EsperIO Reference Documentation

Version: 2.1.0

provided by

Table of Contents

Preface

This document describes input and output adapters for the Esper Java event stream and complex event processor.

If you are new to Esper, the Esper reference manual should be your first stop.

If you are looking for information on a specific adapter, you are at the right spot.

Chapter 1. Adapter Overview

Input and output adapters to Esper provide the means of accepting events from various sources, and for making available events to destinations.

The following input and output adapters exist:

Table 1.1. Input and Output Adapters

Adapter	Description
CSV Input Adapter	The CSV input adapter can read one or more CSV-formatted input sources, transform the textual values into events, and play the events into the engine. The adapter also makes it possible to run complete simulations of events arriving in time-order from different input streams.
Spring JMS Input and Output Adapter	JMS adapters based on the JmsTemplate offered by Spring 2. Provides unmarshalling of JMS `javax.jms.Message` messages for sending into an engine instance, and marshaling of `com.espertech.esper.EventBean` events into JMS messages.

1.1. Adapter Library Classes

1.1.1. The Adapter Interface

The `Adapter` interface allows client applications to control the state of an input and output adapter. It provides state transition methods that each input and output adapter implements.

An input or output adapter is always in one of the following states:

- Opened - The begin state; The adapter is not generating or accepting events in this state
- Started - When the adapter is active, generating and accepting events
- Paused - When operation of the adapter is suspended
- Destroyed

The state transition table below outlines adapter states and, for each state, the valid state transitions:

Table 1.2. Adapter State Transitions

Start State	Method	Next State
Opened	start()	Started
Opened	destroy()	Destroyed
Started	stop()	Opened
Started	pause()	Paused

Start State	Method	Next State
Started	destroy()	Destroyed
Paused	resume()	Started
Paused	stop()	Opened
Paused	destroy()	Destroyed

1.1.2. Using AdapterInputSource

The `com.espertech.esperio.AdapterInputSource` encapsulates information about an input source. Input adapters use the `AdapterInputSource` to determine how to read input. The class provides constructors for use with different input sources:

- `java.io.Reader` to read character streams
- `java.io.InputStream` to read byte streams
- `java.net.URL`
- Classpath resource by name
- `java.io.File`

Adapters resolve Classpath resources in the following order:

1. Current thread classloader via `Thread.currentThread().getContextClassLoader().getResourceAsStream`
2. If the resource is not found: `AdapterInputSource.class.getResourceAsStream`
3. If the resource is not found: `AdapterInputSource.class.getClassLoader().getResourceAsStream`

Chapter 2. The CSV Input Adapter

This chapter discusses the CSV input adapter. CSV is an abbreviation for comma-separated values. CSV files are simple text files in which each line is a comma-separated list of values. CSV-formatted text can be read from many different input sources via `com.espertech.esperio.AdapterInputSource`. Please consult the JavaDoc for additional information on `AdapterInputSource` and the CSV adapter.

2.1. Introduction

In summary the CSV input adapter API performs the following functions.

- Read events from an input source providing CSV-formatted text and send the events to an Esper engine instance
    - Read from different types of input sources
    - Use a timestamp column to schedule events being sent into the engine
    - Playback with options such as file looping, events per second and other options
    - Use the Esper engine timer thread to read the CSV file
- Read multiple CSV files using a timestamp column to simulate events coming from different streams

The following formatting rules and restrictions apply to CSV-formatted text:

- Comment lines are prefixed with a single hash or pound # character
- Strings are placed in double quotes, e.g. `"value"`
- Escape rules follow common spreadsheet conventions, i.e. double quotes can be escaped via double quote
- A column header is required unless a property order is defined explicitly
- If a column header is used, properties are assumed to be of type String unless otherwise configured
- The value of the timestamp column, if one is given, must be in ascending order

2.2. Playback of CSV-formatted Events

The adapter reads events from a CSV input source and sends events to an engine using the class `com.espertech.esperio.csv.CSVInputAdapter`.

