Oracle® Fusion Middleware

Developing Applications for Oracle Event Processing

12*c* Release (12.1.3)

E28538-08

November 2015

How to design and create Oracle Event Processing scalable applications to process streaming events.

ORACLE

Oracle Fusion Middleware Developing Applications for Oracle Event Processing, 12c Release (12.1.3)

E28538-08

Primary Author: Oracle® Corporation

Contents

Part II Deploy, Test, and Debug

13 Assemble and Deploy

14 Testing 1-2-3

Preface

This document describes how to create, deploy, and debug Oracle Event Processing applications.

Audience

This document is intended for developers who want to create Oracle Event Processing applications.

Related Documents

For more information, see the following:

- *Administering Oracle Event Processing*
- *Getting Started with Oracle Event Processing*
- *Getting Started with Oracle Edge Analytics*
- *Schema Reference for Oracle Event Processing*
- *Customizing Oracle Event Processing*
- *Using Visualizer for Oracle Event Processing*
- *Customizing Oracle Event Processing*
- *Developing Applications with Oracle CQL Data Cartridges*
- *Oracle CQL Language Reference for Oracle Event Processing*
- *Java API Reference for Oracle Event Processing*
- *Java API Reference for Oracle Edge Analytics*
- *Using Oracle Stream Explorer*
- *Getting Started with Oracle Stream Explorer*
- *Oracle Database SQL Language Reference* at: `http://docs.oracle.com/cd/E16655_01/server.121/e17209/toc.htm`
- SQL99 Specifications (ISO/IEC 9075-1:1999, ISO/IEC 9075-2:1999, ISO/IEC 9075-3:1999, and ISO/IEC 9075-4:1999)

- Oracle Event Processing Forum: http://forums.oracle.com/forums/forum.jspa?forumID=820.

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

The following table lists sections that have been added or changed. See Introduction to Application Development for a description of new features for this release.

For information about customizing adapters, event beans, and event store providers, see *Using Visualizer for Oracle Event Processing*.

The support for QuickFix Adapter has been deprecated in this release.

Sections	Description
Entire Guide	**Product renamed to Oracle Event Processing.**
Part I: Getting Started with Creating Oracle Event Processing Applications	This section is now in the book *Getting Started with Oracle Event Processing*.
Part II: Oracle Event Processing IDE for Eclipse	This section is replaced by chapters on Oracle JDeveloper in the book *Getting Started with Oracle Event Processing*.
Part III: Developing Oracle Event Processing Applications	This is now Part I: Developing Oracle Event Processing Applications
Chapter 8 Walkthrough: Assembling a Simple Application	This chapter is now in the book *Getting Started with Oracle Event Processing*.
Introduction to Application Development	This chapter has been rewritten to introduce Oracle Event Processing for developers.
Application and Resource Configuration	This chapter was added to provide information about the application configuration files and how to create them.

Part I

Application Development

Part I contains the following chapters:

- Introduction to Application Development
- Application and Resource Configuration
- Events and Event Types
- Adapters
- Channels
- Oracle CQL Processors
- Event Beans
- Cached Event Data
- Web Services
- Parameterized Applications.

1

Introduction to Application Development

An Oracle Event Processing application monitors and processes streaming data in real time. Streaming data flows into, through, and out of an application. Raw data flows into the application and is converted into events. Events flow through application stages for processing and filtering according to your application requirements. At the end, the application converts the processed and filtered events back to data in a format that is suitable for the destination, which could be, for example, storage, display on a web page, or further processing by another application.

If you are new to Oracle Event Processing application development, start with *Getting Started with Oracle Event Processing*. The getting started guide presents an overview of Oracle Event Processing, provides hands-on walkthroughs, and describes the sample applications. This guide explains how to create, configure, and deploy an Oracle Event Processing application with the components provided in the platform. If you want to build an application with customized adapters or event beans, see *Customizing Oracle Event Processing*.

This chapter includes the following sections:

1.1 New in this Release

The 12c release includes the following new features:

- QuickStart Installation that provides Oracle JDeveloper with the Oracle event Processing plug-in and an integrated Oracle WebLogic Server. The integrated Oracle WebLogic Server enables you to write applications that exchange event data with Oracle SOA Suite. See EDN Adapters.
- Oracle JDeveloper supports Oracle Event Processing application development. When you launch Oracle JDeveloper in the Studio Developer (All Features) role, it

provides a full feature set for creating Oracle Event Processing applications. See *Getting Started with Oracle Event Processing*.

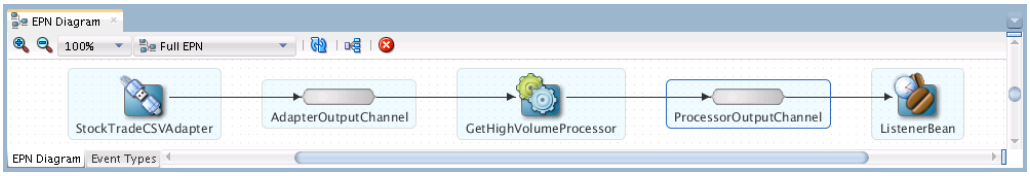
- Oracle CQL Features.
  - Oracle CQL Pattern components in Oracle JDeveloper. The Oracle CQL pattern components provide templates to help you form standard event queries, such as detecting missing events, averaging a series of events, partitioning by event pattern, and multiple forms of selections. *Getting Started with Oracle Event Processing*
  - Oracle CQL fault handler enables you to write code to handle faults that occur in code that does not have an inherent fault handling mechanism. See Oracle CQL Processors.
  - A CQL aggregation aggregates events into a Java collection so you can use the Collection APIs to manipulate the events. See *Oracle CQL Language Reference for Oracle Event Processing*.
  - Hadoop and Oracle NoSQLDB Big Data extensions. Hadoop is a data cartridge extension for an Oracle CQL processor to provide access to large quantities of data in a Hadoop distributed file system (HDFS). HDFS is a non-relational data store. NoSQLDB is a data cartridge extension for an Oracle CQL processor to provide access to large quantities of data in an Oracle NoSQL Database. The Oracle NoSQLDB Database stores data in key-value pairs. See *Developing Applications with Oracle CQL Data Cartridges*.
  - Spatial cartridge performance enhancements for better monitoring of geometric shapes that pass through the EPN. See *Developing Applications with Oracle CQL Data Cartridges*.
- Adapters. Oracle Event Processing provides a number of different kinds of inbound and outbound adapters to handle different types of data such as CSV, Event Delivery Network (EDN), REST, RMI, Oracle Business Rules (OBR), HTTP, and JMS. See Adapters.
- Testing tools. Testing tools include the load generator with the CSV inbound adapter to simulate a data feed, an event inspector service to debug Oracle CQL queries, and EPN shell commands to test the EPN from the command line. See Testing 1-2-3.
- Parameterized applications contain application metadata in parameters that is used to configure and customize the application. See Parameterized Applications.
- Java Persistence API (JPA) is available for you to use in your Oracle Event Processing Java code. The Oracle Event Processing installation includes the EclipseLink open source mapping and persistence framework to support the use of JPA in your applications. See EclipseLink, JPA, and Oracle Coherence.
- Custom data cartridges: The Data Cartridge SPI is now public so users and vendors can create cartridges to plug into Oracle Event Processing to extend Oracle CQL. See *Java API Reference for Oracle Event Processing* and *Developing Applications with Oracle CQL Data Cartridges*.
- JAXB support. Oracle Event Processing provides a simplified interface for using Java Architecture for XML Binding (JAXB) mapping capabilities in adapters and event beans. See JAXB Support.

1.2 EPN Diagram

Oracle Event Processing application development centers on the Event Processing Network (EPN) application model. The EPN diagram represents how event data flows into, through, and out of an Oracle Event Processing application. You assemble the EPN diagram in Oracle JDeveloper by selecting and configuring EPN components and providing logic as needed. In an EPN diagram, event data flows from left to right.

The figure shows the EPN diagram for the TradeReport application. Data enters the EPN through the StockTradeCSV adapter on the left, which handles data in the form of comma-separated values (CSV). The StockTradeCSVadapter logic translates the incoming CSV data into Oracle Event Processing events. The AdapterOutputChannel carries the newly generated events to the Oracle CQL processor.

The GetHighVolumeProcessor component queries the events as they stream through and selects stock trades that have a volume greater than 4000. The ProcessorOutputChannel component sends the selected events to the ListenerBean component, which prints their stock symbol and volume information to the command line.



Getting Started with Oracle Event Processing describes how to use Oracle JDeveloper to create Oracle Event Processing applications. A walkthough of the TradeReport application and a fraud detection application are included.

1.3 Component Configuration

When you develop an Oracle Event Processing application, you assemble and configure a network of components into an EPN. Each component has a role in processing the data. The following sections describe EPN components and their roles.

Events and Event Types

An event type is a data structure that defines the data contained in an event. Event types are the foundation of the EPN because they determine how event data funnels through the EPN and the operations that can be performed on it. When you start your application, the first thing to do is to create the event type or types for your EPN because you will need to configure components such as adapters, channels, relational database tables, and big data storage with the appropriate event type.

Adapters

Oracle Event Processing provides a selection of input and output adapters to accommodate every type of data that might flow into and out of the EPN. For example, you can access Java Message Service (JMS) objects, an HTTP Publish-Subscribe server, and financial market feeds. You can also develop your own adapters to integrate systems that are not supported by default. See *Using Visualizer for Oracle Event Processing*.

You configure adapters with an event type and other relevant configuration information. The specific configuration depends on whether the adapter handles event input or output and the source of the data. For example, in the TradeReport

application, the input CSV adapter configuration specifies the location of the CSV file, and values that tell the adapter when to start reading the CSV file and how long to wait between consecutive readings.

Channels

You configure a channel with an event type so that it can transfer events of that type to the next stage in the EPN that is appropriate for the given event type. A channel can represent either a stream or a relation.

A stream or relation channel inserts events into a collection and sends the resulting stream to the next EPN stage. Events in a stream can never be deleted from the stream. Events in a relation can be inserted into, deleted from, and updated in the relation. For insert, delete, and update operations, events in a relation must always be referenced to a particular point in time. See Streams and Relations for more information.

Oracle CQL Processors

You configure Oracle CQL processors with Oracle CQL query code to examine events as they pass through. The Oracle JDeveloper Components window provides CQL Patterns to facilitate the formation of Oracle CQL queries. The wizard for each CQL Pattern prompts you for the correct configuration data to ensure that you form a valid Oracle CQL query.

Beans

A bean defines application event logic written in the Java programming language that conforms to standard Spring-based beans.

An event bean is a Java class that implements logic to listen for and work on events. This type of Java class is called a listener Java class. A listener that receives events (event sink) might create new events when it finds a certain type of data and send the new events to the next stage for further processing. A listener event sink can also initiate other processes in the same or in another application based on the event data.

Spring beans are managed by the Spring framework, and are a good choice if you want to integrate your bean to an existing Spring deployment. Event beans use Oracle Event Processing conventions for configuring beans so that they are managed by the Oracle Event Processing server. With an event bean, for example, you get the support of Oracle Event Processing server features such as monitoring and event record and playback. You can use event record and playback to debug an application.

Caching

You can integrate a cache system with your Oracle Event Processing application to make a cache available as source or destination for data and event data that your application uses. Integrating a cache can provide access to relatively static data at a speed that is suited to an application that handles streaming data.

A cache is a temporary storage area for events that you can create to improve the overall performance of your Oracle Event Processing application. A cache is not necessary for the application to function correctly. To increase the availability of the events and increase the performance of their applications, Oracle Event Processing applications can publish to or consume events from a cache.

A caching system defines a named set of configured caches. Oracle Event Processing distributes the configuration for remote cache communications across multiple servers. The Spring context file supports caching configuration. Listeners that are configured with a Spring context file receive events from the cache.

Data-Related Components

Table: The Table component provides access to a relational database. You configure the Table component with an ID, event type, and a data source to feed specific events into a relational database table. Oracle Event Processing provides the Hadoop and NoSQLDB data cartridges for accessing big data storage.

Hadoop: A data cartridge extension for an Oracle CQL processor to access large quantities of data in a Hadoop distributed file system (HDFS). HDFS is a non-relational data store. The Oracle CQL processor provides the Oracle CQL query code for the big data access. You configure Hadoop with an ID, event type, the path to the database, and the file separator character.

NoSQLDB: A data cartridge extension for an Oracle CQL processor to access large quantities of data in an Oracle NoSQL Database. The Oracle NoSQLDB Database stores data in key-value pairs. The Oracle CQL processor provides the Oracle CQL query code for the big data access. You configure NoSQLDB with an ID, event type, store name, and store locations.

1.4 Streams and Relations

An Oracle Event Processing application handles events that arrive in a stream as raw event data. The raw event data enters the EPN through an adapter that converts the raw event data into an event. An event is an ordered set of values (tuple).

Events are similar to a table row in a relational database in that an event has a schema. The event schema defines the properties and types for each event value. Events are unlike a table row in a database in that a table row contains static data. In a stream of events, when an event arrives, including which event arrives before or after another event, can make a difference. Your application needs to be able to account for time and sequence.

For example, in an application that processes stock trades, events made up of stock symbol, price, last price, percentage change, and volume information would arrive one after the other in the order in which each trade was executed. Your application logic might look for trades of one stock that occurred immediately after trades of another.

In an event processing application, the sequence in which events occur in a stream is as important as the data types and values of each event property. Oracle Event Processing programming conventions reflect the importance of time and sequence.

Your code needs to discover which events are related to one another based on certain criteria, such as a shared stock symbol. Your code also needs to discover sequence patterns, such as trades within fifteen seconds of one another. To account for both the sequential and relational aspects of event data, Oracle Event Processing implements the concepts of streams and relations through low latency channels.

- A stream is a potentially infinite sequence of events where each event has its own time stamp. In a stream, the events must be ordered by time, one after the other, so that time stamps do not decrease from one event to the next. There can be events in a stream that have the same time stamp.

- In a relation, sequence might be unimportant. Instead, events in a relation are related because they meet certain criteria. For example, events in a relation might be the result of a query executed against a stream of stock trades, where the query looks for trade volumes above a particular level.

In a stream of stock trade events, the events arrive in sequence and each event has its own time stamp. To isolate the share price for trades that occurred within 5 seconds of

one another, configure an Oracle CQL processor to query the stream when it arrives from the channel with the following Oracle CQL code:

```
select price from StockTradeChannel [range 5 seconds]
```

Because the query uses the `[range 5 seconds]` window to isolate the events, the output of this query is a relation. Although the events returned from the query have time stamps, they are unordered in the relation. Because the incoming events are in a stream, the query executes continuously against every 5 seconds' worth of events as they pass into the Oracle CQL processor. As new events come along, those meeting the query terms are inserted into the relation, while those that do not meet the query terms are deleted from the relation.

This is important because the integrity of the order in a stream is important. Technically, a stream is a continuously moving and ordered set of events. In a stream, every event is inserted into the stream one after the other. When you get a subset of the stream from a CQL query, you no longer have the order. Before you pass a relation to the next stage in the EPN, you can convert the relation back into a stream with the `IStream` operator.

For more information, *Customizing Oracle Event Processing*.

1.5 Application Scalability and High Availability

A scalable Oracle Event Processing application incorporates Oracle Event Processing design patterns with implementation and configuration conventions to ensure that the application operation scales as the event load increases. You can achieve scalability and high availability by integrating application design patterns, server resources, and configuration conventions so that your deployed application continues to operate even in the event of software or hardware failures.

For more information, see the following:

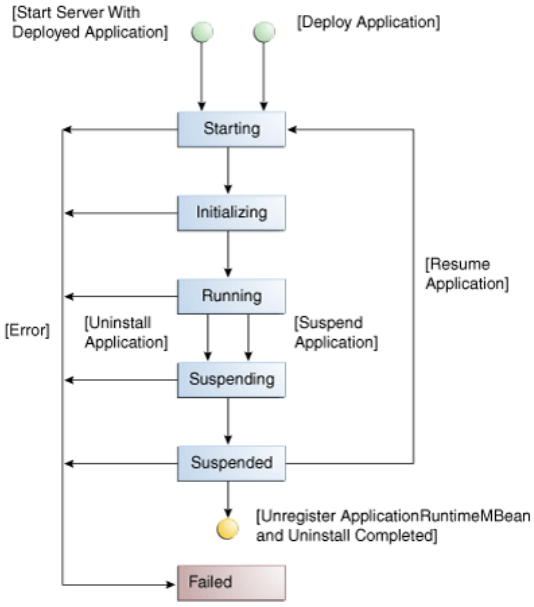
High Availability Applications.

Scalable Applications.

1.6 Application Life Cycle

Figure 1-1 shows a state diagram for the Oracle Event Processing application life cycle. In this diagram, the state names (`STARTING`, `INITIALIZING`, `RUNNING`, `SUSPENDING`, `SUSPENDED`, and `FAILED`) correspond to the `ApplicationRuntimeMBean` method `getState` return values. These states are specific to Oracle Event Processing. They are not OSGi bundle states.

Figure 1-1 Oracle Event Processing Application Life Cycle State Diagram



Note:

For information on Oracle Event Processing server life cycle, see *Administering Oracle Event Processing*.

This section describes the life cycle of an application deployed to the Oracle Event Processing server and the sequence of `com.bea.wlevs.ede.api` API callbacks. The information explains how Oracle Event Processing manages an application's life cycle so that you can better use the life cycle APIs in your application. For a description of these life cycle APIs (such as `RunnableBean` and `SuspendableBean`), see:

- API Overview
- *Java API Reference for Oracle Event Processing*.

The life cycle description is broken down into actions that a user performs, including those described in the following sections.

Install an Application or Start the Server with the Application Deployed

Oracle Event Processing performs the following actions:

1. Oracle Event Processing installs the application as an OSGI bundle. OSGI resolves the imports and exports, and publishes the service.
2. Oracle Event Processing creates beans (for both standard Spring beans and those that correspond to the Oracle Event Processing tags in the EPN assembly file). For each bean, Oracle Event Processing:
   - Sets the properties on the Spring beans. The `<wlevs:instance-property>` values are set on adapters and event-beans.
   - Injects appropriate dependencies into services specified by `@Service` or `@ServiceReference` annotations.

- Injects appropriate dependencies into static configuration properties.
- Calls the `InitializingBean.afterPropertiesSet` method.
- Calls configuration callbacks (`@Prepare`,`@Activate`) on Spring beans as well as factory-created stages.

  For more information, see Resource Access Configuration.

3. Application state is now `INITIALIZING`.
4. Oracle Event Processing registers the MBeans.
5. Oracle Event Processing calls the `ActivatableBean.afterConfigurationActive` method on all `ActivatableBeans`.
6. Oracle Event Processing calls the `ResumableBean.beforeResume` method on all `ResumableBeans`.
7. For each bean that implements `RunnableBean`, Oracle Event Processing starts it running in a thread.
8. Application state is now `RUNNING`.

Suspend the Application

Oracle Event Processing performs the following actions:

1. Oracle Event Processing calls the `SuspendableBean.suspend` method on all `SuspendableBeans`.
2. Application state is now `SUSPENDED`.

Resume the Application

Oracle Event Processing performs the following actions:

1. Oracle Event Processing calls the `ResumableBean.beforeResume` method on all `ResumableBeans`
2. For each bean that implements `RunnableBean`, starts it running in a thread.
3. Application state is now `RUNNING`.

Uninstall the Application

Oracle Event Processing performs the following actions:

1. Oracle Event Processing calls the `SuspendableBean.suspend` method on all `SuspendableBeans`.
2. Oracle Event Processing unregisters MBeans.
3. calls the `DisposableBean.dispose` method on all `DisposableBeans`.
4. Oracle Event Processing uninstalls application bundle from OSGI.

Updating the application

This is equivalent to first uninstalling an application and then installing it again.

See those user actions in this list.

Call Methods of Stream and Relation Sources and Sinks

You cannot call a method on a stream or relation source or sink from a life cycle callback because components might not be ready to receive events until after these phases of the application life cycle completes.

For example, you cannot call `StreamSender` method `sendInsertEvent` from a life cycle callback such as such as `afterConfigurationActive` or `beforeResume`.

You can call a method on a stream or relation source or sink from the run method of beans that implement `RunnableBean`.

See Event Beans.

1.7 API Overview

The APIs enable you to programmatically implement functionality for all aspects of Oracle Event Processing applications as described in this documentation set. This section presents an overview of the API packages in terms of their intended usages and includes cross-references to where you can learn more.

For the full reference documentation (Javadocs) for all classes and interfaces, see *Java API Reference for Oracle Event Processing*. See also *Getting Started with Oracle Event Processing* for sample code that uses the Oracle Event Processing APIs.

- Configuration
- Adapters
- ChannelsEvent Repositories
- Event-Driven Environment
- Event Bean Life Cycle
- JAXB
- Caching
- Cache Loader
- Cluster Group Management
- Management Beans
- High Availability
- Testing and Utility Tools
- Cartridge Framework
- Spring Support.

Configuration

The `com.bea.wlevs.configuration` package provides interfaces to activate, prepare, and roll back configuration objects. When you implement the `Prepare` interface, provide a method that accepts, checks, and stores a configuration object. The Java type of the configuration object is determined by JAXB. By default, the Java class name is the same as the name of the XML Schema complex type that describes the configuration data for the applicable stage. See the `/Oracle/Middleware/my_oep/`

`oep/ xsd/wlevs_application_config.xsd` schema for schema details. See also Application and Resource Configuration.

Adapters

Oracle Event Processing provides several packages that provide interfaces and classes for managing adapter behavior. See Adapters.

Packages:

- `com.bea.wlevs.adapters.httppubsub.api` package provides interfaces for converting inbound JavaScript Object Notation (JSON) messages to event types and back again. To customize the way inbound and outbound JSON messages are converted to an event type and back to JSON format, create a custom converter bean and use this API.
- `com.bea.wlevs.adapters.httppubsub.support` package provides classes for establishing a connection to an HTTP publish-subscribe server.
- `com.bea.wlevs.adapters.jms.api` package provides interfaces for converting inbound JMS messages to event types and back again. If you want to customize the way inbound and outbound JMS messages are converted to an event type and back, create a custom converter bean. and use this API.
- `com.bea.wlevs.ede.api`: package provides interfaces for creating custom adapters. See Oracle Fusion Middleware Customizing Oracle Event Processing Components.

Channels

The `com.bea.wlevs.channel` package provides an interface for implementing event partitioning and a class for managing the number of events in channels. See Channels and Scalable Applications.

Event Repositories

To manage events and event types, Oracle Event Processing uses an event store repository and an event type repository. The event store repository persists the event and the event type repository persists the event type.

Packages:

- `com.bea.wlevs.eventstore` package provides interfaces and classes to manage the event store repository. See *Using Visualizer for Oracle Event Processing*.
- `com.bea.wlevs.ede.api` package provides the `EventTypeRepository` interface to manage the event type repository. See Events and Event Types.

Event-Driven Environment

The `com.bea.wlevs.ede.api` package provides interfaces for creating and customizing Oracle Event Processing application code that responds to events. The package provides interfaces for creating event beans and adapters and making them event sinks and event sources. Other interfaces in this package enable you to manage all aspects of how events flow through the EPN, such as event creation, event flow through channels, event metadata and properties, the event type repository, external data sources, EPN stages, fault handling, event bean life cycle, and so on.

For sample Java code that uses some of these APIs, see Events and Event Typesand Event Beans. See also Resource Access Configuration for information about using Oracle Event Processing annotations and deployment XML to configure resource injection.

Event Bean Life Cycle

The `com.bea.wlevs.ede.api` package also enables control over event bean life cycle. You can manage event bean initialization, configure dynamic activation, use threading, suspend and resume processing, and release resources when the application is undeployed. See Application Life Cycle for information about the event bean and application life cycles.

Note that the Spring framework implements similar bean life cycle interfaces. However, the equivalent Spring interfaces do not allow you to manipulate beans that were created by factories, while the Oracle Event Processing interfaces do.

JAXB

Oracle Event Processing provides a simplified interface for using Java Architecture for XML Binding (JAXB) mapping capabilities in adapters and event beans to marshall and unmarshall event data between XML and Java objects. See JAXB Support.

Packages:

- `com.oracle.cep.mappers.api` package provides interfaces for marshalling and unmarshalling event data for most applications requirements.
- `com.oracle.cep.mappers.jaxb` package provides interfaces that provide specialized method signatures for marshalling and unmarshalling.

Caching

You can configure a caching system so that applications have ready access to event data. The caches in the system can be a combination of Oracle Coherence distributed caching, Oracle Event Processing local caching, and caching solutions provided by third parties. You can access the events in the caches with Oracle CQL and Java classes. See Cached Event Data.

Packages:

- `com.bea.wlevs.cache.spi` package provides interfaces that enable you to create a caching system that can be used by Oracle Event Processing applications.
- `com.bea.wlevs.cache.spi.coherence` package provides interfaces that enable you to extend the caching system to include Oracle Coherence caching.

Cache Loader

The `com.oracle.cep.cacheloader` package provides the `CsvCacheLoader` class for loading CSV events into a Coherence cache. See Cached Event Data.

Cluster Group Management

The `com.bea.wlevs.ede.api.cluster` package provides interfaces for managing server groups within multiserver domains (clusters). You can get information about the configuration, implement event beans and adapters to listen for cluster membership changes, set the group name for the containing EPN, and get information about a group server. See *Administering Oracle Event Processing*.

Management Beans

Management beans (MBeans) enable you to programmatically access configuration and runtime information to perform tasks. There are two types of MBeans (tasks): configuration and run time. Configuration MBeans contain information about EPN component configuration. Run time MBeans contain information about component throughput and latency. See *Administering Oracle Event Processing*.

Packages:

- `com.bea.wlevs.management` package contains interfaces for managing constants used by client applications and to provide a super-interface for all Oracle WebLogic Event Server MBeans.
- `com.bea.wlevs.management.configuration` package provides interfaces for managing applications, adapters, caches, configuration, Oracle CQL processors, event beans, stages, streams, and table sinks and sources.
- `com.bea.wlevs.management.diagnostic` package provides interfaces for managing diagnostic profiles. A diagnostic profile is an XML file that contains application stage information for testing throughput and latency. See *Using Visualizer for Oracle Event Processing*.
- `com.bea.wlevs.management.diagnostic.notification` package provides a class for wrapping diagnostic change notifications sent by background probes. See *Using Visualizer for Oracle Event Processing*.
- `com.bea.wlevs.diagnostic` package provides interfaces and classes for listening for newly deployed applications and removed applications. When applications are deployed and undeployed a profile manager (group of diagnostic profiles) is also created and removed and corresponding profile manager events are issued.
- `com.bea.wlevs.management.runtime` package provides interfaces for getting runtime information about the application, the application Oracle CQL processors, the domain, the server, and EPN stages.
- `com.bea.wlevs.monitor` package provides interfaces for monitoring the throughput and latency of application endpoints in the event server.
- `com.bea.wlevs.monitor.management` package provides interfaces for receiving monitoring metrics for an application stage and for monitoring latency between endpoints in the EPN.
- `com.bea.wlevs.deployment.mbean` package provides interfaces to manage application deployment.
- `com.bea.wlevs.eventinspector.management` package provides interfaces and classes for controlling the behavior of event tracing and event injection. See Testing 1-2-3.
- `com.oracle.cep.cluster.ha.adapter.management` package provides interfaces and classes for managing JMX communications in a high availability environment.

High Availability

Oracle Event Processing provides application design patterns and high availability adapters, to enable you to increase the backup and failover processing capabilities of your applications. See High Availability Applications.

Packages:

- `com.oracle.cep.cluster.ha.adapter` package provides interfaces and classes for queue trimming.

- `com.oracle.cep.cluster.ha.adapter.inbound` package provides classes for creating a high availability broadcast inbound adapter. This adapter is for applications that use system time and need to be highly available.
- `com.oracle.cep.cluster.ha.adapter.management` package provides interfaces and classes for managing JMX communications in a high availability environment.
- `com.oracle.cep.cluster.ha.adapter.runtime` package provides interfaces and class implementations for managing JMX interfaces to other high availability interfaces.
- `com.oracle.cep.cluster.ha.api` package provides interfaces and classes for simple fail over functionality.
- `com.oracle.cep.cluster.hagroups` package provides interfaces and classes for creating event beans and adapter that listen for property group membership changes, make the changes available, and enable subscriptions to broadcast group members.
- `com.oracle.cep.cluster.hagroups.runtime` package provides interfaces and classes to get notification group information.

Testing and Utility Tools

Oracle Event Processing provides different ways to test your application depending on what and how you want to test. See Testing 1-2-3.

Packages:

- `com.bea.wlevs.eventinspector.management` package provides interfaces and classes for managing event tracing and injection.
- `com.oracle.cep.shell` package provides interfaces and classes for programmatically invoking commands for testing Oracle Event Processing applications.
- `com.bea.wlevs.util` package provides interfaces and classes for marking methods as requiring an OSGi service reference, getting and setting error messages, parsing parameters, returning OSGi importer services cardinality, and loading a service class.

Cartridge Framework

The com.oracle.cep.cartridge package provides interfaces and classes that form the Data Cartridge Framework. The Data Cartridge Framework is a service provider interface (SPI) that enables users and vendors to create cartridges to extend Oracle CQL functionality. See *Developing Applications with Oracle CQL Data Cartridges*.

Spring Support

The `com.bea.wlevs.spring.support` package provides interfaces and classes for using Spring functionality in Oracle Event Processing applications.

1.8 Spring Framework

The Spring Framework provides Java-based APIs and a configuration model that you can use to create portable and flexible enterprise applications.

For more information about Spring:

- Spring Framework API 3.1.1:http://docs.spring.io/spring/docs/3.1.1.RELEASE/javadoc-api/
- The Spring Framework - Reference Documentation 3.1:http://docs.spring.io/spring/docs/3.1.1.RELEASE/spring-framework-reference/html/

1.9 OSGi Service Platform

The OSGi Service Platform provides a dynamic application execution environment where you can install, update, ore remove OSGI bundles (modules) dynamically

For more information about OSGi:

- OSGi Release 4 Service Platform Javadoc: http://www.osgi.org/Release4/Javadoc
- OSGi Release 4 Core Specification: http://www.osgi.org/osgi_technology/download_specs.asp?section=2#Release4

Service Annotations

Use the `com.bea.wlevs.util.Service` (`@Service`) annotation to specify a component method that is injected with an OSGi service reference. You typically add this annotation to JavaBean setter methods where needed. The `@Service` annotation has the following attributes.

Table 1-1 Attributes of the com.bea.wlevs.util.Service JWS Annotation Tag

Name	Description	Data Type	Required ?
`serviceBeanName`	The name of the bean that backs the injected service. Can be null.	`String`	No.
`cardinality`	Valid values for this attribute are: • `ServiceCardinality.C0__1` • `ServiceCardinality.C0__N` • `ServiceCardinality.C1__1` • `ServiceCardinality.C1__N` Default value is `ServiceCardinality.C1__1.`	`enum`	No.
`contextClassloader`	Valid values for this attribute are: • `ServiceClassloader.CLIENT` • `ServiceClassloader.SERVICE_PROVIDER` • `ServiceClassloader.UNMANAGED` Default value is `ServiceClassloader.CLIENT`.	`enum`	No.
`timeout`	Timeout for service resolution in milliseconds. Default value is 30000.	`int`	No.
`serviceType`	Interface (or class) of the service to be injected Default value is `Service.class`.	`Class`	No.

Table 1-1 (Cont.) Attributes of the com.bea.wlevs.util.Service JWS Annotation Tag

Name	Description	Data Type	Required ?
`filter`	Specifies the filter used to narrow service matches. Value may be `null`.	`String`	No.

The following example shows how to use the `@Service` annotation. For another example, see Access the Event Type Repository.

```
@Service(filter = "(Name=StockDs)")
public void setDataSourceService(DataSourceService dss) {
    initStockTable(dss.getDataSource());
}
```

2

Application and Resource Configuration

An Oracle Event Processing EPN has two types of configuration files: assembly files and component configuration files. The assembly file is a context file that describes the EPN diagram stages and structure. The configuration file describes component configuration and the dynamic parameters of the EPN stages. Dynamic parameters are parameters that can be changed at runtime through the Oracle Event Processing Visualizer or programmatically through the JMX APIs.

This chapter includes the following sections:

2.1 Application Configuration

Oracle Event Processing application configuration settings are stored in XML files that are based on standard schemas. When you install Oracle Event Processing, the XSD files for the schemas are installed in the *`Oracle/Middleware`*`/oep/xsd` directory.

By default, Oracle JDeveloper generates one assembly file named `<Project-Name>.context.xml`, and one default configuration file named `processor.xml`. An application can have one or more assembly files and one or more configuration files. You decide how many configuration files to use and what to name them when you build the EPN. Your project must have one configuration file named `prcessor.xml` to contain the Oracle CQL processor configuration settings.

When you create components such as adapters, the `processor.xml` file displays as the default configuration file in the new component wizard. If you take the default, the component configuration information is stored in the default `procesor.xml`. To put all of your adapter configurations in one file named `adapter.xml`, change `processor.xml` to `adapters.xml` in the wizard.

In the component configuration wizard, if you specify a new file name such as `adapters.xml`, but use only default settings, Oracle JDeveloper does not generate the new file because there are no configuration settings to store in it. You can either create the component again with a custom setting or use the File menu to create a new empty configuration file. See *Getting Started with Oracle Event Processing*.

The assembly and configuration files are stored in the following locations within your project:

- Assembly files: `<Project-Name>/META-INF/spring/*.xml`.

- Configuration files: `<Project-Name>/META-INF/wlevs/*.xml`.

You can modify the configuration by editing the application assembly file or by editing the component configuration file. You can edit anything you want in the files, but you have to be careful to keep the assembly file ID value consistent with the configuration file name value. If you change the ID value in the assembly file, you have to change the name value in the configuration file to match, and vice versa. You can change any other information in one file only. Oracle JDeveloper uses the ID and name value pairing to keep the information in the application assembly and component configuration files synchronized.

The following components have a configuration file that defaults to `processor.xml`. Oracle CQL patterns must be placed in the `processor.xml` file, but all other components in this list can use a configuration file by another name.

- All adapters
- Channels
- Oracle CQL Patterns
- Local Cache System
- Cache
- RMIOutbound extension

The Coherence Cache System has a default `coherence-cache-` file. You can change the name of this file.

Component configuration files are deployed as part of the Oracle Event Processing application bundle. You can later update this configuration at runtime using Oracle Event Processing Visualizer, the `wlevs.Admin` utility, or by manipulating the appropriate JMX MBeans.

2.2 Assembly File Structure

The `spring-wlevs-v12_1_3_0.xsd` schema file describes the EPN assembly file structure. This schema file is installed in the *`Oracle/Middleware`*`/oep/xsd` directory. See *Schema Reference for Oracle Event Processing*.

The EPN assembly file has a top-level root element named `beans` that contains a sequence of sub-elements. Each individual sub-element contains the configuration data for an Oracle Event Processing component.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:osgi="http://www.springframework.org/schema/osgi"
    xmlns:wlevs="http://www.bea.com/ns/wlevs/spring"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/osgi
        http://www.springframework.org/schema/osgi/spring-osgi.xsd
        http://www.bea.com/ns/wlevs/spring
        http://www.bea.com/ns/wlevs/spring/spring-wlevs-v12_1_3_0.xsd">

    <wlevs:event-type-repository>
        <wlevs:event-type type-name="HelloWorldEvent">
            <wlevs:class>com.bea.wlevs.event.example.helloworld.HelloWorldEvent</
```

```
wlevs:class>
        </wlevs:event-type>
    </wlevs:event-type-repository>

    <wlevs:adapter id="helloworldAdapter"
        class="com.bea.wlevs.adapter.example.helloworld.HelloWorldAdapter" >
        <wlevs:instance-property name="message" value="HelloWorld - the current time
is:"/>
    </wlevs:adapter>

    <wlevs:channel id="helloworldInputChannel" event-type="HelloWorldEvent" >
        <wlevs:listener ref="helloworldProcessor"/>
        <wlevs:source ref="helloworldAdapter"/>
    </wlevs:channel>

    <wlevs:processor id="helloworldProcessor" />

    <wlevs:channel id="helloworldOutputChannel"
        event-type="HelloWorldEvent" advertise="true">
        <wlevs:listener>
            <bean class="com.bea.wlevs.example.helloworld.HelloWorldBean"/>
        </wlevs:listener>
        <wlevs:source ref="helloworldProcessor"/>
    </wlevs:channel>
</beans>
```

2.2.1 Nested Stages in an EPN Assembly File

When you define a child stage within a parent stage in an EPN, the child stage is said to be nested. Only the parent stage can specify the child stage as a listener.

The following example shows the EPN assembly source in which `HelloWorldBean` is nested within the `helloworldOutputChannel`. Only the parent `helloworldOutputChannel` may specify the nested bean as a listener.

```
<wlevs:adapter id="helloworldAdapter"
class="com.bea.wlevs.adapter.example.helloworld.HelloWorldAdapter" >
    <wlevs:instance-property name="message" value="HelloWorld - the current time
is:"/>
</wlevs:adapter>

<wlevs:channel id="helloworldInputChannel" event-type="HelloWorldEvent" >
    <wlevs:listener ref="helloworldProcessor"/>
    <wlevs:source ref="helloworldAdapter"/>
</wlevs:channel>

<wlevs:processor id="helloworldProcessor" />

<wlevs:channel id="helloworldOutputChannel" event-type="HelloWorldEvent"
advertise="true">
    <wlevs:listener>
        <bean class="com.bea.wlevs.example.helloworld.HelloWorldBean"/>
    </wlevs:listener>
    <wlevs:source ref="helloworldProcessor"/>
</wlevs:channel>
```

Alternatively, you can define this EPN so that all stages are nested as Example 2-1 shows. The `helloworldAdapter`, the outermost parent stage, is the only stage accessible to other stages in the EPN.

Example 2-1 EPN Assembly File with All Stages Nested

```
<wlevs:adapter id="helloworldAdapter"
class="com.bea.wlevs.adapter.example.helloworld.HelloWorldAdapter" >
    <wlevs:instance-property name="message"
        value="HelloWorld - the current time is:"/>
    <wlevs:listener>
        <wlevs:channel id="helloworldInputChannel" event-type="HelloWorldEvent" >
            <wlevs:listener>
                <wlevs:processor id="helloworldProcessor">
                    <wlevs:listener>
                        <wlevs:channel id="helloworldOutputChannel"
                            event-type="HelloWorldEvent">
                            <wlevs:listener>
                                <bean
                                    class="com.bea.wlevs.example.helloworld.HelloWorl
dBean"/>
                            </wlevs:listener>
                        </wlevs:channel>
                    </wlevs:listener>
                </wlevs:processor>
            </wlevs:listener>
        </wlevs:channel>
    </wlevs:listener>
</wlevs:adapter>
```

2.2.2 Foreign Stages in an EPN Assembly File

You can refer to a stage that is in another Oracle Event Processing application. A stage from another application is considered a *foreign stage*. You do this by `id` attribute when you define both the source and target stage in the same application.

Note:

You cannot connect an Oracle CQL processor stage to a channel that is a foreign stage.

To refer to a stage you define in a different application, you use the following syntax:

`FOREIGN-APPLICATION-NAME``:`*`FOREIGN-STAGE-ID`*

Where *`FOREIGN-APPLICATION-NAME`* is the name of the application in which you defined the foreign stage and *`FOREIGN-STAGE-ID`* is the `id` attribute of the foreign stage.

The following example shows how the reference in `application1` to the foreign stage `HelloWorldBeanSource` that you define in application `application2`.

```
<wlevs:stream id="helloworldInstream" >
    <wlevs:listener ref="helloworldProcessor"/>
    <wlevs:source ref="application2:HelloWorldBeanSource"/>
</wlevs:stream>

<wlevs:event-bean id="HelloWorldBeanSource"
    class="com.bea.wlevs.example.helloworld.HelloWorldBeanSource"
    advertise="true"/>
```

The following stages cannot be foreign stages:

- Cache

When creating Oracle Event Processing applications with foreign stages, you must consider foreign stage dependencies when assembling, deploying, and redeploying your application. For more information, see Reference Foreign Stages.

2.3 Component Configuration File Structure

The `wlevs_application_config.xsd` schema file describes the structure of component configuration files. When you install Oracle Event Processing, XSD files such as this one are included in the directory *`Oracle/Middleware/oep/xsd`*.

This XSD schema imports the following schemas:

- `wlevs_base_config.xsd`: Defines common elements that are shared between application configuration files and the server configuration file
- `wlevs_eventstore_config.xsd`: Defines event store-specific elements.
- `wlevs_diagnostic_config.xsd`: Defines diagnostic elements.

See *Schema Reference for Oracle Event Processing*.

The structure of application configuration files is as follows. There is a top-level root element named `config` that contains a sequence of sub-elements. Each individual sub-element contains the configuration data for an Oracle Event Processing component (Oracle CQL processor, channel, or adapter). For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<n1:config xmlns:n1="http://www.bea.com/ns/wlevs/config/application"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <processor>
        <name>helloworldProcessor</name>
        <rules>
            <query id="helloworldRule">
                <![CDATA[ select * from helloworldInputChannel [Now] >
            </query>
        </rules>
    </processor>
    <channel>
        <name>helloworldInputChannel</name>
        <max-size>10000</max-size>
        <max-threads>2</max-threads>
    </channel>
    <channel>
        <name>helloworldOutputChannel</name>
        <max-size>10000</max-size>
        <max-threads>2</max-threads>
    </channel>
</n1:config>
```

2.4 Component and Server Configuration

Use the `ConfigurationPropertyPlaceholderConfigurer` class to reference existing configuration file properties, in both component configuration and server configuration files, using a symbolic placeholder. This allows you to define a value in one place and refer to that one definition rather than hard-coding the same value in many places.

You might want to do this if you want to configure Java Message Service (JMS) without hard-coding JMS information such as the factory name in the assembly file for your Oracle Event Processing application.

Use the `com.bea.wlevs.spring.support.ConfigurationPropertyPlaceholderConfigurer` class, to create a JMS adapter and provide placeholders for the server connection factory name, user name, password, and the location to a separate file that contains the actual factory name, user name, and password values. The `ConfigurationPropertyPlaceholderConfigurer` class is implemented on top of the Spring framework.

The server configuration file is used by Oracle Event Processing server administrators. This file contains configuration information that is specific to a domain, and is located in `/Oracle/Middleware/my_oep/user_projects/domains/<domain_name>/<server_name>/config/`.

To use reference existing configuration file properties, insert a `ConfigurationPropertyPlaceholderConfigurer` bean in the assembly file for your project as shown below.

```
<bean class="com.bea.wlevs.spring.support.ConfigurationPropertyPlaceholderConfigurer"/>
```

For complete details, see the `com.bea.wlevs.spring.support.ConfigurationPropertyPlaceholderConfigurer` class in the *Java API Reference for Oracle Event Processing*.

2.5 Resource Access Configuration

Because Oracle Event Processing applications are low latency high-performance event-driven applications, they run on a lightweight container and are developed with a POJO-based programming model. In POJO (Plain Old Java Object) programming, business logic is implemented in the form of POJOs, and then injected with the services they need. This is popularly called *dependency injection*. The injected services can range from those provided by Oracle Event Processing services, such as configuration management, to those provided by another Oracle product such as Oracle Kodo, to those provided by a third party.

By using Oracle Event Processing and standard Java annotations and deployment XML, you can configure the Oracle Event Processing Spring container to inject resources (such as data sources or persistence managers, and so on) into your Oracle Event Processing application components.

The Spring container typically injects resources during component initialization. However, it can also inject and re-inject resources at runtime and supports the use of JNDI lookups at runtime.

Oracle Event Processing supports the following types of resource access:

- Resource Access Annotations
- Static Resource Injection
- Dynamic Resource Injection
- Dynamic Resource Lookup Using JNDI
- Resource Name Resolution.

In the following sections, consider the example resource that Example 2-2 shows. This is a data source resource named `StockDS` that you specify in the Oracle Event Processing server file.

Example 2-2 Sample Resource: Data Source StockDS

```
<config ...>
    <data-source>
        <name>StockDs</name>
        ...
        <driver-params>
            <url>jdbc:derby:</url>
            ...
        <driver-params>
    </data-source>
...
</config>
```

2.5.1 Resource Access Annotations

Use the `javax.annotation.Resource` (`@Resource`) annotation to configure resource access at design time and the corresponding deployment XML to override this configuration at deploy time.

2.5.2 Static Resource Injection

Static resource injection refers to the injection of resources during the initialization phase of the component life cycle. Once injected, resources are fixed, or static, while the component is active or running.

You can configure static resource injection using:

- Static Resource Names
- Dynamic Resource Names.

2.5.2.1 Static Resource Names

When you configure static resource injection using static resource names, the resource name you use in the `@Resource` annotation or Oracle Event Processing assembly XML file must exactly match the name of the resource as you defined it. The resource name is static in the sense that you cannot change it without recompiling.

To configure static resource injection using static resource names at design time, you use the standard `javax.annotation.Resource` annotation as shown in the example below.

To override design time configuration at deploy time, you use Oracle Event Processing assembly file XML.

In the following examples the resource name `StockDs` exactly matches the name of the data source in the Oracle Event Processing server file.

```
< wlevs:event-bean id="simpleBean" class="...SimpleBean"/>
    <wlevs:resource property="dataSource" name="StockDs"/>
</wlevs:event-bean>
```

If the name of the EventBean set method matches the name of the resource, then the `@Resource` annotation `name` attribute is not needed. Similarly, in this case, the `wlevs:resource` element `name` attribute is not needed.

```
import javax.annotation.Resource;

public class SimpleBean implements EventBean {
...
    @Resource ()
    public void setStockDs (DataSource dataSource){
        this.dataSource = dataSource;
    }
}

< wlevs:event-bean id="simpleBean" class="...SimpleBean"/>
    <wlevs:resource property="dataSource"/>
</wlevs:event-bean>
```

Example 2-3 Static Resource Injection Using Static Resource Names: Annotations

```
import javax.annotation.Resource;

public class SimpleBean implements EventBean {
...
    @Resource (name="StockDs")
    public void setDataSource (DataSource dataSource){
        this.dataSource = dataSource;
    }
}
```

2.5.2.2 Dynamic Resource Names

A dynamic resource name is one that is specified as part of the dynamic or external configuration of an application. Using a dynamic resource name, the deployer or administrator can change the resource name without requiring that the application developer modify the application code or the Spring application context.

To add a dynamic resource name to a component, such as an adapter or POJO, you must first specify custom configuration for your component that contains the resource name.

```
<simple-bean>
    <name>SimpleBean</name>
    <trade-datasource>StockDs</trade-datasource>
</simple-bean>
```

To configure static resource injection using dynamic resource names at design time, use the standard `javax.annotation.Resource` annotation.

To override design time configuration at deploy time, you use Oracle Event Processing assembly file XML.

```
import javax.annotation.Resource;

public class SimpleBean implements EventBean {
...
    @Resource (name="trade-datasource")
    public void setDataSource (DataSource dataSource){
        this.dataSource = dataSource;
    }
}

< wlevs:event-bean id="simpleBean" class="...SimpleBean"/>
    <wlevs:resource property="dataSource" name="trade-datasource"/>
</wlevs:event-bean>
```

2.5.3 Dynamic Resource Injection

Dynamic resource injection refers to the injection of resources dynamically while the component is active in response to a dynamic configuration change using Spring container method injection.

To configure dynamic resource injection at design time, use the standard `javax.annotation.Resource` annotation as Example 2-4 shows.

The component calls the `getDataSource` method at runtime whenever it needs to retrieve a new instance of the resource that the resource name `trade-datasource` refers to.

Typically, the component calls the `getDataSource` method during the `@Prepare` or `@Activate` methods when dynamic configuration changes are handled. For more information see *Schema Reference for Oracle Event Processing*.

Another strategy is to always call the `getDataSource` prior to using the data source. That is, the application code does not store a reference to the data source as a field in the component.

Example 2-4 Dynamic Resource Injection: Annotations

```
import javax.annotations.Resource;

public class SimpleBean implements EventBean {
...
    @Resource ("trade-datasource")
    public abstract DataSource getDataSource ();
    ...
}
```

2.5.4 Dynamic Resource Lookup Using JNDI

Oracle Event Processing supports the use of JNDI to look up resources dynamically.

```
import javax.naming.InitialContext;

public class SimpleBean implements EventBean {
...
    public abstract void getDataSource () throws Exception {
        InitialContext initialContext= new InitialContext ();
        return initialContext.lookup ("StockDs");
    }
}
```

The JNDI name `StockDs` must exactly match the name of the data source in the Oracle Event Processing server file.

Note:

You must disable security when starting the Oracle Event Processing server in order to use JNDI. Oracle does not recommend the use of JNDI for this reason.

For more information, see *Administering Oracle Event Processing*.

2.5.5 Resource Name Resolution

Oracle Event Processing server resolves resource names by examining the naming scopes that Table 2-1 lists.

Table 2-1 Resource Name Resolution

Naming Scope	Contents	Resolution Behavior
Component	The property names of the component's custom configuration	Mapping
Application	The names of the configuration elements in the application configuration files	Matching
Server	The names of the configuration elements in the server configuration file	Matching
JNDI	The names registered in the server's JNDI registry	Matching

Each naming scope contains a set of unique names. The name resolution behavior is specific to a naming scope. Some naming scopes resolve names by simple matching. Other scopes resolve names by mapping the name used to do the lookup into a new name. Once a name is mapped, lookup proceeds recursively beginning with the current scope.

3

Events and Event Types

An event type is a data structure that defines the data contained in an event. When raw event data comes into the Oracle Event Processing application, the application binds that data to an event of a particular event type. In your application, you define the event type in terms of its data set and the corresponding data types.

This chapter includes the following sections:

3.1 How Events Function

An event is structured data that relates to something that happens or is happening. For example, if your application reacts to changes to a cluster of servers, events capture snapshot data that is collected by the device that monitors the servers. Or if your application monitors trends and patterns related to stock market trades, events contain event data that corresponds to stock trades.

Event data can arrive at an application in many forms. By creating an event type to represent the data inside the application, you create a predictable way for application logic to work with the data.

Events carry event data through the event processing network (EPN). When you design the event type, keep in mind how you plan to access, process, and manipulate the event data in your code.

3.2 Choose a Data Structure for the Event Type

An event type can get its structure from a `JavaBean` class, a tuple, or a `java.util.Map` class. Oracle recommends that you use `JavaBean` classes to structure event types. JavaBeans provide greater flexibility within your application and simplify integration with existing systems.

JavaBean event types are flexible. For example, you assign a JavaBean event type to a property of a tuple or `java.util.Map` event type. The following code shows the event type `Student` that defines its `address` property as the JavaBean event type `Address`.

```
<wlevs:event-type-repository>
<wlevs:event-type type-name="Student">
  <wlevs:properties>
    <wlevs:property name="name" type="char"/>
    <wlevs:property name="address" type="classpackage.Address"/>
  </wlevs:properties>
</wlevs:event-type>
</wlevs:event-type-repository>
```

JavaBeans also enable you to closely control event type instantiation by implementing an event type builder class. For more information on event type builder classes, see Control Event Type Instantiation with an Event Type Builder Class.

Table 3-1 compares JavaBean classes, tuples, and `java.util.Map` classes.

Table 3-1 Data Types for Event Types

Data Type	Description	Benefits and Limitations
JavaBean	A Java class written to JavaBean conventions. In addition to being used by logic you write, the type's accessor (get and set) methods will be used by the Oracle Event Processing server and Oracle CQL processor to retrieve and set event property values.	**Benefits:** This type is the best practice because it provides the greatest flexibility and ease of use for application logic that handles events. You access property values directly through accessor methods. A JavaBean class is more likely to be useful when integrating your Oracle Event Processing application with other systems. For control over how the type is instantiated, you can implement an event type builder class. **Limitations:** Requires writing a JavaBean class, rather than simply declaring the event type in a configuration file. Oracle CQL does not support JavaBean properties in GROUP BY, PARTITION BY, and ORDER BY, although you can work around this by using an Oracle CQL view.

Table 3-1 (Cont.) Data Types for Event Types

Data Type	Description	Benefits and Limitations
Tuple	A structure that you create and register declaratively in the EPN assembly file. For more information, see Create and Register a Tuple Event Type.	**Benefits:** Requires no Java programming to create the event type. An event type is created by declaring it in the EPN assembly file. Useful for quick prototyping. **Limitations:** Using instances of this type in Java application logic requires programmatically accessing the event type repository to get the instance's property values. A tuple is also unlikely to be useful when integrating the Oracle Event Processing with other systems.
`java.util.Map`	Based on an instance of `java.util.Map`. You don't implement or extend the `Map` interface. Rather, you specify that the interface should be used when configuring the event type in the EPN assembly file. If you write Java code to access the type instance, you treat it as a `Map` instance. For more information, see Create and Register a Map Event Type.	**Benefits:** Requires no Java programming to create the type. An event type is created by declaring it in the EPN assembly file. Useful for quick prototyping. **Limitations:** Does not perform as well as other types.

3.3 Design Constraints

Keep in mind the following CSV adapter and database table constraints when you design event types.

CSV Adapter Constraints

When you declaratively specify the properties of an event type for use with CSV adapters, you can only use the data types that Table 3-2 describes.

Table 3-2 CSV Adapter Types

Type	Usage
`char`	Single or multiple character values. Use for both `char` and `java.lang.String` values. Optionally, you may use the `length` attribute to specify the maximum length of the `char` value for the property with name `id`. The default length is 256 characters. If you need more than 256 characters you should specify an adequate length.
`int`	Numeric values in the range that `java.lang.Integer` specifies.
`float`	Numeric values in the range that `java.lang.Float` specifies.
`long`	Numeric values in the range that `java.lang.Long` specifies.
`double`	Numeric values in the range that `java.lang.Double` specifies.

For more information, see:

- Testing 1-2-3

Database Table Constraints

You can use a relational database table as a source of event data, binding data from the table to your event type instance at runtime. When your event data source is a database table, you must follow the guidelines represented by the following tables.

When an event type will receive data from a database table, a property configured for the type will each receive data from a particular column in the database. When configuring the event type, note that its property child elements have attributes that have particular meanings and value constraints, as described in Table 3-3.

Table 3-3 EPN Assembly File event-type Element Property Attributes

Attribute	Description
`name`	The name of the table column you want to access as specified in the SQL create table statement. You do not need to specify all columns.
`type`	The Oracle Event Processing Java type from Table 3-4 that corresponds to the column's SQL data type.
`length`	The column size as specified in the SQL create table statement.

When you specify the properties of an event type for use with a relational database table, you must observe the additional JDBC type restrictions listed in Table 3-4.

When you join a stream with the Derby database, where the join condition compares two timestamp values - one value is from the stream attribute and the other value is from the Derby data source attribute, the Derby database performs the predicate evaluation. However, the Derby database supports only the `yyyy-MM-dd-HH.mm.ss[.nnnnnn]` format. For the Derby database to perform the evaluation correctly, the stream timestamp value must use the Derby database format.

Table 3-4 SQL Column Types and Oracle Event Processing Type Equivalents

SQL Type	Oracle Event Processing Java Type	com.bea.wlevs.ede.api.Type	Description
`ARRAY`	`[Ljava.lang.Object`		Array, of depth 1, of `java.lang.Object`.
`BIGINT`	`java.math.BigInteger`	`bigint`	An instance of `java.math.BigInteger`.
`BINARY`	`byte[]`		Array, of depth 1, of `byte`.
`BIT`	`java.lang.Boolean`	`boolean`	An instance of `java.lang.Boolean`.
`BLOB`	`byte[]`		Array, of depth 1, of `byte`.
`BOOLEAN`	`java.lang.Boolean`	`boolean`	An instance of `java.lang.Boolean`.
`CHAR`	`java.lang.Character`	`char`	An instance of `java.lang.Character`.

Table 3-4 (Cont.) SQL Column Types and Oracle Event Processing Type Equivalents

SQL Type	Oracle Event Processing Java Type	com.bea.wlevs.ede.api.Type	Description
CLOB	byte[]		Array, of depth 1, of byte.
DATE	java.sql.Date	timestamp	An instance of java.sql.Date.
DECIMAL	java.math.BigDecimal		An instance of java.math.BigDecimal.
BINARY_DOUBLE or DOUBLE	java.lang.Double	double	An instance of java.lang.Double
BINARY_FLOAT or FLOAT	java.lang.Double	float	An instance of java.lang.Double
INTEGER	java.lang.Integer	int	An instance of java.lang.Integer.
JAVA_OBJECT	java.lang.Object	object	An instance of java.lang.Object.
LONGNVARCHAR	char[]	char	Array, of depth 1, of char.
LONGVARBINARY	byte[]		Array, of depth 1, of byte.
LONGVARCHAR	char[]	char	Array, of depth 1, of char.
NCHAR	char[]	char	Array, of depth 1, of char.
NCLOB	byte[]		Array, of depth 1, of byte.
NUMERIC	java.math.BigDecimal		An instance of java.math.BigDecimal.
NVARCHAR	char[]	char	Array, of depth 1, of char.
OTHER	java.lang.Object	object	An instance of java.lang.Object.
REAL	java.lang.Float	float	An instance of java.lang.Float
SMALLINT	java.lang.Integer	int	An instance of java.lang.Integer.
SQLXML	xmltype	xmltype	For information about processing XMLTYPE data in Oracle CQL, see *Oracle CQL Language Reference for Oracle Event Processing*.
TIME	java.sql.Time		An instance of java.sql.Time.
TIMESTAMP	java.sql.Timestamp	timestamp	An instance of java.sql.Timestamp.
TINYINT	java.lang.Integer	int	An instance of java.lang.Integer.
VARBINARY	byte[]		Array, of depth 1, of byte.

Table 3-4 (Cont.) SQL Column Types and Oracle Event Processing Type Equivalents

SQL Type	Oracle Event Processing Java Type	com.bea.wlevs.ede.api.Type	Description
VARCHAR	`char[]`	`char`	Array, of depth 1, of `char`.

For more information, see: Configure a Table Source.

3.4 Event Type Repository

Oracle Event Processing manages event types in an event type repository. The Oracle Event Processing server accesses the assembly file at run time to retrieve the information it needs to manage the application. The following example shows an event type entry in the repository:

```
<wlevs:event-type-repository>
  <wlevs:event-type type-name="TradeEvent">
    <wlevs:class>tradereport.TradeEvent</wlevs:class>
  </wlevs:event-type>
</wlevs:event-type-repository>
```

To define and edit event types, you can use the Oracle JDeveloper Event tab, work in the assembly file directly, or call APIs from your application code. The Event tab displays when you open the EPN diagram for an Oracle JDeveloper project.

For more information, see Access the Event Type Repository.

3.5 Properties

When you create an event type, you add the `<wlevs:properties>` and `<wlevs:property>` elements to the `<wlevs:event-type>` element to define the event type properties. Properties have `name` and `type` attributes that define the kind of information, such as ticker name, ticker symbol, and closing price, and the corresponding data type, such as `String`, `String`, and `Double`. For more information about `the <wlevs:event-type>` element, see *Schema Reference for Oracle Event Processing*.

