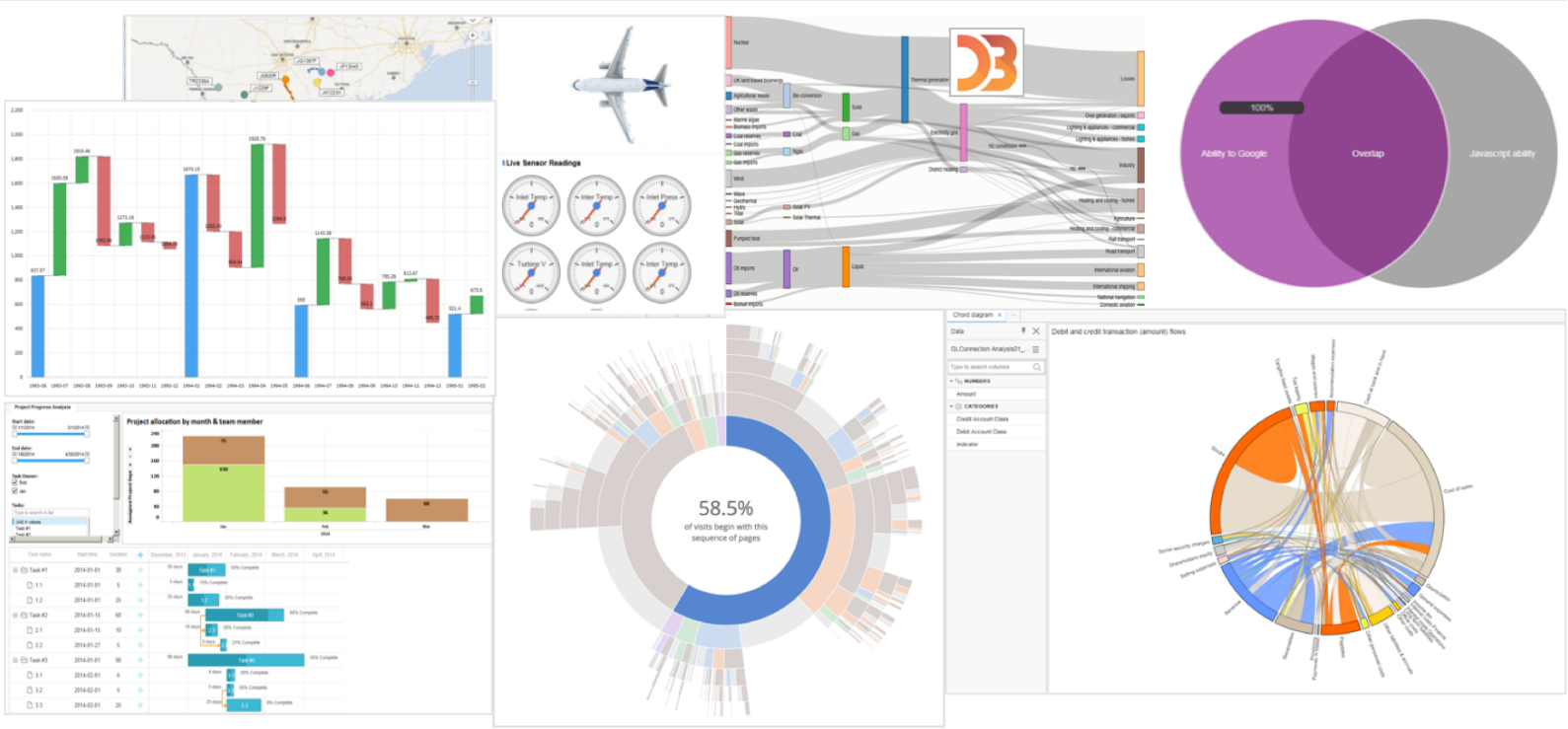
JavaScript Visualization Framework – JSViz V3.4

User Guide



TIBCO® Analytics

August 2017

1. Introduction

Within the Spotfire community, one area of discussion that crops up continually is that of new visualization types. Customers often ask for a visualization type during a POC that Spotfire does not currently support. Others ask for simple customizations of existing visualizations, such as moving labels around, that sound straightforward but are almost impossible to implement without extensive coding.

There are now a number of JavaScript libraries available for creating rich interactive graphics within a standard web browser. One such is D3 which has a large library of example visualizations available on their website http://d3js.org/. There are many others, including Highcharts and Google Charts.

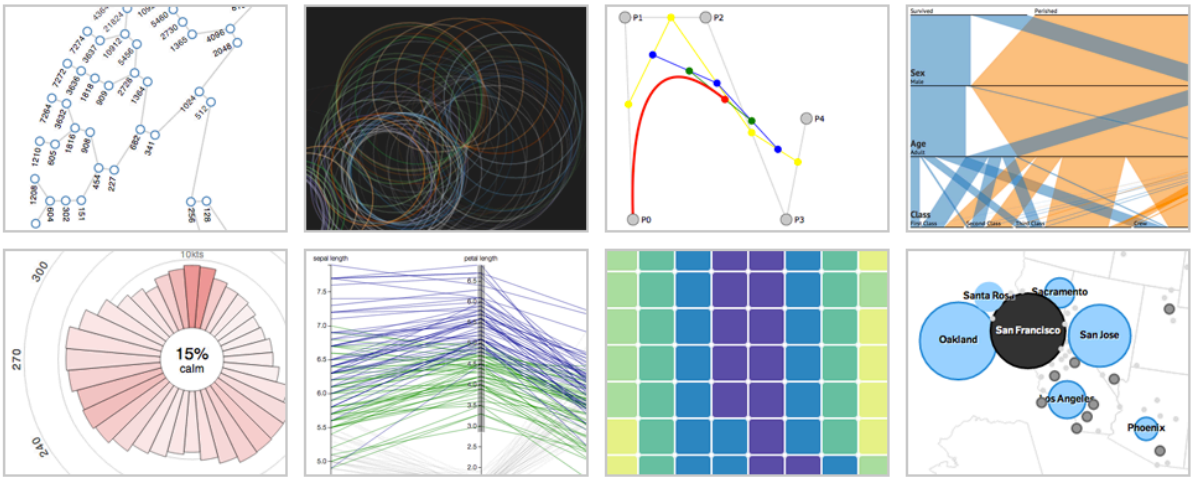


Figure 1 : Example Visualizations from d3js.org website

The JavaScript Visualization Custom Extension, or JSViz, is a plugin for TIBCO Spotfire that allows users to create their own visualizations using JavaScript libraries such as d3 but still allow them to seamlessly integrate with the Spotfire platform. Note that although d3 was an initial driver for implementing the custom extension, it can actually work with any JavaScript or HTML based code.

2. Key Concepts

The diagram below shows how JSViz takes data from within Spotfire and converts it into JSON format. This data is combined with JavaScript code, CSS and HTML files to create visualizations.

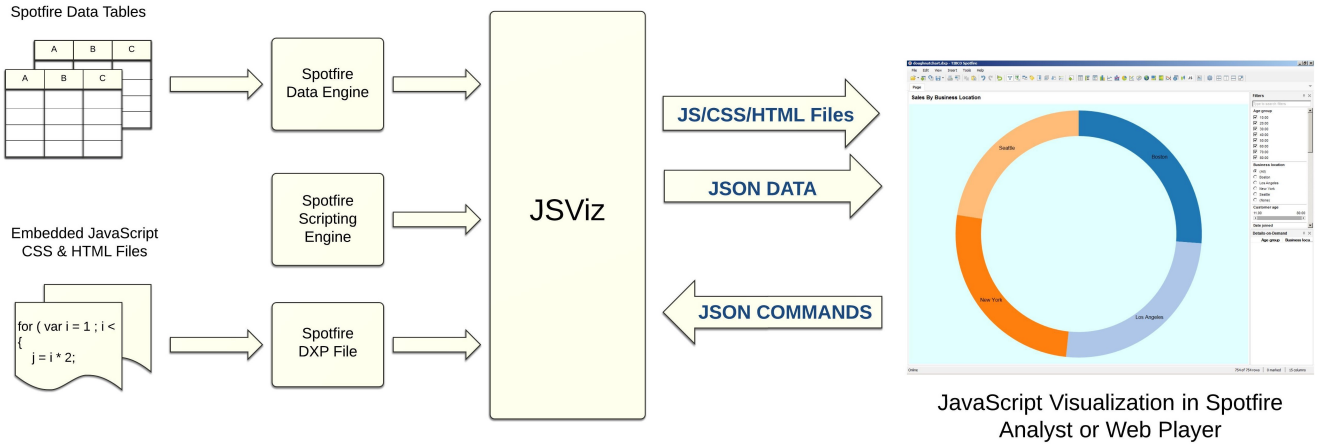


Figure 2 : JSViz Overview

In addition, the JavaScript code can send commands back to JSViz to update the visualization state which then results in the visualization being redrawn.

Turning Spotfire Data into JSON

JSViz allows the data from one or more Spotfire In-Memory data tables to be turned into JSON data that can then be used to create a visualization using JavaScript code.

The JSON data also contains other information:

- Overview Information – total rows, visible rows, marked rows
- Current Marking
- Configuration Data – unique to a visualization type e.g. Color Palette
- Spotfire User Name
- Legend visibility flag

The JSON data is updated whenever the underlying data changes in Spotfire, for instance when a user make changes via the Filter Panel or Marks items.

Appendix A details the structure and contents of the JSON payload sent from JSViz to a visualization.

Turning JSON into Visualizations

JSViz uses HTML, JavaScript and CSS to create visualizations. Typically most JS libraries take the JSON data described above and convert it into SVG objects, however anything that can be done on a regular web page can be done in JSViz.

```
{
  "columns": ["Sales","Location"],
...
  "data": [
    {"items": [1249558,"Boston"]},
    {"items": [1213626,"Los Angeles"]},
    {"items": [1231992,"New York"]},
    {"items": [1074072,"Seattle"]}
  ]
...
}
```

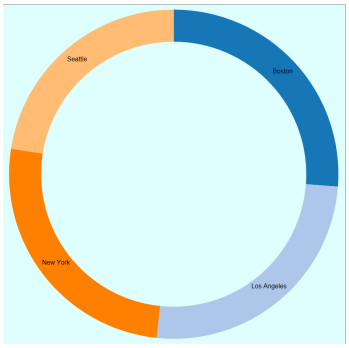


Figure 3 : Converting JSON into a Doughnut Chart

The JavaScript, CSS and HTML files that make up the visualization can either be load from a web server via their URL or embedded directly within the Spotfire DXP file to allow for offline usage.

JSViz as a Spotfire Visualization

JSViz visualizations can be constructed to look and feel like other Spotfire visualizations. Marking and Filtering can control JSViz visualizations and JSViz visualizations can control other visualizations through marking. JSViz visualizations can be copied/pasted and moved around within Spotfire just like any other visualization. They can also Exported and Printed.

To allow the creation of more advanced analytic applications, JSViz also allows the reading and writing of Spotfire Document Properties and the execution of Spotfire IronPython scripts.

This level of interactivity is accomplished by allowing visualization JavaScript code to send commands back to JSViz. The full list of commands is as follows:

- Update Marking
- Execute an IronPython Script
- Update a Document Property
- Update Configuration Data
- Update Runtime State

Appendix B details the function of each of the available commands, along with the structure and contents of the JSON for each command.

JSViz in any Spotfire Client

JSViz visualizations must be authored in Spotfire Analyst, but can then be consumed in Spotfire Analyst and/or Spotfire Consumer and can also be used within Spotfire Automation Services for batch processing.

3. Installation and Configuration

Package Contents

The JSViz installation package consists of a zip file that contains the following folders:

- "For Spotfire Server" - this folder contains the deployable modules for the extension packaged as a Spotfire Distribution (SDN) file. This is the only file required for installation to Spotfire Analyst, Web Player and Automation Services.
- "For Spotfire Analyst" - these are a set of sample DXP files that demonstrate features of JSViz with various JavaScript libraries.
- "For Web Server" - these are the JavaScript, CSS, HTML and data files that are used with the above sample files.
- "Documentation" - this folder contains this User Guide and a set of tutorials to get started with JSViz.

System Requirements

This version of JSViz is compatible with Spotfire 7.5 and above.

Installation

JSViz is installed by adding the installation file from the "For Spotfire Server" folder to a Spotfire Distribution on the Spotfire Server:

- JSVis_<version>.sdn

This is the same process that is used for a standard Spotfire installation. See the following section in the *TIBCO Spotfire Server and Environment Installation and Administration* Manual for more details:

- Administration → Deployments and Deployment Area → Adding software packages to a deployment area

When Spotfire Analyst is next started it will prompt that updates are available and the packages will be downloaded.

All Web Player and Automation Services instances will need to be manually updated. See the following sections in the *TIBCO Spotfire Server and Environment Installation and Administration* Manual for more details:

- Administration → Nodes and Services → Updating Services

Spotfire Licenses

There are two licenses that control access to the JavaScript Visualization Extension:

- `JSVisualization` – users with this license can view JavaScript visualizations
- `JSVisualizationAdminLicense` – users with this license can modify JavaScript visualizations

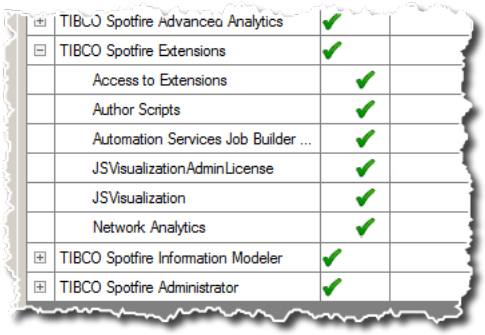


Figure 4 : TIBCO Spotfire Licenses

All users that want to view DXP files containing JSViz visualizations must be granted the first license. Users that wish to modify JSViz visualizations require both licenses.

Web Site Setup and Sample Files Deployment

The sample JavaScript, CSS and HTML files that are provided as part of the installation can be hosted on a web server.

Appendix D shows how to setup a virtual directory on the IIS instance hosting the Spotfire Web Player. However, any web server such as Tomcat or JBoss can be used to host the JS files.

The sample files can be deployed onto the Web Site created above by copying the files from the folder "`For Web Server`" into a folder on the Web Site.

Once the files have been deployed, you can validate that they are installed correctly by opening the Tester file for the Doughnut Chart sample in a web browser:

```
http://<hostname>/jsviz/testing/Tester-DoughnutChart.html
```

Where <hostname> is either the name of the machine running IIS, the IP Address of that machine or `localhost` if you are running the browser on the actual machine.

