SAS® Event Stream Processing 6.2: JavaScript API Reference

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.

Member	Description
`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.
`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.

Member	Description
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
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":projectLoade

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.

Visualization Objects That Use the Connect API (Preproduction)

Overview

Important: THIS PRELIMINARY DOCUMENTATION AND CODE ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. The Institute shall not be liable whatsoever for any damages arising out of the use of this documentation or code, including any direct, indirect, or consequential damages. The Institute reserves the right to alter or abandon use of this documentation at any time. In addition, the Institute will provide no support for the materials contained herein.

The Connect API enables you to communicate with ESP servers from any platform that uses JavaScript objects. These objects can be embedded in web pages. They also support Node.js, so they can be run from the command line.

The Connect API uses the ESP Server Connection API and its associated subscriber and publisher objects. It provides graphical objects that you can use to display event stream processing data.

Getting Started

1. Obtain the file `connect.tar` from `$DFESP_HOME/tools`.
2. Create a working directory, `@work@`.
3. Change directory to `@work@`.
4. Unpack the file:

   ```
   $ tar xf connect.tar
   ```

5. Start the HTTP server using port 33000:

   ```
   $ python -m http.server 33000
   ```

After the HTTP server is running, you can develop web pages. Depending on the API features that you want to use, you must add certain declarations to the head section of the page.

For maps, include the following declarations:

```
<link rel="stylesheet" href="/esp/style/leaflet.css" />
<script src="/esp/js/libs/leaflet/leaflet.js"></script>
```

For model viewer visualizations, include the following declarations:

```
<script src="/esp/js/libs/viz.js/viz.js"></script>
<script src="/esp/js/libs/viz.js/full.render.js"></script>
```

Include the following code to bring in the Connect API:

```
<script src="/esp/js/libs/plotly/plotly-latest.min.js"></script>
<script data-main="/esp/js/libs/main" src="/esp/js/libs/require.js"></script>
<link rel="stylesheet" href="/esp/style/espapi.css" />
```

You must supply the `esp` function, which creates connections to the ESP server and loads projects. For example, consider a web page that displays some colored areas and publishes events into an ESP model when a user clicks in those areas. The `esp` function would look something like the following. Be sure to enclose this code the `<script>` tag.

```
var _api = null;
var _conn = null;
var _visuals = null;

function
esp(api)
{
    _api = api;

    var parms = api.getParms();

    if (parms.hasOwnProperty("server") == false)
    {
        alert("you must specify ESP server URL");
        return;
    }

    _visuals = _api.createVisuals(parms);
    _conn = _api.connect(parms.server,{interval:0});

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

    _conn.loadProject("myproject",model,{loaded:ready},{overwrite:true});
}
```

This function creates a connection to an ESP server. The server must be supplied in the URL or the page displays an error. It also loads a simple model into the server so that it can publish events into it. Since this is an asynchronous operation, the code provides a delegate that implements the loaded function to be called when the project has been loaded into the server.

The function also creates a Visuals object to be used later in creating the visualizations.

After the project has successfully loaded, the API invokes the loaded delegate function when the server returns. If you are working with a server with a model already loaded, then you likely do not need to go through the load process.

Because the ready function was specified, the loaded function might look like this:

```
var _publisher = null;

function
ready()
{
    _publisher = _conn.getPublisher({window:"myproject/cq/clicks"});

    _visuals.createLogViewer("logs",_conn,{header:"Log Viewer"});
    _visuals.createModelViewer("model",_conn,{header:"Model Viewer",counts:true,schema:true});

    var clicks  = _conn.getEventCollection({window:"myproject/cq/clicksAggr"});

    _visuals.createBarChart("barchart",clicks,{y:"count",header:"Clicks Chart",
                            xrange:[0,100],orientation:"horizontal"});
    _visuals.createGauge("gauges",clicks,{value:"count",segments:5,header:"Clicks Indicators",
                         width:200,range:[0,100],bar_color:"rgba(255,255,255,.7)"});

    _visuals.size();
}
```

This function does the following tasks:

- creates a Publisher object to be used to publish click events into the ESP model
- creates a log viewer and a model viewer to monitor what is going on in the ESP server
- creates an EventCollection object to gather click events from the Aggregation window.
- creates a bar chart and a set of gauges to visualize the click events
- uses the Visuals size function to lay out the page

Each visualization requires an HTML page element (usually `<div>`) in which to render. The ID of that element is specified as the first parameter of the creation function.

For this page, the body looks like this:

```
<body onresize="_visuals.size()">

    <div id="banner">
        <table style="width:100%" cellspacing="0" cellpadding="0">
            <tr>
                <td id="bannerTitle">ESP Basic Example</td>
            </tr>
        </table>
    </div>

    <div id="content">
        <div class="container">
            <div class="component" style="width:100%">
```

```
                Click on the colored areas to generate events</div>
               <div id="red" class="component" style="width:32%;height:100px;background:red"
                onmousedown="javascript:publish(event)"></div>
               <div id="green" class="component" style="width:32%;height:100px;background:green"
                onmousedown="javascript:publish(event)"></div>
               <div id="blue" class="component" style="width:32%;height:100px;background:blue"
                onmousedown="javascript:publish(event)"></div>
               <div id="barchart" class="component" style="width:48%;height:300px"></div>
               <div id="gauges" class="component" style="width:48%;height:300px"></div>
               <div id="logs" class="component" style="width:48%;height:400px"></div>
               <div id="model" class="component" style="width:48%;height:400px"></div>
           </div>
       </div>

       <div id="footer"> </div>
   </body>
```

When you lay out a page like this one, the page automatically rearranges itself to best fit the graphics on the page. You can use the Visuals.size() function to size the header, content, and footer sections of the graphic.

Programming Objects

ServerConnection Object

Create a ServerConnection object to create the data sources that feed visualizations. The ServerConnection is a persistent connection to the ESP server. Create an instance of this object through the API instance passed into the `esp` function:

```
conn = api.connect("http://host:port");
```

The visualization API uses this connection to communicate with the ESP server. Also, the connection is used to monitor the health of the server. When the server goes down while the connection is active, the API waits for the server to return. When the server comes back, all data sources are reconnected automatically.

ServerConnection Methods

getEventCollection

Create and return an EventCollection object.

getEventCollection(*parameters*)

Table 3 *Parameters for getEventCollection*

Parameter	Description
`window`	Specify the path of the Source window in the form ***`project/contquery/window`***.
`pagesize`	Specify the page size for the collection. The default value is 50.

Parameter	Description
`filter`	Specify a functional filter for the collection.
`interval`	Specify the time, in milliseconds, for the server to wait before delivering any events that occurred. If not specified, the interval defaults to 1 second.

Example Code 1 *Examples*

```
brokerAlerts = conn.getEventCollection({window:"secondary/cq/brokerAlertsAggr"})
venueAlerts = conn.getEventCollection({window:"secondary/cq/venueAlertsAggr",
                                       filter:"in($city,'raleigh')})
```

getEventStream

Create and return an EventStream object. An event stream is similar to a UNIX tail of an event stream processing window.

getEventStream(*parameters*)

Table 4 *Parameters for getEventStream*

Parameter	Description
`window`	Specify the path to the window from which to obtain the event. Use the format */project/contquery/window*.
`maxevents`	Specify the maximum number of events to store in the collection. The default value is 100.
`interval`	Specify the time, in milliseconds, for the server to wait before delivering any events that occurred. If not specified, the interval defaults to 1 second.

Example Code 2 *Example*

```
rates = conn.getEventStream({window:"primary/cq/counter",maxevents:20})
```

getPublisher

Create and return a publisher object.

getPublisher(*parameter*)

Table 5 *Parameter for getPublisher*

Parameter	Description
`window`	Specify the path to the Source window. Use the format */project/contquery/window*.