Assembly File

The following assembly file entries show a simple event type with one event type and one property defined by the `<wlevs:class>` element. The properties for this event type are defined in a JavaBean class.

```
<wlevs:event-type-repository>
  <wlevs:event-type type-name="TradeEvent">
    <wlevs:class>tradereport.TradeEvent</wlevs:class>
  </wlevs:event-type>
</wlevs:event-type-repository>
```

The following assembly file entries show a message count event type with properties defined by the `<wlevs:properties>` element, which encloses three `<wlevs:property>` elements.

```
<wlevs:event-type-repository>
        <wlevs:event-type id="messagecounts" type-name="SimpleEvent">
            <wlevs:properties>
                <wlevs:property name="msg" type="char" />
```

```
                <wlevs:property name="count" type="long" />
                <wlevs:property name="time_stamp" type="timestamp" />
            </wlevs:properties>
        </wlevs:event-type>
...
</wlevs:event-type-repository>
```

3.6 Interval and Time Stamp Properties

Event types also support the day-to-second and year-to-month interval properties and the time stamp with local time zone properties.

3.6.1 Interval Properties

The following assembly file entries show the interval properties.

```
<wlevs:event-type-repository>
  <wlevs:event-type type-name="IntervalDataTypeEvent">
    <wlevs:properties>
      <wlevs:property name="Comment" type="char" length="256" />
        <wlevs:property name="intervalProp" type="interval day(1) to second(2)"/>
        <wlevs:property name="intervalymProp" type="interval year(2) to month"/>
        <wlevs:property name="intervaldhProp" type="interval day to hour"/>
        <wlevs:property name="intervaldmProp" type="interval day to minute"/>
        <wlevs:property name="intervalhsProp" type="interval hour(1)
                to second(2)"/>
        <wlevs:property name="intervalhmProp" type="interval hour to minute"/>
        <wlevs:property name="intervalmsProp" type="interval minute(2)
                to second(2)"/>
        <wlevs:property name="intervaldProp" type="interval day(1)"/>
        <wlevs:property name="intervalyProp" type="interval year(2)"/>
        <wlevs:property name="intervalmProp" type="interval month"/>
    </wlevs:properties>
  </wlevs:event-type>
</wlevs:event-type-repository>
```

Day-to-second combinations:

```
INTERVAL DAY[(day_precision)]
TO SECOND[(fractional_seconds_precision)]
```

`day_precision` is the number of digits in the `DAY` date-time field. Accepted values are 0 to 9. The default is 2.

`fractional_seconds_precision` is the number of digits in the fractional part of the `SECOND` date-time field. Accepted values are 0 to 9. The default value is 6.

Year-to-month combinations:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

`year_precision` is the number of digits in the `YEAR` date-time field. The default value for `year_precision` is 2.

3.6.2 Time Stamp with Local Time Zone Properties

The following assembly file entries show the time stamp with local time zone properties.

```
<wlevs:event-type-repository>
<wlevs:event-type type-name="IntervalDataTypeEvent">
<wlevs:properties>
  <wlevs:property name="Comment" type="char" length="256" />
  <wlevs:property name="timestamptzProp" type="timestamp with time zone"/>
  <wlevs:property name="timestampltzProp" type="timestamp with local time zone"/>
</wlevs:properties>
</wlevs:event-type>
</wlevs:event-type-repository>
```

With time zone:

`TIMESTAMP [(fractional_seconds_precision)] WITH TIME ZONE`

`fractional_seconds_precision` optionally specifies the number of digits Oracle stores in the fractional part of the `SECOND` datetime field. When you create a column of this data type, the value can be a number in the range 0 to 9. The default value is 6.

With local time zone:

`TIMESTAMP [(fractional_seconds_precision)] WITH LOCAL TIME ZONE`

`fractional_seconds_precision` optionally specifies the number of digits Oracle stores in the fractional part of the `SECOND` datetime field. When you create a column of this data type, the value can be a number in the range 0 to 9. The default value is 6.

3.7 Create and Register a JavaBean Event Type

First, identify the event data that the event type carries and then decide the properties the event type requires. This section walks you through the following steps. To make the JavaBean an event source or sink, see Event Beans .

3.7.1 Data Types

You can use the following Java types for the properties:

- The fully qualified name of a Java class. The name must conform to the `Class.forName` rules and be available in the application class loader.
- A Java primitive such as `int` or `float`.
- An array by appending square brackets (`[]`)to the primitive or class name. For example, `short[]` or `java.lang.Integer[]`.

3.7.2 Create a JavaBean Event Type Declaratively

1. Create a JavaBean class to represent your event type.

```
package com.bea.wlevs.example.algotrading.event;

public final class MarketEvent {
    private final Long timestamp;
    private final String symbol;
    private final Double price;
    private final Long volume;
    private final Long latencyTimestamp;

    public MarketEvent() {}

    public Double getPrice() {
        return this.price;
    }
    public void setPrice(Double price) {
```

```
        this.price = price;
    }

    public String getSymbol() {
        return this.symbol;
    }
    public void setSymbol(String symbol) {
        this.symbol = symbol;
    }

    public Long getTimestamp() {
        return this.timestamp;
    }
    public void setTimestamp(Long timestamp) {
        this.timestamp = timestamp;
    }

    public Long getLatencyTimestamp() {
        return this.latencyTimestamp;
    }
    public void setLatencyTimestamp(Long latencyTimestamp) {
        this.latencyTimestamp = latencyTimestamp;
    }

    public Long getVolume() {
        return this.volume;
    }
    public void setVolume(Long volume) {
        this.volume = volume;
    }

    // Implementation for hashCode and equals methods.
}
```

2. Compile the JavaBean class.
3. Register your JavaBean event type in the Oracle Event Processing event type repository:

```
<wlevs:event-type-repository>
    <wlevs:event-type type-name="MarketEvent">
      <wlevs:class>
          com.bea.wlevs.example.algotrading.event.MarketEvent
      </wlevs:class>
    </wlevs:event-type>
</wlevs:event-type-repository>
```

3.7.3 Create a JavaBean Event Type Programmatically

Steps 1 and 2 are the same as steps 1 and 2 in Create a JavaBean Event Type Declaratively. Then, for step 3, do the following.

To register a JavaBean event type programmatically, use the `EventTypeRepository` class as shown:

```
EventTypeRepository rep = getEventTypeRepository();
rep.registerEventType("MarketEvent",
com.bea.wlevs.example.algotrading.event.MarketEvent.getClass()
);
```

For more information, see Access the Event Type Repository.

3.7.4 Usages

Once you create a JavBean even type, you can reference it in your application Java code. The following code references the `MarketEvent` event type in the `onInsertEvent` method implementation. The `onInsertEvent` method is from an event sink class that receives events. For more information on event sinks, see Event Sink Interfaces.

```
public void onInsertEvent(Object event) throws EventRejectedException {
    if (event instanceof MarketEvent){
        MarketEvent marketEvent = (MarketEvent) event;
        System.out.println("Price: " + marketEvent.getPrice());
    }
}
```

The following Oracle CQL rule shows how to reference the `MarketEvent` event type in a `SELECT` statement. It assumes an upstream channel called `marketEventChannel` with a `MarketEvent` event type.

```
<query id="helloworldRule">
    <![CDATA[ SELECT MarketEvent.price FROM marketEventChannel [NOW] >
</query>
```

Also, with property data types implemented as JavaBeans, Oracle CQL code can get values within those properties by using standard JavaBean-style property access. For example, the following configuration snippet declares a `StudentType` event type that is implemented as a JavaBean class. The `school.Student` class is a JavaBean with an `address` property that is an `Address` JavaBean class. The following query suggests how you might access values of the `Address` object underlying the address property. This query selects student addresses whose postal code begins with 97.

```
<query id="studentAddresses">
    FOR StudentType SELECT student.address
    FROM
        StudentType as student
    WHERE
        student.address.postalCode LIKE '^97'
</query>
```

EventRejectedException Behavior in onInsertEvent Implementations

You need to explicitly throw `EventRejectedException` in `onInsertEvent` implementations for exceptions you do not want to get dropped. You can raise an `EventProcessingException` and it is propagated all the way to the source of the error through a CQL processor. An EventRejectedException can chain exceptions from its downstream listeners, in case there is more than one exception. The CQL processor converts the `EventRejectedException` to a soft exception. See Fault Handling for more information.

3.8 Create and Register a Tuple Event Type

First, identify the event data that the event type carries and then decide the properties the event type requires. When you design your event, you must restrict the properties to the data types described in Design Constraints.

With a tuple-based event type, your Java code must always set and get its property values with the `EventTypeRepository` APIs.

Note:

The order in which the EPN processes tuples with the same time stamp is not guaranteed when the EPN is made up of multiple streams.

Data Types

When you specify the tuple event type properties declaratively in the application assembly file, you can use any of the native Oracle CQL data types in the property type.

The following XML shows the use of different types in the application assembly file.

```
<wlevs:event-type-repository>
        <wlevs:event-type type-name="SimpleEvent">
            <wlevs:properties>
                <wlevs:property name="id" type="char" length="1000" />
                <wlevs:property name="msg" type="char" />
                <wlevs:property name="count" type="double" />
                <wlevs:property name="time_stamp" type="timestamp" />
        </wlevs:properties>
    </wlevs:event-type>
...
</wlevs:event-type-repository>
```

Procedures

- Create a Tuple Event Type in the Assembly File
- Use a Tuple Event Type in Java Code
- Use a Tuple Event Type Instance in Oracle CQL Code

3.8.1 Create a Tuple Event Type in the Assembly File

Register your event type declaratively in the Oracle Event Processing event type repository with the `wlevs:event-type-repository` element and the `wlevs:event-type` child element.

Create a Tuple Event Type in the Assembly File

The following XML stanzas create a the `CrossRateEvent` tuple event type with the properties `price`, `fromRate`, and `toRate`.

```
<wlevs:event-type-repository>
    <wlevs:event-type type-name="CrossRateEvent">
        <wlevs:properties>
            <wlevs:property name="price" type="double"/>
            <wlevs:property name="fromRate" type="char"/>
            <wlevs:property name="toRate" type="char"/>
        </wlevs:properties>
    </wlevs:event-type>
</wlevs:event-type-repository>
```

See *Schema Reference for Oracle Event Processing*.

3.8.2 Use a Tuple Event Type in Java Code

Before you can use a tuple event type in Java code, you must create an event type repository. You use the event type repository to get the property names and values so you can work on them in your code. To create an event type repository, include the `com.bea.wlevs.ede.api.EventTypeRepository` class.

The following code is part of an event sink class. The code creates an event type repository with a call to the `setEventTypeRespository` method. The Oracle Event Processing server then calls the `onInsertEvent` method with an `event` parameter. The `onInsertEvent` method gets information about the `event` from the event type repository.

For more information about creating an `EventTypeRepository` object, see Access the Event Type Repository.

```
@Service
// Create an event type repository
public void setEventTypeRepository(EventTypeRepository etr) {
    etr_ = etr;
}
...
// Called by the server to pass in the event type instance.
public void onInsertEvent(Object event) throws EventRejectedException {

    // Get the event type for the current event instance
    EventType eventType = etr_.getEventType(event);

    // Get the event type name
    String eventTypeName = eventType.getTypeName();

    // Get the event property names
    String[] propNames = eventType.getPropertyNames();

    // See if property you're looking for is present
    if(eventType.isProperty("fromRate")) {
        // Get property value
        Object propValue =
            eventType.getProperty("fromRate").getValue(event);
    }
    // Throw com.bea.wlevs.ede.api.EventRejectedException to have an
    // exception propagated up to senders. Other errors will be
    // logged and dropped.
}
```

3.8.3 Use a Tuple Event Type Instance in Oracle CQL Code

The following Oracle CQL rule shows how to reference the `CrossRateEvent` in a `SELECT` statement. `FxQuoteStream` is a channel with the `CrossRateEvent` event type.

```
<query id="FindCrossRatesRule"><![CDATA[
    select ((a.price * b.price) + 0.05) as internalPrice,
        a.fromRate as crossRate1,
        b.toRate as crossRate2
    from FxQuoteStream [range 1] as a, FxQuoteStream [range 1] as b
    where
        NOT (a.price IS NULL)
    and
        NOT (b.price IS NULL)
    and
        a.toRate = b.fromRate
></query>
```

3.9 Create and Register a Map Event Type

First, identify the event data that the event type carries and then decide the properties the event type requires. You create a `java.util.map` event type by adding the configuration XML to the application assembly file. An event type based on a hash map is called a map-based event type.

- Data Types
- To create and register a java.util.Map event type:
- Usages

Data Types

You can use the following Java types for the properties:

- The fully qualified name of a Java class. The name must conform to the `Class.forName` rules and be available in the application class loader.
- A Java primitive such as `int` or `float`.
- An array by appending square brackets (`[]`)to the primitive or class name. For example, `short[]` or `java.lang.Integer[]`.

The following XML code shows examples of event property declarations in the event repository.

```
<wlevs:event-type-repository>
    <wlevs:event-type type-name="AnotherEvent">
        <wlevs:properties type="map">
          <wlevs:property>
             <entry key="name" value="java.lang.String"/>
             <entry key="employeeId" value="java.lang.Integer[]"/>
             <entry key="salary" value="float"/>
             <entry key="projectIds" value="short[]"/>
          </wlevs:property>
        <wlevs:properties>
    </wlevs:event-type>
</wlevs:event-type-repository>
```

To create and register a java.util.Map event type:

First, identify the event data that the event type carries and then decide the properties the event type requires.

- **To register declaratively**, edit the EPN assembly file using the `wlevs:event-type-repository` element `wlevs:event-type` child element as shown:

  ```
  <wlevs:event-type-repository>
      <wlevs:event-type type-name="AnotherEvent">
          <wlevs:properties type="map">
              <wlevs:property name="name" value="java.lang.String"/>
              <wlevs:property name="age" value="java.lang.Integer"/>
              <wlevs:property name="address" value="java.lang.String"/>
          </wlevs:properties >
      </wlevs:event-type>
  </wlevs:event-type-repository>
  ```

  At runtime, Oracle Event Processing generates a bean instance of the `AnotherEvent` class for you. The `AnotherEvent` class has three properties: `name`, `age`, and `address`.

- **To register programmatically**, use the `EventTypeRepository` class as shown:

```
EventTypeRepository rep = getEventTypeRepository();
java.util.Map map = new Map({name, java.lang.String},
    {age, java.lang.Integer}, {address, java.lang.String});
rep.registerEventType("AnotherEvent", map);
```

Usages

```
public void onInsertEvent(Object event)
        throws EventRejectedException {

        java.util.Map anEvent = (java.util.Map) event;
        System.out.println("Age: " + anEvent.get("age"));
}
```

The following Oracle CQL rule shows how you can reference the `MarketEvent` in a `SELECT` statement:

```
<query id="helloworldRule">
    <![CDATA[ select age from eventChannel [now] >
</query>
```

3.10 Access the Event Type Repository

The Oracle Event Processing event type repository keeps track of the event types defined for your application. When you create an event type in Oracle JDeveloper, it becomes available to the Oracle Event Processing application.

In some cases, you might need to write code that explicitly accesses the repository. For example, when your event type is created as a tuple, Java logic that accesses instance of the type will need to first retrieve the type definition using the repository API, then use the API to access the instance property values.

The `EventTypeRepository` is a singleton OSGi service. Because it is a singleton, you only need to specify its interface name to identify it. You can get a service from OSGi in any of the following ways:

- EPN Assembly File
- Spring-DM @ServiceReference Annotation
- Oracle Event Processing @Service Annotation

For more information, see *Java API Reference for Oracle Event Processing*.

3.10.1 EPN Assembly File

You can access the `EventTypeRepository` by specifying an `osgi:reference` in the EPN assembly file.

```
<osgi:reference id="etr" interface="com.bea.wlevs.ede.api.EventTypeRepository" />
<bean id="outputBean" class="com.acme.MyBean" >
    <property name="eventTypeRepository" ref="etr" />
</bean>
```

Then, in the `MyBean` class, you can access the `EventTypeRepository` using the `eventTypeRepository` property initialized by Spring.

```
package com.acme;

import com.bea.wlevs.ede.api.EventTypeRepository;
import com.bea.wlevs.ede.api.EventType;
```

```
public class MyBean {
    private EventTypeRepository eventTypeRepository;

    public void setEventTypeRepository(EventTypeRepository eventTypeRepository) {
        this.eventTypeRepository = eventTypeRepository;
    }

    public void onInsertEvent(Object event) throws EventRejectedException {
        // get the event type for the current event instance
        EventType eventType = eventTypeRepository.getEventType(event);

        // Throw com.bea.wlevs.ede.api.EventRejectedException to have an
        // exception propagated up to senders. Other errors will be
        // logged and dropped.

    }
}
```

3.10.2 Spring-DM @ServiceReference Annotation

You can access the `EventTypeRepository` by using the Spring-DM `@ServiceReference` annotation to initialize a property in your Java source.

```
import org.springframework.osgi.extensions.annotation.ServiceReference;
import com.bea.wlevs.ede.api.EventTypeRepository;
...
@ServiceReference
setEventTypeRepository(EventTypeRepository etr) {
    ...
}
```

3.10.3 Oracle Event Processing @Service Annotation

You can access the `EventTypeRepository` with the Oracle Event Processing `@Service` annotation to initialize a property in your Java source.

```
import com.bea.wlevs.util.Service;
import com.bea.wlevs.ede.api.EventTypeRepository;
...
@Service
setEventTypeRepository(EventTypeRepository etr) {
    ...
}
```

3.11 Share Event Types Between Application Bundles

Each Oracle Event Processing application gets its own Java classloader and loads application classes using that class loader. This means that, by default, one application cannot access the classes in another application. However, because the event type repository is a singleton service, you can configure the repository in one bundle and then explicitly export the event type classes so that applications in separate bundles (deployed to the same Oracle Event Processing server) can use these shared event types.

The event type names in this case are scoped to the entire Oracle Event Processing server instance. This means that you will get an exception if you try to create an event type that has the same name as an event type that has been shared from another bundle, but the event type classes are different.

To share event type classes, add their package name to the `Export-Package` header of the `MANIFEST.MF` file of the bundle that contains the event type repository you want to share.

Be sure you deploy the bundle that contains the event type repository *before* all bundles that contain applications that use the shared event types, or you will get a deployment exception.

For more information, see:

- Choose a Data Structure for the Event Type

  Reference Foreign Stages
- *Java API Reference for Oracle Event Processing*.

3.12 Control Event Type Instantiation with an Event Type Builder Class

You can create an event type builder to have more control over how event type instances are created. For example, using an event type builder you can ensure that the properties of a configured event are correctly bound to the properties of an event type class, such as one you have implemented as a JavaBean. You would need an event type builder in a case, for example, where event property names assumed in CQL code are different from the names of properties declared in the class.

For example, assume the event type has a `firstname` property, but the CQL rule that executes on the event type assumes the property is called `fname`. Assume also that you cannot change either the event type class (because you are using a shared event class from another bundle, for example) or the CQL rule to make them compatible with each other. In this case you can use an event type builder factory to change the way the event type instance is created so that the property is named `fname` rather than `firstname`.

At runtime, an event type builder class receives property values from the Oracle Event Processing server and uses those values to create an instance of the event type class you created. Your event type builder then returns the instance to the server. In this way, your builder class is in effect an intermediary, instantiating event types in cases where the server is unable to determine how to map configured properties to event type properties.

Creating and using an event type builder involves implementing the builder class and configuring a JavaBean event type to use the builder, as described in the following sections:

- Implement an Event Type Builder Class
- An Event Type that Uses an Event Type Builder

3.12.1 Implement an Event Type Builder Class

When you program the event type builder factory, you must implement the `EventBuilder.Factory` inner interface of the `com.bea.wlevs.ede.api.EventBuilder` interface; see the *Java API Reference for Oracle Event Processing* for details about the methods you must implement, such as `createBuilder` and `createEvent`.

The following example of an event type builder factory class is taken from the FX sample:

```
package com.bea.wlevs.example.fx;

import java.util.HashMap;
import java.util.Map;
import com.bea.wlevs.ede.api.EventBuilder;
import com.bea.wlevs.example.fx.OutputBean.ForeignExchangeEvent;

public class ForeignExchangeBuilderFactory implements EventBuilder.Factory {

    // Called by the server to get an instance of this builder.
    public EventBuilder createBuilder() {
        return new ForeignExchangeBuilder();
    }

    // Inner interface implementation that is the builder.
    static class ForeignExchangeBuilder implements EventBuilder {

        // A Map instance to hold properties until the event type is instantiated.
        private Map<String,Object> values = new HashMap<String,Object>(10);

        // Called by the server to put an event type property. Values from the map
        // will be used to instantiate the event type.
        public void put(String property, Object value) throws IllegalStateException {
            values.put(property, value);
        }

        // Called by the server to create the event type instance once property
        // values have been received.
        public Object createEvent() {
            return new ForeignExchangeEvent(
                (String) values.get("symbol"),
                (Double) values.get("price"),
                (String) values.get("fromRate"),
                (String) values.get("toRate"));
        }
    }
}
```

3.12.2 An Event Type that Uses an Event Type Builder

When you register the event type in the EPN assembly file, use the `<wlevs:property name="builderFactory">` child element of the `wlevs:event-type` element to specify the name of the event type builder class. The hard-coded `builderFactory` value of the `name` attribute alerts Oracle Event Processing that it should use the specified factory class, rather than its own default factory, when creating instances of this event. For example, in the FX example, the builder factory is registered as shown in bold:

```
<wlevs:event-type-repository>
  <wlevs:event-type type-name="ForeignExchangeEvent">
    <wlevs:class>com.bea.wlevs.example.fx.OutputBean$ForeignExchangeEvent</wlevs:class>
    <wlevs:property name="builderFactory">
      <bean id="builderFactory"
            class="com.bea.wlevs.example.fx.ForeignExchangeBuilderFactory"/>
    </wlevs:property>
  </wlevs:event-type>
</wlevs:event-type-repository>
```

4

Adapters

Adapters manage data entering and leaving the EPN. Oracle Event Processing provides a number of different kinds of inbound and outbound adapters to handle different types of data such as CSV, RMI, and HTTP. All adapters have a `provider` property that is a reference to the OSGi-registered adapter factory service and defines the type of data that the adapter handles.

Inbound adapters receive event data from a data stream entering the EPN, assign the data to an event according to the event type, and send the data to the next stage in the EPN. Outbound adapters receive events processed by the EPN, convert the events to their output form, and send the converted data to an output data source such as another EPN, a non-EPN application, a CSV file, or a web page.

This chapter includes the following sections:

For information about the high availability adapters, see High Availability Applications.

See Testing 1-2-3 for information about how to use the csvgen adapter with the load generator to simulate a data feed to test your application.

4.1 Create Adapters

The best way to create most adapters is with Oracle JDeveloper. The Oracle JDeveloper components window provides the following inbound and outbound adapters: CSV, EDN, RMI, HTTP, and JMS. For the other adapters, edit the configuration files directly. Each adapter section in this chapter provides example assembly and configuration file configurations so that you can see the settings.

Before you create an adapter, use Oracle JDeveloper to create an event type to assign to the adapter. See *Getting Started with Oracle Event Processing* for information about creating adapters, event types, and other components in Oracle JDeveloper.

This chapter describes some of the assembly and configuration file settings for the different types of adapters. For complete information about adapter settings, see *Schema Reference for Oracle Event Processing*. See also the `Oracle/Middleware/my_oep/oep/wlevs_application_config.xsd` directory in your Oracle Event Processing installation for adapter schema information.

4.2 Cluster Distribution Service

The cluster distribution feature provides a mechanism for various Oracle Event Processing adapter types to distribute incoming events to all of the servers in a cluster. An individual adapter or event bean instance can be configured to distribute events, and in this case, all input events processed by that adapter are sent (distributed) to all servers in the cluster.

The distribution adapter ensures that all input events are sent (distributed) to all servers in the cluster. To convert an input adapter to a distribution adapter, add the `distributeInput` element and set it to `true` as follows. The `distributionThreadsCount` property is optional and defaults to 1.

```
<wlevs:adapter id="myLoadgenAdapter" provider="loadgen">
  <wlevs:instance-property name="distributeToClusterGroup" value="true"/>
  <wlevs:instance-property name="distributionThreadsCount" value="1"/>
</wlevs:adapter>
```

Oracle Event Processing supports the Cluster Distribution service for the loadgen, CSV inbound, and JMS inbound (queue) adapters. Oracle Event Processing does not support the Cluster Distribution service for the CSV outbound, JMS inbound (topic), JMS outbound, and HTTP publish-subscribe adapters. It is an error to configure topic destinations for input distribution. A topic configuration generates a warning log message and is ignored.

In addition to the adapter types listed above, you can also configure an event bean to distribute all of the events it receives to all cluster members by specifying the provider for the bean to be `clusterGroupDistributor` as shown in the following example:

```
<wlevs:event-bean id="distributor-bean" provider="clusterGroupDistributor">
   <wlevs:instance-property name="distributionThreadsCount" value="1"/>
   ... other event bean properties ...
</wlevs:event-bean>
```

Oracle JDeveloper does not currently provide a component for the Cluster Distribution service. However, you can create a Cluster Distribution service by adding entries to the assembly and configuration files for your Oracle Event Processing application.

4.3 Password Encryption

Some of the adapters have user name and password child elements. Oracle Event Processing provides the `encryptMSAConfig` command so that you can encrypt the file that contains the password. See *Administering Oracle Event Processing* for information about how to use the `encryptMSAConfig` command.

4.4 JAXB Support

Oracle Event Processing provides a simplified interface for using Java Architecture for XML Binding (JAXB) mapping capabilities in adapters and event beans to marshall and unmarshall event data between XML and Java objects. The JAXB interface supports the JAXB 2.2 specification and EclipseLink Moxy provider extensions.

You can configure the mapping operations in the following ways:

- Map from an XML schema to Java objects to output a set of annotated Java classes.
- Map from one set of Java objects to another set of Java Objects or to XML using JAXB annotations.
- Map from an existing XML schema to an existing, predefined Java object representation. This approach uses the EclipseLink Moxy extensions and requires an external `metadata` file that contains the mapping details. The `metadata` file is referenced by the application configuration.

4.4.1 EclipseLink Moxy

EclipseLink Moxy provides extensions that enable you to map between an existing XML schema and a predefined set of Java classes without modifying the XML schema or the Java classes without providing annotations. You provide the mapping information in an external metadata file using a XPath syntax.

The flexible EclipseLink Moxy extensions enable you to perform complex operations. For example, you can map a subset of complex XML data to a much simpler event representation. You can also flatten a deeply nested XML document into a flat Java bean event format for processing by Oracle CQL.

You specify EclipseLink Moxy external `metadata` in XML. Access the schema at: `http://www.eclipse.org/eclipselink/xsds/eclipselink_oxm_2_2.xsd`.

4.4.2 APIs

The adapter or event bean that requires JAXB functionality obtains the functionality by injection of a bean that implements the `com.oracle.cep.mappers.api.Mapper` interface. The `Mapper` interface follows:

```
public interface Mapper {
  Marshaller createMarshaller() throws MapperException;
  Unmarshaller createUnmarshaller() throws MapperException;
}
```

The adapter or other EPN component code uses the injected bean to create marshalling and unmarshalling objects. The `com.oracle.cep.mappers.api.Marshaller` and `com.oracle.cep.mappers.api.Unmarshaller` interfaces shown below work for most applications.

```
public interface Marshaller {
  void marshal(Object object, javax.xml.transform.Result result)
               throws MapperException;
 }

public interface Unmarshaller {
  Object unmarshal(javax.xml.transform.Source source)
                   throws MapperException;
}
```

Some applications might need specialized method signatures for marshalling and unmarshalling such as an unmarshall method that takes the target class as an argument. In these cases, use the `com.oracle.cep.mappers.jaxb.JAXBMarshallerImpl` and `com.oracle.cep.mappers.jaxb.JAXBUnmarshallerImpl`interfaces instead. These interfaces provide methods that correspond to the full set of marshall and unmarshall methods that are supported by the `javax.xml.bind.Marshaller` and `javax.xml.bind.Unmarshaller` interfaces.

Assembly File

The following assembly file entries call a mapper bean with properties to specify the event type and the metadata file.

```
<bean id="mapperBean" class="com.oracle.cep.mappers.jaxb.JAXBMapperImpl" >
  <property name="eventTypeName" value="CallCenterActivity" />
  <property name="metadata" value="external_metadata_case1.xml" />
</bean>
```

If you want to call a factory to make the mapper bean, specify the following for the bean element:

```
<bean id="mapperBean" class="com.oracle.cep.mappers.jaxb.JAXBMapperFactory"
      factorymethod="create"/>
```

Configuration File

The following configuration file entries specify properties for the mapper bean.

```
<jaxb-mapper>
  <name>mapperBean</name>
  <event-type-name>CallCenterActivity</event-type-name>
  <metadata>external_metadata_case1.xml</metadata>
</jaxb-mapper>
```

Properties

A mapper bean supports properties. All of the properties except `metadataMap` can be configured as assembly file properties or as elements in the configuration file.

Table 4-1 Mapper Bean Properties and Elements

Table 4-1 (Cont.) Mapper Bean Properties and Elements

Assembly File Property Name	Configuration File Element Name	Description
eventTypeName	event-type-name	The name of an event type registered by the application in the event type repository. The event type corresponds to a Java class. The package name of this class is used as the context path when initializing the JAXBContext represented by the mapper bean. The configuration must specify either an `eventTypeName` or a `contextPath` to be used in constructing the context path for the `JAXBContext` represented by the mapper. The packages must exist on the `classpath` of the application and contain either schema generated classes, JAXB annotated classes, or classes referenced by Moxy external metadata. The classes will be used as the Java object graph for marshalling and unmarshalling operations. See *Schema Reference for Oracle Event Processing* for information about the `JAXBContext.newInstance` method.
`contextPath`	`context-path`	A colon-separated list of Java package names. The specified context path to initialize the `JAXBContext` represented by the mapper bean. The configuration must specify either an `eventTypeName` or a `contextPath` to be used in constructing the context path for the `JAXBContext` represented by the mapper. The packages must exist on the `classpath` of the application and contain either schema generated classes, JAXB annotated classes, or classes referenced by Moxy external metadata. The classes will be used as the Java object graph for marshalling and unmarshalling operations.
`validate`	`validate`	`Boolean` value that defaults to `false`. When `true`, you must provide the `schema`. Schema validation occurs during marshalling and unmarshalling.
`schema`	`schema`	The file name of the XML schema file used for validation. Package the schema file with the application in the `META-INF/wlevs/mappers/jaxb` directory.

Table 4-1 (Cont.) Mapper Bean Properties and Elements

Assembly File Property Name	Configuration File Element Name	Description
`metadata`	`metadata`	The name of the file that contains the EclipseLink Moxy external metadate for mapping customization. Package the file with the application in the `Meta-INF/wlevs/mappers/jaxb directory`.
`metadataMap`	N/A	The Spring `<map>` element that contains one entry that corresponds to each component of the `contextPath`. For each entry, the key is the package name from the `contextPath` and the corresponding value is the name of a file that contains the EclipseLink Moxy external `metadata` for that package.

If the application uses EclipseLink Moxy-specific external `metadata`, the location of the `metadata` is specified by either the `metadata` property or the `metadataMap` property. The `metadataMap` property is required when there is more than one package on the `contextPath`. There is no support for specifying the `metadataMap` property in a component configuration file.

4.5 CSV Adapters

CSV adapters handle inbound and outbound data that is separated by commas. Use a CSVInbound adapter to accept data in the form of comma-separated values entering the EPN, and use a CSVOutbound adapter to send data in comma-separated values out of the EPN.

You can test an Oracle Event Processing Application that uses CSV inbound adapters with the load generator provided in your Oracle Event Processing installation. The load generator reads an ASCII file that contains sample data. You must use the CSV Inbound adapter because it is coded to decipher the data packets generated by the load generator. See Load Generator and the csvgen Adapter.

Note:

With the `java.sql.TimeStamp` type, the CSV adapter reads and writes data in the format yyyy-mm-dd'T'hh:mm:ss[.fffffffff]. For example, `"2012-12-12T12:12:12.120"`.

The best way to create CSV adapters is through the Oracle JDeveloper components window. The following assembly and configuration files generated by Oracle JDeveloper show the CSV inbound and outbound adapter configurations.

Assembly File

The inbound CSV adapter translates data read from the `StockData.csv` file into an event with the `TradeEvent` event type.

The `wlevs:listener` element specifies the component that listens to the inbound CSV adapter for TradeEvent activity. In this example, the listener is

`AdapterOutputChannel`. The `AdapterOutputChannel` component listens for and receives trade events from `StockTradeCSVInboundAdapter` to send to the next stage.

```
<wlevs:adapter id="StockTradeCSVInboundAdapter" provider="csv-inbound">
    <wlevs:listener ref="AdapterOutputChannel"/>
    <wlevs:instance-property name="eventType" value="TradeEvent"/>
    <wlevs:instance-property name="sourceUrl"
    value="file:/scratch/mpawlan/oep9-19/oep/utils/load-generator/StockData.csv"/>
  </wlevs:adapter>
```

The outbound adapter assembly file configuration is similar to the inbound adapter, but includes an `append` attribute. When set to `true`, Oracle Event Processing appends data to an existing output file. When set to `false`, Oracle Event Processing creates a new file or overwrites an existing file of the same name.

```
 <wlevs:adapter id="StockTradeCSVOutboundAdapter" provider="csv-outbound">
    <wlevs:instance-property name="eventType" value="TradeEvent"/>
    <wlevs:instance-property name="outputFile" value="/scratch/mpawlan/oep9-19/oep/
utils/load-generator/StockData.csv"/>
    <wlevs:instance-property name="append" value="false"/>
  </wlevs:adapter>
```

You can provide an absolute or relative path for the `outputFile` value. For the relative path, you can specify `../filename.csv. ./result.csv`, or `upload/result.csv`. When you specify a relative path, make sure that the abstract path includes the parent directory. For example, in UNIX, specify a file in the current directory as `./result.csv` instead of simply `result.csv`.

Configuration File

The adapter elements in the configuration file show the adapter `name` attribute and its value. The adapter name must match the adapter `id` attribute in the assembly file.

```
<csv-adapter>
  <name>StockTradeCSVInboundAdapter</name>
  <event-interval units="nanoseconds">5</event-interval>
</csv-adapter>

<csv-adapter>
  <name>StockTradeCSVOutboundAdapter</name>
</csv-adapter>
```

4.6 EDN Adapters

Event Delivery Network (EDN) inbound and outbound adapters use JAXB to enable an EPN to interface with an Oracle SOA Suite event network. The EDN adapters have a `raw-xml-content` configuration element that specifies whether to represent the EDN XML data transmission as raw XML (if true) or as a Java object using JAXB. In the JAXB case, the adapter expects the Oracle Event Processing application bundle to include the appropriate set of schema (`xjc`) generated classes on its class path.

You configure an EDN adapter with an event type and a reference to an EDL file. During initialization, the adapter searches the EDL file that contains an event definition QName that matches the configured event type. If the configured event type is found in the EDL, the adapter registers a subscription with EDN for the corresponding QName.

To learn how to use EDN adapters and an EDL file, see *Getting Started with Oracle Event Processing*.

4.6.1 Usage

Use an EDNInbound adapter to receive incoming data from the Oracle SOA Suite event network. The EDN input adapter subscribes to a specified EDN event type and converts the incoming EDN events to an Oracle Event Processing event type for processing by an Oracle Event Processing application.

Use an EDNOutbound adapter to send outbound data to the Oracle SOA Suite event network. The EDN output adapter converts the Oracle Event Processing events into corresponding EDN events and publishes them to the EDN. The published events can be new events that originate in the Oracle Event Processing application or EDN events that were received by the EDN input adapter, processed by the Oracle Event Processing application, and sent to the EDN output adapter.

4.6.2 Create EDN Adapters

The best way to create EDN adapters is through the Oracle JDeveloper components window. The following assembly and configuration files generated by Oracle JDeveloper show the EDN inbound and outbound adapter configurations.

See *Schema Reference for Oracle Event Processing* for information about the SOA Suite side of the EDN adapter.

Assembly File

The following assembly file entries show the elements and attribute settings for the inbound and outbound EDN adapters created in the Fraud Detection walkthrough from *Getting Started with Oracle Event Processing*.

- The input EDN adapter listens to the EDN input channel for events of type `FraudCheckRequest`.
- The EDN output adapter sends events of type `FraudCheckRequest` to the next stage in the EPN.

```
<wlevs:adapter id="edn-inbound-adapter" provider="edn-inbound">
  <wlevs:listener ref="ednInputChannel"/>
  <wlevs:instance-property name="eventType" value="FraudCheckRequest"/>
</wlevs:adapter>

<wlevs:adapter id="edn-outbound-adapter" provider="edn-outbound">
  <wlevs:instance-property name="eventType" value="FraudCheckResponse"/>
</wlevs:adapter>
```

Configuration file

The following configuration file entries show the configuration settings for the EDN inbound and outbound EDN adapters created in the Fraud Detection walkthrough from *Getting Started with Oracle Event Processing*

Note that you have to put the EDL and schema (xsd) files in the fixed path of the bundled JAR file.

```
<edn-adapter>
  <name>edn-outbound-adapter</name>
  <edl-file>FraudCheckEvent.edl</edl-file>
  <validate>false</validate>
  <raw-xml-content>false</raw-xml-content>
  <jndi-provider-url>t3://localhost:7101</jndi-provider-url>
  <jndi-factory>weblogic.jndi.WLInitialContextFactory</jndi-factory>
```

```
  <user>weblogic</user>
  <password>welcome1</password>
</edn-adapter>

<edn-adapter>
  <name>edn-inbound-adapter</name>
  <edl-file>FraudCheckEvent.edl</edl-file>
  <schema-file>FraudCheckType.xsd</schema-file>
  <validate>false</validate>
  <raw-xml-content>false</raw-xml-content>
  <jndi-provider-url>t3://localhost:7101</jndi-provider-url>
  <jndi-factory>weblogic.jndi.WLInitialContextFactory</jndi-factory>
  <user>weblogic</user>
  <password>welcome1</password>
</edn-adapter>
```

4.7 File Adapter

The File adapter reads data from a file into the EPN and converts the data to an event.

Oracle JDeveloper does not currently provide a component for the File adapter. However, you can create a File adapter by adding entries to the assembly and configuration files for your Oracle Event Processing application.

Assembly File

The `path` property provides the location of the input file. As the adapter reads the data from the input file, it converts the incoming data to an event of type `OrderArrivalEvent`. There is an `initialDelay` of 5000 nanoseconds before the File adapters starts to read the file. The downstream `OrderArrival` channel listens for events of type `OrderArrivalEvent`.

```
<wlevs:adapter id="inputAdapter" provider="file" >
        <wlevs:instance-property name="path"
            value="@wlevs.domain.home@/inpOrderArrival.txt"/>
        <wlevs:instance-property name="eventType" value="OrderArrivalEvent"/>
        <wlevs:instance-property name="initialDelay" value="5000"/>
        <wlevs:listener ref="OrderArrival"/>
</wlevs:adapter>
```

Configuration File

```
 <adapter>
    <name>inputAdapter</name>
 </adapter>
```

4.8 HTTP Publish-Subscribe Adapter

Use the HTTP Publisher adapter to send JavaScript Object Notation (JSON) event data out of the EPN to a web-based user interface. Use the HTTP Subscriber adapter to accept JavaScript Object Notation (JSON) event data entering the EPN. JSON event data comes from an HTTP server where user actions generate events.

The HTTP Publish-Subscribe server in Oracle Event Processing is based on the Bayeux protocol that is proposed by the cometd project. The Bayeux protocol defines a contract between the client and the server for communicating with asynchronous messages over HTTP.

You can create a remote or a local HTTP Publisher adapter, and a remote HTTP Subscriber adapter. Whether an HTTP adapter is local or remote is determined by the local or remote URL you supply to the required `<server-url>` child element.

The best way to create HTTP publish and subscribe adapters is to use Oracle JDeveloper.

Note:

Byte arrays are not supported as property types in event types used with the HTTP Publish and Subscribe adapters.

Assembly File

```
<wlevs:adapter id="http-pub-adapter" provider="httppub"/>

<wlevs:adapter id="http-sub-adapter" provider="httpsub" />
```

Configuration File

For every local and remote adapter, provide a URL to the server in the `server-url` property. The server can be an Oracle Event Processing server, a WebLogic Server instance, or any third-party HTTP Publish-Subscribe server.

For every local adapter for publishing, add a `server-context-path` element to specify the path to the local HTTP Publish-Subscribe server associated with the Oracle Event Processing instance hosting the current Oracle Event Processing application.

By default, each Oracle Event Processing server is configured with an HTTP Publish-Subscribe server with path `/pubsub`. If you have created a new local HTTP Publish-Subscribe server or changed the default configuration, then specify the location of the server in the server file. In the file specify the `http-pubsub` element `path` value with the location of the server. You can locate the server file in your Oracle Event Processing installation at: `/Oracle/Middleware/my_oep/examples/domains/<my_domain>/defaultserver`.

The `channel` child element specifies the channel that the adapter publishes or subscribes to.

```
<http-pub-sub-adapter>
  <name>http-pub-adapter</name>
  <server-url>http://myhost.com:9102/pubsub</server-url>
  <channel>/channel2</channel>
  <event-type>com.mycompany.httppubsub.PubsubEvent</event-type>
  <user>wlevs</user>
  <password>wlevs</password>
</http-pub-sub-adapter>

<http-pub-sub-adapter>
  <name>http-sub-adapter</name>
  <server-url>http://myhost.com:9102/pubsub</server-url>
  <channel>/channel2</channel>
  <event-type>com.mycompany.httppubsub.PubsubEvent</event-type>
</http-pub-sub-adapter>
```

4.9 HTTP Publish-Subscribe Adapter Custom Converter Bean

The HTTP Publish-Subscribe adapter converts incoming JavaScript Object Notation (JSON) messages to event types and back again. To customize the way inbound and outbound JSON messages are converted to an event type and back, create a custom converter bean.

4.9.1 Bayeux Protocol

The HTTP Publish-Subscribe (pub-sub) server is based on the Bayeux protocol that is proposed by the cometd project. The Bayeux protocol defines a contract between the client and the server for communicating with asynchronous messages over HTTP. The pub-sub server can communicate with any client that understands the Bayeux protocol.

You can develop your web client with the following frameworks:

- Dojo JavaScript library that supports the Bayeux protocol. Oracle Event Processing does not provide this library. You can find information about it at: http://dojotoolkit.org/.
- WebLogic Workshop Flex plug-in that enables development of a Flex client that uses the Bayeux protocol to communicate with a pub-sub server.

For information about securing an HTTP pub-sub server channel, see *Administering Oracle Event Processing*.

4.9.2 Create a Custom Converter Bean

A custom converter bean is a Java class that implements the following interfaces:

- `InboundMessageConverter` interface to convert inbound JSON messages to events.
- `OutboundMessageConverter` interface to convert events to JSON messages.

See the *Java API Reference for Oracle Event Processing* for a full description of these APIs.

Inbound HTTP Pub-Sub JSON Message

The custom converter bean for an inbound HTTP pub-sub JSON message implements the `com.bea.wlevs.adapters.httppubsub.api.InboundMessageConverter` interface. This interface has only the `convert` method:

```
public List convert(JSONObject message) throws Exception;
```

The `message` parameter is the inbound HTTP pub-sub message in JSON format. The return value is a `List` of events to pass to the next node in the EPN.

Outbound HTTP Pub-Sub JSON Message

The custom converter bean for an outbound HTTP pub-sub message implements the `com.bea.wlevs.adapters.httppubsub.api.OutboundMessageConverter` interface. This interface has only the `convert` method:

```
public List<JSONObject> convert(Object event) throws Exception;
```

The `event` parameter is an event received by the outbound HTTP pub-sub adapter from the source node in the EPN. The return value is a `List` of JSON messages.

Example

The following example shows a custom converter bean that implements both the `InboundMessageConverter` and `OutboundMessageConvert` interfaces. You can use this bean for both inbound and outbound HTTP pub-sub adapters.

Note:

You can use the GSON Java library to convert Java objects to JSON format. For more information, see http://www.json.org and http://code.google.com/p/google-gson.

```
package com.sample.httppubsub;
import com.bea.wlevs.adapters.httppubsub.api.InboundMessageConverter;
import com.bea.wlevs.adapters.httppubsub.api.OutboundMessageConverter;
import com.bea.httppubsub.json.JSONObject;
import java.util.List;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
public class TestConverter implements InboundMessageConverter, OutboundMessageConverter {
  public List convert(JSONObject message) throws Exception {
    List eventCollection = new ArrayList();
    PubsubTestEvent event = new PubsubTestEvent();
    event.setMessage("From TestConverter: " + message);
    eventCollection.add(event);
    return eventCollection;
  }
  public List<JSONObject> convert(Object event) throws Exception {
    List<JSONObject> list = new ArrayList<JSONObject>(1);
    Map map = new HashMap();
    map.put("message", ((PubsubTestEvent) event).getMessage());
    list.add(new JSONObject(map));
    return list;
  }
}
```

4.10 JMS Adapters

Use JMS adapters to connect the Java Message Service (JMS) with an Oracle Event Processing EPN to receive and send JMS messages. The Oracle Event Processing JMS adapters support any JMS service provider that provides a Java client that is compliant with Java EE.

The JMS Inbound adapter converts the incoming JMS messages to Oracle Event Processing events and the JMS outbound adapter converts Oracle Event Processing events to JMS messages. You can customize the inbound conversion by writing your own Java class. See *Customizing Oracle Event Processing*.

The best way to create JMS adapters is through the Oracle JDeveloper components window. The following assembly and configuration files generated by Oracle JDeveloper show the JMS inbound and outbound adapter configurations.

Note:

An exception that occurs in the `MessageConverter` object associated with a outbound JMS adapter does not cause the underlying JMS transaction to roll back. If the exception occurs outside of the `MessageConverter` object within the outbound JMS adapter, then an existing JMS transaction is rolled back.

4.10.1 Service Providers

Oracle Event Processing is tested against the following service providers:

- WebLogic T3 Client, which is a Java RMI client that uses Oracle T3 protocol to communicate with Oracle WebLogic Server.
- Version 10.0, 10.3, and 10.3.1 of Oracle WebLogic Server JMS
- The current version of Tibco EMS JMS

If the service provider you want to use is not in the list, you can configure Oracle Event Processing JMS adapters for use with your service provider by contacting your service provider and getting the `jndi-provider-url` and `jndi-factory` information needed for the `jms-adapter` configuration.

4.10.2 Inbound Adapter Configuration

Assembly File

```
<wlevs:adapter id="jms-inbound-adapter" provider="jms-inbound" />
```

Configuration File

The inbound adapter converts incoming JMS messages to a `TradeEvent`. The JNDI factory and service provider are `weblogic.jndi.WLInitialContextFactory` and `t3://localhost:7101`. The incoming client finds the adapter with the JNDI name of `JNDIName`. After the JMS adapter converts the JMS message to an event, the adapter sends the events to the JNDI destination of `Queue1`.

The optional `connection-jndi-name` element provides the JNDI name of the JMS connection factory. The required `destination-jndi-name` element provides the JNDI name of the JMS destination. The `session-transacted` element when `false` indicates that the session is not transactional.

```
<jms-adapter>
  <name>jms-inbound-adapter</name>
  <event-type>TradeEvent</event-type>
  <jndi-provider-url>t3://localhost:7101</jndi-provider-url>
  <jndi-factory>weblogic.jndi.WLInitialContextFactory</jndi-factory>
  <connection-jndi-name>JNDIName</connection-jndi-name>
  <destination-jndi-name>Queue1</destination-jndi-name>
  <session-transacted>false</session-transacted>
</jms-adapter>
```

4.10.2.1 Single and Multithreaded Inbound JMS Adapters

By default, an inbound JMS adapter is single-threaded. When the inbound JMS adapter is single-threaded, event order is guaranteed.

To improve scalability, you can configure an inbound JMS adapter to use multiple threads to read messages from the JMS destination. When the inbound JMS adapter is multi-threaded, event order is not guaranteed. To use multiple threads, configure the adapter with a work manager with the `work-manager` child element. You can specify a dedicated work manager to be used only by the adapter, or you can share a work manager among several components such as other adapters and Jetty.

4.10.2.2 Configure a JMS Adapter for Durable Subscriptions

You can configure an inbound JMS adapter to be a client in a durable subscription to a JMS topic. A durable subscription ensures that the adapter receives published messages even when the adapter becomes inactive. When the inbound adapter connects to the JMS server, it registers the durable subscription and subsequent messages sent to the topic are retained during periods when the subscriber is disconnected (unless they expire) and delivered when the subscriber reconnects.

A durable subscription assumes that the publisher that is publishing JMS messages to the topic is using the persistent delivery mode. Note that publisher might be the Oracle Event Processing outbound JMS adapter (in other words, its `delivery-mode` value must be `persistent`, the default value).

Create a Durable Subscription in the Adapter

1. Ensure that the JMS message publisher is delivering messages in persistent mode.
2. Specify a client ID for the connection factory. On Oracle WebLogic Server, the client ID can be set on the connection factory administratively with the console. You should have a dedicated connection factory configured for each adapter instance that is using durable subscribers.
3. Set the following three `jms-adapter` properties:
   - `destination-type` to `TOPIC`.
   - `durable-subscription` to `true`.
   - `durable-subscription-name` to a unique subscription identifier.

4.10.3 Outbound Adapter Configuration

The outbound JMS adapter converts events into a JMS map message and sends the JMS message to a JMS destination. You can also customize this conversion by writing your own Java class to specify exactly how you want the event types to be converted into outgoing JMS messages. See *Customizing Oracle Event Processing*.

Assembly File

```
<wlevs:adapter id="jms-outbound-adapter" provider="jms-outbound"/>
```

Configuration File

The JMS Outbound adapter configuration is almost the same as the JMS Inbound adapter configuration. The outbound adapter specifies a JMS destination and provides a user name and password to access the JMS destination. This session is transactional and non-persistent.

```
<jms-adapter>
  <name>jms-outbound-adapter</name>
  <event-type>TradeEvent</event-type>
  <jndi-provider-url>t3://localhost:7101</jndi-provider-url>
  <jndi-factory>weblogic.jndi.WLInitialContextFactory</jndi-factory>
  <connection-jndi-name>Topic</connection-jndi-name>
  <destination-jndi-name>Queue2</destination-jndi-name>
  <user>weblogic</user>
  <password>welcome1</password>
  <session-transacted>true</session-transacted>
  <delivery-mode>nonpersistent</delivery-mode>
</jms-adapter>
```

4.11 JMS Custom Message Converter Bean

To customize the conversion between JMS messages and event types, create inbound and outbound converter beans and package them with your Oracle Event Processing Application.

4.11.1 Implement Interfaces

The inbound and outbound converter beans implement methods in the following two inbound and outbound interfaces. See the *Java API Reference for Oracle Event Processing* for a full description of these APIs.

- **Inbound**:
  `com.bea.wlevs.adapters.jms.api.InboundMessageConverter`. You have to implement its `convert` method. The `return` value is a `List` of events to be passed downstream.

  ```
  public List convert(Message message)
       throws MessageConverterException, JMSException;
  ```

  `message` parameter: Corresponds to the incoming JMS message.

- **Outbound**:
  `com.bea.wlevs.adapters.jms.api.OutboundMessageConverter` interface. You have to implement its `convert` method. The `return` value is a `List` of JMS messages.

  ```
  public List<Message> convert(Session session, Object event)
       throws MessageConverterException, JMSException;
  ```

  `session` parameter: The `javax.jms.Session` to use to create the messages.

  `event` parameter: An event received by the outbound JMS adapter from the source stage in the EPN.

4.11.2 Implement the Inbound JMS Adapter

This example shows you how to implement the `convert` method for the inbound JMS adapter.

1. In Oracle JDeveloper, add a Java class to your application project.
2. Implement the
   `com.bea.wlevs.adapters.jms.api.InboundMessageConverter` interface.

   The following example shows a possible implementation.

```
package com.customer;
import com.bea.wlevs.adapters.jms.api.InboundMessageConverter;
import com.bea.wlevs.adapters.jms.api.MessageConverterException;
import com.bea.wlevs.adapters.jms.api.OutboundMessageConverter;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.TextMessage;
import java.util.ArrayList;
import java.util.List;
public class MessageConverter implements InboundMessageConverter,
    OutboundMessageConverter {
    public List convert(Message message) throws MessageConverterException, JMSException {
        TestEvent event = new TestEvent();
        TextMessage textMessage = (TextMessage) message;
        event.setString_1(textMessage.getText());
        List events = new ArrayList(1);
        events.add(event);
        return events;
    }
    public List<Message> convert(Session session, Object inputEvent)
```

```
        throws MessageConverterException, JMSException {
        TestEvent event = (TestEvent) inputEvent;
        TextMessage message = session.createTextMessage(
            "Text message: " + event.getString_1()
        );
        List<Message> messages = new ArrayList<Message>();
        messages.add(message);
        return messages;
    }
}
```

3. Specify the converter in your application EPN assembly file.

   - Register the converter class using a `bean` element.
   - Associate the converter class with the JMS adapter by adding a `wlevs:instance-property` with `name` set to `converterBean` and `ref` set to the `id` of `bean`.

   The following example shows how to register and associate the converter class.

```
...
    <bean id="myConverter" class="com.customer.MessageConverter"/>
    <wlevs:adapter id="jmsInbound" provider="jms-inbound">
        <wlevs:instance-property name="converterBean" ref="myConverter"/>
        <wlevs:listener ref="mySink"/>
    </wlevs:adapter>
```

4.11.3 Implement the Outbound JMS Adapter

This example shows you how to implement the `convert` method for the outbound JMS adapter.

1. Using the Oracle JDeveloper (or your preferred IDE), add a Java class to your application project.
2. Implement the `com.bea.wlevs.adapters.jms.api.OutboundMessageConverter` interface.

   The example shows a possible implementation.

```
package com.customer;
import com.bea.wlevs.adapters.jms.api.InboundMessageConverter;
import com.bea.wlevs.adapters.jms.api.MessageConverterException;
import com.bea.wlevs.adapters.jms.api.OutboundMessageConverter;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.TextMessage;
import java.util.ArrayList;
import java.util.List;
public class MessageConverter implements InboundMessageConverter,
    OutboundMessageConverter {
    public List convert(Message message) throws MessageConverterException, JMSException {
        TestEvent event = new TestEvent();
        TextMessage textMessage = (TextMessage) message;
        event.setString_1(textMessage.getText());
        List events = new ArrayList(1);
        events.add(event);
        return events;
    }
    public List<Message> convert(Session session, Object inputEvent)
        throws MessageConverterException, JMSException {
        TestEvent event = (TestEvent) inputEvent;
        TextMessage message = session.createTextMessage(
```

```
            "Text message: " + event.getString_1()
        );
        List<Message> messages = new ArrayList<Message>();
        messages.add(message);
        return messages;
    }
}
```

3. Specify the converter in your application EPN assembly file.

   - Register the converter class using a `bean` element.
   - Associate the converter class with the JMS adapter by adding a `wlevs:instance-property` with `name` set to `converterBean` and `ref` set to the `id` of `bean`.

   The following example shows how to register and associate the converter class.

```
<bean id="myConverter" class="com.customer.MessageConverter"/>
  <wlevs:adapter id="jmsOutbound" provider="jms-outbound">
    <wlevs:instance-property name="converterBean" ref="myConverter"/>
  </wlevs:adapter>
```

4.12 Oracle Business Rules Adapter

The Oracle Business Rules (OBR) adapter is an event bean that wraps the business rules engine from the Oracle Business Rules product. The OBR adapter lets you assert and retract events as facts to trigger business rules. You can configure OBR rules to generate events and add business logic to an Oracle CQL processor downstream to process the events. For example you can invoke `StreamSender.sendInsertEvent` within the rules file to send data out of an OBR adapter as an event.

Oracle JDeveloper does not provide a drag and drop component for the assembly file or the EPN diagram, but it does provide a drag-and-drop component for the configuration file.

You can create an OBR adapter by adding entries to the assembly file and by dragging and dropping the OBR adapter into the configuration file. For more information about creating OBR adapters by adding entries, see the OBR documentation at: `http://www.oracle.com/technetwork/middleware/business-rules/documentation/index.html`.

Assembly File

The `event-type-repository` element specifies the event type repository for the application. In the following example, the repository has a single event type named `HelloWorldEvent` and is implemented by the `HelloWorldEvent.java` class.

The next adapter specifies an `id` equal to `helloworldAdapter` ID with a value of the `HelloWorldAdapter` Java class. An adapter is created from the `HelloWorldAdapter` Java class. The OBR adapter configuration includes a `message` element with the specified message text. The `HelloWorldAdapter` class prints the `message` during application execution. In this example, the `HelloWorldAdapter` class is the event source.

The OBR adapter declaration comes after the channel and Oracle CQL processor configurations: `<wlevs:adapter id="OBRAdapter" provider = "obr">` followed by the `decisionFunction` and `dictionaryURL` properties. The `dictionaryURL` property is the path to the OBR dictionary file that contains the rules, and `decisionFunction` property is the OBR function you want to use. The `handler1` property is a handle for other components to access this information.

Note:

The OBR adapter does not handle automatic Fact retraction. If the upstream processor outputs a stream, retract the Fact in the rule file when appropriate or when the last rule is triggered according to rule priority.

At the bottom is the `HelloWorldBeand` configuration. The `HelloWorldBean` is a Java class that instantiates the `HelloworldEvent` and `HelloWorldAdapter` classes.

```
<wlevs:event-type-repository>
  <wlevs:event-type type-name="HelloWorldEvent">
    <wlevs:class>com.bea.wlevs.event.example.helloworld.HelloWorldEvent
    </wlevs:class>
  </wlevs:event-type>
</wlevs:event-type-repository>
<wlevs:adapter id="helloworldAdapter"
  class="com.bea.wlevs.adapter.example.helloworld.HelloWorldAdapter" >
  <wlevs:instance-property name="message" value="HelloWorld - The time is:"/>
</wlevs:adapter>

<wlevs:channel id="helloworldInputChannel" event-type="HelloWorldEvent" >
  <wlevs:listener ref="helloworldProcessor"/>
  <wlevs:source ref="helloworldAdapter"/>
</wlevs:channel>

<wlevs:processor id="helloworldProcessor" />

<wlevs:channel id="helloworldOutputChannel" event-type="HelloWorldEvent"
  advertise="true" max-threads="0" max-size="0" >
<wlevs:listener ref="OBRAdapter"/>
<wlevs:source ref="helloworldProcessor"/>
</wlevs:channel>

<wlevs:adapter id="OBRAdapter" provider = "obr">
<wlevs:instance-property name="decisionFunction" value="handler1" />
<wlevs:instance-property name="dictionaryUrl" value="file:helloworld.rules"/>
<wlevs:listener ref="OutputBean"/>
</wlevs:adapter>

<wlevs:event-bean id="OutputBean"
  class="com.bea.wlevs.example.helloworld.HelloWorldBean">
</wlevs:event-bean>
```

Configuration File

The configuration file declares the Oracle CQL processor and query rules to use to process the `HelloworldEvent` received from the OBR adapter. It also provides the OBR adapter handler (`handler1`) for accessing the OBR rules. The `dictionary-url` element specifies the path to the OBR dictionary file that contains the rules and decision function you want to use. The `decision-function` element specifies the name of the OBR decision function you want to use.

```
<processor>
  <name>helloworldProcessor</name>
  <rules>
    <query id="helloworldRule">
      <![CDATA[ select * from helloworldInputChannel[range 10 slide 5] ]] >
       select * from helloworldInputChannel[now]
    </query>
```

```
  </rules>
</processor>
<obr-adapter>
  <name>OBRAdapter</name>
  <dictionary-url>file:helloworld.rules</dictionary-url>
  <decision-function>handler1</decision-function>
</obr-adapter>
```

4.13 REST Adapter

The Representational State Transfer (REST) inbound adapter receives HTTP Post data from an external client through the HTTP protocol. A REST adapter can accept data in the following forms and convert that data into the Oracle Event Processing event configured on the inbound REST adapter: XML, CSV, and JavaScript Object Notation (JSON).

