SAS® Event Stream Processing 5.2: JavaScript API

Overview

The SAS Event Stream Processing JavaScript (ESPJS) API consists of a set of JavaScript objects and methods that communicate with running ESP servers. Using the ESPJS API, you can create models and perform publish and subscribe operations in SAS Event Stream Processing from within web pages and across other platforms that support JavaScript.

The ESPJS API enables you to do the following:

- Read and analyze SAS Event Stream Processing models. The ESPJS API provides access to information on projects, continuous queries, and windows.
- Subscribe to windows to receive events. Events are delivered to a delegate object that you provide when you subscribe.
- Publish events into a Source window.
- Access ESP server logs.

Because most of the capabilities provided by the ESPJS API are asynchronous in nature, the API uses delegates. A *delegate* is a JavaScript object that implements certain functions that are invoked by SAS Event Stream Processing when an event occurs. The `ServerDelegate`, for example, has the following methods:

`connected(`*`server`*`)`
: Invoked when you successfully connect to an ESP server with the ESPJS API.

`disconnected(`*`server`*`)`
: Invoked when you lose connection to an ESP server.

Note: Of the methods that are supported for a delegate, only the methods of interest need to be implemented. If an event occurs that affects an ESPJS object and that object's delegate does not support the appropriate method, the event is ignored.

Getting Started

To use ESPJS, the JavaScript API files that are shipped with SAS Event Stream Processing must be accessible to the web page, application, or server that uses the ESPJS API.

Consider, for example, that you are developing a web application named *myapp* on an Apache Tomcat server. To ensure that the application can access ESPJS, you need to unpack the ESPJS TAR file that is located in the SAS Event Stream Processing installation directory to the Tomcat directory where that application is developed. With the Tomcat installation directory defined as $TOMCAT_HOME in a UNIX session, use the following commands:

```
$ cd $TOMCAT_HOME/webapps
$ mkdir myapp
$ cd myapp
$ tar xf $DFESP_HOME/tools/espjsapi.tar
```

After the files have been moved to the **myapp** directory, you need to add the following line to the `<head>` element of the web page with the application:

```
<script data-main="esp/js/libs/common/espapiMain" src="esp/js/libs/common/require.js"></script>
```

This line invokes the function `setupEsp(espjs)`. The `setupEsp(espjs)` function provides a handle to ESPJS:

```
var   _espapi = null;  // this is our handle into ESPJS

function
setupEsp(espapi)
{
   _espapi = espapi;
}
```

A complete starting page that incorporates ESPJS takes the following form:

```
<html>

<head>
<title>My App</title>
<script data-main="esp/js/libs/common/espapiMain" src="esp/js/libs/common/require.js"></script>

<script type="text/javascript">

var   _espapi = null;      // this is our handle into ESPJS

function
setupEsp(espapi)
{
   _espapi = espapi;
}

</script>
</head>

<body>
This is my page.
</body>

</html>
```

ESPJS Objects

Model Objects

The components of SAS Event Stream Processing models are represented by JSON objects without type definition in ESPJS.

The Project Object

The project object represents a project in SAS Event Stream Processing.

Table 1 *Project Object Data Members*

Member	Description
`name`	The name of the project.
`key`	The key of the project within its server. This value is the same as the `name` member or the `$projectName`.
`xml`	The XML definition for the project.
`hasReadPermission`	If the user has Read permission on the project, this value is true. Otherwise, it is false. This value is only relevant when the server is running with SASLogon Auth and with user access permissions enabled.
`hasWritePermission`	If the user has Write permission on the project, this value is true. Otherwise, it is false. This value is only relevant when the server is running with SASLogon Auth and with user access permissions enabled.
`contqueries`	An array of the continuous queries that are contained within the project.

The Continuous Query Object

The continuous query object represents a continuous query.

Table 2 *Continuous Query Object Data Members*

Member	Description
`name`	The name of the continuous query.
`key`	The key of the continuous query within its server. This value takes the form `$projectName/$continuousQueryName`.
`hasReadPermission`	If the user has Read permission on the continuous query, this value is true. Otherwise, it is false. This value is only relevant when the server is running with SASLogon Auth and with user access permissions enabled.

Member	Description
hasWritePermission	If the user has Write permission on the continuous query, this value is true. Otherwise, it is false. This value is only relevant when the server is running with SASLogon Auth and with user access permissions enabled.
windows	An array of the windows that are contained within the continuous query.
edges	An array of the edges that are contained within the continuous query.

The Window Object

The window object represents a window.