Validating the Installation

Once JSViz is installed, licensed users should see the JSViz icon appear on the visualizations toolbar in Spotfire Analyst:

Figure 5 : JSViz Icon on Spotfire Analyst Visualization Toolbar

As an additional validation, open the "`doughnutchart.dxp`" sample from the "`For Spotfire Analyst`" folder:

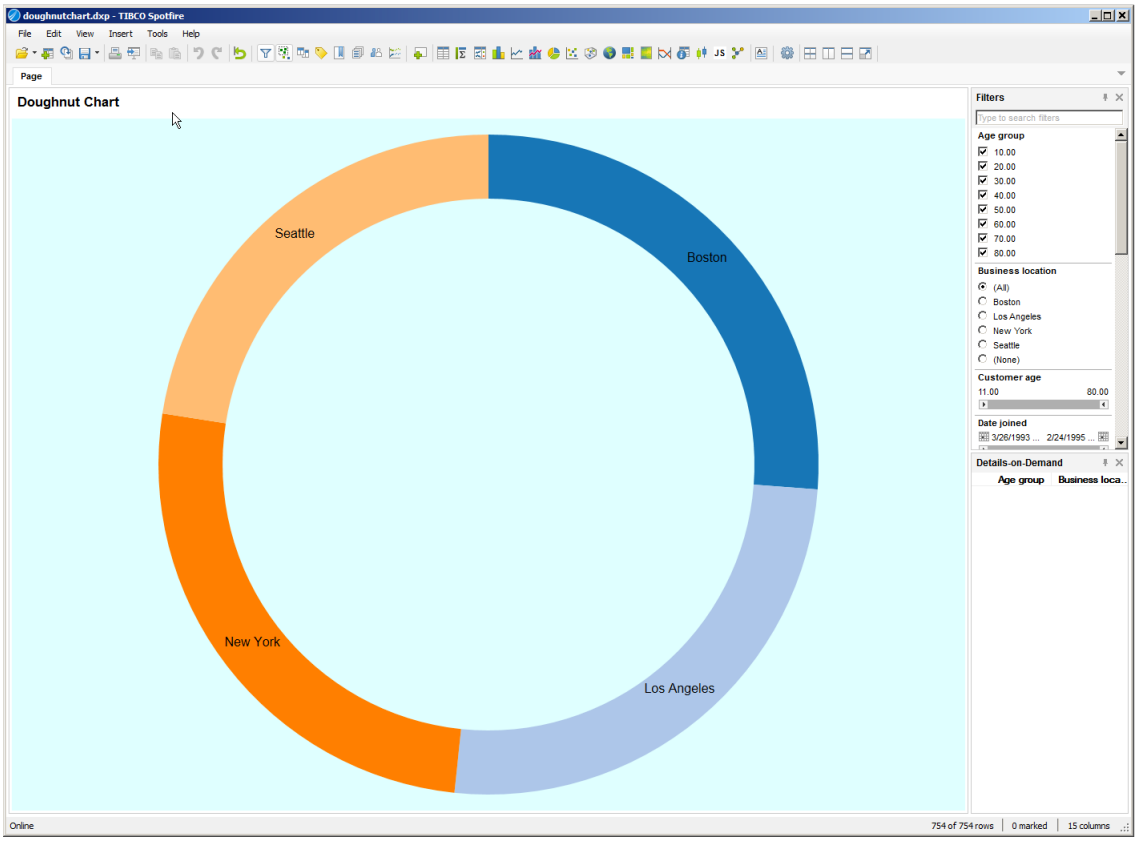


Figure 6 : Doughnutchart.dxp Sample in Spotfire Analyst

4. Visualization Building Workflow

In this section we will walk through the basic process of building a visualization. The subsequent chapters will then dive more deeply into each step, describing the many features of JSViz and how to use and configure them.

The basic workflow is described in the image below:

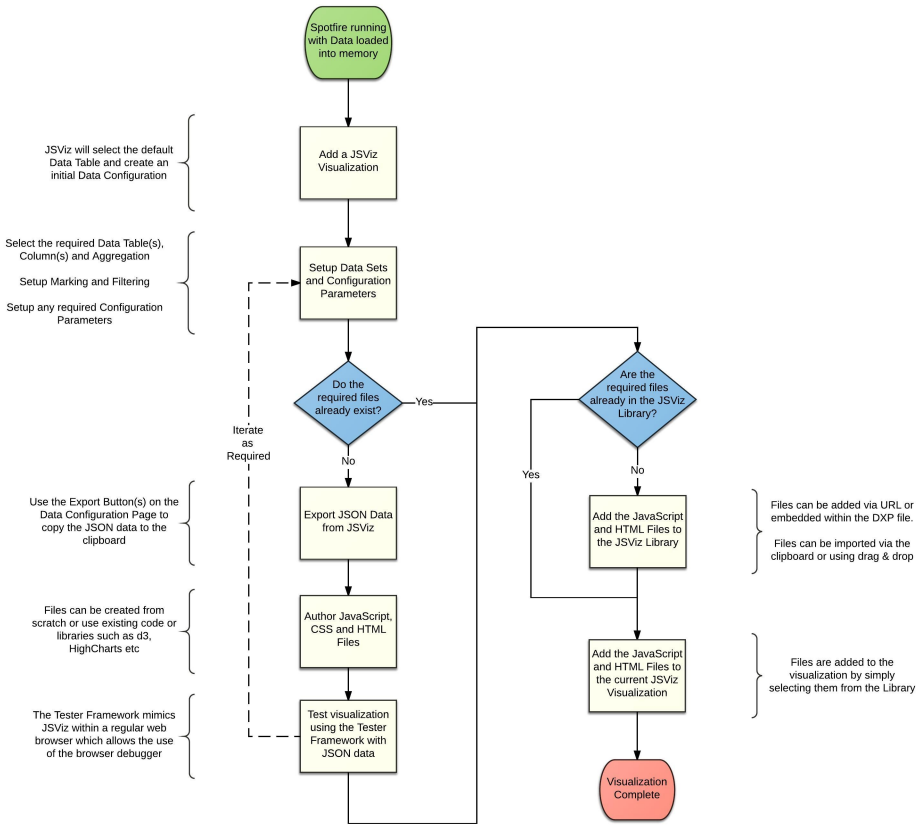


Figure 7 : Basic JSViz Workflow

Adding a JSViz Visualization

This is as simple as clicking on the JSViz icon or adding JSViz from the menu list of visualization types within Spotfire Analyst. JSViz will initialize itself and create a default data configuration based on the default in-memory Spotfire Data Table.

Data Configuration

Most likely the default configuration created with JSViz will not meet your exact requirements so some measure of configuration will be required. This will fall into two basic scenarios:

1. You are re-using some existing JavaScript/CSS files and know the format of data they require
2. You are creating a new visualization and do not know (yet) the format of data required

In Scenario #1 you will use the Data Configuration panel to format the JSON data correctly for the visualization and probably just take a quick look at the data to confirm it is correctly formatted.

In Scenario #2 you will most likely have an initial attempt to configure the data how you think it should be, but then you will need to create your JavaScript/CSS files and try the data. You will then need to iterate over the process of tweaking the data format and the JavaScript/CSS code until they provide the visualization you require.

JSON Export

From the Data Configuration Property Page, you can export the full JSON that is being passed to your visualization code to the Windows clipboard. There are two export modes:

- Formatted – this is intended for human consumption and is formatted with indentation to allow easier reading and interpretation
- Minimized – this is intended for use with the Tester Framework and is formatted on a single line with all indentation, spaces and carriage returns removed

Tester Framework

The Tester Framework consists of a template HTML file called Tester.html that mimics the JSViz plugin and a set of sample files. The basic idea is to copy the Tester.html file to create a specific HTML testing file that you can use to mimic the behavior of JSViz and test and debug your visualization within a Web Browser.

The basic process is as follows:

1. Deploy the Tester files on a Web Server as detailed in the installation instructions.
2. Copy the Tester.html file to create a new testing file for you visualization. For example Tester-MyChart.html
3. Edit Tester-MyChart.html to reference the required JavaScript and CSS files for your visualization. The original Tester.html file has some commented out example entries that you can use as a starting point.

```
<!--
  Place JS and CSS Includes here - uncomment as required
-->

<!--
<script type="text/javascript" src="http://localhost/jsviz/lib/jQuery/jQuery.js"></script>
<script type="text/javascript" src=" http://localhost/jsviz/lib/d3/d3.v3.min.js "></script>
<script type="text/javascript" src="http://localhost/jsviz/src/js/Introspection.js"></script>
-->

<!--
  Place JS and CSS for chart being tested here - uncomment as required
-->

<!--
<link type="text/css" href="http://localhost/jsviz/src/css/<MyChart>.css" rel="stylesheet">
<script type="text/javascript" src="http://localhost/jsviz/src/js/<MyChart>.js"></script>
<script type="text/javascript" src="http://localhost/jsviz/lib/JSViz/JSViz.js"></script> >
-->
```

4. Edit the line in Tester-MyChart.html that assigns the JSON data to the "`sfdata`" variable and paste in the minimized version of the JSON data exported from your JSViz chart. See the section on Previewing JSON Data in Chapter 6 for more details.

```
/*
Paste in the JSON data copied to the clipboard from within your JS Visualization instance
*/

var sfdata = {"columns":[...
```

   Note: you do not need to put a semi-colon at the end of this line.

5. Open the Tester file in a Web Browser using the name of the web server hosting the files and the installation path on that web server. In the example below the files are installed on the local machine under an application folder called "`jsviz`":

   `http://localhost/jsviz/test/Tester-MyChart.html`

6. On the Tester page that appears, press the "`Call renderCore`" button to run the visualization. If everything is setup correctly, you should see you visualization.

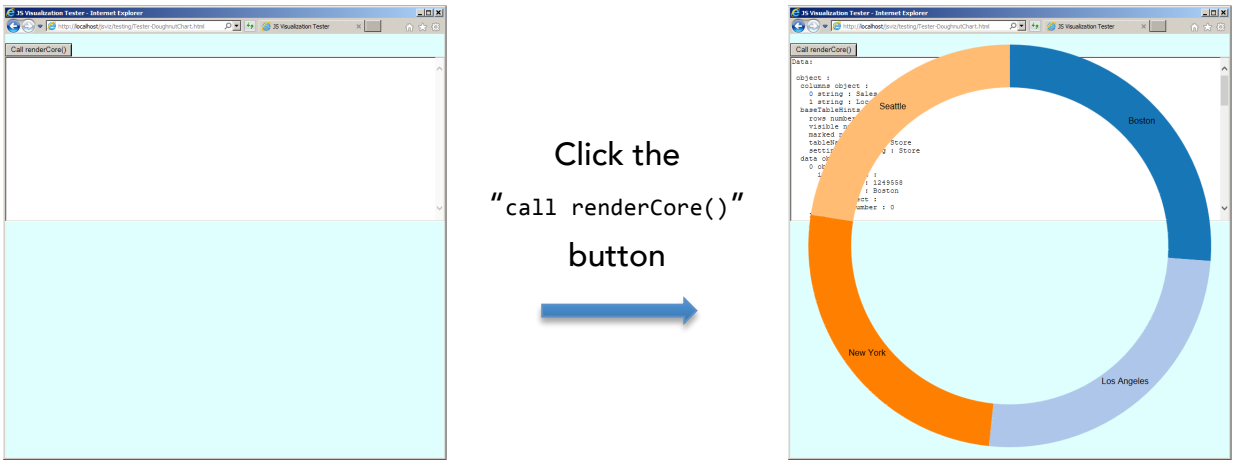


Figure 8 : Calling the renderCore() method from within the Tester

7. If the visualization does not appear or appears incorrect, use the Web Browser debugger to step through the code to understand what is causing the issue.

Hint: Remember to refresh the Web Browser cache whenever the JavaScript or CSS files are updated.

HTML File

By default, JSViz automatically generates the HTML for the browser page that will contain the JavaScript visualization within Spotfire.

However, the user can choose to create a customized HTML page. If an HTML file is found in the JSViz library then it will be used instead of the default HTML content.

JavaScript/CSS Files

If a previous JSViz visualization exists in the DXP file, all the previously used JavaScript and CSS files are available to be re-used. If not then you will need to add the files to the JSViz Library before they can be used within a visualization.

Files can be added to the JSViz Library in one of two ways:

- By Reference – these are referred to as Linked Content and are accessed via their http(s) URL
- By Content – these are referred to as Embedded Content and are stored within the DXP file.

Once the required files have been added to the JSViz Library, they can be added to the visualization in the required order. At this point the visualization should appear.

Debugging in Spotfire Analyst

The embedded Chromium web browser control in Spotfire 7.5 allows the use of its built-in debugger to assist with developing visualization code.

To open the debug console, select the JSViz visualization and press Ctrl-Alt-Shift-F12.

There is also the option to completely refresh an individual visualization by pressing Ctrl-Alt-Shift-F5.

Logging

JSViz provides the facility to send messages from the visualization JavaScript code to JSViz using the "`log`" command. These messages are then written to a logfile.

The logging functionality is only available within Spotfire Analyst and must be enabled each time the DXP file is opened. More details on logging can be found in the Logging Section.

5. Adding and configuring a new JSViz Visualization

A new JSViz visualization can be added via one of the usual methods of adding visualizations:

- By clicking on the JSViz icon in the toolbar
- By selecting "JavaScript Visualization" from the "New Visualization" menu

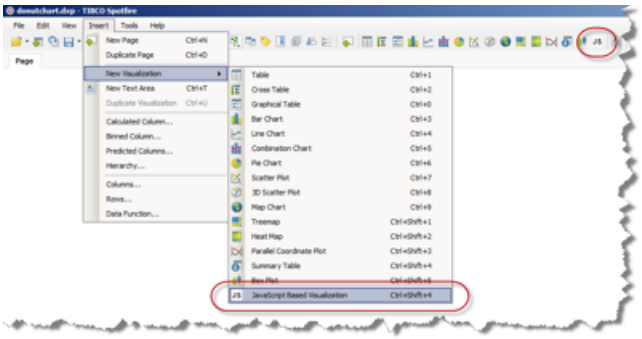


Figure 9 : Adding a JavaScript Visualization via the Menu or the Toolbar

Note: Spotfire only allows Visualizations to be added once some data has been loaded.

When the JSViz visualization is first added, the visualization area will be blank:

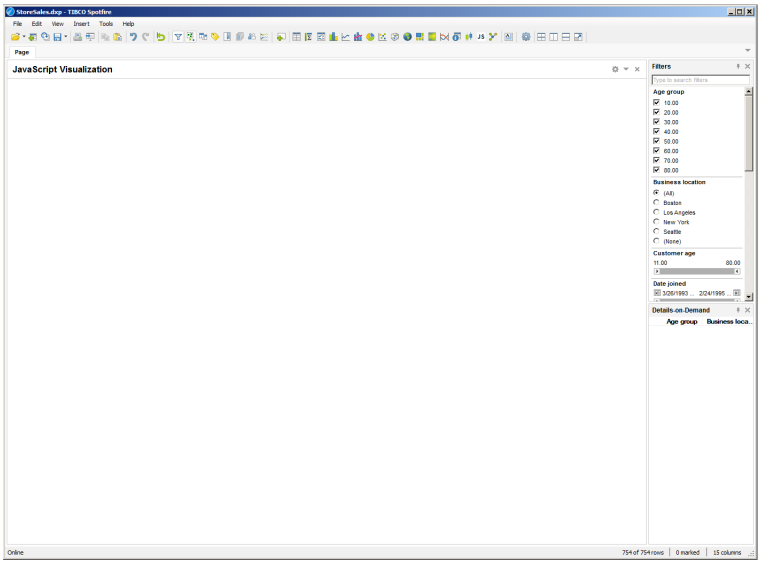


Figure 10 : Spotfire Analyst with New JSViz Visualization Added

The JSViz visualization configures itself as follows:

- A default Data Configuration is created
- The default Data Table in the DXP file is selected
- The first categorical column is selected for Grouping of data
- The first continuous column is selected for aggregation

Opening the Properties Dialog

The Properties Dialog allows the different configuration settings of a JSViz visualization to be modified. Settings are grouped together into logical groups which are shown on different Property Pages.

