THE POWER TO KNOW.

SAS® Event Stream Processing 6.1: Using SAS Event Stream Processing with Other Applications

Using SAS Event Stream Processing with SAS Cloud Analytic Services Actions

Overview

The SAS Cloud Analytic Services (CAS) server provides a cloud-based run-time environment for data management and analytics. CAS can run on a single node or on multiple nodes of a grid as a distributed server. The distributed server consists of one controller node and one or more worker nodes.

A *caslib* is an in-memory space to hold tables, access control lists, and data source information. All data is available to CAS through a caslib. Clients can add or upload data. The server can load data from server-side data sources.

All operations in CAS that use data are performed after you set up a caslib. Authorized users can add or manage caslibs with the CASLIB statement.

CAS *actions* are tasks that are performed at the request of the user. These actions are grouped into action sets. Actions perform common tasks such as data management, server management, and administration. For more information, see *SAS Viya Actions and Action Sets by Name and Product*.

You can establish a caslib as a connection point between a CAS server and an ESP server:

1 Start the ESP Server.

2 Open SAS Studio.

3 Declare a caslib. Specify the port and server name of the ESP Server. Specify the source type as **`esp`**.

   In the code that follows, the caslib reference name is espStatic. The caslib connects to an ESP Server named hdp04.orion.com through port 5555. The specified source type is **`esp`**.

```
/*Create an "ESP caslib" to read from ESP windows*/
caslib espStatic desc="vwap ESP window"
```

```
    datasource=(port=5555,server="hdp04.orion.com",srcType="esp")
    global;
```

This ESP caslib has global scope. This means that is it accessible from all sessions, subject to access controls.

This ESP caslib can now be used as a data source of event data for any single-pass CAS analytical action. When running that analytical action, you must specify the following information:

- The ESP caslib name
- The **`project_name/query_name/window`** path as the espUri path

When you use an ESP caslib, you do not need SAS Event Stream Processing licenses installed on the CAS worker nodes. These nodes do not run the ESP server, which does require a license. The workers are using the publish/subscribe interface to connect to the ESP server.

For more information about CAS, see *SAS Cloud Analytic Services: Fundamentals*. For more information about the programming language elements to use to start and manage CAS, see the *SAS Viya Programming: CAS Language Reference*.

CAS Action Sets for Use with SAS Event Stream Processing

loadStreams Action Set

The loadStreams action set enables you to load event data into in-memory tables that are populated with data from a SAS Event Stream Processing event. Using these actions requires a license for SAS Event Stream Processing and a running ESP server.

Specifically, the following actions are available:

Table 1 *loadStreams Actions*

Action Name	Description
appendSnapshot	Appends the current snapshot of a window to the table
loadSnapshot	Loads the current snapshot of a window to the table
loadStream	Streams data from a window into the table
nMetaData	Captures all model metadata for projects, queries, and windows

These actions connect to a specific window and consume event data for a specified number of seconds. They load event data into an in-memory table. After that, the action disconnects from the ESP server. You can run single-pass or multi-pass analytical actions against the data in the in-memory table.

You can run any single-pass analytical action against data streams published by a SAS Event Stream Processing model in real time with the loadStream action. The loadStream action stays connected to the ESP server and continues to gather data. The action periodically (based on a number of rows or a specified length of time) appends the data gathered to an output table. The output table in SAS Cloud Analytic Services must be global in scope. The global table continues to grow until this action is stopped.

```
proc cas;

session.sessionId;                                                  /* 1 */
```

```
loadStreams.loadStream /                                  /* 2 */
      casLib="espStatic"
      espUri="trades_proj/trades_cq/Trades"               /* 3 */
      casOut={caslib="mycas",name="streamTrades",promote=true};   /* 4 */

table.fetch /                                             /* 5 */
      table={caslib="mycas",name="streamTrades",
             vars={"security","tradeID","quantity","price"}};

run;
```

1 Display the session ID. You can use the ID to stop the subsequent loadStream action.

2 Run the loadStream action specifying the caslib for the ESP server (espStatic in this example).

3 The espUri requires a format: ***`project_name/query_name/window_name`***.

4 Specify a different caslib and table name for the output. The output table must be global in scope, so set `promote='true'`.

5 Use the fetch action to capture the output of the table snapshot.

The fetch action does not start until you explicitly stop the loadStream action. In SAS Studio, the program displays the running status, and you can simply click **Cancel** to stop it. You can also stop the process by issuing stopAction, specifying the session ID. The Trades example is not running live streaming content, so the fetch action should look much like this:

```
session.stopAction uuid='46415f51-904d-6a4f-9a0e-9ef1fa9e1360';
```

espCluster Action Set

The espCluster action set, which interfaces directly with SAS Event Stream Processing, enables a CAS server to start and view a cluster of ESP servers. SAS Event Stream Processing must be installed and configured on the worker nodes of a CAS cluster. The ESP Server on each worker node must be licensed.

This action set is essentially a sophisticated SAS Event Stream Processing client that uses the publish/subscribe interface for the analytical action event stream processing events.

Note: The espCluster action set is not supported on Microsoft Windows. In a UNIX environment, the action set is not supported on a single node system.

