Building Web Widgets for Industrial Graphics

Rubén Rueda
1.0 / 03-05-2023

AVEVA Group Limited
High Cross, Madingley Road
Cambridge CB3 0HB, UK
Tel +44 (0)1223 556655
Fax +44 (0)1223 556666

aveva.com

Index

1. Introduction

1.1. What is a Web Widget?

A web widget is a small web component composed by self-contained code blocks that provides on-screen user interface elements. We can use web widgets to extend the functionality of AVEVA System Platform. For example, we can get access to the device camera, microphone, screen sharing, etc, through the *MediaDevices Web API*, we can leverage *CSS animations*, render 3D models with WebGL (you can use *three.js* library to make it easier) and many other web technologies, all from within System Platform.

1.2. Why use Web Widgets?

Web widgets are a perfect fit for OMI Web and makes moving our application to the cloud seamlessly since they are fully supported in cloud Unified Operations Centre. It allows users to utilize controls, libraries, and frameworks and functionalities that are not available in the local environment.

1.3. Objective

Upon completion of the steps detailed in this document, you will be able to integrate an interactive chart built with a web library to AVEVA System Platform.

1.4. Tools Needed

The tools needed to start building web widgets are:

- AVEVA System Platform
- Text editor (preferably Virtual Studio Code)
- Package tools

2. Web Widget Code

2.1. Web Widget Fundamentals

The three main web programming languages used in developing web widgets (or any web component or application) are *HTML*, *CSS* and *JavaScript*. HTML provides the basic structure of the code. CSS provides stylistic enhancements to the code. JavaScript is used to control the behaviours of the different elements in the code.

Frameworks are tools that provides ready-made components or solutions that are customized in order to speed up development. Popular web frameworks are *Angular*, *React*, *Vue* and *jQuery*.

For Enterprise Visualization we often want to show data in different kind of charts. For this, we can use web charting libraries like *HighCharts*, *AmCharts*, *AnyChart*, *Plotly*, *D3* and many others.

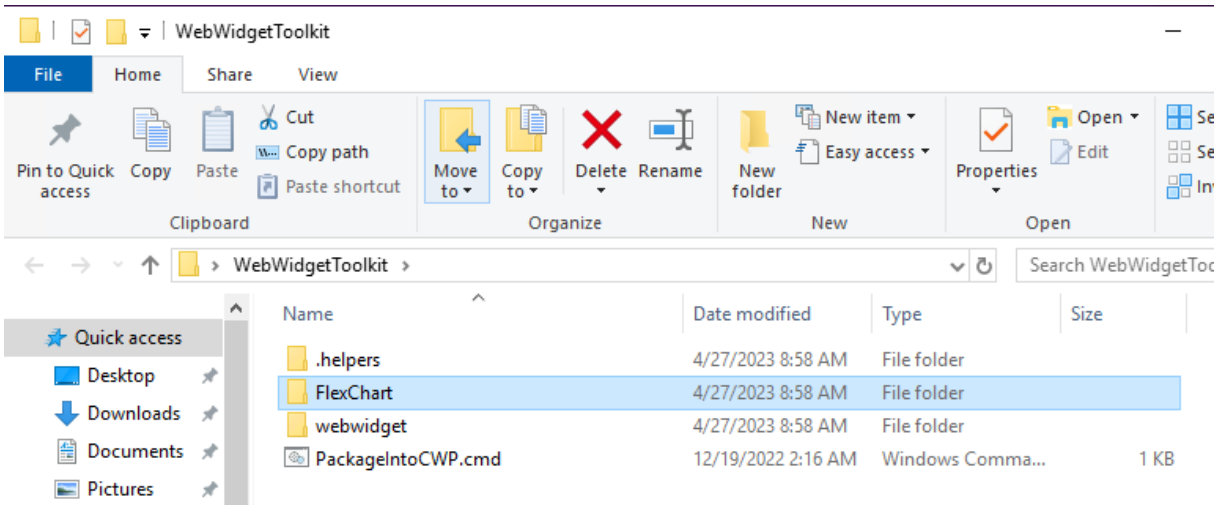
On the Hands-on section we are going to explore how to grab code from a demo chart and create our HTML, CSS and JS files.

