

# **Oracle® Complex Event Processing**

Application Development Guide

Release 10gR3 (10.3)

September 2008

Copyright © 2007, 2008, Oracle and/or its affiliates. All rights reserved.

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

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

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

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

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

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

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

# Contents

## 1. Introduction and Roadmap

Document Scope and Audience . . . . .	1-1
Oracle CEP Documentation Set . . . . .	1-2
Guide to This Document . . . . .	1-2
Samples for the Oracle CEP Application Developer . . . . .	1-3

## 2. Overview of Creating Oracle Complex Event Processing Applications

Overview of the Oracle Complex Event Processing Programming Model . . . . .	2-1
Components of the Oracle CEP Event Processing Network . . . . .	2-2
Event Sources and Event Sinks . . . . .	2-3
Component Configuration Files . . . . .	2-3
How Components Fit Together . . . . .	2-4
Application Lifecycle . . . . .	2-5
Oracle CEP APIs . . . . .	2-7
Oracle CEP Development Environment for Eclipse . . . . .	2-8
Downloading the Latest Oracle CEP IDE . . . . .	2-9
Installing the Oracle CEP IDE Distributed With the Product . . . . .	2-9
Creating Oracle CEP Applications: Typical Steps . . . . .	2-10
Creating the EPN Assembly File . . . . .	2-11
Creating the Event Types . . . . .	2-15
Using an Event Type Builder Factory . . . . .	2-17

Sharing Event Types Between Application Bundles . . . . .	2-19
Next Steps . . . . .	2-19

## 3. Creating Custom Adapters and Event Beans

Overview of Adapters . . . . .	3-2
Overview of Event Beans . . . . .	3-2
Overview of Event Sources and Event Sinks . . . . .	3-3
Standard Spring Beans as Event Sources. . . . .	3-3
Event Beans as Event Sources . . . . .	3-3
Event Sinks . . . . .	3-4
Overview of Adapter and Event Bean Factories. . . . .	3-4
Creating Adapters or Event Beans: Typical Steps . . . . .	3-5
Programming Adapters and Event Bean Classes as Event Sources: Guidelines . . . . .	3-5
Programming Event Sinks: Guidelines . . . . .	3-8
Programming the Optional Factory Class . . . . .	3-10
Accessing a Relational Database. . . . .	3-11
Updating the EPN Assembly File . . . . .	3-11
Registering the Adapter or Event Bean Factory . . . . .	3-11
Declaring the Adapter and Event Bean Components in your Application . . . . .	3-12
Configuring the Adapter or Event Bean . . . . .	3-13
Example of an Adapter Configuration File . . . . .	3-15
Creating an Adapter or Event Bean in Its Own Bundle . . . . .	3-15
Extending the Configuration of an Adapter or Event Bean . . . . .	3-17
Creating the XSD Schema File . . . . .	3-19
Complete Example of an Extended XSD Schema File . . . . .	3-21
Programming Access to the Configuration of an Adapter or Event Bean. . . . .	3-22
Passing Login Credentials from an Adapter to the Data Feed Provider . . . . .	3-23
Updating the Adapter Code to Access the Login Credential Properties . . . . .	3-27

## 4. Using the Java Message Service (JMS) Adapters

Overview of Using the JMS Adapters . . . . .	4-1
Using JMS Adapters: Typical Steps . . . . .	4-2
Creating a Custom Converter Between JMS Messages and Event Types . . . . .	4-3
Updating the EPN Assembly File With JMS Adapters . . . . .	4-4
Configuring the JMS Adapters . . . . .	4-7
Encrypting Passwords in the JMS Adapter Configuration File . . . . .	4-11

## 5. Using and Creating HTTP Publish-Subscribe Adapters

Overview of HTTP Publish-Subscribe Functionality in Oracle CEP . . . . .	5-1
Converting Between JSON Messages and Event Types . . . . .	5-2
Overview of the Built-In Pub-Sub Adapter for Local Publishing . . . . .	5-3
Overview of the Built-In Pub-Sub Adapter for Remote Publishing . . . . .	5-5
Overview of the Built-In Pub-Sub Adapter for Subscribing . . . . .	5-7
Using the Built-In HTTP Pub-Sub Adapters in an Application: Typical Steps . . . . .	5-9
Creating a Custom Converter Between the HTTP Pub-Sub Messages and Event Types	5-10
Configuring the HTTP Pub-Sub Adapters . . . . .	5-11
Updating the EPN Assembly File . . . . .	5-15

## 6. Configuring the Stream Component

Overview of the Stream Configuration File . . . . .	6-1
Creating the Stream Configuration File: Main Steps . . . . .	6-2
Example of an Stream Configuration File . . . . .	6-4

## 7. Configuring the Complex Event Processor

Overview of the Complex Event Processor Configuration File . . . . .	7-1
Configuring the Complex Event Processor: Main Steps . . . . .	7-2
Example of a Processor Configuration File . . . . .	7-5

## 8. Using Oracle CEP Caching

Overview of Oracle Complex Event Processing Caching .....	8-1
Use Case: Publishing Events to a Cache .....	8-2
Use Case: Consuming Data From a Cache .....	8-2
Use Case: Updating and Deleting Data in a Cache .....	8-3
Additional Caching Features .....	8-3
Caching APIs .....	8-3
Typical Steps to Use Oracle CEP Caching .....	8-4
Configuring the Oracle CEP Caching System and Caches .....	8-5
Declaring the Caching System in the EPN Assembly File .....	8-9
Declaring a Cache in the EPN Assembly File .....	8-10
Configuring and Programming a Business POJO to Access a Cache .....	8-11
Configuring and Programming an Adapter to Access a Cache .....	8-12
Configuring and Programming a User-Defined EPL Function to Access a Cache ..	8-13
Configuring a Cache as a Listener .....	8-15
Specifying the Key Used to Index a Cache .....	8-15
Specifying a Key Property in EPN Assembly File.....	8-16
Using a Metadata Annotation to Specify a Key .....	8-17
Specifying a Composite Key .....	8-17
Configuring a Cache as an Event Source. ....	8-18
Configuring a Cache Loader .....	8-18
Configuring a Cache Store .....	8-19
Referencing a Cache from an EPL Statement. ....	8-19
Restrictions When Using a Cache in an EPL Statement .....	8-20
Typical Steps To Reference a Cache in an EPL Statement .....	8-20
Coherence Support .....	8-21
Declaring the Coherence Provider in the EPN Assembly File .....	8-22
Configuring the Coherence Caching System .....	8-23

The coherence-cache-config.xml File . . . . .	8-23
The tangasol-coherence-override.xml File . . . . .	8-26

## 9. Using a Persistent Store to Record and Playback Events

Overview of Using a Persistent Store to Record and Playback Events . . . . .	9-1
The Event Store . . . . .	9-2
Recording Events . . . . .	9-2
Playing Back Events . . . . .	9-3
Querying Stored Events . . . . .	9-4
Restrictions on the Event Types that Can Be Recorded . . . . .	9-4
Record and Playback Example . . . . .	9-4
Implementing Record and Playback of Events in Your Application: Typical Steps . . . .	9-4
Configuring an Event Store for Oracle CEP Server . . . . .	9-5
Configuring a Component to Store Events . . . . .	9-7
Configuring a Component to Playback Events . . . . .	9-11
Starting and Stopping the Record and Playback of Events . . . . .	9-15
Description of the Database Tables Created by the RDMBS Provider . . . . .	9-16
Creating a Custom Event Store Provider . . . . .	9-17

## 10. Assembling and Deploying Oracle Complex Event Processing Applications

Overview of Application Assembly and Deployment . . . . .	10-1
Assembling an Oracle CEP Application: Main Steps . . . . .	10-2
Creating the MANIFEST.MF File . . . . .	10-4
Accessing Third-Party JAR Files From Your Application . . . . .	10-6
Deploying Oracle CEP Applications: Main Steps . . . . .	10-7

## 11. Using the Load Generator to Test Your Application

Overview of the Load Generator Utility . . . . .	11-1
--	------

Creating a Load Generator Property File .....	11-2
Creating a Data Feed File .....	11-4
Configuring the csvgen Adapter in Your Application .....	11-4

## A. Additional Information about Spring and OSGi



# Introduction and Roadmap

This section describes the contents and organization of this guide—*Oracle Complex Event Processing Application Development Guide*.

**Note:** In this section, *Oracle Complex Event Processing* is also referred to as *Oracle CEP*, for simplicity.

- [“Document Scope and Audience” on page 1-1](#)
- [“Oracle CEP Documentation Set” on page 1-2](#)
- [“Guide to This Document” on page 1-2](#)
- [“Samples for the Oracle CEP Application Developer” on page 1-3](#)

## Document Scope and Audience

This document is a resource for software developers who develop event driven real-time applications. It also contains information that is useful for business analysts and system architects who are evaluating Oracle CEP or considering the use of Oracle CEP for a particular application.

The topics in this document are relevant during the design, development, configuration, deployment, and performance tuning phases of event driven applications. The document also includes topics that are useful in solving application problems that are discovered during test and pre-production phases of a project.

It is assumed that the reader is familiar with the Java programming language and Spring.

# Oracle CEP Documentation Set

This document is part of a larger Oracle CEP documentation set that covers a comprehensive list of topics. The full documentation set includes the following documents:

- *[Oracle CEP Getting Started](#)*
- *[Oracle CEP Application Development Guide](#)*
- *[Oracle CEP Administration and Configuration Guide](#)*
- *[Oracle CEP EPL Reference Guide](#)*
- *[Oracle CEP Reference Guide](#)*
- *[Oracle CEP Release Notes](#)*
- *[Oracle CEP Visualizer Help](#)*
- *[Oracle CEP Type 4 JDBC Drivers](#)*

See the main [Oracle CEP documentation page](#) for further details.

## Guide to This Document

This document is organized as follows:

- This chapter, [Chapter 1, “Introduction and Roadmap,”](#) introduces the organization of this guide and the Oracle CEP documentation set and samples.
- [Chapter 2, “Overview of Creating Oracle Complex Event Processing Applications,”](#) describes at a high-level the programming model used to create Oracle CEP applications. It provides a procedure that lists the typical steps a programmer goes through to create an application.
- [Chapter 3, “Creating Custom Adapters and Event Beans,”](#) describes how to create and configure the adapter components of an Oracle CEP application.
- [Chapter 4, “Using the Java Message Service \(JMS\) Adapters,”](#) describes how to use the built-in JMS adapter.
- [Chapter 5, “Using and Creating HTTP Publish-Subscribe Adapters,”](#) describes how to use the built-in HTTP publish-subscribe adapters, as well as create your own custom adapter.

- [Chapter 6, “Configuring the Stream Component,”](#) describes how to optionally configure the stream components of an Oracle CEP.
- [Chapter 7, “Configuring the Complex Event Processor,”](#) describes how to configure the complex event processor component of an Oracle CEP application.
- [Chapter 8, “Using Oracle CEP Caching,”](#) describes how to use the event caching feature to increase the performance of your applications.
- [Chapter 9, “Using a Persistent Store to Record and Playback Events,”](#) describes how to record events to a physical store, and then later play them back.
- [Chapter 10, “Assembling and Deploying Oracle Complex Event Processing Applications,”](#) describes how to assemble all the components of an application into a deployable bundle, and then how to deploy the bundle to Oracle CEP. After you have deployed the application you can start executing it.
- [Chapter 11, “Using the Load Generator to Test Your Application,”](#) provides detailed information for using the load generator, an Oracle CEP testing tool.
- [Appendix A, “Additional Information about Spring and OSGi,”](#) provides links to additional non-Oracle information about Spring and OSGI.

## Samples for the Oracle CEP Application Developer

In addition to this document, Oracle provides a variety of code samples for Oracle CEP application developers. The examples illustrate Oracle CEP in action, and provide practical instructions on how to perform key development tasks.

Oracle recommends that you run some or all of the examples before programming and configuring your own event driven application.

**Note:** When you initially install Oracle CEP, you must chose the `Custom` option to also install the examples. The `Typical` option does *not* include the examples.

If you previously installed Oracle CEP using the `Typical` option, and you now want to also install the examples, re-run the Oracle CEP installation process and specify the same Oracle CEP home directory; a later step in the installation process allows you to then install just the examples.

The examples are distributed in two ways:

- Pre-packaged and compiled in their own domain so you can immediately run them after you install the product.

- Separately in a Java source directory so you can see a typical development environment setup.

The following four examples are provided in both their own domain and as Java source in this release of Oracle CEP:

- HelloWorld—Example that shows the basic elements of an Oracle CEP application. See [Hello World Example](#) for additional information.

The HelloWorld domain is located in

`ORACLE_CEP_HOME\ocep_10.3\samples\domains\helloworld_domain`, where `ORACLE_CEP_HOME` refers to the Oracle CEP installation directory, such as `c:\oracle_cep`.

The HelloWorld Java source code and configuration files are located in

`ORACLE_CEP_HOME\ocep_10.3\samples\source\applications\helloworld`.

- ForeignExchange (FX)—Example that includes multiple adapters, streams, and complex event processor with a variety of EPL rules, all packaged in the same Oracle CEP application. See [Foreign Exchange \(FX\) Example](#) for additional information.

The ForeignExchange domain is located in

`ORACLE_CEP_HOME\ocep_10.3\samples\domains\fx_domain`, where `ORACLE_CEP_HOME` refers to the Oracle CEP installation directory, such as `c:\oracle_cep`.

The ForeignExchange Java source code and configuration files are located in

`ORACLE_CEP_HOME\ocep_10.3\samples\source\applications\fx`.

- Signal Generation—Example that receives simulated market data and verifies if the price of a security has fluctuated more than two percent, and then detects if there is a *trend* occurring by keeping track of successive stock prices for a particular symbol. See [Signal Generation Example](#) for additional information.

The Signal Generation domain is located in

`ORACLE_CEP_HOME\ocep_10.3\samples\domains\signalgeneration_domain`, where `ORACLE_CEP_HOME` refers to the Oracle CEP installation directory, such as `c:\oracle_cep`.

The Signal Generation Java source code and configuration files are located in

`ORACLE_CEP_HOME\ocep_10.3\samples\source\applications\signalgeneration`.

- Record and Playback—Example that shows how to configure the recording and playback of events to a persistent event store, as well as how to use the built-in HTTP pub-sub

adapter to publish messages to a channel. See [Event Record and Playback Example](#) for additional information.

The Record and Playback domain is located in

`ORACLE_CEP_HOME\ocp_10.3\samples\domains\replay_domain`, where `ORACLE_CEP_HOME` refers to the Oracle CEP installation directory, such as `c:\oracle_cep`.

The Record and Playback Java source code and configuration files are located in

`ORACLE_CEP_HOME\ocp_10.3\samples\source\applications\replay`.



# Overview of Creating Oracle Complex Event Processing Applications

This section contains information on the following subjects:

- [“Overview of the Oracle Complex Event Processing Programming Model” on page 2-1](#)
- [“Oracle CEP Development Environment for Eclipse” on page 2-8](#)
- [“Creating Oracle CEP Applications: Typical Steps” on page 2-10](#)
- [“Creating the EPN Assembly File” on page 2-11](#)
- [“Creating the Event Types” on page 2-15](#)
- [“Next Steps” on page 2-19](#)

## Overview of the Oracle Complex Event Processing Programming Model

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

Oracle CEP defines a set of core services or components used together to assemble event-driven applications; the typical services are adapters, streams, and processors. You can also create your

own business logic POJOs and Spring beans that are part of the application, as well as specialized event beans that are just like Spring beans but with full access to the Oracle CEP framework, such as monitoring and record/playback of events. In addition to these, Oracle CEP includes other infrastructure services, such as caching, clustering, configuration, monitoring, logging, and so on.

All services are deployed on the underlying Oracle microServices Architecture (mSA) technology. Oracle mSA is based upon the [OSGi Service Platform defined by the OSGi Alliance](#).

The following sections provide additional information about the Oracle CEP programming model and creating applications:

- [“Components of the Oracle CEP Event Processing Network” on page 2-2](#)
- [“Event Sources and Event Sinks” on page 2-3](#)
- [“Component Configuration Files” on page 2-3](#)
- [“How Components Fit Together” on page 2-4](#)
- [“Oracle CEP APIs” on page 2-7](#)

## Components of the Oracle CEP Event Processing Network

Oracle CEP applications and their event processing networks (EPNs) are made up of the following basic components:

- **Adapters**—Components that provide an interface to incoming data feeds and convert the data into event types that the Oracle CEP application understands. Adapters can be both incoming (receive data) and outgoing (send data). Oracle CEP includes some built-in adapters, such as HTTP publish-subscribe adapters. Event beans are functionally the same as adapters, but typically event beans are used for intermediate components of the EPN and adapters on the outskirts of an EPN.
- **Event beans**—A bean that is managed by Oracle CEP. It is analogous to a Spring-bean, which is managed by the Spring framework. Event beans are functionally the same as adapters, but typically event beans are used for intermediate components of the EPN and adapters on the outskirts of an EPN.
- **Streams**—Components that function as virtual pipes or channels, connecting components that send events with components that receive events.
- **Processors**—Components that execute user-defined event processing rules against streams.

The user-defined rules are written using the Event Processing Language (EPL).



- **Standard Spring beans**—User-coded POJO. These components are not managed by the Oracle CEP; for example, you cannot monitor these components, record and playback of events, and so on. If you require this additional functionality for you POJO, consider creating an event bean instead.
- **Caches**—Temporary storage area for events, created exclusively to improve the overall performance of your application.

## Event Sources and Event Sinks

All components in an EPN are either event sources (emit events) or event sinks (receive events.) Some components can be both, such as an event bean in the middle of an EPN that receives events from an adapter, for example, and then sends events to a processor.

Components that emit events must implement the `com.bea.wlevs.ede.api.EventSource` interface. This interface has one method, `setEventSender()`; at runtime the event source component is injected with an `com.bea.wlevs.ede.api.EventSender` instance. This `EventSender` instance, in turn, has a `sentEvent()` method that the component invokes to actually send events to the next component in the EPN.

Components that receive events from event sources must implement the `com.bea.wlevs.ede.api.EventSink` interface. This interface has a single callback method, `onEvent()`, that event sinks implement to receive a list of events from the previous component in the EPN.

## Component Configuration Files

Each component in your event processing network (adapter, processor, stream, or event bean) can have an associated configuration file, although only processors are *required* to have a configuration file. The caching system also uses a configuration file, regardless of whether it is a stage in the event processing network. Component configuration files in Oracle CEP are XML documents whose structure is defined using standard XML Schema. You create a single file that contains configuration for all components in your application, or you can create separate files for each component; the choice depends on which is easier for you to manage.

The following two schema documents define the default structure of application configuration files:

- [wlevs\\_base\\_config.xsd](#): Defines common elements that are shared between application configuration files and the server configuration file.

- `wlevs_application_config.xsd`: Defines elements that are specific to application configuration files.

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

```
<?xml version="1.0" encoding="UTF-8"?>

<helloworld:config
  xmlns:helloworld="http://www.bea.com/ns/wlevs/example/helloworld">
  <processor>
    <name>helloworldProcessor</name>
    <rules>
      <rule id="helloworldRule"><![CDATA[ select * from HelloWorldEvent
retain 1 event ]]></rule>
    </rules>
  </processor>

  <adapter>
    <name>helloworldAdapter</name>
    <message>HelloWorld - the current time is:</message>
  </adapter>

  <stream monitoring="true" >
    <name>helloworldOutstream</name>
    <max-size>10000</max-size>
    <max-threads>2</max-threads>
  </stream>
</helloworld:config>
```

## How Components Fit Together

Oracle CEP applications are made of services that are assembled together to form an Event Processing Network (EPN).

The server uses the Spring framework as its assembly mechanism due to Spring's popularity and simplicity. Oracle CEP has extended the Spring framework to further simplify the process of assembling applications. This approach allows Oracle CEP applications to be easily integrated

with existing Spring-beans, and other light-weight programming frameworks that are based upon a dependency injection mechanism.

A common approach for dependency injection is the usage of XML configuration files to declaratively specify the dependencies and assembly of an application. You assemble an Oracle CEP application an EPN assembly file before deploying it to the server; this EPN assembly file is an extension of the Spring framework XML configuration file.

After an application is assembled, it must be package so that it can be deployed into Oracle CEP. This is a simple process. The deployment unit of an application is a plain JAR file, which must contain, at a minimum, the following artifacts:

- The compiled application Java code of the business logic POJO.
- Component configuration files. Each processor is required to have a configuration file, although adapters and streams do not need to have a configuration file if the default configuration is adequate and you do not plan to monitor these components.
- The EPN assembly file.
- A MANIFEST.MF file with some additional OSGi entries.

After you assemble the artifacts into a JAR file, you deploy this bundle to Oracle CEP so it can immediately start receiving incoming data.

## Application Lifecycle

This section describes the lifecycle of an application deployed to Oracle CEP and the sequence of `com.bea.wlevs.ede.api` API callbacks. The lifecycle description is broken down into actions that a user performs, such as install an application, suspend it, resume it, and so on.

The point of this section is to show you how Oracle CEP manages an application's lifecycle so that you can better use the lifecycle APIs in your application. See [“Oracle CEP APIs” on page 2-7](#) for a brief description of these lifecycle APIs, such as `RunnableBean` and `SuspendableBean`, as well as the [Javadocs](#).

The application states referenced in this section are returned by the `getState()` method on the `ApplicationMBean`. These states are specific to Oracle CEP; they are not OSGi bundle states. The possible states are: UNINSTALLED, INITIALIZING, RUNNING, and SUSPENDED.

**User action: Install an application or start the server with application already deployed**

- Oracle CEP installs the application as an OSGI bundle. OSGI resolves the imports and exports, and publishes the service.
- Oracle CEP creates beans (for both standard Spring beans and those that correspond to the Oracle CEP tags in the EPN assembly file). For each bean, Oracle CEP:
  - Sets the properties on the Spring beans. The `<wlevs:instance-property>` values are set on adapters and event-beans.
  - Injects appropriate dependencies into services specified by `@Service` or `@ServiceReference` annotations.
  - Injects appropriate dependencies into static configuration properties.
  - Calls `InitializingBean.afterPropertiesSet()`.
  - Calls configuration callbacks (`@Prepare`, `@Activate`) on Spring beans as well as factory-created stages.
- Application state is now **INITIALIZING**.
- Oracle CEP registers the MBeans.
- Oracle CEP calls `ActivatableBean.afterConfigurationActive()` on all `ActivatableBeans`
- Oracle CEP calls `ResumableBean.beforeResume()` on all `ResumableBeans`
- For each bean that implements `RunnableBean`, Oracle CEP starts it running in a thread.
- Application state is now **RUNNING**.

### User action: Suspend application.

- Oracle CEP calls `SuspendableBean.suspend()` on all `SuspendableBeans`.
- Application state is now **SUSPENDED**.

### User action: Resume application.

- Oracle CEP calls `ResumableBean.beforeResume()` on all `ResumableBeans`
- For each bean that implements `RunnableBean`, Oracle CEP starts it running in a thread.
- Application state is now **RUNNING**.

**User action: Uninstall application.**

- Oracle CEP calls `SuspendableBean.suspend()` on all `SuspendableBeans`.
- Oracle CEP unregisters MBeans.
- Oracle CEP calls `DisposableBean.dispose()` on all `DisposableBeans`.
- Oracle CEP uninstalls application bundle from OSGI.

**User action: Update application.**

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

## Oracle CEP APIs

Oracle CEP provides a variety of Java APIs that you use in your adapter or event bean implementation.

This section describes the APIs that you will most typically use in your adapters and event beans; see the [Javadoc](#) for the full reference documentation for all classes and interfaces. See “[Creating Custom Adapters and Event Beans](#)” on page 3-1 as well as the HelloWorld and FX examples in the installed product, for sample Java code that uses these APIs.

- `com.bea.wlevs.ede.api.EventSink`—Components that receive events from an `EventSource`, such as the business logic POJO, must implement this interface. The interface has a callback method, `onEvent()`, in which programmers put the code that handles the received events.
- `EventSource`—Components that send events, such as adapters, must implement this interface. The interface has a `setEventSender()` method for setting the `EventSender`, which actually sends the event to the next component in the network.
- Component life cycle interfaces—If you want some control over the life cycle of the component you are programming, then your component should implement one or more of the following interfaces:
  - `DisposableBean`—Use if you want to release resources when the application is undeployed. Implement the `destroy()` method in your component code.
  - `InitializingBean`—Use if you require custom initialization after Oracle CEP has set all the properties of the component. Implement the `afterPropertiesSet()` method.

- `ActivatableBean`—Use if you want to run some code after all dynamic configuration has been set and the event processing network has been activated. Implement the `afterConfigurationActive()` method.
- `SuspendableBean`—Use if you want to suspend resources or stop processing events when the event processing network is suspended. Implement the `suspend()` method.
- `ResumableBean`—Use if you want to perform some task, such as acquire or configure resources, before the component resumes work.
- `RunnableBean`—Use if you want the component to be run in a thread.

The Spring framework implements similar bean life cycle interfaces; however, the equivalent Spring interfaces do not allow you to manipulate beans that were created by factories, while the Oracle CEP interfaces do.

- `Adapter`, `AdapterFactory`—Adapters and adapter factories must implement these interfaces respectively.
- `EventBuilder`—Use to create events whose Java representation does not expose the necessary setter and getter methods for its properties. If your event type is represented with a `JavaBean` with all required getter and setter methods, then you do not need to create an `EventBuilder`.
- `EventBuilder.Factory`—Factory for creating `EventBuilders`.

## Oracle CEP Development Environment for Eclipse

Oracle provides an IDE targeted specifically to programmers that want to develop Oracle CEP applications. *Oracle CEP Development Environment for Eclipse* (Oracle CEP IDE for short) is a set of plugins for the Eclipse IDE designed to help develop, deploy, and debug applications for Oracle CEP.

The key features of the Oracle CEP IDE are as follows:

- Project creation wizards and templates to quickly get started building event driven applications.
- Advanced editors for source files including Java and XML files common to Oracle CEP applications.
- Integrated server management to seamlessly start, stop, and deploy to Oracle CEP instances all within the IDE.
- Integrated debugging.

- Event Processing Network (EPN) visual design views for orienting and navigating in event processing applications.

Although it is not required or assumed that you are using the Oracle CEP IDE, Oracle recommends that you give it a try.

See the online help, integrated in the Oracle CEP IDE product itself, for detailed instructions on using the IDE.

## Downloading the Latest Oracle CEP IDE

New versions of the IDE are constantly being uploaded to the Oracle Technology Network Web site. Oracle highly recommends that you download the IDE from this site:

<http://www.oracle.com/technology/products/event-driven-architecture/cep-ide/10>

## Installing the Oracle CEP IDE Distributed With the Product

A version of the Oracle CEP IDE is shipped with the Oracle CEP product, although this version might be older than the one on the Oracle Technology Network site. However, if you want to use the version shipped with the product, follow these steps:

1. Obtain the required versions of Eclipse (3.3.2) and WTP (2.0). We recommend you take the entire Europa installation available at the following Web sites:

### Windows:

<http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/europa/winter/eclipse-jee-europa-winter-win32.zip>

### Linux:

<http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/europa/winter/eclipse-jee-europa-winter-linux-gtk.tar.gz>

2. Obtain the zip file containing the Oracle CEP IDE tools.

This file is called `cep-tools-10.3.0.20080917-2044.zip` and is located in the `ORACLE_CEP_HOME/ocp_10.3/eclipse-update-site` directory, where `ORACLE_CEP_HOME` refers to the main directory into which you installed Oracle CEP, such as `/oracle_cep`.

3. Open your Eclipse IDE and select the menu item `Help > Software Updates > Find and Install`.

4. Select the `Search for New Features` option.
5. Select `New Archived Site` and choose the zip file.
6. Complete the Update Manager, selecting to install the Oracle CEP tools.
7. When prompted restart, Eclipse. If you skip this, unreliable behavior can occur.

Once Eclipse has restarted, you should find all of the Oracle CEP IDE tools available to you.

## Creating Oracle CEP Applications: Typical Steps

The following procedure shows the *suggested* start-to-finish steps to create an Oracle CEP application. Although it is not required to program and configure the various components in the order shown, the procedure shows a typical and logical flow recommended by Oracle.

It is assumed in the procedure that you are using an IDE, although it is not required and the one you use is your choice. For one targeted to Oracle CEP developers, see [“Oracle CEP Development Environment for Eclipse” on page 2-8](#).

1. Set up your environment as described in [Setting Up Your Development Environment](#).
2. Design your event processing network (EPN).

This step involves creating the EPN assembly file, adding the full list of components that make up the application and how they are connected to each other, as well as registering the event types used in your application.

This step combines both designing of your application, in particular determining the components that you need to configure and code, as well as creating the actual XML file that specifies all the components. You will likely be constantly updating this XML file as you implement your application, but Oracle recommends you start with this step so you have a high-level view of your application.

For details, see [“Creating the EPN Assembly File” on page 2-11](#).

3. Design the EPL rules that the processors are going to use to select events from the stream.  
See the [EPL Reference Guide](#).
4. Determine the event types that your application is going to use, and, if creating your own JavaBean, program the Java file.

See [“Creating the Event Types” on page 2-15](#).



