



SAS[®] Event Stream Processing: WebSocket API Reference

2025.12

This document might apply to additional versions of the software. Open this document in [SAS Help Center](#) and click on the version in the banner to see all available versions.

Overview	2
Authentication Handshaking for the JavaScript WebSocket API	2
Creating and Using the JavaScript WebSocket Subscriber	3
Additional Parameters for the WebSocket Subscriber	3
Message Parameters	6
Creating and Using the JavaScript WebSocket Publisher	7
Publisher Parameters	8
Obtaining Project Statistics through the JavaScript WebSocket API	9
Setting Up a Connection to Get Log Messages	10
Using the ESP Server Connection API	10
Overview	10
Connecting to the ESP Server	10
Loading Projects	11
Loading Routers	12
Event Subscribers	13
Event Publishers	21
Retrieving ESP Server Statistics	25
Retrieving Server Logs	29
Retrieving Model Information	29
Sending and Receiving Binary Data	30

Overview

The WebSocket Protocol enables two-way communication between a client that is running untrusted code in a controlled environment and a remote host that accepts the communication that results from that code. For more information about the WebSocket protocol, see [RFC-6455](#).

SAS Event Stream Processing supports two APIs that use the WebSocket protocol to communicate with the ESP server.

- The JavaScript WebSocket API can perform tasks such as subscribing, publishing, and monitoring project statistics. You can use any client language or platform with WebSocket capabilities (such as Java, Javascript, C++, Python, and IOS) to communicate with the ESP server through this API. Because it uses a persistent socket connection, this WebSocket API is faster and more efficient than the RESTful API.
- The [ESP Server Connection API](#) uses the WebSocket protocol to perform ESP functions with minimal system resources. It uses a single WebSocket connection to do the following:
 - subscribe to any number of ESP windows
 - publish events
 - receive server status information
 - monitor the server log

Within the ESP server, the WebSocket publish/subscribe objects use the standard publish/subscribe interfaces.

Authentication Handshaking for the JavaScript WebSocket API

The JavaScript WebSocket API does not support HTTP authentication through the initial request headers. In this case, you can use a handshaking strategy instead.

Every time you make a connection to a WebSocket, one or more response headers are the first data written back to the client. These headers are in HTTP response header format:

```
name: value
```

A blank line indicates the end of the response header information.

The status header indicates the status of the WebSocket. The possible values are as follows:

Status Header Value	Description
200	The WebSocket is ready to be used.
401	The WebSocket connection requires authentication data.

When you get a status of 401, examine the `www-authenticate` header to determine the type of authentication information that you must send over the WebSocket. The possible values are as follows:

Authentication Type	Description	Data Format
<code>oauth</code>	You must send a valid OAUTH token over the WebSocket.	<code>Bearer OAUTH_TOKEN</code>
<code>basic</code>	You must send a user name and password over the WebSocket.	<code>Basic user:password</code> The <code>user:password</code> portion must be BASE64-encoded.

Creating and Using the JavaScript WebSocket Subscriber

The JavaScript WebSocket subscriber performs two tasks:

- Manages a collection of currently viewed events. This collection is called an *event page*.
- Subscribes to and handles event notifications coming from the event stream processing engine.

When you create a subscriber, supply the full pathname (project, continuous query, and window) of the window that uses the WebSocket protocol:

ws://server:port/eventStreamProcessing/v1/subscribers/project/continuous_query/window

You can specify additional parameters that enable you to change the behavior of the subscriber. You can apply these parameters after object creation through `load` and `properties` messages.

Table 1 Additional Parameters for the WebSocket Subscriber

Parameter	Description
<code>mode</code>	Specifies the mode of the subscriber. Valid values are <code>updating</code> and <code>streaming</code> .

Parameter	Description
	An updating subscriber uses event pages. An event page is a set of events of an optional size that comprises the current view of the subscriber. For example, for an HTML graphical application displaying a bar chart of events, the currently displayed set of events is the current event page. The subscriber maintains the keys of these events. When the subscriber gets an event notification from the event stream processing engine and the event is not in the current page, it is ignored. When it is in the current page, the WebSocket client is notified of the update. A streaming subscriber sends each event for which it receives a notification from the ESP engine. These events are put into a list as they are received and then sent to the WebSocket client. The list is trimmed to the current page size of the subscriber before it is sent. When a large number of events are streaming through the engine at a high rate, some events are dropped. The WebSocket client gets a sample of the total set of events streaming through the system.
pagesize	Specifies the number of events to keep in an event page for this subscriber.
separator	Specifies the separator to use between events when you use the properties format.
filter	Specifies a functional filter to apply to the events being processed by the subscriber. For example, suppose you want to subscribe to stock transactions for IBM and AT&T that flow through the <code>largeTrades</code> window: <pre>ws://espsrv01:26000/eventStreamProcessing/v1/subscribers/p/cq/ largeTrades?filter=in(symbol, 'IBM', 'T')</pre>
sort	Specifies how to sort events that are streamed through a window. The format is <i>event_field:sort_direction</i>. The <i>sort_direction</i> value is optional and defaults to descending. For example, suppose you want to subscribe to stock transactions for AT&T that flow through the <code>largeTrades</code> window and sort them by quantity: <pre>ws://espsrv01:26000/eventStreamProcessing/v1/subscribers/p/cq/ largeTrades?filter=eq(symbol, 'T')&sort=quantity</pre> Note: When this parameter is set, each time that data is sent to the WebSocket client, it is sorted. When a window processes a large number of events, performance can degrade. Note: The sort property is meaningful only in updating subscribers. Streaming subscribers always send events in the order received.
format	Specifies the format in which to receive event data. Valid values are xml, json, csv, or properties. The properties format contains each event as a set of <i>name=value</i> pairs separated by new lines. Events are separated by a user-defined separator (a blank line by default). You can set the separator with the <code>separator</code> request parameter.