2.2. Hands-On: Part 1

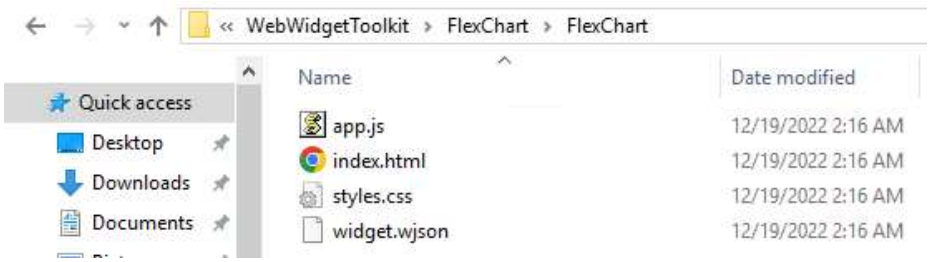
Important: Javascript is case sensitive, be carefull with the coding.

1. Go to Desktop -> Hands On -> Building Web Widgets for Industrial Graphics.
2. Copy the *WebWidgetToolkit* folder into the desktop.
3. Open *WebWidgetToolkit* folder. Make a copy of the folder named webwidget and rename it to the name that you want to give to your web widget. For this lab, we will be naming it **FlexChart**.

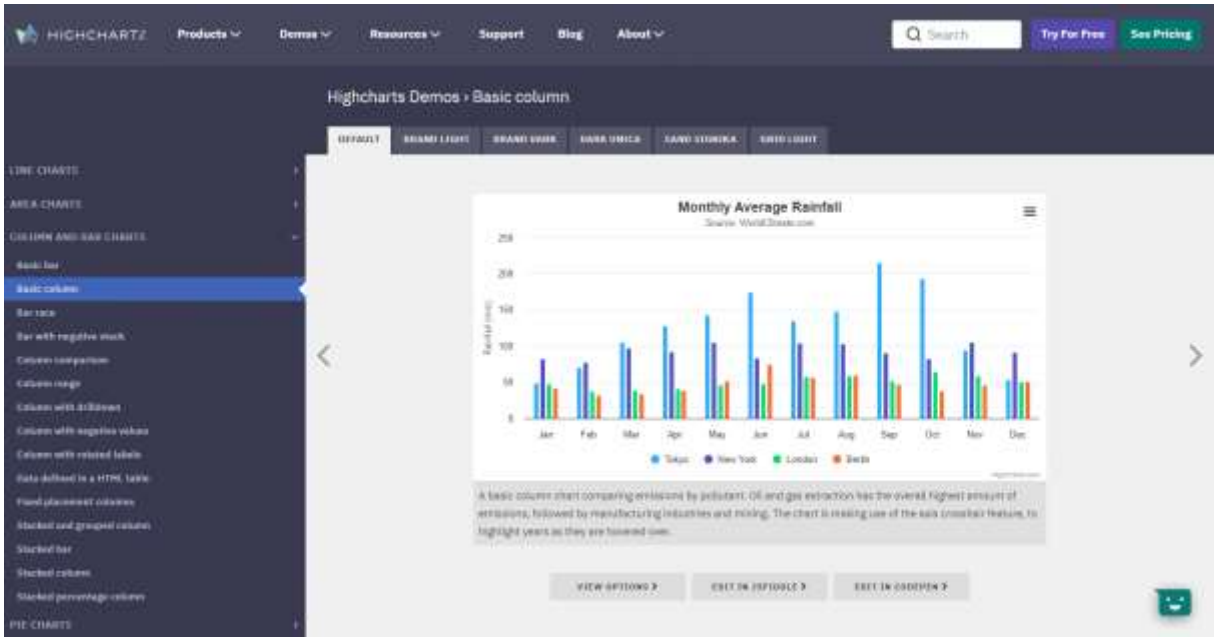
   Note: It is better to make a copy of the webwidget folder to ensure that we don't modify the original boilerplate code.



4. Open the newly created FlexChart folder. You will see two folders: *resources* and *widgetname*. Rename the *widgetname* folder into **FlexChart** (both the parent and child folders must have the same name).
5. Open the child *FlexChart* folder. You will see four files: **app.js**, **index.html**, **styles.css** and **widget.wjson**. Open these files using Visual Studio Code (or any other text editor).