The below code snippet reads the CSV-formatted text file "simulation.csv" expecting the file in the classpath. The `AdapterInputSource` class can take other input sources.

```
AdapterInputSource source = new AdapterInputSource("simulation.csv");
(new CSVInputAdapter(epServiceProvider, source, "PriceEvent")).start();
```

To use the CSVInputAdapter without any options, the event type `PriceEvent` and its property names and value types must be known to the engine. The next section elaborates on adapter options.

- Configure the engine instance for a Map-based event type
- Place a header record in your CSV file that names each column as specified in the event type

The sample application code below shows all the steps to configure, via API, a Map-based event type and play the CSV file without setting any of the available options.

```
Map<String, Class> eventProperties = new HashMap<String, Class>();
eventProperties.put("symbol", String.class);
eventProperties.put("price", double.class);
eventProperties.put("volume", Integer.class);
```

```
Configuration configuration = new Configuration();
configuration.addEventTypeAlias("PriceEvent", eventProperties);

epService = EPServiceProviderManager.getDefaultProvider(configuration);

EPStatement stmt = epService.getEPAdministrator().createEPL(
   "select symbol, price, volume from PriceEvent.win:length(100)");

(new CSVInputAdapter(epService, new AdapterInputSource(filename), "PriceEvent")).start();
```

The contents of a sample CSV file is shown next.

```
symbol,price,volume
IBM,55.5,1000
```

The next code snippet outlines using a `java.io.Reader` as an alternative input source :

```
String myCSV = "symbol, price, volume" + NEW_LINE + "IBM, 10.2, 10000";
StringReader reader = new StringReader(myCSV);
(new CSVInputAdapter(epService, new AdapterInputSource(reader), "PriceEvent")).start();
```

In the previous code samples, the `PriceEvent` properties were defined programmatically with their correct types. It is possible to skip this step and use only a column header record. In such a case you must define property types in the header otherwise a type of String is assumed.

Consider the following:

```
symbol,double price, int volume
IBM,55.5,1000

symbol,price,volume
IBM,55.5,1000
```

The first CSV file defines explicit types in the column header while the second file does not. With the second file a statement like `select sum(volume) from PriceEvent.win:time(1 min)` will be rejected as in the second file `volume` is defaulted to type String - unless otherwise programmatically configured.

2.2.1. Using JavaBean POJO Events

The previous section used an event type based on `java.util.Map`. The adapter can also populate the CSV data into JavaBean events directly, as long as your event class provides setter-methods that follow JavaBean conventions. Note that esperio will ignore read-only properties i.e. if you have a read-only property priceByVolume it will not expect a corresponding column in the input file.

To use Java objects as events instead of Map-based event types, simply register the alias for the Java class and provide the same alias to the CSV adapter.

The below code snipped assumes that a PriceEvent class exists that exposes setter-methods for the three properties. The setter-methods are, for example, `setSymbol(String s)`, `setPrice(double p)` and `setVolume(long v)`.

```
Configuration configuration = new Configuration();
configuration.addEventTypeAlias("PriceEvent", PriceEvent.class);

epService = EPServiceProviderManager.getDefaultProvider(configuration);

EPStatement stmt = epService.getEPAdministrator().createEPL(
```

```
   "select symbol, price, volume from PriceEvent.win:length(100)");

(new CSVInputAdapter(epService, new AdapterInputSource(filename), "PriceEvent")).start();
```

2.3. CSV Playback Options

Use the `CSVInputAdapterSpec` class to set playback options. The following options are available:

- Loop - Reads the CSV input source in a loop; When the end is reached, the input adapter rewinds to the beginning
- Events per second - Controls the number of events per second that the adapter sends to the engine
- Property order - Controls the order of event property values in the CSV input source, for use when the CSV input source does not have a header column
- Property types - Defines a new Map-based event type given a map of event property names and types. No engine configuration for the event type is required as long as the input adapter is created before statements against the event type are created.
- Engine thread - Instructs the adapter to use the engine timer thread to read the CSV input source and send events to the engine
- External timer - Instructs the adapter to use the esper's external timer rather than the internal timer. See "Sending timer events" below
- Timestamp column name - Defines the name of the timestamp column in the CSV input source; The timestamp column must carry long-typed timestamp values relative to the current time; Use zero for the current time