To convert data to events, the REST adapter requires a Java Architecture for XML Binding (JAXB) mapper and a CSV mapper. A mapper is a JavaBean class that implements the marshalling and unmarshalling of the incoming data.

Oracle JDeveloper does not currently provide a component for the REST adapter. However, you can create a REST adapter by adding entries to the assembly and configuration files for your Oracle Event Processing application.

Assembly File

The following assembly file shows the settings for an inbound REST adapter that handles input data of type XML, CSV, and JSON.

```
<bean id="xmlMapperBean" class="com.oracle.cep.mappers.jaxb.JAXBMapperFactory"
   factory-method="create" />
<bean id="csvMapperBean" class="com.oracle.cep.mappers.csv.CSVMapper" />

<bean id="jsonMapperBean"
  class="com.oracle.cep.mappers.jaxb.JAXBMapperFactory"
  factory-method="create" />
<wlevs:adapter id="restInbound" provider="rest-inbound">
<wlevs:instance-property name="mapper" ref="xmlMapperBean" />
<wlevs:instance-property name="csvMapper" ref="csvMapperBean" />
<wlevs:instance-property name="eventTypeName" value="CallCenterActivity" />
<wlevs:instance-property name="contextPath" value="/testhttpadapter" />
</wlevs:adapter>
```

The following assembly file shows the settings for an outbound REST adapter that unmarshalls an event to the XML or JSON content types.

```
<wlevs:adapter id="restXmlOutbound" provider="rest-outbound">
          <wlevs:instance-property name="mapper" ref="xmlMapperBean" />
          <wlevs:instance-property
   name="url" value="http://localhost:9002/testadapter" />
    </wlevs:adapter>

    <wlevs:adapter id="restJsonOutbound" provider="rest-outbound">
        <wlevs:instance-property name="mapper" ref="jsonMapperBean" />
        <wlevs:instance-property name="url"
   value="http://localhost:9002/testadapter" />
    </wlevs:adapter>
```

Configuration File

The following configuration file shows the `rest-adapter` configuration for receiving `POST` data, and the `jaxb-mapper` configuration for handling incoming XML and JSON data.

```
 <rest-adapter>
    <name>restInbound</name>
    <event-type-name>CallCenterActivity</event-type-name>
    <context-path>/testhttpadapter</context-path>
 </rest-adapter>
 <jaxb-mapper>
    <name>xmlMapperBean</name>
    <event-type-name>CallCenterActivity</event-type-name>
    <metadata>external_metadata_case1.xml</metadata>
 </jaxb-mapper>
<json-mapper>
  <name>jsonMapperBean</name>
  <event-type-name>CallCenterActivity</event-type-name>
  <media-type>application/json</media-type>
</json-mapper>
```

The following configuration file shows the settings for an outbound REST adapter that unmarshalls an event to the XML or JSON content types.

```
 <rest-adapter>
   <name>restXmlOutbound</name>
   <url>http://localhost:9002/testrestadapter</url>
 </rest-adapter>

 <rest-adapter>
   <name>restJsonOutbound</name>
   <url>http://localhost:9002/testrestadapter</url>
 </rest-adapter>

<jaxb-mapper>
  <name>xmlMapperBean</name>
  <event-type-name>CallCenterActivity</event-type-name>
  <metadata>external_metadata_case1.xml</metadata>
</jaxb-mapper>

<json-mapper>
  <name>jsonMapperBean</name>
  <event-type-name>CallCenterActivity</event-type-name>
  <media-type>application/json</media-type>
</json-mapper>
```

Note: To support XML content type in the REST inbound and outbound adapters, use the XML Mapper. Adding XML annotations or generating JAXB bindings file automatically is not supported in this release.

4.14 RMI Adapters

Use the RMI Inbound and Outbound adapters to read event information from and write event information to an RMI connection. The best way to create RMI adapters is through the Oracle JDeveloper components window. The following assembly and configuration files generated by Oracle JDeveloper show the RMI inbound and outbound adapter configurations.

Note:

The RMI client connection cannot be closed. See question F1 at http://docs.oracle.com/javase/8/docs/technotes/guides/rmi/faq.html.

Assembly File

The inbound RMI adapter has a JNDI name to enable inbound clients to locate the EPN.

```
<wlevs:adapter id="rmi-inbound-adapter" provider="rmi-inbound">
    <wlevs:instance-property name="jndiName"
      value="TradeReportApplication.TradeReport/rmi-inbound-adapter"/>
  </wlevs:adapter>
<wlevs:adapter id="rmi-outbound-adapter" provider="rmi-outbound"/>
```

Configuration File

The JNDI name enables the RMI outbound adapter to locate the output resource for the event data. The JNDI provider enables directory service implementations to be plugged into the JNDI framework.

In this example, the JNDI provider is the default Oracle WebLogic T3 client. Oracle WebLogic T3 clients are Java RMI clients that use the Oracle T3 protocol to communicate with Oracle WebLogic Server. T3 clients typically outperform other client types.

```
<rmi-adapter>
    <name>rmi-outbound-adapter</name>
    <jndi-name>RMIOutboundJNDIName</jndi-name>
    <jndi-provider-url>t3://localhost:7001</jndi-provider-url>
    <jndi-factory>weblogic.jndi.WLInitialContextFactory</jndi-factory>
 </rmi-adapter>
```

5

Channels

A channel represents the logical conduit through which events flow between other types of components (stages). For example, between adapters and Oracle CQL processors or between Oracle CQL processors and event beans.

This chapter includes the following sections:

5.1 When to Use a Channel

Channels provide buffering, queuing, and concurrency capabilities that enable you to tune the performance of your application later in the design life cycle. By default, the channel `max-threads` attribute is set to 0, which means the channel is in pass-through mode and incurs no performance penalty.

When constructing your EPN, consider the following rules:

- A channel is mandatory when you connect an Oracle CQL processor to a downstream stage.
- A channel is mandatory when you connect a stream or relation to an Oracle CQL processor.

  Note that based on the previous two points, it is mandatory to have a channel between an adapter and a processor. When you use Oracle JDeveloper to connect an adapter to a processor, the channel wizard displays for you to create the channel. See *Getting Started with Oracle Event Processing*.
- A channel is optional when you connect any of the following components to an Oracle CQL processor: an external relation, cache, or table source.

A channel is not needed between a pull source, such as a cache or table, and a processor because the pull source represents an external relation. For an external relation, the only valid operation is a join between a stream and a NOW window operator, and hence it is considered a pull source. The join actually happens outside of the Oracle CQL processor. Because it is a pull, the Oracle CQL processor does not need to be aware of its shape (that is, no DDL is required) and does not need the channel to act as intermediary.

In general, use a channel between components when:

- Buffering is needed between the emitting component and the receiver.
- Queueing or concurrency is needed for the receiving component.
- If a custom adapter is used and thread control is necessary.

5.2 Channel Configuration

When you add a channel to your Event Processing Network (EPN), it has a default configuration. The default channel has a name, an ID, is a system time-stamped stream channel, and has a default heartbeat time out of 100 milliseconds or 100,000,000 nanoseconds.

The default configuration is adequate for most applications.You can modify the configuration by editing the application assembly file or by editing the component configuration file.

When a channel is time stamped by the system, Oracle Event Processing assigns a new time from the CPU clock when a new event arrives and when the configurable heartbeat time out expires.

When a channel is time stamped by an application, the time stamp of an event is determined by the `wlevs:expression` element. A common example of an expression is a reference to a property on the event. If no expression is specified, then the time stamp can be propagated from a prior event. For example, when a channel that is time stamped by the system from one Oracle CQL processor feeds events into a channel that is time stamped by an application of another downstream Oracle CQL processor. In addition, an application can use the `StreamSender.sendHeartbeat` method to send an event of type `heart-beat` downstream to `StreamSink` listeners in the EPN.

Note:

When a channel is both application time stamped and map-based (uses a hash map event type), Oracle Event Processing adds a time stamp. A delete or update operation without a key does not work on a channel with this configuration because application time stamped events hold an always changing `timestamp` property.

This chapter describes some of the assembly and configuration file channel settings For a complete reference, see *Schema Reference for Oracle Event Processing*.

5.2.1 Assembly File

The assembly file shows the channel settings for the helloworldInputChannel. The settings indicate that helloworldProcessor listens to the channel for events, and that events flow into the channel from helloworldAdapter.

```
<wlevs:channel id="helloworldInputChannel" event-type="HelloWorldEvent" >
   <wlevs:listener ref="helloworldProcessor"/>
   <wlevs:source ref="helloworldAdapter"/>
</wlevs:channel>
```

To configure the channel as a relation, add the `is-relation` setting to the assembly file as follows:

```
<wlevs:channel id="helloworldInputChannel" event-type="HelloWorldEvent"
is-relation="true" primary-key="myprimarykey" />
```

If you make the channel a relation, you must also configure the `primary-key` attribute. The primary key is a list of event property names separated by white space or a comma that uniquely identifies each event. See `wlevs:metadata` in *Schema Reference for Oracle Event Processing* for information about how to define a primary key.

To configure the channel to be application time stamped, add the `application-timestamped` and `expression` elements to the assembly file as follows. When you set the `is-total-order` element to `true`, the application time published is always strictly greater than the last value used.

```
<wlevs:application-timestamped is-total-order="true">
   <wlevs:expression>mytime+10</wlevs:expression>
</wlevs:application-timestamped>
```

5.2.2 Configuration File

The configuration file shows the channel configuration settings. The settings customize the channel to buffer process events asynchronously (`max-size`), to use a maximum of 4 threads (max-threads), and to use a heartbeat time out of 10000 nanoseconds (heartbeat).

```
  <channel>
    <name>helloworldInputChannel</name>
    <max-size>10000</max-size>
    <max-threads>4</max-threads>
    <heartbeat>10000</name>
</channel>
```

5.3 Control Which Queries Output to a Downstream Channel

If you configure an Oracle CQL processor with more than one query, then by default, all queries send their results to the downstream channel. You can control which queries send their results to the downstream channel with the `selector` element.

Figure 5-1 shows an EPN with channel `filteredStream` connected to the upstream Oracle CQL processor, `filteredFanoutProcessor`.

Figure 5-1 EPN With Oracle CQL Processor and Downstream Channel

The following example shows the queries configured for the Oracle CQL processor.

```
<processor>
    <name>filterFanoutProcessor</name>
    <rules>
        <query id="Yr3Sector"><![CDATA[
            select cusip, bid, srcId, bidQty, ask, askQty, seq
            from priceStream where sector="3_YEAR"
        ></query>
        <query id="Yr2Sector"><![CDATA[
            select cusip, bid, srcId, bidQty, ask, askQty, seq
            from priceStream where sector="2_YEAR"
        ></query>
        <query id="Yr1Sector"><![CDATA[
            select cusip, bid, srcId, bidQty, ask, askQty, seq
            from priceStream where sector="1_YEAR"
```

```
        ></query>
    </rules>
</processor>
```

If you specify more than one query for an Oracle CQL processor, then by default, all query results are output to the Oracle CQL processor outbound channel (`filteredStream` in Figure 5-1). Optionally, in the component configuration source, you can use the `channel` element `selector` child element to specify a space-delimited list of one or more Oracle CQL query names that can output their results to the channel. In the following example, query results for query `Yr3Sector` and `Yr2Sector` are output to `filteredStream` but not query results for query `Yr1Sector`.

```
<channel>
    <name>filteredStream</name>
    <selector>Yr3Sector Yr2Sector</selector>
</channel>
```

You can configure a `channel` element with a `selector` before you create the queries in the upstream Oracle CQL processor. In this case, you must specify query names that match the names in the `selector`.

Note:

The `selector` child element is only applicable if the upstream stage is an Oracle CQL processor. For more information, see Oracle CQL Processors .

5.4 Batch Processing Channels

By default, a channel processes events as they arrive. You can configure a channel to batch events that have the same time stamp and were output from the same query by setting the `wlevs:channel` attribute `batching` to `true`. Batching events can improve application performance.

```
<wlevs:channel id="priceStream" event-type="PriceEvent" batching="true">
    <wlevs:listener ref="filterFanoutProcessor" />
    <wlevs:source ref="PriceAdapter" />
</wlevs:channel>
```

For related information, see Implement RelationSender, and `batching` in *Schema Reference for Oracle Event Processing*.

5.5 Fault Handling

You can write code to handle exceptions that occur in stages that are downstream from a channel and thrown to the channel. By default, the fault-handling behavior for a channel is as follows:

- If the channel `max-threads` setting is 0 (a pass-through channel), then the exception is thrown again to the next upstream stage in the EPN.
- If the channel `max-threads` setting is greater than 0, then the exception is logged and dropped. any events associated with the fault are also logged and dropped.

You can write a fault handling class and associate the handler with a channel with `max-threads` values that are greater than 0. With a fault handler associated with the channel, exceptions thrown to the channel are received by the handler, which contains code to either handle the fault or throw it again. If your fault handling code throws the

exception again, the exception is logged, but events related to the exception are lost. If you want to keep track of events involved in these exceptions, you must persist them with your code, such as by writing the event data to a data source connected to your EPN.

Note:

To handle an exception thrown by a multithreaded channel, the fault handler must be registered in a component that is upstream from the channel such as a processor. If you do not register the fault handler with an upstream component, the exception is passed upstream, but the fault handler is not invoked.

For information on writing fault handlers, see Fault Handling.

5.6 EventPartitioner Channels

By default, a channel broadcasts each event to every listener. When you configure a channel to use an `EventPartitioner`, each time an incoming event arrives, the channel selects a listener and dispatches the event to that listener instead of broadcasting each event to every listener. You can use an `EventPartitioner` on a channel to improve scalability.

For more information, see *Customizing Oracle Event Processing*.

6

Oracle CQL Processors

An Oracle CQL Processor processes incoming events from various input channels and other data sources. You use Oracle CQL to write the business logic in the form of continuous queries that process incoming events. Oracle CQL filters, aggregates, correlates, and processes events in real time.

Note:

You can create a Java class with methods that enhance the functionality available in Oracle CQL. Within Oracle CQL you reference the compiled class by name and call its methods from a `SELECT` statement. See *Oracle CQL Language Reference for Oracle Event Processing*.

This chapter includes the following sections:

- Processor Data Sources
- Assembly and Configuration Files
- Queries
- CQL Aggregations
- Configure a Table Source
- Configure an Oracle CQL Processor for Parallel Query Execution
- Fault Handling.

This chapter presents an overview of Oracle CQL with examples to help you understand the basic concepts. See Cached Event Data for information about performing CQL queries on caches. See also *Oracle CQL Language Reference for Oracle Event Processing* for a reference to Oracle CQL functions, queries, views, joins, pattern recognition capabilities, and statements.

Oracle JDeveloper provides Oracle CQL Pattern components that provide templates for creating common Oracle CQL queries. See *Getting Started with Oracle Event Processing*.

This chapter describes some of the assembly and configuration file Oracle CQL Processor settings For a complete reference, see *Schema Reference for Oracle Event Processing*.

6.1 Processor Data Sources

Oracle CQL queries can define one or more statements to process incoming event data from one or more input sources and send the outgoing event data to one or more

output channels. Each channel (input or output) and data source has an associated event type.

For example, one input can be a channel and another input can be a Coherence cache. The channel and Coherence cache have different event types because the Coherence cache provides additional information to the Oracle CQL processor query that is related to, but not the same as, the event data coming from the input channel.

If you configure an Oracle CQL processor with more than one query, by default, all queries output their results to all of the output channels. You can control which queries output their results to which output channels by putting a `selector` element on the downstream channel or channels. Use the `selector` element to specify a space delimited list of one or more query names that can output their results to that channel. The Oracle CQL query assigned to the output channel has the correct attributes to match the event type defined on the output channel. For more information, see Control Which Queries Output to a Downstream Channel.

6.2 Assembly and Configuration Files

When you add an Oracle CQL processor to your EPN, the assembly file shows the following entry.

```
<wlevs:processor id="processor"/>
```

When you add an Oracle CQL Pattern such as the Averaging Rule to the EPN in Oracle JDeveloper, the assembly file shows the following entries.

```
<wlevs:processor id="processor"/>
<wlevs:processor id="averaging-rule"/>
```

Configuration File

When you add the Oracle CQL processor to your EPN, the configuration file shows the following entry. By default, you get a template for rules that contains a template for one query.

```
<processor>
  <name>processor</name>
  <rules>
    <query id="ExampleQuery"><![CDATA[
      select * from MyChannel [now] >
    </query>
  </rules>
 </processor>
```

- The `rules` element groups the Oracle CQL statements that the Oracle CQL statements this processor executes.
- The `query` element contains Oracle CQL select statements. The `query` element `id` attribute defines the query name.
- The XML `CDATA` type indicates where to put the Oracle CQL rule.
- The `select` statement is the actual query. The template provides the `[now]` operator so that you can perform a `now` operation as described in NOW and Last Event Windows.

6.3 Queries

The following sections show how to perform basic Oracle CQL processor queries on stock trade events.

- Stream Channels
- Time-Based Relations (Windows)
- Processor Output Control (Slides)

Objective

The objective for this section is understand how to use windows, slides, and views in Oracle CQL queries.

- Windows convert event streams to time-based event relations to make it possible to perform Oracle CQL operations on the events. See Time-Based Relations (Windows).
- Slides enable you to batch events to control how the rate at which the CQL processor outputs events. See Processor Output Control (Slides).
- Views enable you to create an Oracle CQL statement that can be reused by other Oracle CQL queries. See Views.

Event Type Definition

The stock trade events used in the examples for this section are type `StockTradeEventType` with the following field and type definitions:

- `tickerSymbol`: `String`
- `price`: `Double`
- `dailyHigh`: `Double`
- `dailyLow`: `Double`
- `closingValue`: `Double`

6.3.1 Stream Channels

A stream channel inserts events into a collection and sends the stream to the next EPN stage. Events in a stream flow continuously, can never be deleted from the stream, and have no end. You can perform queries on the continuous stream of events flowing into your application.

A query on the input stream channel, `StockTradeIChannel`, to retrieve all stock trade events with the Oracle ticker symbol follows.

```
SELECT tickerSymbol
FROM StockTradeIStreamChannel
WHERE tickerSymbol = ORCL
```

The following configuration file entry shows the query. `ISTREAM` is a relation to stream operator described in Relation to Stream Operators.

```
<processor>
  <rules>
    <query id=rule1 <![CDATA[ISTREAM (SELECT tickerSymbol
```

```
      FROM StockTradeIStreamChannel WHERE tickerSymbol = ORCL)>
    </query>
  </rules>
</processor>
```

6.3.2 Time-Based Relations (Windows)

A relation channel inserts events into a collection and sends the relation to the next EPN stage. A relation is a window of time on the stream that has a beginning and an end. Events in a relation can be inserted into, deleted from, and updated in the relation. For insert, delete, and update operations, events in a relation must be referenced to a particular point in time to ensure the operation takes place on the correct event. All operations on a relation are time based.

Most applications do not use relation channels. You can put a window of time on events coming from a stream channel to create a relation for time-based processing operations. To find the average price for a particular stock, you must determine a time frame (window) in which to calculate the average. When you define a window on a stream, you have a collection of data that is not flowing, and unlike a stream, has a beginning and an end. The window is an in-memory relation on which you can apply a function such as `AVG` and also perform insert, update, and delete operations.

Operators that put a window of time on a stream are called stream to relation operators. The output of stream to relation operations are relations. You use relation to stream operators to convert a relation back to a stream to output a stream that contains every event, only inserted events, or only deleted events.

Oracle CQL processor output typically goes to a stream channel and on to the next stage in the EPN.

6.3.2.1 Stream to Relation Operators

The stream to relation operators are `RANGE` and `ROW`.

RANGE Operator

You can specify a window of time with the time-based window operator, `RANGE`, as follows:

```
SELECT AVG(price)
FROM StockTradeIStreamChannel [RANGE 1 MINUTE]
```

In this example and to keep the example easy to understand, the range is 1 minute, ticks in seconds, and one input event is received every second. The query starts averaging the prices contained in the events at zero seconds and outputs a value of 0 because there is no event in the relation at zero seconds. When the next event arrives at 1 second, the average price is the price in the first event. When the next event arrives at 2 seconds, the average price is the average of the two events in the relation. This continues until 59 seconds (1 minute) is reached.

An important concept with time-based window operators is that the window shifts over the event stream based on time. When 60 seconds have elapsed, the window shifts by one-second to average the prices in the events from 1 to 60 seconds, and when 60 more seconds are reached, the window shifts by one more second to average the prices in the events from 2 to 61 seconds. The window shifting over the relation behavior continues as long as the application runs.

The following configuration file entry shows the query. `ISTREAM` is a relation to stream operator described in Relation to Stream Operators.

```
<processor>
  <rules>
    <query id=rule2 <![CDATA[ISTREAM (SELECT AVG(price)
       FROM StockTradeIStreamChannel [RANGE 1 MIN >
    </query>
  </rules>
</processor>
```

Note:

Very large numbers must be suffixed. Without the suffix, Java treats very large numbers like an integer and the value might be out of range for an integer, which throws an error.

Add a suffix as follows:

`l` or `L` for Long`f` or `F` for float`d` or `D` for double`n` or `N` for big decimal

For example: `SELECT * FROM channel0[RANGE 1368430107027000000l nanoseconds]`

ROW Operator

You can specify a tuple-based window with the time-based `ROWS` operator as follows:

```
SELECT AVG(price)
FROM StockTradeIStreamChannel [ROWS 3]
```

A tuple is an event, so the `ROWS 3` operation means to average the price on three events in the relation starting when the first event arrives. The way it works is that the average operation is performed on the first event that enters the relation. When the second event enters the relation, the average operation is performed on the two events. When the third event enters the relation, the average operation is performed on the three events. No averaging occurs again until the fourth event enters the relation. When the fourth event enters the relation, the second, third, and fourth events are averaged. Likewise, when the fifth event enters the relation, the third, fourth, and fifth events are averaged.

The prior examples have averaged the price for all stocks. To compute the average for specific stocks in the stream, the following query uses a partitioned window.

```
SELECT AVG(price), tickerSymbol
FROM StockTradeIStreamChannel [PARTITION by tickerSymbol ROWS 3]
GROUP BY tickerSymbol
```

A partitioned window creates separate relation-windows for each partition. So in this example with the `PARTITION by tickerSymbol` clause, stocks with the same ticker symbol are grouped by three events and averaged. Without the partition and using only the `GROUP BY` clause, the tuple keeps the last three events as expected, but the ticker symbols in the tuple do not always match, which introduces averaging errors.

The following is the configuration file entry for this query. `ISTREAM` is a relation to stream operator described in Relation to Stream Operators.

```
<procesor>
  <rules>
    <query id="Example"><![CDATA[ISTREAM select tickerSymbol, AVG(price)
         from StockTradeIStream
         [PARTITION by tickerSymbol ROWS 3]
         GROUP BY tickerSymbol) >
```

```
    </query>
  </rules>
</processor>
```

6.3.2.2 Relation to Stream Operators

The relation to stream operators are `ISTREAM`, `DSTREAM`, and `RSTREAM`.

ISTREAM Operator

The `ISTREAM` operator puts an insert event from the relation into the output stream. Events that were deleted or updated in the relation are ignored. When the average changes, the query sends a `delete` event to the relation to remove the previous average and then sends an `insert` event to the relation to add the new average into the relation. The following example uses the `ISTREAM` operator to update the output stream when a new average is calculated.

```
ISTREAM (SELECT AVG(price)
FROM StockTradeIStreamChannel [RANGE 1 MINUTE])
```

The following configuration file entry shows the `ISTREAM` operator.

```
<processor>
  <rules>
    <query id=rule2 <![CDATA[ISTREAM (SELECT AVG(price)
       FROM StockTradeIStreamChannel [RANGE 1 MIN >
    </query>
  </rules>
</processor>
```

DSTREAM Operator

Use the `DSTREAM` operator to find out when a situation is no longer useful such as when a stock has been delisted from the exchange. The following example uses the `DSTREAM` operator to update the output stream with the old average after the new average is calculated in the relation.

```
DSTREAM (SELECT AVG(price)
FROM StockTradeIStreamChannel [RANGE 1 MINUTE])
```

The following configuration file entry shows the `DSTREAM` operator.

```
<processor>
  <rules>
    <query id=rule2 <![CDATA[DSTREAM (SELECT AVG(price)
       FROM StockTradeIStreamChannel [RANGE 1 MIN >
    </query>
  </rules>
</processor>
```

RSTREAM Operator

The `RSTREAM` operator inserts all events into the output stream regardless of whether events were deleted or updated. Use this operator when you need to take downstream action on every output. The following examples uses the `RSTREAM` operator to select all events in the input stream, wait for two events to arrive in the relation, and put the two events from the relation into the output stream.

```
RSTREAM (SELECT *
FROM StockTradeIStreamChannel [ROWS 2])
```

The following configuration file entry shows the `RSTREAM` operator.

```
<processor>
  <rules>
    <query id=rule2 <![CDATA[RSTREAM (SELECT *
       FROM StockTradeIStreamChannel [ROWS 2 >
    </query>
  </rules>
</processor>
```

6.3.2.3 NOW and Last Event Windows

A NOW window to contain the event that happened at the last tick of the system. With the NOW operator, the last input event can be deleted in the next time tick (the new NOW) so you might not have captured what you want. If you truly the last input event, use a last event window. The following example shows how to construct a NOW window.

```
SELECT * FROM StockTradeIStream[NOW]
```

The following configuration file entry shows the NOW operator.

```
<processor>
  <rules>
    <query id=rule2 <![CDATA[ISTREAM (SELECT *
       FROM StockTradeIStreamChannel [NOW>
    </query>
  </rules>
</processor>
```

A last event window captures the last event received. The following example shows how to construct a last event window.

```
SELECT * FROM StockTradeIStream[ROWS 1]
```

The following configuration file entry shows a last event window.

```
<processor>
  <rules>
    <query id=rule2 <![CDATA[ISTREAM (SELECT *
       FROM StockTradeIStreamChannel [ROWS 1 >
    </query>
  </rules>
</processor>
```

6.3.3 Processor Output Control (Slides)

Instead of outputting query results as they happen, you can use the SLIDE operator in a subclause to batch the output events. You can batch the events based on the number of events when you use the ROW operator or an amount of time (time window) when you use the RANGE operator.

Note:

When a slide value is not specified, the query assumes the default value of 1 row for tuple-based windows, and 1 time tick for time-based windows.

Batch by Number of Events

The following example outputs every 2 events (2, 4, 6, 8, ...).

```
SELECT * FROM StockTradeIStreamChannel[ROWS 3 SLIDE 2]
```

The output from the SLIDE operator includes deleted events. When the first two events arrive in the relation, the query outputs both events to the stream. When the next event arrives, there are three events in the relation, but output happens next at the fourth event. When the fourth event arrives, the first event is deleted and output with the third and fourth events.

The following example shows how to use a slide with the RSTREAM operator. In this case, when the fourth event arrives, events 2, 3, and 4 are sent to the output stream. The RSTREAM operator sends all events to the output stream regardless of whether events were deleted or updated.

```
RSTREAM(SELECT * FROM StockTradeIStreamChannel[ROWS 3 SLIDE 2])
```

The following configuration file entry uses an RSTREAM to batch by numbers.

```
<processor>
  <rules>
    <query id=rule2 <![CDATA[RSTREAM (SELECT *
       FROM StockTradeIStreamChannel [ROWS 3 SLIDE 2 >
    </query>
  </rules>
</processor>
```

Batch by Time Window

With a time window, Oracle Event Processing batches events by a time interval (RANGE operator). When you specify the time interval, Oracle CQL sends the events to the output stream at a time that is a multiple of the number you specified. For example, if you specify 5 seconds, the events are sent at 5, 10, 15, 20, and so on seconds. In the case where the first event arrives at 1, 2, or 3 seconds into the interval, the first output will be smaller than the others.

The following example specifies a range of 5 minutes with a slide every 30 seconds.

```
SELECT * FROM StockTradeIStream[RANGE 5 MIN SLIDE 30 SECONDS]
```

The following configuration file entry shows a time-based slide.

```
<processor>
  <rules>
    <query id=rule2 <![CDATA[RSTREAM (SELECT *
       FROM StockTradeIStreamChannel [RANGE 5 MIN 30 SECONDS >
    </query>
  </rules>
</processor>
```

6.3.4 Views

Views enable you to create an Oracle CQL statement that can be reused by other Oracle CQL queries. A `view` element contains Oracle CQL subquery statements. The `view` element `id` attribute defines the view name. A top-level SELECT statement that you create in a `view` element is called a view.

Note:

Subqueries are used with binary set operators such as union, union all, and minus). You must use parentheses in the subqueries so that the right precedence is applied to the query.

The following example shows view `v1` and query `q1` on the view. The view selects from stream `S1` of `xmltype` stream elements. The view `v1` uses the `XMLTABLE` clause to parse data from the `xmltype` stream elements with `XPath` expressions. The query `q1` selects from view `v1` as it would from any other data source. The `XMLTABLE` clause also supports XML name spaces.

An `xmltype` stream contains XML data. With the Oracle CQL `XMLTABLE` clause, you can parse data from an `xmltype` stream into columns using XPath expressions and access the data by column name. XPath expressions enable you to navigate through elements and attributes in an XML document.

Note:

The data types in the view's schema match the data types of the parsed data in the `COLUMNS` clause.

```
<view id="v1" schema="orderId LastShares LastPrice"><![CDATA[
  SELECT
    X.OrderId,
    X.LastShares,
    X.LastPrice
  FROM S1, XMLTABLE (
   "FILL"
   PASSING BY VALUE
   S1.c1 as "."
   COLUMNS
      OrderId char(16) PATH "fn:data(../@ID)",
      LastShares integer PATH "fn:data(@LastShares)",
      LastPrice float PATH "fn:data(@LastPx)"
   ) as X
></view>

<query id="q1"><![CDATA[
  IStream(
    select
      orderId,
      sum(LastShares * LastPrice),
      sum(LastShares * LastPrice) / sum(LastShares)
    from
      v1[now]
    group by orderId
  )
></query>
```

6.4 CQL Aggregations

Oracle CQL supports aggregate functions such as `AVG`, `COUNT`, `SUM`, which are calculated incrementally and `MAX`, and `MIN`, which are not incremental. The aggregate functions aggregate events into a Java collection so that you can use the Collection APIs to manipulate the events.

You can check for conditions on the aggregated results with the `HAVING` clause. In the following example only averages higher than 50 are output.

```
SELECT AVG(price) FROM StockTradeIStreamChannel [RANGE 1 HOUR]
HAVING AVG(price) > 50
```

Oracle CQL provides a variety of built-in single-row functions and aggregate functions based on the Colt open source libraries for high performance scientific and technical computing. The functions which are available as part of Colt library will not support Big Decimal data type and NULL input values. Also the value computation of the functions are not incremental. See the COLT website for details.

6.5 Configure a Table Source

You can access a relational database table from an Oracle CQL query by creating a table component with an associated data source. Oracle Event Processing relational database table event sources are pull data sources, which means that Oracle Event Processing periodically polls the event source.

- You can join a stream only with a `NOW` window and only to a single database table.

  Because changes in the table source are not coordinated in time with stream data, you can only join the table source to an event stream with a `Now` window, and you can only join to a single database table.

- With Oracle JDBC data cartridge, you can integrate arbitrarily complex SQL queries and multiple tables and data sources with your Oracle CQL queries. See *Developing Applications with Oracle CQL Data Cartridges*.

Note:

Oracle recommends the Oracle JDBC data cartridge for accessing relational database tables from an Oracle CQL statement.

Whether you use the `NOW` window or the data cartridge, you must define data sources in the Oracle Event Processing server file as described in *Administering Oracle Event Processing*.

6.5.1 Assembly File

The following assembly file entry shows the setting for a table source with an `id` attribute of `Stock`.

```
<wlevs:table id="Stock" event-type="TradeEvent" data-source="StockDataSource"/>
```

Oracle Event Processing uses the event type and the data source to map a relational table row to the event type. The `TradeEvent` event type is created from a Java class that has the following five private fields that map to columns in the relational database: `symbol`, `price`, `lastPrice`, `percChange`, and `volume`.

Note:

The `XMLTYPE` property is not supported for table sources.

6.5.2 Configuration File

```
<data-source>
  <name>StockDs</name>
  <connection-pool-params>
    <initial-capacity>1</initial-capacity>
    <max-capacity>10</max-capacity>
  </connection-pool-params>
```

```
    <driver-params>
      <url>jdbc:derby:</url>
      <driver-name>org.apache.derby.jdbc.EmbeddedDriver</driver-name>
      <properties>
        <element>
          <name>databaseName</name>
          <value>db</value>
        </element>
        <element>
          <name>create</name>
          <value>true</value>
        </element>
      </properties>
    </driver-params>
    <data-source-params>
      <jndi-names>
        <element>StockDs</element>
      </jndi-names>
      <global-transactions-protocol>None</global-transactions-protocol>
    </data-source-params>
  </data-source>
```

After configuration, you can define Oracle CQL queries that access the `Stock` table as if it were another event stream.

In the following example, the query joins the `StockTradeIStreamChannel` event stream to the `Stock` table:

```
SELECT StockTradeIStreamChannel.symbol, StockTradeIStreamChannel.price,
         StockTradeIStream.lastPrice, StockTradeIStream.percChange,
         StockTradeIStream.volume, Stock
FROM   StockTraceIStreamChannel [Now], Stock
WHERE  StockTradeIStreamChannel.symbol = Stock.symbol
```

Because changes in the table source are not coordinated in time with stream data, you can only join the table source to an event stream with a `Now` window, and you can only join to a single database table.

6.6 Configure an Oracle CQL Processor for Parallel Query Execution

For improved performance, you can enable a CQL query to execute in parallel rather than serially, as it does by default. When the CQL code supports it, you can configure a query so that it can process incoming events in parallel when multiple threads are available to the CQL processor.

You should enable parallel query execution only in cases where the relative order of the query output events is unimportant to the query's downstream client. For example, event ordering probably is not important if your query is intended primarily to filter events, such as to deliver to clients a set of stock transactions involving a particular company, where the transaction sequence is irrelevant.

By default (without enabling parallel execution), queries process events from a channel serially. For events routed through a channel that uses a system time stamp, event order is the order in which events are received; through a channel that is time stamped by an application, event order is the order determined by a time stamp value included in the event. Relaxing the total order constraint allows the configured query to not consider event order for that query, processing events in parallel where possible.

6.6.1 Set Up Parallel Query Execution Support

While specifying support for parallel query execution is at its core a simple configuration task, be sure to follow the other steps below so that you get the most out of the feature.

- Use the `ordering-constraint` attribute to support parallel execution.
- Make sure you have enough threads calling into the processor to meet your performance goals. The maximum amount of parallel query execution is constrained by the number of threads available to the CQL processor. For example, if an adapter upstream of the processor supports the number of threads you need and there is a channel between the adapter and the processor, try configuring the channel with a `max-threads` count of 0 so that it acts as a pass-through.

  If you don't want a pass-through, be sure to configure the query's upstream channel with a `max-threads` value greater than 1. (To make a `max-threads` value setting useful, you'll need to also set the `max-size` attribute to a value greater than 0.) For more information, see Channels .
- Follow other guidelines related to setting the `max-threads` attribute value. For example, to make a `max-threads` value setting useful, you'll need to also set the `max-size` attribute to a value greater than 0.
- Ensure, if necessary, that a bean receiving the query results is thread-aware, such as by using synchronized blocks. For example, you might need to do so if the bean's code builds a list from results received from queries executed on multiple threads.

6.6.2 The ordering-constraint Attribute

You enable parallel query execution by relaxing the default ordering constraint that ensures that events are processed serially. You do this by setting the `ordering-constraint` attribute on a `query` or `view` element.

In the following example, the `ordering-constraint` attribute is set to `UNORDERED` so that the query will execute in parallel whenever possible:

```
<query id="myquery" ordering-constraint="UNORDERED">
    SELECT symbol FROM S WHERE price > 10
</query>
```

The `ordering-constraint` attribute supports the following three values:

- `ORDERED` means that the order of output events (as implied by the order of input events) is important. The CQL engine will process events serially. This is the default behavior.
- `UNORDERED` means that order of the output events is *not* important to the consumer of the output events. This gives the freedom to the CQLProcessor to process events in parallel on multiple threads. When possible, the query will execute in parallel on multiple threads to process the events.
- `PARTITION_ORDERED` means that you're specifying that order of output events within a partition is to be preserved (as implied by the order of input events) while order of output events across different partitions is *not* important to the consumer of the output events. This relaxation provides some freedom to the CQL engine to process events across partitions in parallel (when possible) on multiple threads.

Use the PARTITION_ORDERED value when you want to specify that events conforming to a given partition are processed serially, but that order can be disregarded across partitions and events belonging to different partitions may be processed in parallel. When using the PARTITION_ORDERED value, you must also add the partition-expression attribute to specify which expression for partitioning should be the basis for relaxing the cross-partition ordering constraint.

In the following example, the GROUP BY clause partitions the output based on symbol values. The partition-expression attribute specifies that events in a given subset of events corresponding to a particular symbol value should be handled serially. Across partitions, on the other hand, order can be disregarded.

```
<query id="myquery" ordering-constraint="PARTITION_ORDERED"
    partitioning-expression="symbol">
    SELECT
        COUNT(*) as c, symbol
    FROM
        S[RANGE 1 minute]
    GROUP BY
        symbol
</query>
```

6.6.3 Using partition-order-capacity with Partitioning Queries

In general, you will probably see improved performance for queries by making more threads available and setting the ordering-constraint attribute so that they're able to execute in parallel when possible. As with most performance tuning techniques, a little trial and error with these settings should yield a combination that gets better results.

However, in some cases where your queries use partitioning -- and you've set the ordering-constraint attribute to PARTITION_ORDERED -- you might not see the amount of scaling you'd expect. For example, consider a case in which running with four threads doesn't improve performance very much over running with two threads. In such a case, you can try using the partition-order-capacity value to get the most out of CQL engine characteristics at work with queries that include partitions.

The partition-order-capacity value specifies the maximum amount of parallelism that will be permitted within a given processor instance when processing a PARTITION_ORDERED query. When available threads are handling events belonging to different partitions, the value sets a maximum number of threads that will be allowed to simultaneously run in the query.

As with other aspects of performance tuning, getting the most out of partition-order-capacity may take a bit of experimentation. When tuning with partition-order-capacity, a good starting point is to set it equal to the maximum number of threads you expect to have active in any CQL processor instance. In some cases (for example, at high data rates or with expensive processing downstream from the CQL processor), it may be helpful to set the partition-order-capacity value even higher than the available number of threads. However, you should only do this if performance testing confirms that it's helpful for a given application and load.

The partition-order-capacity value is set from one of four places, two of which you can fall back on when you do not explicitly set it yourself. For information about the settings, see *Schema Reference for Oracle Event Processing*.

These are, in order of precedence.

1. The `partition-order-capacity` element set on a channel configuration. If you specify this on the input channel for a processor, it takes effect for any `PARTITION_ORDERED` queries in that processor.
2. The `partition-order-capacity` property in server configuration. This value will be used for all `PARTITION_ORDERED` queries running on the server unless the value is set on a channel.
3. The `max-threads` value set on a channel configuration. If you specify this on the input channel for a processor, it takes effect for any `PARTITION_ORDERED` queries in that processor
4. A system default value (currently set to 4) is used if you don't specify either a `partition-order-capacity` value or `max-threads` value, or if the `max-threads` value is set to 0 (meaning it's a pass-through channel).

When using `partition-order-capacity`, keep in mind the following:

- The `partition-order-capacity` value is only useful when you're setting the `ordering-constraint` attribute to `PARTITION_ORDERED`.
- Increasing `partition-order-capacity` generally increases parallelism and scaling. For example, if your profiling reveals lock contention bottlenecks, you might find it helpful to increase `partition-order-capacity` to see if contention is reduced.
- Setting `partition-order-capacity` even higher than the number of available threads can be helpful in some cases because of the particular way partitioning is done in the CQL processor.
- There is some resource cost in memory used by specifying very high values.
- Tuning this parameter is very dependent on details of the application and the input rate. Tuning by experimentation may be necessary to determine an optimal value.

6.6.4 Limitations

Think of parallel query execution as a performance enhancement feature that you specify support for so that the CQL processor can use it *whenever possible*. Not all queries can be executed in parallel. This includes queries using certain CQL language features.

For example, if your query uses some form of aggregation -- such as to find the maximum value from a range of values -- the CQL processor may not be able to fully execute the query in parallel (this is needed to guarantee the correct result considering the ordering constraint). Some query semantics in themselves also constrain the query to ordered processing. Such queries will be executed serially regardless of whether you specify support for parallel execution.

Also, the `IStream`, `RStream` and `DStream` operators maintain the state of their operand for processing, making it necessary for the CQL processor to synchronize threads in order to execute the query.

Note that the CQL processor always respects the semantic intention of your query. In cases where the `ordering-constraint` attribute would change this intention, the attribute is coerced to a value that keeps the intention intact.

If you're using the `partitioning-expression` attribute, keep in mind that the attribute supports a single expression only. Entering multiple property names for the value is not supported.

6.7 Fault Handling

You can write code to handle faults that occur in code that does not have an inherent fault handling mechanism. This includes Oracle CQL code and multithreaded EPN channels. By default, the CQL language has no mechanism for handling errors that occur, as does Java with its try/catch structure. To handle faults that occur in CQL, you can write a fault handler, then connect the handler to the EPN stage for which it handles faults, such as an Oracle CQL processor.

You can also associate a fault handler with a multithreaded channel, which is a channel whose `max-threads` setting is greater than 0. This provides fault handling in the case of exceptions that are thrown to the channel from a stage that is downstream of the channel. Note that channels whose `max-threads` setting is 0 are pass-through channels that already rethrow exceptions to their upstream stages. For additional information specific to fault handlers for channels, see Fault Handling.

A fault handler is a Java class that implements the `com.bea.wlevs.ede.api.FaultHandler` interface. You connect the class to an EPN stage by registering your fault handler as an OSGi service and associating it with the stage. For more information about OSGi, see Spring Framework..

Without a custom fault handler, you get the following default fault handling behavior:

- When an exception occurs in Oracle CQL, the CQL engine catches the exception and stops the query processor.
- If an exception occurs in a stage that is downstream to the processor, then that stage is dropped as a listener.
- Exceptions are logged (under the CQLServer category) and the events that are part of the exception clause are discarded.
- Upstream stages are not notified of the failure.

When using custom fault handlers you write, you can:

- Associate a fault handler with an Oracle CQL processor or multithreaded channel so that faults in those stages are thrown as exceptions to the handler. There, you can handle or rethrow the exception.
- Allow query processing to continue as your code either handles the exception or rethrows it to the stage that is next upstream.
- Save event data from being lost while handling a fault. For example, if you have configured a connection to a data source, you could save event data there.
- Log fault and event information when faults occur.
- Use multiple fault handlers where needed in an EPN so that exceptions thrown upstream are handled when they reach other Oracle CQL processors and channels.

Consider associating a fault handler with a stage that does not have its own mechanism for responding to faults, including Oracle CQL processors and multithreaded channels. Other stages, such as custom adapters that have their own exception-handling model, do not benefit from a fault handler.

Queries can continue as your fault handling code evaluates the fault to determine what action should be taken, including rethrowing the fault to a stage that is upstream from the Oracle CQL processor.

For example, the upstream stage of the Oracle CQL processor could be the JMS subscriber adapter, which can roll back the JMS transaction (if the session is transacted) to allow the event to be redelivered. It can also commit the transaction if the event has been redelivered already and found that the problem is not solvable.

Note that when you use a custom fault handler, the query state is reset after a fault as if the query had been stopped and restarted. In contrast the default behavior stops the query and drops all subsequent events.

6.7.1 Implement a Fault Handler Class

You create a fault handler class by implementing the `com.bea.wlevs.ede.api.FaultHandler` interface. After you have written the class, you associate it with the stage for which it handles faults by registering it as an OSGi service. For more information, see Register a Fault Handler.

Your implementation of the `handleFault` method receives exceptions for the EPN stage with which the handler is associated. The exception itself is either an instance of `com.bea.wlevs.ede.api.EventProcessingException` or, if there has been a JVM error, an instance of `java.lang.Error`.

The method also receives a string array that contains the names of upstream stages, or *catchers*, to which the exception goes when your code rethrows it. If there is more than one catcher in the array, your rethrown exception goes to all of them. There are two cases when the catchers array is empty: when the exception occurs while executing a temporal query and when the exception is thrown to a channel's fault handler. In these cases, the fault handler executes in the context of a background thread, and there is no linkage to upstream stages.

An exception that is rethrown from a fault handler travels through upstream EPN stages until it is either caught or reaches a stage that cannot catch it (such as a processor or multithreaded channel that does not have an associated fault handler). Note that if you rethrow an exception, any channels in the catcher's list must have an associated fault handler to catch the exception.

The `EventProcessingException` instance could also be one of the exception types that extend that class, including `CQLExecutionException`, `ArithmeticExecutionException`, and others. See the *Java API Reference for Oracle Event Processing*. The `EventProcessingException` instance provides methods with which your code can retrieve insert, delete, and update events that were involved in generating the fault.

Your implementation of the method should do one of the following:

- Consume the fault in the way that a Java `try` and `catch` statement might. If your implementation does not rethrow the fault, then event processing continues with subsequent events. However, query processing continues with its state reset as if the query had been restarted. The processing state is lost and processing begins fresh with events that follow those that provoked the fault.
- Rethrow the fault so that it is received by upstream stages (or their fault handlers). As when the fault is consumed, queries continue processing events, although the query state is reset with subsequent events. The upstream stage receiving the fault always has the option of explicitly stopping the offending query by using the CQL processor's MBean interface.

Note:

When you update an Oracle CQL query with an MBean, do not send events during the update procedure. If you send events during some queries, the order of the events in the output stream is not guaranteed. For example, when you update an Oracle CQL query from unordered to ordered in an Oracle CQL parallelism execution.

In the following example the code provides a high-level illustration of handling a fault.

```
package com.example.faulthandler;

import com.bea.wlevs.ede.api.FaultHandler;

public class SimpleFaultHandler implements FaultHandler
{
    private String suppress;

    // Called by the server to pass in fault information.
    @Override
    public void handleFault(Throwable fault, String[] catchers) throws Throwable
    {
        // Log the fault.
        return;
    }
}
```

6.7.2 Register a Fault Handler

After you have written a fault handling class, you can associate it with an EPN stage by registering it as an OSGi service. The simplest way to do this is to register the handler declaratively in the EPN assembly file.

Note:

Due to inherent OSGi behavior, runtime fault handler registration from your configuration happens asynchronously, meaning that a small amount of warm-up time might be required before the handler can receive faults. To be sure your handler is ready for the first events that enters the network, add a wait period before the application begins to receive events.

In the following example, the EPN assembly file excerpt shows a `service` element stanza that registers the `SimpleFaultHandler` class as the fault handler for the Oracle CQL processor with an `id` of `exampleProcessor`.

```
<osgi:service interface="com.bea.wlevs.ede.api.FaultHandler">
    <osgi:service-properties>
        <entry key="application.identity" value="myapp"/>
        <entry key="stage.identity" value="exampleProcessor"/>
    </osgi:service-properties>
    <bean class="com.example.faulthandler.SimpleFaultHandler"/>
</osgi:service>

<!-- A processor with a user-defined function. -->
<wlevs:processor id="exampleProcessor" >
```

```
    ...
</wlevs:processor>
```

For more on the schema for registering OSGi services, see http://static.springsource.org/osgi/docs/1.1.x/reference/html/appendix-schema.html. For more on OSGi, see http://en.wikipedia.org/wiki/OSGi.

7

Event Beans

Java is the language you use to write logic for event bean and Spring bean components to add to the EPN. Use an event bean in your EPN to define application logic that works on event data. Use a Spring bean in your EPN when your deployment context and the features you want to use are based on Spring.

Event bean application logic functions as an event sink, an event source, or both. An event sink receives and works on large quantities of event data. An event source sends large quantities of event data. In an EPN, you can configure event beans and adapters with logic to make them behave as event sources, event sinks, or both. In the case of an event bean, the event sink and event source logic comes from its associated JavaBean. In the case of an adapter, the event sink and event source logic comes from its JavaBean event type. See Events and Event Types for information about creating a JavaBean event type.

You can use JAXB in event bean logic. See JAXB Support for information.

This chapter includes the following sections:

7.1 Event Beans and Spring Beans

Event beans and Spring beans are based on Java classes. The Java class you use for an event or Spring bean can conform to the JavaBean specification or not conform, depending on your application requirements. An event bean is an Oracle extension to the regular Spring-based bean.

An event bean can be an event sink, event source, or both an event sink and an event source. You can add event sinks and sources to adapters and event beans.

- An event sink is a JavaBean or Java class that listens for and works on events. An event sink can receive events, retrieve data from the events, and create a new event from the data to send to a downstream component.
- An event source is a JavaBean or Java class that sends events.

If your deployment context and the features you want to use are based on Spring, use a Spring bean. Otherwise, use an event bean. Table 7-1 lists the features provided by event beans and Spring beans.

Table 7-1 Comparison of Event Beans and Spring Beans

Table 7-1 (Cont.) Comparison of Event Beans and Spring Beans

Bean Type	Description
Event bean	Useful as an EPN stage to actively use the capabilities of the Oracle Event Processing server container. An event bean: • Is a type of Oracle Event Processing EPN stage. • Can be monitored by the Oracle Event Processing monitoring framework. • Can make use of the configuration metadata annotations. • Can be set to record and play back events that pass through it. • Can participate in the Oracle Event Processing server bean life cycle by specifying methods in its XML declaration, rather than by implementing Oracle Event Processing server API interfaces.
Spring bean	Useful for legacy integration to Spring. A Spring bean: • Is useful if you have a Spring bean you want to add to an EPN. • Is not a type of Oracle Event Processing EPN stage. • Cannot be monitored by the Oracle Event Processing monitoring framework. • Cannot use the configuration metadata annotations. • Cannot be set to record and play back events that pass through it.

7.1.1 Threading Behavior

Event beans are executed in parallel when they implement either the `Runnable` or the `RunnableBean` interface. The infrastructure uses a work manager associated with the application for spawning the thread.

You can associate a work manager to an application by naming the work manager with the same name as the application. If you do not explicitly specify a work manager for your application, then Oracle Event Processing creates a work manager with default values for the minimum (`MIN`) and maximum (`MAX`) number of threads.

If you need finer control over the threading, a custom event bean can implement the interface `com.bea.wlevs.ede.spi.WorkManagerAware`. In this case, the event bean is injected with the work manager of the application during initialization. The work manager can be used to explicitly manage the threading for the event bean instance.

7.1.2 Receive Heartbeat Events

Implement the `com.bea.wlevs.ede.api.HeartbeatAware` interface if you want your event bean to receive heartbeat events. A heartbeat event is an event of type `heartbeat` that you can use to model the advance of time. This interface has the `onHeartbeat(long timestamp)` callback method to implement.

7.1.3 Create an Event Bean

An event bean is an EPN component that applies logic to events as they pass through. The event bean logic is defined by its JavaBean event type.

This chapter describes some of the assembly and configuration file event bean settings For a complete reference, see *Schema Reference for Oracle Event Processing*.

Assembly File

The following event bean assembly file entry shows the event bean `id`, associated `class`, and that the event bean listens for events from the upstream `BeanOutputChannel` component.

```
<wlevs:event-bean id="eventBean" class="tradereport.TradeEvent" >
   <wlevs:listener ref="BeanOutputChannel"/>
<wlevs:event-bean>
```

Configuration File

The following event bean configuration file entry shows an event bean configured with the `record-parameters` child element:

```
 <event-bean>
    <name>eventBean</name>
      <record-parameters>
        <dataset-name>tradereport_sample</dataset-name>
          <event-type-list>
            <event-type>TradeEvent</event-type>
          </event-type-list>
          <batch-size>1</batch-size>
          <batch-time-out>10</batch-time-out>
        </record-parameters>
  </event-bean>
```

7.1.4 Create a Spring Bean

You can configure a Java class as a Spring bean to include the class in an event processing network. This is a good option if you have an existing Spring bean that you want to incorporate into the EPN or if you want to incorporate Spring features into your Java code.

In a Spring bean you plan to add to an EPN, you can implement the various life cycle interfaces. These include `InitializingBean`, `DisposableBean`, and the active interfaces, such as `RunnableBean`. The Spring bean event source can also use configuration metadata annotations such as `@Prepare`, `@Rollback`, and `@Activate`.

A Spring bean is a Java class managed by the Spring framework. You add a class as a Spring bean by configuring it in the EPN assembly file using the standard `bean` element.

A Spring bean is not an Oracle Event Processing stage. You cannot monitor a Spring bean with the Oracle Event Processing monitoring framework, you cannot use the configuration metadata annotations in a Spring bean, and you cannot set a Spring bean to record and play back events that pass through it.

Assembly File

In the assembly file, you use the `bean` element to declare a custom Spring bean as a component in the event processor network. For example:

```
<bean id="TradeListenerBean"
  class="com.oracle.cep.example.tradereport.TradeListener">
</bean>
```

7.2 Event Sink Interfaces

You create an event sink to receive events in an EPN and apply logic that responds to the event data. A Java class that is an event sink implements one of the interfaces described in this section. Each of these interfaces provides methods that the Oracle

Event Processing server uses to pass events to the class as the events exit the EPN stage connected upstream from the Java class, which is typically a channel.

The interfaces described here provide support for events arriving either as streams or relations. However, interfaces for relations also support receiving events arriving as streams. As described in the following table, the interfaces are hierarchically related.

Interface	Description
`com.bea.wlevs.ede.api.StreamSink`	Implement to receive events sequentially in a stream.
`com.bea.wlevs.ede.api.RelationSink`	Implement to receive events sequentially in a relation. Extends `StreamSink` to receive events in a stream.
`com.bea.wlevs.ede.api.BatchStreamSink`	Implement to receive batched events in a stream. Events might arrive batched by time stamp when the upstream channel allows batching. Extends `StreamSink` to support receiving events unbatched.
`com.bea.wlevs.ede.api.BatchRelationSink`	Implement to receive batched events as a relation. Events might arrive batched by time stamp when the upstream channel allows batching. Extends `RelationSink` to support receiving events unbatched as streams or relations.

EventRejectedException Behavior in onInsertEvent Implementations

You need to explicitly throw `EventRejectedException` in `onInsertEvent` implementations for exceptions you do not want to get dropped. You can raise an `EventProcessingException` and it is propagated all the way to the source of the error through a CQL processor. An EventRejectedException can chain exceptions from its downstream listeners, in case there is more than one exception. The CQL processor converts the `EventRejectedException` to a soft exception. See Fault Handling for more information.

7.2.1 Implement StreamSink

A class that receives events as a stream only receive events that are, from the Oracle Event Processing standpoint, inserted. That is because in a stream, events are always appended to the end of a sequence. Events in a stream are also always received in ascending time order so that their time stamps have non-decreasing values from one event to the one that follows. Non-decreasing time stamps enables the time stamp of one event to be the same as the time stamp of the event that precedes it, but not earlier than that preceding time stamp. The time stamp is either the same or later.

As a result, the interfaces that support receiving events as a stream have one method each for receiving events. The interfaces for receiving events as a relation support receiving multiple kinds of events.

Implement the `StreamSink` interface if your class receives unbatched events as a stream. The `StreamSink` interface has a single method, `onInsertEvent`, which the Oracle Event Processing server calls to pass in each event from the stream as events leave the upstream stage that is connected to your class.

In Example 7-1, a simple `StreamSink` implementation that receives stock trade events where each event is an `Object` instance, and tests to see whether the event is an instance of a particular event type. If so, then the code retrieves values of properties known to be members of that type.

You implement the `BatchStreamSink` interface if you expect your class to receive batched events as a stream. The interface has a single method, `onInsertEvents`, which the Oracle Event Processing server calls to pass in a collection of events received from the upstream stage. The `BatchStreamSink` interface extends `StreamSink`, so can receive unbatched events also.

For more information about event batching, see Batch Processing Channels.

Example 7-1 Implement the StreamSink Interface

```
public class TradeListener implements StreamSink {

    public void onInsertEvent(Object event) throws EventRejectedException {
        if (event instanceof TradeEvent){
            String symbolProp = ((TradeEvent) event).getSymbol();
            Integer volumeProp = ((TradeEvent) event).getVolume();
            // Code to do something with the property values.
        }
    }
}
```

7.2.2 Implement RelationSink

A class that receives events as a relation can receive any of the kinds of events possible in a relation, which are insert events, delete events, and update events. Unlike a stream, events in a relation are unordered and include events that have been updated or deleted by code that created or operated on the relation.

As a result, the interfaces that support receiving events as a relation have methods through which your class can receive insert, delete, or update events.

You implement the `RelationSink` interface if your class receives unbatched events as a relation. The `RelationSink` interface has three methods, one of which it inherits from the `StreamSink` interface: `onInsertEvent`, `onDeleteEvent`, and `onUpdateEvent`. At runtime, the Oracle Event Processing server calls the appropriate method depending on which type of event is received from the upstream channel connected to your class.

```
public class TradeListener implements RelationSink {

    public void onInsertEvent(Object event) throws EventRejectedException {
        if (event instanceof TradeEvent){
            String symbolProp = ((TradeEvent) event).getSymbol();
            Integer volumeProp = ((TradeEvent) event).getVolume();
            // Do something with the inserted event.
        }
    }

    @Override
    public void onDeleteEvent(Object event) throws EventRejectedException {
        if (event instanceof TradeEvent){
            // Do something with the deleted event.
        }
    }

    @Override
```

```
    public void onUpdateEvent(Object event) throws EventRejectedException {
        if (event instanceof TradeEvent){
            // Do something with the updated event.
        }
    }
}
```

Implement the `BatchRelationSink` interface if your class receives batched events as a relation. It has an `onEvents` method designed to receive all three types of events from the batch in `java.util.Collection` instances:

```
onEvents(insertEvents, deleteEvents, updateEvents)
```

In addition, the BatchRelationSink interface extends the `RelationSink` interface to support receiving unbatched events.

At runtime, the Oracle Event Processing server calls the appropriate method to pass in events received from the upstream stage connected to your class.

For more information about event batching, see Batch Processing Channels.

For complete API reference information about the Oracle Event Processing APIs described in this section, see the *Java API Reference for Oracle Event Processing*.

7.3 Event Source Interfaces

You can create a Java class that sends events to a downstream stage in an event processing network. You can create an event source, for example, to send events your Java code has created or altered from event data flowing through the EPN.

A Java class that is an event source implements one of the interfaces described in this section. Each of these interfaces provides a method used by the Oracle Event Processing server to pass into your class an instance of a sender class.

The sender instance your event source receives implements one of the sender interfaces described in this section. The sender interfaces provide methods your code can call to send events as streams or relations, and batched or unbatched to the downstream EPN stage that follows, such as a channel.

The interfaces described here support sending events either as streams or relations. Interfaces for relation also support sending events as streams.

Table 7-2 Interfaces for Implementing an Event Source

Interface	Description
`com.bea.wlevs.ede.api.StreamSource`	Implement this interface to send events as a stream. At runtime, the Oracle Event Processing server injects an instance of a stream sender class.
`com.bea.wlevs.ede.api.RelationSource`	Implement this interface to send events as a relation or stream. At runtime, the Oracle Event Processing server injects an instance of a relation sender class. Extends `StreamSource`, so it also supports stream events.