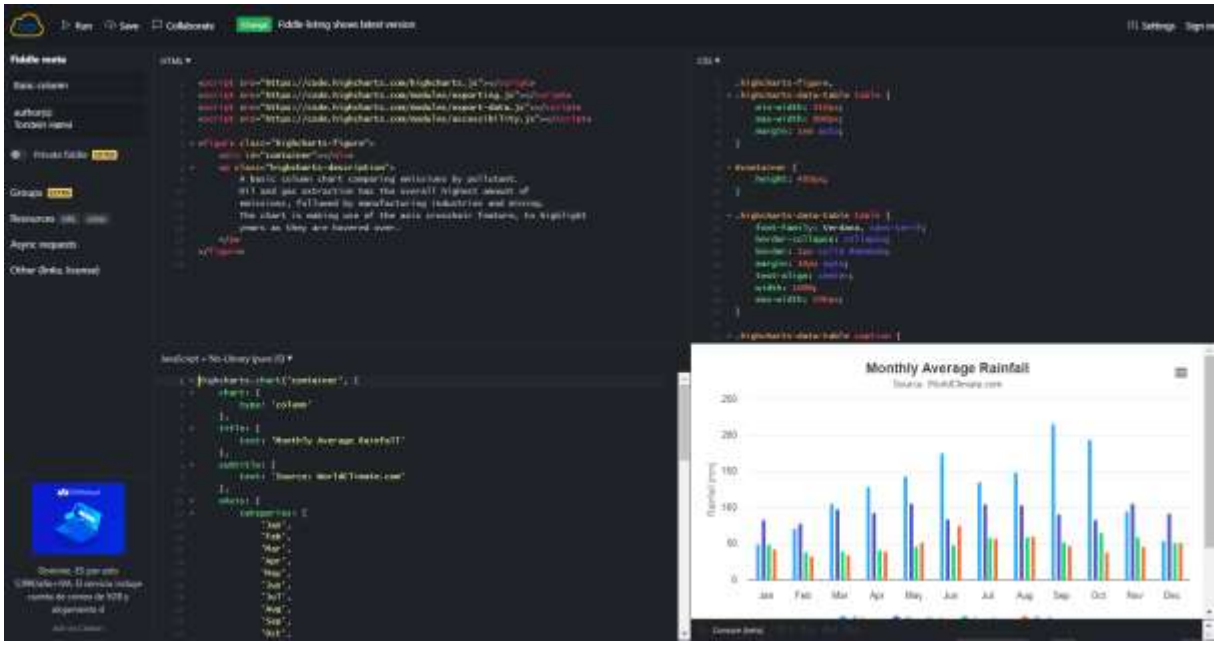


6. Go to *www.highcharts.com/demo/column-basic* using a browser.

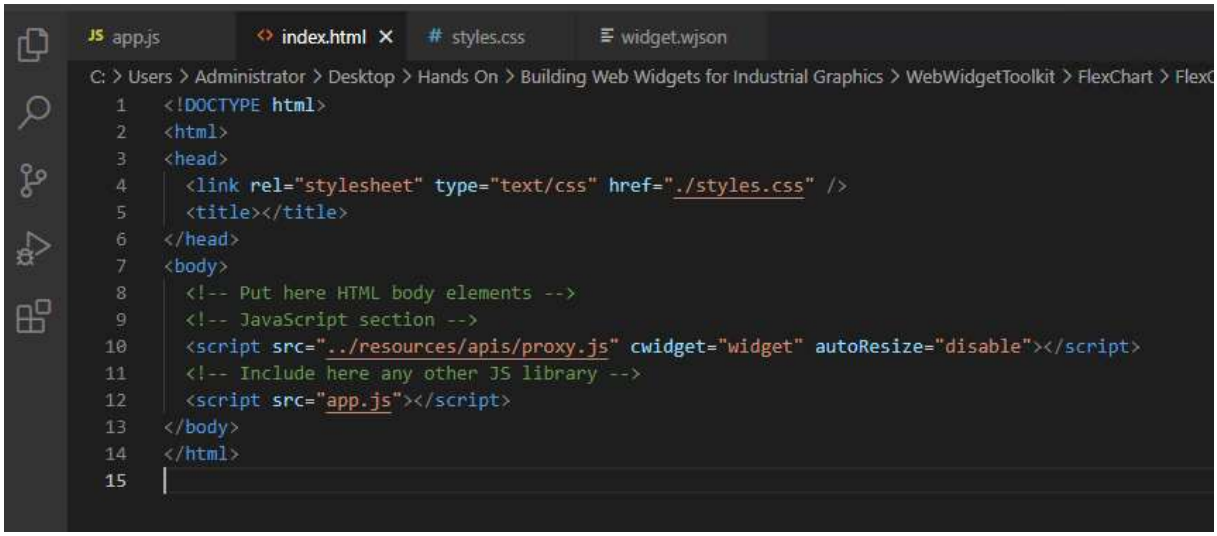