Table 2 *espCluster Actions*

Action Name	Description
listServers	Lists ESP server information
startServers	Starts a cluster of ESP servers After you issue this action, ESP servers continue running until the CAS session ends.

```
loadActionSet(s, "espCluster")                    # 1

cas.espCluster.startservers(                      # 2
  s, nodes=c("grid002.example.com", "grid003.example.com", "grid004.example.com"))

cas.espCluster.listservers(s)                     # 3
```

1. Load the espCluster action set.
2. Start the servers identified by fully qualified domain names. You can list the servers individually or start all of the servers in the cluster by specifying `nodes=c()`.
3. Get a list of the servers in the cluster.

```
$`ESP Servers`
Node                HTTP admin port     PUBSUB port
grid002.example.    23567               24081
grid003.example.    28386               44821
grid004.example.    27650               45067
```

You can use the Cluster Manager to push a project to the ESP server running on the worker nodes. The Cluster Manager connects to raw sources to get input data and appropriately stream them to a model running on the worker nodes.

When you run single-pass statistical algorithms, use the following settings for the caslib for the ESP server cluster (started by the startServers action):

- Set the port to undefined
- Set the authenticationType to none

Running SAS Event Stream Processing in a Hadoop YARN Container

Overview to YARN

Hadoop YARN serves as a resource management framework to schedule and handle computing resources for distributed applications. A Resource Manager manages the global assignment of compute resources for an application. A per-application Application Master manages an application’s scheduling and coordination. YARN provides processing capacity to applications by allocating containers to them. Containers encapsulate resource elements such as memory, CPU, and so on.

For more information about YARN, see the YARN documentation on the Apache Hadoop website.

Starting the Server in the Hadoop YARN Container

To run an ESP server in a Hadoop YARN container, run the following script to implement a YARN client.

$DFESP_HOME/bin/dfesp_yarn_joblauncher -e *localdfesphome* -a *httpport*
<-u *pubsubport*> <-q *yarnqueue* > <-p *yarnpriority* >
<-m *yarnmemory* > <-c *yarncores* > <-1><-o *consulhostport*>

Argument	Description
-e *localdfesphome*	Specify the **$DFESP_HOME** environment variable setting for SAS Event Stream Processing installations on Hadoop nodes.
-a *httpport*	Specify the value for the **-http** argument on the **dfesp_xml_server** command line. This port is opened when the ESP server is started.

Argument	Description
`-u` *`pubsubport`*	Specify the value for the **`-pubsub`** argument on the **`dfesp_xml_server`** command line. This port is opened when the ESP server is started.
`-q` *`yarnqueue`*	Specify the YARN Application Master queue. The default value is **`default`**.
`-p` *`yarnpriority`*	Specify the YARN Application Master priority. The default value is `0`.
`-m` *`yarnmemory`*	Specify the YARN Memory resource requirement for the Application Master and event stream processing server containers
`-c` *`yarncores`*	Specify the YARN Virtual Cores resource requirement for the Application Master and event stream processing server containers.
`-l`	Specify that the Application Master should use the **`/tmp/ESP/log4j.properties`** file found in HDFS.
`-o` *`consulthostport`*	Specify the *`host:port`* of a consult service.

The YARN client submits an Application Master and a script to run an ESP server. The client also passes associated resource requirements to the YARN Resource Manager.

The value of the environment variable **`DFESP_HADOOP_PATH`** must be the location of Hadoop configuration files on the system where you execute **`dfesp_yarn_joblauncher`**. This path must also point to the location of the following Hadoop JAR files, which are required by **`dfesp_yarn_joblauncher`**:

- **`hadoop-yarn-client-*.jar`**
- **`hadoop-yarn-api-*.jar`**
- **`hadoop-yarn-common-*.jar`**
- **`hadoop-common-*.jar`**
- **`hadoop-hdfs-*.jar`**
- **`hadoop-auth-*.jar`**
- supporting JAR files that are usually found in the **`hadoop/share/common/lib`** directory

The ApplicationMaster submitted by **`dfesp_yarn_joblauncher`** is copied to and executed from **`/tmp/ESP`** in HDFS, and is named “sas.esp.clients.yarn.*.*.jar”.

Note: The Application Master is built using Java 1.8, so the Java run-time environment on the Hadoop grid nodes must be compatible with that version. A SAS Event Stream Processing installation must already exist on every node in the grid.

No model XML file is provided. The HTTP client must subsequently manage the model run by the server.

The following command line is executed by the Application Master submitted by **`dfesp_yarn_joblauncher`**:

dfesp_xml_server -http *httpport* <-pubsub *pubsubport* >
-loglevel esp=info

By default, **`dfesp_yarn_joblauncher`** sets the following values for the Application Master:

- YARN Queue = default
- YARN Priority = 0
- YARN Resources: Memory = 32768 MB, Virtual Cores = 4

You can override these values through optional arguments to **`dfesp_yarn_joblauncher`**.

When launched, the Application Master requests one container in which to run the server shell script. That container request specifies the same YARN resources as were requested for the Application Master. Its defaults are 32768 MB and 4 virtual cores, unless you have passed different values to **`dfesp_yarn_joblauncher`**. YARN might kill any running process at any time when it exhausts its resources. Thus, you should tune these memory and core requirements to match the requirements of the running model.

The event stream processing container runs on a Hadoop node that might or might not be the node running the Application Master container.

