Oracle® Fusion Middleware

Getting Started with Oracle Event Processing

12*c* Release (12.1.3)

E28542-07

November 2015

How to get started with developing Oracle Event Processing applications.

ORACLE®

Oracle Fusion Middleware Getting Started with Oracle Event Processing, 12c Release (12.1.3)

E28542-07

Contents

3 Oracle JDeveloper Procedures

4 Create a Basic Application

5 Create a Fraud Detection Application with EDN Adapters

6 Event Processing Samples in Oracle Event Processing

Glossary

Preface

This document provides general background information and detailed code samples to help you learn about Oracle Event Processing and the Oracle CQL.

Audience

This document is intended for users interested in learning about Oracle Event Processing and Oracle CQL. Readers should be familiar with basic Java development. Some knowledge of SQL would be helpful.

Related Documents

For more information, see the following:

- *Developing Applications for Oracle Event Processing*

- *Administering 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

This guide has been updated for the 12c Release. The following table lists the sections that have been added or changed.

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

Sections	Changes Made
Entire Guide	Product renamed to Oracle Event Processing.
Chapter 2, Oracle JDeveloper Quick Reference, and Chapter 3, Oracle JDeveloper Procedures	Application development procedures have been updated to use Oracle JDeveloper instead of Eclipse. All IDE procedures and information are aggregated into these two chapters.
Chapter 4, Create a Basic Application	Moved this walkthrough here from *Developing Applications for Oracle Event Processing* and updated to use Oracle JDeveloper.
Chapter 5, Create a Fraud Detection Application with EDN Adapters	Added this walkthrough to illustrate SOA integration with EDN adapters.
Chapter 7, Oracle Event Processing Samples	Moved here from Oracle Fusion Middleware Developing Application for Oracle Event Processing to form a complete set of examples and walkthroughs.

1

Event Processing Overview in Oracle Event Processing

Oracle Event Processing is a high throughput and low latency platform for developing, administering, and managing applications that monitor real-time streaming events.

This guide introduces you to Oracle Event Processing and Oracle JDeveloper for application development. The step-by-step walkthroughs and sample applications provide a solid foundation for understanding how the parts of the platform work together and how to create an Oracle Event Processing application.

This chapter covers the following topics:

- Oracle Event Processing
- Oracle Event Processing Embedded
- Application Programming Model
- Component Roles in an Event Processing Network
- Oracle CQL
- Technologies in Oracle Event Processing
- Oracle Event Processing High-Level Use Cases.

1.1 Oracle Event Processing

Oracle Event Processing consists of the Oracle Event Processing server, Oracle Event Processing Visualizer, a command-line administrative interface, and the Oracle JDeveloper Integrated Development Environment (IDE).

An Oracle Event Processing server hosts logically related resources and services for running Oracle Event Processing applications. Servers are grouped into and managed as domains. A domain can have one server (standalone-server domain) or many (multiserver domain). You manage the Oracle Event Processing domains and servers through Oracle Event Processing Visualizer and the Oracle Event Processing administrative command-line interface.

Oracle Event Processing Visualizer is a web-based user interface through which you can deploy, configure, test, and monitor Oracle Event Processing applications running on the Oracle Event Processing server.

Oracle Event Processing administrative command-line interface enables you to manage the server from the command line and through configuration files. For example, you can start and stop domains and deploy, suspend, resume, and uninstall an application.

Oracle JDeveloper for the 12c release includes an integrated framework that supports Oracle Event Processing application design, development, testing, and deployment.

Oracle Stream Explorer is developed to simplify the complex event processing operations and make them available even to users without any technical background.

1.2 Oracle Event Processing Embedded

Oracle Event Processing on Oracle Java Embedded systems is an event processing server designed to support event processing applications in embedded environments such as those supported by the Java Embedded Suite (JES). Oracle Event Processing on Oracle Java Embedded systems features represent a subset of Oracle Event Processing features.

The following enhancements have been made in 12c 12.1.3 release:

- Support for JDK 8 and dependency on JES

- Support for JDeveloper IDE to develop and deploy applications

- Improved performance through support for Views and Subqueries

- Ability to match patterns across multiple streams

- Dynamic windows based on Application Timestamps.

- HTTP Pubsub support.

1.3 Application Programming Model

An Oracle Event Processing application receives and processes data streaming from an event source. That data might be coming from one of a variety of places, such as a monitoring device, a financial services company, or a motor vehicle. While monitoring the data, the application might identify and respond to patterns, look for events that meet specified criteria and alert other applications, or do other work that requires immediate action based on quickly changing data.

Oracle Event Processing uses an event-driven architecture called SEDA where an application is broken into a set of stages (nodes) connected by queues. In Oracle Event Processing, the channel component represents queues while all of the other components represent stages. Every component in the EPN has a role in processing the data.

The event processing network (EPN) is linear with data entering the EPN through an adapter where it is converted to an event. After the conversion, events pass through the stages from one end to the other. At various stages in the EPN, the component can execute logic or create connections with external components as needed.

Figure 1-1   Oracle Event Processing Application
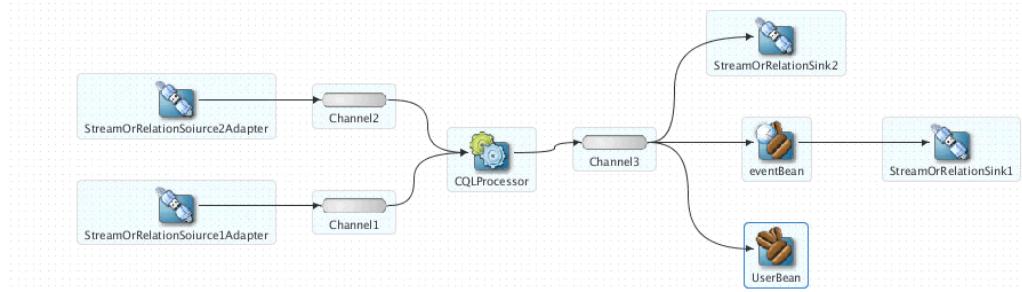
Oracle Event Processing applications have the following characteristics:

- **Applications leverage the database programming model**: Some of the programming model in Oracle Event Processing applications is conceptually an extension of what you find in database programming. Events are similar to database rows in that they are tuples against which you can execute queries with Oracle Continuous Query Language (Oracle CQL). Oracle CQL is an extension to SQL, but designed to work on streaming data.

- **Stages represent discrete functional roles**: The structure of an EPN enables you to execute logic against events flowing through the network. Stages also enable you to capture multiple processing paths with a network that branches into multiple downstream directions based on event patterns that your code finds.

- **Stages transmit events through an EPN by acting as event sinks and event sources**: The stages in an EPN can receive events as event sinks and send events as event sources. An event sink is a Java class that implements logic to listen for and work on specific events.

- **Events are handled as streams or relations**: Channels convey events from one stage to another in the EPN. A channel can convey events in a stream or in a relation. Both stream and relation channels insert events into a collection and send the 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.

1.4 Component Roles in an Event Processing Network

The core of Oracle Event Processing applications is the EPN. You build an EPN by connecting components that have a role in processing events that pass through the network. When you develop an Oracle Event Processing application, you identify which kinds of components are needed to achieve the desired functionality.

The best way to create an EPN is to use Oracle JDeveloper to add, configure, and connect the components. The EPN has a roughly linear shape where events enter from the left, move through the EPN to the right, and exit or terminate at the far right.

The EPN components provide ways to:

- **Exchange event data with external sources**: You can connect external databases, caches, HTTP messages, Java Message Service (JMS) messages, files, and big data

storage to the EPN of your application to add ways for data, including event data, to pass into or out of the EPN.

- **Model event data as event types so that it can be handled by application code**: You implement or define event types that model event data so that application code can access and manipulate it.

- **Query and filter events**: Oracle CQL enables you to query events as you would data in a database. Oracle CQL includes features specifically intended for querying streaming data. You can add Oracle CQL code to an EPN by adding a processor component. All EPN processors are Oracle CQL processors.

- **Execute Java logic to handle events**: You can add Java classes that send and receive events the same way that other EPN stages do. Logic in these classes can retrieve values from events, create new events, and more.

1.5 Oracle CQL

Oracle CQL is an extension to Structured Query Language (SQL) with the same keywords and syntax rules, but with features to support the unique aspects of streaming data.

An event conceptually corresponds to a row in a database table. However, an important difference between an event and a table row is that with events, one event is always before or after another in time, and the stream is potentially infinite and ever-changing.

With a relational database, data is relatively static and changes when a user initiates a transaction such as an add, delete, or change operation. In contrast, event data streams constantly flow into the EPN where your query examines it as it arrives.

To make the most of the sequential, time-oriented quality of streaming data, Oracle CQL enables you to do the following:

- Specify a window of a particular time period or range from which events should be queried. This could be for every five seconds worth of events, for example.

- Specify a window of a particular number of events, called *rows*, against which to query. This might be every sequence of 10 events.

- Specify how often the query should execute against the stream. For example, the query could *slide* every five seconds to a later five-second window of events.

- Separate (partition) an incoming stream into multiple streams based on specified event characteristics. You could have the query create new streams for each of the specified stock symbols found in incoming trade events.

In addition, Oracle CQL supports common aspects of SQL that you might be familiar with, including *views* and *joins*. For example, you can write Oracle CQL code that performs a join that involves streaming event data and data in a relational database table or cache.

Oracle CQL is extensible through data cartridges, with included data cartridges that provide support for queries that incorporate functionality within Java classes. For example, there is data cartridge support for spatial calculations and JDBC queries. The spatial data cartridge supports a large number of moving objects, such as complex polygons and circles, 3D positioning, and spatial clustering.

1.6 Technologies in Oracle Event Processing

Oracle Event Processing is made up of the following standard technologies that provide functionality for developing application logic and for deploying and configuring applications.

- Java programming language: Much of the Oracle Event Processing server functionality is written in the Java programming language. Java is the language you use to write logic for event beans and Spring beans.

- Spring: Oracle Event Processing makes significant use of the Spring configuration model. Spring is a collection of technologies that developers use to connect and configure parts of a Java application. Oracle Event Processing applications also support adding logic as Spring beans, which are Java components that support Spring framework features.

  You can find out more about Spring at the project's web site:http://www.springsource.org/get-started.

- OSGi: Oracle Event Processing application components are assembled and deployed as OSGi bundles. You can find out more about OSGi at: http://en.wikipedia.org/wiki/OSGi.

- XML: Oracle Event Processing application configuration files are written in XML. These files include the assembly file, which defines relationships between EPN stages and other design-time configurations. A separate configuration XML file that contains settings that can be modified after the application is deployed, including Oracle CQL queries.

- SQL: Oracle CQL extends SQL with functionality designed to address the needs of applications that use streaming data.

- Hadoop: Oracle CQL developers can access big data Hadoop data sources from query code. Hadoop is an open source technology that provides access to large data sets that are distributed across clusters. One strength of the Hadoop software is that it provides access to large quantities of data not stored in a relational database.

  For more information about Hadoop, start with the Hadoop project website at http://hadoop.apache.org/.

- NoSQL: Oracle CQL developers can access big data NoSQL data sources from query code. The Oracle NoSQL Database is a distributed key-value database. In Oracle NoSQL, data is stored as key-value pairs, which are written to particular storage stages. Storage stages are replicated to ensure high availability, rapid fail over in the event of a stage failure, and optimal load balancing of queries.

  For more information about Oracle NoSQL, see the Oracle Technology Network page at http://www.oracle.com/technetwork/products/nosqldb/.

- REST Inbound and Outbound Adapters: REST Inbound and Outbound Adapters allow to consume events from `HTTP Post` requests and receive events processed by the EPN.

- MBean API (JMX Technology): Allows administrative operations and dynamic EPN changes.

- Oracle Business Rules: Allows to define your own business logic and build applications.

1.7 Oracle Event Processing High-Level Use Cases

The use cases described in this section illustrate specific uses for Oracle Event Processing applications.

Financial: Responsive Customer Relationship

Acting on an initiative to improve relationships with customers, a retail bank designs an effort to provide coupons tailored to each customer's purchase pattern and geography.

The bank collects automated teller machine (ATM) data, including data about the geographical region for the customer's most common ATM activity. The bank also captures credit card transaction activity. Using this data, the bank can push purchase incentives (such as coupons) to the customer in real time based on where they are and what they tend to buy.

An Oracle Event Processing application receives event data in the form of ATM and credit card activity. Oracle CQL queries filter incoming events for patterns that isolate the customer's geography by way of GPS coordinates and likely purchase interests nearby. This transient data is matched against the bank's stored customer profile data. If a good match is found, a purchase incentive is sent to the customer in real time, such as through their mobile device.

Telecommunications: Real-Time Billing

Due to significant growth in mobile data usage, a telecommunications company with a large mobile customer base wants to shift billing for data usage from a flat-rate system to a real-time per-use system.

The company tracks IP addresses allocated to mobile devices and correlates this with stored user account data. Additional data is collected from deep-packet inspection (DPI) devices (for finer detail about data plan usage) and IP servers, then inserted into a Hadoop-based big data warehouse.

An Oracle Event Processing application receives usage information as event data in real time. Through Oracle CQL queries, and by correlating transient usage data with stored customer account data, the application determines billing requirements.

Energy: Improving Efficiency Through Analysis of Big Data

A company offering data management devices and services needs to improve its data center coordination and energy management to reduce total cost of ownership. The company needs finer-grained, more detailed sensor and data center reporting.

The company receives energy usage sensor data from disparate resources. Data from each sensor must be analyzed for its local relevance, and must be aggregated with data from other sensors to identify patterns that can be used to improve efficiency.

Separate Oracle Event Processing applications provide a two-tiered approach.

One application, deployed in each of thousands of data centers, receives sensor data as event data. Through Oracle CQL queries against events representing the sensor data, this application analyzes local usage, filtering for fault and problem events and sending alerts when needed.

The other application, deployed in multiple central management sites, receives event data from lower-tier applications. This application aggregates and correlates data from

across the system to identify consistency issues and produce data to be used in reports on patterns.

2

Oracle JDeveloper Quick Reference

Oracle Event Processing application development tasks can be performed in a number of ways in Oracle JDeveloper with SOA Product Components for the 12*c* release. If you are new to Oracle JDeveloper, you will discover other ways to locate the same tasks as you become more familiar with it.

For more information, see http://www.oracle.com/technetwork/middleware/complex-event-processing/downloads/index.html.

The following section describes one way to locate each task. This chapter covers the following topics:

- Setting Accessibility Options

- Oracle Event Processing Support

- Open Oracle JDeveloper Windows

- Create an Oracle Event Processing Project

- Project Templates

- Assembly and Component Configuration Files

- Set the Path to Project Source Files

- Perform Project-Level Actions

- Import a Zip or JAR file

- EPN Diagram Features

- Components Window.

- Context Menus.

2.1 Setting Accessibility Options

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

2.2 Oracle Event Processing Support

When you launch Oracle JDeveloper in the Studio Developer (All Features) role, it provides a full feature set for creating Oracle Event Processing applications. Figure 2-1 shows an Oracle Event Processing application open in Oracle JDeveloper.

Figure 2-1   An Oracle Event Processing Application in Oracle JDeveloper
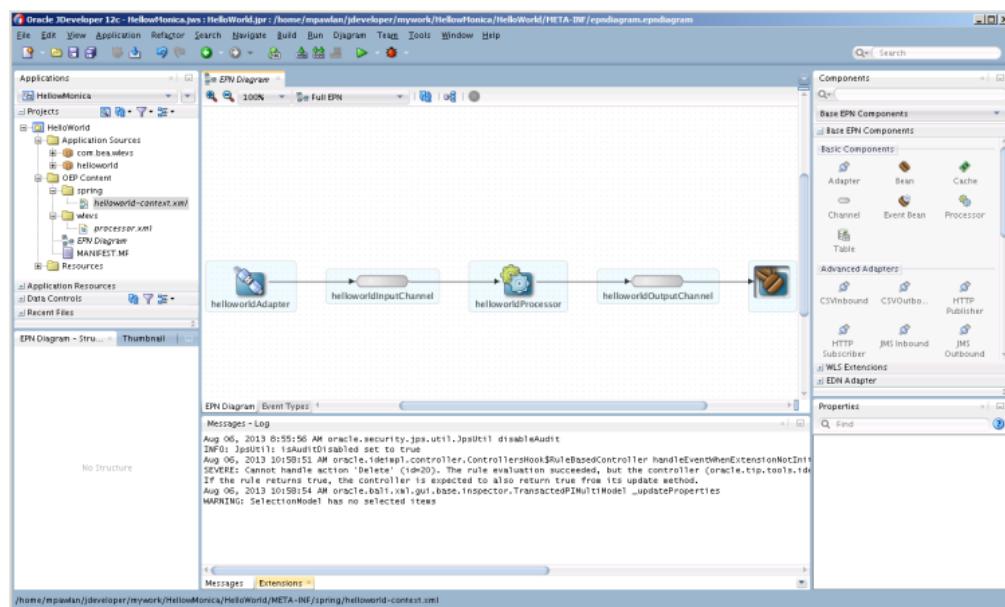
2.3 Open Oracle JDeveloper Windows

Use the Window menu to display the Oracle JDeveloper windows you need, such as the Components window. See Components Window.

2.4 Create an Oracle Event Processing Project

Use the File menu to create the following project files.

Create a New Oracle Event Processing Application
Select File > New > Application > OEP Application.

An application is a container for projects.

Create a New Oracle Event Processing Project
Select File > New > Project > OEP Project.

You add projects to applications.

Import an Oracle Event Processing Bundle
Select File > Import > OEP Bundle into New Project.

You can add an existing project to the application. Use this option to import an Eclipse project as described in Import an Eclipse Project into Oracle JDeveloper.

Create a New Project Library
Select File > New > Project > OEP Library Project.

Add libraries to your projects to provide additional functionality such as utilities and common Java classes that can be shared across projects.

2.5 Project Templates

When you create a project, you can choose from a selection of Oracle Event Processing application templates on which to base the project. The templates provide basic functionality including an EPN and the assembly and configuration files for the following types of projects:

Empty OEP Project template that provides the basic structure for an empty Oracle Event Processing application. Use this application template when none of the other templates meet your needs.

FX template that simulates a foreign currency exchange application.

Hello World template that provides a simple application that sends the Hello World message to the server console.

Signal Generation template that simulates receiving stock market events and generating signals for changes in the price or volume.

2.6 Assembly and Component Configuration Files

The assembly file is a context file that describes the EPN diagram stages and structure. The component configuration file describes component configuration and the dynamic parameters of the EPN stages. An application can have one or more assembly files and one or more configuration files.

Oracle JDeveloper provides default assembly and configuration files that it creates when you add components to the EPN and make connections. By default, the assembly file name is `<Project_Name>.context.xml`, and the configuration file name is `processor.xml`

When you add components to the EPN, you can change the default configuration file name to another file. If the alternate file already exists, Oracle JDeveloper saves the configuration in that file. If the file does not already exist, Oracle JDeveloper creates the file. For example, you might want to store all adapter configuration information in the `adapter.xml` configuration file.

You can also explicitly create assembly and configuration files. See Assembly and Configuration Files.

2.7 Set the Path to Project Source Files

To set the path to the Oracle Event Processing source files, select the project and select **Edit > Properties > Project Source Paths**.

2.8 Perform Project-Level Actions

In the Applications window, select and right-click an Oracle Event Processing project to display a context menu with the following options:

Open EPN Diagram: Displays the Oracle Event Processing EPN diagram. See EPN Diagram Features.

Configure JDBC Context: Use this option to configure a JDBC context. A JDBC context defines an application context for an instance of an Oracle JDBC data cartridge. Use this option only when you use a non-Oracle JDBC driver.

Configure Spatial Context: Use this option to configure a spatial context to manage a large number of moving objects such as complex polygons and circles, 3D positioning, and spatial clustering.

Deploy > oep-profile: Use this option to select an Oracle Event Processing deployment bundle or to assemble a new deployment bundle.

Deploy > New Deployment Profile: Create a deployment profile for your application. An application can have any number of deployment profiles.

Encryption Manager: Use this option to encrypt the application.

2.9 Import a Zip or JAR file

Use Import > OEP Bundle into New Project to import a zip or JAR file into Oracle JDeveloper. You cannot import a zip or JAR file from the command line.

2.10 EPN Diagram Features

The EPN diagram has a number of features that you can use when you create and edit an EPN. The EPN diagram uses an optimized layout by default. After you add, move, or delete components from the EPN diagram, the diagram updates and adjusts the layout.

Open the EPN Diagram

1. Expand **[ProjectName]** > **OEP Content**.

2. Double-click **EPN Diagram**.

   The EPN diagram opens in the EPN Types tab in the middle pane. Next to the EPN tab is the Event Types tab. The Event Types tab enables you to create an event type. EPN tabs also display at the bottom of the left pane: EPN Diagram - Structure and Thumbnail. The structure view shows the EPN diagram component tree.

Create an Event Type

1. Open the **EPN Diagram**.

   The EPN diagram opens in the EPN tab in the middle pane.

2. Select the **Event Types** tab next to the EPN tab.

3. Provide the event type information.

   See Create an Event Type to Carry Event Data for information on how to create an event.

Add Component

Drag a component from the Components window onto an empty area in the EPN to build the EPN diagram. See Components Window.

Delete Component

Right-click the component and select Delete from the context menu or select the component and press the Delete key.

Rename Component

1. Select the component on the EPN diagram.

    Oracle JDeveloper highlights the component.

2. Click the component name.

    The in-line name editor displays.

3. Change the name.

4. Click an empty area on the EPN diagram.

    The in-line name editor closes.

Editors

You can edit an EPN through different Oracle JDeveloper editors: XML Source code, property sheets, EPN Diagram, Manifest Editor, and so on. Validation annotations to indicate errors show in all editors. The editors provide the following features when you create Oracle CQL statements:

*   Syntax highlighting

*   Code completion

*   Syntax validation

*   Dynamic semantic validation

*   Parameterized Oracle CQL statements

You can add a bindings block to an parameterized query. As you code the bindings block, code completion suggests the binding ID with the list of available query IDS within the current processor scope. The following example shows how to add a binding block to a query.

```
<query id="helloworldRule">
<![CDATA[ select :1 from helloworldInputChannel ]]>
<bindings>
    <binding id="helloworldRule">
        <params id="param1">'My message is here: ' || message as message</params>
    </binding>
</bindings>
```

Badges

A badge is a small icon that displays on a stage. The badge displays additional information about the component. For example, if there is a validation error or warning related to the component, the EPN editor displays the error or warning badge on that component. When the situation causing the badge to appear resolves, the badge disappears.

*   Mouse over the badge to display the associated messages. Mouse click on the annotation badge to display the associated messages and a link to detailed information.

Zoom In or Out

You can zoom in or out on the EPN diagram in the following ways:

- Press the Ctrl key and mouse scroll.

- On the editor tool bar, choose a zooming value from the list of predefined zooming values.

- Select or open the thumbnail panel and use the mouse scroll.

Print the EPN Diagram

1. With the EPN editor open, select **File > Print Preview**.

2. In the **Print Preview** dialog, review the settings and click **Print**.

Export the EPN Diagram to an Image

1. Select **Diagram > Publish Diagram**.

2. Enter a file name and choose the appropriate file type.

3. Select **Save**.

Nested Components

When you define a child stage inside a parent stage, the child stage is nested. The nested stage is visible in the EPN diagram in an indented box. You cannot edit nested stages, but you can delete them.

Only the parent stage can specify the child stage as a listener. You can drag references from a nested element, but you cannot drag references to a nested element.

Foreign Components

A foreign component (foreign stage) is a component that is defined in a different application. On an EPN diagram, a foreign stage is visible as a ghost component. To reference a foreign stage, use the following syntax in the assembly file:

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

> **Note:**
>
> When you reference foreign stages, you must consider foreign stage dependencies when assembling, deploying, and redeploying an application.

2.11 Components Window

The Components window provides the Oracle Event Processing components for building an EPN. You drag the component you want to add to your EPN to a blank area on the EPN canvas and use the component wizard to configure it. You add the component to the EPN by dragging a component already in the diagram to the new component. The new component is placed to the right of the component that you dragged. See Create a Basic Application for step-by-step instructions.

The following list describes the components available on the Components window.

Base EPN Components:

- **Adapter**: Use an adapter to connect the EPN to external input or output data sources. The Adapter component represents a generic adapter that you can customize for your application requirements.

- **Bean**: Use a bean to define application event logic written in the Java programming language that conforms to standard Spring-based beans. See http://www.springsource.org/spring-framework.

- **Cache**: Use a cache to set up an area of random access memory (RAM) that holds copies of recently accessed data for ready access by an application. You must have a Cache System component in the EPN to add a Cache component.

- **Channel**: Use a channel to transfer events from stage to stage in the EPN.

- **Event Bean**: Use an event bean to define application event logic written in the Java programming language that conforms to the JavaBeans specification. The event bean is an Oracle extension to the regular Spring-based bean.

- **Processor**: Use a processor when you want to add Oracle CQL query code to your application. Oracle CQL can read from the big data Hadoop and NoSQLDB components.

- **Table**: Use a table as an external relation source. You can also use a table to store events in the database by configuring the table as a listener of an upstream component.

- Advanced Adapters:

  - **CSVInbound**: Use a CSVInbound adapter to accept data in the form of comma-separated values entering the EPN.

  - **CSVOutbound**: Use a CSVOutbound adapter to send data in comma-separated values out of the EPN.

  - **HTTP Publisher**: Use an HTTP Publisher adapter to send JavaScript Object Notation (JSON) event data out of the EPN to a web-based user interface.

  - **HTTP Subscriber**: Use an 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.

  - **JMS Inbound**: Use a JMS Inbound adapter to accept Java Message Service (JMS) topics entering the EPN.

  - **JMS Outbound**: Use a JMS Outbound adapter to send JMS topics out of the EPN.

  - **REST Inbound**: Use a REST Inbound adapter to consume events from `HTTP Post` requests.

  - **REST Outbound**: Use a REST Outbound adapter to receive events processed by the EPN.

- Big Data Extensions

  - **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.

- **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.

- **HBase**: A data cartridge extension for an Oracle CQL processor to access large quantities of data in an HBase Database.

- Cache Systems

  - **Coherence Cache System**: Use a Coherence Cache System component to set up a system to maintain consistent data that is stored in local caches on a shared resource.

  - **Local Cache System**: Use a Local Cache System to speed up network access to data files.

- CQL Patterns

  See Use Oracle CQL Patterns for information about how to use the patterns.

  - **Averaging Rule**: Use an Averaging Rule component to compute an average over a specified number of events (table rows).

  - **Detect Missing Event Rule**: Use a Detect Missing Event Rule component to detect when an expected event does not occur.

  - **Partitioning Rule**: Use a Partitioning Rule component to partition the event panel by an event property and display the specified number of events in the partition.

  - **Select With Subsequent Filtering Query**: Use a Select with Subsequent Filtering Query component to filter events to populate the view with events that pass the filter criteria.

  - **Select From Multiple Streams**: Use a Select From Multiple Streams component to join two streams to select from correlated events.

  - **Select With From**: Use a Select With From component to select events from a channel according to the specified properties.

  - **Select With Pattern Matching**: Use a Select With Pattern Matching component to select events from a channel according to specified property values.

    > **Note:**
    >
    > Oracle JDeveloper does not have an Oracle CQL visual editor. There is an Oracle CQL visual editor in Oracle Event Processing Visualizer. See *Using Visualizer for Oracle Event Processing*.

WLS Extensions:

- **RMIInbound**: Use an RMIInbound adapter to receive incoming data sent from Oracle WebLogic Server over the remote method invocation (RMI) protocol.

- **RMIOutbound**: Use an RMIOutbound adapter to send data to Oracle WebLogic Server over the RMI protocol.

EDN Adapters:

- **EDNInbound**: Use an EDNInbound adapter to receive incoming data from the Oracle SOA Suite event network.

- **EDNOutbound**: Use an EDNOutbound adapter to send outbound data to the Oracle SOA Suite event network.

2.12 Context Menus

Each stage on the EPN editor has a group of context menu items that provide convenient access to various stage-specific functions. Right-click the stage to display its context menu. Using the context menu, you can edit the stage configuration.

For different stages, though the stage wizard is the same, the parameter values and some of the options are greyed out that are read-only.

- **Add Configuration Source**: Adds a configuration file to the project.

- **Define Java Class**: Opens a wizard to that you can create a Java class.

- **Delete Configuration Source**. Deletes a configuration file from the project.

- **Go to Configuration Source**: Opens the corresponding component configuration file and positions the cursor in the appropriate element.

- **Go to Assembly Source**: Opens the corresponding EPN assembly file and positions the cursor in the appropriate element.

- **Go to Java Source**: Opens the corresponding Java source file for this component.

- **Encryption Manager**: Allows to edit all encrypted passwords in one place. This option is enabled on whole EPN diagram and project.

- **Delete**: Deletes the component from both the EPN assembly file and component configuration file (if applicable).

> **Note:**   These navigation options become disabled when a corresponding source artifact cannot be found. For example, if an adapter does not have a corresponding entry in a configuration XML file, its **Go to Configuration Source** menu item is greyed out.

3

Oracle JDeveloper Procedures

The following sections describes how to perform Oracle Event Processing tasks in Oracle JDeveloper.

This chapter covers the following topics:

- Import an Eclipse Project into Oracle JDeveloper
- Add a Library to a Project
- Create an Application Library
- Assembly and Configuration Files
- Configure a Relation Channel
- Configure an Application Time-Stamped Channel
- Create and Register a JavaBean Event Type
- Create and Register a Tuple Event Type
- Create an Event Bean
- Create a Spring Bean
- Configure a Table Source
- Configure a Table Sink
- Use Oracle CQL Patterns
- Configure an Oracle Coherence Caching System and Cache
- Configure a Local Caching System and Cache
- Debug Java Classes
- Testing with the Event Inspector Service
- Start and Stop Oracle JDeveloper and Servers.

3.1 Import an Eclipse Project into Oracle JDeveloper

You can import an Oracle Event Processing Eclipse project into Oracle JDeveloper as a bundle. A bundle is an Oracle Event Processing Eclipse project that is exported as an Archive (zip) or JAR file. There is no command-line command to import an Eclipse project into Oracle JDeveloper.

- Import an Eclipse Project into Oracle JDeveloper

- Build the Imported Project

- Start the Oracle Event Processing Server

- Deploy

Import an Eclipse Project into Oracle JDeveloper

Be aware that you cannot import an Eclipse project that consists of multiple applications or projects. The Eclipse project that you import can only be an Oracle Event Processing project. You cannot import a Coherence or Java project.

When you import a zip or JAR file from Eclipse, Oracle JDeveloper 12c supports JDK1.7 only. You cannot export your JDK 1.6 project from Eclipse and then import the project into Oracle JDeveloper using JDK1.7. You have to move from JDK 1.7 in Eclipse to JDK 1.7 in Oracle JDeveloper.

> **Note:**
>
> You can import a JDK1.6 Eclipse Oracle Event Processing application project, but Oracle JDeveloper does not handle any compilation issues for you. In this case, it is your responsibility to handle compilation issues if they appear.

1. In Eclipse, export your Eclipse project as a zip or JAR file.

    Make sure you include Source files and Resources.

2. In Oracle JDeveloper, select **File > Import > OEP Bundle into New Project**.

    The steps are different depending on whether you have an existing Oracle JDeveloper application or not.

    If you have an existing application, the Import OEP Bundle as Project - Step 1 of 2 dialog displays to import the bundle into the active application.

    a. In the **Step 1 of 2** dialog, provide the name of the project, use or change the directory name, and click **Next**.

        The Import OEP Bundle as Project - Step 3 of 3 dialog displays.

    b. In the **Step 2of 2**dialog, find and select the exported the Eclipse zip file and click **Finish**.

        The project displays in Oracle JDeveloper under Applications.

    If you do not have an existing application, The Import OEP Bundle as Project - Step 1 of 3 dialog displays so that you can create an application for the project.

    a. In the **Step 1 of 3** dialog, provide the name and location for the application and click **Next**.

        The Import OEP Bundle as Project - Step 2 of 3 dialog displays.

    b. In the **Step 2 of 3** dialog, provide the name of the project, use or change the directory name, and click **Next**.

        The Import OEP Bundle as Project - Step 3 of 3 dialog displays.

    c. In the **Step 3 of 3** dialog, find and select the exported the Eclipse zip file and click **Finish**.

The project displays in Oracle JDeveloper under Applications.

Build the Imported Project

1. Select the imported application and select `Build > <project-name>`.

2. If you see errors in the log window indicating you need additional JAR files in the class path, then select the top-level project folder and select **Project Properties**.

    The Project Properties dialog displays.

3. In the **Project Properties** dialog left panel, select **Libraries and Classpath**.

    The Libraries and Classpath dialog displays.

4. In the **Libraries and Classpath** dialog, click **Add Jar/Directory**.

    The Add Archive or Directory dialog displays.

5. In the **Add Archive or Directory** dialog, locate the JAR files that you need to add.

6. Click **OK**.

7. Rebuild the project.

8. Repeat this process until you have located and added all of the required files.

9. If you see problems in the source code, then use the quick fixes to organize and add imports.

Start the Oracle Event Processing Server

See Start and Stop Oracle JDeveloper and Servers.

Deploy

1. Right click the project.

    The context menu displays.

2. From the context menu, select **Deploy > New Deployment Profile**.

    The Create Deployment Profile dialog displays.

3. In the **Create Deployment Profile** dialog, choose a deployment profile, such as OEP Project Deployment Profile, and give it a name.

4. Click **OK**.

    The OEP Project Deployment Profile dialog displays.

5. In the **OEP Project Deployment Profile** dialog, you can either create a new Oracle Event Processing server connection or select an existing connection.

    To select an existing connection:

    a. In the Connection to OEP Server drop-down list, select the existing connection you want to use.

    b. In the **OEP Project Deployment Profile** dialog, accept the defaults or provide the requested profile details.

c. Click **OK**.

To create a new connection:

a. Click the Add (+) button to create an Oracle Event Processing server connection.

The Create OEP Server Connection dialog displays.

b. In the **Create OEP Server Connection** dialog, provide the connection details:

OEP Server Connection Name: SampleOEPConnection OEP Server Home Path: /Oracle/Middleware/my_oep/ Use Default Values: Unchecked. OEP Server Projects Directory: user_projects/domains/sample_domain/ defaultserverUse Default Values: Unchecked Host: localhost Port: 9002 Use Default Values: Unchecked Username: oepadmin User Password: welcome1 Additional Parameters for OEP Server: blank

c. In the **Create OEP Server Connection** dialog, click **Test** Connection.

If everything is okay, then *Success* displays in the message box below the Test Connection button. If you have errors, locate and fix the errors and try again.

d. When you see **Success**, click **OK**.

The OEP Project Deployment Profile dialog displays.

e. In the **OEP Project Deployment Profile** dialog, accept the defaults or provide the requested profile details.

f. Click **OK**.

To regenerate `MANIFEXT.MF`:

- Click the **Recreate MANIFEXT.MF** button to regenerate the manifest file.

  The manifest file along with the imported and exported packages is regenerated.

6. Right click the project and select the deployment profile you just selected or created.

The Deploy <profile-name> dialog displays.

7. In the **Deploy <profile-name>** dialog, select **Deploy OSGi bundle to target platform** and click **Next**.

The Summary dialog displays.

8. In the **Summary dialog**, review the settings and click **Finish**.

9. Wait a few moments while the deployment finishes.

The Deployment finished message displays on the Deployment - Log tab. If there are problems starting the application, it undeploys automatically.

10. View the server log and the list of deployed applications.

To view the server log:

a. If the Resources window is not open, select **Window > Resources**.

The Resources window displays.

b. Navigate to **Resources Window > IDE Connections > OEP Server**.
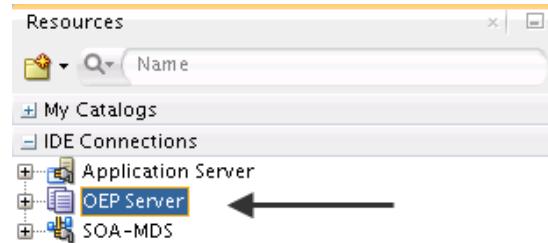
c. Expand OEP Server to see a list of OEP server connections.

d. Right click a connection.

The context menu displays.

e. From the context menu, select **Open OEP Server Log Page**.

The OEP server log page opens and you can see the server log messages.

To view the list of deployed applications:

a. If the Resources window is not open, Select **Window > Resources**.

The Resources window displays.

b. Navigate to **Resources Window > IDE Connections > OEP Server**.
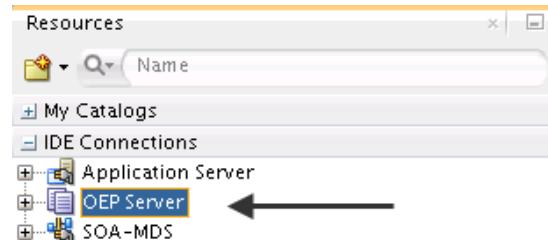
c. Expand OEP Server to see a list of OEP server connections.

d. Navigate to **Resources window > IDE Connections > OEP Server**.

A list of OEP server connections displays.

e. Under **OEP Server** expand the connection for the application you just deployed.

```
-OEP Server    -SampleOEPConnection      +Applications
```

f. Under the expanded connection, expand **Applications**.

The list of deployed applications and their status displays. For example, you might see a listing similar to this:

```
application1 [RUNNING]
my_application2 [SUSPENDED}
```

3.2 Add a Library to a Project

You can add a library JAR file to your application as a resource.

1. In Oracle JDeveloper, right-click the project.

   The context menu displays.

2. In the context menu, select **Add Project Library**.

   The Add Project Library dialog displays.

3. In the Add Project Library dialog in the **Library JAR** field, navigate to the library JAR file you want to add to your application.

4. Click OK.

   The JAR file displays under Resources for the project, and the project manifest file updates accordingly.

3.3 Create an Application Library

You can create an application library to share among applications running in the same domain.

- Create an Oracle Event Processing Library Project
- Create Deployment Profile and Deploy

Create an Oracle Event Processing Library Project

1. In Oracle JDeveloper, select File > New from Gallery.

   The New Gallery dialog displays

2. In the **New Gallery** dialog in the left window, select **OEP Tier**, and in the right window, select **OEP Library Project**.

3. Click **OK**.

   The Create OEP Library Project - Step 1 of 2 dialog displays.

4. In the Create OEP Library Project - **Step 1 of 2** dialog, enter the Project name and optionally a directory location.

5. Click **Next**.

   The Create OEP Library Project - Step 3of 2 dialog displays.

6. In the Create OEP Library Project - **Step 2of 2** dialog, locate the JAR file that you want the library project to contain.

7. In the Create OEP Library Project - **Step 2of 2** dialog, provide the other information or accept the defaults and click **Finish**.

   If your library is a driver, check the Deploy to Library Extensions check box so the library activates in the correct order. See *Developing Applications for Oracle Event Processing*.

   The library project displays in the application under Projects.

Create Deployment Profile and Deploy

1. Right-click the library project. and select **Deploy**.

2. Select either app_lib_profile-*n* or create a new deployment profile for this library.

3. In the Deployment Action dialog, select **Deploy the library JAR to OEP Server** and click **Next**.

4. Review the **Deployment Summary** and click **Finish**.

   The library deploys to the local server.

   > **Note:**
   >
   > In 12*c* the Oracle JDeveloper deployment profile supports only local Oracle Event Processing connections.

5. Restart the Oracle Event Processing server.

3.4 Assembly and Configuration Files

Oracle JDeveloper creates assembly and configuration files as you add components to the EPN and make connections. You can also create your own assembly and configuration files to use instead of the defaults. When a component wizard lists the default `processor.xml` file, you can replace the default with the file you create. An application can have one or more assembly files and one or more configuration files.

- The assembly file is a context file that describes the EPN diagram stages and structure. By default, the assembly file name is `<Project_Name>.context.xml`.

- The configuration file describes component configuration and the dynamic parameters of the EPN stages. By default, the configuration file name is `processor.xml`.

When you add components to the EPN, you can change the default configuration file name to another file. If the alternate file already exists, Oracle JDeveloper saves the configuration in that file. If the file does not already exist, Oracle JDeveloper creates the file and saves the configuration in it. For example, you might want to store all adapter configuration information in the `adapter.xml` configuration file. If you do not specify any configuration settings when you create the component, Oracle JDeveloper does not create a new configuration file because there is no configuration to put in it.

> **Note:**
>
> Identifiers and names in XML files are case sensitive. Use the same case when you reference the component ID in the assembly file.

The walkthroughs in this guide have example assembly and configuration files that you can study. See also *Developing Applications for Oracle Event Processing* for examples of assembly and configuration files for many of the Oracle Event Processing components.

3.4.1 Create an Assembly File

1. In Oracle JDeveloper with the project selected, select **File > New > From Gallery**.

   The New Gallery dialog displays.

2. In the New Gallery dialog under **Categories**, expand **OEP Tier** and select **OEP Files**.

3. In the New Gallery dialog under **Items**, Select **OEP Assembly File** and click **OK**.

   The Create OEP Assembly File dialog displays.

4. In the **Create OEP Assembly File** dialog, provide a file name and directory location or accept the defaults.

   Provide an assembly file name that associates the assembly file with a project.

5. Click **OK**.

   The new assembly file displays in the left pane under the project in OEP Content > Spring.

3.4.2 Create a Component Configuration File

1. In Oracle JDeveloper with the project selected, select **File > New > From Gallery**.

   The New Gallery dialog displays.

2. In the New Gallery dialog under **Categories**, expand **OEP Tier** and select **OEP Files**.

3. In the New Gallery dialog under **Items**, Select **OEP Config File** and click **OK**.

   The Create OEP Config File dialog displays.

4. In the **Create OEP Config File** dialog, provide a file name and directory location or accept the defaults.

   Provide a configuration file name that associates the configuration file with a specific component, group of components, or type of component.

5. Click **OK**.

   The new configuration file displays in the left pane under the project in OEP Content > wlevs.

3.4.3 Add Components to a Configuration File

In Oracle JDeveloper, you can drag components from the Component window to an open configuration file. This does not work with the assembly file.

1. In Oracle JDeveloper, open the configuration file to which you want to add a component.

   The configuration file opens in the middle panel, and the Components window displays in the right panel.

2. Place your cursor in the configuration file where you want to add the component.

   A blinking cursor displays at that location.

3. On the Components window, locate the component you want to add.

4. Drag-and-drop the component onto the open configuration file.

   The wizard for that component displays.

If you chose an invalid location in the configuration file, Oracle JDeveloper displays an error message so you can choose a valid location.

5. Enter the configuration information prompted by the configuration wizard and click **OK**.

   Oracle JDeveloper adds the configuration for that component to the configuration file and updates the assembly file with the corresponding assembly settings as needed.

3.4.4 Add Configuration Settings to a Component

When you add components to the EPN, sometimes you provide custom configuration settings and sometimes you accept the default configuration. When you accept the default configuration, Oracle JDeveloper does not add any entries for that component to the configuration file. Oracle JDeveloper also does not create a configuration file even if you specify a new configuration file name during the configuration process.

Later, if you decide to provide default settings, you need to either create a configuration file as described in Assembly and Configuration Files and add the complete component configuration, or you can generate the configuration in the default `processor.xml` file.

The following example shows how to generate a configuration entry for AdapterOutputChannel in the `processor.xml` file.

Add Channel Configuration

1. Right-click the channel component in the EPN diagram.

   The context menu displays.

2. In the context menu, select **Add Configuration Source**.

   The `processor.xml` file opens and displays the default configuration for the component. You can edit the default configuration to customize it.

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <wlevs:config xmlns:wlevs="http://www.bea.com/ns/wlevs/config/application">
     <channel>
       <name>AdapterOutputChannel</name>
     </channel>
   </wlevs:config>
   ```

3. Make the channel multithreaded by adding the `max-threads` element:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <wlevs:config xmlns:wlevs="http://www.bea.com/ns/wlevs/config/application">
     <channel>
       <name>AdapterOutputChannel</name>
       <max-threads>4</max-threads>
     </channel>
   </wlevs:config>
   ```

   The maximum number of threads that Oracle Event Processing server can use to process events for this channel is four.

3.5 Configure a Relation Channel

The default channel has a name, an ID, and is a system time-stamped, single-threaded stream channel with a default heartbeat time out of 100 milliseconds or 100,000,000 nanoseconds. See *Developing Applications for Oracle Event Processing* for more information about channel configuration. You can change the default channel to a relation by adding the `is-relation="true"` element and attribute to the assembly file.

A relation channel supports insert, delete, and update operations.

1. In the Oracle JDeveloper EPN editor, right-click a **channel** stage and select **Go To Assembly Source**.

   The assembly file displays.

2. In the assembly file, the cursor blinks next the channel definition.

   The channel definition line looks similar to the following example:

   ```
   <wlevs:channel id="TestChannel" event-type="TestEventType"/>
   ```

3. To change the channel to a relation, add an `is-relation="true"` setting:

   ```
   <wlevs:channel id="TestChannel" event-type="TestEventType" is-relation="true"
   primary-key="testPrimaryKey" />
   ```

   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.

3.6 Configure an Application Time-Stamped Channel

You can configure a channel to be time stamped by an application. In this case, the time-stamp of an event is determined by the configurable `wlevs:expression` element. A common example of an expression is a reference to a property on the event. If no expression is on the event, then the time stamp is propagated from a prior event. For example, when you have a channel that is time stamped from the system from one Oracle CQL processor feeding events into an channel that is time stamped by an application from another downstream Oracle CQL processor.

Make sure you have an event type created. A channel needs to know the event type to send the data to the correct stage.