5. Program, and optionally configure, the adapters or event beans that act as inbound, intermediate, or outbound components of your event processing network. You can create your own adapters or event beans, or use the adapters provided by Oracle CEP. For details, see:
  - [“Creating Custom Adapters and Event Beans” on page 3-1](#)
  - [“Using the Java Message Service \(JMS\) Adapters” on page 4-1](#)
  - [“Using and Creating HTTP Publish-Subscribe Adapters” on page 5-1](#)
6. Configure the processors by creating their configuration XML files; the most important part of this step is designing and declaring the initial EPL rules that are associated with each processor.  
See [“Configuring the Complex Event Processor” on page 7-1](#).
7. Optionally configure the streams that stream data between adapters, processors, and the business logic POJO by creating their configuration XML files.  
See [“Configuring the Stream Component” on page 6-1](#).
8. Optionally configure the caching system to publish or consume events to and from a cache to increase the availability of the events and increase the performance of your applications.  
See [“Using Oracle CEP Caching” on page 8-1](#).

Oracle CEP provides a *load generator* testing tool that you can use to test your application, in particular the EPL rules. This testing tool can temporarily replace the adapter component in your application, for testing purposes only of course. For details, see [“Using the Load Generator to Test Your Application” on page 11-1](#).

See [“Next Steps” on page 2-19](#) for the list of steps you should follow after you have completed programming your application, such as packaging and deploying.

## Creating the EPN Assembly File

You use the EPN assembly file to declare the components that make up your Oracle CEP application and how they are connected to each other. You also use the file to register event types of your application, as well as the Java classes that implement the adapter and POJO components of your application.

For an example of an EPN assembly file, see the [foreign exchange \(FX\) example](#). For additional information about Spring and OSGi, see [“Additional Information about Spring and OSGi” on page A-1](#).

As is often true with Spring, there are different ways to use the tags to define your event network. This section shows one way. See [Oracle CEP Spring Tag Reference](#) or the [XSD Schema](#) for the full reference information on the other tags and attributes you can use.

For a typical way to create the EPN assembly file for your application, follow these steps:

1. Using your favorite XML or plain text editor, create an XML file with the `<beans>` root element and namespace declarations as follows:

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

...

</beans>
```

If you are not going to use any of the Spring-OSGI tags in the XML file, then their corresponding namespace declarations, shown in bold in the preceding example, are not required.

2. If you have programmed an adapter factory, add an `<osgi:service ...>` Spring tag to register the factory as an OSGi service. For example:

```
<osgi:service interface="com.bea.wlevs.ede.api.AdapterFactory">
  <osgi:service-properties>
    <prop key="type">hellomsgs</prop>
  </osgi:service-properties>
  <bean
class="com.bea.wlevs.adapter.example.helloworld.HelloWorldAdapterFactory" />
</osgi:service>
```

Specify the Oracle CEP-provided adapter factory

(`com.bea.wlevs.ede.api.AdapterFactory`) for the interface attribute. Use the `<osgi-service-properties>` tag to give the OSGi service a type name, in the example above the name is `hellomsgs`; you will reference this label later when you declare the adapter components of your application. Finally, use the `<bean>` Spring tag to register the

your adapter factory bean in the Spring application context; this class generates instances of the adapter.

**WARNING:** Be sure the type name (`hellomsgs` in the preceding example) is unique across *all* applications deployed to a particular Oracle CEP. The OSGI service registry is per server, not per application, so if two different adapter factory services have been registered with the same type name, it is undefined which adapter factory a particular application will use. To avoid this confusion, be sure that the value of the `<prop key="type">` entry for each OSGI-registered adapter factory in each EPN assembly file for a server is unique.

3. Add a `<wlevs:event-type-repository>` tag to register the event types that you use throughout your application, such as in the adapter implementations, business logic POJO, and the EPL rules associated with the processor components. For each event type in your application, add a `<wlevs:event-type>` child tag.

Event types are simple JavaBeans that you either code yourself (recommended) or let Oracle CEP automatically generate from the meta data you provide in the `<wlevs:event-type>` tag. If you code the JavaBean yourself, use a `<wlevs:class>` tag to specify your JavaBean class. You can optionally use the `<wlevs:property name="builderFactory">` tag to specify the Spring bean that acts as a builder factory for the event type, if you have programmed a factory. If you want Oracle CEP to automatically generate the JavaBean class, use the `<wlevs:metadata>` tag to list each property of the event type. The following example is taken from the FX sample:

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

See [wlevs:event-type-repository](#) for reference information about this tag. See [“Creating the Event Types” on page 2-15](#) for additional information about creating event types.

4. For each adapter component in your application, add a `<wlevs:adapter>` tag to declare that the component is part of the event processing network. Use the required `id` attribute to give it a unique ID and the `provider` attribute to specify the type of data feed to which the adapter will be listening. Use the `<wlevs:instance-property>` child tag to pass the adapter the

properties it expects. For example, the `csvgen` adapter, provided by Oracle CEP to test your EPL rules with a simulated data feed, defines a `setPort()` method and thus expects a `port` property, among other properties. Use the `provider` attribute to specify the adapter factory, typically registered as an OSGi service; you can also use the `csvgen` keyword to specify the `csvgen` adapter.

The following example declares the `helloWorldAdapter` of the `HelloWorld` example:

```
<wlevs:adapter id="helloworldAdapter" provider="hellomsgs"
manageable="true">
  <wlevs:instance-property name="message" value="HelloWorld - the
currenttime is:"/>
</wlevs:adapter>
```

In the example, the property `message` is passed to the adapter. The adapter factory provider is `hellomsgs`, which refers to the type name of the adapter factory OSGi service. The `manageable` attribute, common to all components, enables monitoring for the adapter; by default, manageability of the component is disabled due to possible performance impacts.

See [wlevs:adapter](#) for reference information about this tag, in particular additional optional attributes and child tags.

5. For each processor component in your application, add a `<wlevs:processor>` tag. Use the `id` attribute to give it a unique ID. Use either the `listeners` attribute or `<wlevs:listener>` child tag to specify the components that listen to the processor. The following two examples are equivalent:

```
<wlevs:processor id="preprocessorAmer" listeners="spreaderIn"/>
<wlevs:processor id="preprocessorAmer">
  <wlevs:listener ref="spreaderIn"/>
</wlevs:processor>
```

In the examples, the `spreaderIn` stream component listens to the `preprocessorAmer` processor.

See [wlevs:processor](#) for reference information about this tag, in particular additional optional attributes, such as `manageable` for enabling monitoring of the component.

6. For each stream component in your application, add a `<wlevs:stream>` tag to declare that the component is part of the event processing network. Use the `id` attribute to give it a unique ID. Use the `<wlevs:listener>` and `<wlevs:source>` child tags to specify the components that act as listeners and sources for the stream. For example:

```
<wlevs:stream id="fxMarketAmerOut">
  <wlevs:listener ref="preprocessorAmer"/>
```

```
<wlevs:source ref="fxMarketAmer"/>
</wlevs:stream>
```

In the example, the `fxMarketAmerOut` stream listens to the `fxMarketAmer` component, and the `preprocessorAmer` component in turn listens to the `fxMarketAmerOut` stream.

Nest the declaration of the business logic POJO, called `outputBean` in the example, using a standard Spring `<bean>` tag inside a `<wlevs:listener>` tag, as shown:

```
<wlevs:stream id="spreaderOut" advertise="true">
  <wlevs:listener>
    <!-- Create business object -->
    <bean id="outputBean"
          class="com.bea.wlevs.example.fx.OutputBean"
          autowire="byName"/>
  </wlevs:listener>
</wlevs:stream>
```

The `advertise` attribute is common to all Oracle CEP application assembly tags and is used to register the component as a service in the OSGI registry.

See [wlevs:stream](#) for reference information about this tag, in particular additional optional attributes, such as `manageable` for enabling monitoring of the component.

## Creating the Event Types

Event types define the properties of the events that are handled by Oracle CEP applications. Adapters receiving incoming events from different event sources, such as JMS, or financial market data feeds. You must define these events by an event type before a processor is able to handle them. An event type can be created either programmatically using the `EventTypeRepository` class or declaratively in the EPN assembly file.

You then use these event types in the adapter and POJO Java code, as well as in the EPL rules associated with the processors.

Events are JavaBean instances in which each property represents a data item from the feed. Oracle recommends that you create your own JavaBean class that represents the event type and register the class in the EPN assembly file. By creating your own JavaBean, you can reuse it and you have complete control over what the event looks like. Alternatively, you can specify the properties of the event type in the EPN assembly file using `<wlevs:metadata>` tags, in which case Oracle CEP will use a map; this method is best used for quick prototyping.

The following simple example shows the JavaBean that implements the `HelloWorldEvent`:

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

```
public class HelloWorldEvent {  
    private String message;  
  
    public String getMessage() {  
        return message;  
    }  
  
    public void setMessage (String message) {  
        this.message = message;  
    }  
}
```

The preceding Java class follows standard JavaBeans programming guidelines. See the [JavaBeans Tutorial](#) for additional details.

In addition, Oracle recommends that, if possible, you make your JavaBeans immutable for performance reasons because immutable beans help the garbage collection work much better. Immutable beans are read only (only getters) and have public constructors with arguments that satisfy immutability.

Once you have programmed and compiled the JavaBean that represents your event type, you register it in the EPN assembly file using the `<wlevs:event-type>` child tag of `<wlevs:event-type-repository>`. Use the `<wlevs:class>` tag to point to your JavaBean class, and then optionally use the `<wlevs:property name="builderFactory">` tag to specify a custom event builder factory for the event type, which can be used to create event instances that do not follow the JavaBean specification. If want Oracle CEP to generate the bean instance for you, use the `<wlevs:metadata>` tag to group standard Spring `<entry>` tags for each property. The following example shows both ways:

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

```

        <entry key="name" value="java.lang.String"/>
        <entry key="age" value="java.lang.Integer"/>
        <entry key="address" value="java.lang.String"/>
    </wlevs:metadata>
</wlevs:event-type>
</wlevs:event-type-repository>

```

In the example, `ForeignExchangeEvent` is implemented by the `ForeignExchangeEvent` inner class of `com.bea.wlevs.example.fx.OutputBean`. Instances of `AnotherEvent` will be generated by Oracle CEP. The `AnotherEvent` has three properties: `name`, `age`, and `address`.

See [“Using an Event Type Builder Factory” on page 2-17](#) for additional information about using an event type builder factory.

You can now reference the event types as standard JavaBeans in the Java code of the adapters and business logic POJO in your application. The following snippet from the business logic POJO `HelloWorldBean.java` of the `HelloWorld` application shows an example:

```

public void onEvent(List newEvents)
    throws RejectEventException {
    for (Object event : newEvents) {
        HelloWorldEvent helloWorldEvent = (HelloWorldEvent) event;
        System.out.println("Message: " + helloWorldEvent.getMessage());
    }
}

```

The following EPL rule shows how you can reference the `HelloWorldEvent` in a `SELECT` statement:

```
SELECT * FROM HelloWorldEvent RETAIN 1 event
```

## Using an Event Type Builder Factory

When you register an event type in the repository using the `<wlevs:event-type>` tag, Oracle CEP creates instances of the event using the information in the event type class.

Sometimes, however, you might want or need to have more control over how the event type instances are created. This is required if the POJO that represents the event does not expose the necessary getter and setter methods for the event properties. For example, assume the event type has a `firstname` property, but the EPL rule that executes on the event type assumes the property is called `fname`. Further assume that you cannot change either the event type class (because you are using a shared event class from another bundle, for example) or the EPL rule to make them

compatible with each other. In this case you can use an event type builder factory to change the way the event type instance is created so that the property is named `fname` rather than `firstname`.

When you program the event type builder factory, you must implement the `EventBuilder.Factory` inner interface of the [com.bea.wlevs.ede.api.EventBuilder](#) interface; see the [Javadocs](#) for details about the methods you must implement, such as `createBuilder()` and `createEvent()`.

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

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

import java.util.HashMap;
import java.util.Map;

import com.bea.wlevs.ede.api.EventBuilder;
import com.bea.wlevs.example.fx.OutputBean.ForeignExchangeEvent;

public class ForeignExchangeBuilderFactory implements EventBuilder.Factory {

    public EventBuilder createBuilder() {
        return new ForeignExchangeBuilder();
    }

    static class ForeignExchangeBuilder implements EventBuilder {
        private Map<String, Object> values = new HashMap<String, Object>(10);

        public Object createEvent() {
            return new ForeignExchangeEvent(
                (String) values.get("symbol"),
                (Double) values.get("price"),
                (String) values.get("fromRate"),
                (String) values.get("toRate"));
        }

        public void put(String property, Object value) throws
        IllegalStateException {
            values.put(property, value);
        }
    }
}
```

When you register the event type in the EPN assembly file, use the `<wlevs:property name="builderFactory">` child tag of the `<wlevs:event-type>` tag to specify the name of the factory class. The hard-coded `builderFactory` value of the `name` attribute alerts Oracle CEP that it should use the specified factory class, rather than its own default factory, when creating



instances of this event. For example, in the FX example, the builder factory is registered as shown in bold:

```
<wlevs:event-type-repository>
  <wlevs:event-type type-name="ForeignExchangeEvent">

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

## Sharing Event Types Between Application Bundles

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

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

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

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

## Next Steps

After you have programmed all components of your application and created their configuration XML files:

- Assemble all the components into a deployable OSGi bundle. This step also includes creating the `MANIFEST.MF` file that describes the bundle.

See [“Assembling an Oracle CEP Application: Main Steps” on page 10-2](#).

- Optionally configure the server in your domain to enable logging, debugging, and other services.

See [Configuring Oracle CEP](#).

- Deploy the application to Oracle CEP.

See “[Deploying Oracle CEP Applications: Main Steps](#)” on page 10-7.

- Start Oracle CEP.

See [Stopping and Starting the Server](#).

- Optionally start test clients, such as the load generator.

See “[Using the Load Generator to Test Your Application](#)” on page 11-1.

# Creating Custom Adapters and Event Beans

This section contains information on the following subjects:

- [“Overview of Adapters” on page 3-2](#)
- [“Overview of Event Beans” on page 3-2](#)
- [“Overview of Event Sources and Event Sinks” on page 3-3](#)
- [“Overview of Adapter and Event Bean Factories” on page 3-4](#)
- [“Creating Adapters or Event Beans: Typical Steps” on page 3-5](#)
- [“Programming Adapters and Event Bean Classes as Event Sources: Guidelines” on page 3-5](#)
- [“Programming Event Sinks: Guidelines” on page 3-8](#)
- [“Programming the Optional Factory Class” on page 3-10](#)
- [“Accessing a Relational Database” on page 3-11](#)
- [“Updating the EPN Assembly File” on page 3-11](#)
- [“Configuring the Adapter or Event Bean” on page 3-13](#)
- [“Creating an Adapter or Event Bean in Its Own Bundle” on page 3-15](#)
- [“Extending the Configuration of an Adapter or Event Bean” on page 3-17](#)
- [“Passing Login Credentials from an Adapter to the Data Feed Provider” on page 3-23](#)

## Overview of Adapters

One of the main roles of an adapter is to convert data coming from some stream, such as a market data feed, into Oracle Complex Event Processing (or *Oracle CEP* for short) events. These events are then passed to other components in the application, such as processors. An adapter is usually the entry point to an Oracle CEP application. An adapter can also be the exit point of an application so that it receives events from an intermediate component, converts the data into something that an external application can read, and then sends it out.

The [FX example description](#) shows three adapters that read in data from currency data feeds and then pass the data, in the form of a specific event type, to the processors, which are the next components in the network.

You can create adapters of different types, depending on the format of incoming data and the technology you use in the adapter code to do the conversion. The most typical types of adapters are those that:

- Use a data vendor API, such as Reuters, Wombat, or Bloomberg.
- Convert incoming JMS messages using standard JMS APIs.
- Use other messaging systems, such as TIBCO Rendezvous.
- Use a socket connection to the customer's own data protocol.

Adapters are Java classes that implement specific Oracle CEP interfaces. You register the adapter classes in the EPN assembly file that describes your entire application.

You can optionally change the default configuration of the adapter, or even extend the configuration and add new configuration elements and attributes. There are two ways to pass configuration data to the adapter; the method you chose depends on whether you want to dynamically change the configuration after deployment. If you are *not* going to change the configuration data after the adapter is deployed, then you can configure the adapter in the EPN assembly file. If, however, you do want to be able to dynamically change the configuration elements, then you should put this configuration in the adapter-specific configuration files. Both methods are discussed below.

## Overview of Event Beans

Event beans are very similar to standard Spring beans except that they can be managed by the Oracle CEP management framework. Standard Spring beans are managed by the Spring

framework. You register event beans in the EPN assembly file using the Oracle CEP `<wlevs:event-bean>` tag rather than the standard `<bean>` tag.

An event bean is a type of Stage, it can be monitored by the Event Server monitoring framework, make use of the configuration metadata annotations, and it can be set to record, and play-back events that pass through it. An event bean can also participate in the Event Server bean lifecycle by specifying methods in its XML declaration, rather than by implementing Event Server API interfaces.

A developer can choose to use either a Spring-bean or an event-bean. The former is used for legacy integration to Spring. The latter is used to actively use the capabilities of the Event Server container.

## Overview of Event Sources and Event Sinks

Standard Spring beans and event beans can be event sources, event sinks, or both. Event sources generate events, event sinks receive events.

### Standard Spring Beans as Event Sources

You specify that a standard Spring bean component in your EPN is an event source by implementing the `com.bea.wlevs.ede.api.EventSource` API. The bean can also optionally implement the various lifecycle interfaces, such as `InitializingBean`, `DisposableBean`, and the active interfaces, such as `RunnableBean`. If a Spring-bean implements `Runnable` but not `RunnableBean`, Oracle CEP does not run it in a thread. This is different behavior from an event bean.

The Spring bean event source can make use of the configuration metadata annotations, such as `@Prepare`, `@Rollback`, and `@Activate`.

You register the Spring bean in the EPN assembly file in the standard way using `<bean>`. You can then specify this bean as an event source of some other stage in the EPN.

### Event Beans as Event Sources

The implementation class of an event bean that is an event source may be specified as private to the application or as an advertised OSGI service re-usable by other applications.

For the framework to be able to fully manage the bean as an EPN component, it must be specified as an event-bean rather than a standard Spring bean. Management tasks include monitoring and record/playback.

You register event beans in the EPN assembly file using the `<wlevs:event-bean>` tag. For example:

```
<wlevs:event-bean id="recplayEventSink"
                  class="com.bea.wlevs.example.recplayRecplayEventSink">
    <wlevs:listener ref="playbackHttpPublisher" />
</wlevs:event-bean>
```

## Event Sinks

The functionality of Spring beans and event beans as event sinks is very similar to that of event sources except for the following:

- Spring bean and event bean event sinks must implement the `EventSink` interface.
- Event bean event sinks must provide an `onevent-method`.

Event sinks are not active which means that if they implement the `Runnable` interface, Oracle CEP does not run them in a separate thread.

You reference event sinks in the EPN assembly file using the `<wlevs:listener>` tag:

```
<wlevs:stream id="myStream" >
    <wlevs:listener ref="myEventSink" />
</wlevs:stream>
```

## Overview of Adapter and Event Bean Factories

If your adapter or event bean is going to be used only by a single Oracle CEP application, then you do not need to create a factory. However, if multiple applications are going to use the same adapter or event bean, then you should also program a factory. In this case, every application gets its own instance of the adapter.

Adapter or event bean factories must implement the `com.bea.wlevs.ede.api.Factory` interface. This interface has a single method, `create()`, that you implement to create an adapter or event bean instance.

You register factories in the EPN assembly file using the `<wlevs:factory>` tag:

```
<wlevs:factory provider-name="myprovider" class="my.Implementation"/>
```

Note that if you need to specify service properties, then you must use the `<osgi:service>` tag to register the factory.

## Creating Adapters or Event Beans: Typical Steps

The following procedure describes the typical steps for creating an adapter:

1. Program the adapter or event bean Java class.

If your adapter or event bean is an event source, see [“Programming Adapters and Event Bean Classes as Event Sources: Guidelines” on page 3-5](#). If it is an event sink, see [“Programming Event Sinks: Guidelines” on page 3-8](#).

2. Optionally program the factory class. You only need to do this if many applications are going to use the adapter or event bean.

See [“Programming the Optional Factory Class” on page 3-10](#).

3. Update the EPN assembly file with adapter, event bean, and adapter factory registration info.

See [“Updating the EPN Assembly File” on page 3-11](#)

4. Optionally change the default configuration of the adapter.

See [“Configuring the Adapter or Event Bean” on page 3-13](#).

5. Optionally extend the configuration of the adapter if its basic one is not adequate.

See [“Extending the Configuration of an Adapter or Event Bean” on page 3-17](#).

In the preceding procedure, it is assumed that the adapter or event bean is bundled in the same application JAR file that contains the other components of the event network, such as the processor, streams, and business logic POJO. If you want to bundle the adapter or event bean in its own JAR file so that it can be shared among many applications, see [“Creating an Adapter or Event Bean in Its Own Bundle” on page 3-15](#).

## Programming Adapters and Event Bean Classes as Event Sources: Guidelines

This section describes how to create an inbound adapter that acts as an event source because it receives incoming data and generates events that it sends to the next component in the EPN. Because event beans are functionally the same, the guidelines also apply to programming event beans that act as event sources.

The inbound adapter class typically reads the stream of incoming data, such as from a market data feed, converts it into an Oracle CEP event type that is understood by the rest of the application, and sends the event to the next component in the network.

The following example shows the adapter class of the HelloWorld sample; see the explanation after the example for coding guidelines that correspond to the Java code in bold.

```
package com.bea.wlevs.adapter.example.helloworld;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import com.bea.wlevs.ede.api.EventSender;
import com.bea.wlevs.ede.api.EventSource;
import com.bea.wlevs.ede.api RunnableBean;

import com.bea.wlevs.event.example.helloworld.HelloWorldEvent;

public class HelloWorldAdapter implements RunnableBean, EventSource {

    private static final int SLEEP_MILLIS = 300;

    private DateFormat dateFormat;

    private String message;
    private EventSender eventSender;
    private boolean stopped;

    public HelloWorldAdapter() {
        super();
        dateFormat = DateFormat.getTimeInstance();
    }

    public void run() {
        stopped = false;
        while (!isStopped()) { // Generate messages forever...
            generateHelloMessage();
            try {
                synchronized (this) {
                    wait(SLEEP_MILLIS);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public void setMessage(String message) {
        this.message = message;
    }

    @SuppressWarnings({ "unchecked" })
    private void generateHelloMessage() {
        List eventCollection = new ArrayList();
```



```

        String message = this.message + dateFormat.format(new Date());
        HelloWorldEvent event = new HelloWorldEvent();
        event.setMessage(message);
        eventCollection.add(event);
        eventSender.sendEvent(eventCollection, null);
    }

    public void setEventSender(EventSender sender) {
        eventSender = sender;
    }

    public synchronized void suspend() {
        stopped = true;
    }

    private synchronized boolean isStopped() {
        return stopped;
    }
}

```

Follow these guidelines when programming the adapter Java class; code snippets of the guidelines are shown in bold in the preceding example:

- Import the interfaces and classes of the Oracle CEP API:

```

import com.bea.wlevs.ede.api.EventSender;
import com.bea.wlevs.ede.api.EventSource;
import com.bea.wlevs.ede.api.RunnableBean;

```

Because the adapter is an event source it must implement the `EventSource` interface. If you want the adapter to run in a thread, also implement `RunnableBean`. The `EventSender` interface sends event types to the next component in your application network. For full details of these APIs, see the [Javadoc](#).

- Import the application-specific classes that represent the event types used in the application:

```

import com.bea.wlevs.event.example.helloworld.HelloWorldEvent
import

```

The `com.bea.wlevs.event.example.helloworld.HelloWorldEvent` class is a `JavaBean` that represents the event type used in the application.

- The adapter class must implement the `EventSource` and `RunnableBean` interfaces because it is an event source and will run in its own thread:

```

public class HelloWorldAdapter implements RunnableBean, EventSource {

```

The `EventSource` interface provides the `EventSender` that you use to send events.

- Because the adapter implements the `RunnableBean` interface, your adapter must then implement the `run()` method:

```
public void run() {...
```

This is where you should put the code that reads the incoming data, such as from a market feed, and convert it into an Oracle CEP event type, and then send the event to the next component in the network. Refer to the documentation of your data feed provider for details on how to read the incoming data. See [“Accessing Third-Party JAR Files From Your Application” on page 10-6](#) for information about ensuring you can access the vendor APIs if they are packaged in a third-party JAR file.

In the `HelloWorld` example, the adapter itself generates the incoming data using the `generateHelloMessage()` private method. This is just for illustrative purposes and is not a real-world scenario. The `generateHelloMessage()` method also includes the other typical event type programming tasks:

```
HelloWorldEvent event = new HelloWorldEvent();
event.setMessage(message);
eventCollection.add(event);
eventSender.sendEvent(eventCollection, null);
```

The `HelloWorldEvent` is the event type used by the `HelloWorld` example; the event type is implemented with a `JavaBean` and is registered in the EPN assembly file using the `<wlevs:event-type-repository>` tag. See [“Creating the Event Types” on page 2-15](#) for details. The `setMessage()` method sets the properties of the event; in typical adapter implementations this is how you convert a particular property of the incoming data into an event type property. Finally, the `EventSender.sendEvent()` method sends this new event to the next component in the network.

- Because your adapter implements `EventSource`, you must implement the `setEventSender()` method, which passes in the `EventSender` that you use to send events:

```
public void setEventSender(EventSender sender) { ...
```

- If, as is typically the case, your adapter implements `SuspendableBean`, you must implement the `suspend()` method that stops the adapter when, for example, the application is undeployed:

```
public synchronized void suspend() throws Exception { ...
```

## Programming Event Sinks: Guidelines

The following sample code shows a Spring bean from `HelloWorld` application that acts as an event sink; see the explanation after the example for the code shown in bold:

```

package com.bea.wlevs.example.helloworld;

import java.util.List;

import com.bea.wlevs.ede.api.EventRejectedException;
import com.bea.wlevs.ede.api.EventSink;
import com.bea.wlevs.event.example.helloworld.HelloWorldEvent;

public class HelloWorldBean implements EventSink {

    @SuppressWarnings({ "unchecked" })
    public void onEvent(List newEvents)
        throws EventRejectedException {
        for (Object event : newEvents) {
            if (event instanceof HelloWorldEvent) {
                HelloWorldEvent helloWorldEvent = (HelloWorldEvent) event;
                System.out.println("Message: " +
helloWorldEvent.getMessage());
            }
        }
    }
}

```

The programming guidelines shown in the preceding example are as follows:

- Your bean must import the event type of the application, which in the HelloWorld case is HelloWorldEvent:

```
import com.bea.wlevs.event.example.helloworld.HelloWorldEvent;
```

- Your bean must implement the com.bea.wlevs.ede.api.EventSink interface:

```
public class HelloWorldBean implements EventSink {...
```

- The EventSink interface has a single method that you must implement, onEvent(java.util.List), which is a callback method for receiving events. The parameter of the method is a List that contains the actual events that the bean received from the component that sent it the events:

```
public void onEvent(List newEvents)
```

- The data type of the events is determined by the event type you registered in the EPN assembly file of the application. In the example, the event type is HelloWorldEvent; the code first ensures that the received event is truly a HelloWorldEvent:

```
if (event instanceof HelloWorldEvent) {  
    HelloWorldEvent helloWorldEvent = (HelloWorldEvent) event;  
}
```

This event type is a JavaBean that was configured in the EPN assembly file as shown:

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

See [“Creating the EPN Assembly File” on page 2-11](#) for procedural information about creating the EPN assembly file, and [Oracle CEP Spring Tag Reference](#) for reference information.

- Events are instances of the appropriate JavaBean, so you access the individual properties using the standard `getXXX()` methods. In the example, the `HelloWorldEvent` has a property called `message`:

```
System.out.println("Message: " + helloWorldEvent.getMessage());
```

For complete API reference information about the Oracle CEP APIs described in this section, see the [Javadocs](#).

## Programming the Optional Factory Class

Your adapter factory class must implement the `com.bea.wlevs.ede.api.AdapterFactory` interface, which has a single method, `create()`, in which you code the creation of your specific adapter class. Event beans implement `Factory`.

The following is a possible adapter factory class for the `HelloWorld` example:

```
package com.bea.adapter.wlevs.example.helloworld;  
  
import com.bea.wlevs.ede.api.Adapter;  
import com.bea.wlevs.ede.api.AdapterFactory;  
  
public class HelloWorldAdapterFactory implements Factory {  
    public HelloWorldAdapterFactory() {  
    }  
  
    public synchronized Adapter create() throws IllegalArgumentException {  
        return new HelloWorldAdapter();  
    }  
}
```

```
    }
}
```

For full details of these APIs, see the [Javadoc](#)

## Accessing a Relational Database

You can use the [Java Database Connectivity \(JDBC\)](#) APIs in your adapters, event beans, and standard Spring beans to access data contained in a relational database. Oracle CEP supports [JDBC 3.0](#).

Follow these steps to use JDBC in your components:

1. Configure JDBC for Oracle CEP.

For details, see [Configuring Access to a Relational Database](#).

2. In your bean Java code, you can start using the JDBC APIs as usual, by using a `DataSource` or instantiating a `DriverManager`. For example:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL("jdbc:oracle:thin:user/passwd@localhost:1521/XE");
Connection conn =
ods.getConnection();
```

See [Getting Started with the JDBC API](#) for additional programming information.

## Updating the EPN Assembly File

The adapters, event beans, and adapter factory must be registered in the EPN assembly file, as discussed in the following sections.

For a complete description of the configuration file, including registration of other components of your application, see [“Creating the EPN Assembly File” on page 2-11](#).

## Registering the Adapter or Event Bean Factory

You register factories in the EPN assembly file using the `<wlevs:factory>` tag:

```
<wlevs:factory provider-name="myprovider" class="my.Implementation"/>
```

If you need to specify service properties, then you must use the `<osgi:service>` tag to register the factory as an OSGI service in the EPN assembly file. The scope of the OSGI service registry is the entire Oracle CEP. This means that if more than one application deployed to a given server

is going to use the same adapter factory, be sure to register the adapter factory only *once* as an OSGI service.