You can invoke the **`dfesp_yarn_joblauncher`** again to launch additional event stream processing servers, which all run independently and have no knowledge of any other servers running on the grid.

Managing the Event Stream Processing Server

On successful start-up, the YARN Resource Manager web application should show an entry where the Name column shows “ESP”. Make a note of the associated application ID. Click the Tracking UI link to show the Application Master parameters. This URL connects to a web server running in the event stream processing Application Master itself.

Note the **`Master Host`** and **`Execution Host`** values. These are the nodes where the Application Master and event stream processing server containers are running, respectively. You can drill down to those specific nodes in the Node Manager. There, follow the links for your application ID to find process logs for the Application Master or event stream processing server.

If a failure occurs, the option to view logs depends on whether logging aggregation is enabled on your YARN installation. If it is enabled, log on to any Hadoop node, navigate to the local Hadoop installation, and run “**`/bin/yarn logs -applicationId=`*`ESP_application_id`***”. If it is disabled, log on to the individual Hadoop node and find the yarn logs in the local Hadoop installation directories.

To kill containers started by **`dfesp_yarn_joblauncher`**, log on to any Hadoop node, navigate to the local Hadoop installation, and run **`/bin/yarn application -kill`** ***`ESP_application_id`***.

Connecting to an ESP Server

The http-admin and publish/subscribe ports opened by the ESP server should be reachable by devices outside of the Hadoop grid, if network connectivity is available. The server host name is the name of the Hadoop node where the ESP server is running (shown in the **`Execution Host`** value displayed by the YARN Resource Manager). The port is the http-admin or publish/subscribe port passed to the **`dfesp_yarn_joblauncher`** script.

Clients connecting to this ESP server do not know that the server is running in a YARN container on a Hadoop node. Functional behavior is identical to a stand-alone server running outside a Hadoop grid.

Using SAS Event Stream Processing in the Apache Camel Framework

Overview

The Apache Camel framework enables you to integrate different applications into a single, cohesive architecture. Using the Apache Camel framework, you set up routes that contain endpoints:

```
<route id="injectTrades" startupOrder="10">
   <from uri="systemA://someThing"/>
   <to uri="systemB://someOtherThing"/>
</route>
```

The “from” and “to” elements in the previous code are Apache Camel endpoints that refer to components. Apache Camel supports many out-of-box components. It gives you the tools to develop custom components. An Apache Camel Consumer maps to a “from” endpoint, and an Apache Camel Producer maps to a “to” endpoint.

For more information about the Apache Camel framework, see the documentation.

Installing the Apache Camel Framework

In order to use the Apache Camel framework with SAS Event Stream Processing, you must download and install various files. These include Apache components and specific JAR files.

1 Access and install the following Apache components:

   - Apache Camel, which you can download from http://camel.apache.org/
   - Apache Maven, which you can download from https://maven.apache.org/

   Apache Maven is a build environment that you can use to create projects that leverage components from SAS Event Stream Processing. When you install Apache Maven, make sure that you install the **`bin`** directory subordinate to your Maven install directory in your path.

2 After you have installed the Apache components, install two JAR files into your local Maven repository:

   - ESP API Client JAR
   - ESP Camel JAR

   Note: You can find these JAR files in **`$DFESP_HOME/lib`**. They contain the client API and the Camel components.

3 Install the JAR files into the Maven repository:

```
$ cd $DFESP_HOME/lib
$ mvn install:install-file -Dfile=dfx-esp-api.jar -DgroupId=com.sas.esp
-DartifactId=dfx-esp-api -Dversion=6.1 -Dpackaging=jar
$ mvn install:install-file -Dfile=dfx-esp-camel.jar -DgroupId=com.sas.esp
-DartifactId=dfx-esp-camel -Dversion=6.1 -Dpackaging=jar
$ mvn install:install-file -Dfile=cas-client-3.06.jar -DgroupID=com.sas.esp
-DartifactID=dfx-cas-auth -Dversion=3.0.6 -Dpackaging=jar
```

4 After you have installed the JAR files, you can reference them from your Maven project object model (pom.xml) file. You must have an entry for both the event stream processing client API and the SAS Event Stream Processing Camel components.

Here is the event stream processing client API entry:

```
<dependency>
   <groupId>com.sas.esp</groupId>
   <artifactId>dfx-esp-api</artifactId>
   <version>[6.1]</version>
</dependency>
```

Here is the SAS Event Stream Processing Camel entry:

```
<dependency>
   <groupId>com.sas.esp</groupId>
   <artifactId>dfx-esp-camel</artifactId>
   <version>[4.2]</version>
</dependency>
```

Here is the CAS authentication entry:

```
<dependency>
   <groupID>com.sas.esp</groupID>
```

```
      <artifactID>dfs-cas-auth</artifactID>
      <version>3.0.6</version>
   </dependency>
```

Installing the RabbitMQ Library

Configure the Maven project so that SAS Event Stream Processing uses RabbitMQ as an alternative transport library:

1 Install the RabbitMQ API JAR:

```
$ cd $DFESP_HOME/lib
$ mvn install:install-file -Dfile=dfx-esp-rabbitmq-api.jar -DgroupId=com.sas.esp
-DartifactId=dfx-esp-rabbitmq-api -Dversion=3.2 -Dpackaging=jar
```

2 Update the Maven project object model (pom.xml) to include RabbitMQ dependency information:

```
<dependency>
          <groupId>com.rabbitmq</groupId>
          <artifactId>amqp-client</artifactId>
          <version>3.5.6</version>
        </dependency>
        <dependency>
            <groupId>commons-configuration</groupId>
             <artifactId>commons-configuration</artifactId>
             <version>1.10</version>
        </dependency>
        <dependency>
            <groupId>com.sas.esp</groupId>
            <artifactId>dfx-esp-rabbitmq-api</artifactId>
            <version>3.2</version>
        </dependency>
```

Note: You must define the dependency for the RabbitMQ JAR (`dfx-esp-rabbitmq-api`) in the pom.xml file before you define dependency for the ESP API Client JAR (`dfx-esp-api`).