The next code snippet shows the use of `CSVInputAdapterSpec` to set playback options.

```
CSVInputAdapterSpec spec = new CSVInputAdapterSpec(new AdapterInputSource(myURL), "PriceEvent");
spec.setEventsPerSec(1000);
spec.setLooping(true);

InputAdapter inputAdapter = new CSVInputAdapter(epService, spec);
inputAdapter.start(); // method blocks unless engine thread option is set
```

2.3.1. Sending timer events

The adapter can be instructed to use either esper's internal timer, or to drive timing itself by sending external timer events. If the internal timer is used, esperio will send all events in "real time". For example, if an input file contains the following data:

```
symbol,price,volume,timestamp
IBM,55.5,1000,2
GOOG,9.5,1000,3
MSFT,8.5,1000,3
JAVA,7.5,1000,1004
```

then esperio will sleep for 1001 milliseconds between sending the MSFT and JAVA events to the engine.

If external timing is enabled then esperio will run through the input file at full speed without pausing. The algorithm used sends a time event after all events for a particular time have been received. For the above example file a time event for 2 will be sent after IBM, for 3 after MSFT and 1004 after JAVA. For many of use cases this gives a performance improvement.

2.4. Simulating Multiple Event Streams

The CSV input adapter can run simulations of events arriving in time-order from different input streams. Use the `AdapterCoordinator` as a specialized input adapter for coordinating multiple CSV input sources by timestamp.

The sample application code listed below simulates price and trade events arriving in timestamp order. Via the adapter the application reads two CSV-formatted files from a URL that each contain a timestamp column as well as price or trade events. The `AdapterCoordinator` uses the timestamp column to send events to the engine in the exact ordering prescribed by the timestamp values.

```
AdapterInputSource sourceOne = new AdapterInputSource(new URL("FILE://prices.csv"));
CSVInputAdapterSpec inputOne = new CSVInputAdapterSpec(sourceOne, "PriceEvent");
inputOne.setTimestampColumn("timestamp");

AdapterInputSource sourceTwo = new AdapterInputSource(new URL("FILE://trades.csv"));
CSVInputAdapterSpec inputTwo = new CSVInputAdapterSpec(sourceTwo, "TradeEvent");
inputTwo.setTimestampColumn("timestamp");

AdapterCoordinator coordinator = new AdapterCoordinatorImpl(epService, true);
coordinator.coordinate(new CSVInputAdapter(inputOne));
coordinator.coordinate(new CSVInputAdapter(inputTwo));
coordinator.start();
```

The `AdapterCoordinatorImpl` is provided with two parameters: the engine instance, and a boolean value that instructs the adapter to use the engine timer thread if set to true, and the adapter can use the application thread if the flag passed is false.

2.5. Pausing and Resuming Operation

The CSV adapter can employ the engine timer thread of an Esper engine instance to read and send events. This can be controlled via the `setUsingEngineThread` method on `CSVInputAdapterSpec`. We use that feature in the sample code below to pause and resume a running CSV input adapter.

```
CSVInputAdapterSpec spec = new CSVInputAdapterSpec(new AdapterInputSource(myURL), "PriceEvent");
spec.setEventsPerSec(100);
spec.setUsingEngineThread(true);

InputAdapter inputAdapter = new CSVInputAdapter(epService, spec);
inputAdapter.start(); // method starts adapter and returns, non-blocking
Thread.sleep(5000); // sleep 5 seconds
inputAdapter.pause();
Thread.sleep(5000); // sleep 5 seconds
inputAdapter.resume();
Thread.sleep(5000); // sleep 5 seconds
inputAdapter.stop();
```

Chapter 3. The Spring JMS Input and Output Adapters

This chapter discusses the input and output adapters for JMS based on the Spring JmsTemplate technology. For more information on Spring, and the latest version of Spring, please visit http://www.springframework.org

3.1. Introduction

Here are the steps to use the adapters:

1. Configure an Esper engine instance to use a `SpringContextLoader` for loading input and output adapters, and point it to a Spring JmsTemplate configuration file.
2. Create a Spring JmsTemplate configuration file for your JMS provider and add all your input and output adapter entries in the same file.
3. For receiving events from a JMS destination into an engine (input adapter):
    a. List the destination and un-marshalling class in the Spring configuration.
    b. Create EPL statements using the event type alias matching the event objects or the Map-event type aliases received.
4. For sending events to a JMS destination (output adapter):
    a. Use the `insert-into` syntax naming the stream to insert-into using the same name as listed in the Spring configuration file
    b. Configure the Map event type of the stream in the engine configuration