Add an entry to register the service as an implementation of the `com.bea.wlevs.ede.api.AdapterFactory` interface. Provide a property, with the key attribute equal to `type`, and the name by which this adapter provider will be referenced. Finally, add a nested standard Spring `<bean>` tag to register the your specific adapter class in the Spring application context

For example, the following segment of the EPN assembly file registers the `HelloWorldAdapterFactory` as the provider for type `hellomsgs`:

```
<osgi:service interface="com.bea.wlevs.ede.api.AdapterFactory">
  <osgi:service-properties>
    <prop key="type">hellomsgs</prop>
  </osgi:service-properties>
  <bean
class="com.bea.adapter.wlevs.example.helloworld.HelloWorldAdapterFactory"
/>
</osgi:service>
```

## Declaring the Adapter and Event Bean Components in your Application

In the EPN assembly file, you use the `wlevs:adapter` tag to declare an adapter as a component in the event processor network. Similarly, you use the `<wlevs:event-bean>` tag for event beans. For example:

```
<wlevs:event-bean id="recplayEventSink"
                  class="com.bea.wlevs.example.recplayRecplayEventSink">
  <wlevs:listener ref="playbackHttpPublisher"/>
</wlevs:event-bean>
```

If you registered an optional factory as an OSGI service, then use the `provider` attribute to point to the name you specified as the `type` in your `osgi:service` entry; for example:

```
<wlevs:adapter id="helloworldAdapter" provider="hellomsgs"/>
```

This means that an adapter will be instantiated by the factory registered for the type `hellomsgs`.

You can also use a `<wlevs:instance-property>` child tag of `<wlevs:adapter>` to set any *static* properties in the adapter bean. Static properties are those that you will not dynamically change after the adapter is deployed.

For example, if your adapter class has a `setPort()` method, you can pass it the port number as shown:

```
<wlevs:adapter id="myAdapter" provider="myProvider">
  <wlevs:instance-property name="port" value="9001" />
</wlevs:adapter>
```

## Configuring the Adapter or Event Bean

This section applies to both adapters and event beans. For simplicity the text mentions only adapters. The configuration of an event bean would be enclosed in the `<event-bean>` element.

Each adapter in your application has a default configuration. In particular:

- Monitoring is enabled.

The default adapter configuration is typically adequate for most applications. However, if you want to change this configuration, you must create an XML file that is deployed as part of the Oracle CEP application bundle. You can later update this configuration at runtime using the [wlevs.Admin utility](#) or manipulating the appropriate JMX Mbeans directly.

If your application has more than one adapter, you can create separate XML files for each adapter, or create a single XML file that contains the configuration for all adapters, or even all components of your application (adapters, processors, and streams). Choose the method that best suits your development environment.

The following procedure describes the main steps to create the adapter configuration file. For simplicity, it is assumed in the procedure that you are going to configure all components of an application in a single XML file

See [XSD Schema Reference for Component Configuration Files](#) for the complete XSD Schema that describes the adapter configuration file.

1. Create an XML file using your favorite XML editor. You can name this XML file anything you want, provided it ends with the `.xml` extension.

The root element of the configuration file is `<config>`, with namespace definitions shown in the next step.

2. For each adapter in your application, add an `<adapter>` child element of `<config>`. Uniquely identify each adapter with the `<name>` child element. This name must be the same as the value of the `id` attribute in the `<wlevs:adapter>` tag of the EPN assembly file that defines the event processing network of your application. This is how Oracle CEP knows to which particular adapter component in the EPN assembly file this adapter configuration applies. See [“Creating the EPN Assembly File” on page 2-11](#) for details.

For example, if your application has two adapters, the configuration file might initially look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<helloworld:config
xmlns:helloworld="http://www.bea.com/xml/ns/wlevs/example/helloworld">
  <processor>
    ...
  </processor>

  <adapter>
    <name>firstAdapter</name>
    ...
  </adapter>

  <adapter>
    <name>secondAdapter</name>
    ...
  </adapter>
</helloworld:config>
```

In the example, the configuration file includes two adapters called `firstAdapter` and `secondAdapter`. This means that the EPN assembly file must include at least two adapter registrations with the same identifiers:

```
<wlevs:adapter id="firstAdapter" ...>
  ...
</wlevs:adapter>

<wlevs:adapter id="secondAdapter" ...>
  ...
</wlevs:adapter>
```

**WARNING:** Identifiers and names in XML files are case sensitive, so be sure you specify the same case when referencing the component’s identifier in the EPN assembly file.

3. Optionally use the `monitoring` Boolean attribute of the `<adapter>` element to enable or disable monitoring of the adapter; by default monitoring is enabled. When monitoring is enabled, the adapter gathers runtime statistics and forwards this information to an Mbean:



```
<adapter monitoring="true">
  <name>firstAdapter</name>
</adapter>
```

To truly enable monitoring, you must have also enabled the *manageability* of the component, otherwise setting the `monitoring` attribute to `true` has no effect. You enable manageability by setting the `manageable` attribute of the corresponding adapter component registration in the EPN assembly file to `true`, as shown in bold in the following example:

```
<wlevs:adapter id="firstAdapter" provider="hellomsgs"
manageable="true">
```

## Example of an Adapter Configuration File

The following sample XML file shows how to configure two adapters, `firstAdapter` and `secondAdapter`.

```
<?xml version="1.0" encoding="UTF-8"?>

<sample:config
  xmlns:sample="http://www.bea.com/xml/ns/wlevs/example/sample">

  <adapter>
    <name>firstAdapter</name>
  </adapter>

  <adapter monitoring="true">
    <name>secondAdapter</name>
  </adapter>

</sample:config>
```

## Creating an Adapter or Event Bean in Its Own Bundle

This section applies to both adapters and event beans. For simplicity the text mentions only adapters.

In the procedure described in [“Creating Adapters or Event Beans: Typical Steps” on page 3-5](#), it is assumed that the adapter and adapter factory are bundled in the same application JAR file that contains the other components of the event network, such as the processor, streams, and business logic POJO.

However, you might sometimes want to bundle the adapter in its own JAR file and then reference the adapter in other application bundles. This is useful if, for example, two different applications

read data coming from the same data feed provider and both applications use the same event types. In this case, it makes sense to share a single adapter and event type implementations rather than duplicate the implementation in two different applications.

There is no real difference in *how* you configure an adapter and an application that uses it in separate bundles; the difference lies in *where* you put the configuration, as described in the following guidelines:

- Create an OSGI bundle that contains only the adapter Java class, the adapter factory Java class, and optionally, the event type Java class into which the adapter converts incoming data. For simplicity, it is assumed that this bundle is called `GlobalAdapter`.
- In the EPN assembly file of the `GlobalAdapter` bundle:
  - Register the adapter factory as an OSGI service as usual, as described in [“Registering the Adapter or Event Bean Factory” on page 3-11](#).
  - If you are also including the event type in the bundle, register it as described in [“Creating the Event Types” on page 2-15](#).
  - Do *not* declare the adapter component using the `<wlevs:adapter>` tag. You will use this tag in the EPN assembly file of the application bundle that actually uses the adapter.
- If you want to further configure the adapter, follow the usual procedure as described in [“Configuring the Adapter or Event Bean” on page 3-13](#).
- If you are including the event type in the `GlobalAdapter` bundle, export the `JavaBean` class in the `MANIFEST.MF` file of the `GlobalAdapter` bundle. Use the `Export-Package` header, as described in [“Creating the MANIFEST.MF File” on page 10-4](#).
- Assemble and deploy the `GlobalAdapter` bundle in the usual way, as described in [“Assembling and Deploying Oracle Complex Event Processing Applications” on page 10-1](#).
- In the EPN assembly file of the application that is going to use the adapter, declare the adapter component in the usual way, as described in [“Declaring the Adapter and Event Bean Components in your Application” on page 3-12](#). You still use the `provider` attribute to specify the OSGI-registered adapter factory, although in this case the OSGI registration happens in a different EPN assembly file (of the `GlobalAdapter` bundle) from the EPN assembly file that actually uses the adapter.
- If you have exported the event type in the `GlobalAdapter` bundle, you must explicitly import it into the application that is going to use it. You do this by adding the package to

the `Import-Package` header of the `MANIFEST.MF` file of the application bundle, as described in [“Creating the MANIFEST.MF File” on page 10-4](#).

## Extending the Configuration of an Adapter or Event Bean

This section applies to both adapters and event beans. For simplicity the text mentions only adapters.

Adapters have default configuration data, as described in [“Configuring the Adapter or Event Bean” on page 3-13](#) and [XSD Schema Reference for Component Configuration Files](#). This default configuration is typically adequate for simple and basic applications.

However, you can also extend this configuration by using XSD Schema to specifying a *new* XML format of an adapter configuration file that extends the built-in XML type provided by Oracle CEP. By extending the XSD Schema, you can add as many new elements to the adapter configuration as you want, with few restrictions other than each new element must have a `name` attribute. This feature is based on standard technologies, such as XSD Schema and [Java Architecture for XML Binding \(JAXB\)](#).

The following procedure describes how to extend the adapter configuration:

1. Create the new XSD Schema file that describes the extended adapter configuration. This XSD file must also include the description of the other components in your application (processors and streams), although you typically use built-in XSD types, defined by Oracle CEP, to describe them.  
See [“Creating the XSD Schema File” on page 3-19](#) for details.
2. As part of your application build process, generate the Java representation of the XSD schema types using a JAXB binding compiler, such as the `com.sun.tools.xjc.XJCTask` Ant task from Sun’s GlassFish reference implementation. This Ant task is included in the Oracle CEP distribution for your convenience.

The following sample `build.xml` file shows how to do this:

```
<property name="base.dir" value="." />
<property name="output.dir" value="output" />
<property name="sharedlib.dir"
value="${base.dir}/../../../../../../modules" />
<property name="wlrllib.dir" value="${base.dir}/../../../../../../modules"/>

<path id="classpath">
  <pathelement location="${output.dir}" />
  <fileset dir="${sharedlib.dir}">
    <include name="*.jar" />
  </fileset>
</path>
```

```

        <fileset dir="${wlrtlib.dir}">
            <include name="*.jar"/>
        </fileset>
    </path>

    <taskdef name="xjc" classname="com.sun.tools.xjc.XJCTask">
        <classpath refid="classpath" />
    </taskdef>

    <target name="generate" depends="clean, init">
        <copy file="../../../xsd/wlevs_base_config.xsd"
            todir="src/main/resources/extension" />
        <copy file="../../../xsd/wlevs_application_config.xsd"
            todir="src/main/resources/extension" />
        <xjc extension="true" destdir="${generated.dir}">
            <schema dir="src/main/resources/extension"
                includes="helloworld.xsd"/>
            <produces dir="${generated.dir}" includes="**/*.java" />
        </xjc>
    </target>

```

In the example, the extended XSD file is called `helloworld.xsd`. The build process copies the Oracle CEP XSD files (`wlevs_base_config.xsd` and `wlevs_application_config.xsd`) to the same directory as the `helloworld.xsd` file because `helloworld.xsd` imports the Oracle CEP XSD files.

3. Compile these generated Java files into classes.
4. Package the compiled Java class files in your application bundle.  
See [“Assembling an Oracle CEP Application: Main Steps” on page 10-2](#) for details.
5. Program your adapter as described in [“Programming Adapters and Event Bean Classes as Event Sources: Guidelines” on page 3-5](#). Within your adapter code, you access the extended configuration as usual, as described in [“Programming Access to the Configuration of an Adapter or Event Bean” on page 3-22](#).
6. When you create the configuration XML file that describes the components of your application, be sure you use the extended XSD file as its description. In addition, be sure you identify the namespace for this schema rather than the default schema. For example:

```

<?xml version="1.0" encoding="UTF-8"?>
<helloworld:config
    xmlns:helloworld="http://www.bea.com/xml/ns/wlevs/example/helloworld">

```

```

<adapter>
  <name>helloworldAdapter</name>
  <message>HelloWorld - the current time is:</message>
</adapter>

</helloworld:config>

```

## Creating the XSD Schema File

The new XSD schema file extends the `wlevs_application_config.xsd` XSD schema and then adds new custom information, such as new configuration elements for an adapter. Use standard XSD schema syntax for your custom information.

Oracle recommends that you use the XSD schema in [“Complete Example of an Extended XSD Schema File” on page 3-21](#) as a basic template, and modify the content to suit your needs. In addition to adding new configuration elements, other modifications include changing the package name of the generated Java code and the element name for the custom adapter. You can control whether the schema allows just your custom adapter or other components like processors.

Follow these steps when creating the XSD Schema file that describes your extended adapter configuration; see [“Complete Example of an Extended XSD Schema File” on page 3-21](#) for an example:

1. Using your favorite XML Editor, create the basic XSD file with the required namespaces, in particular those for JAXB. For example:

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  targetNamespace="http://www.bea.com/xml/ns/wlevs/example/helloworld"
  xmlns="http://www.bea.com/xml/ns/wlevs/example/helloworld"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
  xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
  xmlns:wlevs="http://www.bea.com/xml/ns/wlevs/config/application"
  jxb:extensionBindingPrefixes="xjc" jxb:version="1.0"
  elementFormDefault="unqualified"
  attributeFormDefault="unqualified">
  ...
</xs:schema>

```

2. Import the `wlevs_application_config.xsd` XSD schema:

```

<xs:import
  namespace="http://www.bea.com/xml/ns/wlevs/config/application"
  schemaLocation="wlevs_application_config.xsd"/>

```

The `wlevs_application_config.xsd` in turn imports the `wlevs_base_config.xsd` XSD file.

3. Use the `<complexType>` XSD element to describe the XML type of the extended adapter configuration.

The new type must extend the `AdapterConfig` type, defined in `wlevs_application_config.xsd`. `AdapterConfig` extends `ConfigurationObject`. You can then add new elements or attributes to the basic adapter configuration as needed. For example, the following type called `HelloWorldAdapterConfig` adds a `<message>` element to the basic adapter configuration:

```
<xs:complexType name="HelloWorldAdapterConfig">
  <xs:complexContent>
    <xs:extension base="wlevs:AdapterConfig">
      <xs:sequence>
        <xs:element name="message" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

4. Define a top-level element that *must* be named `<config>`.

In the definition of the `config` element, define a sequence of child elements that correspond to the components in your application. Typically the name of the elements should indicate what component they configure (adapter, processor, stream) although you can name them anything you want.

Each element must extend the `ConfigurationObject` XML type, either explicitly using the `<xs:extension base="base:ConfigurationObject"/>` XSD tag or by specifying an XML type that itself extends `ConfigurationObject`. The `ConfigurationObject` XML type, defined in `wlevs_base_config.xsd`, defines a single attribute: `name`.

The type of your adapter element should be the custom one you created in a preceding step of this procedure.

You can use the following built-in XML types, described in `wlevs_application_config.xsd`, for the child elements of `<config>` that correspond to processors or streams:

- `DefaultProcessorConfig`—See [“Overview of the Complex Event Processor Configuration File” on page 7-1](#) for a description of the default processor configuration.
- `DefaultStreamConfig`—See [“Overview of the Stream Configuration File” on page 6-1](#) for a description of the default stream configuration.

For example:

```
<xs:element name="config">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="adapter" type="HelloWorldAdapterConfig"/>
      <xs:element name="processor"
type="wlevs:DefaultProcessorConfig"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

5. Optionally use the `<jxb:package>` child element of `<jxb:schemaBindings>` to specify the package name of the generated Java code:

```
<xs:annotation>
  <xs:appinfo>
    <jxb:schemaBindings>
      <jxb:package name="com.bea.adapter.wlevs.example.helloworld"/>
    </jxb:schemaBindings>
  </xs:appinfo>
</xs:annotation>
```

## Complete Example of an Extended XSD Schema File

Use the following extended XSD file as a template:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
targetNamespace="http://www.bea.com/xml/ns/wlevs/example/helloworld"
  xmlns="http://www.bea.com/xml/ns/wlevs/example/helloworld"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
  xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
  xmlns:wlevs="http://www.bea.com/xml/ns/wlevs/config/application"
  jxb:extensionBindingPrefixes="xjc" jxb:version="1.0"
  elementFormDefault="unqualified" attributeFormDefault="unqualified">

  <xs:annotation>
    <xs:appinfo>
      <jxb:schemaBindings>
        <jxb:package
name="com.bea.adapter.wlevs.example.helloworld"/>
      </jxb:schemaBindings>
    </xs:appinfo>
  </xs:annotation>
```

```
<xs:import
namespace="http://www.bea.com/xml/ns/wlevs/config/application"
schemaLocation="wlevs_application_config.xsd"/>

<xs:element name="config">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="adapter"
type="HelloWorldAdapterConfig"/>
      <xs:element name="processor"
type="wlevs:DefaultProcessorConfig"/>
      <xs:element name="stream"
type="wlevs:DefaultStreamConfig"/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:complexType name="HelloWorldAdapterConfig">
  <xs:complexContent>
    <xs:extension base="wlevs:AdapterConfig">
      <xs:sequence>
        <xs:element name="message" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
</xs:schema>
```

## Programming Access to the Configuration of an Adapter or Event Bean

This section applies to both adapters and event beans. For simplicity the text mentions only adapters.

When you deploy your application, Oracle CEP maps the configuration of each component (specified in the component configuration XML files) into Java objects using the [Java Architecture for XML Binding \(JAXB\)](#) standard. Because there is a single XML element that contains the configuration data for each component, JAXB in turn also produces a single Java class that represents this configuration data. Oracle CEP passes an instance of this Java class to the component (processor, stream, or adapter) at runtime when the component is initialized, and also whenever there is a dynamic change to the component's configuration.

In your adapter implementation, you can use metadata annotations to specify the Java methods that are invoked by Oracle CEP at runtime. Oracle CEP passes an instance of the configuration Java class to the specified methods; you can then program these methods to get specific runtime



configuration information about the adapter. The following example shows how to annotate the `activateAdapter()` method with the `@Activate` annotation to specify the method invoked when the adapter configuration is first activated:

```
@Activate
public void activateAdapter(HelloWorldAdapterConfig adapterConfig) {
    this.message = adapterConfig.getMessage();
}
```

By default, the data type of the adapter configuration Java class is `com.bea.wlevs.configuration.application.DefaultAdapterConfig`. If, however, you have extended the configuration of your adapter by creating your own XSD file that describes the configuration XMLfile, then you specify the type in the XSD file. In the preceding example, the data type of the Java configuration object is

```
com.bea.wlevs.example.helloworld.HelloWorldAdapterConfig.
```

The metadata annotations provided are as follows:

- `com.bea.wlevs.management.Activate`—Specifies the method invoked when the configuration is activated.

See [Activate](#) for additional details about using this annotation in your adapter code.

- `com.bea.wlevs.management.Prepare`—Specifies the method invoked when the configuration is prepared.

See [Prepare](#) for additional details about using this annotation in your adapter code.

- `com.bea.wlevs.management.Rollback`—Specifies the method invoked when the adapter is terminated due to an exception.

See [Rollback](#) for additional details about using this annotation in your adapter code.

## Passing Login Credentials from an Adapter to the Data Feed Provider

If your adapter accesses an external data feed, the adapter might need to pass login credentials (username and password) to the data feed for user authentication.

The simplest, and least secure, way to do this is to hard-code the non-encrypted login credentials in your adapter Java code. However, this method does not allow you to encrypt the password or later change the login credentials without recompiling the adapter Java code.

The following procedure describes a different method that takes these two issues into account. In the procedure, it is assumed that the username to access the data feed is `juliet` and the password is `superSecret`.

1. Decide whether you want the login credentials to be configured statically in the EPN assembly file, or dynamically by extending the configuration of the adapter.

Configuring the credentials statically in the EPN assembly file is easier, but if the credentials later change you must restart the application for the update to the EPN assembly file to take place. Extending the adapter configuration allows you to change the credentials dynamically without restarting the application, but extending the configuration involves additional steps, such as creating an XSD file and compiling it into a JAXB object.

### 2. If you decide to configure the login credentials statically, follow these steps:

- a. Open a command window and set your environment as described in [Setting Up Your Development Environment](#).
- b. Change to the directory that contains the EPN assembly file for your application.
- c. Using your favorite XML editor, edit the EPN assembly file by updating the `<wlevs:adapter>` tag that declares your adapter. In particular, add two instance properties that correspond to the username and password of the login credentials. For now, specify the cleartext password value; you will encrypt it in a later step. Also add a temporary `<password>` element whose value is the cleartext password. For example:

```
<wlevs:adapter id="myAdapter" provider="myProvider">
  <wlevs:instance-property name="user" value="juliet"/>
  <wlevs:instance-property name="password" value="superSecret"/>
  <password>superSecret</password>
</wlevs:adapter>
```

- d. Save the EPN assembly file.
- e. Execute the following `encryptMSAConfig` command to encrypt the value of the `<password>` element in the EPN assembly file:

```
prompt> ORACLE_CEP_HOME/ocp_10.3/bin/encryptMSAConfig .
epn_assembly_file msainternal.dat_file
```

where `ORACLE_CEP_HOME` refers to the main directory into which you installed Oracle CEP, such as `d:\oracle_cep`. The second argument refers to the directory that contains the EPN assembly file; because this procedure directs you to change to the directory, the example shows `"."`. The `epn_assembly_file` parameter refers to the name of your EPN assembly file. Finally, the `msainternal.dat_file` parameter

refers to the location of the `.msainternal.dat` file associated with your domain; by default this file is located in the `DOMAIN_DIR/servername` directory, where `DOMAIN_DIR` refers to the domain directory such as `/oracle_cep/user_projects/domains/mydomain` and `servername` refers to the server instance.

The `encryptMSAConfig` command comes in two flavors: `encryptMSAConfig.cmd` (Windows) and `encryptMSAConfig.sh` (UNIX).

After you run the command, the value of the `<password>` element of the EPN assembly file will be encrypted.

- f. Edit the EPN assembly file. Copy the encrypted value of the `<password>` element to the value attribute of the password instance property. Remove the `<password>` element from the XML file. For example:

```
<wlevs:adapter id="myAdapter" provider="myProvider">
  <wlevs:instance-property name="user" value="juliet"/>
  <wlevs:instance-property name="password"
    value="{Salted-3DES}B7L6nehu7dgPtJJTnTJWRA==" />
</wlevs:adapter>
```

### 3. If you decide to configure the login credentials dynamically, follow these steps:

- a. Extend the configuration of your adapter by adding two new elements: `<user>` and `<password>`, both of type string.

For example, if you were extending the adapter in the HelloWorld example, the XSD file might look like the following:

```
<xs:complexType name="HelloWorldAdapterConfig">
  <xs:complexContent>
    <xs:extension base="wlevs:AdapterConfig">
      <xs:sequence>
        <xs:element name="message" type="xs:string"/>
        <xs:element name="user" type="xs:string"/>
        <xs:element name="password" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

See [“Extending the Configuration of an Adapter or Event Bean” on page 3-17](#) for detailed instructions.

- b. Open a command window and set your environment as described in [Setting Up Your Development Environment](#).

- c. Change to the directory that contains the component configuration XML file for your adapter.
- d. Using your favorite XML editor, update this component configuration XML file by adding the required login credentials using the `<user>` and `<password>` elements. For now, specify the cleartext password value; you will encrypt it in a later step. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<myExample:config
xmlns:myExample="http://www.bea.com/xml/ns/wlevs/example/myExample">
  <adapter>
    <name>myAdapter</name>
    <user>juliet</user>
    <password>superSecret</password>
  </adapter>
</myExample:config>
```

- e. Save the adapter configuration file.
- f. Execute the following `encryptMSAConfig` command to encrypt the value of the `<password>` element in the adapter configuration file:

```
prompt> ORACLE_CEP_HOME/ocep_10.3/bin/encryptMSAConfig .
adapter_config_file msainternal.dat_file
```

where `ORACLE_CEP_HOME` refers to the main directory into which you installed Oracle CEP, such as `d:\oracle_cep`. The second argument refers to the directory that contains the adapter configuration file; because this procedure directs you to change to the directory, the example shows `". "`. The `adapter_config_file` parameter refers to the name of your adapter configuration file. Finally, the `msainternal.dat_file` parameter refers to the location of the `.msainternal.dat` file associated with your domain; by default this file is located in the `DOMAIN_DIR/servername` directory, where `DOMAIN_DIR` refers to the domain directory such as `/oracle_cep/user_projects/domains/mydomain` and `servername` refers to the server instance.

The `encryptMSAConfig` command comes in two flavors: `encryptMSAConfig.cmd` (Windows) and `encryptMSAConfig.sh` (UNIX).

After you run the command, the value of the `<password>` element will be encrypted.

4. Update your adapter Java code to access the login credentials properties you have just configured and decrypt the password.

See [“Updating the Adapter Code to Access the Login Credential Properties”](#) on page 3-27.

5. Edit the `MANIFEST.MF` file of the application and add the `com.bea.core.encryption` package to the `Import-Package` header. See [“Creating the MANIFEST.MF File” on page 10-4](#).
6. Re-assemble and deploy your application as usual. See [“Assembling and Deploying Oracle Complex Event Processing Applications” on page 10-1](#).

## Updating the Adapter Code to Access the Login Credential Properties

This section describes how update your adapter Java code to dynamically get the user and password values from the extended adapter configuration, and then use the `com.bea.core.encryption.EncryptionService` API to decrypt the encrypted password. The code snippets below build on the HelloWorld adapter Java code, shown in [“Programming Adapters and Event Bean Classes as Event Sources: Guidelines” on page 3-5](#).

- Import the additional APIs that you will need to decrypt the encrypted password:

```
import com.bea.core.encryption.EncryptionService;
import com.bea.core.encryption.EncryptionServiceException;
import com.bea.wlevs.util.Service;
```

- Use the `@Service` annotation to get a reference to the `EncryptionService`:

```
private EncryptionService encryptionService;

...

@Service
public void setEncryptionService(EncryptionService encryptionService) {
    this.encryptionService = encryptionService;
}
```

- In the `@Prepare` callback method, get the values of the user and password properties of the extended adapter configuration as usual (only code for the password value is shown):

```
private String password;

...

