Oracle® Fusion Middleware

Customizing Oracle Event Processing

12*c* Release (12.1.3)

E51156-05

November 2016

How to create customized Oracle Event Processing adapters, event beans, channel event partitioners, event store providers, JMS message converter beans, and HTTP Publish-Subscribe servers.

ORACLE

Oracle Fusion Middleware Customizing Oracle Event Processing, 12*c* Release (12.1.3)

E51156-05

Primary Author: Oracle® Corporation

Contents

Preface

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

Audience

This document is intended for programmers creating Oracle Event Processing applications.

Related Documents

For more information, see the following:

- Known Issues for Oracle SOA Products and Oracle AIA Foundation Pack at: http://www.oracle.com//technetwork/middleware/soasuite/documentation/soaknown-2644661.html.
- *Developing Applications for Oracle Event Processing*
- *Getting Started with Oracle Event Processing*
- *Schema Reference for Oracle Event Processing*
- *Using Visualizer for Oracle Event Processing*
- *Administering Oracle Event Processing*
- *Developing Applications with Oracle CQL Data Cartridges*
- *Oracle CQL Language Reference for Oracle Event Processing*
- *Java API Reference for Oracle Event Processing*
- *Using Oracle Stream Explorer*
- *Getting Started with Oracle Stream Explorer*
- *Oracle Database SQL Language Reference* at: `http://docs.oracle.com/cd/E16655_01/server.121/e17209/toc.htm`
- SQL99 Specifications (ISO/IEC 9075-1:1999, ISO/IEC 9075-2:1999, ISO/IEC 9075-3:1999, and ISO/IEC 9075-4:1999)
- Oracle Event Processing Forum: `http://forums.oracle.com/forums/forum.jspa?forumID=820`

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Custom Adapters

Custom adapters exchange event data with external components that are not supported by the adapters provided in Oracle Event Processing.

See Working with QuickFix Adapter in *Developing Applications for Oracle Event Processing* for information about the adapters provided in Oracle Event Processing.

This chapter includes the following sections:

1.1 Implementation Basics

The type of custom adapter you create depends on the format of the incoming data and the technology you use in the adapter code to convert the incoming data to Oracle Stream Explorer events. Custom adapters typically do the following:

- Use APIs from a data vendor, such as Reuters, Wombat, or Bloomberg.
- Use messaging systems, such as TIBCO Rendezvous.
- Establish a socket connection to the customer's own data protocol.
- Authenticate themselves as needed with the input or output external component.
- Receive raw event data from an external component, convert the data into events, and send the events to the next application node.
- Receive events from within the event processing network and convert the event data to a form that is recognized by the target external component.

To implement a custom adapter, write Java code that communicates with the external component. An input custom adapter implementation receives raw event data from an external component, converts the data into events, and sends the events to the next application node. An output custom adapter implementation receives events from within the event processing network and converts the event data to a form that is recognized by the target external component.

The Oracle Event Processing APIs for creating custom adapters are described in *Java API Reference for Oracle Event Processing*.

The following high-level steps describe how to create a custom adapter. These topics are discussed in more detail later in this chapter.

1. Implement a Java class that communicates with the external component.
2. Implement the interfaces that support sending or receiving event type instances.
   - To send events, implement the custom adapter as an event source.
   - To receive events implement the custom adapter as an event sink.

     See *Java API Reference for Oracle Event Processing* for information about event sources and sinks.
3. If the adapter supports being suspended and resumed, such as when it is undeployed and deployed, implement interfaces to handle these events.
4. Use multithreading to improve application scalability.
5. For any required authentications, write Java logic to pass login credentials to the component that provides or receives the event data.
6. Create a factory class when multiple applications access the custom adapter.
7. Add the adapter to an EPN by configuring it in an assembly file.

1.2 Example Input Adapter Code

The example in this section shows the code for a basic input adapter that retrieves raw event data from a file, converts the data into events, and then sends the events to a downstream stage in the EPN. The code is excerpted from the Oracle Spatial sample application.

Note:

The example code presented here is not production ready, but outlines the basic work flow for an input adapter.

The server calls the following `BusStopAdapter` class methods to provide (inject) the information a `BusStopAdapter` object needs at run time.

