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 the ESP client to communicate with the ESP server through this API. You can also use any client language or platform with WebSocket capabilities (such as Java, Javascript, C++, Python, and IOS). 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

Inside 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:

\[\text{name: value}\]

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:

<table>
<thead>
<tr>
<th>Status Header Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>The WebSocket is ready to be used.</td>
</tr>
<tr>
<td>401</td>
<td>The WebSocket connection requires authentication data.</td>
</tr>
</tbody>
</table>

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:

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

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>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mode</strong></td>
<td>Specifies the mode of the subscriber. Valid values are <code>updating</code> and <code>streaming</code>. An <code>updating</code> 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 <code>streaming</code> 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.</td>
</tr>
<tr>
<td><strong>pagesize</strong></td>
<td>Specifies the number of events to keep in an event page for this subscriber.</td>
</tr>
<tr>
<td><strong>separator</strong></td>
<td>Specifies the separator to use between events when you use the <code>properties</code> format.</td>
</tr>
<tr>
<td><strong>filter</strong></td>
<td>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&amp;T that flow through the <code>largeTrades</code> window:</td>
</tr>
</tbody>
</table>

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

| **sort** | Specifies how to sort events that are streamed through a window. The format is `event_field:sort_direction`. The `sort_direction` value is optional and defaults to `descending`. For example, suppose you want to subscribe to stock transactions for AT&T that flow through the `largeTrades` window and sort them by quantity: |

```
ws://espsrv01:26000/eventStreamProcessing/v1/subscribers/p/cq/largeTrades?filter=eq(symbol,'T')&sort=quantity
```

**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.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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 name=value 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 separator request parameter.  
Note: Page information is included only in data that is returned from load commands. Page information does not return with events received during subscription notifications. |
| 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.  
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 interval property, the events are collected for that amount of time and only sent one time per specified interval. |
| schema    | Set to true when you want the WebSocket client to return the window schema when a connection is created. |
| counts    | 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. |
| info      | Use this parameter to instruct the subscriber to periodically deliver event information to the client. The attributes are as follows:  
- page - the current page number  
- pages - the total number of pages  
- events - the number of events in the subscription  
- total - the number of retained events in the window (this matches the events value when no filter exists)  
Suppose you sent the following request:  
dfespxml_client -url "ws://espsrv01:26000/eventStreamProcessing/v1/subscribers/p/cq/largeTrades?format=xml&info=5"  
Something like the following output appears on the console every 5 seconds:  
<info page='1' pages='1024' events='51158' total='51158'/>  
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>
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>page</td>
<td>Specifies a specific page number or a page number relative to the current page number. You can also specify one of the following values:</td>
</tr>
<tr>
<td></td>
<td>- first - first page of events from the window</td>
</tr>
<tr>
<td></td>
<td>- last - last page of events from the window</td>
</tr>
<tr>
<td></td>
<td>- next - next page of events (current page + 1) from the window</td>
</tr>
<tr>
<td></td>
<td>- prev - previous page of events (current page - 1) from the window</td>
</tr>
<tr>
<td>pagesize</td>
<td>Specifies the number of events to keep in an event page for this subscriber.</td>
</tr>
<tr>
<td>sort</td>
<td>Specifies how to sort events that are streamed through a window. The format is: event_field:sort_direction. The sort_direction value is optional and defaults to descending.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Note: The sort property is meaningful only in updating subscribers. Streaming subscribers always send events in the order received.</td>
</tr>
<tr>
<td>interval</td>
<td>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.</td>
</tr>
<tr>
<td>mode</td>
<td>Specifies the mode of the subscriber. Valid values are updating and streaming.</td>
</tr>
<tr>
<td>filter</td>
<td>Specifies a functional filter to apply to the events being processed by the subscriber.</td>
</tr>
</tbody>
</table>

Suppose that you create the following subscriber:

```text
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,87236,1280951996,3436,131.310000,1280951996,0,55222,1012445,0,8123541,1
I,N, 15795961,T,87236,1280951996,3436,131.310000,1400,55111,1012334,0,8093853,1
I,N, 15795959,T,87236,1280951996,5317,26.650000,1300,55999,1012112,0,8374755,1
I,N, 8451702,IBM,87236,1280951996,652,131.314000,1280951995,0,55222,1012445,0,8123541,1
I,N, 8451701,IBM,87236,1280951996,593,131.310000,1200,55111,1012445,0,8382948,0
```