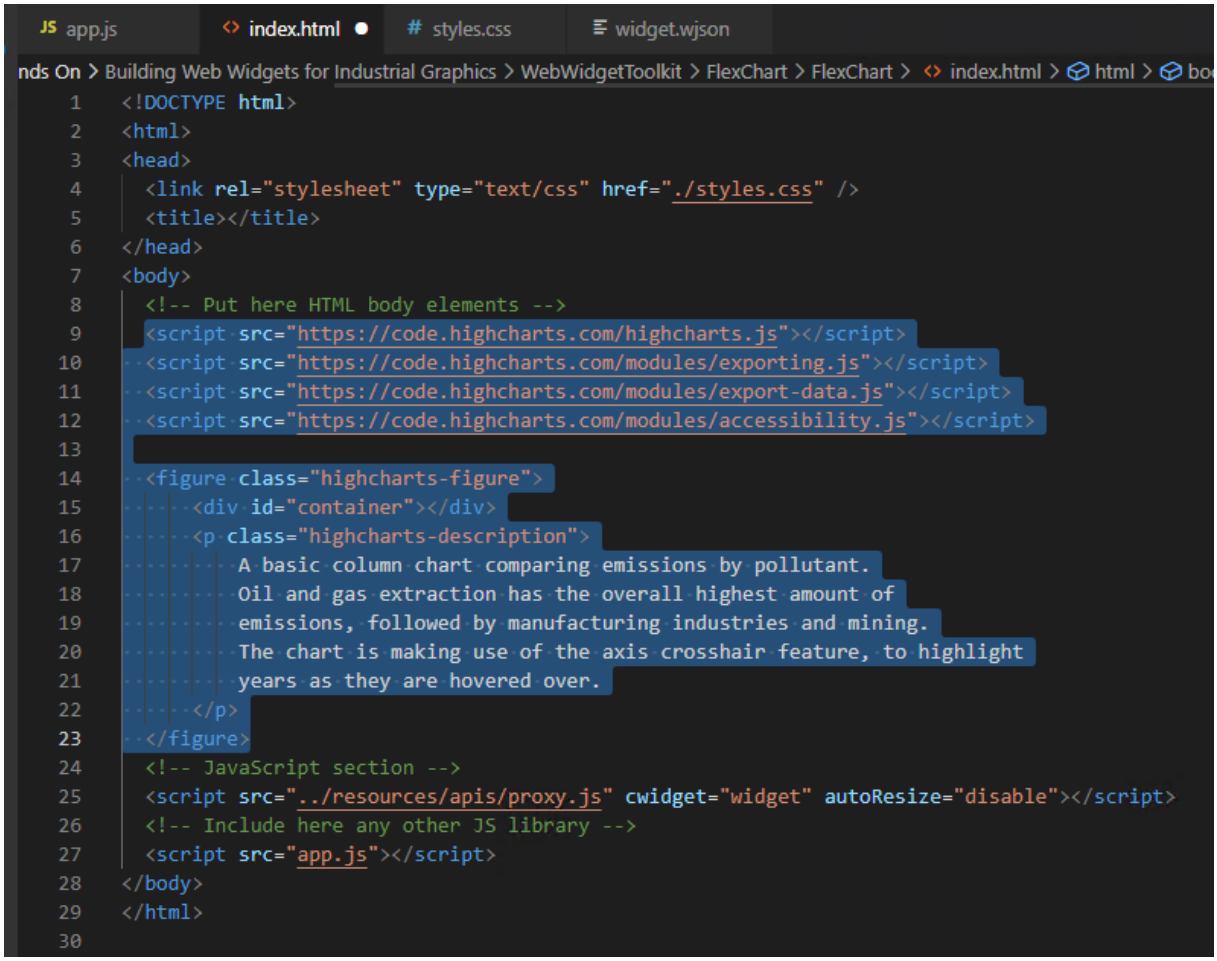
NOTE: Please, review the Highcharts license policy if you want to use these charts.

