Oracle® Complex Event Processing

Application Development Guide

Release 3.0

July 2008

Alpha/Beta Draft

ORACLE®

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 2-5

Oracle CEP Development Environment for Eclipse 2-6

Creating Oracle CEP Applications: Typical Steps 2-7

Creating the EPN Assembly File 2-8

Creating the Event Types 2-12

Next Steps 2-14

3. Creating Custom Adapters and Event Beans

Overview of Adapters and Event Beans 3-1

Creating Adapters or Event Beans: Typical Steps 3-2
Programming the Adapter Class: Guidelines 3-3
Programming the Adapter Factory Class 3-7
Updating the EPN Assembly File 3-8
    Registering the Adapter Factory as an OSGI Service 3-8
    Declaring the Adapter Components in your Application. 3-9
Configuring the Adapter 3-9
    Example of an Adapter Configuration File 3-11
Creating an Adapter in Its Own Bundle 3-11
Extending the Configuration of an Adapter 3-13
    Creating the XSD Schema File 3-15
    Complete Example of an Extended XSD Schema File 3-17
    Programming Access to the Configuration of an Adapter. 3-18
Passing Login Credentials from an Adapter to the Data Feed Provider 3-19
    Updating the Adapter Code to Access the Login Credential Properties 3-22

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 4-2
    Updating the EPN Assembly File With JMS Adapters 4-4
    Configuring the JMS Adapters 4-7

5. Using and Creating HTTP Publish-Subscribe Adapters

Overview of HTTP Publish-Subscribe Functionality in Oracle CEP 5-1
    Overview of the Built-In Pub-Sub Adapter for Local Publishing 5-2
    Overview of the Built-In Pub-Sub Adapter for Remote Publishing 5-4
    Overview of the Built-In Pub-Sub Adapter for Subscribing 5-6

Using the Built-In HTTP Pub-Sub Adapters in an Application: Typical Steps 5-8

Configuring the HTTP Pub-Sub Adapters 5-8

Updating the EPN Assembly File. 5-12

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 Processer Configuration File 7-1

Configuring the Complex Event Processor: Main Steps. 7-2

Example of a Processor Configuration File. 7-5

8. Programming the Business Logic Component

Overview of Programming the Business Logic Component. 8-1

Programming Business Logic: Guidelines 8-1

Accessing a Relational Database. 8-3

9. Using Oracle CEP Caching

Overview of Oracle Complex Event Processing Caching. 9-1

Use Case: Publishing Events to a Cache 9-2

Use Case: Consuming Data From a Cache. 9-2

Use Case: Updating and Deleting Data in a Cache 9-3

Additional Caching Features 9-3

Caching APIs 9-3

Typical Steps to Use Oracle CEP Caching 9-4

Configuring the Oracle CEP Caching System and Caches 9-5

Declaring the Caching System in the EPN Assembly File. 9-9

Declaring a Cache in the EPN Assembly File 9-10

Configuring and Programming a Business POJO to Access a Cache 9-11

Configuring and Programming an Adapter to Access a Cache 9-12

Configuring and Programming a User-Defined EPL Function to Access a Cache . 9-13

Configuring a Cache as a Listener 9-14

Specifying the Key Used to Index a Cache 9-15

Configuring a Cache as an Event Source 9-15

Configuring a Cache Loader 9-16

Configuring a Cache Store 9-17

Referencing a Cache from an EPL Statement 9-17

Restrictions When Using a Cache in an EPL Statement 9-18

Typical Steps To Reference a Cache in an EPL Statement 9-18

10. Storing Events in a Persistent Store

Overview of Storing Events in a Persistent Store 10-1

Use Case: Storing Events 10-1

Use Case: Playing Back Events 10-1

Use Case: Querying Stored Events 10-1

Additional Persistent Store Features 10-2

Implementing Persistent Event Storage for Your Application: Typical Steps 10-2

Deciding Which Components Will Store Events 10-2

Updating Your EPN Assembly File 10-2

Programming the Persistent Event Storage 10-2

Programming the Event Querying 10-2

Configuring the Persistent Storage. 10-2

Example: Storing Events for Your Application 10-2

11. Assembling and Deploying Oracle Complex Event Processing Applications

Overview of Application Assembly and Deployment 11-1

Assembling an Oracle CEP Application: Main Steps 11-2

Creating the MANIFEST.MF File 11-4

Accessing Third-Party JAR Files From Your Application 11-6

Deploying Oracle CEP Applications: Main Steps 11-7

12. Using the Load Generator to Test Your Application

Overview of the Load Generator Utility 12-1

Creating a Load Generator Property File 12-2

Creating a Data Feed File 12-4

Configuring the csvgen Adapter in Your Application 12-4

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*

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, "Programming the Business Logic Component," describes how to program the business logic POJO component of an Oracle CEP application.
- Chapter 9, "Using Oracle CEP Caching," describes how to use the event caching feature to increase the performance of your applications.
- Chapter 10, "Storing Events in a Persistent Store," describes how to record events to a physical store, and then later play them back.
- Chapter 11, "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 12, "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.