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>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 name=value 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 separator request parameter. The default value is csv.</td>
</tr>
<tr>
<td>separator</td>
<td>Specifies the separator to use between events when you use the properties format. The default value is a blank line.</td>
</tr>
<tr>
<td>blocksize</td>
<td>Specifies the number of events to put into an event block for publishing. By default, the publisher injects each event as it is received (blocksize=1).</td>
</tr>
<tr>
<td>dateformat</td>
<td>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.</td>
</tr>
<tr>
<td>rate</td>
<td>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.</td>
</tr>
<tr>
<td>pause</td>
<td>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.</td>
</tr>
<tr>
<td>opcode</td>
<td>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.</td>
</tr>
<tr>
<td>schema</td>
<td>Set to true when you want the WebSocket client to return the window schema when a connection is created.</td>
</tr>
</tbody>
</table>

---

**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.

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

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

You can also use the ESP client to stream the log of an ESP server to its own console:

dfesp_xml_client -url "ws://host:port/eventStreamProcessing/v1/logs"
Using the ESP Server Connection API

Overview

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 tasks:

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

With the existing WebSocket API, each subscriber or publisher creates its own connection to the server. Each of these connections does the following:

- starts a thread in the server that reads information from the client
- starts a publish/subscribe thread

This combination of multiple connections and multiple threads can stress the ESP server. Thus, when you run a model that consists of a large number of windows and then you subscribe to multiple windows, the model's resource consumption can strain the system. For example, running a model with 20 windows requires 20 WebSockets and 20 threads. In contrast, the ESP Server Connection 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. With the existing WebSocket API, subscribing to the same window with three different filters requires three separate WebSockets. Each of these WebSockets must have its own publish/subscribe subscriber within the server. Thus, you have three sockets, three threads, and three publish/subscribe instances to provide the event information. With the ESP Server Connection API, you set up one publish/subscriber per window. You can have any number of subscribers using different event filters. All filters share the same publish/subscriber 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.
Event Subscribers

Overview

Event subscribers deliver events either as soon as the event occurs or after a specified time interval. When you create a subscriber, 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 create a subscriber, 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 both create a subscriber and to set the parameters of an existing subscriber. You can set most of the subscriber parameters (such as `filter`, `pagesize`, and so on.) 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, specify the following parameters:

*Table 4  Parameters Required to Create an Event Stream*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>Specify <code>event-stream</code>.</td>
</tr>
<tr>
<td>action</td>
<td>Specify <code>set</code>.</td>
</tr>
<tr>
<td>id</td>
<td>Specify 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.</td>
</tr>
<tr>
<td>window</td>
<td>Specify the path of the window to which to subscribe. Use the format <code>project/contquery/window</code>.</td>
</tr>
</tbody>
</table>
### Table 5  Optional Parameters to Create an Event Stream

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>schema</strong></td>
<td>Specify <code>true</code> or <code>false</code> to send the schema. When <code>true</code>, the schema is sent upon event stream creation. The default value is <code>false</code>.</td>
</tr>
<tr>
<td><strong>interval</strong></td>
<td>Specify the delivery interval in milliseconds. When you specify a value greater than 0, the stream holds onto events for that interval before delivering them.</td>
</tr>
<tr>
<td><strong>maxevents</strong></td>
<td>Specify the maximum number of events to put into the stream at any one time. The default value is 100.</td>
</tr>
</tbody>
</table>
| **format** | Specify 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 `field=value` pairs with a separator between events |
| **precision** | Specify the floating point precision. |
| **separator** | Specify the separator to use between events when using the properties format. |
| **filter** | Specify a functional filter to use in order to filter the stream.  
For example, suppose that an event has a field named `price`. Specifying "filter": "gt($price, 100)" returns events where `price` is greater than 100.  
For more information about the supporting functions, see [SAS Event Stream Processing: Using Source and Derived Windows](#). |
| **include** | Specify a comma-separated list of event fields to include when the events are delivered to the client. The key fields are always included. |
| **exclude** | Specify a comma-separated list of the event fields to exclude when the events are delivered to the client. The key fields are always included. |

For example:

```json
{
    "request": "event-stream",
    "action": "set",
    "id": "1",
    "window": "secondary/cq/brokerAlertsAggr"
}
```

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. So, 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 not be 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 enables the client to limit the number of events that it must process.