The interfaces listed in Table 7-3 are implemented by sender classes your event source class receives from the Oracle Event Processing server.

Table 7-3 Interfaces Implemented by Sender Classes

Interface	Description
`com.bea.wlevs.ede.api.StreamSender`	Provides a method to send events as a stream.
`com.bea.wlevs.ede.api.RelationSender`	Provides methods to send events as a relation. Extends `StreamSender`, so it also support stream events.
`com.bea.wlevs.ede.api.BatchStreamSender`	Provides a method with which your code can send batched events as a stream. You might send events batched by time stamp if the downstream stage to which you're sending them is a channel configured for batched events. Extends `StreamSender`, so it also provides support for sending events unbatched.
`com.bea.wlevs.ede.api.BatchRelationSender`	Provides a method to send batched events as a relation. You can send events batched by time stamp when the downstream stage is a channel configured for batched events. Extends `RelationSender` to support unbatched events.

7.3.1 Implement StreamSender

A class that is a source of stream events should send only events that are, from the Oracle Event Processing standpoint, inserted. Sending only inserted events models a stream, rather than a relation. Events sent from a stream source should also have non-decreasing time stamps from one event to the event that follows. The time stamp of an event that follows another should either be the same as, or later than, the event that preceded it.

When you implement `StreamSource`, your code can send events batched or unbatched. Your implementation of the `StreamSource` `setEventSender` method receives a sender instance that you can cast to one of the types described in Table 7-3. Use the sender instance in your code to send events as expected by the downstream stage.

If your code sends events to a channel that enables batching, use one of the batched event senders to batch events by time stamp before sending them. For more information, see Batch Processing Channels.

The sender instance provides a `sendHeartbeat` method to send a heartbeat when the receiving channel is configured to be application time stamped.

7.3.2 Implement RelationSender

A class that is a source of events as a relation can send insert, delete, and update events to the downstream stage. When you implement the `RelationSource` interface, your code can send events batched or unbatched. Your implementation of the `RelationSource` `setEventSender` method receives a sender instance that you can cast to one of the types described in Table 7-3. Use the sender instance to send events to the downstream stage.

Keep in mind the following constraints for handling the sender instance your class receives:

- For `sendDeleteEvent`, you must send an instance of the same event type as that configured for the channel.
- For `sendInsertEvent`, a unique constraint violation exception is raised and the input event discarded if an event with the same primary key is already in the relation.
- For `sendUpdateEvent`, an invalid update tuple exception will be raised and the input event will be discarded if an event with the given primary key is not in the relation.

In the following example, a simple `RelationSource` implementation receives a `StreamSender`, then casts the sender to a `RelationSender` to send events as a relation. This class creates a new `TradeEvent` instance from the event type configured in the repository, but the `sendEvents` method could as easily have received an instance as a parameter from another part of the code.

```
package com.oracle.cep.example.tradereport;

import com.bea.wlevs.ede.api.EventType;
import com.bea.wlevs.ede.api.EventTypeRepository;
import com.bea.wlevs.ede.api.RelationSender;
import com.bea.wlevs.ede.api.RelationSource;
import com.bea.wlevs.ede.api.StreamSender;
import com.bea.wlevs.util.Service;

public class TradeEventSource implements RelationSource {

    // Variables for event type respository and event sender. Both
    // will be set by the server.
    EventTypeRepository m_repos = null;
    RelationSender m_sender = null;

    // Called by the server to set the repository instance.
    @Service
    public void setEventTypeRepository(EventTypeRepository repos) {
        m_repos = repos;
    }

    // Called by the server to set the sender instance.
    @Override
    public void setEventSender(StreamSender sender) {
        // Cast the received StreamSender to a RelationSender
        m_sender = (RelationSender)sender;
    }

    /**
     * Sends events to the next EPN stage using the sender
     * received from the server. This code assumes that an event
     * instance isn't received from another part of the class,
     * instead creating a new instance from the repository.
     */
    private void sendEvents(){
        EventType eventType = m_repos.getEventType("TradeEvent");
        TradeEvent tradeEvent = (TradeEvent)eventType.createEvent();
        m_sender.sendDeleteEvent(tradeEvent);
    }
}
```

8

Cached Event Data

You can configure a caching system so that applications have ready access to event data. The caches in the system can be a combination of Oracle Coherence distributed caching, Oracle Event Processing local caching, and caching solutions provided by third parties. You can access the events in the caches with Oracle CQL and Java classes.

This chapter includes the following sections:

8.1 Caching Defined

A *cache* is a temporary storage area for event data. To increase the availability of event data and to increase application performance, you can create a cache so that applications can publish to or consume events from the cache. An application can also access the processed event data written to the cache by other applications.

You can configure any stage in an Oracle Event Processing application that generates events to publish its events to the cache. A cache does not have to be a stage in the EPN. Another component or Spring bean can access events in the cache programmatically with the caching APIs.

A *caching system* is a configured instance of a caching implementation. A caching system defines a named set of configured caches and the configuration for remote communication when any of the caches are distributed across multiple machines.

Oracle Event Processing caching enables an application to perform the following tasks. All of these tasks happen incrementally without halting the application or causing latency spikes.

- Pre-load a cache with event data before an application is deployed.

- Periodically refresh, invalidate, and flush the event data in a cache.
- Dynamically update a cache configuration.

8.1.1 Supported Caching Implementations

Oracle Event Processing supports the following caching implementations:

- **Oracle Event Processing local cache**: a local, in-memory single-JVM cache. This implementation is best for local use (it cannot be used in a cluster). It might also be useful for development in the early stages because it is relatively simple to set up.
- **Oracle Coherence:** a JCache-compliant in-memory distributed data grid solution for clustered applications and application servers. It coordinates updates to the data using cluster-wide concurrency control, replicates data modifications across the cluster using the highest performing clustered protocol available, and delivers notifications of data modifications to any servers that request them. You take advantage of Oracle Coherence features using the standard Java collections API to access and modify data, and use the standard JavaBean event model to receive data change notifications.

  Note:

  Before you can use Oracle Event Processing with Oracle Coherence, you must obtain a valid Oracle Coherence license such as a license for Coherence Enterprise Edition, Coherence Grid Edition, or Oracle WebLogic Application Grid.

  For more information on Oracle Coherence, see `http://docs.oracle.com/middleware/1213/coherence/index.html`.

- **Third-party caches**: you can create a plug-in to allow Oracle Event Processing to work with other, third-party cache implementations.

8.1.2 Use Cases

Caching technology is a great fit for streaming data use cases, where high throughput can be particularly important. Getting data from a cache is usually much faster than getting the same data from a relational database.

The following scenarios describe common use cases for caching in Oracle Event Processing applications.

- **Publish events to a cache**

  A financial application publishes events to a cache while the financial market is open, and then processes data in the cache after the market closes. Publishing events to a cache makes them available to the application or available to other Oracle Event Processing applications running in the server. Publishing events to a cache also allows for asynchronous writes to a secondary storage by the cache implementation.

- **Consume data from a cache**

  Oracle Event Processing applications sometimes need to access non-streaming data. By caching this data, you can increase the performance of the application. The standard components of an Oracle Event Processing application that are allowed

direct programming access to a cache are input- and output-adapters and business POJOs.

Additionally, applications can access a cache from Oracle CQL either by a user-defined function or directly from an Oracle CQL statement. In the case of a user-defined function, programmers use Spring to inject the cache resource into the implementation of the function. For more information, see Application and Resource Configuration.

Applications can also query a cache directly from Oracle CQL statements that run in a processor. In this case, the cache functions as another type of data source to a processor so that querying a cache is similar to querying a channel except that data is pulled from a cache.

An example of using Oracle CQL to query a cache is from a financial application that publishes orders and the trades used to execute the orders to a cache. At the end of the day when the markets close, the application queries the cache to find all the trades related to a particular order.

- **Update and delete data in a cache**

  An Oracle Event Processing application can update and delete data in a cache when required. For example, a financial application might need to update an order in the cache each time individual trades that fulfill the order are executed, or an order might need to be deleted if it has been cancelled. The components of an application that are allowed to consume data from a cache are also allowed to update it.

- **Use a cache in a multiserver domain**

  If you build an Oracle Event Processing application that uses a cache, and you plan to deploy that application in a multiserver domain, then you must use a caching system that supports a distributed cache. In this case, you must use either Oracle Coherence or a third-party caching system that supports a distributed cache.

  For more information, see:

  – *Administering Oracle Event Processing*

  – Configure an Oracle Coherence Caching System and Cache

  – Configure a Third-Party Caching System and Cache

8.2 Configure an Oracle Coherence Caching System and Cache

You can configure your application to use the Oracle Coherence caching system and cache. Use this caching system if you plan to deploy your application to a multiserver domain. When you configure with Oracle Coherence, only the first caching-system can be configured in a server. The Oracle Event Processing server ignores other caching systems that you have configured.

Note:

Before you can legally use Oracle Event Processing with Oracle Coherence, you must obtain a valid Coherence license such as a license for Coherence Enterprise Edition, Coherence Grid Edition, or Oracle WebLogic Application Grid.

For more information on Oracle Coherence, see http://docs.oracle.com/middleware/1213/coherence/index.html.

The following assembly and configuration file settings configure an Oracle Coherence caching system and cache for an Oracle CQL processor. The cache uses an event type to specify the key properties for locating table rows in the relational database. This caching system is advertised, which means other applications can access the data in its caches.

8.2.1 Assembly File

The assembly file settings configure the caching system and cache1. The `value-type` setting is the event type into which you want to load the database values. This cache is advertised.

```
<wlevs:cache id="cache1" value-type="TradeReport" advertise="true">
  <wlevs:caching-system ref="coherence-caching-system"/>
</wlevs:cache>
<wlevs:caching-system id="coherence-caching-system" provider="coherence"/>
```

Note:

When you change the `id` setting for a coherence cache in the EPN diagram, the `id` changes in the assembly file and in the coherence-cache- file. However, if you change the `id` setting in the assembly file source editor, the `id` changes in the assembly file only. In this case, you must manually change the `cache-name` setting in the `coherence-cache-` to match the id setting in the assembly file. You also have to change all references to that cache.

When the cache is advertised, a component in the EPN of an application in a separate bundle can reference the advertised cache. The following example shows how a processor in one bundle can use the `cache-source` element to reference a cache source in another bundle with a `cache-id` of `cacheprovider`:

```
<wlevs:processor id="myProcessor2">
  <wlevs:cache-source ref="cacheprovider:cache-id">
</wlevs:processor>
```

Note:

When you have Oracle Coherence caches in the EPN assembly files of one or more applications deployed to the same Oracle Event Processing server, never configure multiple instances of the same cache with a loader or a store.

You can inadvertently do this by employing multiple applications that each configure the same Oracle Coherence cache with a loader or store in their respective EPN assembly file. If you configure multiple instances of the same cache with a loader or a store, Oracle Event Processing throws an exception.

8.2.2 Configuration File

The `coherence-cache-config.xml` file is the basic Oracle Coherence configuration file and must conform to the Oracle Coherence DTDs, as is true for any Oracle Coherence application.

See the Oracle Coherence documentation for information about `coherence-cache-config.xml`: `http://docs.oracle.com/middleware/1213/coherence/index.html`.

An Oracle Event Processing Oracle Coherence factory must be declared when you use Spring to configure a loader or store for a cache. You specify the factory with the `cachestore-scheme` element and include a factory class that enables Oracle Coherence to call into Oracle Event Processing and retrieve a reference to the loader or store that is configured for the cache. The only difference between configuring a loader or store is that the `method-name` element has a value of `getLoader` when a loader is used and `getStore` when a store is being used. You pass the cache name to the factory as an input parameter.

```
<cache-config>
    <caching-scheme-mapping>
        <cache-mapping>
            <cache-name>myCoherenceCache</cache-name>
            <scheme-name>new-replicated</scheme-name>
        </cache-mapping>
        <cache-mapping>
            <cache-name>myLoaderCache</cache-name>
            <scheme-name>test-loader-scheme</scheme-name>
        </cache-mapping>
        <cache-mapping>
            <cache-name>myStoreCache</cache-name>
            <scheme-name>test-store-scheme</scheme-name>
        </cache-mapping>
        <cache-mapping>
            <cache-name>
                cache1
            </cache-name>
            <scheme-name>
                new-replicated
            </scheme-name>
        </cache-mapping>
    </caching-scheme-mapping>
    <caching-schemes>
        <replicated-scheme>
            <scheme-name>new-replicated</scheme-name>
            <service-name>ReplicatedCache</service-name>
            <backing-map-scheme>
                <class-scheme>
```

```
                    <scheme-ref>my-local-scheme</scheme-
ref>
                </class-scheme>
            </backing-map-scheme>
        </replicated-scheme>
        <class-scheme>
            <scheme-name>my-local-scheme</scheme-name>
            <class-name>com.tangosol.net.cache.LocalCache</class-name>
            <eviction-policy>LRU</eviction-policy>
            <high-units>100</high-units>
            <low-units>50</low-units>
        </class-scheme>
        <local-scheme>
            <scheme-name>test-loader-scheme</scheme-name>
            <eviction-policy>LRU</eviction-policy>
            <high-units>100</high-units>
            <low-units>50</low-units>


<!-- A cachestore-scheme element that gets a loader starts here -->
            <cachestore-scheme>
                <class-scheme>
     <class-factory-name>com.bea.wlevs.cache.coherence.configuration.SpringFactory
     </class-factory-name>
                    <method-name>getLoader</method-name>
                    <init-params>
                        <init-param>
                            <param-type>java.lang.String</param-type>
                            <param-value>myCoherenceCache</param-value>
                        </init-param>
                        <init-param>
                            <param-type>
                                java.lang.String
                            </param-type>
                            <param-value>
                                cache1
                            </param-value>
                        </init-param>
                    </init-params>
                </class-scheme>
            </cachestore-scheme>
     <!-- The cachestore-scheme element ends here -->
        </local-scheme>

        <local-scheme>
            <scheme-name>test-store-scheme</scheme-name>
            <eviction-policy>LRU</eviction-policy>
            <high-units>100</high-units>
            <low-units>50</low-units>


<!-- A cachestore-scheme element that gets a store starts here -->
            <cachestore-scheme>
                <class-scheme>
     <class-factory-name>com.bea.wlevs.cache.coherence.configuration.SpringFactory
     </class-factory-name>
                    <method-name>getStore</method-name>
                    <init-params>
                        <init-param>
                            <param-type>java.lang.String</param-type>
                            <param-value>myCoherenceCache</param-value>
```

```
                        </init-param>
                        <init-param>
                            <param-type>
                                java.lang.String
                            </param-type>
                            <param-value>
                                cache1
                            </param-value>
                        </init-param>
                    </init-params>
                </class-scheme>
            </cachestore-scheme>
     <!-- The cachestore-scheme element ends here -->
        </local-scheme>
    </caching-schemes>
</cache-config>
```

tangosol-coherence-override.xml File (optional)

The `tangosol-coherence-override.xml` file is a global per-server file. It contains what is referred to as the *operational configuration* in the Oracle Coherence documentation. This file contains global, server-wide configuration settings for Oracle Coherence caching. You create this file in an XML editor and put it in the Oracle Event Processing server `config` directory for the server you want to configure.

Note:

Do not include the `tangosol-coherence-override.xml` file when you use Oracle Coherence for clustering.

Add the following XML to the Oracle Coherence configuration file to reference the `tangosol-coherence-override.xml` file. Include the `cluster-name` element to prevent Oracle Coherence from attempting to join existing Oracle Coherence clusters when Oracle Event Processing starts up. This can cause problems and sometimes prevent Oracle Event Processing from starting.

```
...
<coherence xml-override="/tangosol-coherence-override.xml">
  <cluster-config>
    <member-identity>
      <cluster-name>com.bea.wlevs.example.provider</cluster-name>
    </member-identity>
...
</coherence>
```

For more information about Oracle Event Processing clusters, see *Administering Oracle Event Processing*.

8.2.3 Cache Loader Bean

The `com.oracle.cep.cacheloader` package provides the `CsvCacheLoader` class for loading CSV events into a Coherence cache. You use a cache loader with an inbound adapter by replacing the `sourceUrl` property.

The first assembly file CSV adapter configuration shows a CSV inbound adapter that loads a file with the sourceUrl property. The second assembly file CSV adapter entry shows a CSV inbound adapter that loads a cache loader bean.

Load Events in a CSV file

```
<wlevs:adapter id="StockTradeCSVInboundAdapter" provider="csv-inbound">
  <wlevs:listener ref="AdapterOutputChannel"/>
  <wlevs:instance-property name="eventType" value="TradeEvent"/>
  <wlevs:instance-property name="sourceUrl"
    value="file:/scratch/mpawlan/oep9-19/oep/utils/load-generator/StockData.csv"/>
</wlevs:adapter>
```

Load Events with a Cache Loader

```
 <wlevs:cache id="csvcache" key-properties="sequenceNo"
   value-type="TradeEvent" advertise="true">
   <wlevs:caching-system ref="cachesys" />
   </wlevs:cache>
   <bean id="csvloader" class="com.oracle.cep.cacheloader.CsvCacheLoader">
     <property name="cacheName" value="csvcache"/>
     <property name="sourceUrl"
       value="file:///scratch/juhe/view_storage/trade.csv"/>
   </bean>
```

8.3 Configure a Local Caching System and Cache

You can configure your application to use the Oracle Event Processing local caching system and cache. The Oracle Event Processing local caching system is appropriate when you do not plan to deploy your application to a multiserver domain. If you plan to deploy your application to a multiserver domain, use an Oracle Coherence cache.

This chapter describes some of the configuration settings. For complete information, see *Schema Reference for Oracle Event Processing*.

8.3.1 Assembly File

The following assembly file settings configure the local caching system and cache. The `value-type` setting is the event type into which you want to load the database values.

```
<wlevs:cache id="localcache" value-type="HelloWorldEvent">
  <wlevs:caching-system ref="caching-system"/>
    </wlevs:cache>
  <wlevs:caching-system id="caching-system" provider="wlevs" advertise="false"/>
```

8.3.2 Configuration File

The following configuration file settings specify a maximum size and eviction policy for the local caching system.The maximum size specifies the number of cache elements in memory after which the eviction policy occurs. The example also specifies the maximum amount of time in milliseconds that an entry is cached. Default `time-to-live` value is infinite. This example specifies 3600 milliseconds.

```
  <caching-system>
    <name>caching-system</name>
    <cache>
      <name>localcache</name>
      <max-size>64</max-size>
      <eviction-policy>LFU</eviction-policy>
      <time-to-live>3600</time-to-live>
    </cache>
  </caching-system>
```

The following configuration file settings add a `write-behind` element as a child element of cache. The `write-behind` element means Oracle Event Processing invokes the cache store from a separate thread after a create or update of a cache entry. The child elements of write-behind indicate the following:

- The number of updates that are picked up from the store buffer to write back to the backing store (batch-size). The default value is 100.
- The number of attempts that the user thread makes to write to the store buffer. The user thread is the thread that creates or updates a cache entry. If all attempts by the user thread to write to the store buffer fail, it will invoke the store synchronously (`batch-write-attempts`). The default value is 1.
- The time in milliseconds the user thread waits before aborting an attempt to write to the store buffer (`buffer-write-timeout`). The attempt to write to the store buffer fails only when the buffer is full. After the time out, further attempts can be made to write to the buffer based on the value of `buffer-write-attempts`. The default value is 100.

```
<caching-system>
    <name>caching-system-id</name>
    <cache>
        <name>cache-id</name>
        <max-size>100000</max-size>
        <eviction-policy>LRU</eviction-policy
        <time-to-live>3600</time-to-live>
        <write-behind>
            <buffer-size>200</buffer-size>
            <buffer-write-attempts>2</buffer-write-attempts>
            <buffer-write-timeout>200</buffer-write-timeout>
        </write-behind>
    </cache>
</caching-system>
```

The following configuration file settings add a `listeners` child element to configure the behavior of components that listen to the cache. The `listener` element has an `asynchronous` attribute that you can set to either `true` (listeners are invoked asynchronously) or `false` (listeners are invoked synchronously).

The `work-manager-name` child element specifies the work manager to use to asynchronously invoke listeners. This value is ignored if synchronous invocations are enabled. If a work manager is specified for the cache, this value overrides that setting for invoking listeners only. The value of the `work-manager-name` element corresponds to the `name` element of the `work-manager` setting in the Oracle Event Processing `config.xml` server configuration file.

```
<caching-system>
     <name>caching-system-id</name>
     <cache>
        <name>cache-id</name>
        <max-size>100000</max-size>
        <eviction-policy>LRU</eviction-policy
        <time-to-live>3600</time-to-live>
        <write-behind>
            <buffer-size>200</buffer-size>
            <buffer-write-attempts>2</buffer-write-attempts>
            <buffer-write-timeout>200</buffer-write-timeout>
        </write-behind>
        <listeners asynchronous="true">
            <work-manager-name>cachingWM</work-manager-name>
```

```
        </listeners>
      </cache>
</caching-system>
```

8.4 Configure a Cache as an Event Listener

You can configure a cache to receive events as they pass through the network. For example, to specify that a cache listens to a channel, configure the channel with a `wlevs:listener` element that has a reference to the cache.

In the following example, as the channel sends new events to the cache, the events are inserted into the cache. If the channel sends a *remove event* (an old event that exits the output window), then the event is removed from the cache.

```
<wlevs:caching-system id="caching-system-id"/>

<wlevs:cache id="cache-id" name="alternative-cache-name">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>

<wlevs:channel id="tradeStream">
    <wlevs:listener ref="cache-id"/>
</wlevs:channel>
```

8.5 Index a Cache with a Key

The following sections describe the options available to you to specify the key that is used to index the cache. When you do not explicitly specify a key, the event object serves as both the key and value when the event is inserted into the cache. In this case, the event class must include a valid implementation of the `equals` and `hashcode` methods that take into account the values of the key properties.

8.5.1 Assembly File

Specify a property name for the key property in the assembly file with the `key-properties` attribute, as shown in the following example:

```
<wlevs:cache id="myCache" key-properties="key-property-name">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
```

In this case, all events that are inserted into the cache are required to have a property of this name at runtime, otherwise Oracle Event Processing throws an exception. For example, assume the event type being inserted into the cache looks something like the following; note the `key` property (only relevant Java source shown):

```
public class MyEvent {
  private String key;
  public MyEvent() {}
  public MyEvent(String key) { this.key = key; }
  public String getKey() { return key;}
  public void setKey(String key) { this.key = key;}
}
```

The corresponding declaration in the assembly file looks like the following:

```
<wlevs:cache id="myCache" key-properties="key">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
```

8.5.2 Metadata Annotation

you can use the metadata annotation `com.bea.wlevs.ede.api.Key` to annotate the event property in the Java class that implements the event type. This annotation does not have any attributes.

To use a metadata annotation to specify a key:

1. Import the `com.bea.wlevs.ede.api.Key` package.
2. Apply the `@Key` annotation to a method.

   The following example shows how to specify that the `key` property of the `MyEvent` event type is the key; only relevant code is shown:

```
import com.bea.wlevs.ede.api.Key;
public class MyEvent {
  private String key;
  public MyEvent() {}
  public MyEvent(String key) { this.key = key; }
  public String getKey() { return key; }
  @Key
  public void setKey(String key) { this.key = key; }
}
```

8.5.3 Composite Key

You can use the `key-class` attribute of the `wlevs:cache` element to specify a composite key in which multiple properties form the key. The value of the `key-class` attribute must be a `JavaBean` class with public fields that match the fields of the event class. The `JavaBean` class must override the `equals` and `hashCode` methods from the `java.lang.Object class`. The matching is done according to the field name. For example:

```
<wlevs:cache id="myCache" key-class="key-class-name">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
```

For a cache with a composite key composed of key-field1 and key-field2, you can execute both of the following queries:

```
SELECT stream.field2, cache.key-field1 from stream[NOW], cache WHERE
stream.field2=cache.key-field1 AND stream.field2=cache.key-field2

SELECT stream.field1, cache.key-field1 from stream[NOW], cache WHERE
stream.field1=cache.key-field1
```

8.6 Configure a Cache as an Event Source

You can configure a cache as an event source. To use a cache as an event source, you need to implement the `com.bea.wlevs.ede.api.StreamSink` interface. The configuration follows:

```
<wlevs:cache id="cache-id" name="alternative-cache-name"
   caching-system="caching-system-id">
   <wlevs:listener ref="cache-listener-id" />
</wlevs:cache>
```

8.7 Configure a Cache with a Cache Listener

You can configure a cache as a source of events to which another component in the event processing network listens. The listening component can be an adapter or a bean. A class that listens to a cache must implement an interface that provides methods for receiving events, as follows:

- A class that listens to a Coherence cache must implement the `com.tangosol.util.MapListener` interface.
- A class that listens to an Oracle Event Processing local cache must implement the `com.bea.cache.jcache.CacheListener` interface.

```
<wlevs:caching-system id="caching-system-id"/>
...
<wlevs:cache id="cache-id" name="alternative-cache-name">
    <wlevs:caching-system ref="caching-system-id"/>
    <wlevs:cache-listener ref="cache-listener-id" />
</wlevs:cache>
...
<bean id="cacheListenerId" class="com.bea.wlevs.example.provider.coherence"/>
```

In the example, the `cacheListenerId` Spring bean listens to events coming from the cache. In this case, the user-defined class that implements this component, `com.bea.wlevs.example.MyCacheListener`, is listening to an Oracle Coherence cache. It must implement the appropriate Oracle Coherence-specific Java interfaces, including `com.tangosol.util.MapListener`. The following example illustrates this implementation.

```
package com.bea.wlevs.example.provider.coherence;

import com.tangosol.util.MapEvent;
import com.tangosol.util.MapListener;

public class LocalListener implements MapListener {
    public static int deleted = 0;
    public static int inserted = 0;
    public static int updated = 0;

    public void entryDeleted(MapEvent event) { deleted++; }
    public void entryInserted(MapEvent event) { inserted++; }
    public void entryUpdated(MapEvent event) { updated++; }
}
```

8.8 Configure a Third-Party Caching System and Cache

You can configure your application to use a third-party caching system and cache.

Configure a Third-Party Caching System and Cache

1. Create a plug-in to define the third-party caching system as an Oracle Event Processing caching system provider:
   - Implement the `com.bea.wlevs.cache.spi.CachingSystem` interface
   - Create a factory that creates caching systems of this type.
   - Register the factory with an attribute that identifies its provider type.
2. Declare the caching system in the EPN assembly file.

Use the `wlevs:caching-system` element to declare a third-party implementation; use the `class` or `provider` attribute to specify additional information.

For simplicity, you can include the third-party implementation code inside the Oracle Event Processing application bundle itself to avoid having to import or export packages and manage the life cycle of a separate bundle that contains the third-party implementation. In this case the `wlevs:caching-system` element appears in the EPN assembly file as shown in the following example:

```
<wlevs:caching-system id="caching-system-id"
                      class="third-party-implementation-class"/>
```

The `class` attribute specifies a Java class that must implement the `com.bea.wlevs.cache.spi.CachingSystem` interface. For details about this interface, see the *Java API Reference for Oracle Event Processing*.

Sometimes you might not be able to or want to include the third-party caching implementation in the same bundle as the Oracle Event Processing application that is using it. In this case, you must create a separate bundle with a Spring application context that includes the `wlevs:caching-system` element, with the mandatory `advertise` attribute:

```
<wlevs:caching-system id ="caching-system-id"
       class="third-party-implementation-class" advertise="true"/>
```

Alternately, if you want to decouple the implementation bundle from the bundle that references it, or you are plugging in a caching implementation that supports multiple caching systems per Java process, you can specify a factory as a provider:

```
<wlevs:caching-system id ="caching-system-id" provider="caching-provider"/>
<factory id="factory-id" provider-name="caching-provider">
    <class>the.factory.class.name</class>
</factory>
```

The factory class (`the.factory.class.name`) must implement the `com.bea.wlevs.cache.spi.CachingSystemFactory` interface. This interface has the `create` method that returns a `com.bea.wlevs.cache.spi.CachingSystem` instance.

You must deploy this bundle with the application bundle so that the application bundle can start using it.

3. Add one or more caches for this caching system in the EPN assembly file.

```
<wlevs:caching-system id ="caching-system-id" provider="caching-provider"/>
...
<wlevs:cache id="cache-id" name="alternative-cache-name">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
```

Specify the optional `name` attribute only when the name of the cache in the caching system is different from its ID. The `wlevs:caching-system` child element references the already-declared caching system that contains the cache. You must specify this child element only when there is more than one caching system declared (either implicitly or explicitly) or when the caching system is in a different application or bundle.

You can export both the caching system and the cache as an OSGI service with the `advertise` attribute.

```
<wlevs:caching-system id="caching-system-id" advertise="true"/>
...
```

```
<wlevs:cache id="cache-id" name="alternative-cache-name" advertise="true" >
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
```

If the cache is advertised, then a component in the EPN of an application in a separate bundle can reference it. The following example shows how a processor in one bundle can use as a cache source the cache with ID `cache-id` located in a separate bundle (called `cacheprovider`):

```
<wlevs:processor id="myProcessor2">
    <wlevs:cache-source ref="cacheprovider:cache-id"/>
</wlevs:processor>
```

The caching system creates the cache associated with a particular name and returns a reference to the cache. The resulting cache bean implements the `java.util.Map` interface.

4. Configure the third-party caching system and its caches by updating the third-party caching configuration file or files for the application.

   Refer to your third-party cache documentation.

5. Optionally, override the default third-party cache configuration by updating the appropriate configuration file with one or more additional `cache` element child elements. Refer to your third-party cache documentation.

   - Specify that a cache is an event sink by configuring it as a listener to another component in the event processing network.
   - Specify that a cache is an event source to which another component in the event processing network listens.
   - Configure a cache loader or store.

6. When you assemble your application, verify that the `META-INF/MANIFEST.MF` file includes the following import:

   ```
   com.bea.wlevs.cache.spi; version ="<version>"
   ```

   If the `MANIFEST.MF` files does not include this import, update the `MANIFEST.MF` file to add this import before deploying your application.

8.9 Exchange Data Between a Cache and Another Data Source

You can have a cache in an EPN exchange data with another data source, including a database. For example, you can load a cache with data when the application starts or create a read/write relationship between the cache and a database.

If the cache will only be reading data, including when the backing store is read-only, you should use a cache loader. If the cache will read and write data, use a cache store. In both cases, creating the relationship involves specific configuration and a Java class that knows how to communicate with the data source.

8.9.1 Load Cache Data from a Read-Only Data Source

Using a cache loader, you can have a cache in your EPN load data from a read-only data source. A cache loader is a Java class that loads cache objects into a cache. You create a cache loader by writing a Java class that implements the appropriate interfaces to enable the loader class to communicate with the cache. Then you configure a cache

loader by using the `wlevs:cache-loader` child element of the `wlevs:cache` element to specify the bean that does the loading work.

If the backing store is read-write, use a cache store instead (see Exchange Data with a Read-Write Data Source).

When creating a cache loader, you implement interfaces as follows:

- To load cache data into an Oracle Coherence cache, create a class that implements the appropriate Oracle Coherence-specific Java interfaces, including `com.tangosol.net.cache.CacheLoader`. See Example 8-2 for an example.
- To load cache data into an Oracle Event Processing local cache, create a class that implements `com.bea.cache.jcache.CacheLoader` interface. This interface includes the `load` method to customize loading a single object into the cache; Oracle Event Processing calls this method when the requested object is not in the cache. The interface also includes `loadAll` methods that you implement to customize the loading of the entire cache.

In Example 8-1, the `localLoader` bean loads events into an Oracle Coherence cache when the backing store is read-only.

When working with a Coherence cache, note that if you specify a cache loader in your configuration file, you must also specify the corresponding class factory method name in your Coherence cache configuration file. For a cache loader, you specify the `getLoader` method of `com.bea.wlevs.cache.coherence.configuration.SpringFactory`. For example code, see Configure an Oracle Coherence Caching System and Cache.

Example 8-1 Oracle Coherence Cache EPN Assembly File for a Cache Loader

```
<wlevs:caching-system id="caching-system-id"/>
<wlevs:cache id="myCache" advertise="false">
    <wlevs:caching-system ref="caching-system-id"/>
    <wlevs:cache-loader ref="localLoader"/>
</wlevs:cache>
<bean id="localLoader"
      class="com.bea.wlevs.example.provider.coherence.LocalLoader"/>
```

Example 8-2 Oracle Coherence Cache LocalLoader Implementation

```
package com.bea.wlevs.example.provider.coherence;

import java.util.Collection;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import com.bea.wlevs.example.provider.event.ProviderData;
import com.tangosol.net.cache.CacheLoader;

public class LocalLoader implements CacheLoader {
    public static int loadCount = 0;
    public static Set keys = new HashSet();

    public LocalLoader() {
    }
    public Object load(Object key) {
        loadCount++;
        keys.add(key);
        return new ProviderData((String) key);
    }
    public Map loadAll(Collection keys) {
        Map result = new HashMap();
```

```
        for (Object key : keys) {
            result.put(key, load(key));
        }
        return result;
    }
}
```

8.9.2 Exchange Data with a Read-Write Data Source

Using a cache store, you can have a cache in your EPN exchange data with a read-write data source. A cache store is a Java class that exchanges cache objects with a cache. You create a cache store by writing a Java class that implements the appropriate interfaces to enable it to communicate with the data source. Then you add the cache store to the EPN by using the `wlevs:cache-store` child element of the `wlevs:cache` element to specify the bean that communicates with the data source.

If the backing store is read-only, use a cache loader instead (see Load Cache Data from a Read-Only Data Source).

When creating a cache store, you implement interfaces as follows:

- To exchange cache data with an Oracle Coherence cache, create a class that implements the appropriate Oracle Coherence-specific Java interfaces, including `com.tangosol.net.cache.CacheStore`. See Example 8-4for an example.
- To exchange cache data with an Oracle Event Processing local cache, create a class that implements the `com.bea.cache.jcache.CacheStore` interface. This interface includes the `store` method that stores the data in the backing store using the passed key; Oracle Event Processing calls this method when it inserts data into the cache. The interface also includes the `storeAll` method for storing a batch of data to a backing store in the case that you have configured asynchronous writes for a cache with the `write-behind` configuration element.

In Example 8-3, the `localStore` bean loads events into the cache when the backing store is read-write.

Note that if you specify a cache store in your Spring configuration file, you must also specify the corresponding class factory method name in your Coherence cache configuration file. For a cache store, you specify the `getStore` method of `com.bea.wlevs.cache.coherence.configuration.SpringFactory`. For example code, see Configure an Oracle Coherence Caching System and Cache.

Example 8-3 Oracle Coherence Cache EPN Assembly File for a Cache Store

```
<wlevs:caching-system id="caching-system-id"/>
<wlevs:cache id="myCache" advertise="false">
    <wlevs:caching-system ref="caching-system-id"/>
    <wlevs:cache-store ref="localStore"/>
</wlevs:cache>
<bean id="localStore"
      class="com.bea.wlevs.example.provider.coherence.LocalStore"/>
```

Example 8-4 Oracle Coherence Cache LocalStore Implementation

```
package com.bea.wlevs.example.provider.coherence;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

import com.bea.wlevs.example.provider.event.ProviderData;
import com.tangosol.net.cache.CacheStore;
```

```
public class LocalStore implements CacheStore {
    public static int eraseCount = 0;
    public static int storeCount = 0;
    public static int loadCount = 0;

    public void erase(Object key) {
        eraseCount++;
    }
    public void eraseAll(Collection keys) {
        for (Object key : keys) {
            erase(key);
        }
    }
    public void store(Object key, Object value) {
        //
        // Do the store operation here.
        //
    }
    public void storeAll(Map entries) {
        for (Map.Entry entry : (Set <Map.Entry>)entries.entrySet()) {
            store(entry.getKey(), entry.getValue());
        }
    }
    public Object load(Object key) {
        loadCount++;
        return new ProviderData((String) key);
    }
    public Map loadAll(Collection keys) {
        Map result = new HashMap();
        for (Object key : keys) {
            result.put(key, load(key));
        }
        return result;
    }
}
```

8.10 Access a Cache from Application Code

Once you have configured a cache, you can access the cache from several components in an Oracle Event Processing application. This section describes how to do that.

For more information, see the following sections:

- Access a Cache from an Oracle CQL Statement
- Access a Cache from an Adapter
- Access a Cache From a Business POJO
- Access a Cache From an Oracle CQL User-Defined Function
- Access a Cache with JMX.

Before you assemble and deploy the application, edit your `META-INF/MANIFEST.MF` file to import packages that are required in your implementation. For example, if your application implements cache listeners, loaders or stores, your manifest should import `com.tangosol.net.cache` packages, which contain the Coherence APIs.

Oracle Event Processing provides caching APIs that you can use in your application to perform certain tasks. The APIs are in the `com.bea.cache.jcache` package, which includes the APIs used to access a cache and create cache loader, listeners, and stores. If you want to use the loader, listener, and store functionality, then import the `com.tangosol.net` and `com.tangosol.net.cache` packages.

You create, configure, and wire caching systems and caches with the EPN assembly file and component configuration files. This means that you typically never explicitly use the `Cache` and `CachingSystem` interfaces in your application. The only reason to use them is when you have additional requirements over the standard configuration. For example, if you want to provide integration with a third-party cache provider, then you must use the `CachingSystem` interface. If you want to perform operations on a cache that are not part of the `java.util.Map` interface, then you can use the `Cache` interface.

If you create cache listeners, loaders, or stores for an Oracle Event Processing local cache, then the beans you write must implement the `CacheListener`, `CacheLoader`, or `CacheStore` interfaces.

If you create cache listeners, loaders, or stores for an Oracle Coherence cache, then the beans you write must implement the appropriate Oracle Coherence interfaces.

If you create cache listeners, loaders, or stores for a third-party cache, then the beans you write must implement the appropriate third-party cache interfaces.

8.10.1 Access a Cache from an Oracle CQL Statement

You can reference a cache from an Oracle CQL statement in much the same way you reference an event source such as a channel; this feature enables you to enrich standard streaming data with data from a separate source. The code in the following example shows a valid Oracle CQL query that joins trade events from a standard channel named `S1` with stock symbol data from a cache named `stockCache`.

You must abide by these restrictions when using a cache in an Oracle CQL query:

- Whenever you query a cache, you must join against the `[Now]` window.

  This guarantees that the query will execute against a snapshot of the cache. If you join against any other window type, then if the cache changes before the window expires, the query will be incorrect.

  The following example shows an invalid Oracle CQL query that joins a `Range` window against a cache. If the cache changes before this window expires, the query will be incorrect. Consequently, this query will raise Oracle Event Processing server error "`external relation must be joined with s[now]`."

  ```
  SELECT trade.symbol, trade.price, trade.numberOfShares, company.name
  FROM TradeStream [Range 8 hours] as trade, CompanyCache as company
  WHERE trade.symbol = company.id
  ```

  When you use data from a cache in an Oracle CQL query, Oracle Event Processing *pulls* the data rather than it being *pushed*, as is the case with a channel. This means that, continuing with the query executes only when a channel pushes a `trade` event to the query; the stock symbol data in the cache never causes a query to execute, it is only pulled by the query when needed.

- You must specify the key property needed to do a lookup based on the cache key.

  Consider two streams `S` and `C` with schemas (`id`, `group`, `value`) where the cache key is `id`. A valid query is:

  ```
  select count(*) as n from S [now], C
  where S.id = C.id
  ```

- Joins must be executed only by referencing the cache key.

- You cannot use a cache in a view. Instead, use a join.

- Only a single channel source may occur in the `FROM` clause of an Oracle CQL statement that joins cache data source(s).
- If the cache is a processor source, you connect the cache directly to the channel on the EPN.
- If the cache is a processor sink, it can be connected directly to a processor.

Access a Cache from an Oracle CQL Statement

This procedure assumes that you have already configured the caching system and caches. For more information, see:

- Configure a Local Caching System and Cache
- Configure an Oracle Coherence Caching System and Cache
- Configure a Third-Party Caching System and Cache

1. If you have not already done so, create the event type that corresponds to the cache data and register it in the event repository.

   See Events and Event Types.

2. Specify the key properties for the data in the cache.

3. In the EPN assembly file, update the configuration of the cache to declare the event type of its values; use the `value-type` attribute of the `wlevs:cache` element. For example:

   ```
   <wlevs:caching-system id="caching-system-id"/>
   ...
   <wlevs:cache id="cache-id"
                name="alternative-cache-name"
                value-type="CompanyEvent">
       <wlevs:caching-system ref="caching-system-id"/>
   </wlevs:cache>
   ```

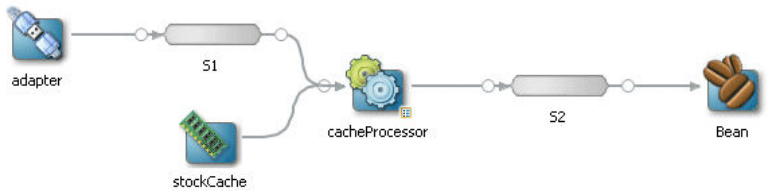
   The value-type attribute specifies the type for the values contained in the cache. This must be a valid type name in the event type repository.

   This attribute is required only if the cache is referenced in an Oracle CQL query. This is because the query processor needs to know the type of events in the cache.

4. In the EPN assembly file, update the configuration of the processor that executes the Oracle CQL query that references a cache:

   a. If the cache is a processor source: you connect the cache directly to the processor on the EPN as Figure 8-1 shows.

   Figure 8-1 Cache as Processor Source

   Update the `wlevs:processor` element a `wlevs:cache-source` child element that references the cache. For example:

```
<wlevs:channel id="S1"/>

<wlevs:processor id="cacheProcessor">
    <wlevs:source ref="S1">
    <wlevs:cache-source ref="cache-id">
</wlevs:processor>
```

In the example, the processor will have data pushed to it from the `S1` channel as usual; however, the Oracle CQL queries that execute in the processor can also pull data from the `cache-id` cache. When the query processor matches an event type in the `FROM` clause to an event type supplied by a cache, such as `CompanyEvent`, the processor pulls instances of that event type from the cache.

b. If the cache is a processor sink: you must connect the processor to the cache using a channel on the EPN (that is, there must be a channel between the processor and the cache sink) as Figure 8-2 shows.

Figure 8-2 Cache as Processor Sink

In this case, the application assembly file looks like this:

```
<wlevs:channel id="channel1" event-type="StockTick">
    <wlevs:listener ref="processor" />
</wlevs:channel>
<wlevs:processor id="processor">
    <wlevs:listener ref="channel2" />
</wlevs:processor>
<wlevs:channel id="channel2" event-type="StockTick">
    <wlevs:listener ref="cache-id" />
</wlevs:channel>
```

```
SELECT  S1.symbol, S1.lastPrice, stockCache.description
FROM    S1 [Now], stockCache
WHERE   S1.symbol = stockCache.symbol
```

8.10.2 Access a Cache from an Adapter

An adapter can also be injected with a cache using the standard Spring mechanism for referencing another bean. A cache bean implements the `java.util.Map` interface which is what the adapter uses to access the injected cache.

First, the configuration of the adapter in the EPN assembly file must be updated with a `wlevs:instance-property` child element, as shown in the following example:

```
<wlevs:caching-system id="caching-system-id"/>
    ...
<wlevs:cache id="cache-id" name="alternative-cache-name">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
...
<wlevs:adapter id="myAdapter" provider="myProvider">
    <wlevs:instance-property name="map" ref="cache-id"/>
</wlevs:adapter>
```

In the example, the `ref` attribute of `wlevs:instance-property` references the `id` value of the `wlevs:cache` element. Oracle Event Processing automatically injects the cache, implemented as a `java.util.Map`, into the adapter.

In the adapter Java source, add a `setMap (Map)` method with the code that implements whatever you want the adapter to do with the cache:

```
package com.bea.wlevs.example;
…
import java.util.Map;
public class MyAdapter implements Runnable, Adapter, EventSource, SuspendableBean  {
...
    public void setMap (Map map) {...}
}
```

8.10.3 Access a Cache From a Business POJO

A business POJO, configured as a standard Spring bean in the EPN assembly file, can be injected with a cache using the standard Spring mechanism for referencing another bean. In this way the POJO can view and manipulate the cache. A cache bean implements the `java.util.Map` interface which is what the business POJO uses to access the injected cache. A cache bean can also implement a vendor-specific sub-interface of `java.util.Map`, but for portability it is recommended that you implement `Map`.

First, the configuration of the business POJO in the EPN assembly file must be updated with a `property` child element, as shown in the following example based on the Output bean of the FX example (see *Getting Started with Oracle Event Processing*):

```
<wlevs:caching-system id="caching-system-id"/>
...
<wlevs:cache id="cache-id" name="alternative-cache-name">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
...
<bean class="com.bea.wlevs.example.helloworld.HelloWorldBean">
    <property name="map" ref="cache-id"/>
</bean>
```

In the example, the `ref` attribute of the `property` element references the `id` value of the `wlevs:cache` element. Oracle Event Processing automatically injects the cache, implemented as a `java.util.Map`, into the business POJO bean.

In the business POJO bean Java source, add a `setMap (Map)` method with the code that implements whatever you want the POJO to do with the cache:

```
package com.bea.wlevs.example.helloworld;
…
import java.util.Map;
public class HelloWorldBean implements EventSink {
...
    public void setMap (Map map) {...}
}
```

8.10.4 Access a Cache From an Oracle CQL User-Defined Function

In addition to standard event streams, Oracle CQL rules can also invoke the member methods of a user-defined function.

These user-defined functions are implemented as standard Java classes and are declared in the component configuration file of the Oracle CQL processor, as shown in the following example:

```
<bean id="orderFunction" class="orderFunction-impl-class"/>
```

The processor in which the relevant Oracle CQL rule runs must then be injected with the user-defined function using the `wlevs:function` child element, referencing the Spring bean with the `ref` attribute:

```
<wlevs:processor id= "tradeProcessor">
    <wlevs:function ref="orderFunction"/>
</wlevs:processor>
```

Alternatively, you can specify the bean class in the `wlevs:function` element:

```
<wlevs:processor id="testProcessor">
    <wlevs:listener ref="providerCache"/>
    <wlevs:listener ref="outputCache"/>
    <wlevs:cache-source ref="testCache"/>
    <wlevs:function function-name="mymod" exec-method="execute" />
        <bean class="com.bea.wlevs.example.function.MyMod"/>
    </wlevs:function>
</wlevs:processor>
```

The following Oracle CQL rule, assumed to be configured for the `tradeProcessor` processor, shows how to invoke the `existsOrder` method of the `orderFunction` user-defined function:

```
INSERT INTO InstitutionalOrder
SELECT er.orderKey AS key, er.symbol AS symbol, er.shares as cumulativeShares
FROM ExecutionRequest er [Range 8 hours]
WHERE NOT orderFunction.existsOrder(er.orderKey)
```

You can also configure the user-defined function to access a cache by injecting the function with a cache using the standard Spring mechanism for referencing another bean. A cache bean implements the `java.util.Map` interface which is what the user-defined function uses to access the injected cache.

First, the configuration of the user-defined function in the EPN assembly file must be updated with a `wlevs:property` child element, as shown in the following example:

```
<wlevs:caching-system id="caching-system-id"/>
    ...
<wlevs:cache id="cache-id" name="alternative-cache-name">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
    ...
<bean id="orderFunction" class="orderFunction-impl-class">
    <wlevs:property name="cache" ref="cache-id"/>
</bean>
```

In the example, the `ref` attribute of the `wlevs:property` element references the `id` value of the `wlevs:cache` element. Oracle Event Processing automatically injects the cache, implemented as a `java.util.Map`, into the user-defined function.

In the user-defined function's Java source, add a `setMap (Map)` method with the code that implements whatever you want the function to do with the cache:

```
package com.bea.wlevs.example;
…
import java.util.Map;
public class OrderFunction {
...
    public void setMap (Map map) {...}
}
```

For more information on user-defined functions, see *Oracle Fusion Middleware Oracle CQL Language Reference for Oracle Event Processing*.

8.10.5 Access a Cache with JMX

At runtime, you can access a cache programmatically using JMX and the MBeans that Oracle Event Processing deploys for the caching systems and caches you define. For more information, see *Administering Oracle Event Processing*.

8.10.5.1 How to Access a Cache With JMX Using Oracle Event Processing Visualizer

The simplest and least error-prone way to access a caching system or cache with JMX is to use the Oracle Event Processing Visualizer. For more information, see *Using Visualizer for Oracle Event Processing*.

8.10.5.2 How to Access a Cache With JMX Using Java

The simplest and least error-prone way to access a caching system or cache with JMX is to use the Oracle Event Processing Visualizer (see How to Access a Cache With JMX Using Oracle Event Processing Visualizer). Alternatively, you can access a caching system or cache with JMX using Java code that you write.

Oracle Event Processing creates a `StageMBean` for each cache that your application uses as a stage. The `Type` of this MBean is `Stage`.

To access a cache with JMX using Java:

1. Connect to the JMX service that Oracle Event Processing server provides.

   For more information, see *Administering Oracle Event Processing*

2. Get a list of cache `StageMbean` using either of:

   - `CachingSystemMBean.getCacheMBeans()`
   - `ApplicationMBean.getStageMBeans()`

3. Get the `ObjectName` for a given `StageMBean` that represents a cache in your caching system:

```
ObjectName cacheName = ObjectName.getInstance (
    'com.bea.wlevs:Name =
newCache,Type=Stage,CachingSystem=newCachingSystem,Application=provider'
);
```

4. Get a proxy instance for the `StageMBean` with this `ObjectName`:

```
StageMBean cache = (StageMBean) MBeanServerInvocationHandler.newProxyInstance(
    server, cacheName, StageMBean.class, false
);
```

5. Use the methods of the `StageMBean` to access the cache.

9

EclipseLink, JPA, and Oracle Coherence

The Oracle Event Processing installation includes the EclipseLink 2.4.2 open source mapping and persistence framework to support the use of the Java Persistence API (JPA) in your applications. JPA is the standard for object-relational mapping (ORM) and enterprise Java persistence.

This chapter presents two sample Oracle Event Processing applications, `HelloWorld` and `JPA-Coherence-Sample-Code`, that use EclipseLink and JPA to read from and write to a database. The JPA-Coherence-Sample-Code also uses a coherence cache for coordinated data updates in an environment with clustered applications and servers.

This chapter includes the following sections:

9.1 High-Level Procedure

Use the following high-level steps to create an Oracle Event Processing application that includes EclipseLink:

1. Create your Oracle Event Processing application including JPA and Oracle Coherence as needed.
2. Create a `persistence.xml` file with the correct JPA configuration. This file contains the properties that control runtime operation.
3. Put the `persistence.xml` file in the `META-INF` directory of your application.
4. Bundle and deploy the application.

Learn more about EclipseLink at `http://eclipse.org/eclipselink/`.

Note: Coherence socket exception occurs when you run CQL sample on AIX 6.1 Japanese platform. To resolve this issue:

Add the `-Djava.net.preferIPv4Stack=true` parameter to the last line of the `startwlevs.sh` script.

Note: Spatial sample is not supported on AIX Platform.

9.2 HelloWorld Example

The HelloWorld example uses EclipseLink to establish a read and write JDBC connection to the data source to access and store HelloWorld events. In this example, HelloWorld events contain date and time information.

The example is comprised of the following files, which are discussed in this section:

- persistence.xml Configuration File
- HelloWorldAdapter.java
- HelloWorldEvent.java
- HelloWorldBean.java

9.2.1 persistence.xml Configuration File

The following `persistence.xml` file has one persistence unit (`persistence-unit`) called `helloworld`. The helloworld persistence unit has a `transaction-type` of `RESOURCE_LOCAL` because Oracle Event Processing is a Java SE environment. The EclipseLink properties specify the settings for database read and write operations and logging. For this example, the managed persistable class that represents objects in the database is `com.bea.wlevs.event.example.helloworld.HelloWorldEvent`.

This `persistence.xml` file has entries for JPA logging that are commented out and set to `false`. You can uncomment these settings and set them to `true` to debug or otherwise monitor the application behavior. For information on property settings, see `http://eclipse.org/eclipselink/documentation/2.4/jpa/extensions/toc.htm`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/persistence         http://
java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
     version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="helloworld" transaction-type="RESOURCE_LOCAL">
    <class>com.bea.wlevs.event.example.helloworld.HelloWorldEvent</class>
      <properties>
        <property name="eclipselink.jdbc.read-connections.min" value="1"/>
        <property name="eclipselink.jdbc.write-connections.min" value="1"/>
      <!--
        <property name="eclipselink.logging.timestamp" value="false"/>
        <property name="eclipselink.logging.thread" value="false"/>
        <property name="eclipselink.logging.session" value="false"/>
        <property name="eclipselink.logging.exceptions" value="false"/>
        <property name="eclipselink.logging.connection" value="false"/>
        <property name="eclipselink.logging.level" value="FINER"/>
      -->
      </properties>
  </persistence-unit>
</persistence>
```

9.2.2 HelloWorldAdapter.java

The `HelloWorldAdapter.java` class is a custom threaded adapter that continuously creates events of type `HelloWorldEvent`. The application constructs

message text of type `DateFormat`, which is used by the `generateHelloMessage` method to create events of type `HelloWorldEvent`.

The Oracle Event Processing framework calls the `setEventSender` method to initialize the `eventSender` private variable with a `StreamSender` instance. The `StreamSender` instance sends events emitted by a `StreamSource` instance to a `StreamSink` listener. In this example the `StreamSink` listener is the `HellowWorldBean` instance.

```
package com.bea.wlevs.adapter.example.helloworld;

import java.text.DateFormat;
import java.util.Date;
import com.bea.wlevs.ede.api.RunnableBean;
import com.bea.wlevs.ede.api.StreamSender;
import com.bea.wlevs.ede.api.StreamSource;
import com.bea.wlevs.event.example.helloworld.HelloWorldEvent;

public class HelloWorldAdapter implements RunnableBean, StreamSource {
    private static final int SLEEP_MILLIS = 300;
    private DateFormat dateFormat;
    private String message;
    private boolean suspended;
    private StreamSender eventSender;

    public HelloWorldAdapter() {
        super();
        dateFormat = DateFormat.getTimeInstance();
    }
    public void run() {
        suspended = false;
        while (!isSuspended()) { // Generate messages forever...
            generateHelloMessage();
            try {
                synchronized (this) {
                    wait(SLEEP_MILLIS);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public void setMessage(String message) {
        this.message = message;
    }
    private void generateHelloMessage() {
        String message = this.message + dateFormat.format(new Date());
        HelloWorldEvent event = new HelloWorldEvent();
        event.setMessage(message);
        eventSender.sendInsertEvent(event);
    }
    public void setEventSender(StreamSender sender) {
        eventSender = sender;
    }
    public synchronized void suspend() {
        suspended = true;
    }
    private synchronized boolean isSuspended() {
        return suspended;
    }
}
```

9.2.3 HelloWorldEvent.java

The `HelloWorldEvent.java` class creates an event from a message. The HelloWorldAdapter.generateHelloMessage method calls the `HelloWorldEvent.setMessage method to create an event from a message.` The `HelloWorldBean` class stores the `message` and its generated `id` to and retrieves them from the data store.

```
package com.bea.wlevs.event.example.helloworld;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class HelloWorldEvent {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private long id;
  private String message;

  public String getMessage() {
    return message;
  }

  public void setMessage (String message) {
    this.message = message;
  }
}
```

9.2.4 HelloWorldBean.java

The `HelloWorldBean.java` class is an event sink and source that retrieves events from `HelloWorldEvent` and performs read and write operations on the database with JPA.

The Oracle Event Processing framework calls the `setEventSender` method to initialize the `m_eventSender` private variable with a `StreamSender` instance. The `onInserEvent` method sends the events emitted by the `StreamSource` instance downstream to `StreamSink` listeners.

```
package com.bea.wlevs.example.helloworld;

import java.util.HashMap;
import java.util.List;
import javax.annotation.Resource;
import javax.sql.DataSource;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import org.springframework.beans.factory.DisposableBean;
import org.eclipse.persistence.config.PersistenceUnitProperties;
import com.bea.wlevs.ede.api.StreamSink;
import com.bea.wlevs.ede.api.StreamSource;
import com.bea.wlevs.ede.api.StreamSender;
import com.bea.wlevs.event.example.helloworld.HelloWorldEvent;

public class HelloWorldBean implements StreamSink, StreamSource, DisposableBean {
```

```
    private static final String PERSISTENCE_UNIT_NAME = "helloworld";
    private EntityManagerFactory m_entityMgrFactory;
    private EntityManager m_entityMgr;
    private DataSource m_ds;
    private boolean m_shuttingDown;
    private StreamSender m_eventSender;

    public void setEventSender(StreamSender sender){
        m_eventSender = sender;
    }
    private void setupEntityManager(){
        if (m_entityMgr!=null)
            return;
        HashMap props = new HashMap();
        props.put(PersistenceUnitProperties.NON_JTA_DATASOURCE, m_ds);
        props.put("eclipselink.ddl-generation", "create-tables");
        props.put("eclipselink.ddl-generation.output-mode", "database");
        m_entityMgrFactory = Persistence.createEntityManagerFactory
                             (PERSISTENCE_UNIT_NAME, props);
        m_entityMgr = m_entityMgrFactory.createEntityManager();
    }
    public void onInsertEvent(Object event){
        if (m_shuttingDown)
            return;
        setupEntityManager();
        if (event instanceof HelloWorldEvent) {
            HelloWorldEvent helloWorldEvent = (HelloWorldEvent) event;
            System.out.println("Message: " + helloWorldEvent.getMessage());
            m_entityMgr.getTransaction().begin();
            try {
                m_entityMgr.persist(helloWorldEvent);
                m_entityMgr.getTransaction().commit();
            } finally {
             if (m_entityMgr.getTransaction().isActive())
              m_entityMgr.getTransaction().rollback();
            }
        }
        Query q = m_entityMgr.createQuery("select t from HelloWorldEvent t");
        List<HelloWorldEvent> hwlist = q.getResultList();
        System.out.println("Stored " + hwlist.size() + " helloworld events");
        m_eventSender.sendInsertEvent(event);
    }
    @Resource(name="derbyDS")
    public void setDataSource(DataSource ds){
        m_ds = ds;
    }
    public void destroy(){
        m_shuttingDown = true;

        if (m_entityMgr!=null){
            m_entityMgr.close();
            m_entityMgr=null;
        }
        if (m_entityMgrFactory!=null){
            m_entityMgrFactory.close();
            m_entityMgrFactory=null;
        }
    }
}
```

9.3 JPA Coherence Example

The JPA Coherence example demonstrates the usage of the EclipseLink JPA implementation for the Coherence `CacheLoader` or `CacheStore` interfaces.

9.3.1 persistence.xml Configuration File

The EclipseLink properties specify the settings for database read and write operations and logging. The managed persistable classes that represents objects in the database are `com.oracle.cep.sample.PriceTarget` and `com.oracle.cep.sample.SaleEvent`.

This `persistence.xml` file has entries for JPA logging that are commented out and set to `false`. You can uncomment these settings and set them to `true` to debug or otherwise monitor the application behavior. For information on property settings, see http://eclipse.org/eclipselink/documentation/2.4/jpa/extensions/toc.htm.

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/persistence      http://
java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
     version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="derby" transaction-type="RESOURCE_LOCAL">
    <class>com.oracle.cep.sample.PriceTarget</class>
    <class>com.oracle.cep.sample.SaleEvent</class>
    <properties>
      <property name="eclipselink.jdbc.read-connections.min" value="1"/>
      <property name="eclipselink.jdbc.write-connections.min" value="1"/>
      <property name="javax.persistence.jdbc.driver"
                value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="javax.persistence.jdbc.url"
                value="jdbc:derby:test1;create=true"/>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
      <property name="eclipselink.ddl-generation.output-mode" value="database"/>
    <!--
      <property name="eclipselink.logging.timestamp" value="false"/>
      <property name="eclipselink.logging.thread" value="false"/>
      <property name="eclipselink.logging.session" value="false"/>
      <property name="eclipselink.logging.exceptions" value="false"/>
      <property name="eclipselink.logging.connection" value="false"/>
      <property name="eclipselink.logging.level" value="FINER"/>
    -->
    </properties>
  </persistence-unit>
</persistence>
```

9.3.2 Classes

The example is comprised of the following classes:

- CoherenceMapListener.java
- PriceTarget.java
- PriceTargetLoader.java
- SaleEvent.java

- SaleEventsGenerator.java

In this example, an initial set of items go on sale and the requested target prices are set up in a data store. The data store is available in the `PriceTarget.java` Coherence cache because it is set up to be used with `CacheLoader`. A stream of `SaleEvents` is generated from the `SaleEventsGenerator` adapter. If the sale prices match the target prices, they are stored in `SaleEvent` Coherence cache. A Coherence `MapListener` implementation verifies that the `SaleEvents` stored in the cache are actually available in the data store as well.