Example Code 3 *Example*

```
publisher = conn.getPublisher({window:"primary/cq/rawTrades"})
```

getStats

Returns the Stats object for the connection

getStats(*parameters*)

Table 6 *Parameters for getStats*

Parameter	Description
`interval`	Specify the interval, in seconds, at which the server sends data to the client.
`mincpu`	Specify the minimum CPU usage, in percentage, which is reported (defaults to 5).
`cpu`	Specify **`true`** or **`false`** to determine whether to report window CPU usage (defaults to **`false`**).
`memory`	Specify **`true`** or **`false`** to determine whether to report memory usage information (defaults to **`true`**).
`counts`	Specify **`true`** or **`false`** to determine whether to report window event counts (defaults to **`false`**).
`config`	Specify **`true`** or **`false`** to determine whether to report server configuration information (defaults to **`false`**).

Example Code 4 *Example*

```
stats = conn.getStats({mincpu:5,cpu:true,memory:true,counts:true})
```

getLog

Returns the Log object for the connection

getLog()

EventCollection Object

An Event Collection is a view into a stateful window. When an Event Collection is created, it sets a page size. This size 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 the window. When an event is not in the current page, that event is ignored. Otherwise, the event is delivered to the client.

```
alerts = conn.getEventCollection({window:"secondary/cq/brokerAlertsAttr"})
```

The Event Collection manages itself and delivers change events to its delegates. To receive collection change notifications, you must register a delegate that implements the `dataChanged` method:

```
function
handle(collection,data,clear)
{
    console.log("data changed: " + JSON.stringify(data,null,"\t"));
}

alerts.addDelegate({dataChanged:handle});
```

The `dataChanged` method receives the collection that changed along with the new data in the `data` parameter.

EventCollection Methods

addDelegate

Add a delegate to receive collection change notifications.

addDelegate(*delegate*)

Table 7 *Required Parameter for addDelegate*

Parameter	Description
`delegate`	Specify the object to receive change notifications.

removeDelegate

Remove a delegate from the collection.

removeDelegate(*delegate*)

Table 8 *Parameter for removeDelegate*

Parameter	Description
`delegate`	Specify the object to remove from the collection.

getData

Return the collection of events. The data is a dictionary of objects that represent events sorted by event key.

getData()

setFilter

Set a functional filter to apply to events in the window.

getData(*filter*)

Table 9 *Parameter for setFilter*

Parameter	Description
`filter`	Specify a functional filter.

Example Code 5 *Example*

```
coll.setFilter("gt($price,200)")
```

load

Load the current page of events from the ESP server.

load()

first

Load the first page of events from the ESP server

first()

last

Load the last page of events from the ESP server.

first()

next

Load the next page of events from the ESP server.

next()

prev

Load the previous page of events from the ESP server.

prev()

play

Set the state of the collection to playing. This means that the ESP server delivers events to the client as they occur. When the state is changed from paused to playing, the server sends an initial page load.

play()

pause

Set the state of the collection to paused. This means that the server does not deliver events to the client as they occur.

pause()

EventStream Object

An Event Stream is a flow of events that is similar to a UNIX tail. When an event occurs in an ESP model in the server, it is placed into the Event Stream. Clients can read this stream and view the events as they occur. When a window produces a very large number of events, you can throttle the number of events injected into the stream by specifying `maxevents` and `interval`.

The `maxevents` keyword parameter limits the number of events that get put into the stream at any one time. The `interval` keyword parameter specifies a time, in milliseconds, that the ESP server waits before putting events into the stream.

Suppose that you set `maxevents` to 1000, and `interval` to 1000 ms (or 1 second). The ESP server collects events from the publish/subscribe interface for 1 second. When more than `maxevents` events occur in that interval, the oldest events are dropped and are not put into the stream. At the end of 1

second, the server puts all of the events that it has into the stream. For clients with heavy processing duties (for example, those that render graphs), this enables the client to limit the number of events that it must process.

When delete events are of no interest, specify `ignore_delete=True` when you create the Event Stream. This discards any delete events received by the stream.

Create Event Streams through the ServerConnection object.

```
rates = conn.getEventStream({window:"primary/cq/counter",maxevents:20})
```

The Event Stream delivers change events to its delegates. If you want stream change notifications, you must register a delegate that implements the `dataChanged` method:

```
function
handle(collection,data,clear)
{
    console.log("data changed: " + JSON.stringify(data,null,"\t"));
}

alerts.addDelegate({dataChanged:handle});
```

The `dataChanged` method receives the stream that changed along with the new data in the `data` parameter.

EventStream Methods

addDelegate

Add a delegate to receive collection change notifications.

addDelegate(*delegate*)

Table 10 *Parameter for addDelegate*

Parameter	Description
`delegate`	Specify the object to receive collection change notifications.

removeDelegate

Remove a delegate from the collection.

removeDelegate(*delegate*)

Table 11 *Parameter for removeDelegate*

Parameter	Description
`delegate`	Specify the object to remove.

getData

Return the collection of events. The data is an array of objects that represent the events.

getData()

setFilter

Set a functional filter to use on events in the window.

setFilter(*filter*)

Table 12 *Parameter for setFilter*

Parameter	Description
`filter`	Specify the functional filter.

Example Code 6 *Example*

```
coll.setFilter("gt($price,200)")
```

Publisher Object

The Publisher object enables you to publish events into an ESP source window. You can add data to the send queue either by one or more sequences of `begin()`, `set()`, `end()` calls. Alternatively, you can use the `add()` method that adds an object to the send queue directly. The publisher stores the objects to be published until the `publish()` method is called.

For example, this first set of calls sets up the send queue and then assigns values to the fields of an event:

```
publisher = conn.getPublisher({window:"p/cq/uievents"});
publisher.begin()
publisher.set("x",mouse.x)
publisher.set("y",mouse.y)
publisher.set("type","click")
publisher.end()
publisher.publish()
```

This second set of calls adds data to the queue:

```
publisher = conn.getPublisher({window:"p/cq/uievents"});
publisher.add({"x":mouse.x,"y":mouse.y,"type":"move")
publisher.add({"x":mouse.x,"y":mouse.y,"type":"click")
publisher.publish()
```

Publisher Methods

begin

Initialize the current data.

begin()

set

Set a field value in the current data.

set(*name*, *value*)

Table 13 Parameters for set

Parameter	Description
`name`	Specify the name of the field to set.
`value`	Specify the value of the field.

end

Close input to the current data. Then add the data to the publish list.

end()

add

Add data to the publish list.

add(*data*)

Table 14 Parameter to add

Parameter	Description
`data`	Specify the data object to add to the publish list.

publish

Publish the publish list into the Source window.

publish()

publishCSV

Publish CSV data into a Source window.

publish(*data*, pause, close)

Table 15 Parameter for PublishCSV

Parameter	Description
`data`	Specify the CSV data.
`pause`	Specify the interval, in milliseconds to pause between events. The default value is 0.
`close`	Specify **`true`** or **`false`** to determine whether to close the publisher when publishing has completed. The default value is **`false`**.

Stats Object

The Stats object enables you to monitor ESP server statistics such as memory usage and CPU usage on a per window basis.

To receive notifications when the data changes, set up a delegate object that implements the `handleStats(self,stats)` method:

```
function
handle(s,stats,memory)
{
    console.log("stats");
    console.log("\t" + JSON.stringify(stats,null,"\t"));
    console.log("memory");
    console.log("\t" + JSON.stringify(memory,null,"\t"));
}

conn.getStats().addDelegate({dataChanged:handle});
```

The Stats array contains CPU and window count information for any window that meets one or both of the following criteria:

- CPU usage is greater than the minimum value.
- The window event count is greater than 0.

The memory information includes values for system, virtual, and resident memory currently consumed in the ESP server.

Stats Methods

addDelegate

Add a delegate to receive Stats change notifications.

addDelegate(*delegate*)

Table 16 *Parameter*

Parameter	Description
`delegate`	Specify the object to receive change notifications.

setOpts

Set a Stats reporting option.

setOpts(*parameter*)

The *parameter* can be any one of the following:

- **`interval`** - the interval, in seconds, at which the server sends data to the client.
- **`cpu`** - the minimum CPU usage, in percentage, which is reported. The default value is 5.
- **`memory`** - specify **`true`** or **`false`** to determine whether to report memory usage information (defaults to **`true`**).

