Oracle® Fusion Middleware

Developing Applications for Oracle Enterprise Scheduler

12*c* (12.1.3)

E28544-03

July 2015

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

ORACLE®

Oracle Fusion Middleware Developing Applications for Oracle Enterprise Scheduler , 12c (12.1.3)

E28544-03

Primary Author: Oracle Corporation

Contributors: Kirk Bittler, Weifeng BaoOracle Fusion Middleware Developing Applications for Oracle Enterprise Scheduler, Shelly Butcher, David Craft, Diane Davison, Carlos Fuentes, Charles Hall, Vaibhav Lole, Solomon Nelson, Shengsong Ni, Rachna Shukla, Steve Traut, Venkat Vengala, Aaron Weisberg, Larry Hoffman

Contents

Preface xiii

Audience xiii
Documentation Accessibility xiii
Related Documents xiii
Conventions xiv

What's New in This Guide xv

1 Introduction to Oracle Enterprise Scheduler

About Oracle Enterprise Scheduler 1-1
Oracle Enterprise Scheduler Overview for Application Developers 1-2
Introduction to Working with Oracle Enterprise Scheduler at Design-Time 1-2
Introduction to Working with Oracle Enterprise Scheduler at Runtime 1-3
Oracle Enterprise Scheduler Job Requests 1-4
Overview of Integration Steps 1-6
Fixed-Rate Scheduling with Oracle Enterprise Scheduler 1-6

2 Planning Job Development

Job Development Flow 2-1
The Hosting Application 2-3
The Client Application 2-3
Create the Job Implementation 2-4
Create Job Metadata 2-4
Automatic Metadata Refresh Post-Submission 2-4

3 Installing and Verifying the Oracle Enterprise Scheduler Installation

Installing Oracle Enterprise Scheduler 3-1
Targeting Oracle Enterprise Scheduler During Domain Creation 3-1
OWSM-PM Targeting With Oracle Enterprise Scheduler 3-2
Introduction to Verifying the Oracle Enterprise Scheduler Installation 3-2
How to Verify the Oracle Enterprise Scheduler Installation Using a Browser 3-3

How to Programmatically Verify the Oracle Enterprise Scheduler Installation 3-3
What Happens at Runtime: How the Oracle Enterprise Scheduler Installation is Verified 3-4

4 Using the Pre-Deployed Native Hosting Application

Introduction 4-1
Properties 4-2
Metadata 4-2
Security Permissions 4-2
Configuring the Policy Stripe 4-2
Support for Multiple Application Stripes 4-3

5 Using Ant to Generate a Hosting Application

Introduction to Generating a Hosting Application with Ant 5-1
Prerequisites for Using the Ant Build Files 5-2
Ant Targets for Creating and Deploying a Hosting Application 5-2
Creating a Hosting Application and Project Workspace with Ant 5-3
Creating a Java Job as a Shared Library with Ant 5-7
Packaging a Java Job as a Shared Library with Ant 5-9
Deploying a Shared Library with Ant 5-10
Packaging a Hosting Application with Ant 5-10
Deploying a Hosting Application with Ant 5-10
Configuring the Generated Ant Targets 5-11

6 Creating a Thin Client Application

Introduction 6-1
Implementation 6-2
Secured Invocation 6-3
RemoteConnector API and the Server Affinity Property 6-4
Examples 6-4
Using JDeveloper to Build a Thin Client Application for MAR Deployment 6-6
Create and Deploy a Thin Client Application for the Standalone Environment 6-6
Using JDeveloper to Create and Configure an EJB and its Job Definition Metadata 6-16

7 Using Oracle JDeveloper to Generate an Oracle Enterprise Scheduler Application

How to Start JDeveloper to Support Building Oracle Enterprise Scheduler Applications 7-1
Understanding Oracle Enterprise Scheduler Application Support Created by Oracle JDeveloper 7-2
Building a Combined Oracle Enterprise Scheduler Application 7-3
Creating the Application and Projects for EssDemoApp Application 7-4
Creating Metadata and an Implementation Class for the EssDemoApp Application 7-7
Adding Application Code to Submit Job Requests 7-10
Setting Oracle Enterprise Scheduler Properties 7-11
Assembling the EssDemoApp Application 7-12

Deploying and Running the EssDemoApp Application 7-21
Building Split Submitting and Hosting Applications 7-24
How to Create the Back-End Hosting Application for EssDemoApp 7-25
How to Create the Front-End Submitter Application for Oracle Enterprise Scheduler 7-35

8 Using the Metadata Service

Introduction to Using the Metadata Service 8-1
Introduction to Metadata Service Name Spaces 8-2
Introduction to Metadata Service Operations 8-2
Introduction to Metadata Service Transactions 8-3
Accessing the Metadata Service 8-3
How to Access the Metadata Service with a Stateless Session EJB 8-3
Accessing the Metadata Service with Oracle JDeveloper 8-4
Querying Metadata Using the Metadata Service 8-4
How to Create a Filter 8-4
How to Query Metadata Objects 8-5

9 Using Parameters and System Properties

Introduction to Using Parameters and System Properties 9-1
What You Need to Know About Application Defined Property and System Property Naming 9-1
What You Need to Know About Parameter Conflict Resolution and Parameter Materialization 9-2
Using Parameters with the Metadata Service 9-4
How to Use Parameters and System Properties in Metadata Objects 9-5
Using Parameters with the Runtime Service 9-6
How to Use Parameters with the Runtime Service 9-6
How to Use Parameters with a Step ID for Job Set Steps 9-7
Using System Properties 9-8

10 Using Tokens and Logical Clusters

Using Token Substitution 10-1
Nested Substitutions 10-2
Automatic Substitution 10-2
Using Logical Clusters 10-3

11 Creating and Using PL/SQL Jobs

Introduction to Using PL/SQL Stored Procedure Job Definitions 11-1
Creating a PL/SQL Stored Procedure for Oracle Enterprise Scheduler 11-2
How to Define a PL/SQL Stored Procedure with the Correct Signature 11-2
Handling Runtime Exceptions in an Oracle Enterprise Scheduler PL/SQL Stored Procedure 11-3
How to Access Job Request Information In PL/SQL Stored Procedures 11-4

What You Need to Know When You Define a PL/SQL Stored Procedure 11-4
Performing Oracle Database Tasks for PL/SQL Stored Procedures 11-4
How to Grant PL/SQL Stored Procedure Permissions 11-4
What You Need to Know About Granting PL/SQL Stored Procedure Permissions 11-5
Creating and Storing Job Definitions for PL/SQL Job Types 11-6
How to Create a PL/SQL Job Type 11-6
How to Create and Store a Job Definition for PL/SQL Job Type 11-7
Using a PL/SQL Stored Procedure with an Oracle Enterprise Scheduler Application 11-8

12 Creating and Using EJB Jobs

Introduction to Creating EJB Jobs 12-1
Planning Job Development 12-2
Creating and Storing Job Definitions for EJB Job Types 12-2
Secured Invocation 12-4
Forward Invocation 12-4
Callback Invocation 12-5
RemoteConnector API and the Server Affinity Property 12-5
CSF Lookup From a Remote Server 12-6
Synchronous Bean 12-6
Metadata 12-6
EJB Job Sample Code 12-7
Asynchronous Bean 12-8
Metadata 12-9
EJB Job Sample Code 12-10

13 Creating and Using Web Service Jobs

Introduction 13-1
Predefined Web Service Job Types 13-2
Cancel and Fault Support 13-3
Configuration Properties for Web Service Jobs 13-4
Oracle Web Services Manager Policy Configuration 13-5
Creating a Web Service Job Definition 13-6
Using Oracle JDeveloper to Create a Job Definition 13-6
Using Oracle Enterprise Manager Fusion Middleware Control to Create a Job Definition 13-10

14 Creating and Using Process Jobs

Introduction to Creating Process Job Definitions 14-1
Creating and Storing Job Definitions for Process Job Types 14-1
How to Create and Store a Process Job Type 14-2
How to Create and Store a Process Type Job Definition 14-4
Using an Agent Handler for Process Jobs 14-5
Choosing an Agent Handler 14-5
Process Job Locale 14-6

15 Defining and Using Schedules

Introduction to Schedules 15-1
Defining a Recurrence 15-1
How to Define a Recurrence with a Recurrence Fields Helper 15-2
How to Define a Recurrence with an iCalendar RFC 2445 Specification 15-4
What You Need to Know When You Use a Recurrence Fields Helper 15-5
What You Need to Know When You Use an iCalendar Expression 15-6
Defining an Explicit Date 15-6
How to Define an Explicit Date 15-6
What You Need to Know About Explicit Dates 15-7
Defining and Storing Exclusions 15-7
How to Define an Exclusion 15-7
How to Create an Exclusions Definition 15-7
Defining and Storing Schedules 15-8
How to Define and Store a Schedule 15-8
What Happens When You Define and Store a Schedule 15-8
What You Need to Know About Handling Time Zones with Schedules 15-9
Identifying Job Requests That Use a Particular Schedule 15-9
Updating and Deleting Schedules 15-10

16 Using the Oracle Enterprise Scheduler Web Service

Introduction to the Oracle Enterprise Scheduler Web Service 16-1
Developing and Using ESSWebservice Applications 16-3
How to Develop and Use an ESSWebservice Java EE Application 16-3
How to Develop and Use an ESSWebservice SOA Application with BPEL 16-4
Setting Web Service Addressing Headers for getCompletionStatus() Operation 16-4
Restrictions When Using ESSWebservice 16-4
ESSWebservice Implementation 16-5
ESSWebservice WSDL File 16-5
Use Case: Using Oracle Enterprise Scheduler ESSWebservice from a BPEL Process 16-5

17 Defining and Using Job Sets

Introduction to Defining and Using Job Sets 17-1
Defining Job Sets 17-2
How to Define a Job Set 17-2
How to Define Serial Job Set Steps 17-4
How to Define Parallel Job Set Steps 17-6
What Happens When You Define a Job Set 17-7
What You Need to Know About Serial Job Sets 17-7
What You Need to Know About Job Set Application Defined Properties and System Properties 17-8
What Happens at Runtime for Job Set State Priorities and State Transitions 17-8

Cross Application Job Sets 17-10
Overview of Cross Application Job Sets 17-11
Requirements for Cross Application Job Sets 17-11
Supporting Input and Output Forwarding in Job Sets 17-12

18 Defining and Using a Job Incompatibility

Introduction to Using a Job Incompatibility 18-1
Job Self Incompatibility 18-2
Defining Incompatibility with Oracle JDeveloper 18-2
How to Define a Global Incompatibility 18-3
How to Define a Domain Incompatibility 18-4
What Happens at Runtime to Handle Job Incompatibility 18-6
What Happens to Subrequests with an Incompatible Parent Request 18-6

19 Using the Runtime Service

Introduction to the Runtime Service 19-1
Accessing the Runtime Service 19-1
How to Access the Runtime Service and Obtain a Runtime Service Handle 19-2
Submitting Job Requests 19-3
How to Submit a Request to the Runtime Service 19-3
What You Should Know About Default System Properties When You Submit a Request . 19-3
What You Should Know About Metadata When You Submit a Request 19-4
DMS ECID and FlowId Support 19-4
Managing Job Requests 19-5
How to Get Job Request Information with getRequestDetail 19-6
How to Change Job Request State 19-6
How to Update Job Request Priority and Job Request Parameters 19-8
Querying Job Requests 19-8
Submitting Ad Hoc Job Requests 19-11
How to Create an Ad Hoc Request 19-11
What Happens When You Create an Ad Hoc Request 19-13
What You Need to Know About Ad Hoc Requests 19-13
Implementing Pre-Process and Post-Process Handlers 19-13
Implementing a Pre-Process Handler 19-13
Implementing a Post-Process Handler 19-14

20 Using Subrequests

Introduction to Using Subrequests 20-1
Creating and Managing Subrequests 20-2
How to Submit Subrequests 20-2
How to Cancel Subrequests 20-2
How to Hold Subrequests 20-3
How to Submit Multiple Subrequests 20-3

How to Manage Paused Subrequests 20-3
How Subrequests Are Processed 20-4
How to Identify Subrequests 20-5
How to Manage Subrequests and Incompatibility 20-5
Creating a Java Procedure that Submits a Subrequest 20-5
Creating a PL/SQL Procedure that Submits a Subrequest 20-8

21 Working with Asynchronous Java Jobs

Introduction to Working with Asynchronous Java Jobs 21-1
Creating an Asynchronous Java Job 21-1
Implementing the Asynchronous Java Job Asynchronous Interface 21-2
Asynchronous Java Job execute() Method 21-2
Invoking a Remote Job from an Asynchronous Java Job 21-2
Calling Back to Oracle Enterprise Scheduler with Status Updates 21-3
Updating the Asynchronous Java Job 21-3
Notifying Oracle Enterprise Scheduler When an Asynchronous Job Completes 21-3
Asynchronous Java Job AsyncCancellable Interface 21-5
Sample Asynchronous Java Job Invoking a BPEL Process Through Event Delivery Network 21-5
A Use Case Illustrating the Implementation of a BPEL Process as an Asynchronous Job 21-10
Introduction to the Recommended Design Pattern 21-11
Potential Approaches 21-11
Use Case Summary 21-11
How to Implement BPEL with an Asynchronous Job 21-12
Use Case: Add Oracle JDeveloper Libraries 21-12
Use Case: Create the Asynchronous Job Definition 21-13
Use Case: Design the Event Payload Schema and Event Definition Files 21-14
Programmatically Raise a Business Event from the Asynchronous Job Methods 21-15
Design the SOA Composite with Meditator and BPEL 21-17
Add Fault Handling and Correlated onMessage Branch for Error and Cancel Job 21-18
Validating the Deployment 21-23
Troubleshooting the Use Case 21-25
Handling Time Outs and Recovery for Asynchronous Jobs 21-25
Asynchronous Request Time Outs 21-25
Handling Asynchronous Jobs Marked for Manual Recovery 21-26
Using RecoverRequest to Manually Recover a Job Request 21-27
Oracle Enterprise Scheduler Interfaces and Classes 21-28

22 Job Request Logs and Output

Request Logs 22-1
System Properties 22-1
Log Header 22-1
Request Logging from a Java Job 22-2

Request Logging from a PL/SQL Job 22-4
Request Logging from a Process Job 22-5
Request Logging and Output From an EJB Job 22-5
Request Logging from a Web Service Job 22-10
APIs for Handling Request Logs 22-11
Request Output 22-11
Using the Request File Directory 22-12
System Properties 22-14
Creating Request Output from a Java Job 22-14
Creating Request Output from a PL/SQL Job 22-20
Creating Request Output from a Process Job 22-24
Creating Request Output from an EJB Job 22-25
Creating Request Output from a Web Service Job 22-25
APIs for Handling Request Output 22-25

23 Oracle Enterprise Scheduler Security

Introduction to Oracle Enterprise Scheduler Security 23-1
Oracle Enterprise Scheduler Metadata Access Control 23-1
Oracle Enterprise Scheduler Job Execution Security 23-2
Configuring Metadata Security for Oracle Enterprise Scheduler 23-2
How to Enable Application Security with Oracle ADF Security Wizard 23-3
Including Security Files in EAR File 23-3
How to Define Principals for Security 23-4
Creating Enterprise Role 23-4
How to Create Grants with Oracle Enterprise Scheduler Metadata Pages 23-5
About MetadataPermission APIs 23-6
What Happens When You Configure Metadata Security 23-7
Configuring Data Security for Oracle Enterprise Scheduler 23-7
How to Change Data Security Permissions 23-7
Examples 23-10
Configuring Web Service Security for Oracle Enterprise Scheduler 23-13
Configuring PL/SQL Job Security for Oracle Enterprise Scheduler 23-13
Elevating Privileges for Oracle Enterprise Scheduler Jobs 23-13
Configuring a Single Policy Stripe in Oracle Enterprise Scheduler 23-13
How to Configure a Single Policy Stripe in Oracle Enterprise Scheduler 23-14
What Happens When You Configure a Single Policy Stripe 23-15
What Happens at Runtime 23-15

List of Tables

3-1 HTTP Response Codes ... 3-4
4-1 Pre-Deployed Native Hosting Application Properties ... 4-2
5-1 Ant Targets in the Included Build File ... 5-3
5-2 Ant Targets in the Generated Build File ... 5-3
5-3 Information Needed by the Ant Target ... 5-4
5-4 Information Needed by the Ant Target ... 5-7
5-5 Build Properties for Customizing Ant Builds ... 5-11
7-1 EJB Resources for the Front-End Submitter Application ... 7-52
8-1 Filter Comparison Operators ... 8-4
8-2 MetadataService Query Fields ... 8-5
9-1 Parameter Precedence Levels ... 9-2
9-2 ParameterInfo Parameter Properties ... 9-4
9-3 System Properties ... 9-8
10-1 EJB Job Type Automatically Substituted Properties ... 10-3
10-2 Web Services Job Type: Automatically Substituted Properties ... 10-3
10-3 Process Job Type: Automatically Substituted Properties ... 10-3
10-4 Properties Associated With a Job Location ... 10-4
11-1 Terminal States for PL/SQL Stored Procedure Results ... 11-3
11-2 Oracle Enterprise Scheduler System Properties for a PL/SQL Stored Procedure Job Type ... 11-7
12-1 EJB Job Type Properties ... 12-3
12-2 Additional Properties ... 12-4
13-1 The Predefined Web Service Job Types ... 13-3
13-2 SOAP Web Service Operation Statuses ... 13-3
13-3 Oracle SOA Suite Status Operations ... 13-3
13-4 Web Service Job Configuration Properties ... 13-4
14-1 System Properties for Process Type Jobs ... 14-2
15-1 Recurrence Field Helper Patterns ... 15-2
16-1 Summary of Operations Available with ESSWebservice ... 16-2
17-1 Job Set Step Property ... 17-2
17-2 Job Set Serial Execution Step Terminal States ... 17-4
17-3 Job Set Terminal State Transitions ... 17-9
17-4 Possible Job Set Runtime States ... 17-9
19-1 Runtime Service Default Value Fields and Corresponding System Properties ... 19-4
19-2 Runtime Service Get Request Methods ... 19-6
19-3 Runtime Service Job Request State Methods ... 19-7
19-4 Runtime Service Update Methods ... 19-8
19-5 Query Filter Fields For Querying the Runtime (Defined in Enum RuntimeService.QueryField) ... 19-9
19-6 Runtime Service Query Methods ... 19-11
19-7 Ad Hoc Request Job Definition System Properties for Job Types ... 19-11
22-1 ContentFactory Methods for Creating Request Logs ... 22-2
22-2 RequestLogger Methods for Creating Request Logs ... 22-3
22-3 ESS_JOB Functions and Procedures for Request Logging ... 22-4
22-4 RuntimeService Methods for Handling Request Logs ... 22-11
22-5 System Properties for Creating Request Output ... 22-14
22-6 ContentFactory Methods for Java Request Output ... 22-15
22-7 RequestOutput Methods for Java Request Output ... 22-15
22-8 OutputContentHelper Methods for Java Request Output ... 22-16
22-9 CommitSemantics Enum Members to Express Commit Semantics ... 22-18
22-10 ESS_JOB Procedures and Functions for Request Output ... 22-20

22-11 RuntimeService Methods for Handling Request Output..... 22-25
23-1 Grant Actions for Metadata Security..... 23-6
23-2 Condition Query Fields and Their Corresponding Request History View Column Entries..... 23-9

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 that have purchased support 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 12c Fusion Middleware documentation set:

- *WLST Command Reference for SOA Suite*
- *Installing and Configuring Oracle SOA Suite and Business Process Management*
- *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 Administering 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.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

What's New in This Guide

This guide has been updated in several ways. The following table lists the sections that have been added or changed.

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	May 2014
Chapter 2: Planning Job Development	New chapter	X
Chapter 3: Installing and Verifying the Oracle Enterprise Scheduler Installation	Added section 3.1	X
Chapter 4: Using the Pre-Deployed Native Hosting Application	New chapter	X
Chapter 5: Using Ant to Generate a Hosting Application	Updated examples 5-1 and 5-2	X
Chapter 6: Creating a Thin Client Application	New chapter	X
Chapter 10: Customizing Metadata	New chapter	X
Chapter 11: Using Tokens and Logical Clusters	New chapter	X
Chapter 13: Creating and Using EJB Jobs	New chapter	X
Chapter 14: Creating and Using Web Service Jobs	New chapter	X
Chapter 15: Creating and Using Process Jobs	Updated section 15.3	X
Chapter 24: Oracle Enterprise Scheduler Security	Updated section 24.3	X

1

Introduction to Oracle Enterprise Scheduler

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

This chapter includes the following sections:

- About Oracle Enterprise Scheduler
- Oracle Enterprise Scheduler Overview for Application Developers
- Fixed-Rate Scheduling with Oracle Enterprise Scheduler

Note:

For Oracle Enterprise Scheduler sample code, be sure to see the Oracle SOA Suite sample.

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 off load 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, binary scripts, web services and EJBs 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.

Fusion Middleware Control provides accessibility options for the pages on which you monitor and manage Oracle Enterprise Scheduler applications. Fusion Middleware Control supports screen readers and provides standard shortcut keys to support keyboard navigation. You can also view the console pages in high contrast or with large fonts for better readability. For information and instructions on configuring accessibility in Fusion Middleware Control, see "Using Oracle Fusion Middleware Accessibility Options" in *Oracle Fusion Middleware Administrator's Guide*.

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, binary process jobs, web services and EJBs
- 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.

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*.

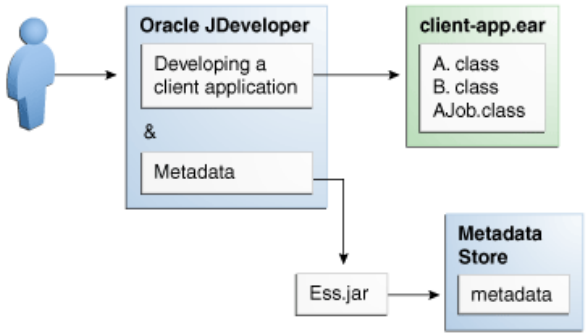
For development purposes, both Oracle Enterprise Scheduler and the Oracle Enterprise Scheduler client application are deployed on the same Oracle WebLogic Server. The Fusion Middleware Control can provide an interface for interacting with Oracle Enterprise Scheduler. Typically, however, you provide a 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.

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:

  1. The type of job to be run, such as Java, PL/SQL, binary script, and so on.
  2. 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.
  3. 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 is 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 runtime when a job request is submitted.
- **Incompatibility**: An incompatibility lets you specify job definitions and job sets that cannot run at the same time.

Introduction to Working with Oracle Enterprise Scheduler at Runtime

At runtime 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. After 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 runtime values for the parameters.

Figure 1-2 Oracle Enterprise Scheduler Runtime Integration

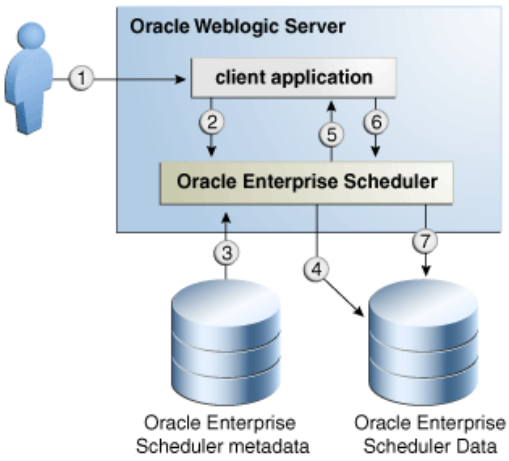


Figure 1-2 shows the sequence involved with running an application using Oracle Enterprise Scheduler, and the following steps:

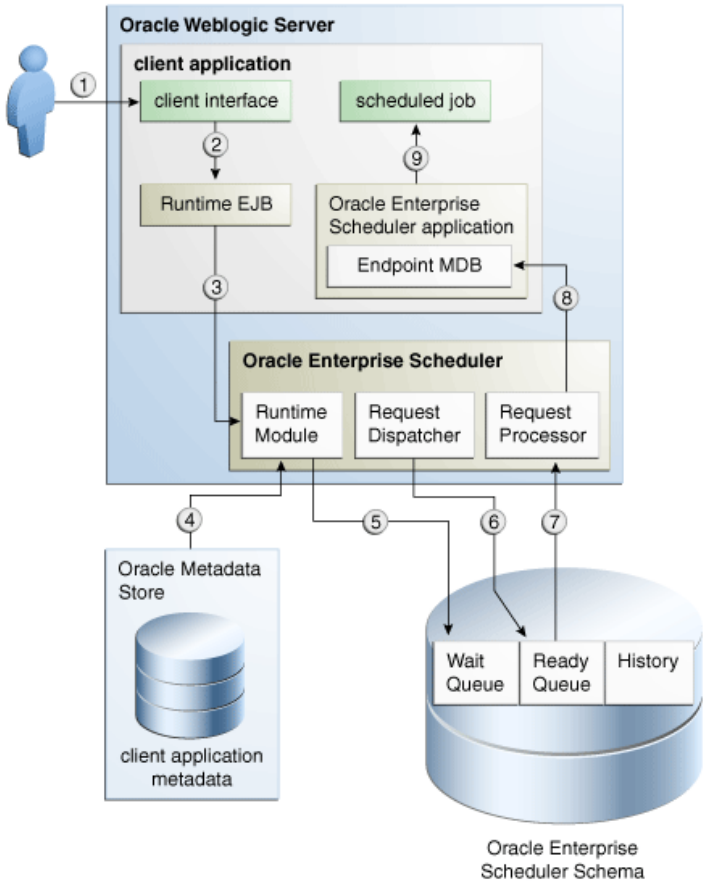
1. User submits a request using a client application.
2. Client application sends the request to Oracle Enterprise Scheduler.
3. Oracle Enterprise Scheduler reads the metadata for the request.
4. Oracle Enterprise Scheduler puts the request in a wait queue in Oracle Enterprise Scheduler data store, along 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.

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 runtime 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 runtime EJB sends the job request to the underlying runtime module in Oracle Enterprise Scheduler.
4. The runtime module accesses the client application metadata from Oracle MDS.
5. The runtime 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 is the same as the application which submitted the request.

Overview of Integration Steps

After 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.

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 are late as well. In contrast, fixed-rate scheduling tries to get things back on schedule after a late request.

2

Planning Job Development

The Oracle Enterprise Scheduler is flexible and provides implementation and deployment options. Some options use out-of-the-box components that are simpler to implement, while other options are more complex but allow for a great deal of customization. This chapter describes the different options you should consider when planning your Oracle Enterprise Scheduler deployment.

This chapter includes the following sections:

- Job Development Flow
- The Hosting Application
- The Client Application
- Create the Job Implementation
- Create Job Metadata

Job Development Flow

This section describes the steps in the job development process and describes the options available for each step. Figure 2-1 contains a diagram that shows the Oracle Enterprise Scheduler components.

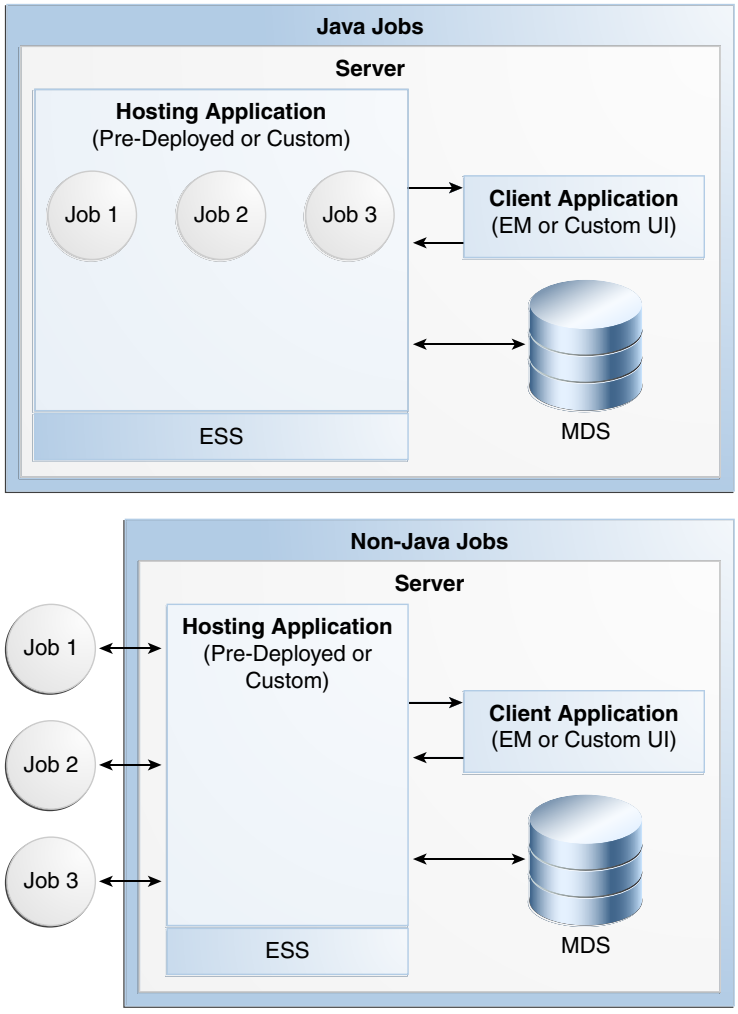
1. Create and deploy the hosting application. You have the following options:
   - Use the pre-deployed native hosting application instead of creating a hosting application.
   - Generate and deploy the hosting application from an Ant script.
   - Use JDeveloper to create the hosting application from scratch and then deploy it.
2. Create and deploy the UI or client application. You have the following options:
   - The client application uses the thin client shared library.
   - The client application uses the client library.
   - Use Oracle Enterprise Manager Fusion Middleware Control as the client application instead of creating a UI or client application.
3. Create and deploy the job implementation. You have the following options:
   - For non-Java-based jobs, you can implement and deploy the job independent of Oracle Enterprise Scheduler.

- For Java-based jobs, the Java class must be part of a custom hosting application.

4. Create job metadata. You have the following options:
   - Define the metadata in Oracle Enterprise Manager Fusion Middleware Control.
   - Use Oracle JDeveloper to create predefined/seeded job metadata for deployment as part of the hosting application.
   - Programmatically create the job metadata using the metadata service API.
5. Provide submission and metadata permissions to the deployed job. You have the following options:
   - Use Oracle Enterprise Manager Fusion Middleware Control to specify permissions.
   - Provision permissions as part of your hosting or client application EAR deployment.

   See chapter Oracle Enterprise Scheduler Security for more information.

Figure 2-1 Oracle Enterprise Scheduler Components



The Hosting Application

Jobs execute in the context of a hosting application. If the job is remote (for example, an EJB), the job is invoked in the hosting application. The pre-deployed native hosting application is convenient to use, but cannot execute custom Java jobs. The pre deployed native hosting application is well suited for custom remote jobs like EJB and web service jobs. See Using the Pre-Deployed Native Hosting Application for details about the pre-deployed native hosting application. See chapter Using Oracle JDeveloper to Generate an Oracle Enterprise Scheduler Application for more information about developing a custom hosting application using JDeveloper. See Using Ant to Generate a Hosting Application for more information about developing a custom hosting application using the Oracle Enterprise Scheduler Ant script.

The Client Application

Client applications are J2EE applications that are typically used to:

- Submit jobs
- Request status
- Read job output and logs
- Possibly, host EJB job implementations that Oracle Enterprise Scheduler can invoke remotely

Client applications can be combined with a hosting application, but this is not a best practice.

Deploying a client application to a server other than the Oracle Enterprise Scheduler server is an advanced use case and requires use of Oracle Enterprise Scheduler internal templates that are only available to other Oracle embedding products.

Client applications use the thin client shared library or the client shared library. The main differences between the two libraries are:

- The thin client shared library does not depend on the Oracle Enterprise Scheduler server or any of the Oracle Enterprise Scheduler data sources being deployed and is ideal if Oracle Enterprise Scheduler deployment is optional. The thin client shared library contacts a hosting application to access the Oracle Enterprise Scheduler metadata and runtime store to do its work.
- The client shared library has an advantage. The Oracle Enterprise Scheduler server need not be running to submit the job and query status because the library allows direct access to the Oracle Enterprise Scheduler metadata and runtime store. It is recommended for use when the client application is co-located on the same WebLogic server as the Oracle Enterprise Scheduler
- Developing client applications using the thin client shared library is easier because the client application is not required to have an `adf-config.xml` file to talk to MDS or have Oracle Enterprise Scheduler EJB deployment descriptors that the client shared library requires.

If the metadata is not automatically provisioned by the client application at deployment time, then the thin client application does not depend on the Oracle Enterprise Scheduler MDS data source.

Refer to Creating a Thin Client Application for more information about creating a client application.

Create the Job Implementation

For non-Java-based jobs (PL/SQL and binary process jobs), you can implement, setup and deploy the process binaries or PL/SQL procedures independent of Oracle Enterprise Scheduler.

For Java-based jobs, the Java implementation must conform to the Oracle Enterprise Scheduler defined interface and must be included as part of a custom hosting application.

For EJB jobs, the EJB interface must conform to an Oracle Enterprise Scheduler defined interface. The interface is in the Oracle Enterprise Scheduler shared library. See Creating and Using EJB Jobs for information about how to create an EJB job implementation.

See chapter Creating and Using Web Service Jobs for information about how to use a SOA composite as a web service job implementation.

Create Job Metadata

The simplest way to create job metadata is to define it through the Oracle Enterprise Manager Fusion Middleware Control. You can also use JDeveloper to create metadata and place it in a MAR archive that is part of a client or hosting application and then deploy the metadata to MDS when the application is deployed. SOA Suite creates the metadata programmatically on first use using the metadata APIs.

See Using the Metadata Service for a description of the metadata API.

Automatic Metadata Refresh Post-Submission

Oracle Enterprise Scheduler ensures that a job request and all of its children use a consistent snapshot of metadata from submission until the request reaches a terminal state. This is accomplished by caching, at job request submission, all metadata known to be used by the request. However, this caching prevents long-running recurring requests from using important metadata changes. In order for incompatibilities to function as expected, new and updated incompatibilities must apply to all relevant requests, whether previously or newly submitted. For job metadata, customizable parameters might have changed and should apply to previously submitted requests. For example, the request category on a job definition might have changed and this must be applied to pre-existing requests so that work allocation functions as expected.

To address these issues, Oracle Enterprise Scheduler automatically refreshes metadata for previously submitted requests that are:

- Singleton requests that have not yet run
- Recurring requests that have more recurrences to run

Cached metadata remains consistent during execution of an instance request tree that consists of the instance parent and all child requests of that instance parent, including jobset steps and sub-requests. For a singleton request, the instance request tree includes the submitted request and any child requests. For a recurring request, each recurrence is an instance request tree that includes the instance parent and any child requests of that instance parent.

3

Installing and Verifying the Oracle Enterprise Scheduler Installation

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

This chapter includes the following sections:

- Installing Oracle Enterprise Scheduler
- Introduction to Verifying the Oracle Enterprise Scheduler Installation
- How to Verify the Oracle Enterprise Scheduler Installation Using a Browser
- How to Programmatically Verify the Oracle Enterprise Scheduler Installation
- What Happens at Runtime: How the Oracle Enterprise Scheduler Installation is Verified

Installing Oracle Enterprise Scheduler

Oracle Enterprise Scheduler does not have its own installer, but is installed by the installer of the embedding product such as Oracle SOA Suite. Refer to the embedding product's installation documentation for details.

The IDE is installed by the Oracle SOA Quick Start for Developers which is described in *Installing Oracle SOA Suite and Business Process Management for Developers*. The installer installs the IDE and automatically configures it to Oracle JDeveloper. Before you run JDeveloper, make sure to set the variable `MW_HOME` to the middleware home location as required by the IDE.

The Oracle Enterprise Scheduler runtime component is installed by the design time or production installer of the embedding product (for example, Oracle SOA suite whose installation is described in *Installing and Configuring Oracle SOA Suite and Business Process Management*). The embedding product may automatically deploy Oracle Enterprise Scheduler, but if it does not, then it can be deployed using the “Oracle Enterprise Scheduler Service Basic" template to a server or cluster. The “Oracle Enterprise Manager Plugin for ESS" template can then be deployed for Oracle Enterprise Manager Fusion Middleware Control functionality.

Targeting Oracle Enterprise Scheduler During Domain Creation

When you extend SOA with Oracle Enterprise Scheduler, `ess-server1` is created and by default the ESS-MGD-SVRS server group is targeted to `ess_server1`. You can use the following steps in the FMW Configuration Wizard to re-target Oracle Enterprise Scheduler to `soa_server1`:

1. Check **ESS-MGD-SVRS** for `soa_server1`

2. Uncheck **ESS-MGD-SVRS** for `ess_server1`
3. Delete `ess_server1`

See *Creating WebLogic Domains Using the Configuration Wizard* for more information about the wizard.

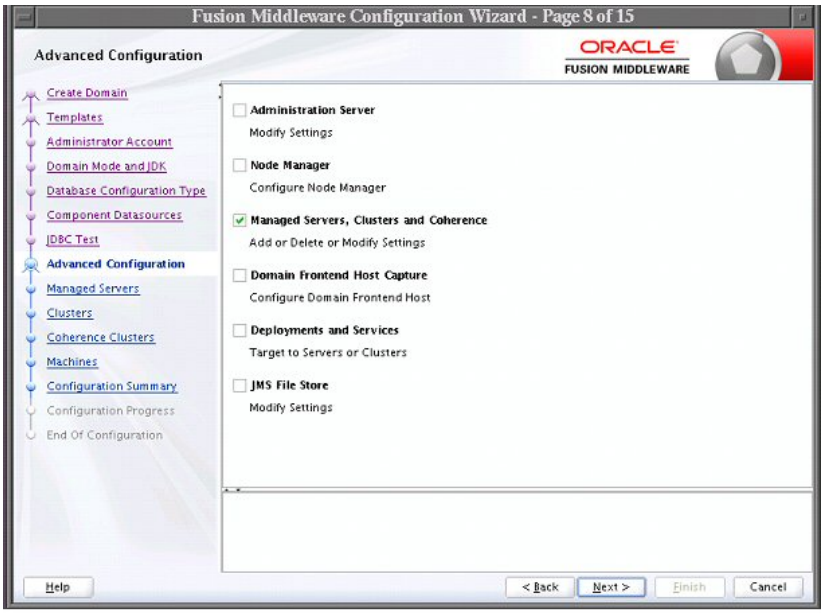
OWSM-PM Targeting With Oracle Enterprise Scheduler

OWSM-PM is intended to be targeted to just one server in a domain. To facilitate this requirement, Oracle Enterprise Scheduler templates no longer target OWSM-PM. If another product in the domain automatically targets OWSM-PM, then there is nothing to do. However, if there are no managed servers in the domain except for Oracle Enterprise Scheduler, or none of these servers has OWSM-PM, then OWSM-PM must be targeted manually.

Targeting OWSM-PM Manually

In the Fusion Middleware Configuration Wizard, select the **Managed Servers, Clusters and Coherence** check box as shown in Figure 3-1.

Figure 3-1 Fusion Middleware Configuration Wizard



On the Managed Server screen, for `ess_server1`, select the WSMPM-MAN-SVR server group. ESS-MGD-SVRS should already be selected.

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.

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
   ```

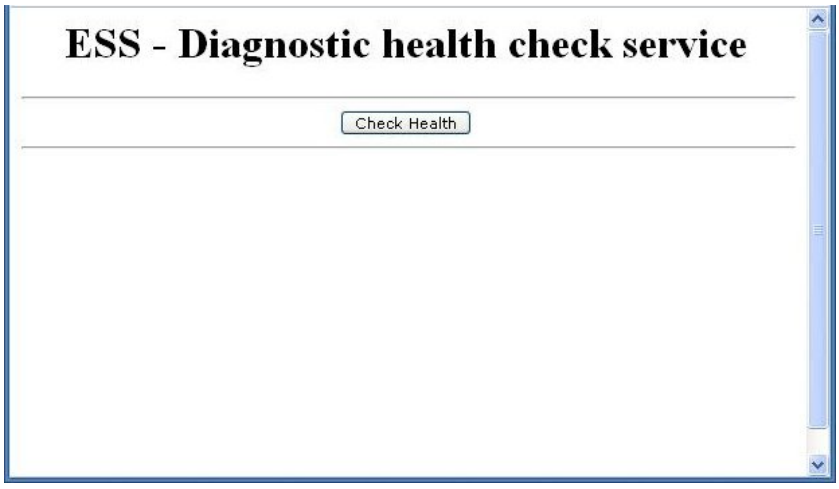
   where `hostName` is the server to which Oracle Enterprise Scheduler is installed and `port` is the port number.

   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 3-2.

   Figure 3-2 Diagnostic Health Check Page

2. Log in to the diagnostic servlet using an Oracle WebLogic Server administrator user name and password.

3. Click the **Check Health** button to verify the installation.

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 3-1.

Table 3-1 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.

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

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.

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.

4

Using the Pre-Deployed Native Hosting Application

The pre-deployed native hosting application is included as part of Oracle Enterprise Scheduler. It greatly simplifies the process of getting Oracle Enterprise Scheduler up and running because you do not have to create your own custom hosting application.

The Oracle Enterprise Scheduler is flexible and provides implementation and deployment options. Planning Job Development is a high-level discussion about how to plan your job development and deployment process

This chapter includes the following sections:

- Introduction
- Properties
- Metadata
- Security Permissions

Introduction

The pre-deployed native hosting application provides a convenient alternative to developing your own custom hosting application. The pre-deployed native hosting application can be used to run any job except a Java-based job.

The pre-deployed native hosting application is collocated with the Oracle Enterprise Scheduler core on the Oracle Enterprise Scheduler server. The pre-deployed native hosting application can be deployed to only one cluster in the domain. The "Enterprise Scheduler Basic" template deploys the pre-deployed native hosting application along with scheduler server components and therefore can be targeted to only one cluster in the domain.

The pre-deployed native hosting application exposes the remote interfaces of Oracle Enterprise Scheduler beans mapped to the following JNDI Names:

- **Runtime service bean:** `java:comp/env/essnative/runtimeservice`
- **Metadata service bean:** `java:comp/env/essnative/metadataservice`
- **Async request bean:** `java:comp/env/essnative/asyncrequest`

If the user uses some other hosting apps, the beans are to be exposed in the above fashion by declaring in weblogic's ejb-jar.xml.

Properties

Table 4-1 shows the pre-deployed native hosting application properties and their default settings.

Table 4-1 Pre-Deployed Native Hosting Application Properties

Property Name	Value
`PolicyStripe`	`EssNativeHostingApp`
`LogicalAppName`	`EssNativeHostingApp`
`J2ee App Name`	`EssNativeHostingApp`
`RuntimeService EJB JNDI name`	`essnative/runtimeservice`
`MetadataService EJB JNDI name`	`essnative/metadataservice`
`AsyncRequestBean JNDI name`	`essnative/asyncrequest`
`MDS Partition`	`essUserMetadata`
`MDS namespace`	`/oracle/apps/ess`

Metadata

Normally, a hosting application contains the MAR metadata to be loaded into MDS, however, you cannot add a MAR archive directly into the pre-deployed native hosting application. If you have a MAR archive that you want to deploy to the pre-deployed native hosting application, you have to deploy it through a client application or using the metadata API (see Using the Metadata Service).

Oracle Enterprise Scheduler promotes the native hosting application's stripe by registering it with OPSS. This in turn exposes the stripe as viewable from Oracle Enterprise Manager Fusion Middleware Control and other MBeans. Application roles and policies can then be configured at runtime for the metadata available in the native hosting application.

Security Permissions

Permission must be granted for all jobs run by the pre-deployed native hosting application. By default, `EssNativeHostingApp` extends support for permissions defined in SOA and Service Bus. This is configured in `ess-config.xml` using the property `HostingAppPolicyStripe` specified with the value "`EssNativeHostingApp,soa-infra,Service_Bus_Console`".

If you install Oracle Enterprise Scheduler with another product (other than SOA or Service Bus) then you must use Oracle Enterprise Scheduler or WLST scripts to extend this property list of stripes.

Configuring the Policy Stripe

The EssNativeHostingApp security policy stripe is configured by customizing the `ess-config.xml` file. The Oracle Enterprise Scheduler property name is `HostingAppPolicyStripe`. The following examples show how to use WLST commands to check the policy stripe value and change it.

Check the value of the current policy stripe:

```
oracle_common/bin/essManageRuntimeConfig.sh \
   -u weblogic -p welcome1 -P 7001 -H localhost -s ess_server1 \
   -A EssNativeHostingApp -n HostingAppPolicyStripe -t ESS
```

Change the value of the policy stripe:

```
 oracle_common/bin/essManageRuntimeConfig.sh \
   -u weblogic -p welcome1 -P 7001 -H localhost -s ess_server1 \
   -m -A EssNativeHostingApp -n HostingAppPolicyStripe -t ESS
   -v MyPolicyStripe
```

See *Oracle Fusion Middleware WLST Command Reference for SOA Suite* for more information about WLST commands.

Support for Multiple Application Stripes

The pre-deployed native hosting application supports multistripes. The pre-deployed native hosting application policy stripes are pre-configured for SOA and Service Bus applications.

```
<EssProperty key="HostingAppPolicyStripe"
   value="EssNativeHostingApp,soa-infra,Services_Bus_Console"/>
```

You can use Oracle Enterprise Manager Fusion Middleware Control or WLST to extend this list of stripes by appending the policy stripes to the value of the `HostingAppPolicyStripe` property. You can also use the Oracle Enterprise Manager Fusion Middleware Control Application Properties page to change the `HostingAppPolicyStripe` property for the pre-deployed native hosting application.

This property is applicable only for the configuration file of the pre-deployed native hosting application. Note that there is no static definition of policy stripes in the pre-deployed native hosting application's `ejb-jar.xml` file, therefore, you must preserve the existing policy stripes specified in the `HostingAppPolicyStripe` property.

5

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. Also, custom hosting applications are generally seeded with metadata that is packaged and deployed to the metadata repository when the application is deployed. Ant-based scripts that generate custom hosting applications do not provide a way to create metadata artifacts. For that reason, after you generate a hosting application, you must open the workspace (`.jws`) in Oracle JDeveloper, and add the necessary metadata before you deploy the application into the server.

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:

- Introduction to Generating a Hosting Application with Ant
- Ant Targets for Creating and Deploying a Hosting Application
- Creating a Hosting Application and Project Workspace with Ant
- Creating a Java Job as a Shared Library with Ant
- Packaging a Java Job as a Shared Library with Ant
- Deploying a Shared Library with Ant
- Packaging a Hosting Application with Ant
- Deploying a Hosting Application with Ant
- Configuring the Generated Ant Targets

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.

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.

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/oracle_common/ess/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 5-1 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 5-2 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.

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 are generated within a JDeveloper-compatible workspace in the target directory you specified. The created workspace has 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 5-3 Information Needed by the Ant Target

Input Prompt	Description
Which template should be used	Possible values are "Fusion" and "Standalone". If you're developing for use with Oracle Fusion Applications, enter Fusion here. If you're not developing for use with Oracle Fusion Applications, enter "Standalone." 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.
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 are found relative to this directory. This feature relies on the `ojdeploy` utility to create, package and deploy artifacts to the server. If the middleware home path does not contain an Oracle JDeveloper directory with `ojdeploy` in the `bin` directory, specify the directory where Oracle JDeveloper is installed.
Hosting application name	The name you want the new hosting application to have.
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 runtime, 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 is the location of the JDeveloper workspace, where artifacts such as the `build.xml` file you use later is created.

To create a hosting application with Ant

1. 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/oracle_common/ess/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.

```
[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] Do you want to add shared library for the (java) job business
logic? (use_jobdef_library) (default=yes)
    [input]      ([yes], no)
no
[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 9 files to
/scratch/WLServers/MW_HOME/standalone_apps/NewDemoApp
     [copy] Copied 15 empty directories to 4 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: 49 seconds

[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] Do you want to add shared library for the (java) job business
logic? (use_jobdef_library) (default=yes)
    [input]      ([yes], no)
yes
    [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 11 files to
/scratch/WLServers/MW_HOME/standalone_apps/NewDemoApp
     [copy] Copied 25 empty directories to 9 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 32 seconds
```

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 is in the build file generated when you created a new hosting application. (For more information, see 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 5-4 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.

```
[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 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 5-1 shows what the generated code looks 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 5-1 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);
    }
}
```

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 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
  ```

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 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
  ```

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 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 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
  ```

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 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 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
  ```

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 5-5 to customize the Ant targets with configuration values of your own.

Table 5-5 Build Properties for Customizing Ant Builds

Build Property	Description
customEss.hostapp.earprofile	-
customEss.hostapp.jarfile	-
customEss.hostapp.jarprofile	-
customEss.hostapp.jprproject	-
customEss.hostapp.jwsfile	-
customEss.hostapp.mds.jdbc	-
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	-

Build Property	Description
fmw.home	-
jdev.home	-
oracle.common	-
ess.server.name	Comma separated names of Oracle Enterprise Scheduler admin/managed servers to which the Oracle Enterprise Scheduler job library and hosting application is deployed.
weblogic.admin.user	The WebLogic Server admin user name.
weblogic.server.host	-
weblogic.server.port	-
weblogic.server.ssl.port	-
weblogic.t3.url	-

```
# 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=${fmw.home}/wlserver_10.3
weblogic.server.host=adc2170657.example.com
WEBLOGIC_HOME=${fmw.home}/wlserver_10.3
weblogic.server.host=adc2170657.example.com
weblogic.server.port=7001
weblogic.server.ssl.port=7002
weblogic.admin.user=weblogic
weblogic.t3.url=t3://${weblogic.server.host}:${weblogic.server.port}
# Comma separated names of ess admin/managed servers to which essjob library
and    hosting app is deployed
ess.server.name=AdminServer
```

6

Creating a Thin Client Application

The thin client application is typically used to submit jobs, query status and optionally used to host EJB job implementations. A thin client application uses the Oracle Enterprise Scheduler thin client library for Oracle Enterprise Scheduler APIs. This chapter describes the thin client library and how to use Oracle JDeveloper to develop a thin client application.

This chapter contains the following sections:

- Introduction
- Implementation
- Using JDeveloper to Build a Thin Client Application for MAR Deployment
- Using JDeveloper to Create and Configure an EJB and its Job Definition Metadata

Introduction

Client applications are J2EE applications that execute in the same WebLogic domain as Oracle Enterprise Scheduler. Client applications can use the Oracle Enterprise Scheduler APIs to do the following:

- Submit jobs
- Query job status
- Look at job output and logs
- Optionally, perform updates to Oracle Enterprise Scheduler metadata
- Host an EJB job implementation that the Oracle Enterprise Scheduler invokes remotely

The Oracle Enterprise Scheduler thin client library is used by client applications to access Oracle Enterprise Scheduler APIs (for example, the metadata service API or the runtime service API). The thin client library is a thin layer that remotely invokes an Oracle Enterprise Scheduler hosting application to perform all operations. The thin client application may optionally have an Oracle Enterprise Scheduler metadata MAR archive with Oracle Enterprise Scheduler metadata developed using Oracle JDeveloper. This metadata is automatically loaded into the Oracle Enterprise Scheduler MDS when the application is deployed. Alternatively, the application can use APIs to create the metadata dynamically.

The thin client shared library differs from the client shared library in the following ways:

- The client shared library includes local EJBs that do all the Oracle Enterprise Scheduler work by directly accessing the MDS and runtime databases. The thin

client library does not include the data sources or EJBs, but instead remotely accesses a hosting application that hosts the EJB and accesses the databases.

- The thin client library more cleanly hides Oracle Enterprise Scheduler internal functionality from the application.
- Because it accesses the databases directly, the client shared library works even if the Oracle Enterprise Scheduler server or cluster is down.
- The thin client library is also useful when deployment of the Oracle Enterprise Scheduler is optional in an embedding product.
- All of the documented APIs exposed by the client shared library are available in the thin client library. Therefore, thin client applications can:
  - Request submission using the runtime service APIs
  - Operate on requests using the runtime service APIs
  - Update metadata artifacts using the Metadata Service API
  - Remotely complete asynchronous requests
- Because the thin client library remotely invokes an Oracle Enterprise Scheduler hosting application to perform all operations, it has to look up remote Oracle Enterprise Scheduler beans instead of local beans. There is some overhead in obtaining the `InitialContext` of a remote Oracle Enterprise Scheduler server. The `RemoteConnector` API provides the following assistance for the callback of Oracle Enterprise Scheduler beans:
  - Helper classes use `RemoteConnectors` to easily connect back to Oracle Enterprise Scheduler beans (for example, `RuntimeService` and `MetadataService`)
  - Log and output can be handled from a remote implementation
  - Asynchronous requests can be completed easily
  - Invocations and callbacks can be secured

Implementation

Consider the following when you use the thin client library to implement a remote EJB job:

- Make sure the bean implements the `RemoteExecutable` interface for execution only, or the `RemoteCancellableExecutable` interface for both the execute and cancel operations.
- Use predefined system properties such as `EJB_OPERATION_NAME` instead of defining specific properties such as `SOA_BEAN_NAME`.
- The `ejb-jar.xml` file should define the `oracle.security.jps.ee.ejb.JpsInterceptor` interceptor. Use the interceptor to obtain the subject propagated from the Oracle Enterprise Scheduler layer and use it in other operations.
- It's best to move the job implementation out of the `ejb-jar.xml` file to ensure that the EJBs are not redeployed when the job logic changes.

Tip:

The application throws a `javax.naming.NamingException` exception if the JNDI context cannot be created with the passed in values. Alternatively, the `ScehdulerException` exception can be thrown when there is a problem with look-ups that involve the credential key store.

Secured Invocation

Secured invocation of the remote EJB is required when the JNDI tree of its server is authenticated. This is also the case when a remote EJB uses secure lookup to call back to Oracle Enterprise Scheduler EJBs. The following sections provides some guidance.

Forward Invocation

The following apply to forward invocation.

- When Oracle Enterprise Scheduler invokes a remote EJB, the subject of the executing job is always propagated.
- When Oracle Enterprise Scheduler executes a job, the `JndiProviderUrl` of the current Oracle Enterprise Scheduler Server is always supplied to the remote EJB through `RequestParameters`.
- If the JNDI tree of the remote server is authenticated, the `JNDI_CSF_KEY` property must be specified in the request parameters or the `EssConfiguration` of the hosting application.
- Oracle Enterprise Scheduler looks up the keystore for the `CsfKey` to retrieve the `PasswordCredential` and connects to the remote server.

Callback Invocation

The following apply to callback invocation.

- If the remote EJB must call back to Oracle Enterprise Scheduler beans, the following properties can be specified:
  - The JNDI names of Oracle Enterprise Scheduler `Runtime`, `Metadata` and `AsyncRequest` beans exposed in `HostingApp` must be specified in request parameters or the `EssConfiguration` of the hosting application. If `EssNativeHostingApp` is used, these entries are not required.
  - If the JNDI tree of the Oracle Enterprise Scheduler server is authenticated, the `ESS_JNDI_CSF_KEY_NAME` property must be specified in the request parameters or `EssConfiguration` of the hosting application. Oracle Enterprise Scheduler ensures that this property is available to the remote EJB through `RequestParameters`.
- A remote EJB can make use of the `RemoteConnector` API to get the remote instances of Oracle Enterprise Scheduler beans. This can be done by passing the following:
  - `RequestParameters`
  - `RequestParameters` and `JndiMappedName` of the bean (for hosting applications other than the native hosting application)

- `RequestParameters`, user name and password (if the Oracle Enterprise Scheduler server is authenticated)
- `InitialContext` (primarily for Java SE clients with `EssNativeHostingApp`)
- `InitialContext` and `jndiMappedName` (primarily for Java SE clients with other hosting applications)

RemoteConnector API and the Server Affinity Property

If your code implementation relies on accessing Oracle Enterprise Scheduler EJBs, use the methods exposed in the `RemoteConnector` API class. The Oracle Enterprise Scheduler requires the server affinity property to be set in the `InitialContext` environment before doing a JNDI lookup and the `RemoteConnector` API class sets this property for you. Note that this is especially important in multi-node cluster scenarios. The `RemoteConnector` class is packaged in the Oracle Enterprise Scheduler client libraries.

If for some reason the `RemoteConector` class cannot be used, you can set the environment map property to the `InitialContext` before doing the look-up for the Oracle Enterprise Scheduler EJBs as shown in the following example.

```
if(PlatformUtils.isWebLogic())
   envProps.put("weblogic.jndi.enableServerAffinity", "true");
```

In a multi-node cluster environment, it is best to set the cluster algorithm to "round-robin-affinity".

Examples

This section contains examples that illustrate how to use the thin client library.

Java EE Application That Uses RemoteConnector

The following code example shows a snippet from a Java EE application that uses `RemoteConnector` through the pre-deployed native hosting application.

```
RemoteConnector essConnector = newRemoteConnector();
//RequestParameters contains the JndiProviderURL of Oracle Enterprise Scheduler
//Server which is auto-populated from the Oracle Enterprise Scheduler end while
//invoking an EJB. The CSF key is auto-populated in RequestParameters from the
//Oracle Enterprise Scheduler end if configured for the
//Oracle Enterprise Scheduler Server and specified in the EssConfig of
HostingApp.
//If CSF key is present, the CSF lookup is done from RemoteConnector to resolve
//authentication.

RuntimeService rts = essConnector.getRuntimeServiceEJB(requestParameters);

//Sample invocation using RuntimeServiceBean.
RuntimeServiceHandle handle = rts.open();
RequestDetail reqDetail = rts.getRequestDetail(handle,
requestExecutionContext.getRequestId());
```

Implementation

The following example shows a skeletal implementation of an EJB job that uses the thin client library. See Creating and Using EJB Jobs for more information about implementing EJB jobs.

```
@Stateless(name = "JMXAdapter")
public class JMXAdapterBean implements RemoteCancellableExecutable
{
@Resource
private SessionContext sctx;
public JMXAdapterBean() {
}

public void execute(RequestExecutionContext requestExecutionContext,
RequestParameters requestParameters) throws
ExecutionErrorException,
ExecutionWarningException,
ExecutionPausedException,
ExecutionCancelledException
{
//"ESS RequestId:" + requestExecutionContext.getRequestId();
//"EJB Operation:" +
requestParameters.getValue(SystemProperty.EJB_OPERATION_NAME);
//"Invoke Message:" + requestParameters.getValue(SystemProperty.INVOKE_MESSAGE);
}
public void cancel(RequestExecutionContext requestExecutionContext,
RequestParameters requestParameters)
{
//"ESS RequestId:" + requestExecutionContext.getRequestId();
//"EJB Operation:" +
requestParameters.getValue(SystemProperty.EJB_OPERATION_NAME);
//"Invoke Message:" + requestParameters.getValue(SystemProperty.INVOKE_MESSAGE);
}
}
```

Subject Propagation

When the Oracle Enterprise Scheduler invokes an EJB job, the subject associated with the hosting application is always propagated to the job. This ensures that the subject that executes the job is available in the business operation of the bean. Add the following code to the `ejb-jar.xml` file to retrieve the subject from within the bean.