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

...
```

```
@Prepare
public void checkConfiguration(HelloWorldAdapterConfig adapterConfig) {
    if (adapterConfig.getMessage() == null
        || adapterConfig.getMessage().length() == 0) {
        throw new RuntimeException("invalid message: " + message);
    }
    this.password= adapterConfig.getPassword();
    ...
}
```

See [“Programming Access to the Configuration of an Adapter or Event Bean” on page 3-22](#) for information about accessing the extended adapter configuration.

- Use the `EncryptionService.decryptStringAsCharArray()` method in the `@Prepare` callback method to decrypt the encrypted password:

```
@Prepare
public void checkConfiguration(HelloWorldAdapterConfig adapterConfig) {
    if (adapterConfig.getMessage() == null
        || adapterConfig.getMessage().length() == 0) {
        throw new RuntimeException("invalid message: " + message);
    }
    this.password= adapterConfig.getPassword();
    try {
        char[] decrypted =
        encryptionService.decryptStringAsCharArray(password);
        System.out.println("DECRYPTED PASSWORD is " + new
        String(decrypted));
    } catch (EncryptionServiceException e) {
        throw new RuntimeException(e);
    }
}
```

The signature of the `decryptStringAsCharArray()` method is as follows:

```
char[] decryptStringAsCharArray(String encryptedString)
                                throws EncryptionServiceException
```

- Pass these credentials to the data feed provider using the vendor API.

# Using the Java Message Service (JMS) Adapters

This section contains information on the following subjects:

- [“Overview of Using the JMS Adapters” on page 4-1](#)
- [“Using JMS Adapters: Typical Steps” on page 4-2](#)

## Overview of Using the JMS Adapters

Oracle CEP provides two JMS adapters that you can use in your event applications to send and receive messages to and from a JMS queue, respectively, without writing any Java code. In particular:

- The inbound JMS adapter receives messages from a JMS queue and automatically converts them into events by matching property names with a specified event type. Typically, you also customize this conversion by writing your own Java class to specify exactly how you want the incoming JMS messages to be converted into one or more event types.
- The outbound JMS adapter sends events to a JMS queue, automatically converting the event into a JMS map message by matching property names with the event type. Typically, you also customize this conversion by writing your own Java class to specify exactly how you want the event types to be converted into outgoing JMS messages.

If you do not provide your own converter class, and instead let Oracle CEP take care of the conversion between messages and event types, the following is true:

- You must specify an event type that Oracle CEP uses in its conversion. See [“Configuring the JMS Adapters” on page 4-7](#) for details.

- The inbound JMS adapter default converter expects incoming JMS messages to be of type `MapMessage`. For each incoming message, an event of the specified event type is created. For each map element in the incoming message, the adapter looks for a property on the event-type and if found, sets the corresponding value.
- The outbound JMS adapter default converter creates JMS messages of type `MapMessage`. For each property of the event, a corresponding element is created in the output `MapMessage`.

Oracle CEP supports the following two JMS providers: Oracle WebLogic JMS and TIBCO EMS JMS.

Oracle CEP includes a WebLogic JMS client. When connecting to Oracle WebLogic server, Oracle CEP uses the T3 client by default. You can use the IIOP WebLogic client by starting Oracle WebLogic Server with the `-useIIOP` command-line argument. This is a server-wide setting that is independent of the JMS code being used (whether it is one of the provided adapters or custom JMS code). It is not possible to mix T3 and IIOP usage within a running Oracle CEP server.

If you are using a JMS provider other than WebLogic JMS, such as TIBCO, you must include the appropriate client jar as a library within your application jar.

For general information about JMS, see [Java Message Service](#) on the Sun Developer Network.

## Using JMS Adapters: Typical Steps

The following procedure describes the typical steps to use the JMS adapters provided by Oracle CEP.

**Note:** It is assumed in this section that you have already created an Oracle CEP application along with its EPN assembly file and component configuration files, and that you want to update the application to use an inbound or outbound JMS adapter. If you have not, refer to [“Overview of Creating Oracle Complex Event Processing Applications” on page 2-1](#) for details.

1. Optionally create a converter Java class if you want to customize the way JMS messages are converted into event types, or vice versa in the case of the outbound JMS adapter. This step is optional because you can let Oracle CEP make the conversion based on mapping property names between JMS messages and a specified event type.

See [“Creating a Custom Converter Between JMS Messages and Event Types” on page 4-3](#).

2. Update the EPN assembly file of the application by adding a `<wlevs:adapter>` tag for each inbound and outbound JMS adapter you want to use in your application.



See [“Updating the EPN Assembly File With JMS Adapters” on page 4-4](#)

3. Configure the JMS properties of the JMS adapter by updating the component configuration file.

See [“Configuring the JMS Adapters” on page 4-7](#).

4. Update the `MANIFEST.MF` file of your application, adding the package `com.bea.core.encryption` to the `Import-Package` header. For example:

```
Import-Package:
    com.bea.core.encryption
    com.bea.wlevs.adapter.defaultprovider;version="2.0.0.0",
    ...
```

See [“Creating the MANIFEST.MF File” on page 10-4](#) for additional information on the manifest file.

## Creating a Custom Converter Between JMS Messages and Event Types

If you want to customize the way a JMS message is converted to an event type, or vice versa, you must create your own converter bean.

The custom converter bean for an inbound JMS must implement the `com.bea.wlevs.adapters.jms.api.InboundMessageConverter` interface. This interface has a single method:

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

The `message` parameter corresponds to the incoming JMS message and the return value is a `List` of events that will be passed on to the next stage of the event processing network.

The custom converter bean for an outbound JMS must implement the `com.bea.wlevs.adapters.jms.api.OutboundMessageConverter` interface. This interface has a single method:

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

The parameters correspond to an event received by the outbound JMS adapter from the source node in the EPN and the return value is a `List` of JMS messages.

See the [Javadoc](#) for a full description of these APIs.

The following example shows the Java source of a custom converter bean that implements both `InboundMessageConverter` and `OutboundMessageConverter`; this bean can be used for both inbound and outbound JMS adapters:

```
package com.customer;

import com.bea.wlevs.adapters.jms.api.InboundMessageConverter;
import com.bea.wlevs.adapters.jms.api.MessageConverterException;
import com.bea.wlevs.adapters.jms.api.OutboundMessageConverter;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.TextMessage;
import java.util.ArrayList;
import java.util.List;

public class MessageConverter implements InboundMessageConverter,
    OutboundMessageConverter {

    public List convert(Message message) throws MessageConverterException,
        JMSException {
        TestEvent event = new TestEvent();
        TextMessage textMessage = (TextMessage) message;
        event.setString_1(textMessage.getText());
        List events = new ArrayList(1);
        events.add(event);
        return events;
    }

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

## Updating the EPN Assembly File With JMS Adapters

For each JMS adapter in your event processing network, you must add a corresponding `<wlevs:adapter>` tag to the EPN assembly file of your application; use the `provider` attribute to specify whether the JMS adapter is inbound or outbound. Follow these guidelines:

- If you are specifying an inbound JMS adapter, set the `provider` attribute to `jms-inbound`, as shown:

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

The value of the `id` attribute, in this case `jmsInbound`, must match the name specified for this JMS adapter in its configuration file. The configuration file configures the JMS queue from which this inbound JMS adapter gets its messages.

Because no converter bean is specified, Oracle CEP automatically converts the inbound message to the event type specified in the component configuration file by mapping property names.

- If you are specifying an outbound JMS adapter, set the `provider` attribute to `jms-outbound`, as shown:

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

The value of the `id` attribute, in this case `jmsOutbound`, must match the name specified for this JMS adapter in its configuration file. The configuration file configures the JMS queue to which this outbound JMS adapter sends messages.

Because no converter bean is specified, Oracle CEP automatically converts the incoming event types to outgoing JMS messages by mapping the property names.

- For both inbound and outbound JMS adapters, if you have created a custom converter bean to customize the conversion between the JMS messages and event types, first use the standard `<bean>` Spring tag to declare it in the EPN assembly file. Then pass a reference of the bean to the JMS adapter by specifying its `id` using the `<wlevs:instance-property>` tag, with the `name` attribute set to `converterBean`, as shown:

```
<bean id="myConverter"
      class="com.customer.MessageConverter"/>

<wlevs:adapter id="jmsOutbound" provider="jms-outbound">
  <wlevs:instance-property name="converterBean" ref="myConverter"/>
</wlevs:adapter>
```

In this case, be sure you do *not* specify an event type in the component configuration file because it is assumed that the custom converter bean takes care of specifying the event type.

As with any other stage in the EPN, add listeners to the `<wlevs:adapter>` tag to integrate the JMS adapter into the event processing network. Typically, an inbound JMS adapter is the first stage in an EPN (because it receives messages) and an outbound JMS adapter would be in a later

stage (because it sends messages). However, the requirements of your own Oracle CEP application define where in the network the JMS adapters fit in.

The following sample EPN assembly file shows how to configure an outbound JMS adapter. The network is simple: a custom adapter called `getData` receives data from some feed, converts it into an event type and passes it to `myProcessor`, which in turn sends the events to the `jmsOutbound` JMS adapter via the `streamOne` stream. Oracle CEP automatically converts these events to JMS messages and sends the messages to the JMS queue configured in the component configuration file associated with the `jmsOutbound` adapter.

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

  <wlevs:event-type-repository>
    <wlevs:event-type type-name="JMSEvent">
      <wlevs:class>com.customer.JMSEvent</wlevs:class>
    </wlevs:event-type>
  </wlevs:event-type-repository>

  <!-- Custom adapter that gets data from somewhere and sends it to myProcessor -->
  <wlevs:adapter id="getData"
                class="com.customer.GetData">
    <wlevs:listener ref="myProcessor"/>
  </wlevs:adapter>

  <wlevs:processor id="myProcessor" />

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

  <!-- Stream for events flowing from myProcessor to outbound JMS adapter -->
  <wlevs:stream id="streamOne">
    <wlevs:listener ref="jmsOutbound"/>
    <wlevs:source ref="myProcessor"/>
  </wlevs:stream>

</beans>
```

The following sample EPN assembly file shows how to configure an inbound JMS adapter. The network is simple: the inbound JMS adapter called `jmsInbound` receives messages from the JMS queue configured in its component configuration file. The Spring bean `myConverter` converts the incoming JMS messages into event types, and then these events flow to the `mySink` event bean.

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

  <wlevs:event-type-repository>
    <wlevs:event-type type-name="JMSEvent">
      <wlevs:class com.customer.JMSEvent</wlevs:class>
    </wlevs:event-type>
  </wlevs:event-type-repository>

  <!-- Event bean that is an event sink -->
  <wlevs:event-bean id="mySink"
    class="com.customer.MySink"/>

  <!-- Inbound JMS adapter with custom converter class; adapter sends events
to mySink event bean-->
  <bean id="myConverter" class="com.customer.MessageConverter"/>
  <wlevs:adapter id="jmsInbound" provider="jms-inbound">
    <wlevs:instance-property name="converterBean" ref="myConverter"/>
    <wlevs:listener ref="mySink"/>
  </wlevs:adapter>
</beans>
```

## Configuring the JMS Adapters

You configure the JMS adapters in their respective configuration files, similar to how you configure other components in the event processing network, such as processors or streams. For general information about these configuration files, see [“Component Configuration Files” on page 2-3](#).

The root element for configuring a JMS adapter is `<jms-adapter>`. The `<name>` child element for a particular adapter must match the `id` attribute of the corresponding `<wlevs:adapter>` tag in the EPN assembly file that declares this adapter.

The following table describes the additional child elements of `<jms-adapter>` you can configure for both inbound and outbound JMS adapters.

**Table 4-1 Child Elements of `<jms-adapter>` for Inbound and Outbound Adapters**

Child Element	Description
event-type	Event type whose properties match the JMS message properties.  Specify this child element <i>only</i> if you want Oracle CEP to automatically perform the conversion between JMS messages and events. If you have created your own custom converter bean, then do not specify this element.
jndi-provider-url	Required. The URL of the JNDI provider.
jndi-factory	Optional. The JNDI factory name. Default value is <code>weblogic.jndi.WLInitialContextFactory</code> , for Oracle WebLogic Server JMS.
connection-jndi-name	Optional. The JNDI name of the JMS connection factory. Default value is <code>weblogic.jms.ConnectionFactory</code> , for Oracle WebLogic Server JMS.
destination-jndi-name, destination-name	Required. Either the JNDI name, or actual name, of the JMS destination. Specify one or the other, but not both.
user	Optional. The username for the external resource.
password	Optional. The password for the external resource.
encrypted-password	Optional. The encrypted password for the external resource.  <a href="#">See “Encrypting Passwords in the JMS Adapter Configuration File” on page 4-11</a> for details on encrypting the password.

The following table lists the optional child elements of `<jms-adapter>` you can configure for inbound JMS adapters only.

**Table 4-2 Optional Child Elements for Inbound JMS Adapters**

Child Element	Description
work-manager	Name of a work manager, configured in the server's <code>config.xml</code> file. This name corresponds to the value of the <code>&lt;name&gt;</code> child element of the <code>&lt;work-manager&gt;</code> element in <code>config.xml</code> . The default value is the work manager configured for the application itself.
concurrent-consumers	Number of consumers to create. Default value is 1. If you set this value to number greater than one, be sure that your converter bean is thread-safe. This is because the converter bean will be shared among the consumers.
message-selector	JMS message selector to use to filter messages.
session-ack-mode-name	Session acknowledgement mode.
session-transacted	Boolean value that specifies whether to use transacted sessions.

The following table lists the optional child elements of `<jms-adapter>` you can configure for outbound JMS adapters only.

**Table 4-3 Optional Child Elements for Outbound JMS Adapters**

Child Element	Description
delivery-mode	Specifies the delivery mode: <code>persistent</code> (default value) or <code>nonpersistent</code> .

For the full schema for the configuration of JMS adapters, see the [XSD Schema](#).

The following configuration file shows a complete example of configuring both an inbound and outbound JMS adapter.

```
<?xml version="1.0" encoding="UTF-8"?>
<nl:config
  xsi:schemaLocation="http://www.bea.com/ns/wlevs/config/application
wlevs_application_config.xsd"
```

## Using the Java Message Service (JMS) Adapters

```
xmlns:nl="http://www.bea.com/ns/wlevs/config/application"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <jms-adapter>
    <name>jmsInbound</name>
    <jndi-provider-url>t3://localhost:7001</jndi-provider-url>
    <destination-jndi-name>Queue1</destination-jndi-name>
    <user>weblogic</user>
    <password>weblogic</password>
    <work-manager>JettyWorkManager</work-manager>
    <concurrent-consumers>1</concurrent-consumers>
    <session-transacted>>false</session-transacted>
  </jms-adapter>

  <jms-adapter>
    <name>jmsOutbound</name>
    <event-type>JMSEvent</event-type>
    <jndi-provider-url>t3://localhost:7001</jndi-provider-url>
    <destination-jndi-name>Topic1</destination-jndi-name>
    <delivery-mode>nonpersistent</delivery-mode>
  </jms-adapter>

</nl:config>
```

The following snippet shows how to configure an inbound JMS adapter connecting to TIBCO EMS JMS:

```
<jms-adapter>
  <name>myJmsAdapter</name>
  <jndi-provider-url>t3://localhost:7222</jndi-provider-url>

<jndi-factory>com.tibco.tibjms.naming.TibjmsInitialContextFactory</jndi-fa
ctory>
  <connection-jndi-name>TibcoQueueConnectionFactory</connection-jndi-name>
  <destination-jndi-name>MyQueue</destination-jndi-name>
</jms-adapter>
```



## Encrypting Passwords in the JMS Adapter Configuration File

Use the following procedure if you want to encrypt the password in the JMS adapter configuration file.

**Note:** The procedure assumes that you are currently using the `<password>` element in the configuration file, along with a cleartext password value, but want to start using the `<encrypted-password>` element to encrypt the password.

1. Open a command window and set your environment as described in [Setting Up Your Development Environment](#).
2. Change to the directory that contains the configuration file for your JMS adapter.
3. Execute the following `encryptMSAConfig` command to encrypt the value of the `<password>` element in the configuration file:

```
prompt> ORACLE_CEP_HOME/ocep_10.3/bin/encryptMSAConfig . config_file
msainternal.dat_file
```

where `ORACLE_CEP_HOME` refers to the main BEA directory into which you installed Oracle CEP, such as `d:\oracle_cep`. The second argument refers to the directory that contains the JMS adapter configuration file; because this procedure directs you to actually change to the directory, the example shows `"."`. The `config_file` parameter refers to the name of your JMS adapter configuration file. Finally, the `msainternal.dat_file` parameter refers to the location of the `.msainternal.dat` file associated with your domain; by default this file is located in the `DOMAIN_DIR/servername` directory, where `DOMAIN_DIR` refers to the domain directory such as `/oracle_cep/user_projects/domains/mydomain` and `servername` refers to the server instance.

The `encryptMSAConfig` command comes in two flavors: `encryptMSAConfig.cmd` (Windows) and `encryptMSAConfig.sh` (UNIX).

After you run the command, the value of the `<password>` element will be encrypted, as shown in bold in the following example:

```
<jms-adapter>
  <name>jmsInbound</name>
  <jndi-provider-url>t3://localhost:7001</jndi-provider-url>
  <destination-jndi-name>Queue1</destination-jndi-name>
  <user>weblogic</user>
  <password>{Salted-3DES}B7L6nehu7dgPtJJTnTJWRA==</password>
  <work-manager>JettyWorkManager</work-manager>
  <concurrent-consumers>1</concurrent-consumers>
  <session-transacted>>false</session-transacted>
</jms-adapter>
```

4. Using your favorite XML editor, edit the JMS adapter configuration file. Change the `<password>` element (whose value is now encrypted) to `<encrypted-password>`, as shown in bold in the following example:

```
<jms-adapter>
  <name>jmsInbound</name>
  <jndi-provider-url>t3://localhost:7001</jndi-provider-url>
  <destination-jndi-name>Queue1</destination-jndi-name>
  <user>weblogic</user>

  <encrypted-password>{Salted-3DES}B7L6nehu7dgPtJJTnTJWRA==</encrypted-pa
ssword>
  <work-manager>JettyWorkManager</work-manager>
  <concurrent-consumers>1</concurrent-consumers>
  <session-transacted>>false</session-transacted>
</jms-adapter>
```

# Using and Creating HTTP Publish-Subscribe Adapters

This section contains information on the following subjects:

- [“Overview of HTTP Publish-Subscribe Functionality in Oracle CEP” on page 5-1](#)
- [“Using the Built-In HTTP Pub-Sub Adapters in an Application: Typical Steps” on page 5-9](#)

## Overview of HTTP Publish-Subscribe Functionality in Oracle CEP

An *HTTP Publish-Subscribe Server* (for simplicity, also called *pub-sub server* in this document) is a mechanism whereby Web clients, such as browser-based clients, subscribe to channels, receive messages as they become available, and publish messages to these channels, all using asynchronous messages over HTTP. A channel is similar to a JMS topic. For additional general information about HTTP Publish-Subscribe Servers, see [Using the HTTP Publish-Subscribe Server](#).

Every instance of Oracle CEP includes a pub-sub server that programmers can use to implement HTTP publish-subscribe functionality in their applications. The pub-sub server is configured in the `config.xml` file along with other server services such as Jetty and JDBC datasources. The pub-sub server is based on the [Bayeux protocol](#) proposed by the [cometd](#) project. The Bayeux protocol defines a contract between the client and the server for communicating with asynchronous messages over HTTP.

In Oracle CEP, programmers access HTTP publish-subscribe functionality by using three built-in HTTP publish-subscribe adapters (called *pub-sub adapters* for short): two for local and remote

publishing and one for subscribing to a channel. Oracle CEP also provides a pub-sub API for programmers to create their own custom pub-sub adapters for publishing and subscribing to a channel, if the built-in pub-sub adapters are not adequate. For example, programmers might want to filter incoming messages from a subscribed channel, dynamically create or destroy local channels, and so on. The built-in pub-sub adapters do not provide this functionality, which is why programmers must implement their own custom pub-sub adapters in this case.

The three built-in pub-sub adapters work like any other adapter: they are stages in the event processing network, they are defined in the EPN assembly file, and they are configured with the standard component configuration files. Typical configuration options include specifying channels, specifying the local or remote pub-sub server, and user authentication.

The following sections provide additional conceptual information about these built-in pub-sub adapters:

- [“Overview of the Built-In Pub-Sub Adapter for Local Publishing” on page 5-3](#)
- [“Overview of the Built-In Pub-Sub Adapter for Remote Publishing” on page 5-5](#)
- [“Overview of the Built-In Pub-Sub Adapter for Subscribing” on page 5-7](#)

The pub-sub server can communicate with any client that can understand the Bayeux protocol. Programmers develop their Web clients using one of the following frameworks:

- [Dojo](#) JavaScript library that supports the Bayeux protocol. Oracle CEP does not provide this library.
- WebLogic Workshop Flex plug-in that enables development of a Flex client that uses the Bayeux protocol to communicate with a pub-sub server.

## Converting Between JSON Messages and Event Types

Oracle CEP can automatically convert incoming JSON messages to event types, and vice versa in the outbound case. However, if you want to customize the way a JSON message (either *inbound* via a HTTP pub-sub adapter for subscribing or *outbound* via an HTTP pub-sub adapter for publishing) is converted to an event type, or vice versa, you must create your own converter bean. See [“Creating a Custom Converter Between the HTTP Pub-Sub Messages and Event Types” on page 5-10](#) for details.

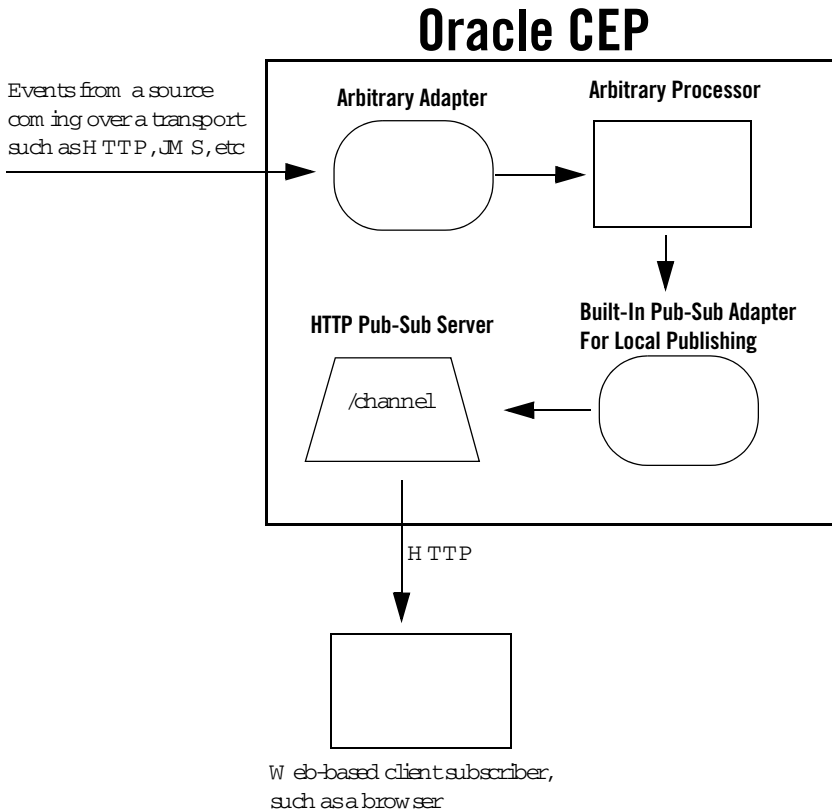
If you do *not* provide your own converter class, and instead let Oracle CEP take care of the conversion between messages and event types, the following is true:

- You must specify an event type that Oracle CEP uses in its conversion. See [“Configuring the HTTP Pub-Sub Adapters” on page 5-11](#) for details.
- The default converter used in the HTTP adapter for subscribing creates a new event of the specified type for each incoming message. For each property of the specified event type, it looks for a corresponding property name in the JSON object that constitutes the message, and if found, sets the corresponding value.
- The default converter used in the HTTP adapter for publishing creates a JSON message for each event. For each property of the specified event type, a corresponding element is created in the output JSON message.

## Overview of the Built-In Pub-Sub Adapter for Local Publishing

The following graphic shows how the built-in pub-sub adapter for local publishing fits into a simple event processing network. The arbitrary adapter and processor are not required, they are just an example of possible components in your application in addition to the pub-sub adapter.

**Figure 5-1 Built-In Pub-Sub Adapter For Local Publishing**



In the preceding graphic:

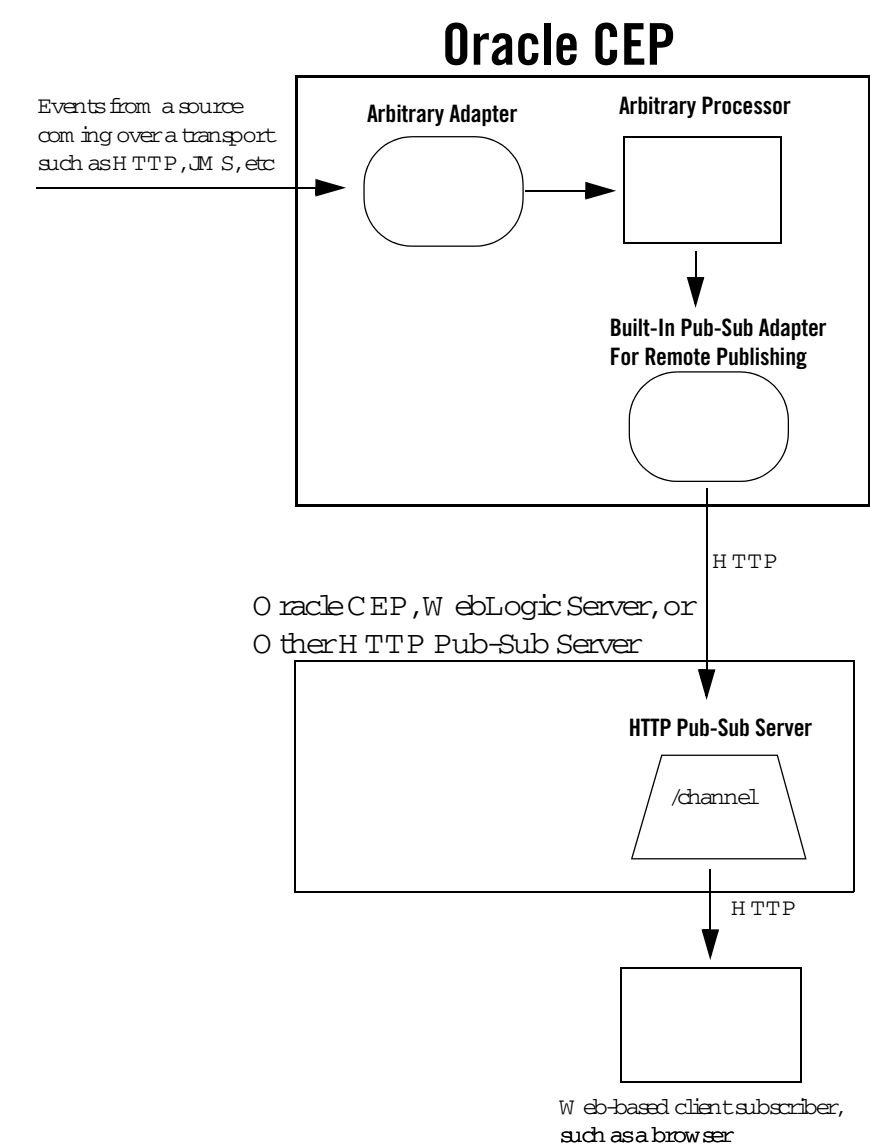
- Events flow from some source into an adapter of an application running in Oracle CEP. This adapter is not required, it is shown only as an example.
- The events flow from the adapter to an arbitrary processor; again, this processor is not required.
- The processor sends the events to the built-in pub-sub adapter for local publishing. The adapter in turn sends the events to the local HTTP pub-sub server configured for the Oracle CEP instance on which the application is deployed. The pub-sub adapter sends the messages to the channel for which it has been configured.

- The local HTTP pub-sub server configured for Oracle CEP then sends the event as a message to all subscribers of the local channel.

## **Overview of the Built-In Pub-Sub Adapter for Remote Publishing**

The following graphic shows how the built-in pub-sub adapter for remote publishing fits into a simple event processing network.

Figure 5-2 Built-In Pub-Sub Adapter For Remote Publishing



In the preceding graphic:

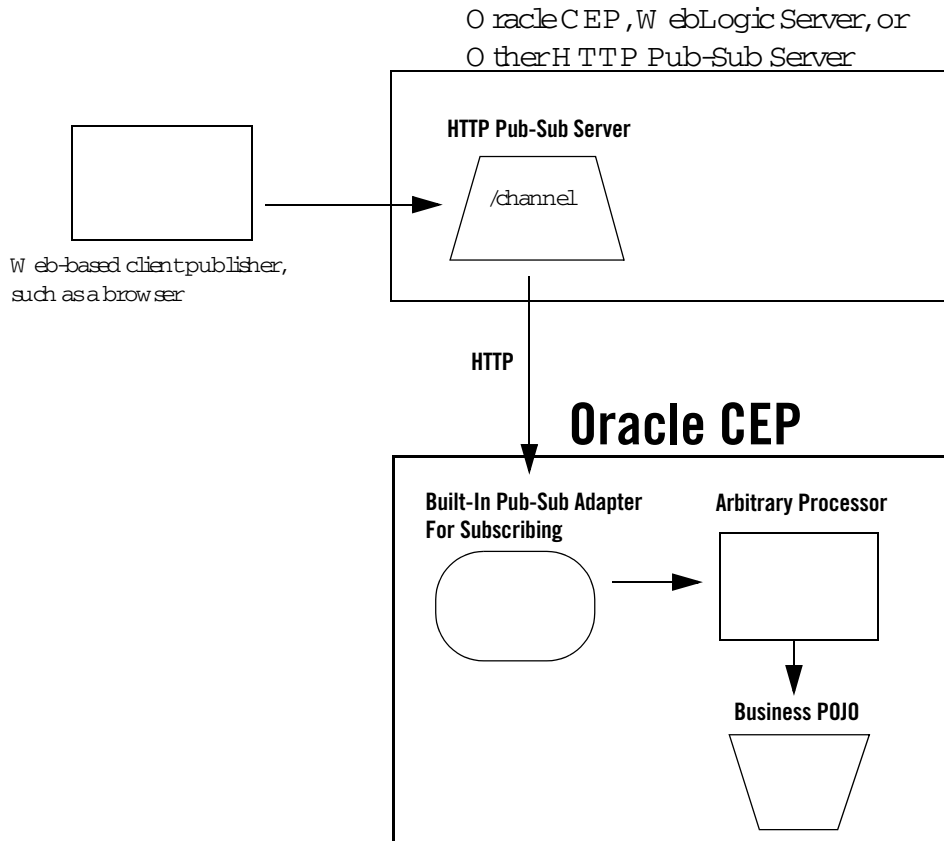


- Events flow from some source into an adapter of an application running in Oracle CEP. The arbitrary adapter is not required, it is shown only as an example.
- The events flow from the adapter to an arbitrary processor; again, this processor is not required.
- The processor sends the events to the built-in pub-sub adapter for remote publishing. The adapter in turn sends the events as messages to the remote HTTP pub-sub server for which the adapter is configured; this HTTP pub-sub server could be on another Oracle CEP instance, a WebLogic Server instance, or any other third-party implementation. The pub-sub adapter sends the messages to the channel for which it has been configured.
- The remote HTTP pub-sub server then sends the message to all subscribers of the channel.

## Overview of the Built-In Pub-Sub Adapter for Subscribing

The following graphic shows how the built-in pub-sub adapter for subscribing fits into a simple event processing network. The arbitrary processor and business POJO are not required, they are just an example of possible components in your application in addition to the pub-sub adapter.

**Figure 5-3 Built-In Pub-Sub Adapter For Subscribing**



In the preceding graphic:

- Messages are published to a remote HTTP pub-sub server, which could be another instance of Oracle CEP, WebLogic Server, or a third-party implementation. The messages are typically published by Web based clients (shown in graphic), by the HTTP pub-sub server itself, or another server application.
- The built-in pub-sub adapter running in an Oracle CEP application subscribes to the HTTP pub-sub server and receives messages from the specified channel. The adapter converts the messages into the event type configured for the adapter.

- The pub-sub adapter sends the events to a processor. This processor is not required, it is shown only as an example of a typical Oracle CEP application.
- The processor sends the events to a business POJO. Again, this business POJO is not required.

## Using the Built-In HTTP Pub-Sub Adapters in an Application: Typical Steps

The following procedure describes typical steps for using the *built-in* pub-sub adapters in your Oracle CEP application.

**Note:** It is assumed in this section that you have already created an Oracle CEP application, along with its EPN assembly file and component configuration files, and that you want to update the application to use the built-in pub-sub adapters. If this is not true, refer to [“Overview of Creating Oracle Complex Event Processing Applications” on page 2-1](#) for general information about creating an Oracle CEP application.