To open the Properties Dialog, click on the configuration cog in the title bar of a JSViz visualization.

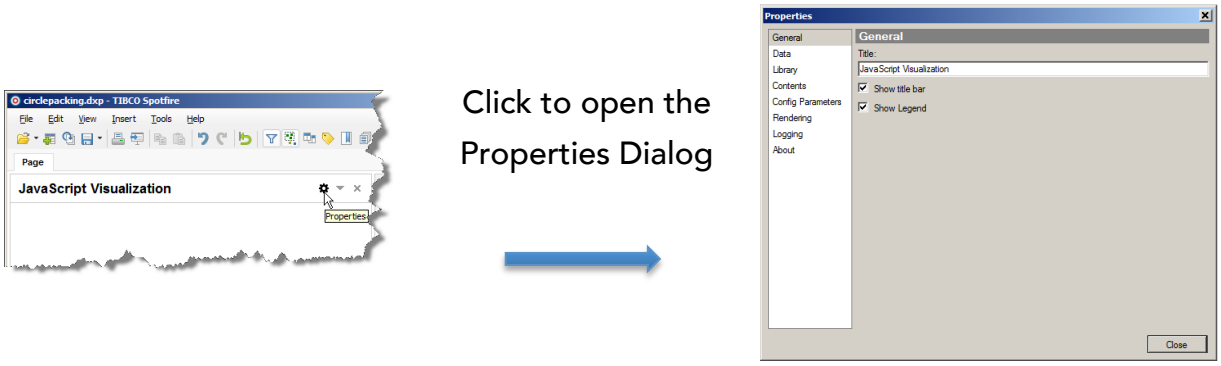
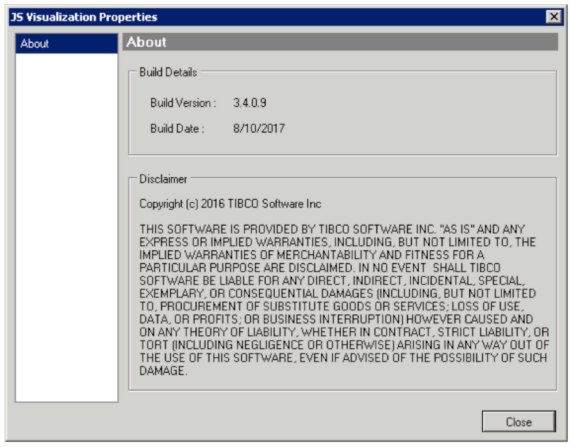


Figure 11 : Opening the Properties Dialog

NOTE: If you have not been granted the `JSVisualizationAdminLicense` license then you will only see the About page. See the section Spotfire Licenses for more details.



Setting the General Properties

The "`General`" Property Page allows the setting of the text that will appear in the title bar of the visualization and whether the visualization should show a legend or not.

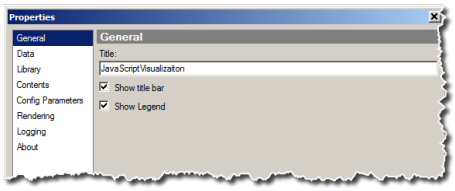


Figure 12 : General Property Page

6. Setting the Data Table Properties

The "Data" Property Page allows the selection of one or more Data Configurations:

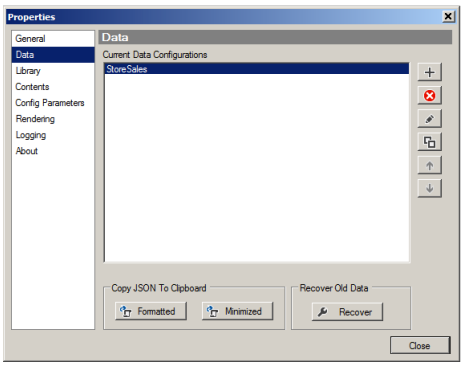


Figure 13 : "Data" Property Page

Each Data Configuration is the combination of the following:

- An in-memory Data Table that exists within the current analysis file
- A filtering scheme to use
- A set of columns from, or custom expressions that refer to that Data Table
- Zero or more columns from the same Data Table to Group the data by
- Settings that control how to use marking, both to and from the visualization
- Settings that control if the data is sent to the JavaScript code as one block or as multiple blocks

Data Configurations are created or edited through their details pages which allow the selection of the above settings:

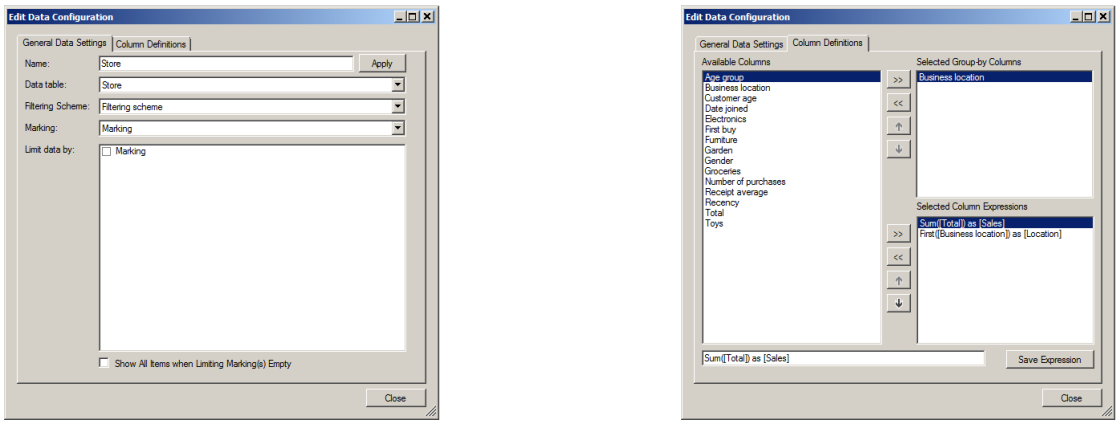


Figure 14 : Data Configuration Editing Pages

The ultimate goal of the Data Configurations is to create the combined JSON string that is passed to the JavaScript code to render the visualization.

The use of multiple Data Configurations allows data from multiple data tables to be passed to a JSViz visualization.

Default Data Configuration

When a JSViz visualization is first created, a default Data Configuration will be created with the following properties:

- Name – the name of the default In-Memory Data Table
- Table – the default In-Memory Data Table
- Filtering Scheme – the default Filtering Scheme for the default Data Table
- Marking – the default Marking for the default Data Table
- Limit Data – no limiting
- Group By - the first categorical column of the default Data Table
- Column Expressions - the first continuous column of the default Data Table
- Paging - disabled

Creating and Editing Data Configurations

The action buttons on the Data Configurations page allow the following actions on the list:

	Add a new Data Configuration
	Delete an existing Data Configuration
	Edit an existing Data Configuration
	Duplicate an existing Data Configuration
	Move the selected Data Configuration up in the list

	Move the selected Data Configuration down in the list

The following sections apply the same, regardless of whether a new Data Configuration is being created or an existing one is being edited.

Editing General Data Settings

This page allows the selection of the data table that will provide the data to the JSViz visualization code and controls how other factors such as Filter Schemes and Marking will affect the data. This process is similar to other built-in Spotfire visualizations.

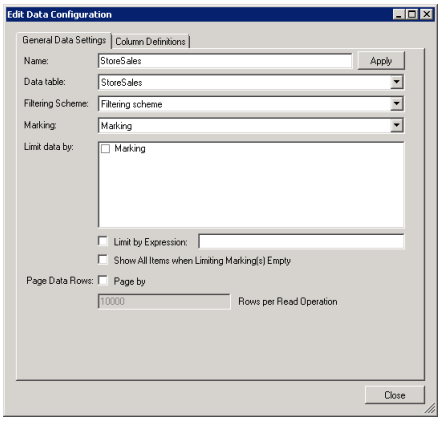


Figure 15 : General Data Settings Property Page

Data Table

This dropdown lists the available in-memory data tables that are eligible to be used with JSViz. Note that in-database sources are not eligible to be used.

Filtering Scheme

A filtering scheme is the set of filtering selections a Spotfire user makes to filter out data. Spotfire allows the use of multiple Filtering Schemes to support advanced visualization use-cases. The dropdown list displays the current Filtering Schemes defined within the current Analysis File.

Marking

This dropdown allows the selection of a Marking set that will be updated when the user Marks items within the visualization.

Note that in order to use this functionality the visualization author must write specific JavaScript code to send the set of marked items back to Spotfire. Not all visualizations will implement this feature, and those that do will vary in how it is implemented.

Limit Data By Marking

This limits the data sent to the JavaScript visualization as a result of other markings within the current DXP file. It functions the same way as other visualizations within Spotfire.

The "Show all items when Limiting Item(s) Empty" option applies to the case where the user has not made any selections in any of the Marking Sets. With this option unchecked, no data will be passed to the visualization. With the option checked, all data items that match the Filtering Scheme will be passed.

Limit By Expression

This allows the limiting of what data is displayed using a Spotfire Expression. Any Spotfire Expression that returns a Boolean value can be used. For example the following Expression will limit data to only those rows where the value in the "Total" column is greater than 500:

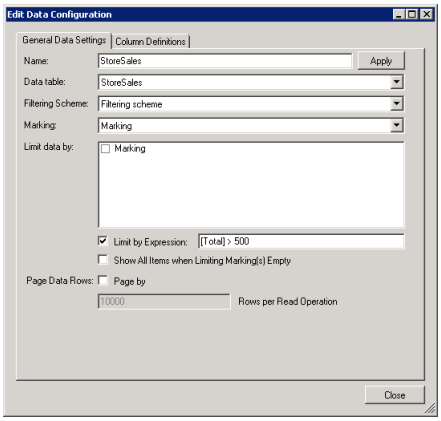


Figure 16 : Example of using Limit Data By Expression

Page Data Rows

This setting controls whether the JSON data will be sent to the JavaScript in one call or will be sent in multiple calls. The library file JSViz.js takes care of re-assembling the data into a single object before calling the `renderCore()` method.

This setting is required when using non-aggregated plots where sending a large amount data in a single call would cause an error inside the browser. The actual limits varies by browser and version.

Editing Column Definitions

This page allows you to choose how data gets passed to the visualization.

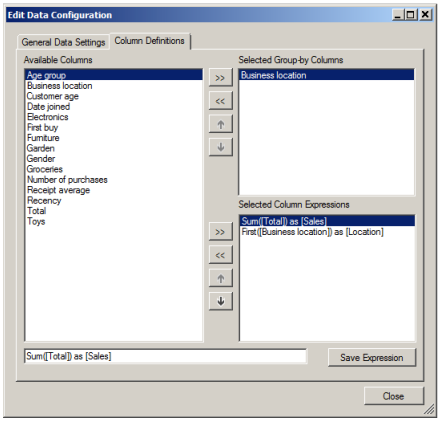


Figure 17 : Column Definitions Property Page

There are three choices to be made:

A. Which columns of data will be passed to the visualization
B. What aggregations, or expressions, will be applied to the data e.g. `Sum`, `Avg` etc.
C. Which columns will be used to group the data for aggregation

Editing Expressions

When adding columns to the Expressions list, you can change the default aggregation by selecting the entry, editing the expansion in the text box, and then selecting Save Expression. This feature can also be used to rename columns using the "`as`" syntax used in Spotfire Custom Expressions.

If the expression is invalid, the Data Engine will reject it and the expression will revert to its previous value.

Previewing JSON data

On the Data Configurations Page, there are two options to export the JSON formatted data that will be passed to the visualization:

Formatted	Copy the JSON data to the clipboard formatted and indented on multiple lines
Minimized	Copy the JSON data to the clipboard minimized to a single line with all spaces and carriage returns removed

Recover Old Data

This button is only used during upgrade of JSViz visualizations from previous versions.

Under normal circumstances JSViz visualizations will be automatically updated to the new Data Configuration model when the visualization is first opened. However when visualizations are on pages that are not visible when the visualization is first opened, there is a possibility that their configuration will not be updated.

In the event of this occurrence, this button can be used to perform a manual update.

7. Managing the JavaScript/CSS/HTML Library

The JSViz library allows the creation of a set of JavaScript, CSS and HTML content items that can then be used within a JSViz visualization. The library is local to an individual DXP file but content items can be exported from one DXP file and imported into another easily and exported content is in a format compatible with Revision Control Systems such as SVN.

The "Library" Property Page allows the manipulation of Content Items:

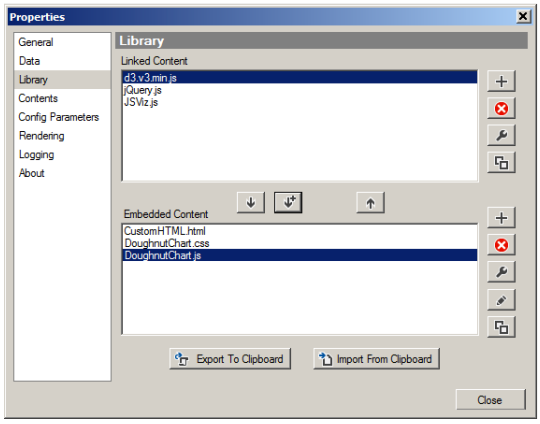


Figure 18 : Library Property Page

Linked Content

These are items that are referenced by their http(s) URL and reside on a Web Server.

The content for these items is pulled from the Web Server when the visualization is displayed, subject to the caching rules of the Web Browser being used. In Spotfire Analyst the Web Browser is Google Chromium whereas in Web Player it is the browser being used by the Spotfire user.

Note: In order for these items to work within a JSViz visualization they must be accessible via their URL from the users Spotfire Analyst or Web Browser.

Embedded Content

These are JavaScript, CSS or HTML items that have been copied entirely into the JSViz library and now reside within the Spotfire DXP file. This allows these items to be used even when the user is not connected to a network or where the original websites are blocked.

Creating and Editing Linked Content

The action buttons on the Library page allow the following actions on Linked Content:

	Add a new Linked Content Item
	Delete an existing Linked Content Item
	Edit the URL, name and type of an existing Linked Content Item
	Duplicate an existing Linked Content Item

The same dialog is used for both creating and editing Linked Content Items.

Linked Content Properties

Enter a name, the URL and Type for the Content Item. Type is either JS for a JavaScript file, CSS for a style sheet file or HTML for an HTML file.