Parameter	Description
	Note: Page information is included only in data that is returned from <code>load</code> commands. Page information does not return with events received during subscription notifications.
<code>interval</code>	Specifies an interval, in milliseconds, at which to deliver collected events to the WebSocket client. Use this interval when you have millions of events streaming through an event stream processing model at a high rate. For example, suppose you create a streaming subscriber to a Source window and then stream 10 million events into it. The system attempts to send all these events to the WebSocket client. Even when the ESP server can handle that load, it often overloads a client. When you set the <code>interval</code> property, the events are collected for that amount of time and only sent one time per specified interval.
<code>schema</code>	Set to true when you want the WebSocket client to return the window schema when a connection is created.
<code>counts</code>	Set to true when you want the WebSocket client to return the total number of events in the subscription window with each event notification. The default value is false .
<code>info</code>	Use this parameter to instruct the subscriber to periodically deliver event information to the client. The attributes are as follows: ■ <code>page</code> - the current page number ■ <code>pages</code> - the total number of pages ■ <code>events</code> - the number of events in the subscription ■ <code>total</code> - the number of retained events in the window (this matches the events value when no filter exists) Suppose you sent the following request: <pre>dfesp_xml_client -url "ws://espsrv01:26000/eventStreamProcessing/v1/subscribers/p/cq/largeTrades?format=xml&info=5"</pre> Something like the following output appears on the console every 5 seconds: <pre><info page='1' pages='1024' events='51158' total='51158'/></pre>
<code>snapshot</code>	Specifies whether to process the current contents of the window. Use only with stateful windows. When false (default), the function processes only future events. When true , the function immediately sends the current contents of the window.

After a subscriber connection is established, you can send messages through it. Send a `load` message to load events from the window.

```
<load page='page_number' pagesize='page_size' sort='field<:direction>'
  interval='delivery_ms' mode='streaming|updating'>
  <filter>...</filter>
</load>
```

Send a `properties` message to set properties on the connection without returning any events.

```
<properties page='page_number' pagesize='page_size' sort='field<:direction>'
  interval='delivery_ms' mode='streaming|updating'>
  <filter>...</filter>
</properties>
```

Here, the `pagesize`, `sort`, and `filter` properties behave exactly as they do when you specify them in the URI to create a subscriber.

Table 2 *Message Parameters*

Parameter	Description
page	Specifies a specific page number or a page number relative to the current page number. You can also specify one of the following values: first - first page of events from the window last - last page of events from the window next - next page of events (current page + 1) from the window prev - previous page of events (current page - 1) from the window
pagesize	Specifies the number of events to keep in an event page for this subscriber.
sort	Specifies how to sort events that are streamed through a window. The format is: <i>event_field:sort_direction</i>. The <i>sort_direction</i> value is optional and defaults to descending. Note: When this parameter is set, each time that data is sent to the WebSocket client, it is sorted. When a window processes a large number of events, performance can degrade. Note: The sort property is meaningful only in updating subscribers. Streaming subscribers always send events in the order received.
interval	Specifies an interval, in milliseconds, at which to deliver collected events to the WebSocket client. Use this interval when you have millions of events streaming through an event stream processing model at a high rate.
mode	Specifies the mode of the subscriber. Valid values are updating and streaming .
filter	Specifies a functional filter to apply to the events being processed by the subscriber.

Suppose that you create the following subscriber:

```
ws://espsrv01:26000/eventStreamProcessing/v1/subscribers/p/cq/largeTrades?
filter=in(symbol,'IBM','T')&pagesize=5
```

Then, after establishing a connection to it, suppose that you send the `<load page='first'/>` message. The subscriber responds:

```
-- page=1;pages=41;events=201;total=51158
I,N, 8451722,IBM,87236,1280951996,3436,131.310000,1900,55222,1012445,0,8123541,1
I,N, 15795961,T,87236,1280951996,5317,26.650000,1400,55111,1012334,0,8093853,1
I,N, 15795959,T,87236,1280951996,5317,26.650000,1300,55999,1012112,0,8374755,1
I,N, 8451702,IBM,87236,1280951995,652,131.340000,1000,55222,101556,0,8093853,1
I,N, 8451701,IBM,87236,1280951995,593,131.310000,1200,55111,1012445,0,8382948,0
```

Suppose that you send the `<load page='next' />` message. The subscriber responds:

```
-- page=2;pages=41;events=201;total=51158
I,N, 15723923,T,87236,1280928628,94,26.660000,1400,55333,1012445,0,8374755,0
I,N, 15795484,T,87236,1280951967,3215,26.660000,7300,55444,1012223,0,8123541,0
I,N, 15723913,T,87236,1280928616,1524,26.680000,1000,55333,1012112,0,8483102,1
I,N, 8451644,IBM,87236,1280951989,2221,131.350000,1600,55444,1012334,0,8382948,1
I,N, 15794640,T,87236,1280951884,5613,26.650000,1200,55000,101556,0,8093853,1
```

Suppose that you send the `<load page='first' sort='price' />` message. The subscriber responds:

```
-- page=1;pages=41;events=201;total=51158
I,N, 8450613,IBM,87236,1280951850,8556,131.450000,1100,55333,1012112,0,8359384,0
I,N, 8450643,IBM,87236,1280951859,3156,131.440000,1300,55333,1012334,0,8377283,1
I,N, 8450713,IBM,87236,1280951866,336,131.430000,1400,55333,1012334,0,8382739,1
I,N, 8451211,IBM,87236,1280951944,7453,131.425000,1655,55111,1012445,0,8483102,1
I,N, 8450860,IBM,87236,1280951894,2783,131.420000,4900,55000,1012334,0,8374755,0
```

