EsperIO Reference

Version 6.0.1

by *EsperTech Inc.* [http://www.espertech.com]

Copyright 2006 - 2017 by EsperTech Inc.

Preface v
1. Adapter Overview 1
1.1. The Adapter Interface 1
2. The File and CSV Input and Output Adapter 3
2.1. Data Flow Operators 3
2.1.1. Introduction 3
2.1.2. FileSink Operator 3
2.1.3. FileSource Operator 4
2.2. CSV Input Adapter API 7
2.2.1. Introduction 7
2.2.2. Playback of CSV-formatted Events 7
2.2.3. CSV Playback Options 10
2.2.4. Simulating Multiple Event Streams 12
2.2.5. Pausing and Resuming Operation 12
3. The Spring JMS Input and Output Adapter 15
3.1. Introduction 15
3.2. Engine Configuration 16
3.3. Input Adapter 16
3.3.1. Spring Configuration 16
3.3.2. JMS Message Unmarshalling 18
3.4. Output Adapter 18
3.4.1. Spring Configuration 18
3.4.2. JMS Message Marshalling 20
4. The AMQP Input and Output Adapter 23
4.1. Introduction 23
4.2. AMQPSink Operator 23
4.3. AMQPSource Operator 24
5. The Kafka Adapter 27
5.1. Classpath Setup 27
5.2. Imports Setup 27
5.3. Input Adapter 27
5.3.1. Input Adapter Configuration and Start 27
5.3.2. Kafka Connectivity 29
5.3.3. Controlling Input Adapter Operation 29
5.4. Output Adapter 33
5.4.1. Output Adapter Configuration and Start 33
5.4.2. Kafka Connectivity 34
5.4.3. Controlling Output Adapter Operation 35
6. The HTTP Adapter 37
6.1. Adapter Overview 37
6.2. Getting Started 37
6.2.1. Plugin Loader Configuration 37
6.2.2. Configuration and Starting via API 38
6.3. HTTP Input Adapter 39

6.3.1. HTTP Service 39
6.3.2. Get Handlers 39
6.4. HTTP Output Adapter 40
6.4.1. Triggered HTTP Get 40
7. The Socket Adapter 43
7.1. Getting Started 43
7.1.1. Plugin Loader Configuration 43
7.1.2. Configuration and Starting via API 44
7.2. Socket Service 44
7.2.1. Object Data Format 45
7.2.2. String CSV Data Format 46
7.2.3. String CSV Data Format With Property Order 47
8. The Relational Database Adapter 49
8.1. Adapter Overview 49
8.2. Getting Started 49
8.2.1. Plugin Loader Configuration 49
8.2.2. Configuration and Starting via API 50
8.3. JDBC Connections 50
8.4. Triggered DML Statement Execution 51
8.5. Triggered Update-Insert Execution 52
8.6. Executor Configuration 54
8.7. Reading and Polling Database Tables 54
8.7.1. Polling and Startup SQL Queries 54
9. XML and JSON Output 57
10. Additional Event Representations 59
10.1. Apache Axiom Events 59

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.

Note

Esper has a fully-evolved API and natively accepts various input event formats and natively produces various output event objects, as part of core Esper.

It is therefore not necessary to use any of the adapters listed herein. Simply use the public Esper APIs directly in your code.

Most adapters present their own configuration as well as API. Some adapters also provide operators for use in data flows.

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 1.1. Adapter State Transitions

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

Chapter 2. The File and CSV Input and Output Adapter

The file input and output adapter consists of:

1. File (including CSV) input and output utilizing data flow operators.
2. The CSV input adapter API.

2.1. Data Flow Operators

2.1.1. Introduction

In order to use the File source and sink data flow operators, add `esperio-csv-`*version*`.jar` to your classpath and import the operator package or class using the static or runtime configuration.

The following code snippet uses the runtime configuration API to import the File adapter classes:

```
epService.getEPAdministrator().getConfiguration()
   .addImport(FileSourceFactory.class.getPackage().getName() + ".*");
```

The File input and output adapter provides the following data flow operators:

Table 2.1. File Operators

Operator	Description
FileSink	Write events to a file. See *Section 2.1.2, "FileSink Operator"*.
FileSource	Read events from a file. See *Section 2.1.3, "FileSource Operator"*.

2.1.2. FileSink Operator

The FileSink operator receives input stream events, transforms events to comma-separated format and writes to a file.

The FileSink operator must have a single input stream.

The FileSink operator cannot declare any output streams.

Parameters for the FileSink operator are (required parameters are listed first):

Table 2.2. FileSink Parameters

Name	Description
file (required)	File name string. An absolute or relative file name.

Name	Description
classpathFile	Boolean indicator whether the file name is found in a classpath directory, false by default.
append	Boolean indicator whether to append or overwrite the file when it exists. false by default causes the existing file, if any, to be overwritten.

The following example declares a data flow that is triggered by `MyMapEventType` events from the event bus (type not declared here) and that writes to the file `output.csv` the CSV-formatted events:

```
create dataflow FileSinkSampleFlow
   EventBusSource -> outstream<MyMapEventType> {}
   FileSink(outstream) {
     file: 'output.csv',
     append: true
   }
```

2.1.3. FileSource Operator

The FileSource operator reads from files, transforms file data and populates a data flow instance with events.

The FileSource operator cannot declare any input streams.

The FileSource operator must have at least one output stream. You can declare additional output streams to hold beginning-of-file and end-of-file indication.

Parameters for the FileSource operator are listed below, with the required parameters listed first:

Table 2.3. FileSource Parameters

Name	Description
file (required, or provide `adapterInputSource`)	File name string
adapterInputSource (required, or provide `file`)	An instance of `AdapterInputSource` if a file name cannot be provided.
classpathFile	Boolean indicator whether the file is found in a classpath directory, false by default.
dateFormat	The format to use when parsing dates; the default is `SimpleDateFormat` of `yyyy-MM-dd'T'HH:mm:ss.SSS` for `Date` and `Calendar` type properties.
format	Specify `csv` (the default) for comma-separate value or `line` for single-line.

Name	Description
hasTitleLine	For use with the `csv` format, boolean indicator whether a title line exists that the operator should read and parse to obtain event property names.
hasHeaderLine	For use with the `csv` format, boolean indicator whether a header line exists that the operator should skip.
numLoops	For use with the `csv` format, number of loops, an integer value that instructs the engine to restart reading the file upon encountering EOF, defaults to zero.
propertyNames	For use with the `csv` format, string array with a list of property names in the same order they appear in the file.
propertyNameLine	For use with the `line` format, specifies the property name of the output event type that receives the line text of type string.
propertyNameFile	For use with the `line` format, specifies the property name of the output event type(s) that receive the file name of type string.

The first output stream holds per-line output events. For use with the `line` format and if declaring two output streams, the second stream holds end-of-file indication. If declaring three output streams, the second stream holds beginning-of-file indication and the third stream holds end-of-file indication.

The `line` format requires that the output stream's event type is an object-array event type that features a single string-type property that the operator populates with each line of the file.