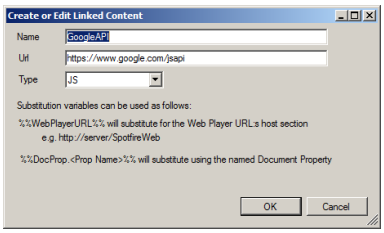


Figure 19 : Create or Edit Linked Content Dialog

Hint: In order for the files to be correctly found when used in both Spotfire Analyst and Spotfire Web Player it is recommended to use the full URL syntax beginning with http(s)://.

Substitution Tokens

In order to allow the URLs to be quickly modified as needed there are two substitution tokens that can be used:

A. `%%DocProp.<Property Name>%%`

This syntax will replace the token with the value of a document property

So if the document property is called "MyBaseURL" has a value of `http://server:port`, then the inclusion string:

```
%%DocProp.MyBaseURL%%/js/test.js
```

will reference the file

```
http://server:port/js/test
```

B. `%%WebPlayerURL%%`

This syntax will replace the token with the base part of the web player URL in which the file will be viewed when opened from the library.

So if the Web Player base URL is `http://server:port/SpotfireWeb`, then the inclusion string

```
%%WebPlayerURL%%/js/test.js
```

will reference the file

```
http://server:port/js/test
```

Creating and Editing Embedded Content

The action buttons on the Library page allow the following actions on Embedded Content:

	Add a new Embedded Content Item
	Delete an existing Embedded Content Item
	Rename an existing Embedded Content Item
	Edit the contents of an existing Embedded Content Item
	Duplicate an existing Linked Content Item

The process of creating a new Linked Content Item, or editing an existing one use the same dialog.

Creating New Embedded Content

New embedded content can be added by either:

- Clicking on the  icon next to the Embedded Content list. Enter a name, and browse to select the file to be embedded. The file type will be automatically set from the file extension if possible.

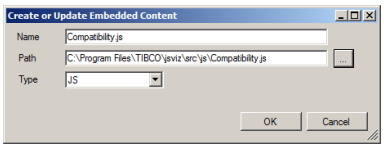


Figure 20 : Edit Inclusion Dialog

- Dragging and dropping a file from Windows Explorer onto the Embedded Content list. Only files with a `.JS`, `.CSS` or `.HTML` extension can be dropped and the name and file type will be automatically set.

Editing Embedded Content

JSViz provides a built-in text editor for editing Embedded Content. The editor uses the Scintilla .NET library and provides color syntax highlighting and useful features such as go to and find.

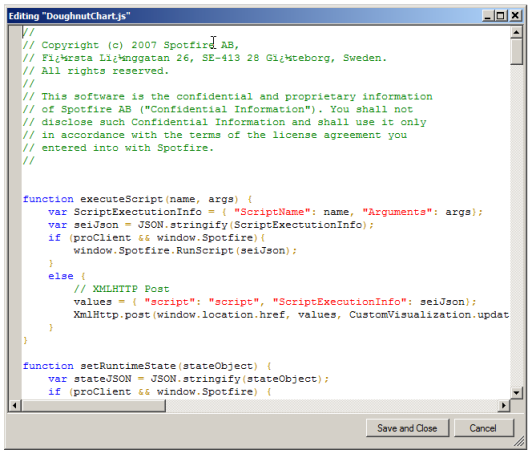


Figure 21 : Editing Embedded Content

Switching content from Linked to Embedded and back

Linked content items can be downloaded and embedded into the DXP file by clicking the down arrow underneath the linked content list. Once downloaded, these items can be reverted back to linked items by clicking the up arrow above the embedded content list.

So items originally added as linked items can be easily toggle between linked and embedded. However items added as embedded items will always remain embedded.

The action buttons on the Library page control switching between Linked and Embedded:

	Switch from Linked Content to Embedded Content. The item retains a pointer to its original URL so it can be switched back to Linked Content if desired.
	Same as above, but also process any CSS `url()` Tags and embed the image content as well. See below.
	Switch back from Embedded Content to Linked Content. This button will be disabled if the item does not have an original URL.

Embedding Image Content

CSS files quite often contain relative references to image tags using CSS image notation like this:

```
body {
  background-image: url("../images/background.gif")
  background-color: #cccccc
}
```

If this CSS files was embedded within JSViz verbatim, then the visualization would not be constructed correctly as the Web Browser would not be able to interpret the relative path for the image.

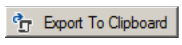
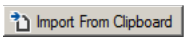
To overcome this issue, JSViz provides the option to read the image file and embed it as a "`data`" resource. This results in the following CSS code:

```
body {
  background-image: url( ... UjIQA7)
  background-color: #cccccc
}
```

This allows the Web Browser to be able to load and use these items.

Library Import and Export

The action buttons at the bottom of the Library page allow the following actions:

	Copy the entire contents of the Library to the Windows Clipboard. The format of the export file is detailed in Appendix E.
	Import the contents of the Windows Clipboard into the Library. After import, content items will be assigned to their respective lists.

Removing a Content Item That Is Being Used

If an attempt is made to remove a Content Item that is being used in a visualization, an error message will be show explaining where the item is being used:

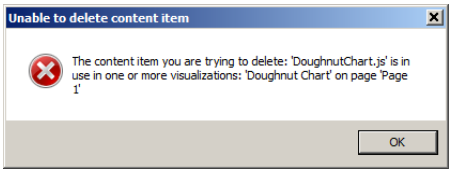


Figure 22 : Error Message when Deleting a Content Item

Note: The message only shows the first visualization that is using the Content Item.

8. Including JS/CSS/HTML Files in a Visualization

The Contents Property Page allows you to choose from the JSViz Library the JavaScript, CSS and HTML files that will be used for individual Visualizations:

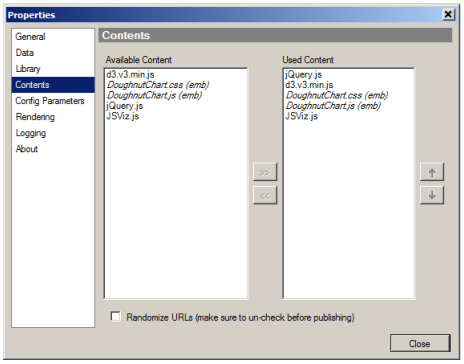


Figure 23 : Contents Property Page

The buttons on the Content Page allow the following actions:

>>	Add the selected Content Item to the Used Content list. This item will be included in the Visualization.
<<	Remove the selected Content Item from the Used Content list. This item will no longer be included in the Visualization.
↑	Move the selected item up the Used Content list.
↓	Move the selected item down the Used Content list.

Items in the Used Content list will be included in the Visualization in the order specified.

Randomizing Inclusions

Normally, when authoring in Spotfire Analyst, the embedded Internet Explorer Control will cache JavaScript and CSS files, so any changes made to Linked Content items will not take effect until Spotfire is restarted or the cache is cleared.

During development and debugging, it is helpful to be able to quickly see any changes as they are made. When the "Randomize URLs" option is checked a parameter "`__id`" with a random value is added to the URL for each JavaScript file. This ensures that the embedded browser control uses the most up-to-date version of each JavaScript file.

```
http://localhost/jsviz/src/js/DonutChart.js
```

will become

```
http://localhost/jsviz/src/js/DonutChart.js?__id=<random integer>
```

It is recommended to turn off this option before publishing a DXP file as it will introduce extra delay during visualization redraws.

9. Using a Custom HTML Page

By default, JSViz automatically generates the HTML for the browser page that will contain the JavaScript visualization within Spotfire. This default HTML content defines the HTML `DIV` tag called "`js_chart`" that will contain the JavaScript visualization.

The default HTML defines the "`js_chart`" `DIV` tag as follows:

```
<div style="position: absolute; width: 100%; height: 100%; top: 0px; left: 0px;"
id="js_chart"></div>
```

A custom HTML can be added either as a Linked or Embedded Content Item. If present, the contents of this file will be used instead of the default HTML.

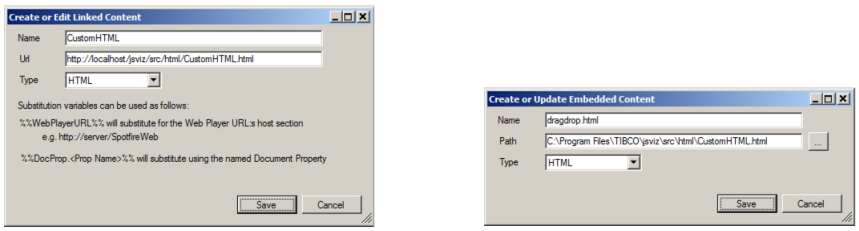


Figure 24 : Creating and Editing a Custom HTML Content Item

Some scenarios for using a Custom HTML page would be:

- To add formatting, content or structure to the visualization canvas
- To display web page content

For example, the following HTML would position the JavaScript visualization within the center cell of a single row HTML table:

```
<table>
    <tr>
        <td></td>
        <td><div id="js_chart"></div></td>
        <td></td>
    </tr>
</table>
```

Sizing and Resizing Considerations

The JSViz sample visualizations assume that they will occupy the entire height and width of the visualization area assigned to the JSViz visualization by Spotfire. This may no longer be true when using a Custom HTML Template as the available visualization area may be further sub-divided.

So it is important to determine the size of the actual `DIV` tag within which the JavaScript visualization will render itself, rather than just using the size of the JavaScript `window` object.

So in the example shown above, the call to determine height and width:

```
var height = window.height;
var width = window.width;
```

Can be replaced by the following code:

```
var td_rect = document.getElementById('chart_table').rows[0].cells[1].getBoundingClientRect ();

var width = td_rect.right - td_rect.left;
var height = td_rect.bottom - td_rect.top;
```

Obviously, this code will only work for the example shown. Determining the size of the `DIV` element will vary for each specific scenario.

10. Using Configuration Data

The "Config Parameters" Property Page allows you to view and edit JSON configuration data that is passed to the visualization. This can be used to control behavior of the visualization such as colors, marker size etc. The actual contents are completely arbitrary and will depend upon the visualization being created.

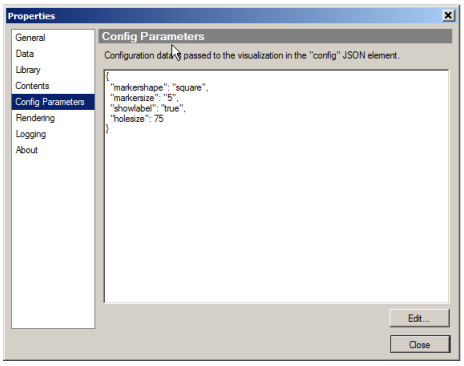


Figure 25 : Config Parameters Property Page

Editing Configuration Data

Clicking on the Edit button opens the built-in text editor for editing Configuration Data. The editor uses the Scintilla .NET library and provides useful features such as go to, find and replace.

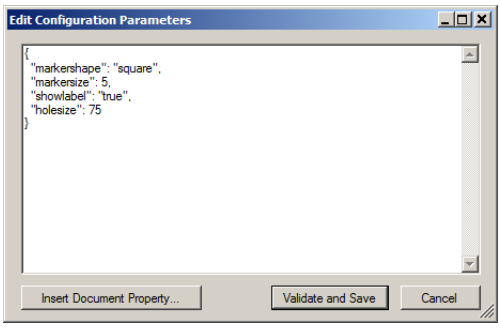


Figure 26 : Editing Configuration Data

Once the configuration data has been edited, clicking the "Validate and Save" button will validate the data using the NewtonSoft JSON library and format the data for display on the Config Parameters Property Page.

If an error occurs during the validation process, an error dialog will display the description of the error. Any errors must be corrected before the updates can be saved.

Typically the error message can be quite long. Usually the root cause of the error can be found at the end or bottom of the message.

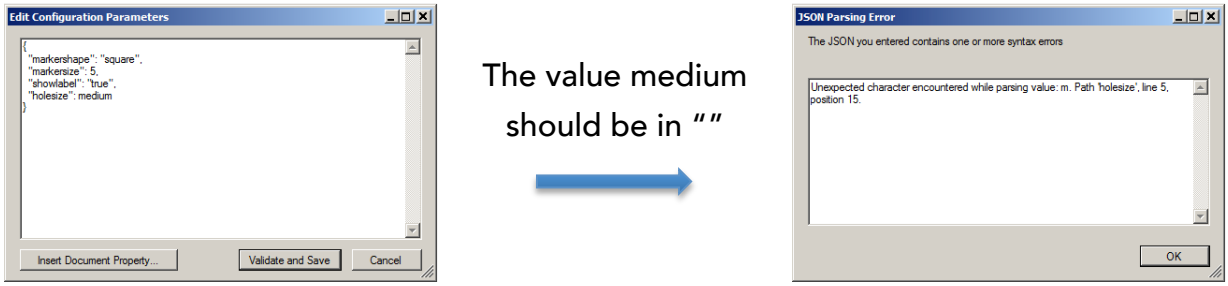


Figure 27 : Example Configuration Data Parsing Error

Using Document Properties

Spotfire Document Properties can be used as part of Configuration Data. The value of the Document Property will be inserted into the Configuration Data and the visualization will be redrawn whenever the value of the Document Property changes.

A placeholder for the Document Property is placed in the Configuration Data using the following syntax:

```
%%DocProp.<Property Name>%%
```

Whenever the JSON data is requested, this placeholder is replaced with the actual value of the Document Property at the specific time.

For example, if the document property called "MarkerSize" has a value of "5", then the configuration fragment:

```
"markersize": %%DocProp.MarkerSize%%
```

This will result in the value 5 being inserted into the Configuration Data JSON stream.

***Note**: Quotes should not be placed around Document Property values, and will be removed during validation if present. When the complete JSON data is requested, the JSON parser will add quotes if needed, based on the data type of the Document Property.*

Inserting Document Properties

Clicking the "Insert Document Property..." button brings up a dialog that lists the Document Properties present in the current Spotfire Document.

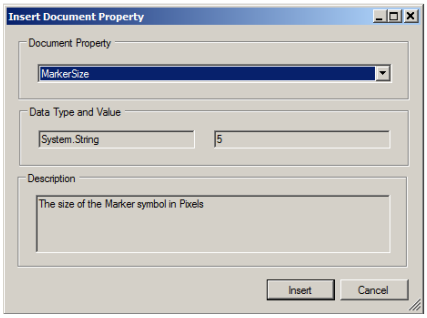


Figure 28 : Insert Document Property Dialog

The dialog displays the Data Type, Description and current value of the selected Document Property. Clicking "Insert" inserts placeholder for the chosen item at the current cursor position or in place of the currently selected text.

Using IronPython Scripting

The JSON configuration data can also be updated from an IronPython Script. This is described in the section "Scripted Configuration".

11. Controlling Rendering

The Rendering Property Page allows control over some of the behavior of JSViz with respect to rendering of visualizations.