When you send the `<info/>` message, the following event information is retrieved:

- the current page and the number of pages
- the number of events in the subscription
- the total number of retained events in the window

This information can be useful when a subscriber client has not received events and needs to know how many events and pages are available. Here is an example:

```
-- page=1;pages=41;events=201;total=51158
```

Creating and Using the JavaScript WebSocket Publisher

The JavaScript WebSocket publisher enables you to publish events into an event stream processing engine. You can publish events in `xml`, `json`, `csv`, or `properties` format. Send the events individually over the stream. The publisher collects events until its block size is met. It then injects collected events over the publish/subscribe interface into the model.

When you create a JavaScript WebSocket publisher, supply the full pathname (project, continuous query, and window) of the window using the WebSocket protocol:

ws://server:port/eventStreamProcessing/v1/publishers/project/continuous_query/window

The specified *window* must be a Source window.

Additional parameters enable you to change the behavior of the publisher.

Table 3 *Publisher Parameters*

Parameter	Description
format	Specifies the format in which to receive event data. Valid values are xml, json, csv, or properties. The properties format contains each event as a set of <i>name=value</i> pairs separated by new lines. Events are separated by a user-defined separator (a blank line by default). You can set the separator with the <code>separator</code> request parameter. The default value is csv.
separator	Specifies the separator to use between events when you use the properties format. The default value is a blank line.
blocksize	Specifies the number of events to put into an event block for publishing. By default, the publisher injects each event as it is received (<code>blocksize=1</code>).
dateformat	Specifies the format used when you send events with formatted dates. The publisher uses this format to decode dates into numeric values. Any valid date format is accepted. There is no default value.
rate	Specifies the maximum event rate to maintain in events per second. When the publisher attains this rate before 1 second has elapsed, it sleeps for the remainder of that second and then resumes publishing. When you do not set the rate, the publisher publishes events as fast as possible. The default value is 0.
pause	Specifies the number of milliseconds to pause between the injection of events. For each event received, when that event causes an event block to be injected into the ESP server, the publisher pauses for the specified number of milliseconds. The default value is 0. This means that there is no pause between event injections.
opcode	Specifies the default opcode to use when an input event does not include the opcode. Valid values are insert, upsert, and delete. The default value is insert.
schema	Set to true when you want the WebSocket client to return the window schema when a connection is created.

Obtaining Project Statistics through the JavaScript WebSocket API

You can obtain per-window usage statistics for a group of projects in the ESP server. Specify the project for which you want statistics in the URI:

ws://server:port/eventStreamProcessing/v1/projectStats/project

You can also specify a filter to gather statistics for a group of projects:

ws://server:port/eventStreamProcessing/v1/projectStats?filter=filter

You can apply parameters to the WebSocket to change its behavior.

Parameter	Description
format	Specifies the format in which to receive the project statistics information. Valid values are xml and json . The default value is xml .
interval	Specifies the interval, in seconds, at which you want project statistics sent. Specify an integer value. The default value is 5.
minCpu	Specifies the minimum CPU average value that you want included in the project statistics. Specify an integer value. The default value is 5.

Here is an example of received project statistics (in **xml** format):

```
<project-stats>
  <project name='p'>
    <contquery name='cq'>
      <window cpu='18.5485' interval='800065' name='trades' />
      <window cpu='17.6821' interval='800064' name='transform' />
      <window cpu='14.7617' interval='800065' name='largeTrades' />
      <window cpu='7.52938' interval='800066' name='frontRunning' />
      <window cpu='6.4803' interval='800071' name='counter' />
      <window cpu='5.53663' interval='800071' name='addBrokerData' />
    </contquery>
  </project>
</project-stats>
```

Setting Up a Connection to Get Log Messages

You can set up a persistent connection to receive log messages when they are logged inside the server. Send the following request:

```
ws://host:port/eventStreamProcessing/v1/logs
```

Using the ESP Server Connection API

Overview

The ESP Server Connection API uses the WebSocket protocol to perform event stream processing functions with minimal system resources. It uses a single WebSocket connection to complete these tasks:

- subscribe to any number of windows
- publish events
- receive server status information
- monitor the server log

This API can handle subscriptions to all windows with a single WebSocket and two threads. You can use the ESP Server Connection API to start any number of server connections and distribute the workload among them. For example, when you have a model with one window with high throughput, you can use a dedicated server connection to subscribe to only that window. You can then create another server connection that subscribes to multiple windows with low throughput.

The ESP Server Connection API can use the publish/subscribe facilities within the ESP server. You can set up one publish/subscribe subscriber per window, and you can have any number of subscribers using different event filters. All filters share the same publish/subscribe resource, but apply their own individual filters to the data before sending it back to the client.

Connecting to the ESP Server

Use the following request to connect to the ESP server and set up the server connection:

```
ws://host:port/eventStreamProcessing/v1/connect
```

This sets up a WebSocket connection that reads and responds to JSON messages.