7. Click on the 'EDIT IN JSFIDDLE >' button.



8. Copy the HTML code of the graph from JSFIDDLE (top-left) and paste it in the **index.html** file, specifically where it says **<!-- Put here HTML body elements -->** code





9. Copy the CSS code from JSFIDDLE (top-right) and replace the code in **styles.css** file. Delete the original code of the file and copy the one from JSFIDDLE.

app.js index.html styles.css widget.wjson

C: > Users > Administrator > Desktop > Hands On > Building Web Widgets for Indus

```css
.highcharts-figure,
.highcharts-data-table table {
    min-width: 310px;
    max-width: 800px;
    margin: 1em auto;
}

#container {
    height: 400px;
}

.highcharts-data-table table {
    font-family: Verdana, sans-serif;
    border-collapse: collapse;
    border: 1px solid #ebebeb;
    margin: 10px auto;
    text-align: center;
    width: 100%;
    max-width: 500px;
}

.highcharts-data-table caption {
    padding: 1em 0;
    font-size: 1.2em;
    color: #555;
}

.highcharts-data-table th {
    font-weight: 600;
    padding: 0.5em;
}

.highcharts-data-table td,
.highcharts-data-table th,
.highcharts-data-table caption {
    padding: 0.5em;
}

.highcharts-data-table thead tr,
.highcharts-data-table tr:nth-child(even) {
    background: #f8f8f8;
}

.highcharts-data-table tr:hover {
    background: #f1f7ff;
}
```

10. Copy the JavaScript code from JSFIDDLE (bottom-left) and paste it inside the function declaration (between line 1 and 3) in the **app.js** file.

app.js index.html styles.css

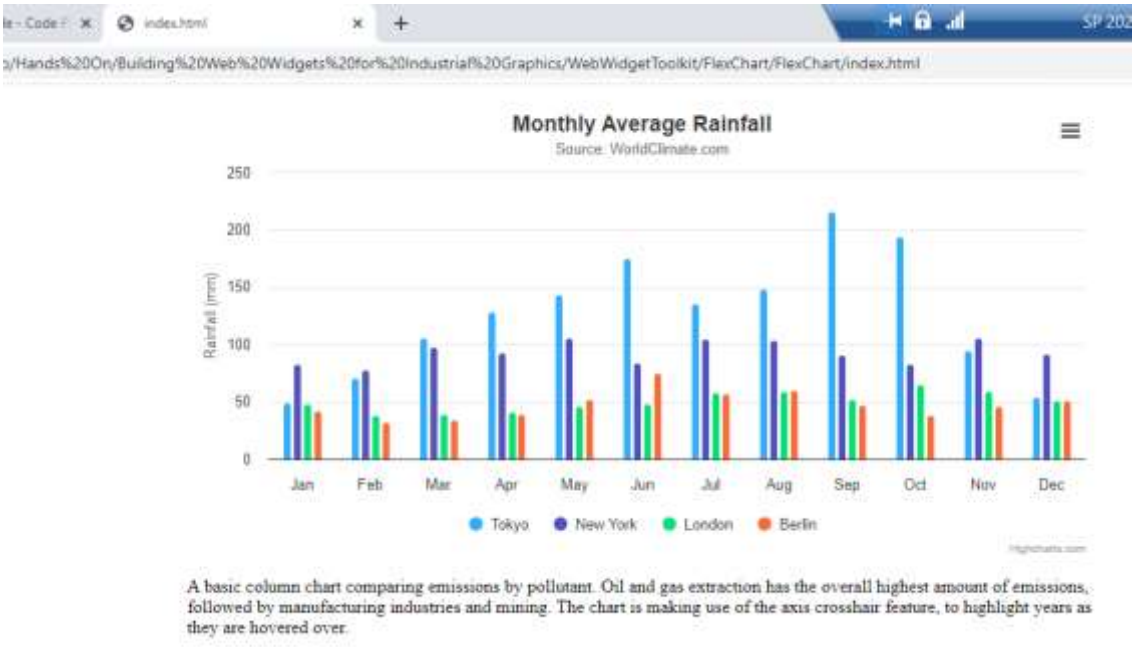
C: > Users > Administrator > Desktop > Hands On > Bu

```js
(function (cwidget) {

})(window.cwidget);
```

```
(function (cwidget) {
    Highcharts.chart('container', {
        chart: {
            type: 'column'
        },
        title: {
            text: 'Monthly Average Rainfall'
        },
        subtitle: {
            text: 'Source: WorldClimate.com'
        },
        xAxis: {
            categories: [
                'Jan',
                'Feb',
                'Mar',
                'Apr',
                'May',
                'Jun',
                'Jul',
                'Aug',
                'Sep',
                'Oct',
                'Nov',
                'Dec'
            ],
            crosshair: true
        },
        yAxis: {
            min: 0,
            title: {
                text: 'Rainfall (mm)'
```

11. Make sure to save the three files that you just modified.
12. You can open the **index.html** file with your browser to check if you've correctly copied the code. In the browser you should see a graph just like the one from the HighCharts website.