- **counts** - specify **true** or **false** to determine whether to report window event counts (defaults to **false**).
- **config** - specify **true** or **false** to determine whether to report ESP server configuration information (defaults to **false**).

getData

Return the Stats information.

getData()

The returned Stats data would look something like this:

```
[
    {
        'project': 'primary',
        'contquery': 'cq',
        'window': 'transform',
        'cpu': 100.598,
        'interval': 1101676.0,
        'count': 0,
        '_key_': 'primary.cq.transform'
    }, {
        'project': 'primary',
        'contquery': 'cq',
        'window': 'rawTrades',
        'cpu': 31.9233,
        'interval': 1101676.0,
        'count': 0,
        '_key_': 'primary.cq.rawTrades'
    }]
```

getMemoryData

Return memory information.

getMemoryData()

The returned memory data would look something like this:

```
{
    'system': 386696,
    'virtual': 1452,
    'resident': 241
}
```

Log Object

The Log object enables you to monitor ESP server logs. In order to receive notifications when the Stats data changes, you must add a delegate object that implements the `handleLog(log,message)` method:

```
function
handleLog(log,message)
{
    console.log("got a message: " + message);
}
```

```
conn.getLog().addDelegate({handleLog:handle});
```

The `log` parameter is an instance of the Log object. The `message` parameter is string containing the text of a log message.

Log Method

addDelegate

Add a delegate to receive log messages.

addDelegate(*delegate*)

Table 17 *Parameter for addDelegate*

Parameter	Description
`delegate`	Specify the object to receive log messages.

Creating the Visuals Instance

The Visuals object is used to create all the charts, tables, and viewers in the ESPJS API. You can create instances of this object by using the API handle delivered upon start-up:

```
function
esp(api)
{
    _api = api;
    _visuals = _api.createVisuals(_api.getParms());
    ...
}
```

The Visuals object is used to create all the charts, tables, and viewers. Here are the keyword parameters that you can specify when creating the Visuals instance:

- `theme` - the color theme
- `colors` - a list of colors in string format (for example, `["#89cff0","#0080ff","#f0bd27","#ff684c","#e03531"]`)
- `font` - the font to use for graphics. The default value is `{family:"AvenirNextforSAS",size:14}`.
- `title_fone` - the font to use for graphics headers. The default value is `{family:"AvenirNextforSAS",size:18}`.

You can specify a color theme for the instance upon creation. These colors are used to render any of the graphics created by the instance.

Table 18 *Color Themes*

Theme	Elements
Plotly color scales	■ Blackbody ■ Bluered ■ Blues ■ Earth ■ Electric ■ Greens ■ Greys ■ Hot ■ Jet ■ Picnic ■ Portland ■ Rainbow ■ RdBu ■ Reds ■ Viridis ■ YlGnBu ■ YlOrRd
SAS Color Themes	■ sas_base ■ sas_corporate ■ sas_dark ■ sas_highcontrast ■ sas_ignite ■ sas_inspire ■ sas_light ■ sas_marine ■ sas_midnight ■ sas_opal ■ sas_sail ■ sas_snow ■ sas_umstead ■ sas_hcb

The call to create the Visuals instance looks something like this:

```
visuals = Visuals({theme:"sas_corporate"});
```

Creating Charts

Overview

After you have an active server connection and you have created your Visuals instance, you can invoke the appropriate chart creation method. After that, use the name of the chart when and where you want to display it:

```
bar = visuals.createBarChart(alerts,y=["total","restrictedTrades"])
bar
```

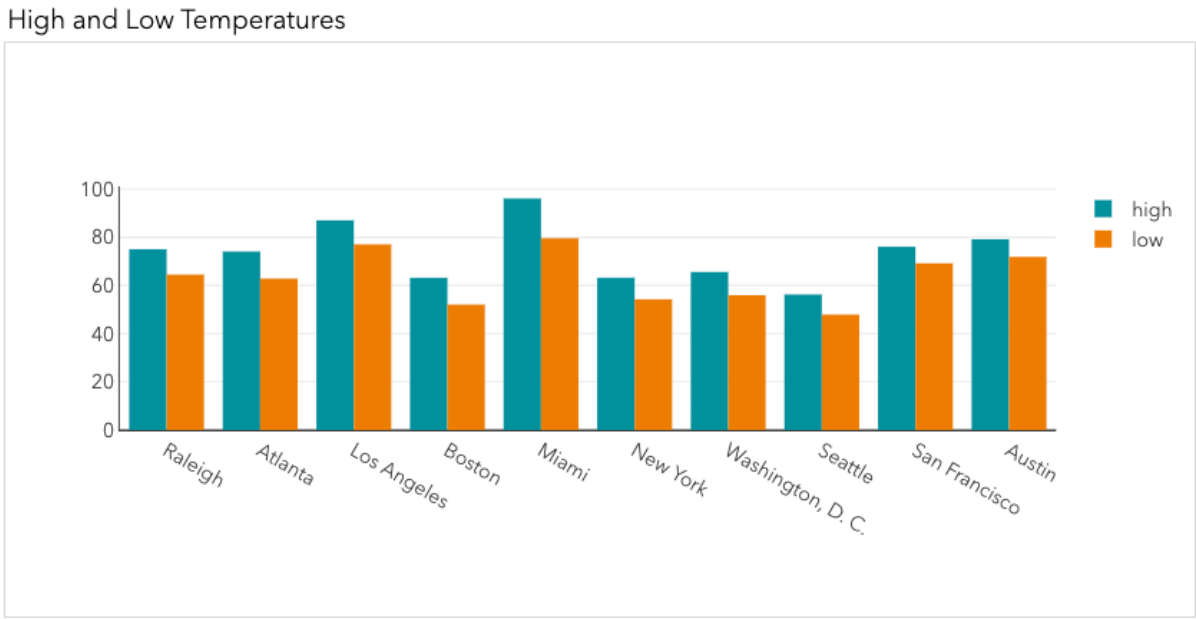
All charts share the following parameters:

Table 19 *Parameters Common to All Charts*

Parameter	Description
`header`	Specify the header of the chart.
`show_controls`	Specify **`true`** or **`false`** to determine whether to show paging controls (for EventCollections) and filtering (all objects). When you set this parameter to **`true`**, a control panel appears at the bottom of the chart. It enables you to perform page navigation, pause or play, and filtering.
`header_text`	Specify the chart header text.

Bar Charts

Use a bar chart to plot multiple numeric variables on X and Y axes.



Use code like the following to create a bar chart:

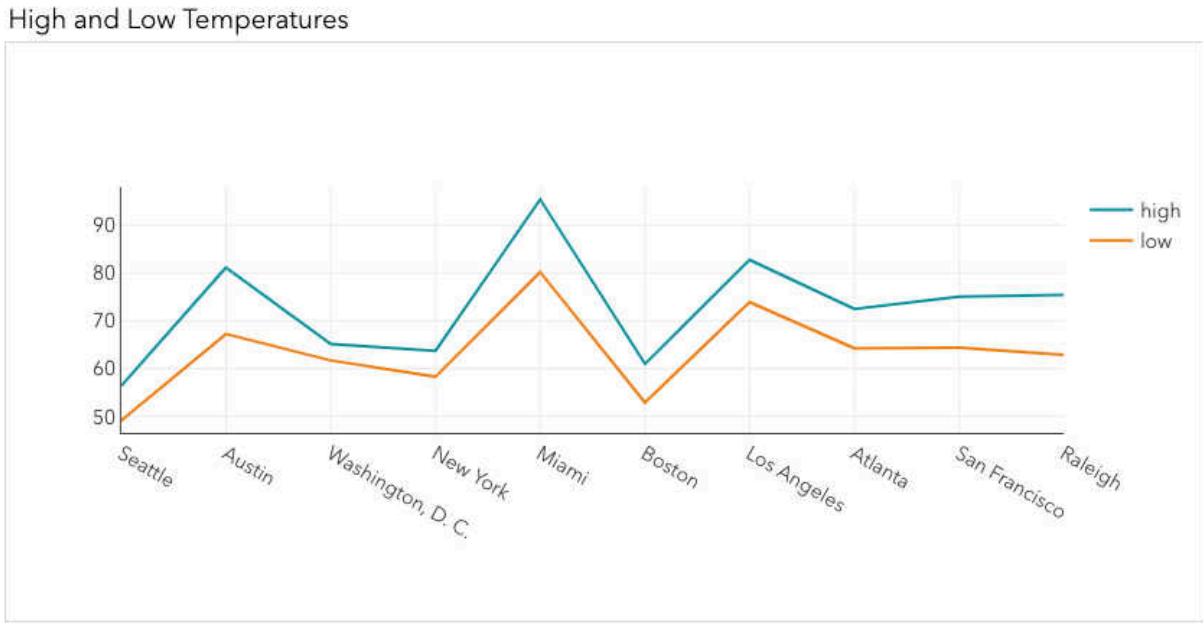
```
violationChart = visuals.createBarChart("mychart",temps,{y:["total","restrictedTrades"],header:"Viola
```