1. Optionally create a converter Java class if you want to customize the way the inbound or outbound messages are converted into event types. This step is optional because you can let Oracle CEP make the conversion based on mapping property names between the messages and a specified event type.

See [“Creating a Custom Converter Between the HTTP Pub-Sub Messages and Event Types” on page 5-10](#).

2. If you are going to use the local HTTP pub-sub server associated with the Oracle CEP instance for local publishing, use Visualizer, the Oracle CEP Administration Tool, to add new channels with the channel pattern required by your application.

For details, see [Configuring the HTTP Publish-Subscribe Server](#) in the Visualizer Online Help.

3. Configure the built-in pub-sub adapters you are going to add to your application by updating the component configuration files.

See [“Configuring the HTTP Pub-Sub Adapters” on page 5-11](#).

4. Update the EPN assembly file, adding declarations for each built-in pub-sub adapter you are adding to your application.

See [“Updating the EPN Assembly File” on page 5-15](#).

5. Update the `MANIFEST.MF` file of your application, adding the package `com.bea.core.encryption` to the `Import-Package` header. For example:

```
Import-Package:  
  com.bea.core.encryption  
  com.bea.wlevs.adapter.defaultprovider;version="2.0.0.0",  
  ...
```

See [“Creating the MANIFEST.MF File” on page 10-4](#) for additional information on the manifest file.

## Creating a Custom Converter Between the HTTP Pub-Sub Messages and Event Types

If you want to customize the way a message (either *inbound* via a HTTP pub-sub adapter for subscribing or *outbound* via an HTTP pub-sub adapter for publishing) is converted to an event type, or vice versa, you must create your own converter bean.

The custom converter bean for an inbound HTTP pub-sub message must implement the `com.bea.wlevs.adapters.httppubsub.api.InboundMessageConverter` interface. This interface has a single method:

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

The `message` parameter corresponds to the incoming HTTP pub-sub message and the return value is a `List` of events that will be passed on to the next stage of the event processing network. The incoming message is assumed to be the JSON format.

The custom converter bean for an outbound HTTP pub-sub message must implement the `com.bea.wlevs.adapters.httppubsub.api.OutboundMessageConverter` interface. This interface has a single method:

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

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

See the [Javadoc](#) for a full description of these APIs.

The following example shows the Java source of a custom converter bean that implements both `InboundMessageConverter` and `OutboundMessageConvert`; this bean can be used for both inbound and outbound HTTP pub-sub adapters:

```
package com.sample.httppubsub;
```

```

import com.bea.wlevs.adapters.httppubsub.api.InboundMessageConverter;
import com.bea.wlevs.adapters.httppubsub.api.OutboundMessageConverter;
import com.bea.httppubsub.json.JSONObject;

import java.util.List;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

public class TestConverter implements InboundMessageConverter,
OutboundMessageConverter {

    public List convert(JSONObject message) throws Exception {
        List eventCollection = new ArrayList();
        PubsubTestEvent event = new PubsubTestEvent();
        event.setMessage("From TestConverter: " + message);
        eventCollection.add(event);
        return eventCollection;
    }

    public List<JSONObject> convert(Object event) throws Exception {
        List<JSONObject> list = new ArrayList<JSONObject>(1);
        Map map = new HashMap();
        map.put("message", ((PubsubTestEvent) event).getMessage());
        list.add(new JSONObject(map));
        return list;
    }
}

```

## Configuring the HTTP Pub-Sub Adapters

You configure the built-in pub-sub adapters in their respective configuration files, similar to how you configure other components in the event processing network, such as processors or streams. For general information about these configuration files, see [“Component Configuration Files” on page 2-3](#).

The following configuration file shows a complete example of configuring each of the three built-in pub-sub adapters; the procedure following the example uses the example to show how to create your own file:

```

<?xml version="1.0" encoding="UTF-8"?>

<n1:config
  xsi:schemaLocation="http://www.bea.com/ns/wlevs/config/application
wlevs_application_config.xsd"
  xmlns:n1="http://www.bea.com/ns/wlevs/config/application"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

```

```
<http-pub-sub-adapter>
  <name>remotePublisher</name>
  <server-url>http://myhost.com:9102/pubsub</server-url>
  <channel>/channel1</channel>
  <event-type>com.mycompany.httppubsub.PubsubEvent</event-type>
  <user>wlevs</user>
  <password>wlevs</password>
</http-pub-sub-adapter>

<http-pub-sub-adapter>
  <name>localPublisher</name>
  <server-context-path>/pubsub</server-context-path>
  <channel>/channel2</channel>
</http-pub-sub-adapter>

<http-pub-sub-adapter>
  <name>remoteSubscriber</name>
  <server-url>http://myhost.com:9102/pubsub</server-url>
  <channel>/channel3</channel>
  <event-type>com.mycompany.httppubsub.PubsubEvent</event-type>
</http-pub-sub-adapter>

</nl:config>
```

The following procedure describes the main steps to configure the built-in pub-sub adapters for your application. For simplicity, it is assumed in the procedure that you are going to configure all components of an application in a single configuration XML file and that you have already created this file for your application.

See [XSD Schema Reference for Component Configuration Files](#) for the complete XSD Schema that describes the configuration of the built-in pub-sub adapters.

1. Open the configuration XML file using your favorite XML editor.
2. For each built-in pub-sub adapter you want to configure, add a `<http-pub-sub-adapter>` child element of the `<config>` root element; use the `<name>` child element to uniquely identify it. This name value will be used later as the `id` attribute of the `<wlevs:adapter>` tag in the EPN assembly file that defines the event processing network of your application. This is how Oracle CEP knows to which particular adapter in the EPN assembly file this adapter configuration applies.

For example, assume your configuration file already contains a processor (contents removed for simplicity) and you want to configure instances of each of the three built-in pub-sub adapters; then the updated file might look like the following; details of the adapter configuration will be added in later steps:

```

<?xml version="1.0" encoding="UTF-8"?>

<nl:config
  xsi:schemaLocation="http://www.bea.com/ns/wlevs/config/application
wlevs_application_config.xsd"
  xmlns:nl="http://www.bea.com/ns/wlevs/config/application"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <processor>
    ...
  </processor>

  <http-pub-sub-adapter>
    <name>remotePublisher</name>
    ...
  </http-pub-sub-adapter>

  <http-pub-sub-adapter>
    <name>remoteSubscriber</name>
    ...
  </http-pub-sub-adapter>

  <http-pub-sub-adapter>
    <name>localPublisher</name>
    ...
  </http-pub-sub-adapter>

</nl:config>

```

3. For each *remote* pub-sub adapter (for both publishing and subscribing), add a `<server-url>` child element of `<http-pub-sub-adapter>` to specify the URL of the *remote* HTTP pub-sub server to which the Oracle CEP application will publish or subscribe, respectively. The remote pub-sub server could be another instance of Oracle CEP, or a WebLogic Server instance, or it could be any third-party HTTP pub-sub server. For example:

```

<http-pub-sub-adapter>
  <name>remotePublisher</name>
  <server-url>http://myhost.com:9102/pubsub</server-url>
  ...
</http-pub-sub-adapter>

```

In the example, the URL of the remote HTTP pub-sub server to which the `remotePublisher` adapter will publish events is `http://myhost.com:8102/pubsub`.

4. For each *local* pub-sub adapter for publishing, add a `<server-context-path>` element to specify the path of the local HTTP pub-sub server associated with the Oracle CEP instance hosting the current Oracle CEP application.

By default, each Oracle CEP server is configured with an HTTP pub-sub server with path `/pubsub`; if, however, you have created a new local HTTP pub-sub server, or changed the

default configuration, then specify the value of the `<path>` child element of the `<http-pubsub>` element in the server's `config.xml` file. For example:

```
<http-pub-sub-adapter>
  <name>localPublisher</name>
  <server-context-path>/pubsub</server-context-path>
  ...
</http-pub-sub-adapter>
```

5. For *all* the pub-sub adapters, whether they are local or remote or for publishing or subscribing, add a `<channel>` child element to specify the channel that the pub-sub adapter publishes or subscribes to, whichever is appropriate. For example:

```
<http-pub-sub-adapter>
  <name>localPublisher</name>
  <server-context-path>/pubsub</server-context-path>
  <channel>/channel2</channel>
</http-pub-sub-adapter>
```

In the example, the `localPublisher` pub-sub adapter publishes to a local channel with pattern `/channel2`.

6. For all pub-sub adapters for subscribing, add an `<event-type>` element that specifies the JavaBean to which incoming messages are mapped. You are required to specify this for all subscribing adapters. At runtime, Oracle CEP uses the incoming key-value pairs in the message to map the message data to the specified event type.

You can also optionally use the `<event-type>` element in a pub-sub adapter for publishing if you want to limit the types of events that are published to just those specified by the `<event-type>` elements. Otherwise, all events sent to the pub-sub adapter are published. For example:

```
<http-pub-sub-adapter>
  <name>remoteSubscriber</name>
  <server-url>http://myhost.com:9102/pubsub</server-url>
  <channel>/channel3</channel>
  <event-type>com.mycompany.httppubsub.PubsubEvent</event-type>
</http-pub-sub-adapter>
```

Be sure this event type has been registered in the EPN assembly file by specifying it as a child element of the `<wlevs:event-type-repository>` element.

7. Finally, if the HTTP pub-sub server to which the Oracle CEP application is publishing requires user authentication, add `<user>` and `<password>` (or `<encrypted-password>`) elements to specify the username and password or encrypted password. For example:

```
<http-pub-sub-adapter>
  <name>remotePublisher</name>
```



```

<server-url>http://myhost.com:9102/pubsub</server-url>
<channel>/channel1</channel>
<event-type>com.mycompany.httppubsub.PubsubEvent</event-type>
<user>wlevs</user>
<password>wlevs</password>
</http-pub-sub-adapter>

```

## Updating the EPN Assembly File

For each pub-sub adapter in your event processing network, you must add a corresponding `<wlevs:adapter>` tag to the EPN assembly file that describes the network; use the `provider` attribute to specify whether the adapter is for publishing or subscribing. Follow these guidelines:

- If you are using a built-in pub-sub adapter for publishing (either locally or remotely), set the `provider` attribute to `httppub`, as shown:

```
<wlevs:adapter id="remotePublisher" provider="httppub"/>
```

The value of the `id` attribute, in this case `remotePublisher`, must match the name specified for this built-in pub-sub adapter in its configuration file. Note that the declaration of the built-in adapter for publishing in the EPN assembly file does *not* specify whether this adapter is local or remote; you specify this in the adapter configuration file.

- If you are using a built-in pub-sub adapter for subscribing, set the `provider` attribute to `httpsub`, as shown:

```
<wlevs:adapter id="remoteSubscriber" provider="httpsub"/>
```

The value of the `id` attribute, in this case `remoteSubscriber`, must match the name specified for this built-in pub-sub adapter in its configuration file.

As with any other stage in the EPN, add listeners to the `<wlevs:adapter>` tag to integrate the pub-sub adapter into the event processing network. Typically, a pub-sub adapter for subscribing is the first stage in an EPN (because it receives messages) and a pub-sub adapter for publishing would be in a later stage (because it sends messages). However, the requirements of your own Oracle CEP application define where in the network the pub-sub adapters fit in.

Also be sure that the event types used by the pub-sub adapters have been registered in the event type repository using the `<wlevs:event-type-repository>` tag.

The following sample EPN file shows an event processing network with two built-in pub-sub adapters for publishing both local and remote publishing); see the text after the example for an explanation:

```
<?xml version="1.0" encoding="UTF-8"?>
```

## Using and Creating HTTP Publish-Subscribe Adapters

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       xmlns:wlevs="http://www.bea.com/ns/wlevs/spring"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd
http://www.bea.com/ns/wlevs/spring
http://www.bea.com/ns/wlevs/spring/spring-wlevs.xsd">

  <wlevs:event-type-repository>
    <wlevs:event-type type-name="com.mycompany.httppubsub.PubsubEvent">
      <wlevs:class>com.mycompany.httppubsub.PubsubEvent</wlevs:class>
    </wlevs:event-type>
  </wlevs:event-type-repository>

  <wlevs:adapter id="receiveFromFeed"
                class="com.mycompany.httppubsub.ReceiveFromFeed">
  </wlevs:adapter>

  <wlevs:processor id="pubsubProcessor" />

  <wlevs:adapter id="remotePublisher" provider="httppub"/>

  <wlevs:adapter id="localPublisher" provider="httppub"/>

  <wlevs:stream id="feed2processor">
    <wlevs:source ref="receiveFromFeed"/>
    <wlevs:listener ref="pubsubProcessor"/>
  </wlevs:stream>

  <wlevs:stream id="pubsubStream">
    <wlevs:listener ref="remotePublisher"/>
    <wlevs:listener ref="localPublisher"/>
    <wlevs:source ref="pubsubProcessor"/>
  </wlevs:stream>

</beans>
```

In the preceding example:

- The `receiveFromFeed` adapter is a custom adapter that receives data from some data feed; the details of this adapter are not pertinent to this topic. The `receiveFromFeed` adapter then sends its events to the `pubsubProcessor` via the `feed2processor` stream.
- The `pubsubProcessor` processes the events from the `receiveFromFeed` adapter and then sends them to the `pubsubStream` stream, which in turn sends them to the two built-in pub-sub adapters: `remotePublisher` and `localPublisher`.

- Based on the configuration of these two pub-sub adapters (see examples in [“Configuring the HTTP Pub-Sub Adapters” on page 5-11](#)), `remotePublisher` publishes events only of type `com.mycompany.httppubsub.PubsubEvent` and publishes them to the a channel called `/channel1` on the HTTP pub-sub server hosted remotely at `http://myhost.com:9102/pubsub`.

The `localPublisher` pub-sub adapter publishes all events it receives to the local HTTP pub-sub server, in other words, the one associated with the Oracle CEP server on which the application is running. The local pub-sub server’s path is `/pubsub` and the channel to which the adapter publishes is called `/channel2`.

The following sample EPN file shows an event processing network with one built-in pub-sub adapter for subscribing; see the text after the example for an explanation:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       xmlns:wlevs="http://www.bea.com/ns/wlevs/spring"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd
http://www.bea.com/ns/wlevs/spring
http://www.bea.com/ns/wlevs/spring/spring-wlevs.xsd">

  <wlevs:event-type-repository>
    <wlevs:event-type type-name="com.mycompany.httppubsub.PubsubEvent">
      <wlevs:class>com.mycompany.httppubsub.PubsubEvent</wlevs:class>
    </wlevs:event-type>
  </wlevs:event-type-repository>

  <wlevs:adapter id="remoteSubscriber" provider="httpsub">
    <wlevs:listener ref="myEventBean"/>
  </wlevs:adapter>

  <bean id="myEventBean"
        class="com.mycompany.httppubsub.MyEventBean">
  </bean>

  <wlevs:stream id="pubsubStream" manageable="true">
    <wlevs:listener>
      <bean id="mySink"
            class="com.mycompany.httppubsub.MySink"/>
    </wlevs:listener>
    <wlevs:source ref="myEventBean"/>
  </wlevs:stream>
```

</beans>

In the preceding example:

- The `remoteSubscriber` adapter is a built-in pub-sub adapter for subscribing.

Based on the configuration of this adapter (see examples in [“Configuring the HTTP Pub-Sub Adapters” on page 5-11](#)), `remoteSubscriber` subscribes to a channel called `/channel3` configured for the remote HTTP pub-sub server hosted at `http://myhost.com:9102/pubsub`. Oracle CEP converts each messages it receives from this channel to an instance of `com.mycompany.httppubsub.PubsubEvent` and then sends it a Spring bean called `myEventBean`.

- The `myEventBean` processes the event as described by the `com.mycompany.httppubsub.MyEventBean` class, and then passes it the `mySink` event source via the `pubsubStream` stream. This section does not discuss the details of these components because they are not pertinent to the HTTP pub-sub adapter topic.

# Configuring the Stream Component

This section contains information on the following subjects:

- [“Overview of the Stream Configuration File” on page 6-1](#)
- [“Creating the Stream Configuration File: Main Steps” on page 6-2](#)
- [“Example of an Stream Configuration File” on page 6-4](#)

## Overview of the Stream Configuration File

Your Oracle Complex Event Processing (or *Oracle CEP* for short ) application contains one or more stream components, or *streams* for short. The streams stream data between other types of components, such as between adapters and processors, and between processors and the business logic POJO.

Each stream in your application has a default configuration. In particular:

- The maximum number of events on the stream is 1048.
- There is one thread assigned to the stream.
- Monitoring is enabled.

The default stream configuration is typically adequate for most applications. However, if you want to change this configuration, you must create an XML file that is deployed as part of the Oracle CEP application bundle. You can later update this configuration at runtime using the [wlevs.Admin utility](#) or manipulating the appropriate JMX Mbeans directly.

If your application has more than one stream, you can create separate XML files for each stream, or create a single XML file that contains the configuration for all streams, or even all components of your application (adapters, processors, and streams). Choose the method that best suits your development environment.

# Creating the Stream Configuration File: Main Steps

The following procedure describes the main steps to create the stream configuration file. For simplicity, it is assumed in the procedure that you are going to configure all components of an application in a single XML file

See [XSD Schema Reference for Component Configuration Files](#) for the complete XSD Schema that describes the stream configuration file.

1. Create an XML file using your favorite XML editor. You can name this XML file anything you want, provided it ends with the .xml extension.

The root element of the configuration file is `<config>`, with namespace definitions shown in the next step.

2. For each stream in your application, add a `<stream>` child element of `<config>`. Uniquely identify each stream with the `<name>` child element. This name must be the same as the value of the `id` attribute in the `<wlevs:stream>` tag of the EPN assembly file that defines the event processing network of your application. This is how Oracle CEP knows to which particular stream component in the EPN assembly file this stream configuration applies. See [“Creating the EPN Assembly File” on page 2-11](#) for details.

For example, if your application has two streams, the configuration file might initially look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<helloworld:config
xmlns:helloworld="http://www.bea.com/xml/ns/wlevs/example/helloworld">
  <processor>
    ...
  </processor>

  <stream>
    <name>firstStream</name>
    ...
  </stream>

  <stream>
    <name>secondStream</name>
```

```

    ...
  </stream>
</helloworld:config>

```

In the example, the configuration file includes two streams called `firstStream` and `secondStream`. This means that the EPN assembly file must include at least two stream registrations with the same identifiers:

```

<wlevs:stream id="firstStream" ...>
  ...
</wlevs:stream>
<wlevs:stream id="secondStream" ...>
  ...
</wlevs:stream>

```

**WARNING:** Identifiers and names in XML files are case sensitive, so be sure you specify the same case when referencing the component's identifier in the EPN assembly file.

3. Optionally add a `<max-size>` child element of the `<stream>` element to specify the maximum size of the stream. Zero-size streams synchronously pass-through events. Streams with non-zero size process events asynchronously, buffering events by the requested size. The default value is 1024 .

```

<stream>
  <name>firstStream</name>
  <max-size>10000</size>
</stream>

```

4. Optionally add a `<max-threads>` child element of the `<stream>` element to specify the maximum number of threads that will be used to process events for this stream. Setting this value has no effect when `<max-size>` is 0. The default value is 1.

```

<stream>
  <name>firstStream</name>
  <max-threads>2</size>
</stream>

```

5. Optionally use the `monitoring` Boolean attribute of the `<stream>` element to enable or disable monitoring of the stream; by default monitoring is enabled. When monitoring is enabled, the stream gathers runtime statistics, such as the number of events inbound and outbound on it, and forwards this information to an Mbean:

```

<stream monitoring="true">
  <name>firstStream</name>
  ...
</stream>

```

To truly enable monitoring, you must have also enabled the *manageability* of the stream, otherwise setting the monitoring attribute to `true` has no effect. You enable manageability by setting the `manageable` attribute of the corresponding component registration in the EPN assembly file to `true`, as shown in bold in the following example:

```
<wlevs:stream id="firstStream" manageable="true">
  <wlevs:listener ref="helloworldProcessor"/>
  <wlevs:source ref="helloworldAdapter"/>
</wlevs:stream>
```

## Example of an Stream Configuration File

The following sample XML file shows how to configure two streamss, `firstStream` and `secondStream`.

```
<?xml version="1.0" encoding="UTF-8"?>
<sample:config
  xmlns:sample="http://www.bea.com/xml/ns/wlevs/example/sample">
  <stream>
    <name>firstStream</name>
    <max-size>10</max-size>
  </stream>
  <stream>
    <name>secondStream</name>
    <max-threads>4</max-threads>
  </stream>
</sample:config>
```



# Configuring the Complex Event Processor

This section contains information on the following subjects:

- [“Overview of the Complex Event Processor Configuration File” on page 7-1](#)
- [“Configuring the Complex Event Processor: Main Steps” on page 7-2](#)
- [“Example of a Processor Configuration File” on page 7-5](#)

## Overview of the Complex Event Processor Configuration File

Your Oracle Complex Event Processing (or *Oracle CEP* for short) application contains one or more complex event processors, or *processors* for short. Each processor takes as input events from one or more adapters; these adapters in turn listen to data feeds that send a continuous stream of data from a source. The source could be anything, from a financial data feed to the Oracle CEP load generator. The main feature of a processor is its associated Event Processing Language (EPL) rules that select a subset of the incoming events to then pass on to the component that is listening to the processor. The listening component could be another processor, or the business object POJO that typically defines the end of the event processing network, and thus does something with the events, such as publish them to a client application.

Each processor in your application must have an associated XML file that defines its initial configuration. This XML file is deployed as part of the Oracle CEP application bundle. You can later update this configuration at runtime using the [wlevs.Admin utility](#) or manipulating the appropriate JMX Mbeans directly.

In addition to configuring the initial set of EPL rules of the processor, you can configure the following in the processor XML file:

- JDBC datasources if your Oracle CEP application requires a connection to a relational database.
- Enable monitoring of the processor.

You are required to create a configuration XML file for each processor in your application. If your application has more than one processor, you can create separate XML files for each processor, or create a single XML file that contains the configuration for all processors. Choose the method that best suits your development environment.

You can optionally create configuration files for the other components in your application (adapters and streams), although if their default configuration is adequate you do not need to change it. If you do create configuration files for these components, you can create separate files or combine them with the processor configuration file(s).

## Configuring the Complex Event Processor: Main Steps

This section describes the main steps to create the processor configuration file. For simplicity, it is assumed in the procedure that you are going to configure all processors in a single XML file, although you can also create separate files for each processor.

See [“Example of a Processor Configuration File” on page 7-5](#) for a complete example of a processor configuration file.

See [XSD Schema Reference for Component Configuration Files](#) for the complete XSD Schema that describes the processor configuration file.

1. Design the set of EPL rules that the processor executes. These rules can be as simple as selecting *all* incoming events to restricting the set based on time, property values, and so on, as shown in the following two examples:

```
SELECT * from Withdrawal RETAIN ALL

SELECT symbol, AVG(price)
FROM (SELECT * FROM MarketTrade WHERE blockSize > 10)
RETAIN 100 EVENTS PARTITION BY symbol WITH LARGEST price
GROUP BY symbol
HAVING AVG(price) >= 100
ORDER BY symbol
```

EPL is similar in many ways to Structure Query Language (SQL), the language used to query relational database tables, although the syntax between the two differs in many ways.

The other big difference is that EPL queries take another dimension into account (time), and the processor executes the EPL continually, rather than SQL queries that are static.

For additional conceptual information about EPL, and examples and reference information to help you design and write your own EPL rules, see the [EPL Reference Guide](#).

2. Create the processor configuration XML file that will contain the EPL rules you designed in the preceding step, as well as other optional features, for each processor in your application.

You can name this XML file anything you want, provided it ends with the `.xml` extension.

The root element of the processor configuration file is `<config>`, with namespace definitions shown in the next step.

3. For each processor in your application, add a `<processor>` child element of `<config>`. Uniquely identify each processor with the `<name>` child element. This name must be the same as the value of the `id` attribute in the `<wlevs:processor>` tag of the EPN assembly file that defines the event processing network of your application. This is how Oracle CEP knows to which particular processor component in the EPN assembly file this processor configuration applies. See “[Creating the EPN Assembly File](#)” on page 2-11 for details.

For example, if your application has two processors, the configuration file might initially look like:

```
<?xml version="1.0" encoding="UTF-8"?>

<n1:config
xsi:schemaLocation="http://www.bea.com/xml/ns/wlevs/config/application
wlevs_application_config.xsd"
xmlns:n1="http://www.bea.com/xml/ns/wlevs/config/application"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <processor>
        <name>firstProcessor</name>
        ...
    </processor>

    <processor>
        <name>secondProcessor</name>
        ...
    </processor>

</n1:config>
```

In the example, the configuration file includes two processors called `myFirstProcessor` and `mySecondProcessor`. This means that the EPN assembly file must include at least two processor registrations with the same identifiers:

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

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

**WARNING:** Identifiers and names in XML files are case sensitive, so be sure you specify the same case when referencing the component's identifier in the EPN assembly file.

4. Add a `<rules>` child element to each `<processor>` to group together one or more `<rule>` elements that correspond to the set of EPL rules you have designed for this processor.

Use the required `id` attribute of the `<rule>` element to uniquely identify each rule. Use the XML CDATA type to input the actual EPL rule. For example:

```
<processor>
  <name>firstProcessor</name>
  <rules>
    <rule id="myFirstRule"><![CDATA[
      SELECT * from Withdrawal RETAIN ALL
    ]]></rule>

    <rule id="mySecondRule"><![CDATA[
      SELECT * from Checking RETAIN ALL
    ]]></rule>
  </rules>
</processor>
```

5. Optionally add a `<database>` child element of the `<processor>` element to define a JDBC data source for your application. This is required if your EPL rules joining a stream of events with an actual relational database table.

Use the `<name>` child element of `<database>` to uniquely identify the datasource.

Use the `<data-source-name>` child element of `<database>` to specify the actual name of the data source; this name corresponds to the `<name>` child element of the `<data-source>` configuration object in the `config.xml` file of your domain. For details about configuring the server, see [Configuring Access to a Relational Database](#).

For example:

```
<processor>
  <name>firstProcessor</name>
  <rules>
    ....
  </rules>
```

```

    <database>
      <name>myDataSource</name>
      <data-source-name>rdbmsDataSource</data-source-name>
    </database>
  </processor>

```

6. Optionally use the `monitoring` Boolean attribute of the `<processor>` element to enable or disable monitoring of the processor; by default monitoring is enabled. When monitoring is enabled, the processor gathers runtime statistics, such as the number of events inbound and outbound on it, and forwards this information to an Mbean:

```

<processor monitoring="true">
  <name>firstProcessor</name>
  <rules>
    ....
  </rules>
</processor>

```

To truly enable monitoring, you must have also enabled the *manageability* of the processor, otherwise setting the `monitoring` attribute to `true` has no effect. You enable manageability by setting the `manageable` attribute of the corresponding component registration in the EPN assembly file to `true`, as shown in bold in the following example:

```

<wlevs:processor id="firstProcessor" manageable="true" />

```

## Example of a Processor Configuration File

The following example shows how to configure one of the sample EPL queries shown in [“Configuring the Complex Event Processor: Main Steps” on page 7-2](#) for the `myProcessor` processor:

```

<?xml version="1.0" encoding="UTF-8"?>

<nl:config
  xsi:schemaLocation="http://www.bea.com/xml/ns/wlevs/config/application
wlevs_application_config.xsd"
  xmlns:nl="http://www.bea.com/xml/ns/wlevs/config/application"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <processor>

    <name>myProcessor</name>

    <rules>

      <rule id="myRule"><![CDATA[

        SELECT symbol, AVG(price)

        FROM (SELECT * FROM MarketTrade WHERE blockSize > 10)

```

## Configuring the Complex Event Processor

```
    RETAIN 100 EVENTS PARTITION BY symbol WITH LARGEST price
    GROUP BY symbol
    HAVING AVG(price) >= 100
    ORDER BY symbol