In summary the Spring JMS input adapter performs the following functions:

- Initialize from a given Spring configuration file in classpath or from a filename. The Spring configuration file sets all JMS parameters such as JMS connection factory, destination and listener pools.
- Attach to a JMS destination and listen to messages using the Spring class `org.springframework.jms.core.JmsTemplate`
- Unmarshal a JMS message and send into the configured engine instance

The Spring JMS output adapter can:

- Initialize from a given Spring configuration file in classpath or from a filename, and attach to a JMS destination
- Act as a listener to one or more named streams populated via `insert-into` syntax by EPL statements
- Marshal events generated by a stream into a JMS message, and send to the given destination

3.2. Engine Configuration

The Spring JMS input and output adapters are configured as part of the Esper engine configuration. EsperIO supplies a `SpringContextLoader` class that loads a Spring configuration file which in turn configures the JMS input and output adapters. List the `SpringContextLoader` class as an adapter loader in the Esper configuration file as the below example shows. The configuration API can alternatively be used to configure one or more adapter loaders.

```
<esper-configuration>

  <!-- Sample configuration for an input/output adapter loader -->
```

```
  <plugin-loader name="MyLoader" class-name="com.espertech.esperio.SpringContextLoader">
    <!-- SpringApplicationContext translates into Spring ClassPathXmlApplicationContext
           or FileSystemXmlApplicationContext. Only one app-context of a sort can be used.
           When both attributes are used classpath and file, classpath prevails -->
    <init-arg name="classpath-app-context" value="spring\jms-spring.xml" />
    <init-arg name="file-app-context" value="spring\jms-spring.xml" />
  </plugin-loader>

</esper-configuration>
```

The loader loads the Spring configuration file from classpath via the `classpath-app-context` configuration, or from a file via `file-app-context`.

3.3. Input Adapter

3.3.1. Spring Configuration

The Spring configuration file must list input and output adapters to be initialized by `SpringContextLoader` upon engine initialization. Please refer to your JMS provider documentation, and the Spring framework documentation on help to configure your specific JMS provider via Spring.

The next XML snippet shows a complete sample configuration for an input adapter. The sample includes the JMS configuration for an Apache ActiveMQ JMS provider.

```
<!-- Spring Application Context -->
<beans default-destroy-method="destroy">

  <!-- JMS ActiveMQ Connection Factory -->
  <bean id="jmsActiveMQFactory" class="org.apache.activemq.pool.PooledConnectionFactory">
    <property name="connectionFactory">
      <bean class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="tcp://localhost:61616"/>
      </bean>
    </property>
  </bean>

  <!--  ActiveMQ destination to use  by default -->
  <bean id="defaultDestination"
        class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="ESPER.QUEUE"/>
  </bean>

  <!--  Spring JMS Template for ActiveMQ -->
  <bean id="jmsActiveMQTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory">
      <ref bean="jmsActiveMQFactory"/>
    </property>
    <property name="defaultDestination">
      <ref bean="defaultDestination"/>
    </property>
  </bean>

  <!-- Provides listener threads -->
  <bean id="listenerContainer"
              class="org.springframework.jms.listener.SimpleMessageListenerContainer">
    <property name="connectionFactory" ref="jmsActiveMQFactory"/>
    <property name="destination" ref="defaultDestination"/>
    <property name="messageListener" ref="jmsInputAdapter"/>
  </bean>

  <!-- Default unmarshaller -->
  <bean id="jmsMessageUnmarshaller"
              class="com.espertech.esperio.jms.JMSDefaultAnyMessageUnmarshaller"/>
```

```
  <!-- Input adapter -->
  <bean id="jmsInputAdapter" class="com.espertech.esperio.jms.SpringJMSTemplateInputAdapter">
    <property name="jmsTemplate">
      <ref bean="jmsActiveMQTemplate"/>
    </property>
    <property name="jmsMessageUnmarshaller">
      <ref bean="jmsMessageUnmarshaller"/>
    </property>
  </bean>

</beans>
```

This input adapter attaches to the JMS destination `ESPER.QUEUE` at an Apache MQ broker available at port `tcp://localhost:61616`. It configures an un-marshalling class as discussed next.