9.3.2.1 CoherenceMapListener.java

The `CoherenceMapListener.java` class listens for events published to the coherence cache.

```
package com.oracle.cep.sample;

import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import org.springframework.beans.factory.DisposableBean;
import com.tangosol.util.MapListener;
import com.tangosol.util.MapEvent;
import com.tangosol.util.ObservableMap;
import com.bea.wlevs.ede.api.InitializingBean;
import com.bea.wlevs.ede.api.StreamSource;
import com.bea.wlevs.ede.api.StreamSender;

public class CoherenceMapListener implements MapListener,
InitializingBean,                                              StreamSource {
    private static final String PERSISTENCE_UNIT_NAME = "derby";
    private EntityManagerFactory m_entityMgrFactory;
    private EntityManager m_entityMgr;
    private ObservableMap m_saleEventCache;
    private StreamSender m_sender;

    public void afterPropertiesSet()
    {
        m_saleEventCache.addMapListener(this);
    }
    public void setEventSender(StreamSender sender)
    {
        m_sender = sender;
    }
    public void setSaleEventCache(ObservableMap cache)
    {
        m_saleEventCache = cache;
    }
    public void entryInserted(MapEvent event)
    {
        verifyEventInStore(event);
    }
    private void verifyEventInStore(MapEvent event){
        if (!(event.getNewValue() instanceof SaleEvent)){
            System.out.println("Unexpected type in SaleEvent cache");
            return;
        }
        if (m_entityMgr==null){
            setupEntityMgr();
```

```
        }
        SaleEvent sale = (SaleEvent) event.getNewValue();
        Query q = m_entityMgr.createQuery("SELECT s FROM SaleEvent s
                 WHERE s.itemID = :itemID");
        q.setParameter("itemID", sale.getItemID());
        List<SaleEvent> saleEvents = q.getResultList();
        if (saleEvents.size()==0)
            System.out.println("ERROR! Matched SaleEvent not found in store");
        else {
          System.out.println("Found sale event for " +
                             saleEvents.get(0).getItemID() + " for $" +
                             saleEvents.get(0).getSalePrice());
          m_sender.sendInsertEvent(sale);
        }
    }
    private void setupEntityMgr() {
      m_entityMgrFactory = Persistence.createEntityManagerFactory(
                           PERSISTENCE_UNIT_NAME);
      m_entityMgr = m_entityMgrFactory.createEntityManager();
    }
    public void entryUpdated(MapEvent event){
        verifyEventInStore(event);
    }
    public void entryDeleted(MapEvent event){
      System.out.println("SaleEvent cache entry deleted.
                         Should not see this event for this sample");
    }
}
```

9.3.2.2 PriceTarget.java

```
package com.oracle.cep.sample;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class PriceTarget implements java.io.Serializable {
    @Id
    private String itemID;
    private double targetPrice;

    public String getItemID() {
        return itemID;
    }
    public void setItemID(String itemID) {
        this.itemID = itemID;
    }
    public double getTargetPrice(){
        return targetPrice;
    }
    public void setTargetPrice(double targetPrice){
        this.targetPrice = targetPrice;
    }
}
```

9.3.2.3 PriceTargetLoader.java

```
package com.oracle.cep.sample;

import java.util.ArrayList;
```

```
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import org.springframework.beans.factory.DisposableBean;
import com.bea.wlevs.ede.api.InitializingBean;

public class PriceTargetLoader implements DisposableBean, InitializingBean {
    private static final String PERSISTENCE_UNIT_NAME = "derby";
    static ArrayList<PriceTarget> s_entriesToLoad = new ArrayList<PriceTarget>();
    static {
      setUpEntriesToLoad();
    }

    private EntityManagerFactory m_entityMgrFactory;
    private EntityManager m_entityMgr;

    public void afterPropertiesSet() {
      m_entityMgrFactory = Persistence.createEntityManagerFactory(
                                                          PERSISTENCE_UNIT_NAME);
      m_entityMgr = m_entityMgrFactory.createEntityManager();
      m_entityMgr.getTransaction().begin();
      try{
        Query q = m_entityMgr.createQuery("SELECT t FROM PriceTarget t
                                           WHERE t.itemID = :itemID");
        for (PriceTarget target : s_entriesToLoad){
          q.setParameter("itemID", target.getItemID());
          List<PriceTarget> targetList = q.getResultList();
          if (targetList.size()==0){
            System.out.println("Persisting target " + target.getItemID());
            m_entityMgr.persist(target);
          } else {
            System.out.println("Found target " + target.getItemID());
          }
        }
        m_entityMgr.getTransaction().commit();
     }
     finally {
       if(m_entityMgr.getTransaction().isActive())
         m_entityMgr.getTransaction().rollback();
     }
   }

   public void destroy() {
     if(m_entityMgr!=null) {
       m_entityMgr.close();
       m_entityMgr=null;
      }
      if(m_entityMgrFactory!=null){
       m_entityMgrFactory.close();
       m_entityMgrFactory=null;
     }
   }
   private static void setUpEntriesToLoad(){
    // 'smith', ipad2, $400
    PriceTarget target = new PriceTarget();
    target.setItemID("ipad2");
    target.setTargetPrice(400);
    s_entriesToLoad.add(target);
    // 'doe', kindle, $100
```

```
    target = new PriceTarget();
    target.setItemID("kindle");
    target.setTargetPrice(100);
    s_entriesToLoad.add(target);
    // walker, rebel, $400
    target = new PriceTarget();
    target.setItemID("rebel");
    target.setTargetPrice(400);
    s_entriesToLoad.add(target);
    // williams, lasko1320, $25
    target = new PriceTarget();
    target.setItemID("lasko1320");
    target.setTargetPrice(25);
    s_entriesToLoad.add(target);
  }
}
```

9.3.2.4 SaleEvent.java

```
package com.oracle.cep.sample;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class SaleEvent implements java.io.Serializable {
    @Id
    private String itemID;
    private double salePrice;

    public SaleEvent() { }
    public SaleEvent(String itemID, double salePrice){
        this.itemID = itemID;
        this.salePrice = salePrice;
    }
    public String getItemID(){
        return itemID;
    }
    public void setItemID(String itemID){
        this.itemID = itemID;
    }
    public double getSalePrice(){
        return salePrice;
    }
    public void setSalePrice(double salePrice) {
        this.salePrice = salePrice;
    }
    public String toString() {
         return "SaleEvent(" + itemID + ":" + salePrice + ")";
    }
}
```

9.3.2.5 SaleEventsGenerator.java

```
package com.oracle.cep.sample;

import java.util.Map;
import java.util.Random;
import com.bea.wlevs.ede.api.RunnableBean;
import com.bea.wlevs.ede.api.StreamSender;
import com.bea.wlevs.ede.api.StreamSource;
import com.bea.wlevs.ede.api.InitializingBean;
```

```
public class SaleEventsGeneraton implements RunnableBean, StreamSource,
                                  InitializingBean{
    private static final int SLEEP_MILLIS = 1000;
    private static final String[] s_itemIDs = {
          "kodaksport",
          "ipodtouch-8GB",
          "ipad2",
          "kindle",
          "garmin1690",
          "rebel",
          "logitech1080",
          "tomtom",
          "ipad2",
          "cuisinart10s",
          "keurig-b70",
          "lasko1320" };
    private static final double[] s_prices = {
          60.0,
          200.0,
          450.0,
          99,
          120,
          400,
          70,
          100,
          399,
          100,
          150,
          20 };

    private boolean m_suspended;
    private Thread m_thread;
    private StreamSender m_sender;
    private Map m_priceTargetCache;

    public void setPriceTargetCache(Map cache){
        m_priceTargetCache = cache;
    }
    public void afterPropertiesSet() {
        // pre-load PriceTarget cache
        for (PriceTarget target : PriceTargetLoader.s_entriesToLoad)
        {
            System.out.println("Getting : " + target.getItemID());
            m_priceTargetCache.get(target.getItemID());
        }
    }
    public void run() {
        m_thread = Thread.currentThread();
        m_suspended = false;
        // send random sale events
        Random random = new Random(System.currentTimeMillis());
        while (!isSuspended())
        {
            int index = random.nextInt(s_itemIDs.length);
            SaleEvent event = new SaleEvent(s_itemIDs[index], s_prices[index]);
            m_sender.sendInsertEvent(event);
            try {
                synchronized (this) { wait(SLEEP_MILLIS); }
            } catch (InterruptedException e) {
                if (isSuspended())
```

```
                    return;
                }
            }
        }
        public void setEventSender(StreamSender sender) {
            m_sender = sender;
        }
        public synchronized void suspend() {
            m_suspended = true;
            if (m_thread!=null)
                m_thread.interrupt();
        }
        private synchronized boolean isSuspended() {
            return m_suspended;
        }
    }
```

10

Web Services

You can use web services platforms to integrate an Oracle Event Processing application with other systems. This chapter explains how to invoke services from an application and how to expose an application as a web service.

This chapter includes the following sections:

10.1 Supported Platforms

Oracle Event Processing supports version 2.0 of the JAX-WS API standard using the Glassfish reference implementation of JAX-WS 2.0, including:

- JAX-WS 2.0 (Java API for XML Web Services, defined in JSR 224)
- WS-I Basic Profile 1.1
- WS-I Attachments Profile 1.0 (SOAP Messages with Attachments)
- WS-I Simple SOAP Binding Profile 1.0
- SOAP 1.1 and 1.2 (Simple Object Access Protocol)
- MTOM (Message Transmission Optimization Mechanism)
- WSDL 1.1 (Web Services Definition Language)
- JAXB 2.0 (Java API for XML Binding, references through a separate JAXB module)
- SAAJ 1.3 (SOAP with Attachments API for Java).

10.2 Invoke a Web Service From an Application

This procedure describes how to create an application that invokes a web service. In this scenario, the application is the web service client.

Invoke a Web Service from an Application:

1. Create or obtain the web service definition language (WSDL) file for the web service.

   This example uses a WSDL named `EchoService.WSDL`.

2. Generate the compiled class files you need to invoke the Web Service with the following command. Keep the entire command on one line:

```
java -cp /Oracle/Middleware/my_oep/modules/
com.bea.core.ws.glassfish.jaxws.tools_12.0.0.0.jar com.sun.tools.ws.WsImport
EchoService.WSDL
```

3. Archive the generated class files the Oracle Event Processing application JAR file.

4. Add the `Export-Package` header and packages to the `MANIFEST.MF` file to export the web services Java packages for the client code:

```
Export-Package: com.oracle.ocep.sample.echoService;
```

5. Add the following packages to the `MANIFEST.MF` file with the `Import-Package` header:

```
Manifest-Version: 1.0
Export-Package: echo
Bundle-Vendor: %project.vendor
Bundle-ClassPath: .,lib/echo.jar
Bundle-Version: 1.0.0
Bundle-Localization: bundle
Bundle-ManifestVersion: 2
Bundle-Name: %project.name
Import-Package: com.bea.wlevs.configuration;version="11.1.1",com.bea.w
 levs.ede;version="11.1.1",com.bea.wlevs.ede.api;version="11.1.1",com.
 bea.wlevs.ede.impl;version="11.1.1",com.bea.wlevs.ede.spi;version="11
 .1.1",com.bea.wlevs.management.spi;version="11.1.1",com.bea.wlevs.spr
 ing;version="11.1.1",com.bea.wlevs.spring.support;version="11.1.1",co
 m.bea.wlevs.util;version="11.1.1",com.ctc.wstx.stax;version="4.0.5",c
 om.sun.xml.bind.v2;version="2.1.14",com.sun.xml.bind.v2.model.annotat
 ion;version="2.1.14",com.sun.xml.messaging.saaj.soap;version="2.1.0",
 com.sun.xml.messaging.saaj.soap.ver1_1;version="2.1.0",javax.jws,java
 x.xml,javax.xml.bind,javax.xml.bind.annotation,javax.xml.namespace,ja
 vax.xml.transform.stream,oracle.jdbc;version="1.1.0.0_11-2-0-2-0",ora
 cle.sql;version="1.1.0.0_11-2-0-2-0",org.apache.commons.logging;versi
 on="1.1.0",org.springframework.beans;version="2.5.6",org.springframew
 ork.beans.factory;version="2.5.6",org.springframework.beans.factory.c
 onfig;version="2.5.6",org.springframework.core.annotation;version="2.
 5.6",org.springframework.ejb.config,org.springframework.osgi.context;
 version="1.2.0",org.springframework.osgi.extensions.annotation;versio
 n="1.2.0",org.springframework.osgi.service;version="1.2.0",org.spring
 framework.util;version="2.5.6",org.xml.sax,org.xml.sax.ext,weblogic.j
 dbc.extensions;version="1.10.0.0",weblogic.xml.stax;version="1.10.0.0
 "
```

6. Add the following lines of code to your application to invoke the web service:

```
EchoService service = new EchoService();
EchoPort port = service.getEchoServicePort();
String echo = port.echo("foo");
```

10.3 Expose an Application as a Web Service

In this example, the application is the web service provider.

To Expose an Application as a Web Service:

1. Create or obtain the WSDL for the web service.

This example uses a WSDL named `EchoService.WSDL`.

2. Implement the service.

   Consider using the `java.jws` annotations `@WebService` and `@WebMethod`.

3. Add a `bea-jaxws.xml` file to your application bundle. Table 10-1 describes the attributes in this file.

```
<endpoints>
  <endpoint>
    <name>EchoService</name>
    <implementation-class>
      com.bea.wlevs.test.echo.impl.EchoServiceImpl
    </implementation-class>
    <url-pattern>/echo</url-pattern>
    <wsdl-location>
      /META-INF/wsdl/echo.wsdl
    </wsdl-location>
    <service-name>
      {http://wsdl.oracle.com/examples/cep/echo}EchoService
    </service-name>
    <port-name>
      {http://wsdl.oracle.com/examples/cep/echo}EchoServicePort
    </port-name>
  </endpoint>
</endpoints>
```

Table 10-1 bea-jaxws.xml File Attributes

Attribute	Description
`name`	The name of the web service.
`implementation-class`	The class that implements the service.
`url-pattern`	The url pattern to access the web service.
`wsdl-location`	Relative path to the wsdl in the bundle.
`service-name`	QName of the service.
`port-name`	QName of the port.

4. Reference the `bea-jaxws.xml` file in the `MANIFEST.MF` file with the `BEA-JAXWS-Descriptor` header:

```
BEA-JAXWS-Descriptor: META-INF/bea-jaxws.xml;
```

5. Add the `Import-Package` header and packages to the `MANIFEST.MF` file to import the following packages to the application:

```
Import-Package: com.ctc.wstx.stax,
   com.bea.core.ws.glassfish.jaxws,
   com.sun.xml.bind.v2,
   com.sun.xml.messaging.saaj.soap,
   com.sun.xml.ws,
   javax.jws,
   javax.xml.bind,
   javax.xml.bind.annotation,
```

```
javax.xml.namespace,
javax.xml.soap,
javax.xml.transform,
javax.xml.transform.stream,
javax.xml.ws,
javax.xml.ws.spi,
org.xml.sax,
weblogic.xml.stax
```

6. Add a `glassfish-ws` element to the Oracle Event Processing server *`DOMAIN_DIR`*`/config/config.xml` file that describes your Oracle Event Processing domain, where *`DOMAIN_DIR`* refers to your domain directory:

```
<glassfish-ws>
    <name>JAXWS</name>
    <http-service-name>JettyServer</http-service-name>
</glassfish-ws>
```

11

Parameterized Applications

Parameterized Applications contain special parameters (variables) that must be configured before the application starts. Generally, you configure a parameterized application when you deploy the application with the Oracle Event Processing EPN shell. After a parameterized applications is configured, it functions as a regular application.

This chapter includes the following sections:

11.1 Application Parameters

Application parameters define property values that can be set when the user launches an application. You can parameterize properties for adapters, channels, event beans, Spring beans, and Oracle CQL parameterized queries with and without views.

You define application parameters (attributes) by grouping attribute definitions (ADs) into object class definitions (OCDs). You create one OCD for each application component that you want to parameterize. An OCD contains one or more ADs to specify the component properties to parameterize and the prompt text.

Place the OCDs in an XML document (metatype file) in the `OSGI-INF/metatype` directory within the Oracle Event Processing application. The metatype file uses and complies with the schema defined by the specification at: `http://www.osgi.org/xmlns/metatype/v1.1.0/metatype.xsd`.

See Example metatype File for an example configuration.

Oracle Event Processing parameterized applications conform to the OSGi MetaType specification at `http://www.osgi.org`. See the Apache implementation at `http://felix.apache.org/documentation/subprojects/apache-felix-metatype-service.html`.

11.2 Object Class Definitions

Every OCD requires the `name`, id, and `description` parameters. The following example is an OCD with no ADs. This OCD defines application metadata, but because there are no ADs, the description displays, but the user is not prompted for input. This type of OCD documents the application.

```
<OCD name="HelloWorld Sample" id="com.oracle.cep.sample.helloworld"
  description="The helloworld OEP application is a sample application
  for Oracle Event Processing.">
</OCD>
```

11.3 Attribute Descriptions

Every AD requires the `name`, id, and `description` parameters. The following example is an OCD with two ADs. The OCD provides a common definition for the channel component and provides parameterized attributes (ADs) for the maximum number of threads and the maximum buffer size of the channel.

When the user runs the application, the application displays the description and the other information that you provided, and waits for the user to enter the requested information and press the Return key.

```
<OCD name="Channel Configuration" id="com.oracle.cep.channel"
  description="The channel definition in the OCEP Application"
  ocep:binding="jmx:EventChannel">
  <AD name="Max Threads" id=".maxThreads" type="Integer" required="true"
    default="0" min="0" max="100"
    description="Number of threads generating messages."
    ocep:binding="MaxThreads" />
  <AD name="Max Size" id=".maxSize" type="Integer" required="true"
    default="0" min="0" max="100"
    description="The maximum size of the FIFO buffer for this channel."
    ocep:binding="MaxSize" />
</OCD>
```

Note:

To avoid errors, always define Max Threads before Max Size in the metatype file.

For the 12c release, the only supported binding is Java Management Extension (JMX). This means that the AD attributes must be bound to a corresponding JMX attribute. See Targeting.

11.4 Targeting

You can connect an OCD and an application component with the `Designate` element. The application class definition can then be a target for multiple designates (components of the same type) to enable the reuse of definitions. To prevent ambiguities, each designate can be associated with one object class definition only.

Oracle Event Processing supports the following designates:

- Adapter, Oracle CQL Processor, event bean, or Spring bean.

- Oracle CQL processor rules such as a query or a view. In the case of a rule, the designate ID must identify the parent component followed by the subcomponent. For example, `helloworldProcessor:q1`.

Oracle Event Processing assigns a parameter to an application component after the call to the `afterPropertiesSet` life cycle method, but prior to the call to `afterConfigurationActive` life cycle method. This timing enables an application to treat an application parameter as a proper application configuration and to distinguish an application parameter from a bean property.

An `ocep:binding` attribute determines how Oracle Event Processing assigns the parameter to the application. Put the ocep:binding attribute in the OCD or in the AD. When you put it in the OCD, the meta-object is bound to an implementation object such as JMX `ObjectInstance`. When you put the OCD in an AD, the meta-attribute is bound to an implementation attribute scoped to the implementation object, such as JMX `MBean` attribute.

11.5 Example metatype File

The following code shows the entire metatype file.

```
<MetaData xmlns="http://www.osgi.org/xmlns/metatype/v1.1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://www.osgi.org/xmlns/metatype/v1.1.0;
    http://www.osgi.org/xmlns/metatype/v1.1.0/metatype.xsd">

  <OCD name="HelloWorld Sample" id="com.oracle.cep.sample.helloworld"
    description="The helloworld OEP application is a sample application
    for Oracle Event Processing.">
  </OCD>

  <OCD name="HelloWorld Channel" id="com.oracle.cep.sample.helloworld.channel" >
    <AD name="Max Threads" id="EventChannel.MaxThreads" type="Integer"
        required="true" default="0" min="0" max="100"
        description="Number of threads generating helloworld messages." />

    <AD name="Max Size"     id=".maxSize"     type="Integer"  required="true"
        default="0"  min="0" max="100"  description="The maximum size of the FIFO
        buffer for this channel." ocep:binding="MaxSize" />
  </OCD>

  <OCD name="HelloWorld Message Filtering"
    id="com.oracle.cep.sample.helloworld.filter" >
    <AD name="Filter" id="CQLProcessor.parameters" type="String"
      description="Message filter." />
  </OCD>

  <Designate pid="helloworld"  >
    <Object ocdref="com.oracle.cep.sample.helloworld" />
  </Designate>

  <Designate pid="helloworldInputChannel"  >
    <Object ocdref="com.oracle.cep.sample.helloworld.channel" />
  </Designate>

  <Designate pid="helloworldProcessor.helloworldRule" >
    <Object ocdref="com.oracle.cep.sample.helloworld.filter" />
  </Designate>
</MetaData>
```

11.6 Where You Can Use Parameterized Applications

You can use parameterized applications in the following three ways:

- Document an Application
- Channel Configuration
- Oracle CQL Processor Query

11.6.1 Document an Application

The following example documents the `com.oracle.cep.parameterizedapp` application. The `pid` attribute of the `designate` element corresponds to the Bundle-SymbolicName in the `MANIFEST.MF`.

```
<OCD name="Parameterized App Testing" id="com.oracle.cep.parameterizedapp"
  description="The application is for parameterized app testing.">
</OCD>

<Designate pid="parameterizedapp">
  <Object ocdref="com.oracle.cep.parameterizedapp" />
</Designate>
```

11.6.2 Channel Configuration

The following example shows how to use an OCD with ADs to configure a channel. The `pid` attribute of the `designate` element corresponds to the channel component in the EPN file.

```
<OCD name="Channel Configuration" id="com.oracle.cep.channel"
 description="The channel definition in the OCEP Application"
 ocep:binding="jmx:EventChannel">
 <AD name="Max Threads" id=".maxThreads" type="Integer" required="true"
   default="0" min="0" max="100"
   description="Number of threads generating messages."
   ocep:binding="MaxThreads" />
 <AD name="Max Size" id=".maxSize" type="Integer" required="true"
   default="0" min="0" max="100"
   description="The maximum size of the FIFO buffer for this channel."
   ocep:binding="MaxSize" />
 <AD name="HEARTBEAT" id=".heartbeat" type="Long" required="false"
   default="5000000000" min="0" max="100000000000"
   description="The value for the heartbeat timeout on this channel.
   The default time unit is nanoseconds."
   ocep:binding="HeartbeatTimeout" />
</OCD>

<Designate pid="helloworldInputChannel">
  <Object ocdref="com.oracle.cep.channel" />
</Designate>

<Designate pid="helloworldOutputChannel">
  <Object ocdref="com.oracle.cep.channel" />
</Designate>
```

11.6.3 Oracle CQL Processor Query

The following example shows how to configure an Oracle CQL processor query. The ADs definition correspond to query parameters 1 and 2 in the sequence.

```
<OCD name="Product Filter" id="com.oracle.cep.solution.product.filter"
  ocep:binding="jmx:CQLProcessor" ocep:multi-valued="true">
  <AD name="range" id=".range" type="Integer" required="true" min="0"
    max="1000000000" default="10"
    description="The range scope"
    ocep:binding="Parameters" />
  <AD name="field" id=".field" type="String" required="true"
    default="'remainingQty'"
    description="select one field you want to do range controlling"
    ocep:binding="Parameters">
    <Option label="remainingQty" value="remainingQty" />
    <Option label="totalQty" value="totalQty" />
    <Option label="price" value="price" />
  </AD>
</OCD>

<Designate pid="productProcessor.productRule">
  <Object ocdref="com.oracle.cep.solution.product.filter" />
</Designate>
```

Add a `processor` element to the `config.xml` file that defines the `productProcessor`, as follows. Note that the ADs definitions correspond to query parameters :1 and :2 in the sequence.

```
<processor>
  <name>productProcessor</name>
  <rules>
    <query id="productRule">
      <![CDATA[
       IStream (select FROM productInputChannel [RANGE :1 on :2) ]]
      >
    </query>
  </rules>
</processor>
```

11.7 Deploy the HelloWorld Application

The following example uses the EPN shell with Apache Felix Gogo to deploy the parameterized HellowWorld application. The EPN shell interfaces with OSGi Bundle Repository (OBR) and uses introspection to locate the available metadata and to form the applicable prompts to the user.

The following example shows the HelloWorld application output when it is deployed with the EPN shell.

Note:

Parametrized applications are not supported on a clustered domain. Also you can deploy only through the EPN shell. For this release, parameterized application deployment is not supported in Oracle JDeveloper or in Oracle Event Processing Visualizer.

```
    ./startwleves.sh -shell
Oracle CEP Shell (using Apache Felix Gogo)
    shell>
    shell> deployapp file:///Users/myuserid/helloworld.jar
    ---- HelloWorld Sample ----
    The helloworld OEP application is a sample application for Oracle OEP.
    Enter Y/N [default is Y] if you would like to set the parameter "Max Threads" :
-- Application parameter "Max Threads"
    --Description : Number of threads generating helloworld messages.
    Type: Integer
Default value: 2
Enter value for "Max Threads" or empty for default value :
    Using default value of "2".
-- Application parameter "Filter"
--Description : Message filter.
    Type: String
Options for parameter "Filter" are :
(0) Select all messages starting with 'Hello'
        (1) Select all messages starting with 'Hi'Select option by entering number
[0,1] or empty for default value :
    <Jan 23, 2012 7:20:40 AM EST> <Notice> <Deployment> <BEA-2045000>
    <The application bundle "helloworld" was deployed successfully>
    <Jan 23, 2012 7:20:41 AM EST> <Notice> <Spring> <BEA-2047000>
    <The application context for "helloworld" was started successfully>
    Message: HelloWorld - the current time is:7:20:41 AM
    Message: HelloWorld - the current time is:7:20:42 AM
```

12

Internationalization

You can use message catalogs to internationalize message text that your application sends to the server log or displays to the user. The messages can provide information about anything occurring in the application, such as which user invoked specific application components, error conditions, or help you debug an application before its release.

This chapter includes the following sections:

12.1 Message Catalogs

A message catalog is a single XML file that contains a collection of messages, with each message indexed by a unique ID. All internationalized text is externalized and defined in message catalogs and each message catalog defines a collection of log messages or simple text. With message catalogs, message strings can be converted to multiple locales without changing or recompiling the application code.

Message IDs are unique across all log message or locale message catalogs. Within the message catalog file, each localized version of the message is assigned a unique message ID and message text specific to the error. Ideally, a message is logged from only one location within the system so that a support team can easily find it. Message IDs in simple text catalogs are unique within each simple text catalog.

There are the following three types of message catalogs:

- Log message catalogs: Informational or error messages that your application logs to the server logs.
- Simple text message catalogs: Simple messages that your application displays to the user.
- Locale message catalogs: A collection of locale-specific messages that correspond to a top-level log message catalog or a simple text catalog that contains the English version of the messages. There are corresponding locale-specific catalogs with one catalog for each additional supported locale.

  The top-level English version catalog has all of the information needed to define the message. The locale-specific catalogs contain only the message ID, the date changed, and the translation of the message for the specific locale.

You create your own catalog of log or simple text messages and use Oracle WebLogic utilities to generate Java classes that have logging methods. You import the Java classes into your application code, and implement your application code to supply runtime values to the logging methods. The log messages generated by the logging

methods are integrated with and treated in the same way as log messages that are generated by the Oracle Event Processing server.

The message catalog files are defined by one of the following XML document type definition (DTD) files:

- `msgcat.dtd`: Describes the syntax of top-level, default catalogs.
- `l10n_msgcat.dtd`: Describes the syntax of locale-specific catalogs.

The DTDs are stored in `/Oracle/Middleware/wlevserver/modules/com.bea.core.i18n.generator_VERSION.jar`. *`VERSION`* points to a particular version that changes.

You can create a single log message catalog for all logging requirements, or create smaller catalogs based on a subsystem or on Java packages. Oracle recommends using multiple subsystem catalogs so you can focus on specific portions of the log during viewing. For simple text catalogs, Oracle recommends that you create a single catalog for each utility to be internationalized

12.1.1 Hierarchy

All messages must be defined in the default, top-level catalog. Catalogs that provide different localizations of the base catalogs are defined in `msgcat` subdirectories named for the locale, for example, `msgcat/de` for Germany. You might have a top-level catalog named `mycat.xml`, and a German translation called `..de/mycat.xml`. Typically the top-level catalog is English. However, English is not required for any catalogs.

Locale designations, for example, `de`, also have a hierarchy as defined in the `java.util.Locale` documentation. A locale can include a language, country, and variant. Language is the most common locale designation. Language can be extended with a country code. For example, `en\US`, indicates American English. The name of the associated catalog is `..en\US\mycat.xml`. Variants are specific to a vendor or browser and are used to introduce minor differences, such as collation sequences, between two or more locales defined by either language or country.

12.1.2 Naming

Because the name of a message catalog file, without the `.xml` extension, is used to generate runtime class and property names, choose the name carefully. Follow these guidelines for naming message catalogs:

- Do not choose a message catalog name that conflicts with the names of existing classes in the target package for which you are creating the message catalog.
- Choose message catalogs names that contain only characters that are allowed in class names.
- Follow class naming standards.

For example, the resulting class names for a catalog named `Xyz.xml` are `XyzLogLocalizer` and `XyzLogger`.

The following considerations also apply to message catalog files:

- Log message IDs are generally six-character strings with leading zeros. Some interfaces also support integer representations.
- Simple text catalogs message IDS can consist of any string value.

- Java lets you group classes into a collection called a package. Oracle recommends that a package name be consistent with the name of the subsystem in which a particular catalog resides. Consistent naming makes OSGi imports easier to define.
- The log Localizer classes are actually `ResourceBundle` property files.

12.1.3 Message Arguments

The message body, message detail, cause, and action sections of a log message can include message arguments, as described by `java.text.MessageFormat`. Make sure your message content conforms to the patterns defined by `java.text.MessageFormat`.

Arguments are values that can be dynamically set at runtime. These values are passed to routines, such as a routine for printing a message. A message can support up to 10 arguments, numbered 0-9. You can include any subset of these arguments in any text section of the message definition (message body, message detail, probable cause), although the message body must include all of the arguments.

- Only the message body section in a simple text message can include arguments.
- You must assign a severity level for log messages. Log messages are generated through the generated Logger methods, as defined by the method attribute.

You insert message arguments into a message definition during development, and these arguments are replaced by the appropriate message content at runtime when the message is logged.

- Arguments are type `String` or representable as a `String` type.
- Numeric data is represented as `{n,number}`.
- Dates are supported as `{n,date}`.

The following excerpt from an XML log message definition shows how to use message arguments. The argument number must correspond to one of the arguments specified in the method attribute. Specifically, `{0}` with the first argument, `{1}` with the second, and so on. In the following example, `{0}` represents the file that cannot be opened, while `{1}` represents the file that is opened in its place.

```
<messagebody>Unable to open file, {0}. Opening {1}. All arguments must be in body.</messagebody>

    <messagedetail> File, {0} does not exist. The server will restore the file
    contents from {1}, resulting in the use of default values for all future
    requests. </messagedetail>

    <cause>The file was deleted</cause>

    <action>If this error repeats then investigate unauthorized access to the
    file system.</action>
```

An example of a method attribute is as follows:

```
-method="logNoFile(String name, String path)"
```

The message example expects two arguments, `{0}` and `{1}`:

- Both are used in the `<messagebody>`
- Both are used in the `<messagedetail>`
- Neither is used in `<cause>` or `<action>`

12.1.4 Formats

The catalog format for top-level and locale-specific catalog files differs. Top-level catalogs define the textual messages for the base locale. Locale-specific catalogs only provide translations of text defined in the top-level version. Also, log message catalogs are defined differently from simple text catalogs.

Log Message Catalog

This example shows the `MyUtilLog.xml` message catalog that has one log message to show how to use the `messagebody`, `messagedetail`, `cause`, and `action` elements.

```
<?xml version="1.0"?>
<!DOCTYPE message_catalog PUBLIC "weblogic-message-catalog-dtd"
"http://www.bea.com/servers/wls90/dtd/msgcat.dtd">
<message_catalog
  l10n_package="programs.utils"
  i18n_package="programs.utils"
  subsystem="MYUTIL"
  version="1.0"
  baseid="600000"
  endid="600100"
  <log_message
    messageid="600001"
    severity="warning"
    method="logNoAuthorization(String arg0, java.util.Date arg1,int arg2)"
    <messagebody>
      Could not open file, {0} on {1,date} after {2,number} attempts.
    </messagebody>
    <messagedetail>
      The configuration for this application will be defaulted to
      factory settings. Custom configuration information resides
      in file, {0}, created on {1,date}, but is not readable.
    </messagedetail>
    <cause>
      The user is not authorized to use custom configurations. Custom
     configuration information resides in file, {0}, created on
     {1,date}, but is not readable.The attempt has been logged to
    </cause>
     the security log.
    <action>
      The user needs to gain approriate authorization or learn to
      live with the default settings.
    </action>
  </log_message>
</message_catalog>
```

Simple Text Catalog

This example shows the `MyUtilLabels.xml` text catalog with one text definition.

```
<?xml version="1.0"?>
<!DOCTYPE message_catalog PUBLIC "weblogic-message-catalog-dtd"
    "http://www.bea.com/servers/wls90/dtd/msgcat.dtd">
<message_catalog>
  l10n_package="programs.utils"
  i18n_package="programs.utils"
  subsystem="MYUTIL"
  version="1.0"
  <message>
    messageid="FileMenuTitle"
    <messagebody>
      File
    </messagebody>
  </message>
</message_catalog>
```

Locale-Specific Catalog

This example shows a French translation of a message that is in the `..\msgcat\fr\MyUtilLabels.xml` file.

```
<?xml version="1.0"?>
<!DOCTYPE message_catalog PUBLIC
   "weblogic-locale-message-catalog-dtd"
   "http://www.bea.com/servers/wls90/dtd/l10n_msgcat.dtd">
<locale_message_catalog
  l10n_package="programs.utils"
  subsystem="MYUTIL"
  version="1.0">
  <message>
    <messageid="FileMenuTitle">
    <messagebody> Fichier </messagebody>
  </message>
</locale_message_catalog>
```

When you enter text in the `messagebody`, `messagedetail`, `cause`, and `action` elements, use a tool that generates valid Unicode Transformation Format-8 (UTF-8) characters, and have appropriate keyboard mappings installed. UTF-8 is an efficient encoding of Unicode character-strings that optimizes the encoding ASCII characters. Message catalogs always use UTF-8 encoding.

12.1.5 Message Catalog Localization

Catalog subdirectories are named after lowercase, two-letter ISO 639 language codes, for example, `ja` for Japanese and `fr` for French. You can find supported language codes in the `java.util.Locale` Javadoc.

You can achieve variations to language codes with uppercase, two-letter ISO 3166 country codes and variants, each of which are subordinate to the language code. The generic syntax is `lang\country\variant`.

For example, `zh` is the language code for Chinese. `CN` is a country code for simplified Chinese, whereas `TW` is the country code for traditional Chinese. Therefore `zh\CN` and `zh\TW` are two distinct locales for Chinese.

Variants are helpful when there is a functional difference in platform vendor handling of specific locales. Examples of vendor variants are `WIN`, `MAC`, and `POSIX`. There may be two variants used to further qualify the locale. In this case, the variants are separated with an underscore (for example, `Traditional_Mac` as opposed to `Modern_MAC`).

Note:

Language, country, and variants are all case sensitive.

A fully-qualified locale would look like `zh\TW\WIN`, identifying traditional Chinese on a Win32 platform. Message catalogs to support the above locale involve the following files:

- `\*.xml` - default catalogs
- `\zh\*.xml` - Chinese localizations
- `\zh\TW\*.xml` - Traditional Chinese localizations
- `\zh\TW\WIN\*.xml` - Traditional Chinese localizations for Win32 code sets

Specific localizations do not need to cover all messages defined in parent localizations.

12.2 Generate Localization Classes

After you create your message catalog XML files, use the `weblogic.i18ngen` utility to create the `Logger` and `TextFormatter` classes that localize the text in log messages. The weblogic.i18ngen utility creates or updates the `i18n_user.properties` properties file that loads the message ID lookup hashtable `weblogic.i18n.L10nLookup`. Any errors, warnings, or informational messages are sent to `stderr`.

For user catalogs to be recognized, the `i18n_user.properties` file must reside in a directory identified in the Oracle Event Processing server class path. Oracle recommends that the `i18n_user.properties` file reside in the Oracle Event Processing server class path. If the `i18n_user.properties` file is in `targetdirectory`, then `targetdirectory` should be in the Oracle Event Processing server class path.

Parse a Message Catalog to Generate the Logger and TextFormatter Classes

The following steps summarize how to create an internationalized message to use with the Oracle Event Processing server.

1. Create or edit a top-level log message catalog or simple text message catalog by defining the messages in the catalog.

   In addition to message text, include information about the type and placement of runtime values that the message contains.

2. Run `weblogic.i18ngen` to validate the catalog you created or edited in Step 1 to generate the `Logger` and `TextFormatter` classes.

   ```
   java weblogic.i18ngen [options] [filelist]
   ```

   The generated classes contain a method for each message. The class is defined according to information specified in the message catalog entry. The classes include methods for logging or getting message text, depending on the type of catalog. The class name ends with Logger or TextFormatter. For details, see weblogic.i18ngen Utility.

Table 12-1 weblogic.i18ngen Utility Options

Option	Description
`-build`	Generates all necessary files and compiles them. Combines the `-i18n`, `-l10n`, `-keepgenerated`, and `-compile` options.
`-d` *`targetdirectory`*	Specifies the root directory for the generated Java source files. User catalog properties are placed in `i18n_user.properties`, relative to the *`targetdirectory`*. Files are placed in appropriate directories based on the `i18n_package` and `l10n_package` values in the corresponding message catalog. The default target directory is the current directory and is created as necessary. If this argument is omitted, all classes are generated in the current directory, without regard to any class hierarchy described in the message catalog.
`-n`	Parse and validate, but do not generate classes.

Table 12-1 (Cont.) weblogic.i18ngen Utility Options

Option	Description
`-keepgenerated`	Keep generated Java source (located in the same directory as the class files).
`-ignore`	Ignore errors.
`-i18n`	Generates internationalizers (for example, `Loggers` and `TextFormatters`).
`-l10n`	Generates localizers (for example, LogLocalizers and TextLocalizers).
`-compile`	Compiles generated Java files using the current `CLASSPATH`. The resulting classes are placed in the directory identified by the -d option. The resulting classes are placed in the same directory as the source. Errors detected during compilation generally result in no class files or properties file being created. i18ngen exits with a bad exit status.
`-nobuild`	Parse and validate only.
`-debug`	Debugging mode.
`-dates`	Causes `weblogic.i18ngen` to update message time stamps in the catalog. If the catalog is writable and time stamps have been updated, the catalog is rewritten.
`filelist`	Process the files and directories in this list of files. If directories are listed, the command processes all XML files in the listed directories. The names of all files must include an XML suffix. All files must conform to the msgcat.dtd syntax. weblogic.i18ngen prints the fully-qualified list of names (Java source) to the stdout log for those files actually generated.

3. Create locale-specific catalogs as needed for the message catalog you created in Step 1.
4. Run `weblogic.l10ngen` to process the locale-specific catalogs you created in Step 3.
5. Code your application to use the `Logger` or `TextFormatter` classes you generated in Step 2.

   Use the OSGi import statements in the application `MANIFEST.MF` file to import the following packages into your Oracle Event Processing application:

   `weblogic.i18n.logging` `weblogic.logging`

   When your application calls one of the `Logger` or `TextFormatter` methods to log or return a message, the method writes the localized version of the message text to the target location. A `Logger` method writes the localized version of the message to the localized log files, and a `TextFormatter` method writes the localized version of the message to the display.
6. Make sure that the `i18n_user.properties` file is in the Oracle Event Processing server class path.

The `weblogic.i18ngen` utility generates the `i18n_user.properties` file that loads the message ID lookup hashtable `weblogic.i18n.L10nLookup`.

Part II

Deploy, Test, and Debug

Part II contains the following chapters:

13

Assemble and Deploy

To deploy and run an Oracle Event Processing application, you assemble the application files into an OSGi bundle and deploy the OSGi bundle to a domain on an Oracle Event Processing server. A deployed application processes client requests in the domain to which it is deployed.

You can assemble and deploy an application in Oracle JDeveloper, with Oracle Event Processing Visualizer, and manually. This chapter explains how to assemble and deploy an application wrapped in an OSGi bundle manually

See *Using Visualizer for Oracle Event Processing* for information about how to assemble and deploy and application and how to deploy and view an application library with Oracle Event Processing Visualizer.

This chapter includes the following sections:

13.1 OSGi bundles

An OSGi bundle contains one or more JAR files that contain the Java classes, packages, and libraries that define an Oracle Event Processing application and its services and resources (dependencies). You can make the application services and resources available to other bundles. You can also create an OSGi bundle that contains only services and resources that are used by other bundled applications. For example, an OSGi bundle can contain a JDBC driver that is accessed by other applications that execute in the same domain.

An OSGi bundle that contains an application and its resources is an application bundle, and an OSGi bundle that contains only resources is an application library bundle.

An OSGi application bundle contains the following files:

- The compiled Java class files that implement the application components, such as event types and event beans.

- One or more Oracle Event Processing configuration files that contain the component configurations.

  Place the configuration files in the `META-INF/wlevs` directory of the OSGi bundle JAR file to deploy. For example, `/Oracle/Middleware/my_oep/user_projects/domains/<domain>/<server>/applications/<OSGi_Bundle_Symbolic_Name>`.

  If you have an application already in the domain directory, then extract the configuration files in the same directory as the application files.

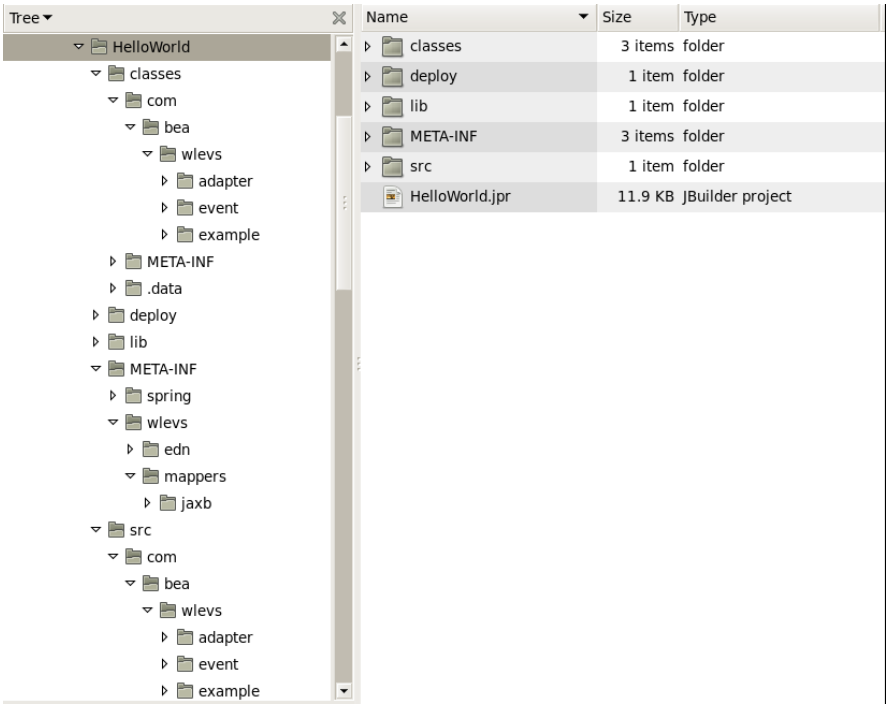
- An assembly file that describes all of the application components and how they connect to each other.

  Place the assembly file in the OSGi bundle JAR file `META-INF/spring` directory.

- A `MANIFEST.MF` file that describes the contents of the JAR file. This file enables you to make the application resources available to other bundles.

The following directory structure shows the structure of an OSGi application bundle:

Figure 13-1 OSGi Application Bundle Structure



13.2 Application Dependencies

Applications depend on imported packages and libraries, which can be shared with other applications executing in the same domain. You specify OSGi bundle dependencies in the `MANIFEST-MF` as follows:

- Use the `Import-Package` attribute to list imported packages and libraries.
- Use the `Export-Package` attribute to list packages that other OSGi bundles need to access. These packages are not bundled in and deployed with the application OSGi bundle, but are deployed to the Oracle Event Processing server application library directory.

13.3 Application Libraries

Application libraries make services and resources available to other applications executing in the same domain. You can use application libraries to add functionality such as drivers or foreign stages to your application. A foreign stage is a stage that is in another Oracle Event Processing application.

You can add an application library to a project as an embedded JAR file, but using an OSGI application library has the following advantages:

- Simplified application assembly and maintenance activities, such as deploying an updated version of the library.
- Artifact reuse.
- Reduced server disk space consumption.

You deploy application libraries to the Oracle Event Processing server into the library and library extensions directories.

13.3.1 Library Directory

By default, the Oracle Event Processing server library directory is `DOMAIN_DIR/servername/modules`, for example:

```
/Oracle/Middleware/my_oep/user_projects/domains/mydomain/myserver/modules
```

Oracle Event Processing loads the libraries in the library directory after the components in the library extensions directory, but before the Oracle Event Processing applications. If your library is a driver, such as a JDBC driver, put the library in the library extensions directory so it activates in the correct order.

13.3.2 Library Extensions Directory

By default, the Oracle Event Processing server library extensions directory is `DOMAIN_DIR/servername/modules/ext`. For example:

```
/Oracle/Middleware/my_oep/user_projects/domains/mydomain/myserver/modules/ext
```

Oracle Event Processing loads the libraries in the library extensions directory first at the same time as the Oracle Event Processing server core modules. Put drivers in the library extensions library to activate them first to override an older driver or to provide access to an alternative driver. If your library is not a driver, put it in the library directory.

For more information, see *Administering Oracle Event Processing*.

13.4 Deployment Order

The Oracle Event Processing server loads components in the following order at Oracle Event Processing server start up time:

1. Load libraries in the library extensions directory.
2. Load libraries in the library directory.
3. Load Oracle Event Processing applications.

The Oracle Event Processing server loads libraries from both the library extensions directory and the library directory based on the lexical order of the library names. Lexical ordering includes the relative directory name plus JAR file name. For example:

- `modules/a.jar` starts before `modules/b.jar`.
- `modules/0/my.jar` starts before `module/my.jar` because `0/my.jar` comes before `my.jar` in lexical order.

This convention enables you to control the order in which the Oracle Event Processing server deploys JAR files by organizing JAR files into appropriately named subdirectories of either the library extensions directory or library directory.

As soon as the application deploys, the Oracle Event Processing server creates an application configuration history, and the configured adapters start to listen for events.

13.5 Configuration History

Configuration changes that you make to Oracle CQL rules or to the Oracle Event Processing high availability adapter configuration are recorded in the history. You can view and roll-back (undo) these changes with the Oracle Event Processing Visualizer or `wlevs.Admin` command-line tool.

You can export the change history to a file and use that file to update your application source with changes made at runtime. For more information, see:

- *Using Visualizer for Oracle Event Processing*
- *Administering Oracle Event Processing*.

13.6 Assemble an OSGi Bundle with appC

The appC utility is a command-line tool that enables you to build, validate, and compile an application. You can create or modify the various files that comprise an Oracle Event Processing project. The appC tool validates against the following schema versions: `spring-wlevs-v12_1_3_0.xsd` and `spring-beans.xsd` and `spring-osgi.xsd`. You must have the JDK installed for this command to work because it depends on the Java compiler.

Note:

The appC tool does not support multiple component configuration files and multiple assembly files. In this case, use the `bundle.sh` script or Oracle JDeveloper.

The appC tool validates the configuration and assembly files and the created OSGi bundle to ensure Oracle CQL validation and that the OSGi bundle deploys successfully.

Note:

Currently, appC performs only schema validations with no extensive semantic validations. Custom adapter providers are not validated.

The `appC.jar` file is located in `/Oracle/Middleware/my_oep/bin`.

Syntax

```
java -jar appC.jar -cmd [cmd option] -basedir <path to workspace> [arguments]
```

cmd options

Choose one of the following command options to execute with the `-cmd` argument.

`buildAll`: Creates the template, generates the manifest, and validates the component configuration file and the assembly file.

`createTemplate`: Creates the workspace file structure with either input XML files or template XML files.

`generateManifest:` Compiles the input source files and generates the OSGi manifest file based on resulting class files.

`validate`: Validates the component configuration and assembly files against their schemas.

arguments

`-cmd`: Execute one of the command options. Specify the root directory with `-basedir` to indicate the root of the project workspace.

`-help`: Prints help information for the command options and arguments to the command line.

`-basedir`: The root of the project workspace that you set up previous to using the appC utility.

`-cp`: The class path to use to compile the bundle classes.

`-config`: The full path to the component configuration file.

`-context`: The full to the assembly file.

`-manifest`: An optional input manifest file. When no manifest file is specified, the appC utility generates one.

Note:

The underlying BND package used in Oracle Event Processing to generate the manifest cannot find reflection usage.

`-name`: The name of the project (OSGi bundle name).

`-destdir`: The full path to the directory for the output JAR file. The default is `basedir`.

`-srcdir`: The full path to the source root directory.

`-excludedirs`: A comma-separated list of directory names to exclude from the source tree. These directories are subdirectories of the source directory root (`srcdir`).

Examples

Compile the input source files and generate the OSGi `MANIFEST.MF` file.

```
java -jar appC.jar -cmd generateManifest -basedir
```

Create the template, generate the `MANIFEST.MF` file, and validate the component configuration file and the assembly file.

```
java -jar appC.jar -cmd buildALL -basedir
```

13.7 Assemble an OSGi Bundle with bundle.sh

Oracle Event Processing provides the `bundler.sh` UNIX shell script that you can use to manually assemble an OSGi bundle that provides services and packages to other bundles. For example, you can deploy a JDBC driver in an OSGi bundle JAR file to make it available to other OSGi applications. For more information, see *Administering Oracle Event Processing*.

Note:

There is no Windows support (no `bundler.cmd`).

The `bundle.sh` shell script reads the source JAR files and creates a target JAR file that includes the content of the source JAR files and a `MANIFEST.MF` file with the appropriate bundle-related entries specified. All Java packages found in the source archive are exported to the target bundle.

With `bundler.sh`, you can also generate a bundle activator. A bundler activator instantiates one or more classes in the JAR file and registers each instantiated object as an OSGi service. This feature enables component bundles to access and manipulate multiple versions of specific factory classes during execution.

The procedure to manually assembly an OSGi bundle consists of the following steps:

- Prepare and Organize the Files
- Create the MANIFEST.MF File
- Include Third-Party JAR Files
- Reference Foreign Stages
- Assemble an OSGi Bundle that Activates.

Note:

See the HelloWorld example source directory for a sample `build.xml` Ant file that performs many of the steps described below.

The `build.xml` file is located in `Oracle/Middleware/my_oep/ oep/ examples/source/applications/helloworld.`

13.7.1 Prepare and Organize the Files

To bundle an application or library into an OSGi bundle manually, you first need to prepare and organize the files to be bundled. For simplicity, this procedure creates a temporary directory that contains the required artifacts, and then jars the contents of this temporary directory. This is a suggested approach, and you are not required to assemble the application this way.

1. Create an empty directory, such as `output`:

```
prompt> mkdir output
```

2. Compile all application Java files into the `output` directory.
3. Create an `output/META-INF/spring` directory.
4. Copy the EPN assembly file that describes the components of your application and how they are connected into the `output/META-INF/spring` directory.
5. Create an `output/META-INF/wlevs` directory.
6. Copy the XML files that configure the components of your application, such as the processors or adapters, into the `output/META-INF/wlevs` directory.
7. Create a `MANIFEST.MF` file that contains information about the bundle.

   See Create the MANIFEST.MF File.
8. If you need to access third-party JAR files from your Oracle Event Processing application, see Include Third-Party JAR Files.
9. Create a JAR file that contains the contents of the `output` directory.

   Be sure to specify the `MANIFEST.MF` file you created in the previous step rather than the default manifest file.

   You can name the JAR file anything you want. In the Oracle Event Processing examples, the name of the JAR file is a combination of Java package name and version, such as:

   ```
   com.bea.wlevs.example.helloworld_1.0.0.0.jar
   ```

   Consider using a similar naming convention to clarify which bundles are deployed to the server.
10. If your application depends on foreign stages, see Reference Foreign Stages.

13.7.2 Create the MANIFEST.MF File

The structure and contents of the `MANIFEST.MF` file is specified by the OSGi Framework. Although the value of many of the headers in the file is specific to your application or business, many of the headers are required by Oracle Event Processing.

In particular, the `MANIFEST.MF` file defines the following:

- Application name: Specified with the `Bundle-Name` header.
- Symbolic application name: Specified with the `Bundle-SymbolicName` header.

  Many of the Oracle Event Processing tools, such as the `wlevs.Admin` utility and JMX subsystem, use the symbolic name of the bundle when referring to the application.
- Application version: Specified with the `Bundle-Version` header.
- Imported packages: Specified with the `Import-Package` header.

  Oracle Event Processing requires that you import the following packages at a minimum:

  ```
  Import-Package:
   com.bea.wlevs.adapter.defaultprovider;version="11.1.1",
  ```

```
 com.bea.wlevs.ede;version="11.1.1",
 com.bea.wlevs.ede.api;version="11.1.1",
 com.bea.wlevs.ede.impl;version="11.1.1",
 org.osgi.framework;version="1.3.0",
 org.springframework.beans.factory;version="2.5.6",
 org.apache.commons.logging;version="1.1.0",
 com.bea.wlevs.spring;version="11.1.1",
 com.bea.wlevs.util;version="11.1.1",
 org.springframework.beans;version="2.5.6",
 org.springframework.util;version="2.0",
 org.springframework.core.annotation;version="2.5.6",
 org.springframework.beans.factory;version="2.5.6",
 org.springframework.beans.factory.config;version="2.5.6",
 org.springframework.osgi.context;version="1.2.0",
 org.springframework.osgi.service;version="1.2.0"
```

- Exported packages: Specified with the `Export-Package` header. You should specify this header only when you need to share one or more application classes with other deployed applications. A typical example is sharing an event bean.

  If possible, you should export packages that include only the interfaces, and not the implementation classes themselves. If other applications use the exported classes, you cannot fully undeploy the application that is exporting the classes.

  Exported packages are server-wide, so be sure their names are unique across the server.

The following is the `MANIFEST.MF` file from the HelloWorld example application.