Member	Description
name	The name of the window.
type	The type of window. Supported window types include: ■ source ■ filter ■ aggregate ■ compute ■ union ■ join ■ copy ■ functional ■ notification ■ pattern ■ counter ■ geofence ■ procedural ■ model-supervisor ■ model-reader ■ train ■ calculate ■ score ■ text-context ■ text-category ■ text-sentiment ■ text-topic

Member	Description
`index`	The window's index type. Supported indexes include: ■ `pi_HASH` ■ `pi_RBTREE` ■ `pi_LN_HASH` ■ `pi_CL_HASH` ■ `pi_FW_HASH` ■ `pi_EMPTY` ■ `pi_HLEVELDB`
`key`	The key of the window within its server. The key takes the following form: `$projectName/$continuousQueryName/$windowName`.

Member	Description

`class`	The class of the window.
	Window classes include:
	`input`
	▪ `source`
	`transformation`
	▪ `filter`
	▪ `aggregate`
	▪ `compute`
	▪ `union`
	▪ `join`
	▪ `copy`
	▪ `functional`
	`utility`
	▪ `notification`
	▪ `pattern`
	▪ `counter`
	▪ `geofence`
	▪ `procedural`
	`analytics`
	▪ `model-supervisor`
	▪ `model-reader`
	▪ `train`
	▪ `calculate`
	▪ `score`
	`textanalytics`
	▪ `text-context`
	▪ `text-category`
	▪ `text-sentiment`
	▪ `text-topic`

`fields`	An array of fields that comprise the schema for the window.
`incoming`	An array of windows that send events to this window.
`outgoing`	An array of windows where this window sends events.

The Field Object

The field object represents a field in a window schema.

Member	Description
`name`	The name of the field.
`espType`	The ESP field type. ESP field types include: ■ `string` ■ `int32` ■ `int64` ■ `double` ■ `money` ■ `date`
`type`	The general field type. General field types include: ■ `string` ■ `int` ■ `float`
`isKey`	A Boolean value indicating if the field is a key.

The Edge Object

The edge object represents an edge between two windows in a model.

Member	Description
`input-window`	The window name of the input window.
`target-window`	The window name of the target window.

API Objects

General API JSON objects establish connection with running ESP servers and support publish and subscribe operations.

The ESPJS Object

The methods of the ESPJS object include:

- `Server`

  `getServer(`*`host`*`,`*`port`*`,[`*`secure`*`],[`*`delegate`*`])`

Description	This method creates a connection to an ESP server with the supplied parameters and returns a server object. The server object can be used to interact with an ESP server.

Delegate Functions	You can optionally specify a delegate object to be notified of significant events that occur within the server object. The delegate can support the following functions: ■ `connected(server)` ■ `disconnected(server)` ■ `error(server)`
Parameters	■ *host* The host on which the target ESP server is running. ■ *port* The HTTP port of the target ESP server. ■ *secure* If the ESP server is running under the HTTPS protocol, this is true. Otherwise, it is false. The default is false. ■ *delegate* The delegate object that receives operational information from the server object.
Examples	`var server = espjs.getServer("myserver",33000,true,{"connected":conn,"disconnected":disc});`

■ `Server`

`getServerFromUrl(url,[delegate])`

Description	This method creates a connection to an ESP server from the elements of the supplied URL and returns a server object. The server object can be used to interact with an ESP server.
Delegate Functions	You can optionally specify a delegate object to be notified of significant events that occur within the server object. The delegate can support the following functions: ■ `connected(server)` ■ `disconnected(server)` ■ `error(server)`
Parameters	■ *url* A URL that contains the protocol, host, and port information of the target ESP server. ■ *delegate* The delegate object that receives operational information from the server object.
Examples	`var server = espjs.getServerFromUrl("https://myserver:33000",{"connected":conn,"disconnected":disc});`

The Server Object

The server object represents the connection to the ESP server.

Server object methods include:

■ `void`

`connect()`

Description	This method sets up a persistent WebSocket connection to an ESP server. Because the connection initiation is asynchronous, the state of the connection is delivered to the server delegate object. If the connection is successfully established, then the `connected(`*`server`*`)` function is called. Otherwise, the `error(`*`server`*`)` function is called.

- `void`

  `disconnect()`

Description	This method shuts down an established connection to an ESP server.

- `void`

  `reconnect([`*`interval`*`])`

Description	This method initiates a loop that attempts to connect to the ESP server. This can be used from within the server delegate to identify a lost connection to the ESP server and to attempt reconnection.
Parameters	■ *`interval`* The interval, in seconds, between connection attempts. The default is 1 second.
Examples	In the following example, a server object uses a delegate to monitor the connection and attempt to reconnect if something goes wrong:

```
var  myserver = espjs.getServer("http://myserver:29000",
                                {"connected":conn,"disconnected":disc,"error":error});

function
conn(server)
{
  console.log("server " + server + " is connected");
  // server is connected, add stuff here
}

function
disc(server)
{
  console.log("server " + server + " is disconnected, commencing reconnect...");
  server.reconnect();
}

function
error(server)
{
  console.log("server " + server + " error, commencing reconnect...");
  server.reconnect();
}
```

- `void`

  `getProjects()`

Description	This function returns all projects in the server as a list of Project data objects. The model must be explicitly loaded for these objects to be available.