Table 20 Parameters for Bar Charts

Parameter	Description
`id`	Specify the ID of the HTML element in which the chart is drawn.
`datasource`	Specify the event collection or event stream that feeds the bar chart.
`x`	Specify an array of classification field values to display on the X axis. If you do not specify a value for `x`, event key values are used.
`y`	Specify an array of numeric field values to display on the Y axis (or X axis, depending on the value of `orientation`).
`orientation`	Specify the orientation of the chart. Valid values are **`vertical`**(default) and **`horizontal`**.

Line Charts

Use a line chart to plot multiple numeric variables on the Y axis.



Use code like the following to create a line chart:

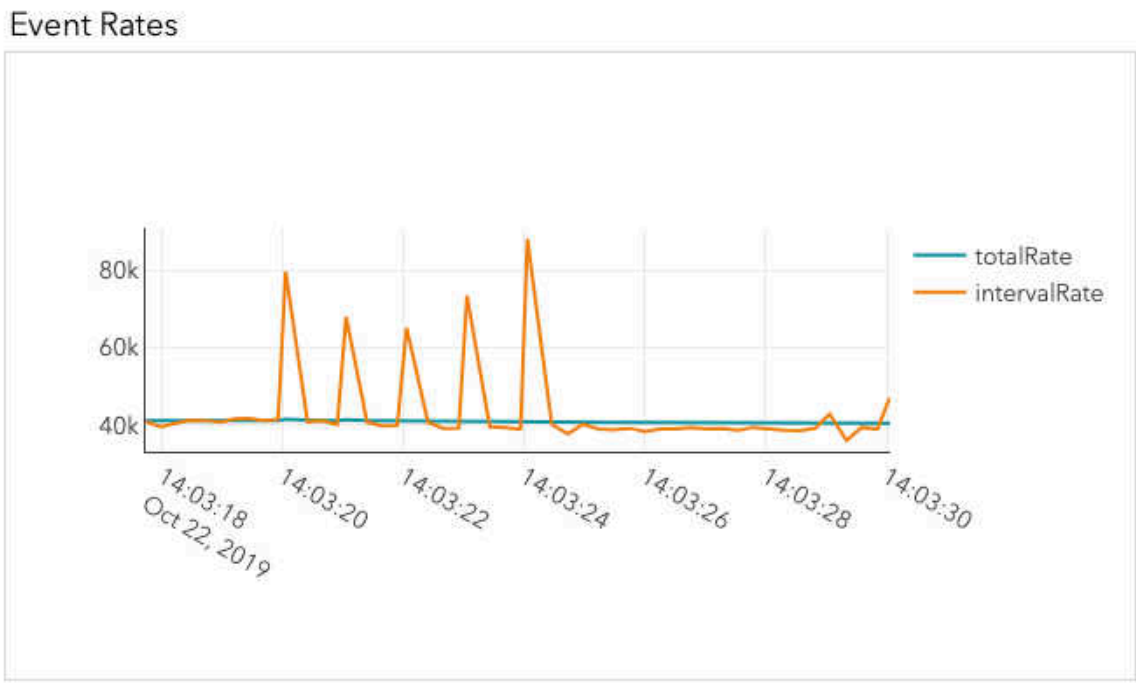
```
rateChart = visuals.createLineChart("rates",eventRate,
                                    {y:["totalRate","intervalRate"],
                                     header:"Event Rate"})
```

Table 21 Parameters for Line Charts

Parameter	Description
`id`	Specify the ID of the HTML element in which the chart is drawn.
`datasource`	Specify the event collection or event stream that feeds the line chart.
`x`	Specify an array of classification field values to display on the X axis. If you do not specify a value for `x`, event key values are used.
`y`	Specify an array of numeric field values to display on the Y axis.
`line_width`	Specify the width of lines in the chart, in pixels. The default value is 2.
`curved`	Specify whether the lines are curved. Valid values are **`true`** or **`false`** (lines are straight).
`fill`	Specify whether the lines are filled underneath. Valid values are **`true`** or **`false`**.

Time Series Graphs

Specify a time series graph to plot multiple numeric variables on the Y axis with a date or time variable on the X axis.



To use this graph, you must specify a field value that is either a `date` or `timestamp`. Use code like this to create a time series graph:

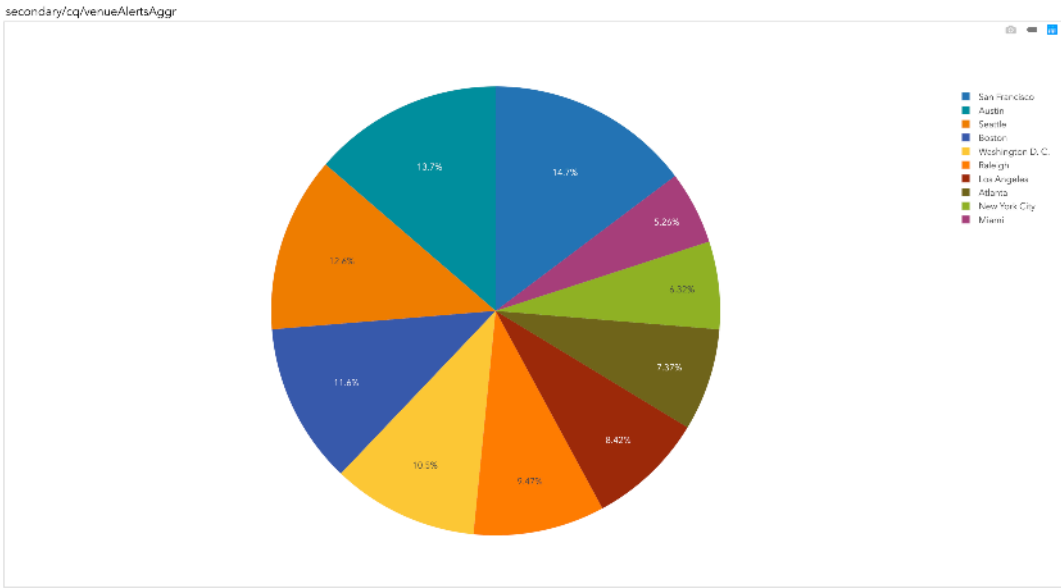
```
rateChart = visuals.createTimeSeries("rates",eventRate,
                                     {time:"_timestamp",y:["totalRate","intervalRate"],
                                      header:"Event Rate"})
```

Table 22 Parameters for Time Series Graphs

Parameter	Description
`id`	Specify the ID of the HTML element in which the chart is drawn.
`datasource`	Specify the event collection or event stream that feeds the time series chart.
`time`	Specify a field value of type `date` or `timestamp`.
`y`	Specify an array of numeric field values to display on the Y axis.
`line_width`	Specify the width of lines in the chart, in pixels. The default value is 2.
`curved`	Specify whether the lines are curved. Valid values are **`true`** or **`false`** (lines are straight).
`fill`	Specify whether the lines are filled underneath. Valid values are **`true`** or **`false`**.