- The `BusStopAdapter.setRepositoryInstance` method injects an `EventTypeRepository` instance that contains the event type configurations for this application. The `EventTypeRepository` instance enables the adapter to get information about the event type that was configured for this adapter.
- The `BusStopAdapter.setEventSender` method injects a `StreamSender` object so that `BusStopAdapter` object can send events to the next EPN node.
- The `BusStopAdapter.setPath` and `BusStopAdapter.setEventType` methods provide event property values.
- The `BusStopAdapter.run` method implementation retrieves raw event data, parses the data into event type instances, and sends the new events to the downstream EPN node.

```
package com.oracle.cep.sample.spatial;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import com.bea.wlevs.ede.api.EventProperty;
import com.bea.wlevs.ede.api.EventRejectedException;
import com.bea.wlevs.ede.api.EventType;
import com.bea.wlevs.ede.api.EventTypeRepository;
import com.bea.wlevs.ede.api.RunnableBean;
import com.bea.wlevs.ede.api.StreamSender;
import com.bea.wlevs.ede.api.StreamSink;
import com.bea.wlevs.ede.api.StreamSource;
import com.bea.wlevs.util.Service;
import java.lang.RuntimeException;

public class BusStopAdapter implements RunnableBean, StreamSource, StreamSink {
  static final Log s_logger = LogFactory.getLog("BusStopAdapter");
  private String m_filePath;
  private String m_eventTypeName;
  private EventType m_eventType;
  private StreamSender m_eventSender;
  private boolean m_stopped;
  private int m_repeat = 1;
  private EventTypeRepository m_etr = null;

  public BusStopAdapter()
  { super(); }

  /**
   * Called by the server to pass in the path
   * to the file with bus stop data.
   *
   * @param path The value specified for the path
   * property in the adapter's configuration in the EPN assembly file.
   */
   public void setPath(String path) throws RuntimeException {
     // Code to create a File instance from the path. This
     // File object retrieves event data from the file.
   }

  /**
   * Called by the server to pass in the name of the event
   * type to which event data should be bound.
   *
   * @param path The value specified for the path
   * property in the adapter's configuration in the EPN assembly file.
  */
   public void setEventType(String typ)
   { m_eventTypeName = typ; }

   /**
    * Called by the server to set an event type
    * repository instance that knows about event
    * types configured for this application.
    *
    * This repository instance will be used to retrieve an
    * event type instance that is populated with
    * event data retrieved from the event data file.
```

```
 *
 * @param etr The event repository.
 */
 @Service(filter = EventTypeRepository.SERVICE_FILTER)
 public void setEventTypeRepository(EventTypeRepository etr)
 { m_etr = etr; }

/**
 * Executes to retrieve raw event data and
 * create event type instances from it, then
 * sends the events to the next stage in the
 * EPN.
 *
 * This method, implemented from the RunnableBean
 * interface, executes when this adapter instance is active.
 */
 public void run() {
   if (m_etr == null){
     throw new RuntimeException("EventTypeRepository is not set");
   }

   // Get the event type from the repository wit the event type name
   // specified as a property of this adapter in the assembly file.
   m_eventType = m_etr.getEventType(m_eventTypeName);
   if (m_eventType == null){
    throw new RuntimeException("EventType(" + m_eventType + ") is not found.");
   }

   BufferedReader reader = null;

   System.out.println("Sending " + m_eventType + " from " + m_filePath);
   while ((m_repeat != 0) && (!m_stopped)) {
     try {
       reader = new BufferedReader(new FileReader(m_filePath));
     } catch (Exception e) {
       m_stopped = true;
       break;
      }
      while (!isStopped()) {
        try {
          // Create an event and assign to it an event type generated
          // from the event data retrieved by the reader.
          Object ev = null;
          ev = readLine(reader);
          if (ev == null){
             reader.close();
             break;
           }
           // Send the newly created event to a downstream node
          //listening to this adpater.
           m_eventSender.sendInsertEvent(ev);
         } catch (Exception e){
           m_stopped = true;
           break;
          }
       }
   }
}
/**
* Called by the server to pass in a sender instance to be used to
* send generated events to a downstream node.
```

```
   *
   * @param sender A sender instance.
   */
   public void setEventSender(StreamSender sender)
   { m_eventSender = sender; }

  /**
   * Returns true if this adapter instance has been
   * suspended, such as because an exception occurred.
   */
   private synchronized boolean isStopped()
   { return m_stopped; }

  /**
   * Reads data from reader, creating event type
   * instances from that data. This method is
   * called from the run() method.
   *
   * @param reader Raw event data from a file.
   * @return An instance of the event type specified
   * as a property of this adapter.
   */
   protected Object readLine(BufferedReader reader) throws Exception
   {
     // Code to read raw event data and return an event type
     // instance from it.
   }

  /**
   * Called by the server to pass in an
   * insert event received from an
   * upstream stage in the EPN.
   */
   @Override
   public void onInsertEvent(Object event) throws EventRejectedException {
     // Code to begin executing the logic needed to
     // convert incoming event data to event type instances.
   }
}
```

1.3 Configure a Custom Adapter

When you create a custom adapter, you add it to an EPN by configuring it in the assembly and component configuration files.

1.3.1 Assembly File Elements and Properties

In the assembly file, use the `wlevs:adapter` element to declare an adapter as a component in the event processor network. Use `wlevs:instance-property` child elements to set static properties in the adapter. Static properties are properties that you do not change dynamically after you deploy the adapter.

In the following example, the `BusStopAdapter` class is configured with properties that correspond to the class methods. The `BusStopAdapter` accessor methods `setPath`, `setEventType`, and `setBuffer` correspond to the `path`, `eventType`, and `buffer` properties in the assembly file. In the assembly file, the property name spellings match the spelling of the accessor methods minus the `get` or `set` prefix.

```
<wlevs:adapter id="BusStopAdapter"
    class="com.oracle.cep.sample.spatial.BusStopAdapter" >
```

```
    <wlevs:instance-property name="path" value="bus_stops.csv"/>
    <wlevs:instance-property name="eventType" value="BusStop"/>
    <wlevs:instance-property name="buffer" value="30.0"/>
</wlevs:adapter>
```

You reference an adapter that is an event sink by using the `wlevs:listener` element. In the following example, a `BusPositionGen` CSV adapter sends events to the `BusStopAdapter`:

```
<wlevs:adapter id="BusPositionGen" provider="csvgen">
    <!-- Code omitted -->
    <wlevs:listener ref="BusStopAdapter"/>
</wlevs:adapter>
```

1.3.2 Configuration File Elements and Properties

In the configuration file, use the `adapter` element to add child elements and properties that can be updated at run time. Each adapter configuration requires a separate `adapter` child element of the `config` element. The following example configures the `BusStopAdapter` adapter for event recording.

```
<?xml version="1.0" encoding="UTF-8"?>
<wlevs:config
    xmlns:wlevs="http://www.bea.com/ns/wlevs/config/application">
    <adapter>
        <name>BusStopAdapter</name>
        <record-parameters>
            <dataset-name>spatial_sample</dataset-name>
            <event-type-list>
                <event-type>BusPos</event-type>
                <event-type>BusStop</event-type>
                <event-type>BusPosEvent</event-type>
                <event-type>BusStopArrivalEvent</event-type>
                <event-type>BusStopPubEvent</event-type>
                <event-type>BusStopPubEvent</event-type>
            </event-type-list>
            <batch-size>1</batch-size>
            <batch-time-out>10</batch-time-out>
        </record-parameters>
    </adapter>
</wlevs:config>
```

1.4 Pass Login Credentials from an Adapter to a Data Feed Provider

If your adapter accesses an external data feed, the adapter might need to pass login credentials to the data feed for user authentication. You can hard code the unencrypted login credentials in your adapter code. With this insecure approach, you cannot encrypt the password or easily change the login credentials.