Note:

Oracle Event Processing requires the following `MANIFEST.MF` setting to deploy to an Oracle WebLogic Server container: `Bundle-ManifestVersion 2`. This is because Oracle Event Processing uses Felix in the Oracle WebLogic Server container.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Version: 1.0.0
Bundle-Vendor: Oracle Corporation
Bundle-Copyright: Copyright (c) 2006-2009 by Oracle.
Export-Package: com.bea.wlevs.event.example.helloworld;version="12.1.2",
 com.bea.wlevs.example.helloworld;version="12.1.2"
Import-Package: com.bea.wlevs.configuration;version="12.1.2"
  com.bea.wlevs.ede.api;version="12.1.2",
  com.bea.wlevs.ede.impl;version="12.1.2",
  com.bea.wlevs.ede.spi;version="12.1.2",
  com.bea.wlevs.ede;version="12.1.2",
  com.bea.wlevs.management.spi;version="12.1.2",
  com.bea.wlevs.spring.support;version="12.1.2",
  com.bea.wlevs.spring;version="12.1.2",
  com.bea.wlevs.util;version="12.1.2",
  org.apache.commons.logging;version="1.1.0",
  org.springframework.beans.factory.config;version="2.5.6",
  org.springframework.beans.factory;version="2.5.6",
  org.springframework.beans;version="2.5.6",
  org.springframework.core.annotation;version="2.5.6",
  org.springframework.osgi.context;version="1.2.0",
  org.springframework.osgi.extensions.annotation;version="1.2.0",
  org.springframework.osgi.service;version="1.2.0",
  org.springframework.util;version="2.5.6"
Bundle-Name: example.helloworld
```

```
Bundle-Description: OCEP example helloworld
Bundle-SymbolicName: helloworld
```

13.7.3 Include Third-Party JAR Files

When you create your Oracle Event Processing applications, you might need to access legacy libraries within existing third-party JAR files. You can ensure access to this legacy code with any of the following approaches:

- Bundle-Classpath
- Operating System Path
- -Xbootclasspath

Bundle-Classpath

The recommended approach is to package the third-party JAR files in your Oracle Event Processing application JAR file. You can put the JAR files anywhere you want.

Note:

This approach gives you little control over the order in which JAR files are loaded and it is possible that dependency conflicts may occur. For this reason, Oracle recommends that you use the Oracle Event Processing server application library approach instead.

To ensure that your Oracle Event Processing application finds the classes in the third-party JAR file, you must update the application class path by adding the `Bundle-Classpath` header to the `MANIFEST.MF` file. Set `Bundle-Classpath` to a comma-separate list of the JAR file path names that should be searched for classes and resources. Use a period (`.`) to specify the bundle itself. For example:

```
Bundle-Classpath: ., commons-logging.jar, myExcitingJar.jar, myOtherExcitingJar.jar
```

If you need to access native libraries, you must also package them in your JAR file and use the `Bundle-NativeCode` header of the `MANIFEST.MF` file to specify their location in the JAR.

Operating System Path

You can create an application library that depends on native code libraries that you do not choose to package as application libraries. In this case, you can put the native code libraries in the operating system path (`bootclasspath`) of the Oracle Event Processing server. When the server starts, the library bundles that need to call this native code load the native code libraries. For more information, see *Administering Oracle Event Processing*.

-Xbootclasspath

If the JAR files include libraries used by all applications deployed to Oracle Event Processing, such as JDBC drivers, you can add the JAR file to the server's boot class path by specifying the `-Xbootclasspath/a` option to the `java` command in the scripts used to start up an instance of the server.

Note:

This approach gives you little control over the order in which JAR files are loaded and dependency conflicts can occur. Oracle recommends that you use the Oracle Event Processing server application library approach instead. For more information, see Operating System Path approach instead.

The name of the server start script is `startwlevs.cmd` (Windows) or `startwlevs.sh` (UNIX), and the script is located in the server directory of your domain directory. The out-of-the-box sample domains are located in *ORACLE_CEP_HOME*`/ocep_11.1/samples/domains`, and the user domains are located in *ORACLE_CEP_HOME*`/user_projects/domains`, where *ORACLE_CEP_HOME* refers to the main Oracle Event Processing installation directory, such as `d:\oracle_cep`.

13.7.4 Access Third-Party JAR Files with -Xbootclasspath

Update the start script by adding the `-Xbootclasspath/a` option to the `java` command that executes the `wlevs_2.0.jar` file. Set the `-Xbootclasspath/a` option to the full path name of the third-party JAR files you want to access system-wide.

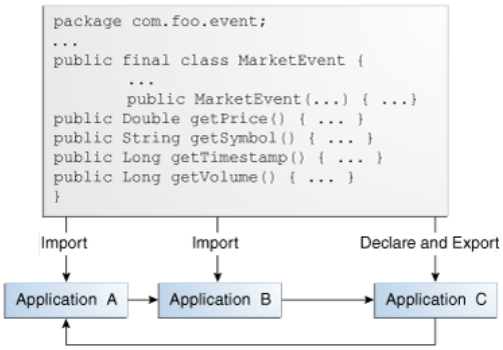
For example, if you want all deployed applications to be able to access a JAR file called `e:\jars\myExcitingJAR.jar`, update the `java` command in the start script as follows. The updated section is shown in bold (in practice, the command should be on one line):

```
%JAVA_HOME%\bin\java -Dwlevs.home=%USER_INSTALL_DIR% -Dbea.home=%BEA_HOME%
    -Xbootclasspath/a:e:\jars\myExcitingJAR.jar
    -jar "%USER_INSTALL_DIR%\bin\wlevs_2.0.jar" -disablesecurity %1 %2 %3 %4 %5 %6
```

13.7.5 Reference Foreign Stages

You can refer to a stage that is in another Oracle Event Processing application. A stage from another application is called a foreign stage. When you assemble applications that depend on foreign stages, be aware of class path dependencies. Consider the application dependency graph that Figure 13-2 shows.

Figure 13-2 Foreign Stage Dependency Graph



In this example, Application A depends on Application B, Application B depends on Application C, and Application C depends on Application A. Application C declares and exports the `MarketEvent` class. Applications A and B import the `MarketEvent`

class that Application C provides. In this example the `MANIFEST.MF` files of the OSGi bundles A and B should contain `Require-Bundle: C`.

Note the following:

- When you redeploy a foreign stage, you must redeploy all foreign stages that depend on that application or foreign stage.

  For example, if you redeploy Application B, you must redeploy Application A.

- If there is a class path dependency between one foreign stage and another, when you deploy the foreign stage that declares and exports the shared class, you must redeploy all foreign stages that import the shared class.

  For example, if you redeploy Application C, you must also redeploy Application A and B because Application A and B have a class path dependency on Application C (`MarketEvent`).

13.7.6 Assemble an OSGi Bundle that Activates

Once you prepare and organize the files, you can use the `bundle.sh` shell script to assemble the files into an OSGi bundle and define the activator classes. You can find the `bundle.sh` script in the */Oracle/Middleware/my_oep/*`bin` directory.

If want to bundle an application library for a new JDBC driver, see *Administering Oracle Event Processing*.

13.7.6.1 Command Location, Syntax, and Arguments

You can find the `bundler.sh` script in the */Oracle/Middleware/my_oep/*`bin` directory. The following shows the bundler.sh command syntax. Table 13-1 describes the command arguments.

```
bundler -source JAR -name NAME -version VERSION
[-factory CLASS+] [-service INTERFACE+] [-fragmenthost HOST]
[-stagedir PATH] [-targetdir PATH]
[+import PACKAGE|REGEX+] [-imods REGEX;MODS+] [-import PACKAGE+]
[+export PACKAGE|REGEX+] [-emods REGEX;MODS+]
[-dimport PACKAGE+] [-explode] [-verbose]
```

Table 13-1 bundler.sh Command-Line Options

Argument	Description
`-source` *JAR*	The path of the source JAR file to be bundled.
`-name` *NAME*	The symbolic name of the bundle. The root of the target JAR file name is derived from the name value.
`-version` *VERSION*	The bundle version number. All exported packages are qualified with a version attribute with this value. The target JAR file name contains the version number.
`-factory` *CLASS+*	An optional argument that specifies a space-delimited list of one or more factory classes that are to be instantiated and registered as OSGi services. Each service is registered with the OSGi service registry with name (`-name`) and version (`-version`) properties. This argument is incompatible with the `-fragmenthost` argument.

Table 13-1 (Cont.) bundler.sh Command-Line Options

Argument	Description
`-service` *`INTERFACE`*+	An optional argument that specifies a space-delimited list of one or more Java interfaces that are used as the object class of each factory object service registration. If no interface names are specified, or the number of interfaces specified does not match the number of factory classes, then each factory object will be registered under the factory class name.
`-fragmenthost` *`HOST`*	An optional argument indicating that the resultant bundle is a fragment bundle and specifies the symbolic name of the host bundle. This argument is incompatible with the `-factory` argument.
`-stagedir` *`PATH`*	An optional argument that specifies where to write temporary files when creating the target JAR file. Default: `./bundler.tmp`
`-targetdir` *`PATH`*	An optional argument that specifies the location of the generated bundle JAR file. Default: current working directory (`.`).
`+import` *`PACKAGE`*| *`REGEX`*+	A space-delimited list of one or more packages or regular expressions that select the packages to *exclude* from the manifest `Import-Package` attribute. By default, all dependent packages will be imported (except `java.*`).
`-imods` *`REGEX;MODS`*+	The import modifiers are applied to the packages matching regular expression.
`-import` *`PACKAGE`*	Additional packages to include on the manifest `Import-Package` attribute. Note that any specified import modifiers will not be applied.
`+export` *`PACKAGE`*| *`REGEX`*+	A space-delimited list of one or more packages or regular expressions that select the packages to *exclude* from the manifest `Export-Package` attribute. By default, all bundle packages will be exported.
`-emods` *`REGEX;MODS`*+	The export modifiers will be applied to the packages matching regular expression.
`-dimport` *`PACKAGE`*+	Packages to include on the manifest `DynamicImport-Package` attribute.
`-explode`	This optional flag specifies that the content of the source JAR should be exploded into the target JAR file. By default, the source JAR is nested within the target JAR file and the generated bundle manifest will contain an appropriate `Bundle-Classpath` attribute.
`-verbose`	An optional flag to enable verbose output.

13.7.6.2 Assemble an OSGi Bundle

1. Execute the `bundler.sh` script to create an OSGi bundle. See Command Location, Syntax, and Arguments.

   The following `bundle.sh` command shows how to use the `bundler.sh` to create an OSGi bundle for an Oracle JDBC driver.

```
bundler.sh \
   -source C:\drivers\com.oracle.ojdbc14_11.2.0.jar \
   -name oracle12c \
   -version 12.1.3.0 \
   -factory oracle.jdbc.xa.client.OracleXADataSource oracle.jdbc.OracleDriver \
   -service javax.sql.XADataSource java.sql.Driver \
   -targetdir C:\stage
```

   The `-source` option specifies a JAR file that is an Oracle driver located in directory `C:\drivers`. The name of the generated bundle JAR is the concatenation of the `-name` and `-version` arguments (`oracle10g_11.2.0.jar`) and is created in the `C:\stage` directory. The bundle JAR contains the following files:

```
   1465 Thu Jun 29 17:54:04 EDT 2006 META-INF/MANIFEST.MF
1540457 Thu May 11 00:37:46 EDT 2006 com.oracle.ojdbc14_11.2.0.jar
   1700 Thu Jun 29 17:54:04 EDT 2006 com/bea/core/tools/bundler/Activator.class
```

   The `-factory` option specifies that there are two factory classes to be instantiated and registered as an OSGi service when the bundle activates, each under a separate object class as Table 13-2 shows.

Table 13-2 Factory Class and Service Interfaces

Factory Class	Service Interface
`oracle.jdbc.xa.client.OracleXADataSource`	`javax.sql.XADataSource`
`oracle.jdbc.OracleDriver`	`java.sql.Driver`

   The `-service` option registers services with a `name` property set to `oracle12c` and a `version` property with a value of `12.1.3.0`. The following example shows the Oracle Event Processing server log messages with the service registrations:

```
...
INFO: [Jun 29, 2006 5:54:18 PM] Service REGISTERED: { version=12.1.3.0, name=oracle12c,
objectClass=[ javax.sql.XADataSource ], service.id=23 }
INFO: [Jun 29, 2006 5:54:18 PM] Service REGISTERED: { version=12.1.3.0, name=oracle12c,
objectClass=[ java.sql.Driver ], service.id=24 }
INFO: [Jun 29, 2006 5:54:18 PM] Bundle oracle11g STARTED
...
```

2. Copy the application library JAR to the appropriate Oracle Event Processing server application library directory:

   a. If your bundle is a driver, you put it in the library extensions directory.

      See Library Extensions Directory.

   b. If your bundle is not a driver, you can put it in the library directory.

      See Library Directory

3. Stop and start the Oracle Event Processing server.

   See *Administering Oracle Event Processing*.

13.8 Deploy an OSGi Bundle

After you assemble your Oracle Event Processing application or library into an OSGi bundle, you deploy it to an Oracle Event Processing server domain. You can deploy an application with Oracle JDeveloper, Oracle Event Processing Visualizer, and with the Deployer utility. This section explains how to use the Deployer utility.

With the Deployer utility, you can deploy an application to either a stand-alone or multiserver domain. You can only deploy to a group when the server is part of a multiserver domain (clustering is enabled). You cannot deploy to a group when the server is part of a standalone server domain (clustering is disabled).

Oracle Event Processing uses the `deployments.xml` file to internally maintain a list of deployed OSGi bundles. This file is located in the *`DOMAIN_DIR/servername`* directory, where *`DOMAIN_DIR`* refers to the main domain directory corresponding to the server instance to which you are deploying your application and *`servername`* refers to the actual server. See *Schema Reference for Oracle Event Processing* for information about this file. The information is provided for your information only; Oracle does not recommend updating the `deployments.xml` file manually.

Before you Begin

Be sure you have configured Jetty for the Oracle Event Processing instance to which you are deploying your application. For more information, see *Administering Oracle Event Processing*.

Open a command window and update your `CLASSPATH` variable to include the `wlevsdeploy.jar JAR` file, which is in the following directory. The Deployer utility is in the JAR file.

```
/Oracle/Middleware/my_oep/bin
```

Deploy an OSGi Bundle with the Deployer Utility

After the OSGi bundle successfully installs and all initialization tasks complete, Oracle Event Processing starts the application and the adapter components listen for incoming events.

1. Assemble your OSGi bundle as described in Assemble an OSGi Bundle with bundle.sh.

2. Open a command window and run the Deployer utility as follows. Keep everything on one line.

   ```
   prompt> java -jar wlevsdeploy.jar -url http://host:port/wlevsdeployer
       -user user -password password -install application_jar_file
   ```

   `host`: The name of the computer where the Oracle Event Processing server is running.

   `port`: The port number where Oracle Event Processing listens. The default value is `9002`. This port is specified in the `DOMAIN_DIR/config/` file that describes the Oracle Event Processing domain. The port number is the value of the `<Port>` child element of the `<Netio>` element in the file:

   ```
   <Netio>
     <Name>NetIO</Name>
   ```

```
  <Port>9002</Port>
</Netio>
```

user: The user name of the Oracle Event Processing administrator.

password: The password of the Oracle Event Processing administrator.

application_jar_file: The OSGi bundle. The OSGi bundle must be located on the same computer where you execute the Deployer utility. For example, if Oracle Event Processing is running on host `ariel`, listening on port `9002`, user name and password of the administrator is `wlevs/wlevs`, and your application JAR file is called `myapp_1.0.0.0.jar` and is located in the `/applications` directory, then the command is the following. Keep everything on one line.

```
prompt> java -jar wlevsdeploy.jar -url http://ariel:9002/wlevsdeployer
        -user wlevs -password wlevs -install /applications/myapp_1.0.0.0.jar
```

The Deployer utility provides additional options over what was described here to resume, suspend, update, uninstall and deploy an OSGi bundle to a specified group of multiserver domains. For more information, see *Administering Oracle Event Processing*.

14

Testing 1-2-3

Oracle Event Processing provides different ways to test your application depending on what and how you want to test.

This chapter includes the following sections:

- Load Generator and the csvgen Adapter
- Event Inspector Service
- EPN Shell
- EPN Command Interface.

14.1 Load Generator and the csvgen Adapter

The load generator utility simulates a data feed so that you can test your application without connecting to a real-world data feed. To test your application with the load generator, you must use the csvgen adapter in your application because the csvgen adapter is specifically coded to decipher the data packets generated by the load generator. Once you have tested your application with the csvgen adapter, you can replaced it with the appropriate input adapter for your application as described in Adapters.

The load generator reads an ASCII file that contains the sample data feed information and sends each line of data in order to a port. The csvgen adapter listens for data at the same port.

A load generator properties file indicates the name of the data file, the port where the server listens, the server host, and the packet type. You can also set the data rate and how long it takes for the load generator to ramp up to a specified final rate.

The following steps present an overview of how to configure and run the load generator utility. Detailed information for each step follows.

1. Create a properties file that contains the configuration properties for particular run of the load generator. Oracle Event Processing provides a default property file you can use if the default property values are adequate. See Create the Properties File.

2. Create a file that contains the actual data feed values. See Create the Data Feed File.

3. Configure the csvgen adapter so that it correctly reads the data feed generated by the load generator. See Configure the csvgen Adapter in Your Application.

4. Run the load generator and specify the properties file you created in step 1 to begin the simulated data feed. For example, if the name of your properties file is `c:\loadgen\myDataFeed.prop`, execute the following command:

```
prompt> runloadgen.cmd c:\loadgen\myDataFeed.prop
```

If you redeploy your application, you must also restart the load generator.

5. To stop and load generator, go to directory where you have load generator running and type `ctrl-c`.

14.1.1 Create the Properties File

Oracle Event Processing provides a default properties file called `csvgen.prop` in the */Oracle/Middleware/my_oep/*`utils/load-generator` directory. The format of the file is simple: each property-value pair is on its own line.

The following example shows the default `csvgen.prop` file; Oracle recommends you use this file as a template for your own property file:

```
# name of file containing your test data
  test.csvDataFile=test.csv
# port the server will listen on for client connection
  test.port=9001
# server host (localhost if not specified)
# test.host=
# do not change the packetType

  test.packetType=CSV
```

Table 14-1 Load Generator Properties

Property	Description	Data Type	Required ?
`test.csvDataFile`	Specifies the file that contains the data feed values.	`String`	Yes
`test.port`	The port number to which the load generator sends the data feed. Each input adapter must be associated with its own `test.port`.	`Integer`	Yes
`test.secs`	Total duration of the load generator run, in seconds. The default value is 30.	`Integer`	No
`test.rate`	Final data rate, in messages per second. The default value is 1.	`Integer`	No
`test.startRate`	Initial data rate, in messages per second. The default value is 1.	`Integer`	No
`test.rampUpSecs`	Number of seconds to ramp up from `test.startRate` to `test.rate`. The default value is 0.	`Integer`	No

14.1.2 Create the Data Feed File

A load generator data feed file contains the sample data feed values that correspond to the event type registered for your Oracle Event Processing application.

The following example show an `EmployeeEvent` and a load generator data feed file corresponding to this event type.

```
<wlevs:event-type-repository>
        <wlevs:event-type type-name="EmployeeEvent">
            <wlevs:properties>
                <wlevs:property name="name" type="char" />
                <wlevs:property name="age" type="int" />
                <wlevs:property name="birthplace" type="char" length="512" />
        </wlevs:properties>
    </wlevs:event-type>
...
</wlevs:event-type-repository>
```

```
Lucy,23,Madagascar
Nick,44,Canada
Amanda,12,Malaysia
Juliet,43,Spain
Horatio,80,Argentina
```

A load generator data feed file follows a simple format:

- Put each data feed item is on its own line.
- Separate the fields of a data feed item with commas.
- Do not include commas in a string field.
- Do not include extraneous spaces before or after the commas, unless the space is literally part of the field value.
- Include only string and numerical data in a data feed file such as integer, long, double, and float.
- Keep within the maximum string length of 256 characters or specify a long string length. To specify a longer string, set the `length` attribute of the `char` property in your event-type for the `birthplace` property.

Note:

The load generator does not fully comply with the CSV specification at `http://www.creativyst.com/Doc/Articles/CSV/CSV01.htm`

For more information about CSV adapter constraints, see Design Constraints.

14.1.3 Configure the csvgen Adapter in Your Application

When using the load generator utility, you must use the `csvgen` adapter in your application because this Oracle Event Processing-provided adapter is specifically coded to read the data packets generated by the load generator.

You register the `csvgen` adapter using the `wlevs:adapter` element in the EPN assembly file of your application, as with all adapters. Set the `provide` attribute to `csvgen` to specify that the provider is the `csvgen` adapter, rather than your own adapter. Additionally, you must specify the following child tags:

- `wlevs:instance-property` element with `name` attribute `port` and `value` attribute *`configured_port`*, where *`configured_port`* corresponds to the value of the `test.port` property in the load generator property file. See Create the Properties File.

- `wlevs:instance-property` element with `name` attribute `eventTypeName` and `value` attribute *`event_type_name`*, where *`event_type_name`* corresponds to the name of the event type that represents an item from the load-generated feed.
- `wlevs:instance-property` element with `name` attribute `eventPropertyNames` and `value` attribute *`ordered_list_of_properties`*, where *`ordered_list_of_properties`* lists the names of the properties in the order that the load generator sends them, and consequently the `csvgen` adapter receives them.

Before showing an example of how to configure the adapter, first assume that your application registers an event type called `PersonType` in the EPN assembly file using the `wlevs:metada` element shown below:

```
<wlevs:event-type-repository>
        <wlevs:event-type type-name="PersonType">
            <wlevs:properties>
               <wlevs:property name="name" type="char"/>
               <<wlevs:property name="age" type="int"/>
               <<wlevs:property name="birthplace" type="char"/>
            </wlevs:properties>
        </wlevs:event-type>
</wlevs:event-type-repository>
```

This event type corresponds to the data feed file shown in Create the Data Feed File.

To configure the `csvgen` adapter that receives this data, use the following `wlevs:adapter` element:

```
<wlevs:adapter id="csvgenAdapter" provider="csvgen">
   <wlevs:instance-property name="port" value="9001"/>
   <wlevs:instance-property name="eventTypeName" value="PersonType"/>
   <wlevs:instance-property name="eventPropertyNames" value="name,age,birthplace"/>
</wlevs:adapter>
```

Note how the bold values in the adapter configuration example correspond to the `PersonType` event type registration.

If you use the `wlevs:class` element to specify your own JavaBean when registering the event type, then the `eventPropertyNames` value corresponds to the JavaBean properties. For example, if your JavaBean has a `getName` method, then one of the properties of your JavaBean is `name`.

For more information on event types, see Events and Event Types.

14.2 Event Inspector Service

Use the Event Inspector service to test and debug Oracle CQL queries during development. With the Event Inspector service you can view (trace) the events that flow out of any EPN stage and inject events into any EPN stage. The Event Inspector service uses a common HTTP pub-sub channel and server to trace and inject events.

Note:

Do not use the Event Inspector service on a production Oracle Event Processing server. Use this service during development only.

A trace event must have its `binding` attribute set to `outbound`, and an injected event must have its `binding` attribute set to `inbound`. Using an Event Inspector client, you can inject:

- A single, simple event by type, such as the `StockTick` event. The specific event property types that you can use depends on the client.
- A single event directly to the HTTP pub-sub channel as a JSON-formatted character string. You can use any event property that JSON can represent.
- Multiple events using a file that contains one or more JSON-formatted character strings. You can use any event property that JSON can represent. The Event Inspector service client parses the file and injects all of its JSON strings to the HTTP pub-sub channel.

You can use the GSON Java library to help you convert Java objects to JSON format when creating your input file. For more information, see:

- `http://www.json.org/`
- `http://code.google.com/p/google-gson`

The Event Inspector service supports Oracle Event Processing Visualizer. See *Using Visualizer for Oracle Event Processing*.

14.2.1 Event Types

The Event Inspector service supports all Oracle Event Processing event types: JavaBean class, `Map`, and tuple. The Event Inspector service converts events to the JavaScript Object Notation (JSON) format before publishing to the trace channel. You must inject events in JSON format.

Note:

Byte arrays are not supported as property types in event types used with the event inspector.

JSON-formatted events must conform to the structure. Table 14-1 lists the required attributes.

```
{
    "event-type": "myEventType",
    "operation": "insert",
    "binding": "outbound",
    "value":{
        "firstname": "Jane",
        "lastname": "Doe",
        "phone": {
            "code": 12345,
            "number": "office"
        },
    }
}
```

Table 14-2 Event Inspector JSON Event Required Attributes

Attribute	Description
`event-type`	The name of the Oracle Event Processing event as you defined it in the application assembly file's `event-type-repository`.

Table 14-2 (Cont.) Event Inspector JSON Event Required Attributes

Attribute	Description
`operation`	Specify the type of event: • `insert`: insert event. • `delete`: delete event • `update`: update event • `heartbeat`: heartbeat event
`binding`	One of: • `inbound`: injected event. • `outbound`: trace event.
`value`	One or more JSON-formatted event properties as defined by the `event-type`.

14.2.2 HTTP Publish-Subscribe Channel and Server

The Event Inspector service uses a dynamic HTTP publish-subscribe (HTTP pub-sub) channel with the following name that is defined in the server `config.xml` file:

/SERVERNAME/APPLICATIONNAME/STAGENAME/DIRECTION

SERVERNAME: The name of the Oracle Event Processing server where the EPN stage runs.

APPLICATIONNAME: the name of the Oracle Event Processing application.

STAGENAME: the name of the EPN stage.

DIRECTION: one of either:

- `input`: Event injection.
- `output`: Event tracing.

For example:

```
/server-1/myapp/MyInputAdapter/input
```

The Event Inspector service uses an HTTP pub-sub server. This can be any of:

- **Local:** You configure the server file with an `event-inspector` element and configure its `pubsub-server-name` child element with the name of the local pubsub server that is running on this machine.
- **Remote:** You configure the server file with an `event-inspector` element and configure its `pubsub-server-url` child element with a URL to an HTTP pub-sub server that is running on a remote machine.
- **Default:** if there is only one HTTP pub-sub server defined in the server file, and you do not specify a local or remote HTTP pub-sub server, the Event Inspector service uses the local HTTP pub-sub server by default.

The Event Inspector service uses the same HTTP pub-sub channel and server for tracing and injecting events.

14.2.3 Configure a Local or Remote Server

You can configure the Event Inspector service with a local or remote HTTP pub-sub server. You configure the Event Inspector HTTP pub-sub server in a component configuration file. When there is only one HTTP pub-sub server defined in the server, and you do not specify a local or remote HTTP pub-sub server, the Event Inspector service uses the local HTTP pub-sub server by default.

Configure for a Local HTTP Publish-Subscribe Server

For any component configuration file that has a component that you want to test, add the `event-inspector` `name` element as follows.

```
<event-inspector>
    <name>myEventInspectorConfig</name>
    <pubsub-server-name>myPubSub</pubsub-server-name>
</event-inspector>
```

The `pubsub-server-name` value `myPubSub` is the value of the `http-pubsub` element `name` child element as defined in the local Oracle Event Processing server file as the following example shows.

```
<http-pubsub>
  <name>myPubSub</name>
  <path>/pubsub</path>
  <pub-sub-bean>
    <server-config>
      <supported-transport>
        <types>
          <element>long-polling</element>
        </types>
      </supported-transport>
      <publish-without-connect-allowed>true</publish-without-connect-allowed>
    </server-config>
    <channels>
        ...
    </channels>
  </pub-sub-bean>
</http-pubsub>
```

Configure for a Remote HTTP Publish-Subscribe Server

For any component configuration file that has a component that you want to test, add the `event-inspector` `name` element as follows.

```
<event-inspector>
  <name>myEventInspectorTraceConfig</name>
  <pubsub-server-url>http://HOST:PORT/PATH</pubsub-server-url>
</event-inspector>
```

`HOST`: The host name or IP address of the remote Oracle Event Processing server.

`PORT`: The remote Oracle Event Processing server `netio` port as defined in the remote Oracle Event Processing server file. Default: 9002.

`PATH`: The value of the `http-pubsub` element `path` child element as defined in the remote Oracle Event Processing server file.

Given the `http-pubsub` configuration that the example shows, a valid `pubsub-server-url` would be as follows:

```
http://remotehost:9002/pubsub
```

The `pubsub-server-name` value `myPubSub` is the value of the `http-pubsub` element name child element as defined in the local Oracle Event Processing server file as the following example shows.

```
<http-pubsub>
  <name>myPubSub</name>
  <path>/pubsub</path>
  <pub-sub-bean>
    <server-config>
      <supported-transport>
        <types>
          <element>long-polling</element>
        </types>
      </supported-transport>
      <publish-without-connect-allowed>true</publish-without-connect-allowed>
    </server-config>
    <channels>
          ...
    </channels>
  </pub-sub-bean>
</http-pubsub>
```

14.2.4 Inject Events

After you configure the Event Inspector service HTTP pub-sub server, you can use Event Inspector clients to inject events. To configure event injection, you can use the Oracle Event Processing Visualizer, or you can edit a component configuration file in your application to specify injection settings that are in place when the application is deployed or redeployed.

Configure event injection in Oracle Event Processing Visualizer with settings that can be discarded when the application is redeployed. See *Using Visualizer for Oracle Event Processing*.

For event injection configuration settings that are in place when the application is deployed or redeployed, configure injection by editing component configuration settings for the stage to which you want to inject.

For example, the component configuration excerpt shown in the example illustrates how you might configure a processor for event injection. The `inject-parameters` element's `active` child element specifies that injection is on, while the `channel-name` element specifies the HTTP pub-sub channel from which injected elements should be sent.

```
<processor>
    <name>FindCrossRates</name>
    <inject-parameters>
        <active>true</active>
        <channel-name>/NonClusteredServer/fx/FindCrossRates/output</channel-name>
    </inject-parameters>
    <rules>
        <!-- Query rules omitted. -->
    </rules>
</processor>
```

For reference information about the elements, see *Schema Reference for Oracle Event Processing*.

14.2.5 Trace Events

After you configure the Event Inspector service HTTP pub-sub server, you can use Event Inspector clients to trace events flowing out of any stage of your EPN. To trace

events, you can either use the Oracle Event Processing Visualizer to configure tracing or you can edit a component configuration file in your application to specify trace settings that are in place when the application is deployed or redeployed.

Configure event tracing in Oracle Event Processing Visualizer with settings that can be discarded when the application is redeployed. See *Using Visualizer for Oracle Event Processing*.

For event tracing configuration settings that are in place when the application is deployed or redeployed, configure tracing by editing component configuration settings for the stage from which you want to trace.

For example, the component configuration excerpt shown in the example illustrates how you might configure a processor for event tracing. The `trace-parameters` element's active child element specifies that tracing is on, while the channel-name element specifies the HTTP pub-sub channel to which traced elements should be sent.

```
<processor>
    <name>FindCrossRates</name>
    <trace-parameters>
        <active>true</active>
        <channel-name>/NonClusteredServer/fx/FindCrossRates/output</channel-name>
    </trace-parameters>
    <rules>
        <!-- Query rules omitted. -->
    </rules>
</processor>
```

For reference information about the elements, see *Schema Reference for Oracle Event Processing*.

14.2.6 Event Inspector API

You can use the Event Inspector API to inject and trace events. The Oracle Event Processing API provides the `EventInspectorMBean` interface that you can implement to control the event tracing and injection behavior. Only the administrator and monitor administrator roles can invoke the `startInject`, `stopInject`, `startTrace`, and `stopTrace` methods. See *Java API Reference for Oracle Event Processing* for information about the `EventInspectorMBean` interface and its methods.

Once you implement the `EventInspectorMBean` interface, you can call its methods from your application to inject and trace events as follows.

Inject Events

1. Get an instance of `com.bea.wlevs.eventinspector.management.EventInspectorMBean` from the server through JMX.
2. Call the `EventInspectorMBean.startInject` method to start event injection on the stage specified in the server `config.xml` file configuration for an HTTP publish-Subscribe channel as explained in HTTP Publish-Subscribe Channel and Server.
3. Publish events to the specified HTTP publish-subscribe channel.
4. Use the `EventInspectorMBean.isInjecting` method to verify that event injection has started on the stage.

5. Call the `EventInspectorMBean.stopInject` method to stop event injection.

Trace Events

1. Get an instance of `com.bea.wlevs.eventinspector.management.EventInspectorMBean` from the server through JMX.
2. Call the EventInspectorMBean.startTrace method to start event tracing on the stage specified in the server config.xml file configuration for an HTTP publish-subscribe channel as explained in HTTP Publish-Subscribe Channel and Server.
3. Use the `EventInspectorMBean`.isTracing method to verify that event tracing has started on the HTTP Publish-Subscribe channel.
4. Call the `EventInspectorMBean.stopTrace` method to stop event tracing.

14.3 EPN Shell

The EPN shell provides shell commands for testing Oracle Event Processing applications. See EPN Command Interface for information about how to perform EPN operations programmatically.

The EPN shell extends the Apache Felix Gogo shell, which provides a standard shell command prompt for OSGi frameworks. See the Apache Felix Gogo documentation at: `http://felix.apache.org/documentation/subprojects/apache-felix-gogo.html`.

To start the server in the EPN shell, go to `/Oracle/Middleware/my_oep/user_projects/domains/<my_domain>/<server>` and start the Oracle Event Processing server in the EPN shell as follows:

UNIX:

```
./startwlevs.sh -shell
...
g!
```

Windows:

```
startwlevs.cmd -shell
...
g!
```

When the EPN shell starts, you see the `g!` prompt. Type `help` to display a list of all of the commands. The commands specific to EPN, are prefixed by `epn:`, for example, `epn:channel`.

```
g! help
```

To display help about a particular command, type `help <command>`. For example, to see help about the `begin` command, type the following:

```
g! help begin
begin - Begins new session for invoking EPN commands
   scope: epn
   parameters:
      CommandSession
```

Note:

Some of the Apache Felix Gogo commands have a `CommandSession` argument, which is an internal shell argument that does not execute with the Apache Felix Gogo commands. When you do a help listing for some of the EPN shell commands, the `CommandSession` parameter is listed by the help, but does not work.

14.3.1 Oracle CQL Queries

By default, an EPN session has an implicit Oracle CQL processor that is connected to an event sink that prints all outputs to the shell console. To test Oracle CQL queries, create an input channel, define the queries, and send events as follows:

```
g! begin
g! channel MyChannel [msg=String]
MyChannel
g! query "select * from MyChannel"
q0
g! send MyChannel [msg='Hi']
11:14:26 618 -> insert event: {msg=Hi}
q! end
```

Use the `begin` command to start an EPN session and the `end` command to end an EPN session. Ending a session destroys all EPN components that were created during that session. A session is not multithread aware.

The `channel` command creates `MyChannel` with an `event-type` that has the `msg` property of type `String`. You can also specify a Java class name for the `event-type`.

The `query` command registers the `"select * query from MyChannel“` in the implicit Oracle CQL processor for this session. You can remove the query from the processor with the `remove` command.

The `send` command dispatches an `insert` event to `MyChannel`. If the session has a single channel only, then the name of the channel is optional. For example, the following two commands are equivalent: `“send MyChannel [msg='Hi']"` and `“send [msg='Hi']"`.

The syntax `“[msg='Hi']"` creates a map that contains a single key-value pair, with a key of `“msg"` and a value of `“Hi"`. This matches the `event-type` created in the first line with `channel MyChannel [msg=String]`.

You can use the `update` and `delete` commands to send an update event and a delete event to a relation-based channel. The `insert` command is equivalent to the `send` command. Also, you can check the current registered statements in a session with the `statement` command and find out the channels you have created in a session with the `channels` command. The `eventtypes` commands enables you to display the structure of all `event-types` in the server.

For more information about Oracle CQL processor queries, see Oracle CQL Processors .

14.3.2 Management Commands

Management commands enable you to list the deployed Oracle Event Processing applications, list libraries, install an application, send events to an existing application, subscribe to channel events, list all public stages, retrieve an OSGi service and call standard Java methods, perform JMX operations, and shut down the server.

List all deployed Oracle Event Processing applications in the running server:

```
g! listapps
com.bea.wlevs.dataservices
```

List all Oracle Event Processing libraries. This commen lists all of the bundles that are unzipped in the modules directory to run the EPN shell.

```
g! llistlibs
org.apache.felix.bundlerepository
org.apache.felix.gogo.runtime
com.oracle.cep.shell
org.apache.felix.gogo.command
org.apache.felix.gogo.shell
```

Install a new application. The following example deploys and immediately starts the application.

```
g! deployapp file:///Users./oepapps/hwlloworld.jar
```

Undeploy an application:

```
g! deployapp file:///Users./oepapps/hwlloworld.jar
g! undeployapp file:///Users./oepapps/hwlloworld.jar
```

Send events to an existing Oracle Event Processing application by specifying the full name of the channel when you use the `send`, `insert`, `update`, and `delete` commands. This example sends an event to the `helloworldInputChannel` in the `helloworld` application. For this to work, you must advertise the channel so the EPN shell can find it. To `advertise` the channel, set the `advertise` attribute to `true` in the assembly file first.

Assembly file setting:

```
<wlevs:channel id="helloworldInputChannel" event-type="HelloWorldEvent"
advertise="true" >
```

EPN shell commands:

```
g! event1=createevent HelloWorldEvent
g! $event1 message "Hi Shell"
g! send helloworld:helloworldInputChannel $event1
```

Subscribe to a channel. The events subscribed to are sent either to the shell console or to a file when you specify a file name. Make sure the channel is advertised by setting the `advertise` attribute to `true` in the assembly file. The following command subscribes to all output from the helloworld application and send the output to the shell console:

```
g! subscribe helloworld:helloworldOutputChannel
```

List all public (advertised) stages in the application.

```
g!introspect helloworld
Application 'helloworld' provides the following OCEP services:
Event Channel 'helloworldOutputChannel' for Event Type 'HelloWorldEvent'
```

Retrieve the `CQLProcessorMBean` for the `helloworld` Oracle CQL processor and invoke the `getAllQueries` operation.

```
g! proc = mbean helloworld:helloworldProcessor CQLProcessor
g! $proc allQueries
```

```
helloworldRule
select * from helloworldInputChannel
```

You can use the retrieved mbean with the `mbean` command to test and manipulate JMX operations.

Stop the server.

```
g! stop
```

14.3.3 Regression Testing

You can automate application testing by running the shell headless and using scripts. For example, you can execute the following script named `send-event.oep` by specifying the `gosh.args` system property in the `startwlevs` command.

```
begin
channel -a [a=Long]
query "select * from ch0"
send 0 [a=1]
send 1 [a=2]
end
```

Note:

In a script, use double quotes to enclose a String tuple value with the `send` command. For example, `send [msg="Enter a value."]`. Single quotes do not work in a script.

To execute the `send-event.oep` script when you start the server, edit the last line of the `startwlevs` command as follows:

```
"$JAVA_HOME/bin/java" -Dgosh.args=send-event.oep $JVM_ARGS $DEBUG_ARGS -
Dwlevs.home="$USER_INSTALL_DIR" -Dbea.home="$BEA_HOME"  -jar "$
{USER_INSTALL_DIR}/bin/wlevs.jar" $ARGS
```

One approach is to have a test driver script that invokes other scripts and directs their output, as follows:

```
source send-event.oep | tac test-output/log/send-event.out
source test-delete.oep | tac test-output/log/test-delete.out
```

14.3.4 EPN Variable

You can set the time stamp format or turn time stamps off for an EPN session.

```
OUT_TIME: FORMATTED | PLAIN | OFF
```

To turn off the time stamp on output events, enter the following:

```
g! OUT_TIME=OFF
```

To indicate formatted time stamping.

```
h! OUT_TIME=FORMATTED
```

14.3.5 EPN Commands

The EPN commands are shown and described in the following list.

`epn:begin`: Begins a new session for invoking EPN commands.

`epn:channel`: Performs different actions based on the parameters:

- Create a channel with a map-based event type.
- Create a named relation channel with update keys and a Java class event type.
- Create a named channel with a map-based event type.
- Create a named channel with a Java class event type.
- Create a channel with a Java class event type.
- Create a named relation channel with update keys and a map-based event type.

`epn:channels`: Lists all channels within the EPN scope.

`epn:createevent`: Creates an event of the provided event type.

`epn:end`: Ends the EPN session.

`epn:eventtypes`: Lists all event types

`epn:heartbeat`: Sends a heartbeat to channel that is time stamped by an application.

`epn:query`: Registers an Oracle CQL query.

`epn:remove`: Removes all statements that are registered within the EPN scope.

`epn:send`: Sends an insert event to a named channel. Same as the `sendinsert` command.

`epn:senddelete`: Deletes a named event in a named channel.

`epn:sendinsert`: Sends an insert event to a named channel. Same as the `send` command.

`epn:sendupdate`: Updates an OSGi bundle with the bundle at the provided URL.

`epn:statements`: Lists all registered Oracle CQL statements within the EPN scope.

`epn:subscribe`: Subscribe to an event channel and output events to a file.

`epn:unsubscribe`: Unsubscribe from an event channel.

`epn:view`: Register an Oracle CQL view.

14.3.6 Management Commands

The management commands are shown and described in the following list.

mngt:deployapp: Deploys an application with optional parameters.

mngt:deployrepapp: Deploys an application from the respository.

mngt:introspect: Introspects an application for its public interfaces.

mngt:listapps: Lists all deployed applications.

mngt:listlibs: Lists all deployed libraries.

mngt:mbean: Retrieves the MBean stage from the local MBean server

mngt:shutdown: Shuts the server down.

mngt:undeployapp: Undeploys the specified application.

14.4 EPN Command Interface

The `EpnCommand` interface is a Java API on top of the Oracle Event Processing EPN CQL programming model. You can use the `EpnCommand` interface in a Java application to manipulate EPN components. You can programmatically start an EPN session and execute EPN commands similar to using the command-line EPN shell described in EPN Shell.

Use `beginSession()` to start an EPN session and `endSession()` to end an EPN session. Ending a session destroys all EPN components that were created during that session. A session is not multithread aware and must be synchronized in your application code.

To use EPN commands in an Oracle Event Processing application, first retrieve the `EpnCommand` OSGi service, which registers the service as an OSGi service factory. Once you retrieve the OSGi service, you can create test cases similar to `JUnit` test cases for testing EPN components and Oracle CQL statements. Only the commands exposed in the `EpnCommand` class as methods can be use for `JUnit`-like test cases.

You can also use the `EpnCommand` interface to create an Oracle Event Processing application through programming.

14.4.1 Session Variables

You can set session variables to indicate the time stamp format to use for output events. The time stamping can be `FORMATTED`, `PLAIN`, or `OFF`.

14.4.2 Methods

`void beginSession()`: Begins a new session for invoking commands. Only a single session can be active at one time. The following list shows the `EpnCommand` methods and parameters. Refer the `EpnCommand` Javadoc for more information.

`endSession()`: Ends the current session. Only a single session can be active at one time.

`void getEventChannels()`: Returns all channels created in the current session.

`EventChannel[] getEventChannel(String channelName)`: Returns the named event channel.

`EventChannel getEventChannel(String channelName)`: Retrieves the named event channel.

Create different types of channels based on the parameters.

- `EventChannel createChannel(String channelName, Class<?> clazz)`: Create a new system time stamped channel with a Java class event type.
- `EventChannel createChannel(boolean relation, String applicationTimestampProp, boolean totalOrder, List<String> keys, String channelName, Class<?> class)`: Create a new channel with a Java class event type.
- `EventChannel createChannel(boolean relation, String applicationTimestampProp, boolean totalOrder, List<String> keys, String channelName, Class<?> class, EventBuilder.Factory factory)`: Create a new channel with a Java class event type.

- `EventChannel createChannel(String channelName, Map<String, String> metadata)`: Create a new system timestamped channel with a metadata-based event type.
- `EventChannel createChannel(boolean relation, String applicationTimestampProp, boolean totalOrder, List<String> keys, String channelName, Map<String, String> metadata)`: Create a new channel with a metadata-based event type.
- `EventChannel createChannel(boolean relation, String applicationTimestampProp, boolean totalOrder, List<String> keys, String channelName, Map<String, String> metadata, EventBuilder.Factory factory)`: Create a new channel with a metadata-based event type.

`Processor[] getProcessors()`: Returns all Oracle CQL processors created in the current session.

`Processor createProcessor()`: Creates an Oracle CQL processor in the current session.

`StreamSource link(StreamSource fromStage, StreamSink toStage)`: Connects a stage to stage.

`void unlink(StreamSource source, StreamSink sink)`: Disconnects the source from the sink.

`Statement createView(Processor processor, String id, String StatementValue)`: Creates an Oracle CQL view.

`Statement createQuery(Processor processor, String id, String statementValue)`: Creates an Oracle CQL query.

`Object createEvent(String eventTypeName)`: Creates an event based on the event type name. The event is created only when it is available in the Event Type repository.

`MapEventObject createMapEvent(EventChannel channel, Map<String, Object> event)`: Create a map event for a metadata-based channel.

`void sendInsert(String channelName, Object event)`: Send an insert event to the channel.

`void sendUpdate(String channelName, Object event)`: Send an update event to the channel.

`void sendDelete(String channelName, Object event)`: Send a delete event to the channel.

`void sendHeartbeat(Long timestamp, String channelName)`: Send a heartbeat to the channel.

14.4.3 Example

The following example shows a basic `JUnit` test case.

```
//Get the OSGI service
ServiceReference ref = ctx.getServiceReference(EpnCommand.class.getName());
commandSession = (EpnCommand) ctx.getService(ref);

//Begin a session
```

```
commandSession.begin();

//Create an Oracle CQL processor
Processor p1 = commandSession.createProcessor("p1");

//Create two stream channels from the MyEvent Java class.
EventChannel ch1 = commandSession.createChannel(false, "c1", MyEvent.class);
EventChannel ch2 = commandSession.createChannel(false, "c2", MyEvent.class);

//Create a listener to retrieve events that equal event e1
//The MyEventListener class implements the StreamSink or RelationSink interface
MyEventListener listener = new MyEventListener();

//Connect channel ch1 and processor p1
commandSession.link((StreamSource) ch1, (StreamSink) p1);

//Connect channel p1 and processor ch2
commandSession.link((StreamSource) p1, (StreamSink) ch2);

//Connect channel ch2 and processor listener
commandSession.link((StreamSource) ch2, (StreamSink) listener);

//Create the query in processor p1
commandSession.createQuery(p1, "q1", "select * from c1");

//Create event e1 and assign two properties, a and 1.
Object e1 = new MyEvent("a", 1);

//Send insert event e1 to channel ch1
ch1.sendInsertEvent(e1);

//Test whether event e1 equals the event retrieved by the listener
assertEquals(e1, listener.getEvent());

//End the session
commandSession.end();
```

15

Debug with Event Record and Playback

You can use the event record and playback feature to debug a running Oracle Event Processing application. While the application runs, you record the events that flow out of an EPN component into a persistent store. You play the events back at a later stage in the application such as in an event bean. In the event bean, you query the events and make fixes to your application based on your findings.

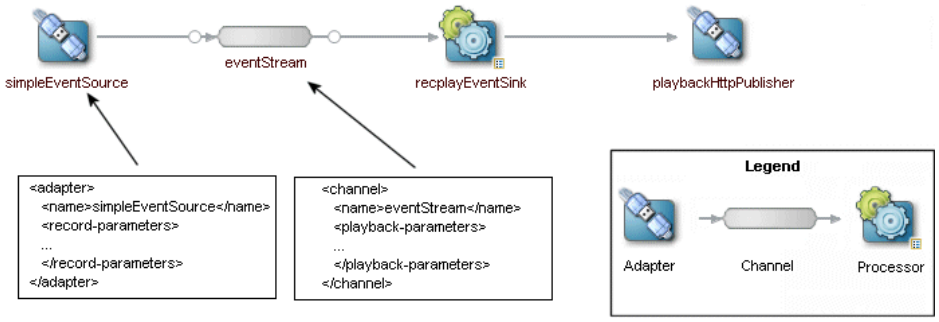
The sample code in this chapter is from the event record and playback example in `/Oracle/Middleware/my_oep/examples/source/applications/recplay`. For details about running and building the example, see *Getting Started with Oracle Event Processing*.

This chapter includes the following sections:

15.1 Event Flow

The following graphic shows the EPN of the event record and playback example to demonstrate where you can record events and where you can play events back. The `simpleEventSource` adapter is configured to record events. The recording happens as events flow out of the adapter. The `eventStream` channel is configured to play back events. The playback happens where events flow into the channel.

Figure 15-1 Configuring Record and Playback in an EPN



15.2 Berkeley DB

When you record events, by default the Oracle Event Processing server stores them in Berkeley DB, which is a persistent event store that is bundled with the Oracle Event Processing server. Berkeley DB is a fast, scalable, transactional database with industrial grade reliability and availability. For more information about Berkeley DB, see:

`http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`.

When you deploy an application that is configured to use the record and playback feature, the Oracle Event Processing server creates the database schema and an instance of Berkeley DB in the following directory.

/Oracle/Middleware/my_oep/`/user_projects/domains/`*domainname/servername*`/bdb`

Note:

The database key is the record time plus the sequence number.

You can use the default Berkeley database configuration as is. You only need to make configuration changes to customize the location of the Berkeley database instance or to tune performance. See Configure Berkeley DB for information about how to configure Berkeley DB.

You can use the event store API to query a store for past events given a record time range and the component from which the events were recorded. The actual query you use depends on the event repository provider; for example, you would use Oracle CQL for the default persistent event store provider included with Oracle Event Processing. You can also use these APIs to delete old events from the event store.

15.3 Record Events

You can configure recording for any component in the EPN that produces events, such as processors, adapters, channels, and event beans. Processors and channels always produce events. Adapters and event beans must implement the `EventSource` interface.

You can configure events from different components in the EPN to be stored in different persistent stores, or that all events go to the same store. Only events that are output by the component are recorded.

You enable the recording of events for a component by updating its configuration file and adding the `record-parameters` element. Using the child elements of `record-parameters`, you specify the event store to which the events are recorded, an initial time period when recording should take place, the list of event types you want to store, and so on.

After you deploy the application and events start flowing through the network, recording begins either automatically because you configured it to start at a certain time or because you dynamically start it using administration tools. For each component you have configured for recording, Oracle Event Processing stores the events that flow out of it to the appropriate store along with a time stamp of the time it was recorded.

15.4 Play Back Events

You can configure playback for any component in the EPN: processors, adapters, streams, and event beans. Typically the playback component is a stage later in the network than the stage that recorded the events.

You enable the playback of events for a component by updating its configuration file and adding the `playback-parameters` element. Using the child elements of `playback-parameters`, you specify the event store from which the events are played back, the list of event types you want to play back (by default all are played back), the time range of the recorded events you want to play back, and so on. By default, Oracle Event Processing plays back the events in a time accurate manner. However, you can configure that the events get played back either faster or slower than they originally flowed out of the component from which they were recorded.

After you deploy the application and events start flowing through the network, you must start the playback with Oracle Event Processing Visualizer or `wlevs.Admin`. Oracle Event Processing reads the events from the persistent store and inserts them into the appropriate place in the EPN.

When a component receives a playback event, the playback event looks exactly like the original event. If a downstream component is configured to record events, then Oracle Event Processing records the arriving playback events and real-time events.

For more information, see:

- *Using Visualizer for Oracle Event Processing*
- *Administering Oracle Event Processing*.

15.5 Configure Berkeley DB

You can use the default Berkeley DB configuration as is. You only need to make configuration changes to customize the location of the Berkeley database instance or to set the cache size to tune performance.

To configure an event store for Oracle Event Processing server:

1. Stop the Oracle Event Processing server instance, if it is running.
2. Open the server `config.xml` file for editing.

   The file is located in `/Oracle/Middleware/my_oep/user_projects/domains/<domain_name>/<server_name>/config.`

3. Edit the `bdb-config` element in the `config.xml` file.

   The following example shows a fully configured `bdb-config` element.

```
<bdb-config>
    <db-env-path>bdb</db-env-path>
    <cache-size>1000</cache-size>
</bdb-config>
```

   Table 15-1 lists the `bdb-config` child elements.

Table 15-1 Child Elements of bdb-config

Child Element	Description
`db-env-path`	Specifies the subdirectory in which OracleEvent Processing server creates the Berkeley DB instances relative to the *`config`* directory of your server. Default: `bdb`
`cache-size`	Specifies the amount of memory, in bytes, available for Berkeley DB cache entries. You can adjust the cache size to tune Berkeley database performance. The cache size must be a power of 2, but it is otherwise limited only by available memory and performance considerations. Default: `je.maxMemoryPercent * JVM maximum memory`.

4. Restart the Oracle Event Processing server.

15.6 Configure a Component to Record Events

You can configure any processor, adapter, channel, or event bean in your application to record events. This section updates an adapter configuration to record events. See *Schema Reference for Oracle Event Processing* for the complete XSD Schema that describes the event recording configuration file elements.

1. Open the component configuration file and add a `record-parameters` child element to the component you want to configure to record events, as follows:

```
<csv-adapter>
  <name>StockTradeCSVAdapter</name>
  <record-parameters>
      ...
  </record-parameters>
</csv-dapter>
```

2. Add child elements to `record-parameters` to specify a data set name, the list of one or more events to be stored, the recording start and stop times, and so on:

```
<csv-adapter>
  <name>StockTradeCSVAdapter/name>
  <record-parameters>
    <dataset-name>recplay_sample</dataset-name>
    <event-type-list>
      <event-type>TradeEvent</event-type>
    </event-type-list>

    <time-range>
      <start>2010-01-20T05:00:00</start>
      <end>2010-01-20T18:00:00</end>
    </time-range>
```

```
    </record-parameters>
  </csv-adapter>
```

Table 15-2 lists the child elements of `record-parameters` that you can specify. Only `dataset-name` is required.

Table 15-2 Child Elements of record-parameters

Child Element	Description
`dataset-name`	Berkeley DB: Identifies the recorded data and places it in a directory of this name below the directory specified by the `db-env-path` setting in the server `config.xml` file. Oracle RDBMS-based provider: Specifies the database area, or schema, in which the tables that store the recorded events are created. When you configure the Oracle RDBMS-based provider, you must specify this element.
`event-type-list`	Berkeley DB: Specifies the event types that are recorded to the event store. If this element is not specified, then Oracle Event Processing records *all* event types that flow out of the component. Oracle RDBMS-based provider: You must specify this element.

`time-range`	Specifies the time period during which recording takes place. Configure the time period with a `start` child element to specify a start time and an `end` child element to specify the end time.
	Express the start and end time as XML Schema `dateTime` values of the form:
	`yyyy-mm-ddThh:mm:ss`
	For example, to have recording start on January 20, 2010, at 5:00 am and end on January 20, 2010, at 6:00 pm, enter the following:
	`<time-range>`
	`  <start>2010-01-20T05:00:00</start>`
	`  <end>2010-01-20T18:00:00</end>`
	`</time-range>`
	For complete details of the XML Schema `dateTime` format, see `http://www.w3.org/TR/xmlschema-2/#dateTime-lexical-representation`.
	If you do not specify a time period, then no events are recorded when the application is deployed and recording happens only after you explicitly start it with Oracle Event Processing Visualizer or `wlevs.Admin`.
	You can specify `time-range` or `time-range-offset`, but not both.

Table 15-2 (Cont.) Child Elements of record-parameters

Child Element	Description

`time-range-offset`	Specifies the time period during which recording takes place. Configure the time period with a `start` child element to specify a start time, and a `duration` child element to specify the length of time to run the recording.
	Express the start time as an XML Schema `dateTime` value of the form:
	`yyyy-mm-ddThh:mm:ss`
	Express the duration in the form:
	`hh:mm:ss`
	For example, to have recording start on January 20, 2010, at 5:00 am and continue for 3 hours, enter the following:
	`<time-range-offset>`
	`<start>2010-01-20T05:00:00</start>`
	`<duration>03:00:00</duration>`
	`</time-range-offset>`
	For complete details of the XML Schema `dateTime` format, see http://www.w3.org/TR/xmlschema-2/#dateTime-lexical-representation.
	If you do not specify a time period, then no events are recorded when the application is deployed and recording happens only after you explicitly start it with Oracle Event Processing Visualizer or `wlevs.Admin`.
	You can specify `time-range` or `time-range-offset`, but not both.

`batch-size`	Specifies the number of events that Oracle Event Processing picks up in a single batch from the event buffer to write the event store. Default value is 1000.
`batch-time-out`	Specifies the number of seconds that Oracle Event Processing waits for the event buffer window to fill up with the `batch-size` number of events before writing to the event store. Default value is 60
`max-size`	When specified, Oracle Event Processing uses a stream when writing to the event store, and this element specifies the size of the stream. Non-zero values indicate asynchronous writes. Default value is 1024.
`max-threads`	When specified, Oracle Event Processing uses a stream when writing to the event store, and this element specifies the maximum number of threads to be used to process events for this stream. Setting this value has no effect when `max-size` is 0. The default value is 1.

15.7 Configure a Component to Play Back Events

You can configure any processor, adapter, channel, or event bean in your application to play back events. The component must downstream from the recording component so that the playback component can receive the events and play them back. This section updates a channel configuration to play back events.

See *Schema Reference for Oracle Event Processing* for the complete XSD Schema that describes the event playback configuration file elements.

1. Open the component configuration XML file and add a `playback-parameters` child element to the component you want to configure to playback events. For example, to configure a channel called `eventStream`:

```
<channel>
  <name>eventStream</name>
  <playback-parameters>
   ...
  </playback-parameters>
</channel>
```

2. Add child elements to `playback-parameters` to specify a data set, one or more events to be played back, and so on. For example:

```
<channel>
  <name>eventStream</name>
  <playback-parameters>
      <dataset-name>recplay_sample</dataset-name>
      <event-type-list>
          <event-type>SimpleEvent</event-type>
      </event-type-list>
  </playback-parameters>
</channel>
```

Table 15-3 lists the child elements of `playback-parameters` that you can specify. Only `dataset-name` is required.

Table 15-3 Child Elements of playback-parameters

Child Element	Description
`dataset-name`	Berkeley DB: Identifies the recorded data and places it in a directory of this name below the directory specified by the `db-env-path` setting in the server `config.xml` file. Oracle RDBMS-based provider: Specifies the database area, or schema, in which the tables that store the recorded events are created. When you configure the Oracle RDBMS-based provider, you must specify this element. When you configure the Oracle RDBMS-based provider, you must specify this element.
`event-type-list`	Berkeley DB: Specifies the event types that are played back from the event store. If this element is not specified, then Oracle Event Processing plays back *all* event types. Oracle RDBMS-based provider: You must specify this element.

Table 15-3 (Cont.) Child Elements of playback-parameters

Child Element	Description

`time-range`	Specifies the time period during which play back takes place with a start and end time. Configure the time period with a `start` child element to specify a start time and an `end` child element to specify the end time.
	Express the start and end time as XML Schema `dateTime` values of the form:
	`yyyy-mm-ddThh:mm:ss`
	For example, to specify that play back to start on January 20, 2010, at 5:00am and end on January 20, 2010, at 6:00 pm, enter the following:
	`<time-range>`
	`  <start>2010-01-20T05:00:00</start>`
	`  <end>2010-01-20T18:00:00</end>`
	`</time-range>`
	For complete details of the XML Schema `dateTime` format, see http://www.w3.org/TR/xmlschema-2/#dateTime-lexical-representation.
	If you do not specify a time period, then no events are played back when the application is deployed and play back happens only after you explicitly start it using Oracle Event Processing Visualizer or `wlevs.Admin`.
	You can specify `time-range` or `time-range-offset`, but not both.

Table 15-3 (Cont.) Child Elements of playback-parameters

Child Element	Description

`time-range-offset`	Specifies the time period during which play back takes place with a start time and a duration. Configure the time period with a `start` child element to specify a start time and a `duration` child element to specify the length of time to play back events.
	Express the start time as an XML Schema `dateTime` value of the form:
	`yyyy-mm-ddThh:mm:ss`
	Express the duration in the form:
	`hh:mm:ss`
	For example, to specify that play back should start on January 20, 2010, at 5:00am and continue for 3 hours, enter the following
	`<time-range-offset>`
	`<start>2010-01-20T05:00:00</start>`
	`<duration>03:00:00</duration>`
	`</time-range-offset>`
	For complete details of the XML Schema `dateTime` format, see http://www.w3.org/TR/xmlschema-2/#dateTime-lexical-representation.
	If you do not specify a time period, then no events are played back when the application is deployed and playback happens after you explicitly start it using Oracle Event Processing Visualizer or `wlevs.Admin`.
	You can specify `time-range` or `time-range-offset`, but not both.

`playback-speed`	Specifies the playback speed as a positive float. The default value is 1, which corresponds to normal speed. A value of 2 means that events are played back 2 times faster than the original record speed. A value of 0.5 means that events will be played back at half the speed.
`repeat`	Specifies whether to playback events again after the playback of the specified time interval completes. Valid values are `true` and `false`. Default value is `false`. A value of `true` means that the repeat of playback continues an infinite number of times until it is deliberately stopped. `False` means that events are played back once.
`max-size`	If specified, Oracle Event Processing uses a stream when playing back events from the event store. This element specifies the size of the stream with non-zero values indicating asynchronous writes. Default value is 1024.

Table 15-3 (Cont.) Child Elements of playback-parameters

Child Element	Description
`max-threads`	If specified, Oracle Event Processing uses a stream to play back events from the event store. This element specifies the maximum number of threads to use to process events for the stream. This value has no effect when `max-size` is 0. The default value is 1.

15.8 Start and Stop the Record and Playback of Events

After you configure the record and playback functionality for the components of an application, and you deploy the application to Oracle Event Processing, the server starts to record events only when you have configured explicit star and stop information in the configuration file. For example, if you included the following element in a component configuration, then recording starts on January 20, 2010 at 5:00 am:

```
<time-range>
    <start>2010-01-20T05:00:00</start>
    <end>2010-01-20T18:00:00</end>
</time-range>
```

To enable the recording and playback of events, use Oracle Event Processing Visualizer or `wlevs.Admin`. Once recording and playback are enabled, they start and stop according to their configuration settings.

For more information, see:

- *Using Visualizer for Oracle Event Processing*
- *Administering Oracle Event Processing*.

Visualizer and `wlevs.Admin` use managed beans (MBeans) to dynamically start and stop event recording and playback and manage the event store configuration. A managed bean is a Java bean that provides a Java Management Extensions (JMX) interface. JMX is the Java EE solution for monitoring and managing resources on a network. You can create your own administration tool and use JMX to manage event store functionality with `com.bea.wlevs.management.configuration.StageMBean`.

For more information, see:

- *Administering Oracle Event Processing*
- *Java API Reference for Oracle Event Processing*.

Part III

Tune and Scale

Part III contains the following chapters:

16

Performance Tuning

This chapter describes techniques for improving Oracle Event Processing application performance by using partitioning and batching, and includes information specific to high availability performance tuning.

This chapter includes the following sections:

- Channel and JMS Performance Tuning
- High Availability Performance Tuning.

16.1 Channel and JMS Performance Tuning

You can tune application performance of by configuring an event partitioner channel, batching events, and partitioning an incoming Java Message Service (JMS) stream.

Event partitioner channel: You can improve scalability by configuring an event partitioner channel. When you configure a channel to use an event partitioner, each time an incoming event arrives, the channel selects a listener and dispatches the event to that listener instead of broadcasting each event to every listener for partitioning events on a channel across its output event sinks. For more information, see *Customizing Oracle Event Processing*.

Batching channel: By default, a channel processes events as they arrive. Alternatively, you can configure a channel to batch events that have the same time stamp and were output from the same query by setting the `wlevs:channel` attribute `batching` to `true`. See Batch Processing Channels for an example.

Scalability with ActiveActiveGroupBean. Use `com.oracle.cep.cluster.hagroups.ActiveActiveGroupBean` to partition an incoming JMS stream in Oracle Event Processing applications with the notification groups that the `ActiveActiveGroupBean` creates. For more information, see Scalable Applications.

16.2 High Availability Performance Tuning

When creating high-availability applications for deployment to multiserver domains, consider the following performance tuning options:

Host configuration: To maximize high availability performance, ensure that all hosts in the multiserver domain are configured with equivalent processing capacity (similar number and type of CPUs), and that all hosts have sufficient memory and disk for the needs of the application

High availability input adapter and quality of service: The Oracle Event Processing high availability input adapter is applicable to all high availability quality of service options. However, because the high availability input adapter increases performance overhead, it is not appropriate for some high availability quality of service options, such as those described in Simple Failover and Simple Failover with Buffering. If you

are using application time from the event, then you do not need to use the input adapter. Application time from the event is always preferable from a performance standpoint.

High availability input adapter configuration: Consider increasing the `batch-size` to reduce the amount of time the primary server spends broadcasting event messages and to reduce the amount of time the secondary servers spend processing these messages. Increasing the `batch-size` can increase the likelihood of missed and duplicate events when the primary fails before broadcasting an event message with a large number of events.

Broadcast output adapter configuration: Decrease the `trimming-interval` to reduce the amount of time the primary server spends broadcasting trimming messages and to reduce the amount of time the secondary servers spend processing these messages. Decreasing the `trimming-interval` may increase recovery time as the new primary server's in-memory queue will be more out of date relative to the old primary.

Oracle Coherence performance tuning options. When you configure Oracle Coherence in a high-availability architecture, consider the following options:

- Increase the Oracle Coherence heartbeat time out machine frequency to reduce the number of heartbeats before failure. See *Oracle Coherence Developer's Guide* at `http://download.oracle.com/docs/cd/E15357_01/coh.360/e15723/tune_perftune.htm`.
- Implement the Oracle Coherence Portable Object Format (POF) for serialization to improve messaging performance. POF is a language agnostic binary format that was designed to be very efficient in both space and time. Using POF instead of Java serialization can greatly improve performance.

17

High Availability Applications

High availability is critical to Oracle Event Processing applications because they continuously monitor streaming data. Oracle Event Processing provides application design patterns and high availability adapters, to enable you to increase the backup and failover processing capabilities of you applications.

This chapter includes the following sections:

- Oracle Coherence
- Architecture
- Life Cycle and Failover
- Deployment Group and Notification Group
- High Availability Adapters
- High Availability and Scalability
- Choose a Quality of Service Option
- Design Applications for High Availability
- Configure High Availability Quality of Service
- Configure High Availability Adapters.

17.1 Oracle Coherence

Oracle Event Processing high availability options depend on Oracle Coherence. You cannot implement Oracle Event Processing high availability options without Oracle Coherence. When considering performance tuning, be sure to evaluate your Oracle Coherence configuration in addition to your Oracle Event Processing application.

For more information, see:

- High Availability Performance Tuning
- *Administering Oracle Event Processing*
- *Oracle Coherence Developer's Guide* at `http://download.oracle.com/docs/cd/E15357_01/coh.360/e15723/toc.htm`.

17.2 Architecture

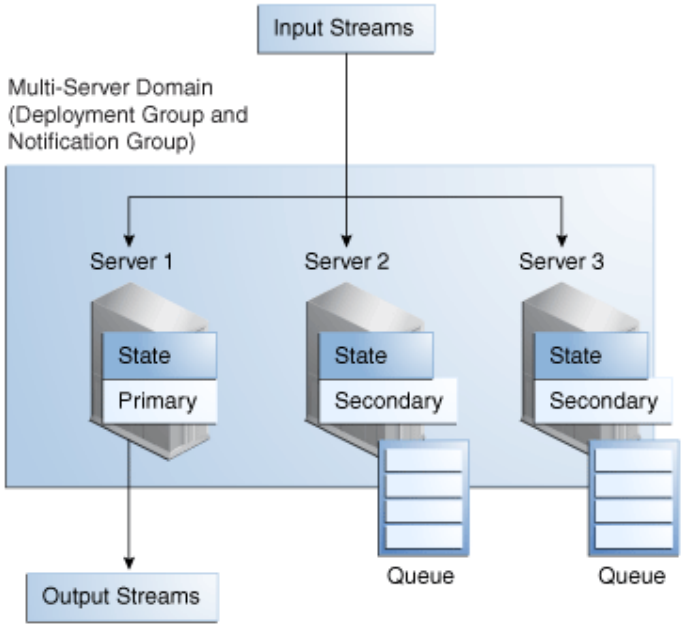
Oracle Event Processing supports an active-active high availability architecture. This approach has the advantages of high performance, simplicity, and short failover time to mitigate the likelihood and impact of data and service faults.

Deploy high availability applications to a group of two or more Oracle Event Processing servers running in a multiserver domain. Oracle Event Processing chooses one server in the group to be the active primary server. The other servers become active secondary servers.

The primary and secondary servers are configured to receive the same input events and process them in parallel but only the primary server outputs events to the Oracle Event Processing application client. Depending on the quality of service you choose, the secondary servers buffer their output events using in-memory queues and the primary server keeps the secondary servers up to date with which events the primary has already output.

Figure 17-1 shows a typical configuration with one active server and two primary servers.

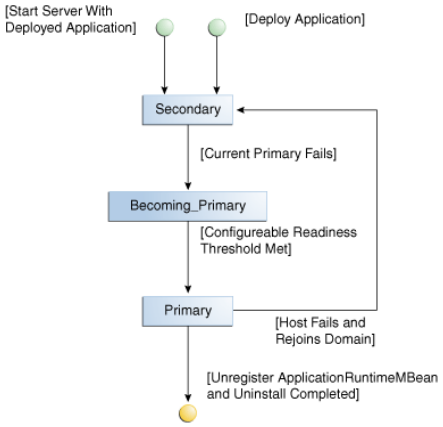
Figure 17-1 Oracle Event Processing High Availability: Primary and Secondary Servers



17.3 Life Cycle and Failover

Figure 17-2 shows a state diagram for the Oracle Event Processing high availability life cycle. In this diagram, the state names (`SECONDARY`, `BECOMING_PRIMARY`, and `PRIMARY`) correspond to the Oracle Event Processing high availability adapter `RuntimeMBean` method `getState` return values. These states are specific to Oracle Event Processing. See High Availability Input Adapter for more information about the high availability adapter.

Figure 17-2 Oracle Event Processing High Availability Life Cycle State Diagram

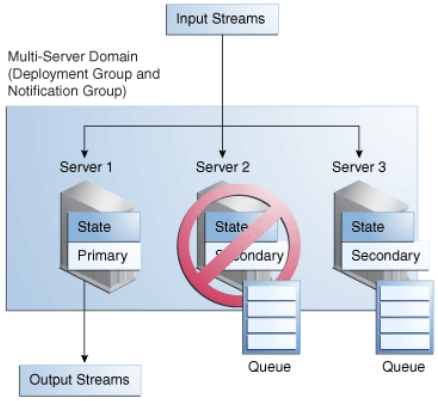


You cannot specify the initial primary server. Initially, the first server in the multiserver domain to start up becomes the primary so by starting servers in a particular order, you can influence primary selection. There is no way to force a particular, running server to become the primary. If a primary fails and comes back up, then it does not automatically become the primary again unless the current primary fails and causes a failover.

17.3.1 Secondary Failure

In general, when a secondary server fails, there is no effect on Oracle Event Processing application operation as Figure 17-3 shows. Regardless of the quality of service you choose, there are no missed or duplicate events.

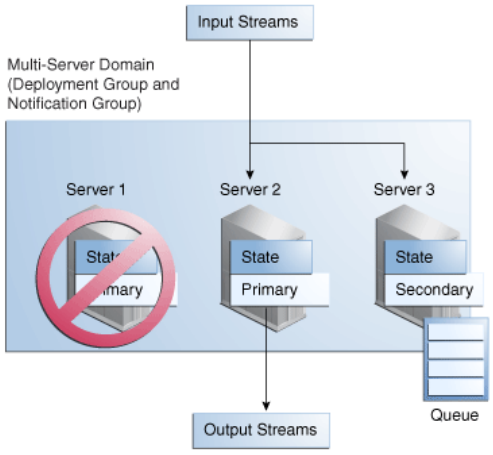
Figure 17-3 Secondary Failure



17.3.2 Primary Failure and Failover

When a primary server fails, as Figure 17-4 shows, Oracle Event Processing performs a failover that can cause missed or duplicate events, depending on the quality of service you choose.

Figure 17-4 Primary Failure and Failover



During failover, Oracle Event Processing selects a new primary and the new primary transitions from the `SECONDARY` state to the `BECOMING_PRIMARY` state. Depending on the quality of service you choose, the new primary does not transition to `PRIMARY` state until a configurable readiness threshold is met. For details, see the specific quality of service option in Choose a Quality of Service Option.

17.3.3 Rejoining the High Availability MultiServer Domain

When a new Oracle Event Processing server is added to an Oracle Event Processing high availability multiserver domain or an existing failed server restarts, the server does not fully join the Oracle Event Processing high availability deployment and notification groups until all applications deployed to it have fully joined. The type of application determines when it can fully join.

If the application must generate exactly the same sequence of output events as existing secondaries (a Type 1 application), then it must be able to rebuild its internal state by processing input streams for some finite period of time (the `warm-up-window-length` period). This `warm-up-window-length` time determines the minimum time it takes for the application to fully join the Oracle Event Processing high availability deployment and notification groups.

If the application does not need to generate exactly the same sequence of output events as existing secondaries (a Type 2 application), then it does not require a `warm-up-window-length` time and fully joins the Oracle Event Processing high availability deployment and notification groups when it deploys.

For more information, see Choose an Adequate warm-up-window-length Time.

17.4 Deployment Group and Notification Group

All of the servers in the multiserver domain belong to the same deployment group. the deployment group is the group to which you deploy an application. For the purposes of Oracle Event Processing high availability, you must deploy the same application to all of the servers in this group.

By default, all of the servers in the multiserver domain also belong to the same notification group. The servers listen to the notification group for membership notifications that indicate when a server fails (and exits the group) or resumes operation (and rejoins the group), and for synchronization notifications from the primary.

If you need to scale your Oracle Event Processing high availability application, use the `ActiveActiveGroupBean` to define a notification group that allows two or more servers to function as a primary server unit while retaining the convenience of a single deployment group that spans all servers (primaries and secondaries).

You must use Oracle Coherence-based clustering to create the multiserver domain deployment group. You can use either default groups or custom groups.

For more information, see:

- High Availability and Scalability
- Oracle Coherence
- *Administering Oracle Event Processing*.

17.5 High Availability Adapters

To implement Oracle Event Processing high availability, add the optional high availability input adapter and the required high availability output adapters to the EPN.

High Availability Input Adapter

The optional high availability input adapter in the primary server communicates with the corresponding high availability input adapters in each secondary server to normalize event time stamps. Oracle Event Processing high availability provides one type of high availability input adapter.

See High Availability Input Adapter.

High Availability Output Adapters

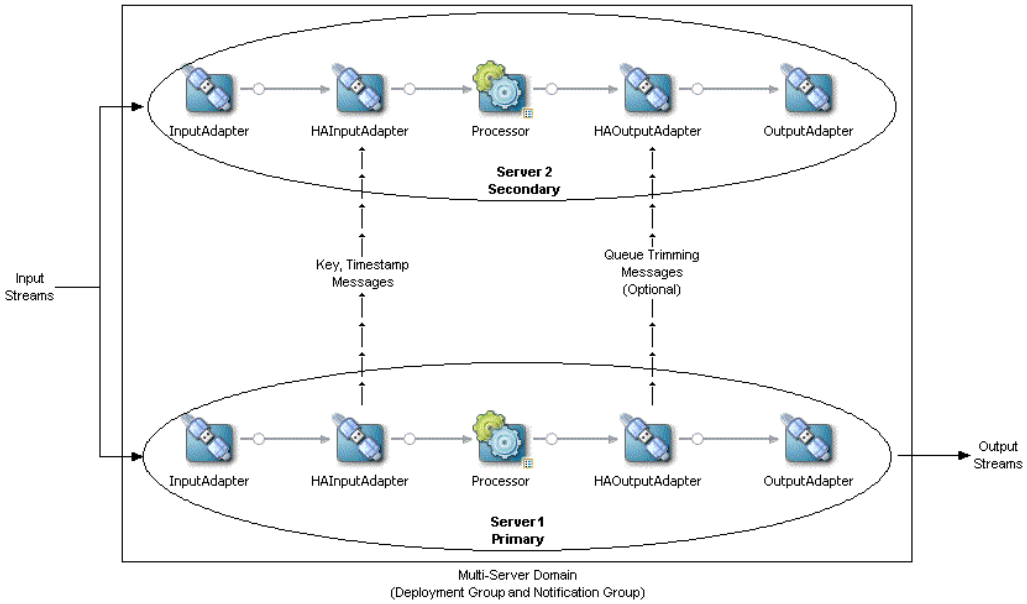
To have high availability functionality in your application, put a high availability output adapter before each output adapter in your EPN. The high availability output adapter in the primary server outputs events to the output streams that connect the Oracle Event Processing application to its downstream client.

The high availability output adapter in the primary also communicates with the corresponding high availability output adapters in each secondary, and depending on the high availability quality of service you choose, can instruct the secondary output adapters to trim their in-memory queues of output events.

For information about the high availability output adapters, see Buffering Output Adapter, Broadcast Output Adapter, and Correlating Output Adapter. Which output adapter you choose is determined by the high availability quality of service you choose. See Choose a Quality of Service Option.

Figure 17-5 shows a simplified EPN with all possible high availability adapters in place. The figure shows no channels and one processor.

Figure 17-5 High Availability Adapters in the EPN



17.5.1 High Availability Input Adapter

Each event is associated with a point in time at which the event occurred. There are two different approaches to generating event timestamps: application timestamps and system timestamps (see Channels for information about application and system time stamps). Application time means that a time value is assigned to each event externally by the application before the event enters a CQL processor. System time means that a time value is associated with an event automatically by Oracle Event Processing when it arrives at a CQL processor.

Application time is generally the best approach for applications that need to be highly available. The application time is associated with an event before the event is sent to Oracle Event Processing, so it is consistent across active primary and secondary instances. System time can cause application instances to generate different results because the time value associated with an event can be different on each instance due to system clocks not being synchronized.

Using system time is not a problem for applications whose CQL queries do not use time-based windows. Applications that use only event-based windows depend only on the arrival order of events rather than the arrival time, so system time can be used in this case.

For applications that use time-based windows and do not have externally generated (application) time stamps, the optional Oracle Event Processing high availability input adapter can be used to provide a consistent time value across all servers. The input adapter instance on the primary server assigns a time (in nanoseconds) to events as they arrive at the adapter and forwards the time values assigned for each event to all secondary servers.

Because a time value is assigned to each event before the event reaches any downstream channels in the EPN, downstream channels should be configured to use application time so that they do not assign a new time value to events as they arrive at the channel.

Input events must have a key that uniquely identifies each event in order to use this adapter.

You can configure the Oracle Event Processing high availability input adapter to send heartbeat events.

The Oracle Event Processing high availability input adapter is applicable to all high availability quality of service options. However, because the high availability input adapter increases performance overhead, it is not appropriate for some high availability quality of service options (such as Simple Failover and Simple Failover with Buffering). For these options, you should instead consider using application time with some incoming event property.

For more information, see:

- Light-Weight Queue Trimming
- Precise Recovery with JMS
- Configure the High Availability Input Adapter.

17.5.2 Buffering Output Adapter

The Oracle Event Processing high availability buffering output adapter implements a buffered queue trimming strategy. The buffer is a sliding window of output events from the stream. The size of the window is measured in milliseconds.

The Oracle Event Processing high availability buffering output adapter applies to simple failover, and simple failover with buffering high availability quality of service options.

For more information, see:

- Simple Failover
- Simple Failover with Buffering
- Configure the Buffering Output Adapter.

17.5.3 Broadcast Output Adapter

The Oracle Event Processing high availability broadcast output adapter implements a distributed queue trimming strategy. The active primary instance broadcasts messages to the active secondary instances in the notification group telling them when to trim their local representation of the queue.

The Oracle Event Processing high availability broadcast output adapter applies to the light-weight queue trimming high availability quality of service option.

For more information, see:

- Light-Weight Queue Trimming
- Configure the Broadcast Output Adapter.

17.5.4 Correlating Output Adapter

The Oracle Event Processing high availability correlating output adapter correlates two event streams, usually from JMS. This adapter correlates an inbound buffer of events with a second source of the same event stream, outputting the buffer if correlation fails after a configurable time interval. Correlated events are trimmed from the queue. Correlated events are assumed to be in-order.

The Oracle Event Processing high availability correlating output adapter applies to precise recovery with JMS high availability quality of service option.

For more information, see:

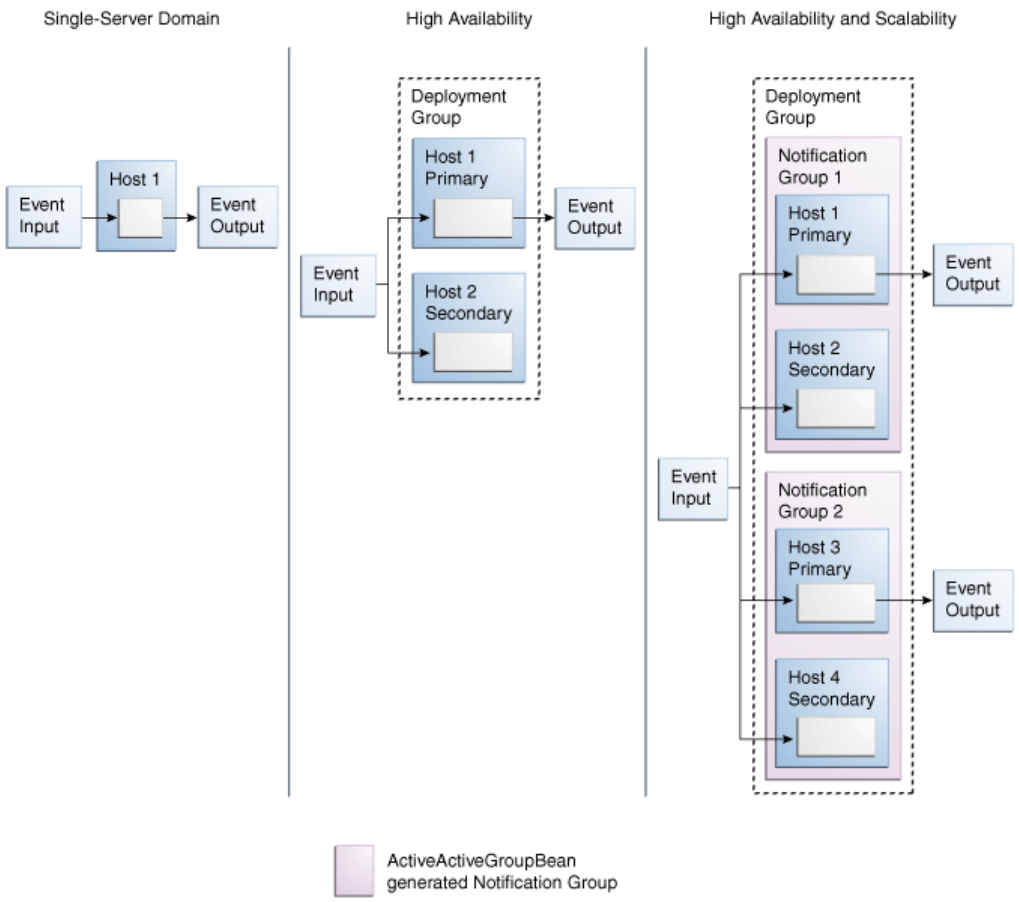
- Precise Recovery with JMS
- Configure the Correlating Output Adapter.

17.6 High Availability and Scalability

If you need to scale your Oracle Event Processing high availability application, use the notification groups Spring bean, `ActiveActiveGroupBean` to increase scalability in JMS applications. The `ActiveActiveGroupBean` defines a notification group that allows two or more servers to function as a high availability unit while retaining the convenience of a single deployment group that spans all servers (primaries and secondaries).

Figure 17-6 shows three Oracle Event Processing application scenarios progressing from the simplest configuration, to high availability, and then to both high availability and scalability.

Figure 17-6 High Availability and Scalability



Most applications begin in a single-server domain without high availability. In this case, the simplest scenario, which is an Oracle Event Processing application running on one Oracle Event Processing server processes an input event stream and produces output events.

High availability scenario: The application is configured to use Oracle Event Processing high availability options, is deployed to the deployment group of a multiserver domain composed of two server, and only the primary server outputs events.

High availability and scalability scenario: The high availability application is configured to use the `ActiveActiveGroupBean` to define notification groups. Each notification group contains two or more servers that function as a single, high availability unit. In this scenario, only the primary server in each notification group outputs events. Should the primary server in a notification group go down, an Oracle Event Processing high availability fail over occurs and a secondary server in that notification group is declared the new primary and resumes outputting events according to the Oracle Event Processing high availability quality of service you configure.

For more information, see Configure Partitioning with High Availability.

17.7 Choose a Quality of Service Option

You can choose any of the quality of service options that Table 17-1 lists. Choose the quality of service option that suits your application's tolerance for missed and duplicate events and expected event throughput. Note that primary and secondary server hardware requirements increase as the quality of service becomes more precise.

Table 17-1 Oracle Event Processing High Availability Quality of Service

High Availability Option	Missed Events?	Duplicate Events?	Performance Overhead
Simple Failover	Yes (many)	Yes (few)	Negligible
Simple Failover with Buffering	Yes (few)	Yes (many)	Low
Light-Weight Queue Trimming	No	Yes (few)	Low-Medium
Precise Recovery with JMS	No	No	High

17.7.1 Simple Failover

The simple failover high availability quality of service is characterized by the lowest performance overhead (fastest recovery time) and the least data integrity (both missed events and duplicate events are possible during failover).

The primary server outputs events and secondary servers discard their output events because they do not buffer output events. If the current active primary fails, a new active primary is chosen and begins sending output events once it is notified.

During failover, many events can be missed or duplicated by the new primary depending on whether it is running ahead of or behind the old primary, respectively.

During the failover window, events can be missed. For example, if you process 100 events per second and failover takes 10 seconds, then you miss 1000 events

The new primary server enters the `PRIMARY` state immediately. There is no configurable readiness threshold that must be met before the new primary server

transitions out of the `BECOMING_PRIMARY` state. When an Oracle Event Processing server rejoins the multiserver domain, it is immediately available as a secondary.

To implement this high availability quality of service, you configure your EPN with a high availability buffering output adapter (with a sliding window of size zero) before each output adapter. To reduce performance overhead, rather than use a high availability input adapter, use application time with some incoming event property.

For more information, see Configure a Simple Failover.

17.7.2 Simple Failover with Buffering

The simple failover with buffering high availability quality of service is characterized by a low performance overhead (faster recovery time) and increased data integrity (no missed events but many duplicate events are possible during failover).

The primary server outputs events and the secondary servers buffer their output events. If the current active primary fails, a new active primary is chosen and begins sending output events once it is notified.

During the failover window, events might be missed. For example, if you are processing 100 events per second and failover takes 10 seconds, then you miss 1000 events. If the secondary buffers are large, a significant number of duplicates can be output. On the other hand, a larger buffer reduces the chances of missed messages.

When an Oracle Event Processing server rejoins the multiserver domain, if your application is an Oracle Event Processing high availability Type 1 application (the application must generate exactly the same sequence of output events as existing secondaries), then it must wait the `warm-up-window-length` time you configure for the Oracle Event Processing high availability output adapter before it is available as a secondary.

To implement this high availability quality of service, you configure your EPN with a high availability buffering output adapter with a sliding window of size greater than zero before each output adapter. To reduce performance overhead, rather than use a high availability input adapter, use application time with some incoming event property.

For more information, see:

- Choose an Adequate warm-up-window-length Time
- Configure Simple Failover With Buffering.

17.7.3 Light-Weight Queue Trimming

This high availability quality of service is characterized by a low performance overhead (faster recovery time) and increased data integrity (no missed events but a few duplicate events are possible during failover).

The active primary communicates to the secondaries the events that it has actually processed. This enables the secondaries to trim their buffer of output events so that it contains only those events that have not been sent by the primary at a particular point in time. Because events are only trimmed after they have been sent by the current primary, this allows the secondary to avoid missing any output events when there is a failover.

The frequency with which the active primary sends queue trimming messages to active secondaries is configurable:

- Every `n` events (`n>0`)

  This limits the number of duplicate output events to at most `n` events at failover.

- Every `n` milliseconds (`n>0`)

The queue trimming adapter requires a way to identify events consistently among the active primary and secondaries. The recommended approach is to use application time to identify events, but any key value that uniquely identifies events works.

The advantage of queue trimming is that output events are never lost. There is a slight performance overhead at the active primary, however, for sending the trimming messages that need to be communicated. This overhead increases as the frequency of queue trimming messages increases.

During failover, the new primary enters the `BECOMING_PRIMARY` state and does not transition into the `PRIMARY` state until its event queue (that it was accumulating as a secondary) has been flushed. During this transition, new input events are buffered and some duplicate events can be output.

When an Oracle Event Processing server rejoins the multiserver domain, if your application is an Oracle Event Processing high availability Type 1 application (an application that must generate exactly the same sequence of output events as existing secondaries), then it must wait the `warm-up-window-length` time you configure for the Oracle Event Processing high availability output adapter before it is available as a secondary.

To implement this high availability quality of service, you configure your EPN with a high availability input adapter after each input adapter and a high availability broadcast output adapter before each output adapter.

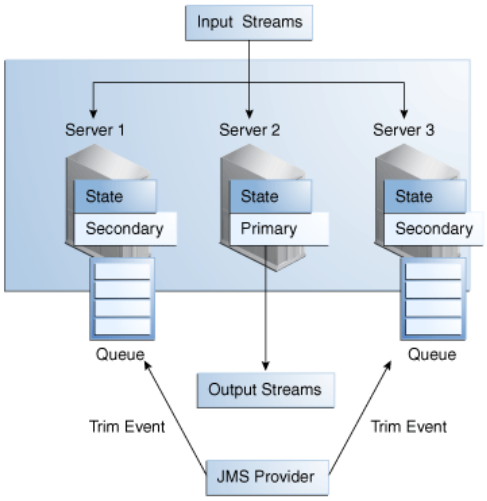
For more information, see Configure Light-Weight Queue Trimming.

17.7.4 Precise Recovery with JMS

The precise recovery with JMS high availability quality of service is characterized by a high performance overhead (slower recovery time) and maximum data integrity (no missed events and no duplicate events during failover). This high availability quality of service is compatible with only JMS input and output adapters.

In the precise recovery with JMS high availability quality of service, the focus is not on transactional guarantees along the event path for a single-server, but on guaranteeing a single output from a set of servers. To achieve guarantee, secondary servers listen over JMS to the event stream being published by the primary. As Figure 17-7 shows, this incoming event stream is a source of reliable queue-trimming messages that the secondaries use to trim their output queues. If JMS is configured for reliable delivery, we can be sure that the stream of events seen by the secondary is precisely the stream of events output by the primary and so failover allows the new primary server to output precisely those events not delivered by the old primary server.

Figure 17-7 Precise Recovery with JMS



During failover, the new primary server enters the `BECOMING_PRIMARY` state and does not transition into the `PRIMARY` state in its event queue (that was accumulating as a secondary) has been flushed. During this transition, new input events are buffered and no duplicate events are output.

When an Oracle Event Processing server rejoins the multiserver domain, if your application is an Oracle Event Processing high availability Type 1 application (the application must generate exactly the same sequence of output events as existing secondaries), it must wait the `warm-up-window-length` time you configure for the Oracle Event Processing high availability output adapter before it is available as a secondary server.

To implement the precise recovery with JMS high availability quality of service, you configure your EPN with a high availability input adapter after each input adapter and a high availability correlating output adapter before each output adapter.

To increase scalability, you can also use the cluster groups bean with high availability quality of service.

For more information, see:

- Configure Precise Recovery With JMS
- Partition an Incoming JMS Event Stream.

17.8 Design Applications for High Availability

Although you can implement Oracle Event Processing high availability declaratively, to fully benefit from the high availability quality of service you choose, you must design your Oracle Event Processing application to take advantage of the high availability options that Oracle Event Processing provides.

When you design your application for high availability, consider the primary use case, design patterns, and Oracle CQL query restrictions discussed here.

17.8.1 Primary High Availability Use Case

You can adapt high availability options to various Oracle Event Processing application designs but in general, Oracle Event Processing high availability is designed for the following use case:

- An application receives input events from one or more external systems.
- The external systems are publish-subscribe style systems that allow multiple instances of the application to connect simultaneously and receive the same stream of messages.
- The application does not update any external systems in a way that would cause conflicts when multiple copies of the application run concurrently.
- The application sends output events to an external downstream system. Multiple instances of the application can connect to the downstream system simultaneously, although only one instance of the application can send messages at any one time.

Within these constraints, the following different cases are of interest:

- The application is allowed to skip sending some output events to the downstream system when there is a failure. Duplicates are also allowed. Use the "Simple Failover" high availability quality of service option.
- The application is allowed to send duplicate events to the downstream system, but must not skip any events when there is a failure. Use the Simple Failover with Buffering and Light-Weight Queue Trimming high availability quality of service options.
- The application must send exactly the same stream of messages/events to the downstream system when there is a failure, modulo a brief pause during which events may not be sent when there is a failure. Use the Precise Recovery with JMS high availability quality of service option.

17.8.2 High Availability Design Patterns

When designing your Oracle Event Processing application for use with Oracle Event Processing high availability options, observe the following design patterns:

- Select the Minimum High Availability Your Application can Tolerate
- Use High Availability Components at All Ingress and Egress Points
- Preserve What You Need
- Limit Oracle Event Processing Application State
- Choose an Adequate warm-up-window-length Time
- Ensure Applications are Idempotent
- Source Event Identity Externally
- Understand the Importance of Event Ordering
- Write Oracle CQL Queries with High Availability in Mind
- Avoid Coupling Servers
- Plan for Server Recovery.

17.8.2.1 Select the Minimum High Availability Your Application can Tolerate

Be sure that the extra cost of precise recovery (per-stage throughput decrease) is actually necessary for your application.

17.8.2.2 Use High Availability Components at All Ingress and Egress Points

Use a high availability input adapter after each regular input adapter and use a high availability output adapter before each regular output adapter.

17.8.2.3 Preserve What You Need

Oracle Event Processing systems receive a large number of raw input events that are queried to generate a smaller number of relevant events. In general it makes sense to preserve the relevant event because there are fewer of them, and they are more valuable.

17.8.2.4 Limit Oracle Event Processing Application State

Oracle Event Processing systems enable you to query windows of events. It can be tempting to build systems with very large windows, but this increases the state that needs to be rebuilt when failure occurs. In general it is better to think of long-term state as something kept in stable storage, such as a distributed cache or a database to leverage the high availability facilities of these technologies.

17.8.2.5 Choose an Adequate warm-up-window-length Time

When a new Oracle Event Processing server is added to a high availability multiserver domain or when an existing failed server restarts, the server does not fully join the Oracle Event Processing high availability deployment and notification groups until all applications deployed to it have fully joined. The type of application determines when it fully joins.

Oracle Event Processing high availability applications can be described as Type 1 or Type 2 applications as Table 17-2 shows.

Table 17-2 Oracle Event Processing High Availability Application Types

Application Type	Must generate exactly the same sequence of output events?	Must be able to rebuild internal state by processing input streams within a finite period of time?	Must wait this period of time before it has fully joined?
Type 1	Yes	Yes	Yes
Type 2	No	No	No

For more information, see Rejoining the High Availability MultiServer Domain.

17.8.2.5.1 Type 1 Applications

A Type 1 application requires the new secondary server to generate exactly the same sequence of output events as existing secondary servers once it fully joins the high availability deployment and notification groups.

A Type 1 application must be able to rebuild its internal state by processing its input streams for a finite period of time (`warm-up-window-length` time), after which it generates the same stream of output events as other secondary servers running the application.

Configure the `warm-up-window-length` time on a high availability output adapter. Specify the length of the `warm-up-window-length` time length in seconds or minutes. For example, if the application contains Oracle CQL queries with range-

based windows of 5, 7, and 15 minutes, then the minimum `warm-up-window-length` time is 15 minutes (the maximum range-based window size). Oracle recommends that the maximum window length be padded with a few minutes to ensure that the necessary state is available. So, in the previous example 17 minutes or 20 minutes would be a good length for the `warm-up-window-length` time.

The server uses system time during the `warm-up-window-length` time period, so the server time is not directly correlated with the application time associated with events being processed.

Type 1 applications must only be interested in events that occurred during a finite amount of time. All range-based Oracle CQL windows must be shorter than the `warm-up-window-length` time and tuple-based windows must be qualified by time. For example, the application should only query the last 10 events if they occurred within the last five minutes. Applications that do not have this property cannot be Type 1 applications and cannot use the `warm-up-window-length` period.

For example, an application that uses an tuple-based partitioned window that has no time qualification cannot use the `warm-up-window-length` period because an arbitrary amount of time is required to rebuild the state of the window.

If a Type 1 application uses the high availability broadcast output adapter, it can trim events with a unique application-specific key, or a monotonic key like application time. Trimming events with application time is encouraged because it is more robust and less susceptible to bugs in the application that can cause an output event to not be generated.

For more information, see:

- Oracle CQL Query Restrictions
- Buffering Output Adapter
- Broadcast Output Adapter
- Correlating Output Adapter

17.8.2.5.2 Type 2 Applications

A Type 2 application does not require the new secondary server to generate the same sequence of output events as existing secondary servers once it fully joins the high availability deployment and notification groups. It simply requires that the new cluster member generate valid output events with respect to the point in time at which it begins processing input events.

A Type 2 application does not require a `warm-up-window-length` period.

Most applications are Type 2 applications. It is common for an application to be brought up at an arbitrary point in time (on the primary Oracle Event Processing server), begin processing events from input streams at that point, and generate valid output events. The input stream is not paused while the application starts and input events are constantly being generated and arriving. You can assume that in many cases a secondary stage that does the same thing, but at a slightly different time, also produces output events that are valid from the point of view of the application, although not necessarily identical to those events produced by the primary server because of slight timing differences.

For example, a financial application that only runs while the market is open might operate as a Type 2 application as follows: All servers can be brought up before the market opens and begin processing incoming events at the same point in the market data stream. Multiple secondary servers can be run to protect against failure, and if the

number of secondary servers is sufficient while the market is open, then do not restart secondary servers that fail or add additional secondary servers because no secondary server needs to recover its state.

17.8.2.6 Ensure Applications are Idempotent

You scan run two copies of an application on different servers and the copies do not conflict in a shared cache or database. If you use an external relation (such as a cache or table), then you must ensure that when a server rejoins the cluster, your application accesses the same cache or table as before. The application must join to the same external relation again. The data source defined on the server must not have been changed to ensure that you are pulling data from same data source.

17.8.2.7 Source Event Identity Externally

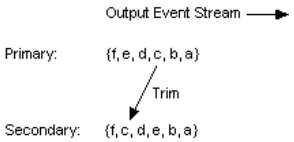
Many high availability solutions require that events be correlated between different servers. To correlate events, the events must be universally identifiable. The best way to make event universally identifiable is use external information, preferably a time stamp, to seed the event. For more information, see Prefer Application Time .

17.8.2.8 Understand the Importance of Event Ordering

Primary and secondary servers must generate the same output events and in exactly the same order when you choose high availability quality of service options that use queue trimming and equality-based event identify (non-monotonic event identifiers that do not increase continually). Generating output events in different orders can cause missed output events or unnecessary duplicate output events when a failure occurs.

Consider the output event streams shown in Figure 17-8. The primary server has output events `a`, `b`, and `c`. After outputting event `c`, the primary sends the secondary a queue trimming message.

Figure 17-8 Event Order



The secondary server trims all events in its queue generated prior to event `c` including event `c` itself. In this case, the set of events trimmed will be `{a, b, e, d, c}`, which is wrong because the primary server has not yet output events `d` and `e`. If a failover occurs after processing the trimming message for event `c`, events are lost.

17.8.2.8.1 Prefer Deterministic Behavior

For an application to generate events in the same order when run on multiple instances, it must be deterministic. The application must not rely on things like:

- Random number generators that can return different results on different machines.
- Methods like `System.getTimeMillis` or `System.nanoTime` that can return different results on different machines because the system clocks are not synchronized.

17.8.2.8.2 Avoid Multithreading

Because thread scheduling algorithms are very timing dependent, multithreading can be a source of nondeterministic behavior in applications. Different threads can be scheduled at different times on different machines.

For example, avoid creating an EPN in which multiple threads send events to a high availability adapter in parallel. If such a channel is an event source for a high availability adapter, it would cause events to be sent to the adapter in parallel by different threads and could make the event order nondeterministic. Also, do not send events to the mediator (JMS server) with multiple threads, which acts as an event source.

For more information on channel configuration to avoid, see:

- *Customizing Oracle Event Processing*
- `max-threads` in *Schema Reference for Oracle Event Processing*.

17.8.2.8.3 Prefer Monotonic Event Identifiers

Event identifiers can be monotonic or non-monotonic. A monotonic identifier is one that increases continually (such as a time value). A non-monotonic identifier does not increase continually and may contain duplicates.

In general, design your application with monotonic event identifiers. With a monotonic event identifier, the Oracle Event Processing high availability adapter can handle an application that may produce events out of order.

17.8.2.9 Write Oracle CQL Queries with High Availability in Mind

Not all Oracle CQL query usage is supported when using Oracle Event Processing high availability. You might need to redefine your Oracle CQL queries to address these restrictions. For more information, see Oracle CQL Query Restrictions .

17.8.2.10 Avoid Coupling Servers

The best high availability performance for Oracle Event Processing systems comes when servers can run without requiring coordination. Generally this can be done when there is no shared state, and the downstream system can tolerate duplicates. Increasing levels of high availability are targeted at increasing the fidelity of event stream that the downstream system sees, but the increase in fidelity has a performance penalty.

17.8.2.11 Plan for Server Recovery

When a secondary server rejoins the multiserver domain, the server needs time to rebuild the application state to match the current primary and active secondaries. See Choose an Adequate warm-up-window-length Time.

The time it takes for a secondary server to become available as an active secondary server after it rejoins the multiserver domain is a factor in the number of active secondary servers you require.

If a secondary is declared to be the new primary server before it is ready, the secondary server throw an exception.

17.8.3 Oracle CQL Query Restrictions

In a high availability application, Oracle CQL queries have the following restrictions. For more information about Oracle CQL and the topics covered in this section, see *Oracle CQL Language Reference for Oracle Event Processing*.

17.8.3.1 Range-Based Windows

In a Type 1 application where the application must generate exactly the same sequence of output events as existing secondaries, all range-based Oracle CQL windows must be shorter than the `warm-up-window-length` time. See Choose an Adequate warm-up-window-length Time.

Channels must use application time when Oracle CQL queries contain range-based Windows. See Prefer Application Time .

17.8.3.2 Tuple-Based Windows

In a Type 1 application where the application must generate exactly the same sequence of output events as existing secondaries, all tuple-based windows must be qualified by time. See Choose an Adequate warm-up-window-length Time.

17.8.3.3 Partitioned Windows

Avoid partitioned windows because there are situations in which a partition cannot be rebuilt. If you do use partitioned windows, configure a `warm-up-window-length` time long enough to give the Oracle Event Processing server time to rebuild the partition. See Choose an Adequate warm-up-window-length Time.

17.8.3.4 Sliding Windows

Oracle CQL queries should not use sliding windows when new stages that join the multiserver domain are expected to generate exactly the same output events as existing stages. See Rejoining the High Availability MultiServer Domain.

17.8.3.5 DURATION Clause and Non-Event Detection

You must use application time when Oracle CQL queries contain a `DURATION` clause for non-event detection. See Prefer Application Time .

17.8.3.6 Prefer Application Time

In Oracle Event Processing each event is associated with a point in time at which the event occurred. Oracle CQL recognizes two types of time:

- Application time: A time value assigned to each event outside of Oracle CQL by the application before the event enters the Oracle CQL processor.
- System time: A time value associated with an event when it arrives at the Oracle CQL processor, essentially by calling `System.nanoTime()`.

Application time is generally the best approach for applications that need to be highly available. The application time is associated with an event before the event is sent downstream, so it is consistent across active primary and secondary servers.

System time can cause application instances to generate different results because the time value associated with an event can be different on each server due to system clocks not being synchronized. You can use system time for applications when Oracle CQL queries do not use time-based windows. Applications that use only event-based

windows depend only on the arrival order of events rather than the arrival time, so you can use system time in this case.

If you must use system time with Oracle CQL queries that do use time-based windows, then you must use a Oracle Event Processing high availability input adapter that intercepts incoming events and assigns a consistent time that spans primary and secondary servers.

17.9 Configure High Availability Quality of Service

You configure Oracle Event Processing high availability quality of service in the assembly and component configuration files. After you make Oracle Event Processing high availability configuration changes, you must redeploy your Oracle Event Processing application.

This section includes the following procedures:

- Configure a Simple Failover
- Configure Simple Failover With Buffering
- Configure Light-Weight Queue Trimming
- Configure Precise Recovery With JMS.

17.9.1 Configure a Simple Failover

You configure simple failover with the Oracle Event Processing high availability buffering output adapter with a sliding window size of zero (0).

This procedure starts with the example EPN that Figure 17-9 shows and adds the required components to configure it for a simple failover.

Figure 17-9 Simple Failover EPN

Configure a simple failover:

1. Create a multiserver domain using Oracle Coherence.

   For more information, see *Administering Oracle Event Processing*.

2. Create an Oracle Event Processing application.

3. Edit the `MANIFEST.MF` file to add the following `Import-Package` entries:

   - `com.bea.wlevs.ede.api.cluster`
   - `com.oracle.cep.cluster.hagroups`
   - `com.oracle.cep.cluster.ha.adapter`
   - `com.oracle.cep.cluster.ha.api`

4. Configure your Oracle Event Processing application EPN assembly file to add an Oracle Event Processing high availability buffering output adapter as the following assembly file entries show.

- Add a `wlevs:adapter` element with `provider` set to `ha-buffering` after channel `helloworldOutputChannel`.
- Update the `wlevs:listener` element in channel `helloworldOutputChannel` to reference the `ha-buffering` adapter by its `id`.
- Add a `wlevs:listener` element to the `ha-buffering` adapter that references the `HelloWorldBean` class.

```
<wlevs:event-type-repository>
  <wlevs:event-type type-name="HelloWorldEvent">
    <wlevs:class>com.bea.wlevs.event.example.helloworld.HelloWorldEvent
    </wlevs:class>
  </wlevs:event-type>
</wlevs:event-type-repository>

<wlevs:adapter id="helloworldAdapter"
  class="com.bea.wlevs.adapter.example.helloworld.HelloWorldAdapter" >
<wlevs:instance-property name="message" value="HelloWorld -
  the current time is:"/>
</wlevs:adapter>

<wlevs:channel id="helloworldInputChannel" event-type="HelloWorldEvent" >
  <wlevs:listener ref="helloworldProcessor"/>
</wlevs:channel>

<wlevs:processor id="helloworldProcessor" />

<wlevs:channel id="helloworldOutputChannel" event-type="HelloWorldEvent"
  advertise="true">
  <wlevs:listener ref="myHaSlidingWindowAdapter"/>
  <wlevs:source ref="helloworldProcessor"/>
 </wlevs:channel>

 <wlevs:adapter id="myHaSlidingWindowAdapter" provider="ha-buffering" >
   <wlevs:listener>
     <bean class="com.bea.wlevs.example.helloworld.HelloWorldBean"/>
   </wlevs:listener>
 </wlevs:adapter>
```

5. Optionally, configure the channel downstream from the input adapter (`helloworldInputChannel`) to configure an application time stamp based on an appropriate event property as assembly file entries show.

   For simple failover, you can use system time stamps because events are not correlated between servers. However, it is possible that slightly different results might be output from the buffer if application time stamps are not used.

   In this example, event property `arrivalTime` is used.

   The `wlevs:expression` should be set to this event property.

```
<wlevs:channel id="helloworldInputChannel" event-type="HelloWorldEvent" >
   <wlevs:listener ref="helloworldProcessor"/>
   <wlevs:source ref="myHaInputAdapter"/>
   <wlevs:application-timestamped>
      <wlevs:expression>arrivalTime</wlevs:expression>
   </wlevs:application-timestamped>
 </wlevs:channel>
```

6. Configure the Oracle Event Processing high availability buffering output adapter.

   Set the instance property `windowLength` to zero (0) as shown.

```
<wlevs:adapter id="myHaSlidingWindowAdapter" provider="ha-buffering" >
   <wlevs:listener>
```

```
      <bean class="com.bea.wlevs.example.helloworld.HelloWorldBean"/>
    </wlevs:listener>
    <wlevs:instance-property name="windowLength" value="0"/>
</wlevs:adapter>
```

7. Optionally, configure the component configuration file to include the Oracle Event Processing high availability buffering output adapter as shown.

```
<processor>
  <name>helloworldProcessor</name>
  <rules>
    <query id="helloworldRule">
      <![CDATA[ select * from helloworldInputChannel [Now] >
    </query>
  </rules>
</processor>

<ha:ha-buffering-adapter >
  <name>myHaSlidingWindowAdapter</name>
  <window-length>0</window-length>
</ha:ha-buffering-adapter >
```

8. Deploy your application to the deployment group you created in step 1.

   Oracle Event Processing automatically selects one of the Oracle Event Processing servers as the primary.

17.9.2 Configure Simple Failover With Buffering

You configure simple failover using the Oracle Event Processing buffering output adapter with a sliding window size greater than zero (0).

This procedure starts with the example EPN that Figure 17-10 shows and adds the required components to configure it for simple failover with buffering.

Figure 17-10 Simple Failover With Buffering EPN

Configure simple failover with buffering:

1. Create a multiserver domain using Oracle Coherence.

   For more information, see *Administering Oracle Event Processing*.

2. Create an Oracle Event Processing application.

3. Edit the `MANIFEST.MF` file to add the following `Import-Package` entries:

   - `com.bea.wlevs.ede.api.cluster`
   - `com.oracle.cep.cluster.hagroups`
   - `com.oracle.cep.cluster.ha.adapter`
   - `com.oracle.cep.cluster.ha.api`

4. Configure your Oracle Event Processing application EPN assembly file to add an Oracle Event Processing high availability buffering output adapter as the following assembly file entries show.

   - Add a `wlevs:adapter` element with `provider` set to `ha-buffering` after channel `helloworldOutputChannel`.
   - Update the `wlevs:listener` element in channel `helloworldOutputChannel` to reference the `ha-buffering` adapter by its `id`.
   - Add a `wlevs:listener` element to the `ha-buffering` adapter that references the `HelloWorldBean` class.

```
<wlevs:event-type-repository>
  <wlevs:event-type type-name="HelloWorldEvent">
     <wlevs:class>com.bea.wlevs.event.example.helloworld.HelloWorldEvent</wlevs:class>
  </wlevs:event-type>
</wlevs:event-type-repository>

<wlevs:adapter id="helloworldAdapter"
  class="com.bea.wlevs.adapter.example.helloworld.HelloWorldAdapter" >
  <wlevs:instance-property name="message" value="HelloWorld - the current time is:"/>
</wlevs:adapter>

<wlevs:channel id="helloworldInputChannel" event-type="HelloWorldEvent" >
  <wlevs:listener ref="helloworldProcessor"/>
</wlevs:channel>

<wlevs:processor id="helloworldProcessor" />

<wlevs:channel id="helloworldOutputChannel" event-type="HelloWorldEvent" advertise="true">
   <wlevs:listener ref="myHaSlidingWindowAdapter"/>
   <wlevs:source ref="helloworldProcessor"/>
</wlevs:channel>

<wlevs:adapter id="myHaSlidingWindowAdapter" provider="ha-buffering" >
  <wlevs:listener>
     <bean class="com.bea.wlevs.example.helloworld.HelloWorldBean"/>
  </wlevs:listener>
</wlevs:adapter>
```

5. Optionally, configure the channel downstream from the input adapter (`helloworldInputChannel`) to configure an application time stamp based on an appropriate event property as shown.

   For simple failover with buffering, you can use system time stamps because events are not correlated between servers. However, it is possible that slightly different results might be output from the buffer if application time stamps are not used.

   In this example, event property `arrivalTime` is used.

   The `wlevs:expression` should be set to this event property.

```
<wlevs:channel id="helloworldInputChannel" event-type="HelloWorldEvent" >
  <wlevs:listener ref="helloworldProcessor"/>
  <wlevs:source ref="myHaInputAdapter"/>
  <wlevs:application-timestamped>
    <wlevs:expression>arrivalTime</wlevs:expression>
  </wlevs:application-timestamped>
</wlevs:channel>
```

6. Configure the Oracle Event Processing high availability buffering output adapter.

   Set the instance property `windowLength` to a value greater than zero (0) as shown.

```
<wlevs:adapter id="myHaSlidingWindowAdapter" provider="ha-buffering" >
  <wlevs:listener>
    <bean class="com.bea.wlevs.example.helloworld.HelloWorldBean"/>
  </wlevs:listener>
     <wlevs:instance-property name="windowLength" value="15000"/>
</wlevs:adapter>
```

7. Optionally, configure the component configuration file to include the Oracle Event Processing high availability buffering output adapter as shown.

```
<processor>
  <name>helloworldProcessor</name>
  <rules>
    <query id="helloworldRule">
      <![CDATA[ select * from helloworldInputChannel [Now] >
    </query>
  </rules>
</processor>

<ha:ha-buffering-adapter >
  <name>myHaSlidingWindowAdapter</name>
  <window-length>15000</window-length>
</ha:ha-buffering-adapter >
```

8. If your application is an Oracle Event Processing high availability Type 1 application where the application must generate exactly the same sequence of output events as existing secondaries) configure the `warm-up-window-length` for the buffering output adapter.

   For more information, see Choose an Adequate warm-up-window-length Time

9. Deploy your application to the deployment group you created in step 1.

   Oracle Event Processing automatically selects one of the Oracle Event Processing servers as the primary.

17.9.3 Configure Light-Weight Queue Trimming

You configure light-weight queue trimming using the Oracle Event Processing high availability input adapter and the broadcast output adapter.

This procedure starts with the example EPN that Figure 17-11 shows and adds the required components to configure it for light-weight queue trimming.

Figure 17-11 Light-Weight Queue Trimming EPN

Configure light-weight queue trimming:

1. Create a multiserver domain using Oracle Coherence.

   For more information, see *Administering Oracle Event Processing*.

2. Create an Oracle Event Processing application.

3. Edit the `MANIFEST.MF` file to add the following `Import-Package` entries:

   - `com.bea.wlevs.ede.api.cluster`

- `com.oracle.cep.cluster.hagroups`
- `com.oracle.cep.cluster.ha.adapter`
- `com.oracle.cep.cluster.ha.api`

4. Configure your Oracle Event Processing application EPN assembly file to add an Oracle Event Processing high availability input adapter as the following example shows.

   - Add a `wlevs:adapter` element with `provider` set to `ha-inbound` after the regular input adapter `helloworldAdapter`.
   - Add a `wlevs:listener` element to the regular input adapter `helloworldAdapter` that references the `ha-inbound` adapter by its `id`.
   - Add a `wlevs:source` element to the `helloworldInputChannel` that references the `ha-inbound` adapter by its `id`.

```
    <wlevs:event-type-repository>
        <wlevs:event-type type-name="HelloWorldEvent">
            <wlevs:class>com.bea.wlevs.event.example.helloworld.HelloWorldEvent</
wlevs:class>
  </wlevs:event-type>
</wlevs:event-type-repository>

<wlevs:adapter id="helloworldAdapter"
  class="com.bea.wlevs.adapter.example.helloworld.HelloWorldAdapter" >
<wlevs:instance-property name="message" value="HelloWorld -
  the current time is:"/>
   <wlevs:listener ref="myHaInputAdapter"/>
</wlevs:adapter>

<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
</wlevs:adapter>

<wlevs:channel id="helloworldInputChannel" event-type="HelloWorldEvent" >
  <wlevs:listener ref="helloworldProcessor"/>
  <wlevs:source ref="myHaInputAdapter"/>
</wlevs:channel>

<wlevs:processor id="helloworldProcessor" />

<wlevs:channel id="helloworldOutputChannel" event-type="HelloWorldEvent"
  advertise="true">
  <wlevs:listener>
    <bean class="com.bea.wlevs.example.helloworld.HelloWorldBean"/>
  </wlevs:listener>
    <wlevs:source ref="helloworldProcessor"/>
  </wlevs:channel>
```

5. Configure your Oracle Event Processing application EPN assembly file to add an Oracle Event Processing high availability broadcast output adapter as shown.

   - Add a `wlevs:adapter` element with `provider` set to `ha-broadcast` after channel `helloworldOutputChannel`.
   - Update the `wlevs:listener` element in channel `helloworldOutputChannel` to reference the `ha-broadcast` adapter by its `id`.
   - Add a `wlevs:listener` element to the `ha-broadcast` adapter that references the `HelloWorldBean` class.

```
<wlevs:event-type-repository>
  <wlevs:event-type type-name="HelloWorldEvent">
    <wlevs:class>com.bea.wlevs.event.example.helloworld.HelloWorldEvent
    </wlevs:class>
  </wlevs:event-type>
</wlevs:event-type-repository>

<wlevs:adapter id="helloworldAdapter"
   class="com.bea.wlevs.adapter.example.helloworld.HelloWorldAdapter" >
 <wlevs:instance-property name="message" value="HelloWorld -
    the current time is:"/>
  <wlevs:listener ref="myHaInputAdapter"/>
</wlevs:adapter>

<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
</wlevs:adapter>

<wlevs:channel id="helloworldInputChannel" event-type="HelloWorldEvent" >
  <wlevs:listener ref="helloworldProcessor"/>
  <wlevs:source ref="myHaInputAdapter"/>
</wlevs:channel>

<wlevs:processor id="helloworldProcessor" />

<wlevs:channel id="helloworldOutputChannel" event-type="HelloWorldEvent"
  advertise="true">
  <wlevs:listener ref="myHaBroadcastAdapter"/>
  <wlevs:source ref="helloworldProcessor"/>
</wlevs:channel>

<wlevs:adapter id="myHaBroadcastAdapter" provider="ha-broadcast" >
  <wlevs:listener>
    <bean class="com.bea.wlevs.example.helloworld.HelloWorldBean"/>
  </wlevs:listener>
</wlevs:adapter>
```

6. Configure the Oracle Event Processing high availability input adapter.

   Consider using one of the following example configurations.

   This example shows a high availability input adapter configuration using all defaults. The mandatory key is based on all event properties and the event property that the high availability input adapter assigns a time value to is an event property named `arrivalTime`.

```
<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
  <wlevs:instance-property name="timeProperty" value="arrivalTime"/>
</wlevs:adapter>
```

   This example shows a high availability input adapter configuration using all defaults. The mandatory key is based on all event properties and the event property that the high availability input adapter assigns a time value to is an event property named `arrivalTime`. Because the events are tuple-based events, you must specify the event type (`MyEventType`) using the `eventType` property.

```
<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
   <wlevs:instance-property name="timeProperty" value="arrivalTime"/>
   <wlevs:instance-property name="eventType" value="MyEventType"/>
</wlevs:adapter>
```

   This example shows a high availability input adapter configuration where the mandatory key is based on one event property (named `id`) and the event property that the high availability input adapter assigns a time value to is an event property named `arrivalTime`.

```
<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
  <wlevs:instance-property name="keyProperties" value="id"/>
  <wlevs:instance-property name="timeProperty" value="arrivalTime"/>
</wlevs:adapter>
```

This example shows a high availability input adapter configuration where the mandatory key is based on more than one event property (properties `orderID` and `accountID`) and the event property that the high availability input adapter assigns a time value to is an event property named `arrivalTime`.

```
<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
  <wlevs:instance-property name="timeProperty" value="arrivalTime"/>
  <wlevs:instance-property name="keyClass" value="com.acme.MyCompoundKeyClass"/>
</wlevs:adapter>
```

A compound key Java class (`com.acme.MyCompoundKeyClass`) is mandatory and its implementation is shown. The `hashCode` and `equals` methods are required. When you specify a `keyClass`, the `keyProperties` instance property is ignored: Oracle Event Processing assumes that the compound key is based on all the getter methods in the `keyClass`.

```
package com.acme;

public class MyCompoundKeyClass {
    private int orderID;
    private int accountID;

    public MyCompoundKeyClass() {}

    public int getOrderID() {
        return orderID;
    }
    public setOrderID(int orderID) {
        this.orderID = orderID;
    }
    public int getAccountID() {
        return accountID;
    }
    public setOrderID(int accountID) {
        this.accountID = accountID;
    }

    public int hashCode() {
    int hash = 1;
    hash = hash * 31 + orderID.hashCode();
    hash = hash * 31 + (accountID == null ? 0 : accountID.hashCode());
    return hash;
    }

    public boolean equals(Object obj) {
        if (obj == this) return true;
        if (obj == null) return false;
        if (!(obj instanceof MyCompoundKeyClass)) return false;
        MyCompoundKeyClass k = (MyCompoundKeyClass) obj;
        return k.accountID == accountID && k.orderID == orderID;
    }
}
```

7. Configure the channel downstream from the high availability input adapter (`helloworldInputChannel`) to configure an application time stamp based on the high availability input adapter `timeProperty` setting as the following example shows.

   The `wlevs:expression` should be set to the `timeProperty` value.

```
<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
   <wlevs:instance-property name="keyProperties" value="id"/>
```

```
   <wlevs:instance-property name="eventType" value="HelloWorldEvent"/>
   <wlevs:instance-property name="timeProperty" value="arrivalTime"/>
</wlevs:adapter>

<wlevs:channel id="helloworldInputChannel" event-type="HelloWorldEvent" >
  <wlevs:listener ref="helloworldProcessor"/>
  <wlevs:source ref="myHaInputAdapter"/>
  <wlevs:application-timestamped>
    <wlevs:expression>arrivalTime</wlevs:expression>
  </wlevs:application-timestamped>
</wlevs:channel>
```

8. Configure the Oracle Event Processing high availability broadcast output adapter.

   Consider using one of the following example configurations:

   This example shows a broadcast output adapter configuration using all defaults. The mandatory key is based on all event properties, key values are non-monotonic (do not increase continually) and total order (unique).

