output, as well as how to write log entries along the way.

Example 17–7 PL/SQL Request Text Output

```

create or replace procedure text_output_example_job
( request_handle in varchar2 )
as
  v_request_id number := null;
  v_content_name varchar2(100) := 'mycontent.txt';
  v_content_handle varchar2(100);
  v_nvarchar2 nvarchar2(100);
begin
  ess_job.write_log(ess_job.LEVEL_FINE,
                   'TEXT_OUTPUT_EXAMPLE_JOB Procedure Begin');

  -- Oracle Enterprise Scheduler request id being executed.
  begin
    v_request_id := ess_runtime.get_request_id(request_handle);
  exception
    when others then
      ess_job.write_log(ess_job.LEVEL_SEVERE,
                       'Bad request handle: '||request_handle);
      raise_application_error(-20000,
                              'Failed to get request id for request handle '||request_handle,
                              true);
  end;

  begin
    -- -----
    -- Delete content entry if it already exists.
    -- -----
    if (not ess_job.output_content_exists(v_content_name)) then
      ess_job.write_log(ess_job.LEVEL_FINEST,

```



```

                                'Content does not exist: ' || v_content_name);
else
    ess_job.write_log(ess_job.LEVEL_INFO,
                    'Deleting existing content: ' || v_content_name);
    ess_job.delete_output_content(v_content_name);
    commit;
end if;

-- -----
-- Write text content. Source data has some non-ascii chars.
-- Illustrate multiple writes.
-- -----
ess_job.write_log(ess_job.LEVEL_FINE,
                'Write text content: ' || v_content_name);

v_content_handle := null;
v_content_handle := ess_job.open_text_output_content(v_content_name);

ess_job.write_text_content(v_content_handle,
                        'Data ');
ess_job.write_ntext_content(v_content_handle,
                        unistr('NTEXT data:\00c4\00c5'));
ess_job.write_text_content(v_content_handle,
                        ' for CONTENT' || v_content_name);

ess_job.close_content(v_content_handle);
v_content_handle := null;
commit;
exception
when others then
    ess_job.write_log(ess_job.LEVEL_WARNING,
                    'Error during text output operations. ' ||
                    'content: ' || v_content_name || chr(10) ||
                    'Error_Stack...' || chr(10) ||
                    dbms_utility.format_error_stack() || chr(10) ||
                    'Error_Backtrace...' || chr(10) ||
                    dbms_utility.format_error_backtrace());
    if v_content_handle is not null then
        ess_job.close_content(v_content_handle);
        v_content_handle := null;
    end if;
    rollback;
    raise_application_error(-20000,
        'Output content operations failed for ' || v_content_name);
end;

ess_job.write_log(ess_job.level_info,
                'TEXT_OUTPUT_EXAMPLE_JOB Procedure End');
end;
/

```

[Example 17-8](#) illustrates how to write binary content for output.

Example 17-8 PLSQL Request Binary Output Example

```

create or replace procedure binary_output_example_job
( request_handle in varchar2 )
as
    v_request_id number := null;
    v_content_name varchar2(100) := 'mycontent.bin';

```

```

v_content_handle varchar2(100);
v_nchar_cs varchar2(100);
v_dest_cs varchar2(100);
v_ntext nvarchar2(100);
v_raw raw(500);
begin
  ess_job.write_log(ess_job.LEVEL_FINE,
    'BINARY_OUTPUT_EXAMPLE_JOB Procedure Begin');

  -- Oracle Enterprise Scheduler request id being executed.
  begin
    v_request_id := ess_runtime.get_request_id(request_handle);
  exception
    when others then
      ess_job.write_log(ess_job.LEVEL_SEVERE,
        'Bad request handle: ' || request_handle);
      raise_application_error(-20000,
        'Failed to get request id for request handle ' || request_handle,
        true);
  end;

  begin
    -- -----
    -- Delete content entry if it already exists.
    -- -----
    if (not ess_job.output_content_exists(v_content_name)) then
      ess_job.write_log(ess_job.LEVEL_FINEST,
        'Content does not exist: ' || v_content_name);
    else
      ess_job.write_log(ess_job.LEVEL_INFO,
        'Deleting existing content: ' || v_content_name);
      ess_job.delete_output_content(v_content_name);
      commit;
    end if;

    -- -----
    -- Write binary content.
    -- This will be UTF-8 representation of a string for a known byte
    -- encoding rather than whatever the charset/national charset
    -- happens to be for this database.
    -- Source data has couple non-ascii chars.
    -- -----

    -- database national character set being used
    select value into v_nchar_cs
      from nls_database_parameters
      where parameter = 'NLS_NCHAR_CHARACTERSET';
    ess_job.write_log(ess_job.LEVEL_FINEST,
      'NLS_NCHAR_CHARACTERSET = ' || v_nchar_cs);

    ess_job.write_log(ess_job.LEVEL_FINE,
      'Write binary content: ' || v_content_name);

    v_content_handle := null;
    v_content_handle := ess_job.open_binary_output_content(v_content_name);

    v_ntext := unistr('Data (NTEXT data:\00c4\00c5) for CONTENT ' ||
      v_content_name);

    v_dest_cs := 'AL32UTF8';

