Overview

SAS Event Stream Processing supports an API that uses the WebSocket protocol to communicate with the ESP server. This 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 the WebSocket 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, the WebSocket API is faster and more efficient than the RESTful API.

Inside the ESP server, the WebSocket publish/subscribe objects use the standard publish/subscribe interfaces. For more information about the WebSocket protocol, see RFC-6455.

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 line 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 user:password portion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>must be BASE64-encoded.</td>
</tr>
</tbody>
</table>

### Creating and Using the WebSocket Subscriber

The 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 WebSocket subscriber, supply the full pathname (project, continuous query, and window) of the window that uses the WebSocket protocol:

```
ws://server:port/SASESP/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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>Specifies the mode of the subscriber. Valid values are <strong>updating</strong> and <strong>streaming</strong>. An <strong>updating</strong> 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. An <strong>streaming</strong> 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>pagesize</td>
<td>Specifies the number of events to keep in an event page for this subscriber.</td>
</tr>
<tr>
<td>separator</td>
<td>Specifies the separator to use between events when you use the <strong>properties</strong> format.</td>
</tr>
<tr>
<td>filter</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 largeTrades window: <code>ws://espsrv01:26000/SASESP/subscribers/p/cq/largeTrades?filter=in(symbol,'IBM','T')</code></td>
</tr>
</tbody>
</table>
### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| **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/SASESP/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. |
| **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:  
`dfesp_xml_client -url "ws://espsrv01:26000/SASESP/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. |
Send a properties message to set properties on the connection without returning any events.

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>first</td>
<td>first page of events from the window</td>
</tr>
<tr>
<td>last</td>
<td>last page of events from the window</td>
</tr>
<tr>
<td>next</td>
<td>next page of events (current page + 1) from the window</td>
</tr>
<tr>
<td>prev</td>
<td>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: <code>event_field:sort_direction</code>. The <code>sort_direction</code> value is optional and defaults to descending.</td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<td>Note: The sort property is meaningful only in updating subscribers. Streaming subscribers always send events in the order received.</td>
<td></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:

```
ws://espsrv01:26000/SASESP/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:
Suppose that you send the `<load page='next'/>` message. The subscriber responds:

```
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,87236,1280928184,5613,26.650000,1200,55000,101556,0,8093853,1
```

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

```
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,87236,1280951886,336,131.430000,1400,55333,1012334,0,8377283,1
I,N, 8451211,IBM,87236,1280951949,7453,131.425000,1655,55111,1012445,0,8483102,1
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 WebSocket Publisher**

The 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 WebSocket publisher, supply the full pathname (project, continuous query, and window) of the window using the WebSocket protocol:

```
ws://server:port/SASESP/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>
</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.  
   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 (blocksize=1). |
| **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 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/SASESP/projectStats/project

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

ws://server:port/SASESP/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):

```xml
<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/SASESP/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/SASESP/logs"`