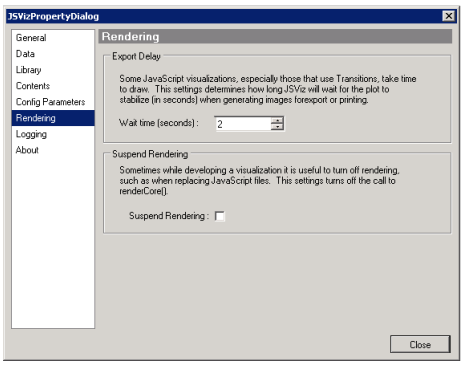


Figure 29 : Rendering Property Page

Export Delay

Many JavaScript visualizations use animations such as Transitions to render the visualization over a short period of time. These visualizations require time to stabilize before taking an image for exporting or printing.

To allow for this, it is possible to specify the number of seconds to wait for the visualization to finish rendering, or stabilize, before taking the snapshot image for printing. A period of 1 or 2 seconds will usually suffice, but longer periods may be required for some visualizations.

Suspend Rendering

During the development cycle it may be necessary to make modifications to the inclusion list or the data configuration that will cause errors to be thrown in the JavaScript code while they are being made.

To prevent this annoying situation, rendering of the visualization can be suspended while such changes are made.

This setting is persisted in the DXP file, so it is important to remember to reset it before saving.

12. Logging

The Logging Property Page allows the enabling of the Log Command which provides JavaScript visualization authors with a means to send log messages to JSViz from within their JavaScript code. These log messages can then be written to a file.

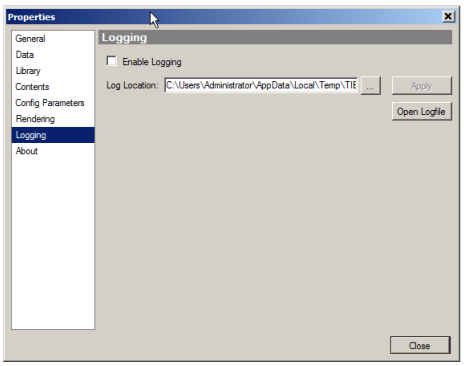


Figure 30 : Logging Property Page

Initial Settings

When the Property Page is first opened, the filename will default to `JavaScriptVisualizationFramework.log` under the Spotfire Temp folder and logging will be disabled.

Enabling Logging

Logging is enabled by checking the "`Enable Logging`" option. The browse button can then be used to select a different filename or location, and the "`Open Logfile`" button can be used to open the chosen file in the windows application associated with the file extension.

Log Format

Each individual call to the Log Command results in the text contents being written to the logfile verbatim on a new line. In the following example the author chose to log the message "`Entering renderCore() ...`":

```
2015-11-10 02:50:18 - [Main Thread] INFO  JavaScriptVisualizationFramework - JavaScriptVisualizationFramework started
2015-11-10 02:50:28 - [Main Thread] DEBUG JavaScriptVisualizationFramework – Entering renderCore() ...
```

For more information on the Log Command and the JavaScript syntax required, see the Appendix B.

NOTE: The settings on this page are NOT persisted to the DXP file by design. You will have to come into this page and set the options each time the DXP file is opened. Also these settings only apply to Spotfire Analyst and have no effect in Spotfire Web Player.

13. About

The About Property Page displays summary information about JSViz such as the Build Version and Build Date along with the Disclaimer information.

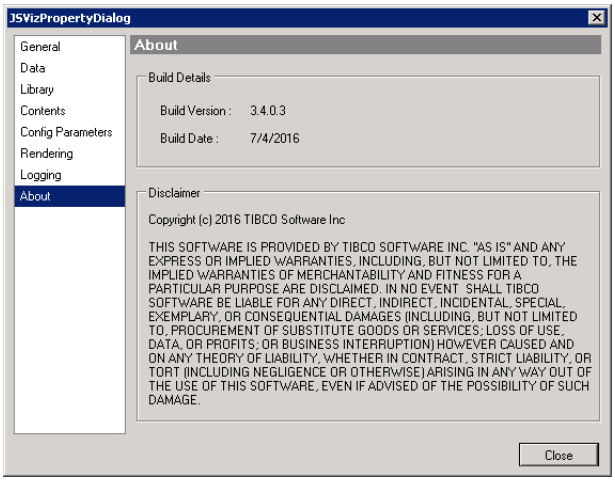


Figure 31 : About Property Page

14. JSViz.js Library File

In order to make the process of creating visualizations easier, the boilerplate code has been centralized into a file called `JSViz.js` which resides in the `lib/JSViz` folder on the Web Server.

This file contains the following items:

- Code required to register the `renderCore()` drawing method at initialization time
- Handlers for Marking and Resizing calls
- Helper functions for logging, running scripts, settings Document Properties etc
- Helper function to re-combine Paged data into a single JSON object

The following sections describe the methods within `JSViz.js` in more detail.

SpotfireLoaded Event Handler

Spotfire 7.5 provides native support for JavaScript based visualizations. JSViz uses this native support and registers a "`render`" function to capture the event that indicates Spotfire has finished initializing and is ready for the visualization to render itself.

This "`render`" function is responsible for ensuring that all required data is either present, or in the case of paged data is downloaded, before calling the main `renderCore()` method. This is accomplished through the use of the jQuery Promise mechanism.

```
//
// The Spotfire 7,5 JavaScript visualization API fires a "SpotfireLoaded"
// event when all the required objects are loaded and initialized.
//
var promises = [];       // Array for asynchronous data loading promises

jQuery ( window ).on ( "SpotfireLoaded", function ()
{
  initMarking ();

  var render = function ()
  {
    promises.length = 0;

    Spotfire.read ( "data", {}, function ( data )
    {
      if ( data )
      {
        var dataObject = JSON.parse(data);

        if (dataObject.pageDataRows)
        {
```

```
            promises.push ( readPagedRows ( dataObject.data,
                                            dataObject.baseTableHints.settingName,
                                            dataObject ) );
          }

          for ( var dataSetting in dataObject.additionalTables )
          {
            if ( dataSetting.data, dataSetting.pageDataRows )
            {
              promises.push ( readPagedRows ( dataSetting.data,
                                              dataSetting.baseTableHints.settingName,
                                              dataObject);
            }
          }

          $.when.apply ( $, promises ).then ( function ()
          {
            setProgress ( false );
            renderCore ( dataObject );
          });
        }
      });
    };

    Spotfire.addEventHandler ( "render", render );

    render();
  });
```

Data Paging Functions

When the "Paged Data Rows" option is selected, data from the Data Engine is sent in blocks of n rows where n is a configurable setting. The helper functions in JSViz.js re-combine that data into a single JSON object before calling the `renderCore()` method.

The `readPagedRows()` function calls the JSViz "`initPaging`" API call passing in a function which will be called successively by JSViz until all the data has been read. This function in turn calls another function `readNextRows()`.

A new jQuery Deferred object is created and passed to the `readNextRows()` function so that it can be resolved once the data read is complete.

The `readPagedRows()` functions then returns the Promise object of the Deferred so that it can be added to the array of Promise objects.

```
function readPagedRows(dataArray, dataSettingName, data)
{
  var deferred = new $.Deferred ();
  var promise  = deferred.promise ();
```

```
  Spotfire.read ( "initPaging", dataSettingName, function ( init )
  {
    if ( init == "OK" )
    {
      setProgress ( true );

      readNextRows ( dataArray, dataSettingName, data, deferred );
    }
  });

  return promise;
}
```

The `readNextRows()` function calls the JSViz "`nextRows`" API call to retrieve a number of rows of data as defined in the JSViz configuration. These rows are then appended to the existing data object until JSViz signifies that the end of the data has been reached, at which point the Deffered object is resolved.

```
function readNextRows ( dataArray, dataSettingName, data, deffered )
{
  Spotfire.read ( "nextRows", dataSettingName, function ( nextRows )
  {
    if(nextRows == "END")
    {
      deferred.resolve ();
    }
    else
    {
      var rows = jQuery.parseJSON ( "[" + nextRows + "]" );
      Array.prototype.push.apply ( dataArray, rows );
      readNextRows ( dataArray, dataSettingName, data, deferred );
    }
  });
}
```

During the process of reading the rows of data, the `setProgress()` method is called to allow insertion of a progress tracker to indicate to the user that the visualization data is being assembled. The default implementation uses a spinner but can be changed to another implementation as required.

```
function setProgress ( bEnabled )
{
     if ( ( window.Spinner ) && ( typeof $('#js_chart').spin == 'function' ) )
     {
         $('#js_chart').spin ( bEnabled ) // Starts or Stops the spinner.
     }
}
```

NOTE: Use of the default spinner requires the inclusion of the `spin.js` and `spinner.js` files. See the doughnutchartswithpaging.dxp sample for more details.

Marking Event Handlers

The `initMarking()` function in `JSViz.js` is called when the visualization is initialized. It registers handlers for basic rectangular marking mouse events.

```
function initMarking ()
{
    jQuery("body").on("mousedown", function(mouseDownEvent)
    {
        var getMarkMode = function(e)
        {
            // shift: add rows
            // control: toggle rows
            // none: replace rows
            if (e.shiftKey)
            {
                return "Add";
            }
            else if (e.ctrlKey)
            {
                return "Toggle";
            }

            return "Replace";
        };

        mouseDownEvent.preventDefault();

        var markMode = getMarkMode(mouseDownEvent);
        //
        // Create initial marking rectangle, will be used if the user only clicks.
        //
        var x = mouseDownEvent.pageX,
        y = mouseDownEvent.pageY,
        width = 1,
        height = 1;

        var $selection = jQuery("<div/>").css({
            'position': 'absolute',
            'border': '1px solid #0a1530',
            'background-color': '#8daddf',
            'opacity': '0.5'
        }).hide().appendTo(this);

        jQuery(this).on("mousemove", function(mouseMoveEvent)
        {
            x = Math.min(mouseDownEvent.pageX, mouseMoveEvent.pageX);
            y = Math.min(mouseDownEvent.pageY, mouseMoveEvent.pageY);
            width = Math.abs(mouseDownEvent.pageX - mouseMoveEvent.pageX);
```

```
            height = Math.abs(mouseDownEvent.pageY - mouseMoveEvent.pageY);

            $selection.css({
                'left': x + 'px',
                'top': y + 'px',
                'width': width + 'px',
                'height': height + 'px'
            });

            $selection.show();
        });

        jQuery(this).on("mouseup", function()
        {
            var rectangle =  {
                'x': x,
                'y': y,
                'width': width,
                'height': height
            };

            if ( typeof(markModel) != 'undefined' )
            {
                markModel ( markMode, rectangle );
            }

            $selection.remove();
            jQuery(this).off("mouseup mousemove");
        });
    });
}
```

When the user completes the marking selection, the `markModel()` function is called if it currently implemented in the users visualization code. The template file `Template.js` includes a stub for the user to implement the `markModel()` function.

```
//
// This method receives the marking mode and marking rectangle coordinates
// on mouse-up when drawing a marking rectangle
//
function markModel(markMode, rectangle)
{
    // Implementation of logic to call markIndices or markIndices2 goes here
}
```

Helper Functions

Within the `JSViz.js` the following helper functions are defined to allow easier access to JSViz methods. The helper functions are written to be independent of whether the code is executing within Spotfire Analyst or Web Player and independent of the version of JSViz.

wait()

This function implements a time delay to allow the visualization to settle. Many JavaScript visualizations use Transitions that animate the visualization over time. When printing or exporting it is therefore necessary to wait for the visualization to settle before capturing an image.

Wrapper Methods

These methods wrap the JSViz API calls described in detail in Appendix B:

- `log()`
- `markIndices()`
- `markIndices2()`
- `setConfig()`
- `setDocumentProperty()`
- `setRuntimeState()`
- `runScript()`

15. JavaScript Environment

This section describes in more detail the environment that exists for the JavaScript charting code to draw in.

Drawing Container

If the user does not include a custom HTML file, JSViz creates a `DIV` element called "`js_chart`". All drawing operations should take place within this `DIV` element. It is defined as follows:

```
<div style="position: absolute; width: 100%; height: 100%; top: 0px; left: 0px;"
id="js_chart"></div>
```

Injected Variables

During the process of adding the JavaScript and CSS files to the embedded browser page, additional snippets of JavaScript code are injected into the page. These code snippets define variables that describe the environment that the visualization code is running in.

proClient

This Boolean variable is defined whenever the JSViz extension is running within the Spotfire Analyst (Professional) thick client. When running within Spotfire Web Player this variable is undefined.

So the following code can be used to conditionally execute code that should only run in Spotfire Analyst:

```
if (proClient) {
    ...
}
```

JSViz

This is an object that is defined in all clients. It contains a "version" sub-object that holds the following values that describe the current version of the JSViz extension:

Property Name	Data Type	Description
major	Integer	The JSViz Major Version. Currently 3.
minor	Integer	The JSViz Minor Version. Currently 3
build	Integer	The internal Build Number.
revision	Integer	The internal Revision Number
date	String	The JSViz Build Date in MM/DD/YYYY format

So the following code can be used to conditionally execute code that should only run in JSViz version 3.x or earlier:

```
if ( JSViz.version.major >= 3 ) {
    ...
}
```

16. IronPython Scripted Setup and Modification

In addition to be able to setup the JavaScript Visualization Extension through its property pages, the settings within the configuration pages can also be accessed and modified via IronPython. This allows scripted setup and modification within Spotfire.

JSVisualizationModel Class

The main class used to control a JSViz visualization is `JSVisualizationModel`. The first step is always to obtain a reference to a `JSVizualizationModel` object, either by instantiating a new one or obtaining a reference to an existing one.

The `JSVisualizationModel` object contains all the Methods and Properties that can be used to configure and manipulate JSViz. The full API Documentation can be found here:

https://s3-us-west-2.amazonaws.com/jsviz/docs/html/index.html

The following sections describe how to use the API to configure JSViz using IronPython.

Required References

Before you can use any JSViz API Objects in an IronPython script, you must setup the required references. The following lines of code are required:

```
import clr
clr.AddReference ( "SpotfirePS.Framework.JSVisualization, Version=1.0.0.0, \
                   Culture=neutral, PublicKeyToken=4d233badaf236513" )
#
# Import the JS Visualization classes
#
from SpotfirePS.Framework.JSVisualization import *
from SpotfirePS.Framework.JSVisualization.Core import *
```

In addition to these imports, you may need other Spotfire imports depending on which inherited Properties or Methods you use:

```
#
# Import Spotfire objects required for manipulating Data Settings
#
from Spotfire.Dxp.Data import PersistentDataView
from Spotfire.Dxp.Data import DataColumn
etc...
```

Adding a new JavaScript Visualization Instance

A new JSViz instance can be added using the `AddNew` method of the `Visuals` object for a given Spotfire `Page`.

The following example shows how to add a new JSViz instance to the current active page:

```
jsviz = Document.ActivePageReference.Visuals.AddNew[JSVisualizationModel]()
jsviz.AutoConfigure()
```

Obtaining a pointer to an existing JavaScript Visualization Instance

One standard method in Spotfire is to loop through the visualizations on a given page and compare the Type Identifier to find a match. If more than one visualization of the required type is present, some further identification method such as matching a name should be used.