1. Create a default channel.

2. In the application assembly file, add a `wlevs:application-timestamped` child element.

3. In the `wlevs:application-timestamped` child element, specify a `wlevs:expression` child element for Oracle Event Processing to use to generate time stamp values. For example:

   ```
   <wlevs:channel id="fxMarketAmerOut" event-type="eventtype" >
       <wlevs:application-timestamped>
           <wlevs:expression>mytime+10</wlevs:expression>
       </wlevs:application-timestamped>
   </wlevs:channel>
   ```

4. Configure the optional `wlevs:application-timestamped` attribute, `is-total-order`. When true, the `is-total-order` attribute indicates that the application time published is always strictly greater than the last value used.

   The attributes are described in *Developing Applications for Oracle Event Processing*.

   For example:

   ```
   <wlevs:channel id="fxMarketAmerOut" event-type="eventtype" >
       <wlevs:application-timestamped is-total-order="true">
           <wlevs:expression>mytime+10</wlevs:expression>
       </wlevs:application-timestamped>
   </wlevs:channel>
   ```

5. Save and close the assembly file.

3.7 Create and Register a JavaBean Event Type

1. Select the Oracle JDeveloper project to which you want to add the event type.

2. Create a JavaBean with a no-argument, public constructor.

3. Optional. Make the class serializable if you plan to cache events in Oracle Coherence.

4. Add the private fields and accessor methods to the JavaBean.

5. In Oracle JDeveloper, with the EPN diagram open, use the Event tab to configure the event type with properties such as the name of the JavaBean.

   The Event tab enables you to declare and edit event types. When you close the Event tab, the event type you created or edited is registered in the corresponding Event Type Repository section of the application assembly file.

   a. Under **Event Type Definitions**, select the application assembly file.

   b. Click the Add (**+**) button.

      The Event Type Details panel displays on the left.

   c. Under **Event Type Details**, select **Properties Defined in JavaBean**.

   d. Provide the name of the JavaBean class.

3.8 Create and Register a Tuple Event Type

This procedure describes how to create and register an Oracle Event Processing event type as a tuple using the Oracle Event Processing IDE event type repository editor. When you design your event, you must restrict your design to the even data types that Oracle Fusion Middleware Developing Application for Oracle Event Processing describes.

Create a Tuple Event Type in Oracle JDeveloper

1. Select the Oracle JDeveloper project to which you want to add the event type.

2. In Oracle JDeveloper with the EPN diagram open, use the Event tab to configure the event type with properties such as the name of the JavaBean.

The Event tab enables you to declare and edit event types. When you close the Event tab, the event type you created or edited is registered in the corresponding Event Type Repository section of the application assembly file.

   a. Under **Event Type Definitions**, select the application assembly file.

   b. Click the Add (**+**) button.

      The Event Type Details panel displays on the left.

   c. Under **Event Type Details**, select **Properties Declaratively**.

   d. In the **Type Name** field, enter a name for the new event type.

   e. Under Event Type Properties use the Add (**+**) button to add a property row to the Event Type Properties list.

   f. Place your cursor inside the **Name** column to edit the property name.

   g. Place your cursor inside the **Type** column and choose a data type from the drop-down list.

      The char data type has a default length of 256 characters that you can edit by placing your cursor inside the **char length** column.

3.9 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.

1. Optionally, create the Java class you want to use as described in *Developing Applications for Oracle Event Processing*.

In step 3, you can select an existing class or create a new one and add the logic later.

2. In Oracle JDeveloper with the EPN diagram open, drag the **Event Bean** component from the Components window to an empty area on the EPN diagram.

The New EventBean wizard displays.

3. In the **New EventBean** wizard, provide the following information:

EventBean ID: A unique identifier for this event bean. EventBean class: Add (+) or choose the JavaBean class (event type) you want to use for this event bean.

Oracle Fusion Middleware Developing Application for Oracle Event Processing for information about making the Java class an event sink, event source, or both.

4. Click **OK**.

Oracle JDeveloper adds the event bean to the EPN.

5. Drag the upstream component to the event bean to place the event bean in its correct location in the EPN.

The EPN diagram adjusts to show the event bean in its correct location.

Example 3-1   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 `Bean Output Channel` component.

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

Example 3-2   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>
```

3.10 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.

1. Optionally, create the JavaBean event type you want to use as described in Oracle Fusion Middleware Developing Application for Oracle Event Processing .

   In step 3, you can select an existing class or create a new one and add the logic later.

2. In Oracle JDeveloper with the EPN diagram open, drag the **Bean** component from the Components window to an empty area on the EPN diagram.

   The New Bean wizard displays.

3. In the **New Bean** wizard, provide the following information:

   Bean ID: A unique identifier for this event bean. Bean class: Add (+) or choose the JavaBean class (event type) with the Spring functionality that you want to use for this bean.

4. Click **OK**.

   Oracle JDeveloper adds the event bean to the EPN.

5. Drag the upstream component to the event bean to place the event bean in its correct location in the EPN.

   The EPN diagram adjusts to show the event bean in its correct location.

3.11 Configure a Table Source

You can access data in a relational database table from an Oracle CQL query by adding a table source component to your application. When you add a table source, you associate it with a data source for read access to the relational database table. Oracle Event Processing relational table sources are pull data sources, which means that Oracle Event Processing periodically checks the event source for new data to read from the database.

- You can join a stream only with a NOW window.

  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.

  You can join more than one database table or view in a join.

- With an 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 table sources in the Oracle Event Processing server file as described in *Administering Oracle Event Processing*.

Create a Table Source

1. In Oracle JDeveloper open the EPN diagram.

2. In the Components window under **Basic Components**, drag the **Table** component to an empty area on the EPN.

   The New Table wizard opens.

3. In the New Table wizard, enter the following values and click **OK**:

   Table ID: Stock Event Type: TradeEvent Data Source: StockDataSource

   By default, the table source stage uses the name of the event type as the default table name in the database. Also, you can explicitly specify the table name with `table-name` elements. The `table-name` element provides the name of the database table from which you want to get the event data.

   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`.

The assembly file has entries to associate the `Stock` table with the `proc` processor follows:

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

<wlevs:processor id="proc">
 <wkevs:table-source ref="Stock" />
</wlevs:processor>
```

> **Note:**
>
> The XMLTYPE property is not supported for table sources.

Create the Data Source

1. In Oracle JDeveloper in the configuration file, add the following lines to define the data source:

```
<data-source>
  <name>StockDataSource</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>StockDataSource</element>
    </jndi-names>
    <global-transactions-protocol>None</global-transactions-protocol>
  </data-source-params>
</data-source>
```

2. Save the file.

Example 3-3   Read Data from the Stock Database Table

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.

3.12 Configure a Table Sink

You can update and delete data in a relational database table from an Oracle CQL query by adding a table sink component to your application. A table sink receives data from an upstream component and performs update and delete operations on the underlying relational database table according to the data received.

You can store incoming events in a relational database by adding a table sink component to your application. When an event comes into the table sink, it is persisted into the database and then send to the downstream stages. Oracle Event Processing does not create the database table. You must create the database table before you run the application. You must also maintain and back up the table as needed.

You create a table sink similar to how you create a table source. After you drag the Table component to the EPN diagram and provide the ID, Event Type, and Data Source, you edit the assembly file entry to include the required `table-name` and `key-properties` elements. These elements are not required for table sources.

The `table-name` element provides the name of the database table to which you want to store the event data. The `key-properties` element provides the unique key value for the database table to enable Oracle CQL queries that perform update and delete operations.

Datatypes for Conversion between CQL, Java, and JDBC

The following table lists the data types for the conversion between CQL, Java, and JDBC:

CQL Native Type	Java Primitive Type	Java Wrapper Type	JDBC Types
BOOLEAN	boolean	Boolean	BOOLEAN, BIT
INT	int	Integer	INTEGER
BIGINT	long	Long	BITINT
FLOAT	float	Float	REAL
DOUBLE	double	Double	DOUBLE
CHAR	char[]	String	VARCHAR, LONGVARCHAR
BYTE	byte[]	Byte[]	VARBINARY, LONGVARBINARY
XMLTYPE	N/A	java.sql.SQLXML	Not Supported
TIMESTAMP	long	java.util.date, java.sql.Time, java.sql.Timestamp	TIMESTAMP
BIGDECIMAL	N/A	java.math.BigDecimal	NUMERIC
INTERVAL	N/A	Not Supported	VARCHAR, LONGVARCHAR
INTERVALYM	N/A	Not Supported	VARCHAR, LONGVARCHAR
OBJECT	N/A	Class	Not Supported

Assembly File

```
<wlevs:table id="StockSink" event-type="TradeEvent" data-source="StockDataSource"
  table-name="StockEvents" key-properties="symbol" />
```

Data Source Configuration

The data source configuration is the same for table sources and table sinks.

Store Data in the StockEvents Database Table

The following Oracle CQL query gets data from an input channel and sends it to the table sink to persist the event data.

```
SELECT * FROM StockTraceIStreamChannel
```

3.13 Use Oracle CQL Patterns

Oracle JDeveloper provides the following seven Oracle CQL patterns to make it easier for you to form Oracle CQL queries in your applications. Each Oracle CQL pattern is stored within the context of an Oracle CQL processor. The processor can already be in the EPN or not already be in the EPN.

- **Averaging Rule**: Use an Averaging Rule component to compute an average over a specified number of events (table rows).

- **Detect Missing Event Rule**: Use a Detect Missing Event Rule component to detect when an expected event does not occur.

- **Partitioning Rule**: Use a Partitioning Rule component to partition the event panel by an event property and display the specified number of events in the partition.

- **Select With Subsequent Filtering Query**: Use a Select Filter Subquery component to filter events to populate the view with events that pass the filter criteria.

- **Select From Multiple Streams**: Use a Select From Multiple Streams component to join two streams to select from correlated events.

- **Select With From**: Use a Select With From component to select events from a channel according to the specified properties.

- **Select With Pattern Matching**: Use a Select With Pattern Matching component to select events from a channel according to specified property values.

Procedure

To add any of the available Oracle CQL patterns to the EPN, perform the following steps. Each Oracle CQL pattern is stored within the context of an Oracle CQL processor. The processor can already be in the EPN or *not* already be in the EPN.

If the processor is already in the EPN, drag and drop the pattern on the existing processor. If the processor is not already in the EPN, start with step 2

1. Indicate the processor in which to store the Oracle CQL pattern:

   a. If the processor is already in the EPN, drag and drop the pattern on the existing processor.

   b. If the processor is not already in the EPN, drag the pattern to an empty spot on the EPN diagram.

Step 1 of the two-step wizard for that pattern displays with default values.

2. In the **Oracle CQL Pattern wizard, Step 1 of 2** screen either accept the defaults or enter the following values. *Note that these values cannot be changed when you drag and drop the Oracle CQL pattern on an existing processor*.

    Processor ID: A unique ID value of the Oracle CQL processor which will store this Oracle CQL pattern. Oracle JDeveloper provides a default unique ID.

    File Name: The name of the configuration file where you want the Oracle CQL pattern configuration stored. Oracle JDeveloper provides the existing `processor.xml` configuration file for the default. If you selected an existing Oracle CQL processor, the file name field is unavailable because Oracle JDeveloper stores the Oracle CQL pattern configuration in the same file with the processor.

3. Click **Next**.

    Step 2 of the two-step wizard for that pattern displays with default values where possible.

4. In the **Oracle CQL Pattern wizard, Step 2 of 2** screen, accept the default values where appropriate and enter values where needed.

    To obtain information about valid values for a field, put your cursor in the field and read the property description in the right panel or Click Help.

    Step 2 has a parameters section on top with the Oracle CQL statement template below. As you provide parameters in the top section, the template reflects your inputs with color coding as shown in the following figure:
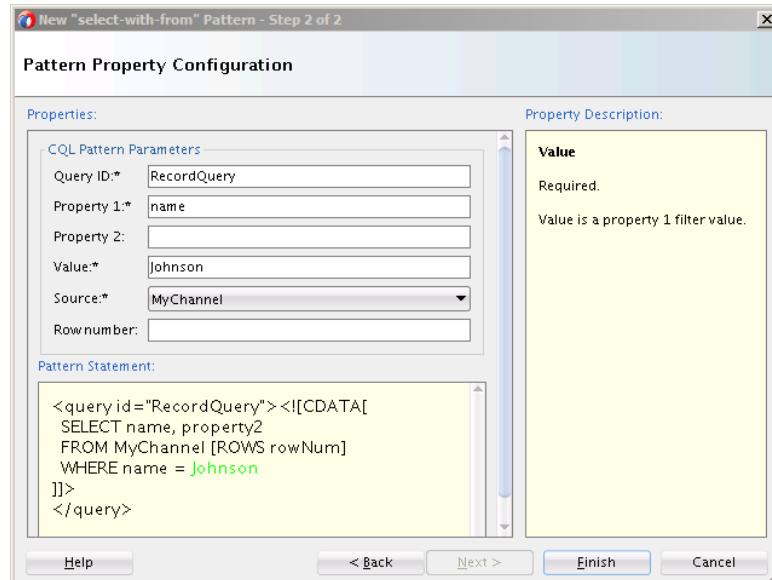
5. Click **Finish**.

    Oracle JDeveloper adds the Oracle CQL processing code to the processor without overwriting any existing rules. In this example the `processor.xml` file contains the following entries for the `MyProcessor` Oracle CQL processor

```
<processor>
  <name>MyProcessor</name>
    <rules>
```

```
            <query id="RecordQuery"><![CDATA[
              SELECT name
              FROM MyChannel
              WHERE name = "Johnson" ]]>
            </query>
        </rules>
    </processor>
```

If the processor is not already in the EPN, the new processor that contains the Oracle CQL pattern code is added to the EPN and connected to the component indicated in the `Source` field.

Valid Event Sources for Views and Queries

Added to an existing Oracle CQL processor:

- All channels that are the sources of events for the processor.

- All caches that are the sources of events for the processor.

- All tables that are the sources of events for the processor.

- All hadoop:files that are the sources of events for the processor.

- All nosql:stores that are the sources of events for the processor.

- All views of the current processor.

Added to a new processor:

- All channels

- All caches

- All tables

- All hadoop:files

- All nosql:stores

3.14 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://www.oracle.com/technetwork/middleware/coherence/overview/index.html.

Create an Oracle Coherence Caching System and Cache

This procedure configures 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.

1. In Oracle JDeveloper, open the EPN for your application.

2. From the **Components** window, select and drag the **Coherence Cache System** component to an empty area on the EPN.

    The Coherence Cache System Step 1 of 4 dialog displays with the following defaults:

    Cache System ID: coherence-caching-system Configuration location: `coherence-cache-config.xml`

    The `coherence-cache-config.xml` file is a per-application configuration file. It contains individual cache information in the `cache-name` element. When you complete this procedure, Oracle JDeveloper places the `coherence-cache-config.xml` file in the `META-INF/wlevs/coherence` directory of the bundle JAR.

3. Click **Next**.

    The Coherence Cache System - Step 2 of 4 dialog displays.

4. In the **Coherence Cache System - Step 2 of 4** dialog, provide the following values:

    Cache name: The name of the first cache in your Oracle Coherent caching system. Value Type: The type for values contained in the cache. Must be a valid type name in the event type repository.

5. Click **Next**.

    The Coherence Cache System - Step 3 of 4 dialog displays.

6. In the **Coherence Cache System - Step 3 of 4** dialog, select the **Advertise** check box.

    Selecting the Advertise check box means that the caching system allows other applications to access this cache system.

7. Click **Next**.

    The Coherence Cache System - Step 4 of 4 dialog displays.

8. In the **Coherence Cache System - Step 4 of 4** dialog, click **Finish**.

Example 3-4   Assembly File

The assembly file contains the information you provided when you created the caching system and `cache1`. 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-config.xml 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-config.xml 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.

Example 3-5   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://www.oracle.com/technetwork/middleware/coherence/overview/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>
```

Example 3-6   *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*.

3.15 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.

Create a Local Caching System and Cache

This procedure creates a local Oracle Event Processing cache that is advertised.

1. In Oracle JDeveloper, open the EPN for your application.

2. From the **Components** window, drag the **Local Cache System** component to an empty area on the EPN.

   The Local Cache System - Step 1 of 5 dialog displays.

3. In the **Local Cache System - Step 1 of 5** dialog, provide the following values:

   Cache System ID: A unique ID to identify this local cache system. File name: The name of the configuration file. The default is `processor.xml`. You might want to name this file `cache.xml` or something similar.

4. Click **Next**.

   The Local Cache System - Step 2 of 5 dialog displays.

5. In the **Local Cache System - Step 2 of 5** dialog, use the Add (**+**) or **Choose** button to specify a class that implements the `com.bea.wlevs.cache.spi.CachingSystem` interface.

6. In the **Local Cache System - Step 2 of 5** dialog, select the **Advertise** check box.

   Selecting the Advertise check box means that the caching system allows other applications to access this cache system.

7. Click **Next**.

   The Local Cache System - Step 3 of 5 dialog displays.

8. In the **Local Cache System - Step 3 of 5** dialog, provide the following values:

   Cache name: The name of the first cache in your Oracle Coherent caching system. Value Type: The event type into which you want to load the database values.

9. Click **Next**.

   The Local Cache System - Step 4 of 5 dialog displays

10. In the **Local Cache System - Step 4 of 5** dialog, accept the defaults or provide the values you want.

11. Click **Finish**.

Example 3-7   Assembly File

The assembly file has the values you specified when you created the local caching system.

```
<wlevs:cache id="localcache" value-type="HelloWorldEvent">
  <wlevs:caching-system ref="caching-system"/>
    </wlevs:cache>
  <wlevs:caching-system id="caching-system" class="hellowworld.MyClass"
advertise="false"/>
```

Example 3-8   Configuration File

The configuration file has the values you specified when you created the cache.

```
<caching-system>
  <name>caching-system</name>
  <cache>
    <name>localcache</name>
    <max-size>64</max-size>
    <eviction-policy>LFU</eviction-policy>
  </cache>
</caching-system>
```

3.16 Debug Java Classes

You can debug the Java classes in your Oracle Event Processing application on a local or remote Oracle Event Processing server.

3.16.1 Debug on a Local Oracle Event Processing Server

- Create a Server Connection

- Use the LocalCon1 Connection for the Project

- Start the Server and Run the LoclCon1 in Debug Mode

- Set Breakpoints

- Deploy the Project

- Debug the Java Class

Create a Server Connection

1.   Select **File > New > From Gallery**.

The New Gallery dialog displays.

2.   In the **New Gallery** dialog in the left window, select **Categories > General > Connections**.

3.   In the **New Gallery** dialog in the right window, select **OEP Connection**.

4.   In the Create OEP Server Connection dialog, complete the information.

OEP Server Connection Name: LocalCon1 OEP Server Home Path: /Oracle/ Middleware/my_oep/ Use Default Values: Unchecked. OEP Server Projects Directory: `user_projects/domains/ basicapp_domain/defaultserver`

Use Default Values: Checked Host: 127.0.0.1 Port: 9002 Use Default Values: Unchecked Username: oepadmin User Password: welcome1 Additional Parameters for OEP Server: blank

Use the LocalCon1 Connection for the Project

You can use the LocalCon1 connection on a new project or change the properties on an existing project to use the LocalCon1 connection.

If you just want to see how this works, create a HelloWorld Oracle Event Processing project as follows:

1. In Oracle JDeveloper, select **File > New > Project**.

2. In the New Gallery dialog, select **OEP Project** and click **OK**.

3. In the **Create OEP Project** wizard, provide **HelloWorldProject** for the name, select **OEP Suite** and click **Next**.

4. In the **Configure Java Settings** dialog, click **Next** to accept the defaults.

5. In the **Configure OEP technology settings** dialog in the **OEP Application Template Name** drop-down list, select **HelloWorld**.

6. In the **Configure OEP technology settings** dialog in the **OEP Server Connections** drop-down list, select **LocalCon1**.

7. Click **Finish**.

To change the connection on an existing project to LocalCon1:

1. Right-click the project and select **Project Properties** from the context menu.

2. In the **Project Properties** dialog in the left window, select **Deployment**.

3. In the **Deployment** window, leave the **User Project Settings** radio button selected, and under **Deployment Profiles**, select the profile you want to edit.

4. Click **Edit**.

5. In the **Deployment Properties** panel in the **Connection to OEP Server** drop-down list, select **LocalCon1**.

6. Click **OK**.

Start the Server and Run the LoclCon1 in Debug Mode

1. Start the Oracle Event Processing server with the -debug option.

   a. Go to /Oracle/Middleware/my_oep/user_projects/domains/ <domain>/defaultserver.

   b. Execute the appropriate startup script:

   Windows:

   ```
   startwlevs.cmd -debug
   ```

   UNIX:

   ```
   ./startwlevs.sh -debug
   ```

2. Right-click the project and select **Project Properties** from the context menu.

   The following messages display in the Messages - Log window:

   ```
   Listening for transport dt_socket at address 8453
   8453 is the default port.
   ```

3. In the **Project Properties** dialog in the left window, select **Run/Debug**.

4. In the right panel under **Run/Debug**, accept the default settings and click **Edit**.

   You can first click New to create a new Run Configuration if you want to.

5. Select **Launch Settings** in the left window, and in the right window, select the **Remote Debugging** check box.

6. Select **Tool Settings > Debugger > Remote** in the left window, and in the right panel set the host and port parameters.

   In this example, the host is LocalHost and the port is 8453.

7. Click **OK** and click **OK** again to dismiss the dialogs.

Set Breakpoints

1. To set breakpoints, open any Java class in the project.

   In the HellowWorld project, you could open the source code file for `HelloWorldBean.java`.

2. Select a method, and press **F5** to toggle a breakpoint to on.

   In the `HelloWorldBean. java` source code select the `onInsertEvent` method.

Deploy the Project

1. Right-click the project and select **Deploy > New Deployment Profile** from the context menu.

2. In the **Create Deployment Profile** dialog in the **Profile Type** drop-down list, select **OEP Project Deployment Profile**.

3. In the **Create Deployment Profile** dialog in the **Deployment Profile Name** field, provide a unique name for the profile.

   For the HellowWorld project, the profile name can be HelloWorldProfile.

4. Click **OK**.

5. In the **Deployment Properties** dialog, check that the information is correct.

   Make any corrections that are needed.

6. Click **OK**.

Debug the Java Class

1. Select the project you want to debug, and **Shift + F9**.

   The Attach to JPDA Debugger dialog displays.

   You can also select the Debug button on the tool bar (red ladybug icon).

2. In the **Attach to JPDA Debugger** dialog, check that the information is correct.

3. Click **OK**.

   The Debugging <Project-Name> - Log panel prints messages that show that the debugger is connected to the server.

3.16.2 Remote Oracle Event Processing Server

Debugging remote standalone OEP is similar to Debug on a Local Oracle Event Processing Server except that you have to run Oracle Event Processing server in debug mode manually with the `-debug` flag on the remote host. When you define a connection to the debugger in Oracle JDeveloper, provide the address of the remote host.

3.16.3 Oracle WebLogic Server

Debugging OEP on WLS looks similar to Debug on a Local Oracle Event Processing Server except that you have to start WLS in debug mode manually and check the debugging port.

3.17 Testing with the Event Inspector Service

You configure the Event Inspector service with a local or remote HTTP publish-subscribe server in a component configuration file. You configure the Event Inspector HTTP publish-subscribe server in a component configuration file. When there is only one HTTP publish-subscribe server defined in the server, and you do not specify a local or remote HTTP publish-subscribe server, the Event Inspector service uses the local HTTP publish-subscribe server by default.

Local HTTP Publish-Subscribe Server

1. Open the EPN editor in the Oracle Event Processing IDE.

2. Right-click any component with a configuration file associated with it and select **Go to Configuration Source**.

3. Add the `event-inspector` name element as the following example shows.

```
<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>
```

4. Save and close the file.

Remote HTTP pub-sub Server

You configure the Event Inspector service with a remote HTTP pub-sub server in a component configuration file. Alternatively, you can configure a local HTTP pub-sub server.

1. Open the EPN editor in the Oracle Event Processing IDE.

2. Right-click any component with a configuration file associated with it and select **Go to Configuration Source**.

3. Add an `event-inspector` element as the following example shows.

    ```
    <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>
    ```

4. Save and close the file.

3.18 Start and Stop Oracle JDeveloper and Servers

You can start and stop Oracle WebLogic Server from within Oracle JDeveloper. You can start and stop Oracle Event Processing from the command line or from within Oracle JDeveloper.

Start Oracle JDeveloper

1. Go to `/Oracle/Middleware/soa/jdeveloper/jdev/bin`.

2. Type `./jdev`.

   The Select Role dialog displays.

3. In the **Select Role** dialog, select **Studio Developer (All Features)** and click **OK**.

   Wait a few moments while Oracle JDeveloper starts.

Stop Oracle JDeveloper

1. Save all of your work.

2. Select **File > Exit**.

Start Oracle Event Processing

1. Go to `/Oracle/Middleware/my_oep/user_projects/domains/<domain>/defaultserver`.

2. Execute the appropriate startup script:

   a. Windows:

   - `startwlevs.cmd`

   b. UNIX:

   - `./startwlevs.sh`

   The parameter `–Dprofile-<xxx>` where `<xxx>` is one of the available profiles (set of loaded bundle features specified in `<MW_HOME>/oep/features/bundleloader_profileName.xml`) can be used with the script `startwlevs` script to start the server.

   The terminal panel displays messages as the server starts. When you see, `<The application context for "com.bea.wlevs.dataservices" was started successfully >`, the Oracle Event Processing server is ready.

Alternately, within Oracle JDeveloper you can start Oracle Event Processing after you have defined a connection to an Oracle Event Processing server. Then, you can find a way to start the Oracle Event Processing server under Application Resources when you expand the `Connections` folder.

Stop Oracle Event Processing

1. Go to `/Oracle/Middleware/my_oep/user_projects/domains/<domain>/defaultserver`.

2. Execute the appropriate stop script:

    a. Windows:

       • `stopwlevs.cmd`

    b. UNIX:

       • `./stopwlevs.sh`

The terminal panel displays messages as the server starts. When you see, `<The application context for "com.bea.wlevs.dataservices" was started successfully >`, the Oracle Event Processing server is ready.

4

Create a Basic Application

This chapter walks through building a basic Oracle Event Processing application. The steps include explanations of key Oracle Event Processing application programming concepts.

This chapter covers the following topics:

- About the Basic Application
- Before You Begin
- Create the Application
- TradeReport Project Files
- Create an Event Type to Carry Event Data
- Add the csvgen Adapter to Receive Simulated Event Data
- Add an Output Channel to Convey Events
- Create a Listener Event Sink to Receive and Report Events
- Add an Oracle CQL Processor to Filter Events
- Add an Output Channel
- Deploy
- Set Up and Start the Load Generator
- Stop the Load Generator and the Server.

4.1 About the Basic Application

The basic Oracle Event Processing application models a simple stock trade alert system. The application receives example data about stock trades, monitors the data for certain characteristics, and based on the results, prints some of the data to the console. The following illustration shows the finished event processing network (EPN) diagram for the application:
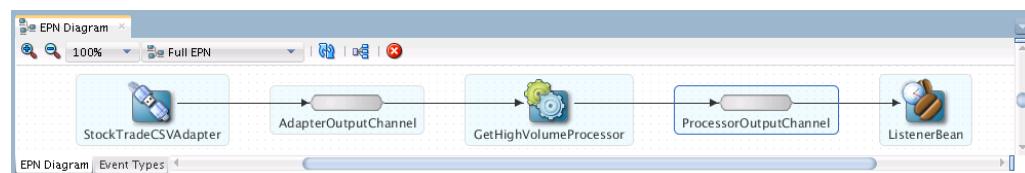
4.2 Before You Begin

This walkthrough requires that you have downloaded and installed the 12c version of Oracle Event Processing including Oracle JDeveloper and the Oracle Event Processing JDeveloper plug-in. Follow the installation instructions that come with the download to ensure you have the correct setup.

Make sure you set the `JAVA_HOME` variable to point to JDK7_u55 or above and set the `PATH` variable to point to the `bin` directory under your JDK installation:

```
export JAVA_HOME=<path to installation directory>
export PATH=${JAVA_HOME}/bin:${PATH}
```

In this walkthrough, the installation directory is `/Oracle/Middleware/my_oep/`.

Optionally, you can set the `WLEVS_HOME` variable to point to the installation directory. The Oracle Event Processing JDeveloper plug-in uses this variable to detect the local Oracle Event Processing server.

> **Note:**
>
> This walkthrough introduces features specific to Oracle Event Processing and assumes that you are familiar with basic Java programming.

4.3 Create the Application

In Oracle JDeveloper an application is the highest level in the control structure. An application is a view of all the objects you need while you work. An application keeps track of all your projects while you develop programs.A project is a logical container for a set of files that define an Oracle JDeveloper program or portion of a program. A project can contain files that represent different tiers of a multi-tier application or different subsystems of a complex application. Project files can reside in any directory and still be contained within a single project.

Start Oracle JDeveloper

1. Go to `/Oracle/Middleware/my_oep/jdeveloper/jdev/bin`.

2. Type `./jdev`.

   The Select Role dialog displays.

3. In the **Select Role** dialog, select **Studio Developer (All Features)** and click **OK**.

   Wait a few moments while Oracle JDeveloper starts.

Create the TradeReport Application

1. In Oracle JDeveloper, Click the **New Application** button.

   The New Gallery dialog displays.

2. In the **New Gallery** dialog, select **OEP Application** and click **OK**.

   The Create OEP Application screen displays.

3. In the **Create OEP Application Step 1 of 4** dialog, enter the following values:

   Application Name: TradeReport Directory: Accept the default Application Package Prefix: Leave blank

4. Click **Next.**

   The Create OEP Application - Step 2 of 4 screen displays.

5. In the **Create OEP Application - Step 2 of 4**, dialog, enter the following values:

   Project Name: TradeReport Directory: Accept the default Project Features: OEP Suite

6. Click **Next**.

   The Create OEP Application - Step 3 of 4 dialog displays.

7. In the **Create OEP Application - Step 3 of 4** dialog, click `Next` to accept the defaults:

   The Create OEP Application - Step 4 dialog displays.

8. In the **Create OEP Application - Step 4 of 4** dialog, examine the default values:

   Empty OEP Project: Provides the basic structure of an Oracle Event Processing application.

   OEP Server Connections: Leave blank. In a later step, you create the Oracle Event Processing server connection.

9. Click **Finish** to accept the defaults.

   The Oracle Event Processing TradeReport application and project displays.

4.4 TradeReport Project Files

The TradeReport application contains the Projects and Applications Resources windows. The Projects window lists the TradeReport project. The TradeReport project contains an `OEP Content` folder with the `spring` and `wlevs` subfolders. On the right side of Oracle JDeveloper under IDE Connections is the Resources window.

Projects Window

- **spring** subfolder that contains the `TradeReport.context.xml` assembly file. The assembly file conforms to the Spring framework and contains the contents and structure of the TradeReport EPN.

  The assembly file also contains the default configuration for each EPN stage. This default configuration cannot be changed at run time without redeploying the application. As you add and connect stages on the EPN diagram, Oracle JDeveloper captures your work in this file. You can also edit this file manually.

  Note:

  The EPN assembly file XML schema extends the Spring framework configuration file. See the Spring website at `http://www.springsource.org/spring-framework`.

- **wlevs** subfolder that contains the default `processor.xml` configuration file. The files in the `wlevs` folder describe components with configurations that can be edited at runtime with Oracle Event Processing Visualizer. As you use Oracle JDeveloper to create components, you can place their configurations in the `processor.xml` file or specify another component configuration file to group component types in the same file. You can also edit configuration files manually.

- **EPN diagram** The EPN diagram represents the components that make up the application. Event data enters your application from the left of the diagram, and moves to the right from stage to stage.

  The EPN diagram shows a graphical representation of the underlying EPN configuration. When you add a component to the EPN, Oracle JDeveloper writes information to the `TradeReport.context.xml` assembly file and the configuration file.

- **MANIFEST.MF** that describes the contents of the OSGi bundle that you deploy to the Oracle Event Processing server.

Resources Window

The Resources window, which is on the right side of Oracle JDeveloper under IDE Connections, provides information about running server connections.

4.5 Create an Event Type to Carry Event Data

Within an Oracle Event Processing application, every event has an event type. The event type is a structure that defines a particular kind of event data in terms of the set of values the event can take, and the operations that can be performed on that data.

Oracle Event Processing supports several data structures for creating a new event type. These data structures are `JavaBean` classes, tuples, and `java.util.Map` classes. A `JavaBean` class is the best practice structure for new event types and is used in this walkthrough to define trade events.

As raw event data comes into the Oracle Event Processing application, the application binds that data to an event of a particular event type. You define the event type in terms of the set of data it can hold and the required type for each data in the set.

In this walkthrough, the event data comes into the application from a CSV file in consistent rows of comma-separated values as follows:

```
IBM,15.5,3.333333333,3000,15 SUN,10.8,-1.818181818,5000,11
ORCL,14.1,0.714285714,6000,14
GOOG,30,-6.25,4000,32
YHOO,7.8,-2.5,1000,8
```

The data columns are not labeled in the CSV file, but if they were labeled, they would have the corresponding Java data type shown in Table 4-1. The Java data types that define an event type are referred to as properties in Oracle Event Processing.

Table 4-1   Mapping Event Data to Event Types

Possible Columns	Java Data Type
Stock Symbol	String
Price per share	Double

Table 4-1   (Cont.) Mapping Event Data to Event Types

Possible Columns	Java Data Type
Percent change	Double
Volume of shares transacted	Integer
Last price	Double

Create the TradeEvent JavaBean

1. Select the **TradeReport** project.

    Oracle JDeveloper highlights the TradeReport project.

2. Select **File > New > From Gallery**.

    The New Gallery dialog displays.

3. In the **New Gallery** dialog, select **General** in the left panel, **Java Class** in the right panel, and click **OK**.

    The Create Java Class dialog displays.

4. In the **Create Java Class** dialog, enter **TradeEvent** in the Name field, and enter or review the following default settings:

    Name: TradeEvent Package: tradereport Extends: java.lang.Object Access Modifiers: public Other Modifiers: <None> Constructors from Superclass: checked Implement Abstract Methods: checked

5. Click **OK**.

    Oracle JDeveloper adds the `tradereport.TradeEvent` JavaBean class to the Project under the Application Sources folder. The stub code displays in the Oracle JDeveloper center window in its own tab:

    ```
    package tradereport;

    public class TradeEvent {
       public TradeEvent() {
          super ();
       }
    }
    ```

Create Private Variables and Accessor Methods

1. In the **TradeEvent** class, add private variables for each of the properties (Java data types) as shown in the following example.

    ```
    package tradereport;

    public class TradeEvent {
      // One variable for each field in the event data.
      private String symbol;
      private Double price;
      private Double lastPrice;
      private Double percChange;
      private Integer volume;
    ```

```
public TradeEvent () {
super();
}
}
```

2. To generate the accessor methods, right click anywhere in the source editor.

   The source editor pop-up menu displays.

3. In the source editor pop-up menu, select **Generate Accessors**.

   The Generate Accessors dialog displays.

4. In the **Generate Accessors** dialog, click the **Select All** button and click **OK**.

5. Close the **TradeEvent.java** tab and save the file.

Configure the TradeEvent Event Type

1. In the **TradeReport** project under the **OEP Content** folder, double-click the EPN Diagram.

   The EPN diagram displays in the center window and is empty.

2. Below the EPN diagram select the **Event Types** tab.

   The Event Type Definitions window displays the `TradeReport.context.xml` folder with Add (**+**) and Delete (**x**) buttons at the top.

3. In the Event Type Definitions window, select the **TradeReport.context.xml** folder and click **Add.**

   Controls to define the event type display below **Event Type Details** on the right.

4. In the **Type Name** field enter **TradeEvent**.

   The event type name does not have to be similar to the JavaBean class name, but by making them similar, it is easier to track which event types go with which classes.

5. Select the **Properties defined in Java bean** radio button and enter or use search to the name of the JavaBean in the **Class** box.

   The name of the JavaBean is **tradereport.TradeEvent**.

   Note:

   The **Properties defined declaratively** radio button is for defining events as tuples.

   ---

6. Close the **EPN diagram** editor and save the file.

View the EPN Assembly File

1. In the left panel under T**radeReport > OEP Content > Spring**, double-click **TradeReport.context.xml**.

   The TradeReport.context.xml file displays in the Source tab.

2. In the **TradeReport.context.xml** file at the bottom of the file, look for the following lines:

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

Notice that Oracle Event Processing manages event types in an event type repository, and that the `TradeEvent` event type contains (maps to) the `tradereport.TradeEvent` class.

4.6 Add the csvgen Adapter to Receive Simulated Event Data

Adapters manage data flowing into and out of the EPN. This example uses a csvgen adapter that works with the load generator utility to simulate a data feed to test your application. 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. The csvgen adapter logic translates data read from the CSV file into an event that has the `TradeEvent` event type.

> **Note:**
>
> Before you deploy an application to the final production environment, you must switch to an input adapter that can read the type of incoming data your application will receive in production.
>
> See *Developing Applications for Oracle Event Processing* for information about the available input and output adapters.

In this procedure, you declare the adapter and set its properties. When completed, the EPN diagram displays the adapter to create the first stage in your TradeReport EPN.

Create the csvgen Adapter and Set Its Properties

1. Open the **TradeReport > META-INF > spring > TradeReport.context.xml** assembly file.

2. Below the event-type-repository XML stanza, add the following XML to declare the csvgen adapter.

```
<wlevs:adapter id="StockTradeCSVAdapter" provider="csvgen">
 <wlevs:instance-property name="port" value ="9200" />
 <wlevs:instance-property name="EventTypeName" value="TradeEvent" />
 <wlevs:instance-property name="eventPropertyNames"
value="symbol,price,percChange,volume,lastPrice" />
</wlevs:adapter>
```

> **Note:**
>
> No white spaces allowed between the `instance-property` name values. The order of the `name` values must match the order in the `StockData.csv` files described in View the Test Data.

The XML stanza declares an instance of the csvgen adapter and assigns to it three properties that configure it for use in your EPN. The adapter uses the properties to map from incoming raw event data to the properties of the event type you defined.

id: A unique identifier for the adapter. The provider attribute value must be csvgen to refer to the csvgen implementation included with Oracle Event Processing.

port: Tells the adapter instance what port to listen on for incoming event data. The value here, 9200, corresponds to the port number to which the load generator will send event data (more on that later).

The eventTypeName: Tells the instance the name of the event type to which incoming event data should be assigned. Here, you give the name of the TradeEvent type you defined earlier.

eventPropertyNames. Tells the instance the names of the event type properties to which data should be assigned. Notice in this case that the eventPropertyNames: A comma-separated list of the same properties you defined in the JavaBean for the event type. In order for the csvgen adapter to map from incoming values to event type properties, the names here must be the same as your event type and must be in the same order as corresponding values for each row of the CSV file.

3. Save and close the **TradeReport.context.xml** assembly file.

   The **StockTradeCSVAdapter** displays on the EPN diagram to create the first stage in your TradeReport EPN network.

4. Open the EPN diagram to see the StockTradeCSVAdapter:

4.7 Add an Output Channel to Convey Events

A channel is a conduit that uses logic to transfer events from one stage in the EPN to the next stage. In this step, you add a channel to carry newly generated events from the StockTradeCSVAdapter to the next stage.

Create the AdapterOutputChannel

1. With the EPN Diagram open, go to **Basic Components** and drag the **Channel** component to an empty space on the EPN diagram.

   The New Channel dialog displays.

2. In **New Channel dialog**, enter the following values:

   Channel ID: AdapterOutputChannel. Event Type: TradeEvent.

3. Click **OK**.

   The AdapterOutputChannel component displays on the EPN diagram.

   The AdapterOutputChannel conveys events of type TradeEvent to the next stage in the EPN diagram. Recall that the TradeEvent event type is implemented with the TradeEvent JavaBean class.

Connect the Adapter to the Channel

1. Click the **StockTradeCSVAdapter** icon and drag it to the **AdapterOutputChannel** icon.

   This action creates a connecting line between the two icons and places the AdapterOutputChannel to the right of the StockTradeCSVAdapter, which indicates that events flow from the adapter to the channel.
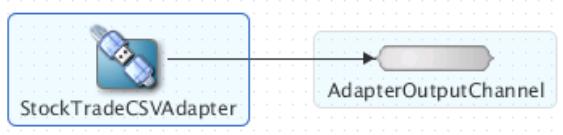
2. Double-click the **AdapterOutputChannel** icon to view the **TradeReport.context.xml** assembly file. A blinking cursor displays next to the line with the channel configuration.

   ```
   <wlevs:adapter id="StockTradeCSVAdapter" provider="csvgen">
    <wlevs:listener ref="AdapterOutputChannel"/>
    <wlevs:instance-property name="eventType" value="TradeEvent"/>
    <wlevs:instance-property name="eventPropertyNames" value="symbol, price,
      lastPrice, percChange, volume" />
   </wlevs:adapter>
   <wlevs:channel id="AdapterOutputChannel" event-type="TradeEvent"/>
   ```

   When you created the connection between the adapter and the channel, Oracle JDeveloper added a reference to a listener. The listener `ref` attribute is set to the `id` attribute of the channel element meaning that the channel listens for events that come from the adapter.

3. Close the **TradeReport.context.xml** tab and save the file.

4.8 Create a Listener Event Sink to Receive and Report Events

Next you add a listener event sink that receives trade events from the channel and checks the information in those events. A listener event sink is a Java class that implements logic to listen for and work on trade events. This type of Java class is also called a listener Java class.

The following procedure shows you how to create a listener event sink that listens for trade events, gets the stock symbol and trade volume information, and prints the stock symbol and trade volume information to the console.

Create the Listener Event Sink

1. Select the **TradeReport** project and select **File > New > Java Class**.

2. In the **Create Java Class** dialog, enter **TradeListener** in the Name field and review the following settings:

   Name: TradeListener Package: tradereport Extends: java.lang.Object Access Modifiers: public Other Modifiers: <NONE> Constructors from Superclass: Checked Implement Abstract Methods: Checked

3. In the **Create Java Class** dialog under the **Implements** area, click the **Add** (+) button to select the interface your listener needs to implement to be an event sink.

4. In the **Class Browser** dialog, use either the Search tab or the Hierarchy tab to locate the **com.bea.wlevs.ede.api.StreamSink** class.

5. Under **Matching Class**, highlight the com.bea.wlevs.ede.api.StreamSink class and click **OK**.

   You return to the Create Java Class dialog.

6. In the **Create Java Class** dialog, click **OK**.

   Oracle JDeveloper adds the TradeListener JavaBean to the project under the Application Sources folder. The stub code displays in the Oracle JDeveloper middle panel.

   ```java
   package tradereport;

   import com.bea.wlevs.ede.api.EventRejectedException;
   import com.bea.wlevs.ede.api.StreamSink;

   public class TradeListener implements StreamSink {
       public TradeListener() {
           super();
       }
       @Override
       public void onInsertEvent(Object object) throws EventRejectedException {
           // TODO Implement this method
       }
   }
   ```

7. In the `TradeListener` class, edit the `onInsertEvent` method as follows:

   ```java
   @Override
   public void onInsertEvent(Object event) throws EventRejectedException {

       if (event instanceof TradeEvent){
           String symbolProp = ((TradeEvent) event).getSymbol();
           Integer volumeProp = ((TradeEvent) event).getVolume();
           System.out.println(symbolProp + ":" + volumeProp);
       }
   }
   ```

   The `onInsertEvent` method listens for trade events, and when it hears a `TradeEvent`, it calls the `tradereport.TradeEvent get` methods to get the stock symbol and the trade volume, and to print the stock symbol and trade volume information to the console.

8. Close the **TradeListener.java** tab and save the file.

Add the Event Sink to the EPN Diagram as an Event Bean

1. Open the EPN Diagram.

2. Under **Base Components**, drag the **Event Bean** component onto an empty area of the EPN Diagram.

   The New Event Bean wizard displays.

3. In the **EventBean ID** field, enter **ListenerBean**.

4. In the **EventBean class** field, enter `tradereport.TradeListener` and click **OK**

5. In the **EPN diagram**, select **AdapterOutputChannel** and drag it to **ListenerBean** to connect them.

   The connection enables trade events to pass from the channel to the listener bean.
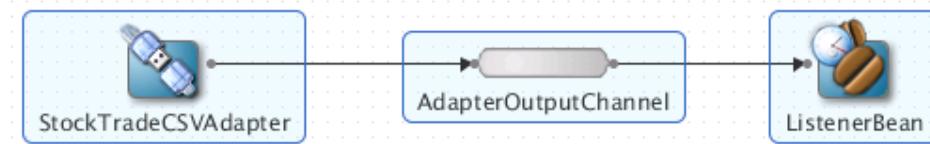
6. Double-click the **AdapterOutputChannel**.

   The `TradeReport.context.xml` file displays with a blinking cursor next to the channel line. The `ref` attribute of the channel listener points to `ListenerBean`.

   ```
   <wlevs:channel id="AdapterOutputChannel" event-type="TradeEvent">
     <wlevs:listener ref="ListenerBean" />
   </wlevs:channel>
   <wlevs:event-bean id="ListenerBean"
     class="tradereport.TradeEvent" />
   ```

7. Close the **TradeReport.context.xml** tab and save the file.

   > **Note:**
   >
   > There are no configuration file entries for the channel beyond the default configuration. You can edit the `processor.xml` file to customize the channel configuration or create a separate configuration file, such as `channel.xml`, for channels and add custom channel configuration to it. See Add Configuration Settings to a Component.

4.9 Add an Oracle CQL Processor to Filter Events

Next add an Oracle CQL processor to filter events based on certain criteria. The Oracle CQL processor goes between AdapterOutputChannel and an output channel that you create in the next section.

The Oracle CQL processor contains Oracle CQL code that you write. The Oracle CQL code queries the events sent to the processor from AdapterOutputChannel. The query retrieves only those trade events that have a volume that is greater than 4000. Oracle Event Processing passes the retrieved events to the output channel, which then sends the events to ListenerBean for processing. Recall that ListenerBean listens for trade events, gets the stock symbol and trade volume information, and prints the stock symbol and trade volume information to the console.

The CQL query selects the symbol and volume properties from each incoming `trade` event, tests the volume property for a value higher than 4000, and outputs a set of 1 qualifying event at a time. The `NOW` operator creates a window of time that contains the event that happened at the last tick of the system.