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 three 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 `WLEVS_HOME\samples\domains\helloworld_domain`, where `WLEVS_HOME` refers to the top-level Oracle CEP directory, such as `c:\beahome\wlevs30`.

The HelloWorld Java source code is located in `WLEVS_HOME\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 `WLEVS_HOME\samples\domains\fx_domain`, where `WLEVS_HOME` refers to the top-level Oracle CEP directory, such as `c:\beahome\wlevs30`.

  The ForeignExchange Java source code is located in `WLEVS_HOME\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 `WLEVS_HOME\samples\domains\signalgeneration_domain`, where `WLEVS_HOME` refers to the top-level Oracle CEP directory, such as `c:\beahome\wlevs30`.

  The Signal Generation Java source code is located in `WLEVS_HOME\samples\source\applications\signalgeneration`.

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 2-6
- "Creating Oracle CEP Applications: Typical Steps" on page 2-7
- "Creating the EPN Assembly File" on page 2-8
- "Creating the Event Types" on page 2-12
- "Next Steps" on page 2-14

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 2-5

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).

- Business Logic POJO—User-coded POJO that receives events from the complex event processor, after the EPL rules have fired. This is a standard Spring bean. These components cannot be 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

No documentation available for Beta.

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

No documentation available for Beta.

Oracle CEP APIs

Oracle CEP provides a variety of Java APIs that you use in your adapter implementation or business logic POJO. These APIs are all packaged in the `com.bea.wlevs.api` package.

This section describes the APIs that you will most typically use in your adapters and POJOs; see the Javadoc for the full reference documentation for all classes and interfaces. See "Creating Custom Adapters and Event Beans" on page 3-1 and "Programming the Business Logic Component" on page 8-1, as well as the HelloWorld and FX examples in the installed product, for sample Java code that uses these APIs.

- `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.
- `EventSender`—The interface that actually sends the events 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.

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* is a set of plugins for the Eclipse IDE designed to help develop, deploy, and debug applications for Oracle CEP 3.0.

The key features of this 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 this IDE, Oracle recommends that you give it a try. For details, see Oracle CEP Development Environment for Eclipse.

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 2-6.

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 2-8.
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 2-12.
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. Program the business object POJO that receives the set of events that were selected with the EPL query and contains the application business logic.

   See "Programming the Business Logic Component" on page 8-1.

9. 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 9-1.

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 12-1.

See "Next Steps" on page 2-14 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.HelloWorldAdapterFactor
   y" />
   </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 2-12 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 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 and let Oracle CEP automatically create JavaBean instances for you; this method is best used for quick prototyping.

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. If an application (the provider) wants to share its classes, the provider must explicitly export the classes in its `MANIFEST.MF` file, and the consumer of the classes must import them. For details, see "Assembling an Oracle CEP Application: Main Steps" on page 11-2.

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 the Spring bean that acts as a builder factory for the event type, if you have programmed a factory. 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`.

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
```

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 11-2.

- 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 11-7.

- 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 12-1.

Creating Custom Adapters and Event Beans

This section contains information on the following subjects:

- "Overview of Adapters and Event Beans" on page 3-1
- "Creating Adapters or Event Beans: Typical Steps" on page 3-2
- "Programming the Adapter Class: Guidelines" on page 3-3
- "Programming the Adapter Factory Class" on page 3-7
- "Updating the EPN Assembly File" on page 3-8
- "Configuring the Adapter" on page 3-9
- "Creating an Adapter in Its Own Bundle" on page 3-11
- "Extending the Configuration of an Adapter" on page 3-13
- "Passing Login Credentials from an Adapter to the Data Feed Provider" on page 3-19

Overview of Adapters and Event Beans

The role 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.

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 must also program adapter factories that create instances of the adapters. Finally, you must register both the adapter classes, and the adapter factories, 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.

Creating Adapters or Event Beans: Typical Steps

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

1. Program the adapter Java class.

   See “Programming the Adapter Class: Guidelines” on page 3-3.

2. Program the adapter factory class.

   See “Programming the Adapter Factory Class” on page 3-7.

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

   See “Updating the EPN Assembly File” on page 3-8

4. Optionally change the default configuration of the adapter.

   See “Configuring the Adapter” on page 3-9.

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

   See “Extending the Configuration of an Adapter” on page 3-13.

In the preceding procedure, it is assumed that the adapter 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 in its own JAR file so that it can be shared among many applications, see “Creating an Adapter in Its Own Bundle” on page 3-11.

Programming the Adapter Class: Guidelines

The adapter class 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.text.DateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import com.bea.wlevs.configuration.Activate;
import com.bea.wlevs.configuration.Prepare;
import com.bea.wlevs.configuration.Rollback;
import com.bea.wlevs.ede.api.Adapter;
import com.bea.wlevs.ede.api.EventSender;
import com.bea.wlevs.ede.api.EventSource;
import com.bea.wlevs.ede.api.SuspendableBean;
import com.bea.wlevs.event.example.helloworld.HelloWorldEvent;
import com.bea.wlevs.adapter.example.helloworld.HelloWorldAdapterConfig;

public class HelloWorldAdapter implements Runnable, Adapter, EventSource,
SuspendableBean {

    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;
    }

    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);
    }

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

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


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

    public synchronized void suspend() throws Exception {
        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 required interfaces and classes of the Oracle CEP API:

  ```
  import com.bea.wlevs.ede.api.Adapter;
  import com.bea.wlevs.ede.api.EventSender;
  import com.bea.wlevs.ede.api.EventSource;
  import com.bea.wlevs.ede.api.SuspendableBean;
  ```

  Your adapter is required to implement the `Adapter` and `EventSource` interfaces; typically, your adapter will also implement the `java.lang.Runnable` and `SuspendableBean` interfaces to control the starting and stopping of the adapter. 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 and adapter configuration:

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

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

  The `com.bea.wlevs.adapter.example.helloworld.HelloWorldAdapterConfig` class represents an intance of the runtime adapter configuration. For details, see "Programming Access to the Configuration of an Adapter" on page 3-18.

- Optionally import the metadata annotations that allow you to access configuration information about your adapter once the application is deployed to Oracle CEP:

  ```
  import com.bea.wlevs.configuration.Activate;
  import com.bea.wlevs.configuration.Prepare;
  import com.bea.wlevs.configuration.Rollback;
  ```

  See "Programming Access to the Configuration of an Adapter" on page 3-18 for details.

- The adapter class must implement the `Adapter` and `EventSource` interfaces. Typically, your adapter will also implement the `SuspendableBean` and `java.lang.Runnable` interfaces:

  ```
    public class HelloWorldAdapter implements Runnable, Adapter,
  EventSource, SuspendableBean {
  ```

  The `Adapter` interface specifies that your Java class is an adapter. The `EventSource` interface provides the `EventSender` that you use to send events.

  The `Runnable` and `SuspendableBean` interfaces enable Oracle CEP to manage the running of the adapter code. An adapter that implements `Runnable` should always also implement `SuspendableBean`. In the `SuspendableBean.suspend()` method, you should put whatever code is needed to stop the running of your adapter. For example, you may set a flag that is checked by the main loop of the `Runnable.run()` method which will cause the loop to be exited. You must implement `SuspendableBean` in order for your application to be properly stopped when it is undeployed.

- If, as is typical, your adapter implements the `java.lang.Runnable` 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 11-6 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 2-12 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.

- The methods annotated with the `@Prepare`, `@Activate`, and `@Rollback` annotations specify the methods that Oracle CEP invokes when the server prepares, activates, or rolls back the adapter's configuration. For details, see "Programming Access to the Configuration of an Adapter" on page 3-18.

- 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 the Adapter 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.

The following is the 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 AdapterFactory {

    public HelloWorldAdapterFactory() {
    }
```

```
    public synchronized Adapter create() throws IllegalArgumentException {
        return new HelloWorldAdapter();
    }
}
```

For full details of these APIs, see the Javadoc

Updating the EPN Assembly File

The adapters and adapter factory must be registered in the EPN assembly file, as discussed in the following sections. If you are using JMS, additional configuration is also required.

For a complete description of the configuration file, including registration of other components of your application, see "Creating the EPN Assembly File" on page 2-8.

Registering the Adapter Factory as an OSGI Service

The adapter factory must be registered 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 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. You can declare one or more adapters in your network. 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

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 2-8 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 in Its Own Bundle

In the procedure described in "Creating Adapters or Event Beans: Typical Steps" on page 3-2, 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 Factory as an OSGI Service" on page 3-8.
  - If you are also including the event type in the bundle, register it as described in "Creating the Event Types" on page 2-12.
  - 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" on page 3-9.
- 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 11-4.
- Assemble and deploy the `GlobalAdapter` bundle in the usual way, as described in "Assembling and Deploying Oracle Complex Event Processing Applications" on page 11-1.
- 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 Components in your Application" on page 3-9. 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 11-4.