You first decide whether you want the login credentials to be configured statically in the assembly file or dynamically by extending the adapter configuration. Configuring the credentials statically in the assembly file is easier, but if the credentials change, you must restart the application for the update to the assembly file to take place. Extending the adapter configuration enables you to change the credentials dynamically without restarting the application, but involves additional steps, such as creating an XSD file and compiling it into a JAXB object.

- Pass Static Login Credentials
- Pass Dynamic Login Credentials

- Access Login Credentials at Run Time.

1.4.1 Pass Static Login Credentials

This section describes how to pass login credentials that you configure statically in the assembly file.

Pass Static Credentials to the Data Feed Provider

1. Open the assembly file.
2. Add two instance properties that correspond to the user name and password of the login credentials.

   Add a `password` element with the cleartext password value. You encrypt the password in step 4.

   ```
   <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>
   ```

3. Save the assembly file.
4. Execute the `encryptMSAConfig` command to encrypt the `password` element in the assembly file. Keep the following command on one line.

   `/Oracle/Middleware/oep/``oep/bin/encryptMSAConfig . assembly_file /Oracle/Middleware/oep/user_projects/domains/<domain>/<server>/.aesinternal.dat`*

   Running the command encrypts the `password` element value in the assembly file.

   `/Oracle/Middleware/oep/`: The Oracle Event Processing installation directory.

   Dot (.): The directory that contains the assembly file. The dot (.) means you are executing the command in the directory where the assembly file is located.

   `assembly_file`: The name of the assembly file.

   `~/aesinternal.dat`: The name and location of the `*.aesinternal.dat` key file associated with your domain. The key file encrypts the `<password />` element in the assembly file.

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

   See Access Login Credentials at Run Time.

6. Edit the `MANIFEST.MF` file of the application and add the `com.bea.core.encryption` package to the `Import-Package` header.
7. Reassemble and deploy your application.

1.4.2 Pass Dynamic Login Credentials

You can extend the a custom adapter configuration by creating your own XSD file.

Create an XSD File

1. Create the new XSD schema file that describes the custom adapter configuration. Add the `user` and `password` elements of type `string`.

   For example, to extend the HelloWorld adapter configuration, the XSD file looks 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 Extend Adapters and Event Beans for detailed instructions.

2. Change to the directory that contains the adapter configuration file.

3. In the adapter configuration file, put the user name and password into the `<user>` and `<password>` elements.

   Specify the cleartext password value. You encrypt it in step 4.

   ```
   <?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>
   ```

4. Execute the `encryptMSAConfig` command to encrypt the `password` element in the assembly file. Keep the following command on one line.

   ```
   /Oracle/Middleware/oep/oep/bin/encryptMSAConfig . assembly_file /Oracle/
   Middleware/oep/user_projects/domains/<domain>/<server>/.aesinternal.dat
   ```

   Running the command encrypts the `password` element value in the assembly file.

   /Oracle/Middleware/oep/: The Oracle Event Processing installation directory.

   Dot (.): The directory that contains the assembly file. The dot (.) means you are executing the command in the directory where the assembly file is located.

   `assembly_file`: The name of the assembly file.

   ~/`aesinternal.dat`: The name and location of the `*.aesinternal.dat` key file associated with your domain. The key file encrypts the `<password />` element in the assembly file.

5. Save the adapter configuration file.

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

See Access Login Credentials at Run Time.

7. Edit the `MANIFEST.MF` file of the application and add the `com.bea.core.encryption` package to the `Import-Package` header.

8. Reassemble and deploy your application.

   When you extend an adapter configuration, you must import the following packages in the OSGi `MANIFEST.MF` file:

```
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="11.1.1",
com.bea.wlevs.configuration.application;version="11.1.1",
com.sun.xml.bind.v2;version="2.0.2"
```

1.4.3 Access Login Credentials at Run Time

To access the login credentials and decrypt the password at run time, you add code to the custom adapter Java class. To decrypt the password, import and use the `com.bea.core.encryption.EncryptionService` class.

Access Login Credential Properties

1. 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;
```

2. Use the OSGi `@Service` annotation to get a reference to the `EncryptionService`.

```
private EncryptionService encryptionService;

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

3. In the `@Prepare` callback method, get the values of the `user` and `password` properties of the extended adapter configuration. The example shows the code to get the `password` value. See Adapter Life Cycle Annotations for information about the `@Prepare`, `@Activate`, and `@Rollback` adapter life cycle annotations.

```
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 Access to the Component Configuration for information about accessing the extended adapter configuration.

4. 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
```

5. Use the vendor API to pass these credentials to the data feed provider.

1.5 Multithreaded Adapters

You can implement or configure an adapter to use multiple threads to read from the data source. A multithreaded adapter can improve performance when its event processing work has high overhead. To implement multithreading, you can use a work manager or implement the `WorkManager` interface to make an adapter multithreaded. You can implement the `Runnable` or the RunnableBean interface to execute custom adapters in parallel.