```
<query id="GetHighVolume"><![CDATA[
   select trade.symbol, trade.volume
   from AdapterOutputChannel [now] as trade
   where trade.volume > 4000
   ]]>
</query>
```

Add a GetHighVolume Processor and Query

1. In the Components window under **Basic Components**, drag the **Processor** component to an empty space on the EPN diagram.

    The New Processor dialog displays.

2. In the New Processor dialog in the **Processor ID** field, enter **GetHighVolumeProcessor**, keep the default File name, which is `processor.xml`, and click **OK**.

    Oracle Event Processing requires that you have at least one configuration file with the name `processor.xml` that contains the processor configuration. You can add other component configurations to this file or create additional configuration files.

3. Right-click the connector from the **AdapterOutputChannel** icon to the **ListenerBean** icon and click **Delete**.

4. Click the **AdapterOutputChannel** component, and drag from it to the **GetHighVolumeProcessor** icon.

    Creating this connection makes the Oracle CQL processor aware of the channel. After you connect the channel to the Oracle CQL processor, you can refer to the channel by its ID value in the Oracle CQL code.
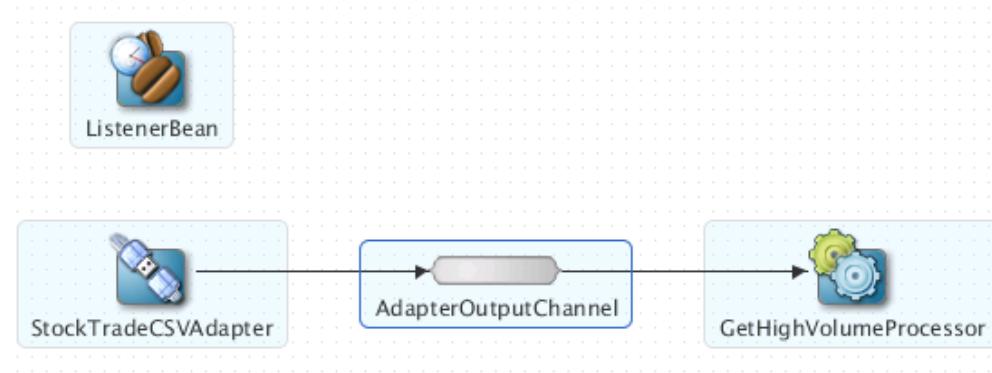
5. Right-click the **GetHighVolumneProcessor** stage.

    The context menu displays.

6. From the context menu, select **Go To Configuration Source**.

    Oracle JDeveloper opens a source editor where you place the Oracle CQL rules to be applied to the streaming event data. The source editor provides a sample query that you can edit or replace.

7. Replace the sample Oracle CQL code with the Oracle CQL code provided here:

    You replace the sample Oracle CQL between `<rules>` `</rules>` with the following Oracle CQL code:

```
<query id="GetHighVolume"><![CDATA[
    select trade.symbol, trade.volume
    from AdapterOutputChannel [now] as trade
    where trade.volume > 4000
    ]]>
</query>
```

8. Close the configuration file tab and save your work.

4.10 Add an Output Channel

1. From **Base Components**, drag the **Channel** component to an empty area on the EPN diagram.

2. In the **New Channel** wizard in the **Channel ID field**, enter **ProcessorOutputChannel** and select **TradeEvent** as the event type.

3. Click **OK**.

4. Select the **GetHighVolumeProcessor** component and drag it to the new channel component to connect the Oracle CQL processor and channel.
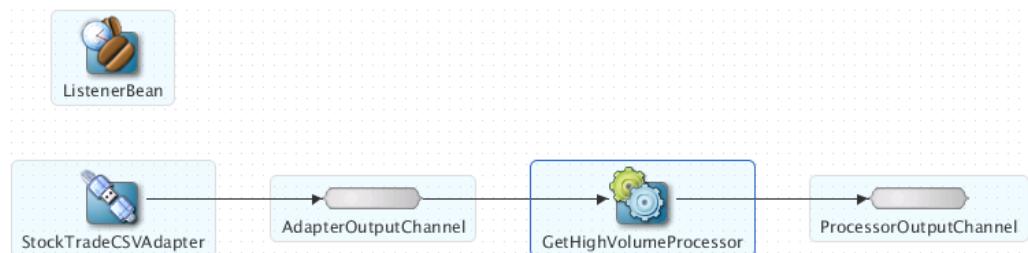
5. Select the **ProcessorOutputChannel** component and drag it to the **ListenerBean** component to connect the channel to the listener.

All of the components in the EPN diagram are now connected.

6. Double-click the **ProcessorOutputChannel** icon to see the channel configuration in the **TradeReport.context.xml** file.

The entry for the ProcessorOutputChannel specifies that events of type TradeEvent pass through this channel.

```
<wlevs:channel id="ProcessorOutputChannel" event-type="TradeEvent">
   <wlevs:listener ref="ListenerBean"/>
   <wlevs:source ref="GetHighVolumeProcessor"/>
 </wlevs:channel>
```

7. Save all of the files in the project.

4.11 Deploy

To deploy the example application for testing, perform the following actions:

- Create an Oracle Event Processing Domain
- Start the Oracle Event Processing Server
- Create an Oracle Event Processing Server Connection

- Create a Deployment Profile

- Deploy the Application

Create an Oracle Event Processing Domain

To create a domain, start the Oracle Event Processing Configuration wizard:

1. Start the Configuration Wizard:

    a. On Windows, navigate to \Oracle\Middleware\my_oep\oep\common\bin \ and type config.cmd.

    b. On UNIX, navigate to /Oracle/Middleware/my_oep/oep/common/bin and type ./config.sh.

    The Configuration wizard Welcome screen displays.

2. On the **Welcome** screen, click **Next**.

    The Choose Create or Update Domain screen displays.

3. On the **Choose Create or Update Domain** screen, select **Create a new OEP domain** and click **Next**.

    The Create or Update Domain screen displays.

4. In the Create or Update Domain screen, in the **User Name** field, enter **oepadmin**, and enter and confirm the password, **welcome1**.

5. Click **Next**, accept the **Configure Server** defaults, and click **Next**.

    The Configure Domain Identity Keystore screen displays.

6. In the **Configure Domain Identity Keystore** screen, enter and confirm the password **welcome1** and click **Next**.

    The Configuration Options screen displays.

7. In the **Configuration Options** screen, click **Next** to not perform any optional configuration.

    The Create OEP Domain screen displays.

8. In the **Create OEP Domain** screen, enter **basicapp_domain** and make a note of its location.

    The location will be something like /Oracle/Middleware/my_oep/ user_projects/domains.

9. Click **Create**, and after a few moments, click **Done**.

Start the Oracle Event Processing Server

1. Go to /Oracle/Middleware/my_oep/user_projects/domains/ basicapp_domain/defaultserver.

2. Execute the appropriate startup script:

    a. On Windows:

    - prompt> startwlevs.cmd

b. On UNIX:

- `prompt> ./startwlevs.sh`

The terminal window displays messages as the server starts. When you see, `<The application context for "com.bea.wlevs.dataservices" was started successfully >`, the Oracle Event Processing server is ready.

Create an Oracle Event Processing Server Connection

1. Select **File > New > From Gallery**.

   The New Gallery dialog displays.

2. In the **New Gallery** dialog under **Categories > General**, select **Connections**.

3. In the New Gallery dialog under **Items**, select **OEP Connection**, and click **OK**.

   The Create OEP Server Connection dialog displays.

4. In the **Create OEP Server Connection** dialog, provide the following information:

   Connection will be created in: IDE Connections: Selected Remote OEP Server: Not checked OEP Server Connection Name: OEPBasicAppConnection OEP Server Home Path: /Oracle/Middleware/my_oep/ Use Default Values: Unchecked. OEP Server Projects Directory: user_projects/domains/ basicapp_domain/ defaultserver Use Default Values: Checked Host: 127.0.0.1 Port: 9002 Use Default Values: Unchecked Username: oepadmin User Password: welcome1 Additional Parameters for OEP Server: blank

5. In the **Create OEP Server Connection** dialog, click **Test Connection**.

   If you see **Success** in the area below the **Test Connection** button, you entered the information correctly. If you see errors, correct them and test again until you see **Success**.

6. When you see the **Success** message, click **OK**.

Create a Deployment Profile

1. Right-click the **TradeReport** project and select **Deploy > New Deployment Profile**.

   The Create Deployment Profile dialog displays.

2. In the **Create Deployment Profile** dialog, provide the following values.

   Profile Type: OEP Project Deployment Profile.

   > **Note:**
   >
   > Make sure you select the correct Profile Type, which is OEP Project Deployment Profile.

   Deployment Profile Name: basicapp_profile.

3. Click **OK**.

   The Deployment Properties dialog displays.

4. In the **Deployment Properties** dialog, verify the information:

Connection to Local OEP Server: OEPBasicAppConnection (127.0.0.1:9002) Symbolic Name: TradeReport.TradeReport Bundle Name: TradeReport.TradeReport Bundle Version: 1.0.0 OSGi JAR file: /home/ <username>/jdeveloper/mywork/TradeReport/TradeReport/deploy/ basicapp_profile.jar.

5. Click **OK**.

Deploy the Application

1. Right-click the **TradeReport** project.

   The context menu displays.

2. In the **context menu**, select **Deploy > basicapp_profile.**

   The Deployment Action dialog displays.

3. In the **Deployment Action** dialog, select **Deploy OSGi bundle to target platform**.

4. Click **Next**.

   The Summary dialog displays.

5. In the **Summary** dialog, confirm the information.

6. Click **Finish**.

   In the **Deployment - Log** panel at the bottom of the middle panel, messages indicate the successful deployment.

7. In the Resources window on the right side under IDE Connections, navigate to **OEP Server > OEPBasicAppConnection > Applications**.

   The BasicApplication.BasicApp[Running} connection displays.

A deployment profile creates an OSGi bundle that contains the required library JAR file.

4.12 Set Up and Start the Load Generator

The load generator enables you to load test data so that you can see how your Oracle Event Processing application behaves when it is deployed into production.

Normally, you start the load generator after you deploy the application. However, you can start the load generator before you deploy, but you will get a message that there is no listener on port 9200. The message goes away after you deploy application.

View the Test Data

1. In the text editor of your choice, open the **StockData.csv** file included with Oracle Event Processing installation.

   By default, the file is at the following location:

   ```
   /Oracle/Middleware/my_oep/oep/utils/load-generator/
   StockData.csv.
   ```

2. Take a look at the **StockData.csv** file, which contains comma-separated values in rows where each row represents a trade.

> **Note:**
>
> The order of the event properties in the `StockData.csv` file must match the order of event properties specified in Create the csvgen Adapter and Set Its Properties.

Verify the Load Generator Properties

1. In the text editor of your choice, open the **StockData.prop** files included with Oracle Event Processing installation.

   By default, the files are at the following location:

   `/Oracle/Middleware/my_oep/oep/utils/load-generator/StockData.prop.`

2. In the **StockData.prop** file, verify the following properties:

   - `test.csvDataFile`: The name of the CSV file that the load generator reads. For this example, the value is **StockData.csv**.

   - `test.port`: The port number to which the load generator sends event data. This should be the port value you specified when you configured the CSV adapter, which is **9200**.

   - `test.packetType`: The type of data format that the load generator will handle. For this example, the value is **CSV**.

   The load generator requires the `test.csvDataFile` and `test.port` properties. The other properties are optional, but you need to set at least `test.packetType` so that the load generator knows that your input is in CSV form.

3. Close the **StockData.prop** file and save if you made any changes.

Start the Load Generator

1. Run the load generator with the `StockData.prop` properties file:

   a. On Windows:

      ```
      prompt> runloadgen.cmd StockData.prop
      ```

   b. On UNIX:

      ```
      prompt> ./runloadgen.sh StockData.prop
      ```

4.13 Stop the Load Generator and the Server

When you are finished with the example, you can stop the load generator and the Oracle Event Processing server.

Stop the Load Generator

1. Change directory to **/Oracle/Middleware/my_oep/oep/utils/load-generator**.

2. Type **Ctrl-c**

Stop the Server

1. Change directory to **/Oracle/Middleware/my_oep/user_projects/domains/ basicapp_domain/defaultserver**.

2. Execute the **stopwlevs** command.

5

Create a Fraud Detection Application with EDN Adapters

This chapter walks through the steps to create and deploy a fraud detection application to present two major new features in the Oracle Event Processing 12c release. The first feature is support for the entire Oracle Event Processing application life cycle with Oracle JDeveloper. The second feature is Event Delivery Network (EDN) adapter support so that an Oracle Event Processing application can send events to and receive events from Oracle SOA Suite. You create and deploy the Oracle Event Processing fraud detection application entirely in Oracle JDeveloper.

This chapter covers the following topics:

- Fraud Detection Scenario
- Before You Begin
- Event Delivery Network Walkthrough.

5.1 Fraud Detection Scenario

In this walkthrough, you create an Oracle Event Processing application that implements a real-time analysis of customer orders. An email address uniquely identifies each customer. While order data (event data) passes to the Oracle Event Processing server, an Oracle Event Processing application dynamically accesses the data and checks for potential fraudulent activity. In this example, event patterns with an aggregated dollar amount for orders by the same person that exceeds $1000 in any 24 hour period indicate possible fraudulent activity.

You can use this fraud detection example application as a base for future real-time fraud detection event-based solutions. Once deployed, the Oracle Event Processing EDN application listens for events coming from the Oracle SOA suite event network.

5.2 Before You Begin

This walkthrough assumes that you have Oracle SOA Suite and Oracle Event Processing installed. In this walkthrough, the top-level installation directory is referred to as /Oracle/Middleware/, the Oracle SOA Suite installation directory is referred to as */Oracle/Middleware/my_soa*, and the Oracle Event Processing installation directory is referred to as */Oracle/Middleware/my_oep*.

You should also have your `JAVA_HOME` variable set to point to JDK7_u55 or above, and the `PATH` variable set to point to the `bin` directory under your JDK installation:

```
export JAVA_HOME=<path to installation directory>
export PATH=${JAVA_HOME}/bin:${PATH}
```

> **Note:**
>
> Although this walkthrough introduces features specific to Oracle Event Processing, it assumes that you are familiar with basic Java programming.

5.3 Event Delivery Network Walkthrough

The following list outlines the high-level tasks required to develop and deploy the Oracle Event Processing fraud detection application:

- Start Oracle WebLogic Server
- Copy the Artifacts Folder
- Create an Oracle Event Processing Domain
- Create a Java Message Service Topic.
- Start the Oracle Event Processing Server
- Use Oracle JDeveloper to Create An Oracle Event Processing Application
- Deploy the Application with JDeveloper
- Create and Deploy the Sample SOA Composite
- Test the Fraud Detection Application.

5.3.1 Start Oracle WebLogic Server

To perform the steps in the walkthrough, start Oracle JDeveloper and Oracle WebLogic Server. Oracle JDeveloper and Oracle WebLogic Server are part of Oracle SOA Suite.

You start the Oracle Event Processing server at a later step in the walkthrough.

Start Oracle JDeveloper and Oracle WebLogicServer

1. In your work area, navigate to `/Oracle/Middleware/soa/jdeveloper/jdev/bin`.

2. Start Oracle JDeveloper by typing `./jdev -clean` at the command line.

   The Oracle JDeveloper initial screen displays.

3. In Oracle JDeveloper, select **Run > Start Server Instance**.

4. If a **Create Default Domain** dialog displays, accept the defaults and enter and confirm a domain password that is at least 7 characters long with at least one numeric character. For example, **welcome1**.

   Oracle WebLogic Server prints messages in the message area while it takes a few minutes to come up. The server is up and running when you see the message: `SOA Platform is running and accepting requests` and a red box below the menu bar and next to the search field at the top of the menu area.

5.3.2 Copy the Artifacts Folder

The `OEP_Fraud_Detection_Walkthrough_Files.zip` file provides supporting files that you need for this walkthrough. The folder contains the event definition files (`*.edl` and `*.xsd`), the sample fraud detection Oracle CQL code, and a sample SOA Composite (`EDNOEPv2`). A SOA composite is a SOA application that interfaces between Oracle SOA Suite and the EDN.

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

Get the Artifacts Folder

1. Go to `http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/complex-event-processing-088095.html`.

2. Locate the `OEP_Fraud_Detection_Walkthrough_Files.zip` file and download it to an accessible location on your computer.

3. Unzip the zip file.

5.3.3 Create an Oracle Event Processing Domain

In this step, you use the `config.sh` command to start the Configuration wizard and create a new domain for the Fraud Check application to use.

Create a New Domain

1. Navigate to your Oracle Event Processing installation to the `/Oracle/Middleware/my_oep/oep/common/bin` directory.

2. In the `/Oracle/Middleware/my_oep/oep/common/bin` directory, type `./config.sh` to start the Configuration wizard.

   The Configuration wizard Welcome screen displays.

3. On the **Welcome** screen, click **Next**.

   The Choose Create or Update Domain screen displays.

4. On the **Choose Create or Update Domain** screen, select **Create a new OEP domain** and click **Next**.

   The Create or Update Domain screen displays.

5. In the Create or Update Domain screen, in the **User Name** field, enter **oepadmin**, and enter and confirm the password, **welcome1**.

6. Click **Next**.

   The Configure Server screen displays.

7. In the **Configure Server** screen, click **Next** to accept the defaults.

   The Configure Domain Identity Keystore screen displays.

8. In the **Configure Domain Identity Keystore** screen, enter and confirm the password **welcome1** and click **Next**.

The Configuration Options screen displays.

9. In the **Configuration Options** screen, click **Next** to not perform any optional configuration.

   The Create OEP Domain screen displays.

10. In the **Create OEP Domain** screen, enter **fraudcheck_domain** and make a note of its location.

    The location will be something like `/Oracle/Middleware/my_oep/user_projects/domains`.

11. Click **Create**, and after a few moments, click **Done**.

Add the EDNConnectionFactory to the Domain

1. Navigate to `/Oracle/Middleware/my_oep/user_projects/domains/fraudcheck_domain/defaultserver` and open the `fraudcheck_domain` **startwlevs.sh** file with a text editor.

2. Add the following system properties to the `JAVA_HOME` command line at the bottom of the file.

   The system properties go before the `-jar $USER_INSTALL_DIR` setting.

   ```
   -Dedn.jms.topic="jms/fabric/EDNTopic" -Dedn.jms.connection-factory="jms/fabric/
   EDNConnectionFactory"
   ```

   The final `JAVA_HOME` line in `startwlevs.sh` looks like this: (all on one line):

   ```
   "$JAVA_HOME/bin/java" $JVM_ARGS $JVM_D64 $DEBUG_ARGS
     -Dwlevs.home="$USER_INSTALL_DIR"
     -Dedn.jms.topic="jms/fabric/EDNTopic"
     -Dedn.jms.connection-factory="jms/fabric/EDNConnectionFactory"
     -jar "${USER_INSTALL_DIR}/bin/wlevs.jar" $ARGS
   ```

   The system properties instruct the Oracle Event Processing server to use the JMS implementation for the EDN rather than the default, which is Advanced Queuing (AQ).

5.3.4 Create a Java Message Service Topic

A Java Message Service (JMS) topic is a mechanism for publishing messages to one or more subscribers. Use the Oracle WebLogic Server administration console to create a JMS topic.

Create a JMS Topic

1. In your work area, open a browser.

   The browser displays.

2. In the browser URL box, type **localhost:7101/console** in the URL box.

   The administration console login screen displays.

3. Log in to the administration console with a user name of **weblogic** and the password **welcome1**.

   The WebLogic Server Administration Console screen displays.

4. In the left panel, under **Domain Structure**, expand **Services > Messaging > JMS Modules**.

5. In the right panel under JMS Modules, click **SOAJMSModule**.

   The Settings for SOAJMSModule screen displays.

6. In the right panel, under **Summary of Resources**, click **New**.

   The Create a New JMS System Module Resource screen displays.

7. In the **Create a New JMS System Module Resource** screen, select the **Topic** radio button and click **Next**.

   The Create a New JMS System Module - JMS Destination Properties screen displays.

8. In the JMS Destination Properties in the **Name** field, enter **EDNTopic**, and in the **JNDI Name** field, enter **jms/fabric/EDNTopic**.

9. Click **Next**.

   The next Create a New JMS System Module screen displays.

10. In the Create a New JMS System Module screen, in the **Subdeployments** drop-down list, select **SOASubDeployment** and make sure the SOAJMSServer radio button is checked.

11. Click **Finish**.

   The EDNTopic JMS topic displays in the Summary of Resources table.

   In the following figure, EDNTopic, displays in the third row of the table underneath EDNConnectionFactory. The EDNConnectionFactory is part of the Oracle WebLogic Server installation. Connection factories are objects that enable JMS clients to create concurrent JMS connections. The EDNConnectionFactory object enables the JMS EDNTopic to create an EDN connection to Oracle SOA Suite.

Summary of Resources

	Name	Type	JNDI Name	Subdeployment	Targets
New	Delete			Showing 11 to 20 of 24	Previous | Next
☐	EDNAQjmsLocalTxForeignServer	Foreign Server	N/A	Default Targetting	DefaultServer
☐	EDNConnectionFactory	Connection Factory	jms/fabric/EDNConnectionFactory	Default Targetting	DefaultServer
☐	EDNOEPTopic	Topic	jms/fabric/EDNOEPTopic	SOASubDeployment	SOAJMSServer
☐	EDNQueue	Queue	jms/fabric/EDNQueue	SOASubDeployment	SOAJMSServer
☐	EDNTopic	Topic	jms/fabric/EDNTopic	SOASubDeployment	SOAJMSServer
☐	NotificationSenderQueue	Queue	jms/Queue/NotificationSenderQueue	SOASubDeployment	SOAJMSServer
☐	NotificationSenderQueueConnectionFactory	Connection Factory	jms/Queue/NotificationSenderQueueConnectionFactory	Default Targetting	DefaultServer
☐	TestFwkQueue	Queue	jms/testfwk/TestFwkQueue	SOASubDeployment	SOAJMSServer
☐	TestFwkQueueFactory	Connection Factory	jms/testfwk/TestFwkQueueFactory	Default Targetting	DefaultServer
☐	TransportDispatcherQueue	Queue	jms/b2b/TransportDispatcherQueue	SOASubDeployment	SOAJMSServer
New	Delete			Showing 11 to 20 of 24	Previous | Next

5.3.5 Start the Oracle Event Processing Server

Start the Oracle Event Processing server so that you can create and deploy the Oracle Event Processing Fraud Detection application.

Start the Oracle Event Processing Server

1. Go to `/Oracle/Middleware/my_oep/user_projects/domains/ fraudcheck/ domain/defaultserver`.

2. Execute the appropriate startup script:

   a. On Windows:

      - `prompt> startwlevs.cmd`

   b. On UNIX:

      - `prompt> ./startwlevs.sh`

   The terminal window displays messages as the server starts. When you see, `<The application context for "com.bea.wlevs.dataservices" was started successfully >`, the Oracle Event Processing server is ready.

5.3.6 Use Oracle JDeveloper to Create An Oracle Event Processing Application

- Start Oracle JDeveloper
- Create a New Application
- Add an Inbound Adapter Stage
- Add an Input Channel Stagestage
- Connect the Input Channel and Inbound Adapter Components
- Set properties on the edn-inbound-adapter
- Add an EPN Oracle CQL processor Stage
- Add CQL code to the Processor
- Connect the Input Channel and the Oracle CQL processor
- Add an Outbound Channel Stage
- Set properties on the Outbound Channel
- Connect the Processor and the Outbound Channel
- Add an Outbound Adapter Stage
- Connect the Outbound Channel and the Outbound Adapter

Start Oracle JDeveloper

If Oracle JDeveloper is not already running, start it.

1. Go to `/Oracle/Middleware/my_soa/jdeveloper/jdev/bin`.

2. Type `./jdev -clean`.

   The Select Role dialog displays.

3. In the **Select Role** dialog, select **Studio Developer (All Features)** and click **OK**.

   Wait a few moments while Oracle JDeveloper starts.

Create a New Application

1. Select **File > New > Application**.

   The New Gallery dialog displays.

2. In the **New Gallery** dialog, select **OEP Application** and click **OK**.

   The Create OEP Application screen displays.

3. In the **Create OEP Application** dialog, enter the following values:

   Application Name: FraudOEPApplication Directory: Accept the default
   Application Package Prefix: Leave blank

4. Click **Next**.

   The Create OEP Application - Step 2 screen displays.

5. In the **Create OEP Application - Step 2** dialog, enter the following values:

   Project Name: FraudCheck Directory: Accept the default Project Features: OEP
   Suite

6. Click **Next**.

   The Create OEP Application - Step 3 dialog displays.

7. In the **Create OEP Application - Step 3** dialog, click **Next** to accept the defaults.

   The Create OEP Application - Step 4 dialog displays.

8. In the **Create OEP Application - Step 4** dialog, Click **Finish** to accept the defaults:

   The **Empty OEP Project** template provides the basic structure of an Oracle Event
   Processing application, which are an empty configuration file and assembly files.

   The **OEP Server Connections** can be left blank at this stage. In a later step, you
   create the Oracle Event Processing server connection.

9. Click **Finish**.

   The Oracle Event Processing FraudCheck project displays. I
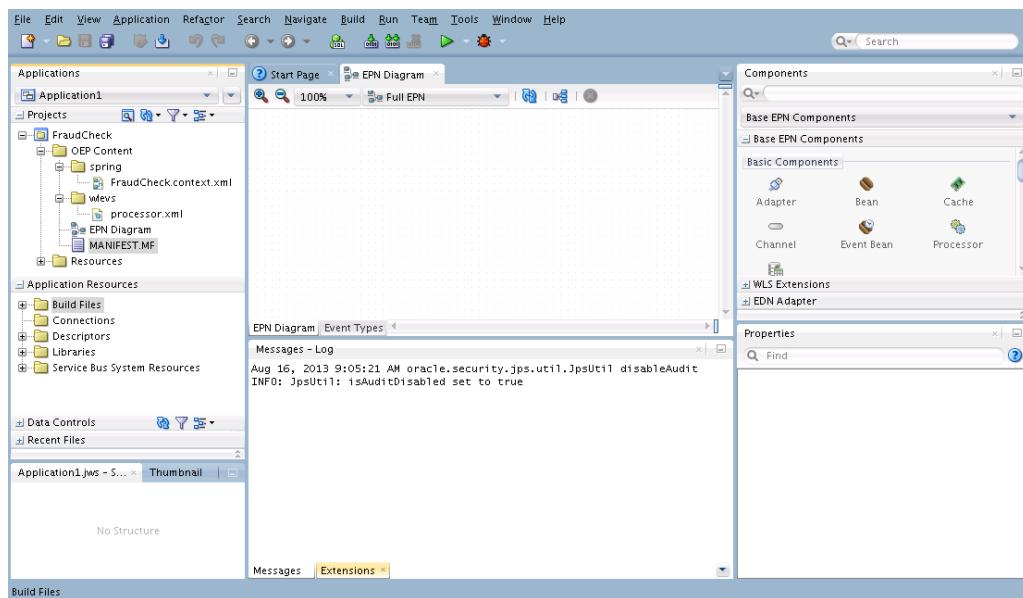
10. If you cannot see the empty EPN Diagram tab and the components panel, go to the Applications panel, expand **FraudCheck > OEP Content** and double-click EPN Diagram.

Add an Inbound Adapter Stage

1. With the EPN diagram open, go to the right panel under **Components**, and open the **EDN Adapter** window

   The EDN inbound and outbound adapters display.

2. Drag the **EDN Inbound Adapter** component from the **EDN Adapter** window to the empty middle panel (canvas).

   The New EDN Adapter wizard starts.

3. In the **New EDN Adapter wizard**, provide the following values:

   Adapter ID: edn-inbound-adapter. File name: adapter.xml.

   Changing the file name from processor.xml to a name that is specific to your usage, distinguishes the files you create from the default files provided by Oracle JDeveloper.

4. Click **Next**.

5. In the **EDN Inbound Adapter Configuration** dialog, specify the following values that relate to your Oracle WebLogic Server configuration.

   JNDI Provider URL: t3://localhost:7101. JNDI Factory: weblogic.jndi.WLInitialContextFactory. User: weblogic. Password: welcome1.

   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.

6. Under **Edl Properties**, load the **Edl File** as follows:

   a. Click the **search** icon (magnifying glass) next to the **EDL File** field.

    b. Navigate to the location where you unzipped the `OEP_Fraud_Detection_Walkthrough_Files.zip` file.

    c. Select the `FraudCheckEvent.edl` file inside the folder and click **OK**.

7. Under **Edl Properties**, select **FraudCheckRequest** from the Event Type drop-down list.

8. In the **EDN Inbound Adapter Configuration** dialog, under **Advanced Properties**, select the schema file associated with the FraudCheckEvent.edl file as follows:

    a. Click the **search** icon (magnifying glass) next to the **Schema File** field.

    b. Navigate to the location where you unzipped the `OEP_Fraud_Detection_Walkthrough_Files.zip` file.

    c. Select the `FraudCheckType.xsd` file inside the folder and click **OK**.

9. Click **Finish**.

Two informational dialogs display about the files you are uploading.

10. Read the informational message and press **OK** to dismiss them.

The EDN diagram displays the edn-inbound-adapter that you just created, and the Fraud Check project lists the files that you uploaded.

11. Select **File > Save** to save your work.

Add an Input Channel Stagestage

1. In the right panel under **Components**, open the **Base EPN Components** window.

The base EPN components display.

2. Drag the **Channel** component to a free space on the canvas.

The New Channel dialog displays.

3. In the **New Channel** dialog, provide the following information:

Channel ID: ednInputChannel. Event Type: <NONE>

4. Click **OK**.

The EDN diagram displays the channel that you just created.

5. Select **File > Save** to save your work.

Connect the Input Channel and Inbound Adapter Components

1. Select and hold **edn-inbound-adapter** with the left mouse button.

2. Drag the **edn-inbound-adapter** to the **ednInputChannel**.

After a few moments, a line displays to connect the adapter to the channel. The components adjust so that the adapter and channel are in a line going left to right with the channel to the left of the adapter. This alignment depicts the flow of information into and out of the EDN from left to right.

3. Select **File > Save** to save your work.

Set properties on the edn-inbound-adapter

1. Select the **ednInputChannel** event stage.

    Oracle JDeveloper highlights the ednInputChannel.

2. In the **Properties** window in the bottom-right corner of Oracle JDeveloper, select **edl:FraudCheckRequest** from the **event-type** drop-down list.

    The FraudCheckRequest event type is now associated with the ednInputChannel. This means that the Fraud Check application checks for FraudCheckRequest events as events come through the ednInputChannel.

3. Select **File > Save** to save your work.

Add an EPN Oracle CQL processor Stage

1. Under **Base EPN Components**, drag the **Processor** component to a free space on the canvas.

    The New Processor dialog displays.

2. In the **New Processor** dialog, provide the following information:

    Processor ID: ednProcessor. File name: processor.xml.

3. Click **OK**.

    The EDN diagram displays the ednprocessor that you just created.

4. Select **File > Save** to save your work.

Add CQL code to the Processor

1. Right-click the **ednProcessor** stage.

    The context menu displays.

2. From the context menu, select **Go To Configuration Source**.

    Oracle JDeveloper opens a source editor where you place the Oracle CQL rules to be applied to the streaming event data. The source editor provides a sample query that you can edit or replace.

3. Replace the sample Oracle CQL with the Oracle CQL provided in the `ProcessorCQLFraudSample.xml` file that is inside your `OEP_Fraud_Detection_Walkthrough_Files` folder.

    You replace the sample Oracle CQL including `<rules>..</rules>` with the following Oracle CQL code:

```
<rules>
  <view id="FraudView"><![CDATA[
```

```
    select S.properties as properties,
      cast@java(S.javaContent,
      com.oracle.oep.FraudCheckRequest.class).getOrderNumber() as orderNumber,
      cast@java(S.javaContent,
      com.oracle.oep.FraudCheckRequest.class).getEmail() as email,
      cast@java(S.javaContent,
      com.oracle.oep.FraudCheckRequest.class).getTotalAmount() as totalAmount
    from ednInputChannel as S
  ]]></view>

<!--
With id=FraudViewAmountOk, the view and query statements detect the case where
the sum of all order amounts from a specific email over a 24 hour period is less
than $1000. In this case, he query issues a FraudCheckResponseEvent with status
OK.
-->

  <view id="FraudViewAmountOk"><![CDATA[
    select email
    from FraudView[range 24 hours]
    group by FraudView.email
    having sum(FraudView.totalAmount) <= 1000.0
  ]]></view>

  <query id="FraudQueryAmountOk"><![CDATA[
    select V1.properties as properties,
    FraudCheckResponse(V1.orderNumber, "OK")
    as javaContent
    from FraudView[partition by email rows 1] as V1, FraudViewAmountOk as V2
    where V1.email = V2.email
  ]]></query>

<!--
With id= FraudViewAmountAlert, the view and query statements detect the case
where the sum of all order amounts from a specific email over a 24 hour period
is greater than $1000. In this case, the query issues a FraudCheckResponseEvent
with status THRESHOLD_EXCEEDED.
-->

  <view id="FraudViewAmountAlert"><![CDATA[
    select email
    from FraudView[range 24 hours]
    group by FraudView.email
    having sum(FraudView.totalAmount) > 1000.0
  ]]></view>

  <query id="FraudQueryAmountAlert"><![CDATA[
    select V1.properties as properties,
    FraudCheckResponse(V1.orderNumber, "THRESHOLD_EXCEEDED")
    as javaContent
    from FraudView[partition by email rows 1] as V1, FraudViewAmountAlert
    as V2 where V1.email = V2.email
  ]]></query>
</rules>
```

4. Select **File > Save** to save your work.

5. Click the **EPN Diagram** tab to return to the EPN diagram.

Connect the Input Channel and the Oracle CQL processor

1. Select and hold **ednInputChannel** with the left mouse button.

Oracle JDeveloper highlights the ednInputChannel.

2. Drag **ednInputChannel** to the **ednProcessor**.

After a few moments, a line displays to connect the input channel to the Oracle CQL processor. The components adjust so that the adapter, channel, and Oracle CQL processor are in a line going left to right.
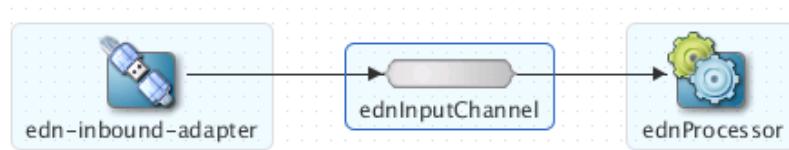
3. Select **File > Save** to save your work.

Add an Outbound Channel Stage

1. Under **Base EPN Components**, drag the **Channel** component to a free space on the canvas.

The New Channel dialog displays.

2. In the **New Channel** dialog, provide the following information:

Channel ID: ednOutputChannel. Event Type: <NONE>

3. Click **OK**.

The EDN diagram displays the channel that you just created.

4. Select **File > Save** to save your work.

Set properties on the Outbound Channel

1. Select **ednOutputChannel**.

Oracle JDeveloper highlights the ednOutputChannel.

2. In the **Properties** window in the bottom-right corner of Oracle JDeveloper, select **edl:FraudCheckResponse** from the **event-type** drop-down list.

3. Select **File > Save** to save your work.

Connect the Processor and the Outbound Channel

1. Select and hold **ednProcessor** with the left mouse button.

Oracle JDeveloper highlights the ednProcessor.

2. Drag **ednProcessor** to **ednOutputChannel**.

After a few moments, a line displays to connect the Oracle CQL processor to the output channel. The components adjust so that the adapter, channel, and Oracle CQL processor are in a line going left to right.
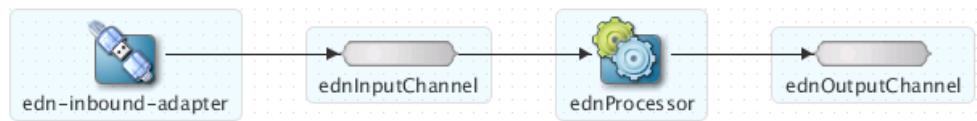
3. Select **File > Save** to save your work.

Add an Outbound Adapter Stage

1. Under **EDN Adapters**, drag the **EPN Outbound Adapter** component from the **EDN Adapter** window to the canvas.

   The New EDN Adapter wizard starts.

2. In the **New EDN Adapter wizard**, provide the following values:

   Adapter ID: edn-outbound-adapter. File name: adapter.xml.

3. Click **Next**.

   The EDN Outbound Adapter Configuration dialog displays.

4. In the **EDN Outbound Adapter Configuration** dialog, specify the following values that relate to your Oracle WebLogic Server configuration.

   JNDI Provider URL: t3://localhost:7101. JNDI Factory: weblogic.jndi.WLInitialContextFactory. User: weblogic. Password: welcome1.

5. In the EDN Outbound Adapter Configuration dialog, under **Edl Properties**, load the **Edl File** as follows:

   a. Click the **search** icon (magnifying glass) next to the **EDL File** field.

   b. Navigate to the location where you unzipped the `OEP_Fraud_Detection_Walkthrough_Files.zip` file.

      Because you already loaded this file, you can also locate it under `$HOME/jdeveloper/mywork/Application1/FraudCheck/META-INF/wlevs/edn`.

   c. Select the `FraudCheckEvent.edl` file inside the folder and click **OK**.

      When you load the FraudCheckEvent.edl file, the Event Type drop-down list is populated with events to use in the Fraud Check application.

6. In the EDN Outbound Adapter Configuration dialog, in the Event Type drop-down list, select **FraudCheckResponse**.

7. Click **Finish**.

   The EDN diagram displays the edn-inbound-adapter that you just created, and the Fraud Check project lists the files that you uploaded.

8. Select **File > Save** to save your work.

Connect the Outbound Channel and the Outbound Adapter

1. Select and hold **ednOutputChannel** with the left mouse button.

   Oracle JDeveloper highlights the ednOutputChannel.

2. Drag **ednOutputChannel** to edn-Outbound-adapter.

After a few moments, a line displays to connect the output channel to the output adapter. The components adjust so that the adapter, channel, and Oracle CQL processor are in a line going left to right.

3. Select **File > Save** to save your work.

5.3.7 Deploy the Application with JDeveloper

Application deployment involves creating a server connection and a deployment profile.

- Create an Oracle Event Processing Server Connection
- Deploy the Application

Create an Oracle Event Processing Server Connection

1. In Oracle JDeveloper, select **File > New > From Gallery**.

The New Gallery dialog displays.

2. In the **New Gallery** dialog under **Categories**, select **Connections**.

3. In the New Gallery dialog under **Items**, select **OEP Connection**, and click **OK**.

The Create OEP Server Connection dialog displays.

4. In the Create OEP Server Connection dialog, provide the following information:

Connection will be created in: IDE Connections: Checked OEP Server Connection Name: FraudDetectionConnection. OEP Server Home Path: /Oracle/ Middleware/my_oep/ Use Default Values: Unchecked. OEP Server Projects Directory: `user_projects/domains/ fraudcheck_domain/ defaultserver`. Use Default Values: Checked. Host: 127.0.0.1. Port: 9002. Use Default Values: Unchecked. Username: oepadmin. User Password: welcome1. Additional Parameters for OEP Server: blank.

5. In the Create OEP Server Connection dialog, click **Test Connection**.

If you see **Success** in the area below the **Test Connection** button, you entered the information correctly. If you see errors, correct them and test again until you see **Success**.

6. In the Create OEP Server Connection dialog, when you see the **Success** message, click **OK**.

FraudDetectionConnection displays under Application Resources > Connections > OEP Server Connection in the left panel.

Create a Deployment Profile

A deployment profile creates an OSGi bundle that contains the required library JAR file.

1. Right-click the **FraudCheck** project and select **Deploy > New Deployment Profile**.

   The Create Deployment Profile dialog displays.

2. In the **Create Deployment Profile** dialog, provide the following values.

   Profile Type: OEP Project Deployment Profile.

   > **Note:**
   >
   > Make sure you select the correct Profile Type, OEP Project Deployment Profile.

   Deployment Profile Name: oep-profile-Production

3. Click **OK**.

   The Deployment Properties dialog displays.

4. In the **Deployment Properties** dialog, edit the information so that it is correct:

   Connection to Local OEP Server: FraudDetectionConnection (127.0.0.1:9002) Symbolic Name: FraudOEPApplication.FraudCheck Bundle Name: FraudOepApplication.FraudCheck Bundle Version: 1.0.0 OSGi JAR file: Application1/FraudCheck/deploy/oep_profile-production.jar

5. Click **OK**.

Deploy the Application

1. Right-click the **FraudCheck** project.

   The context menu displays.

2. In the context menu, select **Deploy > oep_profile-Production.**

   The Deploy oep_profile-Production dialog displays.

3. In the **Deploy oep_profile-Production** dialog, select **Deploy OSGi bundle to target platform**.

4. Click **Next**.

   The Deploy oep_profile-Production Summary dialog displays.

5. In the next Deploy oep_profile-Production **Summary** dialog, confirm the information.

6. Click **Finish**.

   In the Deployment - Log window at the bottom of the middle panel, messages indicate the successful deployment. Your Oracle Event Processing Application is running and waiting for EDN events to arrive for processing.

   In the left panel under Application Resources, you can see the FraudOEPApplication.FraudCheck [Running] connection displays under Connections > OEP Server Connection > OEPConnection > Applications.

The terminal window where you started the Oracle Event Processing window.
```
INFO: Subscribe Event from Topic=jms/fabric/EDNTopic,
JmsType=WLJMS, isXA=false.
```

5.3.8 Create and Deploy the Sample SOA Composite

A sample SOA composite provides FraudCheckRequest events so that the application can check for potential fraudulent activities. How to create the sample composite is not fully described, but it does provide a good code sample for you to leverage in existing or new SOA composites.

The EDNOEPv2Proj SOA composite is provided in the OEP_Fraud_Detection_Walkthrough_Files.zip file. The SOA composite sends EDN events to the Oracle Event Processing application and the resulting EDN events are sent back to the SOA composite. The SOA composite then uses the JCA File adapter to write the information to a file that is saved for you to review and analyze later.

Access the Provided SOA Composite

1. In Oracle JDeveloper, select **File > Open**.

    The Open dialog displays.

2. In the **Open** dialog, navigate to where you unzipped the **OEP_Fraud_Detection_Walkthrough_Files.zip** file.

3. Open the **EDNOEPv2** folder and select **EDNOEPv2.jws** project.

4. Click **Open**.

    Oracle JDeveloper adds the SOA composite to the FraudCheck project.

View and Deploy the SOA Mediators

1. Navigate to **Projects > EDNOEPv2Proj > SOA > Mediators** and view the mediators.

2. Right-click the **EDNEOPv2Proj** project.

    The context menu displays.

3. In the context menu, select **Deploy > New Deployment Profile**.

    The Create Deployment Profile dialog displays.

4. In the Create Deployment Profile dialog under **Profile Type**, select **SOA-SAR** File.

5. In the Ceate Deployment Profile dialog in the **Deployment Profile Name** field, enter **sar_OEP_TEST_COMPOSITE**.

6. Click **OK**.

    The SAR Deployment Properties dialog displays.

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

8. Right-click the **EDNEOPv2Proj** project.

    The context menu displays.

9. In the context menu, select **Deploy > sar_OEP_TEST_COMPOSITE**

    The Deployment Action dialog displays.

10. In the **Deployment Action** dialog, select **Deploy to Application Server**.

11. Click **Next**.

    The Deploy Configuration dialog displays.

12. In the **Deploy Configuration** dialog, check the **Overwrite any existing composites with the same revision ID** check box.

    Keep the default values.

13. Click **Next**.

    The Select Server dialog displays.

14. In the **Select Server** dialog, select **IntegratedWebLogicServer** and click **Next**.

    The SOA Servers dialog displays.

15. In the **SOA Servers** dialog, click **Next** to keep the default values.

    The Summary dialog displays.

16. In the **Summary** dialog, review the settings for accuracy, and click **Finish**.

    When the modified SOA composite successfully deploys, it creates a default JMS mapping for the FraudCheckRequest and FraudCheckResponse event types. It is likely that this JMS mapping uses AQ, instead of WLS JMS. To interoperate with this Oracle Event Processing application, the JMS mapping for these specific event types needs to be changed to use WLS JMS. The mapping change is done with Oracle Enterprise Manager, which is part of Oracle SOA Suite.

Use Oracle Enterprise Manager to Verify the JMS Mapping

1. Open a browser, and enter **localhost:7101/em** into the URL box.

2. Log in with the user name **weblogic** and the password **welcome1**.

3. In the left panel, expand **SOA > soa-infra (Default Server) > default > EDNOEPv2Proj**.

4. Right-click **soa-infra (Default Server)** and select **Business Events** from the context menu.

    The Business Events screen displays in the right panel.

5. On the **Business Events** screen, select the **Events** tab.

    The Namespaces and Events table displays FraudCheckRequest and FraudCheckResponse with a Default link in the JMS Mapping column.

6. In the **Events** tab, select the **Default** link.

    The JMS Mapping dialog displays.

7. In the **JMS Mapping** dialog, verify the following information and make changes as needed.

    Oracle Enterprise Messaging System(OEMS): Oracle Weblogic JMS JNDI Connection Factory (XA, Durable): eis/wls/EDNxaDurableTopic JNDI

Connection Factory (XA, Non-Durable): eis/wls/EDNxaTopic JNDI Connection Factory (Non-XA, Durable): eis/wls/EDNLocalTxDurableTopic JNDI Connection Factory (Non-XA, Non-Durable): eis/wls/EDNLocalTxTopic**JMS Topic Name: jms/fabric/EDNTopic**

8. Press **Apply**.

   The JMS Mapping value changes to Modified.

9. Repeat for the second **Default** link.

Update the Sample SOA Composite for Results Analysis

1. In Oracle JDeveloper, under **EDNOEPv2Proj > SOA**, double-click **EDNOEPv2Proj** to display the diagram.

2. On the diagram, double-click the **EDNToFileOutput** component.

   The File Adapter Configuration Wizard displays.

3. In the **File Adapter Configuration Wizard**, step through the dialogs accepting the provided values until you get to the File Configuration dialog.

   The File Configuration dialog displays.

4. In the **File Configuration** dialog, change the **Directory for Outgoing Files (physical path)** to a location that is valid in your environment.

5. Write the new location down because you will use it later to review and analyze the response information.

6. Click **Next** and complete the remaining steps by accepting the default values.

7. Click **Finish**.

Redeploy the SOA Composite

1. Right-click the **EDNEOPv2Proj** project.

   The context menu displays.

2. From the context menu, select **Deploy > sar_OEP_TEST_COMPOSITE**

   The Deployment Action dialog displays.

3. In the **Deployment Action** dialog, select **Deploy to Application Server**.

4. Click **Next**.

   The Deploy Configuration dialog displays.

5. In the **Deploy Configuration** dialog, check the **Overwrite any existing composites with the same revision ID** check box and keep the default values.

6. Click **Next**.

   The Select Server dialog displays.

7. In the **Select Server** dialog, select **IntegratedWebLogicServer** and click **Next**.

   The SOA Servers dialog displays.

8. In the **SOA Servers** dialog, click **Next** to keep the default values.

The Summary dialog displays.

9. In the **Summary** dialog, review the settings for accuracy, and click **Finish**.

When the modified SOA composite successfully deploys, it creates a default JMS mapping for the FraudCheckRequest and FraudCheckResponse event types. It is likely that this JMS mapping uses AQ, instead of WLS JMS. To interoperate with this Oracle Event Processing application, the JMS mapping for these specific event types needs to be changed to use WLS JMS. The event types change is done with Oracle Enterprise Manager, which is part of Oracle SOA Suite

5.3.9 Test the Fraud Detection Application

Use Oracle Enterprise Manager to test the SOA Composite and its interaction and integration with the new Oracle Event Processing Application.

Log in to Enterprise Manager and navigate to the Test Web Service Screen

1. Open a browser, and enter **localhost:7101/em** into the URL box.

The Enterprise Manager login screen displays.

2. Log in to Enterprise Manager with the user name **weblogic** and the password **welcome1**.

The SOA Infrastructure screen displays.

3. In the SOA Infrastructure screen left panel under Target Navigation, expand **SOA > soa-infra (Default Server) > default** and select **EDNOEPv2Proj**.

4. In the SOA Infrastructure right panel, press the **Test** button.
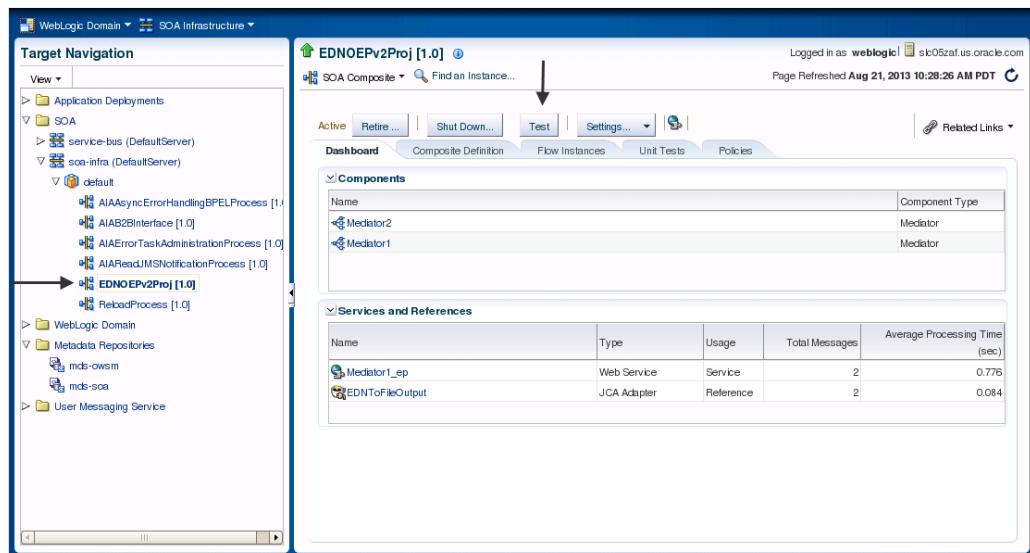
Test the SOA Composite and the Oracle Event Processing Application

1. On the **Test Web Service** screen, scroll down in the right panel until you see the **Input Arguments** section.

2. In the **Input Arguments** section under **SOAP Body**, notice that there are two fields, **email** and **amount**.

   The email and amount fields let you enter an email address and an amount to be passed to the Oracle Event Processing application. The email address is used by the Oracle CQL `Group By` clause to identify each collection of related orders and the dollar amount value.

3. In the **email** field, enter an **email address**, and in the **amount** field, enter **200.00**.

4. Scroll to the top of the page and click **Test Web Service**.

   Check the terminal window where you started Oracle WebLogic server. You see a message like the following that lets you know that a **FraudCheckEvent** has been published to the file:

   ```
   INFO: Publishing Event "{http://xmlns.oracle.com/Application2/Project1/
   FraudCheckEvent}FraudCheckResponse" to Topic="jms/fabric/EDNTopic",
   JmsType=WLJMS, isXA=false
   ```

5. With the same **email** address, enter more **amounts** and click **Test Web Service** until you have submitted more than $1000 worth of events.

6. With a text editor, navigate to the directory for the **oepToedn-output_PRODUCTION.txt** file and open it.

   For the created order number, the record status is THRESHOLD EXCEEDED. This status was determined by the Oracle CQL statements in the Oracle Event Processing Application.

New   Open   Save   Print...   Undo   Redo   Cut   Copy   Paste   Find   Replace

*oepToedn-output.txt   ✕