Extending the Configuration of an Adapter

Adapters have default configuration data, as described in "Configuring the Adapter" on page 3-9 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 3-15 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.

   For example, the HelloWorld sample `build.xml` file includes the following (only relevant sections shown):

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

```
<path id="classpath">
        <pathelement location="${output.dir}" />
        <fileset dir="${sharedlib.dir}">
                <include name="*.jar" />
        </fileset>
        <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 11-2 for details.

5. Program your adapter as described in "Programming the Adapter Class: Guidelines" on page 3-3. Within your adapter code, you access the extended configuration as usual, as described in "Programming Access to the Configuration of an Adapter" on page 3-18.

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, in the HelloWorld configuration file:

```
<?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 from the HelloWorld example 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 3-17 for the HelloWorld 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 then 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 Processer 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

The following extended XSD file is used in the HelloWorld sample application:

```
<?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

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 date 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, because the HelloWorld sample extends the configuration of its adapter, 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 `java` command to encrypt the value of the `<password>` element in the EPN assembly file:

   ```
   prompt> java -jar
   BEA_HOME/modules/com.bea.core.bootbundle_3.0.1.0.jar .
   epn_assembly_file
   ```

   where *`BEA_HOME`* refers to the main directory into which you installed Oracle CEP, such as `d:\beahome`. 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.

   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" on page 3-13 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 `java` command to encrypt the value of the `<password>` element in the adapter configuration file:

```
prompt> java -jar
BEA_HOME/modules/com.bea.core.bootbundle_3.0.1.0.jar .
adapter_config_file
```

where *`BEA_HOME`* refers to the main directory into which you installed Oracle CEP, such as `d:\beahome`. 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 file.

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 3-22.

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 11-4.

6. Re-assemble and deploy your application as usual. See "Assembling and Deploying Oracle Complex Event Processing Applications" on page 11-1.

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 the Adapter Class: Guidelines" on page 3-3.

- 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" on page 3-18 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 map messages from a JMS queue and automatically converts them into events by matching property names with a specified event type. You can optionally 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. You can optionally 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.

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

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.

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 assumed in this section that you have already created an Oracle CEP application along with its EPN assembly file and and component configuration files, and that you want to update the application to use an inbound or outbound JMS adatper. 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 4-2.

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.

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,
JMSException;
```

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, JMSException;
```

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 OutboundMessageConvert; 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 automaticallys 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 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 converstion 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 and sources 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" />

     <!-- 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 Adpaters

Child Element	Description
event-type	Event type whose properties match the JMS message properties. Specify this child element *only* if you want Oracle CEP to automatically perform the conversion between JMS messages and events. If you have created your own custom convert 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 `weblogic.jndi.WLInitialContextFactory`, for Oracle WebLogic Server JMS.
connection-jndi-name	Optional. The JNDI name of the JMS connection factory. Default value is `weblogic.jms.ConnectionFactory`, 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.

Table 4-1 Child Elements of <jms-adapter> for Inbound and Outbound Adpaters

Child Element	Description
password	Optional The password for the external resource.
encrypted-password	Optional. The encrypted password for the external resource.

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 `config.xml` file. This name corresponds to the value of the `<name>` child element of the `<work-manager>` element in `config.xml`.
concurrent-consumers	Number of consumers to create.
message-selector	JMS message selector to use to filter messages.
session-ack-mode-name	Use 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 whether the delivery mode: persistent (default value) or nonpersistent.

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"?>
<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">

    <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>

</n1: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>
```

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 5-8

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 of 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 5-2
- "Overview of the Built-In Pub-Sub Adapter for Remote Publishing" on page 5-4
- "Overview of the Built-In Pub-Sub Adapter for Subscribing" on page 5-6