```
<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>NAME_OF_ENTERPRISE_APPLICATION</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>
<assembly-descriptor>
<interceptor-binding>
<ejb-name>*</ejb-name>
<interceptor-class>oracle.security.jps.ee.ejb.JpsInterceptor</interceptor-class>
</interceptor-binding>
</assembly-descriptor>
```

You can use the following code to invoke an MBean from an EJB operation in the privileged context of the current subject:

```
AccessControlContext accContext = AccessController.getContext();
Subject currentSubject = Subject.getSubject(accContext);
String currentUsername = SubjectUtil.getUserName(currentSubject);
Subject.doAs(currentSubject, new PrivilegedExceptionAction() {
public Object run() {
//logic to invoke MBean
}
});
```

Using JDeveloper to Build a Thin Client Application for MAR Deployment

If your job uses the Oracle Enterprise Scheduler pre-deployed native hosting application, you can simplify the creation of custom job metadata by building a client application that assists in the creation of the metadata and deploys it to the pre-deployed native hosting application MDS partition (`essUserMetadata`).

The following instructions show how to use Oracle JDeveloper to create a thin client application that:

- Adds job metadata
- Creates an enterprise archive (EAR)
- Packages the metadata archive (MAR) in the EAR
- Deploys the metadata to the pre-deployed native hosting application

The instructions in Using JDeveloper to Create and Configure an EJB and its Job Definition Metadata describe how to create an EJB that can be invoked by the EJB job definition added to the pre-deployed native hosting application.

The ADF infrastructure is used to deploy the metadata to a specific partition when an application gets deployed. The MDS partition for the pre-deployed native hosting application is `essUserMetadata`.

JDeveloper provides accessibility options, such as support for screen readers, screen magnifiers, and standard shortcut keys for keyboard navigation. You can also customize JDeveloper for better readability, including the size and color of fonts and the color and shape of objects. For information and instructions on configuring accessibility in JDeveloper, see "Oracle JDeveloper Accessibility Information" in *Developing Applications with Oracle JDeveloper*.

Note:

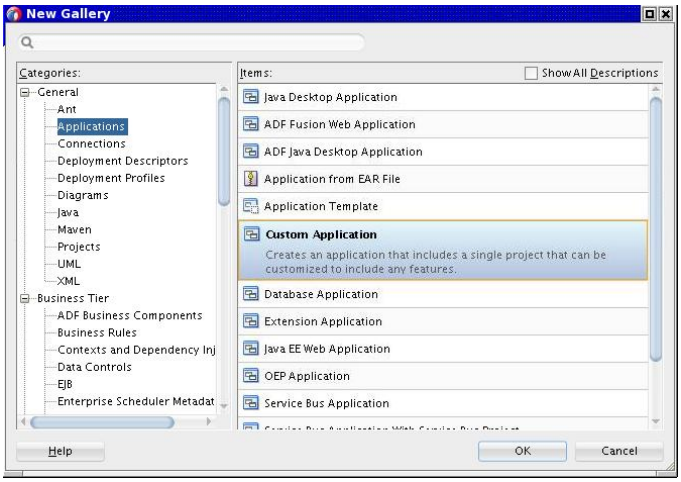
Be sure to set the `MW_HOME` environment variable before you start JDeveloper. For example: `export MW_HOME=/scratch/prh/12c/jdev` If this variable is not set, the **Job Type** dropdown menu is not populated.

Create and Deploy a Thin Client Application for the Standalone Environment

The following steps describe how to use JDeveloper to create and deploy a thin client application.

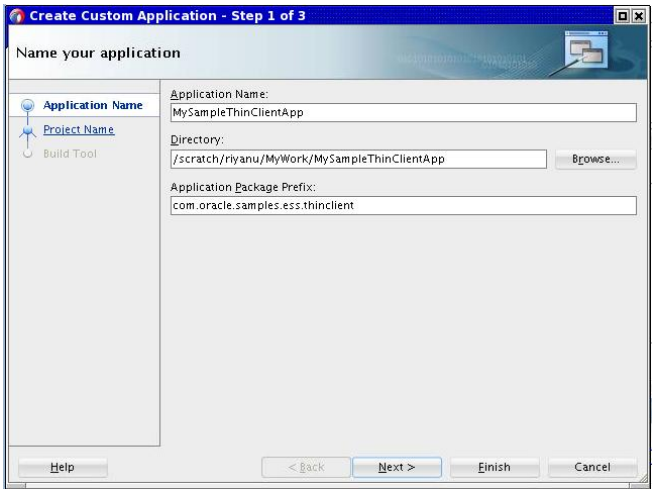
1. In the New Gallery dialog, create a custom application and project as shown in Figure 6-1.

Figure 6-1 New Gallery Dialog



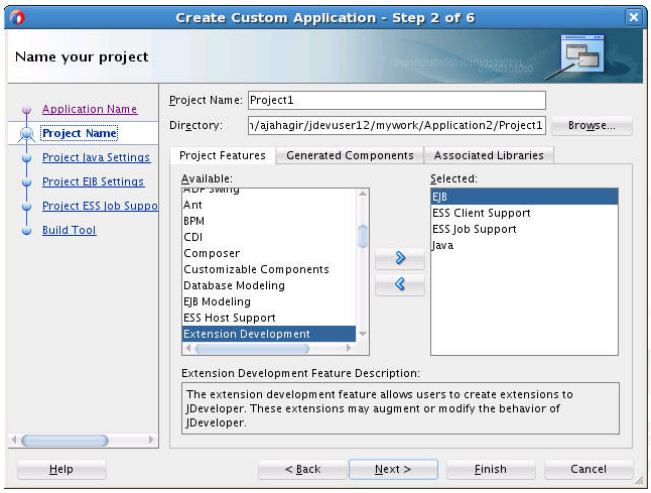
2. In the Create Custom Application dialog, enter the application name and application package prefix as shown in Figure 6-2.

Figure 6-2 Create Custom Application Dialog - Step 1 of 3



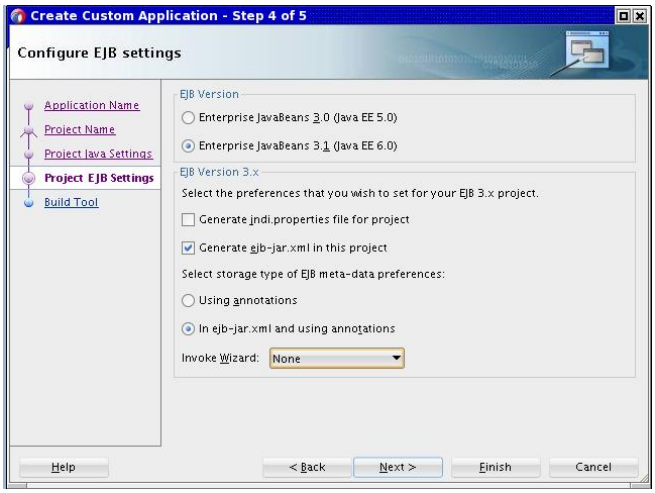
3. Enter the project name, then add **ESS Job Support** and **EJB** into the project features as shown in Figure 6-3.

Figure 6-3 Create Custom Application Dialog - Step 2 of 5



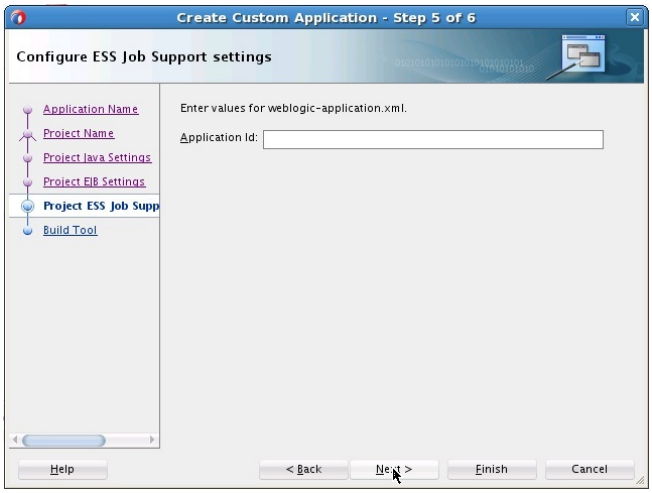
4. Configure the EJB settings as shown in Figure 6-4.

Figure 6-4 Create Custom Application Dialog - Step 4 of 5



5. Configure the application ID value as shown in Figure 6-5

Figure 6-5 Configure ESS Job Support Settings



6. Click **Next**.

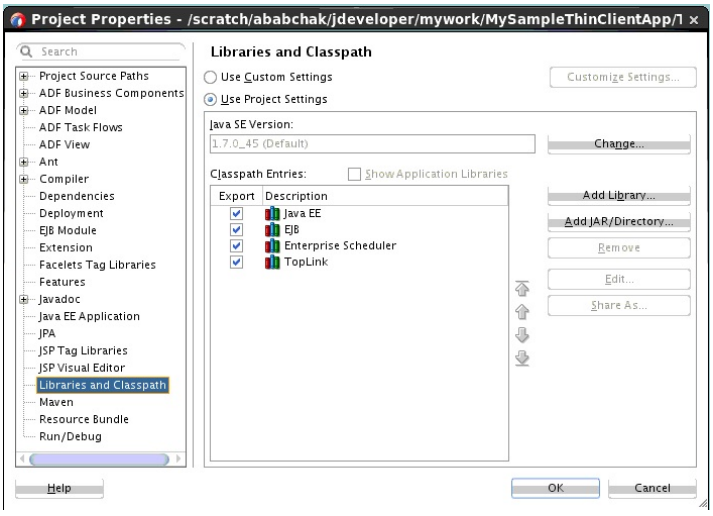
7. Click **Finish** to complete the steps to create a new application.

8. Edit the MANIFEST.MF file generated in the previous step and remove the following lines:

```
essclientapi-Specification-Version: 12
Extension-List: essruntime, essclientapi
Weblogic-Application-Version: 3.0
essclientapi-Extension-Name: oracle.ess.client.api
essruntime-Extension-Name: oracle.ess.runtime
essruntime-Specification-Version: 12
```

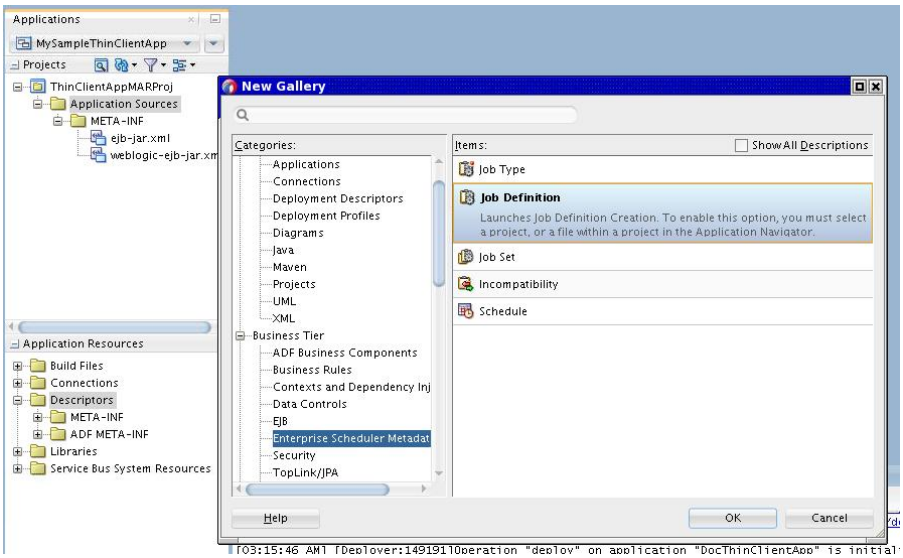
9. Right click the project node in the left tree panel, then select project properties and click “Libraries and Classpath" as shown in Figure 6-6. Make sure to only select “Java EE" and “Enterprise Scheduler" in the **Classpath Entries** pane.

Figure 6-6 Project Properties Dialog



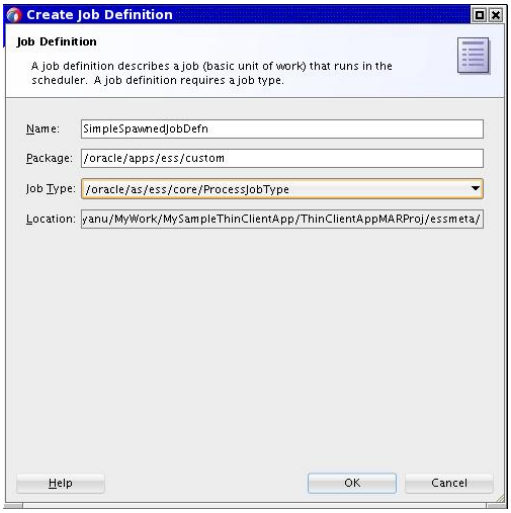
10. In the New Gallery dialog, select “Enterprise Scheduler Metadata" and “Job Definition" as shown in Figure 6-7.

Figure 6-7 New Gallery Dialog



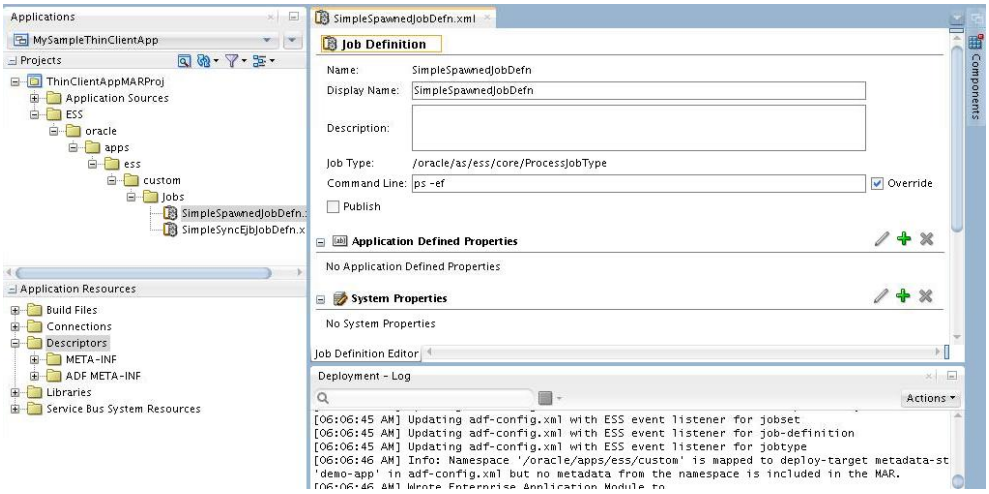
11. In the Create Job Definition dialog, select `/oracle/as/ess/core/ProcessJobType` from the **Job Type** dropdown to add simple spawned job definition metadata as shown in Figure 6-8.

Figure 6-8 Create Job Definition Dialog



12. To complete the addition of spawned job definition metadata, select the **Override** check box and enter a value for the **Command Line** entry. Add a system property named `SYS_effectiveApplication` with a value of `EssNativeHostingApp`.

Figure 6-9 SimpleSpawnedJobDefn.xml Tab



13. Add EJB job definition Metadata. Follow the steps in Using JDeveloper to Create and Configure an EJB and its Job Definition Metadata.

14. Configure the MAR profile.

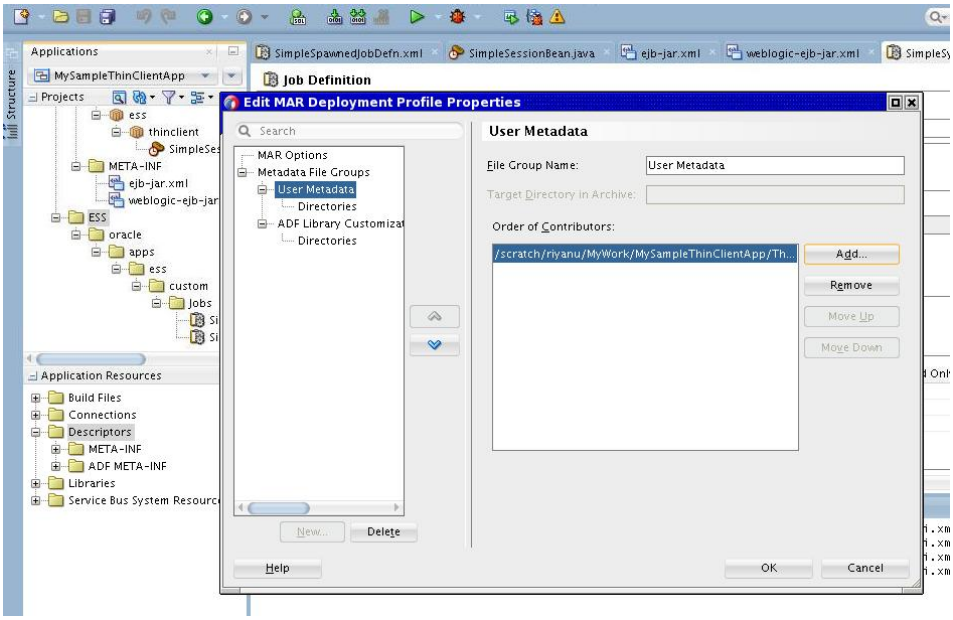
    a. Select “Application Properties" and click “Deployment Node".

    b. Select “MAR Module" in the right side panel and click **Edit**.

    c. Select “User Metadata" in the Edit MAR Deployment Properties dialog as shown in Figure 6-10.

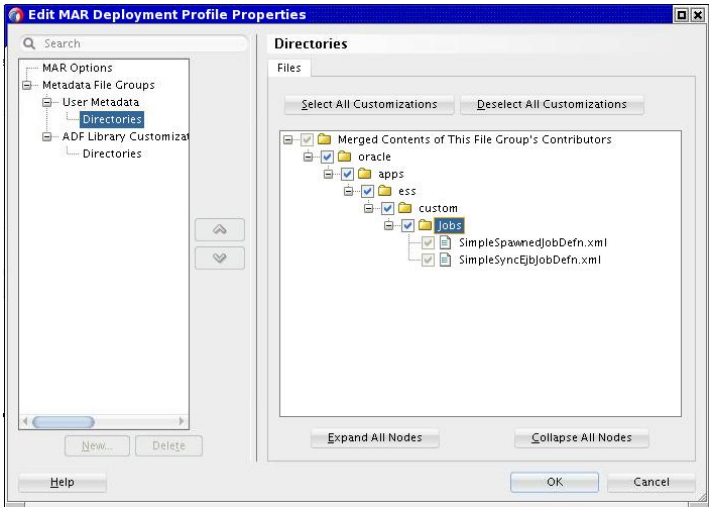
    d. Make sure that the `essmeta` directory (under the path of the project created above) is available. If it is not available, add the directory by manually navigating to the `essmeta` directory.

Figure 6-10 Edit MAR Deployment Profile Properties Dialog



15. Select the appropriate metadata to package in the MAR. click the “Directories" node under “User Metadata" and make sure that the newly added job definitions are selected as shown in Figure 6-11.

Figure 6-11 Edit MAR Deployment Profile Properties Dialog

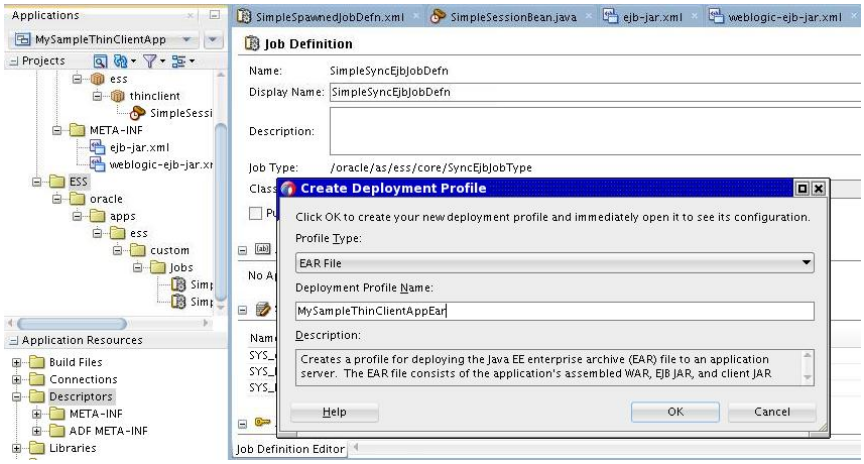


16. Create a deployment profile for an enterprise archive (EAR).

    a. Select “Application Properties" and click the deployment node.

    b. Click **New** in the right panel to invoke the Create Deployment Profile dialog. Choose “EAR File" from the **Profile Type** dropdown as shown in Figure 6-12.

Figure 6-12 Create Deployment Profile Dialog



17. Configure the application assembly for the EAR. In the dialog, make sure the following two profiles are selected:

    - The MAR profile you previously created
    - The EJB profile. This profile is automatically created. If it is not automatically created, create a new “EJB JAR" deployment profile for the project as previously described beginning in step 2.

To create the EJB-JAR deployment profile:

a. In the Application Navigator, in the Projects panel, right-click the **EssHost** project, then click **Project Properties**.

b. In the Project Properties window navigator, click **Deployment**.

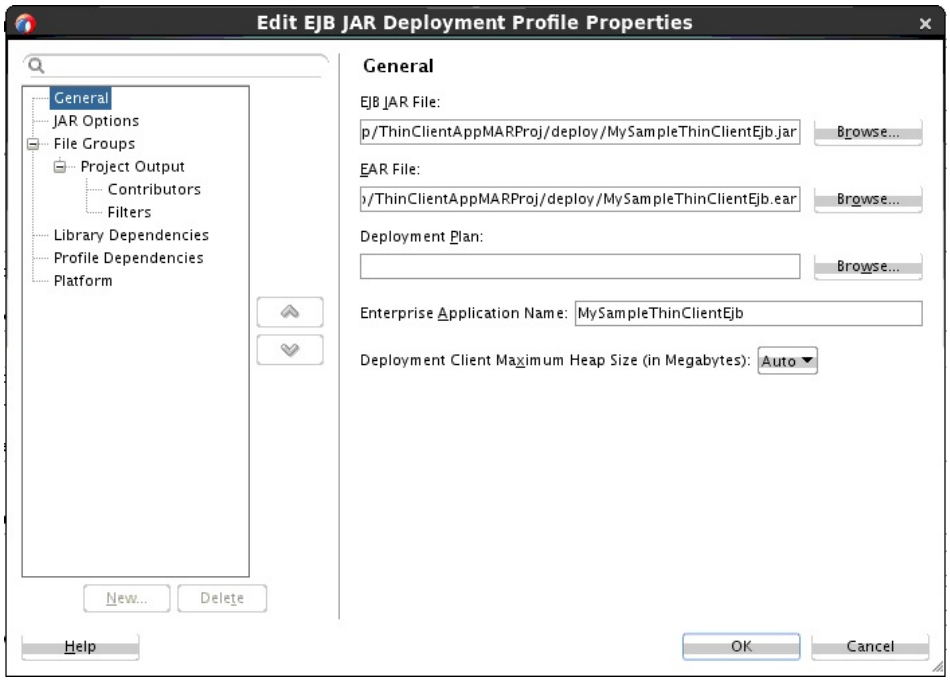
c. Under Deployment Profiles, delete all profiles listed in the window, then click **New**.

d. In the Create Deployment Profile dialog, from the **Profile Type** dropdown list, select **EJB JAR file**.

e. In the **Name** field, enter a name for the EJB. For this example, enter `MySampleThinClientEjb`.

f. Click **OK**.

Figure 6-13 Create the EJB-JAR Deployment Profile



g. In the Edit EJB JAR Deployment Profile Properties dialog navigator on the left, click **General**.

h. In the General window, in the **Enterprise Application Name** field, enter `MySampleThinClientApp`.

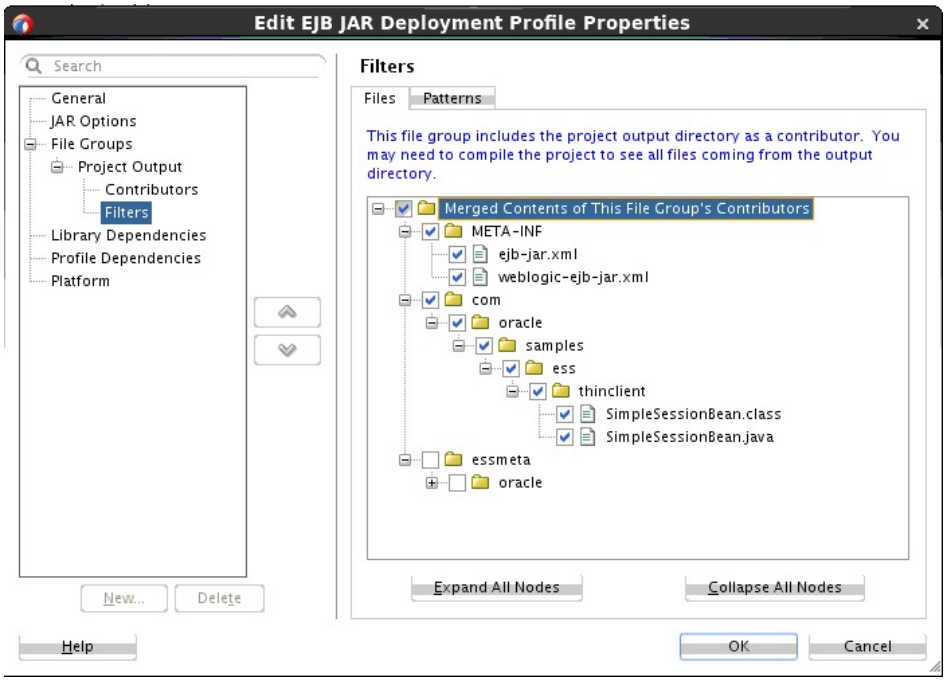
i. In the navigator, expand to **File Groups** > **Project Output** > **Contributors**.

j. In the Contributors window, select the following check boxes:

- **Project Output Directory**
- **Project Source Path**
- **Project Dependencies**

k. In the navigator, expand to **File Groups** > **Project Output** > **Filters**.

Figure 6-14 Edit EAR Deployment Profile Properties Dialog



l. In the Filters window, in the Files tab, ensure that the following folders are selected:

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

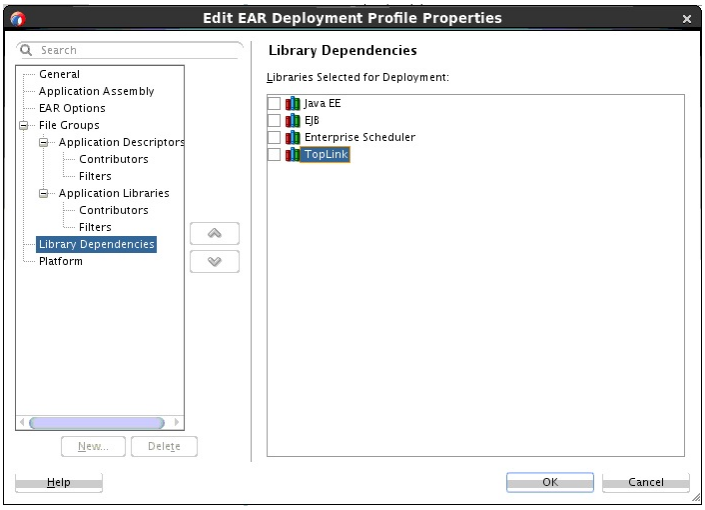
m. In the JAR Option window, deselect the Include **Manifest File** item.

n. Click **OK**.

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

18. Configure the library dependencies for the EAR. Be sure that none of the items are selected in the **Libraries Selected for Deployment** pane.

Figure 6-15 Edit EAR Deployment Profile Properties Dialog

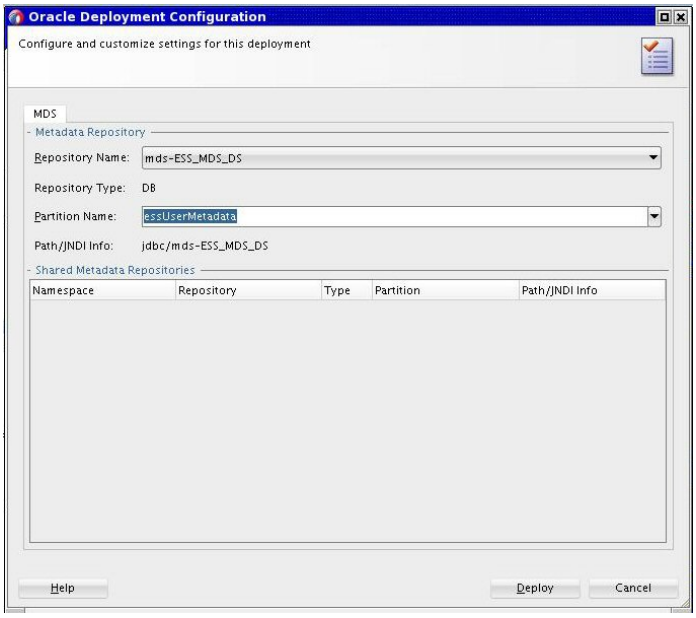


19. Configure the `adf-config.xml` file. When you deploy an ADF-based application from JDeveloper, there is a dialog that asks you to select the MDS partition into which the metadata is to be deployed. If the EAR file generated from this application is to be deployed from the WLS console, certain MDS partition entries must be specified in the `adf-config.xml` file. If this is the case, ensure that the `adf-config.xml` file contains the entries shown in Example 6-1. You can find the `adf-config.xml` file in the Application Resources > Descriptors > ADF META-INF section in bottom of the left panel.

20. Configure the `weblogic-application.xml` file. Make sure the contents of the `weblogic-application.xml` file are as shown in Example 6-2.

21. Deploy the application. To complete the deployment of the EAR, select `essUserMetadata` in the **Partition Name** dropdown in the Deployment Configuration dialog and click **Deploy**.

Figure 6-16 Oracle Deployment Configuration Dialog



Example 6-1 Contents of the adf-config.xml File

```
<?xml version="1.0" encoding="UTF-8" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config">
    <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 path="/oracle/apps/ess/
custom"                         metadata-store-usage="ess_custom_metadata"/>
                </metadata-namespaces>
                <metadata-store-usages>
                    <metadata-store-usage
id="ess_custom_metadata"                          deploy-target="true" default-cust-
store="true">
                        <ns3:metadata-store                          class-
name="oracle.mds.persistence.stores.db.DBMetadataStore"
xmlns:ns3="http://xmlns.oracle.com/mds/config">
                            <ns3:property name="repository-
```

```
name"                              value="mds-ESS_MDS_DS" />
                          <ns3:property name="partition-
name"                              value="essUserMetadata" />
                          <ns3:property name="jndi-
datasource"                                value="jdbc/mds-ESS_MDS_DS" />
                     </ns3:metadata-store>
                </metadata-store-usage>
            </metadata-store-usages>
        </persistence-config>
    </mds-config>
  </adf-mds-config>
</adf-config>
```

Example 6-2 Contents of the weblogic-application.xml File

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

xsi:schemaLocation="http://xmlns.oracle.com/weblogic/weblogic-
application              http://
xmlns.oracle.com"                                xmlns="http://xmlns.oracle.com/
weblogic/weblogic-application">
<listener>
   <listener-class>oracle.mds.lcm.weblogic.WLLifecycleListener</listener-class>
</listener>
<library-ref>
   <library-name>oracle.ess.thin.client</library-name>
</library-ref>
</weblogic-application>
```

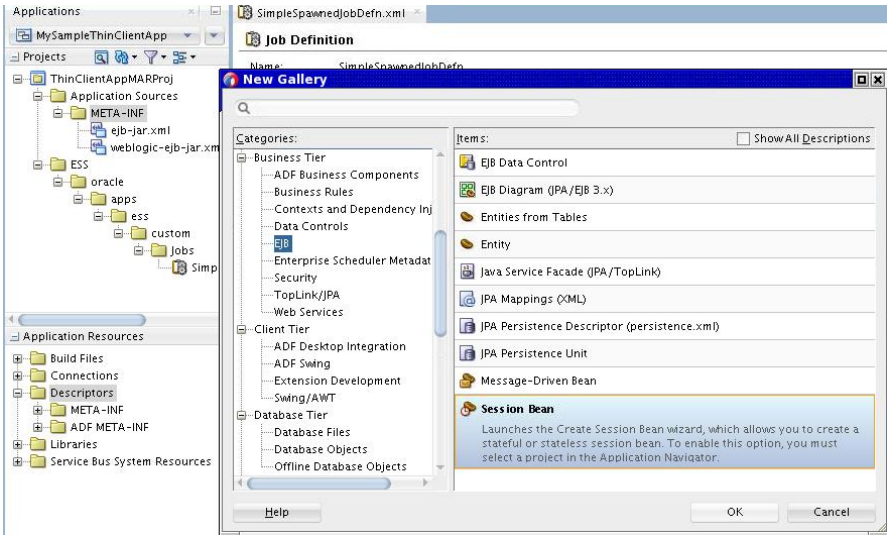
Using JDeveloper to Create and Configure an EJB and its Job Definition Metadata

The following steps describe how to:

- Create a simple synchronous EJB that conforms to Oracle Enterprise Scheduler's job implementation requirements.
- Create EJB job definition metadata and deploy it as a part of the enterprise application.

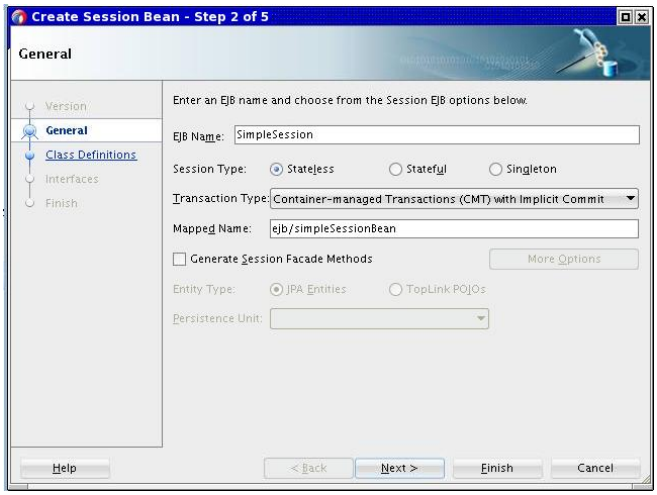
1. Create a session bean. In the New Gallery dialog, select “Session Bean" to create a new EJB as shown in Figure 6-17.

Figure 6-17 New Gallery Dialog



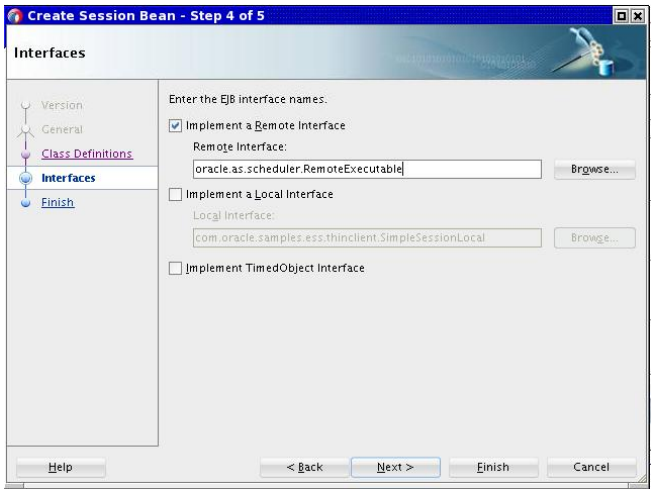
2. Configure the session bean. Enter the EJB name in the **EJB Name** field and enter the mapped name in the **Mapped Name** field as shown in Figure 6-18. Click **Next** to continue.

Figure 6-18 Create Session Bean Dialog - Step 2 of 5



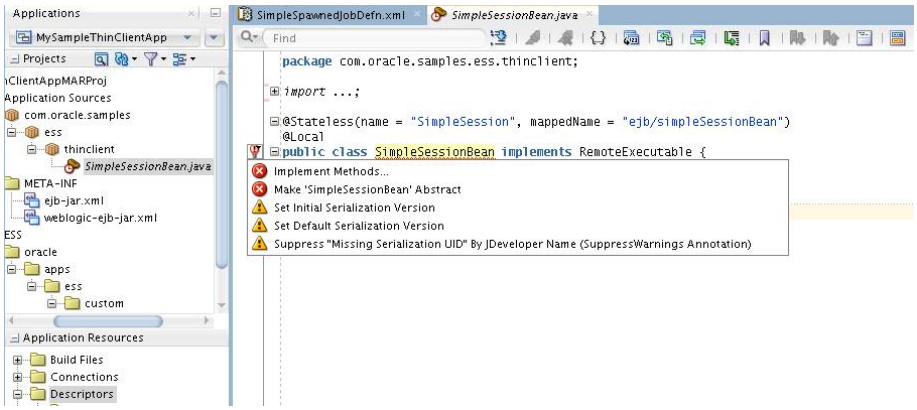
3. Configure the session bean business interface. Make sure that the **Implement a Remote Interface** check box is checked and that `oracle.as.scheduler.RemoteExecutable` is set as the class for the **Remote Interface** field as shown in Figure 6-19. Click **Next** and proceed to the **Finish** step.

Figure 6-19 Create Session Bean Dialog - Step 4 of 5



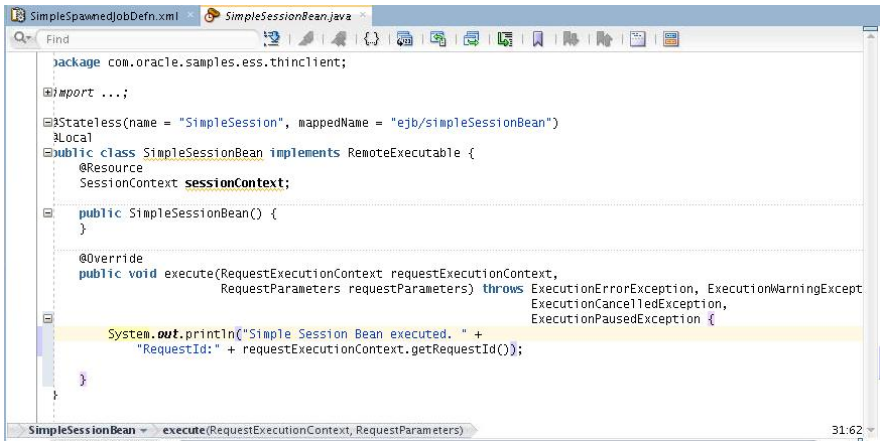
4. Configure the generated session bean. Make sure the generated session bean implements the execute method defined in the `RemoteExecutable` interface

Figure 6-20 Configure the Generated Session Bean



5. Complete the implementation of the session bean. Make sure the bean is complete by implementing the `execute` method.

Figure 6-21 Complete the Implementation of the Session Bean



6. Make sure the `ejb-jar.xml` file contains the declaration shown in Example 6-3.

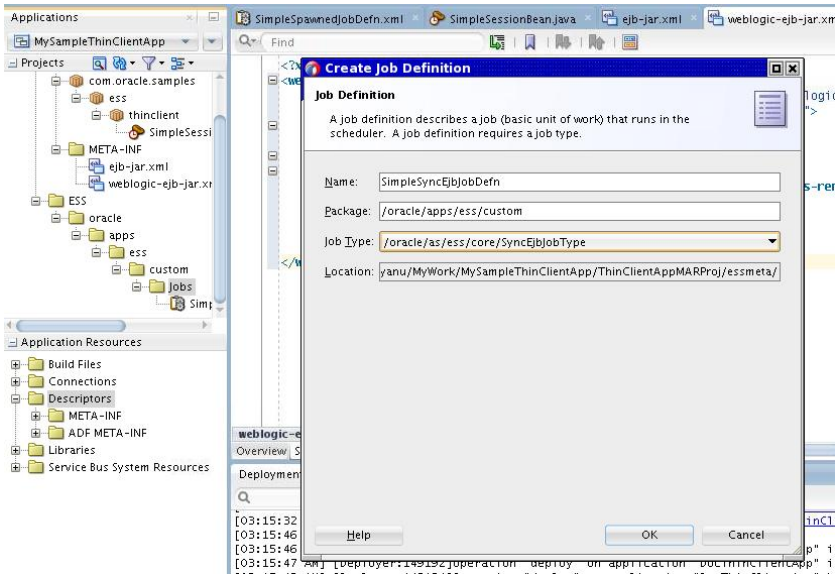
7. Make sure the `weblogic-ejb-jar.xml` file contains the declaration shown in Example 6-4.

   This completes the steps used to create an EJB that can be invoked by Oracle Enterprise Scheduler using the EJB job type.

   The following two steps describe how to use JDeveloper to create an EJB Job definition.

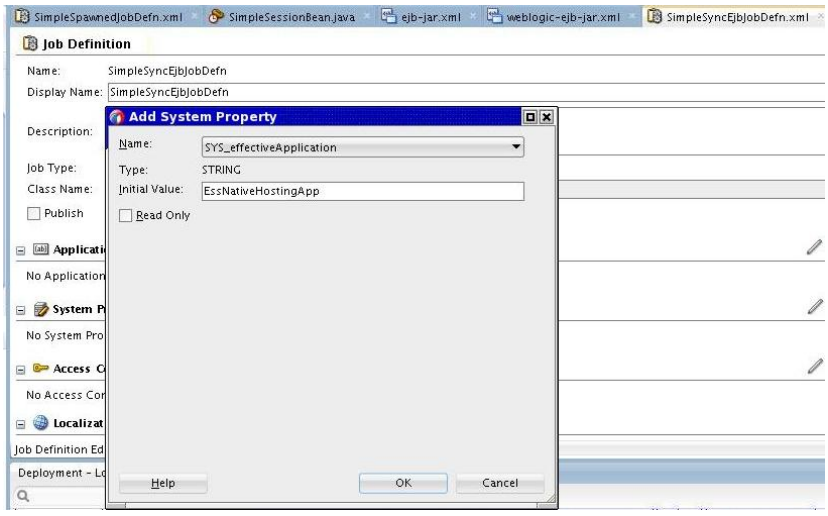
8. Create EJB job definition metadata. In the New Gallery dialog, select “Job Definition" under “Enterprise Scheduler Metadata" and fill it in as shown in Figure 6-22.

Figure 6-22 Create Job Definition Dialog



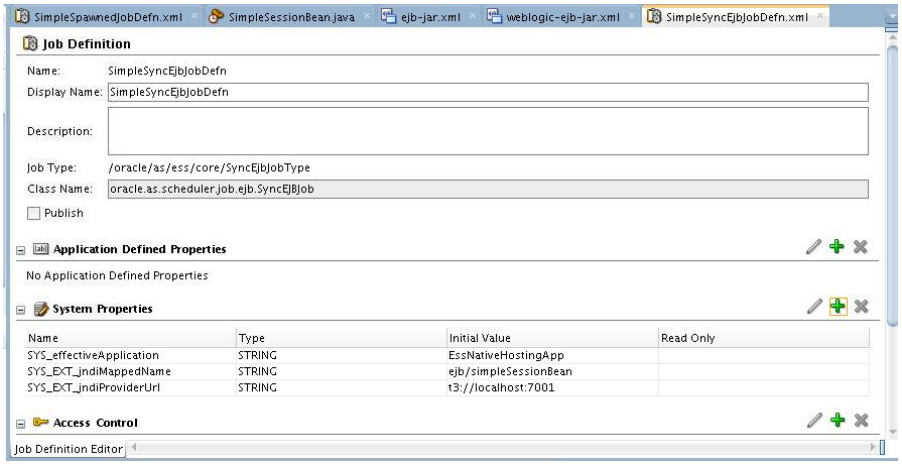
9. Configure system properties in the job definition. In the Add System Property dialog, specify `EssNativeHostingApp` in the **Initial Value** field, and select the system property `SYS_effectiveApplication` from the **Name** dropdown as shown in Figure 6-23.

Figure 6-23 Add System Property Dialog



Repeat the process, making sure that the SYS_EXT_jndiMappedName property is configured with an initial value of jndiName. Also add properties like SYS_EXT_jndiProviderUrl if the application is deployed to a server other than ESSAPP.

Figure 6-24 Job Definition Tab



Example 6-3 Contents of the weblogic-application.xml File

```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<ejb-jar 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."
         version="3.0" xmlns="http://java.sun.com/xml/ns/javaee">
  <enterprise-beans>
     <session>
        <description>Simple Session Bean</description>
        <ejb-name>SimpleSession</ejb-name>                    <ejb-
class>oracle.com.samples.ess.thinclient.SimpleSessionBean
        </ejb-class>
        <session-type>Stateless</session-type>
        <transaction-type>Container</transaction-type>
        <security-identity>
           <use-caller-identity/>
        </security-identity>
     </session>
   </enterprise-beans>
</ejb-jar>
```

Example 6-4 Contents of the weblogic-ejb-jar.xml File

```
<?xml version = "1.0" encoding = 'UTF-8' ?>
<weblogic-ejb-jar xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"                                xsi:schemaLocation="http://
xmlns.oracle.com/weblogic/weblogic-ejb-jar              http://xmlns.oracle.com"
            xmlns="http://xmlns.oracle.com/weblogic/weblogic-ejb-jar">
  <weblogic-enterprise-bean>
     <ejb-name>SimpleSession</ejb-name>
     <stateless-session-descriptor>
        <business-interface-jndi-name-map>
           <business-remote>oracle.as.scheduler.RemoteExecutable</business-
remote>
           <jndi-name>ejb/simpleSessionBean</jndi-name>
        </business-interface-jndi-name-map>
```

```
      </stateless-session-descriptor>
   </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

7

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*.

Note:

For Oracle Enterprise Scheduler sample code, be sure to see the sample site at `https://java.net/projects/oraclesoasuite12c`.

This chapter includes the following sections:

- How to Start JDeveloper to Support Building Oracle Enterprise Scheduler Applications
- Understanding Oracle Enterprise Scheduler Application Support Created by Oracle JDeveloper
- Building a Combined Oracle Enterprise Scheduler Application
- Building Split Submitting and Hosting Applications

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.

JDeveloper provides accessibility options, such as support for screen readers, screen magnifiers, and standard shortcut keys for keyboard navigation. You can also customize JDeveloper for better readability, including the size and color of fonts and the color and shape of objects. For information and instructions on configuring accessibility in JDeveloper, see "Oracle JDeveloper Accessibility Information" in *Developing Applications with Oracle JDeveloper*.

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
   ```

Understanding Oracle Enterprise Scheduler Application Support Created by Oracle JDeveloper

As you create projects in Oracle JDeveloper for developing Oracle Enterprise Scheduler applications, you add underlying support for application functionality by specifying support for particular project technologies.

For more on creating Oracle Enterprise Scheduler applications, see Building a Combined Oracle Enterprise Scheduler Application and Building Split Submitting and Hosting Applications.

When you create an application using Oracle JDeveloper, you select from the following technologies, depending on your application requirements:

- ESS Host Support for developing a hosting application, including:
  - Updating weblogic-application.xml for application support.
  - Updating EJB deployment profile for project support.
  - Updating EAR deployment profile for application support.
  - Adding the Oracle Enterprise Scheduler library.
  - Adding context menu to project (accessed by right-clicking and selecting Enterprise Scheduler Properties), which allows the following ejb-jar.xml properties to be modified: Logical Application Name, Application Policy Stripe, JPS Interceptor Application Name.
- ESS Client Support for developing a client application, including:
  - Updating weblogic-application.xml for application support.
  - Updating EJB deployment profile for project support.
  - Adding the Oracle Enterprise Scheduler library.
  - Adds context menu to project (accessed by right-clicking and selecting Enterprise Scheduler Properties), which allows the following ejb-jar.xml properties to be modified: JPS Interceptor Application Name.
- ESS Job Support for developing scheduler applications, including:
  - Creating or updating a MAR profile.

- Creating a JAR deployment profile for project support.
- Adding the Oracle Enterprise Scheduler library.

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 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:

- Creating the Application and Projects for EssDemoApp Application
- Creating Metadata and an Implementation Class for the EssDemoApp Application
- Adding Application Code to Submit Job Requests

- Setting Oracle Enterprise Scheduler Properties
- Assembling the EssDemoApp Application
- Deploying and Running the EssDemoApp Application

Creating the Application and Projects for EssDemoApp Application

Using Oracle JDeveloper you create an application and projects within the application that contains 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 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.

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:

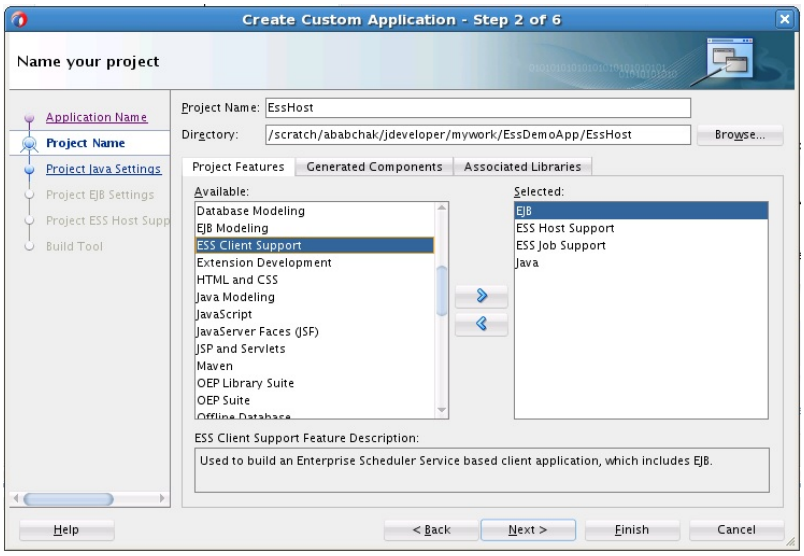
1. Start Oracle JDeveloper as described in How to Start JDeveloper to Support Building Oracle Enterprise Scheduler Applications.
2. In the Select Role dialog, select the **Default Role**, then click **OK**.
3. Click the Application menu, then click **New** and select the **From Gallery** option.
4. In the Name your application window enter the name and location for the new application.
    a. In the New Gallery window, select **Custom Application** listed under the **General Categories Applications** item, then click **OK**.
    b. In the **Application Name** field, enter an application name. For this sample application, enter `EssDemoApp`.
    c. In the **Directory** field, accept the default.
    d. 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.
    e. Click **Next**.
5. In the Name your project window, enter the name for the host project you're creating and select supporting technologies. See Figure 7-1.
    a. In the **Project Name** field, enter a name for your hosting project. For this sample application, enter `EssHost`.
    b. On the **Project Features** 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.

For more on these, see Understanding Oracle Enterprise Scheduler Application Support Created by Oracle JDeveloper.

c. Click **Next**.

Figure 7-1 Create the Custom Application



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.x, 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.

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 provides support for client interaction with the hosting application.

To create the client project:

1. Click the File menu, then click **New > Project**.

2. In the New Gallery, under Categories, expand General, then click **Projects**.
3. Under Items, click **Custom 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. See Figure 7-2.
   a. In the **Project Name** field, enter a name for your client project. For this sample application, enter `EssClient`.
   b. On the **Project Features** 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 & CSS

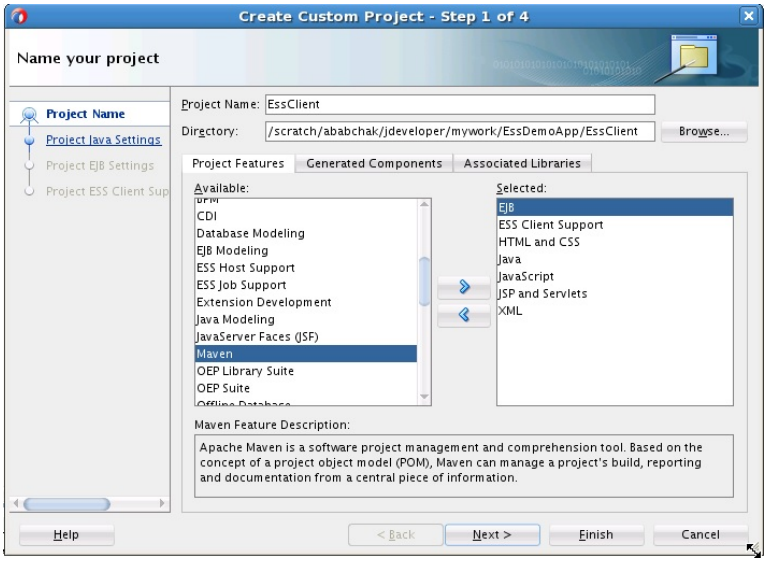
      JSF

      SP and Servlets

      XML

      For more on this, see Understanding Oracle Enterprise Scheduler Application Support Created by Oracle JDeveloper.
   c. Click **Next**.

Figure 7-2 Create a Custom Project



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**.

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.

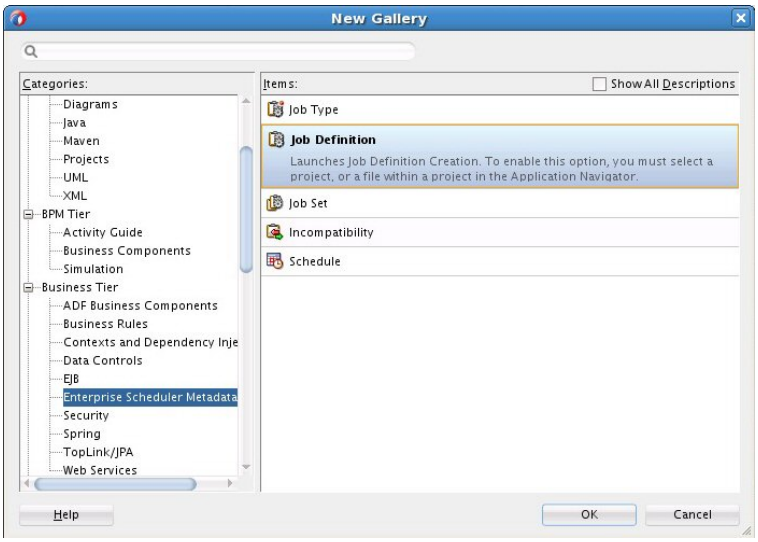
How to Create Metadata for the EssDemoApp Application

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 **Categories** area expand **Business Tier** and select **Enterprise Scheduler Metadata**.
4. In the **Items** area, select **Job Definition** as shown in Figure 7-3.

Figure 7-3 Adding Job Type Metadata to the Sample Application



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

6. 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" is not visible in Oracle JDeveloper.

   c. In the **Job Type** field, from the dropdown list select **/oracle/as/ess/core/JavaJobType**.

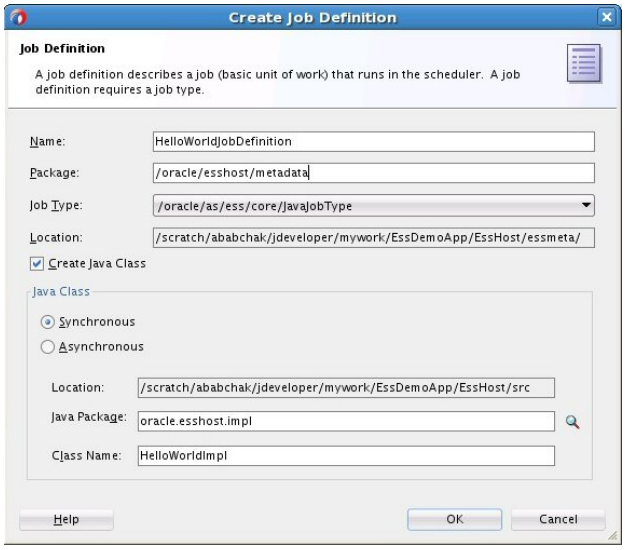
      If job types are not listed in the dropdown, ensure that you started Oracle JDeveloper as described in How to Start JDeveloper to Support Building Oracle Enterprise Scheduler Applications.

   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 is 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 7-4

Figure 7-4 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.

7. 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.

8. 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.

9. Open HelloWorldImpl.java in the source editor.

10. 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 7-1.