Note:

In a single-threaded adapter, event order is guaranteed. Event order is not guaranteed in a multithreaded adapter. You might need to add code to your custom adapter class to manage event order.

Work Manager

The simplest way to implement threading is to configure the adapter with a work manager. A work manager is a server feature through which your application can submit code (classes that implement the `Work` interface) for scheduling on multiple threads. The interface to the work manager feature is `commonj.work.WorkManager`.

In Oracle Event Processing, `WorkManager` instances can be configured on the server in the `config.xml` file. The system also allocates a default work manager for each application to use.

You can obtain a reference to a work manager to use in your adapter in one of two ways:

- You can create a configuration property for the adapter that allows users to specify the name a specific work manager scoped to the server (configured in the `config.xml` file) that will be injected into your adapter. See the configuration details for the built-in JMS adapter for an example of an adapter that supports explicit configuration of a work manager.
- You can implement the `com.bea.wlevs.ede.spi.WorkManagerAware` interface in your adapter. This interface consists of a single method `setWorkManager(WorkManager wm)` that is called by the system when initializing your adapter to inject the default work manager for you application.

Once you have obtained a `WorkManager` instance you can submit work to it that will be executed on the work managers thread pool.

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

WorkManagerAware Interface

If you need finer control over the threading, a custom adapter can implement the `WorkManagerAware` interface. In this case, Oracle Event Processing injects the adapter with the work manager of the application during initialization. Use this work manager to explicitly manage the threading for an adapter or an event bean instance.

RunnableBean Interface

Adapters that need to run some of their code on a separate thread, but do not need to have full control over threading provided by the `WorkManager` API can implement the `com.bea.wlevs.ede.api.RunnableBean` interface. This interface includes a `run` method that is called by the system on one of the threads belonging to the application's default work manager. The `RunnableBean` interface provides adapters with a very simple model for running code on a single work manager thread. If you need to explicitly use multiple work manager threads for more parallelism, use the `WorkManager` interface directly. See Work Manager.

EPN

For reference information on this interface, see the *Java API Reference for Oracle Event Processing*.

1.6 Suspend and Resume Adapter Event Processing

You can implement the following interfaces to add server support for suspending and resuming event processing. For example, you can make the custom adapter stop processing events when the EPN suspends, and make it start processing events again when the EPN resumes. In either or both cases, can manage resources as needed.

Table 1-1 Interfaces to Support Suspending and Resuming an Adapter

Interface	Description
`com.bea.wlevs.ede.api.SuspendableBean`	Provides logic that executes when the EPN suspends. In the `suspend` method, you can suspend resources or stop processing events.

Table 1-1 (Cont.) Interfaces to Support Suspending and Resuming an Adapter

Interface	Description
`com.bea.wlevs.ede.api.ResumeableBean`	Implement this to provide logic that executes when the EPN resumes work. In your implementation of the `beforeResume` method, you can provide code that should execute before the adapter's work resumes.

See the *Java API Reference for Oracle Event Processing*.

1.7 Adapter Life Cycle Annotations

Use the following annotations to specify the methods of a custom adapter implementation that handle a custom adapter life cycle stages: when its configuration is prepared, when its configuration is activated, and when the adapter terminates because of an exception:

- Syntax
- com.bea.wlevs.configuration.Prepare (@Prepare)
- com.bea.wlevs.configuration.Activate (@Activate)
- com.bea.wlevs.configuration.Rollback (@Rollback).

1.7.1 Syntax

Place the applicable `@Prepare`, `@Activate`, or `@Rollback` annotation above custom adapter methods with the following signature to indicate that these methods send configuration information to the adapter:

```
public void methodName(AdapterConfigObject adapterConfig)
```

`AdapterConfigObject` refers to the Java representation of the adapter's configuration XML file that is deployed with the application. The type of this class is `com.bea.wlevs.configuration.application.DefaultAdapterConfig` by default. If you have extended the configuration of the adapter, then the type of this class is whatever is specified in the XSD that describes the extended XML file. For example, in the HelloWorld sample, the type is the following: `com.bea.wlevs.adapter.example.helloworld.HelloWorldAdapterConfig`

At runtime, Oracle Event Processing creates the `AdapterConfigObject` class, populates the class with data from the XML file, and passes the created instance to the adapter. The adapter methods annotated with the `@Activate` and `@Rollback` adapter life cycle annotations use the class to get information about the adapter configuration.

These `@Prepare`, `@Activate`, and `@Rollback` annotations have no attributes. In the following examples, the `HelloWorldAdapterConfig` class represents the adapter's configuration XML file. When you deploy the application, Oracle Event Processing creates an instance of this class. In the HelloWorld example, the adapter configuration is extended. See the example in com.bea.wlevs.configuration.Activate (@Activate).