A basic column chart comparing emissions by pollutant. Oil and gas extraction has the overall highest amount of emissions, followed by manufacturing industries and mining. The chart is making use of the axis crosshair feature, to highlight years as they are hovered over.

3. Interface with System Platform

3.1. WJSON File

The **.wjson** file is used to describe the properties of the web widget. To import it and enable the communication between System Platform and our web widget, we need to include the code below in the **index.html**. This will create a global object called *cwidget*.

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="./styles.css" />
    <title></title>
</head>
<body>
    <!-- Put here HTML body elements -->

    <!-- JavaScript section -->
    <script src="../resources/apis/proxy.js" cwidget="widget" autoResize="disable"></script>
    <!-- Include here any other JS library -->
    <script src="app.js"></script>
</body>
</html>
```

Make sure that the name of the **.wjson** file matches the cwidget attribute of the highlighted script. In the case of the picture above, the **.wjson** file is named as “widget”.

The supported property types in the **.wjson** file are: *Integer*, *String*, *Float*, *Double* and *Boolean*. Note that the *Boolean* values will be represented as string values "True" or "False" in JavaScript. We can see an example below:

app.js index.html styles.css widget.wjson

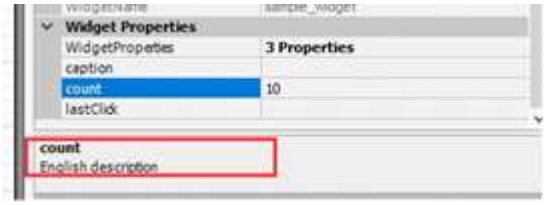
C: > Users > Administrator > Desktop > Hands On > Building Web Widgets for Industria

```
{
    "version": 0,
    "height": 400,
    "width": 400,
    "properties":
    {
        "0": {
            "name": "StrValue",
            "type": "String",
            "value": ""
        },
        "1": {
            "name": "IntValue",
            "type": "Integer",
            "value": "0"
        },
        "2": {
            "name": "FltValue",
            "type": "Float",
            "value": "0.0"
        },
        "3": {
            "name": "DblValue",
            "type": "Double",
            "value": "0.0"
        },
        "4": {
            "name": "BlnValue",
            "type": "Boolean",
            "value": "False"
        }
    }
}
```

For properties configuration in **.wjson** file, except 'name' user also can specify data *type*, *default value*, *description* and *allowEmpty*, like below:

```
"properties": {
  "0": {
    "name": "caption"
  },
  "1": {
    "name": "lastClick"
  },
  "2": {
    "name": "count",
    "type": "Integer",
    "value": "10",
    "desc": {
        "1033": "English description",
        "1036": "French description",
        "1031": "German description",
        "1041": "Japanese description",
        "2052": "German description"
    },
    "allowEmpty": "true"
  }
}
```

If no type here, it's string by default. If user want to see description when design property, 'desc' should be set here.



If allow to set empty for the property in design time, 'allowEmpty' need to be set to true, default is false.

The cwidget global object created by *proxy.js* script will have the properties defined in the **.wjson** file. These properties can be read (In) or written (Out). It also includes a method called on which allows us to subscribe to changes on property values. For this, the *on* method expects 2 arguments: property name and function to be executed on change.

On the next code snippet you can see how to interact with the cwidget object:

```
cwidget.on('Data', function () { // Subscribe to changes on Data property
    console.log(cwidget.Data); // Read Data property
});

function setDataValue (value) {
    cwidget.Data = value; // Write Data property
}
```

3.2. Hands-On: Part 2

1. In the **app.js** file, assign the chart created to a variable called chart, as shown below. This is so that we can refer to the chart object in our future code.

```
(function (cwidget) {
    var chart = Highcharts.chart('container', {
        chart: {
            type: 'column'
        },
        title: {
            text: 'Monthly Average Rainfall'
```

```
        },
```

2. In the **app.js** file, remove the rest of the series and leave only the Tokyo series.