- `void`

  `getContqueries()`

Description	This function returns all continuous queries in the server as a list of Continuous Query data objects. The model must be explicitly loaded for these objects to be available.

- `void`

  `getWindows()`

Description	This function returns all windows in the server as a list of Window data objects. The model must be explicitly loaded for these objects to be available.

- `Object`

  `getProject(`*`key`*`)`

Description	This function returns a Project data object if one exists for the specified key. The model must be explicitly loaded for these objects to be available.
Parameters	■ *`key`* The project name.

- `Object`

  `getContquery(`*`key`*`)`

Description	This function returns a Continuous Query data object if one exists for the specified key. The model must be explicitly loaded for these objects to be available.
Parameters	■ *`key`* The project and the continuous query, separated by a forward slash (/).

- `Object`

  `getWindow(`*`key`*`)`

Description	This function returns a Window data object if one exists for the specified key. The model must be explicitly loaded for these objects to be available.
Parameters	■ *`key`* The project, the continuous query, and the window, separated by a forward slash (/).

- `void`

  `setID(`*`id`*`)`

Description	This function sets the ID of the server.

Parameters	■ *id* The ID of the server.

■ `String`

`getID()`

Description	This function returns the ID of the server as a string.

■ `void`

`setName(`*name*`)`

Description	This function sets the name of the server.
Parameters	■ *name* The name of the server.

■ `String`

`getName()`

Description	This function returns the name of the server as a string.

■ `boolean`

`isSecure()`

Description	If the server is running securely under the HTTPS protocol, this function returns a value of true. Otherwise, it returns a value of false.

■ `boolean`

`isConnected()`

Description	If the server is connected to an ESP server, this function returns a value of true. Otherwise, it returns a value of false.

■ `void`

`loadModel(`*delegate*`,[`*context*`])`

`reloadModel(`*delegate*`,[`*context*`])`

Description	These methods load a SAS Event Stream Processing model from an ESP server. The methods deliver a model from the ESP server to a delegate object. Because this might require sending a request to the server, it cannot be done in a synchronous manner. If a model has already been loaded, `loadModel` sends the model directly to the `loaded(`*`server`*`)` function of the delegate object. The `reloadModel` method always sends a request to the server to retrieve a model.
Delegate Functions	The delegate supports the following functions: ■ `loaded(`*`server`*`)` Invoked with the model if it is successfully retrieved from the server. ■ `error(`*`server`*`)` Invoked if there is a problem interacting with the server.
Parameters	■ *`delegate`* The delegate object that receives the model. ■ *`context`* The context data to attach to the request.

Examples	`var myserver = espjs.getServer("http://espsrv01:29000");`
	
	`myserver.loadModel({"loaded":modelLoaded});`
	
	`function`
	`modelLoaded(server)`
	`{`
	`  for (var i = 0; i < server.projects.length; i++)`
	`  {`
	`  console.log(server.projects[i].name);`
	`  }`
	`}`

■ `void`

`loadProject(`*`name`*`,`*`definition`*`,[`*`options`*`],[`*`delegate`*`])`

`loadProjectUrl(`*`name`*`,`*`url`*`,[`*`options`*`],[`*`delegate`*`])`

Description	These methods load a SAS Event Steam Processing project. The `loadProject` method loads a project from an XML definition. The `loadProjectUrl` method loads a project from a URL. The results of the operation are delivered to the specified delegate object.
Delegate Functions	The delegate supports the following functions: ■ `loaded(`*`name`*`,`*`server`*`)` Invoked when the project successfully loads. ■ `error(`*`name`*`,`*`server`*`,`*`text`*`)` Invoked if there is a problem loading the project. You can specify loading options in the *`options`* parameter.

Parameters

- *name*

  The name of the project to load.

- *definition*

  The XML project definition contained in a string.

- *url*

  The URL from which the server retrieves the XML project definition.

- *delegate*

  The delegate object that is notified of the status of the project load.

- *options*

  The project load options. These are all optional and include the following:

  - *overwrite*

    This can be either true or false depending on whether you want to overwrite the project if it exists. The default is false.

  - *connectors*

    This can be either true or false depending on whether you want to start the connectors upon project startup. The default is true.

  - *start*

    This can be either true or false depending on whether you want to start the project upon loading. The default is true.