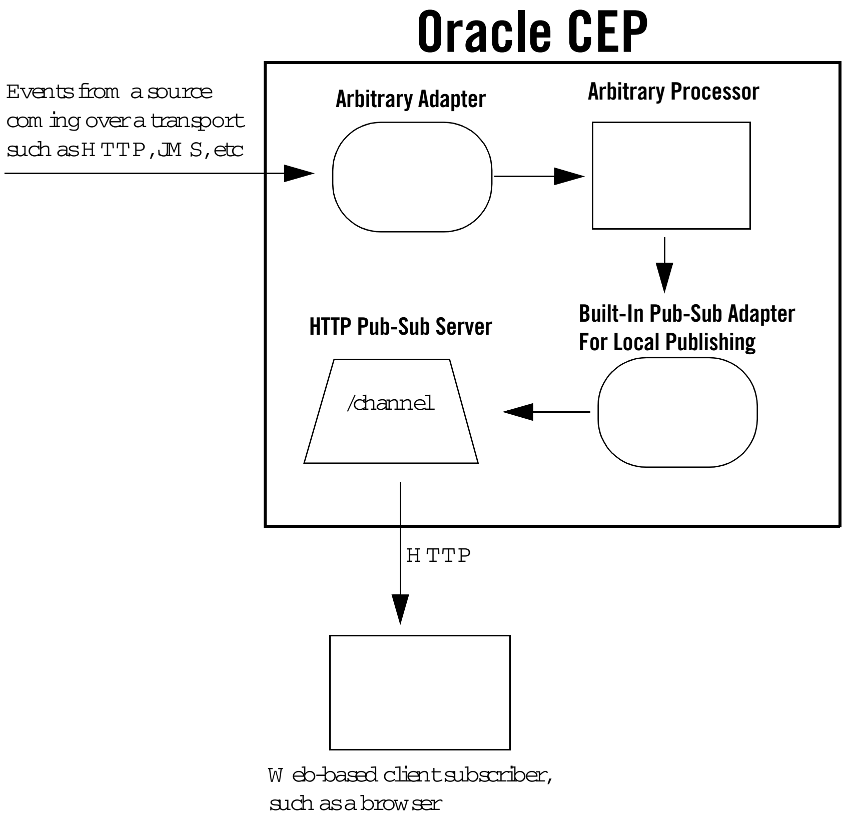
The pub-sub server can communicate with any client that can understand the Bayeux protocol. Programmers developer their Web clients using one of the following frameworks:

- 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.

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


Oracle CEP
Events from a source coming over a transport such as HTTP, JMS, etc
Arbitrary Adapter
Arbitrary Processor
HTTP Pub-Sub Server
Built-In Pub-Sub Adapter For Local Publishing
/channel
HTTP
Web-based client subscriber, such as a browser

In the preceding graphic:

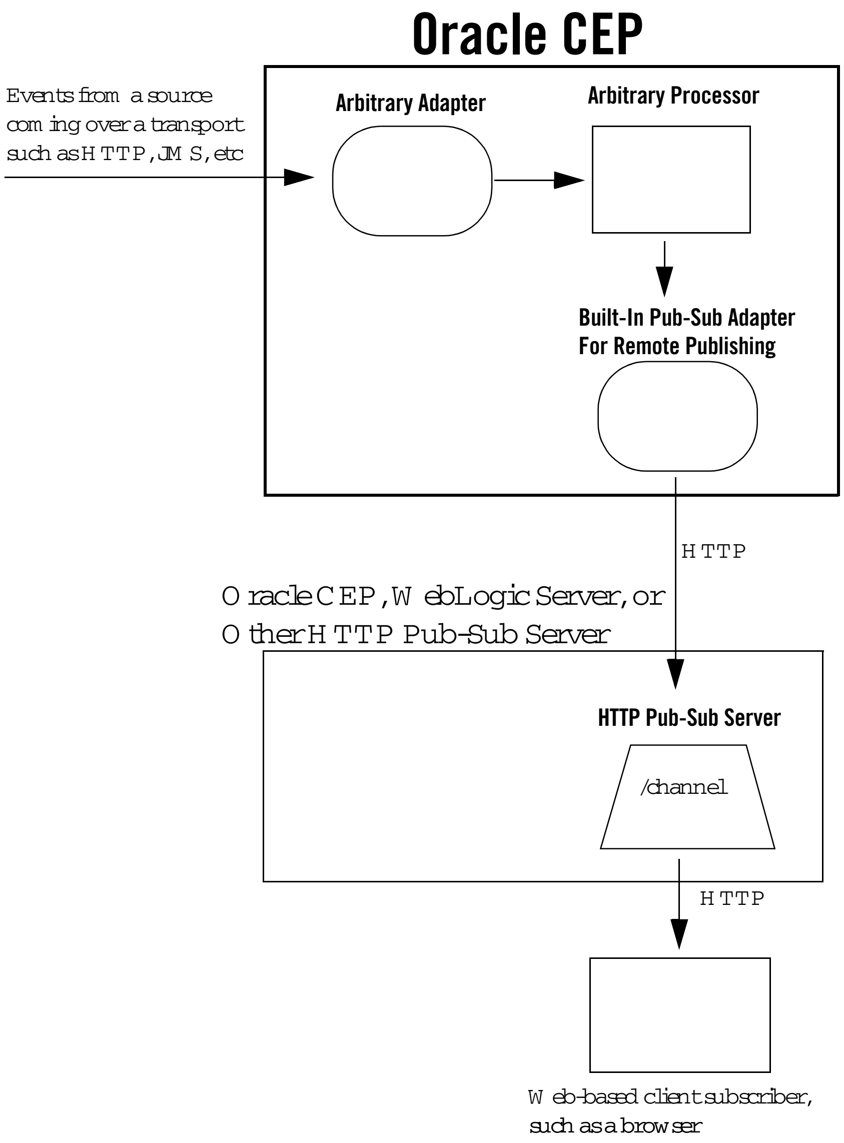
- Events flow from some source into an adapter of an application running in Oracle CEP. This adapter is not required, it is shown only as an example.
- The events flow from the adapter to an arbitrary processor; again, this processor is not required.
- The processor sends the events to the built-in pub-sub 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