To close an event stream, specify the following parameters:

**Table 6 Parameters to Close an Event Stream**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>Specify <code>event-stream</code>.</td>
</tr>
<tr>
<td>id</td>
<td>Specify the <code>id</code> of the event stream that you created.</td>
</tr>
<tr>
<td>action</td>
<td>Specify <code>close</code>.</td>
</tr>
</tbody>
</table>

For example:

```json
{
    "request":"event-stream",
    "id":1,
    "action":"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, specify the following parameters:

**Table 7 Parameters Required to Create an Event Collection**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>Specify <code>event-collection</code>.</td>
</tr>
<tr>
<td>action</td>
<td>Specify <code>set</code>.</td>
</tr>
<tr>
<td>id</td>
<td>Specify 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.</td>
</tr>
<tr>
<td>window</td>
<td>Specify the path of the window to which to subscribe. Use the format <code>project/contquery/window</code>.</td>
</tr>
</tbody>
</table>
### Optional Parameters to Create an Event Collection

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>schema</strong></td>
<td>Specify either json or xml. When this is set, the schema is sent upon event stream creation. The default value is json.</td>
</tr>
<tr>
<td><strong>interval</strong></td>
<td>Specify the delivery interval in milliseconds. When you specify a value greater than 0, the stream holds onto events for that interval before delivering them.</td>
</tr>
<tr>
<td><strong>pagesize</strong></td>
<td>Specify 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.</td>
</tr>
<tr>
<td><strong>info</strong></td>
<td>Specify 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.</td>
</tr>
</tbody>
</table>
| **format** | Specify 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 field=value pairs with a separator between events |
| **precision** | Specify the floating point precision. |
| **separator** | Specify the separator to use between events when using the properties parameter. |
| **filter** | Specify 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** | Specify a comma-separated list of event fields to include when the events are delivered to the client. The key fields are always included. |
| **exclude** | Specify a comma-separated list of the event fields to exclude when the events are delivered to the client. The key fields are always included. |

For example:

```json
{
    "request":"event-collection",
    "action":"set",
    "id":"1",
    "window":"secondary/cq/brokerAlertsAggr"
}
```

To close an event collection, specify the following parameters:
Table 9  Parameters to Close an Event Collection

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>Specify <code>event-collection</code>.</td>
</tr>
<tr>
<td>id</td>
<td>Specify the <code>id</code> of the event collection that you created.</td>
</tr>
<tr>
<td>request</td>
<td>Specify <code>close</code>.</td>
</tr>
</tbody>
</table>

For example:

```
{
    "request":"event-collection",
    "id":1,
    "action":"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**

```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)"
                }
            }
        ]
    }
}
```
"@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),number: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
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": [
            {
                "event": {
                    "opcode": "insert",
                    "timestamp": "156399966126630",
                    "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>
    </event>
    <event opcode="insert" timestamp="1563999844040150">
        <data>data 2</data>
        <date>1563999844</date>
        <doubles>[10.000000;20.000000;35.400000;12.000000;50.000000]</doubles>
    </event>
    <event opcode="insert" timestamp="1563999844040150">
        <data>data 3</data>
        <date>1563999844</date>
        <doubles>[10.000000;20.000000;35.400000;12.000000;50.000000]</doubles>
    </event>
    <event opcode="insert" timestamp="1563999844040150">
        <data>data 4</data>
        <date>1563999844</date>
        <doubles>[10.000000;20.000000;35.400000;12.000000;50.000000]</doubles>
    </event>
    <event opcode="insert" timestamp="1563999844040150">
        <data>data 5</data>
        <date>1563999844</date>
        <doubles>[10.000000;20.000000;35.400000;12.000000;50.000000]</doubles>
    </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 an 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.

Create an event publisher with the following parameters:
### Table 10 Parameters Required to Create an Event Publisher

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>Specify <code>publisher</code>.</td>
</tr>
<tr>
<td>id</td>
<td>Specify the ID. This ID is used when you sent events to the publisher.</td>
</tr>
<tr>
<td>window</td>
<td>Specify the path of the window into which to publish. Use the format <code>project/contquery/window</code>.</td>
</tr>
<tr>
<td>action</td>
<td>Specify <code>set</code>.</td>
</tr>
</tbody>
</table>

For example:

```json
{
  "request":"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, close it when it is no longer needed. For example:

```json
{
  "request":"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

```json
{
  "request":"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"
    }
  ]
}
```
In the JSON code, you must specify `publisher` for the `request` 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.

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 any of four formats:

- CSV
- JSON
- XML
- binary

A publish operation with a URL could take a significant amount of time. Thus, when the server creates a URL publisher, it spawns a new thread to read the data from the URL and publish it into the ESP server. When the thread has completed, a message is sent over the connection to the sending client.