SAS Event Stream Processing Implementation

The SAS Event Stream Processing implementation consists of SAS Event Stream processing Camel Endpoints that are either Consumers (which implement publish/subscribe subscribers) or Producers (which implement publish/subscribe publishers). A Consumer maps to a **`from`** endpoint, and a Producer maps to a **`to`** endpoint. For example, to receive events from one publish/subscribe server and send them to another, execute the following code:

```
...
<endpoint id="subscribe" uri="esp://espsrv01:46003">
  <property key="project" value="project" />
  <property key="contquery" value="query" />
  <property key="window" value="transform" />
</endpoint>

<endpoint id="publish" uri="esp://espsrv01:47003">
  <property key="project" value="project" />
  <property key="contquery" value="query" />
  <property key="window" value="trades" />
</endpoint>
```

```
<route>
  <from uri="ref:subscribe"/>
  <to uri="ref:publish" />
</route>
...
```

The SAS Event Stream Processing components can work with the following formats representing events:

Format	Description
Map<String,Object>	A standard JAVA Map object in which the keys are field names and the values are field values.
List<Map<String,Object>>	A standard JAVA List of JAVA Map objects in which the keys are field names and the values are field values.
XML	Event data in XML.
JSON	Event data in JSON.

A SAS Event Stream Processing Consumer (subscriber) receives events and converts them into one of these formats to send them along the route. A SAS Event Stream Processing Producer (publisher) receives data in one of these formats, converts it into events, and publishes them.

Through these standard formats, SAS Event Stream Processing easily integrates with any of the other components available in the Camel framework and shares data with them. If some transformation is required in order to get the data into the required format, you can use a transformation bean between the endpoints.

Here is an example of transforming the standard comma-separated value event format into one of the supported types:

```
...
<endpoint id="csvData" uri="stream:file">
   <property key="fileName" value="/mnt/data/share/tradesData/trades1M.csv" />
</endpoint>

<endpoint id="inject" uri="esp://espsrv01:46003">
   <property key="project" value="project" />
   <property key="contquery" value="query" />
   <property key="window" value="trades" />
</endpoint>

<route id="injectTrades" startupOrder="10">
   <from uri="ref:csvData"/>
   <bean ref="csvTransform" method="transform" />
   <to uri="ref:inject"/>
</route>

<bean id="csvTransform" class="com.sas.esp.camel.transforms.CsvTransform">
   <property name="schema" value="id*:int64,symbol:string,currency:
int32,time:int64,msecs:int32,price:double,quant:int32,venue:int32,broker:
int32,buyer:int32,seller:int32,buysellflg:int32" />
   <property name="format" value="xml" />
</bean>

...
```

First create an endpoint called csvData to read the CSV data from a file. You also create an endpoint called inject to inject this data into an ESP Source window. The route that you use goes from the file into SAS Event Stream Processing, but you must get the CSV data into one of the supported formats.

To do this, create a bean called csvTransform, which requires a schema and an output format. After that, you can form events from the CSV data and create an XML document to pass along the route. This is consumed by the Producer, which injects the data into the specified Source window. The method attribute for any of the SAS Event Stream Processing transformation beans is always a transform.

Using Camel Components in a Maven Project

In order to reference the SAS Event Stream Processing Camel components by URI, you must create the following file:

```
META-INF/services/org/apache/camel/component/esp
```

The file must contain the following line:

```
class=com.sas.esp.clients.camel.EspComponent
```

Note: This is described in further detail at http://camel.apache.org/writing-components.html.

This enables you to reference the event stream processing components with a URI such as the following:

```
<endpoint id="inject" uri="esp://<pub/sub host>:<pub/sub port>">
```

Configuring Endpoints

You must determine the host and port of the SAS Event Stream Processing publish/subscribe server with which you are going to communicate. Use this information to specify the URI of the component. For example, if your publish/subscribe server is running on port 46003 on machine espsrv01, your URI is:

```
esp://espsrv01:46003
```

Because you are always putting events into or getting events out of windows, you also must specify the project, continuous query, and window in which you are interested. You can do this using either of the following methods:

add parameters to the URI

```
esp://espsrv01:46003?project=myproject&contquery=mycq&window=trades
```

specify an endpoint element with properties :

```
<endpoint id="inject" uri="esp://espsrv01:46003">
   <property key="project" value="project" />
   <property key="contquery" value="query" />
   <property key="window" value="trades" />
</endpoint>
```

Table 3 Properties That You Can Set on an Endpoint

Property	Description	Consumer	Producer	Valid Values
`project`	The event stream processing project.	x	x	A valid project.
`contquery`	The event stream processing continuous query.	x	x	A valid continuous query.
`window`	The event stream processing window.	x	x	A valid window.