Pie Charts

Use a pie chart to represent a numeric value as a slice of the pie.



Use code like the following to create a pie chart:

```
brokerChart = visuals.createPieChart("venueAlerts",venueAlerts,
                                      {value:"total",header:"Broker Alerts"})
```

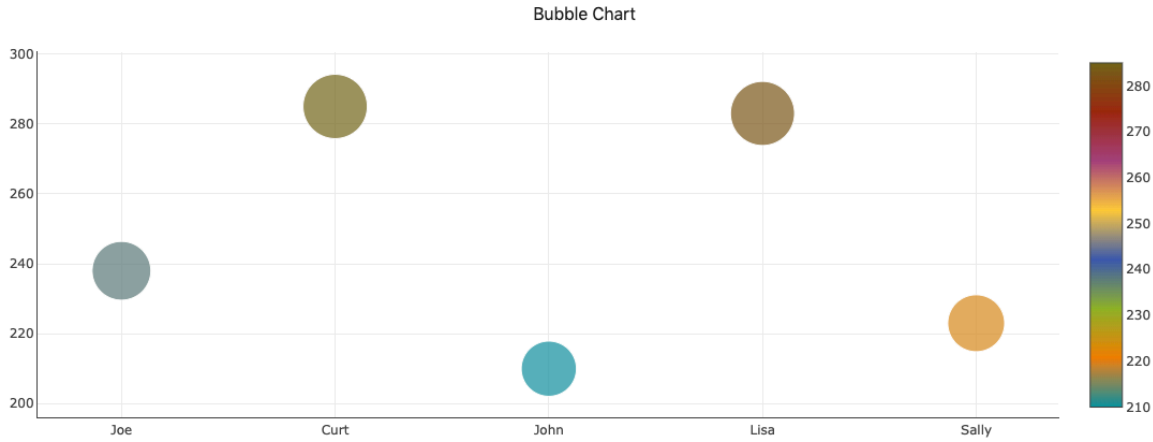
Table 23 Parameters for Pie Charts

Parameter	Description
`id`	Specify the ID of the HTML element in which the chart is drawn.
`datasource`	Specify the event collection or event stream that feeds the pie chart.

Parameter	Description
value	Specify the field value to display in the pie.

Bubble Plots

Use a bubble plot to represent multiple numeric values as y, size, and color visualizations.



Use code like the following to create a bubble plot:

```
brokerChart = visuals.createBubbleChart("brokerAlerts",brokerAlerts,
                                         {y:"restrictedTrades",size:"total",
                                          color:"frontRunningSell",
                                          header:"Broker Alerts"})
```

Table 24 *Parameters for Bubble Plots*

Parameter	Description
id	Specify the ID of the HTML element in which the chart is drawn.
datasource	Specify the event collection or event stream that feeds the bubble plot.
x	Specify an array of classification field values to display on the X axis. If you do not specify a value for x, event key values are used.
y	Specify the field values to display on the Y axis.
size	Specify the field value to use for the size of each bubble.
color	Specify the field value to use for the color of each bubble.

Maps

The map chart uses Leaflet to display a map that is overlaid with markers that represent event data. You can display custom shapes such as circles or polygons on the map.

To add a circle, you must call the `addCircles` method with an EventCollection object that contains events with the circle data. Use the following keyword parameters:

- `lat` - the field representing the latitude of the circle
- `lon` - the field representing the longitude of the circle
- `radius` - the field representing the radius of the circle
- `text` - the field that contains text to display when the user clicks on the circle

To add a polygon, you must call the `addPolygons` method with an EventCollection object that contains events with the shape data. Use the following keyword parameters:

- `coords` - the field that contains a space-separated list of latitude and longitude points in the polygon
- `text` - the field that contains text to display when the user clicks on the polygon
- `order` - the sequencing of the points. When **`lat_lon`** (default), the points are in latitude,longitude order. When **`lon_lat`**, the points are in longitude,latitude order.

Use code like the following to create a map chart:

```
venueMap = visuals.createMap("venueAlerts",venueAlerts,
                                    {lat:"lat",lon:"lon",size:"count",color:"count",
                                     header:"Venue Alerts"})
```

Table 25 *Parameters for Map Charts*

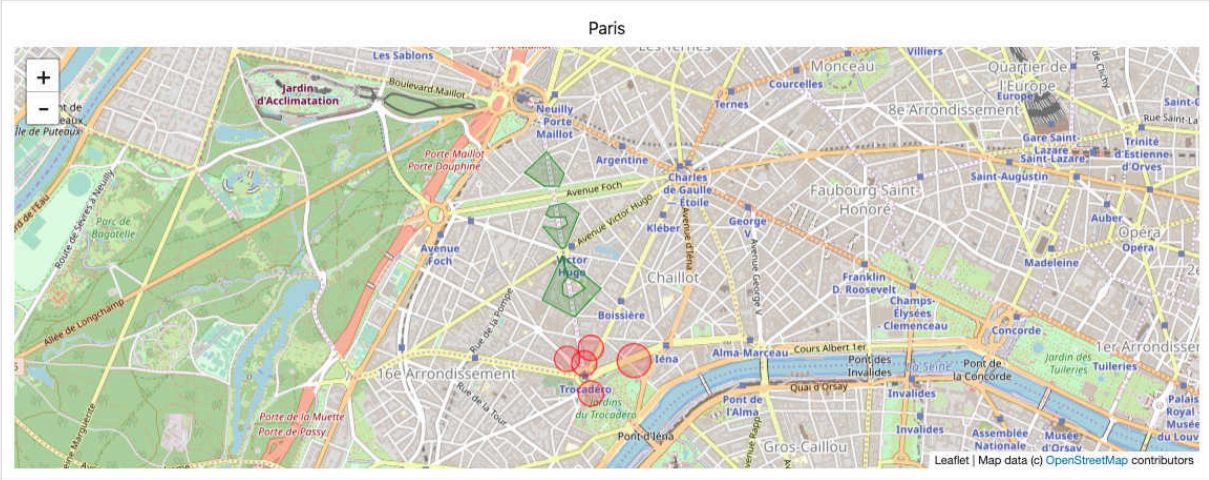
Parameter	Description
`id`	Specify the ID of the HTML element in which the chart is drawn.
`datasource`	Specify an event collection or event stream to feed the map chart.
`lat`	Specify the field value that contains the latitude.

Parameter	Description
`lon`	Specify the field value that contains the longitude.
`size`	Specify either the field value to use to derive the size of each marker or a numeric value for fixed size markers.
`color`	Specify either the field value to use to derive the color of each marker or a fixed color.
`colormap`	Specify the color theme to use to color markers. Use this or `colors`, but not both.
`colors`	Specify a list of colors to use to color markers (for example, **`["#89cff0","#0080ff","#f0bd27","#ff684c","#e03531"]`**. Use this or `colormap`, but not both.
`color_range`	Specify the start and end values against which the data values are compared to derive a color. When this is not set, the minimum and maximum data values are used as the range.
`popup`	Specify the field values to display when the user clicks on a marker.
`marker_border`	Specify **`true`** or **`false`** to determine whether markers have borders.
`marker_opacity`	Specify a value between 0 and 1 to determine marker opacity.
`zoom`	Specify a value between 1 and 18 for the initial zoom value for the map. The default value is 12.
`center`	Specify a (`lat`,`lon`) pair to use as the center of the map. The default value is **`(0,0)`**.
`tracking`	Specify **`true`** or **`false`** to determine whether the map moves such that the initial marker is always in the center.
`circle_border_width`	Specify a numeric value for the width of circle borders, in pixels. The default value is 1.
`circle_border_color`	Specify the color to use for circle borders. The default value is **`black`**.
`circle_fill_color`	Specify the color to use for circle interiors. The default value is **`white`**.
`circle_fill_opacity`	Specify a numeric value for the opacity of circle interiors. The default value is .2.
`poly_border_width`	Specify a numeric value for the width of polygon borders, in pixels. The default value is 1.
`poly_border_color`	Specify the color to use for polygon borders. The default value is **`black`**.
`poly_fill_color`	Specify the color to use for polygon interiors. The default value is **`white`**.
`poly_fill_opacity`	Specify a numeric value for the opacity of polygon interiors. The default value is .2.