Loading Projects

To load projects over the server connection, Base64 encode the XML project definition, and then send the following JSON object:

```
{
  "project": {
    "name": "myproject",
    "id": 1,
    "action": "load",
    "overwrite": true | false,
    "connectors": true | false,
    "validate": true | false,
    "start": true | false,
    "data": Base64 encoded project data,
    "parms": project load options (Javascript object)
  }
}
```

Table 4 Parameters Required to Load a Project

Parameter	Description
name	Specifies the name of the project.
id	Specifies a unique ID within the scope of the connection. Data sent to the client is tagged with this ID. The client can use any method to derive this ID.
action	Specifies load .
connectors	Indicates whether to start connectors. The default value is true .
overwrite	Indicates whether to overwrite the project if it exists. The default value is false .
validate	Indicates whether to validate the project XML code. The default value is true .
start	Indicates whether to start the project. The default value is true .
data	Specifies Base64 data that encodes the XML project definition.

Table 5 *Optional Parameters to Load a Project*

Parameter	Description
parms	When there are tokens in the project code (that is, text enclosed in opening and closing @ signs), the specified values are used to resolve the tokens.

Loading Routers

To load routers over the server connection, encode the XML router definition using Base64 encoding, and then send the following JSON object:

```
{
  "router": {
    "name": "myrouter",
    "id": 1,
    "action": "load",
    "overwrite": true | false,
    "data": Base64 encoded router data,
    "parms": router load options (Javascript object)
  }
}
```

Table 6 *Required Parameters to Load a Router*

Parameter	Description
name	Specifies the name of the router.
id	Specifies a unique ID within the scope of the connection. Data sent to the client is tagged with this ID. The client can use any method to derive this ID.
action	Specifies load .
overwrite	Indicates whether to overwrite the router if it exists. The default value is false .
data	Specifies Base64 data that encodes the XML router definition.

Table 7 *Optional Parameters to Load a Router*

Parameter	Description
parms	When there are tokens in the code (that is, text enclosed in opening and closing @ signs), the specified values are used to resolve the tokens.

Event Subscribers

Overview

Event subscribers deliver events either as soon as the event occurs or after a specified time interval. When you subscribe to events, you must supply an ID that is unique within the scope of the server connection. Anytime that the server delivers information (schema or events) for that subscriber, this ID is included with the data so that it can be mapped to the appropriate resource.

When you subscribe to events, you can choose to send the window schema before any other data. Along with the schema, a subscriber sends ESP event information. You use the `set` action to subscribe to events and to set the parameters of an existing subscriber. You can set most of the subscriber parameters (for example, `filter` and `pagesize`) after the subscriber has been created.

Because subscribers consume resources in the server while they are active, you should always close them after they are no longer needed.

There are two types of event subscribers:

- Event Streams
- Event Collections

Creating and Closing Event Streams

An event stream is a flow of events that is similar to a UNIX tail. When an event occurs in the ESP model in the server, it is placed into the event stream. Clients can read this stream and see the events as they occur.

To create an event stream, send the following JSON object:

```
{
  "event-stream": {
    "action": "set",
    "id": "1",
    "window": "project/contquery/window"
  }
}
```

Specify the following parameters:

Table 8 Parameters Required to Create an Event Stream

Parameter	Description
request	Specifies event-stream .
action	Specifies set .
id	Specifies a unique ID within the scope of the connection. Data sent back to the client is tagged with this ID. The client can use any method of choice to derive this ID.
window	Specifies the path of the window to which to subscribe. Use the format <i>project/contquery/window</i> .

Table 9 Optional Parameters to Create an Event Stream

Parameter	Description
schema	Specifies whether schema data is included. Valid values are true and false . When true , the schema is sent upon event stream creation. The default value is false .
interval	Specifies the delivery interval in milliseconds. When you specify a value greater than 0, the stream holds onto events for that interval before delivering them.
maxevents	Specifies the maximum number of events to put into the stream at any one time. The default value is 100.
format	Specifies the event delivery format. The valid values are as follows: ■ json - events in JSON format ■ xml - events in XML format ■ csv - events in CSV format ■ properties - events in newline separated <i>field=value</i> pairs with a separator between events ■ cbor - events in Concise Binary Object Representation (CBOR) format
precision	Specifies the floating point precision.
separator	Specifies the separator to use between events when using the properties format.
filter	Specifies a functional filter to use in order to filter the stream. For example, suppose that an event has a field named <code>price</code> . Specifying <code>"filter": "gt(\$price,100) "</code> returns events where <code>price</code> is greater than 100.

Parameter	Description
<code>include</code>	Specifies a comma-separated list of event fields to include when the events are delivered to the client. The key fields are always included.
<code>exclude</code>	Specifies a comma-separated list of the event fields to exclude when the events are delivered to the client. The key fields are always included.

When a window produces a huge number of events, you can throttle the number of events that are put into the stream. Do this by specifying values for the `maxevents` and `interval` parameters. The `maxevents` parameter limits the number of events that get put into the stream at any one time. The `interval` parameter specifies a time, in milliseconds, that the ESP server waits before putting events into the stream.

For example, when you set `maxevents` to 1000 and `interval` to 1000 ms (or 1 second), the server collects events from the publish/subscribe interface for 1 second. When more than `maxevents` events occur during that second, the oldest events are dropped and are not put into the stream. At the end of 1 second, the server puts all events into the stream. For clients with heavy processing loads (for example, those that render graphs), this action enables the client to limit the number of events that it must process.

To close an event stream, send the following JSON object:

```
{
  "event-stream": {
    "id": "1",
    "action": "close"
  }
}
```

Specify the following parameters:

Table 10 *Parameters to Close an Event Stream*

Parameter	Description
<code>request</code>	Specifies event-stream .
<code>id</code>	Specifies the <i>id</i> of the event stream that you created.
<code>action</code>	Specifies close .

Creating and Closing Event Collections

An event collection is a view into a stateful window. The view is defined by the current page. When an event collection is created, it sets the page size for that collection. This determines the maximum number of events that are sent from the server to the client. In addition, the server sets up an internal publish/subscribe instance to receive events for that window. When an event is not in the current page, the event is ignored. Otherwise, the event is delivered to the client.