Examples

```
var myserver = espjs.getServer("http://myserver:29000");

myserver.loadProjectUrl("myproject","http://myserver:18080/models/model.xml"{"loaded":projectLoaded});

function
projectLoaded(name,server)
{
  alert("the project " + name + " loaded successfully");
}
```

- `void`

  `unloadProject(`*name*`,[`*delegate*`])`

Description

This method unloads the project `name` from the ESP server. The results of the operation are delivered to the specified delegate object.

Delegate Functions

The delegate supports the following functions:

- `unloaded(`*name*`,`*server*`)`

  Invoked when the project successfully unloads.

- `error(`*name*`,`*server*`,`*text*`)`

  Invoked if there is a problem unloading the project.

Parameters	■ *name* The name of the project to unload. ■ *delegate* The delegate object that is notified of the status of the project unload.

■ `Publisher`

`getPublisher(`*p*`,`*cq*`,`*w*`,[`*delegate*`])`

Description	This method creates an event publisher for the specified project, continuous query, and Source window.
Delegate Functions	You can optionally specify a delegate object if you want to be notified of the status of the publisher. The delegate supports the following functions: ■ `open(`*publisher*`)` Invoked when the publisher is open and ready to publish. ■ `close(`*publisher*`)` Invoked when the publisher loses its connection to the server. ■ `error(`*publisher*`)` Invoked when the publisher encounters an error.
Parameters	■ *p* The project into which you want to publish. ■ *cq* The continuous query into which you want to publish. ■ *w* The window into which you want to publish. ■ *delegate* An optional delegate object to receive status notifications about the publisher.

■ `Subscriber`

`getStreamingSubscriber(`*p*`,`*cq*`,`*w*`,[`*delegate*`])`

`getUpdatingSubscriber(`*p*`,`*cq*`,`*w*`,[`*delegate*`])`

Description	These methods create an event subscriber for the specified project, continuous query, and window. The `getStreamingSubscriber` method creates an updating subscriber that delivers events based on the key values of those events. The `getStreamingSubscriber` method creates a streaming subscriber that delivers raw events.

Delegate Functions	You can optionally specify a delegate object if you want to be notified of the status of the subscriber. The delegate supports the following functions: ■ `events(subscriber,data)` Invoked when the subscriber receives events that have just occurred. ■ `page(this,data,page,pages)` Invoked when the subscriber receives a page of events. ■ `pages(this,page,pages)` Invoked when the subscriber receives updated ESP window event count information. ■ `open(subscriber)` Invoked when the subscriber is open and receiving events. ■ `close(subscriber)` Invoked when the subscriber loses its connection to the server. ■ `error(subscriber)` Invoked when the subscriber encounters an error.
Parameters	■ *p* The project to which you want to subscribe. ■ *cq* The continuous query to which you want to subscribe. ■ *w* The window into which you want to subscribe. ■ *delegate* An optional delegate object to receive status notifications about the subscriber.

■ `ProjectStats`

`getStatsSubscriber(cpu,interval,[counts],[delegate])`

Description	This method creates a ProjectStats object that receives window resource information, including CPU usage and event counts.
Delegate Functions	You can optionally specify a delegate object if you want to be notified of the status of the subscriber. The delegate supports the following functions: ■ `open(subscriber)` Invoked when the subscriber is open and receiving events. ■ `close(subscriber)` Invoked when the subscriber loses its connection to the server. ■ `error(subscriber)` Invoked when the subscriber encounters an error.

Parameters	■ *cpu* The minimum CPU percentage to report. ■ *interval* The interval, in seconds, at which to receive updates. ■ *counts* To receive window event counts, set to true. ■ *delegate* An optional delegate object to receive status notifications about the subscriber.

■ `Logs`

`getLogs(`*delegate*`)`

Description	This method creates a logs object that can receive log entries from an ESP server.
Delegate Functions	You must supply a delegate that supports the following functions. `handle` is required to receive data. ■ `handle(`*subscriber*`,`*data*`)` Invoked when the object receives log data from the server. ■ `open(`*subscriber*`)` Invoked when the subscriber is open and receiving events. ■ `close(`*subscriber*`)` Invoked when the subscriber loses its connection to the server. ■ `error(`*subscriber*`)` Invoked when the subscriber encounters an error.
Parameters	■ *delegate* A delegate object to receive log data and status notifications about the subscriber.

■ `void`

`setLogCapture(`*value*`,[`*size*`])`

Description	This method sets the state and size of the ESP server log capture. The `value` parameter is a Boolean value that determines whether the log capture is on or off, and the optional size parameter can be used to set the number of log messages stored in the log capture cache in the server.
Parameters	■ *value* The state of ESP server log capture, which is set to true for on and false for off. ■ *size* The size of ESP server log cache.