3.3.2. JMS Message Unmarshalling

EsperIO provides a class for unmarshaling JMS message instances into events for processing by an engine in the class `JMSDefaultAnyMessageUnmarshaller`. The class unmarshals as follows:

- If the received Message is of type `javax.xml.MapMessage`, extract the event type alias out of the message and send to the engine via `sendEvent(alias, Map)`
- If the received Message is of type `javax.xml.ObjectMessage`, extract the `Serializable` out of the message and send to the engine via `sendEvent(Object)`
- Else the un-marshaller outputs a warning and ignores the message

The unmarshaller must be made aware of the event type of events within `MapMessage` messages. This is achieved by the client application setting a well-defined property on the message: `InputAdapter.ESPERIO_MAP_EVENT_TYPE`. An example code snippet is:

```
MapMessage mapMessage = jmsSession.createMapMessage();
mapMessage.setObject(InputAdapter.ESPERIO_MAP_EVENT_TYPE, "MyInputEvent");
```

3.4. Output Adapter

3.4.1. Spring Configuration

The Spring configuration file lists all input and output adapters in one file. The `SpringContextLoader` upon engine initialization starts all input and output adapters.

The next XML snippet shows a complete sample configuration of an output adapter. Please check with your JMS provider for the appropriate Spring class names and settings. Note that the input and output adapter Spring configurations can be in the same file.

```
<!-- Application Context -->
<beans default-destroy-method="destroy">

  <!-- JMS ActiveMQ Connection Factory -->
  <bean id="jmsActiveMQFactory" class="org.apache.activemq.pool.PooledConnectionFactory">
    <property name="connectionFactory">
      <bean class="org.apache.activemq.ActiveMQConnectionFactory">
        <property name="brokerURL" value="tcp://localhost:61616"/>
      </bean>
    </property>
  </bean>
```

```
  <!--  ActiveMQ destination to use  by default -->
  <bean id="defaultDestination"
        class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="ESPER.QUEUE"/>
  </bean>

  <!--  Spring JMS Template for ActiveMQ -->
  <bean id="jmsActiveMQTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory">
      <ref bean="jmsActiveMQFactory"/>
    </property>
    <property name="defaultDestination">
      <ref bean="defaultDestination"/>
    </property>
    <property name="receiveTimeout">
      <value>30000</value>
    </property>
  </bean>

  <!--  Marshaller marshals events into map messages -->
  <bean id="jmsMessageMarshaller" class="com.espertech.esperio.jms.JMSDefaultMapMessageMarshaller"/>
  <bean id="myCustomMarshaller" class="com.espertech.esperio.jms.JMSDefaultMapMessageMarshaller"/>

  <!--  Output adapter puts it all together -->
  <bean id="jmsOutputAdapter" class="com.espertech.esperio.jms.SpringJMSTemplateOutputAdapter">
    <property name="jmsTemplate">
      <ref bean="jmsActiveMQTemplate"/>
    </property>
    <property name="subscriptionMap">
      <map>
        <entry>
          <key><idref local="subscriptionOne"/></key>
          <ref bean="subscriptionOne"/>
        </entry>
        <entry>
          <key><idref local="subscriptionTwo"/></key>
          <ref bean="subscriptionTwo"/>
        </entry>
      </map>
    </property>
    <property name="jmsMessageMarshaller">
      <ref bean="jmsMessageMarshaller"/>
    </property>
  </bean>

  <bean id="subscriptionOne" class="com.espertech.esperio.jms.JMSSubscription">
    <property name="eventTypeAlias" value="MyOutputStream"/>
  </bean>

  <bean id="subscriptionTwo" class="com.espertech.esperio.jms.JMSSubscription">
    <property name="eventTypeAlias" value="MyOtherOutputStream"/>
    <property name="jmsMessageMarshaller">
      <ref bean="myCustomMarshaller"/>
    </property>
  </bean>

</beans>
```

3.4.2. JMS Message Marshalling

EsperIO provides a marshal implementation in the class `JMSDefaultMapMessageMarshaller`. This marshaller constructs a JMS `MapMessage` from any event received by copying event properties into the name-value pairs of the message. The configuration file makes it easy to configure a custom marshaller that adheres to the `com.espertech.esperio.jms.JMSMessageMarshaller` interface.