The file name (or `adapterInputSource`) may point to a zip file. If the file name ends with the literal `zip` the operator opens the zip file and uses the first packaged file. All other parameters including the format parameter for CSV or line-formatting then apply to the zipped file.

This example defines a data flow that consists of two operators that work together to read a file and send the resulting events into the engine:

```
create dataflow SensorCSVFlow
  FileSource -> sensorstream<TemperatureEventStream> {
    file: 'sensor_events.csv',
    propertyNames: ['sensor','temp','updtime'],
    numLoops: 3
  }
  EventBusSink(sensorstream){}
```

The data flow above configures the `FileSource` operator to read the file `sensor_events.csv`, populate the `sensor`, `temp` and `updtime` properties of the `TemperatureEventStream` event type (type definition not shown here) and make the output events available within the data flow under the name `sensorstream`.

The data flow above configures the `EventBusSource` operator to send the `sensorstream` events into the engine for processing.

2.1.3.1. FileSource Operator Detailed Example

This example shows the EPL and code to read and count lines in text files.

Below EPL defines an event type to each hold the file line text as well as to indictate the beginning and end of a file (remove the semicolon if creating EPL individually and not as a module):

```
// for beginning-of-file events
create objectarray schema MyBOF (filename string);
// for end of file events
create objectarray schema MyEOF (filename string);
// for line text events
create objectarray schema MyLine (filename string, line string);
```

The next EPL statements count lines per file outputting the final line count only when the end-of-file is reached.

```
// Initiate a context partition for each file, terminate upon end-of-file
create context FileContext
  initiated by MyBOF as mybof
  terminated by MyEOF(filename=mybof.filename);

// For each file, count lines
context FileContext
  select context.mybof.filename as filename, count(*) as cnt
  from MyLine(filename=context.mybof.filename)
  output snapshot when terminated;
```

The below EPL defines a data flow that reads text files line-by-line and that send events into the engine for processing.

```
create dataflow MyEOFEventFileReader
  FileSource -> mylines<MyLine>, mybof<MyBOF>, myeof<MyEOF> {
    format: 'line',
   propertyNameLine: 'line',     // store the text in the event property 'line'
    propertyNameFile: 'filename'   // store the file name in 'filename'
  }
  EventBusSink(mylines, mybof, myeof) {}  // send events into engine
```

The next sample code instantiates and runs data flows passing a file name:

```
EPDataFlowInstantiationOptions options = new EPDataFlowInstantiationOptions();
options.addParameterURI("FileSource/file", "myfile.txt");
EPDataFlowInstance instance = epService.getEPRuntime().getDataFlowRuntime()
    .instantiate("MyEOFEventFileReader",options);
instance.run();
```

2.2. CSV Input Adapter API

This chapter discusses the CSV input adapter API. 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.csv.AdapterInputSource`. Please consult the JavaDoc for additional information on `AdapterInputSource` and the CSV adapter.

2.2.1. 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

2.2.2. 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.addEventType("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.

2.2.2.1. 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 event type name for the Java class and provide the same name 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.addEventType("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();
```

When using JavaBean POJO Events, the event properties types are known from the underlying event type configuration. The CSV file row header does not need to define column type explicitly.

2.2.2.2. Dealing with event with nested properties

Wether you use JavaBean POJO or Map-based event types, EsperIO provides support for nested event properties up to one level of nesting. The row header must then refer to the properties using a `propertyName.nestedPropertyName` syntax. There is no support for mapped or indexed properties.

For example consider the following:

```
public class Point {
 int x;
 int y;

 // with getters & setters
}
public class Figure {
 String name;
 Point point; // point.x and point.y are nested properties

 //with getters & setters
}
```

Or the equivalent representation with nested maps, assuming "Figure" is the declared event type name, the CSV file can contain the following row header:

```
name, point.x, point.y
```

2.2.3. 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
```

2.2.3.1. 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.

2.2.4. 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.

You may not set an event rate per second when using a timestamp column and time-order.

2.2.5. 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 Adapter

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 name matching the event objects or the Map-event type names 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.jms.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 name out of the message and send to the engine via `sendEvent(name, 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="eventTypeName" value="MyOutputStream"/>
  </bean>

  <bean id="subscriptionTwo" class="com.espertech.esperio.jms.JMSSubscription">
    <property name="eventTypeName" 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 name 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 name to be configured with the Engine instance, e.g.:

```
<esper-configuration>
  <event-type name="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. The AMQP Input and Output Adapter

This chapter discusses the input and output adapters for AMQP. AMQP input and output utilizes data flow operators.

4.1. Introduction

In order to use the AMQP data flow operators, add `esperio-amqp-`*version*`.jar` to your classpath and import the operator package or class using the static or runtime configuration.

The following code snippet uses the runtime configuration API to import the AMQP adapter classes:

```
epService.getEPAdministrator().getConfiguration()
  .addImport(AMQPSource.class.getPackage().getName() + ".*");
```

The AMQP input and output adapter provides the following data flow operators:

Table 4.1. AMQP Operators

Operator	Description
AMQPSink	Send messages to an AMQP broker. See *Section 4.2, "AMQPSink Operator"*.
AMQPSource	Receive messages from an AMQP broker. See *Section 4.3, "AMQPSource Operator"*.

4.2. AMQPSink Operator

The AMQPSink operator receives input stream events, transforms events to AMQP messages and sends messages into an AMQP queue.

The AMQPSink operator must have a single input stream.

The AMQPSink operator cannot declare any output streams.

Parameters for the AMQPSink operator are:

Table 4.2. AMQPSink Parameters

Name	Description
host (required)	Host name string
queueName	Queue name string

Name	Description
collector (required)	Transformation class or instance for events to AMQP message
port	Port number integer
username	User name string
password	Password string (also see systemProperties or data flow instantiation options)
vhost	Vhost string
exchange	Exchange name string
routingKey	Routing key string
logMessage	Boolean indicator whether to log a text for each message
waitMSecNextMsg	Number of milliseconds wait between messages, a long-typed value
declareDurable	Boolean indicator whether durable, false by default
declareExclusive	Boolean indicator whether exclusive, false by default
declareAutoDelete	Boolean indicator whether auto-delete, true by default
declareAdditionalArgs	Map of additional arguments passed to AMQP of type Map<String, Object>

Either the `queueName` or the combination of `exchange` and `routingKey` are required parameters.

The collector is required and must be specified to transform events to AMQP messages. The collector instance must implement the interface `ObjectToAMQPCollector`. The adapter provides a default implementation `ObjectToAMQPCollectorSerializable` that employs default serialization.

The following example declares a data flow that is triggered by `MyMapEventType` events from the event bus (type not declared here) that sends serialized messages to an AMQP queue:

```
create dataflow AMQPOutgoingDataFlow
   EventBusSource -> outstream<MyMapEventType> {}
   AMQPSink(outstream) {
     host: 'localhost',
     queueName: 'myqueue',
     collector: {class: 'ObjectToAMQPCollectorSerializable'}
   }
```

4.3. AMQPSource Operator

The AMQPSource operator receives AMQP messages from a queue, transforms messages and populates a data flow instance with events.

The AMQPSource operator cannot declare any input streams.

The AMQPSource operator must have a single output stream.

Parameters for the AMQPSource operator are listed below, with the required parameters listed first:

Table 4.3. AMQPSource Parameters

Name	Description
host (required)	Host name string
queueName (required)	Queue name string
collector (required)	Transformation class or instance for AMQP message to underlying event transformation
port	Port number integer
username	User name string
password	Password string (also see systemProperties or data flow instantiation options)
vhost	Vhost string
exchange	Exchange name string
routingKey	Routing key string
logMessage	Boolean indiator whether to log a text for each message
waitMSecNextMsg	Number of milliseconds wait between messages, a long-typed value
declareDurable	Boolean indicator whether durable, false by default
declareExclusive	Boolean indicator whether exclusive, false by default
declareAutoDelete	Boolean indicator whether auto-delete, true by default
declareAdditionalArgs	Map of additional arguments passed to AMQP of type Map<String, Object>
prefetchCount	Prefetch count integer, defaults to 100
consumeAutoAck	Boolean indicator whether to auto-ack, true by default

The collector is required and must be specified to transform AMQP messages to events. The collector instance must implement the interface `AMQPToObjectCollector`. The adapter provides a default implementation `AMQPToObjectCollectorSerializable` that employs default serialization.

The following example declares a data flow that is receives AMQP messages from a queue, transforms each message and sends each message of type `MyMapEventType` into the event bus:

```
create dataflow AMQPIncomingDataFlow
```

```
  AMQPSource -> outstream<MyMapEventType> {
    host: 'localhost',
    queueName: 'myqueue',
    collector: {class: 'AMQPToObjectCollectorSerializable'},
    logMessages: true
  }
  EventBusSink(outstream){}
```

Chapter 5. The Kafka Adapter

This chapter discusses the EsperIO Kafka input adapter.

This input adapter is for receiving events and event or engine time from Kafka topics.

The scope of this input adapter is a local reader and is not meant for coordinated use by multiple servers, which is the scope of Esper Enterprise Edition. Please see Esper Enterprise Edition for information on the horizontal scale-out architecture based on Kafka (the scope of this input adapter is NOT horizontal scale-out).

5.1. Classpath Setup

Please add the `esperio-kafka-version.jar` jar file to your classpath.

Please also add `kafka-clients-`*version*`.jar` and the Kafka client dependencies to your classpath.

The EsperIO Kafka input adapter supports the new Kafka consumer only and requires Kafka client version 0.10.1.0 and higher.

5.2. Imports Setup

For use with the Kafka output adapter, and when using the `KafkaOutputDefault` annotation, please add the `KafkaOutputDefault` import. For example:

```
configuration.addImport(KafkaOutputDefault.class);
```

5.3. Input Adapter

5.3.1. Input Adapter Configuration and Start

You may configure and start the EsperIO Kafka input adapter either as part of your Esper configuration file in the plugin loader section or via the adapter API.

The following example shows an Esper configuration file with all properties:

```
<?xml version="1.0" encoding="UTF-8"?>
<esper-configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.espertech.com/schema/esper"
  xsi:noNamespaceSchemaLocation="../../esper/etc/esper-configuration-6-0.xsd">

                    <plugin-loader          name="KafkaInput"          class-
name="com.espertech.esperio.kafka.EsperIOKafkaInputAdapterPlugin">
    <!--
      Kafka Consumer Properties: Passed-Through to Kafka Consumer.
    -->
```

```
    <init-arg name="bootstrap.servers" value="127.0.0.1:9092"/>
                                        <init-arg         name="key.deserializer"
 value="org.apache.kafka.common.serialization.StringDeserializer"/>
                                      <init-arg         name="value.deserializer"
 value="com.mycompany.MyCustomDeserializer"/>
    <init-arg name="group.id" value="my_group_id"/>

    <!--
       EsperIO Kafka Input Properties: Define subscription, topics, processor
 and timestamp extractor.
    -->
                            <init-arg      name="esperio.kafka.input.subscriber"
 value="com.espertech.esperio.kafka.EsperIOKafkaInputSubscriberByTopicList"/>
    <init-arg name="esperio.kafka.topics" value="my_topic"/>
                             <init-arg       name="esperio.kafka.input.processor"
 value="com.espertech.esperio.kafka.EsperIOKafkaInputProcessorDefault"/>
                    <init-arg     name="esperio.kafka.input.timestampextractor"

>
  </plugin-loader>
</esper-configuration>
```

Alternatively the equivalent API calls to configure the adapter are:

```
Properties props = new Properties();

// Kafka Consumer Properties
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
 org.apache.kafka.common.serialization.StringDeserializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
 com.mycompany.MyCustomDeserializer.class.getName());
props.put(ConsumerConfig.GROUP_ID_CONFIG, "my_group_id");

// EsperIO Kafka Input Adapter Properties
props.put(EsperIOKafkaConfig.INPUT_SUBSCRIBER_CONFIG,
 EsperIOKafkaInputSubscriberByTopicList.class.getName());
props.put(EsperIOKafkaConfig.TOPICS_CONFIG, "my_topic");
props.put(EsperIOKafkaConfig.INPUT_PROCESSOR_CONFIG,
 EsperIOKafkaInputProcessorDefault.class.getName());
props.put(EsperIOKafkaConfig.INPUT_TIMESTAMPEXTRACTOR_CONFIG,
 EsperIOKafkaInputTimestampExtractorConsumerRecord.class.getName());

Configuration config = new Configuration();
config.addPluginLoader("KafkaInput",
 EsperIOKafkaInputAdapterPlugin.class.getName(), props, null);
```

By adding the plug-in loader to the configuration as above the engine automatically starts the adapter as part of engine initialization.

Alternatively, the adapter can be started and stopped programatically as follows:

```
// start adapter
EsperIOKafkaInputAdapter adapter = new EsperIOKafkaInputAdapter(props,
 "default");
adapter.start();