11. Save and close HelloWorldImpl.java.

Example 7-1 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()
  {
  }
}
```

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).

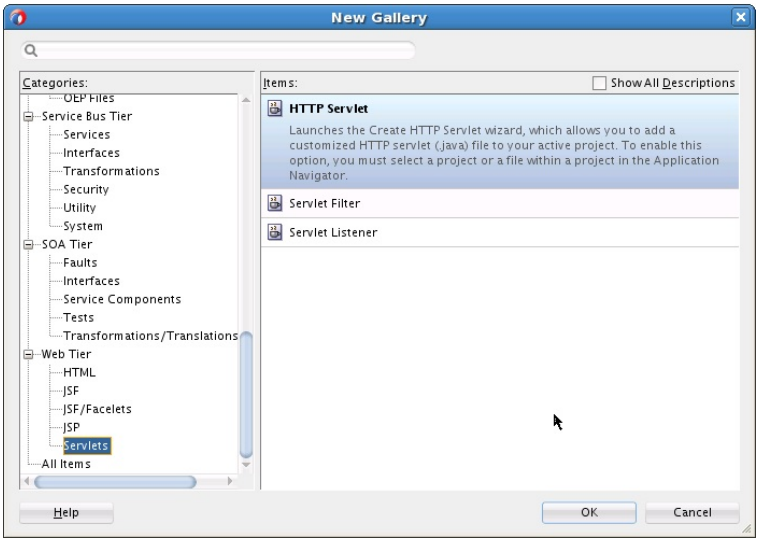
How to Add Application Code to Submit Job Requests

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 7-5.

Figure 7-5 Adding Job Type Metadata to the Sample Application

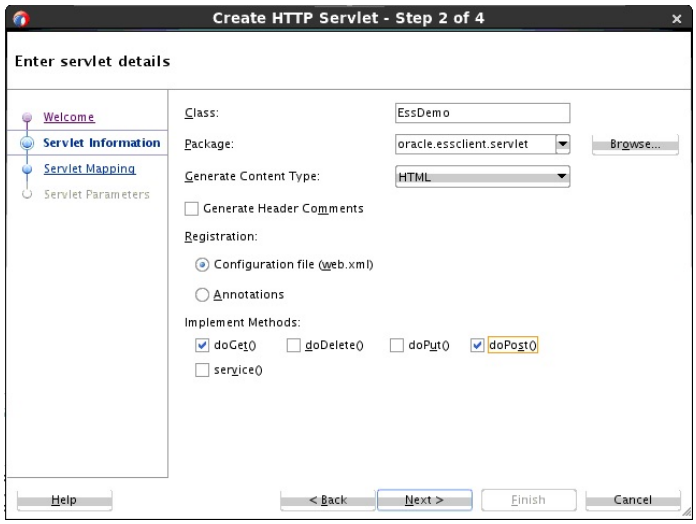


5. Click **OK**. This displays the Create HTTP Servlet wizard.
6. In the Welcome page, click **Next**.
7. In the Create HTTP Servlet - Step 2 of 4: Servlet Information page, specify the following:
   a. In the **Class** field, enter a name for the servlet class. For this example, enter the name: `EssDemo`.
   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 7-6.

Figure 7-6 Creating a Servlet -- Step 2 of 4



e. In the **Registration** area, select the **Configuration file (web.xml)** radio button.

f. Click **Next**.

8. In the Create HTTP Servlet - Step 3 of 4: 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.

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 runtime to determine which set of security policies are applicable.

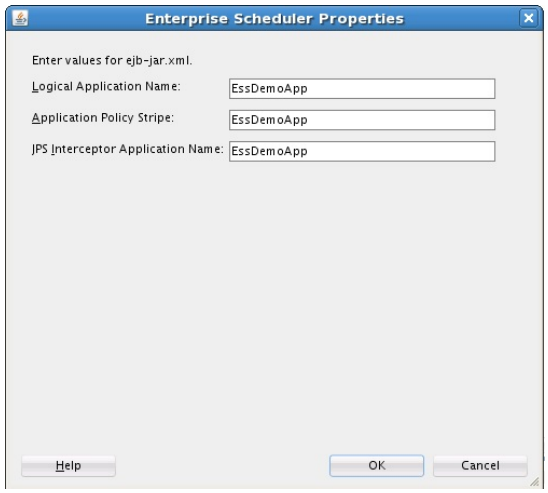
How to Set Oracle Enterprise Scheduler Properties for the Application

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**.

Figure 7-7 Set Values for Oracle Enterprise Scheduler Properties



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.

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

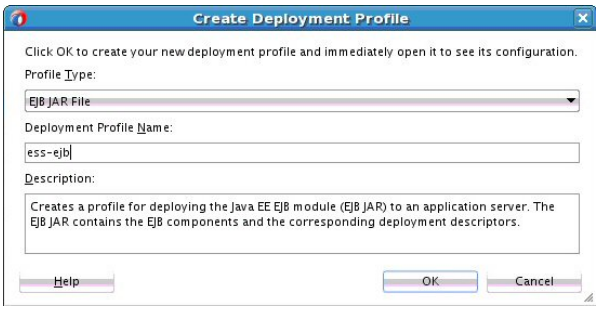
The sample application must contain the required EJB descriptors. You need to create the ejb-jar.xml and weblogic-ejb-jar.xml files and include these files with any Java implementation class that you create.

Oracle Enterprise Scheduler requires an application to assemble and provide an EJB JAR so that Oracle Enterprise Scheduler can find its entry point in the application while running job requests on behalf of the application. This EJB jar should have its required EJB descriptors in ejb-jar.xml and weblogic-ejb-jar, as well as any Java class implementations that are going to be submitted to Oracle Enterprise Scheduler. The descriptor files ejb-jar.xml and weblogic-ejb-jar must contain descriptions for the Oracle Enterprise Scheduler EJBs and should not be modified.

To 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 **Profile 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**.

Figure 7-8 Create the EJB-JAR Deployment Profile



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 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. In the JAR Option window, deselect the **Include Manifest File** item.
14. Click **OK**.
15. In the Project Properties dialog, click **OK**.

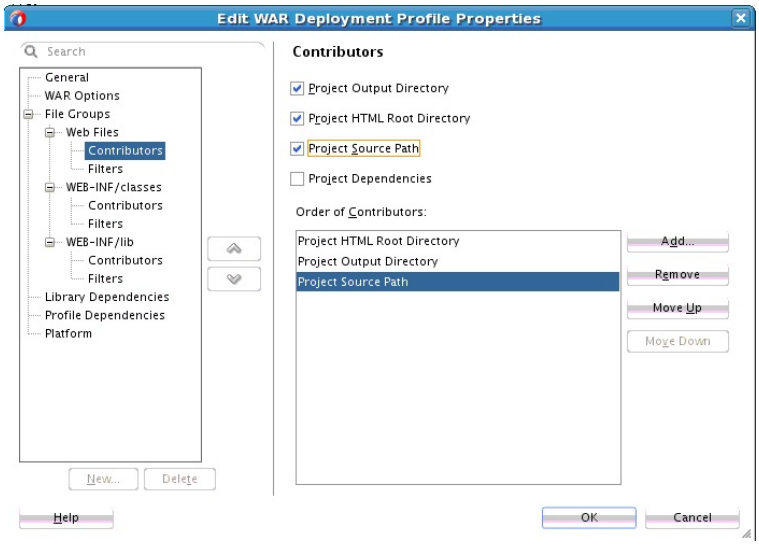
How To Update the WAR Archive Options

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. Delete all profiles listed in the Under Deployment Profiles window, then 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 WAR Options window deselect `Include Manifest File(META-INF/MANIFEST.MF)`.
9. In the General window, select the **Specify Java EE Web Context Root** option. In the field beneath the option, enter `EssDemoApp`.
10. In the navigator, expand to File Groups > Web Files > Contributors.
11. In the Contributors window, select the following check boxes as shown in Figure 7-9:
    - Project Output Directory
    - Project HTML Root Directory
    - Project Source Path

Figure 7-9 Update the WAR Archive Options



12. In the navigator, expand to File Groups > Web Files > Filters.
13. 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**.
14. Navigate to the **Project Properties** > **Libraries and Classpath** window. Use the **Add Library** button to add the following libraries:

    - ADF Common Runtime
    - ADF Faces Runtime11
    - ADF Common Web Runtime
    - ADF Page FlowRuntime
    - ADF Controller Schema
    - ADF Controller Runtime

Create the Application MAR File

To create the MAR 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. Select **MAR File Option** from dropdown menu and enter `MAR_EssDemoApp` as the deployment profile's name.
5. Click **OK**.

6. In the Edit MAR Deployment Profile Properties dialog, in the navigation pane, navigate to Metadata File Groups > User Metadata.
7. Add the `EssDemoApp/EssHost/essmeta` directory.

   This selects the appropriate Oracle Enterprise Scheduler user metadata for the application.
8. Navigate to Metadata File Groups > User Metadata > Directories and select **Directories**. Select the bottom most directory in the tree.

   This is the directory from which the name space is created. For example, when you select **oracle**, the name space is `oracle`. When you select the **product** directory, the names pace is `oracle/apps/product`. To create the name space `/oracle/esshost/metadata`, click the **metadata** directory. The folder you select in this dialog determines the top-level name space in the `adf-config.xml` file.
9. Click **OK**.

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:

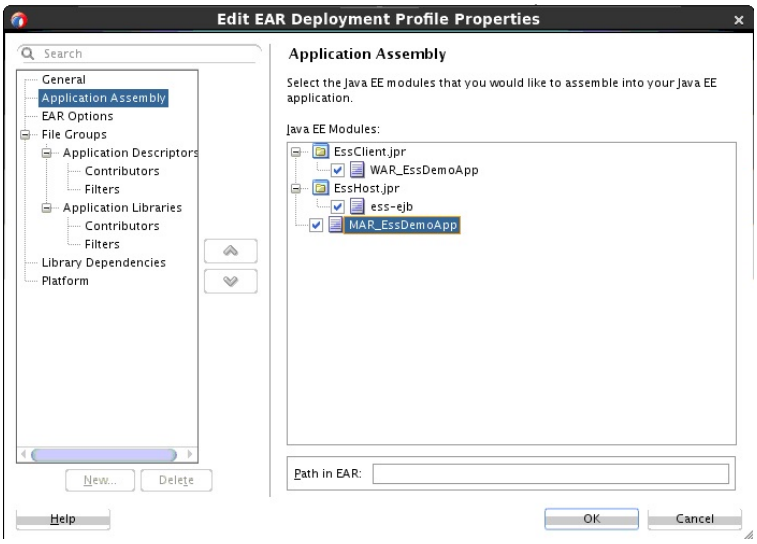
- EJB JAR including the Oracle Enterprise Scheduler Java job implementation.
- WAR archive with the EssDemo servlet.

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. Under Deployment Profiles, delete all profiles listed in the window, then 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. In the EAR Options window, select **Include Manifest File** and add `EssHost/src/META-INF/MANIFEST.MF`.
8. Click **OK**.
9. In the Application Properties dialog, click **OK**.

Figure 7-10 Update the EAR Archive Options



Configure Security for the Application

You must create a user that is assigned to the `EssApplicationRole` role. The following steps describe how to configure security for the back-end hosting application:

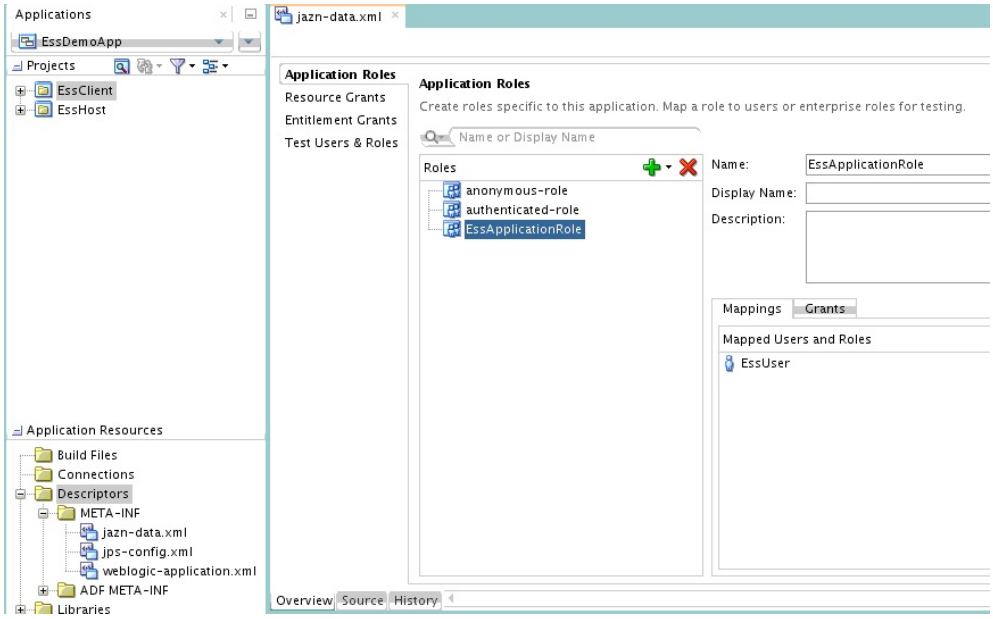
1. Select **Application** > **Secure** > **Configure ADF Security** from the main menu.
2. In the ADF Security page of the Configure ADF Security wizard, select ADF Authentication, then click **Next**.
3. In the Authentication Type page, choose **EssClient.jpr** in the **WebProject** dropdown.
4. Select **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 **META-INF**. This file contains a security context or security stripe named after the application.
6. Select **Application** > **Secure** > **Test Users & Roles** from the main menu.

   A file named `jazn-data.xml` is generated.
7. In the overview editor for the `jazn-data.xml` file, click the **Add** button in the **Users** list.

   Set the name to `EssUser` and set the password to `welcome1`.
8. Click the **Application Roles** navigation tab to open the Application Roles window as shown in Figure 7-11.

Figure 7-11 Application Roles Window



9. Click the **Add** button in the **Roles** list and choose **Add New Role**.
10. Set the name to `EssApplicationRole`.
11. Click the **Add** button in the Mappings tab and choose **Add User**.
12. Select `EssUser` and click **OK**.

Add Resource Grants for ESS Application Role in the Job Definition

The following steps describe how to update the job definition by adding resource grants for the ESS application role.

1. In the HelloWorldJobDefinition.xml Job Definition page, in the **Description** field, enter `HelloWorld Example`.
2. In the System Properties section, click the **Add** button.
3. In the Add System Property dialog, from the **Name** dropdown menu, select **SYS_effectiveApplication**.
4. In the **Initial Value** field, enter `EssDemoApp`.
5. Click **OK**.
6. In the Access Control section, click the **Add** button.
7. In the Add Access Control dialog, from the **Role** dropdown menu, select **EssApplicationRole**. This is the role that you created Configure Security for the Application.
8. Select the **Read** and **Execute** actions.
9. Click **OK**.
10. Verify that the contents of the generated file are identical to Example 7-2.

Example 7-2 jazn-data.xml

```
<?xml version = '1.0' encoding = 'UTF-8' standalone = 'yes'?>
<jazn-data xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/jazn-
data.xsd">
  <jazn-realm default="jazn.com">
   <realm>
    <name>>Username / password is: EssUser / welcome1</name>
     <users>
      <user>
       <name>EssUser</name>
       <credentials>{903}LmqEdVs3zO0/QmP90tihXv4nRq5YqYSL</credentials>
      </user>
     </users>
    </realm>
   </jazn-realm>
   <policy-store>
    <applications>
     <application>
      <name>EssDemoApp</name>
      <app-roles>
       <app-role>
        <name>EssApplicationRole</name>
        <class>oracle.security.jps.service.policystore.ApplicationRole</class>
        <members>
         <member>
        <class>oracle.security.jps.internal.core.principals.JpsXmlUserImpl</
class>
          <name>EssUser</name>
         </member>
        </members>
       </app-role>
      </app-roles>
     <jazn-policy>
      <grant>
       <grantee>
        <principals>
         <principal>
          <class>oracle.security.jps.service.policystore.ApplicationRole</class>
          <name>EssApplicationRole</name>
         </principal>
        </principals>
       </grantee>
       <permissions>
        <permission>
                    <class>oracle.as.scheduler.security.MetadataPermission</
class>
        <name>oracle.esshost.metadata.JobDefinition.HelloWorldJobDefinition</
name>
         <actions>Read,Update,Delete,Execute</actions>
        </permission>
       </permissions>
      </grant>
     </jazn-policy>
    </application>
   </applications>
  </policy-store>
 <system-policy/>
</jazn-data>
```

Configure the weblogic-application.xml File

Use the source editor to remove the following lines from `weblogic-application.xml`:

```
<library-ref>
  <library-name>oracle.applcp.runtime</library-name>
</library-ref>
<library-ref>
  <library-name>oracle.xdo.runtime</library-name>
</library-ref>
```

Update the EssHost MANIFEST File

Replace the content of the EssHost `META-INF/MANIFEST.INF` file with the following lines:

```
Manifest-Version: 1.0
Weblogic-Application-Version: 3.0
Extension-List: essruntime
essruntime-Extension-Name: oracle.ess.runtime
essruntime-Specification-Version: 12
```

Change the Realm Field

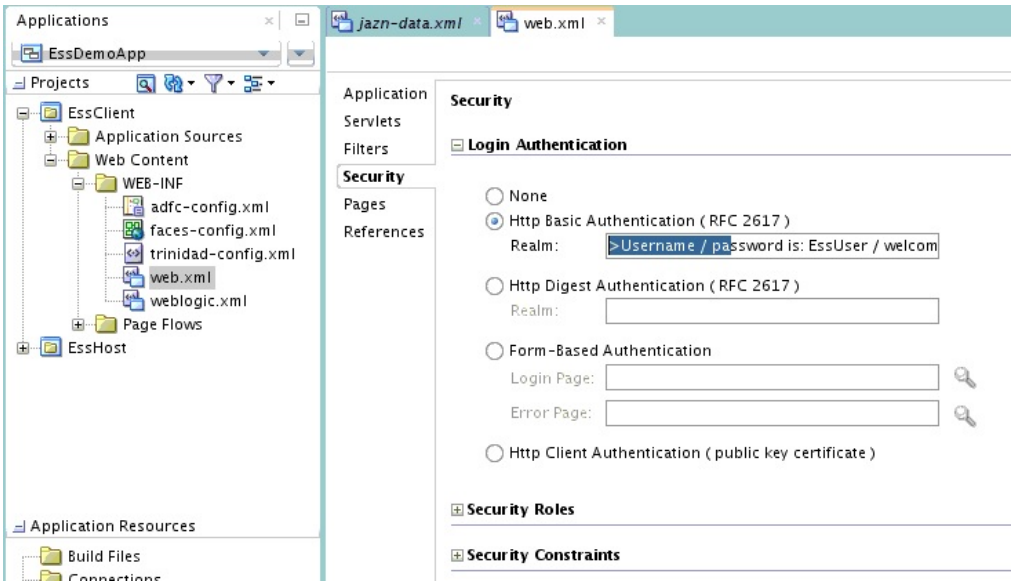
Navigate to **EssClient** > **Web Content** > **WEB-INF** > **web.xml** as shown in Figure 7-12. Change the value in the Security window **Realm** field from:

- `jazn.com`

to:

- `Username / password is: EssUser / welcome1`

Figure 7-12 Change the Realm Field



Edit the adf-config.xml File for the EssDemoApp Application

1. In 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 code shown in Example 7-3.

Example 7-3 adf-config.xml File for a EssDemoApp Application

```
<?xml version="1.0" encoding="UTF-8" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config" xmlns:sec="http://
xmlns.oracle.com/adf/security/config">
  <sec:adf-security-child xmlns="http://xmlns.oracle.com/adf/security/config">
    <CredentialStoreContext
credentialStoreClass="oracle.adf.share.security.providers.jps.CSFCredentialStore"
                        credentialStoreLocation="../../src/META-INF/jps-
config.xml"/>
    <sec:JaasSecurityContext
initialContextFactoryClass="oracle.adf.share.security.JAASInitialContextFactory"

jaasProviderClass="oracle.adf.share.security.providers.jps.JpsSecurityContext"
    authorizationEnforce="false" authenticationRequire="true"/>
  </sec:adf-security-child>
  <adf-mds-config xmlns="http://xmlns.oracle.com/adf/mds/config">
    <mds-config version="11.1.1.000" xmlns="http://xmlns.oracle.com/mds/config">
      <persistence-config>
        <metadata-namespaces>
          <namespace path="/oracle/as/ess/core" metadata-store-usage="ess-core"/>
        </metadata-namespaces>
        <metadata-store-usages>
          <metadata-store-usage id="ess-core" deploy-
target="false"                default-cust-store="false">
            <metadata-store               class-
name="oracle.mds.persistence.stores.db.DBMetadataStore">
              <property name="jndi-datasource" value="jdbc/mds-ESS_MDS_DS"/>
              <property name="repository-name" value="mds-ESS_MDS_DS"/>
              <property name="partition-name" value="essapp-internal-partition"/>
            </metadata-store>
          </metadata-store-usage>
        </metadata-store-usages>
      </persistence-config>
    </mds-config>
  </adf-mds-config>
</adf-config>
```

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.

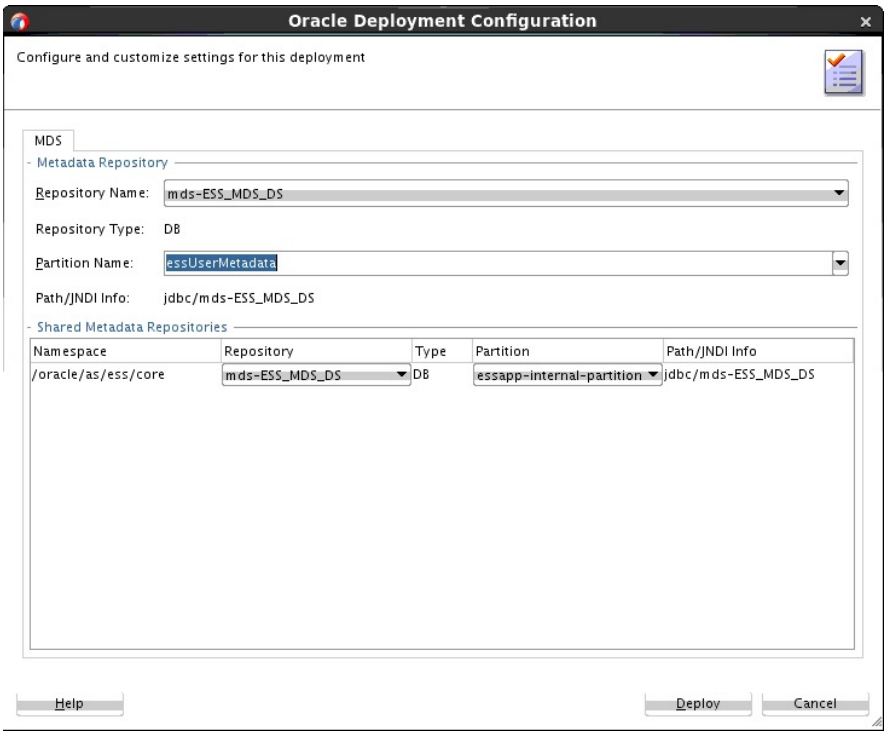
How to Deploy the EssDemoApp Application

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 to make sure the Oracle WebLogic Server is up and running. If the Oracle WebLogic Server is not running, start the server. Make sure Oracle JDeveloper has a connection to the server (for this example, "MyConnection").
2. In the Application Navigator, select the **EssDemoApp** application.
3. In the Application Navigator from the Application Menu select **Deploy > EAR_EssDemoApp > to > MyConnection**.
4. Oracle JDeveloper shows the Deployment Configuration page. Select the appropriate options for your Metadata Repository.
5. Make the following choices when prompted during deployment. In the Metadata Repository section choose the repository and partition names as follows and shown in Figure 7-13:
   a. Repository Name: **mds-ESS_MDS_DS**
   b. Partition Name: `essUserMetadata`

Figure 7-13 Oracle Deployment Configuration Window



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

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:

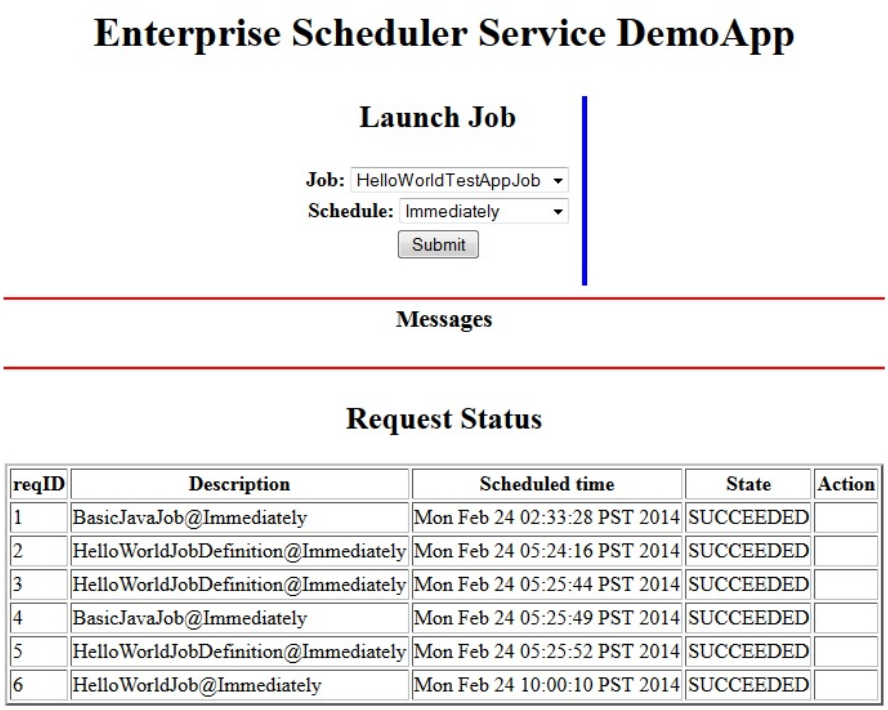
http://*host*:*http-port*/*context-root*/essdemo

For example,

```
http://myserver.example.com:7101/EssDemoApp/essdemo
```

This shows the EssDemo servlet, as shown in Figure 7-14.

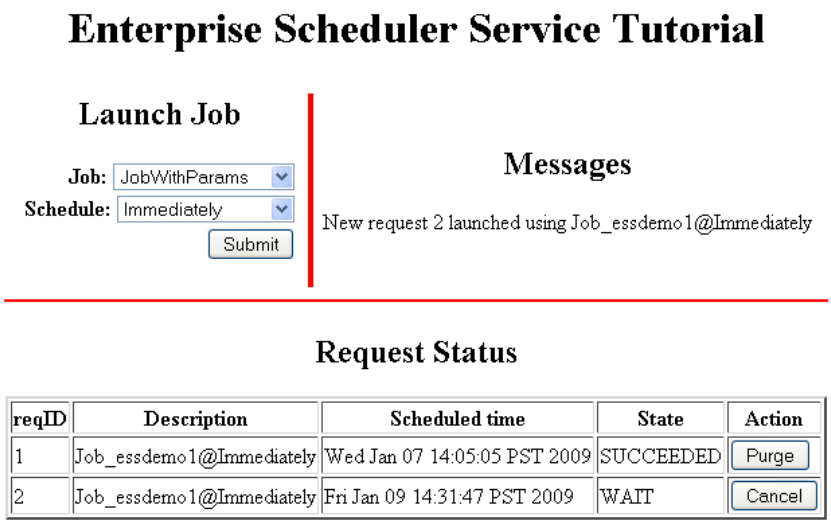
Figure 7-14 Running EssDemo Servlet for Oracle Enterprise Scheduler Sample Application



reqID	Description	Scheduled time	State	Action
1	BasicJavaJob@Immediately	Mon Feb 24 02:33:28 PST 2014	SUCCEEDED	
2	HelloWorldJobDefinition@Immediately	Mon Feb 24 05:24:16 PST 2014	SUCCEEDED	
3	HelloWorldJobDefinition@Immediately	Mon Feb 24 05:25:44 PST 2014	SUCCEEDED	
4	BasicJavaJob@Immediately	Mon Feb 24 05:25:49 PST 2014	SUCCEEDED	
5	HelloWorldJobDefinition@Immediately	Mon Feb 24 05:25:52 PST 2014	SUCCEEDED	
6	HelloWorldJob@Immediately	Mon Feb 24 10:00:10 PST 2014	SUCCEEDED	

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 7-15.

Figure 7-15 Running EssDemo Servlet with Request Status for Submitted Requests



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`.

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 Creating the Application and Projects for EssDemoApp Application, 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:

- How to Create the Back-End Hosting Application for EssDemoApp
- How to Create the Front-End Submitter Application for Oracle Enterprise Scheduler

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.

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.

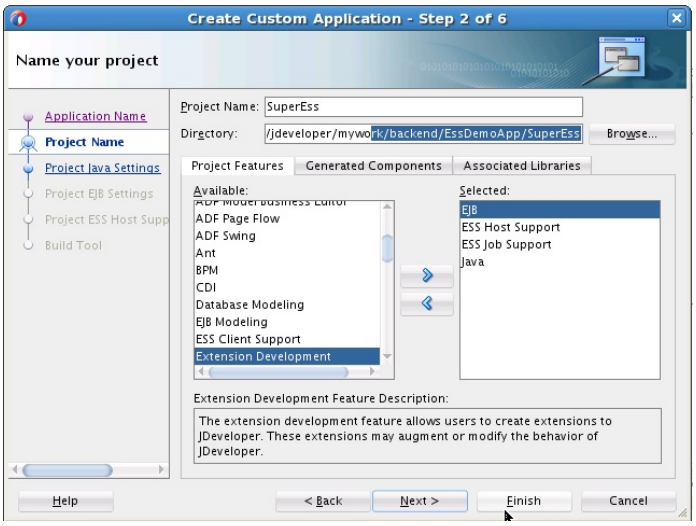
To create the back-end hosting application:

1. From JDeveloper choose **File > New** from the main menu.
2. In the New Gallery, expand **General**, select **Applications** and then **Custom 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 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 Features** 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.

For more on these, see Understanding Oracle Enterprise Scheduler Application Support Created by Oracle JDeveloper.

c. Click **Next**.

Figure 7-16 Create the Back-End Hosting Application



6. In the Configure Java Settings page, change the default package to `oracle.apps.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**.

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:

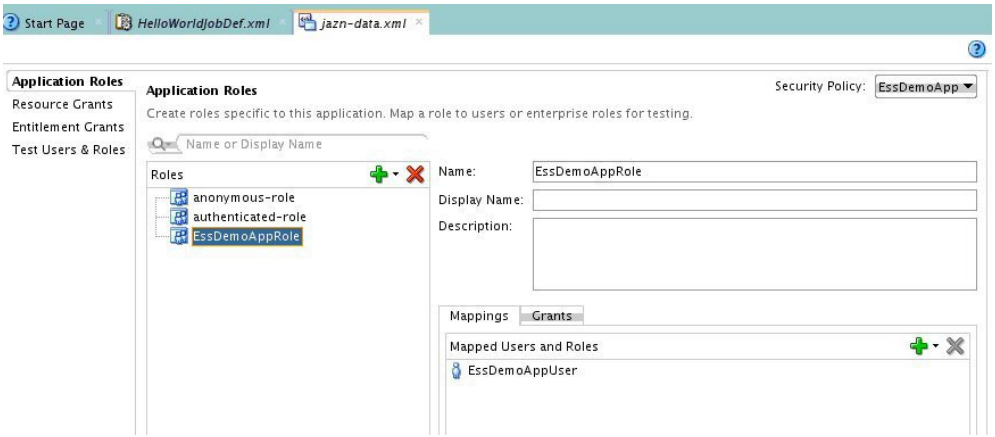
1. Select **Application** > **Secure** > **Configure ADF Security** from the main menu.
2. In the ADF Security page of the Configure ADF Security wizard, select **ADF Authentication**, then click **Next**.
3. In the Authentication Type page, accept the default values as this application does not have 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** > **Test Users & Roles** 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** button 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** button in the **Roles** list and choose **Add New Role**.
10. Set the name to `EssDemoAppRole`.
11. Click the **Add** button in the Mappings tab and choose **Add User**.
12. Select `EssDemoAppUser` and click **OK**.

Figure 7-17 Configuring Security



Defining Metadata for the Back-End Hosting Application

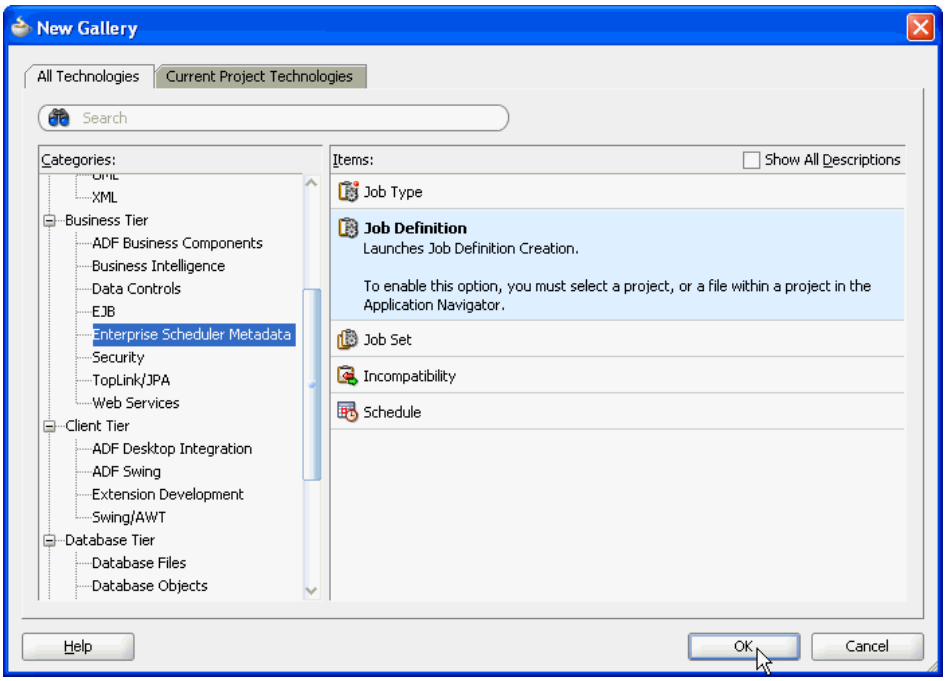
To use the Oracle Enterprise Scheduler split application to submit a job request you need to create metadata that defines a job request, including the following:

- 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 **Categories** area expand **Business Tier** and select **Enterprise Scheduler Metadata**.
4. In the **Items** area, select **Job Definition** as shown in Figure 7-18.

Figure 7-18 Adding Job Definition to the Sample Application



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

6. In the Create Job Definition dialog, specify the following as shown in Figure 7-19:

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

   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 is a synchronous Java job.

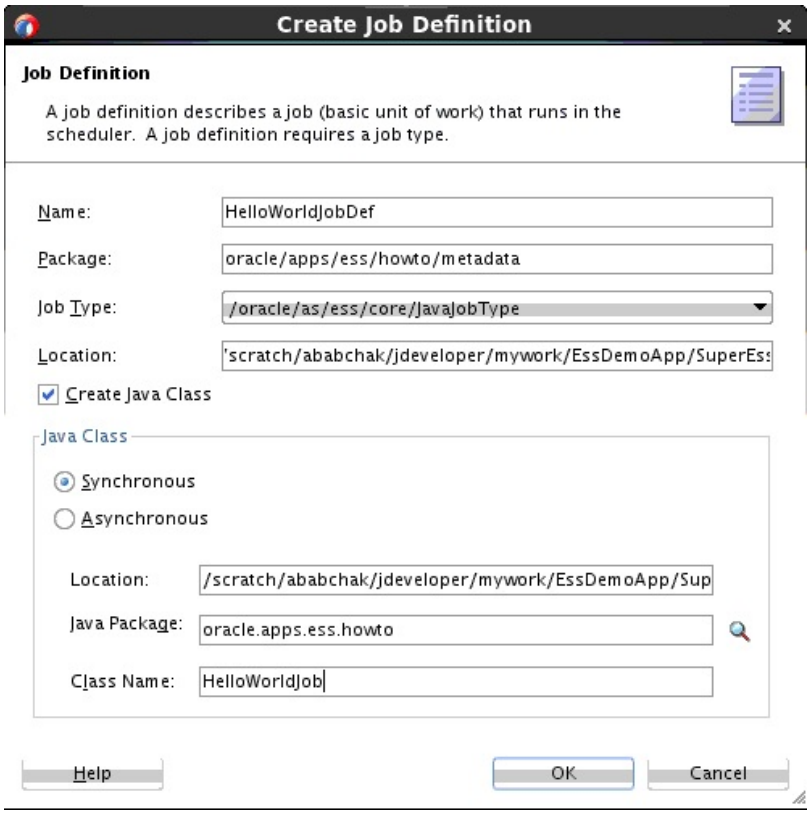
   e. Under **Java Class**, specify details for the Java class you're creating. In the **Java Package** field, enter its package name -- here, enter `oracle.apps.ess.howto`. In the **Class Name** field, enter a name for the class -- here, enter `HelloWorldJob`.

   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.

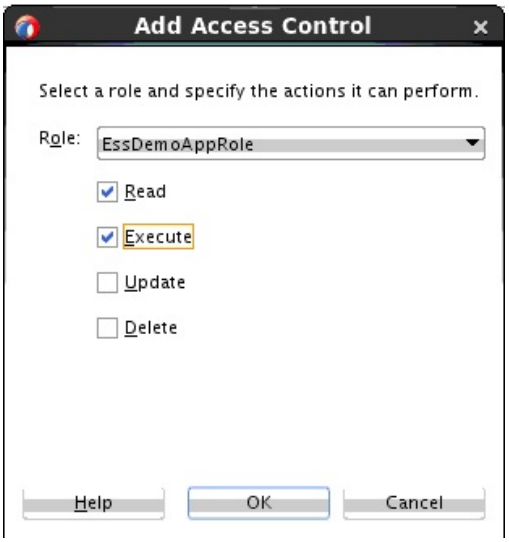
On the Job Definition page, you can edit job definition metadata, including properties that specify parameters for the job, access to this metadata, and a resource bundle to use for localization.

Figure 7-19 Create a Job Definition



7. In the HelloWorldJobDef.xml Job Definition page, in the **Description** field, enter `HelloWorld Example`.
8. In the System Properties section, click the **Add** button.
9. In the Add System Property dialog, from the **Name** dropdown, select **SYS_effectiveApplication**.
10. In the **Initial Value** field, enter `EssDemoApp`.
11. Click **OK**.
12. In the Access Control section, click the **Add** button.
13. In the Add Access Control dialog, from the **Role** dropdown, ensure that **EssDemoAppRole** is selected. This is the role that you created during Configuring Security for the Back-End Hosting Application.
14. Select the Read and Execute actions as shown in Figure 7-20.

Figure 7-20 Add Access Control Dialog



15. Click **OK**.

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 7-4.

Example 7-4 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());
```

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 runtime 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**.

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

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 **Profile 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 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. In the JAR Option window, deselect the **Include Manifest File** item.
12. Click **OK**.
13. In the Project Properties dialog, click **OK**.

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 EJB JAR including the Oracle Enterprise Scheduler Java job implementation.

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** then click the **Add** button to add a contributor and add this directory: `EssDemoApp/SuperEss/essmeta`
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 Profile Type dropdown, select **EAR File**.
8. In the **Name** field, enter `EAR_EssDemoAppEar`.

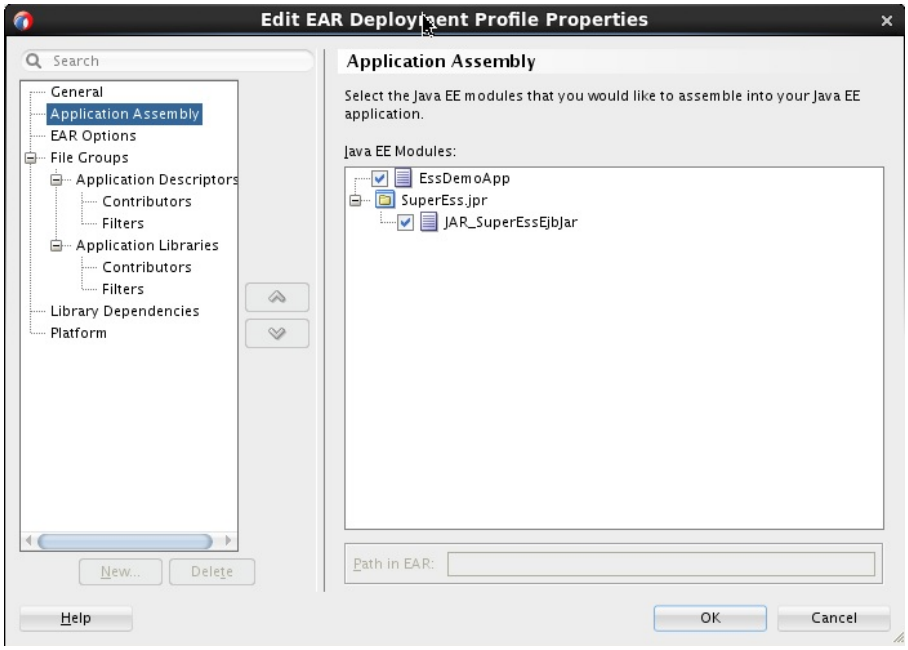
   Click **OK**.
9. In the Edit EAR Deployment Profile Properties dialog, in the navigation pane, select **General**.

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 as shown in Figure 7-21.

Figure 7-21 Edit EAR Deployment Profile Properties



13. In the EAR Options window, select **Include Manifest File** and add `EssDemoApp/SuperEss/src/META-INF/MANIFEST.MF`.

14. Click **OK**.

15. In the Application Properties dialog, click **OK**.

Update the SuperEss MANIFEST File

Replace the content of the SuperEss `META-INF/MANIFEST.INF` file with the following lines:

```
Manifest-Version:
1.0


Weblogic-Application-Version: 3.0
Extension-List: essruntime
essruntime-Extension-Name: oracle.ess.runtime
essruntime-Specification-Version: 12
```

Configure the weblogic-application.xml File

Use the source editor to remove the following lines from `weblogic-application.xml`:

```
<library-ref>
  <library-name>oracle.applcp.runtime</library-name>
</library-ref>
```

```
<library-ref>
  <library-name>oracle.xdo.runtime</library-name>
</library-ref>
```

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.

Edit the adf-config.xml File for the EssDemoApp Application

In the **Application Resources** panel:

1. Expand **Descriptors**.
2. Expand **ADF META-INF**.
3. Double-click **adf-config.xml**.
4. In the source editor, replace the contents of the `adf-config.xml` file with the XML shown in Example 7-5.

Example 7-5 adf-config.xml File

```
<?xml version="1.0" encoding="UTF-8" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config" xmlns:config="http://
xmlns.oracle.com/bc4j/configuration">
  <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-adfm-config xmlns="http://xmlns.oracle.com/adfm/config">
    <defaults changeEventPolicy="ppr"
useBindVarsForViewCriteriaLiterals="true"
useBindValuesInFindByKey="true"/>
    <startup>
       <amconfig-overrides>
         <config:Database jbo.locking.mode="optimistic"/>
      </amconfig-overrides>
    </startup>
  </adf-adfm-config>
   <adf-mds-config xmlns="http://xmlns.oracle.com/adf/mds/config">
    <mds-config version="11.1.1.000" xmlns="http://xmlns.oracle.com/mds/config">
      <persistence-config>
        <metadata-namespaces>
          <namespace path="/oracle/as/ess/core" metadata-store-usage="ess-core"/>
        </metadata-namespaces>
        <metadata-store-usages>
          <metadata-store-usage id="ess-core" deploy-target="false"
default-cust-store="false">
            <metadata-store                    class-
```

```
name="oracle.mds.persistence.stores.db.DBMetadataStore">
              <property name="jndi-datasource" value="jdbc/mds-ESS_MDS_DS"/>
              <property name="repository-name" value="mds-ESS_MDS_DS"/>
              <property name="partition-name" value="essapp-internal-partition"/>
            </metadata-store>
          </metadata-store-usage>
        </metadata-store-usages>
      </persistence-config>
    </mds-config>
  </adf-mds-config>
</adf-config>
```

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.

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 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**.

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 **Custom 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 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 Features** tab, under **Available**, double-click **ESS Client Support**, **JSP and Servlets**, so that both are listed under **Selected** on the right side of the dialog box.

      For more on ESS Client Support, see Understanding Oracle Enterprise Scheduler Application Support Created by Oracle JDeveloper.

   Click **Next**.

4. In the **Default Package** field of the Configure Java settings window, enter `oracle.apps.ess.howto`. Click **Finish**.

5. In the Configure EJB Settings window, under **EJB Version**, select the **Enterprise JavaBeans 3.0** option button.

6. In the Configure EJB Settings window, under **EJB Version 3.x**, select the **Generate ejb-jar.xml in this project** check box.

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 simply shares the users and roles created by the EssDemoApp application.

To configure security for the front-end submitter application:

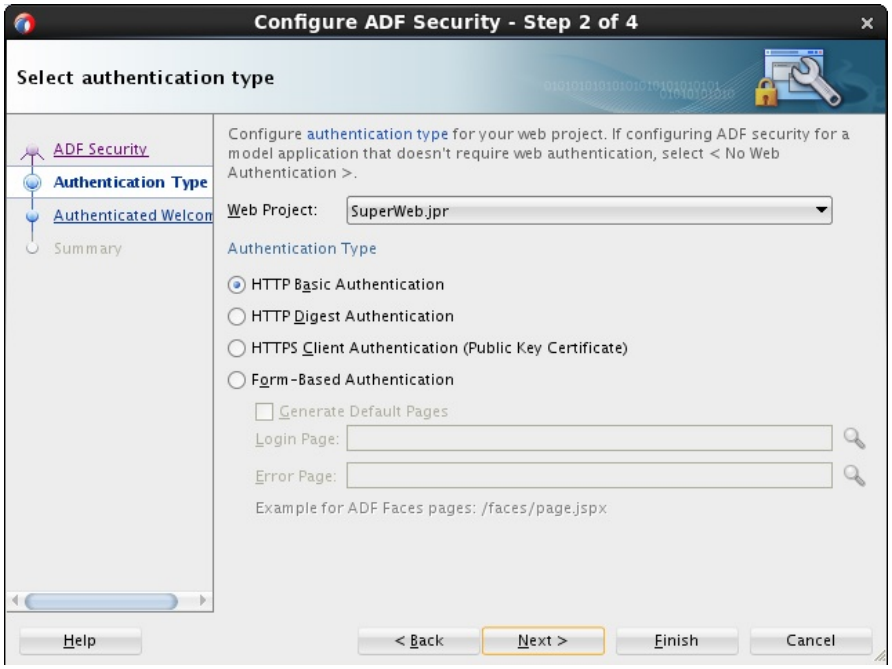
1. Select **Application** > **Secure** > **Configure ADF Security** from the main menu.

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**.

Figure 7-22 Configure ADF Security



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**.

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 Create HTTP Servlet - Step 1 of 3: Servlet Information page, enter `EssDemoAppServlet` in the **Class** field.
4. Change the selection from **Annotation** to **Configuration File(web.xml)**.
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 7-6.

Example 7-6 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 is the reqID), and
             * match it to the value (Purge or Cancel).
             * Assumptions are that there is only one entry in the map
             * per request (one purge or cancel).  Also, that the datatypes
             * for the key and value are 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 is 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 appears 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 cellspacing=6><tr>");

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

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

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

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

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

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

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

        renderStatusTable(out, rl.requestDetails);

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

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

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

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

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

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

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

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

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

```
            return;

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

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

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

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

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

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

            if (scheduledTime == null) {
                scheduledTimeString = "null scheduled time";
            } else {
                scheduledTimeString = String.valueOf(scheduledTime.getTime());
            }

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

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


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

    private RuntimeService getRuntimeService() throws Exception {
```

```
        return JndiUtil.getRuntimeServiceEJB();
    }

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

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

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

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

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

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

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

```
    }
}

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

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

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

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

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

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

```
            }
        }

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

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

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

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

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

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

            RuntimeServicePayload<List<RequestDetail>> myPayload2 =
                new RuntimeServicePayload<List<RequestDetail>>()
            {
```

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

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

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

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

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

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

}
```

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

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

To edit the web.xml file for the 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 7-1.

Table 7-1 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/*`.

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 7-7.

Example 7-7 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 uses 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.ESSApplicationLifecycleListener</
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>
```

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 7-8.

Example 7-8 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>
```

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

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. Delete the other JAR profiles created by default. Only include EJB and WAR.
6. On the Project Properties dialog, click **OK**.

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**.

How to Assemble the MAR and EAR Files for the Front-End Hosting Application

The sample application must 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 **Profile 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 **Profile 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**.

Add ADF Libraries

Navigate to the Project Properties > Libraries and Classpath window. Use the **Add Library** button to add the following libraries:

- ADF Common Runtime
- ADF Faces Runtime11
- ADF Common Web Runtime
- ADF Page FlowRuntime
- ADF Controller Schema
- ADF Controller Runtime

Set Oracle Enterprise Scheduler Properties for the Application

The following steps describe how 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 `EssDemoAppUI` as the value for all three of the following fields:
   - **Logical Application Name**
   - **Application Policy Stripe**
   - **JPS Interceptor Application Name**
3. Click **OK**.

Configure the weblogic-application.xml File

Use the source editor to remove the following lines from `weblogic-application.xml`:

```
<library-ref
  <library-name>oracle.xdo.runtime</library-name>
</library-ref>
<library-ref>
 <library-name>oracle.applcp.runtime</library-name>
</library-ref>
```

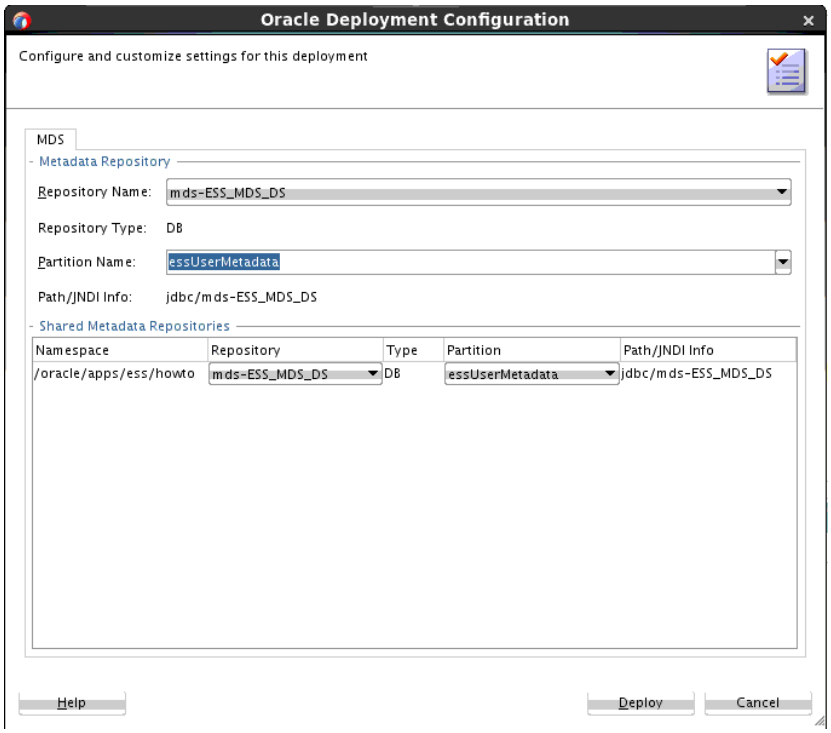
Deploying the Front-End Submitter Application

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` name space. Change its partition to the partition used when deploying `EssDemoApp`. If you used the default value, this should be EssDemoApp_V2.0.

Figure 7-23 Oracle Deployment Configuration Window



4. Click **OK**.

Update the EssHost MANIFEST File

Replace the contents of the EssHost `META-INF/MANIFEST.INF` file with the following lines:

```
Manifest-Version:1.0 Weblogic-Application-Version: 3.0
Extension-List: essruntime
essruntime-Extension-Name: oracle.ess.runtime
essruntime-Specification-Version: 12
```

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.example.com:7101/EssDemoAppUI/essdemoappservlet`

2. Log in as `EssDemoAppUser` with the password `welcome1`.

3. Follow the same steps as in the combined application.

8

Using the Metadata Service

This chapter describes how to use the Oracle Enterprise Scheduler Metadata Service to create, update and manage schedules, job definitions, and other Oracle Enterprise Scheduler metadata to a metadata store. You can also use the Metadata Service query methods to list objects stored in the metadata repository.

This chapter includes the following sections:

- Introduction to Using the Metadata Service
- Accessing the Metadata Service
- Accessing the Metadata Service with Oracle JDeveloper
- Querying Metadata Using the Metadata Service

For information about how to create job definitions, see the following chapter: Creating and Using PL/SQL Jobs , and Creating and Using Process Jobs .

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 create, update and manage 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 metadata 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 handle.

In an Oracle Enterprise Scheduler application you do not need to access or manipulate the MetadataServiceHandle. The application must 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 are 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 name space. Within a given package, two metadata objects with the same name and of the same type cannot be created.

Introduction to Metadata Service Name Spaces

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 name spaces 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 name space `/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.

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.

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 are not part of your transaction.

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.

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 8-1 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 8-1 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();
```

Accessing the Metadata Service with Oracle JDeveloper

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

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).

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 8-1 lists the comparison operators (`comparator` argument).

Table 8-1 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 8-2 shows code that creates a new filter.

Table 8-2 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.REQUEST_CATEGORY`	The name of the request category.
`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.
`MetadataService.QueryField.LOGICAL_CLUSTER_NAME`	The logical cluster associated with the job.

Example 8-2 Creating a Filter with a Filter Comparator for a Query

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

How to Query Metadata Objects

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

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

Example 8-3 shows a sample routine that queries for a list of job types in the metadata.

Example 8-3, 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.

Example 8-3 Using Metadata Service Query Methods

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

9

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 with 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:

- Introduction to Using Parameters and System Properties
- Using Parameters with the Metadata Service
- Using Parameters with the Runtime Service
- Using System Properties

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 Using System Properties.

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).

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 9-1:

- *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 9-1, 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.

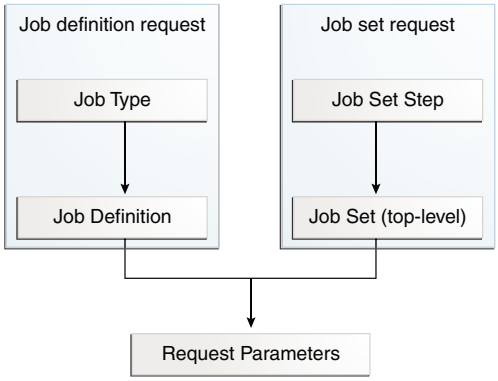
Table 9-1 Parameter Precedence Levels

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

What You Need to Know About Job Definition Parameter Materialization

Figure 9-1 illustrates the order of precedence taken by parameters defined in various components.

Figure 9-1 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.

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 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

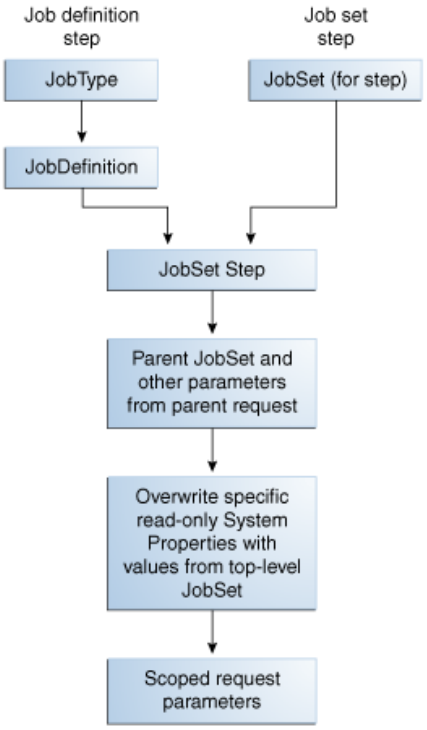
There is an exception for `RETRIES` because a value of 0 may mean that the job is not capable of being restarted. So if a step is defined with `RETRIES = 0`, it is not overridden, but if the step has `RETRIES > 0`, it is overridden with the job set value.

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.

Example 9-2 illustrates the parameter precedence for job set steps.

Figure 9-2 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 9-1 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.

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 9-2.

Table 9-2 ParameterInfo Parameter Properties

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

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

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

How to Use Parameters and System Properties in Metadata Objects

Example 9-1 shows code that uses a `ParameterList` to set parameter and system property values in a metadata object.

Example 9-1, 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.

Example 9-1 Adding Parameters and System Properties in a Metadata Object

```
String name = "JobDescription_name";
MetadataObjectId jobtype;
.
```

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

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 validates 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 allows a previously undefined request parameter to be set by a job during execution.

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 9-2 shows code using a `RequestParameters` object with the `add()` method to set a system property value.

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

Example 9-2 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);
.
.
.
```

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 9-3 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.

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

Example 9-3 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);
.
.
.
```

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 9-3 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 9-3.