1.7.2 com.bea.wlevs.configuration.Prepare (@Prepare)

The Oracle Event Processing server calls methods annotated with `@Prepare` whenever an adapter's state is updated by a configuration change. The following example is from the adapter component of the HelloWorld sample application and shows how to use the `@Prepare` annotation.

```
package com.bea.wlevs.adapter.example.helloworld;
...
import com.bea.wlevs.configuration.Prepare;
import com.bea.wlevs.ede.api.RunnableBean;
import com.bea.wlevs.ede.api.StreamSender;
import com.bea.wlevs.ede.api.StreamSource;
import com.bea.wlevs.event.example.helloworld.HelloWorldEvent;

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

The annotated `checkConfiguration` method checks that the `message` property of the adapter's configuration (set in the extended adapter configuration file) is not `null` or empty. If it `null` or empty, then the method throws an exception.

1.7.3 com.bea.wlevs.configuration.Activate (@Activate)

The Oracle Event Processing server calls methods annotated with `@Activate` after the server has called and has successfully executed all the methods marked with the `@Prepare` annotation. Use the `@Activate` method to get the adapter configuration data to use in the rest of the adapter implementation. The following example shows how to use the `@Activate` annotation in the adapter component of the HelloWorld example:

```
package com.bea.wlevs.adapter.example.helloworld;
...
import com.bea.wlevs.configuration.Activate;
import com.bea.wlevs.ede.api.RunnableBean;
import com.bea.wlevs.ede.api.StreamSender;
import com.bea.wlevs.ede.api.StreamSource;
import com.bea.wlevs.event.example.helloworld.HelloWorldEvent;

public class HelloWorldAdapter implements RunnableBean, StreamSource {
...
    @Activate
    public void activateAdapter(HelloWorldAdapterConfig adapterConfig) {
        this.message = adapterConfig.getMessage();
    }
...
}
```

The following example shows that this XSD file also specifies the fully qualified name of the resulting Java configuration object, as shown in bold:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns="http://www.bea.com/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/ns/wlevs/config/application"
  targetNamespace="http://www.bea.com/ns/wlevs/example/helloworld"
  elementFormDefault="unqualified" attributeFormDefault="unqualified"
  jxb:extensionBindingPrefixes="xjc" jxb:version="1.0">
   <xs:annotation>
      <xs:appinfo>
        <jxb:schemaBindings>
          <jxb:package name="com.bea.wlevs.adapter.example.helloworld"/>
        </jxb:schemaBindings>
      </xs:appinfo>
   </xs:annotation>
   <xs:import namespace="http://www.bea.com/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="channel" 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>
```

Oracle Event Processing creates an instance of this class when the application is deployed. For example, the adapter section of the `helloworldAdapter` configuration file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
  <helloworld:config
  ...
    <adapter>
      <name>helloworldAdapter</name>
      <message>HelloWorld - the current time is:</message>
    </adapter>
</helloworld:config>
```

The annotated `activateAdapter` method uses the `getMessage` method of the configuration object to get the value of the `message` property set in the adapter's configuration XML file. In this case, the value is `HelloWorld - the current time is:`. This value can be used in the main part of the adapter implementation file.

1.7.4 com.bea.wlevs.configuration.Rollback (@Rollback)

The Oracle Event Processing server calls methods annotated with `@Rollback` whenever a component annotated with `@Prepare` was called but threw an exception. The following example from the adapter component of the HelloWorld example, shows how to use the `@Rollback` annotation:

```
package com.bea.wlevs.adapter.example.helloworld;
...
import com.bea.wlevs.configuration.Rollback;
import com.bea.wlevs.ede.api.RunnableBean;
import com.bea.wlevs.ede.api.StreamSender;
import com.bea.wlevs.ede.api.StreamSource;
```

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

public class HelloWorldAdapter implements RunnableBean, StreamSource {
    @Rollback
...
    public void rejectConfigurationChange(HelloWorldAdapterConfig adapterConfig) {
    }
```

In the example, the `rejectConfigurationChange` method is annotated with the `@Rollback` annotation, which means this is the method that is called if the `@Prepare` method threw an exception. In the example above, nothing actually happens.

2

Extend Adapters and Event Beans

You can use an XML schema definition (XSD) or Java Architecture for XML Binding (JAXB) annotations to extend the default configuration for adapters and event beans.

This chapter includes the following sections:

- Annotations
- XSD File
- Access to the Component Configuration.

2.1 Annotations

The simplest and most efficient way to extend component configuration is to annotate your custom adapter or event bean JavaBean class with the annotations provided in the JAXB `javax.xml.bind.annotation` package. For more information, see http://docs.oracle.com/javaee/7/api/javax/xml/bind/annotation/package-summary.html and https://jaxb.java.net/.

Extend Component Configuration with Annotations

1. Implement your custom adapter or event bean JavaBean class.
2. Annotate the attributes of your custom adapter or event bean to specify the component configuration with the applicable annotations from the `javax.xml.bind.annotation` package.

   Important `annotations from the javax.xml.bind.annotation` package include the following. A property without an annotation defaults to the optional configuration property (`@XmlElement`).

   - `@XmlElement`: Property is an optional part of the component configuration.
   - `@XmlElement(required=true)`: Property is a required part of the component configuration.
   - `@XmlTransient`: Property is not part of the component configuration.
   - `@XmlJavaTypeAdapter`: Property elements annotated with this can specify custom handling to accommodate most Java data types.

   Note:

   If you extensively use of `@XmlJavaTypeAdapter`, consider defining your own custom schema. See XSD File.

The following example shows a custom adapter implementation annotated with `javax.xml.bind.annotation` annotations to specify:

- `count`: Not part of the component configuration.
- `doit`: Required part of the component configuration.
- `size`: Required part of the component configuration. Maps to howBig instance property.

```
@XmlType(name="SampleAdapterConfig", namespace="http://www.oracle.com/ns/cep/config/sample")
public class SampleAdapterImpl implements Adapter {
  @XmlTransient
  private int count;

  @XmlElement(name="size")
  private int howBig;

  @XmlElement(required=true)
  private boolean doit;

  ...

  public void setDoit(boolean doit) {
    this.doit = doit;
  }

  public boolean getDoit() {
    return doit;
  }
}
```

3. Within your custom adapter or event bean code, access the extended configuration as Access to the Component Configuration describes.
4. Modify the component configuration XML file that describes the custom components of your application.