// destroy the adapter when done
adapter.destroy();
```

5.3.2. Kafka Connectivity

All properties are passed to the Kafka consumer. This allows your application to add additional properties that are not listed here and according to Kafka consumer documentation.

Required properties are below. `ConsumerConfig` is part of the Kafka API in `org.apache.kafka.clients.consumer.ConsumerConfig`.

Table 5.1. Kafka Consumer Required Properties

Name	API Name	Description
`bootstrap.servers`	`ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG`	Kafka bootstrap server list.
`key.deserializer`	`ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG`	Fully-qualified class name of Kafka message key deserializer.
`value.deserializer`	`ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG`	Fully-qualified class name of Kafka message value deserializer.
`group.id`	`ConsumerConfig.GROUP_ID_CONFIG`	Application consumer group id.

5.3.3. Controlling Input Adapter Operation

The input adapter operation depends on the *subscriber* and *processor*.

The *subscriber* is responsible for calling Kafka consumer subscribe methods, i.e. calls Kafka API `consumer.subscribe(...)`.

The *processor* is responsible for processing Kafka API `ConsumerRecords` messages, i.e. implements `process(ConsumerRecords records)`.

Properties that define the subscriber and consumer are below. `EsperIOKafka` is part of the EsperIO Kafka API in `com.espertech.esperio.kafka.EsperIOKafkaConfig`.

Table 5.2. Kafka Input Adapter Properties

Name	API Name	Description
`esperio.kafka.input.subscriber`	`EsperIOKafkaConfig.INPUT_SUBSCRIBER_CONFIG`	Required property. Fully-qualified class name of subscriber that subscribes to topics and partitions. The class must implement the interface `EsperIOKafkaInputSubscriber`. You may use `com.espertech.esperio.kafka.EsperIOKafkaI` and provide a topic list in `esperio.kafka.topics`.
`esperio.kafka.topics`	`EsperIOKafkaConfig.TOPICS_CONFIG`	Optional property and only required if the subscriber is `EsperIOKafkaInputSubscriberByTopicList`. Specifies a comma-separated list of topic names to subscribe to.
`esperio.kafka.input.processor`	`EsperIOKafkaConfig.INPUT_PROCESSOR_CONFIG`	Required property. Fully-qualified class name of the Kafka consumer records processor that sends events into the engine and may advance engine time. The class must implement the interface `EsperIOKafkaInputProcessor`. You may use `com.espertech.esperio.kafka.EsperIOKafkaI` for default event and time processing.
`esperio.kafka.input.timestampextractor`	`EsperIOKafkaConfig.INPUT_TIMESTAMPEXTRACTOR_CONFIG`	Optional property. Fully-qualified class name of the Kafka message timestamp extractor that extracts a long-typed timestamp from a

Name	API Name	Description
		consumer record, for use as time. The class must implement the interface `EsperIOKafkaInputTimestampExtractor`. You may use `com.espertech.esperio.kafka.EsperIOKafkaI` which returns the time of each consumer record that is part of the consumer record.

5.3.3.1. Subscriber

The subcriber is responsible for calling `consumer.subscribe(...)`.

The adapter provides a default implementation by name `EsperIOKafkaInputSubscriberByTopicList`. Your application may provide its own subscriber by implementing the simple `EsperIOKafkaInputSubscriber` interface.

This default implementation takes the value of `esperio.kafka.topics` and subscribes to each topic.

For reference, we provide the code of the default subscriber below (repository or source jar for full code):

```
public      class      EsperIOKafkaInputSubscriberByTopicList      implements
 EsperIOKafkaInputSubscriber {
  public void subscribe(EsperIOKafkaInputSubscriberContext context) {
                                        String          topicsCSV          =

 EsperIOKafkaConfig.TOPICS_CONFIG);
    String[] topicNames = topicsCSV.split(",");
    List<String> topics = new ArrayList<>();
    for (String topicName : topicNames) {
      if (topicName.trim().length() > 0) {
        topics.add(topicName.trim());
      }
    }
    context.getConsumer().subscribe(topics);
  }
}
```

5.3.3.2. Processor

The processor is responsible for processing Kafka API `ConsumerRecords`.

The adapter provides a default implementation by name `EsperIOKafkaInputProcessorDefault`. Your application may provide its own processor by implementing the simple `EsperIOKafkaInputProcessor` interface.

This default processor can be configured with an optional timestamp extractor that obtains a timestamp for each consumer record. If no timestamp extractor is configured, the default processor does not advance time.

For reference, we provide the (slightly simplified) code of the default processor below (repository or source jar for full code):

```
public        class        EsperIOKafkaInputProcessorDefault        implements
 EsperIOKafkaInputProcessor {

  private EPServiceProvider engine;
  private EsperIOKafkaInputTimestampExtractor timestampExtractor;

  public void init(EsperIOKafkaInputProcessorContext context) {
    this.engine = context.getEngine();

                              String       timestampExtractorClassName       =
 context.getProperties().getProperty(EsperIOKafkaConfig.INPUT_TIMESTAMPEXTRACTOR_CONFIG);
    if (timestampExtractorClassName != null) {
                timestampExtractor   =   (EsperIOKafkaInputTimestampExtractor)

 timestampExtractorClassName);
    }
  }

  public void process(ConsumerRecords<Object, Object> records) {
    for (ConsumerRecord record : records) {

      if (timestampExtractor != null) {
        long timestamp = timestampExtractor.extract(record);
        // advances engine time
        engine.getEPRuntime().sendEvent(new CurrentTimeSpanEvent(timestamp));
      }

      if (record.value() != null) {
        engine.getEPRuntime().sendEvent(record.value());
      }
    }
  }
```

```
  public void close() {}
}
```

The default processor takes the message value and sends it as an event into the engine. The default processor takes the extracted time, if a timestamp extractor is provided, and sends a time span event to the engine to advance engine time.

You must provide your own processor if any additional event transformation is required or if using `epRuntime.send(Map/ObjectArray/Node)` or if the default behavior does not fit for other reasons.

5.4. Output Adapter

5.4.1. Output Adapter Configuration and Start

You may configure and start the EsperIO Kafka output adapter either as part of your Esper configuration file in the plugin loader section or via the adapter API.

The following example shows an Esper configuration file with all properties:

```
<?xml version="1.0" encoding="UTF-8"?>
<esper-configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.espertech.com/schema/esper"
  xsi:noNamespaceSchemaLocation="../../esper/etc/esper-configuration-6-0.xsd">

                    <plugin-loader          name="KafkaOutput"          class-
name="com.espertech.esperio.kafka.EsperIOKafkaOutputAdapterPlugin">
    <!--
      Kafka Producer Properties: Passed-Through to Kafka Consumer.
    -->
    <init-arg name="bootstrap.servers" value="127.0.0.1:9092"/>
                                       <init-arg          name="key.serializer"
 value="org.apache.kafka.common.serialization.StringSerializer"/>
   <init-arg name="value.serializer" value="com.mycompany.MyCustomSerializer"/>

    <!--
      EsperIO Kafka Output Properties: Define a flow controller.
    -->
                      <init-arg      name="esperio.kafka.output.flowcontroller"

>
    <init-arg name="esperio.kafka.topics" value="my_topic"/>
  </plugin-loader>
</esper-configuration>
```

Alternatively the equivalent API calls to configure the adapter are:

```
Properties props = new Properties();

// Kafka Producer Properties
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
 org.apache.kafka.common.serialization.StringSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
 org.apache.kafka.common.serialization.StringSerializer.class.getName());

// EsperIO Kafka Output Adapter Properties
props.put(EsperIOKafkaConfig.OUTPUT_FLOWCONTROLLER_CONFIG,
 EsperIOKafkaOutputFlowControllerByAnnotatedStmt.class.getName());
props.put(EsperIOKafkaConfig.TOPICS_CONFIG, "my_topic");

Configuration config = new Configuration();
config.addPluginLoader("KafkaOutput",
 EsperIOKafkaOutputAdapterPlugin.class.getName(), props, null);
```

By adding the plug-in loader to the configuration as above the engine automatically starts the adapter as part of engine initialization.

Alternatively, the adapter can be started and stopped programatically as follows:

```
// start adapter
EsperIOKafkaOutputAdapter adapter = new EsperIOKafkaOutputAdapter(props,
 "default");
adapter.start();

// destroy the adapter when done
adapter.destroy();
```

5.4.2. Kafka Connectivity

All properties are passed to the Kafka producer. This allows your application to add additional properties that are not listed here and according to Kafka producer documentation.

Required properties are below. `ProducerConfig` is part of the Kafka API in `org.apache.kafka.clients.producer.ProducerConfig`.

Table 5.3. Kafka Producer Required Properties

Name	API Name	Description
`bootstrap.servers`	`ProducerConfig.BOOTSTRAP_SERVERS_CONFIG`	Kafka bootstrap server list.
`key.serializer`	`ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG`	Fully qualified class name of Kafka message key serializer.

Name	API Name	Description
`value.serializer`	`ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG`	Fully-qualified class name of Kafka message value serializer.

5.4.3. Controlling Output Adapter Operation

The output adapter operation depends on the *flow controller*, which is reponsible for attaching listeners to statements that send messages to Kafka topics.

Properties that define the flow controller are below. `EsperIOKafka` is part of the EsperIO Kafka API in `com.espertech.esperio.kafka.EsperIOKafkaConfig`.

Table 5.4. Kafka Output Adapter Properties

Name	API Name	Description
`esperio.kafka.output.flowcontroller`	`EsperIOKafkaConfig.OUTPUT_FLOWCONTROLLER_CONFIG`	Required property. Fully-qualified class name of flow controller that produces messages. The class must implement the interface `EsperIOKafkaOutputFlowController`. You may use `com.espertech.esperio.kafka.EsperIOKafkaO` and provide a topic list in `esperio.kafka.topics`.
`esperio.kafka.topics`	`EsperIOKafkaConfig.TOPICS_CONFIG`	Specifies a comma-separated list of topic names to produce to, for use with the above flow controller and not required otherwise.

5.4.3.1. Flow Controller

The flow controller is responsible for allocating a `KafkaProducer` and associating statement listeners to the producer, for listeners to send messages to Kafka topics.

The adapter provides a default implementation by name `EsperIOKafkaOutputFlowControllerByAnnotatedStmt`. Your application may provide its own subscriber by implementing the simple `EsperIOKafkaOutputFlowControllerContext` interface.

5.4.3.1.1. Default Flow Controller

`EsperIOKafkaOutputFlowControllerByAnnotatedStmt`

The flow controller takes the value of `esperio.kafka.topics` and produces a message to each topic for each statement listener output event.

The flow controller attaches a listener to all statements that have the `@KafkaOutputDefault` annotation. Please ensure that the annotation is part of your imports. The adapter considers all newly-created statements that have the annotation.

Thus please create the EPL as follows:

```
@KafkaOutputDefault select * from ......
```

The flow controller produces JSON output. It uses the engine JSON renderer that can be obtained from `epService.getEPRuntime().getEventRenderer().getJSONRenderer(statement.getEventType());`.

The statement listeners that the flow controller attaches do not provide a key or partition id to the producer. The listeners simply invoke `new ProducerRecord(topic, json)` for output event and each topic. The value serializer must be the string serializer.

For reference, please find the source code of the flow controller in the repository.

Chapter 6. The HTTP Adapter

This chapter discusses the EsperIO HTTP adapter.

6.1. Adapter Overview

The EsperIO HTTP input and output adapter can be used to send events into an Esper engine instance as well as perform HTTP requests triggered by output events generated by an Esper engine instance.

To send events into an Esper engine instance for processing you declare an HTTP service, which causes the adapter to expose an HTTP protocol server on the configured port to handle incoming requests. Your configuration then attaches Get handlers that receive Get requests that post events into the engine with data from each request.

Output events generated by an Esper engine instance can trigger an HTTP Get operation to a URI of your choice. For this purpose define a triggering event stream and the desired target URI and parameters.

6.2. Getting Started

You may configure the EsperIO HTTP adapter either as part of your Esper configuration file in the plugin loader section or via the adapter API. Add the `esperio-http-version.jar` file to your classpath.

For input adapter operation, add the `httpcore-`*version*`.jar` to your classpath. If using Java NIO add the `httpcore-nio-`*version*`.jar` to your classpath in addition.

For output adapter operation, add the `httpclient-`*version*`.jar` to your classpath.

A sample adapter configuration file is provided in `esperio-http-sample-config.xml` in the `etc` folder of the distribution. A configuration file must be valid according to schema `esperio-http-configuration-6-0.xsd`.

6.2.1. Plugin Loader Configuration

You may place the configuration XML directly into your Esper configuration file as the example below shows:

```
<esper-configuration>
  <plugin-loader name="EsperIOHTTPAdapter"
      class-name="com.espertech.esperio.http.EsperIOHTTPAdapterPlugin">
    <config-xml>
      <esperio-http-configuration>
```

```
          .....as outlined below or contents of esperio-http-sample-config.xml
 here...
      </esperio-http-configuration>
    </config-xml>
  </plugin-loader>
</esper-configuration>
```

Alternatively you can provide a URL in the Esper configuration file to point to your adapter configuration file:

```
<esper-configuration>
  <plugin-loader name="EsperIOHTTPAdapter"
      class-name="com.espertech.esperio.http.EsperIOHTTPAdapterPlugin">
      <init-arg name="esperio.http.configuration.file"
          value="file:/path/esperio-http-sample-config.xml" />
  </plugin-loader>
</esper-configuration>
```

6.2.2. Configuration and Starting via API

If using Spring or if your application requires API access, the following code snippet configures and starts the adapter via API.

The class for configuring an EsperIO HTTP adapter is `com.espertech.esperio.http.config.ConfigurationHTTPAdapter`. The adapter class itself is `EsperIOHTTPAdapter`.

The code snippet below is a sample that configures using driver manager and starts the adapter via API:

```
ConfigurationHTTPAdapter adapterConfig = new ConfigurationHTTPAdapter();

// add additional configuration
Request request = new Request();
request.setStream("TriggerEvent");
request.setUri("http://localhost:8077/root");
adapterConfig.getRequests().add(request);

// start adapter
EsperIOHTTPAdapter   httpAdapter   =   new   EsperIOHTTPAdapter(adapterConfig,
 "engineURI");
httpAdapter.start();

// destroy the adapter when done
httpAdapter.destroy();
```

6.3. HTTP Input Adapter

6.3.1. HTTP Service

A service is required for the adapter to receive events via a HTTP client connection.

The synopsis is as follows:

```
<esperio-http-configuration>
  <service name="[name]" port="[port]" [nio="true|false"]/>
  <!-- add additional configuration here -->
</esperio-http-configuration>
```

The *name* attribute value is required and provides the name of the HTTP service for use in logging and for get-handlers as described below.

The *nio* attribute is optional and can be used to enable Java NIO (disabled by default).

If configuring via the adapter API or Spring, use the `com.espertech.esperio.http.config.Service` class.

An example XML to configure a service and single get-handler is:

```
<esperio-http-configuration>
  <service name="myservice" port="8079" nio="false"/>
  <get service="myservice" pattern="*"/>
</esperio-http-configuration>
```

6.3.2. Get Handlers

One or more handlers for HTTP Get operations can be installed for a service and are used to receive events.

Define a `get` element in the adapter configuration file (or use the `GetRequest` class) for every handler to register for a service.

The synopsis is as follows:

```
<get service="[service]" pattern="[pattern]"/>
```

The *service* attribute value is required and provides the name of the HTTP service to register the Get operation handler for.

A value for the *pattern* attribute is required and may be either `*` for all URIs, `*[uri]` for all URIs ending with the given URI or `[uri]*` for all URI starting with the given URI.

A sample Get-handler configuration follows:

```
<get service="myservice" pattern="*"/>
```

When posting events to the engine, the Get request URI must contain a `stream` parameter that carries the name of the stream (event type) to insert into. Each event property to be populated in the input event must be part of the Get request parameter values.

For example, the URI `http://localhost:8079/sendevent?stream=MyFirewallEvent&name=Joe&changed=true` entered into a browser sends an input event of type `MyFirewallEvent` setting the `name` property of the event to "Joe" and the `changed` property of the event to true.

Note that if your target type is a Java object event, your event class must provide setter-methods according to JavaBean conventions. The event class should also provide a default constructor taking no parameters. If your event class does not have a default constructor, your application may configure a factory method via `ConfigurationEventTypeLegacy`.

6.4. HTTP Output Adapter

6.4.1. Triggered HTTP Get

This facility instructs the adapter to perform an HTTP Get request when a triggering event occurs, passing event properties as URI parameters.

Define a `request` element in the adapter configuration file (or use the `Request` class) for every HTTP Get to execute.

The synopsis is as follows:

```
<request stream="[stream]" uri="[uri_with_placeholders]"/>
```

A value for the *stream* attribute is required and provides the name of the stream that triggers the HTTP Get. The adapter expects a stream by this name to exist at adapter start time.

The *uri_with_placeholders* attribute value is required. You may place event property placeholders inside the URI to format the URI as needed. Placeholders are of the format `${`*property_name*`}`.

A sample request configuration follows:

```
<request    stream="TriggerFirewallStream"    uri="http://myremotehost:80/root/
event"/>
```

Assuming the `HttpTriggerStream` has event properties `name` and `ipaddress` then a sample Get request URI is as follows:

```
http://myremotehost:80/root/event?
stream=TriggerFirewallStream&name=Joe&ipaddress=120.1.0.0
```

You may parameterize the URI via placeholders by placing `${`*property_name*`}` and the special placeholder `${stream}` into the URI string.

The next example configuration defines URI parameters via placeholder:

```
<request    stream="TriggerFirewallStream"    uri="http://myremotehost:80/root/
${stream}?violation&amp;name=${name};violationip=${ipaddress}"/>
```

The URI generated by the adapter:

```
http://myremotehost:80/root/TriggerFirewallStream?
violation&name=Joe&violationip=120.1.0.0
```

Chapter 7. The Socket Adapter

This chapter discusses the EsperIO Socket adapter.

The EsperIO Socket input adapter can be used to send events into an Esper engine instance via socket client, either as Java objects or as CSV name-value pair strings.

7.1. Getting Started

You may configure the EsperIO Socket adapter either as part of your Esper configuration file in the plugin loader section or via the adapter API. Add the `esperio-socket-version.jar` file to your classpath. There are no other dependent jar files required.

A sample adapter configuration file is provided in `esperio-socket-sample-config.xml` in the `etc` folder of the distribution. A configuration file must be valid according to schema `esperio-socket-configuration-6-0.xsd`.

7.1.1. Plugin Loader Configuration

You may place the configuration XML directly into your Esper configuration file as the example below shows:

```
<esper-configuration>
  <plugin-loader name="EsperIOSocketAdapter"
      class-name="com.espertech.esperio.socket.EsperIOSocketAdapterPlugin">
    <config-xml>
      <esperio-socket-configuration>
     <socket name="mysocketOne" port="7101" data="object"/>
        <socket  name="mysocketTwo"  port="7102"  data="csv"  hostname="myhost"
 backlog="20" unescape="true"/>
      </esperio-socket-configuration>
    </config-xml>
  </plugin-loader>
</esper-configuration>
```

Alternatively you can provide a URL in the Esper configuration file to point to your adapter configuration file:

```
<esper-configuration>
  <plugin-loader name="EsperIOSocketAdapter"
      class-name="com.espertech.esperio.socket.EsperIOSocketAdapterPlugin">
      <init-arg name="esperio.socket.configuration.file"
          value="file:/path/esperio-socket-sample-config.xml" />
  </plugin-loader>
```

```
</esper-configuration>
```

7.1.2. Configuration and Starting via API

If using Spring or if your application requires API access, the following code snippet configures and starts the adapter via API.

The class for configuring an EsperIO Socket adapter is `com.espertech.esperio.socket.config.ConfigurationSocketAdapter`. The adapter class itself is `EsperIOSocketAdapter`.

The code snippet below is a sample that configures using driver manager and starts the adapter via API:

```
ConfigurationSocketAdapter adapterConfig = new ConfigurationSocketAdapter();

SocketConfig socket = new SocketConfig();
socket.setDataType(DataType.CSV);
socket.setPort(port);
adapterConfig.getSockets().put("SocketService", socket);

// start adapter
EsperIOSocketAdapter socketAdapter = new EsperIOSocketAdapter(adapterConfig,
 "engineURI");
socketAdapter.start();

// destroy the adapter when done
socketAdapter.destroy();
```

7.2. Socket Service

Add a socket configuration for each unique port that you want to expose a socket receive service for use by socket client connections.

The synopsis is as follows:

```
<esperio-socket-configuration>
  <socket name="[name]" port="[port]" data="[csv|object|property_ordered_csv]"
    [hostname="hostname"] [backlog="backlog"] [unescape="true|false"]/>
</esperio-socket-configuration>
```

The required *name* attribute provides the name of the socket service for use in logging.

The required *port* attribute provides the port that the socket service accepts client connections.

The required *data* attribute specifies whether the data arriving through the socket is formatted as a Java binary object stream or as CSV string values.

The optional *hostname* attribute can provide the host name passed to the server socket (`ServerSocket`).

The optional *backlog* attribute can provide the backlog number of connections passed to the server socket. This number defaults to 2 when a host name is passed but no backlog is provided.

The optional *unescape* attribute is false by default. When false the adapter does not unescape (Java escape rules) values. When true the adapter performs an unescape on all values.

If configuring via the adapter API or Spring, use the `com.espertech.esperio.socket.config.SocketConfig` class.

7.2.1. Object Data Format

When sending events as Java objects, configure the `data` attribute value to `object` and use `ObjectOutputStream` to write events to the client socket. When sending a `java.util.Map` event, your Map must contain a String value for the key `stream` which must denote a configured Map event type.

This example XML configures a socket accepting client connections that provide Java objects:

```
<esperio-socket-configuration>
  <socket name="objectStreamSocket" port="8079" data="object"/>
</esperio-socket-configuration>
```

When `object` data type is configured, clients connections are expected to send `java.io.Serializable` or `java.io.Externalizable` objects using `ObjectOutputStream`.

Below is a block of sample code that for use in clients to the adapter. It assumes the `MyEvent` class implements either of the above interfaces:

```
// connect first
Socket requestSocket = new Socket("localhost", port);
ObjectOutputStream                  out                  =                  new
 ObjectOutputStream(requestSocket.getOutputStream());

// send a few events, here we send only one
out.writeObject(new MyEvent("Hello World"));
out.flush();

// Consider resetting the output stream from time-to-time, after sending a number
 of objects.
// This is because the stream may cache strings etc. . The reset is:
// out.reset();
```

```
// close when done
out.close();
requestSocket.close();
```

7.2.2. String CSV Data Format

When sending events as CSV strings, the format of the string should be:

```
stream=[type],[name]=[value] [,...] (newline)
```

The CSV string must end with a newline character: Each event is one line. Each CSV element must be in the `[name]=[value]` format. Your CSV must contain a value for `stream` which must denote a configured event type. The adapter parses each string value and populates an instance of the target type.

This next example XML configures a socket accepting client connections that provide events as CSV-formatted strings with name-value pairs :

```
<esperio-socket-configuration>
  <socket name="csvStreamSocket" port="8079" data="csv"/>
</esperio-socket-configuration>
```

A piece of client code that sends an event of type `MyEvent` may look as follows:

```
// connect first
String newline = System.getProperty("line.separator");
Socket requestSocket = new Socket("localhost", port);
BufferedWriter wr = new BufferedWriter(
    new OutputStreamWriter(socket.getOutputStream()));

// send a few events
wr.write("stream=MyEvent,price=20.d,upcCode=A0001");
wr.write(newline);
wr.flush();

// close when done
wr.close();
requestSocket.close();
```

Note that if your target type is a Java object event, your event class must provide setter-methods according to JavaBean conventions. The event class should also provide a default constructor taking no parameters. If your event class does not have a default constructor, your application may configure a factory method via `ConfigurationEventTypeLegacy`.

7.2.3. String CSV Data Format With Property Order

Similar to the string CSV data format as discussed earlier, this data format allows specifying a property order as well as the event type name.

The format of the string that represents an event is the CSV format (no `name=` or `stream=` texts are required):

```
value [, value [,...]] (newline)
```

This next example XML configures a socket accepting client connections that provide events as CSV-formatted strings with name-value pairs :

```
<esperio-socket-configuration>
     <socket  name="csvStreamSocket"  port="8079"  data="property_ordered_csv"
 stream="MyEvent" propertyOrder="price,upcCode"/>
</esperio-socket-configuration>
```

Set the `data` attribute to `property_ordered_csv`. The `stream` attribute is a required configuration and must contain the event type name. The `propertyOrder` attribute is also required and must contain the property names of properties of the event type, in the same order that the value for the property arrives on each line.

As part of the sample client code shown above, the following line sends an event with values `20.0` and `A0001`.

```
wr.write("20.0,A0001");
wr.write(newline);
```

Chapter 8. The Relational Database Adapter

This chapter discusses the EsperIO adapter for relational databases.

8.1. Adapter Overview

The EsperIO relational database adapter can write events to a database table.

If your application only reads from tables, the Esper jar file and Esper configuration suffices and there is no additional EsperIO DB adapter configuration or jar file required. Please see below tips for reading or polling tables.

The EsperIO DB adapter supports two means to write to a database table:

1. Execute a SQL DML (Data Manipulation, i.e. Update, Insert, Delete or stored procedure call) statement as a response to a triggering event.
2. Execute an Update-Insert: The adapter attempts an Update of a row by key and if unsuccessful (update returns zero rows updated) the adapter performs an Insert.

The adapter also provides infrastructure for queuing database write requests for execution by a thread pool.

8.2. Getting Started

You may configure the EsperIO DB adapter either as part of your Esper configuration file in the plugin loader section or via the adapter API. Add the `esperio-db-version.jar` file to your classpath as well as the JDBC driver. There are not other dependent jar files required by the adapter.

A sample adapter configuration file is provided in `esperio-db-sample-config.xml` in the `etc` folder of the distribution. A configuration file must be valid according to schema `esperio-db-configuration-6-0.xsd`.

8.2.1. Plugin Loader Configuration

You may place the configuration XML directly into your Esper configuration file as the example below shows:

```
<esper-configuration>
  <plugin-loader name="EsperIODBAdapter"
      class-name="com.espertech.esperio.db.EsperIODBAdapterPlugin">
    <config-xml>
      <esperio-db-configuration>
```

```
      .....as outlined below or contents of esperio-db-sample-config.xml here...
      </esperio-db-configuration>
    </config-xml>
  </plugin-loader>
</esper-configuration>
```

Alternatively you can provide a URL in the Esper configuration file to point to your adapter configuration file:

```
<esper-configuration>
  <plugin-loader name="EsperIODBAdapter"
      class-name="com.espertech.esperio.db.EsperIODBAdapterPlugin">
      <init-arg name="esperio.db.configuration.file"
          value="file:/path/esperio-db-sample-config.xml" />
  </plugin-loader>
</esper-configuration>
```

8.2.2. Configuration and Starting via API

If using Spring or if your application requires API access, the following code snippet configures and starts the adapter via API.

The class for configuring an EsperIO DB adapter is `com.espertech.esperio.db.config.ConfigurationDBAdapter`. The adapter class itself is `EsperIODBAdapter`.

The code snippet below is a sample that configures using driver manager and starts the adapter via API:

```
ConfigurationDBAdapter adapterConfig = new ConfigurationDBAdapter();
ConfigurationDBRef configDB = new ConfigurationDBRef();
configDB.setDriverManagerConnection("DRIVER", "URL", new Properties());
adapterConfig.getJdbcConnections().put("db1", configDB);

// add additional configuration such as DML and Upsert

// start adapter
EsperIODBAdapter dbAdapter = new EsperIODBAdapter(adapterConfig, "engineURI");
dbAdapter.start();
```

8.3. JDBC Connections

The configuration for the source of JDBC connections follows the Esper configuration. Please consult the Esper documentation or sample adapter configuration file for details.

Your configuration should set `auto-commit` to true thereby each change to database tables is automatically committed.

The adapter obtains a new connection for each operation and closes the connection after each operation. For optimum performance consider configuring a connection pool.

A sample JDBC connection configuration is shown in below XML. The API class is `ConfigurationDBRef` (an Esper core engine class). You may also configure a `DataSource` or `DataSource` factory as outlined in the Esper docs.