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 9-3 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`

Name	Description
BIZ_ERROR_EXIT_CODE	Specifies the process exit code for a Process job request that denotes an execution business error. If this property is not specified, the system treats a process exit code of 4 as an execution business error. This property is optional for a Process job type. It is not used for other job types. Type: STRING
CLASS_NAME	Specifies the Java executable for a Java job request. This should be the name of a Java class that implements the oracle.as.scheduler.Executable interface. This property is required for a Java job type. It is not used for other job types. Type: STRING
CMDLINE	Specifies the command line used to invoke an external program for a Process job request. This property is required for a Process job type. It is not used for other job types. Type: STRING
CMDLINE_UNIX	Specifies the full command line for executing a Process type request executable on a Unix or Unix-like operating system. Typically, this property is specified in the job type and the executable name, path, and arguments are used to indicate values to be substituted at runtime. See the following properties: EXECUTABLE_NAME, EXECUTABLE_DIR_UNIX, EXECUTABLE_SUFFIX_UNIX, PROCESS_ARGUMENTS Type: STRING
CMDLINE_WINDOWS	Specifies the full command line for executing a Process type request executable on a Windows operating system. Typically, this property is specified in the job type and the executable name, path, and arguments are used to indicate values to be substituted at runtime. See properties: EXECUTABLE_NAME, EXECUTABLE_DIR_WINDOWS, EXECUTABLE_SUFFIX_WINDOWS, PROCESS_ARGUMENTS Type: STRING
EFFECTIVE_APPLICATION	Specifies the logical name of the Java EE application that is the effective application used to process the request. A job definition, job type, or a job set step can be associated with a different application by defining the EFFECTIVE_APPLICATION system property. This property can only be specified using metadata and cannot be specified as a submission parameter. Type: STRING
EJB_OPERATION_NAME	Specifies the operation name of the EJB. This can be used by the Bean implementation to branch to appropriate business methods. This property is used for the EJB job type. Type: STRING
ENVIRONMENT_VARIABLES	Specifies the environment variables to be set for the spawned process of a Process job request.The property value should be a comma separated list of name value pairs (name=value) representing the environment variables to be set. This property is optional for a Process job type. It is not used for other job types. Type: STRING
ESS_ASYNC_REQUEST_JNDI_MAPPED_NAME	Specifies the mapped name of the AsyncRequest EJB of Oracle Enterprise Scheduler bound to the JNDI of an Oracle Enterprise Scheduler server. Type: STRING

Name	Description
ESS_JNDI_CSF_KEY_NAME	Specifies the name that denotes the CSF KEY name of a JNDI provider of the underlying Oracle Enterprise Scheduler server. This property can be set in EssConfig of a hosting application. Type: STRING
ESS_RUNTIME_JNDI_MAPPED_NAME	Specifies the mapped name of the RuntimeService EJB of the Oracle Enterprise Scheduler bound to the JNDI of an Oracle Enterprise Scheduler server. This property is used for the EJB job type. Type: STRING
ESS_METADATA_JNDI_MAPPED_NAME	Specifies the mapped name of the MetadataService EJB of the Oracle Enterprise Scheduler bound to a JNDI of Oracle Enterprise Scheduler server. Type: STRING
EXECUTABLE_NAME	Specifies the name of the executable for a Process type request. The value should not include the path to the executable. See properties: EXECUTABLE_DIR_UNIX, EXECUTABLE_DIR_WINDOWS Type: STRING
EXECUTABLE_DIR_UNIX	Specifies the directory where the executable resides for a Process type request on a Unix or Unix-like operating system. Type: STRING
EXECUTABLE_DIR_WINDOWS	Specifies the directory where the executable resides for a Process type request on a Windows operating system. Type: STRING
EXECUTABLE_SUFFIX_UNIX	Specifies the file extension of the executable for a Process type request if executed on a generic Unix or Unix-like operating system. The default is no extension. Type: STRING
EXECUTABLE_SUFFIX_WINDOWS	Specifies the file extension of the executable for a Process type request if executed on a Windows operating system. The default is no extension. Type: STRING
EXECUTE_AUTO_EXPORT	Specifies whether the request's previously imported output content is automatically exported to the request's output directory before the job's execute stage runs. This property is applicable to the execute stage for Process, synchronous Java, and asynchronous Java job types. It does not apply to the update stage of asynchronous Java job types or PL/SQL job types. Valid values are: • true: All previously imported output content are exported to files in the request's output directory before the job's execute stage. • false: No output content is automatically exported. The job may choose to manually export output content. If this property is not specified, the system default false is used. Type: BOOLEAN

Name	Description
EXECUTE_PAST	Specifies whether instances of a repeating request with an execution time in the past should be generated. Instances are never generated before the requested start time nor after the requested end time. To cause past instances to be generated, you must set this property to TRUE and specify the requested start time as the initial time from which instances should be generated. Note that a null requested start time defaults to the current time. Valid values for this property are: • TRUE: All instances specified by a schedule are generated regardless of the time of generation. • FALSE: Instances with a scheduled execution time in the past (that is, before the time of generation) are not generated. If this property is not specified, the system defaults to TRUE. Type: BOOLEAN
EXTERNAL_ID	Specifies an identifier for an external portion of an asynchronous Java job. For example, an asynchronous Java job usually invokes some remote process and then returns control to Oracle Enterprise Scheduler. This property can be used to identify the remote process. This property should be set by the job implementation of asynchronous Java jobs when the identifier is known. It is never set by Oracle Enterprise Scheduler. Type: STRING
EXTERNAL_JOB_TYPE	Specifies an indicator of the type of the remote component of the job. For requests that have a remote component such as asynchronous Java jobs, WebService jobs, or EJB jobs this property specifies the nature of the remote job. Currently supported external job types are the names of the elements in the SystemProperty.ExternalJobType property. The supported values are SOA, OSB, ADFBC This property is optional. If it is not specified, Oracle Enterprise Scheduler does not associate the request with an external job type, regardless of how the job is implemented. Type: STRING
GROUP_NAME	Specifies the name of the Oracle Enterprise Scheduler isolation group to which this request is bound. This property is automatically set by Oracle Enterprise Scheduler during request submission. Type: STRING
INPUT_LIST	Specifies input to a request. The input to a serial job set is forwarded as input to the first step only. The input to a parallel job set is forwarded as input to all the parallel steps. Oracle Enterprise Scheduler imposes no format on the value of this property. Type: STRING
INVOKE_MESSAGE	Specifies the XML message payload used as the input for invoking the remote web service. This property is used for the EJB job type and WebService job type. This property is a pass-through parameter for the EJB job type. Type: STRING
JNDI_CSF_KEY	Specifies the CSF alias that is mapped to the user name and password in keystore. This specific user name/password is the credential needed to access the secured JNDI for JndiMappedName lookup. This property is needed only if the JNDI tree is secured. This property is used for the EJB job type. Type: STRING

Name	Description
JNDI_MAPPED_NAME	Specifies the mapped name of an EJB that is bound to the JNDI of a local/remote server. This property is used for the EJB job type. Type: STRING
JNDI_PROVIDER_URL	Specifies the URL of the JNDI provider pertaining to a remote server. This property is optional, needed only if the EJB and Oracle Enterprise Scheduler are remotely located. If this property is not specified, the job is executed in a local server. This property is used for the EJB job type. Type: STRING
LISTENER	Specifies the event listener class associated with the request. This should be the name of a Java class that implements the oracle.as.scheduler.EventListener interface. Type: STRING
LOCALE	Specifies the locale associated with the request. Type: STRING
LOGICAL_CLUSTER_NAME	Specifies the name of a logical cluster. A logical cluster consists of information related to a physical cluster and is usually stored in the hosting application's configuration. The logical cluster name is a reference to a set of physical cluster information in the application's configuration. If the property is not specified, no logical cluster is associated with the request. Type: STRING
OUTPUT_LIST	Specifies output from a request. The output of a serial job set is the OUTPUT_LIST of the last step. The output of a parallel job set is the concatenation of the OUTPUT_LIST of all the steps, in no guaranteed order, with oracle.as.scheduler.SystemProperty.OUTPUT_LIST_DELIMITER as a separator. Type: STRING
POST_PROCESS	Specifies the post-process callout handler class. This should be the name of a Java class that implements the oracle.as.scheduler.PostProcessHandler interface. Type: STRING
PRE_PROCESS	Specifies the pre-process callout handler class. This should be the name of a Java class that implements the oracle.as.scheduler.PreProcessHandler interface. Type: STRING
PRIORITY	Specifies the request processing priority. The priority interval is [0..9] with 0 as the lowest priority and 9 as the highest. Default: If this property is not specified, the system default value used is 4. Type: INTEGER
PROCEDURE_NAME	Specifies the name of the PL/SQL stored procedure to be called for a SQL job request. The stored procedure should be specified using schema.name format. The property is required for a SQL job type. It is not used for other job types. Type: STRING
PRODUCT	Specifies the product within the application that submitted the request. Type: STRING

Name	Description
PROCESS_ARGUMENTS	Specifies the arguments passed to the executable of a Process type spawned process. Type: STRING
REDIRECTED_OUTPUT_FILE	Specifies the file where standard output and error streams are redirected for a Process job request. This represents the full path of the log file where the standard output and error streams are redirected for the spawned process when the request is executed. This property is optional for a Process job type. It is not used for other job types. Type: STRING
REPROCESS_DELAY	Specifies the callout handler processing delay time. This represents the time, in minutes, to delay request processing when a delay is requested by a callback handler. Default: If this property is not specified, the system default used is 5. Type: INTEGER
REQUEST_CATEGORY	Specifies an application-specific label for a request. The label, defined by an application or system administrator, allows administrators to group job requests according to their own specific requirements. Type: STRING
REQUEST_EFFECTIVE_ENCODING	Specifies the effective encoding associated with a Process job request. SpawnLauncher determines the Locale setting for a spawned job request in the following precedence order: **1.** LC_ALL/LANG specified in environment properties (SystemProperty.ENVIRONMENT_VARIABLES) for the request **2.** LC_ALL/LANG specified in the hosting application ess-config.xml file **3.** Weblogic server LC_ALL/LANG The effective encoding is computed before the process is spawned and is stored in this property. This is later used to determine the encoding to use for the request log and output. Type: STRING
REQUEST_EXPIRATION	Specifies the expiration time for a request. This represents the time, in minutes, that a request expires after its scheduled execution time. A expiration value of zero (0) means that the request never expires. If this property is not specified, the system default value used is 0. Request expiration only applies to requests that are waiting to run. If a request waits longer than the specified expiration period, it does not run. After a request starts running the request expiration no longer applies. Type: INTEGER
REQUEST_LOG_LEVEL	Specifies the log level for request logging. Valid values for log level are the String representations of levels defined in java.util.logging. The level is obtained using Level.getName(). The default log level is "INFO". Type: STRING

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

Name	Description
SUCCESS_EXIT_CODE	Specifies the process exit code for a Process job request that denotes an execution success. If this property is not specified the system treats a process exit code of 0 as execution success. This property is optional for a Process job type. It is not used for other job types. Type: STRING
SUPPORT_OUTPUT_FILES	Specifies whether the request creates temporary or output files. The property applies during these stages: pre-processing, execution, async update, and post-processing. The request can always use the API to create output content directly in the content store. The property value specifies the action to take. If this property is not specified, no directories are created. Non-valid values are treated as though the property is not specified. Valid values are: • SystemProperty.SUPPORT_OUTPUT_FILES_NONE • SystemProperty.SUPPORT_OUTPUT_FILES_WORK • SystemProperty.SUPPORT_OUTPUT_FILES_OUTPUT Type: STRING
UPLOAD_CONTENT_TO_REPOSITORY	Specifies whether to upload request log and output files to a separate repository, such as Universal Content Management (UCM), from the internal repository when the request execution completes. Property value specifies the action to take. If this property is not specified, content is not uploaded. Non-valid values are treated as though the property were not specified. Valid value is: SystemProperty.UPLOAD_CONTENT_TO_REPOSITORY_COPY Type: STRING
USE_ALTERNATE_ENV	Specifies whether to use an alternative environment from a callout rather than the normal application environment. If this property is not specified, the normal application environment is used. Type: BOOLEAN
USE_EXTENDED_SETUP	Specifies whether to initiate capabilities like ApplSessions prior to invoking job-related code such as the job executable or pre-process handler, post-process handler. Extended functionality is invoked only in an environments where it is available. If this property is not specified, no extended functionality is set up prior to job execution. Type: BOOLEAN
USER_FILE_DIR	Specifies a base directory in the file system where files, such as input and output files, may be stored for use by the request executable. Oracle Enterprise Scheduler supports a configuration parameter that specifies a file directory where requests may store files. At request submission, a USER_FILE_DIR property is automatically added for the request if the configuration parameter is currently set and USER_FILE_DIR property was not specified for the request. If the property is added, it is initialized to the value of the configuration parameter. The property is not added if the configuration parameter is not set at the time of request submission. Type: STRING

Name	Description
USER_FILE_DIR_SHARED	Specifies whether the request's USER_FILE_DIR (configured RequestFileDir) directory is shared. This property represents the value of RequestFileDirShared. This property is valid for a request in standard or extended request mode. Valid values are: • true: USER_FILE_DIR is a shared directory. • false: USER_FILE_DIR is a local directory. If this property is not specified, system default false is used. Type: BOOLEAN
USER_NAME	Specifies the name of the user used to execute the request. Normally this is the submitting user unless the RUNAS_APPLICATIONID property was set in the job metadata. This property is automatically set by Oracle Enterprise Scheduler during request submission. Type: STRING
WARNING_EXIT_CODE	Specifies the process exit code for a Process job request that denotes an execution warning. If this property is not specified, the system treats a process exit code of 3 as execution warning. This property is optional for a Process job type. It is not used for other job types. Type: STRING
WORK_DIR_ROOT	Specifies the working directory for the spawned process of a Process job request. This property is optional for a Process job type. It is not used for other job types. Type: STRING
WS_WSDL_URL	Specifies the relative URL for web service WSDL. The base URL is given by the WS_WSDL_BASE_URL system property. This property is used for a WebService job type. Type: STRING
WS_WSDL_BASE_URL	Specifies a base URL that can be used in conjunction with the WS_WSDL_URL system property to provide a full URL for the web service WSDL. The property is usually used in conjunction with the LOGICAL_CLUSTER_NAME system property. It is meant to be a generic base URL that is common for all web service WSDLs in the cluster. This property is used for a WebService job type. This property is optional. If it is not specified, equivalent information may be retrieved from the information associated with the LOGICAL_CLUSTER_NAME system property of the request if it is configured in the hosting application's configuration. Type: STRING
WS_TARGET_NS	Specifies the target name space for the web service. This property is used for a WebService job type. Type: STRING
WS_ENDPOINT_URL	Specifies the relative URL for a web service endpoint. The base URL is given by the WS_ENDPOINT_BASE_URL system property. This property is used for a WebService job type. Type: STRING

Name	Description
WS_ENDPOINT_BASE_URL	Specifies a base URL that can be used in conjunction with the WS_ENDPOINT_URL system property to provide a full URL for the web service endpoint. This property is usually used in conjunction with the LOGICAL_CLUSTER_NAME system property. It is meant to be a generic base URL that is common for all web service endpoints in the cluster. This property is used for a WebService job type. This property is optional. If it is not specified, equivalent information may be retrieved from the information associated with the LOGICAL_CLUSTER_NAME system property of the request if it is configured in the hosting application's configuration. Type: STRING
WS_SERVICE_NAME	Specifies the WSDL service name for a web service operation. This property is used for a WebService job type. Type: STRING
WS_PORT_NAME	Specifies the WSDL port name for a web service operation. This property is used for a WebService job type. Type: STRING
WS_OPERATION_NAME	Specifies the WSDL operation name for a web service operation. This property is used for a WebService job type. Type: STRING
WS_CANCEL_OPERATION_NAME	Specifies the WSDL operation name for a web service cancel operation. This property is used for a WebService job type. Type: STRING
WS_CANCEL_MESSAGE	Specifies the XML message payload used as the input for invoking cancel on remote web service. This property is used for a WebService job type. Type: STRING

10

Using Tokens and Logical Clusters

In order to make job definitions easily portable from test environments to production environments, it is best for job definitions not to contain environment-specific information such as host names and port numbers. The Oracle Enterprise Scheduler token substitution and logical cluster features allow you to abstract metadata so that it can be easily changed to correctly fit the target deployment during the T2P process.

This chapter contains the following sections:

- Using Token Substitution
- Using Logical Clusters

Using Token Substitution

To improve the flexibility of configuration and reduce the need for provisioning, Oracle Enterprise Scheduler allows you to include substitutable tokens in request parameters and environment properties.

Tokens take the following form: `${`*TokenPrefix*`:`*token*`}`

Where *token* is the name of the token and *TokenPrefix* is where the substitution value is specified. Supported token prefixes are: `APP_ENV`, `ESS_ENV`, and `ESS_REQ`. The examples given below are simple, but illustrate the capabilities of Oracle Enterprise Scheduler substitution.

- **APP_ENV**

  The substitution value comes from the application environment properties.

  For example, the following environment variable is specified in the hosting application configuration properties using Oracle Enterprise Manager Fusion Middleware Control or WLST:

  ```
  AppEnvVar1=foo
  ```

  A request parameter is specified in a job definition:

  ```
  MyParam=${APP_ENV:appEnvVar1}
  ```

  After the substitution, the value of `MyParam` is `foo`.

- **ESS_ENV**

  Substitution values come from the Oracle Enterprise Scheduler server environment. This includes the following three possible sources:

  – **JVM system properties**: The token value is the name of the system property.

  – **JVM environment variables**: The token value is the name of the environment variable.

  - **JRF**: Token values are `jrfServerLogPath`, `jrfServerConfigDirectory`, and `jrfDomainConfigDirectory`.

  The following example shows the use of the `ESS_ENV` token prefix in the specification of an environment variable in the hosting application configuration properties.

  `wlstLocation = ${ESS_ENV:common.components.home}/common/bin/wlst.sh`.

  When the substitution is performed, the Oracle Enterprise Scheduler process job environment includes an environment variable named `wlstLocation` whose value is a complete path. For example:

  `/myInstallHome/mw_home/oracle_common/common/bin/wlst.sh`.

- **ESS_REQ**

  `ESS_REQ` substitutions come from information specific to the Oracle Enterprise Scheduler request in question. The following are the supported tokens:

  - `REQUEST_ID`
  - `REQUEST_HANDLE`
  - `IS_RESUMED`
  - `PAUSED_STATE`
  - Request parameter names for those requests

Nested Substitutions

Nested substitutions are automatically resolved when the top-level substitution is done. For example, the following request parameters are specified in a job definition for a process job:

```
MyParam1=${ESS_REQ:MyParam2}
MyParam2=${APP_ENV:MyEnvProp1}
```

An application's environment properties includes the following:

```
MyEnvProp1=${ESS_ENV:weblogic.Name}
```

The value of the JVM system property `weblogic.Name` might be something like `ess_server1`. When substitution is performed on `MyParam1`, it resolves to the value of `MyParam2`, which in turn resolves to the value of `MyEnvProp1`. The the result of the nested substitution is that the value of `MyParam1` is `ess_server1`.

Automatic Substitution

Automatic Oracle Enterprise Scheduler substitution is available for process job command lines and environment properties, as well as for some request properties used by EJB and web service jobs.

Table 10-1, Table 10-2, and Table 10-3 list the automatically substituted request parameters for EJB jobs, web service jobs, and process jobs.

Table 10-1 EJB Job Type Automatically Substituted Properties

Property Name
SYS_EXT_jndiProviderUrl
SYS_EXT_essRuntimeJndiMappedName
SYS_EXT_essMetadataJndiMappedName
SYS_EXT_essAsyncRequestJndiMappedName
SYS_EXT_essJndiCsfKey
SYS_EXT_invokeMessage

Table 10-2 Web Services Job Type: Automatically Substituted Properties

Property Name
SYS_EXT_wsEndpointBaseUrl
SYS_EXT_wsEndpointUrl
SYS_EXT_wsWsdlBaseUrl
SYS_EXT_wsWsdlUrl
SYS_EXT_invokeMessage
SYS_EXT_wsCancelMessage

Table 10-3 Process Job Type: Automatically Substituted Properties

Property Name
SYS_cmdLine
SYS_EXT_cmdLine.Unix
SYS_EXT_cmdLine.Windows
SYS_environmentVariables

For more information about these properties refer to the following chapters:

- Creating and Using EJB Jobs
- Creating and Using Web Service Jobs
- Creating and Using Process Jobs .

Using Logical Clusters

Oracle Enterprise Scheduler provides the means by which EJB and web service jobs can define an abstract job location. The job location is specified by the Oracle Enterprise Scheduler `SYS_logicalClusterName` system property and specifies a logical cluster name (LCN). If the job definition for an EJB or web service job specifies a value for an LCN, certain environment-specific properties are specified using Oracle Enterprise Manager Fusion Middleware Control or WLST commands at the hosting application level rather than in the job definition. All job definitions with the same

LCN share the value of the properties entered in the hosting application configuration properties using Oracle Enterprise Manager Fusion Middleware Control or WLST commands.

Note:

Oracle Enterprise Manager Fusion Middleware Control refers to logical cluster functionality as "job location." The terms "logical clusters" and "job location" can be used interchangeably.

If a job definition specifies a value for the `SYS_logicalClusterName` property, then the value is used as a prefix for a set of application configuration properties that define attributes of the logical cluster. Table 10-4 lists the properties associated with job location, where *prefix* represents the logical cluster name. Note that these properties need not be specified if they are configured in the hosting application.

Table 10-4 Properties Associated With a Job Location

Property Name	Corresponding System Property	Description
`LCN.`*prefix*`.JndiProviderUrl`	`SYS_EXT_jndiProviderUrl`	The JNDI provider for the cluster. Used with the EJB job type. The corresponding system property, `SYS_EXT_jndiProviderUrl`, need not be specified in the job.
`LCN.`*prefix*`.WsEndpointBaseUrl`	`SYS_EXT_wsEndpointBaseUrl`	The host and port. For example, `http://`*host*`:`*port*`/`. Used with the EJB job type. The corresponding system property, `SYS_EXT_wsEndpointBaseUrl`, need not be specified in the job.
`LCN.`*prefix*`.WsWsdlBaseUrl`	`SYS_EXT_wsWsdlBaseUrl`	The host and port of the WSDL. For example, `http://`*host*`:`*port*`/`. Used with the EJB job type. The Corresponding system property, `SYS_EXT_wsWsdlBaseUrl`, need not be specified in the job.

For example, if a job defines the `SYS_logicalClusterName` property as `SOA_Cluster1`, then the application configuration might contain the following properties:

```
LCN.SOA_Cluster1.WsEndpointBaseURL=http://host:port/
LCN.SOA_Cluster1.WsWsdlBaseURL=http://host:port/
LCN.SOA_Cluster1.JndiProviderURL=t3://host1;port1;host2:port2/
```

Note:

The value of the `SYS_logicalClusterName` property cannot contain the “." character.

11

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:

- Introduction to Using PL/SQL Stored Procedure Job Definitions
- Creating a PL/SQL Stored Procedure for Oracle Enterprise Scheduler
- Performing Oracle Database Tasks for PL/SQL Stored Procedures
- Creating and Storing Job Definitions for PL/SQL Job Types

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

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 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 Database Scheduler (an Oracle Database feature). When a PL/SQL stored procedure job request is ready to

execute, Oracle Enterprise Scheduler creates an immediate, run-once Oracle Database Scheduler job. This Oracle Database Scheduler job is created by the Oracle Enterprise Scheduler runtime schema user associated with the container instance that executes the PL/SQL request, and is owned by the application procedure owner. The Oracle Database Scheduler job procedure is a PL/SQL wrapper procedure owned by the Oracle Enterprise Scheduler runtime schema user. Finally, when the Oracle Database Scheduler job runs, the wrapper procedure calls the application stored procedure using dynamic SQL. After the PL/SQL stored procedure completes, either by a successful return or by raising an exception, the Oracle Database Scheduler job finishes and creates an event that informs Oracle Enterprise Scheduler that the remote executable finished.

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

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 11-1 shows a sample `HELLO_WORLD` stored procedure for use with Oracle Enterprise Scheduler.

Example 11-1 HELLO_WORLD PL/SQL Stored Procedure

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

    -- Retrieve value of an existing request property.
```

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

    -- Update an existing request property with a new value.
    -- This procedure is responsible for commit/rollback of the update operation.
    begin
        v_prop_name := 'myJobdefProp';
        ess_runtime.update_reqprop_varchar2(v_request_id, v_prop_name,
                                            'myUpdateValue');
        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;
/
```

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

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 must raise an exception. `RAISE_APPLICATION_ERROR` requires that the error code range from -20000 to -20999.

Table 11-1 indicates the Oracle Enterprise Scheduler state based on the result of the PL/SQL stored procedure.

Table 11-1 Terminal States for PL/SQL Stored Procedure Results

Final State	Description
`SUCCEEDED`	If the PL/SQL stored procedure returns normally, without raising an exception, the request state transitions to the `SUCCEEDED` state, bearing any subsequent errors completing the request.
`WARNING`	If the PL/SQL stored procedure returns with an exception, the request state is based on the SQL error code of the exception. The request transitions to the `WARNING` terminal state if the SQL error code ranges from -20900 to -20919.

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

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.

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 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.

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 must load the PL/SQL stored procedure in the Oracle Database and grant the required permissions.

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 runtime schema that is associated with the deployed Java EE application that is going to submit the Oracle Enterprise Scheduler PL/SQL stored procedure request.

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

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

user_schema: specifies the name of the application user schema.

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

To grant Oracle Database permissions:

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

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

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

   ```
   CREATE OR REPLACE SYNOMYM user_schema.ESS_RUNTIME for ess_schema.ESS_RUNTIME;
   ```

3. In the Oracle Database, grant execute on the ESS_JOB package to the application user schema. This step can be skipped if ESS_JOB is not used. For example:

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

4. In the Oracle Database, create a private synonym for the ESS_JOB package. This is a convenience step that allows the PL/SQL stored procedure to reference the ESS_JOB as simply ESS_JOB rather than using the full *schema_name* .ESS_JOB. This step can be skipped if ESS_JOB is not used. For example:

   ```
   CREATE OR REPLACE SYNONYM user_schema.ESS_JOB for ess_schema.ESS_JOB;
   ```

5. In the Oracle Database, grant execute on a PL/SQL stored procedure owned by the Oracle Enterprise Scheduler runtime schema user that serves as the Oracle Enterprise Scheduler job procedure. For example:

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

As an example, if the Oracle Enterprise Scheduler runtime schema is TEST_ESS, the application user schema is HOWTO, and the PL/SQL procedure is named HELLO_WORLD, the DBA operations are:

```
GRANT EXECUTE ON test_ess.ess_runtime to howto;
CREATE OR REPLACE SYNONYM howto.ess_runtime for test_ess.ess_runtime;
GRANT EXECUTE ON test_ess.ess_job to howto;
CREATE OR REPLACE SYNONYM howto.ess_job for test_ess.ess_job;
GRANT EXECUTE ON test_ess.ESS_SCHJOB_PROC to howto;
```

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

The 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. The two steps shown for DBA tasks use to grant permissions on the ESS_JOB package are only required if the ESS_JOB package is referenced by a PL/SQL procedure. The step shown for the ESS_SCHJOB_PROC procedure is always required since it allows the Oracle Enterprise Scheduler wrapper procedure to be called.

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`.

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 Using the Metadata Service .

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

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 *owner*`.`*name* format, where *owner* is the schema owner of the job procedure and *name* is the procedure name. This property must be specified for either the job type or job definition.

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

If the `SQL_JOB_CLASS` system property is not specified, a default Oracle Database Scheduler job class created by Oracle Enterprise Scheduler is used for the Oracle Database Scheduler job. The default job class is associated with the default resource consumer group. It belongs to the default service, which means 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 grant is needed for an Oracle Enterprise Scheduler SQL request to use that default job class.

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 11-2 describes the `SystemProperties` that are appropriate for a PL/SQL stored procedure job type.

Table 11-2 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 Database Scheduler job class to be assigned to the Oracle Database 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 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 Using a PL/SQL Stored Procedure with an Oracle Enterprise Scheduler Application for sample code.

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:

After 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.

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.

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

This section shows sample code in which job type and job definition application metadata are created for a SQL job type.

```
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;
      }
   }
}
```

12

Creating and Using EJB Jobs

This chapter describes how to use Oracle Enterprise Scheduler to create Enterprise Java Bean (EJB) jobs.

This chapter contains the following sections:

- Introduction to Creating EJB Jobs
- Planning Job Development
- Creating and Storing Job Definitions for EJB Job Types
- Secured Invocation
- Synchronous Bean
- Asynchronous Bean

Introduction to Creating EJB Jobs

The EJB job type allows you to create Java-based jobs and take advantage of the convenience of the pre-deployed hosting application. In addition, the EJB can be located remotely on a different server. Unlike Java SE-based jobs, EJB jobs are not required to be embedded inside the hosting application. This allows them to be located remotely and to be used with the pre-deployed hosting application. An EJB job can be invoked from any hosting application, including the pre-deployed hosting application. Note that the EJB implementation must be in the same WebLogic domain as the scheduler server.

The EJB conforms to an interface defined by Oracle Enterprise Scheduler (defined in a shared library). The EJB is non-transactional. Both synchronous and asynchronous EJB jobs are supported. When running asynchronously, the EJB returns quickly and the Oracle Enterprise Scheduler EJB is invoked later when the job completes. The EJB implementation can work with any of the three shared libraries—server (`oracle.ess`), client (`oracle.ess.client`), and thin client (`oracle.ess.thin.client`). The thin client shared library does not depend on Oracle Enterprise Scheduler data sources. See Creating a Thin Client Application for more information about using the thin client library.

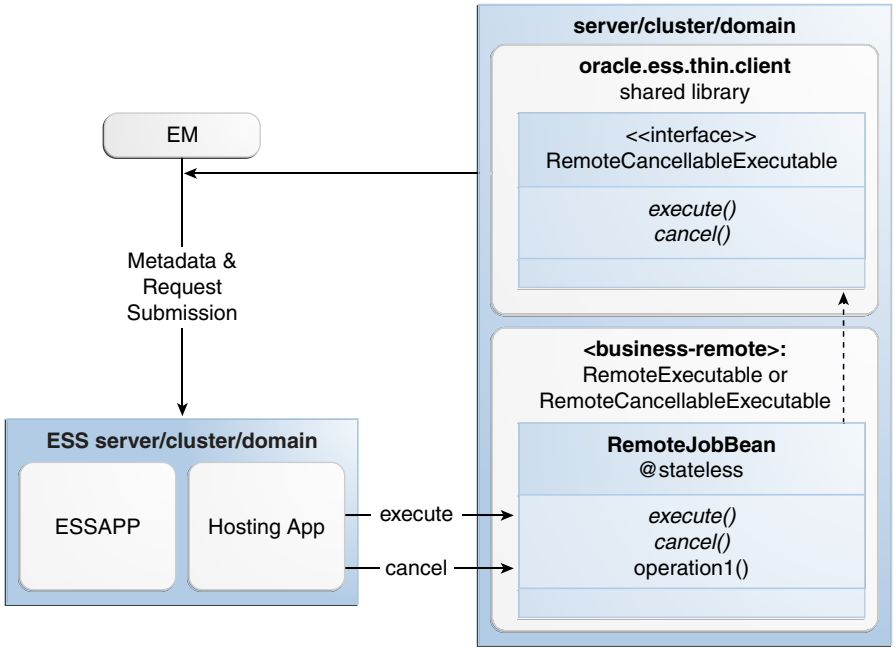
The EJB can submit sub-requests and it can write to output and a log. The EJB interface is similar or the same as a Java job and can do similar things. To improve performance, it is possible to consolidate multiple job implementations under a single shared EJB.

An EJB job is a Java job that is executed remotely using the EJB remote business interface. The execution type is `JAVA_TYPE`. The remote job is an EJB deployed in a remote server. The remote business interface of this bean extends the `oracle.as.scheduler.RemoteExecutable` interface and defines the `execute` method. The contract between Oracle Enterprise Scheduler and the servicing

component is defined with the `execute` method. Figure 12-1 shows the components in a typical EJB job deployment.

The EJB must be located in the same domain as the hosting application and the `Subject` object is propagated to the EJB. For JNDI lookup operations, you can supply optional credentials. The default identity is "anonymous".

Figure 12-1 EJB Job Environment



Planning Job Development

The Oracle Enterprise Scheduler is flexible and provides implementation and deployment options. Planning Job Development is a high-level discussion about how to plan your job development and deployment process.

Creating and Storing Job Definitions for EJB Job Types

To use EJB type jobs with Oracle Enterprise Scheduler, you must 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 must also set certain system properties. You can store the job definition in the metadata repository using the Metadata Service. Sample metadata files are provided later in this chapter.

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

When you specify the `JobType` for the job definition, you can also specify `SystemProperties` that define the characteristics associated with the `JobType`. Table 12-1 and Table 12-2 describe the properties that specify how the request should be processed.

Table 12-1 EJB Job Type Properties

Property Name [Field in SystemProperty class]	Description
`SYS_EXT_jndiProviderUrl [JNDI_PROVIDER_URL]`	Optional. Specifies the URL of the remote server. Required only if the EJB is remotely located. The `JndiProviderUrl` can be specified to contain tokens that are resolved at runtime. The following are two examples: • `${WIRING:urn:oracle:fmw:soa:t3}` This URN is resolved at runtime and the actual URL value is fetched. • `t3://localhost:19283` Where `localhost` is the host where EJB's are deployed and `19283` is the server port number.
`SYS_EXT_jndiMappedName [JNDI_MAPPED_NAME]`	Required. Specifies the JNDI lookup name of a remote EJB implementation. Example: `ejb/fileAdapter`
`SYS_EXT_ejbOperationName [EJB_OPERATION_NAME]`	Optional. Specifies a pass-through parameter used by the EJB implementation to branch to the appropriate business methods. Example: `manageFileAdapter`
`SYS_EXT_jndiCSFKey [JNDI_CSF_KEY]`	Required only if the JNDI tree of the EJB server is secured. Points to the CSF alias that is mapped to the user name and password in the keystore. This specific user name/password pair is the credential required to access the secured JNDI for `JndiMappedName` lookup. This property can be added to either the `RequestParameters` object or to the Oracle Enterprise Scheduler configuration of the hosting application.

Note:

You can use Oracle Enterprise Manager Fusion Middleware Control or WLST scripts to configure the CSF key aliases as a post installation step. Prior to the post installation step, the Keystore's CSF map can be set to the default value of `oracle.ess.security`.

Table 12-2 lists the properties that can be added either to the `RequestParameters` object or to the Oracle Enterprise Scheduler configuration of the hosting application. In a production environment, environment specific data should not be entered into the job definition because the job definition is replicated when going from the test environment to the production environment. Instead, this data should be entered separately as configuration data with the hosting application. The Oracle Enterprise Scheduler token substitution and logical cluster features allow you to abstract metadata so that it can be easily changed to correctly fit the target deployment during the T2P process. See Using Tokens and Logical Clusters for information about using these features.

Table 12-2 Additional Properties

Property Name [Field in SystemProperty class]	Description
`SYS_EXT_essJndiCsfKey [ESS_JNDI_CSF_KEY_NAME]`	Optional. Specifies the CSF key alias of the authenticated Oracle Enterprise Scheduler server. This property is required only if the Oracle Enterprise Scheduler JNDI tree is authenticated. Example: `ess-jndi-csf-key`
`SYS_EXT_essRuntimeJndiMappedName{ESS_RUNTIME_JNDI_MAPPED_NAME]`	Specifies the JNDI mapped name of Oracle Enterprise Scheduler's `RuntimeService` bean that is defined in the hosting application and bound to the Oracle Enterprise Scheduler server's JNDI tree. This property is required only if you use a hosting application other than `EssNativeHostingApp` and the remote bean has to call the Oracle Enterprise Scheduler runtime bean (for example, to write output or log information, submit requests or operate on requests).
`SYS_EXT_essMetadataJndiMappedName[ESS_METADATA_JNDI_MAPPED_NAME]`	Specifies the JNDI mapped name of Oracle Enterprise Scheduler's `MetadataService` bean defined in the hosting application and bound to the Oracle Enterprise Scheduler server's JNDI tree. This property is required only if you use a hosting application other than `EssNativeHostingApp` and the remote bean requires access to Oracle Enterprise Scheduler's metadata bean.
`SYS_EXT_essAsyncRequestJndiMappedName[ESS_ASYNC_REQUEST_JNDI_MAPPED_NAME]`	Specifies the JNDI mapped name of Oracle Enterprise Scheduler's `AsyncRequest` bean defined in the hosting application and bound to Oracle Enterprise Scheduler server's JNDI tree. This property is required only if: • The EJB invocation is asynchronous • You use a hosting application other than `EssNativeHostingApp` • The remote bean has to call back to Oracle Enterprise Scheduler beans (for example, an asynchronous callback).

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

Secured Invocation

Secured invocation of remote EJBs is required when the JNDI tree of its server is authenticated. This is also true when a remote EJB calls back to Oracle Enterprise Scheduler EJBs using secured lookup. The following sections provide some guidance.

Forward Invocation

The following apply to forward invocation.

- When Oracle Enterprise Scheduler invokes a remote EJB, the subject of the executing job is always propagated.
- When Oracle Enterprise Scheduler executes a job, the `JndiProviderUrl` of the current Oracle Enterprise Scheduler Server is always supplied to the remote EJB through `RequestParameters`.
- If the JNDI tree of the remote server is authenticated, the `JNDI_CSF_KEY` property must be specified in the request parameters or the `EssConfiguration` of the hosting application.
- Oracle Enterprise Scheduler looks up the keystore for the `CsfKey` to retrieve the `PasswordCredential` and connects to the remote server.

Callback Invocation

The following apply to callback invocation.

- If the remote EJB must call back to Oracle Enterprise Scheduler beans, the following properties can be specified:
  - The JNDI names of Oracle Enterprise Scheduler `Runtime`, `Metadata` and `AsyncRequest` beans exposed in `HostingApp` must be specified in request parameters or the `EssConfiguration` of the hosting application. If `EssNativeHostingApp` is used, these entries are not required.
  - If the JNDI tree of the Oracle Enterprise Scheduler server is authenticated, the `ESS_JNDI_CSF_KEY_NAME` property must be specified in the request parameters or `EssConfiguration` of the hosting application. Oracle Enterprise Scheduler ensures that this property is available to the remote EJB through `RequestParameters`.
- A remote EJB can make use of the `RemoteConnector` API to get the remote instances of Oracle Enterprise Scheduler beans. This can be done by passing the following:
  - `RequestParameters`
  - `RequestParameters` and `JndiMappedName` of the bean (for hosting applications other than the native hosting application)
  - `RequestParameters`, user name and password (if the Oracle Enterprise Scheduler server is authenticated)
  - `InitialContext` (primarily for Java SE clients with `EssNativeHostingApp`)
  - `InitialContext` and `jndiMappedName` (primarily for Java SE clients with other hosting applications)

RemoteConnector API and the Server Affinity Property

If your code implementation relies on accessing Oracle Enterprise Scheduler EJBs, use the methods exposed in the `RemoteConnector` API class. The Oracle Enterprise Scheduler requires the server affinity property to be set in the `InitialContext` environment before doing a JNDI lookup and the `RemoteConnector` API class sets this property for you. Note that this is especially important in multi-node cluster

scenarios. The `RemoteConnector` class is packaged in the Oracle Enterprise Scheduler client libraries.

If for some reason the `RemoteConector` class cannot be used, you can set the environment map property to the `InitialContext` before doing the look-up for the Oracle Enterprise Scheduler EJBs as shown in the following example.

```
if(PlatformUtils.isWebLogic())
   envProps.put("weblogic.jndi.enableServerAffinity", "true");
```

In a multi-node cluster environment, it is best to set the cluster algorithm to "round-robin-affinity".

CSF Lookup From a Remote Server

If the beans of Oracle Enterprise Scheduler Services are authenticated, remote applications must use a secured lookup to make callbacks to Oracle Enterprise Scheduler. You can use Oracle Enterprise Scheduler's `RemoteConnector` API which uses the `ESS_JNDI_CSF_KEY_NAME` property available in the request parameters to do the look-up. But to assist this CSF lookup, the code that invokes the `RemoteConnector` must grant permission for credential store access. The following XML fragment can be added to the `jazn-data.xml` file of the remote application.

```
 <jazn-policy>
    <grant>
      <grantee>
        <codesource>
          <url>file:${domain.home}/servers/${weblogic.Name}/
tmp/                    _WL_user/<AppName>/-</url>
        </codesource>
      </grantee>
      <permissions>
      <permission>
<class>oracle.security.jps.service.credstore.CredentialAccessPermission
          </class>
          <name>context=SYSTEM,mapName=oracle.ess.security,keyName=*</name>
          <actions>READ</actions>
        </permission>
        <permission>
          <class>oracle.security.jps.JpsPermission</class>
          <name>IdentityAssertion</name>
          <actions>execute</actions>
        </permission>
        <permission>
          <class>oracle.security.jps.JpsPermission</class>
          <name>AppSecurityContext.setApplicationID.*</name>
        </permission>
      </permissions>
    </grant>
  </jazn-policy>
```

Synchronous Bean

This section contains examples that illustrate how to create a synchronous bean.

Metadata

This section shows metadata as it applies to synchronous beans.

The following example shows a sample job definition for an EJB job located in the file `oracle/apps/ess/custom/Jobs/EssGatewayJobDefn.xml`

```
<?xml version = '1.0'?>
<job-definition xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"                  xmlns="http://xmlns.oracle.com/scheduler"
               name="SoaEjbJobDefn"
               job-type="/oracle/as/ess/core/JobType/SyncEjbJobType.xml">
<description/>
  <display-name>EssGatewayBean</display-name>
  <parameter-list>
    <parameter name="SYS_EXT_jndiKeyName" data-type="string" read-only="true">
      ejb/essGatewayBean</parameter>
    <parameter name="SYS_EXT_jndiProviderUrl" data-type="string"        read-
only="true">URL</parameter>
    <parameter name="SYS_EXT_ejbOperationName" data-type="string"
      read-only="true">activateFileAdapter</parameter>
    <parameter name="SYS_effectiveApplication" data-type="string">
      ESS_NATIVE_HOSTING_APP_LOGICAL_NAME</parameter>
  </parameter-list>
</job-definition>
```

EJB Job Sample Code

This section shows a sample implementation of a synchronous EJB job expected by Oracle Enterprise Scheduler.

The following code snippet shows a fragment of the `ejb-jar.xml` file that defines this bean.

```
<session>

<description>ESS Gateway Bean</description>
      <ejb-name>EssGateway</ejb-name>
      <ejb-class>com.soa.beans.EssGatewayBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <security-identity>
        <use-caller-identity/>
      </security-identity>
</session>
```

The following code snippet shows a fragment of the `weblogic-ejb-jar.xml` file that defines this bean.

```
<weblogic-enterprise-bean>
      <ejb-name>FileAdapterBean</ejb-name>
      <stateless-session-descriptor>
         <business-interface-jndi-name-map>
            <business-remote>oracle.as.scheduler.RemoteCancellableExecutable
         </business-remote>
            <jndi-name>ejb/essGatewayBean</jndi-name>
         </business-interface-jndi-name-map>
      </stateless-session-descriptor>
 </weblogic-enterprise-bean>

import javax.ejb.Stateless;
import oracle.as.scheduler.SystemProperty;
import oracle.as.scheduler.ExecutionCancelledException;
import oracle.as.scheduler.ExecutionErrorException;
```

```
import oracle.as.scheduler.ExecutionPausedException;
import oracle.as.scheduler.ExecutionWarningException;
import oracle.as.scheduler.RemoteCancellableExecutable;
import oracle.as.scheduler.RequestExecutionContext;
import oracle.as.scheduler.RequestParameters;

@Stateless(name = "EssGateway", mappedName = "ejb/essGatewayBean")
public class EssGatewayBean implements RemoteCancellableExecutable
{
    public EssGatewayBean()
    {
    }

    public void execute(RequestExecutionContext context,
                    RequestParameters parameters) throws ExecutionErrorException,
                    ExecutionWarningException, ExecutionCancelledException,
                    ExecutionPausedException
    {
        //Get the value of 'SYS_EXT_ejbOperationName' property
        String opName =
(String)parameters.getValue(SystemProperty.EJB_OPERATION_NAME);

        if("manageFileAdapter".equals(opName))
        {
            //
            //Call business method of this bean or some other bean
            //
            //Hint: User defined properties can be set in RequestParameters while
            //submitting the job and can be retrieved here for further
processing.
        }
    }

    public void cancel(RequestExecutionContext context,
                    RequestParameters parameters)
    {
        //
        //Logic to cancel the execution of a business method.
        //
        // Execute the actual logic of cancellation, notifies back to ESS
        // by throwing ExecutionCancelledException through execute method.
    }
}
```

Asynchronous Bean

Asynchronous EJBs are typically used:

- For long-running operations
- For processor-intensive tasks
- For background tasks
- To increase application throughput
- To improve application response time if the method invocation result is not required immediately

The synchronous EJB job is more appropriate for short-running user business methods.

There are a couple of ways for Oracle Enterprise Scheduler to execute a bean asynchronously:

- **Explicit asynchrony**: Use a synchronous stateless bean to invoke a message-driven bean asynchronously. (Add a Java Message Service message to a topic/queue that is listened to by a message-driven bean)
- **Implicit asynchrony**: Use the EJB Asynchronous annotation to declare a business method (other than execute, cancel methods) to behave asynchronously.

When Oracle Enterprise Scheduler invokes a bean asynchronously, it does not wait for the execute method to finish. For that reason, the bean implementation has to notify the Oracle Enterprise Scheduler after processing finishes. The `RemoteAsyncHelper` class can be used for this purpose. Alternatively, the `AsyncRequestBean` obtained from `RemoteConnector` can be used to notify Oracle Enterprise Scheduler with a status update.

Note:

Oracle Enterprise Scheduler can invoke a synchronous bean asynchronously. However, if you use this method the bean must be modeled in a way that long-running methods are marked for asynchrony.

Note:

As specified in the EJB standard, you cannot use the `@Asynchronous` annotation in the `execute` method or the entire class because the execute method throws custom exceptions which are not permitted. Oracle Enterprise Scheduler requires the `execute` method to throw custom exceptions.

Metadata

This section shows metadata as it applies to asynchronous beans.

This example shows a sample job definition for an EJB job located in the file `oracle/apps/ess/custom/Jobs/AsyncJobDefn.xml`

```
<?xml version = '1.0'?>
<job-definition xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://xmlns.oracle.com/scheduler" name="EssAsyncJobDefn"
    job-type="/oracle/as/ess/core/JobType/AsyncEjbJobType.xml">
  <display-name>EssGatewayBean</display-name>
  <parameter-list>
    <parameter name="SYS_EXT_jndiKeyName" data-type="string"
       read-only="true">ejb/essAsyncGatewayBean</parameter>
    <parameter name="SYS_EXT_jndiProviderUrl" data-type="string"
      read-only="true">t3://localhost:10801</parameter>
    <parameter name="SYS_EXT_ejbOperationName"
      data-type="string"read-only="true">activateFileAdapter</parameter>
    <parameter name="SYS_effectiveApplication" data-type="string">
      ESS_NATIVE_HOSTING_APP_LOGICAL_NAME</parameter>
  </parameter-list>
</job-definition>
```

EJB Job Sample Code

This section includes sample code that illustrates how to implement asynchrony using both the explicit and implicit methods described in Asynchronous Bean.

Sample Implementation of Asynchrony Using a Message-Driven Bean

The following code sample shows a synchronous stateless bean that is used to invoke a message driven bean asynchronously.

```
package com.soa.test;

import java.io.Serializable;
import java.util.ArrayList;
import javax.ejb.Stateless;
import javax.jms.ObjectMessage;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.Session;

import javax.naming.InitialContext;

import oracle.as.scheduler.AsyncRequestBeanRemote;
import oracle.as.scheduler.ExecutionCancelledException;
import oracle.as.scheduler.ExecutionErrorException;
import oracle.as.scheduler.ExecutionPausedException;
import oracle.as.scheduler.ExecutionWarningException;
import oracle.as.scheduler.RemoteCancellableExecutable;
import oracle.as.scheduler.RequestExecutionContext;
import oracle.as.scheduler.RequestParameters;
import oracle.as.scheduler.request.RemoteConnector;

@Stateless(name = "EssAsyncPilot")
public class EssAsyncPilotBean implements RemoteCancellableExecutable
{
    public EssAsyncPilotBean() {
    }

    public void execute(RequestExecutionContext requestExecutionContext,
            RequestParameters requestParameters)
            throws ExecutionErrorException, ExecutionWarningException,
            ExecutionPausedException, ExecutionCancelledException {
        // Delegate the job request cancellation to message driven bean
        postToQueue("execute", requestExecutionContext, requestParameters);
    }

    public void cancel(RequestExecutionContext requestExecutionContext,
            RequestParameters requestParameters) {
        RemoteConnector connector = new RemoteConnector();
        AsyncRequestBeanRemote asyncRequest;

        // Delegate the job request cancellation to message driven bean
        try {
            postToQueue("cancel", requestExecutionContext, requestParameters);
        } catch (Exception e) {
            //Mark this request as ERROR ed
        }
```

```
        /*
         * Other ways to cancel the job request.
         * asyncRequest = connector.getAsyncRequestEJB(requestParameters);
         * asyncRequest.onCancel(requestExecutionContext);
         *
         * (or)
         *
         * asyncRequest = connector.getAsyncRequestEJB(requestParameters);
         * asyncRequest.setRequestStatus(
         *      requestExecutionContext, AsyncStatus.CANCEL, "Cancelling the
job");
         *
         * (or simply)
         *
         * RemoteAsyncHelper asyncHelper = new RemoteAsyncHelper(
         *      requestExecutionContext, requestParameters);
         * asyncHelper.onCancel();
         *
         */
    }

    private void postToQueue(String instruction,
            RequestExecutionContext context, RequestParameters params) {
        try {
            QueueConnectionFactory qconFactory;
            QueueConnection qcon;
            QueueSession qsession;
            QueueSender qsender;
            Queue queue;
            ObjectMessage msg;
            InitialContext ic = new InitialContext();

            qconFactory = (QueueConnectionFactory) ic
                    .lookup("EssAsyncJmsConnFactory");
            qcon = qconFactory.createQueueConnection();
            qsession = qcon.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

            queue = (Queue) ic.lookup("EssAsyncJmsQueue");

            qsender = qsession.createSender(queue);
            msg = qsession.createObjectMessage();
            qcon.start();

            ArrayList<Serializable> objsList = new ArrayList<Serializable>(2);
            objsList.add(context);
            objsList.add(params);
            objsList.add(instruction);
            msg.setObject(objsList);
            qsender.send(msg);

            System.out.println("The message, " + instruction
                    + ", has been sent to the EssAsyncJmsQueue.");
            qsender.close();
            qsession.close();
            qcon.close();
        } catch (Exception e) {
            System.out.print("error " + e);
        }
    }
}
```

```
import java.util.List;
import java.io.Serializable;
import javax.ejb.MessageDriven;

import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;

import oracle.as.scheduler.RequestExecutionContext;
import oracle.as.scheduler.RequestParameters;
import oracle.as.scheduler.async.RemoteAsyncHelper;

/**
 * This message driven bean sample relies on execute/cancel instructions.
 * Upon completion of execution or cancellation, this bean notifies
 * ESS about its status so that the job request is marked for completion.
 */
@MessageDriven(mappedName = "ejb/essAsyncJms")
public class EssAsyncJmsBean implements MessageListener {
    public void onMessage(Message message) {
        if (message instanceof ObjectMessage) {
            ObjectMessage objMessage = (ObjectMessage)message;
            try {
                List<Serializable> objsList =
(List<Serializable>)objMessage.getObject();
                RequestExecutionContext ctx =
(RequestExecutionContext)objsList.get(0);
                RequestParameters params = (RequestParameters)objsList.get(1);
                String instruction = (String)objsList.get(2);
                RemoteAsyncHelper asyncHelper = new RemoteAsyncHelper(ctx,
params);
                if ("cancel".equalsIgnoreCase(instruction)) {
                    //EssAsyncJmsBean.cancel: Cancelling the Execution
                    try {
                        //Do the actual cancellation
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                    }
                    asyncHelper.onCancel();
                    //EssAsyncJmsBean.cancel: Completed cancellation
                } else {
                    //EssAsyncJmsBean.execute: Started the Execution ");
                    try {
                        //Do the actual execution
                        Thread.sleep(5000);
                    } catch (InterruptedException e) {
                    }
                    asyncHelper.onSuccess();
                    //EssAsyncJmsBean.execute: Completed the Execution ");
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

Sample Implementation of Asynchrony Using Annotations

The following code snippet uses the EJB `Asynchronous`to declare a bean or its methods to behave asynchronously.

```
package com.soa.test;

import java.util.concurrent.Future;

import javax.annotation.Resource;

import javax.ejb.AsyncResult;
import javax.ejb.Asynchronous;
import javax.ejb.SessionContext;
import javax.ejb.Stateless;

import javax.xml.transform.Result;

import oracle.as.scheduler.ExecutionCancelledException;
import oracle.as.scheduler.ExecutionErrorException;
import oracle.as.scheduler.ExecutionPausedException;
import oracle.as.scheduler.ExecutionWarningException;
import oracle.as.scheduler.RemoteExecutable;
import oracle.as.scheduler.RequestExecutionContext;
import oracle.as.scheduler.RequestNotFoundException;
import oracle.as.scheduler.RequestParameters;
import oracle.as.scheduler.RuntimeServiceException;
import oracle.as.scheduler.SchedulerException;
import oracle.as.scheduler.async.RemoteAsyncHelper;

@Stateless(name = "EssAsyncAnnotatedBean", mappedName = "ejb/essAsyncAnnBean")
public class EssAsyncAnnotatedBean implements RemoteExecutable {
    @Resource
    SessionContext sessionContext;

    public void execute(RequestExecutionContext requestExecutionContext,
                        RequestParameters requestParameters) throws
    ExecutionErrorException, ExecutionWarningException,
    ExecutionPausedException,ExecutionCancelledException
    {
        RemoteAsyncHelper asyncHelper = null;

        try {
            asyncHelper = new RemoteAsyncHelper(requestExecutionContext,
requestParameters);

            //Initiate processing
            initiateProcessing(requestExecutionContext, requestParameters);

            //Get processed results
            Future<Result[]> results =
getProcessedResults(requestExecutionContext, requestParameters);

            //do further processing

            //Finally, complete the request
            asyncHelper.onSuccess();
        }
        catch (Exception e)
        {
            try
            {
                asyncHelper.onBizError(e.getMessage());
            }
            catch (Exception f)
            {
```

```
            }
        }
    }

    @Asynchronous
    public void initiateProcessing(RequestExecutionContext
requestExecutionContext,
                                   RequestParameters requestParameters)
    {
        //startProcessing
    }

    @Asynchronous
    public Future<Result[]> getProcessedResults(RequestExecutionContext
requestExecutionContext,
                                                RequestParameters
requestParameters)
    {
        Result[] resultsArr = null;
        //do processing
        return new AsyncResult<Result[]>(resultsArr);
    }

}
```

13

Creating and Using Web Service Jobs

This chapter describes how to use Oracle Enterprise Scheduler to create Web Service jobs and contains the following sections:

- Introduction
- Predefined Web Service Job Types
- Cancel and Fault Support
- Configuration Properties for Web Service Jobs
- Oracle Web Services Manager Policy Configuration
- Creating a Web Service Job Definition

Introduction

Web services provide a standard means to expose services on the web. Web services are accessible from a URL and use SOAP and XML as their payloads. Web services are described by the WSDL standard that defines the interface and the URL of the web service. The following are examples of web services

- SOA suite composites
- Oracle Service Bus proxy services
- ADF Business Component web services

Web services can expose one-way, synchronous, or asynchronous operations. A one-way web service operation is a fire-and-forget operation where the web service does not return a response. A synchronous web service operation returns a response as part of the same web service invocation. Typically, a web service client blocks until the synchronous operation response is received. An asynchronous web service operation involves two one-way messages: one for the web service operation request and a separate one for the response. Asynchronous web service operations typically represent long running operations. A web service client invokes an asynchronous web service operation, but does not wait for the response. Instead, the client specifies a callback URL at which to receive the response from the web service. The web service processes the request in the background and uses a callback operation to return the response to the client-specified callback URL.

Oracle Enterprise Scheduler supports web service jobs that use synchronous, one-way and asynchronous interfaces. The web service job definition can be defined using JDeveloper (as part of a hosting application or client application) or using Oracle Enterprise Manager Fusion Middleware Control. When the web service job type is selected, a wizard leads the user through a simple set of steps to define the web service job (see the example in Using Oracle Enterprise Manager Fusion Middleware Control

to Create a Job Definition). This wizard obtains the WSDL URL and asks the user to select the WSDL service, port type, and operation. It then creates sample XML for the payload based on the WSDL, and allows the user to update it. Asynchronous and synchronous web service may optionally have a designated operation for cancel. If there is a cancel operation, the operation is selected and the sample XML code for the cancel operation is modified. The wizard populates a set of predefined system properties in the job definition with values entered or derived from what the user enters in the wizard.

Note:

The WSDL URL in a web services job type must be a concrete WSDL URL. It cannot be an abstract WSDL URL.

The job definition can have user defined parameters. Elements or attributes in the invoke or `cancel` payload XML code can specify that one of these parameters be plugged in as the element value by specifying a token substitution instruction. For example, plug in the parameter `customerID` with the token substitution command `${ESS_REQ:customerID}`. This allows the job submitter to just enter parameter values and have the XML payload constructed from them. Token substitutions can also be specified for the WSDL base URL and WSDL relative URL system properties. For more information about token substitution see Using Tokens and Logical Clusters .

Web service jobs are secured by Oracle Web Services Manager (OWSM) policies. In Oracle Enterprise Manager Fusion Middleware Control or JDeveloper, you can attach OWSM directly attached policies for the job definition for the invocation (client policy) and the callback (service policy) actions. You can use globally attached policies to define policies globally or you can secure individual job definitions with directly attached policies.

Note:

See the following sections in *Securing Web Services and Managing Policies with Oracle Web Services Manager* for more information about using Oracle Web Services Manager (OWSM) policies:

- "Attaching Policies Globally Using Fusion Middleware Control"
- "Attaching Policies Globally Using WLST"
- "Attaching Policies Directly Using WLST"

Progress messages are supported for asynchronous web service jobs. These messages are written to the job log. In the callback operation, the job can indicate if the job succeeded or failed. The callback message comprises the job's output

Predefined Web Service Job Types

As described in Introduction, Oracle Enterprise Scheduler supports three predefined web service job types. The web service predefined job types are shown in Table 13-1.

Table 13-1 The Predefined Web Service Job Types

Predefined Job Type	Description
`/oracle/as/ess/core/JobType/AsyncWebserviceJobType`	(Asynchronous) The caller invokes the web service, the web service runs asynchronously in the background, and the web service calls back to the caller at a callback URL
`/oracle/as/ess/core/JobType/SyncWebserviceJobType`	(Synchronous) The caller blocks until the response is returned (request/response)
`/oracle/as/ess/core/JobType/OnewayWebserviceJobType`	(One-way) The caller does not expect a response. The web service runs in the background (fire-and-forget).

The job type you specify in the web service job definition implicitly determines whether the configured web service operation is invoked using an asynchronous, synchronous or one-way (fire-and-forget) operation.

Cancel and Fault Support

Supporting a cancel operation for web service jobs is optional. The web service may support a cancel operation that allows a running web service invocation to be canceled. The cancel operation must not be an abort operation (hammer-on-head style), where the composite is terminated and never calls back to complete the original operation. A well-behaved cancel implementation by a web service provider ensures the original web service operation returns an "operation canceled response," with a predefined `wsa:Action` code (see Table 13-2) in the SOAP response header. The cancel web service operation must be a synchronous web service operation.

Both synchronous and asynchronous web service jobs can indicate whether the web service operation was canceled or resulted in a fault (error) by specifying the appropriate value in the `wsa:Action` SOAP response message header. If the callback response SOAP message does not match the “Canceled" or “Fault" response (through one of the mechanisms listed below), then the job state is “Succeeded".

The Table 13-2 shows the different web service operation statuses that can be specified using the SOAP `wsa:Action` header.

Table 13-2 SOAP Web Service Operation Statuses

Action Code Name	Action URI
Cancelled	"http://xmlns.oracle.com/schedulercallback/wsOperationCancelled"
Fault	• Standard web service addressing: "`http://schemas.xmlsoap.org/ws/2004/08/addressing/fault`" • Oracle application server: "`http://xmlns.oracle.com/oracleas/schema/oracle-fault-11_0/Fault`"

The Oracle SOA Suite does not support setting `wsa:Action` message headers. As an alternative you can add one of the strings listed in Table 13-3 to the SOAP `body` element of a callback message.

Table 13-3 Oracle SOA Suite Status Operations

Operation	String
Cancelled	"wsOperationCancelled"

Operation	String
Fault	"wsOperationFault"

Configuration Properties for Web Service Jobs

Table 13-4 lists the properties associated with the web service job type.

Table 13-4 Web Service Job Configuration Properties

Property Name	Description
`SYS_EXT_wsWsdlBaseUrl`[1]	The base URL part of WSDL URL.
`SYS_EXT_wsWsdlUrl`[1]	The relative part of the web service WSDL URL (must be a concrete WSDL URL). Either the `SYS_EXT_wsEndpointUrl` property or the `SYS_EXT_wsWsdlUrl` property must be completely specified. For example, either `SYS_EXT_wsWsdlBaseUrl` and `SYS_EXT_wsWsdlUrl` are both configured, or `SYS_EXT_wsEndpointBaseUrl` and `SYS_EXT_wsEndpointUrl` are both configured.
`SYS_EXT_wsEndpointBaseUrl`[1]	The base URL part of endpoint URL.
`SYS_EXT_wsEndpointUrl`[1]	The relative part of the web service endpoint URL (must be a concrete WSDL URL). Either the `SYS_EXT_wsEndpointUrl` property or the `SYS_EXT_wsWsdlUrl` property must be completely specified. For example, either `SYS_EXT_wsWsdlBaseUrl` and `SYS_EXT_wsWsdlUrl` are both configured, or `SYS_EXT_wsEndpointBaseUrl` and `SYS_EXT_wsEndpointUrl` are both configured.
`SYS_EXT_wsTargetNamespace`	The target name space.
`SYS_EXT_wsServiceName`	The service name.
`SYS_EXT_wsPortName`	The port name.
`SYS_EXT_wsOperationName`	The operation name.
`SYS_EXT_invokeMessage`[1]	The XML submit message used to invoke the web service.
`SYS_EXT_wsCancelOperationName`	Optional. The cancel operation name.
`SYS_EXT_wsCancelMessage`[1]	Optional. The XML message for the web service cancel operation.
`SYS_externalJobType`	Optional. The supported values are “ADFBC", “OSB" or “SOA". Any other value is invalid.

[1] This property can be specified using token substitution. Refer to Using Tokens and Logical Clusters for more information.

If the `SYS_EXT_wsEndpointBaseUrl` property and the `SYS_EXT_wsEndpointUrl` property are specified in the job definition, Oracle Enterprise Scheduler has enough information to invoke the web service. If the `SYS_EXT_wsEndpointBaseUrl` and `SYS_EXT_wsEndpointUrl` properties are not specified in the job definition and the `SYS_EXT_wsWsdlBaseUrl` and the `SYS_EXT_wsWsdlUrl` properties are specified, Oracle Enterprise Scheduler retrieves the WSDL at runtime (before invoking the job), gets the `EndpointUrl` and `TargetNamespace` property values from the WSDL and invokes the web service.

The SYS_EXT_wsServiceName, SYS_EXT_wsPortName and SYS_EXT_wsOperationName properties must be specified to identify the specific web service operation to be invoked.

The SYS_EXT_invokeMessage property contains the XML message (SOAP body payload) for invocation. This can either be an XML template or full XML.

An XML template contains tokens that are replaced at runtime. The job submitter specifies the parameter values to substitute for the tokens in the template. If full XML is used without tokens, no substitution is required and the specified XML in the job definition is used “as is" for job invocation.

Note:

The angle brackets (“<“and “>") in XML statements must be escaped.

After it is invoked, the remote web service can log progress messages to update its status. These messages are logged by the web service job and are available in the request logs. The web service response XML is captured as job output.

If the SYS_EXT_wsCancelMessage and SYS_EXT_wsCancelOperationName properties are configured with a cancel message, the message is invoked when a cancel operation is initiated on a running web service job. The cancel operation is always invoked as a synchronous web service operation.

The cancel message SOAP header is automatically populated with the WS-Addressing relatesToId property set to the wsa:messageId associated with the invoke web service operation. The cancel operation uses the same OWSM policy as the invoke operation. If the SYS_EXT_wsCancelMessage property is not configured, it indicates that the web service does not support cancellation and therefore cannot be canceled.

The SYS_externalJobType property allows web service job definitions to specify a web service type (ADFBC, Service Bus or SOA). Intended for future customized web service job implementations.

Oracle Web Services Manager Policy Configuration

The web service job type uses decoupled Oracle Web Services Manager (OWSM) policy subjects (Job-Invoke, Job-Callback) and associated globally attached policies and directly attached policies for web service invocation and callback operations.

The Job-Invoke policy subject is associated with all web service job types (one-way, synchronous and asynchronous), whereas the Job-Callback policy subject is available only for the asynchronous web service job type. The Job-Invoke and Job-Callback globally attached policies can be specified at the domain level and configured using EM or WLST.

If a Job-Invoke globally attached policy or a directly attached policy is not specified for a web service job definition, an attempt is made to invoke the web service anonymously. This only works for the one-way and synchronous job type, because anonymous callbacks are not supported for the asynchronous web service job type.

Job-Invoke and Job-Callback directly attached policies are specific to individual web service job definitions and are captured in the policy assembly descriptor associated with the web service job definition. These directly attached policies can be specified at design time using JDeveloper or at runtime using Oracle Enterprise Manager Fusion Middleware Control, or using WLST commands.

Globally attached policies for web service job policy subjects can be set up using Oracle Enterprise Manager Fusion Middleware Control or using a WLST script to configure domain-level globally attached policies for web service job policy subjects. Example 13-1 shows how such a script might look.

Example 13-1 WLST Script to Configure Globally Attached Policies