The following code generates a map of Paris and adds custom shapes to it:

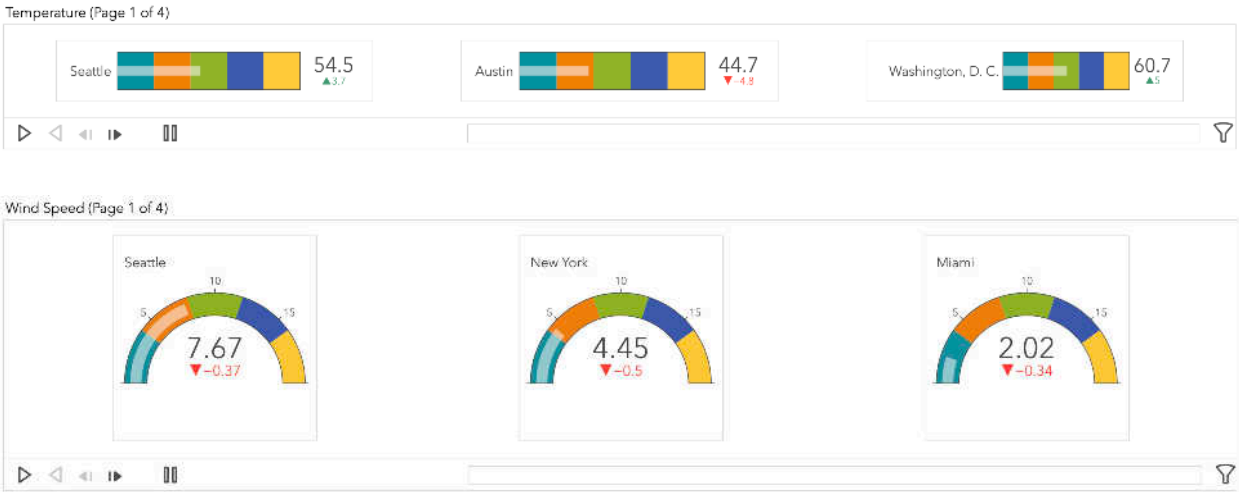
```
paris = visuals.createMap("paris",tracker,{lat:"GPS_latitude",lon:"GPS_longitude",
                                  size:10,color:"speed",color_range:[0,50],
                                  title:"Paris",popup:["vehicle","speed"],marker_border:false,
                                  colors:["#e03531","#ff684c","#f0bd27","#51b364","#8ace7e"],
                                  zoom:15,tracking:true,center:[48.875,2.287583]})
```

```
paris.addCircles(circles,{lat:"POI_y",lon:"POI_x",radius:"POI_radius",text:"POI_desc"})
paris.addPolygons(polygons,{coords:"poly_data",text:"poly_desc",order:"lon_lat"})
```



Gauges

Use gauges to display a single numeric value in a graphic divided into segments.



For each event, a gauge is displayed that shows the specified numeric value of the event and a pointer that shows where that value lies in the specified range. The key value of the event is displayed at the top of the gauge with the current gauge value.

When you specify a single gauge color, it applies to the leftmost segment. The remaining segments are colored with an increasingly darker shade of the specified color. Alternatively, you can specify a color palette that is applied in the segments left to right.

Use code like this to create gauges:

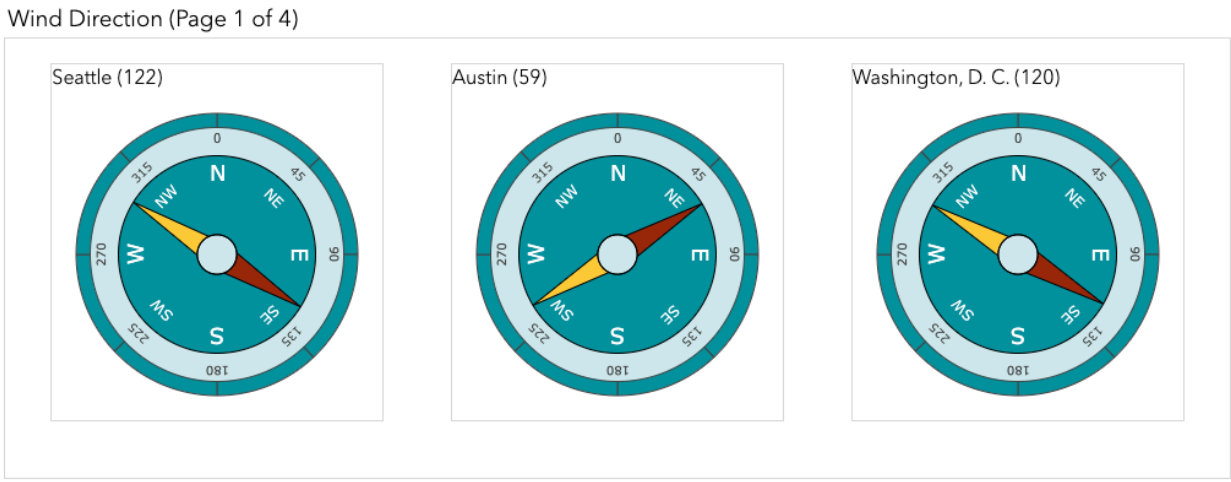
```
var colors = ["#8ace7e","#51b364","#f0bd27","#ff684c","#e03531"];

gauge = visuals.createGauge("windspeed",speed,{segments:5,value:"total",
                                                size:300,range:[0,1000],colors:colors,
                                                shape:"bullet",header:"Wind Speed"})
```

Table 26 Parameters for Gauges

Parameter	Description
`id`	Specify the ID of the HTML element in which the chart is drawn.
`datasource`	Specify an event collection or event stream to feed the gauges.
`segments`	Specify the number of segments to display in the gauge. The default value is 3.
`value`	Specify the field value that contains the numeric value to display in the gauge.
`size`	Specify the pixel size of the gauge. The default value is 300.
`shape`	Specify a numeric value between 40 and 89 that determines how much of the circle is occupied by the gauge. The default value is 50.
`range`	Specify the start and end values for the range displayed in the gauge. The default value is `(0,100)`.
`columns`	Specify the number of columns to display. The default value is 3.
`color`	Specify the color to use for the leftmost segment in the gauge. The remaining segments are colored with a darker gradient of this value.
`colors`	Specify a color palette to display in the gauge.
`line_width`	Specify the pixel width of the lines used to draw the gauge.
`delta`	Specify **`true`** or **`false`** to determine whether to display the change in values.

Compasses

Use a compass to display a numeric value that represents a navigational heading.