   For more information, see Configure a Custom Adapter.
5. When you create the component configuration XML file that describes the components of your application, be sure to use the extended XSD file as its description. Also be sure to identify the name space for this schema rather than use the default schema.

   The following example shows a component configuration file for the `SampleAdapterImpl` custom adapter in step 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<app:config
  xmlns:app="http://www.bea.com/ns/wlevs/config/application"
  xmlns:sample="http://www.oracle.com/ns/cep/config/sample"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://www.bea.com/ns/wlevs/config/application
    http://www.bea.com/ns/wlevs/config/application/wlevs_application_config.xsd
    http://www.oracle.com/ns/cep/config/sample
    http://www.oracle.com/ns/cep/config/sample/ocep_sample_config.xsd">
  <processor>
    <name>clusterProcessor</name>
    <rules>
      <query id="clusterRule"><![CDATA[ select * from clusterInstream [Now] >
      </query>
    </rules>
  </processor>
    <sample:adapter>
```

```
      <name>myadapter</name>
      <config>
        <size>15</size>      <!-- optional -->
        <doit>true</doit>    <!-- required -->
      </config>
    </sample:adapter>
</app:config>
```

Note:

The extended component configuration schema requires a nested `config` element.

6. Package and deploy your application.

2.2 XSD File

You can extend an adapter or an event bean configuration with an XML Schema Definition (XSD) file. Add as many custom elements to the XSD file as you need with few restrictions other than each new element must have a `name` attribute.

When you set up the name spaces, Oracle recommends that you use `elementFormDefault="unqualified"` so that locally defined elements do not have a names pace, but global elements leverage the `targetNamespace`. This avoids name clashes across schemas without requiring excessive prefixing. For more information, see http://www.xfront.com/HideVersusExpose.html.

You can also manually generate a custom schema that does not extend the application schema. This allows you to create custom configuration in your own name space without having to define all of the other elements. This mechanism functions like the annotation approach after you generate the schema.

2.2.1 Extend the Component Configuration with an XSD File

1. Create the new XSD Schema file that describes the extended adapter or event bean 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 Event Processing, to describe them.

   See Create an XSD File.

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 the GlassFish reference implementation. This Ant task is included in the Oracle Event Processing distribution for your convenience.

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

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

For more information, see https://jaxb.java.net/nonav/2.0.2/docs/xjcTask.html.

3. Compile these generated Java files into classes.
4. Package the compiled Java class files in your application bundle.
5. Program your custom adapter or event bean.

   For more information, see: Implementation Basics.
6. Within your custom adapter or event bean code, access the extended configuration as Access to the Component Configuration describes.
7. When you create the component configuration XML file that describes the components of your application, be sure to use the extended XSD file as its description. In addition, be sure you identify the names pace for this schema rather than the default schema.

   The following example shows a component configuration file for the XSD you created in Create an XSD 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>
```

2.2.2 Create an XSD File

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

Oracle recommends that you use the XSD in Complete Example of an XSD Schema File 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.

Create a New XSD File

1. Create the basic XSD file with the required names paces, 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/ns/wlevs/config/application"
  jxb:extensionBindingPrefixes="xjc" jxb:version="1.0"
  elementFormDefault="unqualified" attributeFormDefault="unqualified">
...
</xs:schema>
```

2. Import the `wlevs_application_config.xsd` file:

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

   The `wlevs_application_config.xsd` file imports the `wlevs_base_config.xsd` file.

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

   The new type must extend the `AdapterConfig` type, which is defined in `wlevs_application_config.xsd`. `AdapterConfig` extends `ConfigurationObject`. You can 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 indicate what component they configure (`adapter`, `processor`, `channel`) although you can name then anything you want.

   Each element must extend the `ConfigurationObject` XML type, either explicitly using the `xs:extension` element with `base` attribute value

`base:ConfigurationObject` or by specifying an XML type that extends `ConfigurationObject`. The `ConfigurationObject` XML type, defined in `wlevs_base_config.xsd`, defines a single attribute: `name`.

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

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

- `DefaultProcessorConfig`: For a description of the default processor configuration, see *Java API Reference for Oracle Event Processing*
- `DefaultStreamConfig`: For a description of the default channel configuration, see Oracle Fusion Middleware Developing Applications with Oracle Event Processing.

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>
```

2.2.3 Complete Example of an XSD Schema File

Use the following extended XSD file as a template:

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

2.3 Access to the Component Configuration

When you deploy your application, Oracle Event Processing maps the configuration of each component specified in the component configuration files into Java objects that adhere to the JAXB standard. Because there is a single XML element that contains the configuration data for each component, JAXB also produces a single Java class that represents the configuration data. Oracle Event Processing passes an instance of this Java class to the component (processor, channel, or adapter) at runtime when the component initializes and whenever a dynamic change to the component configuration occurs.

You can access this component configuration at run time in the following ways:

- With resource injection as Resource Injection describes.

  This is the simplest and most efficient way to access component configuration at runtime.

- With callbacks as Life Cycle Callbacks.

  This is the most flexible way to access component configuration at run time. Consider this option if resource injection does not meet your needs.

Note:

Clients that need to use a schema from a provider must import the appropriate package from the provider bundle so the provider's `ObjectFactory` is visible to the client bundle.

2.3.1 Resource Injection

By default Oracle Event Processing configures adapters and event beans by direct injection of their JavaBean properties followed by configuration callbacks.