```
<wlevs:adapter id="myHaSlidingWindowAdapter" provider="ha-broadcast" >
   <wlevs:listener>
      <bean class="com.bea.wlevs.example.helloworld.HelloWorldBean"/>
   </wlevs:listener>
</wlevs:adapter>
```

   This example shows a broadcast output adapter configuration where the mandatory key is based on one event property (named `timeProperty`), key values are monotonic (they do increase continually) and not total order (not unique).

```
<wlevs:adapter id="myHaSlidingWindowAdapter" provider="ha-broadcast" >
   <wlevs:listener>
     <bean class="com.bea.wlevs.example.helloworld.HelloWorldBean"/>
   </wlevs:listener>
     <wlevs:instance-property name="keyProperties" value="timeProperty"/>
     <wlevs:instance-property name="monotonic" value="true"/>
     <wlevs:instance-property name="totalOrder" value="false"/>
  </wlevs:adapter>
```

   This example shows a broadcast output adapter configuration where the mandatory key is based on more than one event property (properties `timeProperty` and `accountID`), key values are monotonic (they do increase continually) and total order (unique).

```
<wlevs:adapter id="myHaSlidingWindowAdapter" provider="ha-broadcast" >
 <wlevs:listener>
    <bean class="com.bea.wlevs.example.helloworld.HelloWorldBean"/>
  </wlevs:listener>
  <wlevs:instance-property name="keyClass" value="com.acme.MyCompoundKeyClass"/>
  <wlevs:instance-property name="monotonic" value="true"/>
  <wlevs:instance-property name="totalOrder" value="true"/>
</wlevs:adapter>
```

   A compound key Java class (`com.acme.MyCompoundKeyClass`) is mandatory and its implementation is shown in the following example. The `hashCode` and `equals` methods are required. When you specify a `keyClass`, the `keyProperties` instance property is ignored: Oracle Event Processing assumes that the compound key is based on all the getter methods in the `keyClass`.

```
package com.acme;

public class MyCompoundKeyClass {
    private int timeProperty;
    private int accountID;
```

```
    public MyCompoundKeyClass() {}

    public int getTimeProperty() {
        return orderID;
    }
    public setTimeProperty(int timeProperty) {
        this.timeProperty = timeProperty;
    }
    public int getAccountID() {
        return accountID;
    }
    public setOrderID(int accountID) {
        this.accountID = accountID;
    }

    public int hashCode() {
    int hash = 1;
    hash = hash * 31 + timeProperty.hashCode();
    hash = hash * 31 + (accountID == null ? 0 : accountID.hashCode());
    return hash;
    }

    public boolean equals(Object obj) {
        if (obj == this) return true;
        if (obj == null) return false;
        if (!(obj instanceof MyCompoundKeyClass)) return false;
        MyCompoundKeyClass k = (MyCompoundKeyClass) obj;
        return k.accountID == accountID && k.orderID == orderID;
    }
}
```

9. Optionally, configure the component configuration file to include the Oracle Event Processing high availability input adapter and buffering output adapter as shown.

```
<processor>
  <name>helloworldProcessor</name>
  <rules>
    <query id="helloworldRule">
      <![CDATA[ select * from helloworldInputChannel [Now] >
    </query>
  </rules>
</processor>

<ha:ha-inbound-adapter>
 <name>myHaInputAdapter</name>
</ha:ha-inbound-adapter>

<ha:ha-broadcast-adapter>
  <name>myHaBroadcastAdapter</name>
  <trimming-interval units="events">10</trimming-interval>
</ha:ha-broadcast-adapter>
```

10. If your application is an Oracle Event Processing high availability Type 1 application where the application must generate exactly the same sequence of output events as existing secondaries, configure the `warm-up-window-length` for the broadcast output adapter.

11. Oracle Event Processing automatically selects one of the Oracle Event Processing servers as the primary.

17.9.4 Configure Precise Recovery With JMS

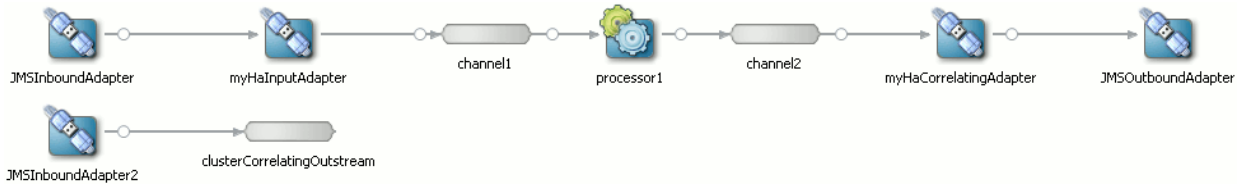
You configure precise recovery with JMS using the Oracle Event Processing high availability input adapter and correlating output adapter.

This procedure describes how to create the example EPN that Figure 17-12 shows. For more information about this Oracle Event Processing high availability quality of service, see Precise Recovery with JMS.

Note:

The JMS destination used by JMS adapters for precise recovery must be topics, rather than queues.

Figure 17-12 Precise Recovery With JMS EPN



Configure precise recovery with JMS:

1. Create a multiserver domain using Oracle Coherence.

   For more information, see *Administering Oracle Event Processing*.

2. Create an Oracle Event Processing application.

3. Edit the `MANIFEST.MF` file to add the following `Import-Package` entries:

   - `com.bea.wlevs.ede.api.cluster`
   - `com.oracle.cep.cluster.hagroups`
   - `com.oracle.cep.cluster.ha.adapter`
   - `com.oracle.cep.cluster.ha.api`

4. Configure your Oracle Event Processing application EPN assembly file to add an Oracle Event Processing high availability input adapter as shown:

   - Add a `wlevs:adapter` element with `provider` set to `ha-inbound` after the regular input adapter `JMSInboundAdapter`.
   - Add a `wlevs:listener` element to the regular input adapter `JMSInboundAdapter` that references the `ha-inbound` adapter by its `id`.
   - Add a `wlevs:source` element to the channel `channel1` that references the `ha-inbound` adapter by its `id`.

```
<wlevs:event-type-repository>
    <wlevs:event-type type-name="StockTick">
        <wlevs:properties>
            <wlevs:property name="lastPrice" type="double" />
            <wlevs:property name="symbol" type="char" />
        </wlevs:properties>
    </wlevs:event-type>
</wlevs:event-type-repository>

<wlevs:adapter id="JMSInboundAdapter" provider="jms-inbound">
    <wlevs:listener ref="myHaInputAdapter"/>
</wlevs:adapter>
```

```
<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
</wlevs:adapter>

<wlevs:channel id="channel1" event-type="StockTick">
    <wlevs:listener ref="processor1" />
    <wlevs:source ref="myHaInputAdapter"/>
</wlevs:channel>
```

5. Configure your Oracle Event Processing application EPN assembly file to add an Oracle Event Processing high availability correlating output adapter as shown.

   - Add a `wlevs:adapter` element with `provider` set to `ha-correlating` after channel `channel2`.
   - Update the `wlevs:listener` element in channel `channel2` to reference the `ha-correlating` adapter by its `id`.
   - Add a `wlevs:listener` element to the `ha-correlating` adapter that references the regular output adapter `JMSOutboundAdapter`.

```
<wlevs:event-type-repository>
  <wlevs:event-type type-name="StockTick">
    <wlevs:properties>
      <wlevs:property name="lastPrice" type="double" />
      <wlevs:property name="symbol" type="char" />
    </wlevs:properties>
  </wlevs:event-type>
</wlevs:event-type-repository>

<wlevs:adapter id="JMSInboundAdapter" provider="jms-inbound">
  <wlevs:listener ref="myHaInputAdapter"/>
</wlevs:adapter>

<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
</wlevs:adapter>

<wlevs:channel id="channel1" event-type="StockTick">
  <wlevs:listener ref="processor1" />
  <wlevs:source ref="myHaInputAdapter"/>
</wlevs:channel>

<wlevs:processor id="processor1">
  <wlevs:listener ref="channel2" />
</wlevs:processor>

<wlevs:channel id="channel2" event-type="StockTick">
  <wlevs:listener ref="myHaCorrelatingAdapter" />
</wlevs:channel>

<wlevs:adapter id="myHaCorrelatingAdapter" provider="ha-correlating" >
  <wlevs:listener ref="JMSOutboundAdapter"/>
</wlevs:adapter>

<wlevs:adapter id="JMSOutboundAdapter" provider="jms-outbound">
</wlevs:adapter>
```

6. Configure the Oracle Event Processing high availability input adapter.

   Consider using one of the following example configurations:

   This example shows a high availability input adapter configuration using all defaults. The mandatory key is based on all event properties and the event property that the high availability input adapter assigns a time value to is an event property named `arrivalTime`.

```
<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
  <wlevs:instance-property name="timeProperty" value="arrivalTime"/>
</wlevs:adapter>
```

This example shows a high availability input adapter configuration using all defaults. The mandatory key is based on all event properties and the event property that the high availability input adapter assigns a time value to is an event property named `arrivalTime`. Because the events are tuple-based events, you must specify the event type (`MyEventType`) using the `eventType` property.

```
<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
  <wlevs:instance-property name="timeProperty" value="arrivalTime"/>
  <wlevs:instance-property name="eventType" value="MyEventType"/>
</wlevs:adapter>
```

This example shows a high availability input adapter configuration where the mandatory key is based on one event property (named `sequenceNo`) and the event property that the high availability input adapter assigns a time value to is an event property named `inboundTime`.

```
<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
  <wlevs:instance-property name="keyProperties" value="sequenceNo"/>
  <wlevs:instance-property name="timeProperty" value="inboundTime"/>
</wlevs:adapter>
```

This example shows a high availability input adapter configuration where the mandatory key is based on more than one event property (properties `orderID` and `accountID`) and the event property that the high availability input adapter assigns a time value to is an event property named `arrivalTime`.

A compound key Java class (`com.acme.MyCompoundKeyClass`) is mandatory and its implementation is shown. The `hashCode` and `equals` methods are required. When you specify a `keyClass`, the `keyProperties` instance property is ignored: Oracle Event Processing assumes that the compound key is based on all the getter methods in the `keyClass`.

```
<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
  <wlevs:instance-property name="timeProperty" value="arrivalTime"/>
  <wlevs:instance-property name="keyClass" value="com.acme.MyCompoundKeyClass"/>
</wlevs:adapter>

package com.acme;

public class MyCompoundKeyClass {
    private int orderID;
    private int accountID;

    public MyCompoundKeyClass() {}

    public int getOrderID() {
        return orderID;
    }
    public setOrderID(int orderID) {
        this.orderID = orderID;
    }
    public int getAccountID() {
        return accountID;
    }
    public setOrderID(int accountID) {
        this.accountID = accountID;
    }

    public int hashCode() {
    int hash = 1;
    hash = hash * 31 + orderID.hashCode();
    hash = hash * 31 + (accountID == null ? 0 : accountID.hashCode());
```

```
    return hash;
    }

    public boolean equals(Object obj) {
        if (obj == this) return true;
        if (obj == null) return false;
        if (!(obj instanceof MyCompoundKeyClass)) return false;
        MyCompoundKeyClass k = (MyCompoundKeyClass) obj;
        return k.accountID == accountID && k.orderID == orderID;
    }
}
```

7. Configure the channel downstream from the high availability input adapter (`channel1`) to configure an application time stamp based on the high availability input adapter `timeProperty` setting as the following example shows.

   The `wlevs:expression` should be set to the `timeProperty` value.

```
<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
  <wlevs:instance-property name="eventType" value="HelloWorldEvent"/>
  <wlevs:instance-property name="keyProperties" value="sequenceNo"/>
  <wlevs:instance-property name="timeProperty" value="inboundTime"/>
</wlevs:adapter>

<wlevs:channel id="channel1" event-type="StockTick">
  <wlevs:listener ref="processor1" />
  <wlevs:source ref="myHaInputAdapter"/>
  <wlevs:application-timestamped>
    <wlevs:expression>inboundTime</wlevs:expression>
  </wlevs:application-timestamped>
</wlevs:channel>
```

8. Configure the Oracle Event Processing high availability correlating output adapter `failOverDelay`.

   The following example shows a correlating output adapter configuration where the `failOverDelay` is 2000 milliseconds.

```
<wlevs:adapter id="myHaCorrelatingAdapter" provider="ha-correlating" >
  <wlevs:listener ref="JMSOutboundAdapter"/>
  <wlevs:instance-property name="failOverDelay" value="2000"/>
</wlevs:adapter>
```

9. Create a second regular JMS input adapter.

   The following example shows a JMS adapter named `JMSInboundAdapter2`.

```
    <wlevs:adapter id="JMSInboundAdapter2" provider="jms-inbound">
    </wlevs:adapter>
```

   The following JMS input adapter must be configured identically to the first JMS input adapter (in this example, `JMSInboundAdapter`). The following example shows the component configuration file for both the JMS input adapters. Note that both have exactly the same configuration, including the same provider.

```
    <jms-adapter>
        <name>JMSInboundAdapter</name>
        <jndi-provider-url>t3://localhost:7001</jndi-provider-url>
        <destination-jndi-name>./Topic1</destination-jndi-name>
        <user>weblogic</user>
        <password>weblogic</password>
        <work-manager>JettyWorkManager</work-manager>
        <concurrent-consumers>1</concurrent-consumers>
    </jms-adapter>

    <jms-adapter>
        <name>JMSInboundAdapter2</name>
```

```
        <jndi-provider-url>t3://localhost:7001</jndi-provider-url>
        <destination-jndi-name>./Topic2</destination-jndi-name>
        <user>weblogic</user>
        <password>weblogic</password>
        <work-manager>JettyWorkManager</work-manager>
        <concurrent-consumers>1</concurrent-consumers>
    </jms-adapter>
    ...
</wlevs:config>
```

10. Create a channel to function as the correlated source.

    You must configure this channel with the second regular JMS input adapter as its source.

    The following example shows a correlated source named `clusterCorrelatingOutstream` whose source is `JMSInboundAdapter2`.

```
    <wlevs:adapter id="JMSInboundAdapter2" provider="jms-inbound">
    </wlevs:adapter>

    <wlevs:channel id="clusterCorrelatingOutstream" event-type="StockTick" advertise="true">
        <wlevs:source ref="JMSInboundAdapter2"/>
    </wlevs:channel>
```

11. Configure the Oracle Event Processing high availability correlating output adapter with the `correlatedSource`.

    The following example shows a correlating output adapter configuration where the `correlatedSource` is `clusterCorrelatingOutstream`.

```
<wlevs:adapter id="myHaCorrelatingAdapter" provider="ha-correlating" >
  <wlevs:listener ref="JMSOutboundAdapter"/>
  <wlevs:instance-property name="failOverDelay" value="2000"/>
  <wlevs:instance-property name="correlatedSource"      ref="clusterCorrelatingOutstream"/>
</wlevs:adapter>
```

12. If your application is an Oracle Event Processing high availability Type 1 application where the application must generate exactly the same sequence of output events as existing secondaries, configure the `warm-up-window-length` for the correlating output adapter.

13. Configure the component configuration file to enable `session-transacted` for both inbound JMS adapters and the outbound JMS adapter as the following example shows:

```
    <jms-adapter>
        <name>JMSInboundAdapter</name>
        <jndi-provider-url>t3://localhost:7001</jndi-provider-url>
        <destination-jndi-name>./Topic1</destination-jndi-name>
        <user>weblogic</user>
        <password>weblogic</password>
        <work-manager>JettyWorkManager</work-manager>
        <concurrent-consumers>1</concurrent-consumers>
        <session-transacted>true</session-transacted>
    </jms-adapter>

    <jms-adapter>
        <name>JMSInboundAdapter2</name>
        <jndi-provider-url>t3://localhost:7001</jndi-provider-url>
        <destination-jndi-name>./Topic2</destination-jndi-name>
        <user>weblogic</user>
        <password>weblogic</password>
        <work-manager>JettyWorkManager</work-manager>
        <concurrent-consumers>1</concurrent-consumers>
        <session-transacted>true</session-transacted>
```

```
    </jms-adapter>
    ...
    <jms-adapter>
        <name>JMSOutboundAdapter</name>
        <event-type>JMSEvent</event-type>
        <jndi-provider-url>t3://localhost:7001</jndi-provider-url>
        <destination-jndi-name>./Topic2</destination-jndi-name>
        <delivery-mode>nonpersistent</delivery-mode>
        <session-transacted>true</session-transacted>
    </jms-adapter>
    ...
</wlevs:config>
```

14. Optionally, configure the component configuration file to include the Oracle Event Processing high availability input adapter and correlating output adapter as shown.

```
<ha:ha-inbound-adapter>
    <name>myHaInputAdapter</name>
</ha:ha-inbound-adapter>
...
<ha:ha-correlating-adapter>
    <name>myHaBroadcastAdapter</name>
    <fail-over-delay>2000</fail-over-delay>
</ha:ha-correlating-adapter>
```

15. Optionally, add an `ActiveActiveGroupBean` to your EPN to improve scalability.

    For more information, see Partition an Incoming JMS Event Stream.

16. Oracle Event Processing automatically selects one of the Oracle Event Processing servers as the primary.

17.10 Configure High Availability Adapters

You configure Oracle Event Processing high availability adapters in the EPN assembly file and component configuration files, similar to how you configure other components in the EPN, such as channels or processors.

After making any Oracle Event Processing high availability configuration changes, you must redeploy your Oracle Event Processing application. See Deploy an OSGi Bundle.

This section includes the following procedures:

- Configure the High Availability Input Adapter
- Configure the Buffering Output Adapter
- Configure the Broadcast Output Adapter
- Configure the Correlating Output Adapter.

17.10.1 Configure the High Availability Input Adapter

The Oracle Event Processing high availability broadcast input adapter is implemented by the `BroadcastInputAdapter` interface.

Assembly File

The root element to declare an Oracle Event Processing high availability input adapter is `wlevs:adapter` with the `provider` element set to `ha-inbound`. You also specify a wlevs:listener element for the Oracle Event Processing high availability input adapter in the input adapter.

```
<wlevs:adapter id="jmsAdapter" provider="jms-inbound"
    <wlevs:listener ref="myHaInputAdapter"/>
</wlevs:adapter>

<wlevs:adapter id="myHaInputAdapter" provider="ha-inbound">
    <wlevs:instance-property name="keyProperties" value="id"/>
    <wlevs:instance-property name="timeProperty" value="arrivalTime"/>
    <wlevs:instance-property name="eventType" value="MyEventType"/>
</wlevs:adapter>

<wlevs:channel id="inputChannel" event-type="MyEventType ">
    <wlevs:source ref="myHaInputAdapter"/>
    <wlevs:application-timestamped>
        <wlevs:expression>arrivalTime</wlevs:expression>
    </wlevs:application-timestamped>
</wlevs:channel>
```

Table 17-3 describes the additional child element.

Table 17-3 Child Elements of wlevs:adapter for the High Availability Input Adapter

Child Element	Description
`wlevs:instance-property`	Specify one or more `instance-property` element `name` and `value` attributes.

Table 17-4 lists the supported instance properties with their name and value attributes.

Table 17-4 High Availability Input Adapter Instance Properties

Name	Value
`timeProperty`	Specify the name of the event property to which the high availability input adapter assigns a time value. This is the same property that you use in the `wlevs:application-timestamped` element of the downstream EPN component to which the high availability input adapter is connected.
`keyProperties`	Specify a space delimited list of one or more event properties that the Oracle Event Processing high availability input adapter uses to identify event instances. If you specify more than one event property, you must specify a keyClass. Default: all event properties.
`keyClass`	Specify the fully qualified Java class name of to use for the compound key. By default, all JavaBean properties in the `keyClass` are assumed to be `keyProperties`, unless the `keyProperties` setting is used.

Table 17-4 (Cont.) High Availability Input Adapter Instance Properties

Name	Value
`eventType`	Specify the type name of the events that the Oracle Event Processing high availability input adapter receives from the actual input adapter. This is the same event type that you use in the downstream EPN component to which the high availability input adapter is connected. For tuple events, this property is mandatory. For all other Java class-based event types, this property is optional.

Configuration File

The root element for configuring an Oracle Event Processing high availability input adapter is `ha-inbound-adapter`. The `name` child element must match the `id` attribute of the corresponding `wlevs:adapter` element in the assembly file as shown.

```
<ha:ha-inbound-adapter>
    <name>myHaInputAdapter</name>
    <heartbeat units="millis">1000</heartbeat>
    <batch-size>10</batch-size>
</ha:ha-inbound-adapter>
```

Table 17-5 describes the additional child elements.

Table 17-5 Child Elements

Child Element	Description
`heartbeat`	Specify the length of time that the high availability input adapter can be idle before it generates a heartbeat event to advance time. Valid integer values for the `units` attribute: • `nanos`: wait the specified number of nanoseconds. • `millis`: wait the specified number of milliseconds. • `secs`: wait the specified number of seconds. Default: Heartbeats are not sent.
`batch-size`	Specify the number of events in each timing message that the primary broadcasts to its secondaries. A value of `n` means that `n {key, time}` pairs are sent in each message. You can use this property for performance tuning. Default: 1 (disable batching).

17.10.2 Configure the Buffering Output Adapter

The Oracle Event Processing high availability buffering output adapter is implemented by the `SlidingWindowQueueTrimmingAdapter` interface.

Assembly File

The root element for declaring an Oracle Event Processing high availability buffering output adapter is `wlevs:adapter` with `provider` element set to `ha-buffering` as the following example shows.

```
<wlevs:adapter id="mySlidingWindowingAdapter" provider ="ha-buffering">
  <wlevs:listener>
     <bean class="com.bea.wlevs.example.cluster.ClusterAdapterBean"/>
  </wlevs:listener>
  <wlevs:instance-property name="windowLength" value="15000"/>
</wlevs:adapter>
```

Table 17-6 describes the additional child elements.

Table 17-6 Child Elements

Child Element	Description
`wlevs:listener`	Specify the regular output adapter downstream from this Oracle Event Processing high availability buffering output adapter.
`wlevs:instance-property`	Specify one or more `instance-property` element `name` and `value` attributes as Table 17-7 describes.

Table 17-7 lists the instance properties.

Table 17-7 Instance Properties

Name	Value
`windowLength`	Specify the sliding window size as an integer number of milliseconds. Default: `15000`.

Configuration File

The root element for configuring an Oracle Event Processing high availability buffering output adapter is `ha-buffering-adapter`. The `name` child element for a particular adapter must match the `id` attribute of the corresponding `wlevs:adapter` element in the assembly file as shown.

```
<ha:ha-buffering-adapter >
    <name>mySlidingWindowingAdapter</name>
    <window-length>15000</window-length>
    <warm-up-window-length units="minutes">6</warm-up-window-length>
</ha:ha-buffering-adapter >
```

Table 17-8 describes the additional child elements of `ha-buffering-adapter` you can configure for an Oracle Event Processing high availability buffering output adapter.

Table 17-8 Child Elements

Child Element	Description
`window-length`	Specify the sliding window size as an integer number of milliseconds. Default: `15000`.

Table 17-8 (Cont.) Child Elements

Child Element	Description
`warm-up-window-length`	Specify the length of time that the high availability input adapter can be idle before it generates a heartbeat event to advance tim Valid integer values for the `units` attribute: • `seconds`: wait the specified number of seconds. • `minutes`: wait the specified number of minutes. Default: `units` is `seconds`.

17.10.3 Configure the Broadcast Output Adapter

The Oracle Event Processing high availability broadcast output adapter is implemented by the `GroupBroadcastQueueTrimmingAdapter` class.

Assembly File

The root element to declare an Oracle Event Processing high availability broadcast output adapter is `wlevs:adapter` with `the provider` element set to `ha-broadcast` as shown.

```
<wlevs:adapter id="myBroadcastAdapter" provider="ha-broadcast">
    <wlevs:listener ref="actualAdapter"/>
    <wlevs:instance-property name="keyProperties" value="time"/>
    <wlevs:instance-property name="monotonic" value="true"/>
</wlevs:adapter>
```

Table 17-9 describes the additional child elements.

Table 17-9 Child Elements

Child Element	Description
`wlevs:listener`	Specify the regular output adapter downstream from this Oracle Event Processing high availability broadcast output adapter.
`wlevs:instance-property`	Specify one or more `instance-property` element `name` and `value` attributes as Table 17-10 describes.

Table 17-10 lists the instance properties.

Table 17-10 Instance Properties

Name	Value
`keyProperties`	Specify a space delimited list of one or more event properties that the Oracle Event Processing high availability broadcast output adapter uses to identify event instances. If you specify more than one event property, you must specify a keyClass. Default: all event properties.

Table 17-10 (Cont.) Instance Properties

Name	Value
`keyClass`	Specify the fully qualified class name of a Java class used as a compound key. By default, all JavaBean properties in the `keyClass` are assumed to be `keyProperties`, unless the `keyProperties` setting is used. A compound key may be monotonic and may be totalOrder.
`monotonic`	Specify whether the key value is constantly increasing (like a time value). Valid values are: • `true`: the key is constantly increasing. • `false`: the key is not constantly increasing. Default: `false`.
`totalOrder`	Specify whether event keys are unique. Applicable only when instance property monotonic is set to `true`. Valid values are: • `true`: event keys are unique. • `false`: event keys are not unique. Default: `true`.

Configuration File

The root element for configuring an Oracle Event Processing high availability broadcast output adapter is `ha-broadcast-adapter`. The `name` child element for a particular adapter must match the `id` attribute of the corresponding `wlevs:adapter` element in the EPN assembly file that declares this adapter as shown.

```
<ha:ha-broadcast-adapter>
    <name>myBroadcastAdapter</name>
    <trimming-interval units="events">10</trimming-interval>
    <warm-up-window-length units="minutes">6</warm-up-window-length>
</ha:ha-broadcast-adapter>
```

Table 17-11 describes the additional child elements.

Table 17-11 Child Elements

Child Element	Description
`trimming-interval`	Specify the interval at which trimming messages are broadcast as an integer number of `units`. You can use this property for performance tuning (see High Availability Performance Tuning). Valid values for attribute `units`: • `events`: broadcast trimming messages after the specified number of events are processed. • `millis`: broadcast trimming messages after the specified number milliseconds. Default: `units` is `events`.

Table 17-11 (Cont.) Child Elements

Child Element	Description
`warm-up-window-length`	Specify the length of time it takes the application to rebuild state after a previously failed secondary restarts or a new secondary is added. Valid integer values for the `units` attribute: • `seconds`: wait the specified number of seconds. • `minutes`: wait the specified number of minutes. Default: `units` is `seconds`. For more information, see Choose an Adequate warm-up-window-length Time.

17.10.4 Configure the Correlating Output Adapter

The Oracle Event Processing high availability correlating output adapter is implemented by the `CorrelatedQueueTrimmingAdapter` interface.

Assembly File

The root element to declare an Oracle Event Processing high availability correlating output adapter is `wlevs:adapter` with `the provider` element set to `ha-correlating` as shown.

```
<wlevs:adapter id="myCorrelatingAdapter" provider="ha-correlating">
    <wlevs:listener>
        <bean class="com.bea.wlevs.example.cluster.ClusterAdapterBean"/>
    </wlevs:listener>
    <wlevs:instance-property name="correlatedSource" ref="clusterCorrOutstream"/>
    <wlevs:instance-property name="failOverDelay" value="2000"/>
</wlevs:adapter>
```

Table 17-12 describes the additional child elements.

Table 17-12 Child Elements

Child Element	Description
`wlevs:listener`	Specify the regular output adapter downstream from this Oracle Event Processing high availability buffering output adapter.
`wlevs:instance-property`	Specify one or more `instance-property` element `name` and `value` attributes as Table 17-13 describes.

Table 17-13 lists the instance properties.

Table 17-13 Instance Properties

Name	Value
`correlatedSource`	Specify the event source to correlate against. Events seen from this source are purged from the trimming queue. Events still in the queue at failover are replayed.

Table 17-13 (Cont.) Instance Properties

Name	Value
`failOverDelay`	Specify the delay timeout in milliseconds that is used to decide how soon after failover correlation should restart. Default: 0 ms.

Configuration File

The root element for configuring an Oracle Event Processing high availability correlating output adapter is `ha-correlating-adapter`. The `name` child element for a particular adapter must match the `id` attribute of the corresponding `wlevs:adapter` element in the EPN assembly file that declares this adapter as shown.

```
<ha:ha-correlating-adapter>
    <name>myCorrelatingAdapter</name>
    <window-length>15000</window-length>
    <warm-up-window-length units="minutes">6</warm-up-window-length>
</ha:ha-correlating-adapter>
```

Table 17-14 describes the child elements.

Table 17-14 Child Elements

Child Element	Description
`fail-over-delay`	Specify the delay timeout in milliseconds that is used to decide how soon after failover correlation should restart. Default: 0 ms.
`warm-up-window-length`	Specify the length of time it takes the application to rebuild state after a previously failed secondary restarts or a new secondary is added as an integer number of `units`. Valid values for attribute `units`: • `seconds`: wait the specified number of seconds. • `minutes`: wait the specified number of minutes. Default: `units` is `seconds`. For more information, see Choose an Adequate warm-up-window-length Time.
window-length	The length of the saved buffer of events in milliseconds.
trimming-interval	The interval at which events should be trimmed from a secondary buffer. Units can be `events` or `millis`.
heartbeat	The value (n) for the heartbeat time out on this adapter. A heartbeat is generated when `n` time units go by without any event being generated on this adapter. The default time unit is nanoseconds.
batch-size	The batch size in terms of events for sending event time stamps to the secondary. By default, batching is disabled.

18

Scalable Applications

You can build scalability into your application design with partitioning and parallel processing, and by taking high availability options into consideration. Oracle Event Processing enables you to use default or custom partitioning and parallel processing settings on channels and the upstream adapter. You can also partition an incoming JMS event stream and configure the JSMS Event stream group pattern matching.

This chapter includes the following sections:

- Default Channel Scalability Settings
- Partition an Incoming JMS Event Stream
- Notification Group Naming Conventions
- Custom Channel Event Partitioner.

18.1 Default Channel Scalability Settings

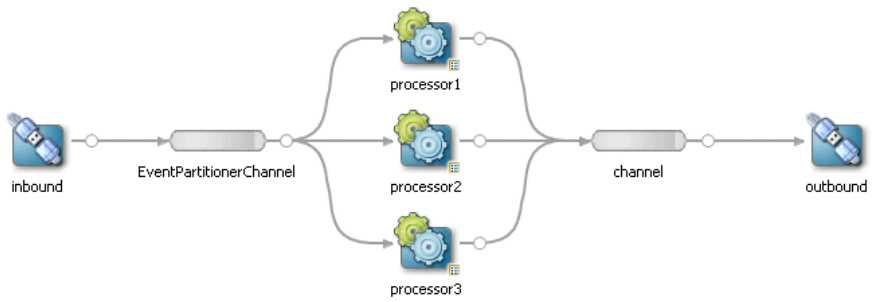
You can configure a channel to use the default event property-based event partitioner. With this default configuration, every time an incoming event arrives, the channel selects a listener and dispatches the event to that listener instead of broadcasting every event to every listener.

Note:

Batching is not supported when you configure a channel with an event partitioner.

Figure 18-1 shows an EPN that uses an event partitioner property to partition a channel. In this example, the inbound adapter sends events of type `PriceEvent`, which has two properties: stock symbol and stock price. The example partitions the channel on the `symbol` property and shows you how to add multithreading to either the channel or the upstream adapter.

Figure 18-1 EventPartitioner EPN



- Configure Partitioning on the Channel
- Configure Parallel Processing on the Channel
- Configure Parallel Processing on the Upstream Adapter.

18.1.1 Configure Partitioning on the Channel

See *Customizing Oracle Event Processing* for information about customizing an event partitioner on a channel.

1. Add a channel to your EPN.

   In Figure 18-1, the channel is `EventPartitionerChannel`.

2. Connect the channel to an upstream adapter.

   In Figure 18-1, the upstream adapter is `inbound`.

3. Connect the channel to two or more listeners.

   In Figure 18-1, the channel is connected to Oracle CQL processors `processor1`, `processor2`, and `processor3`.

4. Edit the assembly file to add a `partitionByEventProperty` instance property to the channel element.

   The `value` of this `instance-property` is the name of the event property by which the channel partitions events.

   In this example, the channel partitions events by the event property `symbol`.

```
<wlevs:event-type-repository>
  <wlevs:event-type type-name="PriceEvent">
    <wlevs:properties>
       <wlevs:property name="symbol" type="char" />
       <wlevs:property name="price" type="long" />
    </wlevs:properties>
  </wlevs:event-type>
</wlevs:event-type-repository>

<wlevs:channel id="EventPartitionerChannel" event-type="PriceEvent">
  <wlevs:instance-property name="partitionByEventProperty" value="symbol" />
   <wlevs:listener ref="processor1" />
   <wlevs:listener ref="processor2" />
   <wlevs:listener ref="processor3" />
   <wlevs:source ref="inbound" />
</wlevs:channel>
```

18.1.2 Configure Parallel Processing on the Channel

If you want the channel to allocate threads, set the max-threads property to the number of listeners in the EPN.

If you want to provide increased concurrency downstream from the channel, you can associate a thread pool with the channel by setting the `max-threads` property on the channel. The best value for the maximum number of threads can depend on many factors including the details of the Oracle CQL queries in downstream processors (do the queries allow parallel execution), and the behavior observed while running the application (are all the CPU cores utilized). As a starting point in tuning the maximum number of threads, it is reasonable to set it equal to the number of listeners on the channel.

In this example, there are 3 listeners.

```
<wlevs:channel id="EventPartitionerChannel" event-type="PriceEvent" max-threads="3" >
  <wlevs:instance-property name="eventPartitioner" value="true" />
  <wlevs:listener ref="processor1" />
  <wlevs:listener ref="processor2" />
  <wlevs:listener ref="processor3" />
  <wlevs:source ref="inbound" />
</wlevs:channel>
```

18.1.3 Configure Parallel Processing on the Upstream Adapter

1. Edit the EPN assembly file to configure the channel to set the `max-threads` attribute to 0.

   ```
   <wlevs:channel id="EventPartitionerChannel" event-type="PriceEvent"
     max-threads="0" >
     <wlevs:instance-property name="eventPartitioner" value="true" />
     <wlevs:listener ref="processor1" />
     <wlevs:listener ref="processor2" />
     <wlevs:listener ref="processor3" />
     <wlevs:source ref="inbound" />
   </wlevs:channel>
   ```

2. Edit the Oracle Event Processing server file to add a `work-manager` element.

   Selecting the appropriate `min-threads-constraint` and `max-threads-constraint` for the work manager can depend on a number of factors, including the factors discussed in Configure Parallel Processing on the Channel for setting thread counts on a channel and whether the work manager is dedicated to a specific adapter or shared with other components (other adapters or the Jetty service). As a starting point in tuning, it' is reasonable to set the `min-threads-constraint` and `max-threads-constraint` properties equal to the number of listeners downstream from the adapter if the work manager is dedicated to a single adapter instance.

   If this work manager is not shared by more than one component (that is, it is dedicated to the upstream adapter in this configuration), then set the `min-threads-constraint` and `max-threads-constraint` elements equal to the number of listeners.

   ```
   <work-manager>
       <name>adapterWorkManager</name>
       <min-threads-constraint>3</min-threads-constraint>
       <max-threads-constraint>3</max-threads-constraint>
   </work-manager>
   ```

   For more information, see *Schema Reference for Oracle Event Processing*.

3. Edit the component configuration file to configure the upstream adapter with this `work-manager`.

   ```
   <adapter>
       <name>inbound</name>
       <work-manager-name>adapterWorkManager</work-manager-name>
       ...
   </adapter>
   ```

18.2 Partition an Incoming JMS Event Stream

You can add the `ActiveActiveGroupBean` class to the assembly file to partition an incoming JMS event stream by a selector in a multiserver domain.

- Configure Partitioning without High Availability
- Configure Partitioning with High Availability

18.2.1 Configure Partitioning without High Availability

1. Create a multiserver domain.

   In this example, the deployment group name is `MyDeploymentGroup`.

   See *Administering Oracle Event Processing*.

2. Configure the Oracle Event Processing server configuration file on each Oracle Event Processing server to add the appropriate `ActiveActiveGroupBean` notification group to the `groups` child element of the `cluster` element.

   The Oracle Event Processing server configuration file is located in `/Oracle/Middleware/my_oep/user_projects/domains/<domain_name>/<server_name>/config`.

   Table 18-2 shows `cluster` elements for Oracle Event Processing servers `ocep-server-1`, `ocep-server-2`, `ocep-server-3`, and `ocep-server-4`. The deployment group is `MyDeploymentGroup` and the notification groups are defined using default `ActiveActiveGroupBean` notification group naming.

   Optionally, you can specify your own group naming convention as Notification Group Naming Conventions describes.

Table 18-1 Server Configuration File Groups Element Configuration

Partition	cluster Element

`ocep-server-1`	`<cluster>`
	`  <server-name>ocep-server-1</server-name>`
	`  ...`
	`  <enabled>coherence</enabled>`
	`  ...`
	`  <groups>MyDeploymentGroup, ActiveActiveGroupBean_group1</groups>`
	`</cluster>`

`ocep-server-2`	`<cluster>`
	`  <server-name>ocep-server-2</server-name>`
	`  ...`
	`  <enabled>coherence</enabled>`
	`  ...`
	`  <groups>MyDeploymentGroup, ActiveActiveGroupBean_group2</groups>`
	`</cluster>`

`ocep-server-3`	`<cluster>`
	`  <server-name>ocep-server-3</server-name>`
	`  ...`
	`  <enabled>coherence</enabled>`
	`  ...`
	`  <groups>MyDeploymentGroup, ActiveActiveGroupBean_group3</groups>`
	`</cluster>`

Table 18-1 (Cont.) Server Configuration File Groups Element Configuration

Partition	cluster Element
ocep-server-4	<cluster> <server-name>ocep-server-4</server-name> ... <enabled>coherence</enabled> ... <groups>MyDeploymentGroup, ActiveActiveGroupBean_group4</groups> </cluster>

3. Create an Oracle Event Processing application.
4. Add an `ActiveActiveGroupBean` element to the assembly file as follows.

```
<bean id="clusterAdapter" class="com.oracle.cep.cluster.hagroups.ActiveActiveGroupBean">
</bean>
```

5. Define a parameterized `message-selector` in the `jms-adapter` element for the JMS inbound adapters.
   a. Edit the component configuration file to add `group-binding` child elements to the `jms-adapter` element for the JMS inbound adapters.
   b. Add one `group-binding` element for each possible JMS message-selector value as shown.

```
<jms-adapter>
  <name>JMSInboundAdapter</name>
  <event-type>StockTick</event-type>
  <jndi-provider-url>t3://ppurich-pc:7001</jndi-provider-url>
  <destination-jndi-name>./Topic1</destination-jndi-name>
  <user>weblogic</user>
  <password>weblogic1</password>
  <work-manager>JettyWorkManager</work-manager>
  <concurrent-consumers>1</concurrent-consumers>
  <session-transacted>true</session-transacted>
  <message-selector>${CONDITION}</message-selector>
  <bindings>
    <group-binding group-id="ActiveActiveGroupBean_group1">
      <param id="CONDITION">acctid > 400</param>
    </group-binding>
    <group-binding group-id="ActiveActiveGroupBean_group2">
      <param id="CONDITION">acctid BETWEEN 301 AND 400</param>
    </group-binding>
    <group-binding group-id="ActiveActiveGroupBean_group3">
      <param id="CONDITION">acctid BETWEEN 201 AND 300</param>
    </group-binding>
    <group-binding group-id="ActiveActiveGroupBean_group4">
      <param id="CONDITION">acctid <= 200</param>
    </group-binding>
  </bindings>
</jms-adapter>
```

In this configuration, when the application is deployed to an Oracle Event Processing server with a `cluster` element `groups` child element that contains `ActiveActiveGroupBean_group1`, then the `CONDITION` parameter is defined as `acctid > 400` and the application processes events whose `acctid` property is greater than 400.

Note:

Each in-bound JMS adapter must listen to a different topic. For more information, see Adapters.

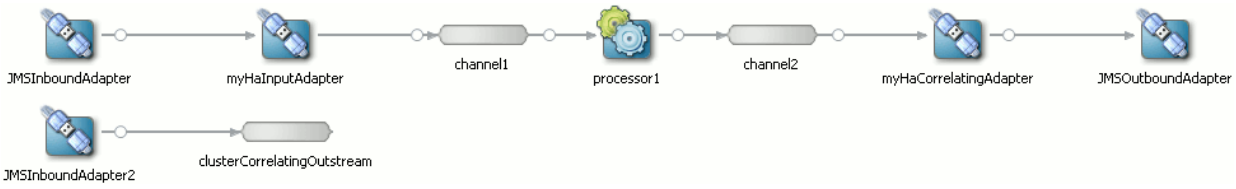
6. Deploy your application to the deployment group of your multiserver domain.

   At runtime, each Oracle Event Processing server configures its instance of the application with the `message-selector` that corresponds to its `ActiveActiveGroupBean` notification group. This partitions the JMS topic so that each instance of the application processes a subset of the total number of messages in parallel.

18.2.2 Configure Partitioning with High Availability

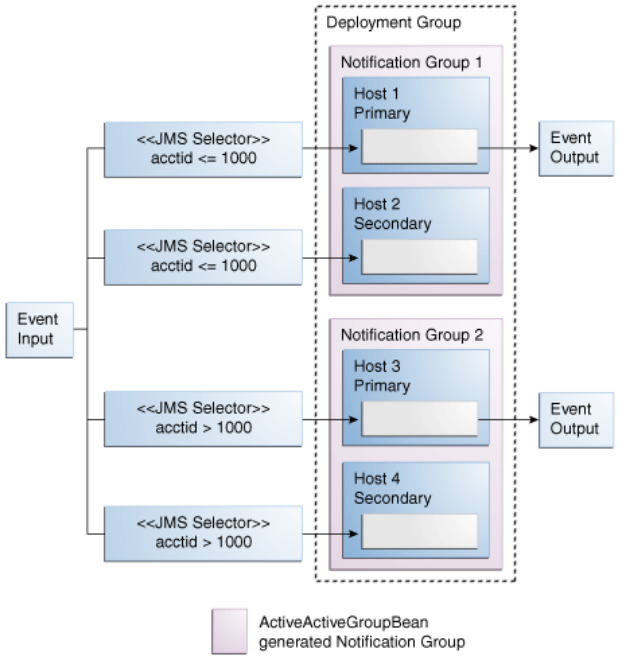
This procedure uses the example application from Configure Precise Recovery With JMS. Figure 18-2 shows the EPN diagram, and Example 18-1 and Example 18-2 show the corresponding assembly and configuration files.

Figure 18-2 Precise Recovery With JMS EPN



The procedure creates the Oracle Event Processing high availability configuration shown in Figure 18-3.

Figure 18-3 ActiveActiveGroupBean With High Availability



Configure Scalability in a JMS Application with High Availability

1. Create a multiserver domain.

In this example, the deployment group is named `MyDeploymentGroup`.

See *Administering Oracle Event Processing*.

2. Configure the Oracle Event Processing server configuration file on each Oracle Event Processing server to add the appropriate `ActiveActiveGroupBean` notification group to the `groups` child element of the `cluster` element.

   The Oracle Event Processing server configuration file is located in `/Oracle/Middleware/my_oep/user_projects/domains/<domain_name>/<server_name>/config`.

   Table 18-2 shows `cluster` elements for Oracle Event Processing servers `ocep-server-1`, `ocep-server-2`, `ocep-server-3`, and `ocep-server-4`. The deployment group is `MyDeploymentGroup` and notification groups are defined using default `ActiveActiveGroupBean` notification group names.

   Note that `ocep-server-1` and `ocep-server-2` use the same notification group name (`ActiveActiveGroupBean_group1`) and `ocep-server-3` and `ocep-server-4` use the same notification group name (`ActiveActiveGroupBean_group2`).

Table 18-2 Server Configuration File Groups Element Configuration

Partition	cluster Element

`ocep-server-1`	`<cluster>`
	`  <server-name>ocep-server-1</server-name>`
	`  ...`
	`  <enabled>coherence</enabled>`
	`  ...`
	`  <groups>MyDeploymentGroup, ActiveActiveGroupBean_group1</groups>`
	`</cluster>`

`ocep-server-2`	`<cluster>`
	`  <server-name>ocep-server-2</server-name>`
	`  ...`
	`  <enabled>coherence</enabled>`
	`  ...`
	`  <groups>MyDeploymentGroup, ActiveActiveGroupBean_group1</groups>`
	`</cluster>`

`ocep-server-3`	`<cluster>`
	`  <server-name>ocep-server-3</server-name>`
	`  ...`
	`  <enabled>coherence</enabled>`
	`  ...`
	`  <groups>MyDeploymentGroup, ActiveActiveGroupBean_group2</groups>`
	`</cluster>`

`ocep-server-4`	`<cluster>`
	`  <server-name>ocep-server-4</server-name>`
	`  ...`
	`  <enabled>coherence</enabled>`
	`  ...`
	`  <groups>MyDeploymentGroup, ActiveActiveGroupBean_group2</groups>`
	`</cluster>`

3. Create an Oracle Event Processing high availability application.

For more information, see High Availability Applications.

4. Add an `ActiveActiveGroupBean` element to the assembly file as shown.

```
<bean id="clusterAdapter" class="com.oracle.cep.cluster.hagroups.ActiveActiveGroupBean">
</bean>
```

5. Edit the component configuration file to configure a `jms-adapter` element for the inbound JMS adapters as shown.

   You must set each inbound JMS adapter to listen to a different topic and set `session-transacted` to `true`.

```
<?xml version="1.0" encoding="UTF-8"?>
<wlevs:config
        xmlns:wlevs="http://www.bea.com/ns/wlevs/config/application"
        xmlns:ha="http://www.oracle.com/ns/cep/config/cluster">
    ...
    <jms-adapter>
        <name>JMSInboundAdapter</name>
        <event-type>StockTick</event-type>
        <jndi-provider-url>t3://ppurich-pc:7001</jndi-provider-url>
        <destination-jndi-name>./Topic1</destination-jndi-name>
        <session-transacted>true</session-transacted>
    ...    </jms-adapter>
    <jms-adapter>
        <name>JMSInboundAdapter2</name>
        <event-type>StockTick</event-type>
        <jndi-provider-url>t3://ppurich-pc:7001</jndi-provider-url>
        <destination-jndi-name>./Topic2</destination-jndi-name>
        <session-transacted>true</session-transacted>
    ...    </jms-adapter>
</wlevs:config>
```

6. Define a parameterized `message-selector` in the `jms-adapter` element for each JMS inbound adapter.

   a. Edit the component configuration file to add `group-binding` child elements to the `jms-adapter` element for the JMS inbound adapters.

   b. Add one `group-binding` element for each possible JMS `message-selector` value as shown.

```
<jms-adapter>
    <name>JMSInboundAdapter</name>
    <event-type>StockTick</event-type>
    <jndi-provider-url>t3://ppurich-pc:7001</jndi-provider-url>
    <destination-jndi-name>./Topic1</destination-jndi-name>
    <session-transacted>true</session-transacted>
    <message-selector>${CONDITION}</message-selector>
    <bindings>
        <group-binding group-id="ActiveActiveGroupBean_group1">
            <param id="CONDITION">acctid <= 1000</param>
        </group-binding>
        <group-binding group-id="ActiveActiveGroupBean_group2">
            <param id="CONDITION">acctid > 1000</param>
        </group-binding>
     </bindings>
</jms-adapter>
```

   In this configuration, when the application is deployed to an Oracle Event Processing server with a `cluster` element `groups` child element that contains `ActiveActiveGroupBean_group1`, then the `CONDITION` parameter is defined as `acctid <= 1000` and the application processes events whose `acctid` property is less than or equal to 1000. Similarly, when the application is deployed to an Oracle Event Processing server with a `cluster` element `groups` child

element that contains `ActiveActiveGroupBean_group2`, then the `CONDITION` parameter is defined as `acctid > 1000` and the application processes events whose `acctid` property is greater than 1000.

7. Edit the component configuration file to configure a `jms-adapter` element for the outbound JMS adapter as shown:

   Configure the out-bound JMS adapter with the same topic as the correlating in-bound adapter (in this example, `JMSInboundAdapter2`: `./Topic2`), and set `session-transacted` to `true`.

```
<?xml version="1.0" encoding="UTF-8"?>
<wlevs:config
        xmlns:wlevs="http://www.bea.com/ns/wlevs/config/application"
        xmlns:ha="http://www.oracle.com/ns/cep/config/cluster">
    ...
    <jms-adapter>
        <name>JMSInboundAdapter</name>
        <event-type>StockTick</event-type>
        <jndi-provider-url>t3://ppurich-pc:7001</jndi-provider-url>
        <destination-jndi-name>./Topic1</destination-jndi-name>
        <session-transacted>true</session-transacted>
    ...    </jms-adapter>
    <jms-adapter>
        <name>JMSInboundAdapter2</name>
        <event-type>StockTick</event-type>
        <jndi-provider-url>t3://ppurich-pc:7001</jndi-provider-url>
        <destination-jndi-name>./Topic2</destination-jndi-name>
        <session-transacted>true</session-transacted>
    ...    </jms-adapter>
    <jms-adapter>
        <name>JMSOutboundAdapter</name>
        <event-type>StockTick</event-type>
        <jndi-provider-url>t3://ppurich-pc:7001</jndi-provider-url>
        <destination-jndi-name>./Topic2</destination-jndi-name>
        <session-transacted>true</session-transacted>
    ...    </jms-adapter>
</wlevs:config>
```

8. Deploy your application to the deployment group of your multiserver domain.

   At runtime, each Oracle Event Processing server configures its instance of the application with the `message-selector` that corresponds to its `ActiveActiveGroupBean` notification group. This partitions the JMS topic so that each instance of the application processes a subset of the total number of messages in parallel.

   If the active Oracle Event Processing server in an `ActiveActiveGroupBean` group goes down, the Oracle Event Processing server performs an Oracle Event Processing high availability failover to the standby Oracle Event Processing server in that `ActiveActiveGroupBean` group.

Example 18-1 Precise Recovery With JMS EPN Assembly File

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >

    <wlevs:event-type-repository>
        <wlevs:event-type type-name="StockTick">
            <wlevs:properties>
                <wlevs:property name="lastPrice" type="double" />
                <wlevs:property name="symbol" type="char" />
            </wlevs:properties>
        </wlevs:event-type>
    </wlevs:event-type-repository>
```

```
    <wlevs:adapter id="JMSInboundAdapter" provider="jms-inbound">
        <wlevs:listener ref="myHaInputAdapter"/>
    </wlevs:adapter>

    <wlevs:adapter id="myHaInputAdapter" provider="ha-inbound" >
        <wlevs:instance-property name="keyProperties" value="sequenceNo"/>
        <wlevs:instance-property name="timeProperty" value="inboundTime"/>
    </wlevs:adapter>

    <wlevs:channel id="channel1" event-type="StockTick">
        <wlevs:listener ref="processor1" />
        <wlevs:source ref="myHaInputAdapter"/>
        <wlevs:application-timestamped>
            <wlevs:expression>inboundTime</wlevs:expression>
        </wlevs:application-timestamped>
    </wlevs:channel>

    <wlevs:processor id="processor1">
        <wlevs:listener ref="channel2" />
    </wlevs:processor>

    <wlevs:channel id="channel2" event-type="StockTick">
        <wlevs:listener ref="myHaCorrelatingAdapter" />
    </wlevs:channel>

    <wlevs:adapter id="myHaCorrelatingAdapter" provider="ha-correlating" >
        <wlevs:instance-property name="correlatedSource" ref="clusterCorrelatingOutstream"/>
        <wlevs:instance-property name="failOverDelay" value="2000"/>
        <wlevs:listener ref="JMSOutboundAdapter"/>
    </wlevs:adapter>

    <wlevs:adapter id="JMSOutboundAdapter" provider="jms-outbound">
    </wlevs:adapter>

    <wlevs:adapter id="JMSInboundAdapter2" provider="jms-inbound">
    </wlevs:adapter>

    <wlevs:channel id="clusterCorrelatingOutstream" event-type="StockTick" advertise="true">
        <wlevs:source ref="JMSInboundAdapter2"/>
    </wlevs:channel>
</beans>
```

Example 18-2 Precise Recovery With JMS Component Configuration Assembly File

```
<?xml version="1.0" encoding="UTF-8"?>
<wlevs:config
        xmlns:wlevs="http://www.bea.com/ns/wlevs/config/application"
        xmlns:ha="http://www.oracle.com/ns/cep/config/cluster">
    <processor>
        <name>processor1</name>
        <rules>
            <query id="helloworldRule">
                <![CDATA[ select * from channel1 [Now] >
            </query>
        </rules>
    </processor>
    <jms-adapter>
        <name>JMSInboundAdapter</name>
        <event-type>StockTick</event-type>
        <jndi-provider-url>t3://ppurich-pc:7001</jndi-provider-url>
        <destination-jndi-name>./Topic1</destination-jndi-name>
        <session-transacted>true</session-transacted>
    ...
    </jms-adapter>
    <jms-adapter>
        <name>JMSInboundAdapter2</name>
        <event-type>StockTick</event-type>
        <jndi-provider-url>t3://ppurich-pc:7001</jndi-provider-url>
        <destination-jndi-name>./Topic2</destination-jndi-name>
```

```
        <session-transacted>true</session-transacted>
    ...
    </jms-adapter>
    <jms-adapter>
        <name>JMSOutboundAdapter</name>
        <event-type>StockTick</event-type>
        <jndi-provider-url>t3://ppurich-pc:7001</jndi-provider-url>
        <destination-jndi-name>./Topic2</destination-jndi-name>
        <session-transacted>true</session-transacted>
    ...
    </jms-adapter>
</wlevs:config>
```

18.3 Notification Group Naming Conventions

By default, the `ActiveActiveGroupBean` class creates notification groups with the following name where *X* is a string.

```
ActiveActiveGroupBean_X
```

At runtime, `ActiveActiveGroupBean` scans the existing groups defined on the Oracle Event Processing server and applies the following default pattern match. When ActiveActiveGroupBean finds a match, it creates a notification group with that name.

```
ActiveActiveGroupBean_\\w+
```

Optionally, you can define your own group pattern to specify a different notification group naming pattern.

1. Configure the assembly file to add a `groupPattern` attribute to your `ActiveActiveGroupBean` element as shown.

   ```
   <bean id="clusterAdapter" class="com.oracle.cep.cluster.hagroups.ActiveActiveGroupBean">
       <property name="groupPattern" value="MyNotificationGroupPattern*"/>
   </bean>
   ```

2. Specify a value for the `groupPattern` attribute that matches the cluster group naming convention you want to use for notification groups.

18.4 Custom Channel Event Partitioner

Most channels use the default event partitioning described in *Developing Applications for Oracle Event Processing*where if no partitioner is specified and if the `partitionByEventProperty` element is not present, the channel sends events to all listeners. The `partitionByEventProperty` element provides a level of customization by partitioning on the specified event with a default partitioning algorithm. This section explains how you can further customize how events are dispatched to the channel listeners by programmatically configuring a custom partitioner that provides finer control over the default partitioning algorithm. For example, you can create an event partitioner that is based on a property range

18.4.1 EventPartitioner Interface

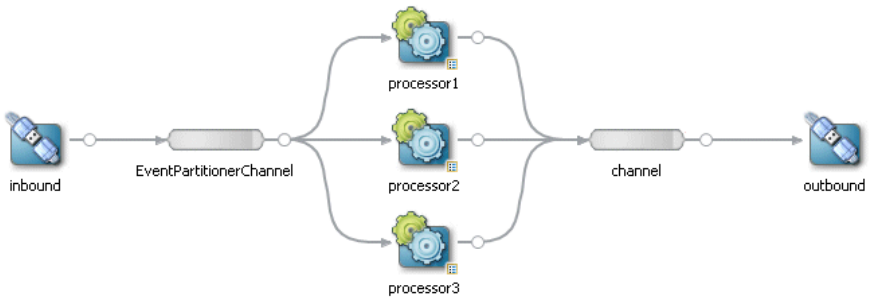
Use the com.bea.wlevs.channel.EventPartitioner interface to partition events across a channel to customize how events are dispatched to the channel listener.

Note:

When you implement custom partitioning and parallel processing, make sure to add code to preserve event order and to carefully manage multithreading.

Figure 18-4 shows an EPN that uses an event partitioner to partition a channel. In this example, the inbound adapter sends events of type `PriceEvent`, which has two properties: stock symbol and stock price. The example partitions the channel on the `symbol` property and shows you how to add multithreading to the channel or to the upstream adapter.

Figure 18-4 Event Partitioner EPN



18.4.2 Implement the EventPartitioner Interface

1. In Oracle JDeveloper, open your Oracle Event Processing application.
2. Edit your `MANIFEST.MF` file to import package `com.bea.wlevs.channel`.
3. Select the project and select **File > New > From Gallery**.

   The New Gallery dialog displays.
4. In the **New Gallery** dialog, select **General** in the left panel and **Java Class** in the right panel, and click **OK**.

   The Create Java Class dialog displays.
5. In the **Create Java Class** dialog, provide a class name, package name, and extends information.
6. Under **Optional Attributes** and **Implements**, use the Add **(+)** button to locate the `com.bea.wlevs.channel.EventPartitioner` interface.
7. Click **OK**.

   A new `EventPartitioner` class is created.
8. Complete the implementation of your `EventPartitioner` as shown.

```
package com.acme;

import com.bea.wlevs.channel.EventPartitioner;
import com.bea.wlevs.ede.api.EventProcessingException;
import com.bea.wlevs.ede.api.EventType;

public class MyEventPartitioner implements EventPartitioner {

  private final EventType eventType;
```

```
  private int numberOfPartitions;

  @Override
  public void activateConfiguration(int numberOfPartitions, EventType eventType) {
     this.numberOfPartitions = numberOfPartitions;
     this.eventType = eventType;
  }

  @Override
  public int partition(Object event) throws EventProcessingException {
    int dispatchToListener = 0;
    ... // Your implementation.
    return dispatchToListener;
  }
}
```

The `activateConfiguration` method is a callback that the Oracle Event Processing server invokes before `ActivatableBean.afterConfigurationActive` and before your `EventPartitioner` class's `partition` method is invoked.

When you associate this `EventPartitioner` with a channel, the channel will invoke your `EventPartitioner` class's `partition` method each time the channel receives an event.

Your `partition` method must return the index of the listener to which the channel should dispatch the event. The index must be an `int` between `0` and `numberOfPartitions - 1`.

9. Add a channel to your EPN.

   In Figure 18-4, the channel is `EventPartitionerChannel`.

10. Connect the channel to an upstream adapter.

    In Figure 18-4, the upstream adapter is `inbound`.

11. Connect the channel to two or more listeners.

    In Figure 18-4, the channel is connected to Oracle CQL processors `processor1`, `processor2`, and `processor3`.

    If you want to the channel to perform load balancing, each listener must be identical.

12. Edit the EPN assembly file to add an `eventPartitioner` instance property to the channel element.

    The `value` of this `instance-property` is the fully qualified class name of the `EventPartitioner` instance the channel will use to partition events. This class must be on your Oracle Event Processing application class path.

    In this example, the channel uses `EventPartitioner` instance `com.acme.MyEventPartitioner` to partition events.

```
<wlevs:channel id="EventPartitionerChannel" event-type="PriceEvent"   max-threads="0" >
  <wlevs:instance-property name="eventPartitioner"
    value="com.acme.MyEventPartitioner" />
  <wlevs:listener ref="filterFanoutProcessor1" />
  <wlevs:listener ref="filterFanoutProcessor2" />
  <wlevs:listener ref="filterFanoutProcessor3" />
  <wlevs:source ref="PriceAdapter" />
</wlevs:channel>
```