```
connect(adminuser, adminpassword, adminurl)

beginRepositorySession()
deletePolicySet('domain-default-job-invoke-client-policies')
describeRepositorySession()
commitRepositorySession()

beginRepositorySession()
deletePolicySet('domain-default-job-callback-service-policies')
describeRepositorySession()
commitRepositorySession()

print "-- create domain-default-job-invoke-client-policies --"
beginRepositorySession()
describeRepositorySession()
createPolicySet('domain-default-job-invoke-client-policies', 'job-invoke',
'Domain("*")')
attachPolicySetPolicy("oracle/
wss11_saml_token_with_message_protection_client_policy")
describeRepositorySession()
commitRepositorySession()

print "-- create domain-default-job-callback-service-policies --"
beginRepositorySession()
describeRepositorySession()
createPolicySet('domain-default-job-callback-service-policies', 'job-callback',
'Domain("*")')
attachPolicySetPolicy("oracle/
wss11_saml_or_username_token_with_message_protection_service_policy")
describeRepositorySession()
commitRepositorySession()
```

Creating a Web Service Job Definition

Both Oracle JDeveloper and Oracle Enterprise Manager Fusion Middleware Control offer convenient graphical user interfaces to help you create web service job definitions. Using Oracle JDeveloper to Create a Job Definition describes how to use Oracle JDeveloper to create a job definition and Using Oracle Enterprise Manager Fusion Middleware Control to Create a Job Definition describes how to use Oracle Enterprise Manager Fusion Middleware Control to do the same.

Using Oracle JDeveloper to Create a Job Definition

You can use Oracle JDeveloper to create a web service job definition while creating your application. Refer to Using Oracle JDeveloper to Generate an Oracle Enterprise Scheduler Application general information about how to use Oracle JDeveloper to create applications that work with the Oracle Enterprise Scheduler.

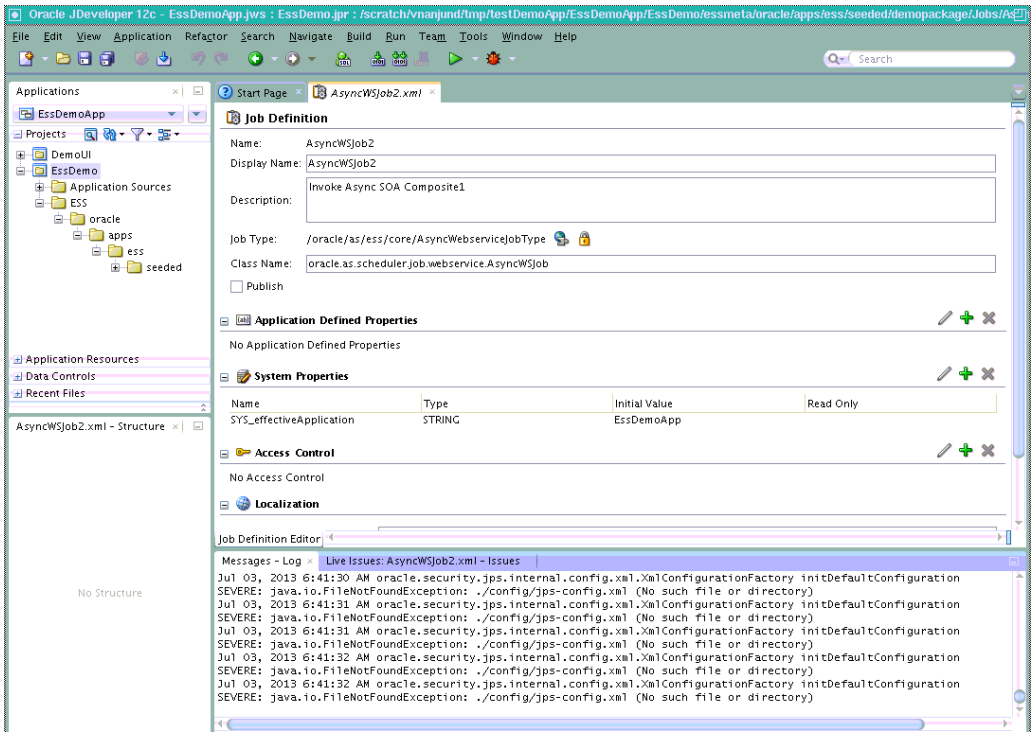
JDeveloper provides accessibility options, such as support for screen readers, screen magnifiers, and standard shortcut keys for keyboard navigation. You can also customize JDeveloper for better readability, including the size and color of fonts and the color and shape of objects. For information and instructions on configuring

accessibility in JDeveloper, see "Oracle JDeveloper Accessibility Information" in *Developing Applications with Oracle JDeveloper*.

The following steps show you how to create a job definition for an asynchronous web service job type.

1. Navigate to the Job Definition tab. Fill in the **Name**, **Display Name**, and **Description** fields and choose an appropriate web service job type as shown in the example in Figure 13-1. Have the WSDL URL for the target web service available.

Figure 13-1 Oracle JDeveloper: Job Definition Tab

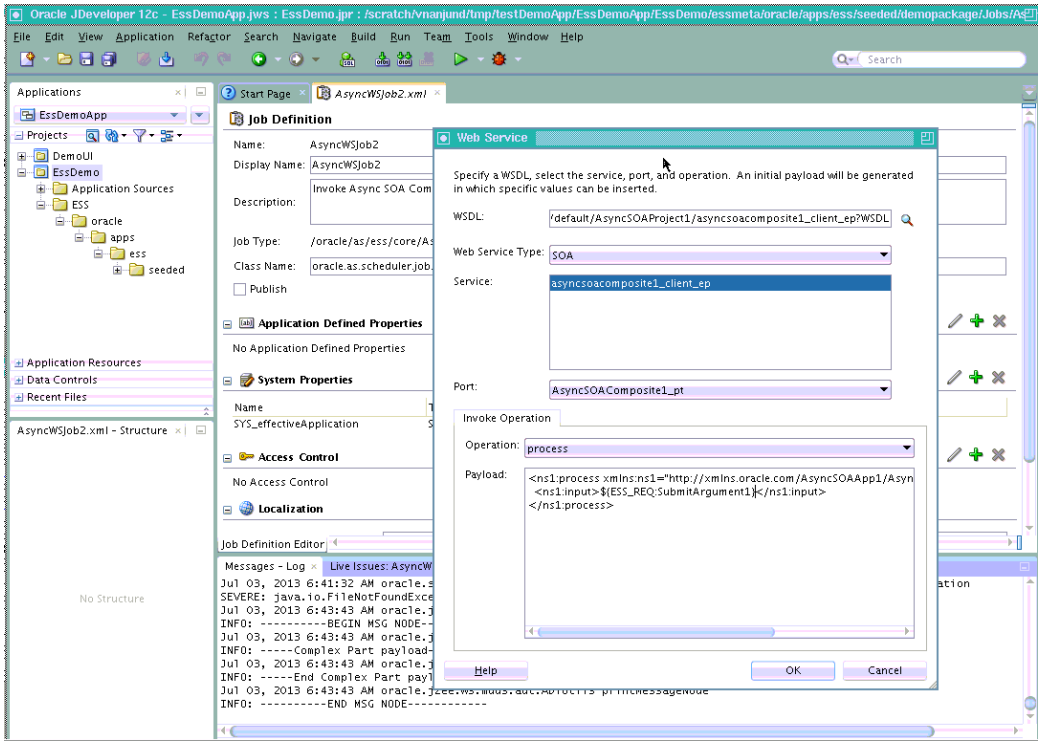


2. Click the **Web Service Explorer** button to launch the **Web Service** configuration wizard as shown in Figure 13-2. Enter the WSDL URL, the Service, Port, Operation and configure the payload XML as shown in the example, then click **OK**.

Note:

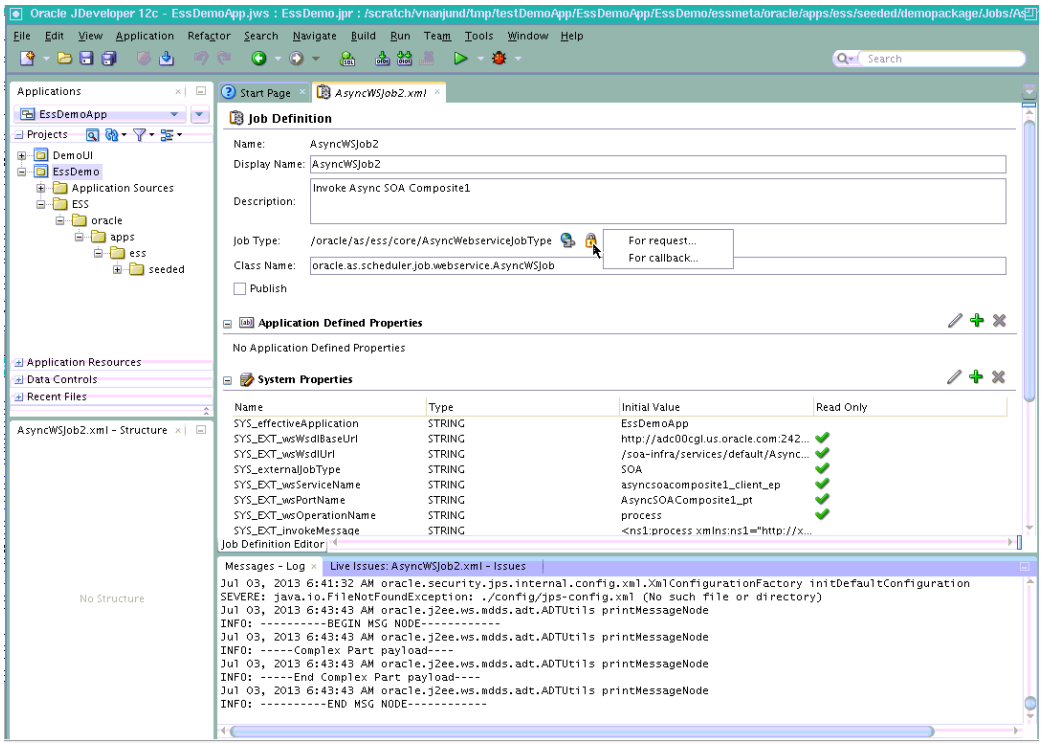
The example shown in Figure 13-2 shows the invoke XML payload with the substitutable token `SubmitArgument1`, whose value is provided at submission request time. Token substitution is described in Using Tokens and Logical Clusters .

Figure 13-2 Oracle JDeveloper: Web Service Popup



3. Click the **Specify Security Policies** button as shown in Figure 13-3. Select **For Request** to configure the directly attached policy for the Job-Invoke policy subject.

Figure 13-3 Oracle JDeveloper: Job Definition Tab



4. The Job-Invoke policy subject is available for all web service job definition types (one-way, synchronous and asynchronous. Select and attach the required OWSM client policy for the Job-Invoke directly attached policy. You should see a screen like the one shown in Figure 13-4.

Figure 13-4 Oracle JDeveloper: ESS Web Service Policies Popup

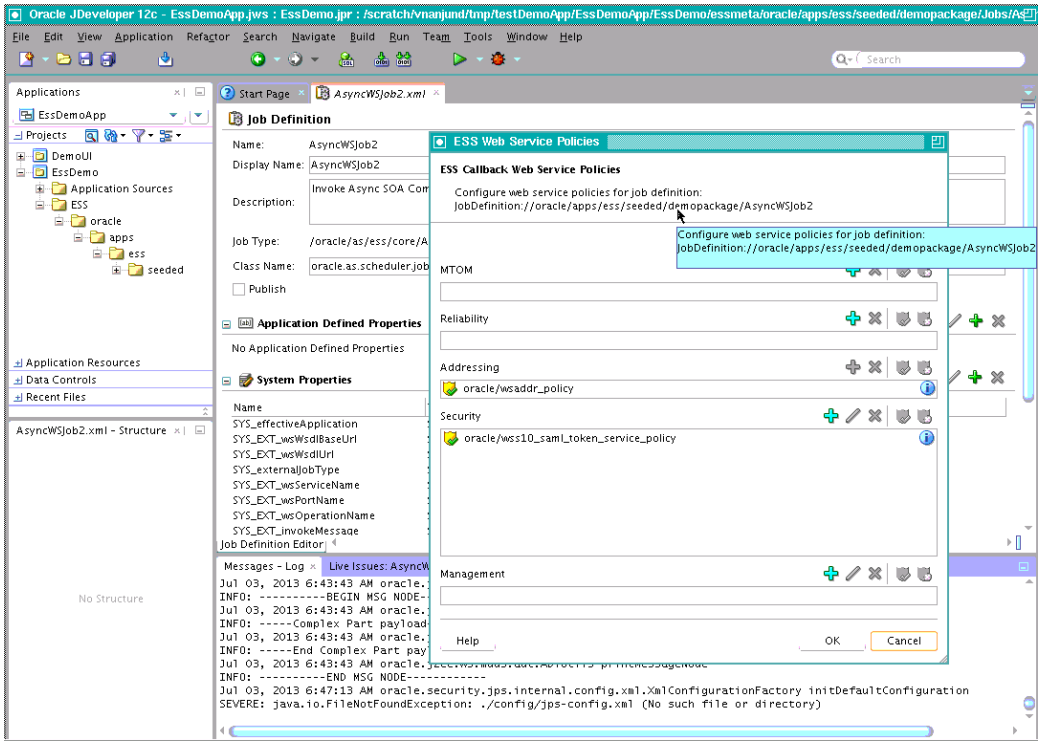
5. This completes directly attached policy configuration for a synchronous or one-way web service job definition. For asynchronous job definitions, you can also configure the directly attached policy for the Job-Callback policy subject.

   To configure a Job-Callback directly attached policy for an asynchronous job definition, repeat step 3 of this procedure and instead of **For request,** select **For callback**. Select and attach the required OWSM service policy for the Job-Callback directly attached policy as shown in Figure 13-4.

Note:

Post deployment, you can use Oracle Enterprise Manager Fusion Middleware Control to change job policies associated with web service job definitions.

Figure 13-5 Oracle JDeveloper: ESS Web Service Policies Popup

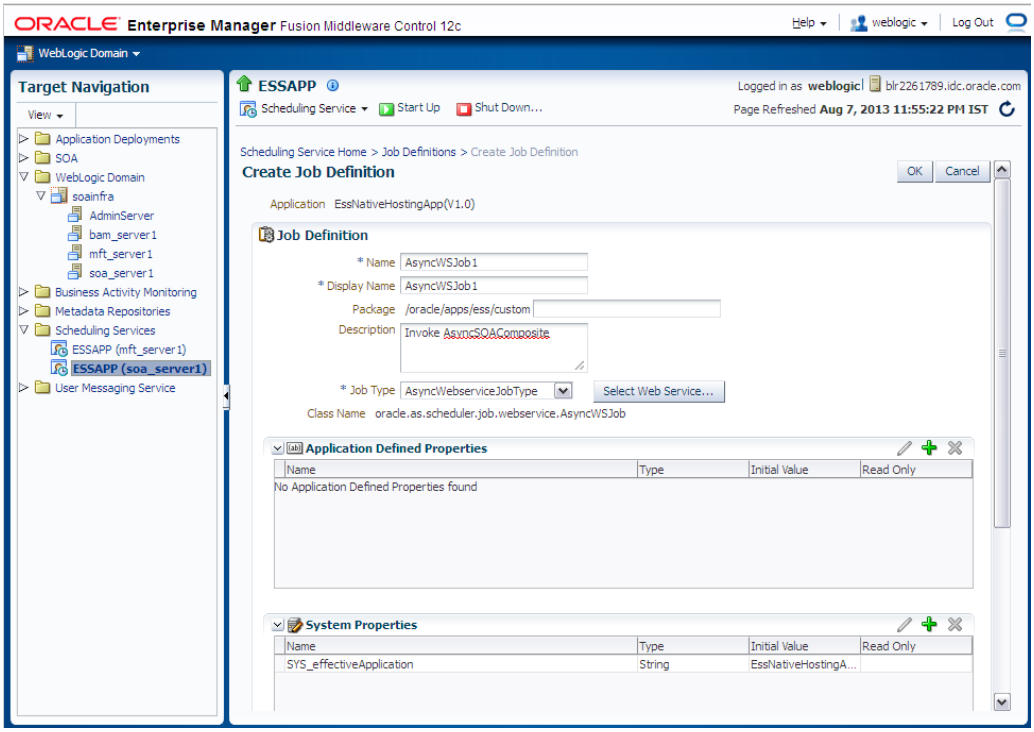


Using Oracle Enterprise Manager Fusion Middleware Control to Create a Job Definition

This procedure shows how to use the Oracle Enterprise Manger to create and configure a web service job definition.

1. Start and log in to Oracle Enterprise Manager Fusion Middleware Control.
2. From the navigation pane, expand the **Scheduling Services** folder and select the Oracle Enterprise Scheduler application.
3. From the Scheduling Services menu, select **Job Metadata** > **Job Definitions** and then click the **Create** button.
4. Fill in the **Name**, **Display Name**, and **Description** fields and choose the appropriate web service job type from the **Job Type** dropdown as shown in the example in Figure 13-6.

Figure 13-6 Fusion Middleware Control Console: Create Job Definition Page



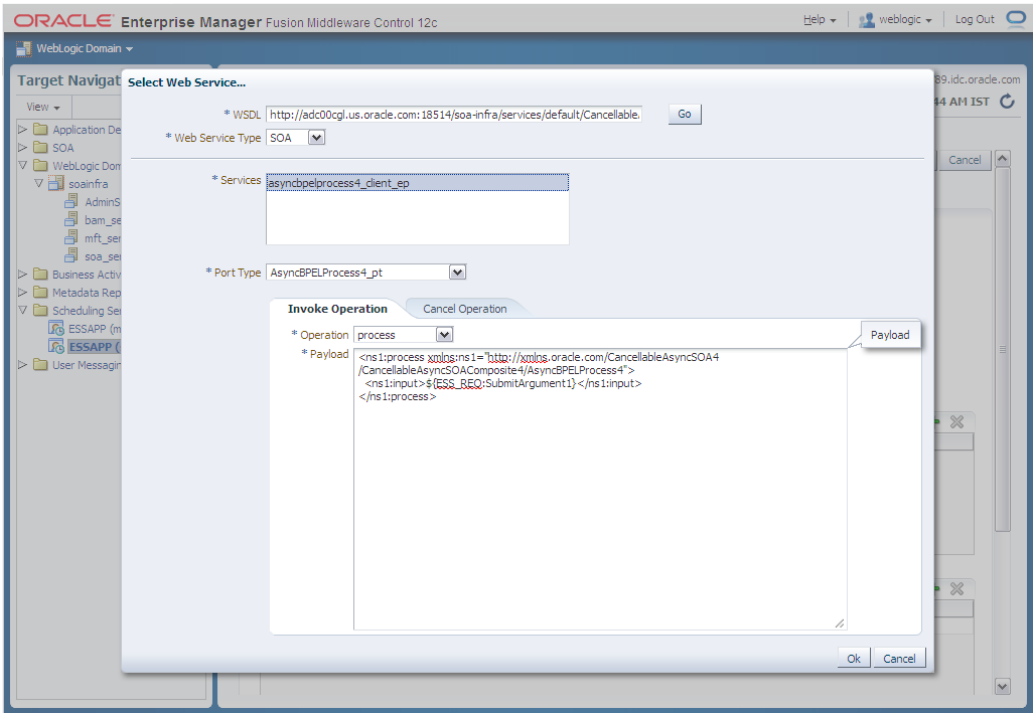
5. Click the **Select Web Service** button and enter the WSDL URL in the Select Web Service popup window. After you enter the URL, select the Service, Port Type, Operation and configure the Invoke Operation XML payload.

   The example shown in Figure 13-7 shows the invoke XML payload with the substitutable token `SubmitArgument1`, whose value is provided at submission request time.

Note:

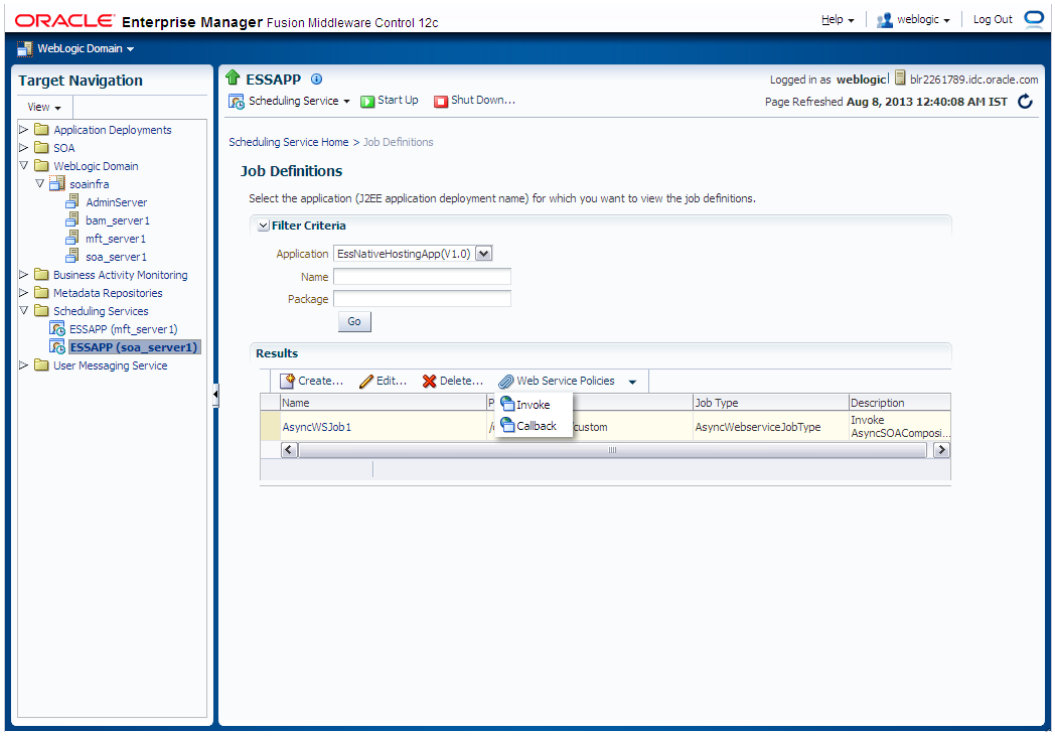
Token substitution is described in Using Tokens and Logical Clusters .

Figure 13-7 Fusion Middleware Control Console: Select Web Service Popup



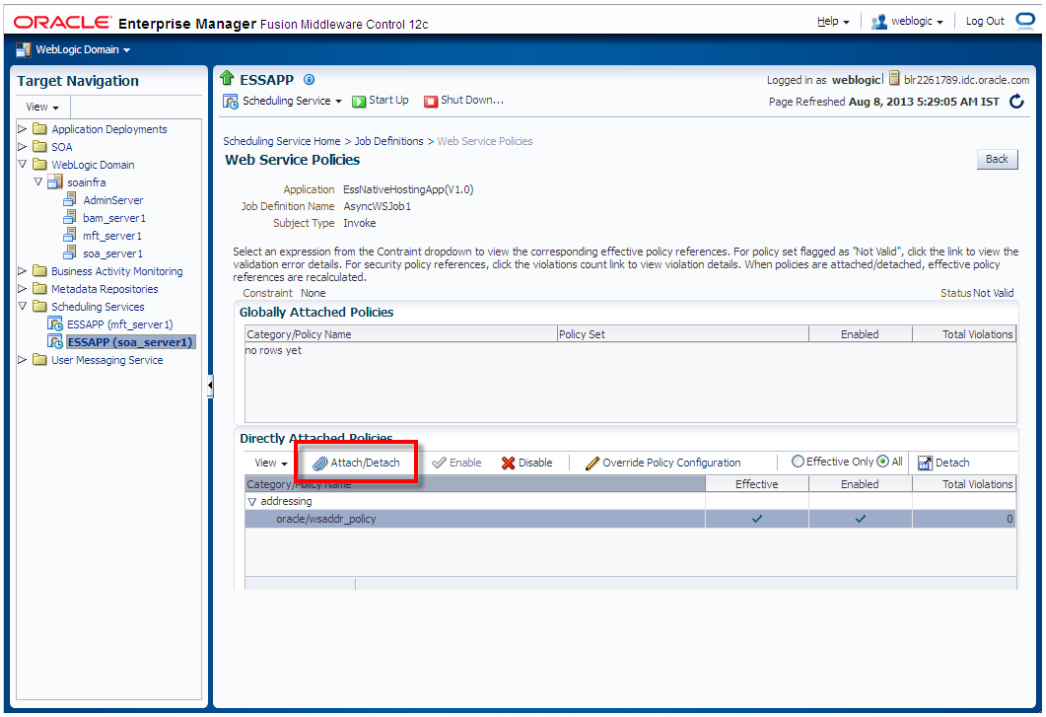
6. After you create the job definition, return to the Job Definitions page and select the job definition name ("AsyncWSJob1" in this example) in the **Results** table. Click the **Attach/Detach Policy** button and select **Invoke** as shown in Figure 13-8 to configure the directly attached policy for the Job-Invoke policy subject. The Job-Invoke policy subject is available for all web service job type definitions.

Figure 13-8 Fusion Middleware Control Console: Job Definitions Page



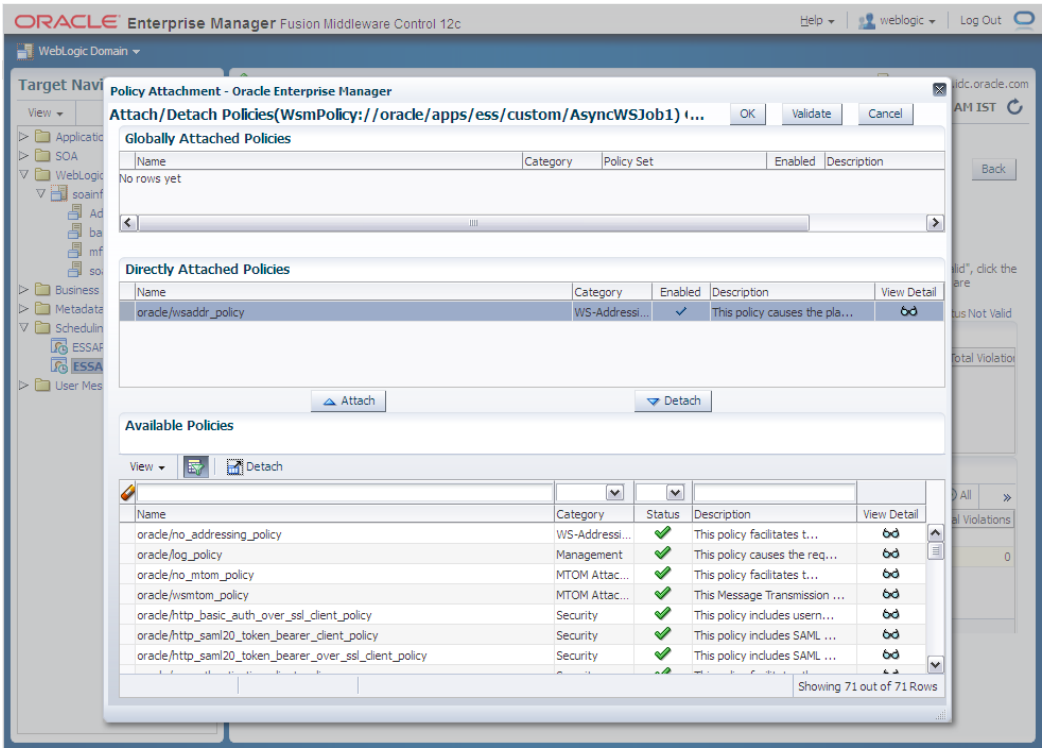
7. In the Web Services Policy page, select the policy and click the **Attach/Detach** button to attach the required OWSM client policy for the Job-Invoke directly attached policy. This is shown in Figure 13-9. The **Attach/Detach Policies** popup is displayed.

Figure 13-9 Fusion Middleware Control Console: Web Service Policies Page



8. Select the policy and click **OK** to complete the attachment.

Figure 13-10 Fusion Middleware Control Console: Policy Attachment Popup



9. This completes directly attached policy configuration for a synchronous or one-way web service job definition. For an asynchronous job definitions, you can also configure the directly attached policy for the Job-Callback policy subject.

   To configure Job-Callback directly attached policy for an asynchronous job definition, repeat step 6 of this procedure and instead of **Invoke,** select **Callback**. Select and attach the required OWSM service policy for the Job-Callback directly attached policy.

14

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:

- Introduction to Creating Process Job Definitions
- Creating and Storing Job Definitions for Process Job Types
- Using an Agent Handler for Process Jobs
- Process Job Locale

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

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 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 Using the Runtime Service.

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 Using the Metadata Service .

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 14-1 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 14-1 System Properties for Process Type Jobs

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

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

Example 14-1 shows a sample job definition with a `PROCESS_TYPE`.

As shown in Example 14-1, 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 Accessing the Metadata Service. Use the Metadata Service `addJobType()` method to store the `JobType` in metadata.
- The `MedatdataObjectId`, returned by addJobType(), uniquely identifies metadata objects in the metadata repository using a unique identifier.

Example 14-1 Creating an Oracle Enterprise Scheduler Job Definition and Setting Job Definition Properties

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

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

        try
        {
            ParameterInfo pinfo;
            ParameterList plist;

            mshandle = metadata.open();

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

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

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

```
            jobType.setParameters(plist);

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

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

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

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

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

            jobDef.setParameters(plist);

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

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:

After 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 14-1 shows how to create a job definition using the job definition constructor and the job type. Table 14-1 describes some of the system properties that are associated with the job definition.

As shown in Example 14-1, 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 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`.

Using an Agent Handler for Process Jobs

Oracle Enterprise Scheduler requires an agent handler to manage individual process jobs. The agent handler validates, spawns, monitors and controls process job execution, and also returns the exit status of process jobs to Oracle Enterprise Scheduler. The agent handler also monitors Oracle Enterprise Scheduler availability and handles job cancellation requests. In the event of abnormal job termination (or job cancellation requests), the agent handler 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 agent handler can generate its log under the `/tmp` folder. Log generation 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.

Choosing an Agent Handler

Oracle Enterprise Scheduler provides two different agent handlers, the Java agent handler and the Perl agent handler. Both agent handlers are functionally equivalent with the exception that the Java agent handler does not support `terminate-spawned-process-on-restart` behavior on Windows.

By default, Oracle Enterprise Scheduler uses the Java agent handler for requests in standard and extended mode. It always uses the Perl agent handler for requests in Fusion mode. To use the Perl agent handler in standard and extended request modes, you must add the `PerlCommand` property to the `ess-config.xml` file associated with the hosting application running the process job as shown in the following example.

```
<EssProperties>
   <EssProperty key="RequestFileDirectory" value="/tmp/ess/
requestFileDirectory"/>
   <EssProperty key="RequestFileDirectoryShared" value="false"/>
     ...
   <EssProperty key="PerlCommand" value="/usr/bin/perl"/>
</EssProperties>
```

You can use token substitution to specify environment dependent values like directory names. Refer to Using Tokens and Logical Clusters for more information.

The Oracle Enterprise Scheduler Perl agent handler requires Oracle Perl version 5.10 or later. Instructions for installing Perl to support process jobs can be found in the chapter "Configuring Perl to Support Process Jobs" in Oracle Fusion Middleware Administering Oracle Enterprise Scheduler.

Note:

If you run Oracle Enterprise Scheduler in a Fusion Applications environment you must use the Perl agent handler.

Process Job Locale

Individual process jobs can use different locales and encoding as determined by the locale environment variable settings applicable to the process job at execution time. For a process job, Oracle Enterprise Scheduler imports the request log and output file into the content store after completing the request.

Locale environment variables for a process job can be specified at multiple places including the process job definition and the hosting application's `ess-config.xml` file. The locale resolution logic for a process job uses the following precedence order to determine the effective `LC_ALL` and `LANG` environment variable values for the request:

1. `SYS_environment` variables associated with the request (highest precedence)
2. The hosting application's `ess-config.xml` file
3. The WebLogic server locale (lowest precedence)

For every process job, the effective locale and encoding is determined based on the above precedence order (with the effective `LC_ALL` value overriding the effective `LANG` value). This encoding applies to the log only, and not the output.

15

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:

- Introduction to Schedules
- Defining a Recurrence
- Defining an Explicit Date
- Defining and Storing Exclusions
- Defining and Storing Schedules
- Identifying Job Requests That Use a Particular Schedule
- Updating and Deleting Schedules

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.

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.

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 15-1 shows the recurrence fields helper classes available to specify a recurrence pattern.

Table 15-1 Recurrence Field Helper Patterns

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

Recurrence Field	Description
`MONTH_OF_YEAR`	Defines the months of the year
`TIME_OF_DAY`	Defines the time of the day
`WEEK_OF_MONTH`	Enumerations for the week of a month
`YEAR`	Encapsulate the value of a year

Example 15-1 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.

In Example 15-1, 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 15-2 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.

In Example 15-2, 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 15-3 shows a sample recurrence created using the `RecurrenceFields` helper class and a monthly frequency.

Example 15-3 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 15-4 shows a sample recurrence created using the `RecurrenceFields` helper class and a monthly frequency specified with a start date and time.

Example 15-4 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.

Example 15-1 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();
```

Example 15-2 Defining a Recurrence with Four Hourly Frequency

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

Example 15-3 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 15-4 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();
```

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 15-5 shows a sample recurrence created using an iCalendar expression.

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.

Example 15-5 Defining a Recurrence with an iCalendar String Expression

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

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 15-1).
- 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 does 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 are 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.

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 are 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 does not generate any occurrences before the start date or after the end date (and if specified, any associated start time or end time).

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.

How to Define an Explicit Date

Example 15-6 shows an explicit date definition.

Example 15-6 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 15-6 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.

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.

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.

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 15-7 shows how to create an `Exclusion` instance using a recurrence.

Example 15-7 defines an individual exclusion. For information about creating a list of Exclusions, see How to Create an Exclusions Definition.

Example 15-7 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);
```

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`.

Example 15-8 Creating and Storing a List of Exclusions in an ExlusionDefinition

```
Collection<Exclusion> exclusions = new ArrayList<Exclusion>();
Exclusion e = new Exclusion("Independence Day", recur);
exclusions.add(e);
ExclusionsDefinition exDef1 =
new ExclusionsDefinition("OrclHolidays1", "Annual Holidays", exclusions);
MetadataServiceHandle handle = m_service.open();
```

```
MetadataObjectId exId1 = m_service.addExclusionDefinition(handle,exDef1,
"METADATA_UNITTEST_PROD");
```

Finally, when you want to associate an `ExclusionsDefinition` with a schedule, you use the schedule `addExclusion()` method.

Example 15-8 shows how to create an `ExclusionDefinition` and store the definition to the metadata repository.

Note in Example 15-8 that the `ExclusionsDefinition` constructor requires three arguments.

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 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 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 Defining and Storing Exclusions

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,schedule,"HOW_TO_PROD");
   ```

3. Define the date, recurrences and exclusions.
4. Store the schedule using `addScheduleDefinition`.
5. Close the session with a `finally` block.

What Happens When You Define and Store a Schedule

Example 15-9 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 15-10 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 15-9 Creating a Schedule Recurrence with RecurrenceFields Helpers

```
Recurrence recur = new Recurrence(RecurrenceFields.FREQUENCY.WEEKLY, 1);
 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 15-10 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);
      }
   }
```

What You Need to Know About Handling Time Zones with Schedules

You can use a `java.util.TimeZone` object to set the time zone for a schedule. Use the Schedule `setTimeZone()` method to set or clear the `TimeZone` for a Schedule. The Schedule method `getTimeZone()`returns a `java.util.TimeZone` value if the Schedule object has as `TimeZone` set.

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 Administering Oracle Enterprise Scheduler*.

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 Administering Oracle Enterprise Scheduler*.

16

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:

- Introduction to the Oracle Enterprise Scheduler Web Service
- Developing and Using ESSWebservice Applications
- ESSWebservice WSDL File
- Use Case: Using Oracle Enterprise Scheduler ESSWebservice from a BPEL Process

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 requires 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. Although the Oracle Enterprise Scheduler internal job execution model is asynchronous, the ESSWebservice APIs need not 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 16-1 summarizes the operations available with ESSWebservice.

Table 16-1 Summary of Operations Available with ESSWebservice

Operation	Communication Type	Description
`addPPAction`	Synchronous	Adds a post-processing action to a step in a job set request. This method is called prior to submitting the request. The method provides support for action previously supported by add_printer, add_notification, add_layout in concurrent processing. The parameters to these legacy routines are passed as arguments to addPPAction in the order in which they were declared in the original routine.
`addPPActions`	Synchronous	Similar to `addPPAction`, except that you can package multiple actions in your request.
`cancelRequest`	Synchronous	Cancels the processing of a request that is not in a terminal state.
Oracle Enterprise SchedulerOracle Enterprise SchedulerOracle Enterprise Scheduler Oracle Enterprise SchedulerOracle Enterprise Scheduler	Synchronous	Marks a request in a terminal state for deletion. This does not physically remove any data, although the request is no longer be accessible by most methods. For parent requests, this operation cascades to all children.
`getCompletionStatus`	Asynchronous	Registers for an asynchronous status update when the request completes. A one-way operation with a separate asynchronous response.
`getRequestExecution Context`	Synchronous	Gets an `oracle.as.scheduler.RequestExectionContext` object from a serialized request execution context string. This operation should only be invoked from a remote running ESS job.
`getRequestDetail`	Synchronous	Gets the runtime details of the specified request.
`getRequestState`	Synchronous	Retrieves the current state of the specified request.
`holdRequest`	Synchronous	Withholds further processing of a request that is in `WAIT` or `READY` state. For parent requests, this operation cascades to all eligible child requests.
`releaseRequest`	Synchronous	Releases a request from the `HOLD` state. For parent requests, this operation cascades to all eligible child requests.
`setAsyncRequestStat us`	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.

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".
`submitRecurringRequest`	Synchronous	Submits a new recurring job request (a request with a schedule).
`submitRequest`	Synchronous	Submits a new job request. For more information, see Use Case: Using Oracle Enterprise Scheduler ESSWebservice from a BPEL Process

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 this type of Java EE application, so 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 and should pass it to the Oracle Enterprise Scheduler web service call wherever required (as defined in the WSDL).

Such an application is a regular Oracle Enterprise Scheduler client application, where the job request submission and management are done using ESSWebservice operations.

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 must 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 hosting 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 must still develop,

package and deploy a corresponding Java EE hosting application that contains all the required Oracle Enterprise Scheduler artifacts.

How to Develop and Use an ESSWebservice SOA Application with BPEL

For SOA clients all the SOA components such as a BPEL process 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.

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 invokes 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 the 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 the 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, the web service client must set these addressing headers.

Restrictions When Using 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.

ESSWebservice Implementation

The Oracle Enterprise Scheduler functionality is exposed as web a service using a Service Endpoint Interface (SEI) annotated with the JAX-WS annotations. The web service 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 a 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. In general, the web service layer reuses the existing data types where possible.