```
<?xml version="1.0" encoding="UTF-8" ?>

<ns0:FraudCheckResponse xmlns:jca="http://xmlns.oracle.com/pcb
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http:
pcbpel/adapter/file/EDNOEPv2/EDNOEPv2Proj/EDNToFileOutput" xml
schemas.xmlsoap.org/ws/2003/05/partner-link/" xmlns:ns0="http:
oep">
  <ns0:OrderNumber>6</ns0:OrderNumber>
  <ns0:Status>OK</ns0:Status>
</ns0:FraudCheckResponse>

<?xml version="1.0" encoding="UTF-8" ?><ns0:FraudCheckResponse
xmlns.oracle.com/pcbpel/wsdl/jca/" xmlns:wsdl="http://schemas.
xmlns:tns="http://xmlns.oracle.com/pcbpel/adapter/file/EDNOEPv
EDNToFileOutput" xmlns:plt="http://schemas.xmlsoap.org/ws/2003
xmlns:ns0="http://www.oracle.com/oep">
  <ns0:OrderNumber>6</ns0:OrderNumber>
  <ns0:Status>THRESHOLD_EXCEEDED</ns0:Status>
</ns0:FraudCheckResponse>

<?xml version="1.0" encoding="UTF-8" ?><ns0:FraudCheckResponse
xmlns.oracle.com/pcbpel/wsdl/jca/" xmlns:wsdl="http://schemas.
xmlns:tns="http://xmlns.oracle.com/pcbpel/adapter/file/EDNOEPv
EDNToFileOutput" xmlns:plt="http://schemas.xmlsoap.org/ws/2003
xmlns:ns0="http://www.oracle.com/oep">
<ns0:OrderNumber>6</ns0:OrderNumber>
<ns0:Status>OK</ns0:Status>
</ns0:FraudCheckResponse>
```

6

Event Processing Samples in Oracle Event Processing

This chapter introduces the sample code provided with the Oracle Event Processing installation and describes how to set up and use the code. You must have installed Oracle Event Processing with the Examples check box checked.

This chapter covers the following topics:

- About the Samples

- Environment Setup

- Use Oracle Event Processing Visualizer with the Samples

- Increase the Performance of the Samples

- HelloWorld Example

- Oracle Continuous Query Language Example

- Oracle Spatial Example

- Foreign Exchange (FX) Example

- Signal Generation Example

- Event Record and Playback Example.

6.1 About the Samples

When you choose to include examples during installation, the Oracle Event Processing installation includes the following samples:

- HelloWorld: Provides a basic skeleton for an Oracle Event Processing application.

- Oracle CQL: Shows how to use the Oracle Event Processing Visualizer Query Wizard to construct Oracle CQL queries to process event streams.

- Oracle Spatial: Shows how to use Oracle Spatial with Oracle CQL queries to process a stream of Global Positioning System (GPS) events. The GPS events track the location of buses and generate alerts when a bus arrives at bus stop positions.

- Foreign Exchange (FX): Includes multiple components.

- Signal Generation: Simulates market trading and trend detection.

- Event record and playback: Shows how to configure event record and playback using a persistent event store.

These samples are provided in the following two forms:

- Ready-to-Run Samples

- Sample Source.

The samples use Ant as their development tool. For details about Ant and installing it on your computer, see `http://ant.apache.org/`.

The Oracle Event Processing installation directory is referred to as */Oracle/Middleware/ my_oep/*.

6.1.1 Ready-to-Run Samples

The ready-to-run samples have domains that are preconfigured to deploy the assembled application. Each domain is a standalone server domain, and the server files are located in the `defaultserver` subdirectory of the domain directory. To deploy the application, start the default server in the domain.

- The sample HelloWorld domain is located in `/Oracle/Middleware/my_oep/ oep/ examples/domains/helloworld_domain`.

  See Run the HelloWorld Example from the helloworld Domain for details.

- The sample CQL domain is located in /Oracle/Middleware/my_oep/oep/ examples/domains/cql_domain.

  See Run the CQL Example for details.

- The sample Oracle Spatial domain is located in */Oracle/Middleware/my_oep/ oep/ examples/domains/spatial_domain*.

  See Run the Oracle Spatial Example for details.

- The sample Foreign Exchange domain is located in */Oracle/Middleware/ my_oep/ oep/examples/domains/fx_domain*.

  See Run the Foreign Exchange Example for details.

- The sample Signal Generation domain is located in */Oracle/Middleware/my_oep/ oep/examples/domains/signalgeneration_domain*.

  See Run the Signal Generation Example for details.

- The sample Record and Playback domain is in */Oracle/Middleware/my_oep/ oep/examples/domains/recplay_domain*.

  See Run the Event Record/Playback Example for details.

6.1.2 Sample Source

The Java and configuration XML source for each sample is provided in a separate source directory that describes a sample development environment.

- The HelloWorld source directory is located in */Oracle/Middleware/my_oep/ oep/ examples/source/applications/helloworld*.

  See Implementation of the HelloWorld Example for details.

- The CQL source directory is located in */Oracle/Middleware/my_oep/oep/ examples/source/applications/cql*.

  See Implementation of the CQL Example for details.

- The Oracle Spatial source directory is in */Oracle/Middleware/my_oep/*`oep/`
  `examples/source/applications/spatial`.

  See Implementation of the Oracle Spatial Example for details.

- The Foreign Exchange source directory is located in `/Oracle/Middleware/`
  `my_oep/ oep/examples/source/applications/fx`.

  See Implementation of the FX Example for details.

- The Signal Generation source directory is located in `/Oracle/Middleware/`
  `my_oep/ oep/ examples/source/applications/signalgeneration`.

  See Implementation of the Signal Generation Example for details.

- The Record and Playback source directory is located in */Oracle/Middleware/my_oep/*
  `oep/examples/source/applications/recplay`.

  See Implementation of the Record and Playback Example for details.

6.2 Environment Setup

To run the examples, your development environment must have JDK7_u55 or above
installed. You must set `JAVA_HOME` as follows.

```
export JAVA_HOME=< path to installation directory >
export PATH=${JAVA_HOME}/bin:${PATH}
```

To build and run the sample source, your development environment must have Ant
installed. You must set `ANT_HOME` as follows:

```
export ANT_HOME=<path to Ant directory>
export PATH=${ANT_HOME}/bin:${PATH}
```

6.3 Use Oracle Event Processing Visualizer with the Samples

The Oracle Event Processing Visualizer is a Web 2.0 application that consumes data
from Oracle Event Processing, displays it in a useful and intuitive way to system
administrators and operators, and for specified tasks, accepts data that is passed back
to Oracle Event Processing so as to change its configuration.

Visualizer is itself an Oracle Event Processing application and is automatically
deployed in each server instance. To use it with the samples, be sure you have started
the server (instructions provided for each sample below) and then invoke the
following URL in your browser:

```
http://host:9002/wlevs
```

where `host` refers to the name of the computer hosting Oracle Event Processing. If it
is the same as the computer on which the browser is running you can use `localhost`.

Security is disabled for the HelloWorld application, so you can click Logon at the login
screen without entering a user name and password. For the FX and signal generation
samples, security is enabled, so use the following user name and password to log in:

Username: oepadmin
Password: welcome1

For more information about Oracle Event Processing Visualizer, see *Developing
Applications for Oracle Event Processing*.

6.4 Increase the Performance of the Samples

When you run Oracle Event Processing on a computer with a larger amount of memory, set the load generator and server heap sizes appropriately for the size of the computer.

On computers with sufficient memory, Oracle recommends a heap size of 1 GB for the server and between 512MB - 1GB for the load generator.

6.5 HelloWorld Example

The HelloWorld sample shows how to create a typical Oracle Event Processing application.

Figure 6-1 shows the HelloWorld example Event Processing Network (EPN). The EPN contains the components that make up the application and defines how they fit together.

Figure 6-1   The HelloWorld Example Event Processing Network

The example includes the following components:

- `helloworldAdapter`: Component that generates *Hello World* messages every second. In a real-world scenario, this component typically reads a stream of data from a source, such as a data feed from a financial institution, and converts it into a stream of events that the Oracle CQL processor can understand. The HelloWorld application also includes a `HelloWorldAdapterFactory` that creates instances of `HelloWorldAdapter`.

- `helloworldInputChannel`: Component that streams the events generated by the adapter (in this case *Hello World* messages) to the Oracle CQL processor.

- `helloworldProcessor`: Component that forwards the messages from the `helloworldAdapter` component to the Plain Old Java Object (POJO) that contains the business logic. In a real-world scenario, this component typically executes additional and possibly much more processing of the events from the stream, such as selecting a subset of events based on a property value, grouping events, and so on using Oracle CQL.

- `helloworldOutputChannel`: Component that streams the events processed by the Oracle CQL processor to the POJO that contains the user-defined business logic.

- `helloworldBean`: POJO component that prints out a message every time it receives a batch of messages from the Oracle CQL processor through the output channel. In a real-world scenario, this component contains the business logic of the application, such as running reports on the set of events from the Oracle CQL processor, sending appropriate emails or alerts, and so on.

6.5.1 Run the HelloWorld Example from the helloworld Domain

The HelloWorld application is pre-deployed to the `helloworld` domain. To run the application, start an instance of Oracle Event Processing server.

Run the HelloWorld example from the helloworld domain:

1. Open a command window and change to the default server directory of the helloworld domain directory, located in install with install with */Oracle/ Middleware/my_oep/*oep/examples/domains/helloworld_domain/defaultserver.

2. Start Oracle Event Processing by executing the appropriate server startup script with the correct command-line arguments:

   a. On Windows:

      - `prompt> startwlevs.cmd`

   b. On UNIX:

      - `prompt> ./startwlevs.sh`

   After the server starts, you should see the following message printed to the output about every second:

   ```
   Message: HelloWorld - the current time is: 3:56:57 PM
   ```

   This message indicates that the HelloWorld example is running correctly.

6.5.2 Build and Deploy the HelloWorld Example from the Source Directory

The HelloWorld sample source directory contains the Java source and other required resources such as configuration XML files, that make up the HelloWorld application. The `build.xml` Ant file contains targets to build and deploy the application to the helloworld domain.

See also Description of the Ant Targets to Build Hello World.

Build and deploy the HelloWorld example from the source directory:

1. If the helloworld Oracle Event Processing server is not already running, follow the procedure in Run the HelloWorld Example from the helloworld Domain to start the server.

   You must have a running server to successfully deploy the rebuilt application.

2. Open a new command window and change to the HelloWorld source directory, located in */Oracle/Middleware/my_oep/oep/examples/source/applications/ helloworld*.

3. Execute the `all` Ant target to compile and create the application JAR file:

   ```
   prompt> ant all
   ```

4. Execute the `deploy` Ant target to deploy the application JAR file to Oracle Event Processing:

   ```
   prompt> ant -Daction=update deploy
   ```

   Caution:

   This target overwrites the existing helloworld application JAR file in the domain directory.

You should see the following message printed to the output about every second:

```
Message: HelloWorld - the current time is: 3:56:57 PM
```

This message indicates that the HelloWorld example has been redeployed and is running correctly.

6.5.3 Description of the Ant Targets to Build Hello World

The `build.xml` file, located in the top level of the HelloWorld source directory, contains the following targets to build and deploy the application:

- `clean`: This target removes the `dist` and `output` working directories under the current directory.

- `all`: This target cleans, compiles, and puts the application into a JAR file called `com.bea.wlevs.example.helloworld_12.1.3.0_0.jar`, and places the generated JAR file into a `dist` directory below the current directory.

- `deploy`: This target deploys the JAR file to Oracle Event Processing using the Deployer utility.

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

6.5.4 Implementation of the HelloWorld Example

The HelloWorld example, because it is relatively simple, does not use all of the components and configuration files described in the general procedure for creating an Oracle Event Processing application.

All the example files are located relative to the /Oracle/Middleware/my_oep/examples/source/applications/helloworld directory.

The files used by the HelloWorld example include:

- An EPN assembly file that describes each component in the application and how all the components are connected together. The EPN assembly file extends the standard Spring context file. The file also registers the event types used in the application. You are required to include this XML file in your Oracle Event Processing application.

  In the example, the file is called `com.bea.wlevs.example.helloworld-context.xml` and is located in the `~/META-INF/spring` directory.

- Java source file for the `helloworldAdapter` component.

  In the example, the file is called `HelloWorldAdapter.java` and is located in the `~/src/com/bea/wlevs/adapter/examples/helloworld` directory.

  For a detailed description of this file and how to program the adapter Java files in general, see Integrating an External Component Using a Custom Adapter in *Developing Applications for Oracle Event Processing*.

- Java source file that describes the `HelloWorldEvent` event type.

  In the example, the file is called `HelloWorldEvent.java` and is located in the `~/src/com/bea/wlevs/event/examples/helloworld` directory.

  For a detailed description of this file, and general information about programming event types, see Defining and Using Event Types in Oracle Fusion Middleware Developing Application for Oracle Event Processing .

- An XML file that configures the `helloworldProcessor` and `helloworldOutputChannel` components. An important part of this file is the set of Oracle CQL rules that select the set of events that the HelloWorld application processes. You are required to include a Oracle CQL processor configuration file in your Oracle Event Processing application, although the adapter and channel configuration is optional.

  In the example, the file is called `config.xml` and is located in the `~/META-INF/wlevs` directory.

- A Java file that implements the `helloworldBean` component of the application, a POJO that contains the business logic.

  In the example, the file is called `HelloWorldBean.java` and is located in the `~/src/com/bea/wlevs/examples/helloworld` directory.

  For a detailed description of this file and general information about programming event sinks, see *Handling Events* with Java in *Oracle Fusion Middleware Developing Application for Oracle Event Processing* .

- A `MANIFEST.MF` file that describes the contents of the OSGi bundle to be deployed to Oracle Event Processing.

  In the example, the `MANIFEST.MF` file is located in the `META-INF` directory.

  For more information about creating this file and a description of creating the OSGi bundle that you deploy to Oracle Event Processing, see *Oracle Fusion Middleware Developing Application for Oracle Event Processing*

The HelloWorld example uses a `build.xml` Ant file to compile, assemble, and deploy the OSGi bundle; see Build and Deploy the HelloWorld Example from the Source Directory for a description of this `build.xml` file if you also use Ant in your development environment.

6.6 Oracle Continuous Query Language Example

The Oracle Continuous Query Language (Oracle CQL) example shows how to use the Oracle Event Processing Visualizer Query Wizard to construct various types of Oracle CQL queries.

Figure 6-2 shows the CQL example Event Processing Network (EPN). The EPN contains the components that make up the application and how they fit together.

> **Note:**
>
> This sample might not start on a configuration of multiple coherence clusters that have the same default multicast address and port numbers. The error message contains text similar to `... has been attempting to join the cluster at address /239.255.0.1:9100 with TTL 4 for 30 seconds without success.`
>
> To get around this problem, specify unique addresses and ports to create a distinct cluster.

Figure 6-2   The CQL Example Event Processing Network
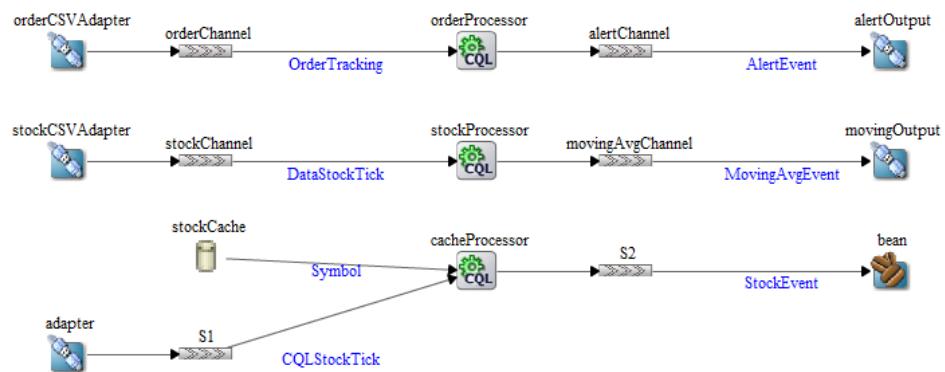
The application contains two separate event paths in its EPN:

- **Missing events**: this event path consists of an adapter `orderCVSAdapter` connected to a channel `orderChannel`. The `orderChannel` is connected to `orderProcessor` which is connected to channel `alertChannel` which is connected to adapter `alertOutput`.

  This event path is used to detect missing events in a customer order workflow.

  For more information on how to construct the query that the `cqlProc` processor executes, see Create the Missing Event Query.

- **Moving average**: The event path consists of channel `stockChannel` connected to processor `stockProcessor`, which is connected to channel `movingAvgChannel`, which is connected to adapter `movingOutput`.

  This event path is used to compute a moving average on stock whose volume is greater than 1000.

- **Cache**: this event path consists of adapter `adapter` connected to channel `S1` connected to Oracle CQL processor `cacheProcessor` connected to channel `S2` connected to bean `Bean`. There is a cache `stockCache` also connected to the Oracle CQL processor `cacheProcessor`. There is also a bean `Loader`.

  This event path is used to access information from a cache in an Oracle CQL query.

  > **Note:**
  >
  > For more information about the various components in the EPN, see the other samples in this book.

6.6.1 Run the CQL Example

For optimal demonstration purposes, Oracle recommends that you run this example on a powerful computer, such as one with multiple CPUs or a 3 GHz dual-core Intel, with a minimum of 2 GB of RAM.

The CQL application is pre-deployed to the `cql_domain` domain. To run the application, you simply start an instance of Oracle Event Processing server.

To run the CQL example:

1. Open a command window and change to the default server directory of the CQL
    domain directory, located in */Oracle/Middleware/my_oep/oep/examples/*
    domains/cql_domain/defaultserver.

2. Start Oracle Event Processing by executing the appropriate script with the correct
    command line arguments:

    a. On Windows:

        - `prompt> startwlevs.cmd`

    b. On UNIX:

        - `prompt> ./startwlevs.sh`

    The CQL application is now ready to receive data from the data feeds.

3. To simulate the data feed for the missing event query, open a new command
    window.

4. Change to the */Oracle/Middleware/my_oep/oep/utils/load-*
    generator.

5. Run the load generator using the `orderData.prop` properties file:

    a. On Windows:

        `prompt> runloadgen.cmd orderData.prop`

    b. On UNIX:

        `prompt> ./runloadgen.sh orderData.prop`

6. Change to the */Oracle/Middleware/my_oep/oep/utils/load-*
    generator.

7. To simulate the data feed for the moving average query, open a new command
    window

8. Run the load generator using the `StockData.prop` properties file:

    a. On Windows:

        `prompt> runloadgen.cmd StockData.prop`

    b. On UNIX:

        `prompt> ./runloadgen.sh StockData.prop`

9. To simulate the data feed for the cache query, you only need to run the example.

    The load data is generated by `Adaptor.java` and the cache data is generated by
    `Loader.java`. You can verify that data is flowing through by turning on
    statistics in the Oracle Event Processing Visualizer Query Plan.

6.6.2 Build and Deploy the CQL Example

The CQL sample source directory contains the Java source, along with other required
resources such as configuration XML files, that make up the CQL application. The
`build.xml` Ant file contains targets to build and deploy the application to the

`cql_domain` domain, as described in Description of the Ant Targets to Build Hello World.

To build and deploy the CQL example from the source directory:

1. If the CQL Oracle Event Processing instance is not already running, follow the procedure in Run the CQL Example to start the server.

   You must have a running server to successfully deploy the rebuilt application.

2. Open a new command window and change to the CQL source directory, located in */Oracle/Middleware/my_oep/oep/examples/source/applications/ cql*.

3. Execute the `all` Ant target to compile and create the application JAR file:

   ```
   prompt> ant all
   ```

4. Execute the `deploy` Ant target to deploy the application JAR file to Oracle Event Processing:

   ```
   prompt> ant -Dusername=oepadmin -Dpassword=welcome1 -Daction=update deploy
   ```

   Caution:

   This target overwrites the existing CQL application JAR file in the domain directory.

   ---

5. If the load generators required by the CQL application are not running, start them as described in Run the CQL Example.

6.6.3 Description of the Ant Targets to Build the CQL Example

The `build.xml` file, located in the top-level directory of the CQL source, contains the following targets to build and deploy the application:

- `clean`: This target removes the `dist` and `output` working directories under the current directory.

- `all`: This target cleans, compiles, and puts the application into a JAR file called `com.bea.wlevs.example.cql_12.1.2.0_0.jar`, and places the generated JAR file into a `dist` directory below the current directory.

- `deploy`: This target deploys the JAR file to Oracle Event Processing using the Deployer utility.

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

6.6.4 Implementation of the CQL Example

This section describes how to create the queries that the CQL example uses, including:

- Create the Missing Event Query

- Create the Moving Average Query.

6.6.4.1 Create the Missing Event Query

This section describes how to use the Oracle Event Processing Visualizer Query Wizard to create the Oracle CQL pattern matching query that `cqlProc` executes to detect missing events.

Consider a customer order workflow in which you have customer order workflow events flowing into the Oracle Event Processing system.

In a valid scenario, you see events in the order that Table 6-1 lists:

Table 6-1   Valid Order Workflow

Event Type	Description
C	Customer order
A	Approval
S	Shipment

However, it is an error if an order is shipped without an approval event as Table 6-2 lists:

Table 6-2   Invalid Order Workflow

Event Type	Description
C	Customer order
S	Shipment

You will create and test a query that detects the missing approval event and generates an alert event:

- "Create the missing event query:"

- "Test the missing event query:"

Create the missing event query:

1. If the CQL Oracle Event Processing instance is not already running, follow the procedure in Run the CQL Example to start the server.

   You must have a running server to use the Oracle Event Processing Visualizer.

2. Invoke the following URL in your browser:

   `http://host:port/wlevs`

   where *host* refers to the name of the computer on which Oracle Event Processing is running and *port* refers to the Jetty NetIO port configured for the server (default value `9002`).

   The Logon screen displays.

3. In the Logon screen, enter the **Username** `oepadmin`, **Password** `welcome1` and click **Login**.

   The Oracle Event Processing Visualizer dashboard displays.

   For more information about the Oracle Event Processing Visualizer user interface, see *Oracle Fusion Middleware Using Visualizer for Oracle Event Processing*.

4. In the right panel, expand **WLEventServerDomain > NonClusteredServer > Applications**.

5. Select the **cql** node.

   The CQL application screen displays.

6. Select the **Event Processing Network** tab.

The Event Processing Network screen displays.

7. Double-click the **orderProcessor** Oracle CQL processor icon.
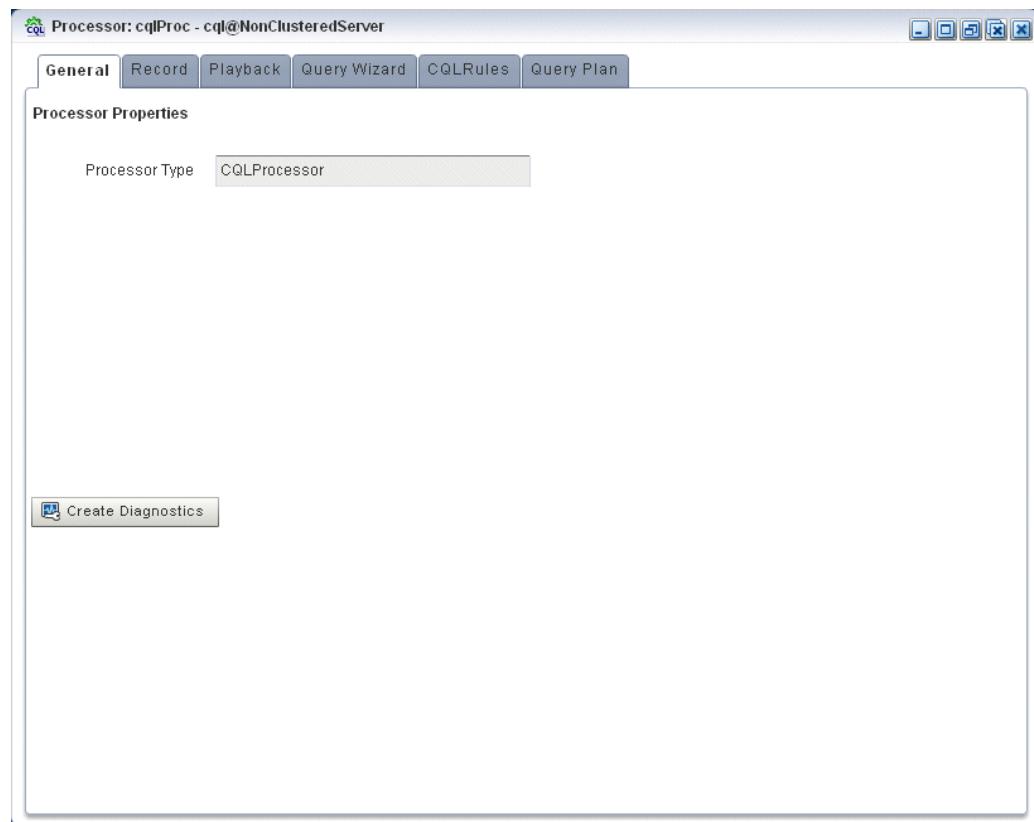
The Oracle CQL processor screen displays.

8. Select the **Query Wizard** tab.

   The Query Wizard screen Displays.
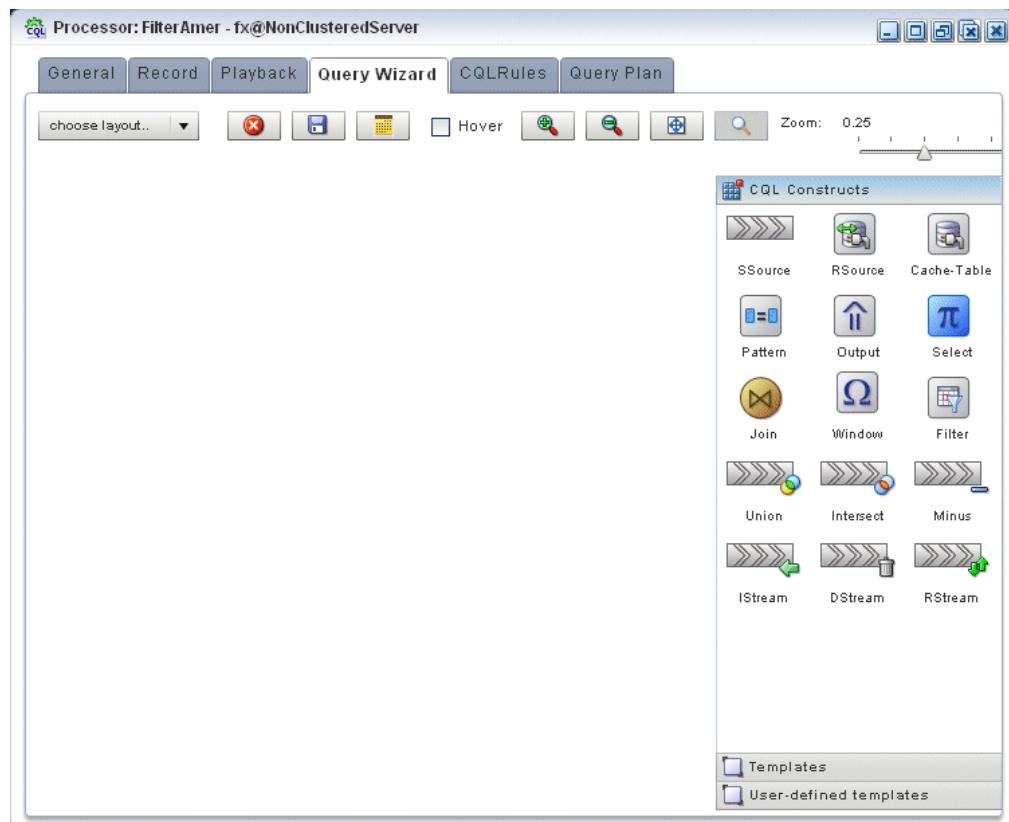
You can use the Oracle CQL Query Wizard to construct an Oracle CQL query from a template or from individual Oracle CQL constructs.

In this procedure, you are going to create an Oracle CQL query from a template.

For more information, see *Oracle Fusion Middleware Using Visualizer for Oracle Event Processing*.

9. Click the Templates tab.

The Templates tab displays.
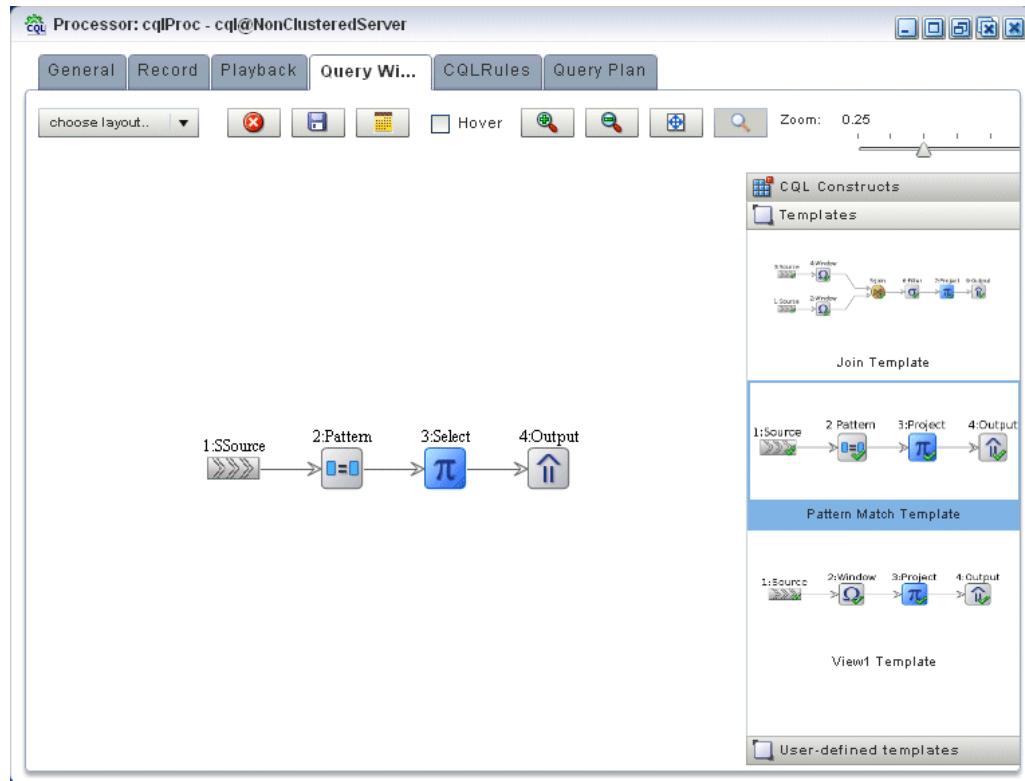
10. Click and drag the **Pattern Match Template** from the Templates palette and drop it anywhere in the Query Wizard canvas.

11. Double-click the **SSource** icon.

   The SSource configuration screen displays.
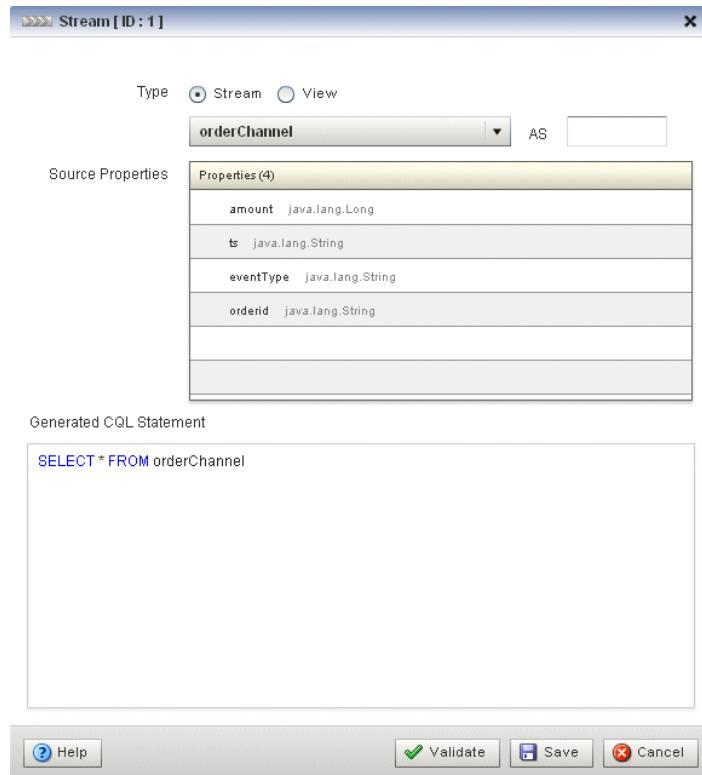
The source of your query is the `orderChannel` stream.

12. Configure the SSource as follows:

   - Select **Stream** as the Type.

   - Select **orderChannel** from the **Select a source** pull-down menu.

13. Click **Save**.

14. Click **Save Query**.

15. Double-click the **Pattern** icon.

   The Pattern configuration screen displays.
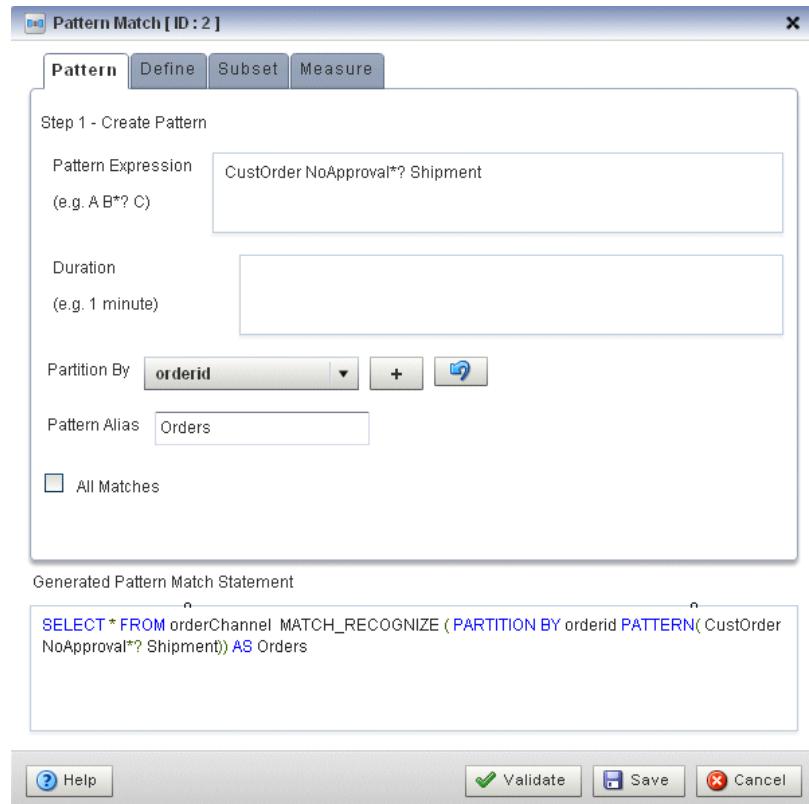
Using the Pattern tab, you will define the pattern expression that matches when missed events occur. The expression is made in terms of named conditions that you will specify on the Define tab in a later step.

16. Enter the following expression in the Pattern Expression field:

```
CustOrder NoApproval*? Shipment
```

This pattern uses the Oracle CQL pattern quantifiers that Table 6-3 lists. Use the pattern quantifiers to specify the allowed range of pattern matches. The one-character pattern quantifiers are maximal (greedy). They attempt to match the biggest quantity first. The two-character pattern quantifiers are minimal (reluctant). They attempt to match the smallest quantity first.

Table 6-3   MATCH_RECOGNIZE Pattern Quantifiers

Maximal	Minimal	Description
*	*?	0 or more times
+	+?	1 or more times.
?	??	0 or 1 time.

For more information, see *Oracle Fusion Middleware Oracle CQL Language Reference for Oracle Event Processing*.

17. Select **orderid** from the **Partition By** pull-down menu and click the Plus Sign button to add this property to the PARTITION BY clause.

This ensures that Oracle Event Processing evaluates the missing event query on each order.

18. Enter **Orders** in the **Alias** field.

This assigns an alias (Orders) for the pattern to simplify its use later in the query.

19. Click the **Define** tab.

The Define tab displays.
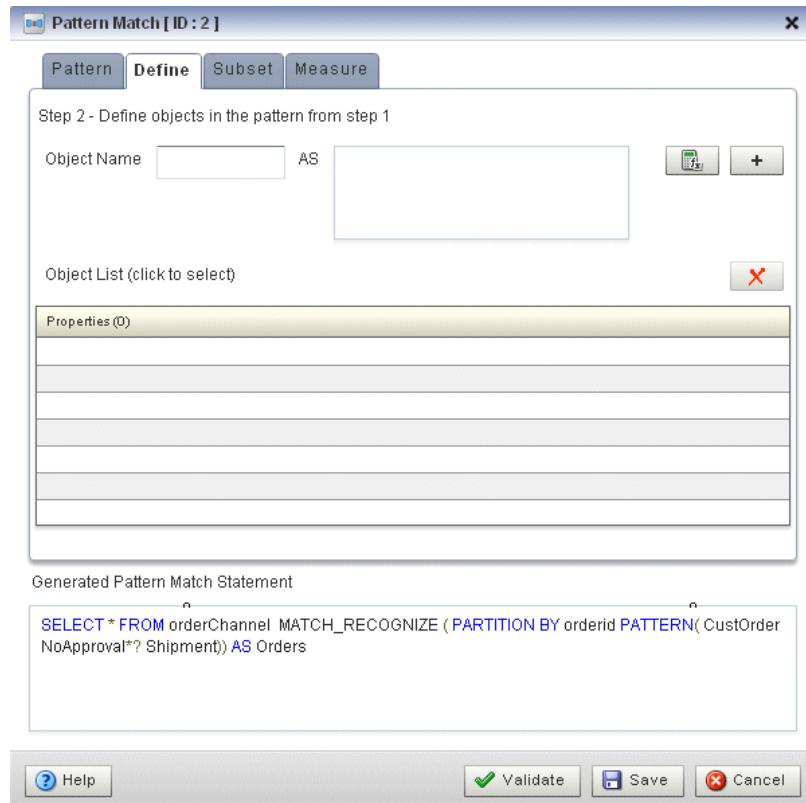
You will now define each of the conditions named in the pattern clause as lists:

Table 6-4   Condition Definitions

Condition Name	Definition
CustOrder	orderChannel.eventType = 'C'
NoApproval	NOT(orderChannel.eventType = 'A')
Shipment	orderChannel.eventType = 'C'

20. Enter **CustOrder** in the **Object Name** field.

21. Click the Expression Builder button and configure the Expression Builder as follows:

- In the **Variables** list, double-click **eventType**.

- In the **Operands** list, double-click **=**.

- After the = operand, enter the value `'C'`.
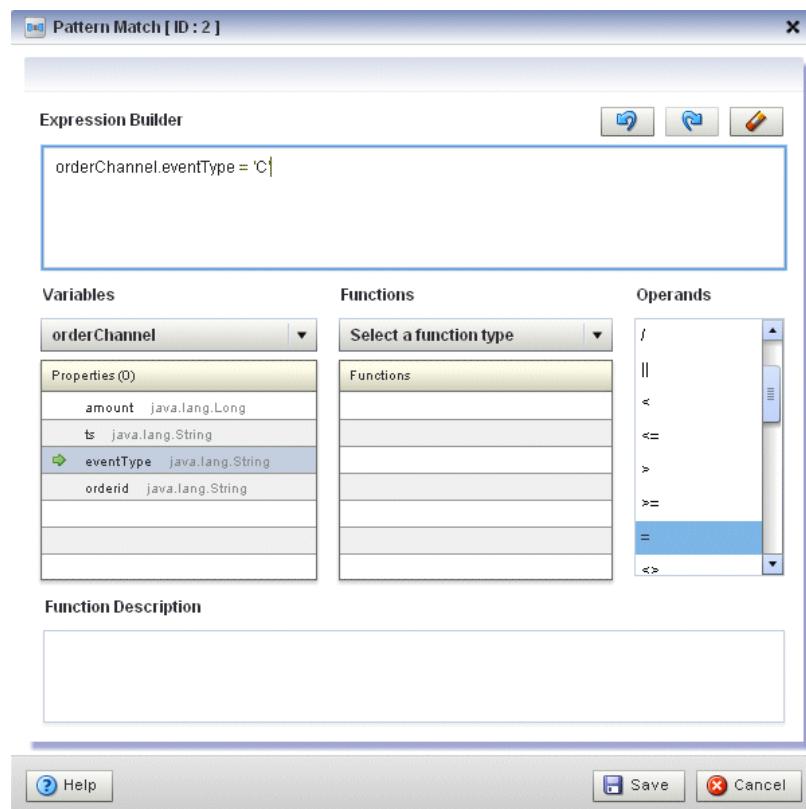
22. Click **Save**.

23. Click the Plus Sign button.

The condition definition is added to the Object List as follows:
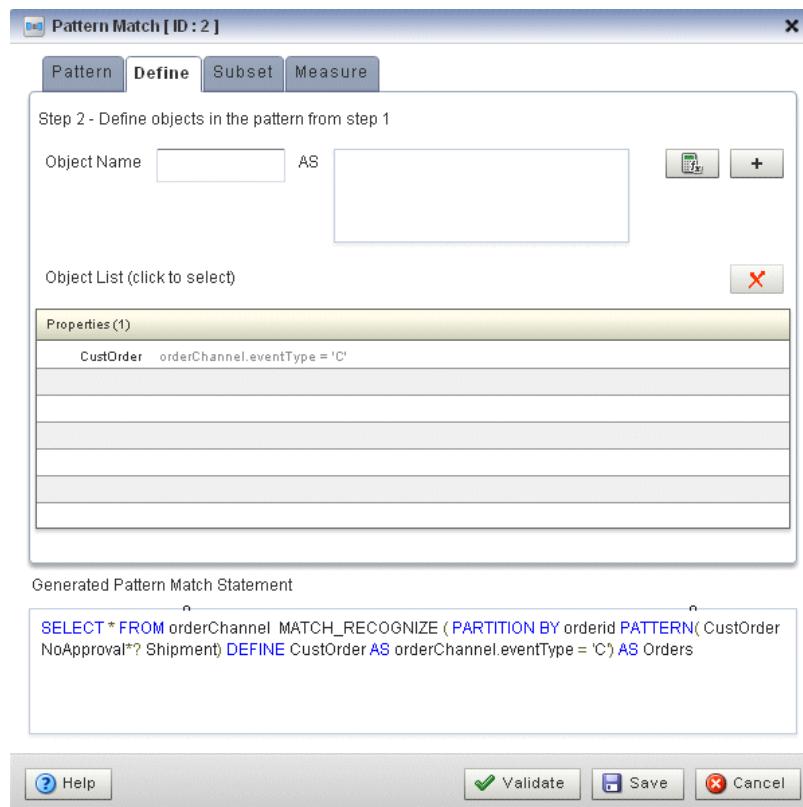
24. Enter **NoApproval** in the **Object Name** field.

25. Click the Expression Builder button and configure the Expression Builder:

   - In the **Variables** list, double-click **eventType**.

   - In the **Operands** list, double-click **=**.

   - After the = operand, enter the value `'A'`.

   - Place parenthesis around the expression.

   - Place the insertion bar at the beginning of the expression, outside the open parenthesis.

   - In the **Operands** list, double-click **NOT**.
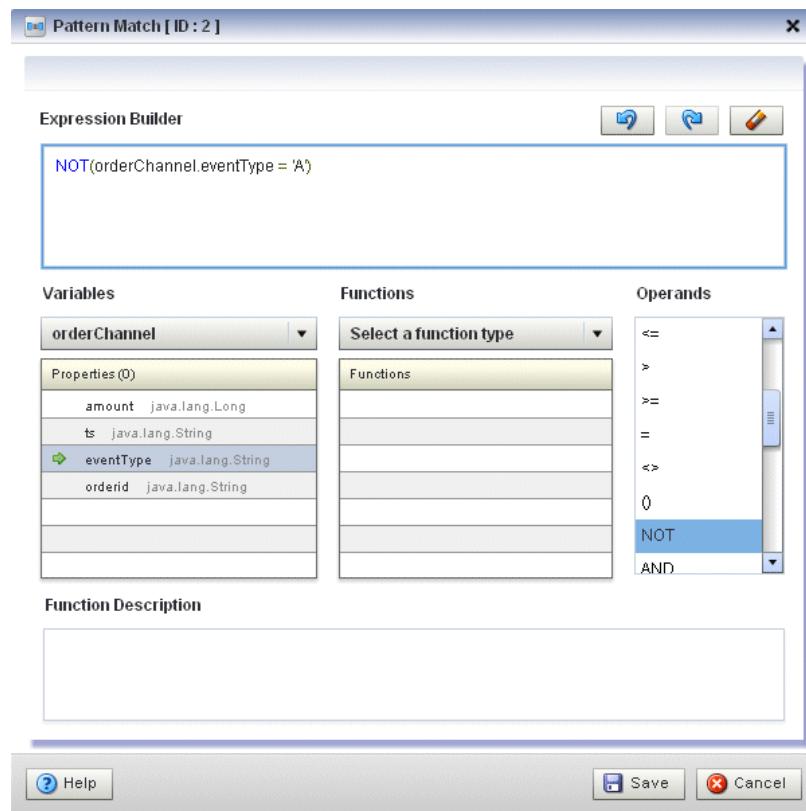
26. Click **Save**.

27. Click the Plus Sign button.

    The condition definition is added to the Object List.

28. Enter **Shipment** in the **Object Name** field.

29. Click the Expression Builder button and configure the Expression Builder :

    - In the **Variables** list, double-click **eventType**.

    - In the **Operands** list, double-click **=**.

    - After the = operand, enter the value `'S'`.
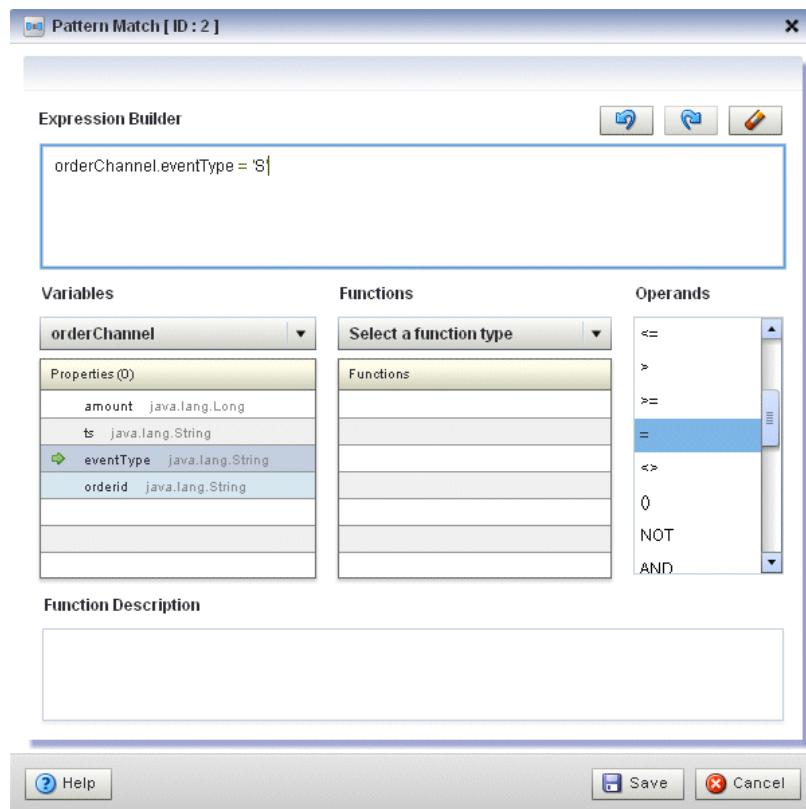
30. Click **Save**.

31. Click the Plus Sign button.

    The Define tab displays.
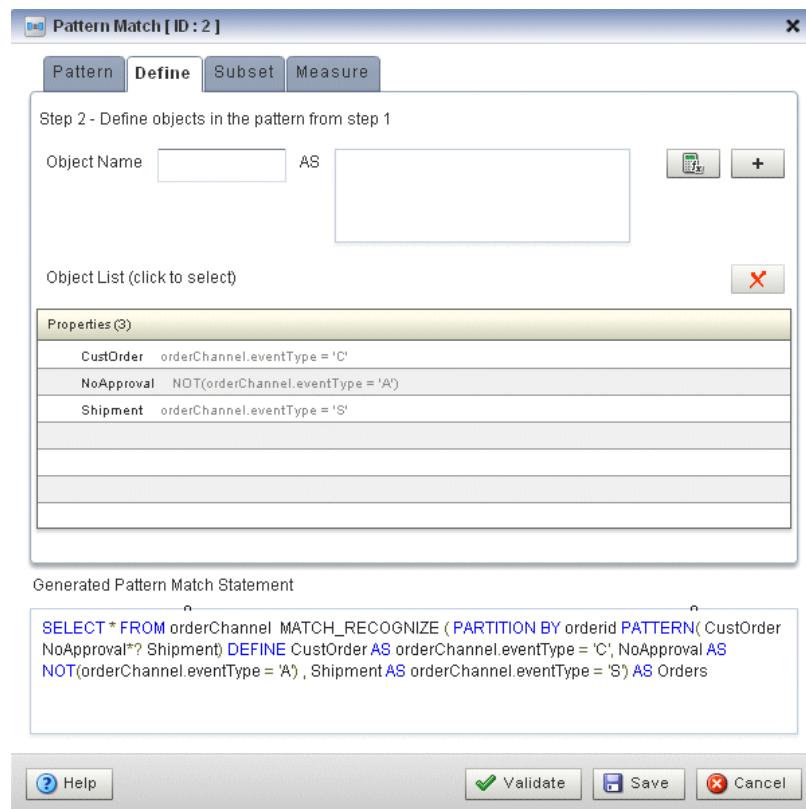
32. Click the **Measure** tab.

The Measure tab displays.
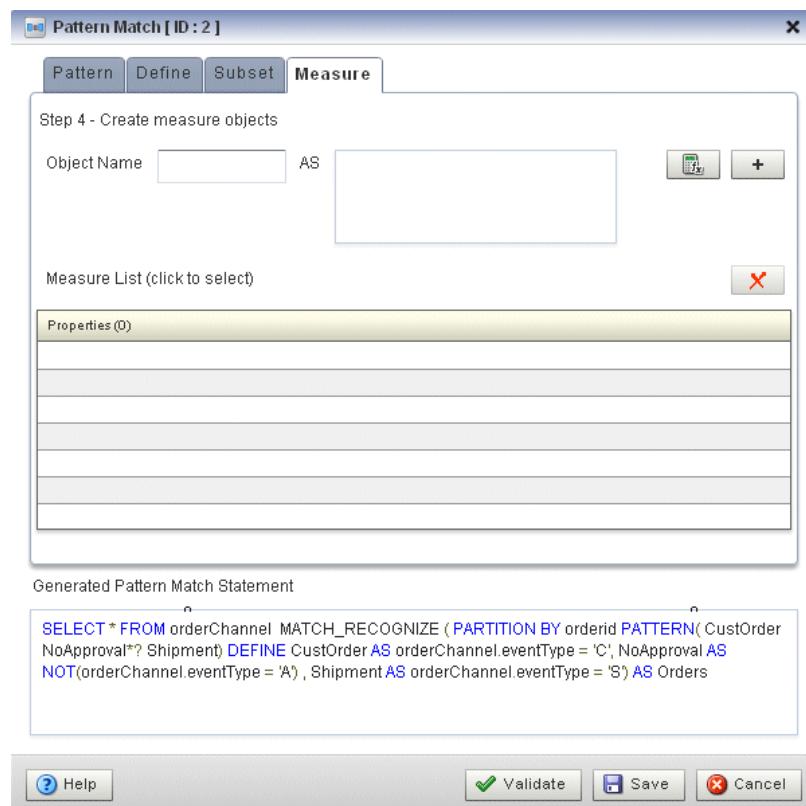
Use the Measure tab to define expressions in a `MATCH_RECOGNIZE` condition and to bind stream elements that match conditions in the `DEFINE` clause to arguments that you can include in the select statement of a query.

Use the Measure tab to specify the following:

- `CustOrder.orderid AS orderid`

- `CustOrder.amount AS amount`

For more information, see Oracle Fusion Middleware Oracle CQL Language Reference for Oracle Event Processing.

33. Enter **orderid** in the **Object Name** field.

34. Click the Expression Builder button and configure the Expression Builder:

- In the **Variables** list, double-click **CustOrder.orderid**.
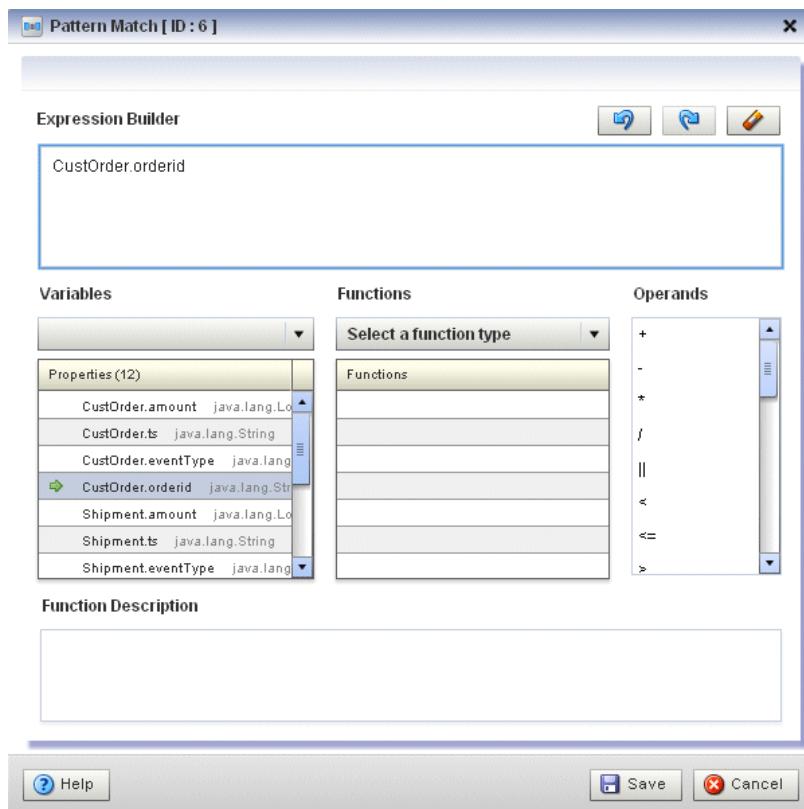
35. Click **Save**.

36. Click the Plus Sign button.

37. Enter **amount** in the **Object Name** field.

38. Click the Expression Builder button and configure the Expression Builder:

- In the **Variables** list, double-click **CustOrder.amount**.
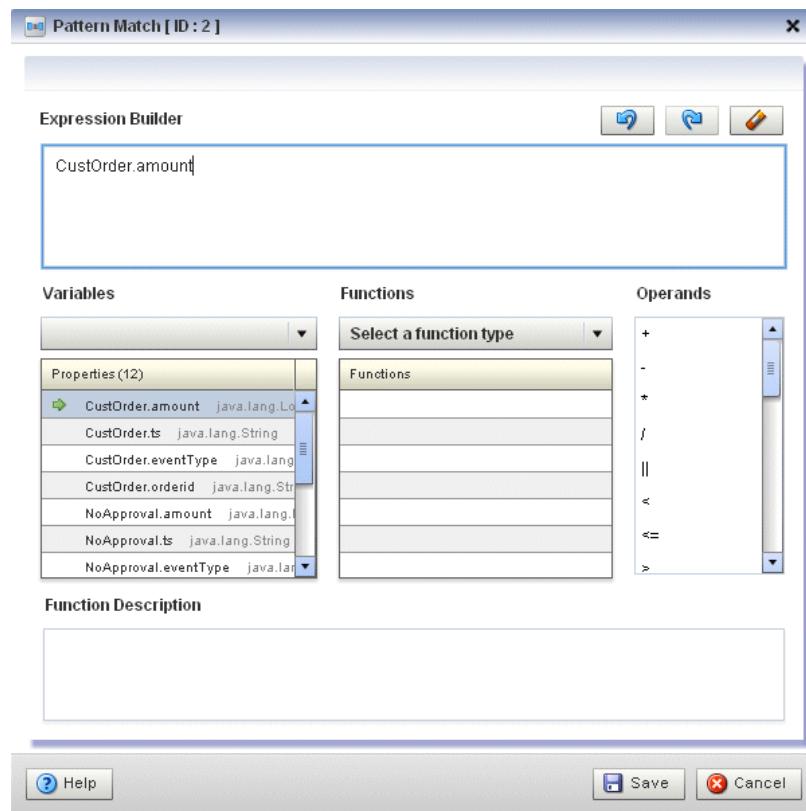
39. Click **Save**.

40. Click the Plus Sign button.

The Measure tab displays.
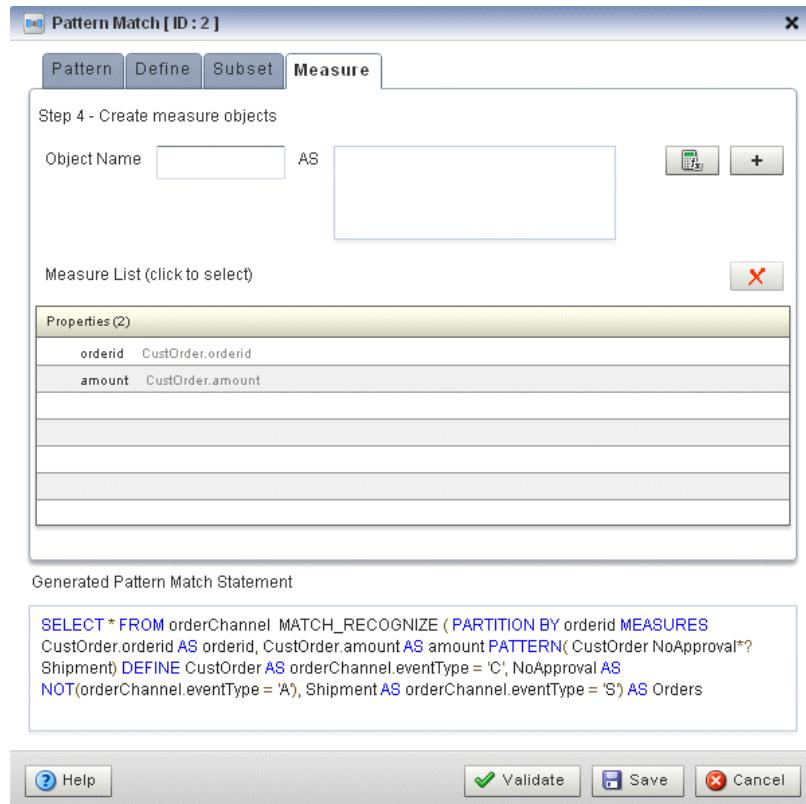
41. Click **Save**.

42. Double-click the **Select** icon.

The Select configuration screen appears as follows:
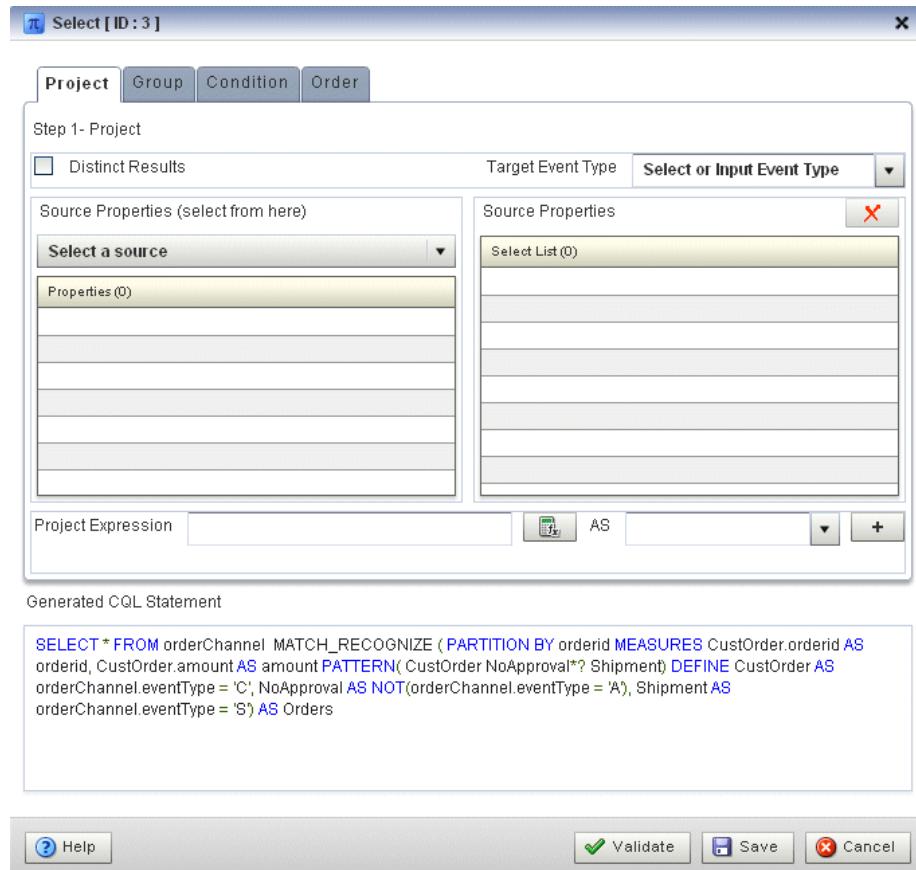
43. Configure the Project tab as follows:

   • Select **AlertEvent** from the **Select or Input Event Type** pull-down menu.

   • Select **Orders** from the **Select a source** pull-down menu.

44. Double-click **orderid** in the **Properties** list and select **orderid** from the **Select or Input Alias** pull-down menu.

45. Click the Plus Sign button to add the property to the Generated CQL Statement.

46. Double-click **amount** in the **Properties** list and select **amount** from the **Select or Input Alias** pull-down menu.

47. Click the Plus Sign button to add the property to the Generated CQL Statement.

48. Click in the Project Expression field and enter the value `"Error - Missing Approval"` and select **alertType** from the **Select or Input Alias** pull-down menu.

49. Click the Plus Sign button to add the property to the Generated CQL Statement.

   The Project tab displays.

50. Click **Save**.

51. Click **Save Query**.

52. Double-click the **Output** icon.

The Output configuration screen displays.
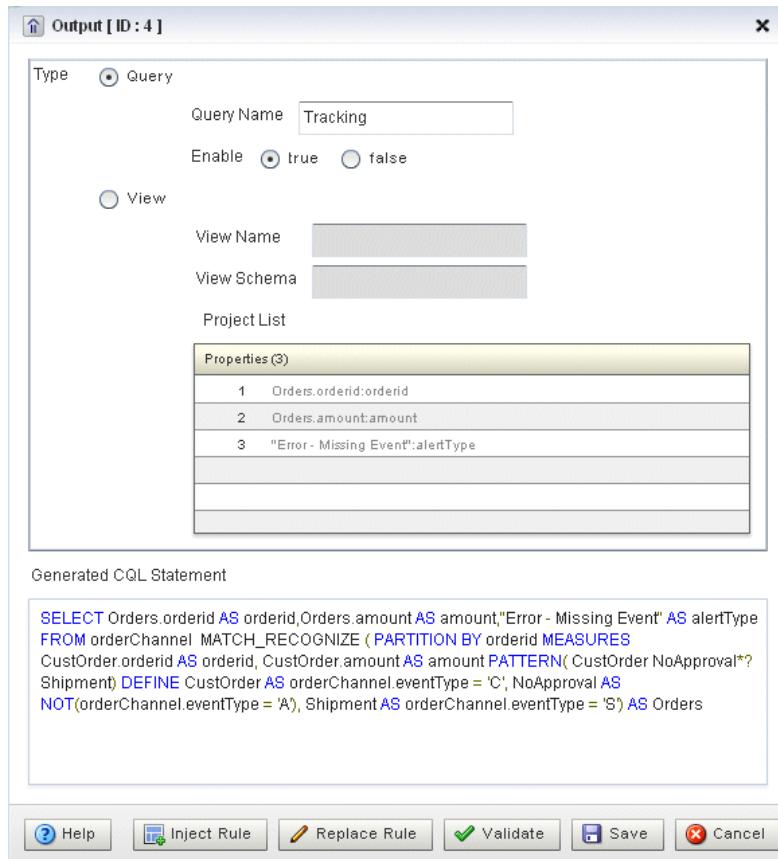
53. Configure the Output as follows:

   • Select **Query**.

   • Enter **Tracking** as the **Query Name**.

54. Click **Inject Rule**.

   The Inject Rule Confirmation dialog displays.

55. Click **OK**.

   The Query Wizard adds the rule to the cqlProc processor.

56. Click **Save**.

57. Click the **CQL Rules** tab.

   The CQL Rules tab displays.

58. Click the **Query** radio button.

   Confirm that your Tracking query is present.
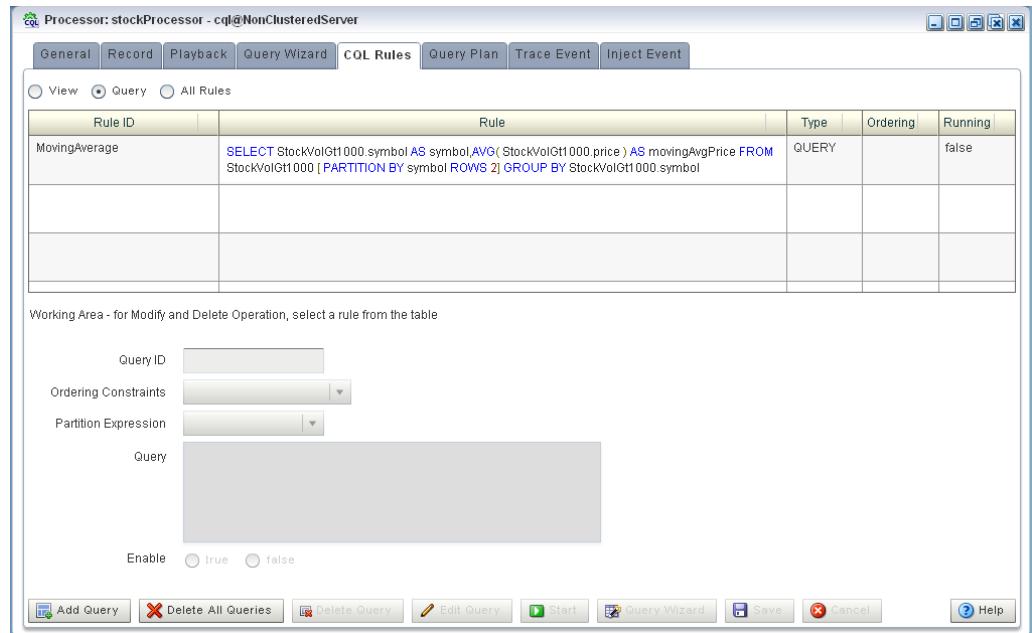
Test the missing event query:

1. To simulate the data feed, change to the */Oracle/Middleware/my_oep/ utils/load-generator* directory.

2. Run the load generator using the `orderData.prop` properties file:

   a. On Windows:

   ```
   prompt> runloadgen.cmd orderData.prop
   ```

   b. On UNIX:

   ```
   prompt> ./runloadgen.sh orderData.prop
   ```

3. In the Oracle Event Processing Visualizer, click the **ViewStream** button in the top panel.

   The Stream Visualizer screen displays.
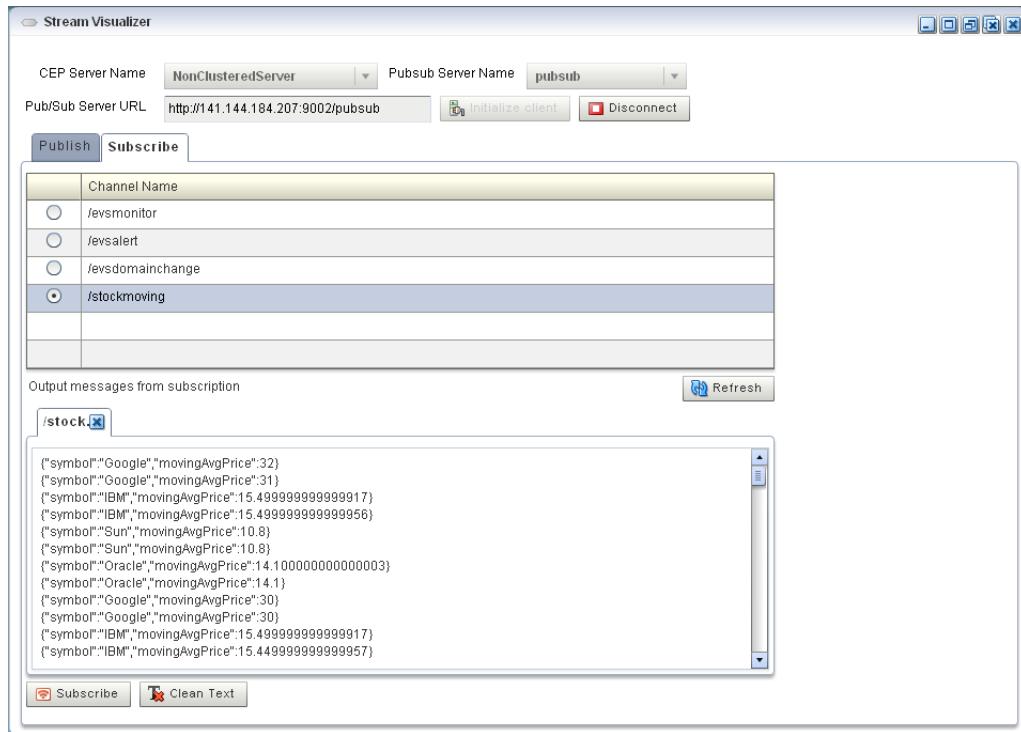
4. Click **Initialize Client**.

5. Click the **Subscribe** tab.

6. Select the **orderalert** radio button.

7. Click **Subscribe**.

   As missing events are detected, the Oracle Event Processing updates the **Received Messages** area showing the `AlertEvents` generated.

6.6.4.2 Create the Moving Average Query

This section describes how to use the Oracle Event Processing Visualizer Query Wizard to create the Oracle CQL moving average query that the `stockProc` processor executes.

You do this in two steps:

- First, you create a view (the Oracle CQL equivalent of a subquery) that serves as the source of the moving average query.

  See "Create a view source for the moving average query:".

- Second, you create the moving average query using the source view.

  See "Create the moving average query using the view source:".

- Finally, you test the moving average query.

  See "Test the moving average query:".

Create a view source for the moving average query:

1. If the CQL Oracle Event Processing instance is not already running, follow the procedure in Run the CQL Example to start the server.

You must have a running server to use the Oracle Event Processing Visualizer.

2. Invoke the following URL in your browser:

```
http://host:port/wlevs
```

where *host* refers to the name of the computer on which Oracle Event Processing is running and *port* refers to the Jetty NetIO port configured for the server (default value 9002).

The Logon screen displays.

3. In the Logon screen, enter the **Username** oepadmin and **Password** welcome1, and click **Login**.

The Oracle Event Processing Visualizer dashboard displays.

For more information about the Oracle Event Processing Visualizer user interface, see *Oracle Fusion Middleware Using Visualizer for Oracle Event Processing*.

4. In the right panel, expand **WLEventServerDomain > NonClusteredServer > Applications**.

5. Select the **cql** node.

The CQL application screen displays.

6. Select the **Event Processing Network** tab.

The Event Processing Network screen displays.

7. Double-click the **stockProcessor** Oracle CQL processor icon.

The Oracle CQL processor screen appears as Figure 6-3 shows.

Figure 6-3   Oracle CQL Processor: General Tab
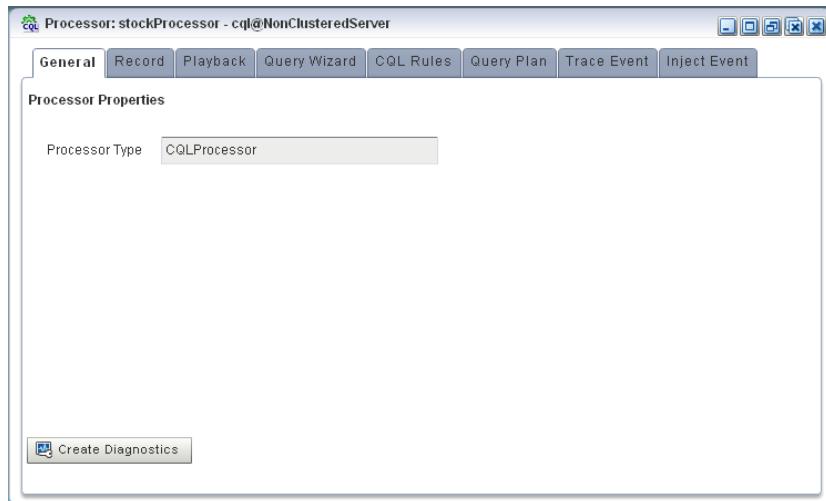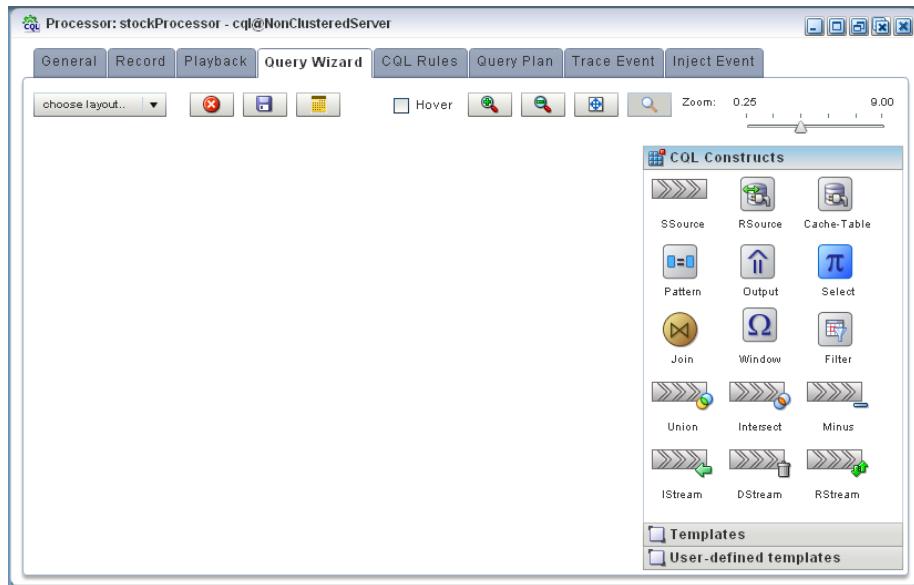
8. Select the **Query Wizard** tab.

The Query Wizard screen displays.

You can use the Oracle CQL Query Wizard to construct an Oracle CQL query from a template or from individual Oracle CQL constructs.

In this procedure, you are going to create an Oracle CQL view and query from individual Oracle CQL constructs.

For more information, see *Oracle Fusion Middleware Using Visualizer for Oracle Event Processing*.

9. Click and drag an SSource icon (Stream Source) from the CQL Constructs palette and drop it anywhere in the Query Wizard canvas as.
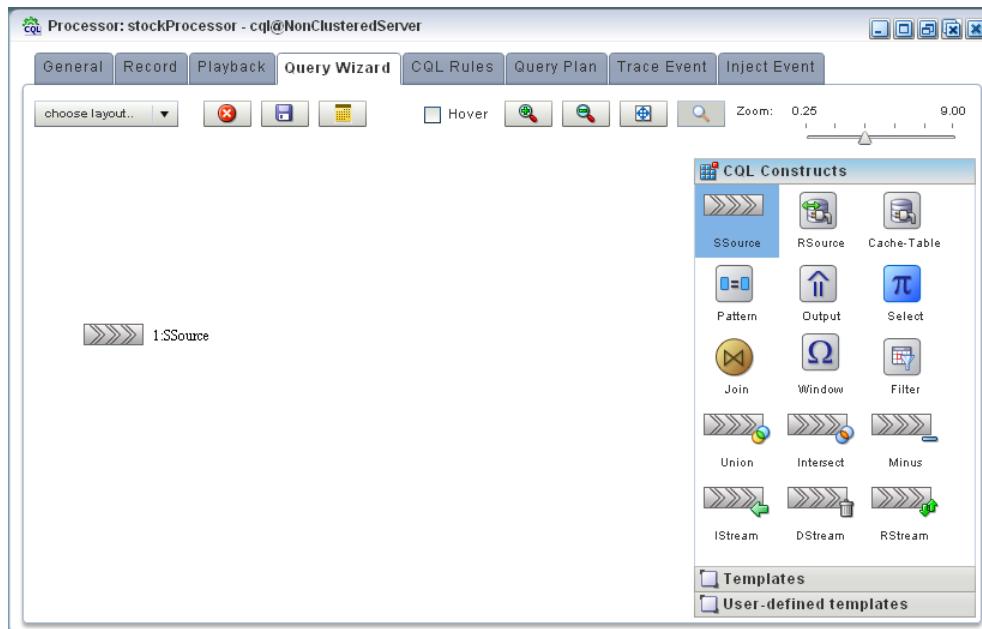
10. Double-click the **SSource** icon.

The SSource configuration screen appears.

The source of your view will be the `stockChannel` stream. You want to select stock events from this stream where the volume is greater than 1000. This will be the source for your moving average query.

11. Configure the SSource as follows:

- Select **Stream** as the Type.

  The source of your view is the `stockChannel` stream.

- Select **stockChannel** from the **Select a source** pull-down menu.

- Enter the alias `StockVolGt1000` in the **AS** field.
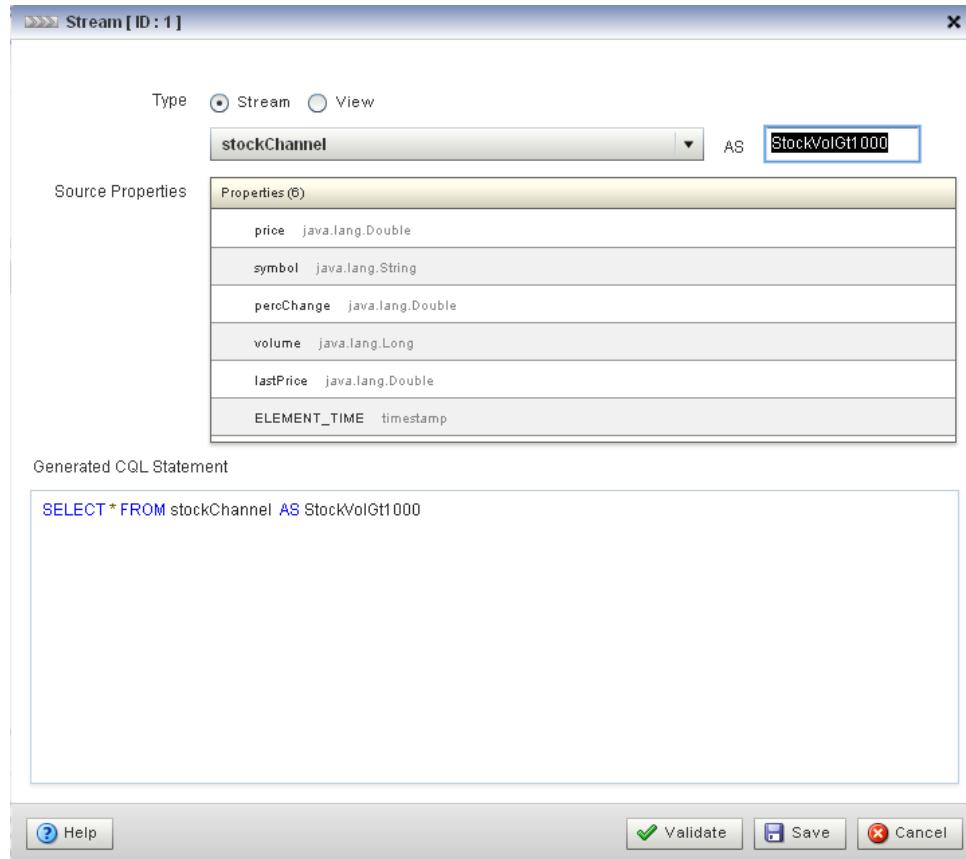
12. Click **Save**.

13. Click **Save Query**.

14. When prompted, enter **StockVolGt1000** in the **Query Id** field.

15. Click **Save**.

Next, you will add an Oracle CQL filter.

16. Click and drag a Filter icon from the CQL Constructs palette and drop it anywhere in the Query Wizard canvas as follows:
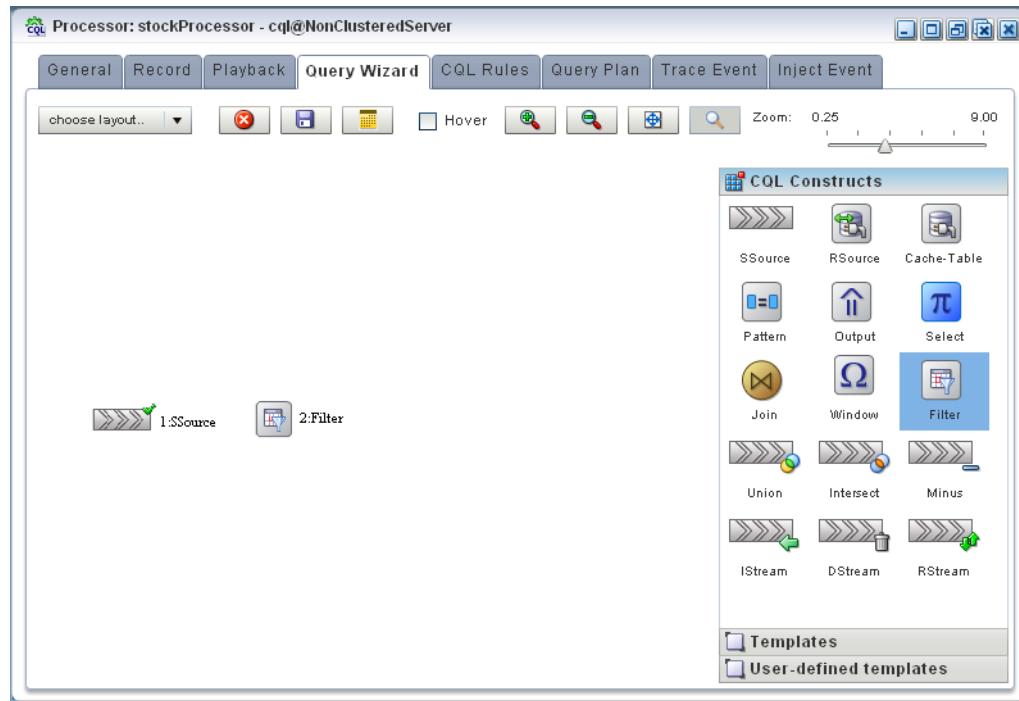
17. Click the SSource icon and drag to the Window icon to connect the Oracle CQL constructs as follows:

18. Double-click the **Filter** icon.

The Filter configuration screen displays:

19. Click the Expression Builder button.

The Expression Builder dialog appears.

20. Configure the Expression Builder as follows:

- Select **StockVolGt100** from the **Select an Event Type** pull-down menu to define the variables you can use in this expression.

- Double-click the **volume** variable to add it to the Expression Builder field.

- Double-click **>** in the **Operands** list to add it to the Expression Builder field.

- Enter the value 1000 after the > operand.
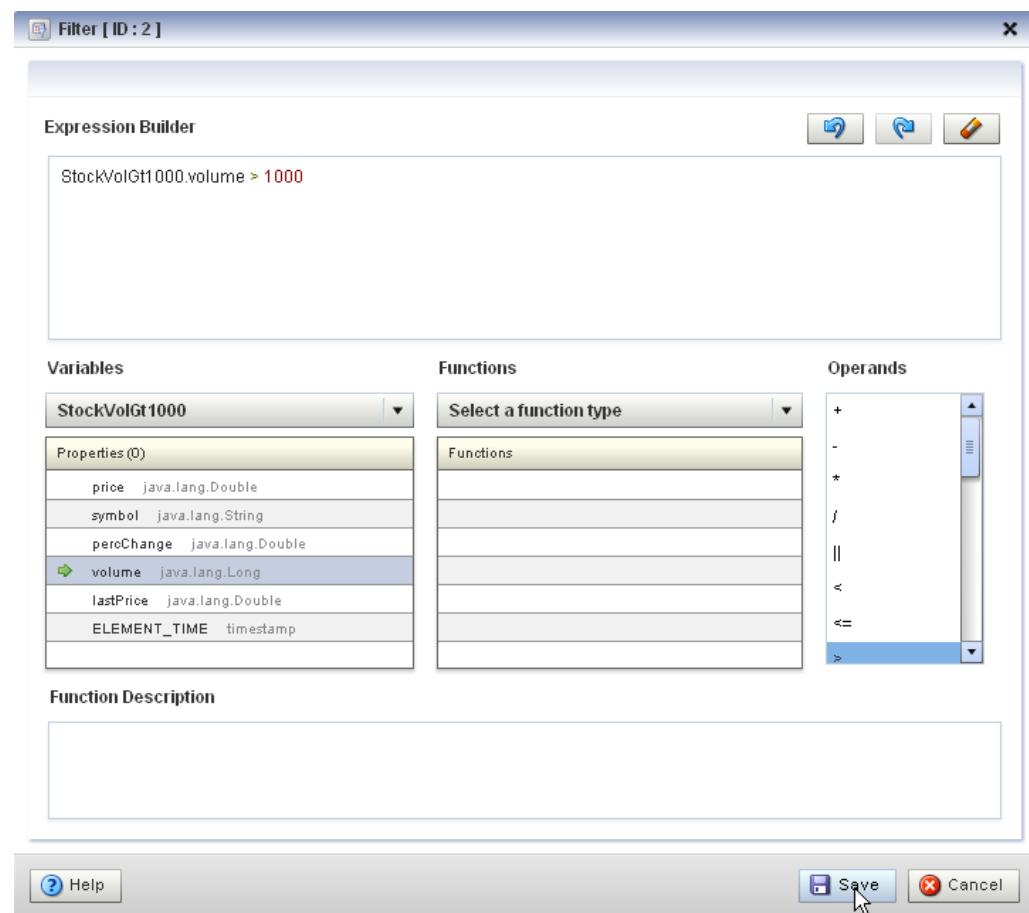
21. Click **Save**.

22. Click **Add Filter**.

    The Query Wizard adds the expression to the Generated CQL Statement as
    follows:

23. Click **Save**.

24. Click **Save Query**.

Next you want to add a select statement.

25. Click and drag a Select icon from the CQL Constructs palette and drop it anywhere in the Query Wizard canvas as follows:
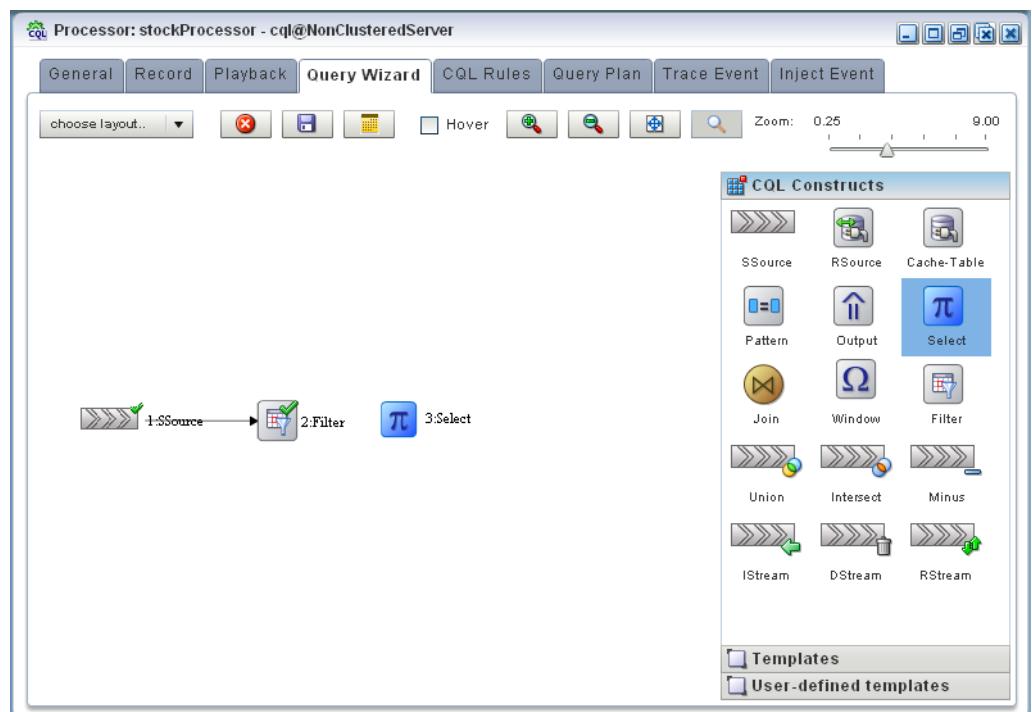
26. Click the **Filter** icon and drag to the **Select** icon to connect the Oracle CQL constructs.

27. Double-click the **Select** icon.

The Select configuration screen appears.

You want to select `price`, `symbol,` and `volume` from your `StockVolGt1000` stream.

28. Configure the Select as follows:

- Select **StockVolGt1000** from the **Select a source** pull-down menu.

- Select the **price** property and click the Plus Sign button.

   The Query Wizard adds the property to Generated CQL Statement

- Repeat for the **symbol** and **volume** properties.

The Select configuration dialog displays.
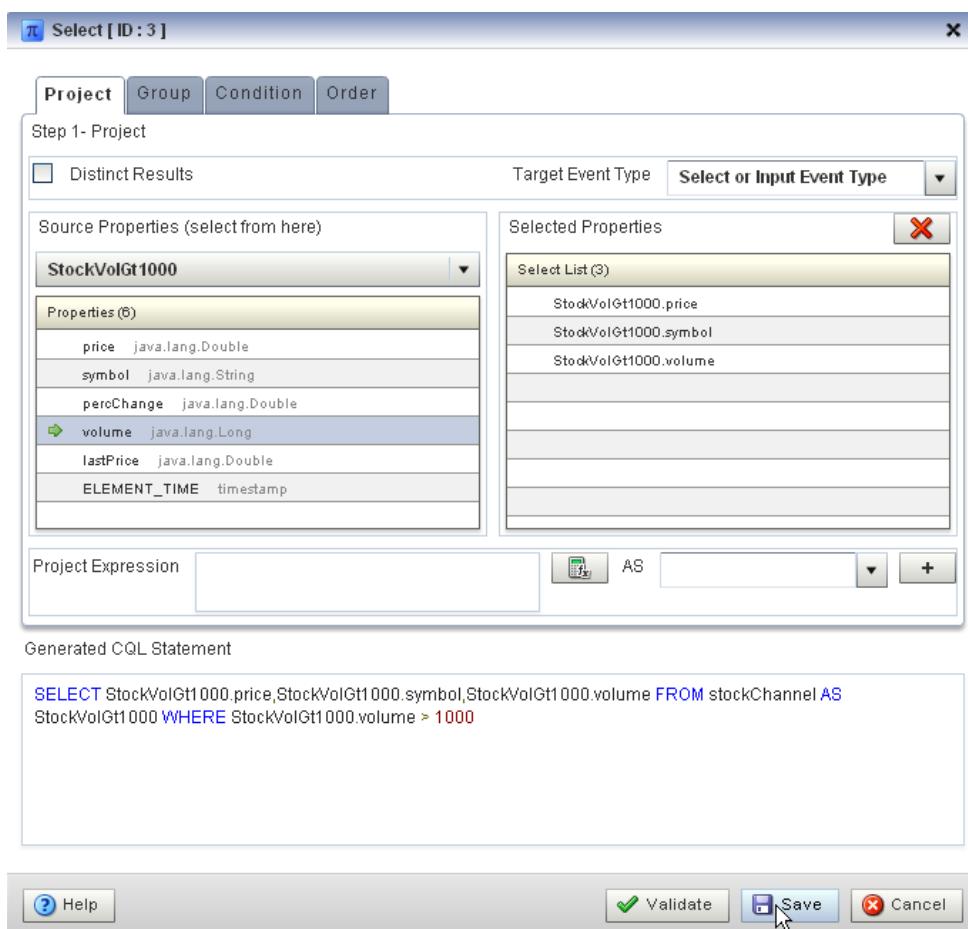
29. Click **Save**.

30. Click Save **Query**.

Finally, you will add an Output.

31. Click and drag an Output icon from the CQL Constructs palette and drop it anywhere in the Query Wizard canvas as follows:
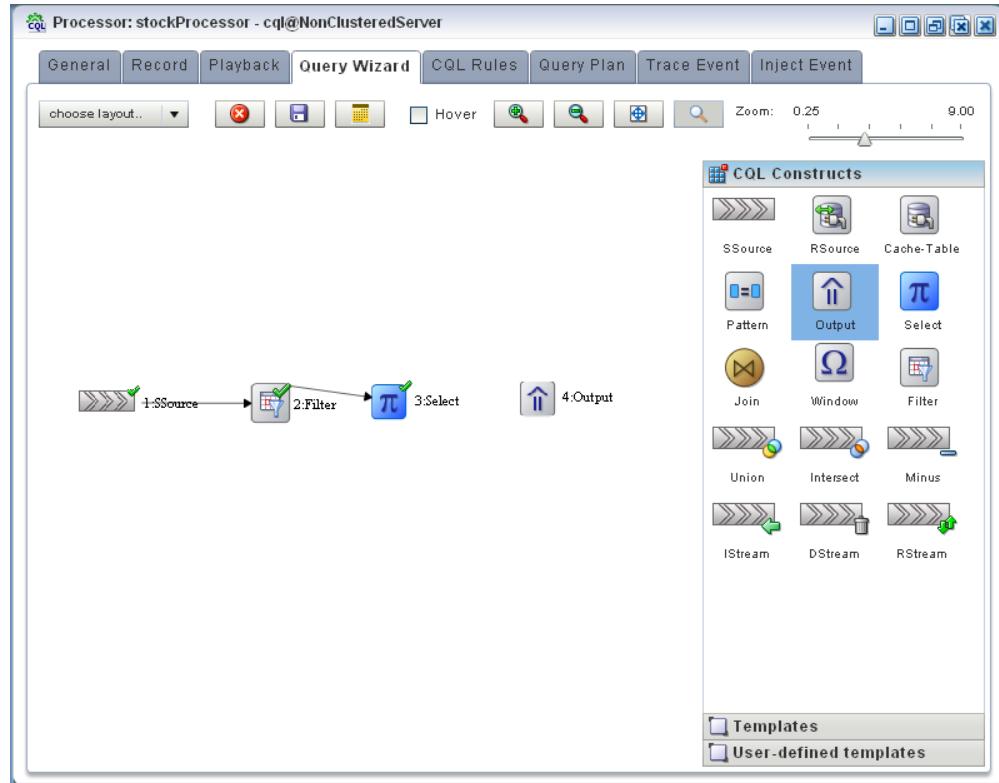
32. Click the **Select** icon and drag to the **Output** icon to connect the Oracle CQL constructs.

33. Double-click the **Output** icon.

The Output configuration screen appears.

34. Configure the Output as follows:

- Select **View**.

- Configure **View Name** as `StockVolGt1000`.

- Delete the contents of the **View Schema** field.

  You can let the Oracle Event Processing server define the view schema for you.
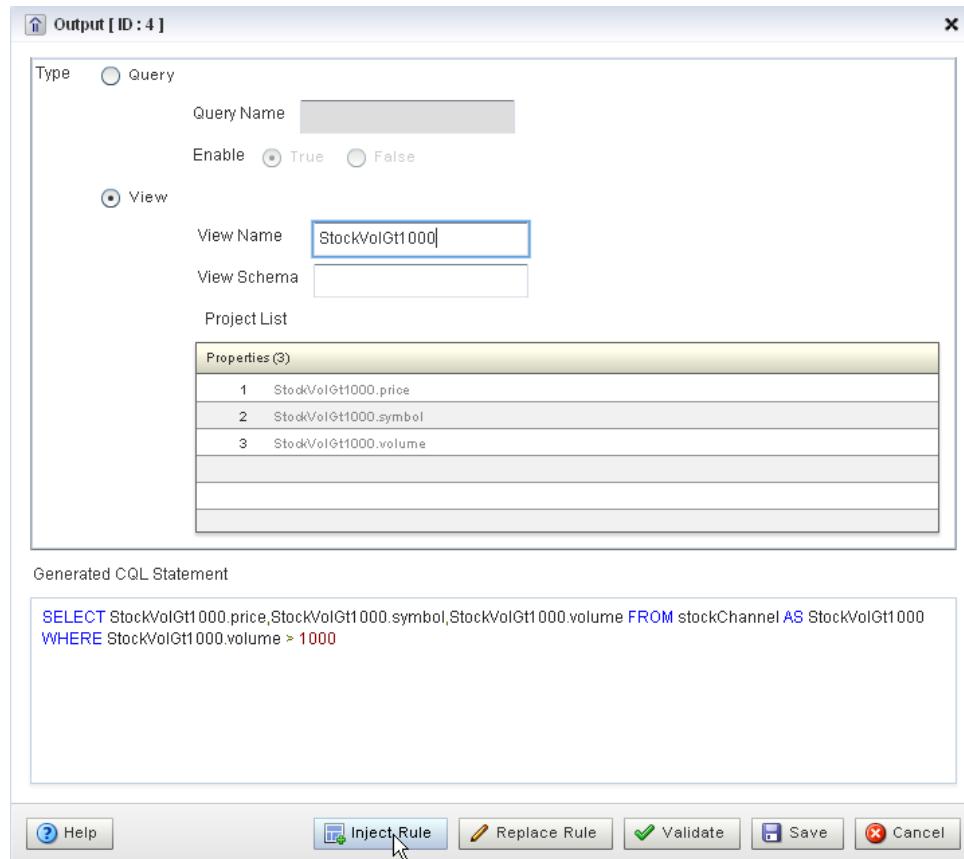
35. Click **Inject Rule**.

The Inject Rule Confirmation dialog appears as follows:

36. Click **OK**.

The Query Wizard adds the rule to the `cqlProc` processor.

37. Click **Save**.

38. Click the **CQL Rules** tab.

The CQL Rules tab displays.

39. Click the **View** radio button.

Confirm that your `StockVolGt1000` view is present.
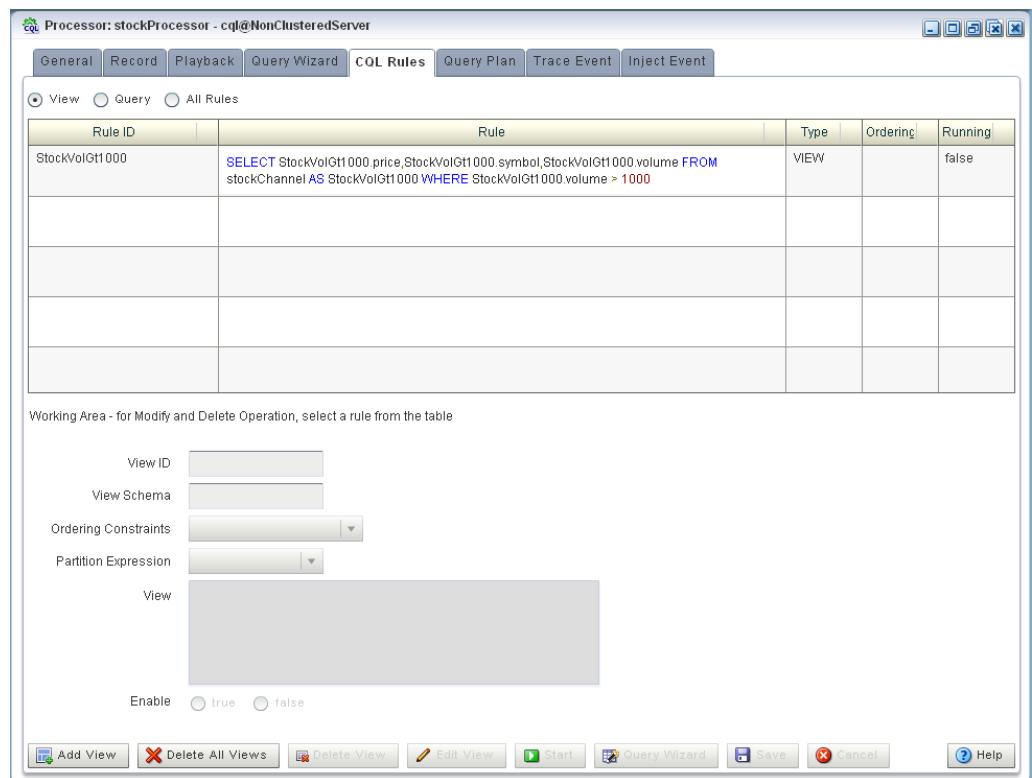
Create the moving average query using the view source:

1. If the CQL Oracle Event Processing instance is not already running, follow the procedure in Run the CQL Example to start the server.

   You must have a running server to use the Oracle Event Processing Visualizer.

2. Invoke the following URL in your browser:

   `http://host:port/wlevs`

   where *host* refers to the name of the computer on which Oracle Event Processing is running and *port* refers to the Jetty NetIO port configured for the server (default value `9002`).

   The Logon screen displays.

3. In the Logon screen, enter the **Username** `oepadmin` and **Password** `welcome1`, and click **Login**.

    The Oracle Event Processing Visualizer dashboard displays.

    For more information about the Oracle Event Processing Visualizer user interface, see *Oracle Fusion Middleware Using Visualizer for Oracle Event Processing*.

4. In the left panel, expand **WLEventServerDomain > NonClusteredServer > Applications**.

5. Select the **cql** stage.

The CQL application screen displays.
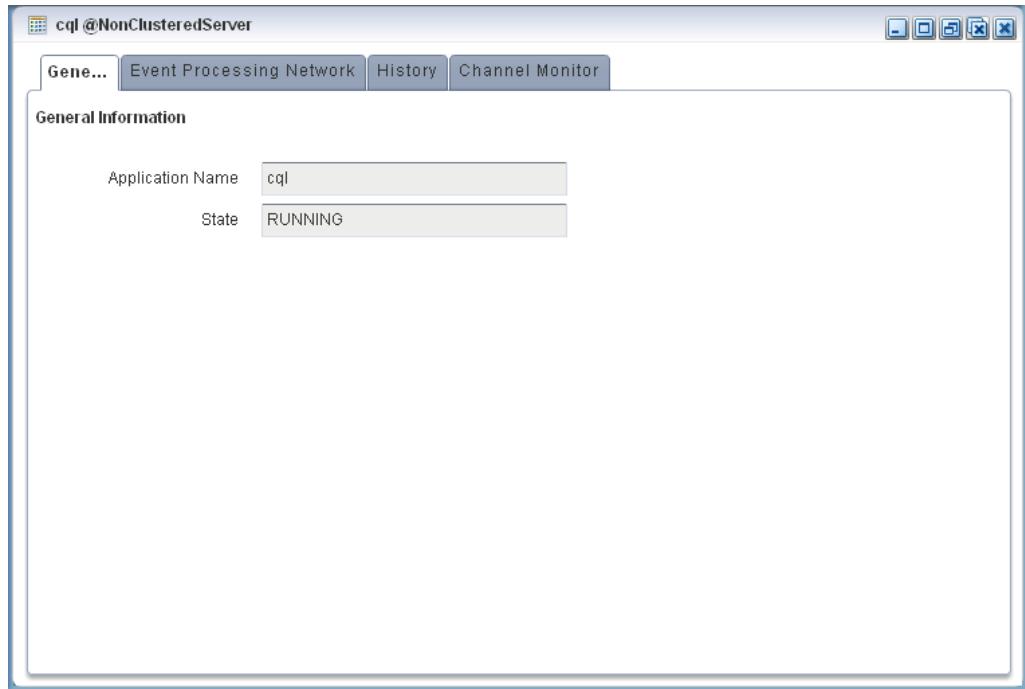
6. Select the **Event Processing Network** tab.

The Event Processing Network screen displays.
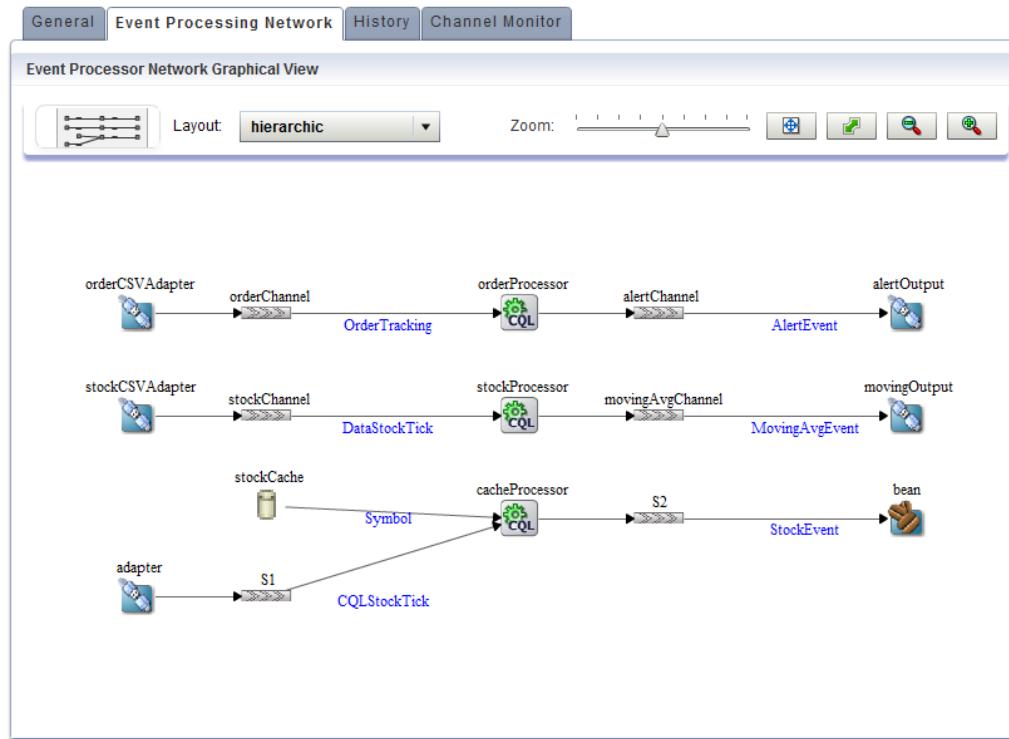
7. Double-click the **stockProcessor** Oracle CQL processor icon.

   The Oracle CQL processor screen displays.
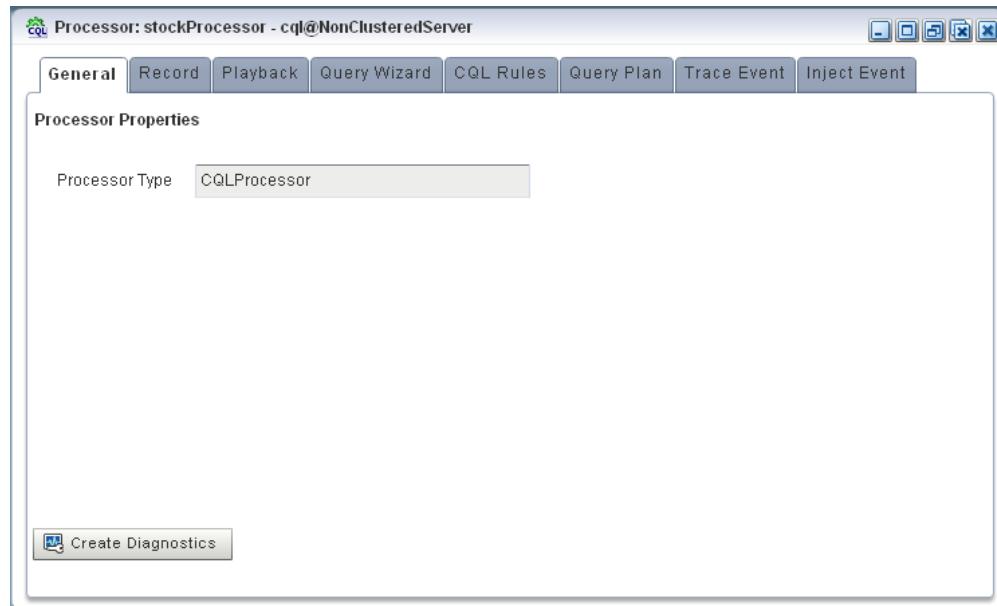
8. Select the **Query Wizard** tab.

   The Query Wizard screen displays. If you have been recently creating or editing queries for this Oracle CQL processor, you might see those queries on the Query Wizard canvas. Otherwise, the canvas will be blank.
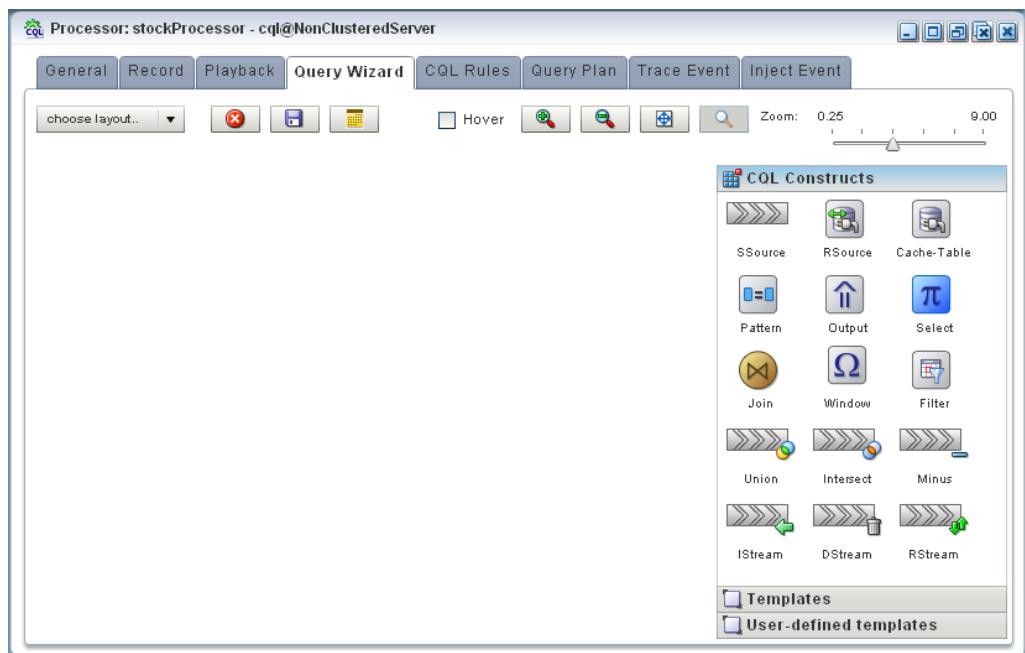
You can use the Oracle CQL Query Wizard to construct an Oracle CQL query from a template or from individual Oracle CQL constructs.

In this procedure, you are going to create an Oracle CQL view and query from individual Oracle CQL constructs.

For more information, see *Oracle Fusion Middleware Using Visualizer for Oracle Event Processing*.

9. Click and drag an SSource icon (Stream Source) from the CQL Constructs palette and drop it anywhere in the Query Wizard canvas as follows:
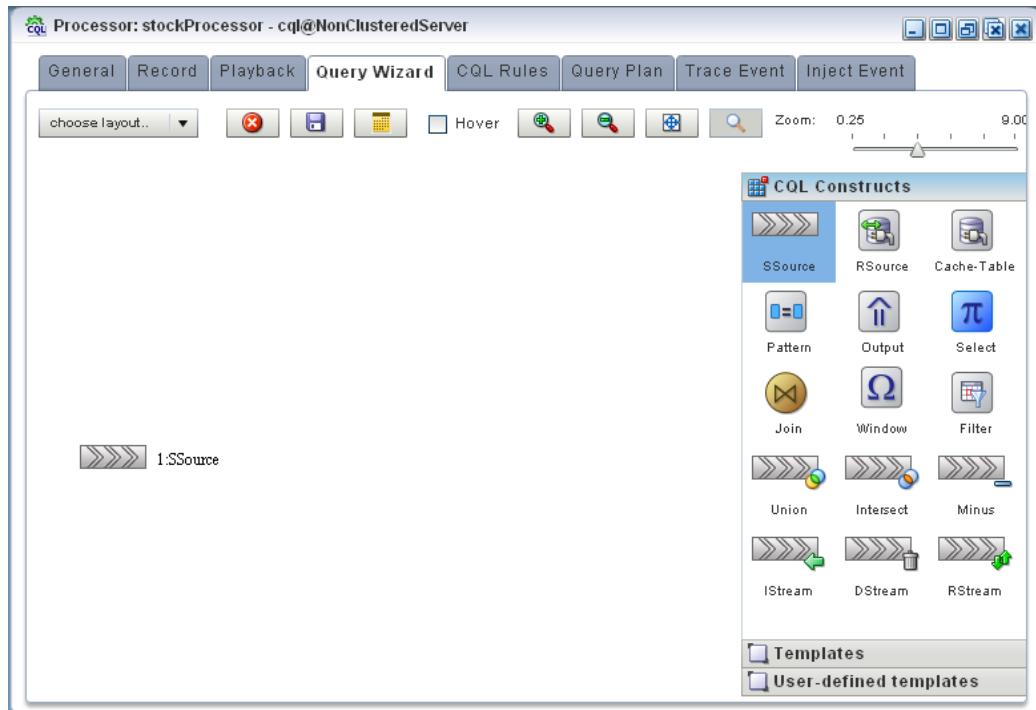
10. Double-click the **SSource** icon.

The SSource configuration screen appears.

11. Configure the SSource dialog as follows:

- Select **View** as the **Type**.

- Select the **StockVolGt1000** view from the **Select a source** pull-down menu.
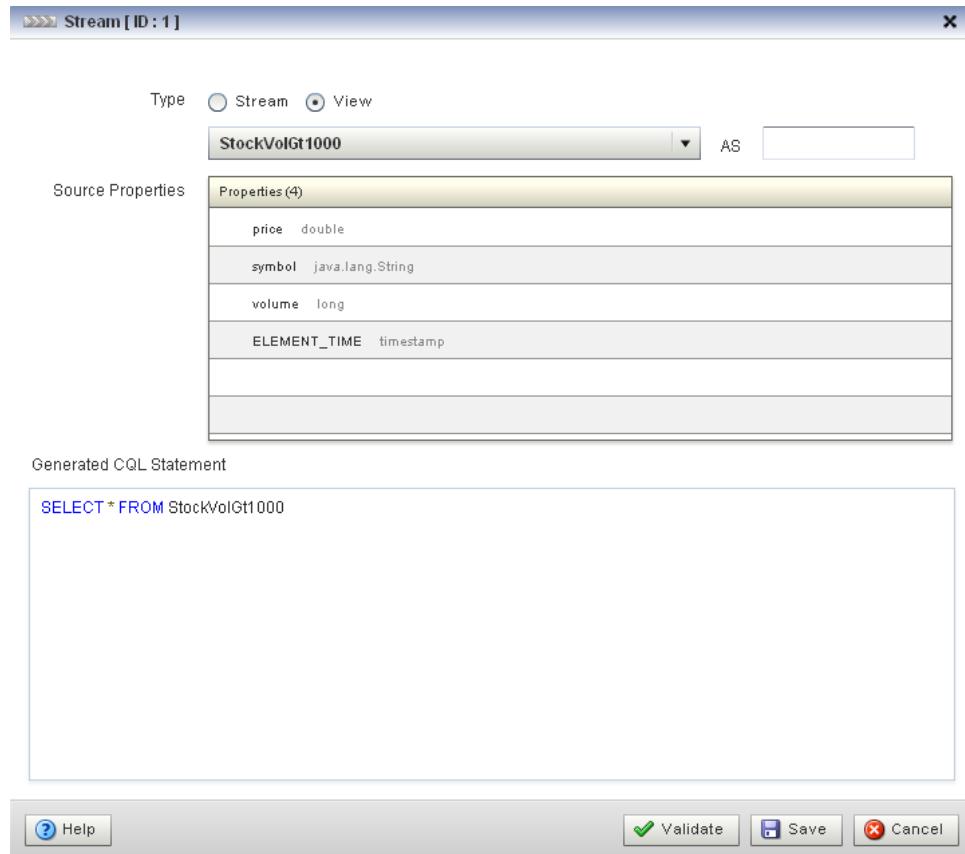
12. Click **Save**.

13. Click **Save Query**.

14. Click and drag a Window icon from the CQL Constructs palette and drop it anywhere in the Query Wizard canvas as follows:
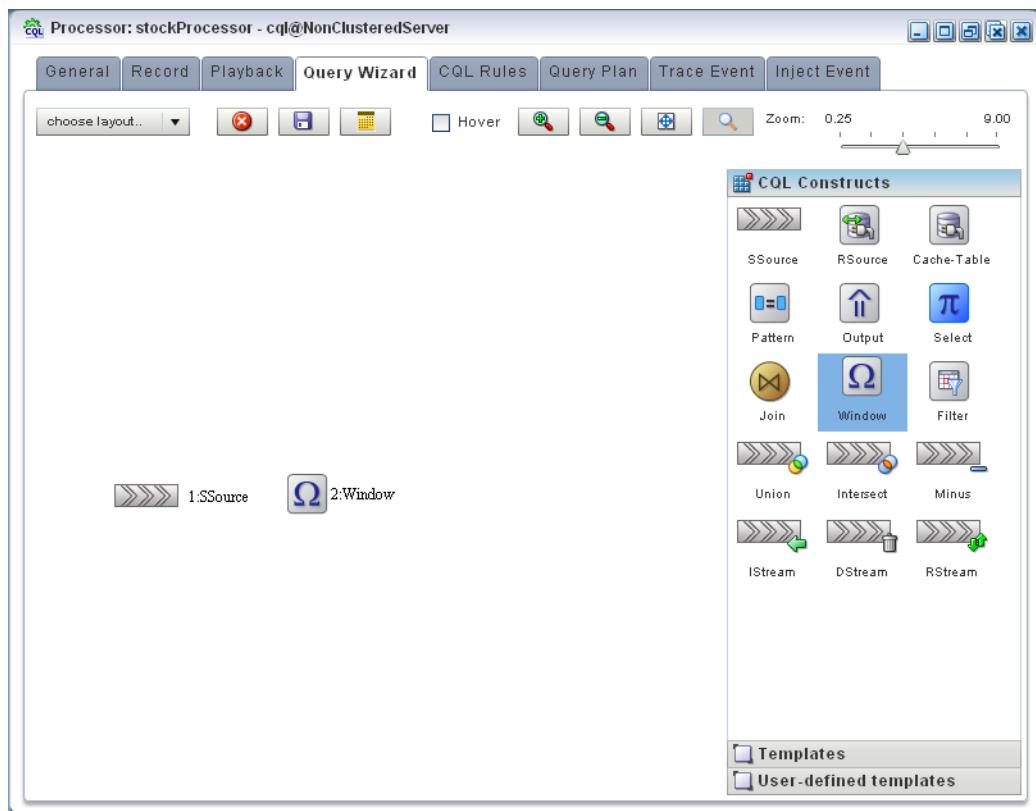
15. Click the **SSource** icon and drag to the **Window** icon to connect the Oracle CQL constructs.

16. Double-click the **Window** icon.

    The SSource configuration screen appears.

    You want to create a sliding window over the last 2 events, partitioned by `symbol`.

17. Configure the Window dialog as follows:

    • Select **symbol** in the **Source Property List** to add it to the **Partition List**.

    • Select **Partition** as the **Type**.

    • Select **Row Based** and enter 2 in the **Row Based** field.

18. Click Add Window.

    The Query Wizard adds the sliding window to the Generated CQL Statement as follows:
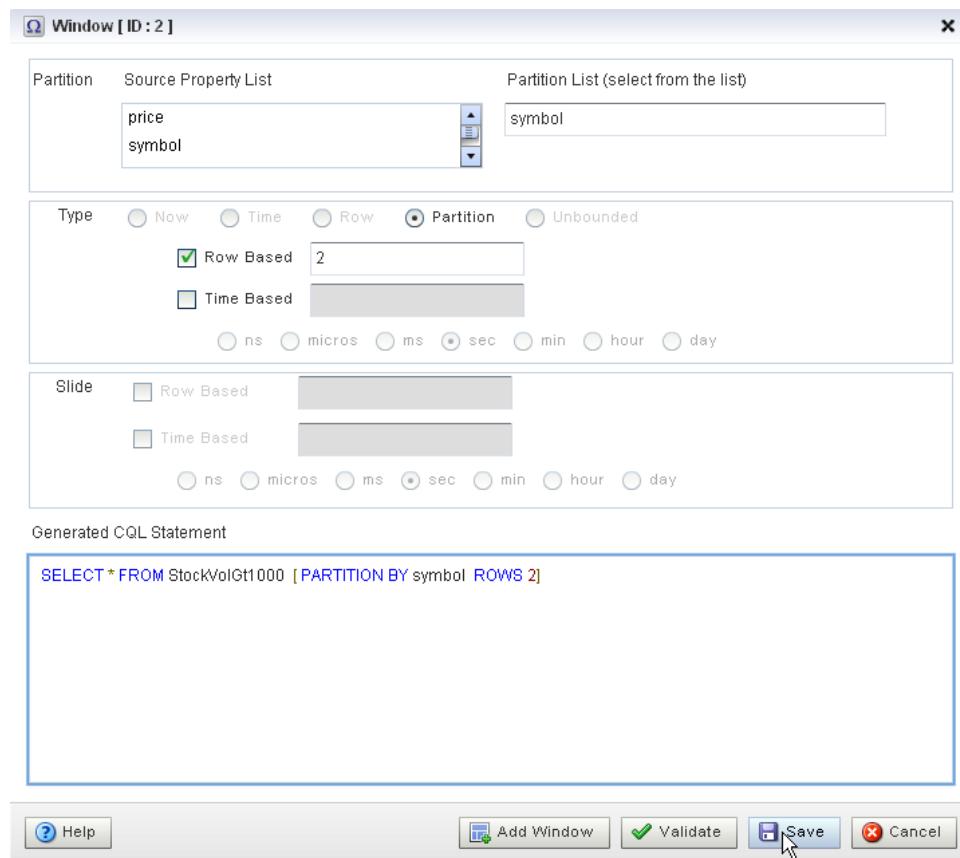
19. Click **Save**.

20. Click **Save Query**.

21. Click and drag a Select icon from the CQL Constructs palette and drop it anywhere in the Query Wizard canvas as follows:
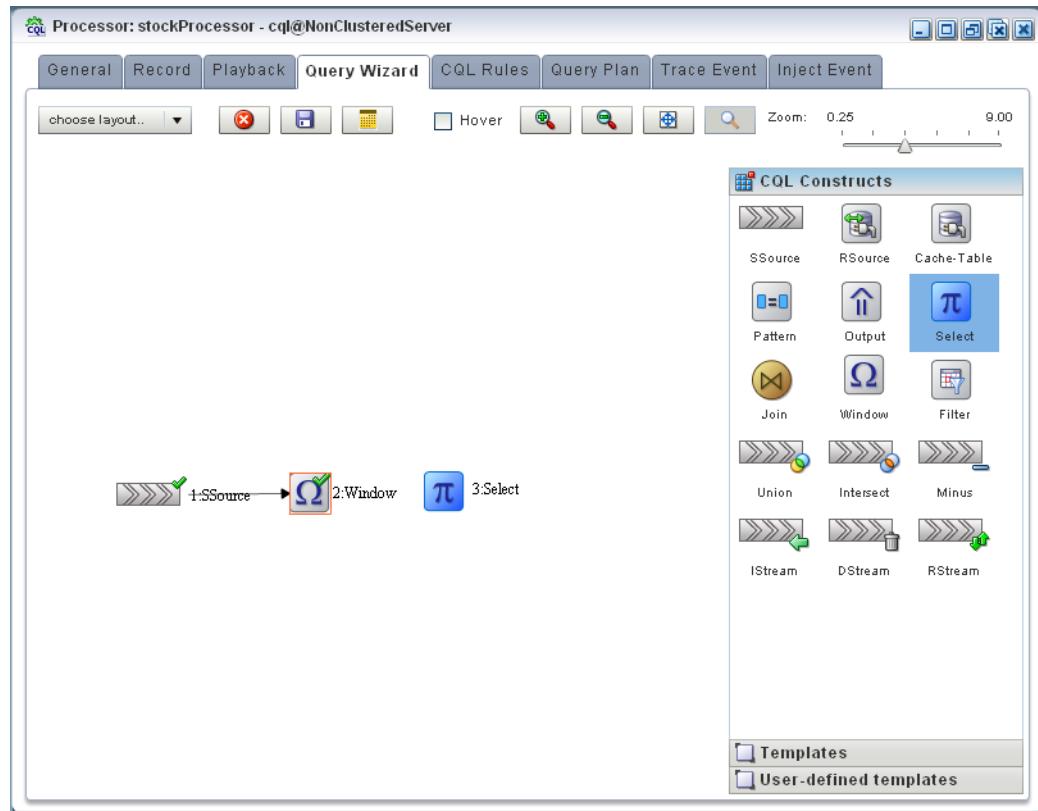
22. Click the **Window** icon and drag to the **Select** icon to connect the Oracle CQL constructs.

23. Double-click the **Select** icon.

The Select configuration screen appears.

24. Select **StockVolGt1000** from the **Select a source** pull-down menu.

This is the source of moving average query: the view you created earlier (see "Create a view source for the moving average query:").

25. Select **MovingAvgEvent** from the **Target Event Type** pull-down menu.

This is the output event your moving average query will produced. You will map properties from the source events to this output event.

26. In the Source Properties list, select **symbol**.

The selected source property is added to the Project Expression as follows:
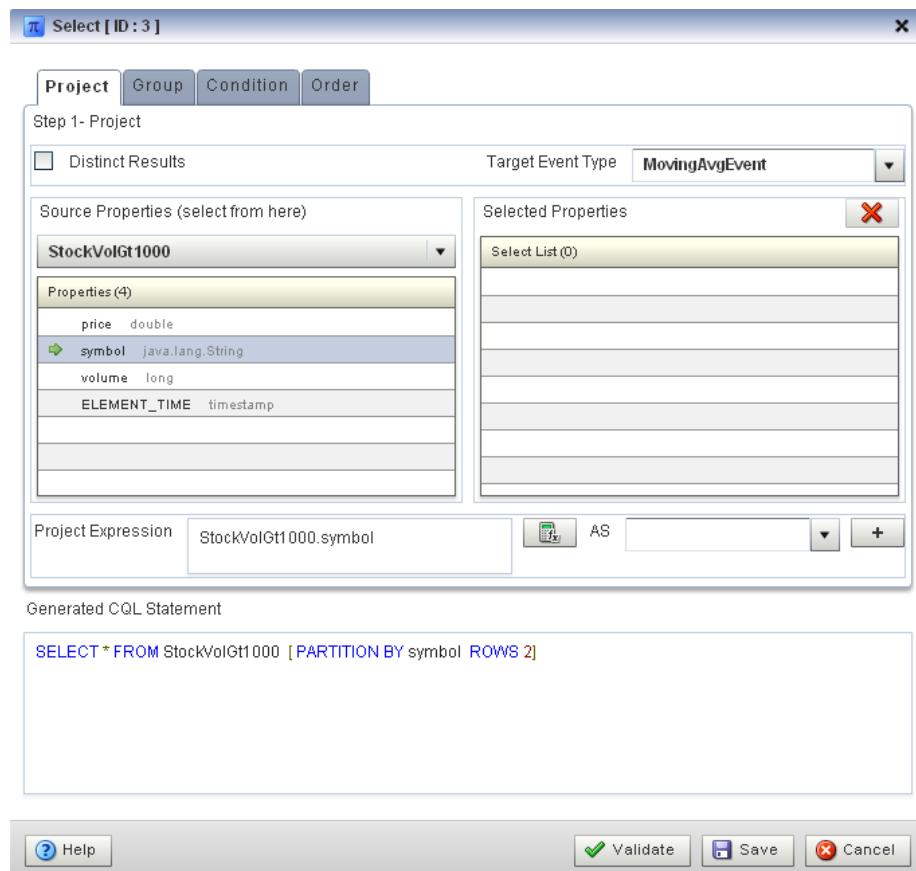
In this case, you just want to map the source property `symbol` to output event property `symbol` as is.

27. Click the pull-down menu next to the **AS** field and select **symbol**.

28. Click the Plus Sign button.

The source property is added to the project expression of the Generated CQL Statement as follows:
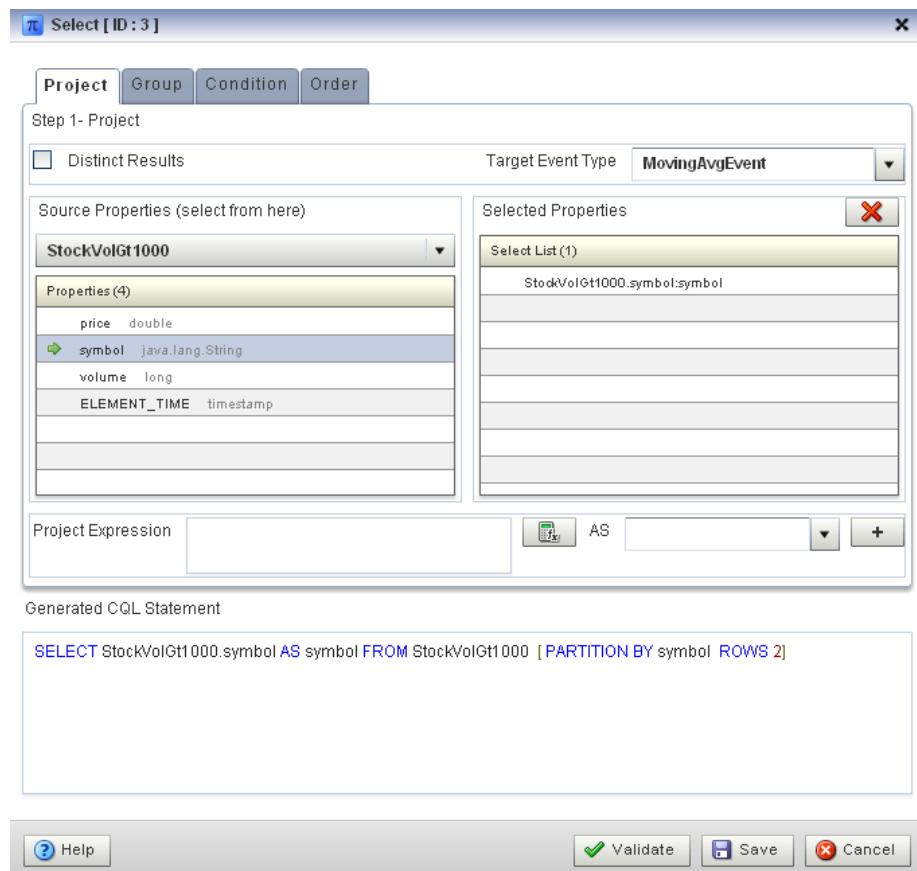
29. In the Source Properties list, select **price**.

The selected source property is added to the Project Expression as follows:
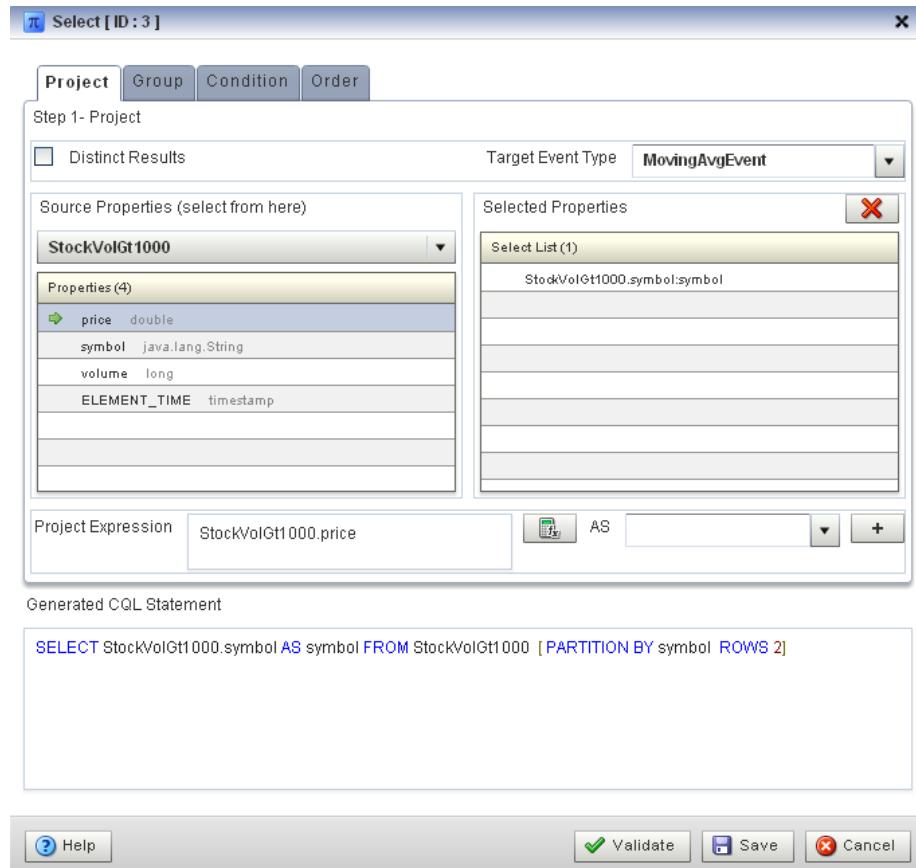
In this case, you want to process the source property `price` before you map it to the output event.

30. Click the Expression Builder button.

The Expression Builder dialog appears.

31. Select **Aggregate Function** from the **Select a function type** pull-down menu.

A list of the aggregate functions that Oracle CQL provides is displayed. You are going to use the **AVG** function.

32. Select the **StockVolGt1000.price** in the Expression Builder field.

33. Double-click the **AVG** function.

The `AVG()` function is wrapped around your selection in the Expression Builder field as follows:
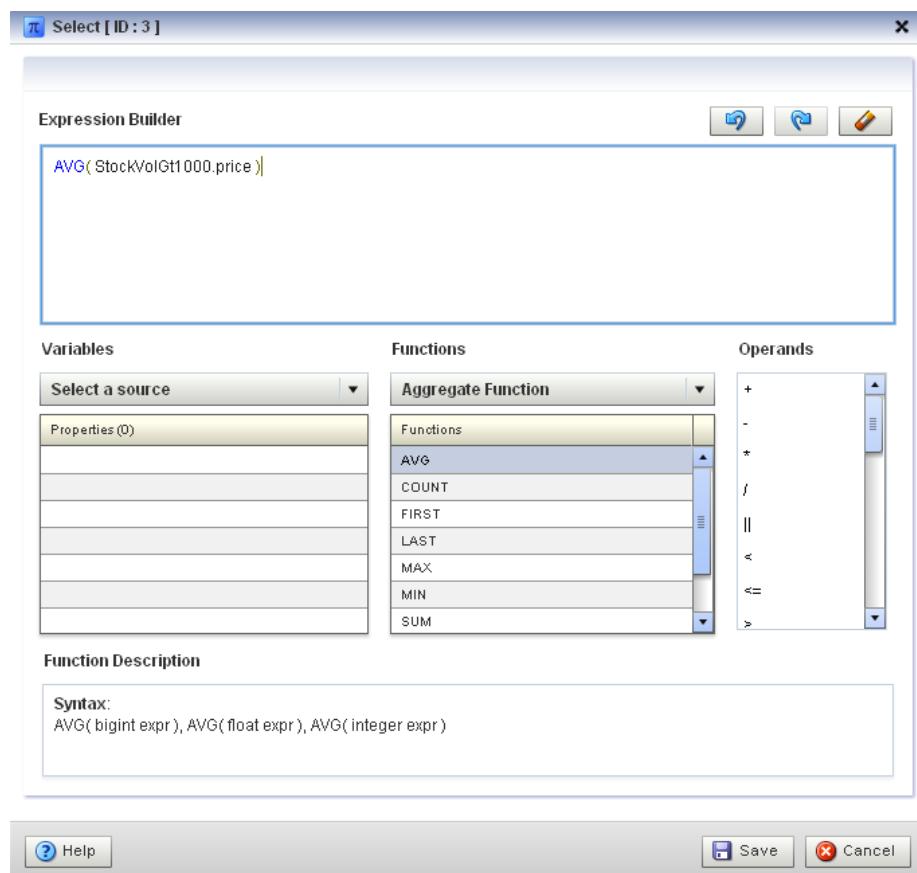
34. Click **Save**.

The expression is added to the Project Expression field as follows:
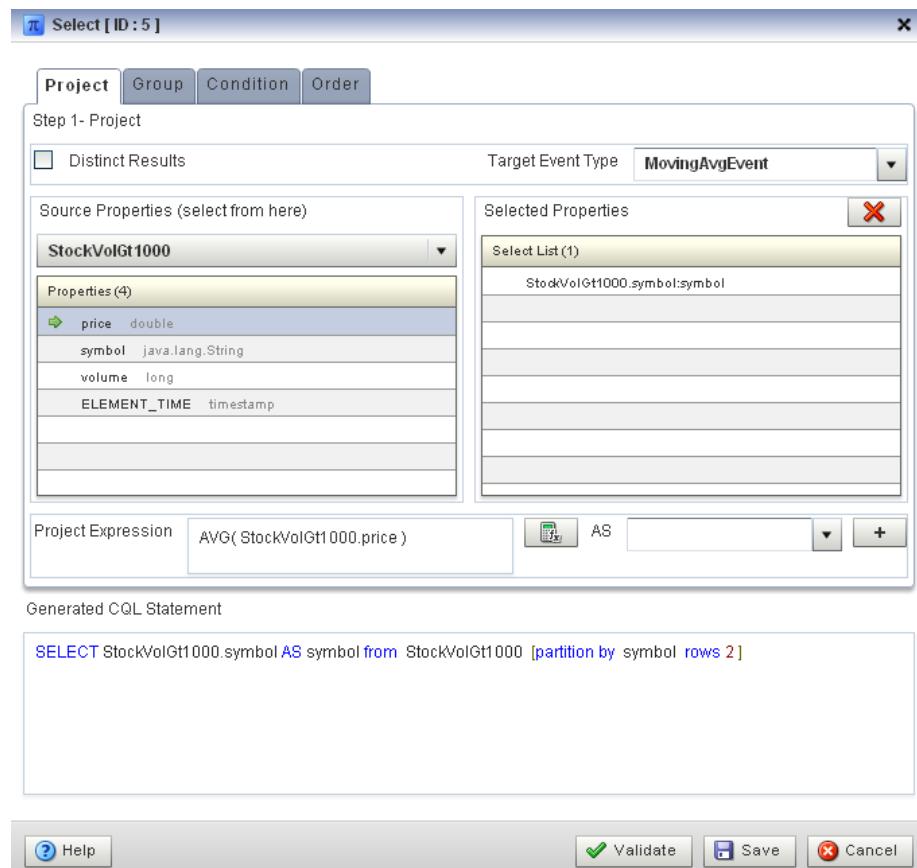
35. Click the pull-down menu next to the **AS** field and select **movingAvgPrice**.

36. Click the plus Sign button.

    The source property is added to the project expression of the Generated CQL Statement as follows:
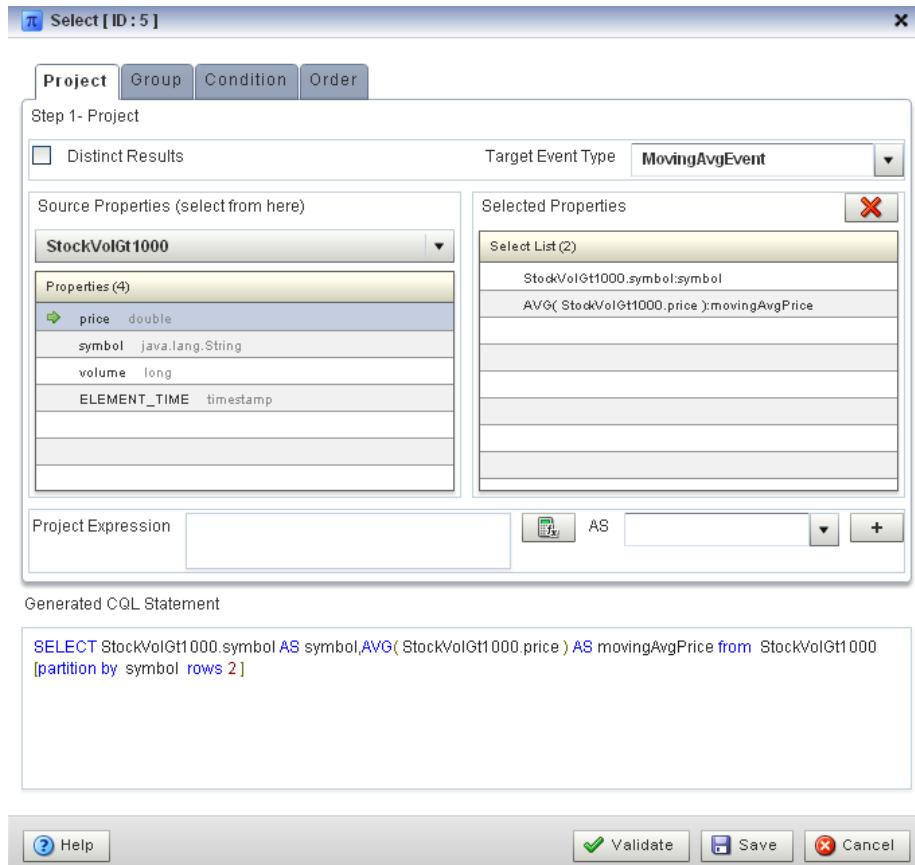
37. Click **Validate**.

A validation error dialog is shown as follows:
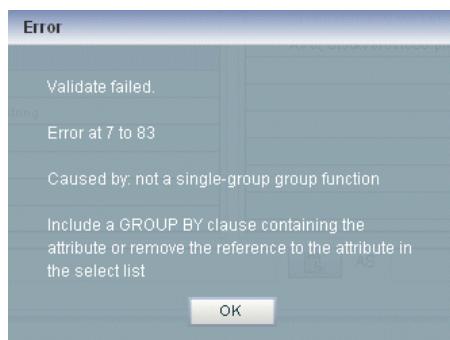
Because you are partitioning, you must specify a `GROUP BY` clause.

38. Select the **Group** tab.

The Group tab appears.

39. Configure the Group tab as follows:

- Select **StockVolGt1000** from the **Select a source** pull-down menu.

- Select **symbol** from the **Properties** list.

- Click the Plus Sign button.

The `symbol` property is added to `GROUP  BY` clause as follows:
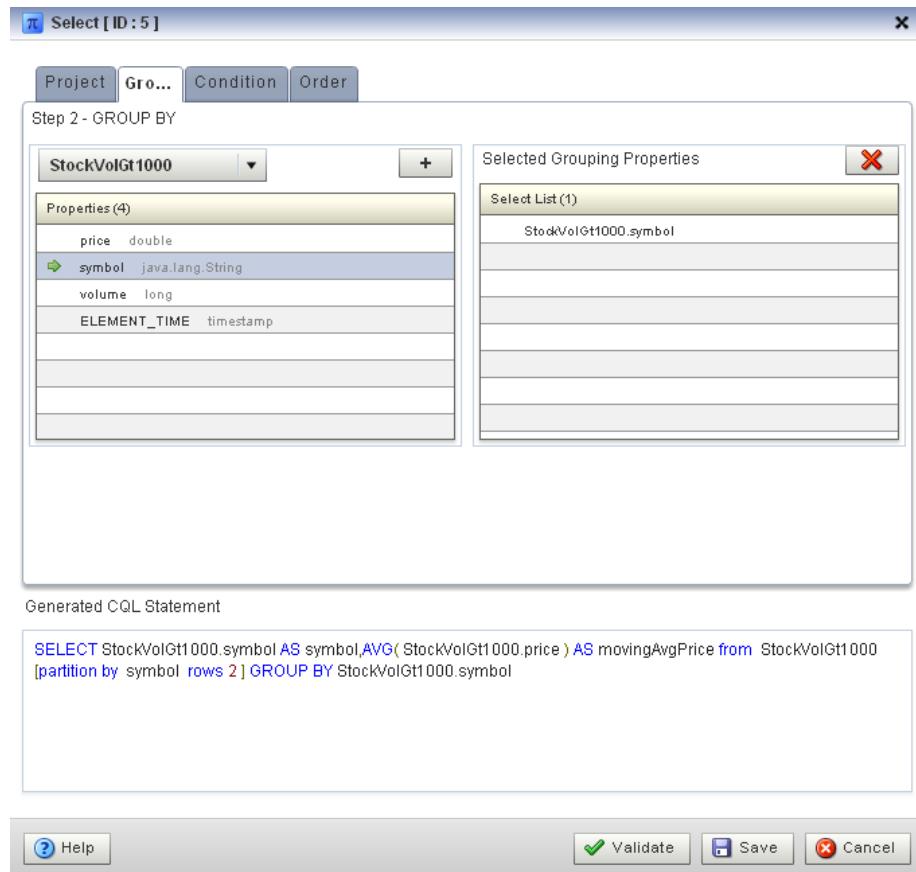
40. Click **Save**.

41. Click **Save Query**.

   Next, you want to connect the query to an output.

42. Click and drag an Output icon from the CQL Constructs palette and drop it anywhere in the Query Wizard canvas as follows:
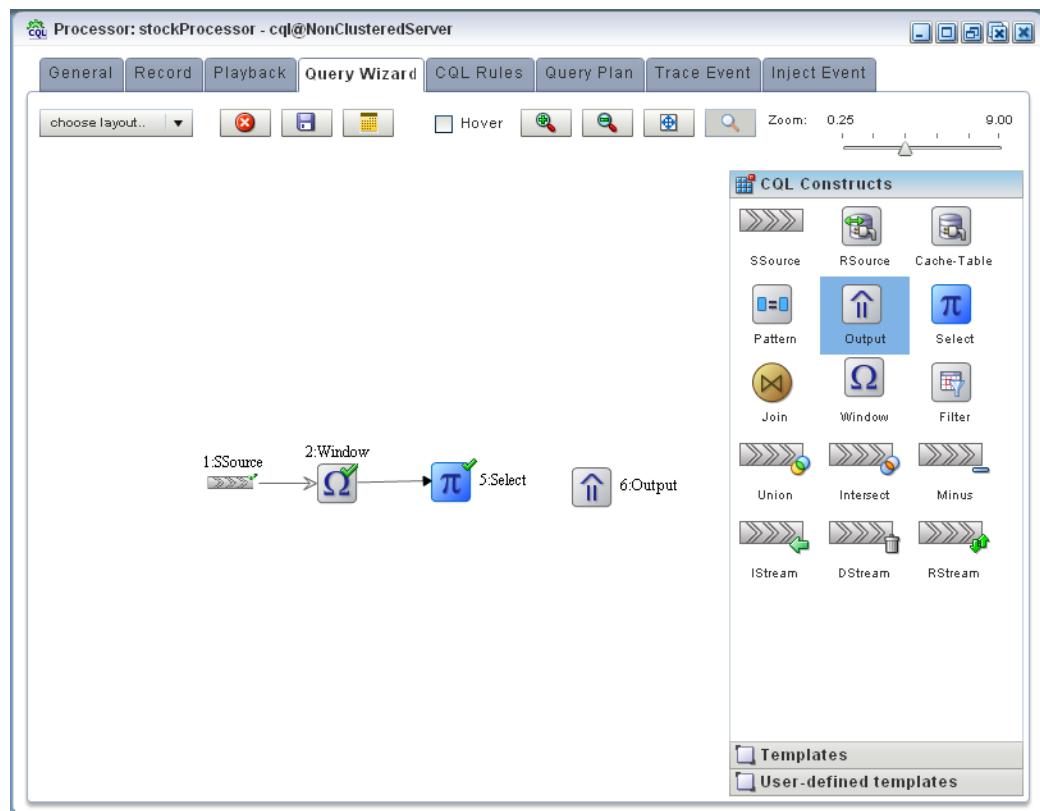
43. Click the **Select** icon and drag to the **Output** icon to connect the Oracle CQL constructs.

44. Double-click the **Output** icon.

    The Output configuration screen appears.

45. Configure the Output as follows:

    • Select **Query**.

    • Enter **MovingAverage** as the **Query Name**.
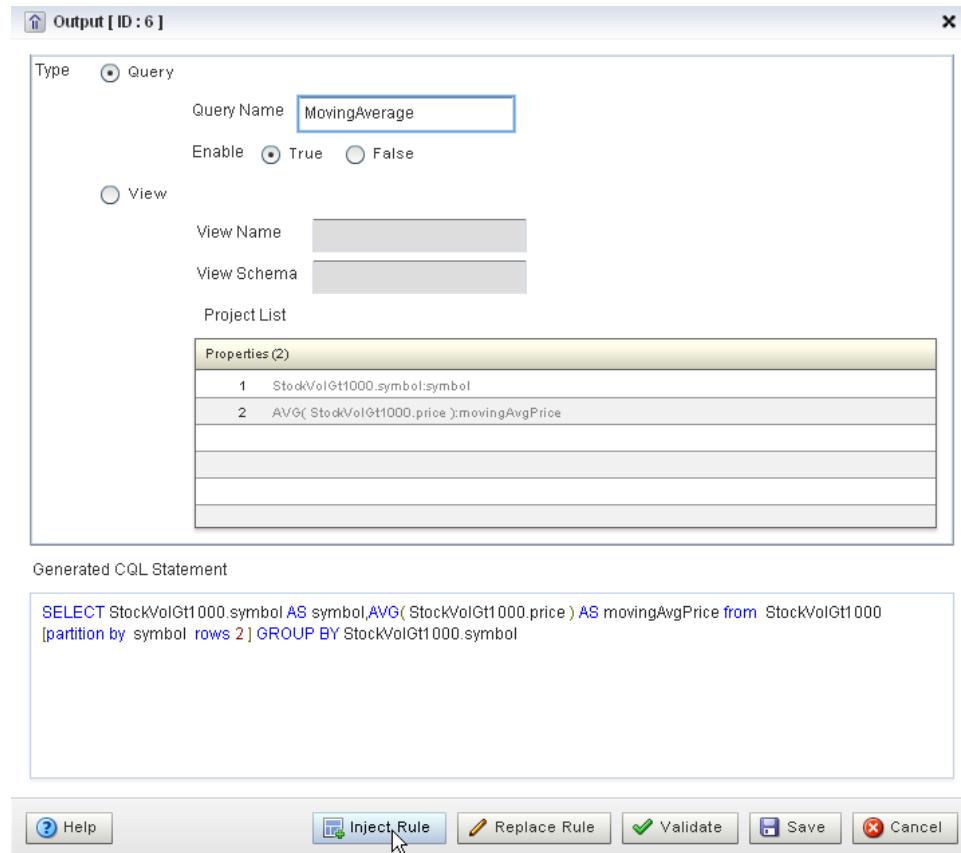
46. Click **Inject Rule**.

   The Inject Rule Confirmation dialog displays.

47. Click **OK**.

   The Query Wizard adds the rule to the `cqlProc` processor.

48. Click **Save**.

49. Click the **CQL Rules** tab.

   The CQL Rules tab displays.

50. Click the **Query** radio button.

   Confirm that your `MovingAverage` query is present.
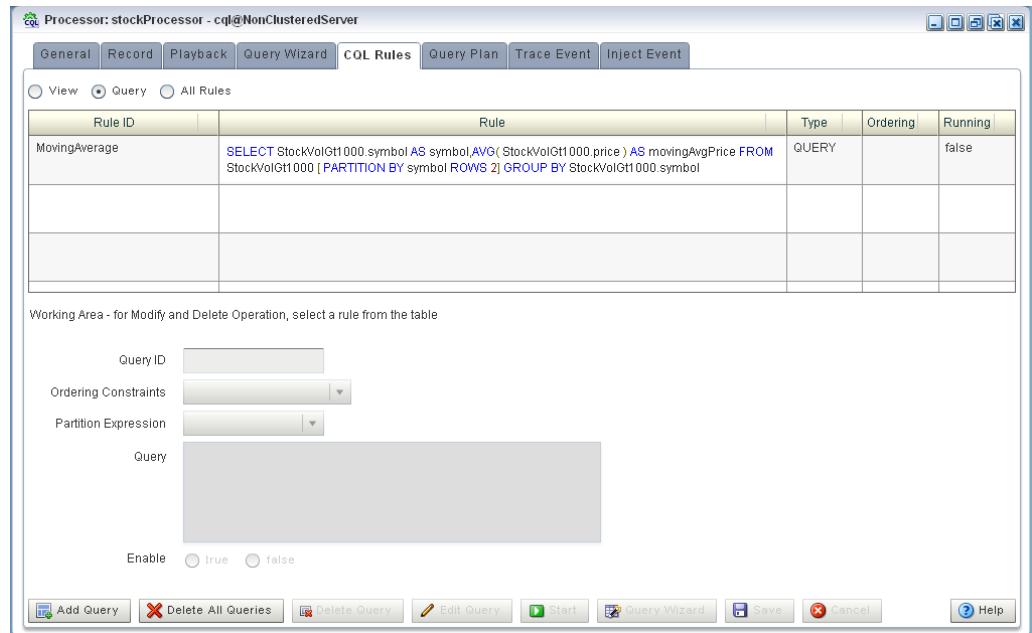
Test the moving average query:

1. To simulate the data feed for the moving average query, change to the */Oracle/ Middleware/my_oep/utils/load-generator* directory.

2. Run the load generator using the `stockData.prop` properties file:

   a. On Windows:

   ```
   prompt> runloadgen.cmd stockData.prop
   ```

   b. On UNIX:

   ```
   prompt> ./runloadgen.sh stockData.prop
   ```

3. In the Oracle Event Processing Visualizer, click the **ViewStream** button in the top panel.

   The Stream Visualizer screen displays.
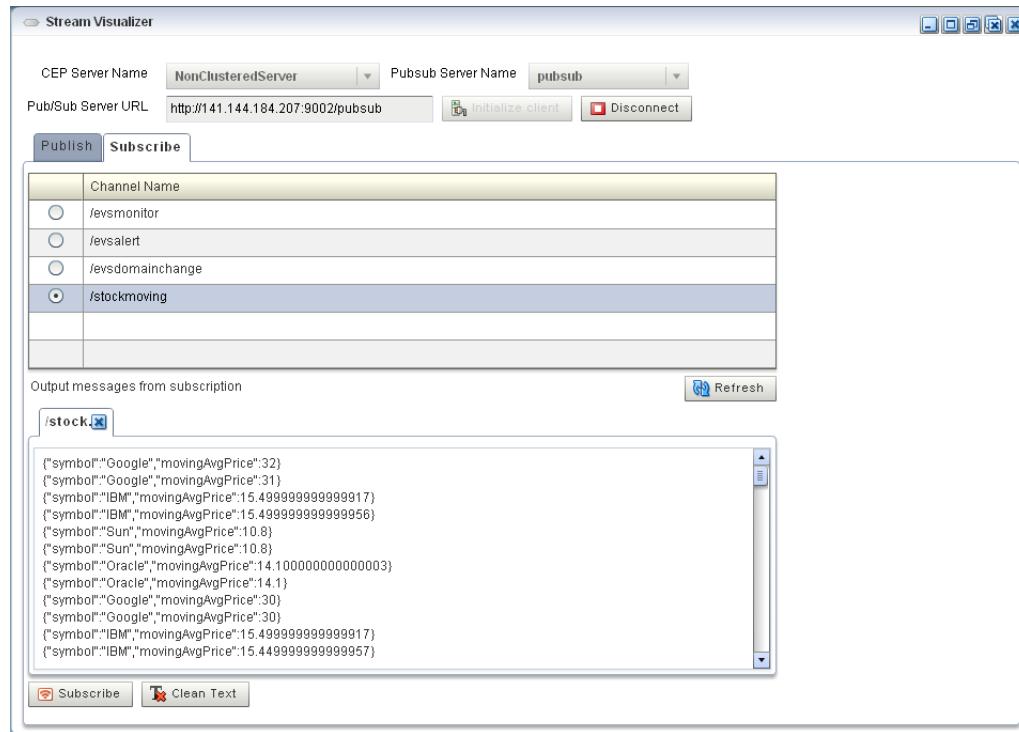
4. Click **Initialize Client**.

5. Enter `/stockmoving` in the **Initialize client** field.

6. Click **Subscribe**.

   As the moving average query outputs events, the Oracle Event Processing updates the **Received Messages** area showing the events generated.

6.7 Oracle Spatial Example

This example shows how to use Oracle Spatial with Oracle CQL queries to process a stream of Global Positioning System (GPS) events to track the GPS location of buses and generate alerts when a bus arrives at its pre-determined bus stop positions.

Figure 6-4 shows Oracle Spatial example Event Processing Network (EPN). The EPN contains the components that make up the application and how they fit together.

Figure 6-4   Oracle Spatial Example Event Processing Network
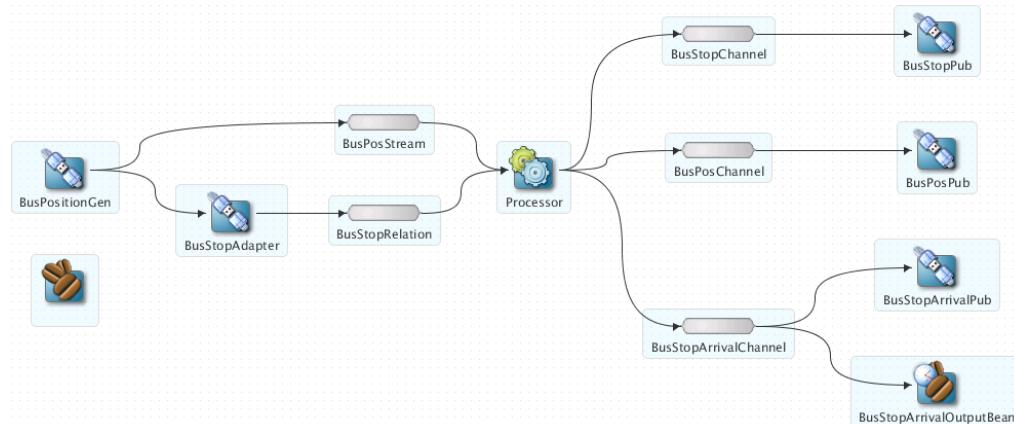
The example includes the following components:

- `BusPositionGen`: Component that simulates an input stream of bus position GPS events. It uses the Oracle Event Processing loadgen utility and csvgen adapter provider to read in comma separated values (CSV) and deliver them to the EPN as `BusPos` events.

- `BusStopAdapter`: Custom adapter component that generates bus stop positions based on */Oracle/Middleware/my_oep/examples/domains/ spatial_domain/defaultserver/applications/spatial_sample/ bus_stops.csv*.

- `BusPosStream`: Component that transmits `BusPos` events to the `Processor` as a stream.

- `BusStopRelation`: Component that transmits `BusPos` events to the `Processor` as a relation.

- `Processor`: Component that executes Oracle CQL queries on the incoming `BusPos` events.

- `BusStopChannel`, `BusPosChannel`, and `BusStopArrivalChannel`: Components that each specify a different selector to transmit the results of a different query from the `Processor` component to the appropriate outbound adapter or output bean.

- `BusStopPub`, `BusPosPub`, and `BusStopArrivalPub`: Components that publish the results of the `Processor` component's queries.

- `BusStopArrivalOutputBean`: POJO event bean component that logs a message for each insert, delete, and update event to help visualize the relation offered by the `BusStopArrivalChannel`.

For more information about data cartridges, see Oracle Fusion Middleware Oracle CQL Language Reference for Oracle Event Processing.

6.7.1 Run the Oracle Spatial Example

The Oracle Spatial application is pre-deployed to the `spatial_domain` domain. To run the application, you simply start an instance of Oracle Event Processing server.

Run the Oracle Spatial example from the spatial_domain domain:

1. Open a command window and change to the default server directory of the Oracle Spatial example domain directory, located in */Oracle/Middleware/ my_oep/oep/ examples/domains/spatial_domain/defaultserver*.

2. Start Oracle Event Processing by executing the appropriate script with the correct command line arguments:

   a. On Windows:

      - `prompt> startwlevs.cmd`

   b. On UNIX:

      - `prompt> ./startwlevs.sh`

      Wait for the console log to show:

```
<Mar 4, 2010 2:13:15 PM EST> <Notice> <Spring> <BEA-2047000> <The
application context for "spatial_sample" was started successfully>
<Mar 4, 2010 2:13:15 PM EST> <Notice> <Server> <BEA-2046000> <Server
STARTED>
```

This message indicates that the Oracle Spatial example is running correctly.

3. On the same host as the Oracle Spatial example is running, launch a browser and navigate to `http://localhost:9002/bus/web/main.html`.

Note:

You cannot run this example on one host and browse to it from another host. This is a limitation of the Google API Key that the example uses and is not a limitation of Oracle Event Processing.

The Oracle Spatial example Web page displays.
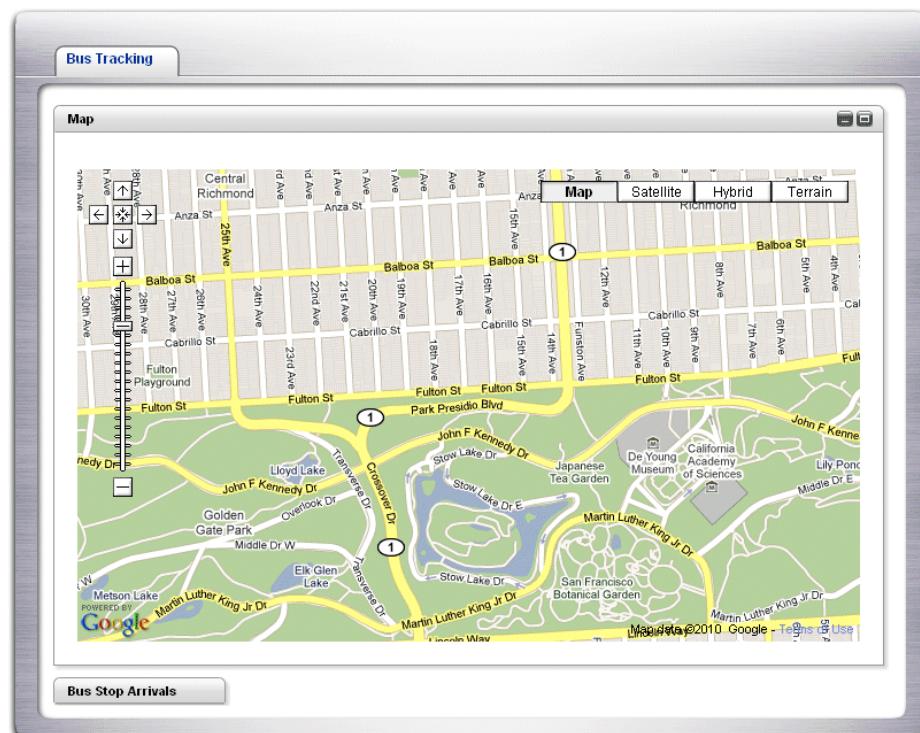
Click the **Bus Top Arrivals** tab to view bus stop arrivals as follows:
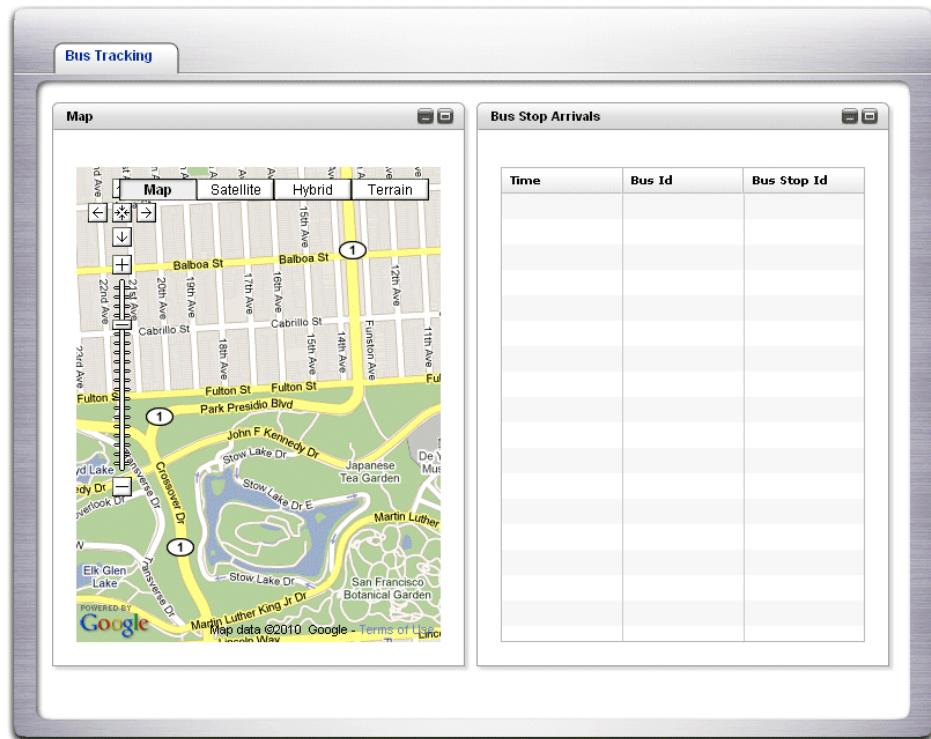
4. Execute the Oracle Event Processing load generator to generate sample data:

   a. Open a command prompt and navigate to /Oracle/Middleware/my_oep/ utils/load-generator.

   b. On Windows, type:

      • `runloadgen.cmd bus_positions.prop`

   c. On UNIX, type:

      • `./runloadgen.sh bus_positions.prop`

5. Observe the bus movements and alerts in the browser as follows:
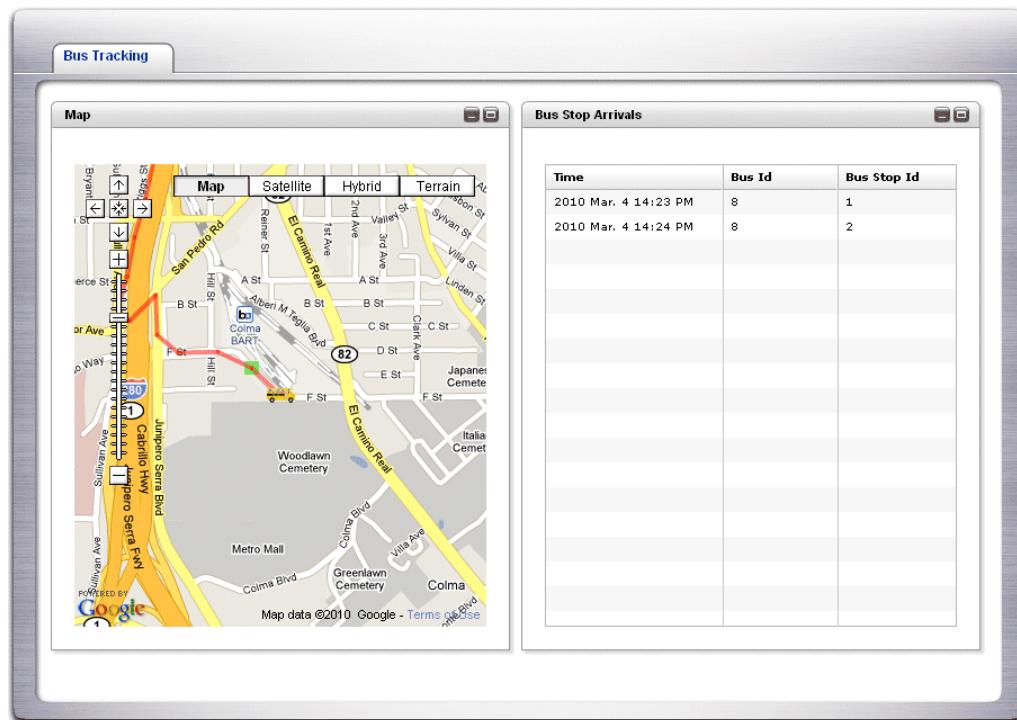
6.7.2 Build and Deploy the Oracle Spatial Example

The Oracle Spatial sample source directory contains the Java source, along with other required resources such as configuration XML files, that make up the Oracle Spatial application. The `build.xml` Ant file contains targets to build and deploy the application to the `spatial_domain` domain.

For more information, see Description of the Ant Targets to Build Hello World.

Build and deploy the Oracle Spatial example from the source directory:

1. If the `spatial_domain` Oracle Event Processing instance is not already running, follow the procedure in Run the Oracle Spatial Example to start the server.

   You must have a running server to successfully deploy the rebuilt application.

2. Open a new command window and change to the Oracle Spatial source directory, located in */Oracle/Middleware/my_oep/oep/examples/source/ applications/ spatial*.

3. Execute the `all` Ant target to compile and create the application JAR file:

   ```
   prompt> ant all
   ```

4. Execute the `deploy` Ant target to deploy the application JAR file to Oracle Event Processing:

   ```
   prompt> ant -Daction=update deploy
   ```

   Caution:

   This target overwrites the existing Oracle Spatial application JAR file in the domain directory.

6.7.3 Description of the Ant Targets to Build the Oracle Spatial Example

The `build.xml` file, located in the top level of the Oracle Spatial source directory, contains the following targets to build and deploy the application:

- `clean`: This target removes the `dist` and `output` working directories under the current directory.

- `all`: This target cleans, compiles, and outs the application into a JAR file called `com.bea.wlevs.example.spatial_12.1.2.0_0.jar`, and places the generated JAR file into a `dist` directory below the current directory.

- `deploy`: This target deploys the JAR file to Oracle Event Processing using the Deployer utility.

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

6.7.4 Implementation of the Oracle Spatial Example

All the files of the Oracle Spatial example are located relative to the */Oracle/Middleware/my_oep/examples/source/applications/spatial* directory.

The files used by the Oracle Spatial example include:

- An EPN assembly file that describes each component in the application and how all the components are connected together. You are required to include this XML file in your Oracle Event Processing application.

  In the example, the file is called `context.xml` and is located in the `~/META-INF/spring` directory.

- A component configuration file that configures the various components on the EPN including the processor component of the application:

  In the example, this file is called `config.xml` and is located in the `~/META-INF/wlevs` directory.

- Java files that implement:

  - `BusStopAdapter`: Custom adapter component that generates bus stop positions based on */Oracle/Middleware/my_oep/examples/domains/spatial_domain/defaultserver/applications/spatial_sample/bus_stops.csv*.

  - `OutputBean`: POJO event bean component that logs a message for each insert, delete, and update event to help visualize the relation offered by the `BusStopArrivalChannel`

  - `OrdsHelper`: Helper class that provides method `getOrds` to return the ordinates from a `JGeometry` as a `List` of `Double` values.

  These Java files are located in the `~/source/applications/spatial/src/com/oracle/cep/sample/spatial` directory.

  For additional information about the Oracle Event Processing APIs referenced in this POJO, see *Java API Reference for Oracle Event Processing*.

- A `MANIFEST.MF` file that describes the contents of the OSGi bundle that will be deployed to Oracle Event Processing.

In the example, the `MANIFEST.MF` file is located in the `META-INF` directory.

For more information about creating this file, as well as a description of creating the OSGi bundle that you deploy to Oracle Event Processing, see *Developing Applications for Oracle Event Processing*.

The Oracle Spatial example uses a `build.xml` Ant file to compile, assemble, and deploy the OSGi bundle; see Build and Deploy the Oracle Spatial Example for a description of this `build.xml` file if you also use Ant in your development environment.

6.8 Foreign Exchange (FX) Example

The foreign exchange example, called FX for simplicity, is a more complex example than the HelloWorld example because it includes multiple Oracle CQL processors that handle information from multiple data feeds. In the example, the data feeds are simulated using the Oracle Event Processing load generator utility.

Figure 6-5 shows the FX example Event Processing Network (EPN). The EPN contains the components that make up the application and how they fit together.

Figure 6-5   FX Example Event Processing Network
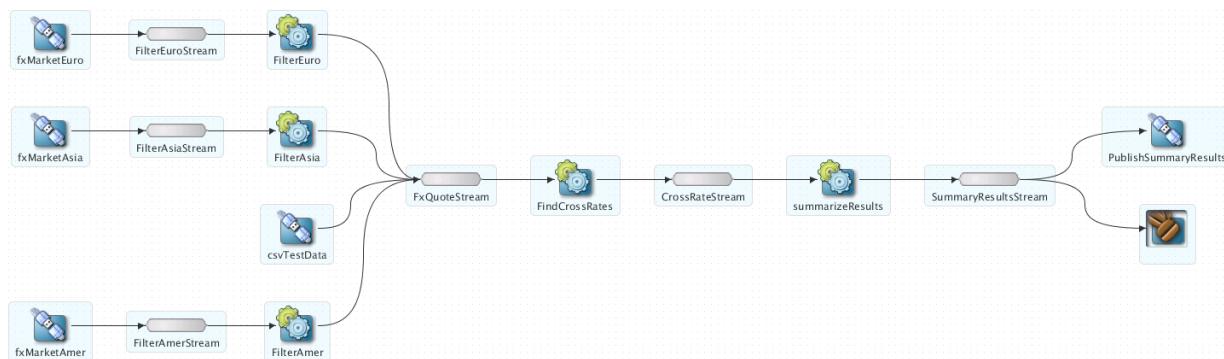
In this scenario, three data feeds, simulated using the load generator, send a constant pair of values from different parts of the world; the value pairs consist of a currency pair, such as USDEUR for US dollar - European euro, and an exchange rate between the two currencies. The `fxMarketAmer`, `fxMarketAsia`, and `fxMarketEuro` adapters receive the data from the feeds, convert them into events, and pass them to the corresponding `FilterAmer`, `FilterAsia`, and `FilterEuro` processors. Each Oracle CQL processor performs an initial stale check to ensure that no event is more than 1 second old and then a boundary check to ensure that the exchange rate between the two currencies is within a current boundary. The Oracle CQL processor also only selects a specific currency pair from a particular channel; for example, the server selects USDEUR from the simulated American data feed, but rejects all other pairs, such as USDAUD (Australian dollar).

After the data from each data feed provider passes this initial preparation phase, a different Oracle CQL processor, called `FindCrossRates`, joins all events across all providers, calculates the mid-point between the maximum and minimum rate, and then applies a trader-specified spread. Finally, the Oracle CQL processor forwards the rate to the POJO that contains the business code; in this example, the POJO simply publishes the rate to clients.

The Oracle Event Processing monitor is configured to watch if the event latency in the last step exceeds some threshold, such as no updated rates in a 30 second time-span, and if there is too much variance between two consecutive rates for the same currency

pair. Finally, the last rate of each currency pair is forwarded to the Oracle Event Processing http pub-sub server.

6.8.1 Run the Foreign Exchange Example

For optimal demonstration purposes, Oracle recommends that you run this example on a powerful computer, such as one with multiple CPUs or a 3 GHz dual-core Intel, with a minimum of 2 GB of RAM.

The Foreign Exchange (FX) application is pre-deployed to the `fx_domain` domain. To run the application, you simply start an instance of Oracle Event Processing server.

Run the foreign exchange example:

1. Open a command window and change to the default server directory of the FX domain directory, located in */Oracle/Middleware/my_oep/oep/examples/ domains/fx_domain/defaultserver*.

2. Start Oracle Event Processing by executing the appropriate script with the correct command line arguments:

   a. On Windows:

   - `prompt> startwlevs.cmd`

   b. On UNIX:

   - `prompt> ./startwlevs.sh`

3. When prompted, enter **wlevs** for the **user name** and **password**.

   The FX application is now ready to receive data from the data feeds.

4. To simulate an American data feed, open a new command window.

5. Change to the */Oracle/Middleware/my_oep/utils/load-generator* directory.

6. Run the load generator using the `fxAmer.prop` properties file:

   a. On Windows:

   `prompt> runloadgen.cmd fxAmer.prop`

   b. On UNIX:

   `prompt> ./runloadgen.sh fxAmer.prop`

7. Repeat steps 4 - 6 to simulate an Asian data feed, using the `fxAsia.prop` properties file:

   a. On Windows:

   `prompt> runloadgen.cmd fxAsia.prop`

   b. On UNIX:

   `prompt> ./runloadgen.sh fxAsia.prop`

8. Repeat steps 4 - 6 to simulate an European data feed, using the `fxEuro.prop` properties file:

a. On Windows:

```
prompt> runloadgen.cmd fxEuro.prop
```

b. On UNIX:

```
prompt> ./runloadgen.sh fxEuro.prop
```

After the server status messages scroll by in the command window from which you started the server, and the three load generators start, you should see messages similar to the following being printed to the server command window (the message will likely be on one line):