```
49          series: [{
50              name: 'Tokyo',
51              data: [49.9, 71.5, 106.4, 129.2, 144.0, 176.0, 135.6, 148.5, 216.4,
52                  194.1, 95.6, 54.4]
53
54          }, {
55              name: 'New York',
56              data: [83.6, 78.8, 98.5, 93.4, 106.0, 84.5, 105.0, 104.3, 91.2, 83.5,
57                  106.6, 92.3]
58
59          }, {
60              name: 'London',
61              data: [48.9, 38.8, 39.3, 41.4, 47.0, 48.3, 59.0, 59.6, 52.4, 65.2, 59.3,
62                  51.2]
63
64          }, {
65              name: 'Berlin',
66              data: [42.4, 33.2, 34.5, 39.7, 52.6, 75.5, 57.4, 60.4, 47.6, 39.1, 46.8,
67                  51.1]
68
69          }]
70      });
71  })(window.cwidget);
```

3. Copy the data list of the Tokyo series: [49.0, 71.5, 106.4, 129.2, 144.0, 176.0, 135.6, 148.5, 216.4, 194.1, 95.6, 54.4] and replace the last code line of the **app.js** with the following code. The code ensures that should there be no *cwidget* global variable, return the Data object created. Note that the data list is now a string representation of the list.

```
})(window.cwidget || {
    Data: "[49.9, 71.5, 106.4, 129.2, 144.0, 176.0, 135.6, 148.5, 216.4, 194.1,
95.6, 54.4]"
});
```

```
49          series: [{
50              name: 'Tokyo',
51              data: [49.9, 71.5, 106.4, 129.2, 144.0, 176.0, 135.6, 148.5, 216.4, 194.1, 95.6, 54.4]
52
53          }]
54      });
55  })(window.cwidget || {
56      Data: "[49.9, 71.5, 106.4, 129.2, 144.0, 176.0, 135.6, 148.5, 216.4, 194.1, 95.6, 54.4]"
57  });
```

4. Replace the data attribute of the Tokyo series with the line:

```
data: cwidget.Data ? JSON.parse(cwidget.Data) : []
```

5. This will assign the data property as the parsed cwidget.Data value if this property has a value, or an empty array otherwise. Because the Data property is a string (representing a JavaScript Object), we need to parse it.

```
        series: [{
            name: 'Tokyo',
            data: cwidget.Data ? JSON.parse(cwidget.Data) : []

        }]
    });
})(window.cwidget || {
    Data: "[49.9, 71.5, 106.4, 129.2, 144.0, 176.0, 135.6, 148.5, 216.4, 194.1, 95.6, 54.4]"
});
```

6. Between the series information and the line you edited in Step 4, add the code as shown in the picture below. This code will allow us to subscribe to changes whenever we have data for the Tokyo series.

```
cwidget.on('Data', function() {
   chart.series[0].setData(JSON.parse(cwidget.Data));
});
```

```
        series: [{
            name: 'Tokyo',
            data: cwidget.Data ? JSON.parse(cwidget.Data) : []