  ]]></rule>
</rules>
</processor>
</nl:config>
```

In the example, the `<name>` element specifies that the processor for which the single EPL rule is being configured is called `myProcessor`. This in turn implies that the EPN assembly file that defines your application must include a corresponding `<wlevs:processor id="myProcessor" />` tag to link this EPL rules with an actual `myProcessor` processor instance.

# Using Oracle CEP Caching

This section contains information on the following subjects:

- [“Overview of Oracle Complex Event Processing Caching” on page 8-1](#)
- [“Typical Steps to Use Oracle CEP Caching” on page 8-4](#)
- [“Referencing a Cache from an EPL Statement” on page 8-19](#)
- [“Coherence Support” on page 8-21](#)

## Overview of Oracle Complex Event Processing Caching

Oracle Complex Event Processing (or *Oracle CEP* for short) applications can optionally publish or consume events to and from a cache to increase the availability of the events and increase the performance of their applications. A cache is a temporary storage area for events, created exclusively to improve the overall performance of your application; it is not necessary for the application to function correctly.

Oracle CEP includes its own caching implementation for local single-JVM caches; this implementation uses an in-memory store. If you require a different kind of cache, such as one that provides replicated or partitioned caching, then you can plug-in different cache providers, such as Oracle Coherence. See [“Coherence Support” on page 8-21](#) for details about using Oracle Coherence.

In this document, a *caching system* refers to a configured instance of a caching implementation, either the local one included in the product or a third-party one. A caching system defines a

named set of configured *caches* as well as the configuration for remote communication if any of the caches are distributed across multiple machines.

A caching system is always configured at the application level, regardless of its implementation (Oracle CEP or third-party). Other Oracle CEP applications in separate bundles can also use the caching system.

A cache can be considered a stage in the event processing network in which an external element (the cache) consumes or produces events; this is similar to an adapter that uses a JMS destination. You configure a caching system, along with one or more caches, for an application declaratively using the EPN assembly file. A cache, however, does not have to be an actual stage in the network; another component or Spring bean can access a cache programmatically using the caching APIs.

The next sections describe specific use cases of the Oracle CEP caching feature.

## Use Case: Publishing Events to a Cache

An example of this use case is a financial application that publishes events to a cache while the financial market is open and then processes data in the cache after the market closes.

Publishing events to a cache makes them highly available or available to other Oracle CEP applications running in the server. Publishing events to a cache also allows for asynchronous writes to a secondary storage by the cache implementation. You can configure any stage in an Oracle CEP application that generates events (input adapter, stream, business POJO, or processor) to publish its events to the cache.

## Use Case: Consuming Data From a Cache

Oracle CEP applications may sometimes need to access non-streaming data in order to do its work; caching this data can increase the performance of the application.

The standard components of an Oracle CEP application that are allowed direct programming access to a cache are input- and output-adapters and business POJOs.

Additionally, applications can access a cache from EPL, either by a user-defined EPL function or directly from an EPL statement. In the case of a user-defined EPL function, programmers use Spring to inject the cache resource into the implementation of the function. Applications can also query a cache directly from an EPL statement that runs in a processor. In this case, the cache essentially functions as another type of stream data source to a processor so that querying a cache is very similar to querying a stream except that data is pulled from a cache.



An example of using EPL to query a cache is from a financial application that publishes orders and the trades used to execute the orders to a cache. At the end of the day when the markets are closed, the application queries the cache in order to find all the trades related to a particular order.

## Use Case: Updating and Deleting Data in a Cache

An Oracle CEP application can both update and delete data in a cache when required.

For example, a financial application may need to update an order in the cache each time individual trades that fulfill the order are executed, or an order may need to be deleted if it has been cancelled. The components of an application that are allowed to consume data from a cache are also allowed to update it.

## Additional Caching Features

In addition to the major caching features described in the preceding sections, Oracle CEP caching includes the following features:

- Pre-load a cache with data before an application is deployed.
- Periodically refresh, invalidate, and flush the data in a cache. All these tasks happen incrementally and without halting the application or causing latency spikes.
- Dynamically update a cache's configuration.
- Support for large local cache sizes (4 - 16 GB).

## Caching APIs

Oracle CEP provides a number of [caching APIs](#) that you can use in your application to perform certain tasks. The APIs are in two packages:

- `com.bea.cache.jcache`—Includes the APIs used to access a cache and create cache loader, listeners, and stores.
- `com.bea.wlevs.cache.spi`—Includes the API used to access a caching system.

The creation, configuration, and wiring of the caching systems and caches is all done using the EPN assembly file and component configuration files. This means that you typically never explicitly use the `Cache` and `CachingSystem` interfaces in your application; the only reason to use them is if you have additional requirements than the standard configuration. For example, if you want to provide integration with a third-party cache provider, then you must use the

CachingSystem interface. If you want to perform operations on a cache that are not part of the `java.util.Map` interface, then you can use the `Cache` interface.

If you create cache listeners, loaders, or stores, then the Spring bean must implement the `CacheListener`, `CacheLoader`, or `CacheStore` interfaces. The following sections describe additional details.

## Typical Steps to Use Oracle CEP Caching

Depending on how you are going to use the cache, there are different tasks that you must perform, as described in the following procedure that in turn point to sections with additional details.

**Note:** It is assumed in this section that you have already created an Oracle CEP application along with its EPN assembly file and that you want to update the application to use caching. If you have not, refer to [“Overview of Creating Oracle Complex Event Processing Applications” on page 2-1](#) for details.

1. Configure the caching system and one or more caches by updating the caching configuration file for the application.

See [“Configuring the Oracle CEP Caching System and Caches” on page 8-5](#).

2. Declare the caching system in the EPN assembly file.

See [“Declaring the Caching System in the EPN Assembly File” on page 8-9](#).

3. Declare the caches in the EPN assembly file.

[“Declaring a Cache in the EPN Assembly File” on page 8-10](#)

4. Optionally configure and program an adapter, business POJO, or EPL user-defined function to access the cache. The configuration is done in the EPN assembly file and the programming is done in the Java file that implements the POJO, adapter, or EPL user-defined function. See:

- [“Configuring and Programming a Business POJO to Access a Cache” on page 8-11](#)
- [“Configuring and Programming an Adapter to Access a Cache” on page 8-12](#)
- [“Configuring and Programming a User-Defined EPL Function to Access a Cache” on page 8-13](#)

5. Optionally specify that a cache is an event sink by configuring it as a listener to another component in the event processing network.

See [“Configuring a Cache as a Listener” on page 8-15](#).

6. Optionally specify that a cache is an event source to which another component in the event processing network listens.  
See [“Configuring a Cache as an Event Source” on page 8-18](#).
7. Optionally configure a cache loader or cache store for a cache. See:
  - [“Configuring a Cache Loader” on page 8-18](#)
  - [“Configuring a Cache Store” on page 8-19](#)
8. Optionally reference the cache in an EPL statement. See [“Referencing a Cache from an EPL Statement” on page 8-19](#).

## Configuring the Oracle CEP Caching System and Caches

You configure a caching system and its caches in the caching configuration file, similar to how you configure other components in the event processing network such as processors and adapters. For general information about these configuration files, see [“Component Configuration Files” on page 2-3](#).

The caching implementation you use determines the number of caching systems you can configure per Oracle CEP instance. For example, the third-party caching implementation Gemfire specifies that you can configure only one caching system per Oracle CEP instance. If, however, you use the caching implementation provided by Oracle CEP, then you can configure multiple caching systems per application.

The following procedure describes the main steps to configure the caching system provided by Oracle CEP for your application. For simplicity, it is assumed in the procedure that you are going to configure all components of an application, including the caching system, in a single configuration XML file and that you have already created this file for your application.

See [XSD Schema Reference for Component Configuration Files](#) for the complete XSD Schema that describes the caching system configuration file elements.

1. Open the configuration XML file using your favorite XML editor.
2. Add a `<caching-system>` child element of the `<config>` root element; use the `<name>` child element to uniquely identify it. This name value will optionally be used later as the `id` attribute of the `<wlevs:caching-system>` tag in the EPN assembly file that defines the event processing network of your application. This is how Oracle CEP knows to which particular caching system in the EPN assembly file this caching configuration applies. See [“Declaring the Caching System in the EPN Assembly File” on page 8-9](#) for details.

For example, assume your configuration file already contains a processor and an adapter (contents removed for simplicity); then the updated file might look like the following

```
<?xml version="1.0" encoding="UTF-8"?>
<nl:config xmlns:nl="http://www.bea.com/ns/wlevs/config/application"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <processor>
    ...
  </processor>

  <adapter>
    ...
  </adapter>

  < caching-system>
    < name> caching-system-id </ name>
  </ caching-system>
</nl:config>
```

3. For each cache you want to create, add a `<cache>` child element of the `<caching-system>` element; use the `<name>` child element to uniquely identify it. This name value will be used later as the `id` attribute of the `<wlevs:cache>` tag in the EPN assembly file that defines the event processing network of your application. This is how Oracle CEP knows to which particular cache in the EPN assembly file this configuration applies. The following example shows two caches in the caching system:

```
<caching-system>
  <name>caching-system-id</name>
  <cache>
    <name>cache-id</name>
    ...
  </cache>
  <cache>
    <name>second-cache-id</name>
    ...
  </cache>
</caching-system>
```

4. For each cache, optionally add the following elements that take simple data types to configure the cache:
  - `<max-size>` : The number of cache elements in memory after which eviction/paging occurs. The maximum cache size is  $2^{31}-1$  entries; default is 64.
  - `<eviction-policy>` : The eviction policy to use when `max-size` is reached. Supported values are: FIFO, LRU, LFU, and NRU; default value is LFU.

- `<time-to-live>`: The maximum amount of time, in milliseconds, that an entry is cached. Default value is infinite.
- `<idle-time>`: Amount of time, in milliseconds, after which cached entries are actively removed from the cache. Default value is infinite.
- `<work-manager-name>`: The work manager to be used for all asynchronous operations. The value of this element corresponds to the `<name>` child element of the `<work-manager>` element in the server's `config.xml` configuration file.

For example:

```
< caching-system>
  < name> caching-system-id</ name>

  < cache>
    < name> cache-id</ name>
    < max-size> 100000</ max-size>
    < eviction-policy> LRU</ eviction-policy>
    < time-to-live> 3600</ time-to-live>
  </ cache>
</ caching-system>
```

5. Optionally add *either* `<write-through>` or `<write-behind>` as a child element of `<cache>` to specify synchronous or asynchronous writes to the cache store, respectively. By default, writes to the store are synchronous (`<write-through>`) which means that as soon as an entry is created or updated the write occurs.

If you specify the `<write-behind>` element, then the cache store is invoked from a separate thread after a create or update of a cache entry. Use the following optional child elements to further configure the asynchronous writes to the store:

- `<work-manager-name>`: The work manager that handles asynchronous writes to the cache store. If a work manager is specified for the cache itself, this value overrides it for store operations only. The value of this element corresponds to the `<name>` child element of the `<work-manager>` element in the server's `config.xml` configuration file.
- `<batch-size>`: The number of updates that are picked up from the store buffer to write back to the backing store. Default value is 1.
- `<buffer-size>`: The size of the internal store buffer that temporarily holds the asynchronous updates that need to be written to the store. Default value is 100.
- `<buffer-write-attempts>`: The number of attempts that the user thread makes to write to the store buffer. The user thread is the thread that creates or updates a cache

entry. If all attempts by the user thread to write to the store buffer fail, it will invoke the store synchronously. Default value is 1.

- `<buffer-write-timeout>` : The time in milliseconds that the user thread waits before aborting an attempt to write to the store buffer. The attempt to write to the store buffer fails only in case the buffer is full. After the timeout, further attempts may be made to write to the buffer based on the value of `buffer-write-attempts`. Default value is 100.

For example:

```
< caching-system>
  < name> caching-system-id </ name>

  < cache>
    < name> cache-id </ name>
    < max-size> 100000 </ max-size>
    < eviction-policy> LRU </ eviction-policy>
    < time-to-live> 3600 </ time-to-live>
    < write-behind>
      < buffer-size> 200 </ buffer-size>
      < buffer-write-attempts> 2 </ buffer-write-attempts>
      < buffer-write-timeout> 200 </ buffer-write-timeout>
    </ write-behind>
  </ cache>
</ caching-system>
```

6. Optionally add a `<listeners>` child element of `<cache>` to configure the behavior of components that listen to the cache.

Use the `asynchronous` Boolean attribute to specify whether listeners should be invoked asynchronously. By default this attribute is `false`, which means listeners are invoked synchronously.

The `<listeners>` element has a single child element, `<work-manager-name>`, that specifies the work manager to be used for asynchronously invoking listeners. This value is ignored if synchronous invocations are enabled. If a work manager is specified for the cache itself, this value overrides it for invoking listeners only. The value of this element corresponds to the `<name>` child element of the `<work-manager>` element in the server's `config.xml` configuration file.

For example:

```
< caching-system>
  < name> caching-system-id </ name>

  < cache>
    < name> cache-id </ name>
```

```

    <max-size>100000</max-size>
    <eviction-policy>LRU</eviction-policy>
    <time-to-live>3600</time-to-live>
    <write-behind>
      <buffer-size>200</buffer-size>
      <buffer-write-attempts>2</buffer-write-attempts>
      <buffer-write-timeout>200</buffer-write-timeout>
    </write-behind>
    <listeners asynchronous="true">
      <work-manager-name>cachingWM</work-manager-name>
    </listeners>
  </cache>
</caching-system>

```

## Declaring the Caching System in the EPN Assembly File

To declare a caching system that uses the Oracle CEP implementation declaratively in the EPN assembly file, use the `<wlevs:caching-system>` tag without any additional attributes, as shown in the following example:

```
<wlevs:caching-system id="caching-system-id"/>
```

The value of the `id` attribute must match the name specified for the caching system in the external configuration metadata. If the application allows other applications to use the caching system, it can specify that the caching system be advertised using the `advertise` attribute (by default set to `false`):

```
<wlevs:caching-system id="caching-system-id" advertise="true"/>
```

You also use the `<wlevs:caching-system>` tag to declare a third-party implementation; use the `class` or `provider` attributes to specify additional information.

**Note:** See [“Coherence Support” on page 8-21](#) for specific instructions on using Oracle Coherence as the caching provider.

For simplicity, you can include the third-party implementation code inside the Oracle CEP application bundle itself to avoid having to import or export packages and managing the lifecycle of a separate bundle that contains the third-party implementation. In this case the `<wlevs:caching-system>` tag appears in the EPN assembly file as shown in the following example:

```

<wlevs:caching-system id="caching-system-id"
  class="third-party-implementation-class"/>

```

The `class` attribute specifies a Java class that must implement the `com.bea.wlevs.cache.spi.CachingSystem` interface. For details about this interface, see the [Oracle CEP Javadocs](#).

Sometimes, however, you might not be able, or want, to include the third-party caching implementation in the same bundle as the Oracle CEP application that is using it. In this case, you must create a *separate* bundle whose Spring application context includes the `<wlevs:caching-system>` tag, with the `advertise` attribute mandatory:

```
<wlevs:caching-system id="caching-system-id"
                      class="third-party-implementation-class"
                      advertise="true" />
```

Alternatively, if you want to decouple the implementation bundle from the bundle that references it, or you are plugging in a caching implementation that supports multiple caching systems per Java process, you can specify a factory as a provider:

```
<wlevs:caching-system id="caching-system-id"
                      provider="caching-provider" />

<factory id="factory-id" provider-name="caching-provider">
  <class>the.factory.class.name</class>
</factory>
```

The factory class (`the.factory.class.name` in the example) must implement the `com.bea.wlevs.cache.spi.CachingSystemFactory` interface. This interface has a single method, `create()`, that returns a `com.bea.wlevs.cache.spi.CachingSystem` instance.

You must deploy this bundle alongside the application bundle so that the latter can start using it.

## Declaring a Cache in the EPN Assembly File

After you have declared a caching system for an application, you can configure one or more caches using the `<wlevs:cache>` tag:

```
<wlevs:caching-system id="caching-system-id" />
...
<wlevs:cache id="cache-id" name="alternative-cache-name">
  <wlevs:caching-system ref="caching-system-id" />
</wlevs:cache>
```

The `name` attribute is optional; specify it only if the name of the cache in the caching system is different from its ID. The `<wlevs:caching-system>` child element references the



already-declared caching system that contains the cache. You must specify this child element only if the caching system is ambiguous: there is more than one caching system declared (either implicitly or explicitly) or if the caching system is in a different application or bundle.

You can export both the caching system and the cache as an OSGI service using the `advertise` attribute.

```
<wlevs:caching-system id="caching-system-id" advertise="true"/>
...
<wlevs:cache id="cache-id" name="alternative-cache-name"
advertise="true" >
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
```

If the cache is advertised, then a component in the EPN of an application in a *separate* bundle can then reference it. The following example shows how a processor in one bundle can use as a cache source the cache with ID `cache-id` located in a separate bundle (called `cacheprovider`):

```
<wlevs:processor id="myProcessor2">
    <wlevs:cache-source ref="cacheprovider:cache-id"/>
</wlevs:processor>
```

The caching system is responsible for creating the cache associated with a particular name and returning a reference to the cache. The resulting cache bean implements the `java.util.Map` interface.

## Configuring and Programming a Business POJO to Access a Cache

A business POJO, configured as a standard Spring bean in the EPN assembly file, can be injected with a cache using the standard Spring mechanism for referencing another bean. In this way the POJO can view and manipulate the cache. A cache bean implements the `java.util.Map` interface which is what the business POJO uses to access the injected cache. A cache bean can also implement a vendor-specific sub-interface of `java.util.Map`, but for portability it is recommended that you implement `Map`.

First, the configuration of the business POJO in the EPN assembly file must be updated with a `<property>` child element, as shown in the following example based on the Output bean of the FX example (see [Helloworld Example](#)):

```
<wlevs:caching-system id="caching-system-id"/>
...
```

```
<wlevs:cache id="cache-id" name="alternative-cache-name">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
...
<bean class="com.bea.wlevs.example.helloworld.HelloWorldBean">
    <property name="map" ref="cache-id"/>
</bean>
```

In the example, the `ref` attribute of `<property>` references the `id` value of the `<wlevs:cache>` tag. Oracle CEP automatically injects the cache, implemented as a `java.util.Map`, into the business POJO bean.

In the business POJO bean Java source, add a `setMap (Map)` method with the code that implements whatever you want the POJO to do with the cache:

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

...

import java.util.Map;

public class HelloWorldBean implements EventSink {

    ...

    public void setMap (Map map) {...}

}
```

## Configuring and Programming an Adapter to Access a Cache

An adapter can also be injected with a cache using the standard Spring mechanism for referencing another bean. A cache bean implements the `java.util.Map` interface which is what the adapter uses to access the injected cache.

First, the configuration of the adapter in the EPN assembly file must be updated with a `<wlevs:instance-property>` child element, as shown in the following example:

```
<wlevs:caching-system id="caching-system-id"/>
...
<wlevs:cache id="cache-id" name="alternative-cache-name">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
```

```

...
<wlevs:adapter id="myAdapter" provider="myProvider">
    <wlevs:instance-property name="map" ref="cache-id"/>
</wlevs:adapter>

```

In the example, the `ref` attribute of `<wlevs:instance-property>` references the `id` value of the `<wlevs:cache>` tag. Oracle CEP automatically injects the cache, implemented as a `java.util.Map`, into the adapter.

In the adapter Java source, add a `setMap (Map)` method with the code that implements whatever you want the adapter to do with the cache:

```

package com.bea.wlevs.example;

...

import java.util.Map;

public class MyAdapter implements Runnable, Adapter, EventSource,
SuspendableBean {

    ...

    public void setMap (Map map) {...}

}

```

## Configuring and Programming a User-Defined EPL Function to Access a Cache

In addition to standard event streams, EPL rules can also invoke the member methods of a user-defined function.

These user-defined functions are implemented as standard Java classes and are declared in the EPN assembly file using the standard Spring bean tags, as shown in the following example:

```

<bean id="orderFunction" class="orderFunction-impl-class"/>

```

The processor in which the relevant EPL rule runs must then be injected with the user-defined function using the `<wlevs:function>` child element, referencing the Spring with the `ref` attribute:

```

<wlevs:processor id= "tradeProcessor">
    <wlevs:function ref="orderFunction"/>
</wlevs:processor>

```

The following EPL rule, assumed to be configured for the `tradeProcessor` processor, shows how to invoke the `existsOrder()` method of the `orderFunction` user-defined function:

```
INSERT INTO InstitutionalOrder
SELECT er.orderKey AS key, er.symbol AS symbol, er.shares as
cumulativeShares
FROM ExecutionRequest er RETAIN 8 HOURS WITH UNIQUE KEY
WHERE NOT orderFunction.existsOrder(er.orderKey)
```

You can also configure the user-defined function to access a cache by injecting the function with a cache using the standard Spring mechanism for referencing another bean. A cache bean implements the `java.util.Map` interface which is what the user-defined function uses to access the injected cache.

First, the configuration of the user-defined function in the EPN assembly file must be updated with a `<wlevs:property>` child element, as shown in the following example:

```
<wlevs:caching-system id="caching-system-id"/>
...
<wlevs:cache id="cache-id" name="alternative-cache-name">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
...
<bean id="orderFunction" class="orderFunction-impl-class">
    <wlevs:property name="cache" ref="cache-id"/>
</bean>
```

In the example, the `ref` attribute of `<wlevs:property>` references the `id` value of the `<wlevs:cache>` tag. Oracle CEP automatically injects the cache, implemented as a `java.util.Map`, into the user-defined function.

In the user-defined function's Java source, add a `setMap (Map)` method with the code that implements whatever you want the function to do with the cache:

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

...

import java.util.Map;

public class OrderFunction {
    ...

    public void setMap (Map map) {...}
```

```
}
```

## Configuring a Cache as a Listener

A cache can be configured as an explicit listener in the event processing network in order to receive events.

For example, to specify that a cache listens to a stream, specify the `<wlevs:listener>` tag with a reference to the cache as a child of the `<wlevs:stream>` tag as shown below:

```
<wlevs:caching-system id="caching-system-id"/>
...
<wlevs:cache id="cache-id" name="alternative-cache-name">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
...
<wlevs:stream id="tradeStream">
    <wlevs:listener ref="cache-id"/>
</wlevs:stream>
```

As the stream sends new events to the cache, they are inserted into the cache. If a *remove event* (an old event that exits the output window) is sent by the stream, then the event is removed from the cache.

## Specifying the Key Used to Index a Cache

When you configure a cache to be a listener, events are inserted into the cache. This section describes the variety of options available to you to specify the key used to index a cache in this instance.

If you do not explicitly specify a key, the event object itself serves as both the key and value when the event is inserted into the cache. In this case, the event class must include a valid implementation of the `equals` and `hashCode` methods that take into account the values of the key properties.

See the following for ways to explicitly specify a key:

- [“Specifying a Key Property in EPN Assembly File” on page 8-16](#)
- [“Using a Metadata Annotation to Specify a Key” on page 8-17](#)
- [“Specifying a Composite Key” on page 8-17](#)

## Specifying a Key Property in EPN Assembly File

The first option is to specify a property name for the key property when a cache is declared in the EPN assembly file using the `key-properties` attribute, as shown in the following example:

```
<wlevs:cache id="cache-id" key-properties="key-property-name">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
```

In this case, all events that are inserted into the cache are required to have a property of this name at runtime, otherwise Oracle CEP throws an exception.

For example, assume the event type being inserted into the cache looks something like the following; note the key property (only relevant Java source shown):

```
public class MyEvent {
    private String key;

    public MyEvent() {
    }

    public MyEvent(String key) {
        this.key = key;
    }

    public String getKey() {
        return key;
    }

    public void setKey(String key) {
        this.key = key;
    }

    ...
}
```

The corresponding declaration in the EPN assembly file would look like the following:

```
<wlevs:cache id="cache-id" key-properties="key">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
```

## Using a Metadata Annotation to Specify a Key

The second option is to use the metadata annotation `com.bea.wlevs.ede.api.Key` to annotate the event property in the Java class that implements the event type. This annotation does not have any attributes.

The following example shows how to specify that the `key` property of the `MyEvent` event type is the key; only relevant code is shown:

```
import com.bea.wlevs.ede.api.Key;

public class MyEvent {
    @Key
    private String key;

    public MyEvent() {
    }

    public MyEvent(String key) {
        this.key = key;
    }

    public String getKey() {
        return key;
    }

    public void setKey(String key) {
        this.key = key;
    }

    ...
}
```

## Specifying a Composite Key

The final option is to use the `key-class` attribute of the `<wlevs:cache>` tag to specify a composite key in which multiple properties form the key. The value of the `key-class` attribute must be a JavaBean whose public fields match the fields of the event class. The matching is done according to the field name. For example:

```
<wlevs:cache id="cache-id" key-class="key-class-name">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
```

## Configuring a Cache as an Event Source

A cache can be configured as a source of events to which another component in the event processing network listens. The listening component can be an adapter or a standard Spring bean. Any component that listens to a cache must implement the `com.bea.cache.jcache.CacheListener` interface. The following example shows how to configure a cache to be an event source for a Spring bean:

```
<wlevs:caching-system id="caching-system-id" />
...
<wlevs:cache id="cache-id" name="alternative-cache-name">
    <wlevs:caching-system ref="caching-system-id" />
    <wlevs:cache-listener ref="cache-listener-id" />
</wlevs:cache>
...
<bean id="cache-listener-id" class="wlevs.example.MyCacheListener"/>
```

In the example, the `cache-listener-id` Spring bean listens to events coming from the cache; the class that implements this component, `wlevs.example.MyCacheListener`, must implement the `com.bea.cache.jcache.CacheListener` interface. You must program the `wlevs.example.MyCacheListener` class yourself.

## Configuring a Cache Loader

A cache loader is an object that loads objects into a cache. You configure a cache loader by using the `<wlevs:cache-loader>` child tag of the `<wlevs:cache>` tag to specify the bean that does the loading work, as shown in the following example:

```
<wlevs:caching-system id="caching-system-id" />
...
<wlevs:cache id="cache-id" name="alternative-cache-name">
    <wlevs:caching-system ref="caching-system-id" />
    <wlevs:cache-loader ref="cache-loader-id" />
</wlevs:cache>
...
<bean id="cache-loader-id" class="wlevs.example.MyCacheLoader"/>
```

In the example, the `cache-loader-id` Spring bean does the cache loading work; it is referenced by the cache using the `ref` attribute of the `<wlevs:cache-loader>` child element.



You must program the `wlevs.example.MyCacheLoader` class yourself, and it must implement the `com.bea.cache.jcache.CacheLoader` interface. This interface includes the `load()` method to customize the loading of a single object into the cache; Oracle CEP calls this method when the requested object is not in the cache. The interface also includes `loadAll()` methods that you implement to customize the loading of the entire cache.

## Configuring a Cache Store

You can configure a cache with a custom store that is responsible for writing data from the cache to a backing store, such as a table in a database. You configure a cache store by using the `<wlevs:cache-store>` child tag of the `<wlevs:cache>` tag to specify the bean that does the actual storing work, as shown in the following example:

```
<wlevs:caching-system id="caching-system-id"/>
...
<wlevs:cache id="cache-id" name="alternative-cache-name">
    <wlevs:caching-system ref="caching-system-id"/>
    <wlevs:cache-store ref="cache-store-id" />
</wlevs:cache>
...
<bean id="cache-store-id" class="wlevs.example.MyCacheStore"/>
```

In the example, the `cache-store-id` Spring bean does the work of writing the data from the cache to the backing store; it is referenced by the cache using the `ref` attribute of the `<wlevs:cache-store>` child element.

You must program the `wlevs.example.MyCacheStore` class yourself, and it must implement the `com.bea.cache.jcache.CacheStore` interface. This interface includes the `store()` method that stores the data in the backing store using the passed key; Oracle CEP calls this method it inserts data into the cache. The interface also includes the `storeAll()` method for storing a batch of data to a backing store in the case that you have configured asynchronous writes for a cache with the `<write-behind>` configuration element.

## Referencing a Cache from an EPL Statement

You can reference a cache from an EPL statement in much the same way you reference a stream; this feature enables you to enrich standard streaming data with data from a separate source. For example, the following EPL query joins trade events from a standard stream with company data from a cache:

```
INSERT INTO EnrichedTradeEvent
SELECT trade.symbol, trade.price, trade.numberOfShares, company.name
FROM TradeEvent trade RETAIN 8 hours, Company company
WHERE trade.symbol = company.id
```

In the example, both `TradeEvent` and `Company` are event types registered in the repository, but they have been configured in such a way that `TradeEvents` come from a standard stream of events but `Company` maps to a cache in the event processing network. This configuration happens outside of the EPL query, which means that the source of the data is transparent in the query itself.

When you use data from a cache in an EPL query, Oracle CEP *pulls* the data rather than it being *pushed*, as is the case with a stream. This means that, continuing with the preceding sample, the query executes only when a stream pushes a trade event to the query; the company data in the cache never causes a query to execute, it is only pulled by the query when needed.

## Restrictions When Using a Cache in an EPL Statement

You must abide by these restrictions when using a cache in an EPL query:

- You must specify the key properties for data in the cache. See [“Specifying the Key Used to Index a Cache” on page 8-15](#) for instructions on specifying the cache key.
- Joins must be executed only by referencing the cache key.
- You cannot specify a `RETAIN` clause for data pulled from a cache. If an event type that gets its data from a cache is included in a `RETAIN` clause, Oracle CEP ignores it.
- You cannot use a cache in a correlated sub-query. Instead, use a join.
- Only a single stream source may occur in the `FROM` clause of an EPL statement that joins cache data source(s). Using multiple cache sources and parameterized SQL queries is supported.

## Typical Steps To Reference a Cache in an EPL Statement

The following procedure assumes you have already configured the caching system and caches; see [“Typical Steps to Use Oracle CEP Caching” on page 8-4](#) for general details.

1. If you have not already done so, create the event type that corresponds to the cache data, such as `Company` in the preceding example, and registered it in the event repository. See [“Creating the Event Types” on page 2-15](#).

2. Specify the key properties for the data in the cache. There are a variety of ways to do this; see [“Specifying the Key Used to Index a Cache” on page 8-15](#) for details.
3. In the EPN assembly file, update the configuration of the cache in the EPN assembly file to declare the event type of its values; use the `value-type` attribute of `<wlevs:cache>`. For example:

```
<wlevs:caching-system id="caching-system-id"/>
...
<wlevs:cache id="cache-id"
    name="alternative-cache-name"
    value-type="Company">
    <wlevs:caching-system ref="caching-system-id"/>
</wlevs:cache>
```

4. In the EPN assembly file, update the configuration of the processor that executes the EPL query that references a cache, adding a `<wlevs:cache-source>` child element that references the cache. For example:

```
<wlevs:stream id="stream-id"/>
<wlevs:processor id="processor-id">
    <wlevs:cache-source ref="cache-id">
    <wlevs:source ref="stream-id">
</wlevs:processor>
```

In the example, the processor will have data pushed to it from the `stream-id` stream as usual; however, the EPL queries that execute in the processor can also pull data from the `cache-id` cache. When the query processor matches an event type in the `FROM` clause to an event type supplied by a cache, such as `Company`, the processor pulls instances of that event type from the cache.

## Coherence Support

This section describes how you can use Oracle Coherence as the caching provider. The steps are similar to the steps to use the local in-memory Oracle CEP cache; for this reason, be sure to read the following sections first:

- [“Overview of Oracle Complex Event Processing Caching” on page 8-1](#)
- [“Typical Steps to Use Oracle CEP Caching” on page 8-4.](#)

The differences are in the way you declare the Coherence cache provider in the EPN assembly file of your application and how you pass the Coherence configuration to Oracle CEP.

Additionally, when you create Spring beans that act as cache listeners, cache loaders, and cache stores, you must use the Coherence-specific Java interfaces rather than the ones described in the preceding sections, such as [“Configuring a Cache Loader” on page 8-18](#) and [“Configuring a Cache Store” on page 8-19](#).

## Declaring the Coherence Provider in the EPN Assembly File

To specify that you want to use Coherence as the caching provider, use the `provider="coherence"` attribute of the `<wlevs:caching-system>` tag in the EPN assembly file of your application, as shown:

```
<wlevs:caching-system id="caching-system-id" provider="coherence"/>
```

You declare a cache of this caching system in the standard way, referencing the id of the caching system you just declared:

```
<wlevs:cache id="myCache" advertise="false">
    <wlevs:caching-system ref="caching-system-id"/>
    <wlevs:cache-loader ref="localLoader"/>
    <wlevs:cache-listener ref="localListener"/>
</wlevs:cache>

<bean id="localLoader"
      class="com.bea.wlevs.example.provider.coherence.LocalLoader"/>

<bean id="localListener"
      class="com.bea.wlevs.example.provider.coherence.LocalListener"/>
```

The `id="myCache"` attribute of the `<wlevs:cache>` element maps to the name of a cache in the Coherence configuration files; this is discussed in more detail later.

An important difference between declaring the Coherence provider and the local in-memory provider is that, when using Coherence, you can configure *either* a `<wlevs:cache-loader>` or a `<wlevs:cache-store>` child element of the `<wlevs:cache>` element in the EPN assembly file, but not both. This is because Coherence combines the loader and store into a single component. You specify a cache loader when the backing store is read-only and a cache store when the backing store is read-write.

If multiple application bundles need to share Coherence caches, then you should put the EPN assembly file that contains the appropriate `<wlevs:cache>` and `<wlevs:caching-system>` in a separate bundle and set their `advertise` attributes to `true`.

## Configuring the Coherence Caching System

Oracle CEP leverages the native configuration provided by Coherence. You do this by packaging the following two Coherence configuration files, with the indicated names, in the application bundle that uses the Coherence caching:

- `coherence-cache-config.xml`—Coherence cache configuration information. Individual caches are identified with the `<cache-name>` element; the value of this element maps to the `id` attribute of the `<wlevs:cache>` element in the EPN assembly file. See [“The coherence-cache-config.xml File” on page 8-23](#) for information about this file as well as an example of the mapping.
- `tangasol-coherence-override.xml`—Coherence cluster configuration. See [“The tangasol-coherence-override.xml File” on page 8-26](#) for information about this file as well as an example.

When assembling your application, put these two files in the `META-INF/wlevs/coherence` directory of the bundle JAR. Note that this directory is different from the directory that stores the component configuration file for the local in-memory Oracle CEP caching provider (`META-INF/wlevs`).

When you declare that a caching system uses the Coherence provider, be sure that all of the caches of this caching system also map to a Coherence configuration and not a local configuration, or Oracle CEP throws an exception.

### The coherence-cache-config.xml File

The `coherence-cache-config.xml` file is the basic Coherence configuration file and must conform to the Coherence DTDs, as it true for any Coherence application.

The following sample shows a simple configuration. See the explanation after the sample for information about the sections in bold.