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
```

Note that you cannot invoke web service operations by directly accessing the ESSWebservice URL from a browser.

Use Case: Using Oracle Enterprise Scheduler ESSWebservice from a BPEL Process

The following example demonstrates how to use the ESSWebService from a BPEL process; in the BPEL process you use ESSWebService to submit a job request. The use case demonstrates one way of using Oracle Enterprise Scheduler for BPEL and SOA users. Experienced SOA users and designers may have other ideas for how to work with Oracle Enterprise Scheduler using the web service.

Oracle JDeveloper is used to create an application and the projects within the application that contain the code and support files for the application.

JDeveloper provides accessibility options, such as support for screen readers, screen magnifiers, and standard shortcut keys for keyboard navigation. You can also customize JDeveloper for better readability, including the size and color of fonts and the color and shape of objects. For information and instructions on configuring accessibility in JDeveloper, see "Oracle JDeveloper Accessibility Information" in *Developing Applications with Oracle JDeveloper*.

To create the ESSWebService sample application, follow these steps:

1. Start Oracle JDeveloper.
2. Click the **New Application** button.
3. In the **New Gallery - 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 example, enter `EssWebApplication`.

   b. In the **Directory** field, accept the default or specify a location for the application to be created.

   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`.

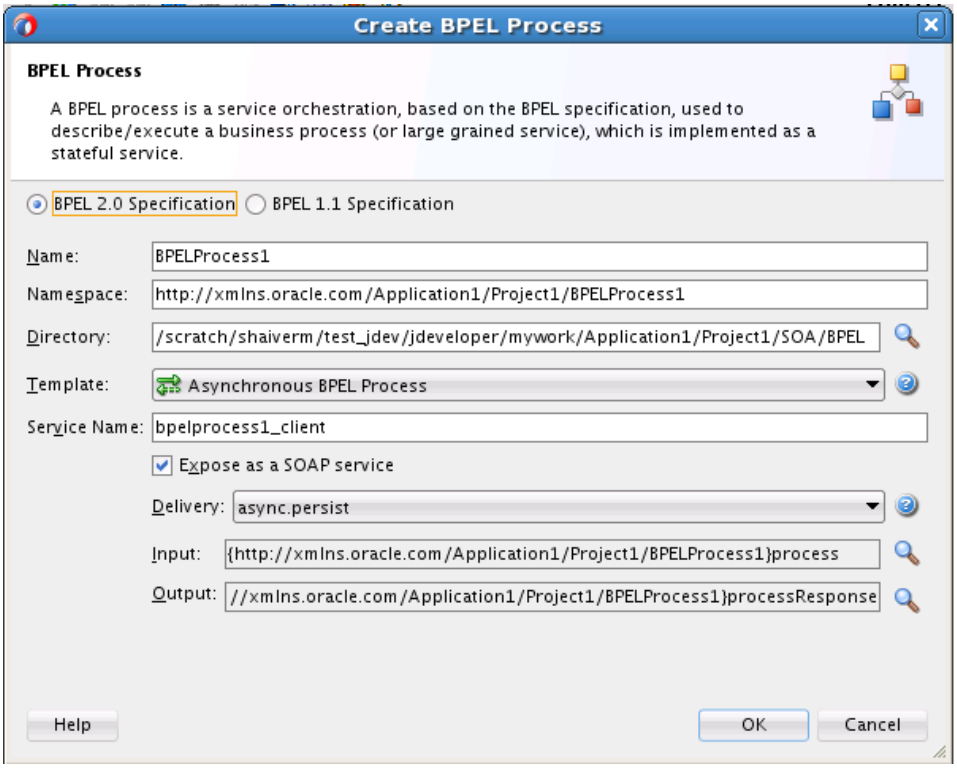
   b. On the **Project Features** tab, select **SOA Suite**.

   c. Click **Next**.

7. In the Configure SOA Settings dialog, select **Composite with BPEL Process** and click **Finish**.

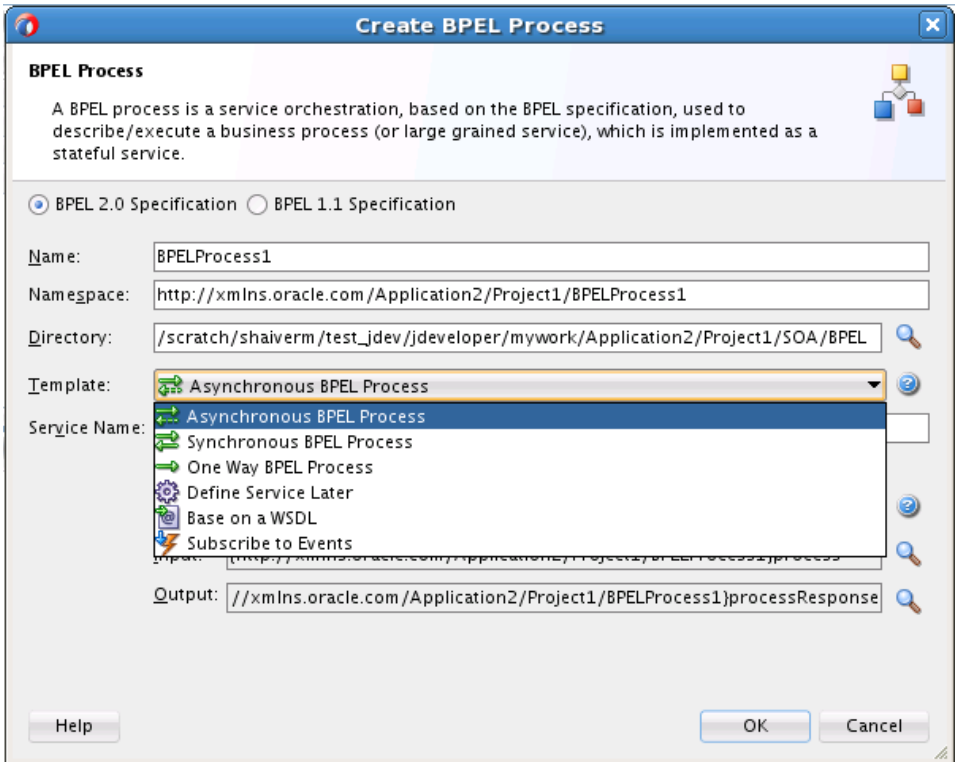
8. Choose one of the two BPEL specifications as shown in Figure 16-1.

Figure 16-1 Choose a BPEL Specification



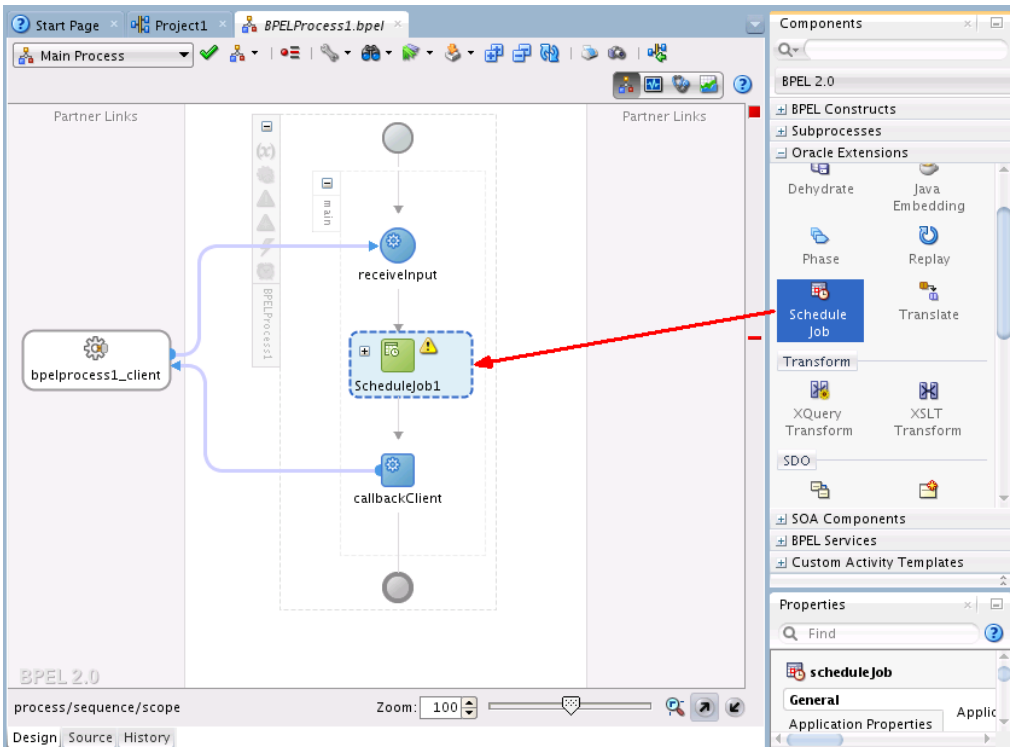
9. Select the service type from the **Template** dropdown menu as shown in Figure 16-2 and click **OK**.

Figure 16-2 Selecting the Service Type



10. In the **Editor** pane (`BPELProcess1.bpel`), drag the **Schedule Job** component from the Oracle Extensions section of the Components palette to the position between the **receiveInput** and **callBackClient** components as shown in Figure 16-3.

Figure 16-3 Adding the Schedule Job Component



11. Create a connection to the metadata server as shown in Figure 16-4.

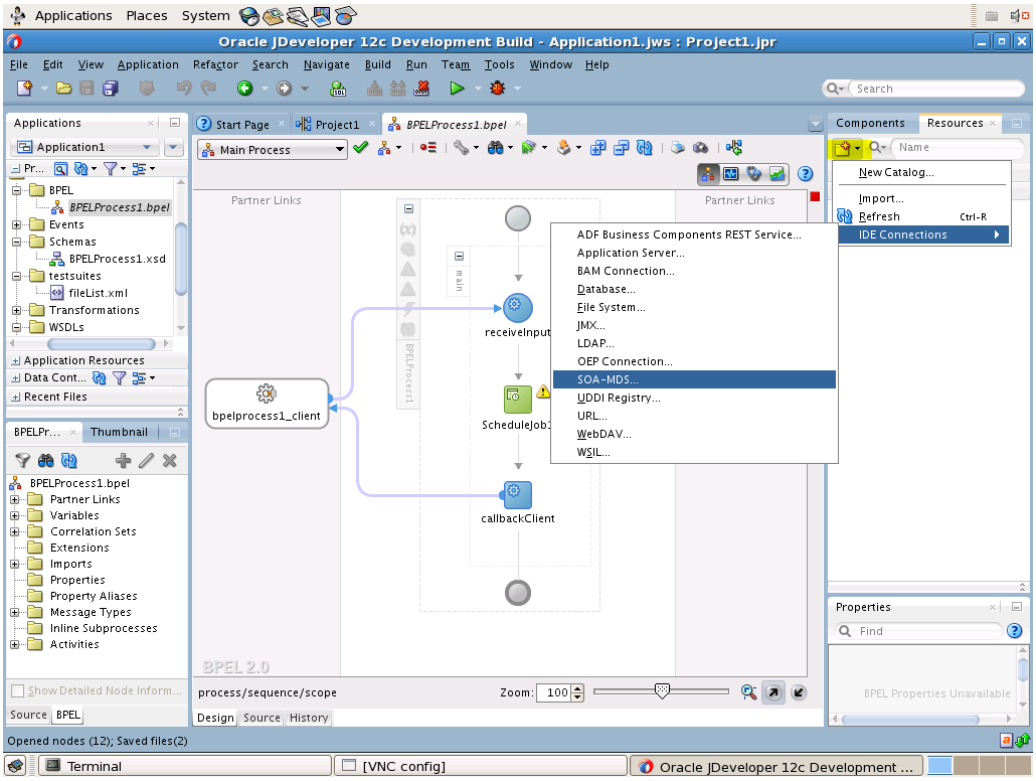
    a. Click the **New** button in the resource window.

    b. In the dropdown menu, select **IDE Connecitons** > **SOA-MDS**. In the Create SOA-MDS Connection dialog, fill in the appropriate information about your MDS server.

Note:

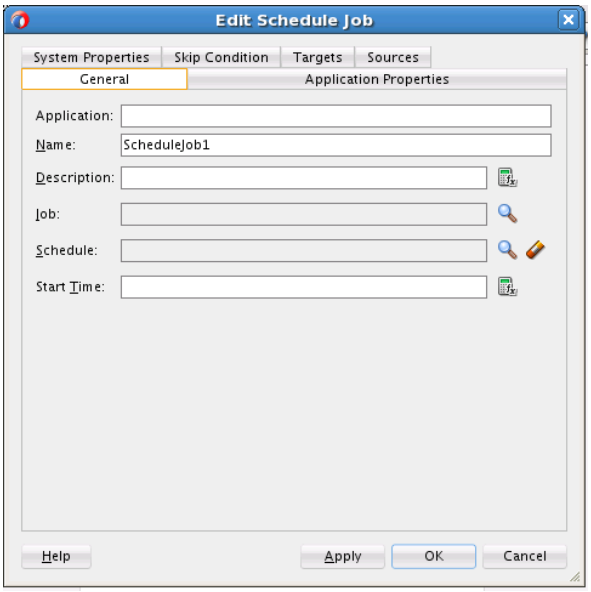
You can also create the connection by choosing **File** > **New Gallery** > **General** > **Connections** > **SOA-MDS Connections**.

Figure 16-4 Creating a Connection to the Metadata Server



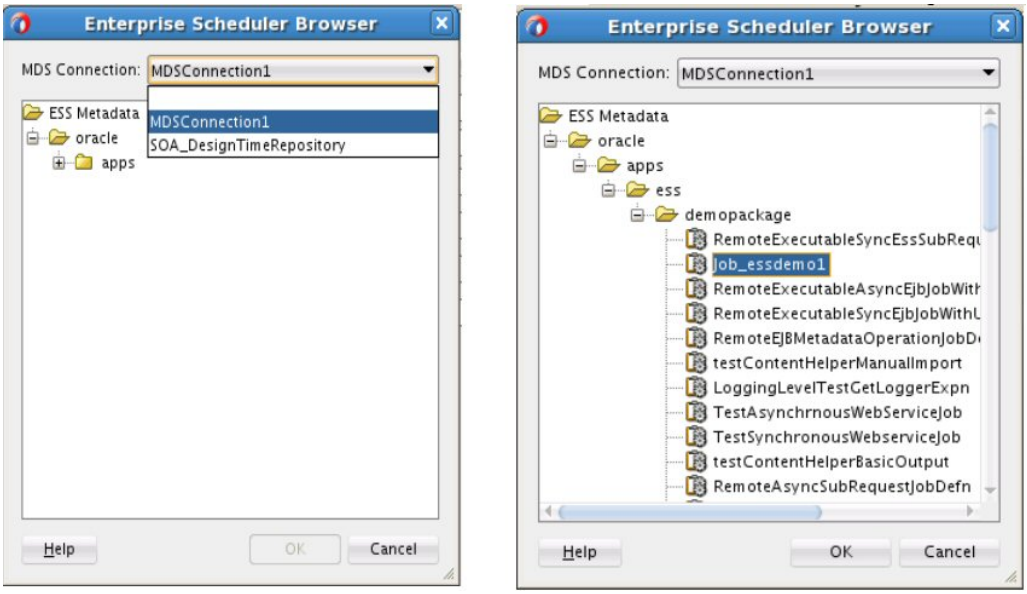
12. Right-click the **Schedulejob1** component and select the **Edit** item. This invokes the Edit Schedule Job dialog shown in Figure 16-5.

Figure 16-5 Editing Schedule Job



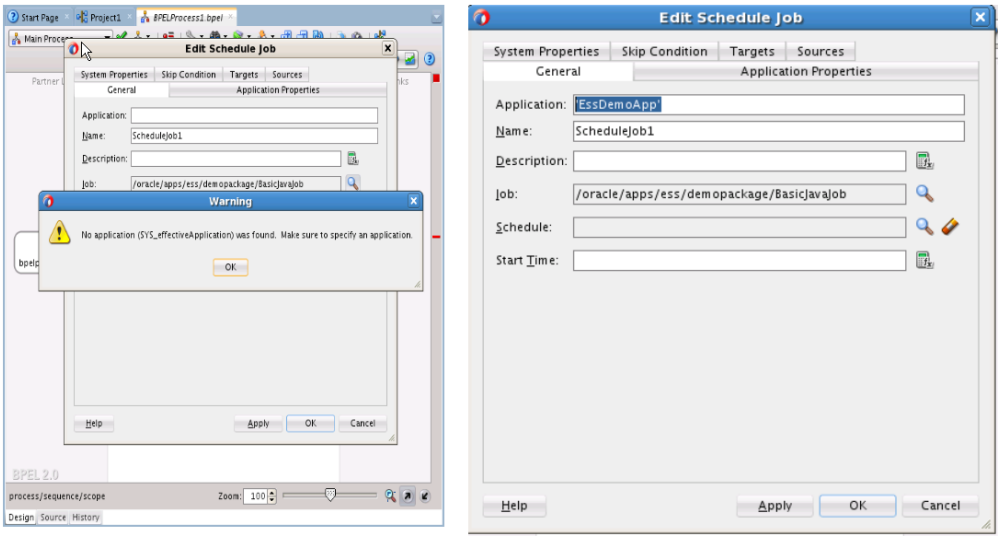
13. Click the **Job** browse button to select the job definition through the MDS connection. Figure 16-6 shows an example.

Figure 16-6 Selecting the Job Definition Through the MDS Connection



14. If the Sys_effectiveApplication property is not defined in the job definition you selected, you are prompted to provide it in **Application** field on the general tab. If **Sys_effectiveApplication** property is defined in the selected job definition, it appears in the **Application** field and cannot be edited. See Figure 16-7.

Figure 16-7 Defining the Sys_effectiveApplication Property (if not Already Defined)



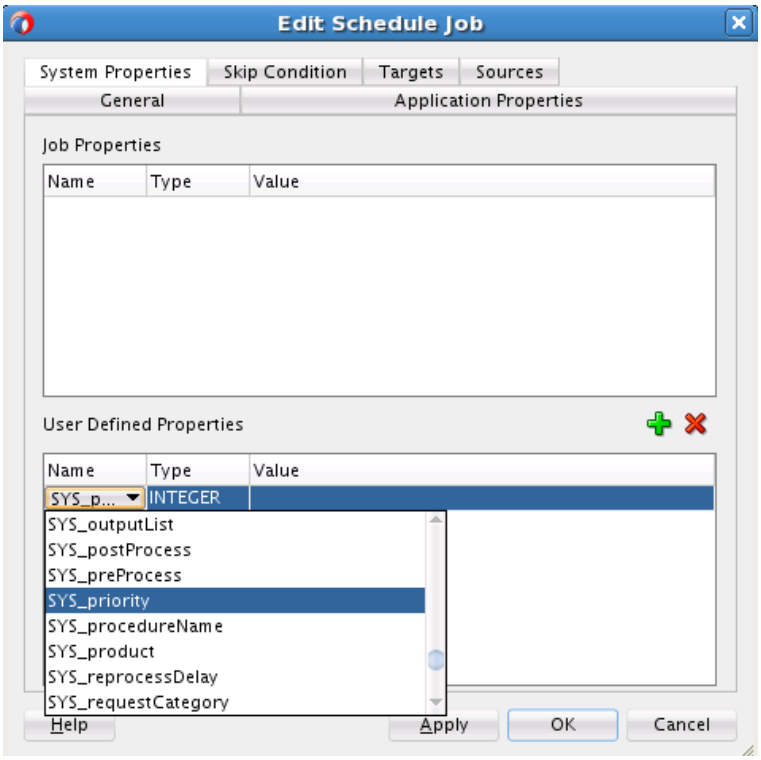
15. Add system properties:

    a. Select the **System Properties** tab.

    b. The **Job Properties** pane should be populated with system properties obtained through the MDS connection from the job definition.

    c. Use the **Add** button to add additional system properties in the **User Defined Properties** pane.

Figure 16-8 Adding User Defined Properties



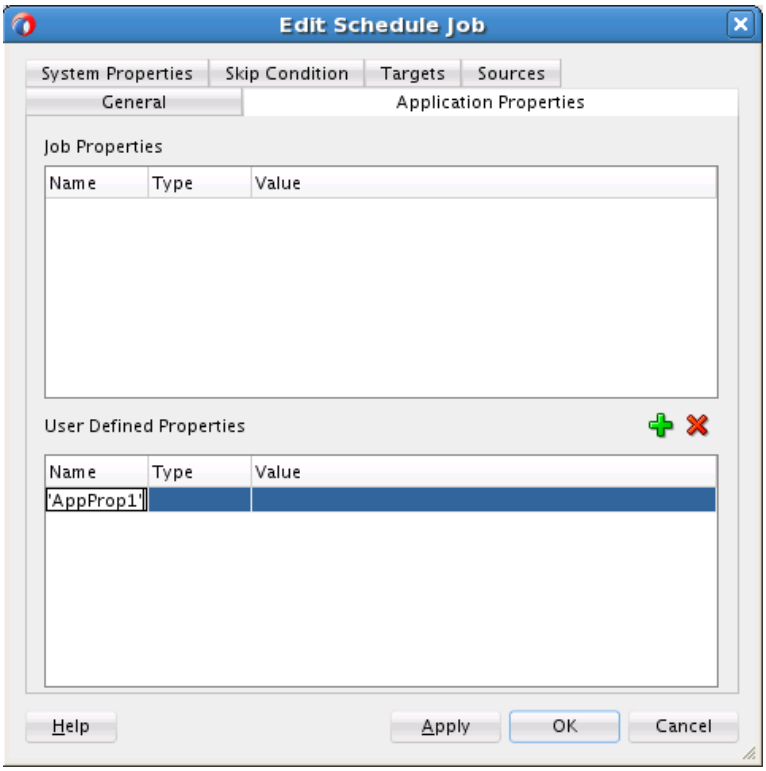
16. Add application properties:

    a. Select the **Application Properties** tab.

    b. The **Job Properties** pane should be populated with properties obtained through the MDS connection from the job definition.

    c. Use the **Add** button to add additional properties in the **User Defined Properties** pane.

Figure 16-9 Adding Additional User Defined Properties



17. Attach the WSDL URL.

    a. Click the **Project Editor** tab (Figure 16-10).

    b. Edit the **ESSService** component. Provide the **Name**, **WSDL URL**, **Port Type** and other information (Figure 16-11).

Figure 16-10 The Project Editor Tab

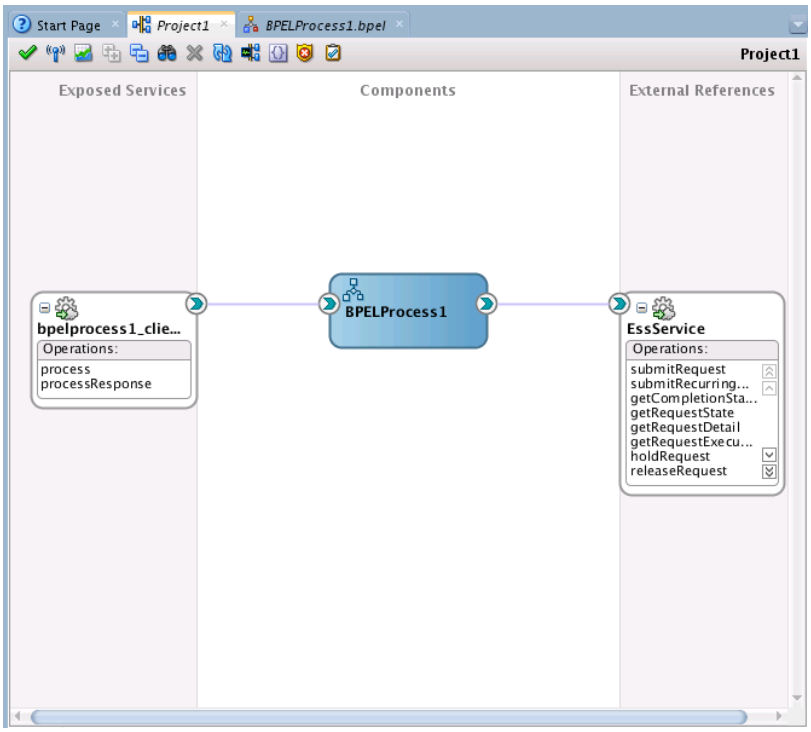
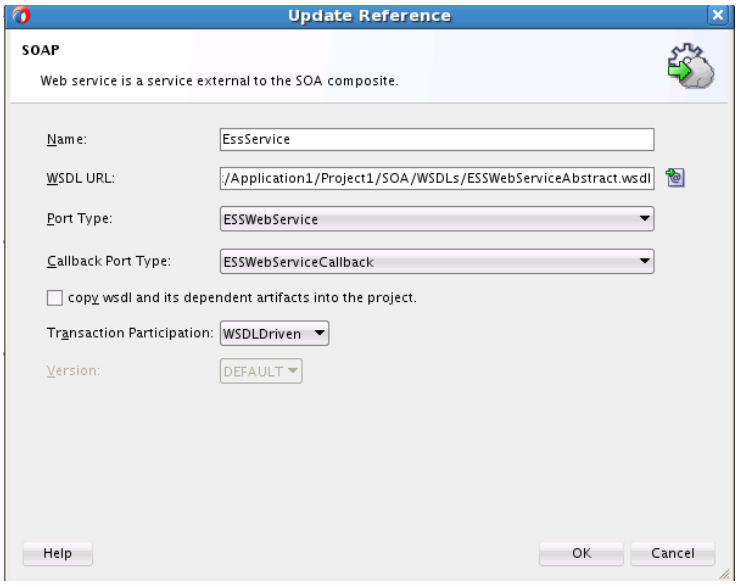


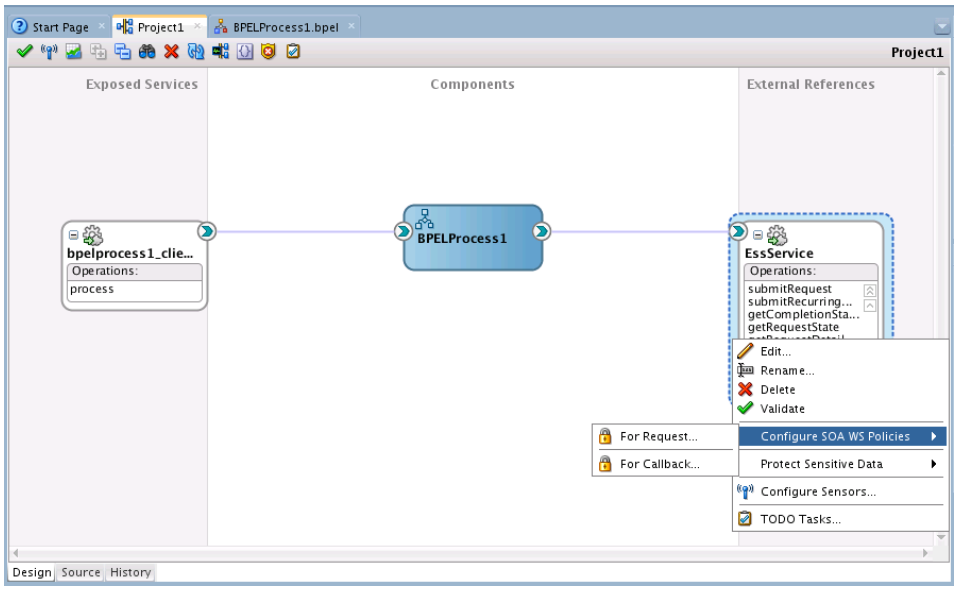
Figure 16-11 The Update Reference Dialog



18. Add a security policy to the service.

    a. In the Project Editor tab, right-click the Oracle Enterprise Scheduler web service and select **Configure SOA WS Policies** > **For Request** to open the Configure SOA WS Policies dialog as shown in Figure 16-12.

Figure 16-12 Opening the Configure SOA WS Policies



b. In the **Security** area of the Configure SOA WS Policies, click the **Add** button to attach the desired security policies. For example, oracle/`wss_http_token_client_policy` as shown in Figure 16-13 and Figure 16-14. If you are creating an asynchronous BPEL process you must also use this process to attach a service policy to the callback.

Figure 16-13 The Configure SOA WS Policies Dialog

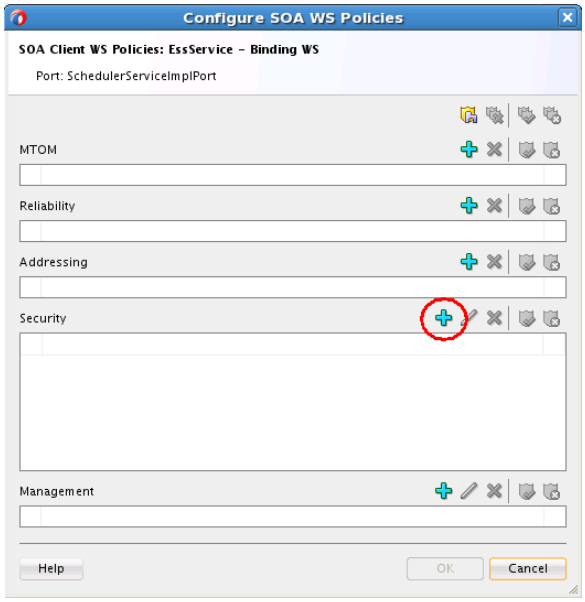
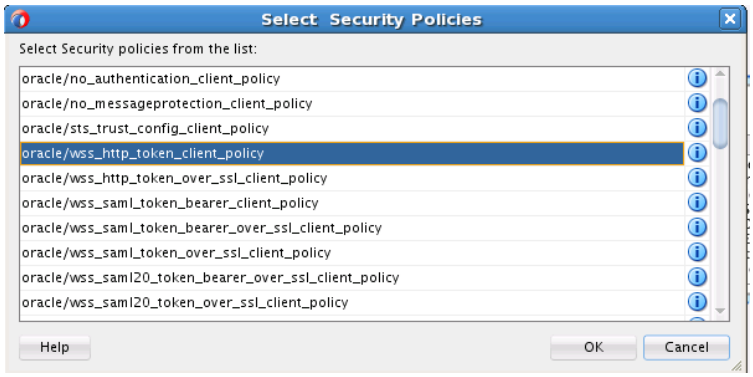


Figure 16-14 The Select Security Policies Dialog

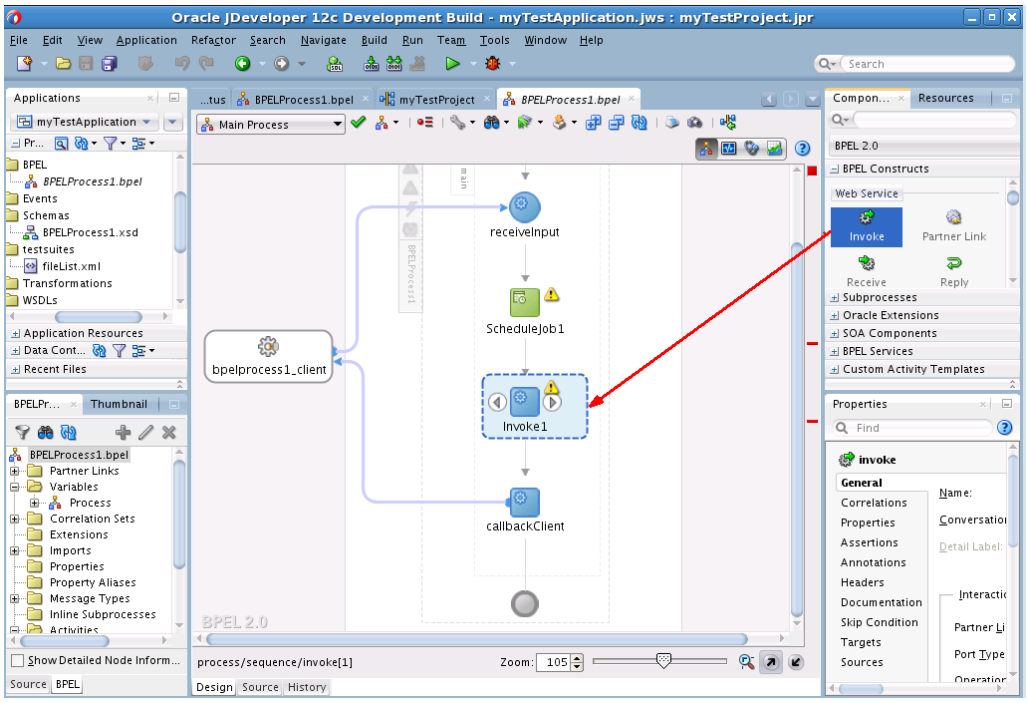


19. Add the **Invoke** activity for the `getCompletionStatus` operation.

    a. Click the **Design** tab to switch the display from the source view back to the design view.

    b. From the **BPEL Constructs** section of the **Component Palette**, drag and drop an **Invoke** component between **Schedulejob1** and **callbackClient** as shown in Figure 16-15.

Figure 16-15 Drag and Drop an Invoke Component Between Schedulejob1 and callbackClient



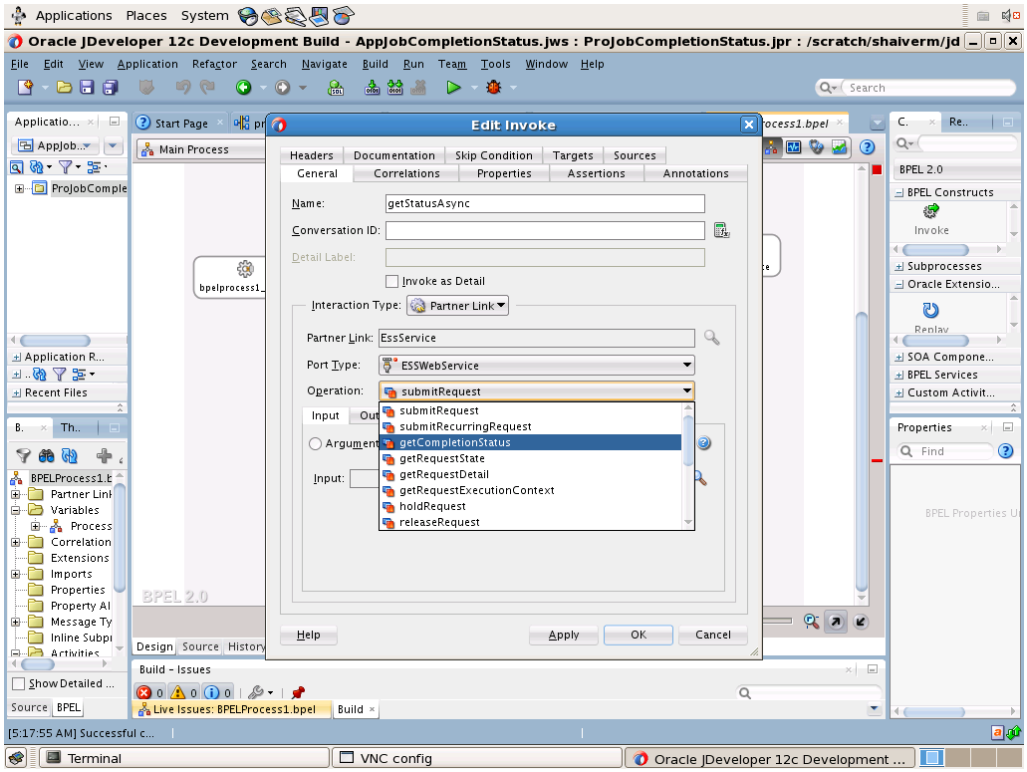
    c. Right-click the **Invoke1** button to open the Edit Invoke dialog. Rename the component **getStatusAsync**.

    d. Drag the arrow from the **getStatusAsync** component to the **EssService** component in the **Partner Links** area. The Edit Invoke dialog opens as shown in Figure 16-16.

e. From the Edit Invoke dialog **Operation** dropdown, select **getCompletionStatus** as shown in Figure 16-16.

f. Create an input variable named `x` and click **OK** to close the Edit Invoke dialog.

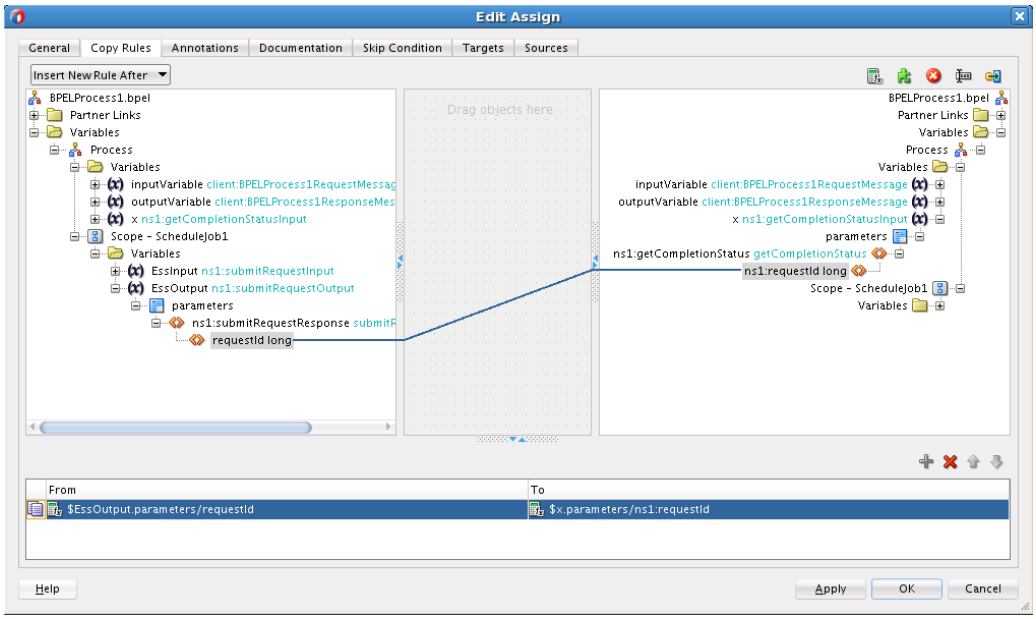
Figure 16-16 The Edit Invoke Dialog



g. Drag and drop an **Assign** component from the **BPEL Construct** area in the **Component Palette** to between the **ScheduleJob1** and the **getStatusAsync** component.

h. Double-click the **Assign** component to open the Edit Assign dialog and map the **ScheduleJob1** output parameter `requestID` to the **getStatusAsync** input parameter `requestID` as shown in Figure 16-17. Click **OK**.

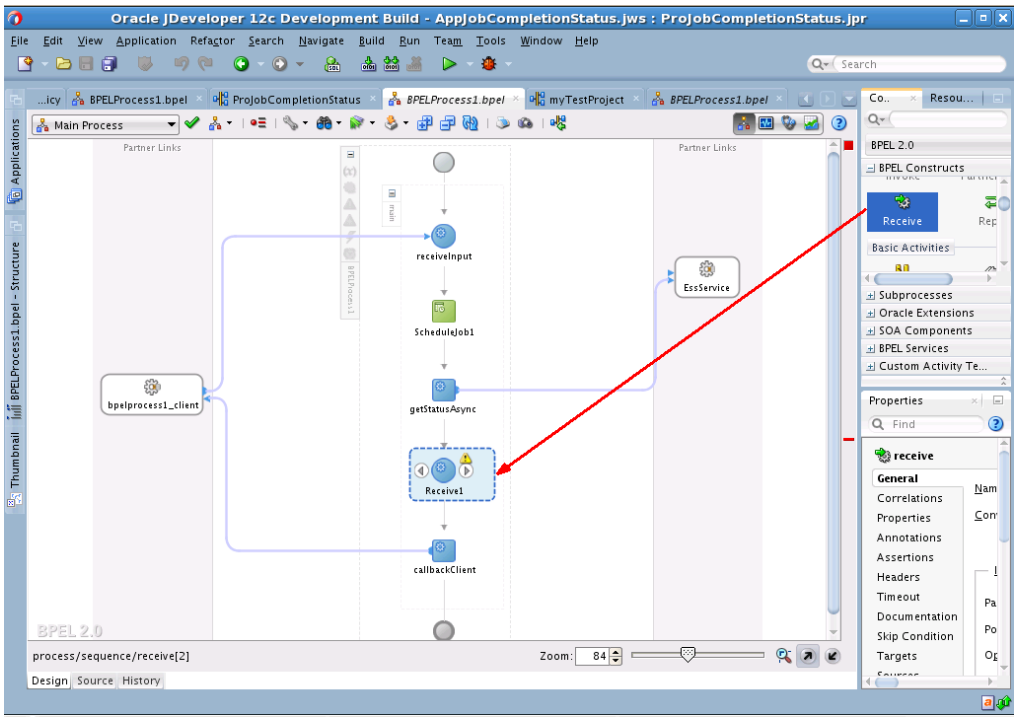
Figure 16-17 Use the Edit Assign Dialog to Map the ScheduleJob1 Output parameter requestID to the getStatusAsync Input Parameter requestID



20. Receive job completion status.

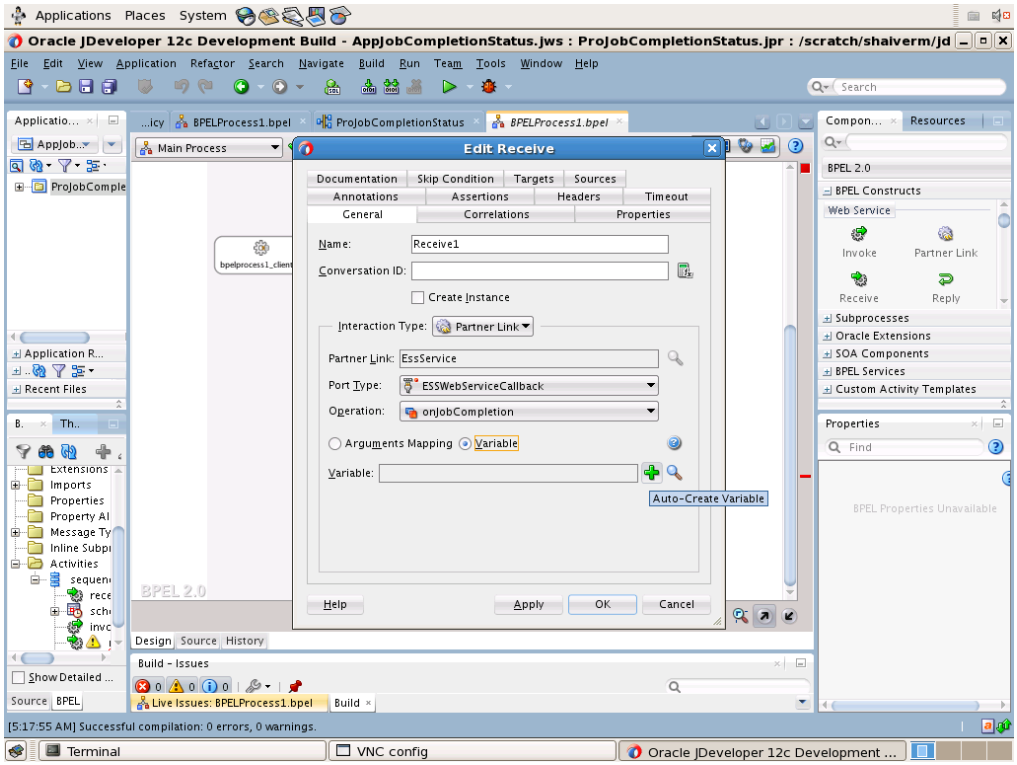
a. From the **BPEL Constructs** section of the **Component Palette**, drag and drop a **Receive** component between the **getStatusAsync** and the **callbackClient** components as shown in Figure 16-18.

Figure 16-18 Drag and Drop a Receive Component Between the getStatusAsync and the callbackClient Components



b. Attach the **Receive1** component to **ESSService** in **Partner Links** area by dragging the arrow from the **Receive1** component to the **ESSService** component. This action also opens the Edit Receive dialog. Rename the **Receive1** component to **OnJobCompletion** as shown in Figure 16-19.

Figure 16-19 Rename the Receive1 component to OnJobCompletion

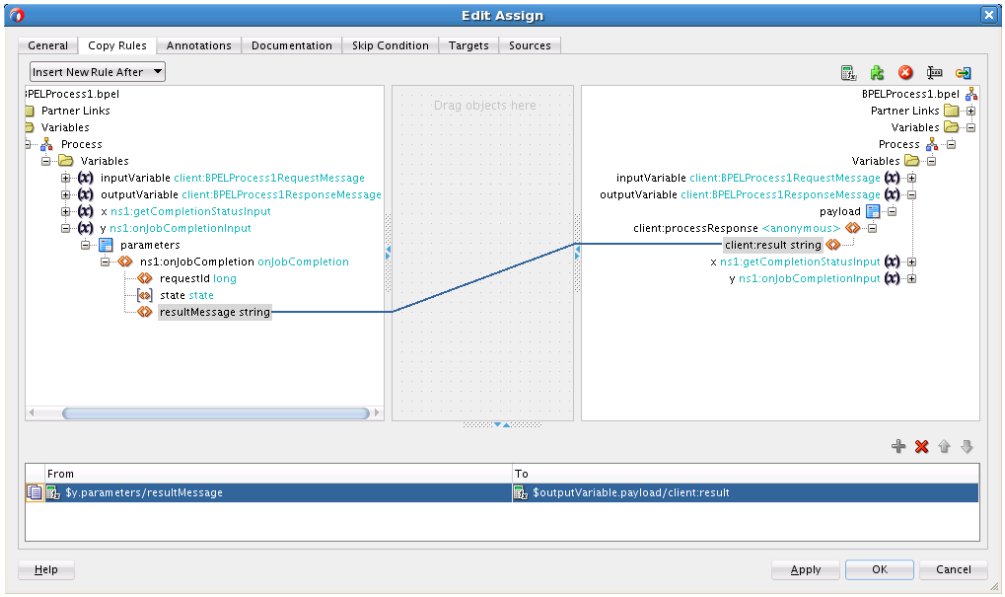


c. Select the **OnJobCompletion** operation and add a variable named `y` as shown in Figure 16-19. Click **OK** to close the Edit Receive dialog.

d. Drag and drop an **Assign** component from the **BPEL Construct** area in the **Component Palette** to between the **onJobCompletion** component and the **callbackClient** component.

e. Double-click the **Assign** component to open the Edit Assign dialog and map the **onJobCompletion** component's output parameter `resultMessage` to the **callbackClient** component's input parameter `result` variable as shown in Figure 16-20. Click **OK**.

Figure 16-20 Map the onJobCompletion Component's Output Parameter resultMessage to the callbackClient Component's input Parameter result Variable



17

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:

- Introduction to Defining and Using Job Sets
- Defining Job Sets
- Cross Application Job Sets
- Supporting Input and Output Forwarding in Job Sets

Introduction to Defining and Using Job Sets

Oracle Enterprise Scheduler provides for collections of job definitions that can be grouped together to run as a single unit called a *job set*. A job set may be nested; thus a job set may contain a collection of job definitions or one or more child job sets. Each job definition or job set included within a job set is called a *job set step*.

A job set is defined as either a *serial* job set or a *parallel* job set. At runtime, Oracle Enterprise Scheduler runs parallel job set steps together, in parallel. When a serial job set runs, Oracle Enterprise Scheduler runs the steps one after another in a specific sequence. Using a serial job set Oracle Enterprise Scheduler supports conditional branching between steps based on the execution status of a previous step.

You can define a serial job set to include a parallel job set, or a parallel job set to include a serial job set. Job sets that include a mix of parallel and serial job sets are called *complex job set*s. For example, when a serial job set contains a child parallel job set, the serial job set runs serially until it reaches the child parallel job set. Then, all the job definitions or job set definitions in the child parallel job set run in parallel. Upon completion of the child parallel job set the serial job set continues running its remaining steps serially. Nested parallel job sets behave the same as non-nested parallel job sets.

For every step in a job set Oracle Enterprise Scheduler supports a property (`SYS_selectState`) that provides runtime flexibility for how a particular step affects the entire job set. This property is defined on a per step basis. Table 17-1 describes SYS_selectState.

Table 17-1 Job Set Step Property

Property	Description
`SYS_selectState`	Specifies whether the result state of a job set step should be included when determining the state of the job set. Specifies whether the execution state of the step affects the eventual state of entire job set. By default, all job set steps affect the job set state. To prevent the state of a particular job set step from affecting the state of the job set, set `SELECT_STATE` to `false` for that step. To allow the state of a job set step to affect the overall state of the job set, set `SELECT_STATE` to `true` for that step.

Oracle Enterprise Scheduler provides the capability for a job set to execute across multiple applications. A job set runs in its hosting application and by default all job set steps also run in this application.

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.

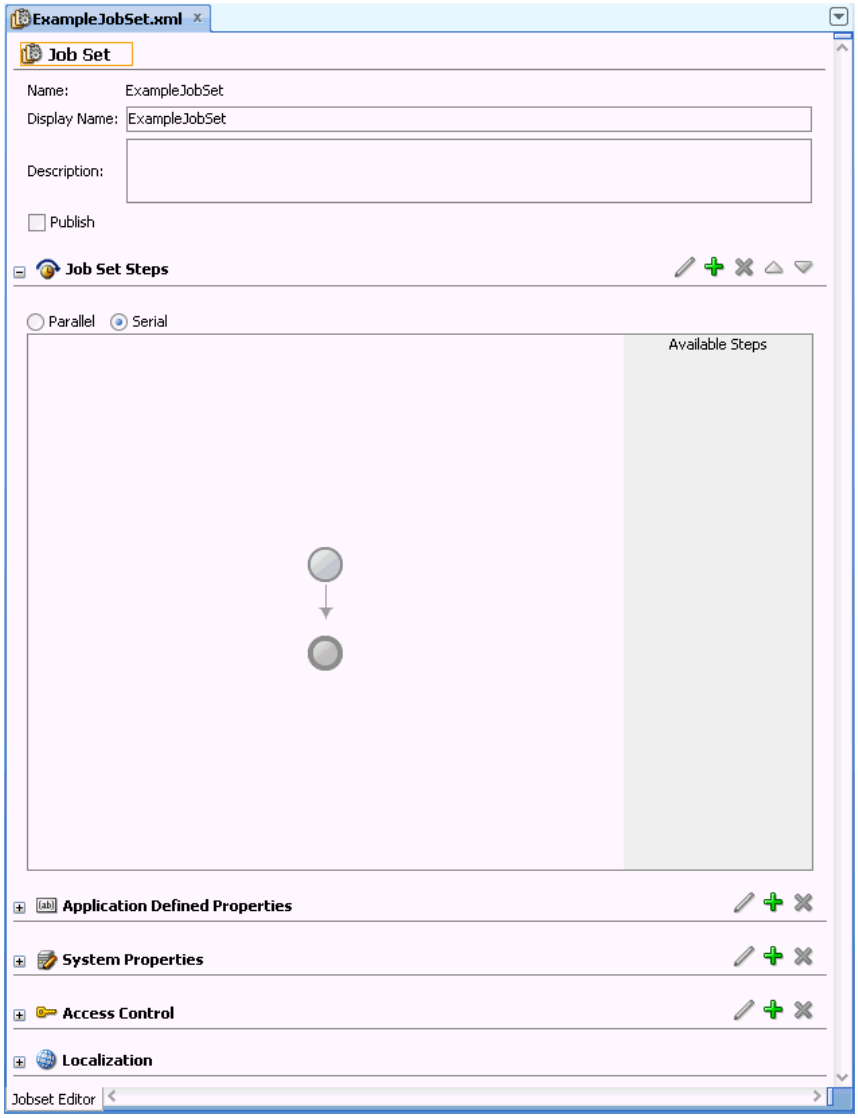
How to Define a Job Set

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, 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 17-1.

Figure 17-1 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 How to Define Serial Job Set Steps or 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 Introduction to Using Parameters and System Properties. For more information, see 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 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 Oracle Enterprise Scheduler Security .

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.

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 17-2 lists the possible terminal states that you can specify using JDeveloper.

Table 17-2 Job Set Serial Execution Step Terminal States

Terminal State	Description
`SUCCEEDED`	Oracle JDeveloper indicates this state with a check mark button. 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 button. A child step or child job set resulted in a warning.
`ERROR`	Oracle JDeveloper indicates this step with an error button. 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** button 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.

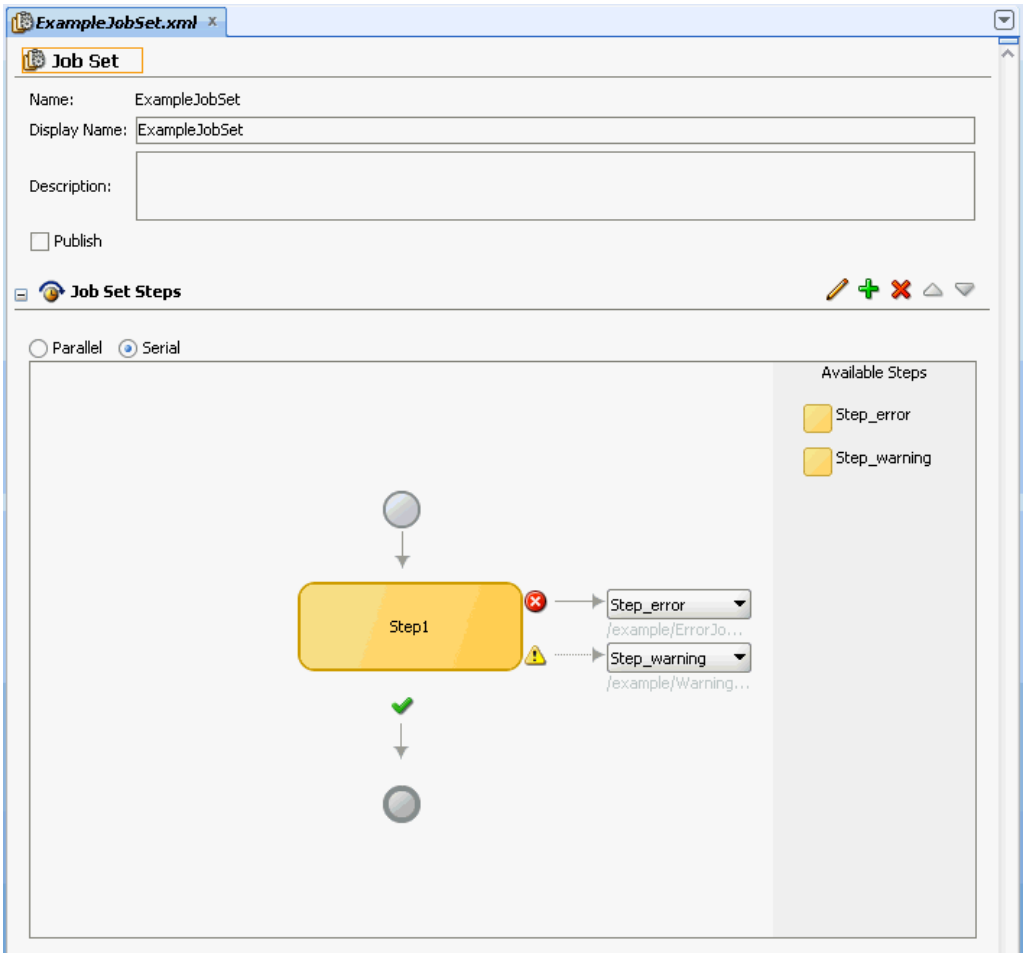
6. If you need to define step level application defined properties, then select the Application Defined Properties tab and add properties for the step.

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 17-2.

Figure 17-2 Job Set with a Step Added



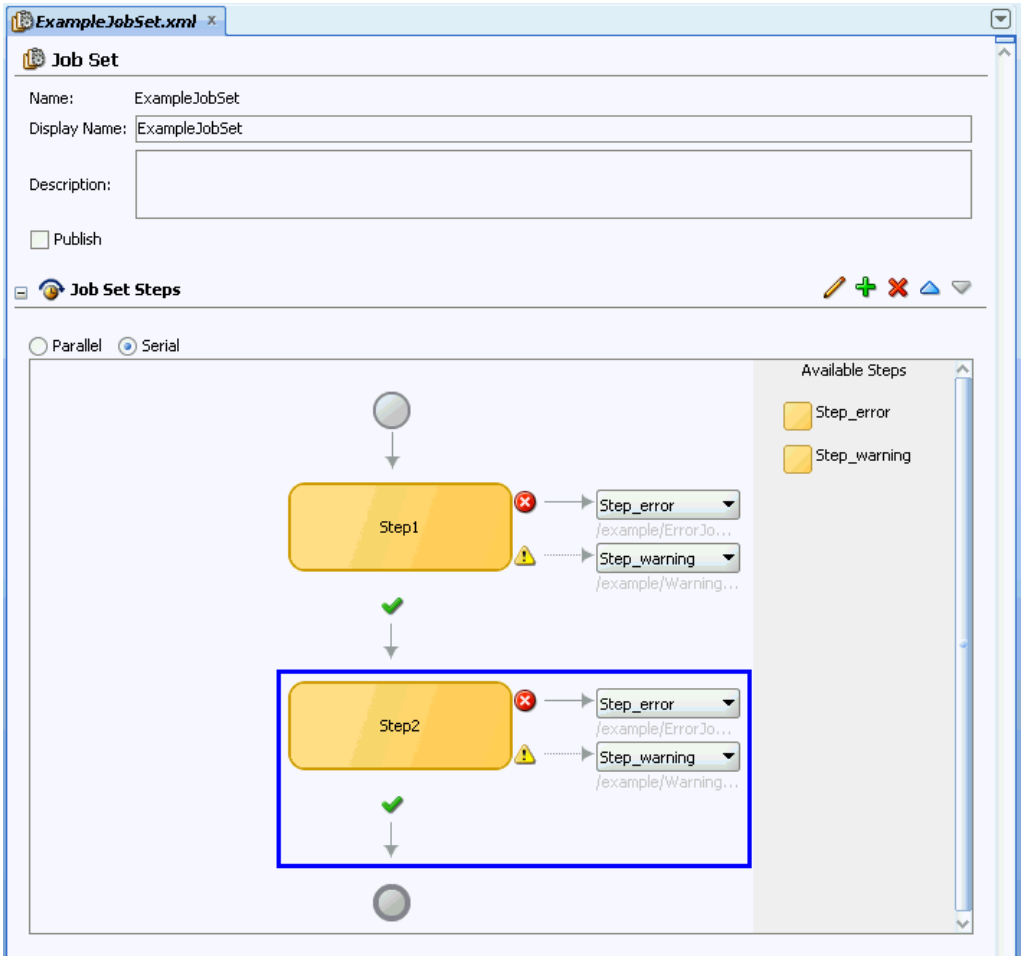
10. From the dropdown list next to the error icon, select Stop or select the step for the `ERROR` terminal state for the step. For example, from the dropdown list select **Step_error** (Step_error must be defined).

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 must be defined).

12. Click the **Add** button and add additional steps as needed.

13. Click **OK**, as shown in Figure 17-3.

Figure 17-3 Job Set with Two Steps Added



How to Define Parallel Job Set Steps

You can add parallel job set steps to a job set.

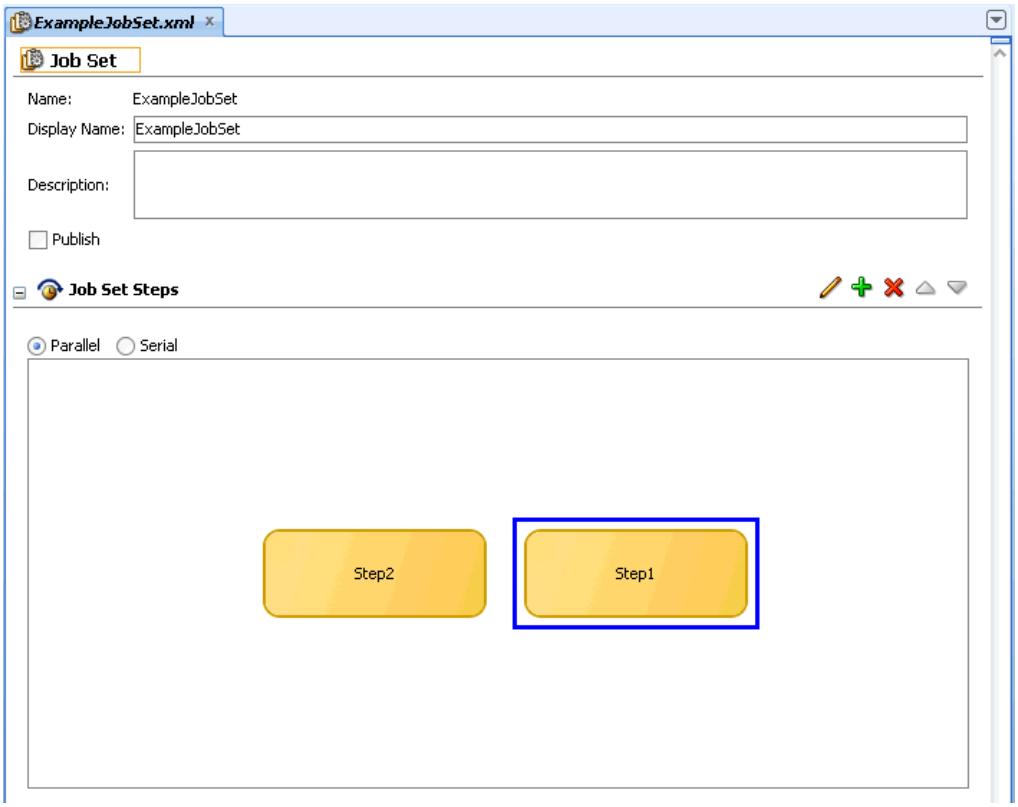
To add parallel job set steps:

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** button 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** button.
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 17-4.

Figure 17-4 Adding Job Set Steps to a Parallel Job Set



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.

What You Need to Know About Serial Job Sets

When you define a serial job set, the associated XML document includes job set steps and links. Oracle Enterprise Scheduler enforces the following limitations for serial job set definitions:

- To prevent looping within a job set, job set definitions should not contain circular execution paths. A circular execution path, or a loop, is defined at the job set level as follows: loop is a path from one job set step along the links of any number of

other steps back to the same job set step. For example, in a job set with a flow from `Job_A`, to `Job_B`, to `Job_C` defined, Oracle Enterprise Scheduler does not allow you to define an execution path from `Job_B` or `Job_C` back to `Job_A`. For example you could a create circular execution path, or a loop, if one of the links in a job set step for success, error, or warning links back to the same job set step. Thus, each job set step can link to any of the available job definitions or job sets, or they could all use the same job definition or job set as a link for the success, error and warning case. There is only a possible loop based on the path through the job set steps, as identified by the job set step ID. Oracle Enterprise Scheduler validates job sets at submission time to try to prevent job set step level looping. Also, Oracle JDeveloper does not allow you to create a job set containing a job set step level loop.

- To prevent looping within a job set, job set definitions should not contain self-referencing execution paths. For example, in a job set with `Job_B` defined, Oracle Enterprise Scheduler does not allow you to define an execution path from `Job_B` to `Job_B` itself if `Job_B` ends up with a terminal state of `ERROR`. However using the `RETRIES` property available for a job definition or a job set, you can have multiple executions up to the configured `RETRIES` number.
- When there is no job set link defined for a terminal state of a step, it implies that the job set should stop if the step ends with the unspecified terminal state. For example if there is no link defined for a step `Job_D` for the state `WARNING`, and if the step `Job_D` ends up with the state of `WARNING`, the job set stops execution.

Each job set step can be defined to use any of the available job definitions or job sets, and multiple steps may use the same job definition or job set.

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 Using Parameters with the Metadata Service and Using Parameters with the Runtime Service.

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 17-2. 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 17-3 shows the possible job set terminal states with the level indicated in the Priority column.

Table 17-3 Job Set Terminal State Transitions

Terminal State	Description	Priority
ERROR	If any step in a job set finishes with the terminal state of ERROR, the entire job set is marked with the terminal state of ERROR no matter what the state of the other steps. For serial job sets, if one step goes to ERROR, subsequent steps do not execute. For parallel job sets, all steps begin at the same time, and the job set state is not determined until the job set steps reach a terminal state.	The ERROR state has the highest priority.
WARNING	If any step in a job set ends up with the terminal state of WARNING, and there is no step with the terminal state of ERROR then the job set is marked with the terminal state WARNING. When the terminal state is WARNING, post processing begins.	Lower than ERROR
EXPIRED	The job set transitions to EXPIRED state if at least one of the child requests expires while there is no step that ends with the terminal state of ERROR or WARNING.	Lower than ERROR and WARNING
CANCELLED	Based on the actual outcome of a cancellation attempt, the job set can transition to CANCELLED if at least one child request successfully processes the cancellation attempt and transitions to CANCELLED state. The cancellation might have been requested on the entire job set or just a specific child request. Further the transition to CANCELLED follows the priorities of terminal states. Therefore the job set transitions to CANCELLED terminal state only if there is no step that ends with the state of ERROR, WARNING, or EXPIRED and there is at least one step with terminal state of CANCELLED. When a job set is canceled, steps that have not been added or run are considered to be CANCELLED for the purpose of final state.	Lower than ERROR, WARNING, and EXPIRED
SUCCEEDED	The job set is considered as SUCCEEDED if and only if all child requests completed with the terminal state of SUCCEEDED.	The SUCCEEDED state has the lowest priority among all terminal states

Table 17-4 lists additional possible states for a job set:

Table 17-4 Possible Job Set Runtime States

State	Description
WAIT	This is the initial state of the submitted job set request. After 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 go from WAIT to READY to RUNNING. This is true for all job set steps, whether the step is a job definition or nested job set.
RUNNING	The submitted job set transitions from WAIT to READY to RUNNING. Nested job sets start in READY and transition to RUNNING.

State	Description
CANCELLING	A job set transitions to CANCELLING when the user requests a cancellation for the entire job set. This can be done by calling cancelRequest() with the request ID of the parent request representing the job set. Passing the parent request ID indicates that the user wants to cancel entire job set irrespective of its current, non-terminal, state and the states of its child requests. In such cases, a cancellation is attempted on all child requests that are still active and have not already transitioned to a terminal state. On the other hand if cancellation is attempted only on a specific child request in the job set, there won't be any state change for the parent request and only the particular child request transitions to CANCELLING if possible. If the cancel happens during post-processing, the state is set to WARNING rather than CANCELLED. If the job set finishes before the cancel is issued, the job set can have state SUCCEEDED.
COMPLETED	This state indicates that the job set or job set step has finished executing and post-processing begins.
BLOCKED	The BLOCKED state is not a terminal state. However any request can remain in a BLOCKED state for a long period until the blocking condition is eliminated (such as incompatibility). In the case of a job set, any individual step might be BLOCKED while other steps either complete or may be running. The job set itself, however, remains in a RUNNING state. Eventually if all steps in the job set complete except the ones that are in the BLOCKED state, the job set cannot continue further until the blocking step is ready to run. When the blocked step unblocks and completes, the job set can proceed. After the steps complete, the job set eventually goes to the appropriate terminal state. For a serial job set, the job set may stop at a step that is in BLOCKED state. In such cases, all previous steps are complete and the job set cannot continue until the blocked step executes. However for a parallel job set, multiple steps can remain in BLOCKED state. Further, while some steps are blocked, other steps can still continue to run.
HOLD	The HOLD state is very similar to the BLOCKED state. Following the same rules for the BLOCKED state, a job set cannot continue running while a step is in HOLD state. A serial job set cannot continue if the current step in the execution flow is stuck at HOLD state. In the case of a parallel job set, if at least one step is stuck in HOLD state while all other steps have completed, the job set can complete when the step is no longer in HOLD state.

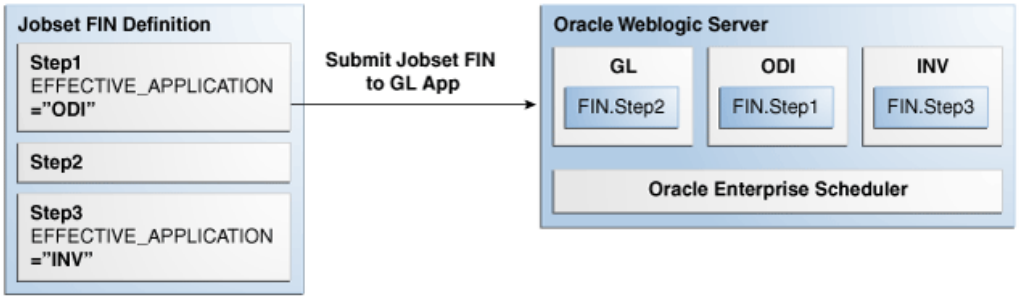
Cross Application Job Sets

Oracle Enterprise Scheduler provides the capability for a job or a job set to execute across multiple applications as shown in Figure 17-5:

- 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 17-5 Cross Application Job Set Steps



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. The system property `SYS_effectiveApplication` should be defined on the job definition or job set (rather than the job set step). For a nested job set that defines `SYS_effectiveApplication`, the application applies to any child requests of that nested job set. If it is a nested job set, the jobs in the nested job set execute in the effective application. When `SYS_effectiveApplication` is defined for a job, the request for the job set and any child requests of the job set are associated with the effective application, meaning the `APPLICATION` system property for those requests is set to the effective application.

The `SYS_effectiveApplication` system property may only be defined in metadata, specifically job set, job set step, job type, and job. The property `SYS_effectiveApplication` is not supported in the request parameters. The effective application must be in the same cluster as the hosting application, or an error results. If a submitted job set defines the effective application, that value must be the same as the hosting application, or the job set submission is 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.

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.

Supporting Input and Output Forwarding in Job Sets

Sometimes a step in a job set requires input from the previous step in the job set. Oracle Enterprise Scheduler uses two system properties `SYS_inputList` and `SYS_outputList` 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 must be passed on to the next step in a job set, the job adds the information to the `SYS_outputList` property. Upon completion of the job request execution, Oracle Enterprise Scheduler forwards the `SYS_outputList` property of the request so that it becomes the `SYS_inputList` 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 `SYS_outputList` 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 `SYS_inputList` property set to the value of the `SYS_outputList` 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 `SYS_outputList` 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 `SYS_outputList` for that job set has 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 `SYS_outputList` property of a step to the value of the `SYS_inputList` 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.

18

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:

- Introduction to Using a Job Incompatibility
- Defining Incompatibility with Oracle JDeveloper
- What Happens at Runtime to Handle Job Incompatibility

For information about how to create and submit job requests see t Creating and Using PL/SQL Jobs , and Creating and Using Process Jobs . For more information on using job sets, see 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.

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 one or more 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. 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. Parameters specified through `parameterVO` are submitted as request properties having the property name `submit.argument1`, ... `submit.argument#`. To use such a parameter as the domain incompatibility property, specify `submit.argument1`, ... `submit.argument#` for the incompatibility property name.
- **Global** (`GLOBAL`): where one or more 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 one or more entities (job definition or job set) and the resource over which they must be incompatible. A resource is not specified for a global incompatibility. Each entity can be flagged as being self-incompatible. If an incompatibility is defined for only one entity that entity must be flagged as 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 of a request when it is about to executed for the first time. It also determines the resource (property) value for any domain incompatibility at that time. That information is used throughout the subsequent processing life cycle of the request, including any retries of the request. For job set requests, Oracle Enterprise Scheduler determines which, if any, incompatibility definitions reference the job definition of any potential job set step when the top-most job set request is about to be executed rather than when individual step requests are executed.

For a Schedule-based submission, Oracle Enterprise Scheduler creates new child requests for instances of the Schedule. Only one instance request is executed at a given time. Oracle Enterprise Scheduler tracks metadata changes made to incompatibility definitions and may refresh the incompatibility definitions, if any, when an instance request is about to be executed for the first time. This means the incompatibility definitions used when the next instance request is executed may be different than the incompatibility definitions used when a prior instance request was executed.

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.

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

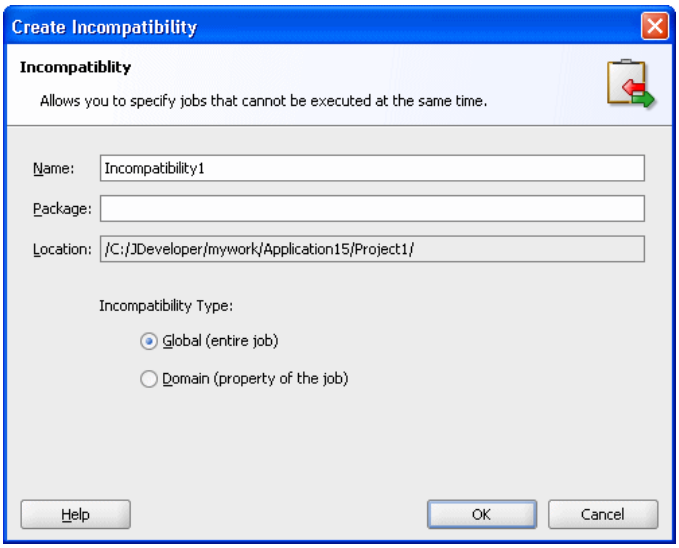
How to Define a Global Incompatibility

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 18-1.

Figure 18-1 Create Incompatibility Window

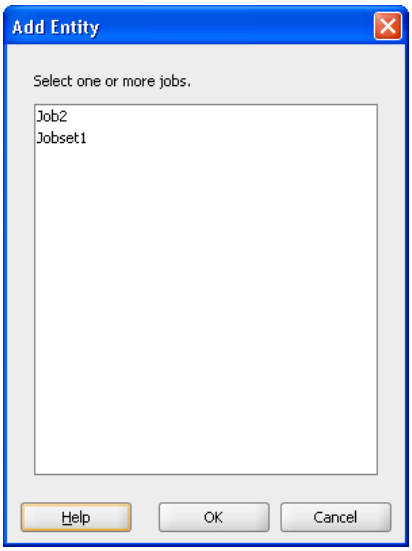


4. Use the Create Incompatibility dialog to specify the following:
   a. In the **Name** field, enter a name for the incompatibility or accept the default name.
   b. In the **Package** field, enter a package name for the incompatibility.
   c. The **Location** field displays the full path of the directory 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 18-2.

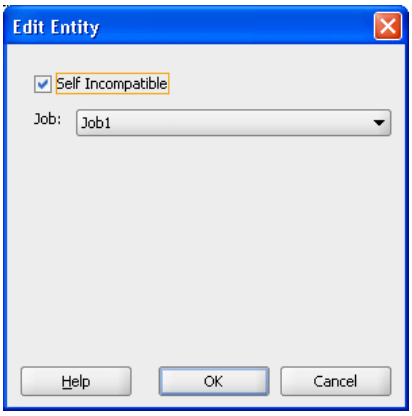
Figure 18-2 Incompatibility Add Entity Window



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 18-3.

Figure 18-3 Edit Entity Window for Global Incompatibility



9. To specify self incompatibility, select **Self Incompatible**.

10. **Save** the incompatibility.

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:

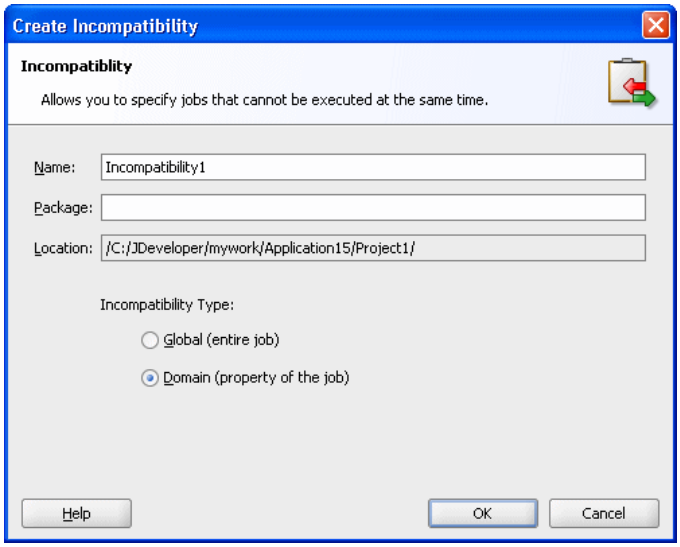
   a. In the **Name** field, enter a name for the incompatibility or accept the default name.

   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 18-4.

   Figure 18-4 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.

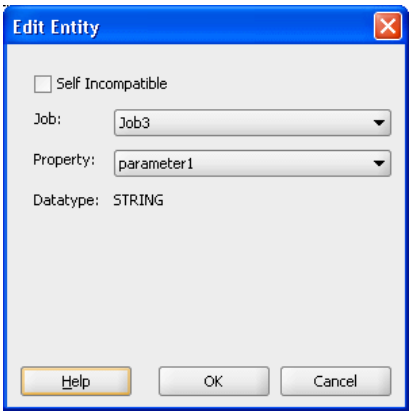
7. Select one or more jobs or job sets to add to the incompatibility and click **OK**.

   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 18-5.

Figure 18-5 Incompatibility Edit Entity Window



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

What Happens at Runtime to Handle Job Incompatibility

Oracle Enterprise Scheduler handles incompatibility definitions according to the incompatibility type, global or domain (property-based), at runtime. When a job request is about to be executed, 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 or property, to use for that incompatibility. Oracle Enterprise Scheduler checks to determine if there are any incompatible requests already executing. If so, the request is blocked until all requests with which it is incompatible have completed.

Note:

The value of the property for a domain incompatibility is obtained from the request parameters at request execution. That value usually originates either in the job definition or in a request parameter specified at request submission. If no such parameter is found, that incompatibility is ignored during subsequent request processing. The request is compatible with any other request with regard to that incompatibility definition. This initial property value is used as the incompatibility resource value even if the property is subsequently altered.

What Happens to Subrequests with an Incompatible Parent Request

A request which is incompatible with another request is also incompatible with the sub-requests of that request (the children). A request that has been blocked by a sub-request parent remains blocked while any sub-requests execute and until the sub-request parent request is resumed and completes.

19

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.

Note:

The runtime service also includes log and output APIs. These APIs are document separately in Job Request Logs and Output .

This chapter includes the following sections:

- Introduction to the Runtime Service
- Accessing the Runtime Service
- Submitting Job Requests
- Managing Job Requests
- Querying Job Requests
- Submitting Ad Hoc Job Requests
- Implementing Pre-Process and Post-Process Handlers

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.

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`.

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 19-1 shows a lookup for the Oracle Enterprise Scheduler runtime service using the `RuntimeServiceLocalHome` object.

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`.

Example 19-1 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);
        }
    }
```

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 Submitting Ad Hoc Job Requests).