Note that this marshaller uses `javax.jms.MapMessage` name-value pairs and not general `javax.jms.Message` properties. This means when you'll read the event properties back from the JMS MapMessage, you will have to use the `javax.jms.MapMessage.getObject(...)` method.

The `SpringJMSTemplateOutputAdapter` is configured with a list of subscription instances of type `JMSSubscription` as the sample configuration shows. Each subscription defines an event type alias that must be configured and used in the `insert-into` syntax of a statement.

To connect the Spring JMS output adapter and the EPL statements producing events, use the `insert-into` syntax to direct events for output. Here is a sample statement that sends events into `MyOutputStream`:

```
insert into MyOutputStream select assetId, zone from RFIDEvent
```

The type `MyOutputStream` must be known to an engine instance. The output adapter requires the alias to be configured with the Engine instance, e.g.:

```
<esper-configuration>
  <event-type alias="MyOutputStream">
    <java-util-map>
      <map-property name="assetId" class="String"/>
      <map-property name="zone" class="int"/>
    </java-util-map>
  </event-type>
</esper-configuration>
```

Chapter 4. Additional Event Representations

4.1. Apache Axiom Events

The plug-in event representation based on Apache Axiom can process XML documents by means of the Streaming API for XML (StAX) and the concept of "pull parsing", which can gain performance improvements extracting data from XML documents.

The instructions below have been tested with Apache Axiom version 1.2.5. Please visit `http://ws.apache.org/commons/axiom/` for more information. Apache Axiom requires additional jar files that are not part of the EsperIO distribution and must be downloaded separately.

There are 3 steps to follow:

1. Enable Apache Axiom by adding the Axiom even representation to the engine configuration.
2. Register your application event type aliases.
3. Process `org.apache.axiom.om.OMDocument` or `OMElement` event objects.

To enable Apache Axiom event processing, use the code snippet shown next, or configure via confiugration XML:

```
Configuration config = new Configuration();
config.addPlugInEventRepresentation(new URI("type://xml/apacheaxiom/OMNode"),
    AxiomEventRepresentation.class.getName(), null);
```

Your application may register Axiom event types in advance. Here is sample code for adding event types based on Axiom:

```
ConfigurationEventTypeAxiom desc = new ConfigurationEventTypeAxiom();
desc.setRootElementName("measurement");
desc.addXPathProperty("measurement", "/sensor/measurement", XPathConstants.NUMBER);
URI[] resolveURIs = new URI[] {new URI("type://xml/apacheaxiom/OMNode/SensorEvent")};
configuration.addPlugInEventType("SensorEvent", resolveURIs, desc);
```

The operation above is available at configuration time and also at runtime via `ConfigurationOperations`. After registering an event type alias as above, your application can create EPL statements.

To send Axiom `OMDocument` or `OMElement` events into the engine, your application code must obtain an `EventSender` to process Axiom `OMElement` events:

```
URI[] resolveURIs = new URI[] {new URI("type://xml/apacheaxiom/OMNode/SensorEvent")};
EventSender sender = epService.getEPRuntime().getEventSender(resolveURIs);

String xml = "<measurement><temperature>98.6</temperature></measurement>";
InputStream s = new ByteArrayInputStream(xml.getBytes());
OMElement omElement = new StAXOMBuilder(s).getDocumentElement();

sender.sendEvent(omElement);
```

Configuring an Axiom event type via XML is easy. An Esper configuration XML can be found in the file `esper-axiom-sample-configuration.xml` in the `etc` folder of the EsperIO distribution.

The configuration XML for the `ConfigurationEventTypeAxiom` class adheres to the schema `esperio-axiom-configuration-2-0.xsd` also in the `etc` folder of the EsperIO distribution.