Oracle CEP
Events from a source coming over a transport such as HTTP, JMS, etc
Arbitrary Adapter
Arbitrary Processor
Built-In Pub-Sub Adapter For Remote Publishing
HTTP
Oracle CEP, WebLogic Server, or Other HTTP Pub-Sub Server
HTTP Pub-Sub Server
/channel
HTTP
Web-based client subscriber, such as a browser

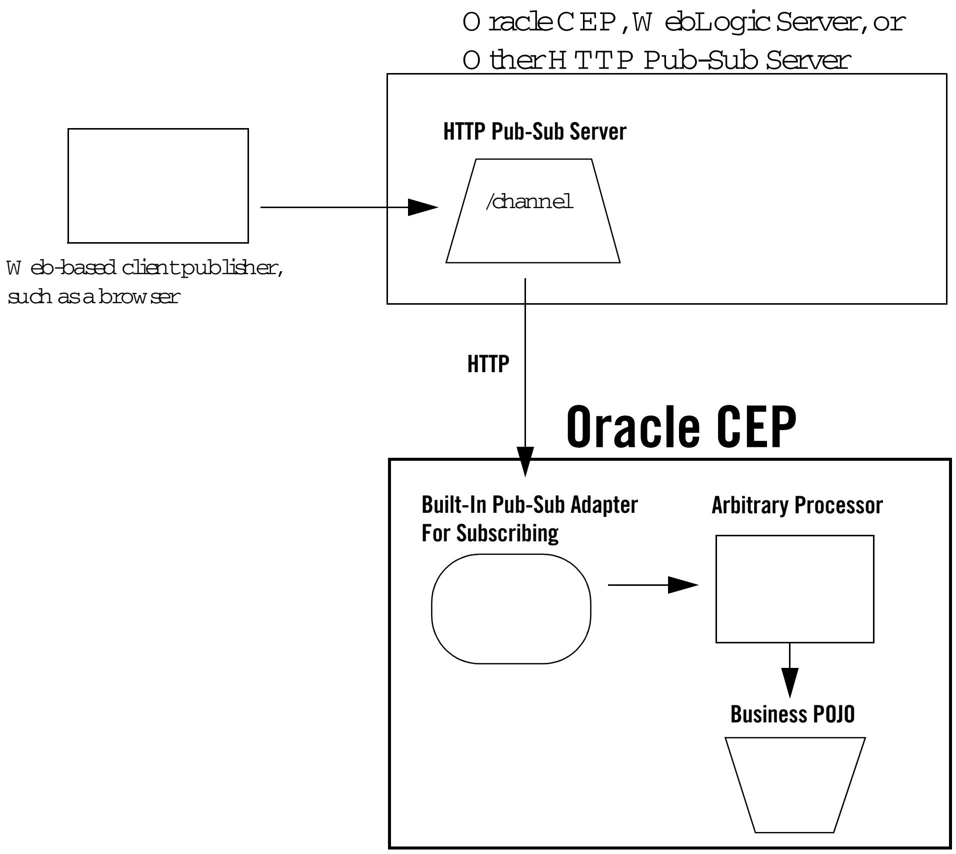
In the preceding graphic:

- 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) or by the HTTP pub-sub server itself.
- 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 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. 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.

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

   See "Configuring the HTTP Pub-Sub Adapters" on page 5-8.

3. 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 5-12.