```
<?xml version="1.0"?>
<!DOCTYPE cache-config SYSTEM "cache-config.dtd">
<cache-config>
  < caching-scheme-mapping>
    < cache-mapping>
      < cache-name>myCoherenceCache</cache-name>
      < scheme-name>new-replicated</scheme-name>
    </cache-mapping>
  </caching-scheme-mapping>
</cache-config>
```

```
<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>
</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>
    <cachestore-scheme>
      <class-scheme>

<class-factory-name>com.bea.wlevs.cache.coherence.configuration.SpringFact
ory</class-factory-name>
  <method-name>getLoader</method-name>
```

```

        <init-params>
            <init-param>
                <param-type>java.lang.String</param-type>
                <param-value>{cache-name}</param-value>
            </init-param>
        </init-params>
    </class-scheme>
</cachestore-scheme>

</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>
    <cachestore-scheme>
        <class-scheme>

<class-factory-name>com.bea.wlevs.cache.coherence.configuration.SpringFact
ory</class-factory-name>
    <method-name>getStore</method-name>
    <init-params>
        <init-param>
            <param-type>java.lang.String</param-type>
            <param-value>{cache-name}</param-value>
        </init-param>
    </init-params>
    </class-scheme>
</cachestore-scheme>
</local-scheme>
</caching-schemes>
</cache-config>

```

In the Coherence configuration file, the `<cache-name>` element that is a child element of `<cache-mapping>` identifies the name of the Coherence cache. The value of this element must exactly match the value of the `id` attribute of the `<wlevs:cache>` element in the EPN assembly file. For example, the following EPN assembly file snippet refers to the `myCoherenceCache` cache in the Coherence configuration file:

```
<wlevs:cache id="myCoherenceCache" advertise="false">
    <wlevs:caching-system ref="coherence-cache"/>
    <wlevs:cache-loader ref="localLoader"/>
    <wlevs:cache-listener ref="localListener"/>
</wlevs:cache>
```

The Coherence configuration file illustrates another requirement when using Coherence with Oracle CEP: a Coherence factory must be declared when using Spring to configure a loader or store for a cache. You do this using the `<cachestore-scheme>` element in the Coherence configuration file to specify a factory class that allows Coherence to call into Oracle CEP 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.

When declaring the Coherence caches in the EPN assembly files of one or more applications deployed to the same Oracle CEP server, you should never configure multiple instances of the same cache with a loader or store. You might inadvertently do this by employing multiple applications that each configure the same Coherence cache with a loader or store in their respective EPN assembly file. If you do this, Oracle CEP throws an exception.

If multiple application bundles need to share Coherence caches, then you should put the EPN assembly file that contains the appropriate `<wlevs:cache>` and `<wlevs:caching-system>` in a separate bundle and set their `advertise` attributes to `true`.

Refer to your [Coherence documentation](#) for detailed information about the `coherence-cache-config.xml` file.

## The tangasol-coherence-override.xml File

The `tangasol-coherence-override.xml` file configures Coherence clusters.

The following sample shows a simple configuration. See the explanation after the sample for information about the sections in bold.

```
<?xml version='1.0'?>

<coherence xml-override="/tangasol-coherence-override.xml">
    <cluster-config>
        <member-identity>
            <cluster-name>com.bea.wlevs.example.provider</cluster-name>
        </member-identity>
```



```

    <multicast-listener>
      <time-to-live
system-property="tangosol.coherence.ttl">4</time-to-live>
      <join-timeout-milliseconds>3000</join-timeout-milliseconds>
    </multicast-listener>

    <packet-publisher>
      <packet-delivery>
        <timeout-milliseconds>30000</timeout-milliseconds>
      </packet-delivery>
    </packet-publisher>
  </cluster-config>

  <logging-config>
    <severity-level
system-property="tangosol.coherence.log.level">5</severity-level>
    <character-limit
system-property="tangosol.coherence.log.limit">0</character-limit>
    </logging-config>
</coherence>

```

This configuration file is fairly standard. The main thing to note is that you should specify a `<cluster-name>` element to prevent Coherence from attempting to join existing Coherence clusters when Oracle CEP starts up; this can cause problems and sometimes even prevent Oracle CEP from starting.

Refer to your [Coherence documentation](#) for detailed information about the `tangosol-coherence-override.xml` file.



# Using a Persistent Store to Record and Playback Events

This section contains information on the following subjects:

- [“Overview of Using a Persistent Store to Record and Playback Events” on page 9-1](#)
- [“Implementing Record and Playback of Events in Your Application: Typical Steps” on page 9-4](#)
- [“Creating a Custom Event Store Provider” on page 9-17](#)

## Overview of Using a Persistent Store to Record and Playback Events

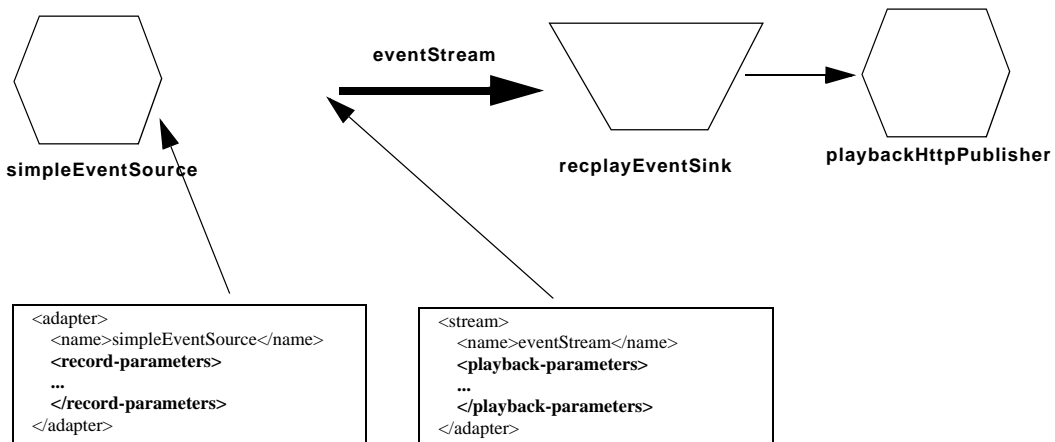
Oracle CEP event repository feature allows you to persist the events that flow out of a component of the event processing network (EPN) to a store, such as a database table, and then play them back at a later stage or explicitly query the events from a component such as an event bean.

A typical use case of this feature is the ability to debug a problem with a currently running application. If you have been recording the events at a node in the EPN when the problem occurred, you can later playback the same list of events to recreate the problem scenario for debugging purposes.

The following graphic shows the EPN of the Event Record and Playback example and demonstrates at what point events are recorded and where they are played back. The `simpleEventSource` adapter has been configured to record events; as indicated, the record happens as events flow *out* of the adapter. The `eventStream` stream has been configured to

playback events; as indicated, the playback happens at the point where events flow *into* the stream.

**Figure 9-1 Configuring Record and Playback in an EPN**



## The Event Store

Oracle CEP provides an RDBMS-based implementation of the event store; this implementation stores data in a relational database. You can use either the database server included with Oracle CEP (Apache Derby) or another database server, such as Oracle DBMS.

Oracle CEP includes a default database server, [Apache Derby](#), that you can use to record events. Apache Derby is an open source relational database implemented entirely in Java. By default, Derby creates its database files and log file (`derby.log`) from the directory in which you started Oracle CEP, such as `DOMAIN_DIR/servername`. You can change this default location by setting the system property `derby.system.home` to a different directory.

You can also create a custom event store provider to store events in a non-RDBMS persistent store. For details, see [“Creating a Custom Event Store Provider” on page 9-17](#).

## Recording Events

You can configure recording for any component in the event processing network (EPN) that produces events: processors, adapters, streams, and event beans. Processors and streams always produce events; adapters and event beans must implement the `EventSource` interface.

Additionally, you can configure that events from different components in the EPN be stored in different persistent stores, or that all events go to the same store. Note that only events that are outputted by the component are recorded.

You enable the recording of events for a component by updating its configuration file and adding the `<record-parameters>` element. Using the child elements of `<record-parameters>`, you specify the event store to which the events are recorded, an initial time period when recording should take place, the list of event types you want to store, and so on.

After you deploy the application and events start flowing through the network, recording begins either automatically because you configured it to start at a certain time or because you dynamically start it using administration tools. For each component you have configured for recording, Oracle CEP stores the events that flow out of it to the appropriate store along with a timestamp of the time it was recorded.

## Playing Back Events

You can configure playback for any component in the event processing network (EPN): processors, adapters, streams, and event beans. Typically the playback component is a node later in the network than the node that recorded the events.

You enable the playback of events for a component by updating its configuration file and adding the `<playback-parameters>` element. Using the child elements of `<playback-parameters>`, you specify the event store from which the events are played back, the list event types you want to play back (by default all are played back), the time range of the recorded events you want to play back, and so on. By default, Oracle CEP plays back the events in a time accurate manner; however, you can also configure that the events get played back either faster or slower than they originally flowed out of the component from which they were recorded.

After you deploy the application and events start flowing through the network, you must start the playback by using the administration tools (Visualizer or `wlevs.Admin`). Oracle CEP reads the events from the appropriate persistent store and inserts them into the appropriate place in the EPN.

It is important to note that when a component gets a playback event, it looks exactly like the original event. Additionally, a component later in the network has been configured to record events, then Oracle CEP records the playback events as well as the "real" events.

## Querying Stored Events

You can use the event store API to query a store for past events given a record time range and the component from which the events were recorded. The actual query you use depends on the event repository provider; for example, you would use EPL for the RDBMS provider included in Oracle CEP. You can also use these APIs to delete old events from the event store.

## Restrictions on the Event Types that Can Be Recorded

In this release, the following restrictions apply to the event types that can be recorded to the event store if you use the Oracle RDBMS-based provider:

- The fields of the event type must be simple data types. In other words, the fields cannot be complex types such as arrays, `java.util.Map`, or nested objects.
- The simple fields must be one of the supported Java types, as listed in [“Supported Java Types” on page 9-17](#).

## Record and Playback Example

The sample code in this section is taken from the `recplay` example, located in the `ORACLE_CEP_HOME\ocep_10.3\samples\source\applications\recplay` directory, where `ORACLE_CEP_HOME` refers to the main Oracle CEP installation directory, such as `d:\oracle_cep`.

See [Event Record and Playback Example](#) for details about running and building the example.

## Implementing Record and Playback of Events in Your Application: Typical Steps

Depending on how you are going to use the event repository, there are different tasks that you must perform, as described in the following procedure that in turn point to sections with additional details.

**Note:** It is assumed in this section that you have already created an Oracle CEP application along with component configuration file(s) for the components and that you want to update the application so that components record or playback events. If you have not, refer to [“Overview of Creating Oracle Complex Event Processing Applications” on page 2-1](#) for details.

1. Configure an event store for your Oracle CEP server instance. You must perform this configuration step even if you plan to use the event store provider provided by Oracle CEP, [Apache Derby](#).  
See [“Configuring an Event Store for Oracle CEP Server” on page 9-5](#).
2. Configure a component in your EPN to record events by updating the component’s configuration file. The component can be a processor, adapter, stream, or event bean. Only events flowing out of the component are recorded.  
See [“Configuring a Component to Store Events” on page 9-7](#).
3. Configure a component in your EPN to playback events by updating the component’s configuration file. The component can be a processor, adapter, stream, or event bean. Only components that are also event sinks can playback events; events are played to the input side of the component.  
See [“Configuring a Component to Playback Events” on page 9-11](#).
4. Redeploy your application for the changes to take effect.
5. If you have not specified an explicit start and end time for recording events, you must use Visualizer or `wlevs.Admin` to start recording. You must always use these administration tools to start and end the playback of events.  
See [“Starting and Stopping the Record and Playback of Events” on page 9-15](#).

## Configuring an Event Store for Oracle CEP Server

Oracle CEP stores recorded events in a database, which means that before you can start using the record and playback feature in your own application, you must specify where the database server is located along with the name of the database server that will contain the recorded events. You do this by updating the `config.xml` file of your Oracle CEP server instance, as described in later procedures.

You do not, however, create the actual tables that store the recorded events; Oracle CEP automatically does this for you after you deploy the application that uses the record and playback feature and recording starts to take place. See [“Description of the Database Tables Created by the RDMBS Provider” on page 9-16](#) for information about how Oracle CEP creates these tables.

To configure Oracle CEP for an event store, including the default Apache Derby, follow these steps:

1. Stop your Oracle CEP server instance, if it is running.

2. Using your favorite XML editor, open the server's `config.xml` file for edit. The `config.xml` file is located in the `DOMAIN_DIR/servername/config` directory of your server, where `DOMAIN_DIR` refers to the domain directory, such as `/oracle_cep/user_projects/domains/myDomain` and `servername` refers to the name of your server, such as `defaultserver`.
3. Configure access to your relational database by adding a `<data-source>` element to the `config.xml` file. For details, see [Configuring Access to a Relational Database](#).

If you want to use the default Apache Derby database, add the following `<data-source>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<n1:config xmlns:n1="http://www.bea.com/ns/wlevs/config/server"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <domain>
    <name>myDomain</name>
  </domain>

  ...

  <data-source>
    <name>derby1</name>
    <connection-pool-params>
      <initial-capacity>15</initial-capacity>
      <max-capacity>50</max-capacity>
    </connection-pool-params>
    <driver-params>
      <url>jdbc:derby:dbtest1;create=true</url>
      <driver-name>org.apache.derby.jdbc.EmbeddedDriver</driver-name>
    </driver-params>
  </data-source>

</n1:config>
```

You can, of course, name the data source anything you want, as well as specify different connection pool parameters and database name (the sample above connects to the database `dbtest1` and specifies that the database should be created if it does not already exist). The only element that must be the same as in the sample above is `<driver-name>`, whose value must be `org.apache.derby.jdbc.EmbeddedDriver`. You are required to add a `<transaction-manager>` element, although you can name it anything you want.

4. Add a `<transaction-manager>` element to the `config.xml` file:

```
<data-source>
...
</data-source>
```



```
<transaction-manager>
  <name>myTransactionManager</name>
</transaction-manager>
```

5. Add an `<rdbms-event-store-provider>` element to the `config.xml` file that in turn references the previously configured data source. For example, to use the Apache Derby database configured in the previous step, add the following XML snippet:

```
<data-source>
  <name>derby1</name>
  ...
</data-source>

<rdbms-event-store-provider>
  <name>my-rdbms-provider</name>
  <data-source-name>derby1</data-source-name>
</rdbms-event-store-provider>
```

Later, when you configure the components that record and playback the events, you will specify the name of this event store.

Oracle CEP is now configured for an event store and you can configure specific components in your application to record or playback events.

## Configuring a Component to Store Events

You can configure any processor, adapter, stream, or event bean in your application to record events. As with all other component configuration, you specify that a component records events by updating its configuration file. For general information about these configuration files, see [“Component Configuration Files” on page 2-3](#).

This section describes the main steps to configure a component to record events. For simplicity, it is assumed in the procedure that you are configuring an adapter to record events and that you have already created its component configuration file.

See [XSD Schema Reference for Component Configuration Files](#) for the complete XSD Schema that describes the event recording configuration file elements.

Using your favorite XML editor, open the component configuration XML file and add a `<record-parameters>` child element to the component you want to configure to record events. For example, to configure an adapter called `simpleEventSource`:

```
<?xml version="1.0" encoding="UTF-8"?>
<nl:config xmlns:nl="http://www.bea.com/ns/wlevs/config/application"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

## Using a Persistent Store to Record and Playback Events

```
<adapter>
  <name>simpleEventSource</name>
  <record-parameters>
    ...
  </record-parameters>
  ...
</adapter>
...
```

```
</nl:config>
```

Add child elements to `<record-parameters>` to specify the name of the event store provider, the events that are stored, the start and stop time for recording, and so on. For example:

```
<adapter>
  <name>simpleEventSource</name>
  <record-parameters>
    <dataset-name>recplay_sample</dataset-name>
    <event-type-list>
      <event-type>SimpleEvent</event-type>
    </event-type-list>
    <provider-name>my-rdbms-provider</provider-name>
    <batch-size>1</batch-size>
    <batch-time-out>10</batch-time-out>
  </record-parameters>
</adapter>
```

The following table lists the child elements of `<record-parameters>` that you can specify. Although only `<dataset-name>` is required, Oracle recommends that you also include `<provider-name>` to explicitly specify the name of the event store provider.

**Table 9-1 Child Elements of `<record-parameters>`**

Child Element	Description
<code>&lt;dataset-name&gt;</code>	<p>Specifies the group of data that the user wants to group together.</p> <p>In the case of the Oracle RDBMS-based provider, it specifies the database area, or schema, in which the tables that store the recorded events are created.</p> <p>When configuring the Oracle RDBMS-based provider, you are required to specify this element.</p>
<code>&lt;provider-name&gt;</code>	<p>Specifies the name of the event store provider.</p> <p>The value of this element corresponds to the value of the <code>&lt;name&gt;</code> child element of the <code>&lt;rdbms-event-store-provider&gt;</code> element in the <code>config.xml</code> file of the Oracle CEP server instance.</p> <p>See <a href="#">“Configuring an Event Store for Oracle CEP Server” on page 9-5</a></p> <p>When configuring the Oracle RDBMS-based provider, you are required to specify this element.</p>
<code>&lt;event-type-list&gt;</code>	<p>Specifies the event types that are recorded to the event store. If this element is not specified, then Oracle CEP records <i>all</i> event types that flow out of the component.</p> <p>Use the <code>&lt;event-type&gt;</code> child component to list all events, such as:</p> <pre>&lt;event-type-list&gt;   &lt;event-type&gt;EventOne&lt;/event-type&gt;   &lt;event-type&gt;EventTwo&lt;/event-type&gt; &lt;/event-type-list&gt;</pre> <p>When configuring the Oracle RDBMS-based provider, you are required to specify this element.</p>

**Table 9-1 Child Elements of <record-parameters>**

Child Element	Description
<time-range>	<p>Specifies the time period during which recording should take place using a start and end time.</p> <p>The time period is configured by using a &lt;start&gt; child element to specify a start time and an &lt;end&gt; child element to specify the end time. The start and end time formats are both MM-dd-yyyy:HH:mm:ss, such as 10-20-2007:11:22:07. For example, to specify that recording should start on April 7, 2009, at 6:00am and end on April 10, 2009, at 10:00 pm, enter the following</p> <pre>&lt;time-range-offset&gt;   &lt;start&gt;04-07-2009:06:00:00&lt;/start&gt;   &lt;end&gt;04-10-2009:22:00:00&lt;/end&gt; &lt;/time-range-offset&gt;</pre> <p>If you do not specify a time period, then no events are recorded when the application is deployed and recording will only happen after you explicitly start it using Visualizer or wlevs.Admin.</p> <p>You can specify &lt;time-range&gt; or &lt;time-range-offset&gt;, but not both.</p>
<time-range-offset>	<p>Specifies the time period during which recording should take place, using a start time and a duration.</p> <p>The time period is configured by using a &lt;start&gt; child element to specify a start time and &lt;duration&gt; child element to specify the amount of time after the start time that recording should stop. The start time format is MM-dd-yyyy:HH:mm:ss, such as 10-20-2007:11:22:07. The duration format is HH:mm:ss, such as 01:00:00. For example, to specify that recording should start on April 7, 2009, at 6:00am and continue for 3 hours, enter the following</p> <pre>&lt;time-range-offset&gt;   &lt;start&gt;04-07-2009:06:00:00&lt;/start&gt;   &lt;duration&gt;03:00:00&lt;/duration&gt; &lt;/time-range-offset&gt;</pre> <p>If you do not specify a time period, then no events are recorded when the application is deployed and recording will only happen after you explicitly start it using Visualizer or wlevs.Admin.</p> <p>You can specify &lt;time-range&gt; or &lt;time-range-offset&gt;, but not both.</p>
<batch-size>	<p>Specifies the number of events that Oracle CEP picks up in a single batch from the event buffer to write the event store.</p> <p>Default value is 1000.</p>

**Table 9-1 Child Elements of <record-parameters>**

Child Element	Description
<batch-time-out>	Specifies the number of seconds that Oracle CEP waits for the event buffer window to fill up with the <batch-size> number of events before writing to the event store.  Default value is 60
<max-size>	If specified, Oracle CEP uses a stream when writing to the event store, and this element specifies the size of the stream, with non-zero values indicating asynchronous writes.  Default value is 1024.
<max-threads>	If specified, Oracle CEP uses a stream when writing to the event store, and this element specifies the maximum number of threads that will be used to process events for this stream. Setting this value has no effect when <max-size> is 0.  The default value is 1.
<store-policy-parameters>	Specifies policy parameters, specific to the event store provider. Use the <name> and <value> child elements to specify a particular parameter, such as:  <pre> &lt;store-policy-parameters&gt;   &lt;name&gt;supportsTransactions&lt;/name&gt;   &lt;value&gt;true&lt;/value&gt; &lt;/store-policy-parameters&gt; </pre>

## Configuring a Component to Playback Events

You can configure any processor, adapter, stream, or event bean in your application to playback events, although the component must be a node downstream of the recording component so that the playback component will actually receive the events and play them back. As with all other component configuration, you specify that a component plays back events by updating its configuration file. For general information about these configuration files, see [“Component Configuration Files” on page 2-3](#).

This section describes the main steps to configure a component to play back events. For simplicity, it is assumed in the procedure that you are configuring a stream to playback events from a node upstream in the EPN that has recorded events, and that you have already created the stream’s configuration file.

See [XSD Schema Reference for Component Configuration Files](#) for the complete XSD Schema that describes the event playback configuration file elements.

Using your favorite XML editor, open the component configuration XML file and add a `<playback-parameters>` child element to the component you want to configure to playback events. For example, to configure a stream called `eventStream`:

```
<?xml version="1.0" encoding="UTF-8"?>
<nl:config xmlns:nl="http://www.bea.com/ns/wlevs/config/application"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <stream>
    <name>eventStream</name>
    <playback-parameters>
      ...
    </playback-parameters>
  </stream>

  ...
</nl:config>
```

Add child elements to `<playback-parameters>` to specify the name of the event store provider, the events that are played back, and so on. For example:

```
<stream>
  <name>eventStream</name>
  <playback-parameters>
    <dataset-name>recplay_sample</dataset-name>
    <event-type-list>
      <event-type>SimpleEvent</event-type>
    </event-type-list>
    <provider-name>test-rdbms-provider</provider-name>
  </playback-parameters>
</stream>
```

The following table lists the child elements of `<playback-parameters>` that you can specify. Although only `<dataset-name>` is required, Oracle recommends that you also include `<provider-name>` to explicitly specify the name of the event store provider.

**Table 9-2 Child Elements of `<playback-parameters>`**

Child Element	Description
<code>&lt;dataset-name&gt;</code>	<p>Specifies the group of data that the user wants to group together.</p> <p>In the case of the Oracle RDBMS-based provider, it specifies the database area, or schema, in which the tables that store the recorded events are queried for the playback events.</p> <p>When configuring the Oracle RDBMS-based provider, you are required to specify this element.</p>
<code>&lt;provider-name&gt;</code>	<p>Specifies the name of the event store provider.</p> <p>The value of this element corresponds to the value of the <code>&lt;name&gt;</code> child element of the <code>&lt;rdbms-event-store-provider&gt;</code> element in the <code>config.xml</code> file of the Oracle CEP server instance.</p> <p>See <a href="#">“Configuring an Event Store for Oracle CEP Server” on page 9-5</a>.</p> <p>When configuring the Oracle RDBMS-based provider, you are required to specify this element.</p>
<code>&lt;event-type-list&gt;</code>	<p>Specifies the event types that are played back from the event store. If this element is not specified, then Oracle CEP plays back <i>all</i> event types.</p> <p>Use the <code>&lt;event-type&gt;</code> child component to list all events, such as:</p> <pre>&lt;event-type-list&gt;   &lt;event-type&gt;EventOne&lt;/event-type&gt;   &lt;event-type&gt;EventTwo&lt;/event-type&gt; &lt;/event-type-list&gt;</pre> <p>When configuring the Oracle RDBMS-based provider, you are required to specify this element.</p>

**Table 9-2 Child Elements of <playback-parameters>**

Child Element	Description
<time-range>	<p>Specifies the time period of the recorded events that you want to play back. In other words, this element acts as a filter so that only the events recorded during the specified time period are played back. This element does <i>not</i> specify a time period when the playback itself occurs.</p> <p>The time period is configured by using a &lt;start&gt; child element to specify a start time and an &lt;end&gt; child element to specify the end time. The start and end time formats are both MM-dd-yyyy:HH:mm:ss, such as 10-20-2007:11:22:07. For example, to specify that you want to playback events that were recorded starting on April 7, 2009, at 6:00am and ending on April 10, 2009, at 10:00 pm, enter the following</p> <pre>&lt;time-range-offset&gt;   &lt;start&gt;04-07-2009:06:00:00&lt;/start&gt;   &lt;end&gt;04-10-2009:22:00:00&lt;/end&gt; &lt;/time-range-offset&gt;</pre> <p>If you do not specify a time period, then all events are played back.</p> <p>You can specify &lt;time-range&gt; or &lt;time-range-offset&gt;, but not both.</p>
<time-range-offset>	<p>Specifies the time period of the recorded events that you want to play back. In other words, this element acts as a filter so that only the events recorded during the specified time period are played back. This element does <i>not</i> specify a time period when the playback itself occurs.</p> <p>The time period is configured by using a &lt;start&gt; child element to specify a start time and &lt;duration&gt; child element to specify an amount of time after the start time. The start time format is MM-dd-yyyy:HH:mm:ss, such as 10-20-2007:11:22:07. The duration format is HH:mm:ss, such as 01:00:00. For example, to specify that you want to playback events that were recorded starting on April 7, 2009, at 6:00am and continued for three hours enter the following:</p> <pre>&lt;time-range-offset&gt;   &lt;start&gt;04-07-2009:06:00:00&lt;/start&gt;   &lt;duration&gt;03:00:00&lt;/duration&gt; &lt;/time-range-offset&gt;</pre> <p>If you do not specify a time period, then all events are played back.</p>
<playback-speed>	<p>Specifies the playback speed as a positive float.</p> <p>The default value is 1, which corresponds to normal speed. A value of 2 means that events will be played back 2 times faster than the original record speed. Similarly, a value of 0.5 means that events will be played back at half the speed.</p>



**Table 9-2 Child Elements of <playback-parameters>**

Child Element	Description
<repeat>	<p>Specifies whether to playback events again after the playback of the specified time interval is over.</p> <p>Valid values are <code>true</code> and <code>false</code>. Default value is <code>false</code>. A value of <code>true</code> means that the repeat of playback continues an infinite number of times until it is deliberately stopped. <code>False</code> means that events will be played back only once.</p>
<max-size>	<p>If specified, Oracle CEP uses a stream when playing back events from the event store, and this element specifies the size of the stream, with non-zero values indicating asynchronous writes.</p> <p>Default value is 1024.</p>
<max-threads>	<p>If specified, Oracle CEP uses a stream when playing back events from the event store, and this element specifies the maximum number of threads that will be used to process events for this stream. Setting this value has no effect when &lt;max-size&gt; is 0.</p> <p>The default value is 1.</p>
<store-policy-parameters>	<p>Specifies policy parameters, specific to the event store provider. Use the &lt;name&gt; and &lt;value&gt; child elements to specify a particular parameter, such as:</p> <pre> &lt;store-policy-parameters&gt;   &lt;name&gt;supportsTransactions&lt;/name&gt;   &lt;value&gt;true&lt;/value&gt; &lt;/store-policy-parameters&gt; </pre>

## Starting and Stopping the Record and Playback of Events

After you configure the record and playback functionality for the components of an application, and you deploy the application to Oracle CEP, the server starts to record events only if you specified an explicit start/stop time in the initial configuration.

For example, if you included the following element in a component configuration:

```

<time-range-offset>
  <start>04-07-2009:06:00:00</start>
  <end>04-10-2009:22:00:00</end>
</time-range-offset>

```

then recording will automatically start on April 7, 2009.

The only way to start the playback of events, however, is by using Visualizer or `wlevs.Admin`. You also use these tools to dynamically start and stop the recording of events. For details, see:

- [Commands for Controlling Event Record and Playback](#) (`wlevs.Admin`)
- [Record and Play Back Events Flowing Through an EPN](#) (Visualizer)

Visualizer and `wlevs.Admin` use managed beans (MBeans) to dynamically start and stop event recording and playback, as well as manage the event store configuration. A managed bean is a Java bean that provides a Java Management Extensions (JMX) interface. JMX is the Java EE solution for monitoring and managing resources on a network. You can create your own administration tool and use JMX to manage event store functionality by using the `com.bea.wlevs.management.configuration.StageMBean`.

## Description of the Database Tables Created by the RDMBS Provider

When you enable event recording for a component, Oracle CEP automatically creates, if they do not already exist, the database tables that stores the actual data. The server connects to the database using the data source configured for the event store provider.

Oracle recommends that you allow Oracle CEP to create and manage these tables; however, if you want more control you can create your own tables.

The following guidelines describe how Oracle CEP creates the database tables, and thus the requirements if you decide to create your own tables. If the default name of a database object (schema, table, column) is a reserved word, Oracle CEP adds a suffix to come up with a non-reserved word.

- Events are stored in a schema, whose name is specified by the value of the `<dataset-name>` element.
- Every Java event type in the event store has a corresponding table in the schema. The name of the table is same as the name of the event type.
- Every field in the Java event type has a corresponding column in the table. The name of the column is the same as the name of the field. See [“Supported Java Types” on page 9-17](#) for the list of supported Java types for the fields of the event type.
- The table contains an additional column `recordTime` that stores the time the event was recorded.
- The table has an index on the `recordTime` column.

The following table lists the supported Java types that can be fields of the event type.

**Table 9-3 Supported Java Types**

Java Type	Corrponding Java SQL Type
<code>boolean</code> , <code>java.lang.Boolean</code>	<code>java.sql.Types.BIT</code>
<code>byte</code> , <code>java.lang.Byte</code>	<code>java.sql.Types.TINYINT</code>
<code>short</code> , <code>java.lang.Short</code>	<code>java.sql.Types.SMALLINT</code>
<code>int</code> , <code>java.lang.Integer</code>	<code>java.sql.Types.INTEGER</code>
<code>long</code> , <code>java.lang.Long</code>	<code>java.sql.Types.BIGINT</code>
<code>float</code> , <code>java.lang.Float</code>	<code>java.sql.Types.REAL</code>
<code>double</code> , <code>java.lang.Double</code>	<code>java.sql.Types.DOUBLE</code>
<code>java.lang.String</code>	<code>java.sql.Types.VARCHAR</code>
<code>java.math.BigDecimal</code>	<code>java.sql.Types.BIGINT</code>
<code>java.sql.Date</code> , <code>java.util.Date</code>	<code>java.sql.Types.TIMESTAMP</code>
<code>java.sql.Time</code>	<code>java.sql.Types.TIME</code>
<code>java.sql.Timestamp</code>	<code>java.sql.Types.TIMESTAMP</code>

## Creating a Custom Event Store Provider

Oracle CEP provides an [event store API](#) that you can use to create a custom event store provider. Oracle provides an RDBMS-based implementation for storing events in a relational database, or one that supports JDBC connections. If you want to store events in a different kind of database, or for some reason the Oracle RDBMS provider is not adequate for your needs, then you can create your own event store provider using the event store API.

The event store API is in the [com.bea.wlevs.eventstore](#) package; the following list describes the most important interfaces:

- `EventStore`—Object that represents a single event store. The methods of this interface allow you to persist events to the store and to query the contents of the store using a provider-specific query.

- `EventStoreManager`—Manages event stores. Only one instance of the `EventStoreManager` ever exists on a given Oracle CEP server, and this instance registers itself in the OSGI registry so that event store providers can in turn register themselves with the event store manager. You use this interface to find existing event stores, create new ones, get the provider for a given event store, and register an event provider. The event store manager delegates the actual work to the event store provider.
- `EventStoreProvider`—Underlying repository that provides event store services to clients.

# Assembling and Deploying Oracle Complex Event Processing Applications

This section contains information on the following subjects:

- [“Overview of Application Assembly and Deployment” on page 10-1](#)
- [“Assembling an Oracle CEP Application: Main Steps” on page 10-2](#)
- [“Deploying Oracle CEP Applications: Main Steps” on page 10-7](#)

## Overview of Application Assembly and Deployment

The term *application assembly* refers to the process of packaging the components of an application, such as the Java files and XML configuration files, into an OSGI bundle that can be deployed to Oracle Complex Event Processing, or *Oracle CEP* for short. The term *application deployment* refers to the process of making an application available for processing client requests in an Oracle CEP domain.

In the context of Oracle CEP assembly and deployment, an application is defined as an [OSGi bundle](#) JAR file that contains the following artifacts:

- The compiled Java class files that implement some of the components of the application, such as the adapters, adapter factory, and POJO that contains the business logic.
- One or more Oracle CEP configuration XML files that configure the components of the application. The only type of component that is required to have a configuration file is the complex event processor; all other components (adapters and streams) do not require configuration files if the default configuration of the component is adequate. You can

combine all configuration files into a single file, or separate the configuration for individual components in their own files.

The configuration files must be located in the `META-INF/wlevs` directory of the OSGi bundle JAR file if you plan to dynamically deploy the bundle. If you have an application already present in the domain directory, then the configuration files need to be extracted in the same directory.

- An EPN assembly file that describes all the components of the application and how they are connected to each other.

The EPN assembly file must be located in the `META-INF/spring` directory of the OSGi bundle JAR file.

- A `MANIFEST.MF` file that describes the contents of the JAR.

The OSGi bundle declares dependencies by specifying imported and required packages. It also provides functionality to other bundles by exporting packages. If a bundle is required to provide functionality to other bundles, you must use `Export-Package` to allow other bundles to reference named packages. All packages not exported are not available outside the bundle.

See [“Assembling an Oracle CEP Application: Main Steps” on page 10-2](#) for detailed instructions on creating this deployment bundle.

After you have assembled the application, you deploy it by making it known to the Oracle CEP domain using the Deployer utility (packaged in the `wlevsdeploy.jar` file). For detailed instructions, see [“Deploying Oracle CEP Applications: Main Steps” on page 10-7](#).

Once the application is deployed to Oracle CEP, the configured adapters immediately start listening for events for which they are configured, such as financial data feeds and so on.

**Note:** Oracle CEP applications are built on top of the Spring Framework and OSGi Service Platform and make extensive use of their technologies and services. See [“Additional Information about Spring and OSGi” on page A-1](#) for links to reference and conceptual information about Spring and OSGi.

## Assembling an Oracle CEP Application: Main Steps

Assembling an Oracle CEP application refers to bundling the artifacts that make up the application into an [OSGi bundle](#) JAR file. These artifacts include compiled Java classes, the XML files that configure the components of the application (such as the processors or adapters), the EPN assembly file, and the `MANIFEST.MF` file.

For simplicity, the following procedure creates a temporary directory that contains the required artifacts, and then jars up the contents of this temporary directory. This is just a suggestion and you are not required, of course, to assemble the application using this method.

See [“Additional Information about Spring and OSGi” on page A-1](#) for links to reference and conceptual information about Spring and OSGi.

**Note:** See the HelloWorld example source directory for a sample `build.xml` Ant file that performs many of the steps described below. The `build.xml` file is located in `ORACLE_CEP_HOME\ocep_10.3\samples\source\applications\helloworld`, where `ORACLE_CEP_HOME` refers to the main Oracle CEP installation directory, such as `d:\oracle_cep`.

To assemble an Oracle CEP application:

1. Open a command window and set your environment as described in [Setting Up Your Development Environment](#).
2. Create an empty directory, such as `output`:  

```
prompt> mkdir output
```
3. Compile all application Java files into the `output` directory.
4. Create an `output/META-INF/spring` directory.
5. Copy the EPN assembly file that describes the components of your application and how they are connected into the `output/META-INF/spring` directory.  
 See [“Creating the EPN Assembly File” on page 2-11](#) for details about this file.
6. Create an `output/META-INF/wlevs` directory.
7. Copy the XML files that configure the components of your application (such as the processors or adapters) into the `output/META-INF/wlevs` directory. You create these XML files during the course of creating your application, as described in [“Overview of the Oracle Complex Event Processing Programming Model” on page 2-1](#).
8. Create a `MANIFEST.MF` file that contains descriptive information about the bundle.  
 See [“Creating the MANIFEST.MF File” on page 10-4](#).
9. If you need to access third-party JAR files from your Oracle CEP application, see [“Accessing Third-Party JAR Files From Your Application” on page 10-6](#).
10. Create a JAR file that contains the contents of the `output` directory. Be sure you specify the `MANIFEST.MF` file you created in the previous step rather than the default manifest file.

You can name the JAR file anything you want. In the Oracle CEP examples, the name of the JAR file is a combination of Java package name and version, such as:

```
com.bea.wlevs.example.helloworld_1.0.0.0.jar
```

Consider using a similar naming convention to clarify which bundles are deployed to the server.

See the [Apache Ant documentation](#) for information on using the `jar` task or the [J2SE documentation](#) for information on using the `jar` command-line tool.

## Creating the MANIFEST.MF File

The structure and contents of the `MANIFEST.MF` file is specified by the OSGi Framework. Although the value of many of the headers in the file is specific to your application or business, many of the headers are required by Oracle CEP. In particular, the `MANIFEST.MF` file defines the following:

- Application name—Specified with the `Bundle-Name` header.
- Symbolic application name—Specified with the `Bundle-SymbolicName` header. Many of the Oracle CEP tools, such as the `wlevs.Admin` utility and JMX subsystem, use the symbolic name of the bundle when referring to the application.
- Application version—Specified with the `Bundle-Version` header.
- Imported packages—Specified with the `Import-Package` header. Oracle CEP requires that you import the following packages at a minimum:

```
Import-Package:
com.bea.wlevs.adapter.defaultprovider;version="2.0.0.0",
com.bea.wlevs.ede;version="2.0.0.0",
com.bea.wlevs.ede.api;version="2.0.0.0",
com.bea.wlevs.ede.impl;version="2.0.0.0",
org.osgi.framework;version="1.3.0",
org.springframework.beans.factory;version="2.0.5",
org.apache.commons.logging;version="1.1.0",
com.bea.wlevs.spring;version="2.0.0.0",
com.bea.wlevs.util;version="2.0.0.0",
org.springframework.beans;version="2.0.5",
org.springframework.util;version="2.0",
org.springframework.core.annotation;version="2.0.5",
org.springframework.beans.factory;version="2.0.5",
org.springframework.beans.factory.config;version="2.0.5",
org.springframework.osgi.context;version="1.0.0",
org.springframework.osgi.service;version="1.0.0"
```



If you have extended the configuration of an adapter, then you must also import the following packages:

```
javax.xml.bind;version="2.0",
javax.xml.bind.annotation;version=2.0,
javax.xml.bind.annotation.adapters;version=2.0,
javax.xml.bind.attachment;version=2.0,
javax.xml.bind.helpers;version=2.0,
javax.xml.bind.util;version=2.0,
com.bea.wlevs.configuration;version="2.0.0.0",
com.bea.wlevs.configuration.application;version="2.0.0.0",
com.sun.xml.bind.v2;version="2.0.2"
```

- **Exported packages**—Specified with the `Export-Package` header. You should specify this header only if you need to share one or more application classes with other deployed applications. A typical example is sharing an event type `JavaBean`.

If possible, you should export packages that include only the interfaces, and not the implementation classes themselves. If other applications are using the exported classes, you will be unable to fully undeploy the application that is exporting the classes.

Exported packages are server-wide, so be sure their names are unique across the server.

The following complete `MANIFEST.MF` file is from the `HelloWorld` example, which extends the configuration of its adapter:

```
Manifest-Version: 1.0
Archiver-Version:
Build-Jdk: 1.5.0_06
Extension-Name: example.helloworld
Specification-Title: 1.0.0.0
Specification-Vendor: Oracle.
Implementation-Vendor: Oracle.
Implementation-Title: example.helloworld
Implementation-Version: 1.0.0.0
Bundle-Version: 2.0.0.0
Bundle-ManifestVersion: 1
Bundle-Vendor: Oracle.
Bundle-Copyright: Copyright (c) 2006 by Oracle.
Import-Package: com.bea.wlevs.adapter.defaultprovider;version="2.0.0.0",
               com.bea.wlevs.ede;version="2.0.0.0",
               com.bea.wlevs.ede.impl;version="2.0.0.0",
```

```
com.bea.wlevs.ede.api;version="2.0.0.0",
org.osgi.framework;version="1.3.0",
org.apache.commons.logging;version="1.1.0",
com.bea.wlevs.spring;version="2.0.0.0",
com.bea.wlevs.util;version="2.0.0.0",
net.sf.cglib.proxy,
net.sf.cglib.core,
net.sf.cglib.reflect,
org.aopalliance.aop,
org.springframework.aop.framework;version="2.0.5",
org.springframework.aop;version="2.0.5",
org.springframework.beans;version="2.0.5",
org.springframework.util;version="2.0",
org.springframework.core.annotation;version="2.0.5",
org.springframework.beans.factory;version="2.0.5",
org.springframework.beans.factory.config;version="2.0.5",
org.springframework.osgi.context;version="1.0.0",
org.springframework.osgi.service;version="1.0.0",
javax.xml.bind;version="2.0",
javax.xml.bind.annotation;version=2.0,
javax.xml.bind.annotation.adapters;version=2.0,
javax.xml.bind.attachment;version=2.0,
javax.xml.bind.helpers;version=2.0,
javax.xml.bind.util;version=2.0,
com.bea.wlevs.configuration;version="2.0.0.0",
com.bea.wlevs.configuration.application;version="2.0.0.0",
com.sun.xml.bind.v2;version="2.0.2"
Bundle-Name: example.helloworld
Bundle-Description: WLEvS example helloworld
Bundle-SymbolicName: helloworld
```

## Accessing Third-Party JAR Files From Your Application

When creating your Oracle CEP applications, you might need to access legacy libraries within existing third-party JAR files. There are two ways to ensure access to this legacy code:

- **Recommended.** Package the third-party JAR files in your Oracle CEP application JAR file. You can put the JAR files anywhere you want.

However, to ensure that your Oracle CEP application finds the classes in the third-party JAR file, you must update the application classpath by adding the `Bundle-Classpath` header to the `MANIFEST.MF` file. Set `Bundle-Classpath` to a comma-separated list of the JAR file path names that should be searched for classes and resources. Use a period (.) to specify the bundle itself. For example:

```
Bundle-Classpath: ., commons-logging.jar, myExcitingJar.jar,
myOtherExcitingJar.jar
```

If you need to access native libraries, you must also package them in your JAR file and use the `Bundle-NativeCode` header of the `MANIFEST.MF` file to specify their location in the JAR.

- If the JAR files include libraries used by *all* applications deployed to Oracle CEP, such as JDBC drivers, you can add the JAR file to the server's boot classpath by specifying the `-Xbootclasspath/a` option to the `java` command in the scripts used to start up an instance of the server.

The name of the server start script is `startwlevs.cmd` (Windows) or `startwlevs.sh` (UNIX), and the script is located in the server directory of your domain directory. The out-of-the-box sample domains are located in

`ORACLE_CEP_HOME/ocp_10.3/samples/domains`, and the user domains are located in `ORACLE_CEP_HOME/user_projects/domains`, where `ORACLE_CEP_HOME` refers to the main Oracle CEP installation directory, such as `d:\oracle_cep`.

Update the start script by adding the `-Xbootclasspath/a` option to the `java` command that executes the `wlevs_2.0.jar` file. Set the `-Xbootclasspath/a` option to the full pathname of the third-party JAR files you want to access system-wide.

For example, if you want all deployed applications to be able to access a JAR file called `e:\jars\myExcitingJAR.jar`, update the `java` command in the start script as follows (updated section shown in bold):

```
%JAVA_HOME%\bin\java -Dwlevs.home=%USER_INSTALL_DIR%
-Dbea.home=%BEA_HOME% -Xbootclasspath/a:e:\jars\myExcitingJAR.jar -jar
"%USER_INSTALL_DIR%\bin\wlevs_2.0.jar" -disablesecurity %1 %2 %3 %4 %5 %6
```

## Deploying Oracle CEP Applications: Main Steps

The following procedure describes how to deploy an application to Oracle CEP using the Deployer utility. It is assumed in the procedure that you have assembled your application as described in [“Assembling an Oracle CEP Application: Main Steps” on page 10-2](#).

See [Deployer Command-Line Reference](#) for complete reference information about the Deployer utility, in particular options to the utility that are supported in addition to the ones described in

this section. See [“Additional Information about Spring and OSGi” on page A-1](#) for links to reference and conceptual information about Spring and OSGi.

1. Open a command window and set your environment as described in [Setting Up Your Development Environment](#).
2. Update your CLASSPATH variable to include the `wlevsdeploy.jar` JAR file, located in the `ORACLE_CEP_HOME/ocp_10.3/bin` directory where, `ORACLE_CEP_HOME` refers to the main Oracle CEP installation directory, such as `/oracle_cep`.

**Note:** If you are running the deployer utility on a remote computer, see [Running the Deployer Utility Remotely](#) for instructions.

3. Be sure you have configured Jetty for the Oracle CEP instance to which you are deploying your application.

See [Configuring Oracle CEP](#).

4. In the command window, run the `Deployer` utility using the following syntax to install your application:

```
prompt> java -jar wlevsdeploy.jar -url http://host:port/wlevsdeployer
-user user -password password -install application_jar_file
```

where

- `host` refers to the hostname of the computer on which Oracle CEP is running.
- `port` refers to the port number to which Oracle CEP listens; its value is 9002 by default. This port is specified in the `config.xml` file that describes your Oracle CEP domain, located in the `DOMAIN_DIR/config` directory, where `DOMAIN_DIR` refers to your domain directory. The port number is the value of the `<Port>` child element of the `<Netio>` element:

```
<Netio>
  <Name>NetIO</Name>
  <Port>9002</Port>
</Netio>
```

- `user` refers to the username of the Oracle CEP administrator.
- `password` refers to the password of the Oracle CEP administrator.
- `application_jar_file` refers to your application JAR file, assembled into an OSGi bundle as described in [“Assembling an Oracle CEP Application: Main Steps” on page 10-2](#). This file must be located on the same computer from which you execute the `Deployer` utility.

For example, if Oracle CEP is running on host `ariel`, listening at port 9002, username and password of the administrator is `wlevs/wlevs`, and your application JAR file is called `myapp_1.0.0.0.jar` and is located in the `/applications` directory, then the command is:

```
prompt> java -jar wlevsdeploy.jar -url http://ariel:9002/wlevsdeployer  
-user wlevs -password wlevs -install /applications/myapp_1.0.0.0.jar
```

After the application JAR file has been successfully installed and all initialization tasks completed, Oracle CEP automatically starts the application and the adapter components immediately start listening for incoming events.

The Deployer utility provides additional options to resume, suspend, update, and uninstall an application JAR file, as well as deploy an application to a specified group of a multi-server domain. For details, see [Deployer Command-Line Reference](#).

Oracle CEP uses the `deployments.xml` file to internally maintain its list of deployed application OSGi bundles. This file is located in the `DOMAIN_DIR/servername` directory, where `DOMAIN_DIR` refers to the main domain directory corresponding to the server instance to which you are deploying your application and `servername` refers to the actual server. See [XSD Schema For the Deployment File](#) for information about this file. This information is provided for your information only; Oracle does not recommend updating the `deployments.xml` file manually.



# Using the Load Generator to Test Your Application

This section contains information on the following subjects:

- [“Overview of the Load Generator Utility” on page 11-1](#)
- [“Creating a Load Generator Property File” on page 11-2](#)
- [“Creating a Data Feed File” on page 11-4](#)
- [“Configuring the csvgen Adapter in Your Application” on page 11-4](#)

## Overview of the Load Generator Utility

The load generator is a simple utility provided by Oracle Complex Event Processing (or *Oracle CEP* for short ) to simulate a data feed. The utility is useful for testing the EPL rules in your application without needing to connect to a real-world data feed.

The load generator reads an ASCII file that contains the sample data feed information and sends each data item to the configured port. The load generator reads items from the sample data file in order and inserts them into the stream, looping around to the beginning of the data file when it reaches the end; this ensures that a continuous stream of data is available, regardless of the number of data items in the file. You can configure the rate of sent data, from the rate at which it starts, the final rate, and how long it takes the load generator to ramp up to the final rate.

In your application, you must use the Oracle CEP-provided `csvgen` adapter, rather than your own adapter, to read the incoming data; this is because the `csvgen` adapter is specifically coded to decipher the data packets generated by the load generator.

To use the load generator, follow these steps:

1. Optionally create a property file that contains configuration properties for particular run of the load generator; these properties specify the location of the file that contains simulated data, the port to which the generator feeds the data, and so on.

Oracle CEP provides a default property file you can use if the default property values are adequate.

See [“Creating a Load Generator Property File” on page 11-2](#).

2. Create a file that contains the actual data feed values.

See [“Creating a Data Feed File” on page 11-4](#).

3. Configure the `csvgen` adapter so that it correctly reads the data feed generated by the load generator. You configure the adapter in the EPN assembly file that describes your Oracle CEP application.

See [“Configuring the csvgen Adapter in Your Application” on page 11-4](#).

4. Be sure that you configure a builder factory for creating your event types. Although specifying event type builder factories is typically an optional task, it is required when using the load generator.

See [“Creating the Event Types” on page 2-15](#) for details.

5. Open a new command window and set your environment as described in [Setting Up Your Development Environment](#).

6. Change to the `ORACLE_CEP_HOME\ocep_10.3\utils\load-generator` directory, where `ORACLE_CEP_HOME` refers to the main Oracle CEP installation directory, such as `d:\oracle_cep`.

7. Run the load generator specifying the properties file you created in step 1 to begin the simulated data feed. For example, if the name of your properties file is `c:\loadgen\myDataFeed.prop`, execute the following command:

```
prompt> runloadgen.cmd c:\loadgen\myDataFeed.prop
```

If you redploy your application, you must also restart the load generator.

## Creating a Load Generator Property File

The load generator uses an ASCII properties file for its configuration purposes. Properties include the location of the file that contains the sample data feed values, the port to which the utility should send the data feed, and so on.



Oracle CEP provides a default properties file called `csvgen.prop`, located in the `ORACLE_CEP_HOME\ocep_10.3\utils\load-generator` directory, where `ORACLE_CEP_HOME` refers to the main Oracle CEP installation directory, such as `d:\oracle_cep`.

The format of the file is simple: each property-value pair is on its own line. The following example shows the default `csvgen.prop` file; Oracle recommends you use this file as template for your own property file:

```
test.csvDataFile=test.csv
test.port=9001
test.packetType=CSV
test.mode=client
test.senders=1
test.latencyStats=false
test.statInterval=2000
```

**WARNING:** If you create your own properties file, you must include the `test.packetType`, `test.mode`, `test.senders`, `test.latencyStats`, and `test.statInterval` properties exactly as shown above.

In the preceding sample properties file, the file that contains the sample data is called `test.csv` and is located in the same directory as the properties file. The load generator will send the data feed to port 9001.

The following table lists the additional properties you can set in your properties file.

**Table 11-1 Load Generator Properties**

Property	Description	Data Type	Required?
<code>test.csvDataFile</code>	Specifies the file that contains the data feed values.	String	Yes.
<code>test.port</code>	The port number to which the load generator should send the data feed.	Integer	Yes.
<code>test.secs</code>	Total duration of the load generator run, in seconds. The default value is 30.	Integer	No.
<code>test.rate</code>	Final data rate, in messages per second. The default value is 1.	Integer	No.

**Table 11-1 Load Generator Properties**

Property	Description	Data Type	Required?
<code>test.startRate</code>	Initial data rate, in messages per second. The default value is 1.	Integer	No.
<code>test.rampUpSecs</code>	Number of seconds to ramp up from <code>test.startRate</code> to <code>test.rate</code> . The default value is 0.	Integer	No.

## Creating a Data Feed File

The file that contains the sample data feed values correspond to the event type registered for your Oracle CEP application. The file follows a simple format:

- Each item of a particular data feed is on its own line.
- Separate the fields of a data feed item with commas.
- Do not include extraneous spaces before or after the commas, unless the space is literally part of the field value.
- Include only string and numerical (integer, long, double, float, etc) data in a data feed file.

The following example shows a sample data feed file where each item corresponds to a person with name, age, and birthplace fields:

```
Lucy,23,Madagascar
Nick,44,Canada
Amanda,12,Malaysia
Juliet,43,Spain
Horatio,80,Argentina
```

## Configuring the csvgen Adapter in Your Application

You must use the `csvgen` adapter in your application because this Oracle CEP-provided adapter is specifically coded to read the data packets generated by the load generator.

You register the `csvgen` adapter using the `<wlevs:adapter>` tag in the EPN assembly file of your application, as with all adapters. Use the `provider="csvgen"` attribute to specify that the

provider is the csvgen adapter, rather than your own adapter. Additionally, you must specify the following child tags:

- `<wlevs:instance-property name="port" value=configured_port>`, where *configured\_port* corresponds to the value of the `test.port` property in the load generator property file. See [“Creating a Load Generator Property File” on page 11-2](#).
- `<wlevs:instance-property name="eventName" value=event_type_name>`, where *event\_type\_name* corresponds to the name of the event type that represents an item from the load-generated feed.
- `<wlevs:instance-property name="eventPropertyNames" value=ordered_list_of_properties>`, where *ordered\_list\_of\_properties* lists the names of the properties in the order that the load generator sends them, and consequently the csvgen adapter receives them.

Before showing an example of how to configure the adapter, first assume that your application registers an event type called `PersonType` in the EPN assembly file using the `<wlevs:metada>` method as shown:

```
<wlevs:event-type-repository>
  <wlevs:event-type type-name="PersonType">
    <wlevs:metadata>
      <entry key="name" value="java.lang.String"/>
      <entry key="age" value="java.lang.Integer"/>
      <entry key="birthplace" value="java.lang.String"/>
    </wlevs:metadata>
  </wlevs:event-type>
</wlevs:event-type-repository>
```

This event type corresponds to the data feed file shown in [“Creating a Data Feed File” on page 11-4](#).

To configure the csvgen adapter that receives this data, use the following `<wlevs:adapter>` tag:

```
<wlevs:adapter id="csvgenAdapter" provider="csvgen">
  <wlevs:instance-property name="port" value="9001"/>
  <wlevs:instance-property name="eventName" value="PersonType"/>
  <wlevs:instance-property name="eventPropertyNames"
    value="name,age,birthplace"/>
</wlevs:adapter>
```

Note how the bolded values in the adapter configuration example correspond to the `PersonType` event type registration.

If you use `<wlevs:class>` to specify your own JavaBean when registering the event type, then the `eventPropertyNames` value corresponds to the JavaBean properties. For example, if your JavaBean has a `getName()` method, then one of the properties of your JavaBean is `name`.

# Additional Information about Spring and OSGi

Oracle Complex Event Processing applications are built on top of the Spring Framework and OSGi Service Platform. Therefore, it is assumed that you are familiar with these technologies and how to program within the frameworks.

For additional information about Spring and OSGi, see:

- [Spring Framework API 2.5](#)
- [The Spring Framework - Reference Documentation](#) (from Interface21)
- [Spring-OSGi Project](#)
- [OSGi Service Platform Javadoc \(Release 4\)](#)
- [OSGi Release 4 Core Specification](#)