To create an event collection, send the following JSON object:

```
{
  "event-collection": {
    "action": "set",
    "id": "1",
    "window": "project/contquery/window"
  }
}
```

Specify the following parameters:

Table 11 Parameters Required to Create an Event Collection

Parameter	Description
request	Specifies event-collection .
action	Specifies set .
id	Specifies a unique ID within the scope of the connection. Data sent back to the client is tagged with this ID. The client can use any method of choice to derive this ID.
window	Specifies the path of the window to subscribe to. Use the format <i>project/contquery/window</i> .

Table 12 Optional Parameters to Create an Event Collection

Parameter	Description
schema	Specifies either json or xml . When this is set, the schema is sent upon event stream creation. The default value is json .
interval	Specifies the delivery interval in milliseconds. When you specify a value greater than 0, the stream holds onto events for that interval before delivering them.
pagesize	Specifies the number of events in a page. The client can view a single page at a time, so the page size is the maximum number of events that the client can view at any time.
info	Specifies a numeric value that, when set, causes the server to send collection information to the client at intervals specified by the value. When the client wants to know the number of pages in a collection, this capability provides that data.
format	Specifies the event delivery format. The valid values are as follows: ■ json - events in JSON format ■ xml - events in XML format

Parameter	Description
	■ csv - events in CSV format ■ properties - events in newline separated <i>field=value</i> pairs with a separator between events ■ cbor - events in CBOR format
precision	Specifies the floating point precision.
separator	Specifies the separator to use between events when using the properties parameter.
filter	Specifies a functional filter to use in order to filter the stream. For example, "filter": "gt(\$price,100)" returns events where price is greater than 100.
include	Specifies a comma-separated list of event fields to include when the events are delivered to the client. The key fields are always included.
exclude	Specifies a comma-separated list of the event fields to exclude when the events are delivered to the client. The key fields are always included.

To close an event collection, send the following JSON object:

```
{
  "event-collection": {
    "id": "1",
    "action": "close"
  }
}
```

Specify the following parameters:

Table 13 Parameters to Close an Event Collection

Parameter	Description
id	Specifies the <i>id</i> of the event collection that you created.
action	Specifies close .

Event Subscriber Output

Window Schema

Window schema are returned upon stream creation in either JSON or XML. The subscriber ID is specified so that the client can map the schema to the appropriate resource. The schema output looks like this:

Example Code 1 *Event Subscriber Window Schema in JSON*

```
{
  "schema": {
    "@id": "1",
    "fields": [
      {
        "field": {
          "@key": "true",
          "@name": "id",
          "@type": "string"
        }
      },
      {
        "field": {
          "@name": "data",
          "@type": "string"
        }
      },
      {
        "field": {
          "@name": "i32s",
          "@type": "array(i32) "
        }
      },
      {
        "field": {
          "@name": "i64s",
          "@type": "array(i64) "
        }
      },
      {
        "field": {
          "@name": "doubles",
          "@type": "array(dbl) "
        }
      },
      {
        "field": {
          "@name": "number",
          "@type": "double"
        }
      },
      {
        "field": {
```

```

        "@name": "date",
        "@type": "date"
    },
    {
        "field": {
            "@name": "time",
            "@type": "stamp"
        }
    }
],
"schema-
string": "id*:string,data:string,i32s:array(i32),i64s:array(i64),doubles:array(dbl),num
ber:double,date:date,time:stamp"
}
}

```

Example Code 2 Event Subscriber Window Schema in XML

```

<schema id="1">
  <fields>
    <field key="true" name="id" type="string"/>
    <field name="data" type="string"/>
    <field name="i32s" type="array(i32)"/>
    <field name="i64s" type="array(i64)"/>
    <field name="doubles" type="array(dbl)"/>
    <field name="number" type="double"/>
    <field name="date" type="date"/>
    <field name="time" type="stamp"/>
  </fields>
  <schema-string>id*:string,data:string,i32s:array(i32),i64s:array(i64),
    doubles:array(dbl),number:double,date:date,time:stamp</schema-string>
</schema>

```

Events

Events are delivered as they occur or after the delivery interval has passed. Events can be delivered in any of the following four formats:

- JSON
- XML
- CSV
- properties

In each of these formats, the subscriber ID is specified so that the client can map the events to the appropriate resource. For the JSON and XML formats, the subscriber ID is embedded within the object. For the properties and CSV formats, the event data is preceded by a JSON header that looks like this:

```

{
  "events": {
    "id": "1",
    "page": 0,
    "pages": 1,
    "total": 5,
    "events": 5,

```

```

        "format": "csv"
    }
}
I,N, Joe,9,14,17,0,7,47
I,N, Curt,8,11,21,2,12,54
I,N, John,4,10,18,3,7,42
I,N, Lisa,10,12,23,3,8,56
I,N, Sally,5,10,15,3,11,44

```

Note: When the events are not part of a page load, the page information is absent.