■ `void`

`getGuids(delegate,[num],[context])`

Description	This method retrieves any number of generic unique IDs (GUIDs) from the ESP server and delivers them to the specified delegate.
Delegate Functions	■ `handle(server,guids,context)` Invoked when the GUIDs have been successfully retrieved from the server. The GUIDs are delivered in an array of strings in the *guids* parameter. ■ `error(server)` Invoked when the server object senses a connection error to the ESP server.
Parameters	■ *delegate* The delegate that receives the GUIDs. ■ *num* The number of GUIDs to retrieve. The default is 1. ■ *context* The context data to attach to the request. The data is delivered to the delegate along with the GUIDs.

Examples	`...`
	`server.getGuids({"handle":showGuids},10);`
	`...`
	`function showGuids(server,guids) { for (var i = 0; i < guids.length; i++) { console.log("GUID: " + guids[i]); } }`

The Publisher Object

The publisher object is used to publish events to an ESP server. There are two ways to add events to the publisher for delivery to SAS Event Stream Processing using the publisher object:

- Add events to the publisher with the `begin`, `set`, and `end` methods. The `begin` method initializes an object to add to the publish queue. The `set` method sets the value for a specified field in the current object. The `end` method finalizes the current object and places it in the publish queue.

```
...

publisher.begin();
publisher.set("element",event.target.id);
publisher.set("x",event.clientX);
publisher.set("y",event.clientY);
publisher.end();

...
```

- Add JavaScript objects to the publisher that puts the objects directly into the publish queue.

```
...

publisher.add({"element":event.target.id,"x":event.clientX,"y":event.clientY});

...
```

When you are ready to publish the events, just call the `publish` method:

```
...

publisher.publish();

...
```

Calling the `publish` method sends all events in the current queue to SAS Event Stream Processing. After a `publish` call, the event queue is emptied.

Publisher object methods include:

- `void`
  `start()`

Description	This method initiates the publisher's connection to the ESP server.

- `void`
  `stop()`

Description	This method shuts down the connection to the ESP server.

- `void`
  `setName(`*`name`*`)`

Description	This method sets the name of the publisher.
Parameters	■ *`name`* The name of the publisher.

- `String`
  `getName()`

Description	This method returns the name of the publisher as a string.

- `String`
  `getProject()`

Description	This method returns the name of the project for the publisher as a string.

- `String`

  `getContquery()`

Description	This method returns the name of the continuous query for the publisher as a string.

- `String`

  `getWindow()`

Description	This method returns the name of the window for the publisher as a string.

- `void`

  `begin()`

Description	This method initializes an object before setting field values.

- `void`

  `set(`*`name`*`,`*`value`*`)`

Description	This method sets a value for the current object. The name of the value is assumed to be a valid field name for the intended Source window.
Parameters	■ *`name`* The name of a field that maps to a schema field name. ■ *`value`* The value of the field.

- `void`

  `end()`

Description	This method terminates value setting and commits the current object to the publish queue. This does not publish the object.

- `void`

  `add(`*`o`*`)`

Description	This method adds an object to the publish queue. The names of the object fields should map to schema field names of the target Source window.
Parameters	■ *`o`* The object to add to the publish queue.

- `void`

`publish()`

Description	This method sends all objects currently in the publish queue to the ESP server. The publish queue is then cleared.

The Subscriber Object

The subscriber object retrieves events from the ESP server. Each subscriber consists of two components:

- A server-side component that resides in the ESP server and uses the native ESP publish/subscribe engine.
- A client-side component that communicates directly with its server-side counterpart to send and receive data.

There are two types of subscribers:

- Updating subscribers

  An *updating subscriber* uses event pages to retrieve and report streaming data. An *event page* is a set of events of a specific size that comprises the current view of the subscriber. An example is an HTML graphical application with a bar chart of events that displays the current event page at any point in time. The server-side component maintains the keys of these events. If the server-side component receives an event notification from the ESP server and the event is not in the current page, it is ignored. If it is in the current page, the client-side component is notified of the update.

- Streaming subscribers

  The server-side component of a *streaming subscriber* processes and sends an event for each event notification that it receives from the ESP server. The events are put into a list as they are received and sent to the client-side component. The list is trimmed to the current event page size of the subscriber before being sent. When a large number of events stream through SAS Event Stream Processing at a high rate, some events are dropped, and the client-side component picks up a sample of the total set of events streaming through the system. If the client-side component resides in a relatively slow application, such as a web page that renders charts, increasing the delivery interval of high-throughput windows can improve performance. You can change the delivery interval, in milliseconds, with the `setInterval(`*`interval`*`)` method. This method directs the server-side component to only deliver events at each interval.

A subscriber can receive events from the ESP server publish/subscribe engine as they occur or by explicitly requesting event data from the server.

Note: Only updating subscribers can request event page data.