How to Submit a Request to the Runtime Service

You create a job request by calling `submitRequest()`. Depending on your requirements, 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 19-2 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 Submitting Ad Hoc Job Requests. You can also submit a sub-request. For more information, see Using Subrequests .

Note:

When you submit a job request using the runtime service:

- You obtain the runtime service handle as shown in Example 19-1.
- The runtime service internally uses the metadata service to obtain job definition metadata with the supplied `MetadataObjectId`, `jobDefnId`.

Example 19-2 Creating a Job Request with submitRequest()

```
long requestID = 0L;
MetadataObjectId jobDefnId;

RequestParameters p = new RequestParameters();

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

requestID = runtime.submitRequest(r, "My Java job", jobDefnId, start, p);
```

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. Certain system properties can be associated with a job request. If you do not set these properties anywhere in the properties

hierarchy when a job request is submitted, then Oracle Enterprise Scheduler provides default values.

Table 19-1 shows the default runtime service field names and the corresponding system properties.

Table 19-1 Runtime Service Default Value Fields and Corresponding System Properties

Value	Runtime Service Default Value Field	Corresponding System Property	Description
0	`DEFAULT_REQUEST_EXPIRATION`	`SYS_requestExpiration`	The default expiration time, in minutes, for a request. The default value is 0 which means the request never expires.
4	`DEFAULT_PRIORITY`	`SYS_priority`	The default system priority associated with a request.
5	`DEFAULT_REPROCESS_DELAY`	`SYS_reprocessDelay`	The default period, in minutes, in which processing must be postponed by a callout handler that returns `Action.DELAY`.
0	`DEFAULT_RETRIES`	`SYS_retries`	The default number of times a failed request is retried. The default value is 0 which means a failed request is not retried.
0	`DEFAULT_ASYNC_REQUEST_TIMEOUT`	`SYS_request_timeout`	Specifies the time in minutes that the processor waits for an asynchronous response after the job execution has begun. After the time elapses, the request is timed out.

What You Should Know About Metadata When You Submit a Request

All Oracle Enterprise Scheduler Metadata associated with a job request is persisted in the runtime store at the time of request submission. Persisted metadata objects include job definition, job type, job set, schedule, incompatibility definitions, and exclusion definition. Metadata is stored in the context of a top level request, and each metadata object is uniquely identified by the absolute parent request ID and its metadata ID. Each unique metadata object is stored only once for a top-level request, even if the definition is used multiple times in the request. This ensures that every child request uses the same definition.

When a request is submitted, all known metadata for the request is persisted. For subrequests, the metadata is not know until the subrequest is submitted, so subrequest metadata is persisted when the subrequest is submitted, after first checking that the metadata object is not already persisted in the runtime store.

Metadata persisted in the runtime store is removed when the absolute parent request is deleted.

DMS ECID and FlowId Support

Oracle Enterprise Scheduler associates a DMS ECID and `FlowId` value with every request. Oracle Enterprise Scheduler usually obtains the ECID and `FlowId` from the current DMS execution context, if present, at request submission and uses that ECID and `FlowId` value during subsequent processing of the request. For example, Oracle

Enterprise Scheduler sets up a DMS execution context that associates the ECID and `FlowId` with the request when it initiates the job executable.

If a DMS `FlowId` property is not present in the DMS execution context at request submission, then a new `FlowId` is associated with the request. For example, if the request is not submitted by SOA, there might not be a `FlowId` present on the DMS execution context and Oracle Enterprise Scheduler associates a new `FlowId` with the request.

If no DMS execution context is present at request submission, then a new ECID and new `FlowId` are associated with the request. For example, if a request is submitted using the Oracle Enterprise Scheduler PL/SQL interface, there might be no DMS context information available from the database session when the PL/SQL submit procedure is called. A new ECID and new `FlowId` are associated with the request after it is successfully validated by the Oracle Enterprise Scheduler mid-tier.

ECID and FlowID for Child Requests

In general, child requests inherit the ECID and `FlowId` from their parent request. For example, Oracle Enterprise Scheduler uses the ECID and `FlowId` of the parent request when a job set step request is created.

A sub-request is a submitted request, therefore the ECID and `FlowId` of the current DMS execution context of the sub-request submission is associated with the sub-request. Usually the ECID and `FlowId` values are the same as those of the parent request because Oracle Enterprise Scheduler sets up a DMS execution context that has the ECID and `FlowId` of the parent request prior to initiating the parent job executable. It is possible that the application or some component layer changed the ECID or `FlowId` prior to Oracle Enterprise Scheduler receiving the sub-request submission. If that is the case, the parent and sub-request might have a different ECID or `FlowId`.

If a schedule is specified at request submission, the submitted request represents an absolute parent that does not execute. Oracle Enterprise Scheduler automatically creates child instance requests according to the specified schedule and a new ECID and `FlowId` is used for each child instance. The child instance request represents an instance parent request and may have children of its own; for example, a sub-request or job set step request. Any such children typically have the same ECID and `FlowId` as its instance parent request.

DMS FlowId and SOA CorrelationFlowId

Oracle Enterprise Scheduler uses the DMS FlowId property whose property name is "FlowId". SOA has several properties that might be present on a DMS execution context. Two such properties are the SOA `CorrelationFlowId` and SOA `FlowId`. properties. The DMS `FlowId` property ("FlowId") is used to propagate the value for the SOA `CorrelationFlowId`. The DMS property name "oracle.soa.tracking.FlowId" is used to propagate the value for the SOA `FlowId` property. For that reason, the `FlowId` property associated with an Oracle Enterprise Scheduler request submitted by SOA might match the SOA `CorrelationFlowId` value.

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

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 19-2 shows the runtime service methods that allow you to obtain job request information.

Table 19-2 Runtime Service Get Request Methods

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

Example 19-3shows code that determines if there is any immediate child request in the HOLD state.

Example 19-3 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;
        }
    }
```

How to Change Job Request State

Using the runtime service, with a `requestID`, you can change the state of a job request. Table 19-3 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 19-3 Runtime Service Job Request State Methods

Runtime Service Method	Description
`cancelRequest()`	Cancels the processing of a request that is not in a terminal state.
`deleteRequest()`	Marks a request in a terminal state for deletion.
`holdRequest()`	Withholds further processing of a request that is in `WAIT` or `READY` state.
`releaseRequest()`	Releases a request from the `HOLD` state.

Example 19-4 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.

Note:

Note the following in Example 19-4:

- You obtain the runtime service handle, `rHandle`, as shown in Example 19-1.
- 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.

Example 19-4 Runtime Service releaseRequest() Usage

```
rHandle = runtime.open();
try
{
            runtime.holdRequest(rHandle,reqid);
.
.
.
            runtime.releaseRequest(rHandle, reqid);
.
.
.
}
finally
{
        if (rHandle != null)
        {
            runtime.close(rHandle);
        }
 }
```

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 19-4 shows the runtime service methods that allow you to lock and update up a job request.

Table 19-4 Runtime Service Update Methods

Runtime Service Method	Description
`lockRequest()`	Acquires a lock for the given request. The lock is released when `close()` operation is subsequently invoked or the encompassing transaction is committed. If an application tries to invoke this operation while the lock is being held by another thread, this method blocks until the lock is released. Use this method to ensure data consistency when updating request parameters or system properties.
`setRequestParameter()`	Updates the property value of the specified request subject to the property read-only constraints.

Example 19-5 shows code that updates a job request parameter. This code would be wrapped in a try/finally block as shown in Example 19-1.

Example 19-5 shows the following:

- Obtain the runtime service handle, `rhandle`, as shown in Example 19-1.
- Acquire a lock for either the request using `lockRequest()`
- Perform the update operation with `setRequestParameter()`
- 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`.

Example 19-5 Sample Runtime Service Parameter Update

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

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 includes 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 How to Create a Filter.

When you create a filter for a query, you can use any of the field names shown in Table 19-5 when querying the runtime store.

Table 19-5 Query Filter Fields For Querying the Runtime (Defined in Enum RuntimeService.QueryField)

Name	Description
ABSPARENTID	The absolute parent request ID of a request.
APPLICATION	The application name.
ASYNCHRONOUS	Indicates if the job is asynchronous, synchronous or unknown. The value of the field is not set until the request is processed. The field data type is `java.lang.Boolean`. The value may be `NULL` if the nature of the job has not yet been determined.
CLASSNAME	The name of the executable class that processed the request
COMPLETED_TIME	The date and time that Oracle Enterprise Scheduler finished processing the request. This field represents the time the process phase was set to COMPLETED.
DEFINITION	The job definition ID (Metadata Object ID).
ELAPSEDTIME	The amount of time, in milliseconds, that elapsed while the request was running.
ENTERPRISE_ID	The enterprise ID.
ERROR_TYPE	The request error type.
EXTERNAL_ID	The identifier for an external portion of an Oracle Enterprise Scheduler asynchronous Java job.
EXTERNAL_JOB_TYPE	Indicates the type of the remote job
INSTANCEPARENTID	The request ID of the instance parent request.
JOB_TYPE	The job type ID (Metadata Object ID).
LOGICAL_CLUSTER_NAME	Indicates the logical cluster on which a remote job is executed.
NAME	The request description.
PARENTREQUESTID	The parent request ID.
PRIORITY	The priority of the request.
PROCESS_PHASE	The process phase of the request.
PROCESSEND	The date and time that the process ended. The `PROCESSSTART` is set only when a request transitions from `READY` to `RUNNING`. This implies that (`PROCESSEND` - `PROCESSSTART`) encompasses the entire span of execution: from the time the state becomes `RUNNING` to the time it transitions to a terminal state.
PROCESSOR	The name of the instance that processed the request.

Name	Description
PROCESSSTART	The date and time that the process started. The PROCESSSTART is set only when a request transitions from READY to RUNNING. This implies that (PROCESSEND - PROCESSSTART) encompasses the entire span of execution: from the time the state becomes RUNNING to the time it transitions to a terminal state.
PRODUCT	The product name.
READYWAIT_TIME	The amount of time, in milliseconds, a request has been waiting to run since it became READY.
REQUEST_CATEGORY	The request category specified for the request.
REQUEST_DMS_ECID	The DMS ECID used for processing of a request.
REQUESTEDEND	The requested end time.
REQUESTEDSTART	The requested start time.
REQUESTID	The request ID of a submitted request.
REQUESTTYPE	The type of request (that is, an element of RequestType)
RESULTINDEX	Controls the starting and ending index of the returned results. This field allows users to express result constraints such as "return only results 10 through 20".
RETRIED_COUNT	The retried count associated with a job. This field represents the number of times the job was retried.
REQUESTTRIGGER	The Trigger ID (Metadata Object ID).
SCHEDULE	The schedule ID (Metadata Object ID).
SCHEDULED	The time when the request is scheduled to be executed.
STATE	The job request state.
SUBMISSION	The submission time of the request.
SUBMITTER	The submitter of the request.
SUBMITTER_DMS_ECID	The DMS ECID from the DMS context at request submission.
SUBMITTER_FLOWID	The SOA/DMS FlowId from the DMS context at request submission.
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 19-6 shows the runtime service method for querying job requests and Example 19-6 shows the use of this method.

Table 19-6 Runtime Service Query Methods

Runtime Query Method	Description
`queryRequests()`	Gets a summary of requests.

Example 19-6 Using queryRequest() Method

```
Filter filter =
     new Filter(RuntimeService.QueryField.DEFINITION.fieldName(),
                Filter.Comparator.EQUALS,
                myJavaSucJobDef.toString())
     .and(RuntimeService.QueryField.STATE.fieldName(),
          Filter.Comparator.EQUALS,
          new Integer(12) );

 //
 Enumeration requests =
     runtime.queryRequests(h, filter,
                RuntimeService.QueryField.REQUESTID, true);
```

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.

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 19-7.

Table 19-7 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 19-7 shows sample code for an ad hoc request submission that uses a schedule.

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 19-7.
- 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.

Example 19-7 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 5 seconds.", recur);

            // adhoc submission, no metadata definitions passed
            reqId = runtime.submitRequest(h,
                                  "testAdhocJavaWithSchedule",
                                JobType.ExecutionType.JAVA_TYPE,
schedule,                                    null, Calendar.getInstance(),
null, p);
```

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 `setRequestParameter()` or `getRequestDetail()` as you would with any other job request.

There is only one `submitRequest` signature that creates 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.

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.

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.

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.

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 causes 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 remains in the RUNNING state.
- `WARN` informs Oracle Enterprise Scheduler that request processing should commence but that a warning should be logged. The request remains in the RUNNING state.
- `CANCEL` informs Oracle Enterprise Scheduler that request pre-processing has been canceled. The request transitions to the 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 transitions 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 transitions to an error state not subject to retries.

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.

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 causes the request to transition to WARNING state.

- `PROCEED` to inform Oracle Enterprise Scheduler that request post-processing completed successfully. The request transitions to the 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 transitions to WARNING state.
- `CANCEL` informs Oracle Enterprise Scheduler that request post-processing has been canceled. The request transitions 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 transitions to the WARNING state.
- `BIZ_ERROR` to inform Oracle Enterprise Scheduler that the handler has experienced a business error. The request transitions to the WARNING state.

20

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.

This chapter includes the following sections:

- Introduction to Using Subrequests
- Creating and Managing Subrequests
- Creating a Java Procedure that Submits a Subrequest
- Creating a PL/SQL Procedure that Submits a Subrequest

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.

After 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. After 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.

Creating and Managing Subrequests

- How to Submit Subrequests
- How to Cancel Subrequests
- How to Hold Subrequests
- How to Submit Multiple Subrequests
- How to Manage Paused Subrequests
- How Subrequests Are Processed
- How to Identify Subrequests
- How to Manage Subrequests and Incompatibility

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 does not process the request while the parent request is running. A subrequest can be processed only after the parent request has paused.

How to Cancel Subrequests

There are two main ways a subrequest can be canceled, either by the user cancelling the subrequest directly or as a result of the parent request being canceled. 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 after 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 canceled, 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. After 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.

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.

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 is 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.

How to Manage Paused Subrequests

- Indicating Paused Status
- Storing the Paused State for a Parent Request

Indicating Paused Status

A Java executable can submit subrequests using `RuntimeService.submitRequest`. After the subrequest has been submitted, the parent request must indicate to Oracle Enterprise Scheduler that it is pausing to allow the subrequest to be processed. This is accomplished by the parent throwing an `ExecutionPausedExcpetion` which causes the request to transition to PAUSED state. After the subrequests have completed, the parent request is run again as a resumed request. The `RequestExecutionContext` can be used to determine if the executable is being run as a resumed request.

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()`.

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 is not 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. After 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. After Oracle Enterprise Scheduler picks up the parent request from the ready queue, the request state is set to `RUNNING` and the request executable called as a resumed request.

If a request is paused without submitting any subrequests, it is treated as if all subrequests had finished. That is, it is 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 after 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 canceled.

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 using 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.

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.

Creating a Java Procedure that Submits a Subrequest

This is an example of the Java class for a Java job type that submits subrequests. The procedure submits two subrequests, pausing between each one. Each subrequest uses the same `JobDefinition` but specifies a different value for the request parameter named `SubRequestData`. The `oracle.as.scheduler.Executable.execute` method of the parent request is called a total of three times for a given Oracle Enterprise Scheduler request and the following summaries the expected conditions and actions for each.

In the first call to execute method as a non-resumed request:

Entry condition:

- `RequestExecutionContext.isResumed()` is `false`
- `RequestExecutionContext.getPausedState()` is `null`

Method Action:

- Submit a subrequest with request parameter value of 'MyData1'

- Throw `ExecutionPausedException` with `pausedState` of 'MyPausedState1"

Oracle Enterprise Scheduler transitions the request to `PAUSED` state, execute the subrequest, and then resume the request after the subrequest has completed.

First call to execute method as resumed request:

Entry condition:

- `RequestExecutionContext.isResumed()` is `true`
- `RequestExecutionContext.getPausedState()` is 'MyPausedState1'

Method Action:

- Submit a subrequest with request parameter value of 'MyData2'
- Throw `ExecutionPausedException` with `pausedState` of 'MyPausedState2"

Oracle Enterprise Scheduler transitions the request to `PAUSED` state, execute the subrequest, and then resume the request after the subrequest has completed.

Second call to execute method as resumed request:

Entry condition:

- `RequestExecutionContext.isResumed()` is `true`
- `RequestExecutionContext.getPausedState()` is 'MyPausedState2'

Method Action:

- Exit normally, no exception.

Oracle Enterprise Scheduler transitions the request to `SUCCEEDED` state.

Example 20-1 shows a Java procedure with a subrequest.

Example 20-1 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
    }

}
```

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 is 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 does not set parent request to `PAUSED` state, but instead, it the parent state is set to `ERROR` or `WARNING` depending on the SQL error code. Furthermore, the subrequests are automatically `CANCELLED` and are not executed.

After the subrequest has finished, PL/SQL procedure for the parent request is 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 using 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 using the `ess_runtime.mark_paused` procedure when the request was paused.

This is an example of the PL/SQL stored procedure for a SQL job type that submits subrequests using the `ESS_RUNTIME` package. The procedure submits two subrequests, pausing between each one. Each subrequest uses the same `JobDefinition` but specifies a different value for the request parameter named `SubRequestData`. The PL/SQL stored procedure would be called a total of three times for a given Oracle Enterprise Scheduler request and the following summaries the expected conditions and actions for each.

First call to procedure as non-resumed request:

Entry condition:

- `ess_runtime.is_resumed` is `false`
- `ess_runtime.get_paused_state` is `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 transitions the request to `PAUSED` state, execute the subrequest, and then resume the request after the subrequest has completed.

First call to procedure as resumed request:

Entry condition:

- `ess_runtime.is_resumed` is `true`
- `ess_runtime.get_paused_state` is '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 transitions the request to `PAUSED` state, execute the subrequest, and then resume the request after the subrequest has completed.

Second call to procedure as resumed request:

Entry condition:

- `ess_runtime.is_resumed` is `true`
- `ess_runtime.get_paused_state` is 'MyPausedState2'

Procedure Action:

- Exit normally, no exception.

Oracle Enterprise Scheduler transitions the request to `SUCCEEDED` state.

Example 20-2 shows a PL/SQL procedure with a subrequest.

Example 20-2 PL/SQL Procedure with Subrequest

```
-- Application stored procedure.
procedure plsql_subreq_sample
( request_handle in varchar2 )
is
   v_reqid  number;
   v_is_resumed  boolean;
   v_paused_str  varchar2(100);
   v_paused_state1  varchar2(100) := 'MyPausedState1';
   v_paused_state2  varchar2(100) := 'MyPausedState2';
begin
   -- Request id of this subrequest parent.
   v_reqid := ess_runtime.get_request_id(request_handle);

   -- Check is this is a resumed request
   v_is_resumed := ess_runtime.is_resumed(request_handle);

   if (v_is_resumed = false) then
      -- This parent request is being run for the first time.
      -- Submit a subrequest and exit as success to allow ESS to pause this
parent.
      submit_subrequest(request_handle, v_paused_state1, 'MyData1');
   else
      -- Parent is being run as resumed request.
      v_paused_str := ess_runtime.get_paused_state(v_reqid);
      if (v_paused_state1 = v_paused_str) then
         -- Request being resumed after first pause.
         -- Submit a subrequest and exit as success to allow ESS to pause this
parent.
         submit_subrequest(request_handle, v_paused_state2, 'MyData2');
      elsif (v_paused_state2 = v_paused_str) then
         -- Request being resumed after second pause.
         -- Parent is done. Just return as success.
         null;
      end if;
   end if;
end;

-- Helper procedure to submit subrequest and call mark_paused.
-- Caller should exit normally, without an error, to allow
-- ESS mid-tier to pause the parent and execute the sub-request.
procedure submit_subrequest
( request_handle in varchar2,
  paused_state_value in varchar2,
  reqprop_value in varchar2 )
is
   v_sub_reqid  number := null;
   v_req_props  ess_runtime.request_prop_table_t := null;
   v_idx  pls_integer := 0;
begin
   v_req_props := ess_runtime.request_prop_table_t();
   v_idx := 0;

   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 := reqprop_value;

   -- Must commit or rollback work done in this block.
```

```
   begin
      -- Submit the subrequest.
      v_sub_reqid := ess_runtime.submit_subrequest(
            request_handle => request_handle,
            definition_name => 'sampleJob',
            definition_package => 'samplePkg',
            props => v_req_props);

      -- Indicate that the parent request should pause.
      -- The actual state change does not occur until ESS is notififed
      -- that this run of the parent executable finished.
      ess_runtime.mark_paused(request_handle, paused_state_value);

      -- This procedure is responsible for the txn commit.
      commit;
   exception
      -- Rollback txn on failure and raise an error.
      rollback;
      raise_application_error(-20000,
                              'Error submitting sub-request. '||
                              'SQLCODE='||SQLCODE || ', SQLERRM=['||SQLERRM||']',
                              true);
      end;
   end;
end;
```

21

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:

- Introduction to Working with Asynchronous Java Jobs
- Creating an Asynchronous Java Job
- A Use Case Illustrating the Implementation of a BPEL Process as an Asynchronous Job
- How to Implement BPEL with an Asynchronous Job
- Handling Time Outs and Recovery for Asynchronous Jobs
- Oracle Enterprise Scheduler Interfaces and Classes

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 can be used to initiate a 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.

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.

Implementing the Asynchronous Java Job Asynchronous Interface

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

Asynchronous Java Job execute() Method

The duty of an asynchronous Java jobs's `execute()` method is to set up the external mechanism in which the real work runs; this should start the external mechanism and then return. The asynchronous Java job invocation `execute()` method may not do any actual work. An exception can be thrown during the execute method to tell Oracle Enterprise Scheduler that this job had a problem during initialization and failed to run. The exception during the execute method does not tell Oracle Enterprise Scheduler that the actual work running on the external mechanism encountered a problem. It is the responsibility of the job owner to make sure any resources that may have been started or used are released, since Oracle Enterprise Scheduler does no further processing if it catches an exception. Assuming no exception is thrown, Oracle Enterprise Scheduler puts the job into the running state and then releases the handle on the job's object so that it may be garbage collected.

Invoking a Remote Job from an Asynchronous Java Job

An asynchronous Java job that invokes an external web service 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. The Oracle Enterprise Scheduler generic

callback service address can be obtained using the **WebServices WSDL URLs** link on the Oracle Enterprise Scheduler instance home page at `http://`*host*`:`*port*`/ess/`.

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 21-15). 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`.

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 determined from the asynchronous callback to Oracle Enterprise Scheduler.

For more information about the Updatable interface, see Example 21-12.

Notifying Oracle Enterprise Scheduler When an Asynchronous Job Completes

When an asynchronous Java jobs's `execute()` method is successful and the job request is running on the external mechanism, Oracle Enterprise Scheduler continues processing other jobs. When the job request is complete or encounters an error, it must communicate back to Oracle Enterprise Scheduler. This communication channel is the responsibility of the external mechanism.

Using the Web Service to Notify When an Asynchronous Job Completes

When you invoke the Oracle Enterprise Scheduler web service operation, `setAsyncRequestStatus`, this sets the asynchronous request's status and associated information. Associated with this operation, the following pieces of information are needed:

`setAsyncRequestStatus(String requestExecutionContext, AsyncStatus status, String statusMessage)`

Where:

- *`requestExecutionContext`* is a string that should be passed in as part of the initiating event. This parameter is derived from the Oracle Enterprise Scheduler job's `RequestExecutionContext` object.
- *`status`* is one of the following: `SUCCESS`, `ERROR`, `WARNING`, `PAUSE`, `CANCEL`, `BIZ_ERROR` or `UPDATE`.
- *`statusMessage`* is:
  - An error message if the status is `ERROR` or `BIZ_ERROR`.
  - A warning message if the status is `WARNING`.
  - A paused state if the status is `PAUSED`.
  - A customized string you define and have the job interpret accordingly if the status is `UPDATE`.
  - The value is ignored if the status is `SUCCESS` or `CANCEL`.

For more information about implementing a web service in a web application, see the chapters "Integrating Web Services Into a Fusion Web Application" in *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* and "Securing and Administering WebLogic Web Services" in *Oracle Fusion Middleware Security and Administrator's Guide for Web Services*.

Using EJB to Notify When an Asynchronous Job Completes

The external mechanism can communicate status to Oracle Enterprise Scheduler using the asynchronous request EJB, that provides local and remote EJB interfaces. A helper class is provided that 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. The following is an example:

Async request bean: `java:comp/env/essnative/asyncrequest`

The method:

```
/**
   * Set the status of an ESS asynchronous java job.
   *
   * @param context The <code>RequestExecutionContext</code> with which the
   * request was started.
   * @param status The status of the request.
   * @param statusMessage An error message if the status is ERROR,
   * a warning message if the status is WARNING,
   * the paused state if the status is PAUSED.
```

```
     * The value is ignored if the status is SUCCESS or CANCEL.
     * @throws RequestNotFoundException If a request is not found for the
     * <code>context</code>.
     */
    public void setRequestStatus(String context,
                                 String status,
                                 String statusMessage)
        throws RuntimeServiceException, RequestNotFoundException
```

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 canceled.

Note:

Currently, there is no way to terminate a running asynchronous Oracle ADF Business Components web service process.

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 Using the Web Service to Notify When an Asynchronous Job Completes.
4. After Oracle Enterprise Scheduler has the completion information, it completes 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 21-1.

Example 21-1 Job that Initiates a BPEL Process Through an Event Mediator

```
import oracle.as.scheduler.RequestParameters;
import oracle.as.scheduler.ExecutionCancelledException;
import oracle.as.scheduler.ExecutionErrorException;
```

```
import oracle.as.scheduler.ExecutionPausedException;
import oracle.as.scheduler.ExecutionWarningException;
import oracle.as.scheduler.RequestExecutionContext;

import javax.xml.namespace.QName;
import oracle.fabric.blocks.event.BusinessEventConnection;
import oracle.fabric.blocks.event.BusinessEventConnectionFactory;
import oracle.fabric.common.BusinessEvent;
import oracle.integration.platform.blocks.event.BusinessEventBuilder;
import
oracle.integration.platform.blocks.event.BusinessEventConnectionFactorySupport;
import oracle.xml.parser.v2.XMLDocument;
import org.w3c.dom.Element;


// Async imports
import oracle.as.scheduler.AsyncExecutable;
import oracle.as.scheduler.AsyncCancellable;


public class BPELJob implements AsyncExecutable, AsyncCancellable
{
    public BPELJob() {
    }

    public void execute(RequestExecutionContext ctx, RequestParameters params)
        throws ExecutionErrorException,
               ExecutionWarningException,
               ExecutionCancelledException,
               ExecutionPausedException
    {
        // Publish an event to the Event Mediator
        publishEvent(ctx.getRequestId() + "", ctx.toString(), "ESS_EVENT");
    }

    // Cancel

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

    // Event publishing

    private final String eventName = "ESSDemoEvent";
    private final String eventElement = "ESSDemoEventElement";
    private final String eventNamespace =
            "http://xmlns.oracle.com/apps/ta/essdemo/events/edl";
    private final String schemaNamespace =
            "http://xmlns.oracle.com/apps/ta/essdemo/events/schema";

    private XMLDocument buildEventPayload(String correlationId, String key,
String
                                          eventType) {
        Element masterElem, childElem1, childElem2, childElem3;
        XMLDocument document = new XMLDocument();
        masterElem = document.createElementNS(schemaNamespace, eventElement);
        document.appendChild(masterElem);
        childElem1 = document.createElementNS(schemaNamespace, "requestId");
```

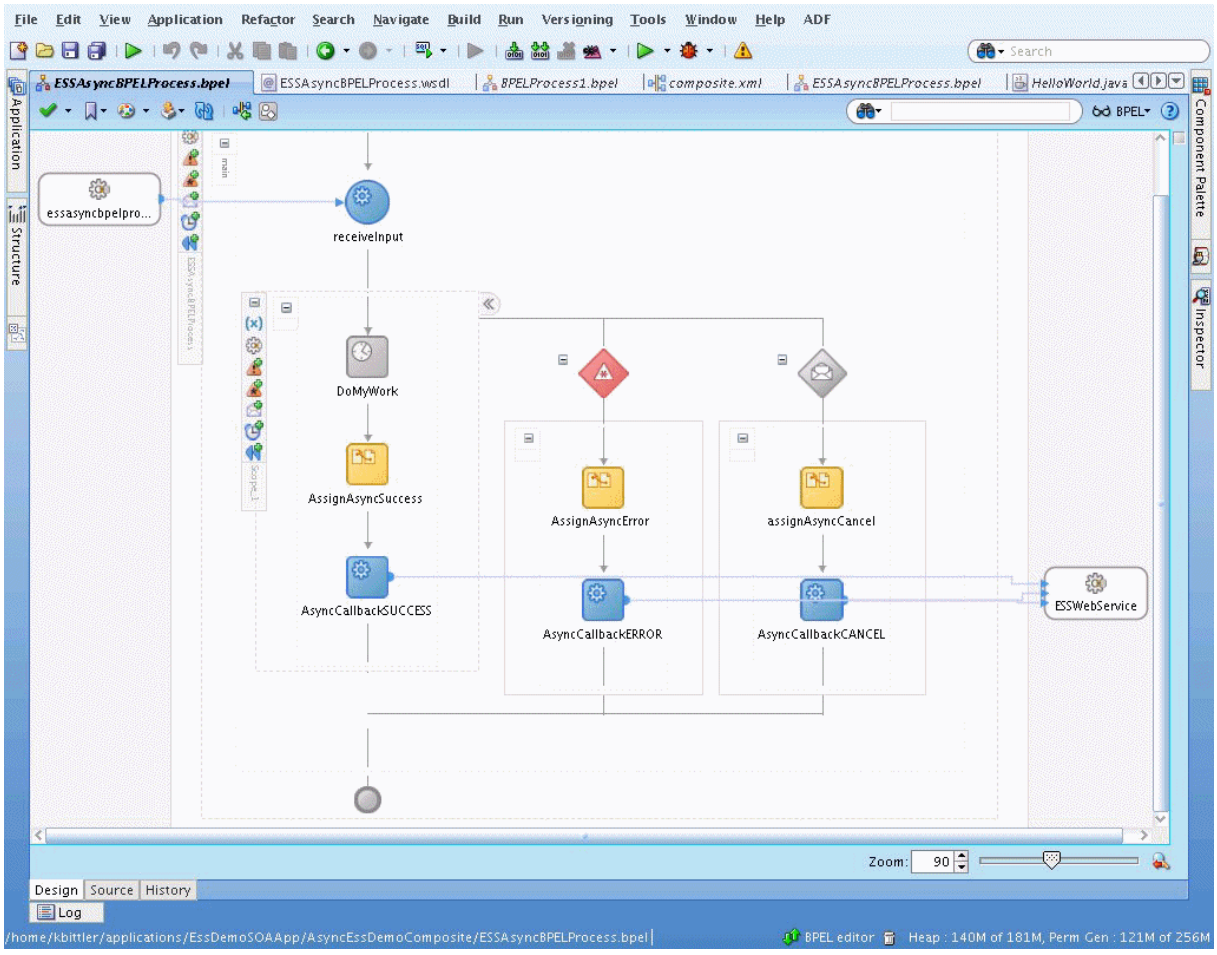
```
        childElem1.appendChild(document.createTextNode(correlationId));
        masterElem.appendChild(childElem1);
        childElem2 = document.createElementNS(schemaNamespace,
                                              "executionContext");
        childElem2.appendChild(document.createTextNode(key));
        masterElem.appendChild(childElem2);
        childElem3 = document.createElementNS(schemaNamespace, "eventType");
        childElem3.appendChild(document.createTextNode(eventType));
        masterElem.appendChild(childElem3);
        return document;
    }


    private void publishEvent(String correlationId, String key, String eventType)
        throws ExecutionErrorException
    {

            try {
                 // Get event connection
                 BusinessEventConnectionFactory cf =
                        BusinessEventConnectionFactorySupport.
                        findRelevantBusinessEventConnectionFactory(true);

                 if (cf != null) {
                     BusinessEventConnection conn =
                             cf.createBusinessEventConnection();

                     //  Build event
                     BusinessEventBuilder builder =
                             BusinessEventBuilder.newInstance();

                     // Specify the event name and namespace. In this prototype,
                     // they are constants, eventNamespace, eventName
                     builder.setEventName(new QName(eventNamespace, eventName));

                     // Specify the event payload. In this prototype, the
                     // getXMLPayload custom method constructs the payload
                     builder.setBody(buildEventPayload(correlationId, key,
                                     eventType).getDocumentElement());
                     BusinessEvent event = builder.createEvent();

                     //  Publish event
                     conn.publishEvent(event, 5);

                     // For debug only
                     System.out.println("Event was sent sucessfully");
                 } else {
                     // For debug only
                     System.out.println("cf is null");
                 }
              } catch (Exception exp) {
                 String errorMsg = "Failed sending event for correlation ID "
+                         correlationId);
                 exp.printStackTrace();
                 throw new ExecutionErrorException(errorMsg, exp);
              }
        } // publishEvent
```

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 21-1.

- The real work of the process is done in the DoMyWork module.
- If the work completes successfully, control flows 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 is 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 jumps 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 21-1 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 21-2, Figure 21-3, and Figure 21-4 for the `AssignAsyncError` assignment activity.

Figure 21-2 AsyncCallBackError Argument Mapping for statusMessage Element

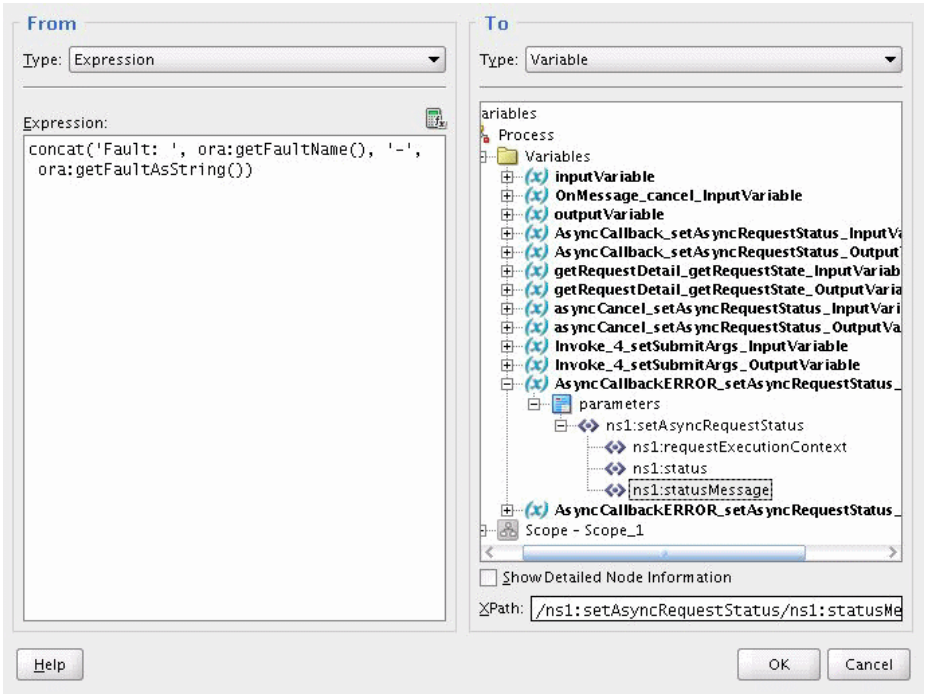


Figure 21-3 AsyncCallbackError Argument Mapping for requestExecutionContext

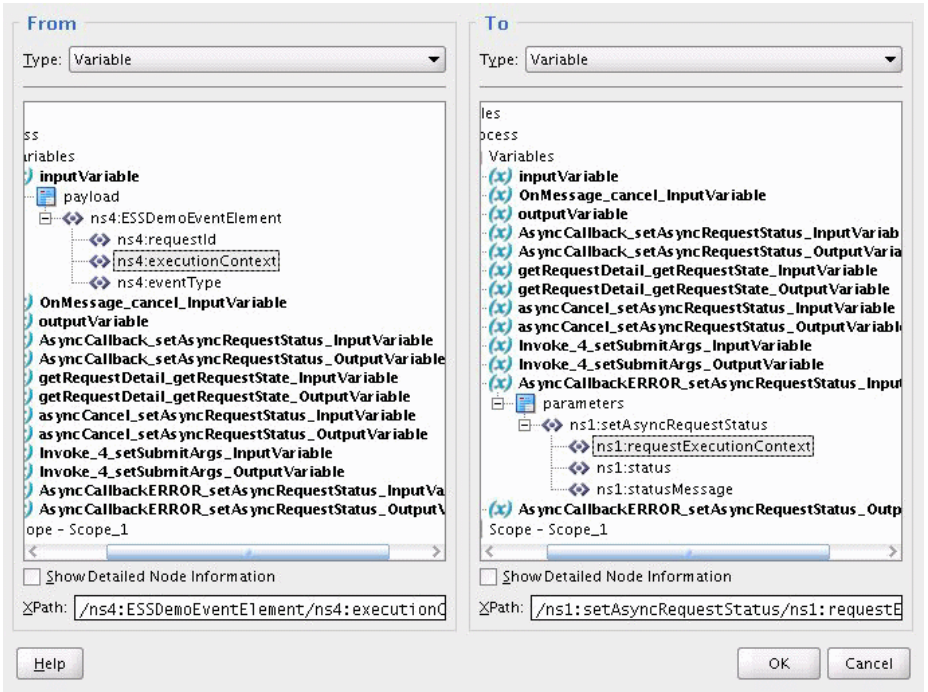
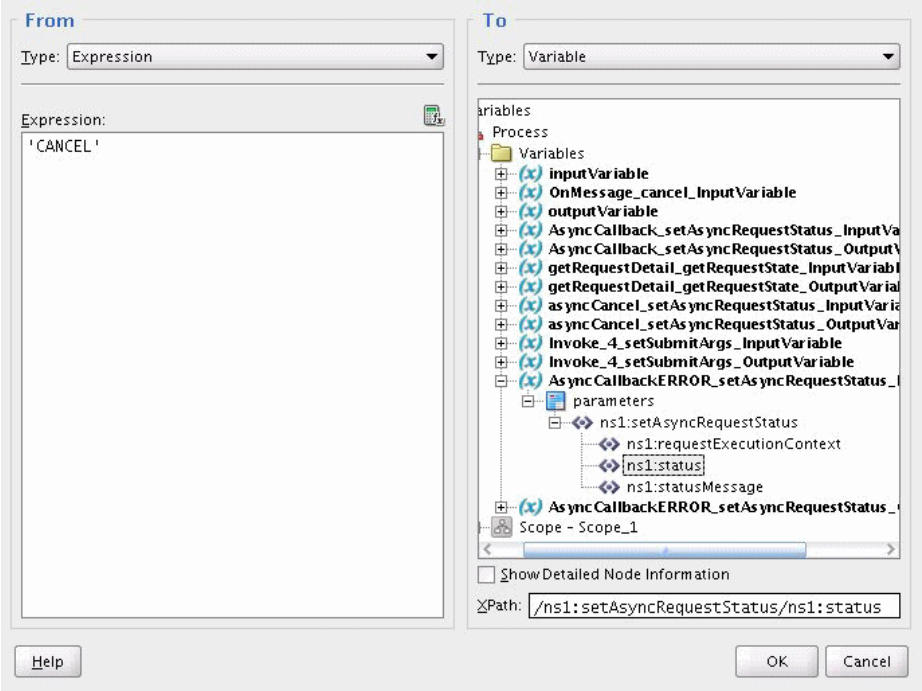


Figure 21-4 AsyncCallbackError Argument Mapping for status Element



A Use Case Illustrating the Implementation of a BPEL Process as an Asynchronous Job

Use cases for implementing a BPEL process as an asynchronous job are as follows:

- Gaining approval for a task using human workflow notifications and other SOA-specific activities.
- Notifying Oracle Enterprise Scheduler that a job has completed, while allowing other jobs to run or proceed to the next job in a set.

Design Pattern Summary

Asynchronous Oracle Enterprise Scheduler jobs are Java jobs that implement the `AsyncExecutable` interface, which is invoked by Oracle Enterprise Scheduler by implementing the `execute()` method. This method enables initiating a long running or remote task where the `execute()` method completes (such as raising a business event), while Oracle Enterprise Scheduler keeps the job in `RUNNING` status. The remote task completes and notifies Oracle Enterprise Scheduler of its completion using a status message using one of the following implementations:

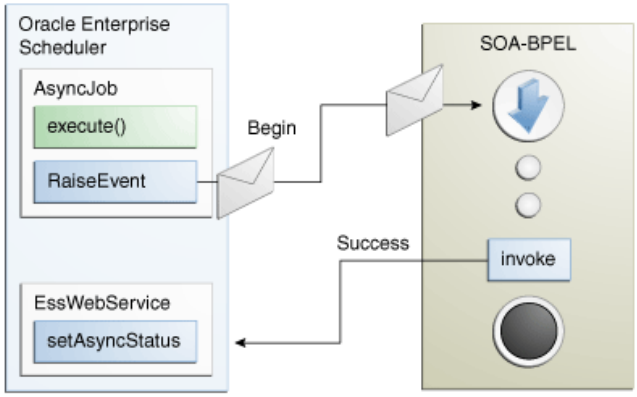
- The `RuntimeService` EJB
- The Oracle Enterprise Scheduler web service `setAsyncRequestStatus` operation.

This pattern assumes the remote task to be invoked is a BPEL process which is triggered by raising a business event in the `execute()` method of the asynchronous job. Upon termination of the process through completion, error or cancellation, the BPEL process invokes the Oracle Enterprise Scheduler web service and sets the status accordingly.

Involved Components

Oracle Enterprise Scheduler, SOA Meditator and BPEL, as shown in Figure 21-5.

Figure 21-5 BPEL Call from Oracle Enterprise Scheduler Asynchronous Job



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.

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 using 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.

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 sub-requests for each expense to trigger the asynchronous BPEL functionality per expense. This sub-request is implemented as an asynchronous Oracle Enterprise Scheduler job which raises a business event, completing it's Java execute() method, and staying in a running state while BPEL is initiated, submits the Human Task notification and awaits the outcome from user interaction. After this outcome is obtained, BPEL invokes the Oracle Enterprise Scheduler web service signaling that this particular sub-request is completed.

How to Implement BPEL with an Asynchronous Job

Implementing an Oracle Enterprise Scheduler asynchronous job in BPEL requires performing the following steps:

1. Author the Oracle Enterprise Scheduler Java job to implement the `AsyncExecutable` and `AsyncCancellable` interfaces by writing `execute()` and `cancel()` methods.
2. Create the asynchronous Oracle Enterprise Scheduler job definition.
3. Design the event payload schema (XSD) and event definition (EDL) files.
4. Programmatically raise a business event from the asynchronous Oracle Enterprise Scheduler job execute() and (optionally) cancel methods.
5. Design the SOA Composite with Meditator and BPEL.
6. Add fault handling and correlated `onMessage` branch for error and cancel job status updates.

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 implements 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 21-2.

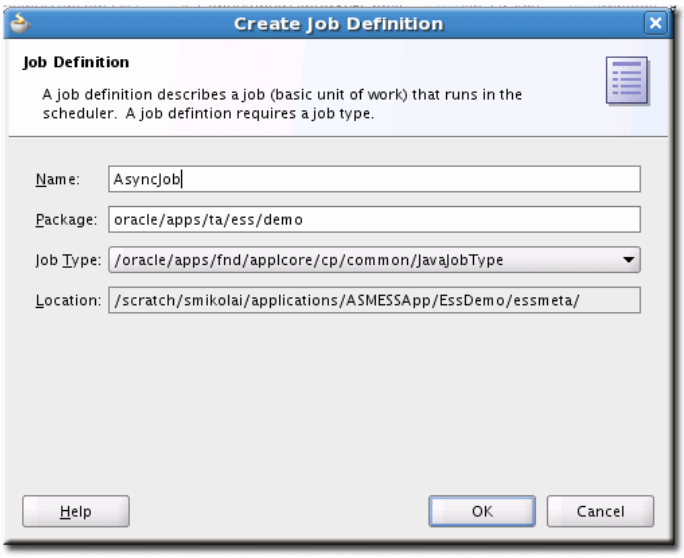
Example 21-2 Adding Oracle JDeveloper Libraries

```
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
```

Use Case: Create the Asynchronous Job Definition

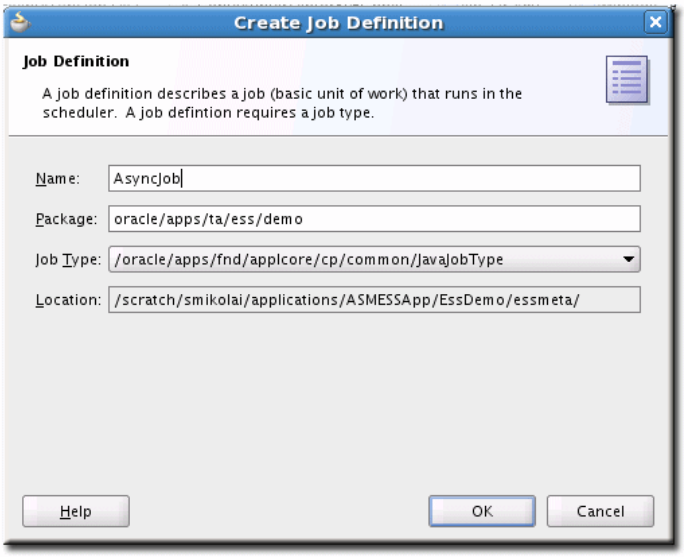
In your Oracle Enterprise Scheduler JDeveloper workspace, click "New', choose the Enterprise Scheduler Service technology group and select "Job Definition". Enter the name off your Oracle Enterprise Scheduler job definition, choose the provided "JavaJobType" and select the class build in step 1 as the overriding Java class for this job definition, as shown in Figure 21-6.

Figure 21-6 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 21-7.

Figure 21-7 Create Job Definition with Job Type Defined

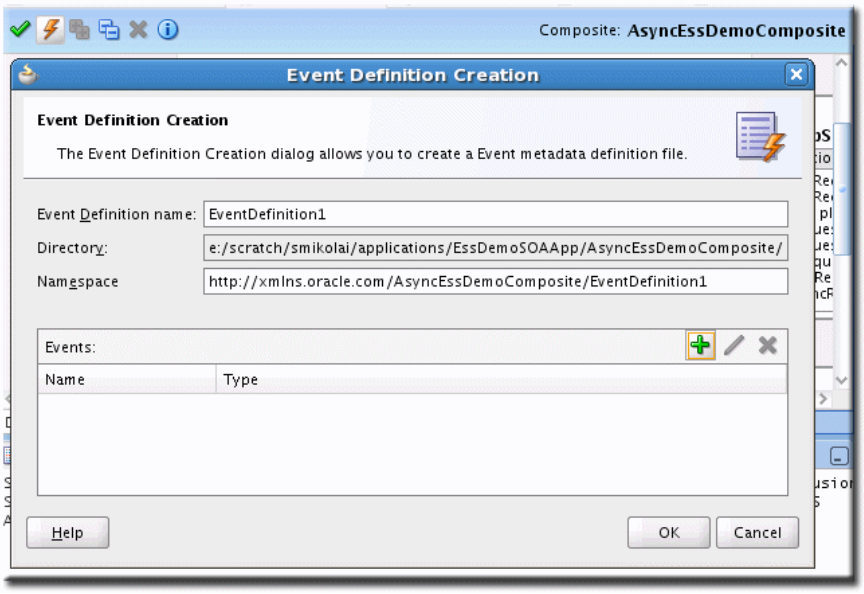


Use Case: Design the Event Payload Schema and Event Definition Files

The SOA composite designer has UI features to assist in designing business event payload definitions (EDL); however your schema (.xsd) must be designed first. Example 21-3 shows a sample XSD file.

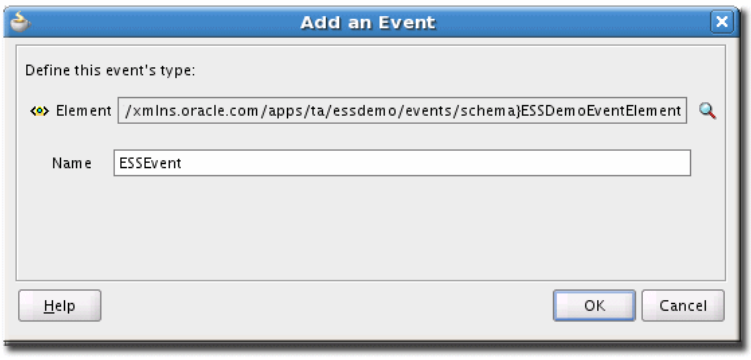
With the payload element type completed, you can either create the EDL by hand or use the event definition builder. To use the builder, open the SOA composite editor and click the lightning bolt button at the top of the UI to open the Event Definition Creation window, as shown in Figure 21-8

Figure 21-8 Event Definition Creation



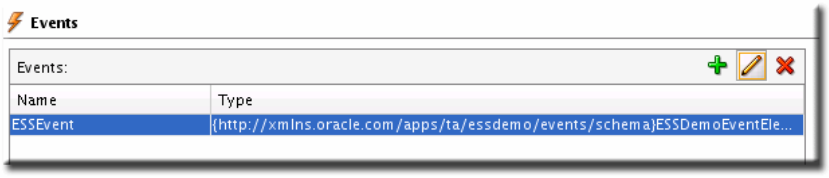
Next, assign a name and name space and click **Add** to add a new event to this definition, as shown in Figure 21-9.

Figure 21-9 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 21-10.

Figure 21-10 Events List



Example 21-4 shows a sample of the EDL file that is created.

Example 21-3 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>
```

Example 21-4 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>
```

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/canceled/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 must design an event schema (.xsd) and definition (EDL) in order to declaratively build the SOA composite which subscribes 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 name space 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 then invokes the cancel() method. It's recommended that both methods raise events that contain similar payload types/name spaces 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 21-5.

Example 21-5 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)        throws ExecutionErrorException    {            // 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) {                String errorMsg = "Failed sending event for
correlation ID " +                           correlationId);
exp.printStackTrace();                throw new
ExecutionErrorException(errorMsg);             }        } // publishEvent
```

Design the SOA Composite with Meditator and BPEL

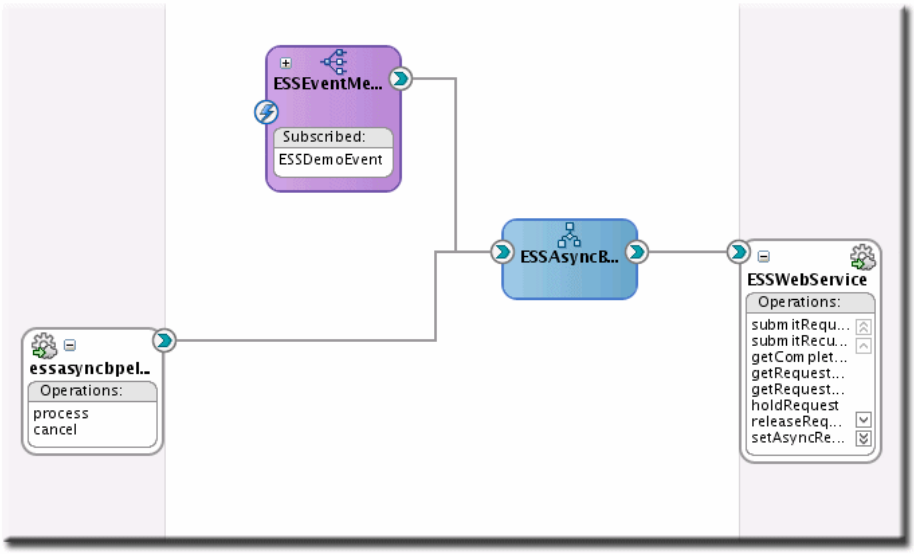
Since this use case depends on BPEL functionality it is necessary to build a SOA composite which contains a Mediator for event subscription which can then transform the payload and initiate the BPEL process.

In your SOA workspace, create a new SOA composite. To setup the composite for this pattern, add a Mediator that subscribes to your Oracle Enterprise Scheduler raised event and wire it to a BPEL process. Add a service reference to the Oracle Enterprise Scheduler web service WSDL. For example,

```
http://myhost.com:7001/ess/esswebservice?WSDL
```

Continue to build the required functionality in the BPEL process using one or more nested scopes. Bear in mind that your functionality should reside within at least one primary scope on which you can add an `onMessage` event (for in-flight cancel message receipt) and fault handler branches, as shown in Figure 21-11.

Figure 21-11 Composite with BPEL and ESSWebService



For more information about invoking the Oracle Enterprise Scheduler web service, see Using the Oracle Enterprise Scheduler Web Service .

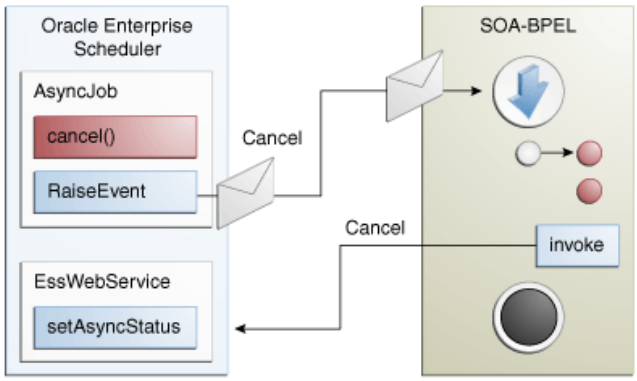
Add Fault Handling and Correlated onMessage Branch for Error and Cancel Job

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. After 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 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 canceled, Oracle Enterprise Scheduler then updates the job's status to `CANCELLING` and wait for the remote functionality to tidy up and confirm that it has canceled, as shown in Figure 21-12.