Once a reference to the JSViz visualization is obtained, it must be cast to the correct type using the `As[]()` method.

```
for viz in Document.ActivePageReference.Visuals:
    if viz.TypeId == JSVisualizationIdentifiers.JSVisualization:
        break
#
# Cast the visualization object to the correct type
#
jsviz = viz.As[JSVisualizationModel]()
```

Getting or Setting Properties

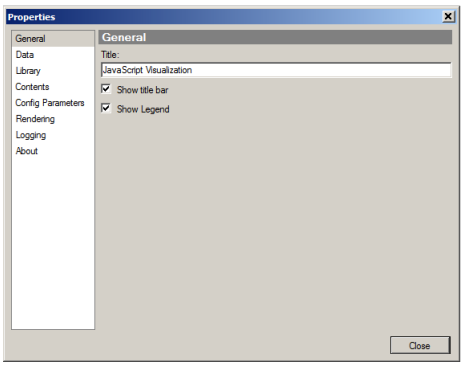
Once a correctly typed handle to a visualization instance has been obtained, a specific property can be read from or written to directly. This example shows getting or setting the Configuration Data for a visualization using the `SettingsParameter2` property.

```
#
# Read the Configuration Data JSON string
#
config = jsviz.SettingsParameters2

#
# Set the Configuration Data from a JSON string
#
jsviz.SettingsParameters2 = config
```

Properties by Dialog Page

The following sections associates the JSViz API Properties with the UI controls in the Properties Dialog. This allows you to figure out which API Methods and Properties you would need to use to mimic a change made in the Configuration Dialog.

General Page



Title

This maps to the `Title` Property of `CustomVisualization` which is inherited from `VisualContent`.

```
# Set the Title
jsviz.Title = "My Chart"
```

Show title bar

This maps to the `ShowTitle` Property of the `Visual` Property of `JSVisualizationModel`.

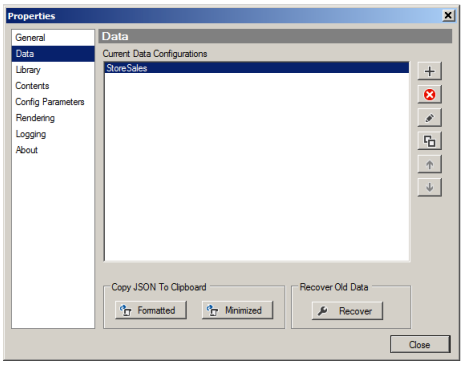
```
# Hide the Title
jsviz.Visual.ShowTitle = False
```

Show Legend

This maps to the `LegendVisible` Property of `JSVisualizationModel`.

```
# Hide the Legend
jsviz.LegendVisible = False
```

Note: The value of the `LegendVisible` property is passed to the visualization in the JSON stream, but is not guaranteed to have any function, depending upon the visualization implementation.



Data Configurations List

The list of Data Configurations maps to the `DataSettings` Property of `JSVisualizationModel`. This is a `DataSettingList` object which is a list of `DataSetting` objects, which are described in more detail in the next section.

```
# Get the list of data configurations
dclist = jsviz.DataSettings

# Empty the list of data configurations
jsviz.DataSettings.Clear ()

# Add a new data configuration
datatable = Document.ActiveDataTableReference
datasettingname = jsviz.GetNextDataSettingName(
datatable.Name )
datasetting = DataSetting ( datasettingname )
datasetting.DataTable = datatable
datasetting.AutoConfigure ()
jsviz.DataSettings.Add ( datasetting )
```

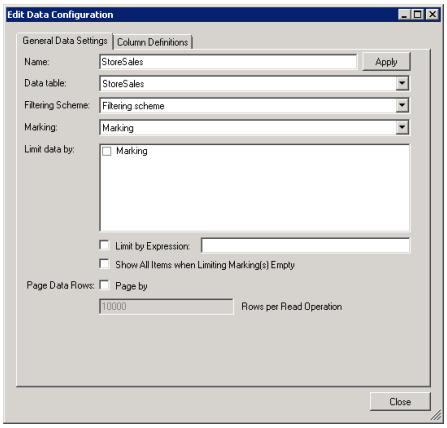
Copy JSON to Clipboard

An *unformatted* copy of the current JSON data being passed to the visualization can be retrieved by calling the `GetDataAsJson()` Method of `JSVisualizationModel`. If required, the data can be formatted and manipulated using a JSON Parser.

```
# Get the unformatted JSON
rawjson = jsviz.GetDataAsJson (False)
```

Data Configuration – General Page

The controls on this page map to Properties of a particular `DataSetting` object. In the examples below it assumed that `ds` is an instance of `DataSetting` object.



Name

```
# Set the Data Setting Name
ds.Name = "My Data Setting"
```

Data table

```
# Set the Data Table
ds.DataTable = Document.ActiveDataTableReference
```

Filtering Scheme

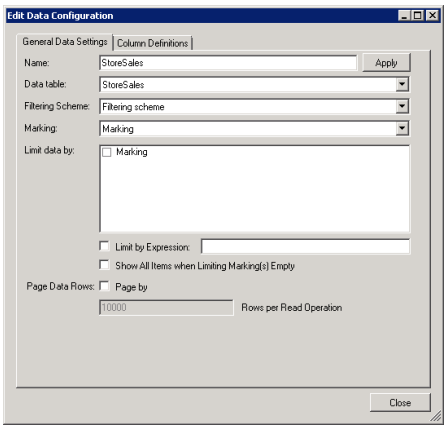
This maps to the `Filtering` Property of a particular `DataSetting` object. This is a Spotfire `DataFilteringSelection` object.

```
# Set the Filtering Scheme
datamanager = ds.Context.GetService[DataManager]()
ds.Filtering = \
datamanager.Filterings.DefaultFilteringReference
```

Marking

This maps to the `Marking` Property of a particular `DataSetting` object. This is a Spotfire `DataMarkingSelection` object.

```
# Set the Marking
datamanager = ds.Context.GetService[DataManager]()
ds.Marking = \
datamanager.Markings.DefaultMarkingReference
```



Limit data by

The checked items in the list map to the `LimitByMarkings` Property of a particular `DataSetting` object. This is a `MarkingReferenceList` object which contains `MarkingReference` objects.

Each `MarkingReference` object wraps a Spotfire `DataMarkingSelection` object.

```
# Clear the Limit data by list
ds.LimitByMarkings.Clear ()

# Limit data by the Default Marking
datamanager = ds.Context.GetService[DataManager]()
defaultmarking = \
datamanager.Markings.DefaultMarkingReference
mr = MarkingReference ( defaultmarking )
ds.LimitByMarkings.Clear ()
ds.LimitByMarkings.Add ( mr )
```

Limit by Expression

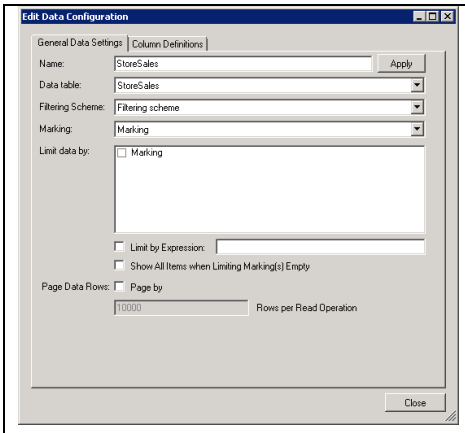
This maps to two properties. The first, `FilterByExpression`, is set to True to enable and the second, `FilterExpression`, is set to the Spotfire Expression.

```
# Limit Data by Expression
ds.FilterByExpression = True
ds.FilterExpression = "[Total] < 5000"
```

Show All Items When Limiting Marking(s) Empty

This maps to the `ShowAllOnEmptyMarkings` Property of a particular `DataSetting` object.

```
# Turn off Show All Items...
ds.ShowAllOnEmptyMarkings = False
```



Page Data Rows

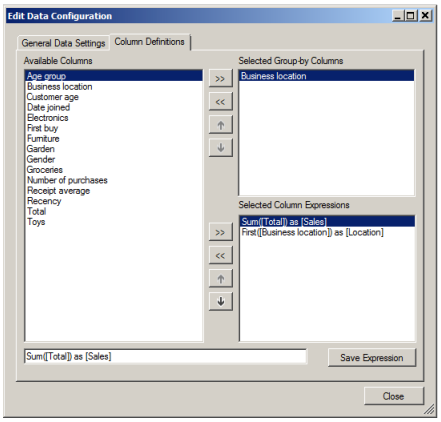
This maps to two properties. The first, `PageDataRows`, is set to True to enable and the second, `PageRowSize`, is set to number of rows per read operation.

```
# Enable Paging
ds.PageDataRows = True
ds.PageRowSize  = 5000
```

Data Configuration – Column Definitions Page

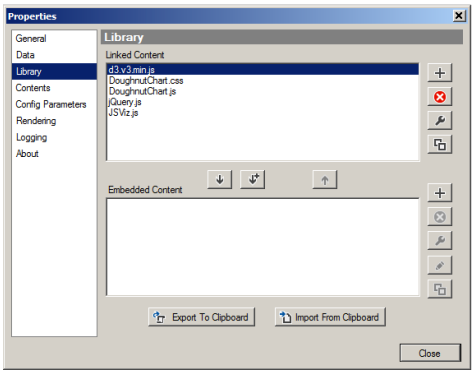
The items in the selected lists on this page define a Spotfire `PersistentDataView` object that forms the `View` Property of a particular `DataSetting` object. The contents of any existing `PersistentDataView` object cannot be modified so a new `PersistentDataView` object must be created and then substituted for the existing one. The `Autoconfigure` Method on a `DataSetting` object creates a default `PersistentDataView` object.

In the examples below it assumed that `ds` is an instance of `DataSetting` object.



The following steps will create the setup shown in the image to the left.

```
# Create a list of Columns to Group By
lstGroupByColumns = List[DataColumn]()
colGroupBy = ds.DataTable.Columns["Business
location"]
lstGroupByColumns.Add ( colGroupBy )

# Create a list of Column Expressions
lstColumnExpressions = List[str]()
lstColumnExpressions.Add ("Sum([Total]) as
[Sales]")
lstColumnExpressions.Add ( "First(["Business
location"]) as ["Location"]" )

# Create a new PersistentDataView object
vw1 = PersistentDataView ( "my view",
lstColumnExpressions, lstGroupByColumns,
ds.DataTable, ds.Filtering )

# Replace the current view with our new one
ds.View = vw1
```



Content Repository

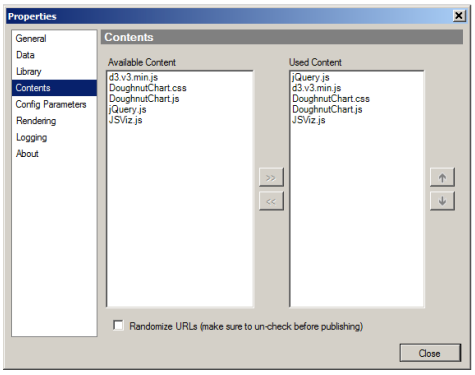
When working with a `JSVizualizationModel` object for the first time in a DXP file, it is necessary to create a new `ContentRepository` object.

```
# Create a new Content Repository (if needed)
cr = Document.CustomNodes. \
       AddNewIfNeeded[ContentRepository]()
```

Creating Content Items

Content Items are created using the `UrlReference` class. They are then added to the Content Repository using a unique name.

```
# Create a Linked Content Item
name = "jQuery.js"
url  = "http://server/path/jQuery.js"

urlReference = UrlReference ( name, \
                              url, \
                              None, \
                              ContentType.JS, \
                              False )
# Add it to the Content Repository
cr[name] = urlReference

# Create an Embedded Content Item
name = "my.js"
url  = "http://server/path/filename.js"
content  = "var i = 0;"

urlReference = UrlReference ( name, \
                              url, \
                              content, \
                              ContentType.JS, \
                              True )
# Add it to the Content Repository
cr[name] = urlReference
```



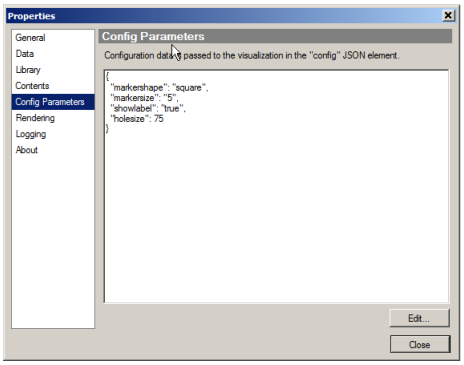
Inclusion List

The `UrlInclusions` Property of the `JSVisualizationModel` object specifies which Content Items in the Content Repository are used to draw the visualization.

New items will be added to the end of the list and Items are presented to the Visualization in the order they appear in the list.

```
# Add a Content Item to the Inclusion List
name = "jQuery.js"
jsvisual.UrlInclusions.Add ( name )

# Remove a Content Item from the Inclusion List
name = "jQuery.js"
jsvisual.UrlInclusions.Remove ( name )
```

Configuration Parameters



Configuration data is a JSON string which can be accessed via the `SettingsParameter2` Property of the `JSVisualizationModel` class.

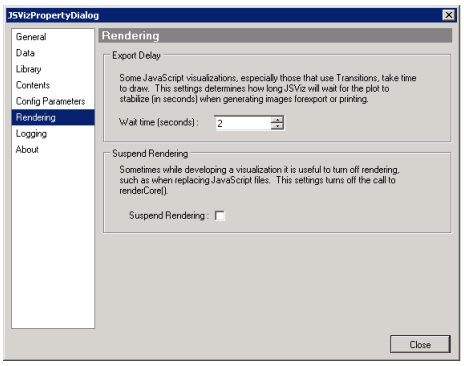
```
#
# Read the Configuration Data JSON string
#
config = jsviz.SettingsParameters2

#
# Set the Configuration Data from a JSON string
#
jsviz.SettingsParameters2 = config
```

Document Properties can be inserted as the value of a parameter using the following syntax:

```
%%DocProp.<Property Name>%%
```

The value should be in quotes if it is a string.

Rendering



Export Delay

This maps onto the `WaitForRendering` Property of the `JSVisualizationModel` class.

```
# Set the Export Delay to 5 seconds
jsviz.WaitForRendering = 5
```

Suspend Rendering

This maps onto the `SuppressRendering` Property of the `JSVisualizationModel` class.

```
# Suspend Rendering
jsviz.SuppressRendering = True

# Resume Rendering
jsviz.SuppressRendering = False
```

17. Implementation Details

This section describes in greater details the internal workings of JSViz, so as to allow a better understanding of its various features and limitations. The implementation is different for the Spotfire Desktop Client and the Spotfire Web Player.

Desktop Client

In the Desktop Client, a Chromium Web Browser control is used as the container for the JavaScript files containing the display code. The actual files JavaScript Files can either be hosted on a Web Server instance or embedded within the DXP file.