The delegate object that you provide when creating the subscriber is the mechanism you use to receive these events:

- `server.getUpdatingSubscriber(`*`p`*`,`*`cq`*`,`*`w`*`,`*`delegate`*`)`
- `server.getStreamingSubscriber(`*`p`*`,`*`cq`*`,`*`w`*`,`*`delegate`*`)`

Define the following functions in the delegate:

- `events(`*`subscriber`*`,`*`data`*`)`

  Invoked when the subscriber receives events that have just occurred. The event data is contained in the *`data`* parameter as an array of JavaScript objects.

- `page(`*`subscriber`*`,`*`data`*`,`*`page`*`,`*`pages`*`)`

  Invoked when the subscriber receives a page of events. The event data is contained in the *`data`* parameter as an array of JavaScript objects. The *`page`* and *`pages`* parameters indicate the page number and the total number of pages, respectively.

- `pages(`*`subscriber`*`,`*`page`*`,`*`pages`*`)`

Invoked when the subscriber receives updated window event count information. You receive notification only if you have set the *info* option on the subscriber to an integer value greater than 0. For example, `subscriber.setOption("info",5);` sends window event count information to the subscriber every 5 seconds. This is useful if you want to track the total number of pages in the window.

- `open(`*`subscriber`*`)`

  Invoked when the subscriber is open and receiving events.

- `close(`*`subscriber`*`)`

  Invoked when the subscriber loses its connection to the server.

- `error(`*`subscriber`*`)`

  Invoked when the subscriber encounters an error.

Subscriber object methods include:

- `void`
  `start()`

Description	This method initiates the subscriber's connection to the ESP server.

- `void`
  `stop()`

Description	This method shuts down the connection to the ESP server.

- `void`
  `setName(`*`name`*`)`

Description	This method sets the name of the subscriber.
Parameters	■ *`name`* The name of the subscriber.

- `String`
  `getName()`

Description	This method returns the name of the subscriber as a string.

- `String`
  `getProject()`

Description	This method returns the name of the project for the subscriber as a string.

- `String`
  `getContquery()`

Description	This method returns the name of the continuous query for the subscriber as a string.

- `String`

  `getWindow()`

Description	This method returns the name of the window for the subscriber as a string.

- `void`

  `setFilter(`*`filter`*`)`

Description	This method sets a functional filter on the subscriber. The filter follows the functional window syntax that allows you to use event fields in functions. Here is an example:
	`contains($brokerName,'Larry','Moe')`
	`gt($price,1000)`
	Filters run on the server. Events that do not match the criteria specified in the filter are not delivered to the client.

Parameters	■ *`name`* The name of a field that maps to a schema field name. ■ *`value`* The value of the field.

- `String`

  `getFilter()`

Description	This method returns the current filter as a string.

- `void`

  `clearFilter()`

Description	This method clears the current filter.

- `void`

  `setSort(`*`field`*`,[`*`direction`*`])`

Description	This method sets the sort properties for a subscriber. The *`field`* parameter is the event field by which the events are sorted. The *`direction`* parameter can be either `ascending` or `descending`. The default is `descending`. **Note:** Using a sort field can significantly affect performance because events must be sorted as they occur. Because each event that occurs can change the sort order of the current page, setting a sort field on a subscriber forces the subscriber to always use a `page(`*`subscriber`*`,`*`data`*`,`*`page`*`,`*`pages`*`)` delegate to receive events.

Parameters	■ *field* The sort field. ■ *direction* The sort direction, either `ascending` or `descending`.

■ `void`

`clearSort()`

Description	This method clears the sort properties.

■ `void`

`setPageSize(`*size*`)`

Description	This method sets the event page size. This size is the maximum number of events that are stored in a page on the server. An updating subscriber always has a current page containing a set of keys of interest. Whenever an event affects an event in the current page, the client is notified.
Parameters	■ *size* The page size.

■ `int`

`getPage()`

Description	This method returns the current page number as an integer.

■ `int`

`getPages()`

Description	This method returns the number of available pages in the subscriber as an integer.

■ `void`

`load()`

Description	This method loads the current page. It sends a message to the server to return all events in the current page.

■ `void`

`first()`

Description	This method loads the first page. It sends a message to the server to return all events in the first page.

- `void`

  `last()`

Description	This method loads the last page. It sends a message to the server to return all events in the last page.

- `void`

  `next()`

Description	This method loads the next page. It sends a message to the server to return all events in the next page.

- `void`

  `prev()`

Description	This method loads the previous page. It sends a message to the server to return all events in the previous page.

- `void`

  `updatePages()`

Description	This method sends a message to the server to return updated information on the number of pages in the window. The response is sent to the `pages` function in the delegate if it exists.

- `void`

  `play()`

Description	This method sends a message to the server to start delivering events to the client side. When a subscriber is started, it is put in play mode. The subscriber must be explicitly paused to ignore events in the server-side component. Calling this method causes the server-side component to send the current page of events to the client.