Property	Description	Consumer	Producer	Valid Values
`format`	The data format for this component. This is usually used with a Consumer to specify the format of the event data to send down the route. It can be used with a Producer. If the component receives a message with a body that is a String, the producer uses the format (either XML or JSON) for conversion.	x	x	map, list, XML, JSON
`blocksize`	The size of the event blocks to inject into a Source window.		x	Unsigned integer
blob	The name of an event field that is used to contain the entire message body. If this property is set, the Producer receives a message and creates a MapMap<String,Object> where the key is the field indicated by the blob property. The value is the entire message body represented as a String. This data is injected into the appropriate Source window. **Note:** When using the blob property, the Source window into which the event is being injected must set both 'insert-only=true' and 'autogen-key=true'.		x	A valid event field
`authUser`	The name to use with SASLogon authorization.	x	x	
`oAuthToken`	The OAuth token to use with OAuth authentication.	x	x	

Using Transformation Beans

When data is not in a format immediately usable by the event stream processing components, you must use a transformation bean to convert the data into a usable format. You can use a transformation bean to convert events in the SAS Event Stream Processing CSV format into a usable format. Place the transformation beans in a route between the from and to endpoints.

Here is an example:

```
...
       <route id="injectTrades">
      <from uri="ref:csvData"/>
      <bean ref="csvTransform" method="transform" />
      <to uri="ref:inject"/>
   </route>

...

<bean id="csvTransform" class="com.sas.esp.clients.camel.transforms.CsvTransform">
   <property name="schema" value="id*:int64,symbol:string,currency:
int32,time:int64,msecs:int32,price:double,quant:int32,venue:int32,broker:
int32,buyer:int32,seller:int32,buysellflg:int32" />
```

```
        <property name="format" value="xml" />
        <property name="dateformat" value="ddMMMyyyy HH:mm:ss.S a" />
    </bean>
    ...
```

Each bean takes certain parameters to help it convert the data to be usable by SAS Event Stream Processing. The CSV transformation bean requires the event schema and the output data format. Note that the bean lies between a “from” endpoint that contains a file reference and a “to” endpoint to publish events into a window.

Table 4 Transformation Beans in the SAS Event Stream Processing Camel Package

Class	Description
com.sas.esp.clients.camel.transforms.CsvTransform	Transforms CSV data into ESP event data schema The event schema for the events represented by the CSV data. format The data format to use to write events. Valid values are map, list, XML, and JSON. dateformat The date format to use.
com.sas.esp.clients.camel.transforms.RssTransform	Transforms RSS data into ESP event data format The data format to use to write events. Valid values are list, XML, and JSON.

Examples

Where to Find Examples

A set of examples is available at **$DFESP_HOME/examples/java/camel**.

CSV Injection

The following example reads trade data from a CSV file and injects the trades into the broker surveillance model. It also subscribes to the brokerAlertsAggr window and writes these events to the console in JSON format.

1 Edit **src/main/resources/esp.properties** so that it contains your publish/subscribe server information:

```
espServer=esp://espsrv01:46003
tradesFile=data/trades1M.csv
```

2 Start your ESP server:

```
$ dfesp_xml_server -model file://model.xml -http-admin <http admin port>
-http-pubsub <http pub/sub port> -pubsub <esp pub/sub port> -nocleanup
```

3 Start the project:

```
$ mvn camel:run
```

Distributed Modeling

This example distributes broker surveillance model between two servers. The first server takes the trade data and performs all the dimensional additions (broker info, venue data) to the event. This model is in primary.xml and ends with a Functional window called transform. The events generated by transform contain a full set of trade information. The project subscribes to this window in server1 and forwards the events to server2, which looks for the broker alerts. Another route is used to subscribe to brokerAlertsAggr window in server2 and dump the events to the screen in Map format.

1 Edit **`src/main/resources/esp.properties`** so that it contains your publish/subscribe server information:

```
espServer1=esp://espsrv01:46003
espServer2=esp://espsrv01:47003
tradesFile=data/trades1M.csv
```

2 Start your primary event stream processing server:

```
$ dfesp_xml_server -model file://primary.xml -http-admin <http admin port>
-http-pubsub <http pub/sub port> -pubsub <esp pub/sub port> -nocleanup
```

3 Start your secondary event stream processing server:

```
$ dfesp_xml_server -model file://secondary.xml -http-admin <http admin port>
 -http-pubsub <http pub/sub port> -pubsub <esp pub/sub port> -nocleanup
```

4 Start the project:

```
$ mvn camel:run
```

RSS

This example uses the Camel RSS Component to set up a route that reads data from any number of RSS feeds and injects them into SAS Event Stream Processing. You should be able to add any RSS feeds to the route.

```
<route>
   <from uri="rss:http://feeds.reuters.com/reuters/businessNews" />
   <from uri="rss:http://feeds.reuters.com/reuters/topNews" />
   <from uri="rss:http://feeds.reuters.com/reuters/technologyNews" />
   <bean ref="rssTransform" method="transform" />
   <to uri="ref:publishNews" />
</route>

...
```