```

```

v_raw := utl_raw.cast_to_raw(convert(v_ntext, v_dest_cs, v_nchar_cs));
ess_job.write_binary_content(v_content_handle, v_raw);
ess_job.write_log(ess_job.LEVEL_FINE,
                  'Wrote ' || utl_raw.length(v_raw) || ' bytes' ||
                  ' using ' || v_dest_cs || ' charset');

ess_job.close_content(v_content_handle);
v_content_handle := null;
commit;
exception
when others then
    ess_job.write_log(ess_job.LEVEL_WARNING,
                    'Error during binary output operations. ' ||
                    'content_name=' || v_content_name || chr(10) ||
                    'Error_Stack...' || chr(10) ||
                    dbms_utility.format_error_stack());
    if v_content_handle is not null then
        ess_job.close_content(v_content_handle);
        v_content_handle := null;
    end if;
    rollback;
    raise_application_error(-20000,
        'Output content operations failed for ' || v_content_name);
end;

ess_job.write_log(ess_job.level_info,
                  'BINARY_OUTPUT_EXAMPLE_JOB Procedure End');
end;
/

```

17.2.6 Creating Request Output from a Process Job

You create output from a process job logic by writing the content to the location specified by the `ESS_OUTPUT_WORK_DIR` environment variable that is available for all process jobs. As with other jobs, ensure that the `SYS_EXT_supportOutputFiles` system property is set to "output" so that the environment variable will be defined for the job.

After your process code writes the file, Oracle Enterprise Scheduler will automatically import output files in the directory into the content store as binary content.

Oracle Enterprise Scheduler Security

This chapter describes Oracle Enterprise Scheduler Security security features that provide access control for its resources and application identity propagation for job execution.

- [Section 18.1, "Introduction to Oracle Enterprise Scheduler Security"](#)
- [Section 18.2, "Configuring Metadata Security for Oracle Enterprise Scheduler"](#)
- [Section 18.3, "Configuring Web Service Security for Oracle Enterprise Scheduler"](#)
- [Section 18.4, "Configuring PL/SQL Job Security for Oracle Enterprise Scheduler"](#)
- [Section 18.5, "Elevating Privileges for Oracle Enterprise Scheduler Jobs"](#)
- [Section 18.6, "Configuring a Single Policy Stripe in Oracle Enterprise Scheduler"](#)