- `void`

  `pause()`

Description	This method sends a message to the server to stop delivering events to the client side. Any events that are received from SAS Event Stream Processing on the server side while the subscriber is paused are ignored.

- `boolean`

  `isStreaming()`

Description	If the subscriber is a streaming subscriber, a value of true is returned. Otherwise, a value of false is returned.

- `boolean`

  `isUpdating()`

Description	If the subscriber is an updating subscriber, a value of true is returned. Otherwise, a value of false is returned.

The ProjectStats Object

The projectstats object retrieves processing information on windows. This information includes the CPU usage of each window along with the event counts.

You create a projectstats object with the `getStatsSubscriber` method:

```
ProjectStats
server.getStatsSubscriber(cpu,interval,counts,delegate)
```

The *`cpu`* parameter specifies the minimum CPU usage to report. If *`cpu`* is set to 5, for example, the projectstats object only retrieves information on windows that are using at least 5% of the CPU when the window data is collected. The *`interval`* parameter specifies the interval, in seconds, at which the information is sent from the server to the client. If the *`counts`* parameter is set to true, the window event count is also included in the delivered information.

The delegate supports the following functions:

- `handle(`*`stats`*`,`*`data`*`)`

  Invoked when data is delivered.

  The *`data`* parameter is an array of JavaScript objects containing the following fields:

  - project
  - contquery
  - window
  - cpu
  - interval
  - count

- `open(`*`stats`*`)`

  Invoked when the projectstats object is open and receiving data.

- `close(`*`stats`*`)`

  Invoked when the projectstats object loses its connection to the server.

- `error(`*`stats`*`)`

  Invoked when the projectstats object encounters an error.

Projectstats object methods include:

- `void`

  `setParms(`*`cpu`*`,`*`interval`*`,`*`counts`*`)`

Description	This method sets the parameters for the instance. This restarts the projectstats instance.

Parameters	■ *cpu* The minimum CPU percentage to report. ■ *interval* The interval, in seconds, at which to receive updates. ■ *counts* If you want to receive window event counts, set this value to true.

■ `void`
`start()`

Description	This method initiates a connection to the ESP server to start the projectstats collector.

■ `void`
`stop()`

Description	This method shuts down the connection to the ESP server to stop the projectstats collector.

The Logs Object

The logs object allows you to view ESP server logs.

You create a logs object with the `getLogs` method:

```
Logs
server.getLogs(delegate)
```

The delegate supports the following functions:

■ `handle(logs,data)`
Invoked when data is delivered. The *data* parameter contains an ESP server log entry.

■ `open(logs)`
Invoked when the logs object is open and receiving data.

■ `close(logs)`
Invoked when the logs object loses its connection to the server.

■ `error(logs)`
Invoked when the logs object encounters an error.

Logs object methods include:

■ `void`
`start()`

Description	This method initiates a connection to the ESP server to start the log collector.

■ `void`

`stop()`

Description	This method shuts down the connection to the ESP server to stop the log collector.

Authentication

To connect to an ESP server that requires user authentication, the server delegate must have valid authentication information. You can write a simple function that authenticates the server delegate with the ESP server. Specify this function in the server delegate's `authenticate` field:

```
var server = _espapi.getServerFromUrl(url,{"connected":conn,"disconnected":disc,"error":error,
                                           "authenticate":myauth});
```

The function *myauth* retrieves authorization data and calls the `setAuthorization(auth)` function on the server. The function can then send a request to authenticate with the server using the authentication data that it retrieved.

myauth must have the following signature:

```
function myauth(server,scheme,request)
```

If the ESP server requires authentication, it responds to all unauthorized requests with a 401 "Unauthorized" status code. If the ESP server responds to an unauthorized ESPJS request with a 401 status code, the *myauth* function is called with the server requested as the *server*, the authentication scheme sent back from the server as the *scheme*, and the request that was denied by the server as the *request*.

You can define the *myauth* function in several ways. For example, if you are running against an ESP server using OAUTH2 and you have a token, your function might take the following form:

```
function
myauth(server,scheme,request)
{
   if (scheme == "bearer")
   {
      var   token = "eyJhbGciOiJIUzI1NiIsImtpZCI6ImxlZ2Fj...";
      server.setAuthorization("bearer " + token);
      request.send();
   }
}
```

If you are prompting the user for credentials, you need to save the server and the request to set the authorization information when the user enters their credentials.