Example Code 3 Event in JSON Format

```

{
  "events": {
    "@id": "1",
    "entries": [
      {
        "@opcode": "insert",
        "@timestamp": 1563999661266330,
        "data": "data 1",
        "date": 1563999661,
        "doubles": [10.000000;20.000000;35.400000;12.000000;50.000000],
        "i32s": [10;20;30;40;50],
        "i64s": [10;20;30;40;50],
        "id": "4f3e6038-593d-4cbc-a1b0-428d03865de9",
        "number": 30.000000,
        "time": 1563999661000000
      }
    ]
  }
}

```

Example Code 4 Event in XML Format

```

<events id="1">
  <entries>
    <event opcode="insert" timestamp="1563999844040150">
      <data>data 1</data>
      <date>1563999844</date>
      <doubles>[10.000000;20.000000;35.400000;12.000000;50.000000]</doubles>
      <i32s>[10;20;30;40;50]</i32s>
      <i64s>[10;20;30;40;50]</i64s>
      <id>e3b0a6a1-9743-4aee-b111-34b40b2cde7d</id>
      <number>67.000000</number>
      <time>1563999844000000</time>
    </event>
  </entries>
</events>

```

Example Code 5 Event in Properties Format

```

{
  "events": {
    "id": "1",
    "format": "properties"
  }
}

```

```

}
opcode=insert
timestamp=1564000645319757
data=data 1
date=1564000645
doubles=[10.000000;20.000000;35.400000;12.000000;50.000000]
i32s=[10;20;30;40;50]
i64s=[10;20;30;40;50]
id=aa4fa535-54f5-4189-a74a-2bcd7c6cc9de
number=28.000000
time=1564000645000000
* ++ *

```

Example Code 6 *Event in CSV Format*

```

{
  "events":{
    "id":"1",
    "format":"csv"
  }
}
I,N, 62443978-ef9a-46eb-a14f-f8f199400b9b,data 1,[10;20;30;40;50],[10;20;30;40;50],
[10.000000;20.000000;35.400000;12.000000;50.000000],26.000000

```

Event Publishers

Overview

Event publishers publish data into the Source window of a SAS Event Stream Processing model. There are two ways to publish events:

- Send the events directly over the connection into the model.
- Send a URL over the connection and have the server pull events into the model from that URL.

Publishers

Publishers enable you to stream event data to the server over the connection as each event occurs. The input format of the event data can be JSON or CBOR.

Unlike data and URL publishers, the server does not need to spawn a new thread to publish data each time it creates a publisher. A publisher sends a message to the client when there is an error. Otherwise, the publisher remains silent.

Create a publisher with the following parameters:

Table 14 Parameters Required to Create a Publisher

Parameter	Description
id	Specifies the ID . This ID is used when you send events to the publisher.
window	Specifies the path of the window to publish to. Use the format <i>project/contquery/window</i> .
action	Specifies set .

Table 15 Optional Parameter to Create a Publisher

Parameter	Description
schema	Specifies whether schema data is included. Valid values are true and false . The default value is false .
format	Specifies the event delivery format. The valid values are as follows: ■ json - events in JSON format ■ cbor - events in CBOR format

Here is an example of how to create a publisher:

```
{
  "publisher": {
    "id":1,
    "window":"primary/cq/trades",
    "action":"set"
  }
}
```

After you create it, the publisher is available to receive events which it publishes into its associated Source window. Because the publisher uses resources as long as it exists, you should close it when it is no longer needed. For example, you could close a publisher by using the following code:

```
{
  "publisher": {
    "id":1,
    "action":"close"
  }
}
```

In order to publish events into a window, you send JSON objects over the connection that contains the publisher ID along with the event data. Those JSON objects look like this:

Example Code 7 JSON Objects Sent That Contain the Publisher ID and Event Data

```
{
  "publisher": {
    "id":1,
    "action":"publish",
    "data": [
```

```
[
  {
    "id":1,
    "symbol":"A",
    "price":100,
    "quant":1000,
    "broker":"moe"
  },
  {
    "id":2,
    "symbol":"A",
    "price":50,
    "quant":600,
    "broker":"moe"
  }
],
{
  "id":3,
  "symbol":"A",
  "price":100,
  "quant":1000,
  "broker":"moe"
}
]
```

In the JSON code, you must specify **publisher** and provide the publisher **id**. The action must be **publish**. The data field contains the event data. Specify event data as an array of JSON objects whose fields map to the fields in the event. When you include events in a nested array, they are published into the ESP model as a single transaction.

Data Publishers

Data publishers enable you to send Base64 encoded event data to the server over the connection. The input format of the event data can be one of the following:

- CSV
- JSON
- XML

Because this operation could take a significant amount of time, the server spawns a new thread to publish data each time that it creates a data publisher. When the thread has completed, a message is sent over the connection to the sending client.

Create a data publisher with the following parameters:

Table 16 Parameters Required to Create a Data Publisher

Parameter	Description
id	Specifies a unique ID within the scope of the connection. This ID is returned when the data publisher has finished its work.

Parameter	Description
window	Specifies the path of the Source window into which to publish the events. Use the format <i>project/contquery/window</i> .
data	Specifies the Base64 encoded event data. The default value is JSON .

Table 17 Optional Parameter to Create a Data Publisher

Parameter	Description
informat	Specifies the input format of the event data. Valid values include JSON, XML, and CSV.

Here is an example of how to create a data publisher:

```
{
  "data-publisher": {
    "id": "1",
    "window": "primary/cq/rawTrades",
    "data": "<Base64 encoded event data>",
    "informat": "csv"
  }
}
```

When all events have been read and published into the ESP server, the server sends a completion message back to the client:

```
{
  "data-publisher": {
    "@id": "1",
    "@complete": true
  }
}
```

URL Publishers

URL publishers enable you to send a URL to the server over the connection. The URL points to an event data source in one of the following formats:

- CSV
- JSON
- XML
- binary

Because this operation can take a significant amount of time, the server spawns a new thread to read the data from the URL and publish each time that it creates a URL publisher. When the thread has completed, a message is sent back over the connection to the sending client.