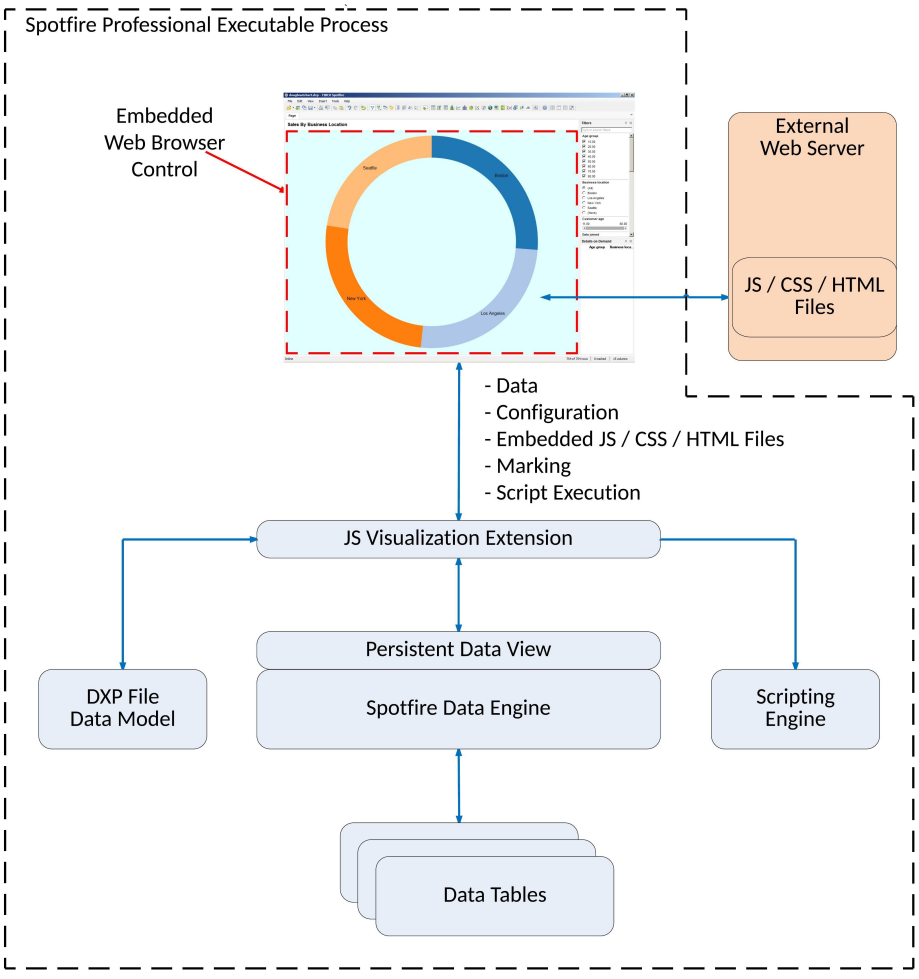


Figure 32 : Desktop Client Implementation

The sequence of events is as follows:

1. The Spotfire Analyst client starts up and loads the JSViz extension.
   As the extension is part of the memory space of the Spotfire Analyst executable, it has access to the internal components of Spotfire such as the Data Engine, the Scripting Engine etc.
2. A visualization that uses the JSViz extension is created or a DXP file is loaded containing an existing instance.
3. The view for the JSViz visualization is created. This is an instance of a Chromium Web Browser control.
4. The JSViz extension instructs the Web Browser control to load the JavaScript, CSS and HTML files that make up the JSViz visualization. These can either be loaded from an external Web Server via URL or directly from JSViz as embedded content.
5. When the loading of the content files is completed, the Spotfire JavaScript API fires a "`SpotfireLoaded`" event. This event is captured within the visualization code and used to register the main "`renderCore()`"drawing method. Marking and Resizing handlers are also registered.
6. Whenever the user changes a marking or filter that affects the underlying data for the visualization the Spotfire JavaScript API calls the "`renderCore()`"drawing method to render the JavaScript visualization. Also if Marking or Resizing occurs, the required handlers are called.

Whenever the user performs an action that issues a command to JSViz, such as marking some data points in the visualization or executing a script, these events are sent to the Spotfire JavaScript API which routes them to the JSViz extension.

Web Player Client

When deployed within the Web Player, the JavaScript or HTML visualization code is hosted inside an IFrame created by the Web Player framework. Once again the actual files can either be hosted on a Web Server instance or embedded within the DXP file

Unlike other visualizations in the Web Player, which are rendered as image files, the JavaScript drawing code is executed within the users Web Browser.

Whenever the framework detects that a change has been made to the underlying data, for instance due to marking or filtering, it refreshes the contents of the IFrame.

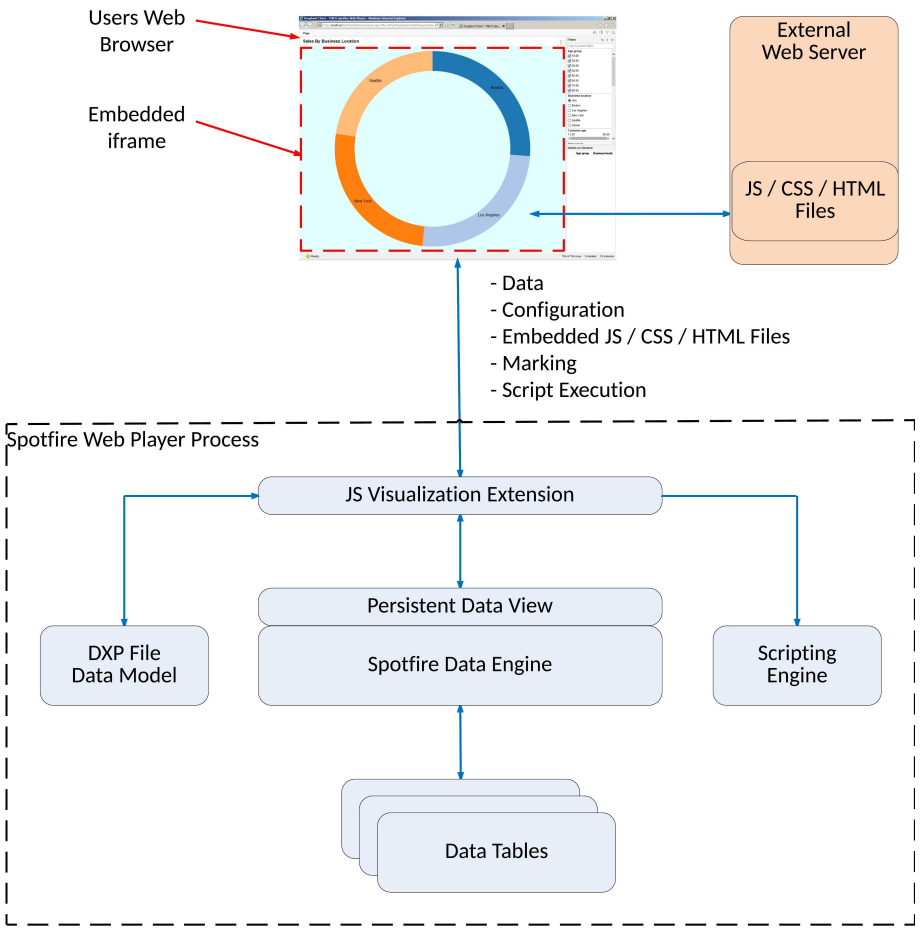


Figure 33 : Web Player Client Implementation

The sequence of events is as follows:

1. The Web Player starts up and loads the JSViz extension. As the extension is part of the memory space of the Spotfire Web Player executable, it has access to the internal components of Spotfire such as the Data Engine, the Scripting Engine etc.
2. The user opens a DXP file containing a visualization that uses the JSViz extension.
3. The Web Player server creates an IFrame on the HTML page that will host the JSViz instance.
4. The IFrame requests its content from the Web Player server which routes the request through to the JSViz extension. The extension returns the body HTML for the visualization page to the user's browser.
   Within the HTML will either be tags referring to the location of the JavaScript and CSS files that make up the visualization or the actual contents of the files depending upon whether the content is linked or embedded.
5. When the loading of the content files is completed, the Spotfire JavaScript API fires a "`SpotfireLoaded`" event. This event is captured within the visualization code and used to register the main "`renderCore()`"drawing method. Marking and Resizing handlers are also registered.
6. Whenever the user changes a marking or filter that affects the underlying data for the visualization the Spotfire JavaScript API calls the "`renderCore()`"drawing method to render the JavaScript visualization. Also if Marking or Resizing occurs, the required handlers are called.

Whenever the user performs an action that issues a command to JSViz, such as marking some data points in the visualization or executing a script, these events are sent to the Web Player via the Spotfire JavaScript API which routes them to the JSViz extension.

Appendix A – JSViz Data, JSON Structure and Contents

The JSON payload passed to visualizations carries the following information:

- Column Names
- Hint Information (# rows total, visible and marked)
- Row and Column Data including Marking
- Runtime State Information (unique to each visualization)
- Configuration Data
- Legend Visibility
- Spotfire User Name

Notice that the data does not contain any information about the axis, scales etc. This information is coded into the HTML or JavaScript page. Only standard Spotfire visualization information reflecting filtering, marking and brush link information is passed on to the JavaScript visualization.

Multiple Data Tables

The original version of JSViz only allowed passing data for a single Data Table. In order to maintain backward compatibility, the JSON format continues to pass data for the Primary Data Table using the previous format, with the additional data tables passed in a new JSON node called "`additionalTables`".

This new node will be ignored by existing visualizations, guaranteeing backwards compatibility but will be able to be used by newer visualizations to work with multiple Data Tables.

Column Names – the "columns" node

This section contains the names of the columns of the Primary Data Table as configured on the data configuration page.

```
{
  "columns":[
    "Name1",
    "Name2",
    ...
  ],
  ...
}
```

Note: the column names appear in the same order as the row and column data.

Hint Information – the "baseTableHints" node

This information is based on the summary provided in the bottom right hand corner of Spotfire Analyst:

- `rows` – the total number of rows in the Data Table
- `visible` – the number of rows visible after Filtering
- `marked` – the number of rows current marked
- `tableName` – the name of the Data Table
- `settingName` – the name of the Data Configuration that created this JSON

```
{
  ...
  "baseTableHints":{
    "rows":999,
    "visible":723,
    "marked":32,
    "tableName":"table1",
    "settingName":"settings1"
  },
  ...
}
```

Primary Data Table Row and Column data – the "data" node

This node contains the data to be visualized, however the exact format will depend on how the Data Configuration is setup.

The "data" node is an array containing an entry for each row of data. When visualizing aggregated data, there will be a row of data for each item in the aggregation group. When visualizing raw or un-aggregated data there will be a row of data for each Visible row in the Data Table.

```
{
  ...
  "data":[
    {
      "items":[
        "textvalue",
        numericvalue
        ...
      ],
```

```
        "hints":{
          "marked": true,
          "index": 0
        }
      },
      {
        "items":[
          "textvalue",
          numericvalue
          ...
        ],
        "hints":{
          "index": 1
        }
      },
      ...
    ],
    ...
  }
```

Within each row of data there are two child nodes:

- `items` – an array containing the name and value for each Column in the dataset.
- `hints` – this contains a flag to indicate if this row of data is Marked within Spotfire, along with the index or order of the item in the dataset

Additional Data Tables – the "additionalTables" node

For each additional table beyond the Primary Data Table setup in the Data Configuration page, a JSON node consisting of "`columns`", "`baseTableHints`" and "`data`" nodes is added under the "`additionalTables`" node.

```
"additionalTables": [
  {
    "columns": [
      ...
    ]
    "baseTableHints": {
      ...
      "tableName": "table2",
      "settingName": "settings2"
    },
    "data": [
      ...
```

```
    ]
  },
  {
    "columns": [
      ...
    ]
    "baseTableHints": {
      ...
      "tableName": "table3",
      "settingName": "settings3"
    },
    "data": [
      ...
    ]
  },
  ...
],
```

This format allows backwards compatibility with earlier JSViz releases that were only able to handle data from a single Data Table.

Runtime State Information – the "runtime" node

The Web Player is a stateless environment. So any action a user takes with a visualization that is not passed into the visualization from Spotfire, such as expanding or collapsing a hierarchical tree or re-arranging items in a Sankey Chart would not be re-drawn correctly the next time the page is refreshed.

To remedy these situations, the Runtime State interface allows a control to persist its current visual state. The contents of this section can be any JSON formatted data.

```
{
  ...
  "runtime":{ ... },
  ...
}
```

The actual format of the data depends entirely on the visualization being used. Typically most JavaScript libraries provide a JSON interface specifically to persist and restore the runtime state of a control.

This data is not persisted into the model so the state of the control will not be stored beyond the lifetime of the DXP file being open. If this behavior is required, it can be accomplished through the use of write-back scripts and information links.

Configuration Data – the "config" node

JSViz allows the storing of configuration properties that can be passed to the JS drawing code to change the look and feel of the visualization. Some examples might be marker shape, size etc.

The data can be any JSON formatted data. The entire configuration block is passed to the drawing code whenever data is requested. The individual elements can then be referenced from within the actual JS drawing code.

The value of Spotfire Document Properties can also be passed in configuration data. See the section titled "Page #6: Configuration Data" for more details.

Configuration data can contain the value of Document Properties and can also be manipulated using IronPython scripting.

Drawing Mode – the "static" node

This is a simple Boolean flag that allows the visualization to draw itself differently depending upon whether it drawing for an interactive user session or during a static export or print operation.

For example, if the value is True then the visualization could choose to not use Transitions when drawing the visualization as there is no human to see the effect.

Legend Visibility – the "legend" node

This is a simple Boolean flag that allows the visualization to respect the value of the "Show/Hide Controls" button on the visualization control bar.

```
{
  ...
  "legend": true,
  ...
}
```

If the value is True then the legend should be shown, otherwise the legend should be hidden.

Spotfire User Name – the "user" node

This node allows the current user name within Spotfire to be passed to the visualization.

```
{
  ...
  "user": "spotfireadmin",
```

```
    ...
  }
```

Rendering Delay – the "wait" node

This indicates the amount of time JSViz should wait for the visualization to stabilize before returning from a render call. Many JavaScript visualization libraries use Transitions when drawing so this delay provides a wait of allowing the Transitions to complete before proceeding.

CSS Data – the "style" node

This node contains the CSS information passed from Spotfire. It can be used to make a visualization conform to the styling being used by other visualizations and by the Spotfire application itself.