Use code like this to create a compass:

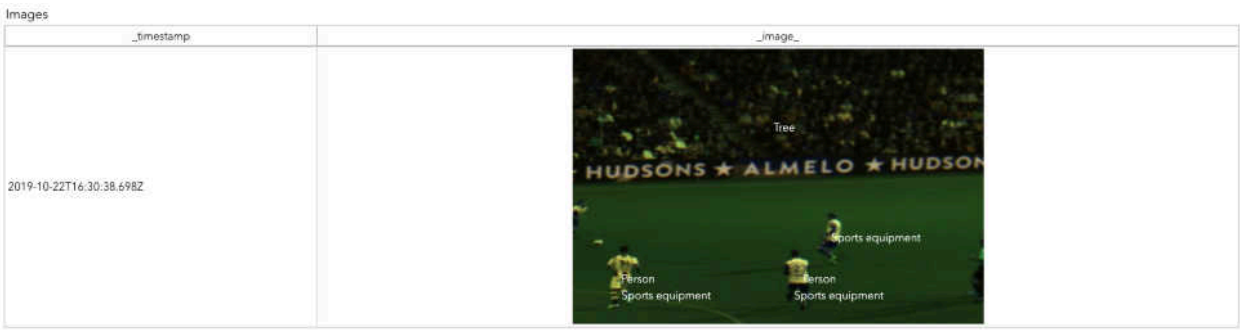
```
compass = visuals.createCompass("heading",heading,
                                        {heading:"total",size:300,
                                         heading:"Compass"})
```

Table 27 Parameters for Compasses

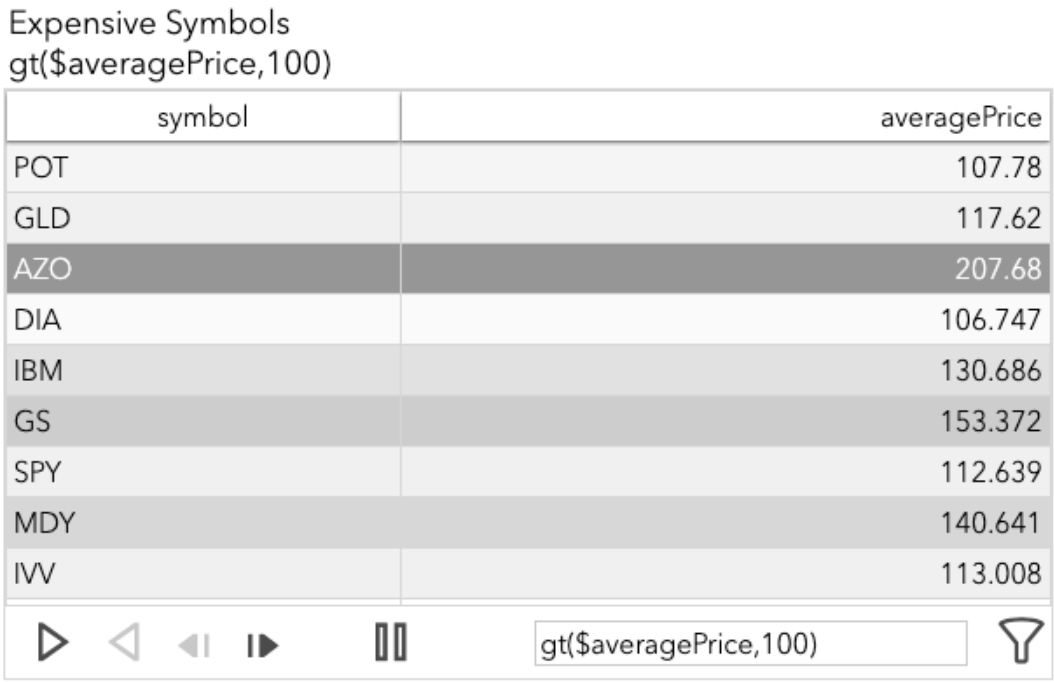
Parameter	Description
`id`	Specify the ID of the HTML element in which the chart is drawn.
`datasource`	Specify the event collection or event stream that feeds the compass.
`heading`	Specify the field value that contains the navigational heading to display in the compass.
`size`	Specify the pixel size of the gauge. The default value is 300.
`columns`	Specify the number of columns to display. The default value is 3.
`heading_color`	Specify the color to use for the heading arrow. The default value is **`#89cff0`**.
`reciprocal_color`	Specify the color to use for the opposite arrow angle. The default value is **`white`**.
`bg_color`	Specify the color to use for the center of the compass. The default value is **`#f8f8f8`**.
`outer_color`	Specify the color to use for the outer circle of the compass. The default value is **`#89cff0`**.
`line_width`	Specify the pixel width of the lines used to draw the gauge. The default value is 1.

Tables

A table can display any number of event fields. When you have an event field that contains an image, the image is displayed within the table. When the event has object detection information within it, the objects are labeled inside the image.



You can also apply a gradient color to the table rows depending on the value of a certain field.



Use code like this to create a table:

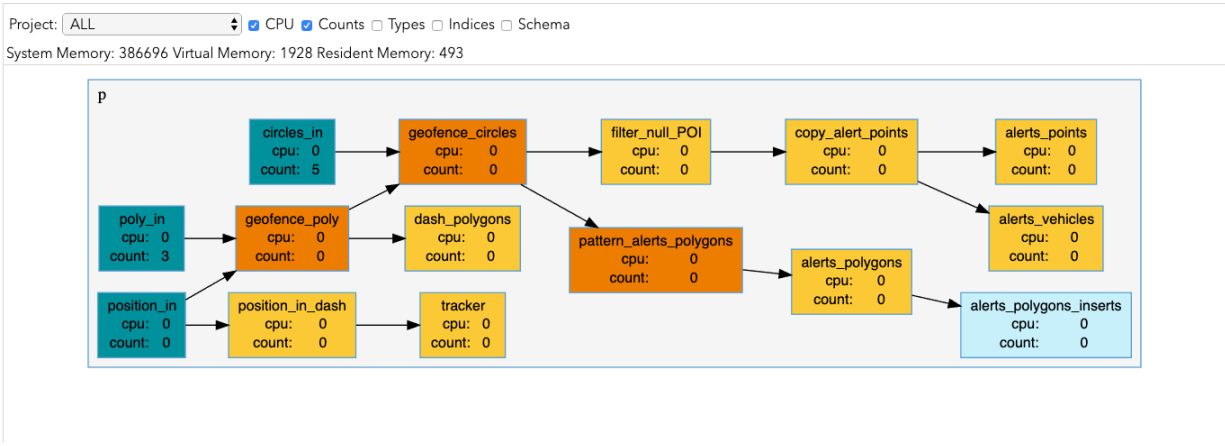
```
table = visuals.createTable("symbols",expensiveSymbols,
                                    {values:["brokerName","total","restrictedTrades"],
                                     header:"Expensive Symbols"})
```

Table 28 *Parameters for Tables*

Parameter	Description
`id`	Specify the ID of the HTML element in which the chart is drawn.
`datasource`	Specify the event collection or event stream that feeds the table.
`values`	Specify the event values to display in the table. **Note:** When you do not specify a `values` parameter, all event fields are displayed.
`reversed`	Specify **`true`** or **`false`** to determine whether to add the most recent events to the beginning or end of the table. The default value is **`false`**.
`image_width`	Specify the width of images that are included in the table.
`image_height`	Specify the height of images that are included in the table
`color`	Specify the numeric field to use to color the rows of the table.
`base_color`	Specify the base color to use when coloring the table rows. By default, the lightest theme color is used.

Event Stream Processing Model Viewer

The Model Viewer displays a graphical representation of the model in the form of a directed graph. You can view as many attributes of each window as you choose. Some model attributes are dynamic and some are static. The Model Viewer also displays ESP server memory usage information. System memory, virtual memory, and resident memory information is displayed above the directed graph. This information changes as the model processes the event stream.



Use code like the following to create a model viewer:

```
viewer = visuals.createModelViewer("model",conn,{cpu:true,counts:true,type:false,index:false});
```