Table 18 Required Parameters for a URL Publisher

Parameter	Description
id	Specifies a unique ID within the scope of the connection. This ID is returned when the URL publisher has completed its work.
window	Specifies the path of the Source window into which to publish the events. Use the format <i>project/contquery/window</i> .
url	Specifies the URL that contains the event data.

Table 19 Optional Parameter for the URL Publisher

Parameter	Description
informat	Specifies the input format of the event data. Valid values include bin, JSON, XML, and CSV.

Here is an example of how to create a URL publisher:

```
{
  "url-publisher": {
    "id": "1",
    "window": "primary/cq/rawTrades",
    "url": "file://data/trades10M.bin",
    "informat": "bin"
  }
}
```

After all events have been read from the URL and published, the server sends a completion message back to the client:

```
{
  "url-publisher": {
    "@id": "1",
    "@complete": true,
    "url": "file://data/trades10M.bin"
  }
}
```

Retrieving ESP Server Statistics

You can retrieve ESP server information such as per-window CPU usage, event counts, and memory usage of the server connection.

Here is an example of retrieving server statistics:

```
{
  "stats": {
```

```

    "minCpu":5,
    "interval":1,
    "format":"xml",
    "counts":true,
    "config":true,
    "memory":true
  }
}

```

Use the following parameters to retrieve server statistics:

Table 20 *Optional Parameters to Retrieve Server Statistics*

Parameter	Description
minCpu	Specifies the minimum window CPU percentage to report. The default value is 5.
interval	Specifies the interval, in seconds, at which to deliver the statistics. The default value is 1.
format	Specifies the delivery format. The valid values are as follows: ■ xml - statistics in XML format (default) ■ json - events in JSON format
counts	Specifies whether window counts are returned. Valid values are true and false . The values of the window counts represent the number of events that are currently in the window. The default value is false .
config	Specifies whether server configuration information is returned. Valid values are true and false . The default value is false .
memory	Specifies whether information about server memory usage is returned. Valid values are true and false . The default value is false .

Note: Content varies depending on the request attributes.

Here are examples of server statistic information that is returned in XML and JSON formats:

Example Code 8 *Server Statistic Information in XML Format*

```

<stats>
  <project name="primary" state="started">
    <contquery name="cq">
      <window count="1" cpu="0.000497167" interval="1005698" name="counter"/>
    </contquery>
  </project>
  <project name="secondary" state="started">
    <contquery name="cq">
      <window count="5" cpu="0" interval="1005697" name="brokerAlertsAggr"/>
      <window count="10" cpu="0" interval="1005697" name="venueAlertsAggr"/>
      <window count="5" cpu="0" interval="1005697" name="violationCounts"/>
    </contquery>
  </project>
</stats>

```

```

    </contquery>
  </project>
  <server-memory>
    <system>386696</system>
    <virtual>1363</virtual>
    <resident>246</resident>
  </server-memory>
  <properties>
    <property name="http">26000</property>
    <property name="loglevel">common.http=debug,server.connection=debug</property>
    <property name="model">file://model.xml</property>
    <property name="nocleanup"/>
    <property name="novalidate"/>
    <property name="port">26001</property>
    <property name="pubsub">26001</property>
  </properties>
  <meteringInfo connected="false" enabled="true" host="localhost" port="31001"/>
</stats>

```

Example Code 9 Server Statistic Information in JSON Format

```

{
  "stats":{
    "project":{
      "@name":"secondary",
      "@state":"started",
      "contquery":[
        "@name":"cq",
        {
          "window":{
            "@count":"5",
            "@cpu":"0",
            "@interval":"1005488",
            "@name":"brokerAlertsAggr"
          }
        },
        {
          "window":{
            "@count":"10",
            "@cpu":"0",
            "@interval":"1005485",
            "@name":"venueAlertsAggr"
          }
        },
        {
          "window":{
            "@count":"5",
            "@cpu":"0",
            "@interval":"1005485",
            "@name":"violationCounts"
          }
        }
      ]
    },
    "server-memory":{
      "system":"386696",
      "virtual":"1371",

```

```

        "resident": "246"
    },
    "properties": [
        {
            "property": {
                "@name": "http",
                "*value": "26000"
            }
        },
        {
            "property": {
                "@name": "loglevel",
                "*value": "common.http=debug,server.connection=debug"
            }
        },
        {
            "property": {
                "@name": "model",
                "*value": "file://model.xml"
            }
        },
        {
            "property": {
                "@name": "nocleanup"
            }
        },
        {
            "property": {
                "@name": "novalidate"
            }
        },
        {
            "property": {
                "@name": "port",
                "*value": "26001"
            }
        },
        {
            "property": {
                "@name": "pubsub",
                "*value": "26001"
            }
        }
    ],
    "meteringInfo": {
        "@connected": "false",
        "@enabled": "true",
        "@host": "localhost",
        "@port": "31001"
    }
}

```

Retrieving Server Logs

You can retrieve the ESP server logs over the server connection. You can specify the logs to be delivered in either XML, JSON, or plain text. You can also specify a regular expression filter. When you are looking for specific entries in a busy log, a filter enables you to see only those messages that meet specific criteria.

Here is an example of a filter within a server log request:

```
{
  "logs": {
    "capture": true,
    "filter": "",
    "format": "text"
  }
}
```

Use the following parameters to request server logs:

Table 21 *Parameters Required to Retrieve Server Logs*

Parameter	Description
capture	Specifies whether to turn log capture on or off. Valid values are true and false . Because log capture uses server resources, turn this off unless logs are required.

Table 22 *Optional Parameters to Retrieve Server Logs*

Parameter	Description
filter	Specifies a regular expression that is run against each log message to determine whether the message is returned.
format	Specifies xml , json , or text . The default value is xml .