You can also use a new transformation bean to transform the RSS data into a supported format:

```
<bean id="rssTransform" class="com.sas.esp.clients.camel.transforms.RssTransform">
   <property name="opcode" value="upsert" />
</bean>

...
```

In the following project, the RSS data is keyed by title:

```
<project name='project' pubsub='auto' threads='4'>
   <contqueries>
      <contquery name='cq' trace='src'>
         <windows>
            <window-source name='src'>
               <schema-string>title*:string,author:string,link:string,
description:string,categories:string,pubDate:date</schema-string>
            </window-source>
         </windows>
      </contquery>
   </contqueries>
</project>

...
```

1 Edit **`src/main/resources/esp.properties`** so that it contains your publish/subscribe server information:

```
espServer=esp://espsrv01:46003
```

2 Start your event stream processing server:

```
$ dfesp_xml_server -model file://model.xml -http-admin <http admin port>
-http-pubsub <http pub/sub port> -pubsub <esp pub/sub port> -nocleanup
```

3 Start the project:

```
$ mvn camel:run
```

Weather

This example uses the Camel Weather Component to set up a route that reads weather data for any number of locations and injects it into SAS Event Stream Processing. You should be able to add any locations to the route. The locations can be defined as endpoints as shown in the following example:

```
<endpoint id="cary" uri="weather:foo">
   <property key="location" value="cary,nc"/>
   <property key="mode" value="XML"/>
   <property key="units" value="IMPERIAL"/>
</endpoint>

<endpoint id="morehead" uri="weather:foo">
   <property key="location" value="moreheadcity,nc"/>
   <property key="mode" value="XML"/>
   <property key="units" value="IMPERIAL"/>
</endpoint>

<endpoint id="chapelHill" uri="weather:foo">
   <property key="location" value="chapelhill,nc"/>
   <property key="mode" value="XML"/>
   <property key="units" value="IMPERIAL"/>
</endpoint>

...
```

You can then add these endpoints to your route:

```
<route>
   <from uri="ref:cary"/>
   <from uri="ref:morehead"/>
   <from uri="ref:chapelHill"/>
   <to uri="ref:publishWeather"/>
</route>

...
```

1 Edit **`src/main/resources/esp.properties`** so that it contains your publish/subscribe server information:

```
espServer=esp://espsrv01:46003
```

2 Start your event stream processing server:

```
$ dfesp_xml_server -model file://model.xml -http-admin <http admin port>
-http-pubsub <http pub/sub port> -pubsub <esp pub/sub port> -nocleanup
```

3 Start the project:

```
$ mvn camel:run
```

Using SAS Event Stream Processing with Apache NiFi

Setting Up Apache NiFi

Apache NiFi is a system to automate the flow of data between systems. You can download Apache NiFi from https://nifi.apache.org/. For more information about Apache NiFi, see the FAQ.

1 Download Apache NiFi version 1.*x* into a working directory, **`$WORK`**, on a UNIX system, **`$HOST`**.

2 Enter the following commands at the command prompt:

   a **`cd $WORK`**

   b **`gunzip nifi-1.x-bin.tar.gz`**

   c **`tar xvf nifi-1.x-bin.tar`**

   d **`export NIFI_HOME=$WORK/nifi-1.x`**

   e **`export PATH=$NIFI_HOME/bin:$PATH`**

3 A NiFi Archive (NAR) file enables components and their dependencies to be packaged together. Copy the SAS Event Stream processing NAR file (nifi-esp-nar-*version*.nar) from **`$DFESP_HOME/lib`** to **`$NIFI_HOME/lib`**.

4 Edit **`$NIFI_HOME/conf/nifi.properties`** to change the following line to use an available port:

```
nifi.web.http.port=8080
```

For example, if port 31005 is available, the line reads as follows:

```
nifi.web.http.port=31005
```

5 Start Apache NiFi as follows: **`$NIFI_HOME/bin/nifi.sh run`**.

Note: The DFESP_HOME environment variable must be set before starting Apache NiFi with SAS Event Stream Processing. Update the **`$NIFI_HOME/bin/nifi-env.sh`** script to set DFESP_HOME if you are starting Apache NiFi at system start-up.

6 Start Apache NiFi in a browser. Specify the available port that you selected in step 4: **`http://$HOST:`*`available_port`*`/nifi`**

You should now be able to design your Apache NiFi flows. The SAS Event Stream Processing processors ListenESP and PutESP should be available. When processor updates become available, you need to update only the NAR file in **`$NIFI_HOME/lib`**.

PutESP Processor

The PutESP processor enables you to publish events from Apache NiFi into an ESP engine. It requires that you specify an ESP project, continuous query, and Source window hierarchy for the processor.

Given this information, the PutESP processor can accept FlowFiles in Apache Avro format, which is binary. For each object in the Avro input, PutESP takes any data that maps to a field in the ESP source window schema and creates an ESP event. When all the events have been created, they are sent into ESP.