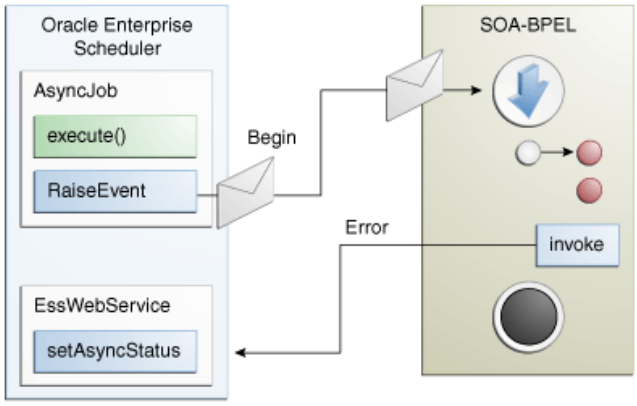
Figure 21-12 BPEL Handling Cancellation



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 21-13.

Figure 21-13 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 contain the functionality for this asynchronous job. This allows 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.

Create Correlation Set and Define Initiate Activity

In order to support receiving the cancel event while the BPEL process is in the middle of performing other activities or waiting for an asynchronous callback the process must be configured with a correlation set. A correlation set is key value that is built from one or more incoming payload attributes which are used to uniquely identify the BPEL process to the BPEL engine whereby additional service requests that contain matching sets of attributes can be routed to the process that is currently running instead of initiating a new one. While correlation is standard functionality used for asynchronous request responses, it can also be used to change the flow of execution in a BPEL process through scope-level `onMessage` branches.

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

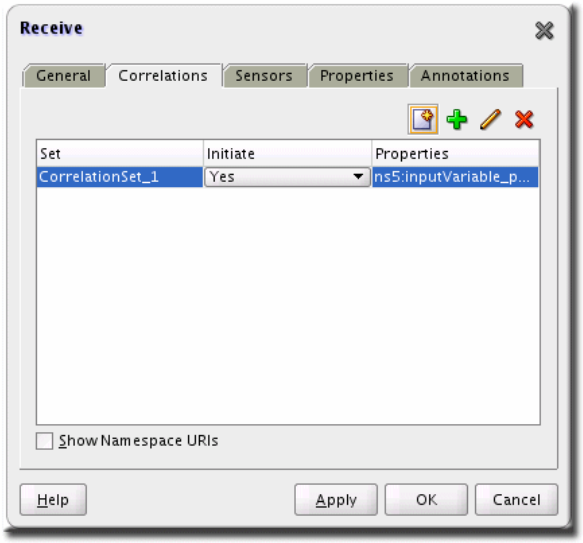
Note that coarctation sets have an "initiate" property which indicates which activity is the starting point for this correlation set's life cycle. In this case, the start of the BPEL process is the point at which the correlation set's life cycle should begin allowing correlated events to route to this process at any point during the process.

To create a correlation set:

- Click the "New" button 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 is always unique.
- Set the initiate flag on the correlation to "Yes" on the activity for which the correlation set's life cycle should begin.

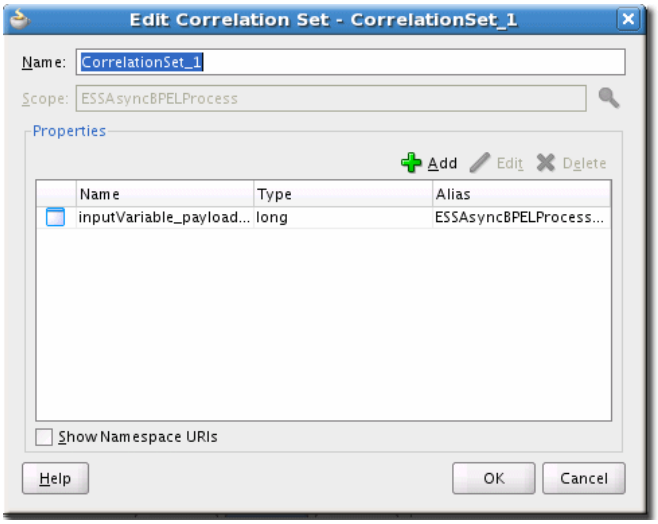
Primary (first) Receive Activity with Defined Correlation Set and "Initiate" flagged to "Yes", as shown in Figure 21-14.

Figure 21-14 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 21-15.

Figure 21-15 Edit Correlation Set



Create the onMessage Branch with Use of Correlation Set

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

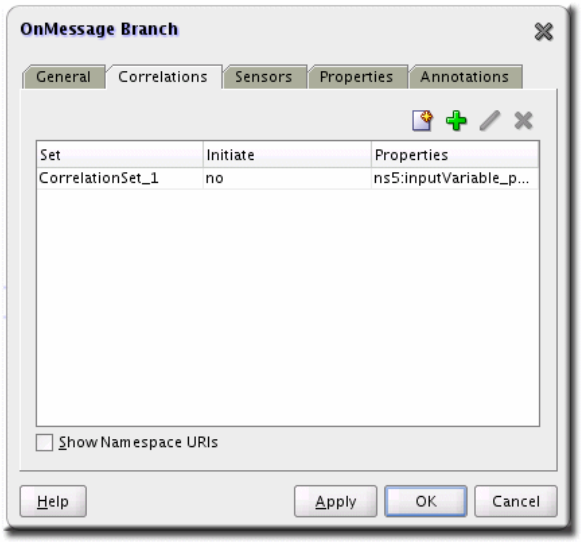
Note:

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 21-16.

- On the nested scope containing the process functionality, click the 'Add onMessage branch' button 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 '+' button and select the previously created correlation set ensuring that the initiate flag is set to 'No' and click "Ok".

Figure 21-16 BPEL OnMessage Branch



Create the Fault Branch

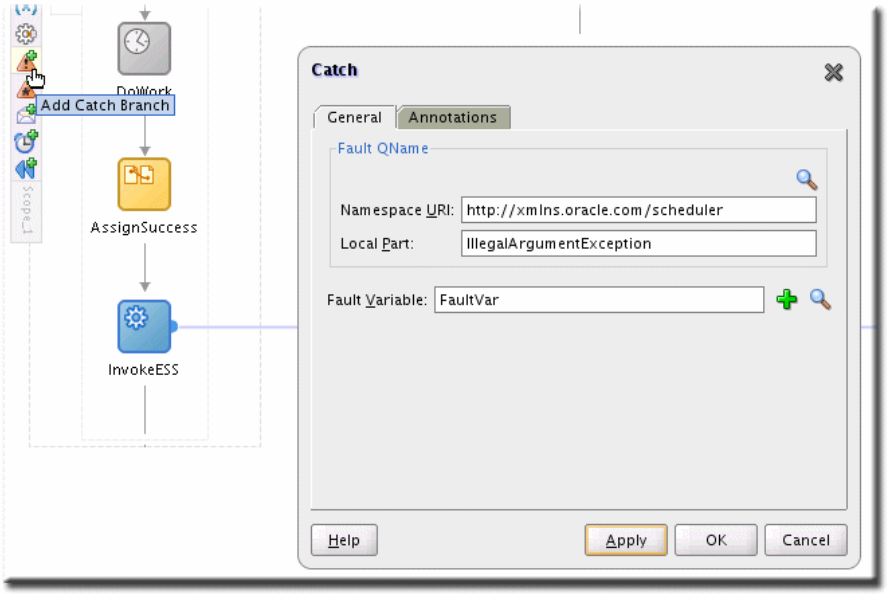
Through the course of performing the various activities in the nested work scope BPEL may encounter faults from business services or system functionality. In most cases, business services define one or more WSDL-defined faults that can be thrown back to the calling process. Ordinarily, a BPEL CatchAll fault branch traps 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 buttons to create the desired fault handling branches, then double-click the named fault handling branches and define the named fault those branches catch.

Note the available status, as shown in Figure 21-17.

Figure 21-17 Catch Branch for BPEL Flow



Populate the onMessage and Fault Branch

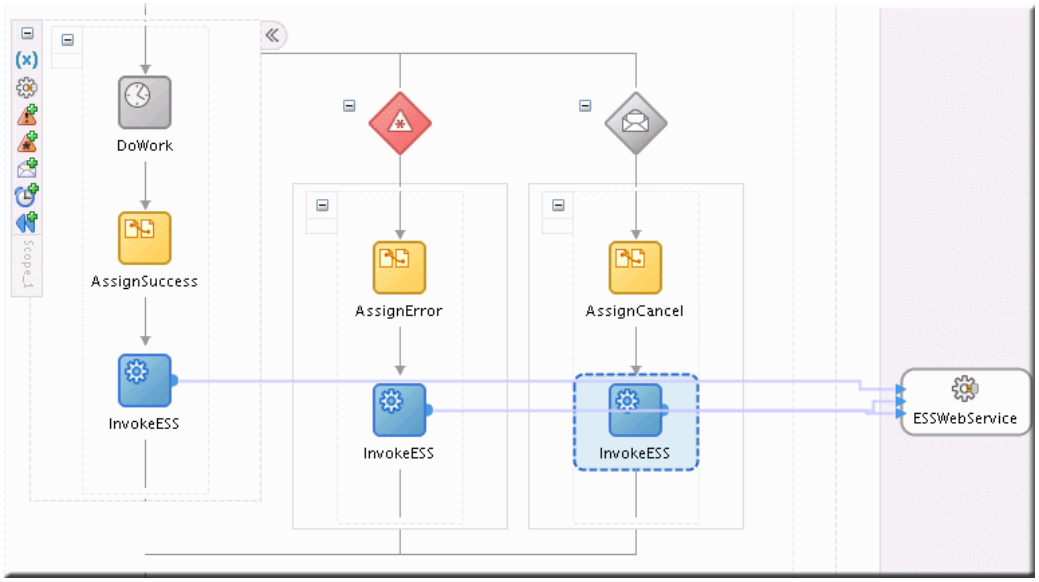
You need to populate the onMessage and Fault branch with cleanup activities as needed and invoke Oracle Enterprise Scheduler web service with appropriate status.

In the event of a fault or receipt of the cancellation message through the onMessage branch the Oracle Enterprise Scheduler infrastructure must be updated directly using 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 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 21-18.

Figure 21-18 Entire BPEL Flow Sample



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 21-6.

4. Your subscribing mediator has 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 21-7.

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.

Example 21-6 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>
```

Example 21-7 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
```

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 runtime, see the chapter "Troubleshooting Oracle Enterprise Scheduler" in *Oracle Fusion Middleware Administering Oracle Enterprise Scheduler*.

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.

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.

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.

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 21-8.

A similar query can be run using `REQUEST_HISTORY_VIEW`, as shown in Example 21-9.

Example 21-8 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);
```

Example 21-9 Using REQUEST_HISTORY_VIEW

```
SELECT requestId FROM request_history_view WHERE timedout='Y' AND state=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 Administering Oracle Enterprise Scheduler*.

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.

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. 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 are 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 Administering Oracle Enterprise Scheduler*.

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 21-10.

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 is thrown. The exception message lists child requests that are incomplete. Example 21-11 shows the `recoverRequest` syntax.

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 Administering Oracle Enterprise Scheduler*.

Example 21-10 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());
```

Example 21-11 recoverRequest

```
   /**
     * Attempts to force a request to complete under certain conditions.
     * <p>
     * 1. The request must already by in a terminal state, {@code
     *    State.CANCELLING}, or {@code State.ERROR_MANUAL_RECOVER}.
     *    If a request is in another state,
     *    {@code RuntimeService.cancel} must be called first. If the
     *    request does not eventually transition to {@code State.CANCELLED},
     *    then this operation may be invoked on the request.
     * 2. All child requests of the given request must already be complete.
     * <p>
     * A <b>completed></b> request is a request in a terminal state with
     * a process phase of {@code ProcessPhase.Complete}.
     * <p>
     * Note that this operation locks the request.
     * <p>
     * @param requestId the request identifier of the request.
     * @throws IOException if a protocol error occurred.
     * @throws InstanceNotFoundException if the request is not found
     * @throws OperationException if the given request has child requests
     *  that are not complete.
     * @throws RuntimeOperationsException if a RuntimeService subsystem failure
     * occurs.
     */
    public void recoverRequest( long requestId )
        throws IOException, InstanceNotFoundException, OperationsException,
               RuntimeOperationsException;
```

Oracle Enterprise Scheduler Interfaces and Classes

Sample code illustrating the Oracle Enterprise Scheduler asynchronous callback interfaces and classes are shown in Example 21-12, Example 21-13, Example 21-14 and Example 21-15.

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

Example 21-12 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 );
}
```

Example 21-13 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
 * proceeds.
 */
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( );
}
```

Example 21-14 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 changes to the corresponding value. The new state of the request
 * depends 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 21-15 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 ;
```

22

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:

- Request Logs
- Request Output

Request Logs

Oracle Enterprise Scheduler provides APIs your job can use to request logging. All job types, except for process jobs, use the logging APIs. Oracle Enterprise Scheduler also provides APIs to handle request logs in the content store.

Oracle Enterprise Scheduler supports a single log per request. The log has 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 Administering Oracle Enterprise Scheduler*.

System Properties

The system property `SYS_EXT_requestLogLevel` constrains which messages logged using the API are stored in the request log.

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 `SYS_EXT_requestLogLevel` does not apply to process jobs because they do not use the logging API.

Log Header

Your logging code writes entries to a log that begins with the following heading information. This header is automatically prefixed to each record written to the log using the logging APIs, therefore it does not apply to process jobs.

```
####[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.
```

Request Logging 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 `RequestLogger` that can be used to log messages to the request log. Oracle Enterprise Scheduler uses the current value of the `SYS_EXT_requestLogLevel` system property to constrain logging level each time the logger is retrieved.

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 22-1) to get a `RequestLogger` instance for adding log entries. Note that the request logger does 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 22-1.
- Use the `RequestLogger` class to write the log. See Table 22-2.

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 22-1 ContentFactory Methods for Creating Request Logs

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

After you have a logger instance, you can use the methods in Table 22-2 to add entries.

Table 22-2 RequestLogger Methods for Creating Request Logs

Method	Description

`log( Level level, String msg )`
`fine( String msg )`
`finer( String msg )`
`finest( String msg )`
	These methods log messages at the specified levels.
	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 `INFO`.
	When using the log method, the `java.util.logging.Level` supports the following values, in descending order.
	• `SEVERE`
	• `WARNING`
	• `INFO`
	• `CONFIG`
	• `FINE`
	• `FINER`
	• `FINEST`

Java Request Logging Example

Example 22-1 shows a very simple Java job example that does logging.

Example 22-1 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);
    }
```

```
  }
}
```

Request Logging 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.

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 22-3 ESS_JOB Functions and Procedures for Request Logging

Method	Description
`procedure write_log( p_level in integer, p_text in varchar2 );`	Writes `p_text` as a message to request log content for the Oracle Enterprise Scheduler request associated with the current session. 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`. Log level values correspond to those defined in `java.util.logging.Level`. Use the following values for the `p_level` parameter (shown in descending order): • `LEVEL_SEVERE` • `LEVEL_WARNING` • `LEVEL_INFO` • `LEVEL_CONFIG` • `LEVEL_FINE` • `LEVEL_FINER` • `LEVEL_FINEST`

PL/SQL Request Logging Example

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

```
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;
/
```

Request Logging 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 imports 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 described in Process Job Locale.

Note that you cannot 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 does not constrain log contents. Oracle Enterprise Scheduler always appends the contents of the log file to the request log in the content store.

Request Logging and Output From an EJB Job

Because an EJB Job is a Java job that gets executed remotely, the remote EJB job implementation can write content to logs and output. Use the `oracle.as.scheduler.request.RemoteContentHelper` API to handle this content. The following bullet items describe the high-level steps involved. Example 22-2 shows an EJB job implementation that illustrates this functionality.

- Set up the `RemoteContentHelper`. The parameters `RequestExecutionContext` and `RequestParameters` can be obtained from the `execute()` method of the job implementation.

  ```
  RemoteContentHelper rch=new RemoteContentHelper(RequestExecutionContext,
  RequestParameters);
  ```

- Write content to the log file.

  ```
  rch.log(Level.INFO, logMessage)
  ```

- Write text output.

  ```
  RuntimeServiceHandle rsh = null;
   ContentHandle ch = null;
   String contentName = "Content1";
   String contentData = "Some text content";
   try {
      rsh = rch.getRuntimeService().open();
      ch = rch.openOutputContent(rsh, contentName, ContentType.Text,
  EnumSet.of(ContentHandle.ContentOpenOptions.Write));
      rchelper().write(ch, contentData);
   } catch (Exception ex) {
  ```

```
      throw new ExecutionErrorException(ex);
   } finally {
      rch.closeOutputContent(ch);
      rch.getRuntimeService().close(rsh);
   }
```

- Write binary output.

```
RuntimeServiceHandle rsh = null;
 ContentHandle ch = null;
 String contentName = "Content1";
 String contentData = "Some binary content"; //Binary content can differ in
the     variable declaration.
 try {
     rsh = rch.getRuntimeService().open();
     ch = rch.openOutputContent(rsh, contentName, ContentType.Binary,
EnumSet.of(ContentHandle.ContentOpenOptions.Write));
     rchelper().write(ch, contentData.getBytes());
 } catch (Exception ex) {
     throw new ExecutionErrorException(ex);
 } finally {
     rch.closeOutputContent(ch);
     rch.getRuntimeService().close(rsh);
 }
```

Example 22-2 shows an EJB job implementation that demonstrates how to handle output and logs.

Example 22-2 SimpleSyncEssBean

```
import java.util.ArrayList;
import java.util.logging.Level;
import java.util.EnumSet;

import javax.ejb.Stateless;

import oracle.as.scheduler.RuntimeServiceHandle;
import oracle.as.scheduler.ExecutionCancelledException;
import oracle.as.scheduler.ExecutionErrorException;
import oracle.as.scheduler.ExecutionPausedException;
import oracle.as.scheduler.ExecutionWarningException;
import oracle.as.scheduler.RemoteExecutable;
import oracle.as.scheduler.RequestExecutionContext;
import oracle.as.scheduler.RequestParameters;
import oracle.as.scheduler.request.RemoteContentHelper;
import oracle.as.scheduler.request.ContentDetail;
import oracle.as.scheduler.request.ContentHandle;
import oracle.as.scheduler.request.ContentType;

@Stateless(name = "SimpleSyncEssBean")
public class SimpleSyncEssBean implements RemoteExecutable {
    private RemoteContentHelper m_rch = null;

    public void execute(RequestExecutionContext requestExecutionContext,
                        RequestParameters requestParameters) throws
ExecutionErrorException, ExecutionWarningException,

ExecutionCancelledException,

ExecutionPausedException {
        long requestId = requestExecutionContext.getRequestId();
        // setup helper now so it is available to println
```

```
        m_rch = setupRemoteContentHelper(requestExecutionContext,
requestParameters);

        ContentType contentType = ContentType.Text;
        String contentName = requestId + "Output.txt";

        ArrayList<String> srcLines = new ArrayList<String>();
        srcLines.add("SimpleSyncEssBean Output Details \n");
        boolean success = false;
        String exmsg = "";

        try {
            createOutputContent(contentName, contentType, srcLines);
            success = verifyOutputContent(contentName, contentType, srcLines);
        } catch (ExecutionErrorException ex) {
            exmsg = ex.getMessage();
        }

        if (!success) {
            throw new ExecutionErrorException("Output test failed: " +
contentName                 + ", " + exmsg);
        } else {
            printToLog("Output test succeeded: " + contentName);
        }
        printToLog(" SimpleSyncEssBean job succeeded. RequestId:" + requestId);
    }

    /**
     * Creates the remote content helper for log/output.
     */
    private RemoteContentHelper
setupRemoteContentHelper(RequestExecutionContext        ctx, RequestParameters
params) throws ExecutionErrorException {

        RemoteContentHelper rch = null;
        try {
            rch = new RemoteContentHelper(ctx, params);
        } catch (Exception ex) {
            throw new ExecutionErrorException(ex);
        }

        return rch;
    }

    /**
     * Gets the remote content helper that is setup.
     * Throws if it is not setup.
     */
    private RemoteContentHelper rchelper() throws ExecutionErrorException {
        if (null == m_rch) {
            throw new ExecutionErrorException("RemoteContentHelper is not
setup");
        }

        return m_rch;
    }


    private void printToLog(String message) throws ExecutionErrorException {
        System.out.println(message);
        if (m_rch != null) {
```

```
            try {
                m_rch.log(Level.INFO, message);
            } catch (Exception ex) {
                // ignore
            }
        } else {
            rchelper();
        }

    }

    /**
     * Writes dataList as either Text or Binary output content.
     */
    private void createOutputContent(String contentName, ContentType contentType,
                                     ArrayList<String> dataList) throws
ExecutionErrorException {
        RuntimeServiceHandle rsh = null;
        ContentHandle ch = null;
        try {
            rsh = rchelper().getRuntimeService().open();
            ch =
                rchelper().openOutputContent(rsh, contentName, contentType,

EnumSet.of(ContentHandle.ContentOpenOptions.Write));

            for (String data : dataList) {
                if (ContentType.Text == contentType) {
                    rchelper().write(ch, data);
                } else {
                    rchelper().write(ch, data.getBytes());
                }
            }
        } catch (Exception ex) {
            throw new ExecutionErrorException(ex);
        } finally {
            if (ch != null) {
                try {
                    rchelper().closeOutputContent(ch);
                } catch (Exception ex) {
                    printToLog("Error while closing ch: " + ex.getMessage());
                    //ex.printStackTrace();
                    throw new ExecutionErrorException(ex);
                }
            }
            if (rsh != null) {
                try {
                    rchelper().getRuntimeService().close(rsh);
                } catch (Exception ex) {
                    printToLog("Error while closing rsh: " + ex.getMessage());
                    //ex.printStackTrace();
                    throw new ExecutionErrorException(ex);
                }
            }
        }
    }

    /**
     * Verifies Text or Binary output content.
     */
    private boolean verifyOutputContent(String contentName,         ContentType
```

```
contentType, ArrayList<String> srcDataList)        throws
ExecutionErrorException {

        String actualData = getOutputContent(contentName, contentType);

        StringBuffer sb = new StringBuffer();
        for (String str : srcDataList) {
            sb.append(str);
        }
        String srcData = sb.toString();

        boolean success = srcData.equals(actualData);

        if (!success) {
            printToLog("Test failed for " + contentName);
            printToLog("Expected data: <\n" + srcData + ">");
            printToLog("Actual data: <\n" + actualData + ">");
        }

        try {
            if (rchelper().outputContentExists(contentName)) {
                ContentDetail detail =
rchelper().getOutputContentDetail(contentName);
                printToLog("ContentDetail:\n" + detail);
            } else {
                printToLog("The output content details are not present:\n");
            }

        } catch (Exception ex) {
            String exm = "Failed to get output content detail: " + contentName;
            printToLog(exm);
            throw new ExecutionErrorException(ex);
        }

        return success;
    }


    /**
     * Gets either Text or Binary output content as String.
     */
    private String getOutputContent(String contentName, ContentType
contentType)        throws ExecutionErrorException {
        RuntimeServiceHandle rsh = null;
        long requestId = rchelper().getRequestExecutionContext().getRequestId();
        ContentHandle ch = null;
        String result = null;
        try {
            rsh = rchelper().getRuntimeService().open();
            ch =
                rchelper().openOutputContent(rsh, contentName, contentType,

EnumSet.of(ContentHandle.ContentOpenOptions.Read));

            if (ContentType.Text == contentType) {
                char[] chars = rchelper().getTextContent(ch, Integer.MAX_VALUE);
                result = new String(chars);
            } else {
                byte[] bytes = rchelper().getBinaryContent(ch,
Integer.MAX_VALUE);
                result = new String(bytes);
```

```
            }
        } catch (Exception ex) {
            throw new ExecutionErrorException(ex);
        } finally {
            if (ch != null) {
                try {
                    rchelper().closeOutputContent(ch);
                } catch (Exception ex) {
                    printToLog("Error while closing ch: " + ex.getMessage());
                    //ex.printStackTrace();
                    throw new ExecutionErrorException(ex);
                }
            }
            if (rsh != null) {
                try {
                    rchelper().getRuntimeService().close(rsh);
                } catch (Exception ex) {
                    printToLog("Error while closing rsh: " + ex.getMessage());
                    //ex.printStackTrace();
                    throw new ExecutionErrorException(ex);
                }
            }
        }

        return result;
    }

}
```

Request Logging from a Web Service Job

Progress messages are only available to asynchronous jobs. The Oracle Enterprise Scheduler `AsyncWebServiceJob` callback endpoint supports a new `logAsyncWSJobProgressMessage` one-way operation. This operation allows an asynchronous SOA composite (or other asynchronous web service) to log progress messages using ws-addressing for correlation and the same callback endpoint and associated callback OWSM policy. This provides a simple, convenient option for logging progress messages from asynchronous SOA composites.

The Oracle Enterprise Scheduler asynchronous web service job callback endpoint WSDL provides the necessary information for using this operation. A portion of the WSDL is shown in Example 22-3.

Example 22-3 Asynchronous Web Service Job Callback Endpoint WSDL

```
 <xsd:complexType name="logAsyncWSJobProgressMessage">
          <xsd:sequence>
               <xsd:element name="level" type="tns:logLevel" form="qualified"/>
               <xsd:element name="message" type="xsd:string" form="qualified"/>
          </xsd:sequence>
     </xsd:complexType>

     <xsd:element name="logAsyncWSJobProgressMessage"
type="tns:logAsyncWSJobProgressMessage"/>
    <message name="logAsyncWSJobProgressMessageInput">
        <part name="parameters" element="tns:logAsyncWSJobProgressMessage"/>
    </message>

    <portType name="RequestPort">
        <operation name="logAsyncWSJobProgressMessage">
```

```
            <input
message="tns:logAsyncWSJobProgressMessageInput"
xmlns:ns1="http://www.w3.org/2006/05/addressing/wsdl"
ns1:Action="logAsyncWSJobProgressMessage"/>
        </operation>
    </portType>
```

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 How to Access the Runtime Service and Obtain a Runtime Service Handle.

Table 22-4 RuntimeService Methods for Handling Request Logs

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

Request Output

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

The output your code creates can be created directly in the Oracle Enterprise Scheduler content store, or it can be created in the file system and imported to the content store. Oracle Enterprise Scheduler can import it automatically or your job can use Oracle Enterprise Scheduler APIs to import it explicitly.

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, and then deletes request-specific subdirectories.

Oracle Enterprise Scheduler provides APIs for managing output content in the content store.

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 imported into the content store.

Your code can write temporary and output files to their respective request-specific directories at runtime. Oracle Enterprise Scheduler imports 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 Common Request File Directory Behavior, Shared Request File Directory Behavior, and Local Request File Directory Behavior.

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. Runtime 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 22-4. Oracle Enterprise Scheduler sets `USER_FILE_DIR` (`SYS_userFileDir`) based on `RequestFileDirectory` and then makes the property read-only. The job cannot override this by setting `USER_FILE_DIR`.

Example 22-4 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>
```

Common Request File Directory Behavior

Oracle Enterprise Scheduler automatically imports all request output files from the output directory to the content store, and deletes 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 does not overwrite and is ignored. In other words, content created with the API has precedence.

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

Shared Request File Directory Behavior

Any files created by the request remains 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 are available to all stages of the request.

Error Handling When a Shared Request File Directory is Used

Oracle Enterprise Scheduler creates the request work directory before the job request transitions to the `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 must 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 must remove the output files.

Local Request File Directory Behavior

Oracle Enterprise Scheduler creates 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 requires 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 automatically exports 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.

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 are 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 results 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 results 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 results in a system error for the request.

If the request is a process job, an error importing the request log is logged and not treated as an error. The log is 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 are not imported because the job could still be running. The log and output files are 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.

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 does not create any output or temporary files.

Table 22-5 System Properties for Creating Request Output

Method	Description
`SYS_EXT_supportOutputFiles`	String property indicating whether a job creates 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.

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 create output content directly in the content store, or your job can create files in the request's output directory in the file system. If your job creates files in the request's output directory, you can either explicitly import those files to the

content store or allow Oracle Enterprise Scheduler to automatically import the files 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.

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 and manage output content, including `RequestOutput` and `OutputContentHelper` output. See Table 22-6.
- Use the `RequestOutput` class to create output content directly in the content store. See Table 22-7.
- Use the `OutputContentHelper` class to explicitly manage content you create as files in the request output file directory, and to interact with the content store. See Table 22-8.

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 22-6 ContentFactory Methods for Java Request Output

Method	Description
`getRequestOutput( RuntimeServiceHandle rsh, long requestId, ContentType contentType, String contentName )`	Returns a `RequestOutput` instance with output for the specified request and creates the output content for the request. Each write uses the specified request service handle; your calling code is responsible for committing or rolling back the transaction.
`getOutputContentHelper( long requestId )`	Returns a `OutputContentHelper` instance for creating output content for requests with `Standard` or `Extended` request mode. Each operation is performed in a separate transaction.
`getOutputContentHelper( long requestId, RuntimeServiceHandle rsh )`	Returns a `OutputContentHelper` instance for creating output content for requests with `Standard` or `Extended` request mode. Each operation uses 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 write to the output content you are creating.

Table 22-7 RequestOutput Methods for Java Request Output

Method	Description
`writeln( String str )`	Appends `str` to the text output content, followed by a line feed character.
`write( String str )`	Appends `str` to the text output content.

Method	Description
`write( String str, int offset, int length )`	Appends `str` to the text output content.
`write( char[] chars )`	Appends `chars` to the text output content.
`write( char[] chars, int offset, int length )`	Appends chars to the text output content.
`write( byte[] bytes )`	Appends bytes to the binary output content.
`write( byte[] bytes, int offset, int length )`	Appends bytes to the binary output content.

Methods of the `oracle.as.scheduler.request.OutputContentHelper` class do the heavy lifting for output file 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 22-9.

Table 22-8 OutputContentHelper Methods for Java Request Output

Method	Description
`workDirectoryExists( )`	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. The job must define the `SYS_EXT_supportOutputFiles` system property with a value of `work` or `output` to cause Oracle Enterprise Scheduler to create the work directory.
`outputDirectoryExists( );`	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. The job must define the `SYS_EXT_supportOutputFiles` system property with a value of `output` to cause Oracle Enterprise Scheduler to create the output directory.
`isRequestWorkDirectoryShared( );`	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 is available to all subsequent stages of the request.
`getResolvedWorkDirectory( );`	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 automatically deletes the work directory at the end of request execution if the `RequestFileDirectory` is shared, or at the end of each stage (pre-processing, execution, update, post-processing) if the `RequestFileDirectory` is local.

Method	Description
`getResolvedOutputDirectory( );`	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.
`importOutputFiles( List<String> fileNames, CommitSemantics semantics );`	Returns an `ImportExportResult` 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 is not imported.
`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 is not 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 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 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 does not). 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 22-9 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 is committed.
`IgnoreErrors`	Attempt the operation on all files regardless of errors. If the handle is internal, it is committed.
`Transactional`	Stop the operation for all files when any file has an error. This is not valid with a user-provided handle.

Java Request Output Examples

Example 22-5 illustrates how to create output content using the `RequestOutput` API. The output content is created directly in the content store.

Example 22-6 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.

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 must be updated and imported.

```
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 =
helper.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(fileNamesList, CommitSemantics.IgnoreErrors);

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

Example 22-5 Creating Output Content using RequestOutput

```
import oracle.as.scheduler.request.ContentFactory;
import oracle.as.scheduler.request.ContentType;
import oracle.as.scheduler.request.RequestOutput;

class ExampleOutputContentCreator
{
    RuntimeService runtimeService = getRuntimeService();
    RuntimeServiceHandle handle;
    try {
      handle = runtimeService.open();
      RequestOutput requestOutput = ContentFactory.getRequestOutput(
            handle, requestId, ContentType.Text, “SampleOutput.txt");

      requestOutput.writeln(“Output data in sample output content.");
    }
    finally {
      runtimeService.close(handle);
    }
}
```

Example 22-6 Creating an Output File for Automatic Import

```
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
    }
}
```

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.

PL/SQL Package Support for Creating Output

Table 22-10 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.
`open_output_content( p_content_name in varchar2, p_content_type in integer ) return varchar2;`	Returns an opaque handle from opening the output content `p_content_name` for the request associated with the current session. 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_content_type` represents content type with one of the content type constants: `CONTENT_TYPE_TEXT` (value: 1) for text content. `CONTENT_TYPE_BINARY` (value: 2) for binary content. This returns an opaque handle that is passed to subsequent procedures that operate on that output content. 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 `write_text_content` or `write_ntext_content` procedures must be used to write data for text content, while the `write_binary_content` procedure must be used to write data for binary content. You should call close_content to free any resources associated with the handle returned by this method. The close should be done prior to transaction commit or rollback. NOTE: The content output support has DML semantics. The caller is responsible for the commit/rollback.

Method	Description
`write_ntext_content( p_content_handle in varchar2, p_data in nvarchar2 )` `write_text_content( p_content_handle in varchar2, p_data in varchar2 );`	Writes data as `p_data` to the output content associated with the given handle. These operations are supported only for `CONTENT_TYPE_TEXT` content type. `p_content_handle` is the output handle from a prior `open_output_content` call. NOTE: The content output support has DML semantics. The caller is responsible for the commit/rollback.
`write_binary_content( p_content_handle in varchar2, p_data in raw );`	Writes binary data as `p_data` to the output content associated with the given handle. This operation is supported only for `CONTENT_TYPE_BINARY` content type. `p_content_handle` is the output handle from a prior open_output_content call. NOTE: The content output support has DML semantics. The caller is responsible for the commit/rollback.
`close_content( p_content_handle in varchar2 );`	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_content_handle is the output handle from a prior `open_output_content` call. 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
`output_content_exists( p_content_name in varchar2 ) return boolean;`	Returns true if an output content entry having the specified name already exists for the request associated with the current session. `p_content_name` is the name of the output content entity.
`delete_output_content( p_content_name in varchar2 );`	Deletes the specified output content entry for the request associated with the current session. `p_content_name` is the name of the output content entity.

PL/SQL Output Creation Examples

Example 22-7 illustrates how to write text content into output, as well as how to write log entries along the way.

Example 22-8 illustrates how to write binary content for output.

Example 22-7 PL/SQL Output Creation Examples

```
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_ntext 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_Backtace...' || 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 22-8 Manual Export and Import of Request Output

```
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 is the 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;
/
```

Creating Request Output from a Process Job

You create output from 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 is defined for the job.

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

Creating Request Output from an EJB Job

See Request Logging and Output From an EJB Job for a description and examples about how to create output from an EJB job.

Creating Request Output from a Web Service Job

For synchronous and asynchronous web services, request output content is automatically created from the response message payload as described below. The output content is text data with the name `WebServiceJobOutput-REQUESTID.txt`, where `REQUESTID` is the request ID of the request.

For a synchronous web service, the response message payload is captured as the request output.

For an asynchronous web service, the callback/response message payload received at `EssWsJobAsyncCallbackService` is captured as the request output when the job status is `SUCCESS`.

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 How to Access the Runtime Service and Obtain a Runtime Service Handle.

Table 22-11 RuntimeService Methods for Handling Request Output

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

Method	Description
`closeContent( RuntimeServiceHandle handle, ContentHandle contentHandle )`	Closes the previously opened log or output content and releases the handle.

23

Oracle Enterprise Scheduler Security

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

This chapter includes the following sections:

- Introduction to Oracle Enterprise Scheduler Security
- Configuring Metadata Security for Oracle Enterprise Scheduler
- Configuring Data Security for Oracle Enterprise Scheduler
- Configuring Web Service Security for Oracle Enterprise Scheduler
- Configuring PL/SQL Job Security for Oracle Enterprise Scheduler
- Elevating Privileges for Oracle Enterprise Scheduler Jobs
- Configuring a Single Policy Stripe in Oracle Enterprise Scheduler

Introduction to Oracle Enterprise Scheduler Security

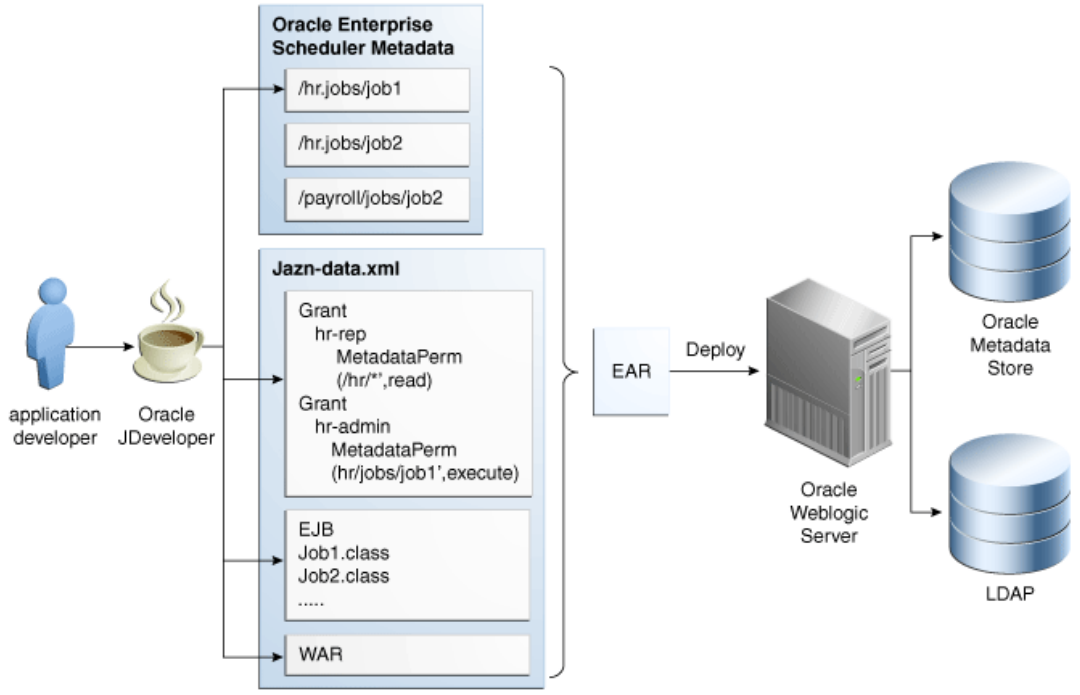
Oracle Enterprise Scheduler Security includes the following:

- Protected operations on `MetadataService`; protected by `MetadataPermission`, which enforces metadata access control. Access control on metadata objects. Only a privileged user may create, delete, and update job and schedule metadata. For more information see 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 Oracle Enterprise Scheduler Job Execution Security.

Oracle Enterprise Scheduler Metadata Access Control

At design time the metadata creator must 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 23-1 shows the metadata security summary.

Figure 23-1 Design Time Metadata Security for Oracle Enterprise Scheduler



Oracle Enterprise Scheduler Job Execution Security

During job submission, the user under whose permissions the job request is submitted is called the submitting user. At request execution time all user Java code including pre-processing, post-processing, Java jobs, and substitution, is run as the submitting user, retaining all roles and credentials.

If the job metadata specifies `SYS_RUNAS_APPLICAITONID`, however, the job runs under the elevated privileges of an application ID. For more information, see Elevating Privileges for Oracle Enterprise Scheduler Jobs.

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 must 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 must 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 them with a metadata object.

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. If you selected **ADF Authentication and Authorization** > **Form-Based Authentication** 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**.

Including Security Files in EAR File

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.

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.
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.

How to Define Principals for Security

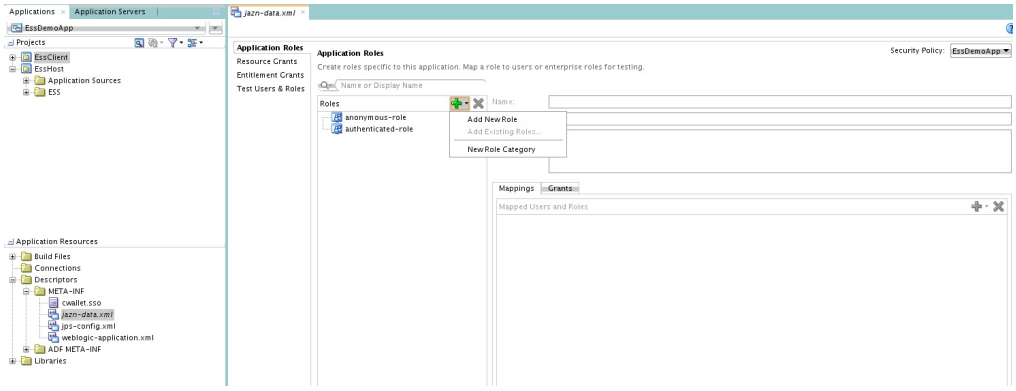
You need to define roles before the roles are used in Oracle Enterprise Scheduler security. There are two types of roles that may be defined:

- 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 create the application role:

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 **Applications Role** tab. Note, if the **Application Role** tab is not shown, try closing `jazn-data.xml` and then opening it again.
5. Click the **Add** button in the **Roles** list and choose **Add New Role**.
6. Set the name to `EssApplicationRole` in the **Name** field.

Figure 23-2 Creating the Application Role



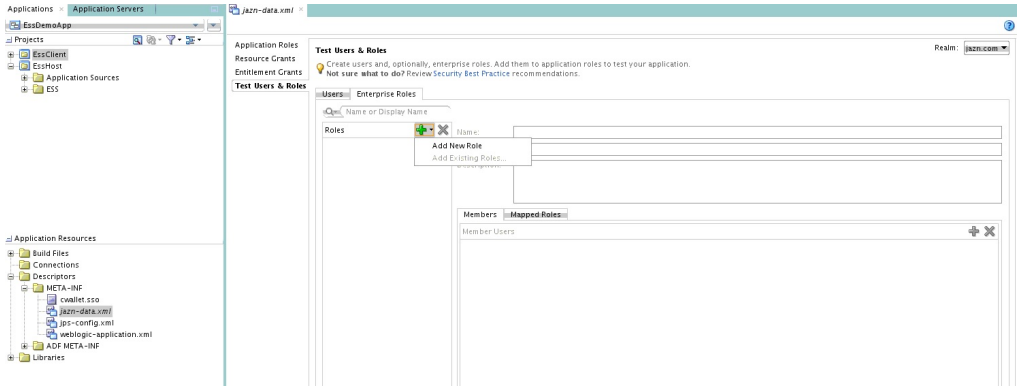
Creating Enterprise Role

To create the enterprise role:

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 **Test Users & Roles** tab. Note, if the **Test Users & Roles** tab is not shown, try closing `jazn-data.xml` and then opening it again.

5. Select the Enterprise Roles tab.
6. Click the **Add** button in the **Roles** list and choose **Add New Role**.
7. Set the name to `EssEnterpriseRole` in the **Name** field.

Figure 23-3 Creating the Enterprise Role



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 are done using the Oracle Enterprise Scheduler Oracle JDeveloper add-in.

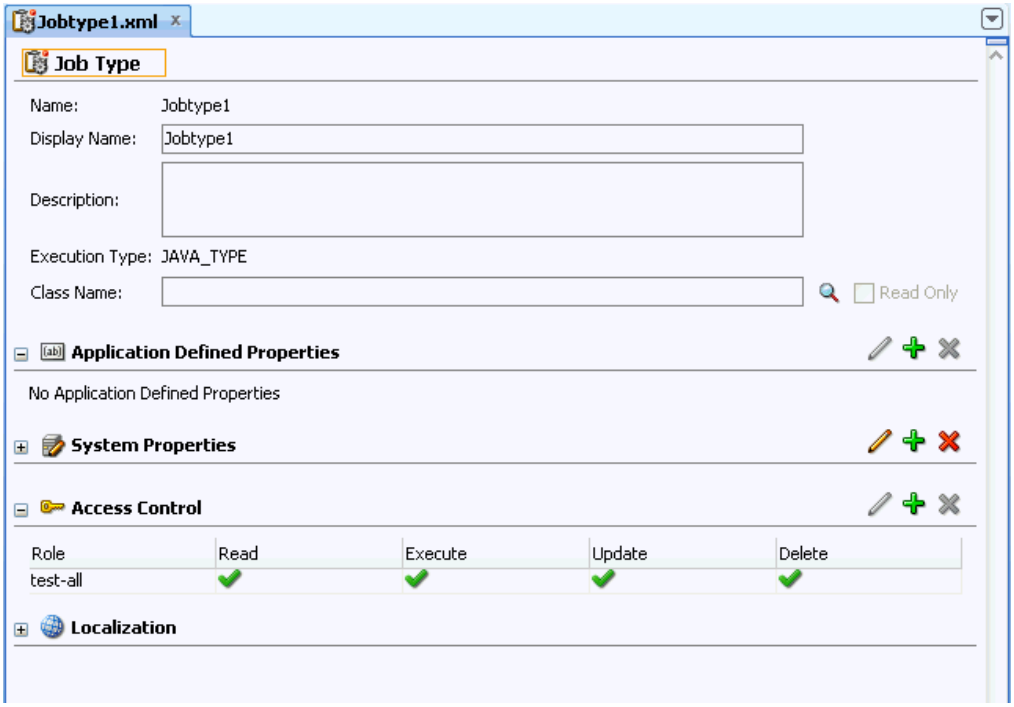
First, create any required Oracle Enterprise Scheduler metadata in an application using **File** > **New** > **Business Tier** > **Enterprise Scheduler Metadata**.

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 23-4.
6. Repeat for as many roles as needed.

Figure 23-4 Security Roles for Oracle Enterprise Scheduler Metadata



About MetadataPermission APIs

Grants for metadata are part of the class `oracle.as.scheduler.security.MetadataPermission`. The name, or target of the permission is based on the package, metadata object type, and name of the metadata object being protected; this identifier can be retrieved from `MetdataObjectId#toPermissionString()`.

Table 23-1 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 23-1 Grant Actions for Metadata Security

Action	Implies	Metadata Functions
READ	None	get<*Type*>(), query<*Type*>()
EXECUTE	READ	submitRequest()
CREATE	READ	add<*Type*>()
UPDATE	READ	update<*Type*>()
DELETE	READ	delete<*Type*>()

If you are submitting ad-hoc requests, you can have full wildcard ("*") permission with both EXECUTE and CREATE actions. When submitting ad-hoc requests, that is, using `submitRequest()` without certain `MetadataObjectIds`, you can grant permissions with the full wildcard ("*") name using the EXECUTE and CREATE actions.

What Happens When You Configure Metadata Security

Each time a user application calls a `MetdataService` or `RuntimeService` method, Oracle Enterprise Scheduler checks the current subject for privileges on the metadata accessed by the methods. For example, submitting a request requires `EXECUTE` permissions on the job definition or job set metadata object associated with the submission. Methods that change metadata, for example calling `updateJobDefinition()`, require `UPDATE` permissions.

For all `MetadataService` methods except queries, an exception is thrown when the user tries to access a metadata object for which the user does not have permission.

The `MetadataService` query methods have different behavior. When a user performs a query Oracle Enterprise Scheduler only returns metadata objects that have `READ` permission. Thus a user who has no permissions on metadata objects receives an empty list for all queries, but this user would not see an exception thrown due to lack of permissions.

The value of `SystemProperty.USER_NAME` is overwritten at submission time; the user cannot spoof an identity at submission time using `SystemProperty.USER_NAME`.

Configuring Data Security for Oracle Enterprise Scheduler

The Oracle Enterprise Scheduler standalone data security implementation is based on functional security permission checking using the `RuntimeDataPermission` class. By default, Oracle Enterprise Scheduler supports the following levels of data security enforcement:

- A user can only operate on the requests that he or she submits.
- A user can be granted permission to see *all* requests regardless of who submits them by assigning the user to the Admin, Operator, or Monitor roles.
- A user can view the output generated by the requests that he or she submits or executes.

If you want to add or change the default data security permissions, you must reconfigure the `jazn-data.xml` file as shown in How to Change Data Security Permissions.

How to Change Data Security Permissions

This section describes how configure the `jazn-data.xml` file to change the a job request's data security from the "out-of-the-box" default settings described in Configuring Data Security for Oracle Enterprise Scheduler.

The following is an example of a`jazn-data.xml` permission section that defines `RuntmeDataPermission`.

```
<permission>
  <name>definition like 'oracle.apps.ess.custom.soa.*',product like         "SOA
%",PROPERTY1=VALUE1</name>
  <class>oracle.as.scheduler.security.RuntimeDataPermission</class>
  <actions>ESS_REQUEST_READ,ESS_REQUEST_OUTPUT_READ</actions>
</permission
```

The permission conditions defined in the example above grant permission to a user to read the details of a job request and to read job request output if the job request matches the following criteria:

- Its job definition matches the pattern: `oracle.apps.ess.custom.soa.*` *and*
- If the value of the product field has "SOA"' *and*
- A user-defined property named `PROPERTY1` is set with the value of `VALUE1`

Conditions

Conditions are specified in the `<name>` element and use the "=" `and` "`like/LIKE`" operators. The "%" and "*" wildcard characters can be used with the "`like/LIKE`" operators. Note that you cannot use the "%" wildcard character in a field name that specifies a metadata object name, however, you can use it if the field name specifies other data such as products, and request categories. For example, the following examples work correctly:

- `definition LIKE 'oracle.apps.ess.custom.soa.*'`
- `requestCategory LIKE 'EssFineGrained%',workAssignment LIKE 'FineGrainedWA%',className LIKE '%BasicJavaJob%'`

While the following entry does *not* work correctly:

- `definition LIKE 'oracle.apps.ess.custom.soa.%'`

Conditions are specified as sets of key-value pairs with the following specific delimiter syntax.

- Key-value pairs in conditions are separated by the "," character.
- Whitespace characters cannot be specified directly before or after the "," delimiter.
- The value portion of a key-value condition can be unquoted, single-quoted, or double-quoted.
- Multiple conditions specified in the same element are interpreted as logically ANDed together.
- You can logically OR conditions by defining two or more permission resources for a same grantee.

The following rules apply to field names and `reqProp` keys:

- The field name must be the value represented by `fieldName` of `RuntimeService.QueryField`. For example, `QueryField.DEFINITION.fieldName()`.
- You cannot specify `ResultIndex` as the field name because there is no equivalent field in Request History View for `ResultIndex`. The Request History View is the Database View created out of the `Request_history` table. The SQL query of the standalone data security gets executed against this view.
- `reqProp` is a user property defined in Oracle Enterprise Scheduler job metadata or during request submissions. For example, `PARTITION_NAME='SOA'`.
- Table 23-2 shows the list of valid field names and their corresponding request history view column entries:

Table 23-2 Condition Query Fields and Their Corresponding Request History View Column Entries

Query Field	Request History View Column
QueryField.REQUESTID	requestid
QueryField.APPLICATION	application
QueryField.USERNAME	userName
QueryField.PRODUCT	product
QueryField.REQUEST_CATEGORY	requestCategory
QueryField.PRIORITY	priority
QueryField.NAME	name
QueryField.ABSPARENTID	absParentId
QueryField.TYPE	type
QueryField.DEFINITION	definition
QueryField.STATE	state
QueryField.SCHEDULE	schedule
QueryField.PROCESSSTART	processStart
QueryField.PROCESSEND	processEnd
QueryField.REQUESTEDSTART	requestedStart
QueryField.REQUESTEDEND	requestedEnd
QueryField.SUBMISSION	submission
QueryField.PARENTREQUESTID	parentRequestId
QueryField.WORKASSIGNMENT	workAssignment
QueryField.SCHEDULE	scheduled
QueryField.REQUESTTRIGGER	requesttrigger
QueryField.PROCESSOR	processor
QueryField.CLASSNAME	processor
QueryField.ELAPSEDTIME	elapsedtime
QueryField.WAITTIME	waittime
QueryField.SUBMITTER	submitter
QueryField.SUBMITTERGUID	submitterguid

Condition Examples

The following examples show validly specified conditions:

```
• fieldName=value
• fieldName like 'value%'
• fieldName like "value*"
• reqProp="value"
• reqProp LIKE "value*"
• PermissionName : <name>fieldName1=value1,fieldName2 like
'value2%',reqProp="value",reqProp like 'value*'</name>
```

Actions

Available actions include a combination of the following:

- ESS_REQUEST_READ
- ESS_REQUEST_UPDATE
- ESS_REQUEST_HOLD
- ESS_REQUEST_CANCEL
- ESS_REQUEST_LOCK
- ESS_REQUEST_RELEASE
- ESS_REQUEST_DELETE
- ESS_REQUEST_PURGE
- ESS_REQUEST_OUTPUT_READ
- ESS_REQUEST_OUTPUT_DELETE

Examples

Example 23-1 shows how permissions can be specified directly in the <permission> element.

Example 23-2 shows how permissions can be specified using permission sets.

Note:

The content defined in the <permission-sets> - <permission-set> - <member-resources> - <member-resource> - <resource-name> element must be the same as the content defined in the <resources> - <resource> - <name> element.

Example 23-1 PL/SQL Request Text Output

```
<app-roles>
  <app-role>
    <name>riyanu_soa_role</name>
    <class>oracle.security.jps.service.policystore.ApplicationRole</class>
    <members>
      <member>
        <name>riyanu_soa_group</name>
```

```
        <class>weblogic.security.principal.WLSGroupImpl</class>
      </member>
    </members>
  </app-role>
</app-roles>
<grant>
  <grantee>
    <description>Allow soa role to pass through</description>
    <principals>
      <principal>
        <class>oracle.security.jps.service.policystore.ApplicationRole</class>
        <name>riyanu_soa_role</name>
      </principal>
    </principals>
</grantee>
<permissions>
  <permission>
    <name>oracle.apps.ess.custom.soa.*</name>
    <class>oracle.as.scheduler.security.MetadataPermission</class>
    <actions>READ,EXECUTE,CREATE,DELETE,UPDATE</actions>
  </permission>
  <permission>
    <name>definition like oracle.apps.ess.custom.soa.*,
partitionName=SOA_PARTITION,product=SOA</name>
    <class>oracle.as.scheduler.security.RuntimeDataPermission</class>
    <actions>ESS_REQUEST_READ,ESS_REQUEST_UPDATE,
ESS_REQUEST_OUTPUT_READ</actions>
  </permission>
</permissions>
</grant>
```

Example 23-2 PLSQL Request Binary Output Example

```
<app-roles>
  <app-role>
    <name>riyanu_soa_role</name>
    <class>oracle.security.jps.service.policystore.ApplicationRole</class>
    <members>
      <member>
        <name>riyanu_soa_group</name>
        <class>weblogic.security.principal.WLSGroupImpl</class>
      </member>
    </members>
  </app-role>
</app-roles>
<resource-types>
  <resource-type>
    <name>ESSSOAMetadataResourceType</name>
    <display-name>ESSSOAMetadataResourceType</display-name>
    <description>ESS SOA Metadata Resource</description>
    <matcher-class>oracle.as.scheduler.security.MetadataPermission</matcher-
class>
    <actions-delimiter>,</actions-delimiter>
    <actions>create,read,update,delete,execute</actions>
  </resource-type>
  <resource-type>
    <name>ESSSOARequestResourceType</name>
    <display-name>ESSSOARequestResourceType</display-name>
    <description>Resource type for simple ESS request accesscontrol</description>
    <matcher-class>oracle.as.scheduler.security.RuntimeDataPermission     </
matcher-class>
```

```
    <actions-delimiter>,</actions-delimiter>
    <actions>ESS_REQUEST_READ,ESS_REQUEST_UPDATE,ESS_REQUEST_CANCEL,
ESS_REQUEST_DELETE,ESS_REQUEST_OUTPUT_READ</actions>
  </resource-type>
</resource-types>
  <resources>
    <resource>
       <name>oracle.apps.ess.custom.soa.*</name>
       <type-name-ref>ESSSOAMetadataResourceType</type-name-ref>
       <description>All SOA ESS metadata</description>
       <display-name>All SOA ESS metaddata</display-name>
    </resource>
    <resource>
       <name>definition like "oracle.apps.ess.custom.soa.*",
partitionName=SOA_PARTITION,product=SOA</name>
       <type-name-ref>ESSSOARequestResourceType</type-name-ref>
       <description>Any ESS Multi request</description>
       <display-name>Any ESS Multi request</display-name>
    </resource>
  </resources>
  <permission-sets>
    <permission-set>
      <name>READ_ALL_SOA_MULTI_METADATA_RIYANU</name>
      <display-name>Read privilege on all SOA ESS metadata
      </display-name>
      <description>Read privilege on all SOA ESS metadata</description>
      <member-resources>
        <member-resource>
          <type-name-ref>ESSSOAMetadataResourceType</type-name-ref>
          <resource-name>oracle.apps.ess.custom.soa.*</resource-name>
          <actions>create,read,execute</actions>
        </member-resource>
      </member-resources>
    </permission-set>
  <permission-set>
    <name>READ_ALL_ESS_SOA_REQUESTS_RIYANU</name>
    <display-name>All privileges on all ESS Requests</display-name>
    <description>Allow read, update, cancel, hold, delete all the ESS
requests</description>
    <member-resources>
      <member-resource>
        <type-name-ref>ESSSOARequestResourceType</type-name-ref>
        <resource-name>definition like
"oracle.apps.ess.custom.soa.*",         partitionName=SOA_PARTITION,product=SOA</
resourcename>
        <actions>ESS_REQUEST_READ,ESS_REQUEST_OUTPUT_READ</actions>
       </member-resource>
    </member-resources>
  </permission-set>
</permission-sets>
<jazn-policy>
  <grant>
    <grantee>
      <principals>
        <principal>
          <class>oracle.security.jps.service.policystore.ApplicationRole</class>
          <name>riyanu_soa_role</name>
        </principal>
      </principals>
    </grantee>
    <permission-set-refs>
```

```
      <permission-set-ref>
        <name>READ_ALL_SOA_MULTI_METADATA_RIYANU</name>
      </permission-set-ref>
      <permission-set-ref>
        <name>READ_ALL_ESS_SOA_REQUESTS_RIYANU</name>
      </permission-set-ref>
    </permission-set-refs>
  </grant>
</jazn-policy>
```

Configuring Web Service Security for Oracle Enterprise Scheduler

For information about securing the Oracle Enterprise Scheduler web service, see Using the Oracle Enterprise Scheduler Web Service .

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.

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.

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 runtime 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.

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 23-3 shows an `applicationStripe` configuration for the policy context `ESS_FUNCTIONAL_TEST_APP_STRIPE`.
4. If your application has a web module, configure the web module `JpsFilter` to use the same `applicationStripe` in the file `web.xml`. Example 23-4 shows a code sample.

Example 23-3 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>
```

Example 23-4 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>
```

What Happens When You Configure a Single Policy Stripe

At design time, an application stripe manifests as:

- An `<application>` element under the `<policystore>` element in the `jazn-data.xml` file.
- A node under the node `cn=<Weblogic.domain.name>,cn=JPSContext,cn=<root.node>`, such as `cn=ATGDemo,cn=base_domain,cn=JPSContext,cn=MY_Node`.

What Happens at Runtime

At runtime, an application stripe manifests as an instance of the class `oracle.security.jps.service.policystore.ApplicationPolicy`.