Consider the annotated custom adapter implementation.

```
@XmlType(name="SampleAdapterConfig", namespace="http://www.oracle.com/ns/cep/config/sample")
public class SampleAdapterImpl implements Adapter {
    private boolean doit;

    public void setDoit(boolean doit) {
        this.doit = doit;
    }
```

```
    public boolean getDoit() {
        return doit;
    }
}
```

Consider the configuration file for an instance of this custom adapter.

```
<?xml version="1.0" encoding="UTF-8"?>
<app:config
      xmlns:app="http://www.bea.com/ns/wlevs/config/application"
      xmlns:sample="http://www.oracle.com/ns/cep/config/sample"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="
            http://www.bea.com/ns/wlevs/config/application
            http://www.bea.com/ns/wlevs/config/application/wlevs_application_config.xsd
            http://www.oracle.com/ns/cep/config/sample
            http://www.oracle.com/ns/cep/config/sample/ocep_sample_config.xsd">
  <processor>
    <name>clusterProcessor</name>
    <rules>
      <query id="clusterRule"><![CDATA[ select * from clusterInstream [Now] ></query>
    </rules>
  </processor>
    <sample:adapter>
        <name>myadapter</name>
        <config>
            <doit>true</doit>
        </config>
    </sample:adapter>
</app:config>
```

At runtime Oracle Event Processing calls the `setDoIt` method of the adapter instance and passes the value `true` as the argument value.

Note:

The extended component configuration schema requires a nested `config` element.

2.3.2 Life Cycle Callbacks

In your adapter or event bean implementation, you can use metadata annotations to specify the Java methods to be invoked by Oracle Event Processing at runtime. Oracle Event Processing passes an instance of the configuration Java class to these specified methods. You can program these methods to get specific run time configuration information about the adapter or event bean.

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

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

By default, the data type of the adapter configuration Java class is `com.bea.wlevs.configuration.application.DefaultAdapterConfig`. If you have extended the configuration of the adapter (or event bean) by creating your own XSD file that describes the configuration XML file, then you specify the type in the XSD file. In the preceding example, the data type of the Java configuration object is `com.bea.wlevs.example.helloworld.HelloWorldAdapterConfig`.

2.3.3 Life Cycle Callback Annotations

You can use the following metadata annotations to specify life cycle callback methods:

- `com.bea.wlevs.management.Activate`: Specifies the method invoked when the configuration is activated.
- `com.bea.wlevs.management.Prepare`: Specifies the method invoked when the configuration is prepared.
- `com.bea.wlevs.management.Rollback`: Specifies the method invoked when the adapter is terminated due to an exception.

2.3.4 Life Cycle for Adapters and Event Beans

Oracle Event Processing uses the following life cycle to create adapter and event beans.

1. Create the adapter or event bean instance.
2. Inject static properties.
3. Call the `afterPropertiesSet` method.
4. Prepare phase:
   a. If `@Prepare` with one or more configuration arguments is present, call it. Otherwise, directly inject configuration properties.
   b. If `@Prepare` without arguments is present, call it.
5. Activate phase:
   a. If `@Activate` with one or more configuration arguments is present, call it.
   b. If `@Activate` without arguments is present, call it.
6. Call `afterConfigurationActive`.
7. Continue with other configuration.

3

Adapter and Event Bean Factories

You can use a single custom adapter or event bean implementation in multiple EPNs by implementing and configuring an adapter or event bean factory class. The factory class instantiates an adapter or event bean object to Oracle Event Processing applications that request one.

For detail on the APIs described here, see the *Java API Reference for Oracle Event Processing*.

This chapter includes the following sections:.

3.1 Create an Adapter Factory

1. In your adapter class, implement the `com.bea.wlevs.ede.api.Adapter` interface so that the adapter can be returned by the factory.

   This is a marker interface, so there are no methods to implement.

   ```
   public class BusStopAdapter implements Adapter, StreamSource {
       // Adapter implementation code.
   }
   ```

2. Create an adapter factory class that implements the `com.bea.wlevs.ede.api.AdapterFactory` interface.

   The `create` method creates and returns instances of the custom adapter.

   ```
   import com.oracle.cep.sample.spatial.BusStopAdapter;
   import com.bea.wlevs.ede.api.AdapterFactory;

   public class BusStopAdapterFactory implements AdapterFactory {
     public BusStopAdapterFactory() {}
     public synchronized BusStopAdapter create() throws IllegalArgumentException {
     // Your code might have a particular way to create the instance.
        return new BusStopAdapter();
     }
   }
   ```

3. In an assembly file, use the `wlevs:factory` element to register the factory class.

   ```
   <wlevs:factory provider-name="busStopAdapterProvider"
     class="com.oracle.cep.sample.spatial.BusStopAdapterFactory"/>
   ```

4. Use the `osgi:service` element to register the factory as an OSGi service in the assembly file.

   The OSGi service registry scope is all of Oracle Event Processing. If more than one application deployed to a given server uses the same adapter factory, register the factory once as an OSGi service.

a. 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 is referenced.

b. Add a nested standard Spring `bean` element to register your specific adapter class in the Spring application context.

```
<osgi:service interface="com.bea.wlevs.ede.api.AdapterFactory">
  <osgi:service-properties>
    <entry key="type" value="busStopAdapterProvider"</entry>
  </osgi:service-properties>
  <bean class="com.oracle.cep.sample.spatial.BusStopAdapterFactory" />
</osgi:service>
```

5. In applications that use instances of the adapter, configure the adapter by specifying the configured factory as a provider (rather than specifying the adapter by its class name), as shown in the following example:

```
<wlevs:adapter id="BusStopAdapter"
  provider="busStopAdapterProvider">
  // ...
</wlevs:adapter>
```

3.2 Create an Event Bean Factory

1. In your event bean class, implement the `com.bea.wlevs.ede.api.EventBean` interface so that the bean can be returned by the factory.