Example Application

Consider an example page web application, hosted on an Apache Tomcat server, that does the following:

- Initializes ESPJS
- Creates a connection to a running ESP server
- Loads a simple, single window project that can capture click data from the page
- Upon notification of a successful project load, creates a publisher and a subscriber to the clicks window

- Sets up a logging `div` to show information received from the ESP server and three colored `div` to receive click events

When a user clicks anywhere inside the web page, an event is published to SAS Event Stream Processing. A subscriber receives events from the same window and sends the event objects to the logging `div` in JSON format:

```
{"_opcode":"insert","_timestamp":"1519907814537502","element":"red","id":"e0a5135b-3868-41d7-a476-8b0f6c94033c",
 "x":"332","y":"419"}
```

To run the application:

1. Follow the instructions in "Getting Started" to create a starting page for *myapp* that can access ESPJS.
2. Edit the `body` of the index.html file that is located in **`$TOMCAT_HOME/webapps/espjs`** to read as follows:

```
<body id="theBody">
   <table style="width:100%">
      <tr>
         <td colspan="3">ESPJS Output</td>
      </tr>
      <tr>
         <td colspan="3">
            <div id="output" style="width:100%;height:300px;border:1px solid #c0c0c0;
                              font-family:courier new;font-size:12pt;overflow:auto"></div>
         </td>
      </tr>
      <tr>
         <td>
            <div id="red" style="width:100%;height:100px;border:1px solid #c0c0c0;background:red"></div>
         </td>
         <td>
            <div id="green" style="width:100%;height:100px;border:1px solid #c0c0c0;background:green"></div>
         </td>
         <td>
            <div id="blue" style="width:100%;height:100px;border:1px solid #c0c0c0;background:blue"></div>
         </td>
      </tr>
   </table>
</body>
```

3. In a browser, navigate to the web page to make sure that you can access it:

   `http://myserver:yourport/myapp`

4. Add an ESPJS initialization function to do the following:
   - Save the ESPJS handle for future use.
   - Create a server object to communicate with an ESP server.
   - Use the server object to load a project to an ESP server.

   In addition to the initialization function, add a logging function to show ESPJS output and information messages.

   The following code defines these functions in the `script` of the index.html file:

```
<script type="text/javascript">

var   _espapi = null;      // this is our handle into ESPJS
var   _server = null;      // this is our server
```

```
function
setupEsp(espapi)        //a
{
   _espapi = espapi;

   log("ESPJS initialized");

   var   espUrl = "http://myserver:port";

   _server = _espapi.getServerFromUrl(espUrl);

   var   model = "<project threads='4' pubsub='auto'>\
               <contqueries>\
                 <contquery name='cq' trace='clicks'>\
                   <windows>\
                     <window-source name='clicks' insert-only='true' autogen-key='true'>\
                       <schema>\
                         <fields>\
                           <field name='id' type='string' key='true'/>\
                           <field name='element' type='string'/>\
                           <field name='x' type='int32'/>\
                           <field name='y' type='int32'/>\
                         </fields>\
                       </schema>\
                     </window-source>\
                   </windows>\
                 </contquery>\
               </contqueries>\
             </project>";

   _server.loadProject("myapp",model,{"loaded":projectLoaded,"error":projectError},
                       {"overwrite":true});                              //b
}

function
log(text)        //c
{
   var   output = document.getElementById("output");
   var   s = output.innerHTML;
   if (s.length > 0)
   {
      s += "<br/>";
   }
   s += text;
   output.innerHTML = s;
   output.scrollTop = output.scrollHeight;
}

</script>
```

a Define the initialization function `setupEsp`.

b Call the `loadProject` method using a delegate to receive notification when the project successfully loads.

c Define the logging function `log`.

5 Add the following code block to the `script` of the index.html file:

```
var   _publisher = null;     //a
var   _subscriber = null;     //b

function
projectLoaded(name,server)
{
   log("project " + name + " loaded by server " + server);

   _publisher = _server.getPublisher("myapp","cq","clicks");
   _publisher.start();

   _subscriber = _server.getStreamingSubscriber("myapp","cq","clicks",{"events":handler});
   _subscriber.start();

   document.body.addEventListener("click",sendEvent);     //c
}

function
sendEvent(event)     //d
{
   _publisher.begin();
   _publisher.set("element",event.target.id);
   _publisher.set("x",event.clientX);
   _publisher.set("y",event.clientY);
   _publisher.end();
   _publisher.publish();
}

function
handler(subscriber,data)
{
   for (var i = 0; i < data.length; i++)
   {
      log(JSON.stringify(data[i]));
   }
}

function
projectError(server,text)
{
   log("error: " + server + " : " + text);
}
```

a Create a publisher object to send events into the *myapp/cq/clicks* window.

b Create a subscriber object to subscribe to the *myapp/cq/clicks* window. These events are delivered to the `handler(`*`subscriber`*`,`*`data`*`)` function.

c Add an event listener to capture clicks on the page.

d Call the `sendEvent(`*`event`*`)` method to publish the click information to SAS Event Stream Processing.

This ESPJS enabled web page interacts with an ESP server to load a project and to publish and receive events.