Retrieving Model Information

The models that run on an ESP server are accessible through the server connection. You can specify to retrieve model information in either XML or JSON format.

Here is an example of retrieving model information:

```
{
  "model": {
```

```

    "schema":true,
    "counts":true,
    "index":true,
    "perms":true,
    "pubsub":true,
    "xml":true,
    "format":"json"
  }
}

```

Use the following parameters to retrieve model information:

Table 23 *Optional Parameters to Retrieve Model Information*

Parameter	Description
schema	Specifies whether schema data is included. Valid values are true and false . The default value is false .
counts	Specifies whether window counts are included. Valid values are true and false . The values of window counts represent the number of events that are currently in the window. The default value is false .
index	Specifies whether window index types are included. Valid values are true and false . The default value is false .
perms	Specifies whether permissions are included. Valid values are true and false . The default value is false .
pubsub	Specifies whether to include only windows that are publish/subscribe enabled. Valid values are true and false . The default value is false .
xml	Specifies whether window XML is included. Valid values are true and false . The default value is false .
format	Specifies xml or json . The default value is false .

Sending and Receiving Binary Data

Overview

The ESP Server Connection API uses a binary representation of JSON called Concise Binary Object Representation (CBOR) to enable the ESP server to send and receive binary data. Using CBOR avoids intensive operations such as Base64 encoding and decoding, and other data transformations when handling binary data. For more information, see [CBOR Documentation](#).

Note: The ESP Server Connection API adds a type named 'B' to the standard implementation. This type signifies true binary data. When the ESP server encounters this type, data is treated as strictly binary and no transformations are performed. Event data containing blobs for fields such as image data can be injected directly into an ESP model as binary data. This enables the ESP server to handle very high throughput rates for large data.

Encoding and Decoding in JavaScript

The `api.encode()` and `api.decode()` functions within the JavaScript connect API enable you to encode and decode JavaScript objects into and out of CBOR. To encode your requests to the ESP server, you can use code like this:

```
var request = {
  "publisher":{
    "id":"publish",
    "action":"set",
    "window":"myproject/cq/clicks",
    "schema":"true"
  }
}

ws.send(api.encode(request))
```

To decode data that comes back from the server (for example, when you subscribe to events in CBOR format), you can use code like this:

```
function
handleMessage(message)
{
  if (typeof(message) == "string")
  {
    console.log(message);
  }
  else
  {
    var buffer = new ArrayBuffer(message.length);
    var view = new Uint8Array(buffer);
    for (var i = 0; i < message.length; ++i)
    {
      view[i] = message[i];
    }

    var o = api.decode(buffer);
    console.log(api.stringify(o));
  }
}
```

The following NodeJS program loads an event stream processing model and then publishes to and subscribes from events. All requests are encoded in CBOR format before they are sent to the ESP server. The event subscriber specifies CBOR format, so it receives binary events from the ESP server.

```
var api = require("espconnect");

function
```

```

handleMessage(message)
{
    if (typeof(message) == "string")
    {
        console.log(message);
    }
    else
    {
        var buffer = new ArrayBuffer(message.length);
        var view = new Uint8Array(buffer);
        for (var i = 0; i < message.length; ++i)
        {
            view[i] = message[i];
        }

        var o = api.decode(buffer);
        console.log(api.stringify(o));
    }
}

// set up a websocket to communicate to the ESP server over the connect API
var url = "ws://espsrv01:7777/eventStreamProcessing/v1/connect"
var ws = api.createWebSocket(url, {open:connected,message:handleMessage});

function
connected()
{
    var model = "<project threads='4' pubsub='auto'>\
        <contqueries>\
            <contquery name='cq' trace='clicks'>\
                <windows>\
                    <window-source name='clicks' insert-only='true'
index='pi_EMPTY'>\
                        <schema-string>id*:string,element:string,x:int32,y:int32</
schema-string>\
                            </window-source>\
                            <window-copy name='copy'>\
                                <retention type='bytime_sliding'>10 seconds</retention>\
                            </window-copy>\
                            <window-aggregate name='clicksAggr'>\
                                <schema-string>element*:string,count:int32</schema-string>
\
                                    <output>\
                                        <field-expr>ESP_aCount()</field-expr>\
                                    </output>\
                                </window-aggregate>\
                            </windows>\
                            <edges>\
                                <edge source='clicks' target='copy' />\
                                <edge source='copy' target='clicksAggr' />\
                            </edges>\
                        </contquery>\
                    </contqueries>\
                </project>";

    var request = {

```

```

        "project":{
            "name":"myproject",
            "id":"sendproj",
            "action":"load",
            "overwrite":"true",
            "data":new Buffer.from(model).toString("base64")
        }
    };

    ws.send(api.encode(request))

    // create and send a request to set up a publisher

    request = {
        "publisher":{
            "id":"publish",
            "action":"set",
            "window":"myproject/cq/clicks",
            "schema":"true"
        }
    };
    ws.send(api.encode(request));

    // create and send a request to set up a subscriber

    request = {
        "event-collection":{
            "id":"subscriber",
            "window":"myproject/cq/clicksAggr",
            "schema":"true",
            "format":"cbor",
            "interval":"0"
        }
    };
    ws.send(api.encode(request));

    // create and send 10 requests to set publish data

    for (var i = 0; i < 10; i++)
    {
        request = {
            "publisher":{
                "id":"publish",
                "action":"publish",
                "data":[
                    {
                        "id":api.guid(),
                        "element":"red",
                        "x":"502",
                        "y":"145"
                    }
                ]
            }
        };
        ws.send(api.encode(request))
    }
}

```

}