18.1 Introduction to Oracle Enterprise Scheduler Security

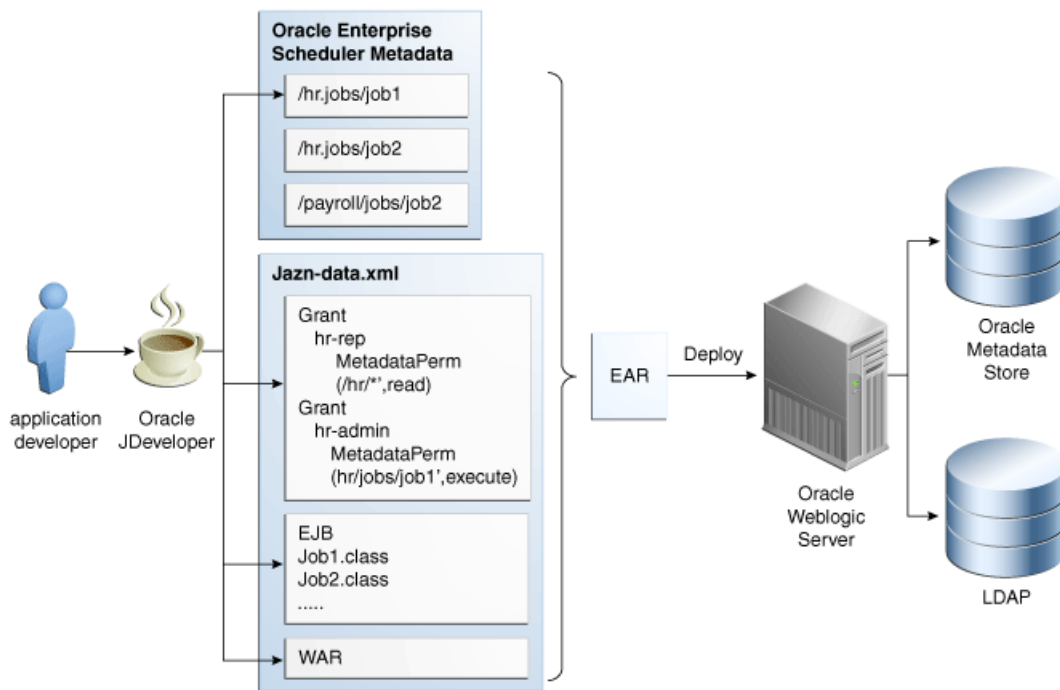
Oracle Enterprise Scheduler Security includes the following:

- 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 18.1.1, "Oracle Enterprise Scheduler Metadata Access Control."](#)
- 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 18.1.2, "Oracle Enterprise Scheduler Job Execution Security."](#)

18.1.1 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 18–1](#) shows the metadata security summary.

Figure 18–1 Design Time Metadata Security for Oracle Enterprise Scheduler



18.1.2 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_APPLICATIONID`, however, the job runs under the elevated privileges of an application ID. For more information, see [Section 18.5, "Elevating Privileges for Oracle Enterprise Scheduler Jobs."](#)

18.2 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 `calculateFees`, Oracle Enterprise Scheduler ensures that `teller1` has `READ` permission for the Metadata object `calculateFees` 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.

18.2.1 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 5.6.3, "How to Assemble the EAR File for Oracle Enterprise 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 5.6.3, "How to Assemble the EAR File for Oracle Enterprise 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.

18.2.2 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**.

18.2.3 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.

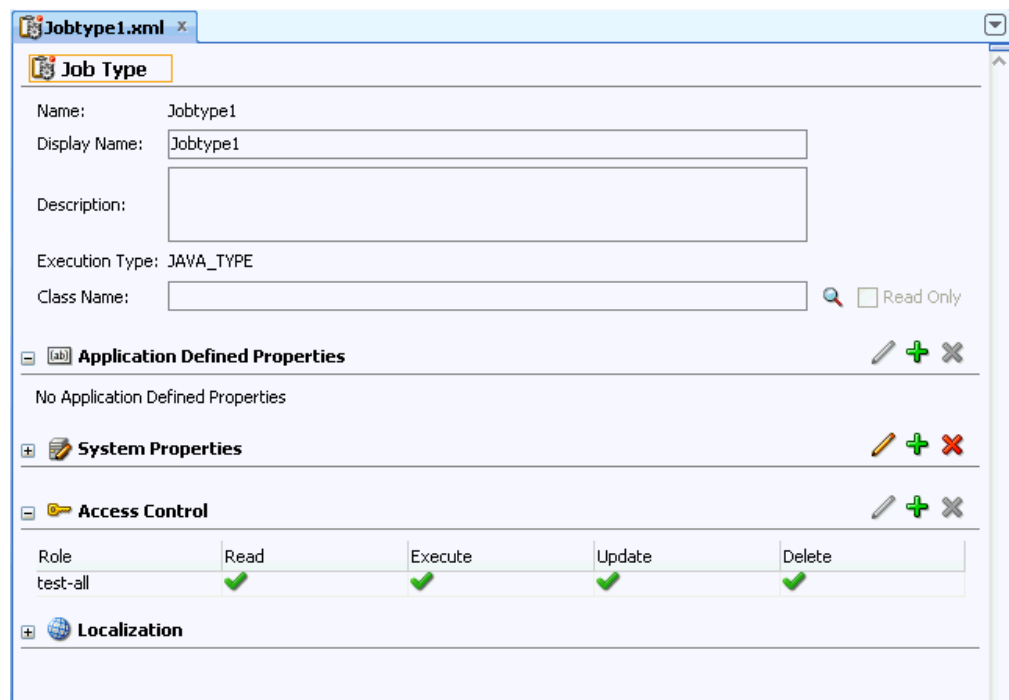
First, create any required Oracle Enterprise Scheduler Metadata in an application using **File > New > Business Tier > Enterprise Scheduler Metadata**. For more information on creating Metadata, see [Section 5.5, "Creating Metadata for Oracle Enterprise Scheduler Sample Application."](#)

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 18–2](#).
6. Repeat for as many roles as needed.