Table 29 *Parameters for the Model Viewer*

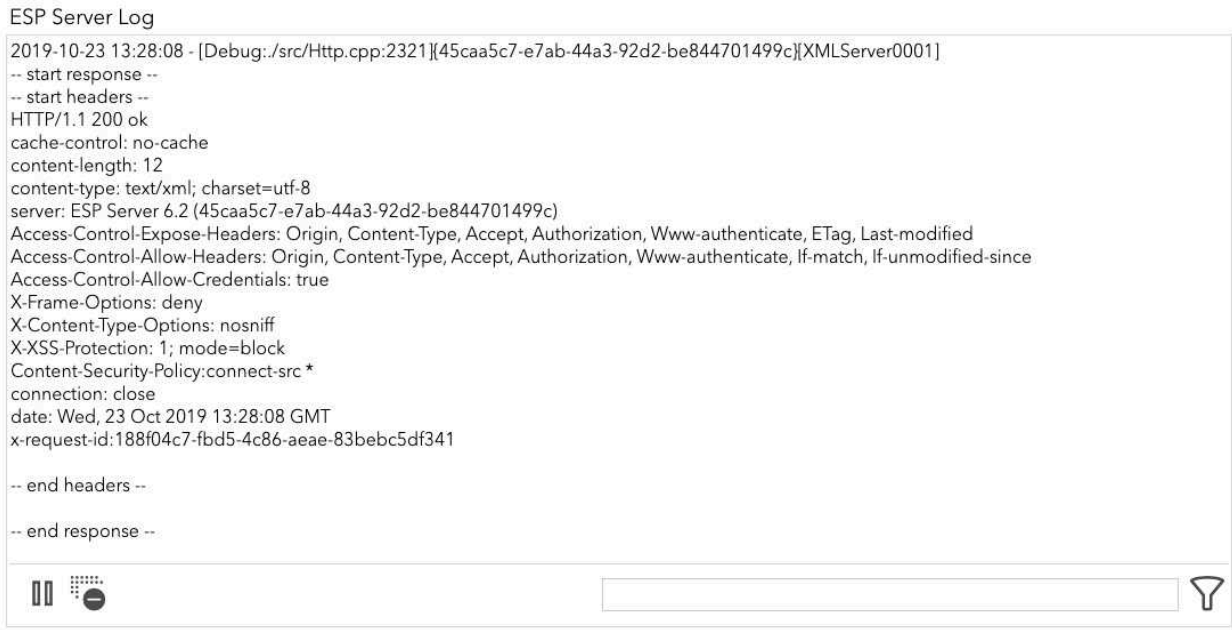
Parameter	Description
`id`	Specify the ID of the HTML element in which the chart is drawn.
`conn`	Specify the connection to the ESP server.
`cpu`	Specify **`true`** or **`false`** to display the CPU usage of the window. The default value is **`false`**.
`counts`	Specify **`true`** or **`false`** to display the number of events currently in the window. The default value is **`false`**.
`type`	Specify **`true`** or **`false`** to display the window type. The default value is **`false`**.
`index`	Specify **`true`** or **`false`** to display the window index type. The default value is **`false`**.
`schema`	Specify **`true`** or **`false`** to display the window index type. The default value is **`false`**.
`cpu_color`	Use a color gradient that begins with the specified value to color models nodes by CPU usage.

By default, the model viewer displays all projects running in the ESP server. However, if you want to focus on a specific project, then specify that before displaying the model viewer:

```
var viewer = visuals.createModelViewer("viewer",conn,{cpu:true,counts:true,type:false,index:false,cpu
viewer.project = "primary";
```

Event Stream Processing Log Viewer

Use the Log Viewer object to view a live ESP server log. Messages appear at the same time that they appear on the console. The Log Viewer displays the most recent log messages at the top.



Use code like this to create a log viewer:

```
visuals.createLogViewer(conn,{filter:"ERROR"})
```

Table 30 *Parameters for a Log Viewer*

Parameter	Description
`id`	Specify the ID of the HTML element in which the chart is drawn.
`conn`	Specify the connection to the ESP server.
`filter`	Specify a filter to use on log entries. When set, you get a text field in which you can change the filter.

Laying Out Visualizations

Because all you need to render the visualizations provided with these objects is an HTML element, you can lay out your pages as you choose. The Flexbox Layout makes it easy to create a simple, responsive page that presents your visuals in an optimal fashion.

All of the ESPJS examples use the Flexbot Layout mechanism. Also, these objects provide sizing support for a standard layout through the Visuals.size() method.

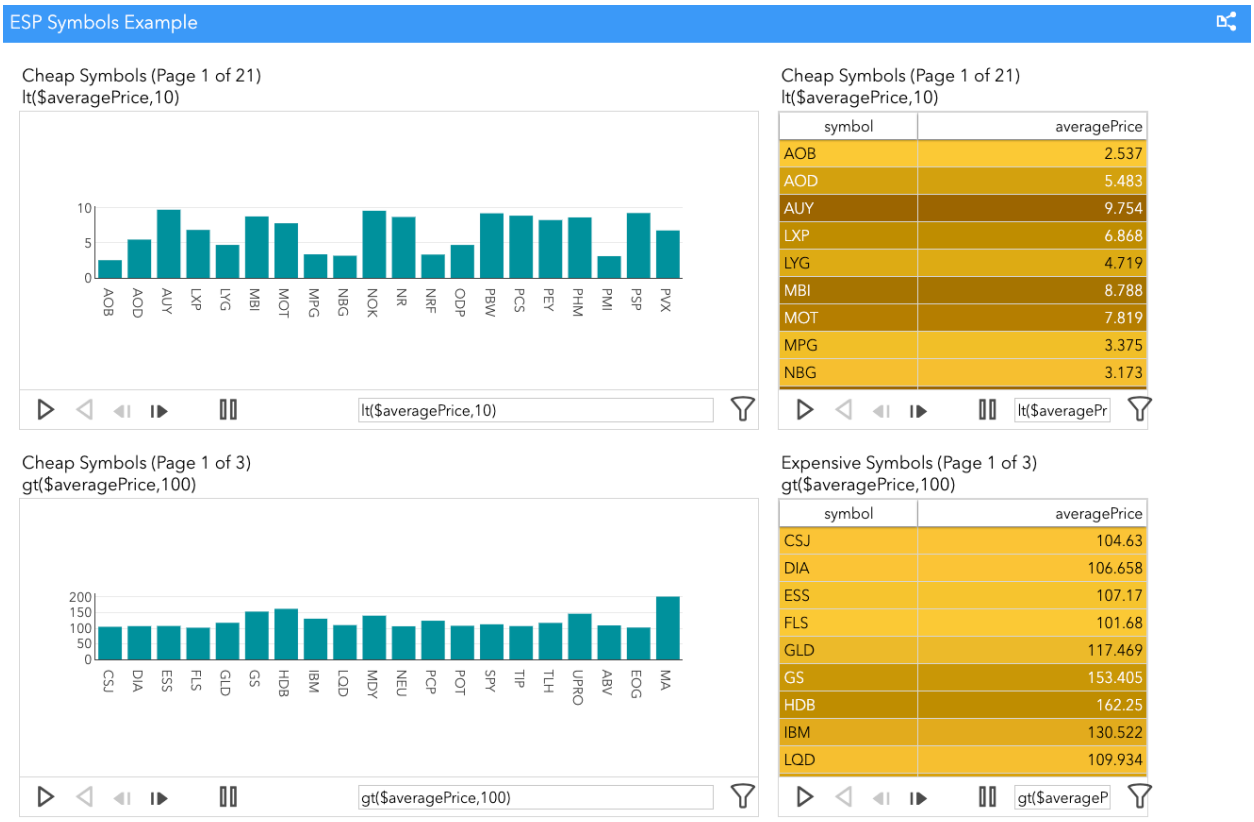
For example, consider this HTML:

```
<body onresize="_visuals.size()">

    <div id="banner">
        <table style="width:100%" cellspacing="0" cellpadding="0">
            <tr>
                <td>ESP Symbols Example</td>
                <td class="icon"><a class="icon" href="javascript:publish()"> </a></td>
            </tr>
        </table>
    </div>

    <div id="content">
        <div class="container">
            <div id="cheapBar" class="component" style="width:60%;height:400px"></div>
            <div id="cheapTable" class="component" style="width:30%;height:400px"></div>
            <div id="expensiveBar" class="component" style="width:60%;height:400px"></div>
            <div id="expensiveTable" class="component" style="width:30%;height:400px"></div>
        </div>
    </div>

    <div id="publish" style="display:none">
        <input type="button" value="Publish"></input>
    </div>

    <div id="footer"> </div>

</body>
```

The page has three components:

- a banner at the top of the page that displays title information and that can contain icons
- page content that contains visualized event information
- a footer at the bottom of the page that displays messages and status

That code produces a layout like this:



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"></d
            </td>
            <td>
               <div id="green" style="width:100%;height:100px;border:1px solid #c0c0c0;background:green"
            </td>
            <td>
               <div id="blue" style="width:100%;height:100px;border:1px solid #c0c0c0;background:blue"><
            </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.