The parameters to create a URL publisher are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| request   | Specify `url-publisher`.
| id        | Specifies a unique ID within the scope of the connection. This ID is returned when the URL publisher has finished its work. |
| window    | Specifies the path of the Source window into which to publish the events. Use the format `project/contquery/window`. |
| url       | Specifies the URL that contains the event data. |

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>Specify the data format.</td>
</tr>
</tbody>
</table>
For example:

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

When the format can be derived from the URL extension, this parameter is not required.

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

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

---

**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.

The parameters to retrieve server statistics are as follows:

*Table 13  Parameters Required to Retrieve Server Statistics*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>Specify <code>stats</code>.</td>
</tr>
</tbody>
</table>

*Table 14  Optional Parameters to Retrieve Server Statistics*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minCpu</td>
<td>Specify the minimum window CPU percentage to report. The default value is 5.</td>
</tr>
<tr>
<td>interval</td>
<td>Specify the interval, in seconds, at which to deliver the statistics. The default value is 1.</td>
</tr>
<tr>
<td>format</td>
<td>Specify the delivery format. Valid values are:</td>
</tr>
<tr>
<td></td>
<td>- <code>xml</code> - statistics in XML format (default)</td>
</tr>
<tr>
<td></td>
<td>- <code>json</code> - events in JSON format</td>
</tr>
<tr>
<td>counts</td>
<td>Specify <code>true</code> or <code>false</code> to determine whether window counts are returned. The default value is <code>false</code>.</td>
</tr>
<tr>
<td>config</td>
<td>Specify <code>true</code> or <code>false</code> to determine whether server configuration information is returned. The default value is <code>false</code>.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>memory</td>
<td>Specify true or false to determine whether server memory usage information is returned. The default value is false.</td>
</tr>
</tbody>
</table>

For example:

```json
{
    "request": "stats",
    "minCpu": 5,
    "interval": 1,
    "format": "xml",
    "counts": true,
    "config": true,
    "memory": true
}
```

Returned server statistic information looks like the following examples.

**Example Code 8  Server Statistic Information in XML Format**

```xml
<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="0.000497167" interval="1005698" name="brokerAlertsAggr"/>
            <window count="10" cpu="0.000497167" interval="1005698" name="venueAlertsAggr"/>
            <window count="5" cpu="0.000497167" interval="1005698" name="violationCounts"/>
        </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**

```json
{

```
"stats":{
    "project":{
        "@name":"secondary",
        "@state":"started",
        "contquery":[
            {
                "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"
            }
        }
    ]
}}
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.

Use the following parameters to request server logs:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>Specify logs.</td>
</tr>
<tr>
<td>capture</td>
<td>Specify true or false to determine whether to turn log capture on or off. Because log capture uses server resources, turn this off unless logs are required.</td>
</tr>
</tbody>
</table>
Table 16  Optional Parameters to Retrieve Server Logs

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter</td>
<td>Specify a regular expression that is run against each log message to determine whether the message is returned.</td>
</tr>
<tr>
<td>format</td>
<td>Specify xml, json, or text. The default value is xml.</td>
</tr>
</tbody>
</table>

For example:

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

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.

Use the following parameters to retrieve model information

Table 17  Parameter Required to Retrieve Model Information

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>Specify model.</td>
</tr>
</tbody>
</table>

Table 18  Optional Parameters to Retrieve Model Information

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schema</td>
<td>Specify true or false to determine whether schema data is included. The default value is false.</td>
</tr>
<tr>
<td>counts</td>
<td>Specify true or false to determine whether window event counts are included. The default value is false.</td>
</tr>
<tr>
<td>index</td>
<td>Specify true or false to determine whether window index types are included. The default value is false.</td>
</tr>
<tr>
<td>perms</td>
<td>Specify true or false to determine whether permissions are included. The default value is false.</td>
</tr>
<tr>
<td>pubsub</td>
<td>Specify true or false to determine whether to include only publish/subscribe enabled windows. The default value is false.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>xml</td>
<td>Specify <code>true</code> or <code>false</code> to determine whether window XML is included. The default value is <code>false</code>.</td>
</tr>
<tr>
<td>format</td>
<td>Specify <code>xml</code> or <code>json</code>. The default value is <code>false</code>.</td>
</tr>
</tbody>
</table>

For example:

```json

{  
  "request":"model",  
  "schema":true,  
  "counts":true,  
  "index":true,  
  "perms":true,  
  "pubsub":true,  
  "xml":true,  
  "format":"json"
}
```