Figure 18–2 Security Roles for Oracle Enterprise Scheduler Metadata



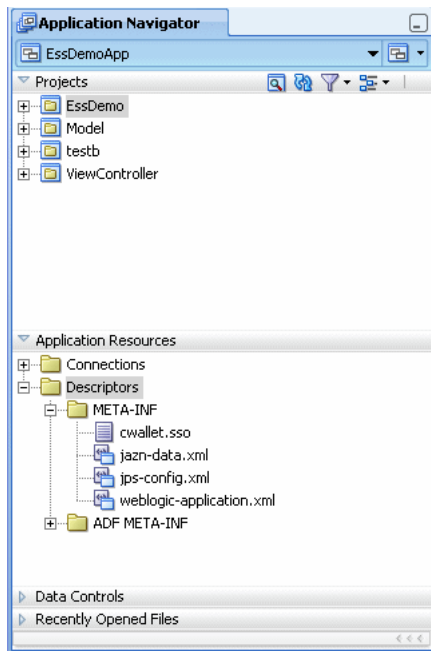
18.2.4 How to Create Grants with Oracle ADF Security Wizard

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 18–3](#).

Figure 18–3 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 18–1](#) 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 18–1 Sample Permission Grants for Security Using Oracle ADF

Name	Actions	Effect
<code>package-part.JobDefinition</code> <code>.MyJavaSucJobDef</code>	EXECUTE	Grants the ability to submit requests for a single Metadata item.
<code>mypackage.subpackage.*</code>	CREATE,EXECUTE	Grants to ability to create and execute any new Metadata items in <code>/mypackage/subpackage</code>
<code>JobDefinition.SYS_</code> <code>AdHocRequest</code>	CREATE,EXECUTE	Grants ad hoc submission permission
<code>mypackage.*</code>	CREATE,EXECUTE,DELETE	Grants wide-open permissions

18.2.5 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 `MetadataObjectId#toPermissionString()`.

Table 18–2 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 18–2 Grant Actions for Metadata Security

Action	Implies	Metadata Functions
READ	None	<code>get<Type>()</code> , <code>query<Type>()</code>
EXECUTE	READ	<code>submitRequest()</code>
CREATE	READ	<code>add<Type>()</code>
UPDATE	READ	<code>update<Type>()</code>
DELETE	READ	<code>delete<Type>()</code>

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.

18.2.6 What Happens When You Configure Metadata Security

Each time a user application calls a `MetadataService` 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`.

18.3 Configuring Web Service Security for Oracle Enterprise Scheduler

For information about securing the Oracle Enterprise Scheduler web service, see [Section 11.9, "Securing the Oracle Enterprise Scheduler Web Service."](#)

18.4 Configuring PL/SQL Job Security for Oracle Enterprise Scheduler

The PL/SQL job does not enforce data security checks when calling `ess_runtime` package apis.

18.5 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.

18.6 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.

18.6.1 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 18–1](#) shows an `applicationStripe` configuration for the policy context `ESS_FUNCTIONAL_TEST_APP_STRIPE`.

Example 18–1 *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>
```

```
</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 18–2](#) shows a code sample.

Example 18–2 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>
```

18.6.2 What Happens When You Configure a Single Policy Stripe

At design time, an application stripe manifests as:

- An `<application>` element under the `<polycystore>` element in the `jazn-data.xml` file.
- A node under the node `cn=<Weblogic.domain.name>, cn=JPSText, cn=<root.node>`, such as `cn=ATGDemo, cn=base_domain, cn=JPSText, cn=MY_Node`.

18.6.3 What Happens at Runtime

At run time, an application stripe manifests as an instance of the class `oracle.security.jps.service.polycystore.ApplicationPolicy`.