   This is a marker interface, so there are no methods to implement.

```
public class TradeListener implements EventBean, StreamSink {
  // Bean implementation code.
}
```

2. Create an event bean class that implements the `com.bea.wlevs.ede.api.EventBeanFactory` interface.

   The create method creates and returns instances of the event bean.

```
import com.oracle.cep.example.tradereport.TradeListener;
import com.bea.wlevs.ede.api.EventBeanFactory;

public class TradeListenerFactory implements EventBeanFactory {
  public TradeListenerFactory() { }
  public synchronized TradeListener create() throws IllegalArgumentException {
  // Your code might have a particular way to create the instance.
    return new TradeListener();
  }
}
```

3. In an assembly file, use the `wlevs:factory` element to register the factory class.

```
<wlevs:factory provider-name="tradeListenerProvider"
  class="com.oracle.cep.example.tradereport.TradeListenerFactory"/>
```

4. Use the `osgi:service` element to register the factory as an OSGi service in the assembly file.

   The OSGi service registry scope is all of Oracle Event Processing. If more than one application deployed to a given server uses the same adapter factory, register the factory once as an OSGi service.

a. Add an entry to register the service as an implementation of the `com.bea.wlevs.ede.api.EventBeanFactory` interface. Provide a property, with the `key` attribute equal to `type`, and the name by which this adapter provider is referenced.

b. Add a nested standard Spring `bean` element to register your specific adapter class in the Spring application context.

```
<osgi:service interface="com.bea.wlevs.ede.api.EventBeanFactory">
  <osgi:service-properties>
    <entry key="type" value="tradeListenerProvider"</entry>
    </osgi:service-properties>
  <bean class="com.oracle.cep.example.tradereport.TradeListenerFactory" />
</osgi:service>
```

5. In applications will use instances of the event bean, configure the event bean by specifying the configured event bean factory as a provider (rather than specifying the bean by its class name), as shown in the following example:

```
<wlevs:event-bean id="TradeListenerBean" provider="tradeListenerProvider">
```

4

Assemble an Adapter or Event Bean

You typically bundle custom adapters and event beans in the same application JAR file that contains the other EPN components. Alternately, you can bundle the custom adapter and its factory or the event bean and its factory in their own respective JAR file so that you can reference them from other application bundles.

For example, when two different applications read data coming from the same data feed provider and both applications have the same event types, it can make sense to share a single adapter implementation. With this approach, you do not have to duplicate the custom adapter implementation in two different applications.

This chapter includes the following sections:

4.1 Assemble a Custom Adapter in its Own Bundle

When you assemble a custom adapter in its own bundle, do not declare it in the assembly file with the `wlevs:adapter` element. You use this element to declare the custom adapter in the assembly file of the application bundle that uses it.

1. Create an OSGI bundle that contains the following Java classes: The custom adapter class, its JavaBean event type class, and its adapter factory class.

   In this procedure, this bundle is called `GlobalAdapter`.

2. In the `GlobalAdapter` bundle assembly file:

   - Register the adapter factory as an OSGi service.
   - Register the custom adapter JavaBean event type class as an OSGi service.

3. In the `MANIFEST.MF`, export the JavaBean event type class with the `Export-Package` header.

4. Assemble and deploy the `GlobalAdapter` bundle.

5. In the assembly file of the application that is going to use the custom adapter, declare the custom adapter component as Configure a Custom Adapter describes.

   You still use the `provider` attribute to specify the OSGI-registered adapter factory, although in this case the OSGi registration happens in a different assembly file (of the `GlobalAdapter` bundle) from the assembly file that actually uses the adapter.

6. If you have exported the JavaBean event type into 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.

4.2 Assemble an Event Bean in its Own Bundle

When you assemble an event bean in its own bundle, do not declare it in the assembly file with the `wlevs:event-bean` element. You use this element to declare the custom event bean in the assembly file of the application bundle that uses them.

1. Create an OSGi bundle that contains the following Java classes: The event bean Java class and its event bean factory class.

   In this procedure, this bundle is called `GlobalEventBean`.

2. In the `GlobalEventBean` assembly file register the event bean factory as an OSGi service.

3. Assemble and deploy the `GlobalEventBean` bundle.

4. In the assembly file of the application that is going to use the event bean, declare the custom event bean component.

   You still use the `provider` attribute to specify the OSGi-registered event bean factory, although in this case the OSGi registration happens in a different assembly file (of the `GlobalEventBean` bundle) from the assembly file that actually uses the event bean.

5

Custom Event Store Provider

You can create a custom persistent event store to store recorded events. For example, you could specify a Relational Database Management System (RDBMS) such as Oracle Database or Derby as your persistent event store.

This chapter includes the following sections:

5.1 Event Store API

Oracle Event Processing provides an event store API that you can use to create a custom event store. Oracle provides an RDBMS-based implementation for storing events in a relational database or for supporting JDBC connections. If you want to store events in a different kind of database or if the Oracle RDBMS provider does not meet your needs, then you can use the Event Store API to create your own event store provider.

5.2 Interfaces

The event store API is in the `com.bea.wlevs.eventstore` package. The following list describes the most important interfaces:

- `EventStore:` Represents a single event store. The methods enable you to persist events to the store and to use provider-specific queries to query the event store.
- `EventStoreManager`: Manages event stores. One instance of the `EventStoreManager` can ever exist on a given Oracle Event Processing server. This instance registers in the OSGi registry to enable event store providers to register with the event store manager. Use this interface to find existing event stores, create new event stores, get the provider for a given event store, and register an event provider. The event store manager delegates the work to the event store provider.
- `EventStoreProvider`: Underlying repository that provides event store services to clients.

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