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>

</n1: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 ant 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"?>

<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">

    <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>

</n1: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 and sources 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 both local and remote publishing); 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="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 5-8), `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="pubsubCounterBean"/>
    </wlevs:adapter>

    <bean id="pubsubCounterBean"
          class="com.mycompany.httppubsub.EventCounter">
        <property name="expectedEvents" value="12"/>
    </bean>

    <wlevs:stream id="pubsubStream" manageable="true">
        <wlevs:listener>
            <bean id="mySink"
                  class="com.mycompany.httppubsub.MySink"/>
        </wlevs:listener>
        <wlevs:source ref="pubsubCounterBean"/>
    </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 5-8), `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 `pubsubCounterBean`.

- The `pubsubCounterBean` processes the event as described by the `com.mycompany.httppubsub.EventCounter` 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 2-8 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 Processer 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 Processer 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 2-8 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 joing 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"?>

<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>myProcessor</name>
    <rules>
      <rule id="myRule"><![CDATA[

      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

      ]]></rule>
    </rules>
  </processor>

</n1: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.

Programming the Business Logic Component

This section contains information on the following subjects:

- "Overview of Programming the Business Logic Component" on page 8-1
- "Programming Business Logic: Guidelines" on page 8-1
- "Accessing a Relational Database" on page 8-3

Overview of Programming the Business Logic Component

The business logic component is typically the last component in your event network, the one that receives results from the EPL queries associated with the processor components. This is also the component in which you program your application business code. For example, the business logic component might publish the events to a Web site, pass the events on to a legacy application, and so on.

This component is a plain old Java object, or POJO. Programming the component is very simple with few required guidelines. You can also use the JDBC API to access data in a relational database, as described in "Accessing a Relational Database" on page 8-3

Programming Business Logic: Guidelines

The simplest way to describe the guidelines to programming the business POJO is to show an example.

The following sample code shows the business logic POJO for the HelloWorld application; 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 {

    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 POJO must import the event type of the application, which in the HelloWorld case is `HelloWorldEvent`:

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

- Your POJO 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 POJO received from the processor, typically via a stream:

  ```
      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 2-8 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.

Accessing a Relational Database

You can use the Java Database Connectivity (JDBC) APIs in your business logic POJO to access data contained in a relational database. Oracle CEP supports JDBC 3.0.

Follow these steps to use JDBC in your business logic POJO:

1. Configure JDBC for Oracle CEP.

   For details, see Configuring Access to a Relational Database.

2. In your business logic POJO 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.

Using Oracle CEP Caching

This section contains information on the following subjects:

- "Overview of Oracle Complex Event Processing Caching" on page 9-1
- "Typical Steps to Use Oracle CEP Caching" on page 9-4
- "Referencing a Cache from an EPL Statement" on page 9-17

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 stores to disk, then you can plug-in different cache providers, such as 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 can use the caching system, but they must be deployed after the application that owns the caching system to ensure that the caching system is made available to them.

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 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 asynchonous 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 the caching API to inject the cache resource into the implementation of the function. Applications can also query a cache directly from an EPL statement that both runs in a processor or is submitted programmatically to the cache. In this case, the cache essentialy functions as another type of stream data source to a processor so that querying a cache is very similar to querying a stream.

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 and access its runtime statistics using JMX.
- 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.api`—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 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 9-5.

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

   See "Declaring the Caching System in the EPN Assembly File" on page 9-9.

3. Declare the caches in the EPN assembly file.

   "Declaring a Cache in the EPN Assembly File" on page 9-10

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 9-11

   – "Configuring and Programming an Adapter to Access a Cache" on page 9-12

   – "Configuring and Programming a User-Defined EPL Function to Access a Cache" on page 9-13

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 9-14.

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 9-15.

7. Optionally configure a cache loader or cache store for a cache. See:
   – "Configuring a Cache Loader" on page 9-16
   – "Configuring a Cache Store" on page 9-17
8. Optionally reference the cache in an EPL statement. See "Referencing a Cache from an EPL Statement" on page 9-17.

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.

If you use a third-party caching implementations such as GemFire, 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 9-9 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"?>

<helloworld:config

xmlns:helloworld="http://www.bea.com/xml/ns/wlevs/example/helloworld">
  <processor>
```

```
      ...
    </processor>

    <adapter>
      ...
    </adapter

    <caching-system>
       <name>caching-system-id</name>
    </caching-system>

</helloworld: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 entires 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-attemps>` : 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, futher 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.

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.api.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 implemenation 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>
```

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 may *not* export the cache as an OSGI service using the `advertise` attribute; only caching systems can be advertised.

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.

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 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.

The first option is to allow you to specify a property name for the key property when a cache is declared, 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.

The second option is to use the metada annotation `com.bea.wlevs.ede.api.Key` to annotate the event property in the Java class that implements the event type.

The third option is to not specify a key at all. In this case the event object itself serves as both the key and value when the event is inserted into the cache. The event class must include a valid implementation of the `equals` and `hashcode` methods that take into account the values of the key properties.