Table 5 Properties

Pub/Sub Host	The host name of the ESP engine publish/subscribe server.
Pub/Sub Port	The port of the ESP engine publish/subscribe server.
Project	The project into which events are published.
Continuous Query	The continuous query into which events are published.
Source Window	The Source window into which events are published.
Record Schema	The Avro schema for input records. If the schema is not set, it generates an Avro schema from the publish target Source window in the ESP model. You can get the Avro schema from a running event stream processing model through the REST interface.
Block Size	If this is set, PutESP limits the events sent at one time into ESP engine to this number.
Input Date Format	If this is set, PutESP expects dates in the incoming data to be strings in this format.
Convert Inserts to Upserts	This should be set if you want to convert any opcodes coming in as inserts to upserts before publishing. Allowable Values: ■ **Yes** Convert inserts to upserts. ■ **No** Do not convert inserts to upserts. Note: Defaults to **No**
Authentication User	The SASLogon authentication user.
OAuth Token	The token to use for OAuth authentication.

Table 6 Relationships

success	If the events are successfully sent to the ESP engine, PutESP sends the originating FlowFile to this relationship.

failure	When PutESP detects a failure in its interaction with the ESP engine, it sends a FlowFile to the failure relationship.

Note: No **Reads Attributes** or **Writes Attributes** specified.

For more information about Apache Avro, see the documentation.

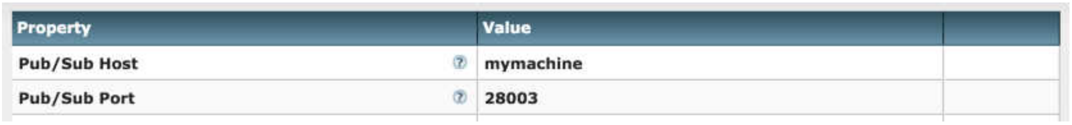
ListenESP Processor

The ListenESP processor enables you to subscribe to any number of windows in a SAS ESP engine and receive events from those windows. The events are collected and converted to AVRO format and sent along the success relationship. The project, continuous query, and window attributes are set on the FlowFile so that downstream processors can act on this information. For example, one can use the **RouteOnAttribute** processor to send events down different paths depending on the originating project, continuous query, and window. You can also specify field values from the ESP events to put on the FlowFile as attributes.

The ListenESP processor requires information for the subscribed ESP windows in order to collect events of interest. You must supply valid ESP engine connection information in the **Pub/Sub Host** and **Pub/Sub Port** properties. For example, if you are running an ESP server on a host called *mymachine*,

```
$ $DFESP_HOME/bin/dfesp_xml_server -model file://model.xml -pubsub 28003
```

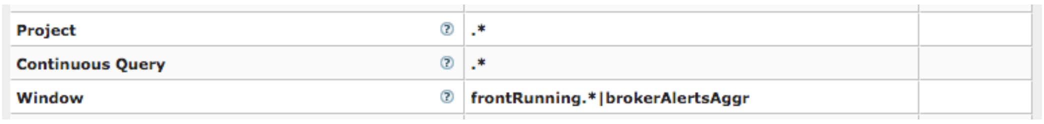
your connection properties are as follows:



Property	Value	
Pub/Sub Host	mymachine	
Pub/Sub Port	28003	

Once you have entered the connection information, you need to specify regular expressions to determine the subscribed windows from which to get the event data. To get all of an ESP element from the engine, whether it is projects, continuous queries, or windows, you simply use the `.*` regular expression. Because many engines contain a single project and continuous query, this is the simplest way to specify that you want them all. Usually, you do not want to subscribe to all the windows in a model.

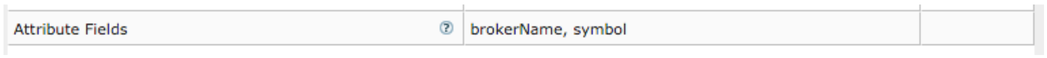
The following example subscribes to all windows whose name begins with `frontRunning` or whose name is `brokerAlertsAggr` in all continuous queries in all projects.



Project	.*	
Continuous Query	.*	
Window	frontRunning.*|brokerAlertsAggr	

If you want to grab field values from each received event, you use the **Attribute Fields** property. This is a comma-separated list of ESP field names. For each event that contains a value for each specified field in this property, values are put onto the FlowFile as an attribute.

The following example tells the processor to put the values of the `brokerName` and `symbol` fields onto the FlowFile.



Attribute Fields	brokerName, symbol	

Table 7 Properties

Pub/Sub Host	The host name of the ESP engine publish/subscribe server.
Pub/Sub Port	The port of the ESP engine publish/subscribe server.
Project	This is a regular expression used to specify the projects in the engine to which to subscribe.

Continuous Query	A regular expression used to specify the continuous queries in the engine to which to subscribe.
Window	A regular expression used to specify the windows in the engine to which to subscribe.
Snapshot	Set to **Yes** to get a snapshot of the events in the window. Allowable Values: ■ **Yes** Grab initial contents of the window. ■ **No** Do not grab initial contents of the window. Note: Defaults to **No**
Output Date Format	If this is set, ListenESP writes date and timestamp fields as strings using this format instead of as numeric values.
Attribute Fields	These are event field values to put onto the FlowFile as attributes.
Authentication User	The SASLogon authentication user.
OAuth Token	The token to use for OAuth authentication.

Table 8 Relationships

success	FlowFiles containing ESP events in AVRO format are passed to this relationship upon arrival.
failure	When ListenESP detects a failure in its interaction with the ESP engine, it sends a FlowFile to the failure relationship.