```
<esperio-db-configuration>
  <jdbc-connection name="db1">
               <drivermanager-connection    class-name="com.mysql.jdbc.Driver"
 url="jdbc:mysql://localhost/test"
       user="root" password="password">
    <connection-settings auto-commit="true" catalog="TEST"/>
  </jdbc-connection>
  <!-- Add DML and Upsert configurations here, as below. -->
</esperio-db-configuration>
```

8.4. Triggered DML Statement Execution

This facility allows running a SQL DML (Data Manipulation) query, i.e. an Update, Insert, Delete query or a stored procedure when an event in a triggering stream occurs.

Define a `dml` element in the adapter configuration file (or use the `DMLQuery` class) for every query to execute.

The synopsis is as follows:

```
<dml connection="[connection]" stream="[stream]"
     [name="name"] [executor-name="executor"] [retry="count"] [retry-interval-
sec="sec"]>
  <sql>[sql]</sql>
  <bindings>
    <parameter pos="[position]" property="[property_name]"/>
    [...parameters]
  </bindings>
</dml>
```

The *connection* attribute value is required and provides the name of the configured JDBC connection.

A value for the *stream* attribute is required and provides the name of the stream that triggers the DML. The adapter expects a stream by this name to exist at adapter start time.

The *name* attribute is optional and is only used for logging errors.

The *executor-name* attribute is optional. If specified, it must be the name of an executor configuration. If specified, the adapter will use the executor service (queue and thread pool) for performing all DML work. If not specified, the adapter performs the DML work in the same thread.

The *retry* attribute is optional. If specified, the adapter will retry a given number of times in case an error is encountered. If *retry-interval-sec* is specified, the adapter waits the given number of seconds between retries.

The *sql* element is required and provides the SQL DML or stored procedure call to execute, with parameters as question mark (?).

The *bindings* element is required and provides the bindings for expression parameters.

The *parameter* element should occur as often as there are parameters in the SQL query. The *position* attribute starts at 1 and counts up for each parameter. The *property* parameter provide the name of the event property of the stream to use as the parameter value.

A sample DML configuration follows:

```
<dml connection="db1" stream="InsertToDBStream"
      name="MyInsertQuery" executor-name="queue1" retry="count">
  <sql>insert into MyEventStore(key1, value1, value2) values (?, ?, ?)</sql>
  <bindings>
    <parameter pos="1" property="eventProperty1"/>
    <parameter pos="2" property="eventProperty2"/>
    <parameter pos="3" property="eventProperty3"/>
  </bindings>
</dml>
```

8.5. Triggered Update-Insert Execution

This facility allows running an SQL Update that is followed by an Insert if the Update did not update any rows.

Define an `upsert` element in the adapter configuration file (or use the `UpsertQuery` class) for every update-insert to execute.

The synopsis is as follows:

```
<upsert connection="[connection]" stream="[stream]" table-name="[table]"
      [name="name"] [executor-name="executor"] [retry="count"] [retry-interval-
sec="sec"]>
  <keys>
   <column property="[property_name]" column="[column_name]" type="[sql_type]"/>
```

```
    [...column]
  </keys>
  <values>
   <column property="[property_name]" column="[column_name]" type="[sql_type]"/>
    [...column]
  </values>
</upsert>
```

The *connection* attribute value is required and provides the name of the configured JDBC connection.

A value for the *stream* attribute is required and provides the name of the stream that triggers the Update-Insert. The adapter expects a stream by this name to exist at adapter start time.

The *table* attribute value is required and provides the database table name.

The *name* attribute is optional and is only used for logging errors.

The *executor-name* attribute is optional. If specified, it must be the name of an executor configuration. If specified, the adapter will use the executor service (queue and thread pool) for performing all work. If not specified, the adapter performs the work in the same thread.

The *retry* attribute is optional. If specified, the adapter will retry a given number of times in case an error is encountered. If *retry-interval-sec* is specified, the adapter waits the given number of seconds between retries.

The *keys* element is required and provides the key columns of the table and the *values* element provides the list of value columns of the table.

The *column* element should occur as many as there are key and value columns in the table. The *property* attribute provides the name of the event property, the *column* attribute provides the database table column name and the *type* is any of the `java.sql.Types` names (case ignored).

A sample Update-Insert configuration follows:

```
<upsert          connection="db1"          stream="UpdateInsertDBTableTrigger"
 name="UpdateInsertSample"
    table-name="MyKeyedTable" executor-name="queue1" retry="3">
  <keys>
    <column property="eventProperty1" column="keyColumn1" type="varchar"/>
    <column property="eventProperty2" column="keyColumn2" type="varchar"/>
  </keys>
  <values>
    <column property="eventProperty3" column="valueColumn1" type="varchar"/>
    <column property="eventProperty4" column="valueColumn2" type="integer"/>
  </values>
</upsert>
```

8.6. Executor Configuration

Executors are named thread pools and queues that may be assigned to perform DML or update-insert work.

Define a `executor` element in the adapter configuration file (or use the `Executor` class) for every thread pool and queue to declare.

Upon adapter start, for each executor the adapter starts the given number of threads and an unbound queue.

The synopsis is as follows:

```
<executors>
  <executor name="[name]" threads="[count]"/>
</executors>
```

The *name* attribute value is required and provides the name of the executor, while the *count* attribute specifies the number of threads for the thread pool.

An example executor configuration::

```
<executors>
  <executor name="threadPool1" threads="2"/>
</executors>
```

An application can obtain a handle to all thread pools and queues via the Esper engine context:

```
ExecutorServices execs = (ExecutorServices)
    provider.getContext().lookup("EsperIODBAdapter/ExecutorServices");
```

8.7. Reading and Polling Database Tables

Herein we provide sample statements and documentation pointers to use Esper EPL for reading from database tables. If only reading and not writing to a database, no configuration or EsperIO jar is file required.

Please consult the Esper SQL access documentation for more information.

8.7.1. Polling and Startup SQL Queries

To execute an SQL query repeatedly, Esper provides the opportunity to join a pattern to an SQL statement. The pattern may provide a single interval or crontab schedule or may also contain multiple schedules or combinations thereof via the pattern `or` operator.

The sample query below simply executes every 10 seconds retrieving all rows from table `MyTable`:

```
select * from pattern[every timer:interval(10)], sql:db1 ['select * from
 MyTable']
```

To perform an incremental query, consider utilizing a variable to parameterize your SQL statement so that only new rows are returned.

The next EPL statements create a variable and pass the variable value to the query to poll for new rows only. It assumes the `timestamp` column in the `MyTable` table holds long-type millisecond values:

```
// Create a variable to hold the last poll timestamp
create variable long VarLastTimestamp = 0

// Poll every 15 seconds between 8am and 4pm based on variable value
insert into PollStream
select * from pattern[every timer:crontab(*, 8-16, *, *, *, */15)],
  sql:db1 ['select * from MyTable where timestamp > ${VarLastTimestamp}']

// Assign last value to variable
on PollStream set VarLastTimestamp = timestamp
```

A sample statement to read a table at startup time is below:

```
select * from pattern[timer:interval(0)], sql:db1 ['select * from MyTable']
```

Chapter 9. XML and JSON Output

Esper supports output event rendering to JSON and XML directly in its output API, please see the Esper documentation set for more information.

Chapter 10. Additional Event Representations

10.1. 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 names.
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 name 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-6-0.xsd` also in the `etc` folder of the EsperIO distribution.