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 processor, 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: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.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: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: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: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: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: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: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 all data in the cache to a backing store.

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 9-15 for instructions on specifying the cache key.
- Joins must be executed only 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 9-4 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 2-12.
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 9-15 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 `<wlevs:value-type>` child element. For 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:value-type name ="Company"/>
</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:source>` child element that references the cache. For example:

```
<wlevs:stream id="stream-id"/>

<wlevs:processor id="processor-id">
  <wlevs: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.

Storing Events in a Persistent Store

This section contains information on the following subjects:

- "Overview of Storing Events in a Persistent Store" on page 10-1
- "Implementing Persistent Event Storage for Your Application: Typical Steps" on page 10-2
- "Example: Storing Events for Your Application" on page 10-2

Overview of Storing Events in a Persistent Store

No documentation available for Beta.

Use Case: Storing Events

No documentation available for Beta.

Use Case: Playing Back Events

No documentation available for Beta.

Use Case: Querying Stored Events

No documentation available for Beta.

Additional Persistent Store Features

No documentation available for Beta.

Implementing Persistent Event Storage for Your Application: Typical Steps

No documentation available for Beta.

Deciding Which Components Will Store Events

Updating Your EPN Assembly File

Programming the Persistent Event Storage

Programming the Event Querying

Configuring the Persistent Storage

Example: Storing Events for Your Application

Assembling and Deploying Oracle Complex Event Processing Applications

This section contains information on the following subjects:

- "Overview of Application Assembly and Deployment" on page 11-1
- "Assembling an Oracle CEP Application: Main Steps" on page 11-2
- "Deploying Oracle CEP Applications: Main Steps" on page 11-7

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 11-2 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 11-7.

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 *`WLEVS_HOME`*`\samples\source\applications\helloworld`, where *`WLEVS_HOME`* refers to the main installation directory, such as `d:\beahome\wlevs30`.

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 2-8 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 11-4.
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 11-6.
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-separate list of the JAR file path names that should be searched for classes and resources. Use a period (`.`) to specify the bundle itself. For example:

```
Bundle-Classpath: ., commons-logging.jar, myExcitingJar.jar,
myOtherExcitingJar.jar
```

If you need to access native libraries, you must also package them in your JAR file and use the `Bundle-NativeCode` header of the `MANIFEST.MF` file to specify their location in the JAR.

- 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 main domain directory. The out-of-the-box sample domains are located in *`WLEVS_HOME`*`/samples/domains`, and the user domains are located in *`BEA_HOME`*`/user_projects/domains`, where *`WLEVS_HOME`* refers to the main Oracle CEP installation directory, such as `d:\beahome\wlevs30`, and *`BEA_HOME`* refers to the directory above *`WLEVS_HOME`*, such as `d:\beahome`.

  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 11-2.

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 *`WLEVS_HOME`*`/bin` directory where, *`WLEVS_HOME`* refers to the main Oracle CEP installation directory, such as `/beahome/wlevs30`.

   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 11-2.

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. 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 12-1
- "Creating a Load Generator Property File" on page 12-2
- "Creating a Data Feed File" on page 12-4
- "Configuring the csvgen Adapter in Your Application" on page 12-4

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 12-2.

2. Create a file that contains the actual data feed values.

   See "Creating a Data Feed File" on page 12-4.

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 12-4.

4. Open a new command window and set your environment as described in Setting Up Your Development Environment.

5. Change to the *`WLEVS_HOME`*`\utils\load-generator` directory, where *`WLEVS_HOME`* refers to the main Oracle CEP installation directory, such as `d:\beahome\wlevs30`.

6. 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 *`WLEVS_HOME`*`\utils\load-generator` directory, where *`WLEVS_HOME`* refers to the main Oracle CEP installation directory, such as `d:\beahome\wlevs30`.

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 12-1 Load Generator Properties

Property	Description	Data Type	Required?
`test.csvDataFile`	Specifies the file that contains the data feed values.	String	Yes.
`test.port`	The port number to which the load generator should send the data feed.	Integer	Yes.
`test.secs`	Total duration of the load generator run, in seconds. The default value is 30.	Integer	No.
`test.rate`	Final data rate, in messages per second. The default value is 1.	Integer	No.
`test.startRate`	Initial data rate, in messages per second. The default value is 1.	Integer	No.
`test.rampUpSecs`	Number of seconds to ramp up from `test.startRate` to `test.rate`. The default value is 0.	Integer	No.

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 12-2.
- `<wlevs:instance-property name="eventTypeName" 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 12-4.

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="eventTypeName" 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