Example Output

The following data is from the "doughnutchart" sample export as Formatted JSON:


```
{
  "columns": [
    "Sales",
    "Location"
  ],
  "baseTableHints": {
    "rows": 754,
    "visible": 754,
    "marked": 0,
    "tableName": "Store",
    "settingName": "Store"
  },
  "data": [
    {
      "items": [
        1249558,
        "Boston"
      ],
      "hints": {
        "index": 0
      }
    },
    {
      "items": [
        1213626,
        "Los Angeles"
      ],
      "hints": {
        "index": 1
      }
    },
    {
      "items": [
        1231992,
```

```
        "New York"
      ],
      "hints": {
        "index": 2
      }
    },
    {
      "items": [
        1074072,
        "Seattle"
      ],
      "hints": {
        "index": 3
      }
    }
  ],
  "additionalTables": [],
  "user": "spotfireadmin",
  "static": false,
  "runtime": {},
  "legend": true,
  "config": {
    "markershape": "square",
    "markersize": "5",
    "showlabel": "true",
    "holesize": 75
  },
  "style": "font-family:\"Arial\",sans-serif;font-size:11px;font-style:Normal;font-
weight:Normal;color:#000000;background-color:transparent;border-style:None;border-top-
color:#FFFFFF;border-right-color:#FFFFFF;border-bottom-color:#FFFFFF;border-left-
color:#FFFFFF;border-top-width:0px;border-right-width:0px;border-bottom-
width:0px;border-left-width:0px;border-top-left-radius:0px;border-top-right-
radius:0px;border-bottom-right-radius:0px;border-bottom-left-radius:0px;padding-
top:0px;padding-bottom:0px;padding-left:0px;padding-right:0px;margin-top:0px;margin-
bottom:0px;margin-left:0px;margin-right:0px;"
}
```

Appendix B – JSViz Commands, JSON Structure and Contents

There are a number of interfaces defined that allow the JavaScript visualization to pass data to TIBCO Spotfire. These interfaces allow the following actions:

- Mark data in Spotfire
- Run a Spotfire IronPython Script
- Update the configuration data for the JavaScript visualization
- Update the runtime configuration data for the JavaScript visualization
- Update the value of a Spotfire Document Property
- Log an event

These commands are executed via the Spotfire JavaScript API by calling the "`modify()`" method on the "`Spotfire`" object.

MarkIndices

This interface allows marking or selection activities in the visualization to update a Marking set on the default Data Configuration in JSViz. The Marking set is configured on the General Data Settings tab for the Data Configuration.

The marking data must follow the `MarkingData` JSON contract, which consists of the following parameters:

- Command name: "`MarkingData`"
- "`markMode`": specifies how the set of points passed will affect the current marking:
  - `Replace` : Replace the current Marking Set with this set of points
  - `Add` : Add this set of points to the current Marking Set
  - `Toggle` : Invert the current Marking Set. Marked points become Unmarked and Unmarked points become Marked
- "`indexSet`" : The indices of items to mark, as passed in the `hints.index` objects in the JSON data stream

JSON:

```
{
  "MarkingData":
  {
    "markMode":"Replace",
    "indexSet":[
      1,
      2,
      3
```

```
    ]
  }
}
```

Example code:

```
markData.markMode = markMode;
markData.indexSet = indicesToMark;

Spotfire.Modify ( "mark", markData );
```

MarkIndices2

This interface allows marking or selection activities in the visualization to update a Marking set on one or more Data Configurations in JSViz. The Marking set is configured on the General Data Settings tab for each Data Configuration.

The marking data must follow the `MarkingData2` JSON contract, which consists of the following parameters:

- Command name: "`MarkingData2`"
- "`markMode`": specifies how the set of points passed will affect the current marking:
  - `Replace` : Replace the current Marking Set with this set of points
  - `Add` : Add this set of points to the current Marking Set
  - `Toggle` : Invert the current Marking Set. Marked points become Unmarked and Unmarked points become Marked
- "`indexSetDict`" : A dictionary object that contains an index set of items to mark for each Data Configuration

JSON:

```
{
  "MarkingData2":
  {
    "markMode":"Replace",
    "indexSetDict": [{"Key":"dataConfig1", "Value":[12,23,105]},
                     {"Key":"dataConfig2", "Value":[43,105,229]}
  }
}
```

Example code:

```
dataConfig1 = "dataConfig1";
indices1    = [12,23,105];
```

```
dataConfig2 = "dataConfig2";
indices2    = [43,105,229];

markData.markMode = markMode;
markData.indexSetDict = [{"Key":dataConfig1, "Value":indices1},
                         {"Key":dataConfig2, "Value":indices2}]

Spotfire.Modify ( "mark2", markData );
```

RunScript

This interface allows an action or event within the visualization to execute an existing Spotfire IronPython Script, passing in the required parameters.

The `ScriptExecutionInfo` data contract specifies the name of the Script to be executed along with any parameters:

JSON:

```
{
    "ScriptExecutionInfo":{
        "ScriptName":"name",
        "Arguments":[
            { "Key": "key1", "Value": "value1" },
            { "Key": "key2", "Value": "value2" }
        ]
    }
}
```

Example code:

```
var ScriptExecutionInfo = { "ScriptName": name, "Arguments": args };

Spotfire.Modify ( "script", ScriptExecutionInfo );
```

SetConfiguration

This interface allows the configuration data for the current visualization to be updated. Configuration data is any data that controls the appearance or behavior of a visualization. It is persisted in the DXP file. Examples might be marker size, line colors and thicknesses etc.

As the configuration data format will be unique to a particular JavaScript visualization, the interface allows any JSON formatted data to be passed.

It is important to remember that Spotfire overwrites the current configuration data in its entirety with the data sent via this method call. So it is essential to always send the entire configuration object with every call.

Example code:

```
var configObject = { "linecolor": "blue", "linesize": 4, ... };

Spotfire.Modify ( "config", configObject );
```

SetRuntimeState

This interface allows runtime state data for the current visualization to be updated. Runtime data is any data such as current selection information that is valid only for the duration of a users session. It is not persisted in the DXP file.

As the runtime data format will be unique to a particular JavaScript visualization, the interface allows any JSON formatted data to be passed.

In the following example a single item, the selected Node ID, is stored. Its value will then be passed back each time data is sent to the visualization.

Example code:

```
var RuntimeStateInfo = { "selectedNode": nodeID, "nodeState": nodeState, ... };

Spotfire.Modify ( "runtime", RuntimeStateInfo );
```

SetDocumentProperty

This interface allows the JavaScript visualization to update the value of an existing Document Property in the hosting DXP file. If the Document Property does not exist, an exception will be thrown.

In the following example the name of the Document Property to be set is in the variable *name* and the value to be set is in the variable *value*.

Example code:

```
var DocumentPropertyInfo = { "PropertyName": name, "PropertyValue": value };

Spotfire.Modify ( "documentproperty", DocumentPropertyInfo );
```

Log

This interface allows the JavaScript visualization authors to send log messages to Spotfire.

Spotfire can be configured to write the log data to a file. See the section titled "Page #8: Logging" for more details.

In the following example a simple log message is sent to indicate that rendering has begun. When running in Spotfire Analyst the message will be sent to JSViz, but when running in Spotfire Web Player the message will appear in the Web Browser Console.

Example code:

```
...

log ( "Entering renderCore..." );

...

function log ( message )
{
    if ( typeof(JSViz) != 'undefined' ){
        // Spotfire Analyst:
        Spotfire.modify ( "log", message );
    }
    else if ( window.console ){
        // Spotfire Web Player:
        console.log ( message );
    }
}
```

Appendix C - Troubleshooting Guide

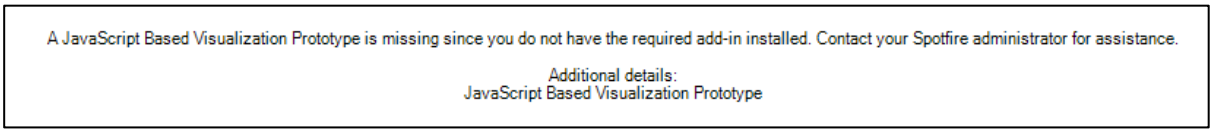
I don't see the option to insert a JavaScript Visualization

Check that the following steps have been taken:

A. Your system administrator has installed the JSViz extension on the Spotfire Server that you connect to.
B. You have accepted the updates (you will need to restart Spotfire if it is already running)
C. Your System Administrator has licensed you to use JSViz.

The system administrator must also update the Web Player deployment before you can use files from the library containing JSViz Visualizations.

I see a message about a missing visualization type when opening an existing DXP file



See the previous item.

My custom JavaScript Visualization doesn't display in Spotfire Analyst – I just see a blank area

Step through each of the content items that make up the visualization to make sure they are setup correctly. If they are linked content items you can copy the URL and make sure that you can open the item in your Web Browser.

Lastly you can use the Chromium Web Browser's built-in debug tools to identify and correct any problems. See the section "Debugging in Spotfire Analyst" in Chapter 4.

My custom visualization displays OK – but my data doesn't show up

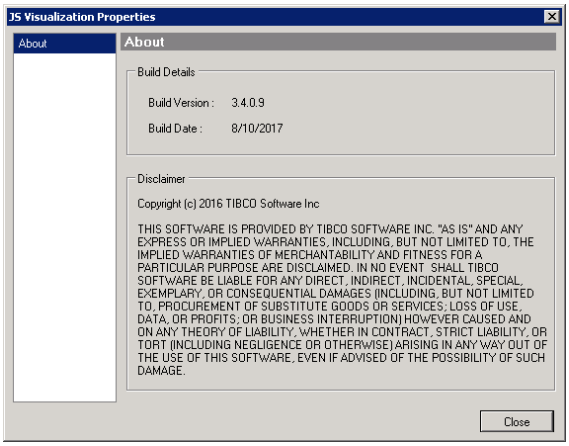
Use the Chromium Web Browser's built-in debug tools to validate that your "`renderCore()`" method is being called correctly with the expected data format. See the section "Debugging in Spotfire Analyst" in Chapter 4.

My JSON request works in the Tester and in the Web Player but not in Spotfire Analyst

Within Spotfire Analyst, JSViz uses an embedded instance of the Chromium Web Browser Control. This control is configured to prevent some behavior that could lead to security issues.

Within the Tester and within the Web Player, the JavaScript code is executed within a regular Web Browser instance. There are limitations on Cross Domain Requests, but these are well documented.

I can't configure a JSViz, the Configuration Dialog is missing everything apart from the About page



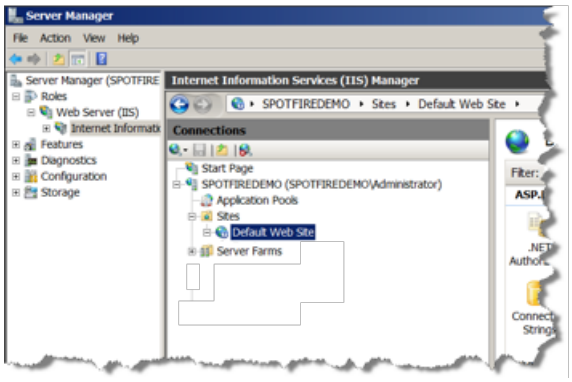
This means that you have not been granted the `JSVisualizationAdminLicense` license.

Appendix D – Web Server Setup

The following steps explain how to setup a site under the IIS default Web Site. This can then be used to store HTML, CSS and JS files for use by JSViz.

Note: This setup is not required and is provided as a courtesy.

Open up the IIS Manager window within the Server Manager console.



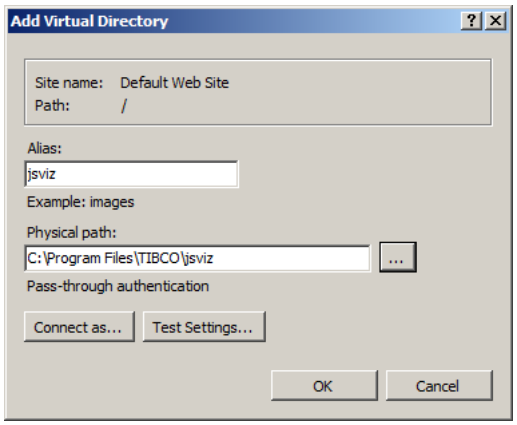
Expand the IIS tree until you see the item "Default Web Site".

Right-click on the "Default Web Site" entry and select "Add Virtual Directory".

Enter the Alias as "`jsviz`".

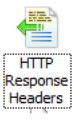
Use the Browse button (…) to navigate to the folder "`C:\Program Files\TIBCO`" and create a sub-folder called "`jsviz`".

Select this directory and click OK. The dialog should now look as shown.



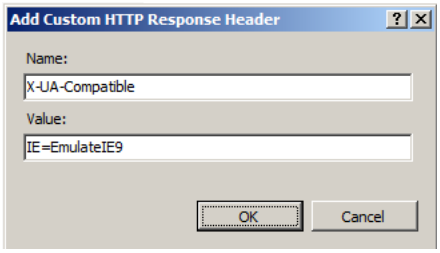
Click OK to create the Virtual Directory.

Click on the newly created "`jsviz`" entry and double click on the "HTTP Response Headers" entry.



Add a new entry as follows:

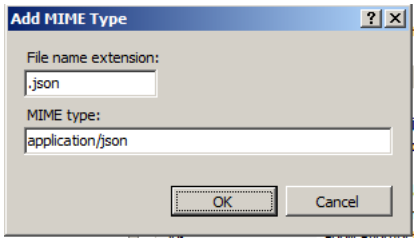
Name: X-UA-Compatible
Value: IE=EmulateIE9



Go back to the "`jsviz`" entry and double click on the "Mime Types" entry.

Add a new entry as follows:

File name extension: .json
MIME type: application/json



Close the Server Manager window.

Sample Files deployment

The sample JavaScript and CSS files that are distributed with the visualization plugin can now be placed in the folder created above.

Copy the files from:

"`For Web Server`"

To:

"`C:\Program Files\TIBCO\jsviz\`"

Validation

Confirm successful setup by opening the Doughnut Chart sample provided in a web browser:

```
http://<hostname>/jsviz/testing/Tester-DoughnutChart.html
```

Where <hostname> is either the name of the machine running IIS, the IP Address of that machine or `localhost` if you are running the browser on the actual machine.

Appendix E – Import / Export Format

The export / import functionality on the Library Property Page uses the following XML-like format.

Linked JS Files

These consist of a single line `<script>` tag containing a name for the item and the location at which the content can be found:

```
<script name="..." src="..." />
```

Embedded JS Files

These are represented by a multi-line `<script>` tag containing attributes for the name of the item and the location from which the content was originally uploaded,, with the actual JS content being located before the end `</script>` tag:

```
<script name="..." src="..." >
...
</script>
```

Note that the JS content is html encoded. It will be decoded using the .net `WebUtility.HtmlDecode()` function when it is imported.

Linked CSS Files

These are represented by a single line `<style>` tag containing attributes for the name of the item and the location at which the content can be found:

```
<style name="..." src="..." />
```

Embedded CSS Files

These are represented by a multi-line `<style>` tag containing attributes for the name of the item and the location from which the content was originally uploaded, with the actual CSS content being located before the end `</style>` tag:

```
<style name="..." src="..." >
...
</style>
```

Note that the CSS content is html encoded. It will be decoded using the .net `WebUtility.HtmlDecode()` function when it is imported.

Linked HTML Files

These consist of a single line `<basehtml>` tag containing a name for the item and the location at which the content can be found:

```
<basehtml name="..." src="..." />
```

Embedded HTML Files

These are represented by a multi-line `<basehtml>` tag containing attributes for the name of the item and the location from which the content was originally uploaded, with the actual HTML content being located before the end `</basehtml>` tag:

```
<basehtml name="..." src="..." >
...
</basehtml>
```

Note that the HTML content is html encoded. It will be decoded using the .net `WebUtility.HtmlDecode()` function when it is imported.