Table 9 Writes Attributes

project	The project containing the current set of events.
contquery	The continuous query containing the current set of events.
window	The window containing the current set of events.
path	This is set to the model container of the event: project or continuous query.
filename	This is set to the window of the event with a timestamp appended.

Note: No Reads Attributes specified

Running SAS Event Stream Processing in a Cisco Kinetic Edge and Fog Processing Module

The Cisco Kinetic platform is a data fabric that is designed to extract data from connected devices, compute anywhere within a distributed network, and move data to various applications. The Cisco Edge and Fog Processing Module (EFM) applies rules on data in motion to reduce, compress, normalize, and transmit data in optimal ways.

You can run one or more SAS Event Stream Processing engines as a microservice in an EFM. The EFM functions as an engine link and uses the Distributed Services Architecture for IoT (DSA) standard to connect to other links that operate on the EFM broker.

An ESP server running in an EFM link runs stand-alone, that is, it has no awareness of any other running ESP servers. It also has no awareness of running in a Kinetic environment. You can use the EFM Dataflow Editor to graphically connect fields within the windows of a running model to other links on the EFM broker.

Note: The Java run-time environment where the EFM link runs must be compatible with Java 1.8.

Follow these steps to install a link that can run one or more engines:

1. Correctly define DFESP_HOME and LD_LIBRARY_PATH (or PATH for Windows) when the broker is started. All ESP server instances that run in an EFM link use the product installation referenced by the DFESP_HOME environment variable. In effect, the user environment is that of the user who started the EFM broker.

   If it is problematic to stop and restart the EFM broker or to modify the user environment, do the following. After the link is installed but before it is started, add DFESP_HOME and LD_LIBRARY_PATH definitions to the link start-up script.

2. Install SAS Event Stream Processing on a machine running EFM. Ensure that the product installation is functional and that the DFESP_HOME environment variable is set correctly.

3. Run the EFM System Administrator, click on your broker in the left-hand "Brokers" panel, and click on "Management" in the main panel. Then run the “Install Link from Zip” action and select file **`$DFESP_HOME/lib/kinetic-dslink-esp-*-zip.zip`**.

   This ZIP file contains the SAS Event Stream Processing Java client JAR file and additional Java glue code to implement connectivity between the SAS Event Stream Processing link and other links through DSA.

4. If DFESP_HOME and LD_LIBRARY_PATH were not correctly set when the broker was started, define them in the link start-up scripts now. The scripts are located here: **`/opt/cisco/kinetic/efm_server/dslinks/`*`yourlink`*`/bin/kinetic-dslink-esp.*.`**

   Add the appropriate definitions to the beginning of the Linux or Windows script.

5. Run the “Rescan” action to show the SAS Event Stream Processing link.

6. Expand the “Links” drop-down menu. Select the SAS Event Stream Processing link and run the “Start Link” action. You then see the new link named “ESP-*” in the lower left “Links” section.

7. Select the SAS Event Stream Processing link and run its “Create Model” action. Paste the XML code of a model into the “Model XML” box.

8. Run the “Add Server” action, and provide the server publish/subscribe port. Wait five seconds for this action to complete. The action causes the link to run the following command line: “**`dfesp_xml_server -model file://`*`xml_model`*` -pubsub `*`pubsub_port`***”. The link also internally runs the model “Introspect” action, which creates a Java publish/subscribe subscriber for every window within the model. It also creates a Java

publish/subscribe publisher for every source window within the model. These clients are invoked whenever another EFM link bound to a field in a window triggers a publish or subscribe action.

9 If the link is restarted, then it uses the currently configured model and repeats the “Introspect” action automatically. Whenever you restart the ESP server, only the server that uses the command line shown in the previous step is started, and you must invoke “Introspect” separately.

To debug a failed ESP server, run the server “SendConsoleToLog” action and examine the log details under “Life cycle”.

Note: When you stop a SAS Event Stream Processing link that contains running ESP servers, server processes are not killed. Use the **`top`** command on Linux to manually kill server processes before you restart the link. Otherwise, the newly started server fails because the publish/subscribe port is not available.

After installing the link, you can run multiple ESP servers within it. Each link can run any model that you have uploaded to it.

To connect a window in a running link to another EFM link, follow these steps:

1 Run the EFM Dataflow Editor and drill down to a window in the “downstream” section of the Data pane.

2 If the window is a source window, then right-click on the window name under “Source windows” and drag the “Publish” action to the Dataflow pane. This creates a data service that exposes all the fields in the source window schema. Fields can then be bound to another block’s output, and the publish `autoRun` property can be enabled to invoke the publish action whenever an input changes. You can also invoke the publish action manually for testing.

3 To subscribe to any window, click on “Output windows” in the Data pane, find the window name in the Metrics pane, and drag it to the data flow pane. Its value property always contains a JSON representation of the most recent event output by the window, including key/value pairs for opcode and flags. You can bind the value property to another block’s input, or bind it to a jsonParser block first.

To see available SAS Event Stream Processing actions in the EFM System Administrator, click on any link or model or project or query or window and then click “Actions and Attributes”. To see available SAS Event Stream Processing actions in the EFM Dataflow Editor, right-click on any engine link or model or project or query or window.

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies. Copyright © 2019, SAS Institute Inc., Cary, NC, USA. All Rights Reserved. May 2019 6.1-P1:espcases

THE POWER TO KNOW.