```
OutputBean:onEvent() +
    <TupleValue>
        <EventType>SpreaderOuputEvent</EventType>
        <ObjectName>FindCrossRatesRule</ObjectName>
        <Timestamp>1843704855846</Timestamp>
        <TupleKind>null</TupleKind>
        <DoubleAttribute>
            <Value>90.08350000074516</Value>
        </DoubleAttribute>
        <CharAttribute>
            <Value>USD</Value>
            <Length>3</Length>
        </CharAttribute>
        <CharAttribute>
            <Value>JPY</Value>
            <Length>3</Length>
        </CharAttribute>
    <IsTotalOrderGuarantee>false</IsTotalOrderGuarantee>
</TupleValue>
```

These messages indicate that the Foreign Exchange example is running correctly. The output shows the cross rates of US dollars to Japanese yen and US dollars to UK pounds sterling.

6.8.2 Build and Deploy the Foreign Exchange Example from the Source Directory

The Foreign Exchange (FX) sample source directory contains the Java source, along with other required resources such as configuration XML files, that make up the FX application. The build.xml Ant file contains targets to build and deploy the application to the fx_domain domain, as described in Description of the Ant Targets to Build Hello World.

Build and deploy the foreign exchange example from the source directory:

1. If the FX Oracle Event Processing instance is not already running, follow the procedure in Run the Foreign Exchange Example to start the server.

You must have a running server to successfully deploy the rebuilt application.

2. Open a new command window and change to the FX source directory, located in */Oracle/Middleware/my_oep/oep/examples/source/applications/fx*.

3. Execute the all Ant target to compile and create the application JAR file:

```
prompt> ant all
```

4. Execute the deploy Ant target to deploy the application JAR file to Oracle Event Processing:

```
prompt> ant -Dusername=wlevs -Dpassword=wlevs -Daction=update deploy
```

> **Caution:**
>
> This target overwrites the existing FX application JAR file in the domain directory.

5. If the load generators required by the FX application are not running, start them as described in Run the Foreign Exchange Example.

After the server starts, you should see the following message printed to the output:

```
{crossRate=USDJPY, internalPrice=119.09934499999781}, {crossRate=USDGBP,
internalPrice=0.5031949999999915}, {crossRate=USDJPY,
internalPrice=117.73945624999783}
```

This message indicates that the FX example has been redeployed and is running correctly.

6.8.3 Description of the Ant Targets to Build FX

The `build.xml` file, located in the top-level directory of the FX source, contains the following targets to build and deploy the application:

- `clean`: This target removes the `dist` and `output` working directories under the current directory.

- `all`: This target cleans, compiles, and puts the application into a JAR file called `com.bea.wlevs.example.fx_12.1.3.0_0.jar`, and places the generated JAR file into a `dist` directory below the current directory.

- `deploy`: This target deploys the JAR file to Oracle Event Processing using the Deployer utility.

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

6.8.4 Implementation of the FX Example

All the files of the FX example are located relative to the */Oracle/Middleware/my_oep/examples/source/applications/fx* directory.

The files used by the FX example include:

- An EPN assembly file that describes each component in the application and how all the components are connected together. You are required to include this XML file in your Oracle Event Processing application.

  In the example, the file is called `com.oracle.cep.sample.fx.context.xml` and is located in the `~/META-INF/spring` directory.

- The processor.xml file configures the processor components for the application:

  The `filterAmer`, `filterAsia`, `filterEuro`, and `FindCrossRates` processors, all in a single file. This XML file includes the Oracle CQL rules that select particular currency pairs from particular simulated market feeds and joins together all the events that were selected by the pre-processors, calculates an internal price for the particular currency pair, and then calculates the cross rate. In the example, this file is called `spreader.xml` and is located in the `~/META-INF/wlevs` directory.

The `summarizeResults` Oracle CQL processor includes the Oracle CQL rule that summarizes the results of the `FindCrossRates` processor. In the example, this file is called `SummarizeResults.xml` and is located in the `~/META-INF/wlevs` directory.

- An XML file that configures the `PublishSummaryResults` http pub-sub adapter. In the example, this file is called `PubSubAdapterConfiguration.xml` and is located in the `~/META-INF/wlevs` directory.

- A Java file that implements the `OutputBean` component of the application, a POJO that contains the business logic. This POJO prints out to the screen the events that it receives, programmed in the `onEvent` method. The POJO also registers into the event type repository the `ForeignExchangeEvent` event type.

  In the example, the file is called `OutputBean.java` and is located in the `~/src/com/oracle/cep/sample/fx` directory.

  For additional information about the Oracle Event Processing APIs referenced in this POJO, see *Java API Reference for Oracle Event Processing*.

- A `MANIFEST.MF` file that describes the contents of the OSGi bundle that will be deployed to Oracle Event Processing.

  In the example, the `MANIFEST.MF` file is located in the `META-INF` directory.

  For more information about creating this file, as well as a description of creating the OSGi bundle that you deploy to Oracle Event Processing, see Overview of Application Assembly and Deployment in *Developing Applications for Oracle Event Processing*.

  The FX example uses a `build.xml` Ant file to compile, assemble, and deploy the OSGi bundle; see Build and Deploy the Foreign Exchange Example from the Source Directory for a description of this `build.xml` file if you also use Ant in your development environment.

6.9 Signal Generation Example

The signal generation sample application receives simulated market data and verifies if the price of a security has fluctuated more than two percent. The application also detects the pattern occurring by keeping track of successive stock prices for a particular symbol; if more than three successive prices are larger than the one before it, this is considered a pattern.

Figure 6-6 shows the signal generation example Event Processing Network (EPN). The EPN contains the components that make up the application and how they fit together.

Figure 6-6   The Signal Generation Example Event Processing Network
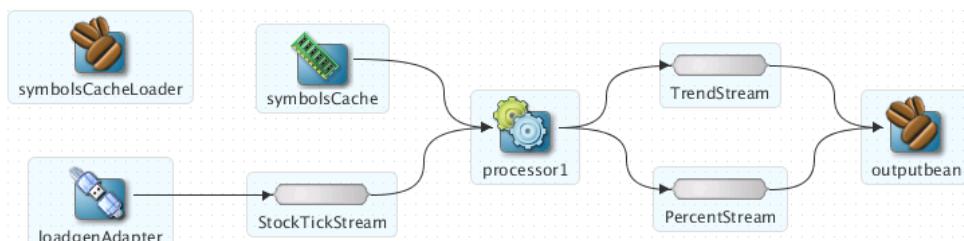
The application simulates a market data feed using the Oracle Event Processing load generator utility; in this example, the load generator generates up to 10,000 messages per second. The example includes an HTML dashboard which displays the matched

events along with the latencies; events consist of a stock symbol, a timestamp, and the price.

The example demonstrates very low latencies, with minimum latency *jitter* under high throughputs. Once the application starts running, the processor matches an average of 800 messages per second. If the application is run on the minimum configured system, the example shows very low average latencies (30-300 microsecond, on average) with minimal latency spikes (low milliseconds).

The example computes and displays latency values based on the difference between a timestamp generated on the load generator and timestamp on Oracle Event Processing. Computing valid latencies requires very tight clock synchronization, such as 1 millisecond, between the computer running the load generator and the computer running Oracle Event Processing. For this reason, Oracle recommends running both the load generator and Oracle Event Processing on a single multi-CPU computer where they will share a common clock.

The example also shows how to use the Oracle Event Processing event caching feature. In particular the single processor in the EPN sends events to both an event bean and a cache.

The example also demonstrates how to use Oracle CQL queries.

6.9.1 Run the Signal Generation Example

For optimal demonstration purposes, Oracle recommends that you run this example on a powerful computer, such as one with multiple CPUs or a 3 GHz dual-core Intel, with a minimum of 2 GB of RAM.

The `signalgeneration_domain` domain contains a single application: the signal generation sample application. To run the signal generation application, you simply start an instance of Oracle Event Processing in that domain.

Run the signal generation example:

1. Open a command window and change to the default server directory of the `signalgeneration_domain` domain directory, located in */Oracle/ Middleware/my_oep/oep/examples/domains/ signalgeneration_domain/defaultserver*.

2. Start Oracle Event Processing by executing the appropriate script with the correct command line arguments:

    a. On Windows:

        • `prompt> startwlevs.cmd`

    b. On UNIX:

        • `prompt> ./startwlevs.sh`

3. When prompted, enter **wlevs** for the **user name** and **password**.

4. Wait until you see console messages like this:

        <Apr 24, 2009 11:40:37 AM EDT> <Notice> <Server> <BEA-2046000> <Server STARTED>
        Throughput (msg per second): 0. Average latency (microseconds): 0
        Throughput (msg per second): 0. Average latency (microseconds): 0
        Throughput (msg per second): 0. Average latency (microseconds): 0

```
Throughput (msg per second): 0. Average latency (microseconds): 0
...
```

The signal generation application is now ready to receive data from the data feeds.

Next, to simulate a data feed, you use a load generator programmed specifically for the example.

5. 0pen a new command window.

6. Change to the */Oracle/Middleware/my_oep/examples/domains/ signalgeneration_domain/defaultserver/utils* directory.

7. Run the startDataFeed command:

a. On Windows:

```
prompt> startDataFeed.cmd
```

b. On UNIX:

```
prompt> ./startDataFeed.sh
```

8. Invoke the example dashboard by starting a browser and opening the following HTML page:

```
http://host:9002/signalgeneration/dashboard.html
```

Replace *host* with the name of the computer on which Oracle Event Processing is running; if it is the same computer as your browser, you can use localhost.

9. In the browser, click **Start** on the HTML page.

You should start seeing the events that match the Oracle CQL rules configured for this example as follows:
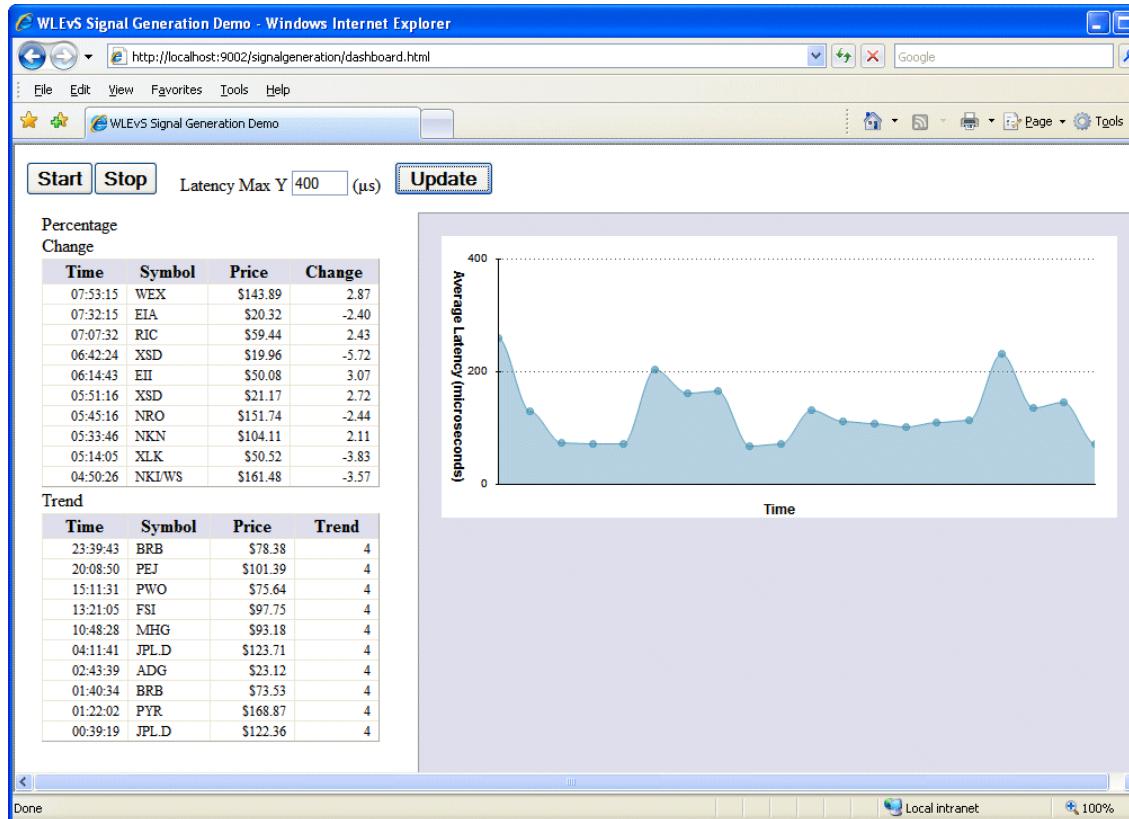
6.9.2 Build and Deploy the Signal Generation Example from the Source Directory

The signal generation sample source directory contains the Java source, along with other required resources, such as configuration XML files, EPN assembly file, and DOJO client JavaScript libraries, that make up the signal generation application. The `build.xml` Ant file contains targets to build and deploy the application to the `signalgeneration_domain` domain, as described in Description of the Ant Targets to Build Signal Generation.

Build and deploy the signal generation example from the source directory:

1. If the signal generation Oracle Event Processing instance is not already running, follow the procedure in Run the Signal Generation Example to start the server. You must have a running server to successfully deploy the rebuilt application.

2. Open a new command window and change to the signal generation source directory, located in */Oracle/Middleware/my_oep/oep/examples/source/ applications/signalgeneration*.

3. Execute the `all` Ant target to compile and create the application JAR file:

   ```
   prompt> ant all
   ```

4. Execute the `deploy` Ant target to deploy the application JAR file to the */Oracle/ Middleware/my_oep/examples/domains/signalgeneration_domain/ defaultserver/applications/signalgeneration* directory:

   ```
   prompt> ant deploy
   ```

> **Caution:**
>
> This target overwrites the existing signal generation application JAR file in the domain directory.

5. If the load generator required by the signal generation application is not running, start it as described in Run the Signal Generation Example.

6. Invoke the example dashboard as described in Run the Signal Generation Example.

6.9.3 Description of the Ant Targets to Build Signal Generation

The `build.xml` file, located in the top-level directory of the signal generation example source, contains the following targets to build and deploy the application:

- `clean`: This target removes the `dist` and `output` working directories under the current directory.

- `all`: This target cleans, compiles, and puts the application into a JAR file called `com.bea.wlevs.example.signalgen_12.1.2.0_0.jar`, and places the generated JAR file into a `dist` directory below the current directory.

- `deploy`: This target deploys the JAR file to Oracle Event Processing using the Deployer utility.

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

6.9.4 Implementation of the Signal Generation Example

All the files of the signal generation are located relative to the */Oracle/ Middleware/my_oep/examples/source/applications/signalgeneration* directory.

The files used by the signal generation example include:

- A EPN assembly file that describes each component in the application and how all the components are connected together.

  In the example, the file is called `epn_assembly.xml` and is located in the `~/ META-INF/spring` directory.

- An XML file that configures the processor component of the application; this file is called `config.xml` and is located in the `~/META-INF/wlevs` directory

  The `config.xml` file configures the `processor1` Oracle CQL processor, in particular the Oracle CQL rules that verify whether the price of a security has fluctuated more than two percent and whether a trend has occurred in its price.

- A Java file that implements the `SignalgenOutputBean` component of the application, a POJO that contains the business logic. This POJO is an `HttpServlet` and an `EventSink`. Its `onEvent` method consumes `PercentTick` and `TrendTick` event instances, computes latency, and displays dashboard information.

  In the example, the file is called `SignalgenOutputBean.java` and is located in the `~/src/oracle/cep/example/signalgen` directory.

For general information about programming event sinks, see Handling Events with Sources and Sinks in Oracle Fusion Middleware Developing Application for Oracle Event Processing .

- A `MANIFEST.MF` file that describes the contents of the OSGi bundle that will be deployed to Oracle Event Processing.

  In the example, the `MANIFEST.MF` file is located in the `META-INF` directory

  For more information about creating this file, as well as a description of creating the OSGi bundle that you deploy to Oracle Event Processing, see Overview of Application Assembly and Deployment in Oracle Fusion Middleware Developing Application for Oracle Event Processing .

- A `dashboard.html` file in the main example directory; this HTML file is the example dashboard that displays events and latencies of the running signal generation application. The HTML file uses Dojo JavaScript libraries from `http://dojotoolkit.org/`, located in the `dojo` directory.

For additional information about the Oracle Event Processing APIs referenced in `ForeignExchangeBuilderFactory`, see *Java API Reference for Oracle Event Processing*.

The signal generation example uses a `build.xml` Ant file to compile, assemble, and deploy the OSGi bundle; see Build and Deploy the Signal Generation Example from the Source Directory for a description of this `build.xml` file if you also use Ant in your development environment.

6.10 Event Record and Playback Example

The record and playback example shows how to configure a component to record events to an event store and then configure another component in the network to playback events from the store. The example uses the Oracle Event Processing-provided default Berkeley database to store the events. The example also shows how to configure a publishing HTTP pub-sub adapter as a stage in the event processing network.

Figure 6-7 shows the event record and playback example Event Processing Network (EPN). The EPN contains the components that make up the application and how they fit together.

Figure 6-7   The Event Record and Playback Example Event Processing Network

The application contains four components in its event processing network:

- `simpleEventSource`: an adapter that generates simple events for purposes of the example. This component has been configured to record events, as shown in the graphic.

  The configuration source for this adapter is:

```
<adapter>
    <name>simpleEventSource</name>
    <record-parameters>
        ...
```

```
        </record-parameters>
    </adapter>
```

- `eventStream`: a channel that connects the `simpleEventSource` adapter and `recplayEventSink` event bean. This component has been configured to playback events.

  The configuration source for this channel is:

  ```
  <channel>
      <name>eventStream</name>
      <playback-parameters>
          ...
      </playback-parameters>
      ...
  </channel>
  ```

- `recplayEventSink`: an event bean that acts as a sink for the events generated by the adapter.

- `playbackHttpPublisher`: a publishing HTTP pub-sub adapter that listens to the `recplayEventSink` event bean and publishes to a channel called `/playbackchannel` of the Oracle Event Processing HTTP Pub-Sub server.

6.10.1 Run the Event Record/Playback Example

The `recplay_domain` domain contains a single application: the record and playback sample application. To run this application, you first start an instance of Oracle Event Processing in the domain, as described in the following procedure.

The procedure then shows you how to use Oracle Event Processing Visualizer to start the recording and playback of events at the `simpleEventSource` and `eventStream` components, respectively. Finally, the procedure shows you how to use Oracle Event Processing Visualizer to view the stream of events being published to a channel by the `playbackHttpPublisher` adapter.

Run the event record/playback example:

1. Open a command window and change to the default server directory of the `recplay_domain` domain directory, located in */Oracle/Middleware/ my_oep/oep/ examples/domains/recplay_domain/defaultserver*.

2. Start Oracle Event Processing by executing the appropriate script with the correct command line arguments:

   a. On Windows:

      - `prompt> startwlevs.cmd`

   b. On UNIX:

      - `prompt> ./startwlevs.sh`

   After the server starts, you should see the following message printed to the output:

   ```
   SimpleEvent created at: 14:33:40.441
   ```

   This message indicates that the Oracle Event Processing server started correctly and that the `simpleEventSource` component is creating events.

3. Invoke the following URL in your browser:

```
http://host:port/wlevs
```

where *host* refers to the name of the computer on which Oracle Event Processing is running and *port* refers to the Jetty NetIO port configured for the server (default value `9002`).

The Logon screen displays.
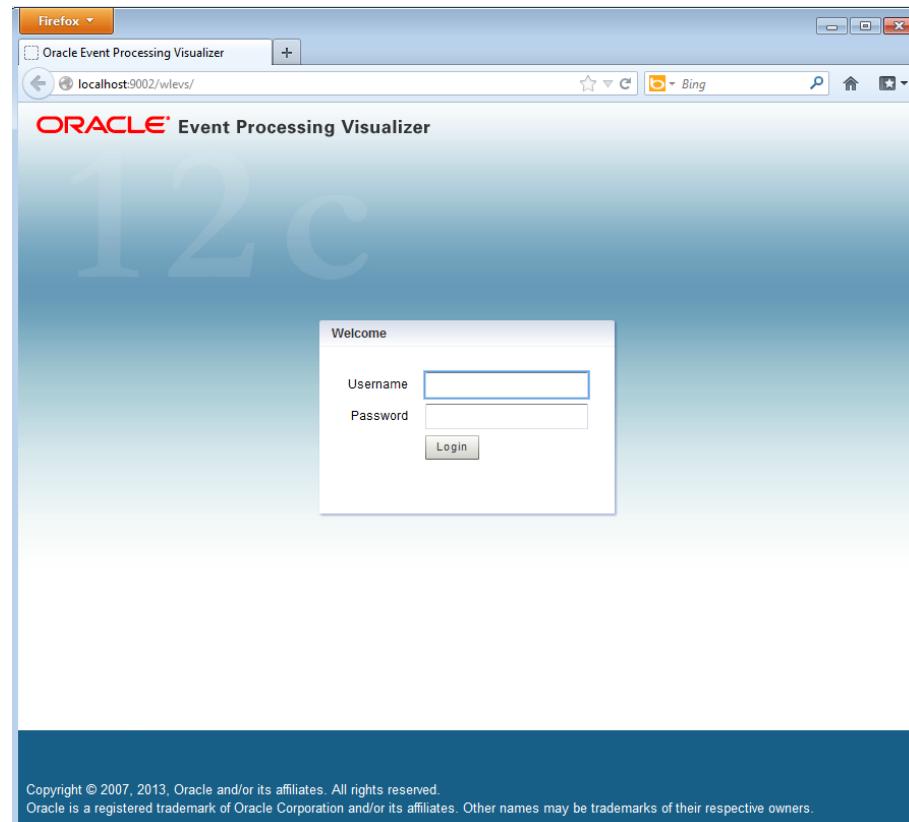
4. In the Logon screen, enter the **Username** `oepadmin` and **Password** `welcome1`, and click **Login**.

The Oracle Event Processing Visualizer dashboard displays.
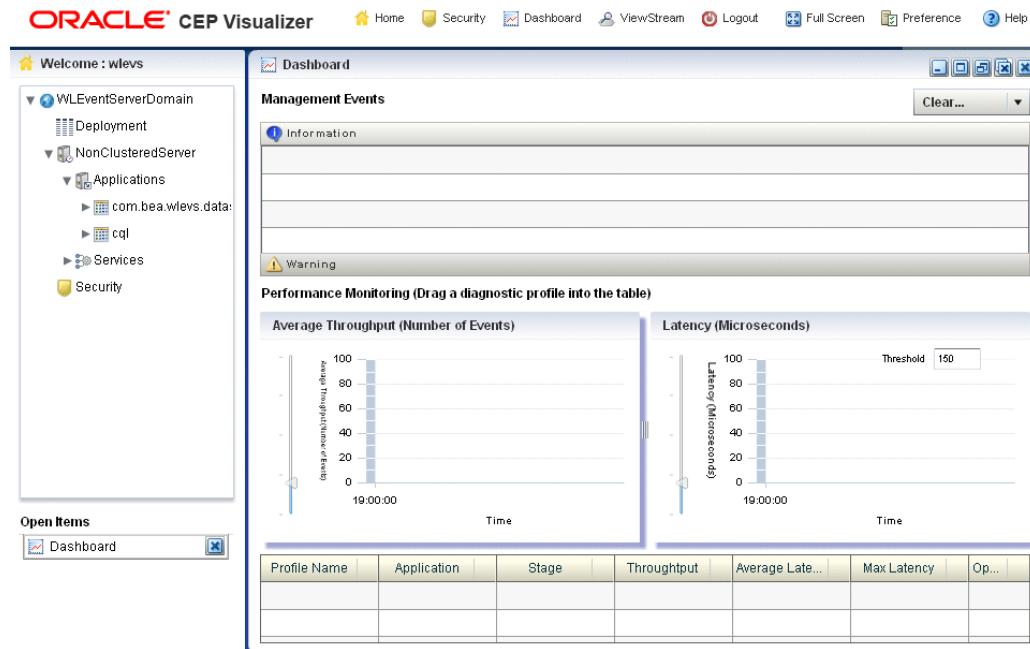
5. In the left panel, select **WLEventServerDomain** > **NonClusteredServer** > **Applications** > **recplay** > **stages** > **simpleEventSource**.

6. In the right panel, select the **Record** tab as follows:
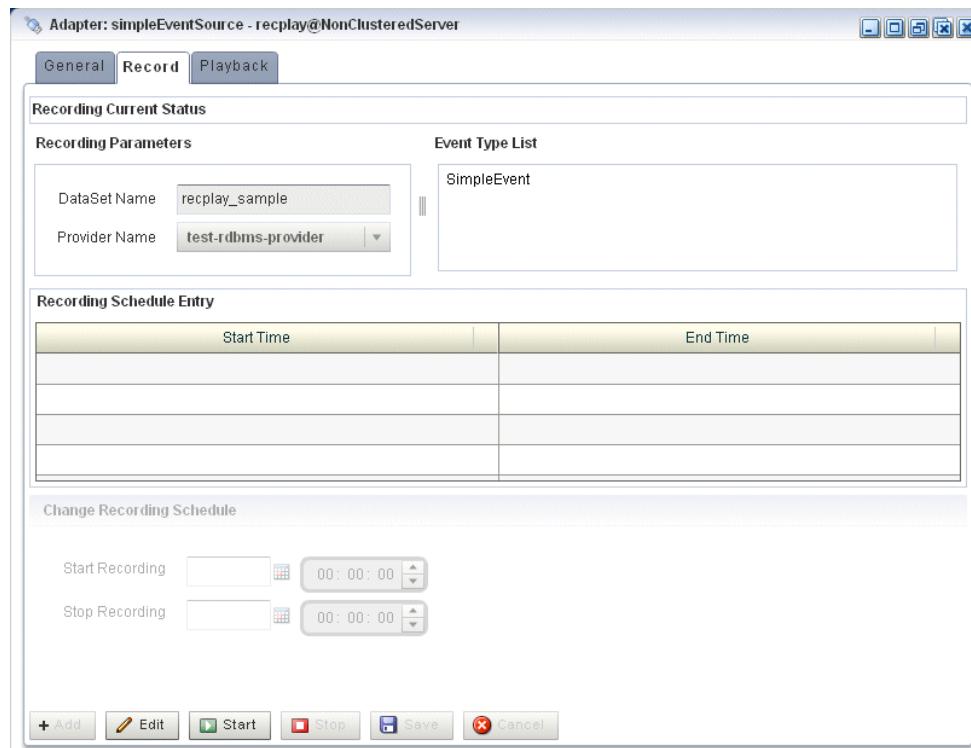
The **DataSet Name** field contains the value of the `record-parameters` child element `dataset-name` element from the `simpleEventSource` adapter application configuration file `/Oracle/Middleware/my_oep/examples/`

```
domains/recplay_domain/defaultserver/applications/recplay/
config.xml.

<adapter>
    <name>simpleEventSource</name>
    <record-parameters>
        <dataset-name>recplay_sample</dataset-name>
        <event-type-list>
            <event-type>SimpleEvent</event-type>
        </event-type-list>
        <batch-size>1</batch-size>
        <batch-time-out>10</batch-time-out>
    </record-parameters>
</adapter>
```

7. At the bottom of the Record tab, click **Start**.

An Alert dialog displays.

8. Click **OK**.

The Current Status field reads **Recording...**.

As soon as you click **OK**, events start to flow out of the `simpleEventSource` component and are stored in the configured database.

You can further configure when events are recorded using the **Start Recording** and **Stop Recording** fields.

9. In the left panel, select **eventStream**.

10. In the right panel, select the **Playback** tab as follows:
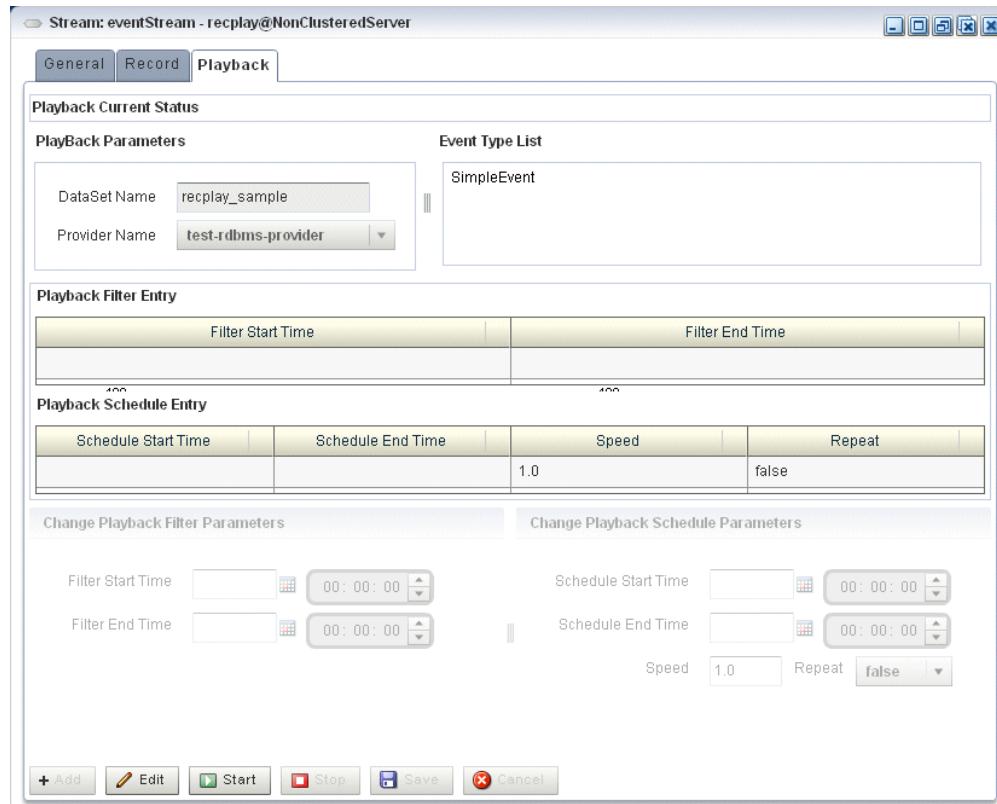
11. At the bottom of the tab, click **Start**.

    An Alert dialog appears as shown in Figure 6-8.

*Figure 6-8   Start Playback Alert Dialog*
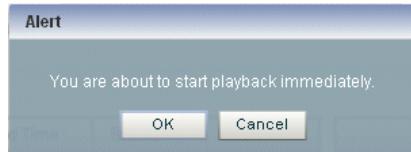
12. Click **OK**.

    The Current Status field reads **Playing...**.

    As soon as you click **OK**, events that had been recorded by the `simpleEventSource` component are now played back to the `simpleStream` component.

    You should see the following messages being printed to the command window from which you started Oracle Event Processing server to indicate that both original events and playback events are streaming through the EPN:

    ```
    SimpleEvent created at: 14:33:11.501
    Played back: Original time=14:15:23.141 Playback time=14:33:11.657
    ```

    You can further configure the playback parameters, such as the recorded time period for which you want playback events and the speed that they are played back, by updating the appropriate field and clicking **Change Parameters**. You must restart the playback after changing any playback parameters.

13. To view the events that the `playbackHttpPublisher` adapter is publishing to a channel, follow these steps:

a. In the top panel, select **Viewstream**.

The Viewstream window displays.
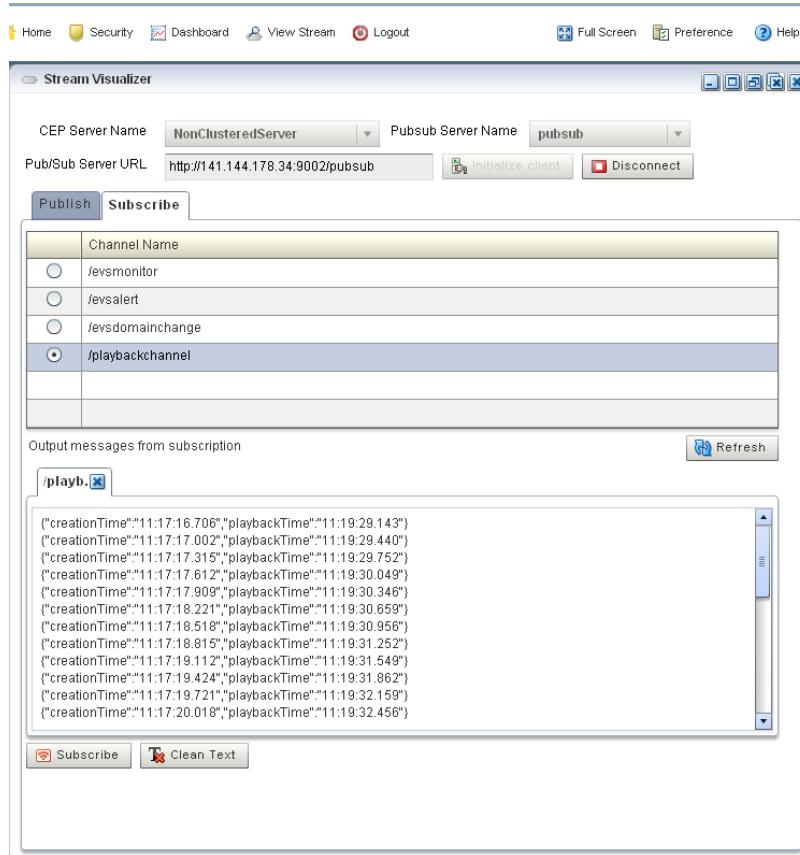
**b.** In the right panel, click **Initialize Client**.

c. In the Subscribe Channel text box, enter `/playbackchannel`.

d. Click **Subscribe**.

The **Received Messages** text box displays the played back event details. The played back events show the time at which the event was created and the time at which it was played back.

6.10.2 Build and Deploy the Event Record/Playback Example

The record and playback sample source directory contains the Java source, along with other required resources, such as configuration XML file and EPN assembly file that make up the application. The `build.xml` Ant file contains targets to build and deploy the application to the signalgeneration_domain domain, as described in Description of the Ant Targets to Build the Record and Playback Example.

Build and deploy the event record/playback example from the source directory:

1. If the record/playback Oracle Event Processing instance is not already running, follow the procedure in Run the Event Record/Playback Example to start the

server. You must have a running server to successfully deploy the rebuilt application.

2. Open a new command window and change to the record/playback source directory, located in */Oracle/Middleware/my_oep/oep/examples/source/ applications/recplay*.

3. Execute the `all` Ant target to compile and create the application JAR file:

```
prompt> ant all
```

4. Execute the `deploy` Ant target to deploy the application JAR file to the */Oracle/ Middleware/my_oep/examples/domains/recplay_domain/ defaultserver/applications/recplay* directory:

```
prompt> ant -Dusername=oepadmin -Dpassword=welcome1 -Daction=update deploy
```

> **Caution:**
>
> This target overwrites the existing event record/playback application JAR file in the domain directory.

After an application redeploy message, you should see the following message printed to the output about every second:

```
SimpleEvent created at: 14:33:40.441
```

This message indicates that the record and playback example has been redeployed and is running correctly.

5. Follow the instructions in Run the Event Record/Playback Example, starting at step 4, to invoke Oracle Event Processing Visualizer and start recording and playing back events.

6.10.3 Description of the Ant Targets to Build the Record and Playback Example

The `build.xml` file, located in the top-level directory of the record/playback source, contains the following targets to build and deploy the application:

- `clean`: This target removes the `dist` and `output` working directories under the current directory.

- `all`: This target cleans, compiles, and puts the application into a JAR file called `com.bea.wlevs.example.recplay_12.1.2.0_0.jar`, and places the generated JAR file into a `dist` directory below the current directory.

- `deploy`: This target deploys the JAR file to Oracle Event Processing using the Deployer utility.

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

6.10.4 Implementation of the Record and Playback Example

All the files of the example are located relative to the /Oracle/Middleware/my_oep/ *examples/source/applications/recplay* directory.

The files used by the record and playback example include:

- An EPN assembly file that describes each component in the application and how all the components are connected together as shown in Figure 6-7.

    In the example, the file is called `com.bea.wlevs.example.recplay-context.xml` and is located in the `META-INF/spring` directory.

- Java source file for the `simpleEventSource` adapter.

    In the example, the file is called `SimpleEventSource.java` and is located in the `~/src/com/bea/wlevs/adapter/example/recplay` directory.

    For a detailed description of how to program the adapter Java files in general, see Overview of Custom Adapters in Oracle Fusion Middleware Developing Application for Oracle Event Processing .

- Java source file that describes the `PlayedBackEvent` and `SimpleEvent` event types. The `SimpleEvent` event type is the one originally generated by the adapter, but the `PlayedBackEvent` event type is used for the events that are played back after having been recorded. The `PlayedBackEvents` look almost exactly the same as `SimpleEvent` except they have an extra field, the time the event was recorded.

    In the example, the two events are called `SimpleEvent.java` and `PlayedBackEvent.java` and are located in the `~/src/com/bea/wlevs/event/example/recplay` directory.

    For a detailed description of this file, as well as general information about programming event types, see *Overview of Oracle Event Processing Event Types* in *Developing Applications for Event Processing with Oracle Stream Explorer*.

- A Java file that implements the `recplayEventSink` event bean of the application, which is an event sink that receives both realtime events from the `simpleEventSource` adapter as well as playback events.

    In the example, the file is called `RecplayEventSink.java` and is located in the `~/src/com/bea/wlevs/example/recplay` directory.

    For more information about event sources and sinks, see Handling Events with Sources and Sinks in Oracle Fusion Middleware Developing Application for Oracle Event Processing .

- An XML file that configures the `simpleEventSource` adapter and `eventStream` channel components. The adapter includes a `<record-parameters>` element that specifies that the component will record events to the event store; similarly, the channel includes a `<playback-parameters>` element that specifies that it receives playback events.

    In the example, the file is called `config.xml` and is located in the `~/META-INF/wlevs` directory.

- A `MANIFEST.MF` file that describes the contents of the OSGi bundle that will be deployed to Oracle Event Processing.

    In the example, the `MANIFEST.MF` file is located in the `META-INF` directory

    For more information about creating this file, as well as a description of creating the OSGi bundle that you deploy to Oracle Event Processing, see Overview of Application Assembly and Deployment.

The record/playback example uses a `build.xml` Ant file to compile, assemble, and deploy the OSGi bundle; see Build and Deploy the Event Record/Playback Example for a description of this `build.xml` file if you also use Ant in your development environment.

Glossary

Adapter

An element of the EPN that interfaces directly to an inbound event source. Adapters understand the inbound protocol, and are responsible for converting the event data into a normalized form that can be queried by a POJO. Adapters forward the normalized event data into a Stream.

Aggregate Function

Aggregate functions return a single aggregate result based on group of tuples, rather than on a single tuple.

See also Function and Single-Row Function.

OEP

Oracle Event Processing.

Channel

A channel represents the physical conduit through which events flow between other types of components, such as between an Adapter and a Processor, and between a Processor and an Event Bean. A channel can model a Stream or Relation.

Condition

An Oracle CQL condition specifies a combination of one or more expressions and logical (Boolean) operators and returns a value of `TRUE`, `FALSE`, or `UNKNOWN`.

Constant value

A fixed data value. Synonymous with Literal.

CQL

Oracle Continuous Query Language. Supersedes EPL.

Data Feed

A synonym for Event Source.

Destination

An Oracle CQL destination identifies a consumer of query results such as the Enterprise Link BAM Adapter, JMS queue or topic, or file.

Deterministic Garbage Collection

Short, predictable pause times for memory heap garbage collection, which is the process of clearing dead objects from the heap, thus releasing that space for new objects.

DStream

A relation-to-stream operator that represents deleted tuples.

EDA

Event-Driven Architecture.

EPL

Oracle Event Processing Language. Superseded by CQL.

EPN

Oracle Event Processing Network. An EPN is the arbitrary interconnection of Adapter, Stream, POJO, and business logic POJOs used by Oracle Event Processing to process events.

Event Bean

A POJO to that contains the business logic executed when a notable event is detected. An event bean is an Event Sink.

Event Rule

A query, expressed in CQL or EPL, executed by a POJO to filter and aggregate events.

Event Sink

A component that consumes events, such as a Processor.

See also Event Source.

Event Source

A component that provides events, such as a sensor, wire service, or stock ticker.

See also Data Feed and Event Sink.

Expressions

An Oracle CQL expression is a combination of one or more values, operators, and Oracle CQL functions that evaluates to a value. An expression generally assumes the data type of its components.

See also Condition and Function.

Format model

A character **literal** that describes the format of date-time or numeric data stored in a character string.

Function

Oracle CQL functions are similar to operators in that they manipulate data items and return a result. Functions differ from operators in the format of their arguments. This format enables them to operate on zero, one, two, or more arguments.

See also Condition, Aggregate Function, and Single-Row Function.

Incremental Processing

A user-defined aggregate function design pattern that improves scalability and performance by ensuring that the cost of (re)computation on arrival of new events will be proportional to the number of new events as opposed to the total number of events seen thus far.

If your user-defined aggregate function supports incremental processing, you specify the `supports incremental processing` clause in the `register function` statement to instruct the Oracle Event Processing Service Engine to supply only the new event data as opposed to performing a rescan over already processed event data.

IStream

A relation-to-stream operator that represents inserted tuples.

Join

A query that combines rows from two or more streams, views, or relations.

Latency

An expression of how much time it takes for data to get from one designated point to another.

Literal

A fixed data value. Synonymous with Constant value.

Monotonic

A sequence of values that are consistently increasing and never decreasing or consistently decreasing and never increasing. The sequence may contain multiple consecutive occurrences of the same value.

Now window

A special case of the time-based sliding window on a stream S that takes a time-interval T as a parameter and is specified by: S [Range T]. A Now window is defined as: S [Now] (short for S [Range 0]). When $T = 0$, the relation at time t consists of tuples obtained from elements of S with timestamp t.

See also Sliding window.

Operators

Oracle CQL operators manipulate data items and return a result. Syntactically, an operator appears before or after an operand or between two operands.

OSGi

A dynamic module system for Java that provides a service-oriented, component-based environment and standardized software life cycle management. Oracle Event Processing applications are packaged and deployed as OSGi bundles. For more information, see http://www.osgi.org/.

Partitioned window

A partitioned sliding window on a stream S takes a positive integer number of tuples N and a subset {A1,... Ak} of the stream's attributes as parameters and is specified by: S[Partition By A1 ... Ak Rows N] or, optionally, S[Partition By A1 ... Ak Rows N Range T].

See also Sliding window.

POJO

A Plain Old Java Object. A Java class that is not required to implement a third-party interface or extend a third-party class. In Oracle Event Processing, you can express your business logic using POJOs.

Processor

An element of the EPN that consumes normalized event data from a stream, processes it using queries (expressed in CQL or EPL), and may generate new events to an output stream.

Query

A query is an operation that retrieves data from one or more streams or views. In this reference, a top-level SELECT statement is called a **query**.

Real-time

A level of computer responsiveness that a user senses as sufficiently immediate or that enables the computer to keep up with some external process (for example, to present visualizations of the weather as it constantly changes).

Relation

A relation is time-varying bag of tuples. Here time refers to an instant in the time domain. At every instant of time, a relation is a bounded set. It can also be represented as a sequence of times tamped tuples that includes insertions, deletions, and updates to capture the changing state of the relation. The updates are required to arrive at the system in the order of increasing timestamps. Like streams, relations have a fixed schema to which all tuples conform.

RStream

A relation-to-stream operator that maintains the entire current state of its input relation and outputs all of the tuples as insertions at each time step.

Single-Row Function

Single-row functions return a single result row for every row of a queried stream or view.

See also Function and Aggregate Function.

Sliding window

A stream-to-relation operator based on the window specification derived from SQL99.

See also: Now window, Partitioned window, Unbounded window_ tuple-based, and Unbounded window_ time-based.

Source

An Oracle CQL source identifies a producer of data that a Oracle CQL query operates on such as the Enterprise Link BAM Adapter, JMS queue or topic, or file.

Spring Framework

A light-weight, open source application framework for Java. Oracle Event Processing server uses the Spring Framework to host Oracle Event Processing applications. For more information, see http://www.springframework.org/.

Stream

A stream is a sequence of times tamped tuples. There could be more than one tuple with the same timestamp. The tuples of an input stream are required to arrive at the system in the order of increasing timestamps. A stream has an associated schema consisting of a set of named attributes, and all tuples of the stream conform to the schema.

A stream is a bag (or multi-set) of tuple-timestamp pairs, which can be represented as a sequence of times tamped tuple insertions.

In Oracle Event Processing, a stream is modeled as a channel component.

See also Tuple and Channel.

Throughput

An Oracle CQL source identifies a producer of data that a Oracle CQL query operates on such as the Enterprise Link BAM Adapter, JMS queue or topic, or file.

Tuple

The phrase *tuple of a stream* denotes the ordered list of data (excluding timestamp data) portion of a stream element (the s of <s,t>). For example, a stock ticker data stream might appear like this where each stream element is made up of <timestamp value>, <stock symbol>, and <stock price>:

```
...
<timestampN>    NVDA,4
<timestampN+1>  ORCL,62
<timestampN+2>  PCAR,38
<timestampN+3>  SPOT,53
<timestampN+4>  PDCO,44
<timestampN+5>  PTEN,50
...
```

In the stream element <timestampN+1> ORCL,62, the tuple is ORCL,62.

See also Stream.

Unbounded window, time-based

A special case of the time-based sliding window on a stream S that takes a time-interval T as a parameter and is specified by: S [Range T]. An Unbounded window is defined as: S [Range Unbounded] (short for S [Range infinity]). When T = infinity, the relation at time t consists of tuples obtained from all elements of S up to t.

See also Sliding window.

Unbounded window, tuple-based

A special case of the tuple-based sliding window on a stream S that takes a number of tuples N as a parameter and is specified by: S [Rows N]. An Unbounded window is defined as: S [Rows Unbounded] (short for S [Rows infinity] and equivalent to S [Range Unbounded]). When T = infinity, the relation at time t consists of tuples obtained from all elements of S up to t.

See also Sliding window.

View

An Oracle CQL view represents an alternative selection on a stream or relation. In Oracle CQL, you use a view instead of a subquery.