        }]
    });
    cwidget.on('Data', function() {
        chart.series[0].setData(JSON.parse(cwidget.Data));
    });
})(window.cwidget || {
    Data: "[49.9, 71.5, 106.4, 129.2, 144.0, 176.0, 135.6, 148.5, 216.4, 194.1, 95.6, 54.4]"
});
```

7. Lastly, in the **widget.wjson** file, change the name of the "0" attribute to "Data" and delete the other properties.

app.js | index.html | styles.css | widget.wjson

C: > Users > Administrator > Desktop > Hands On > Building Web Widgets for Industria

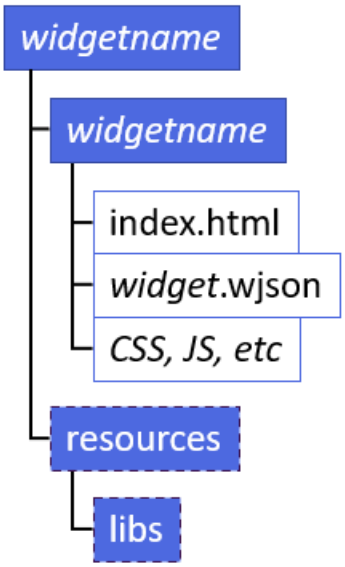
```
{
    "version": 0,
    "height": 400,
    "width": 400,
    "properties":
    {
        "0": {
            "name": "Data",
            "type": "String",
            "value": ""
        }
    }
}
```

8. Make sure to save all the files that you just modified.

4. Import into System Platform

4.1. Web Widget Package

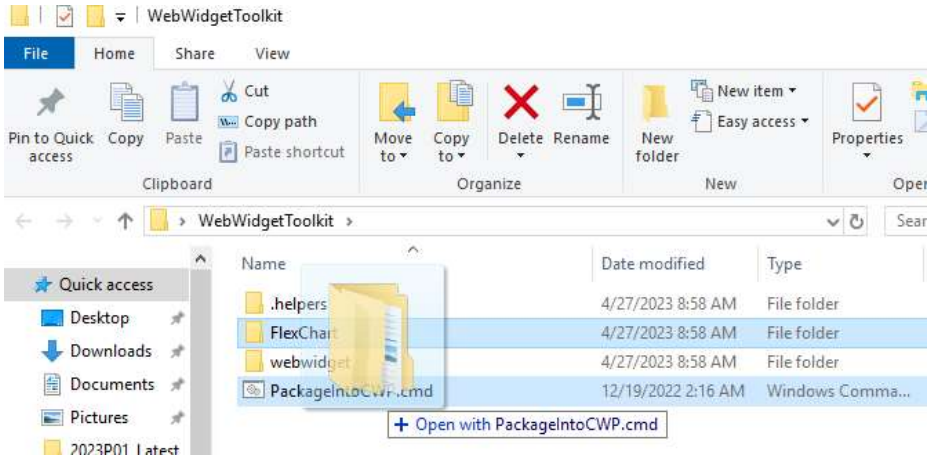
So far, you have created four files: **index.html, style.css, app.js** and **widget.wjson**. These files are placed under a folder **FlexChart** (or any other *widgetname*). The **FlexChart** folder is placed under the **FlexChart** parent folder. Optionally, you could also create a **resources** folder that contains the libraries used in the widget under the parent **FlexChart** folder.



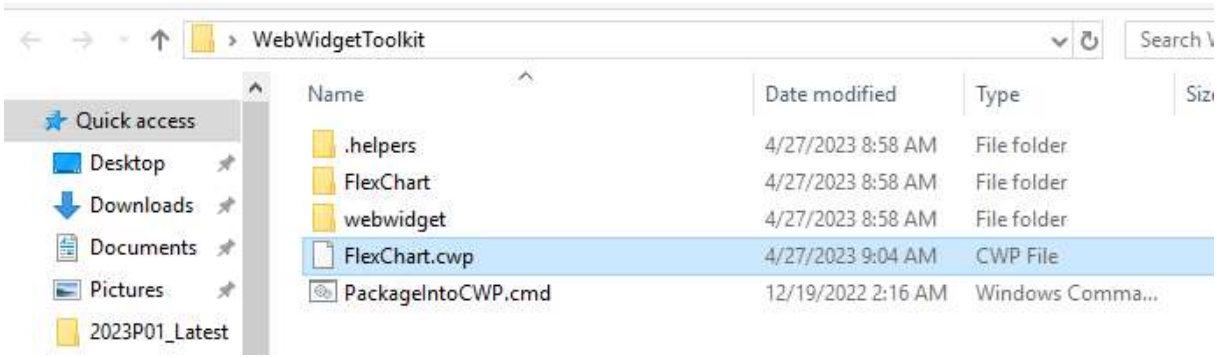
Then, use the PackageTool (**PackageIntoCWP.cmd**) provided in the *WebWidgetToolkit* to compress the parent **FlexChart** folder and create a CWP file. CWP is a compressed file that includes the necessary files and folders for our widget to work. The CWP file must be named the same as the widget name. The CWP is the file we import in AVEVA System Platform as a web widget.

4.2. Hands-On: Part 3

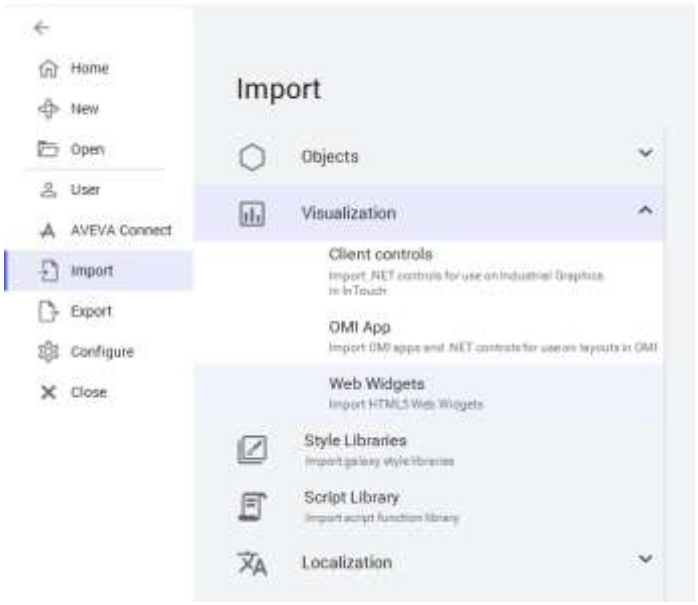
1. Select parent FlexChart and move into PackageIntoCWP.cmd.

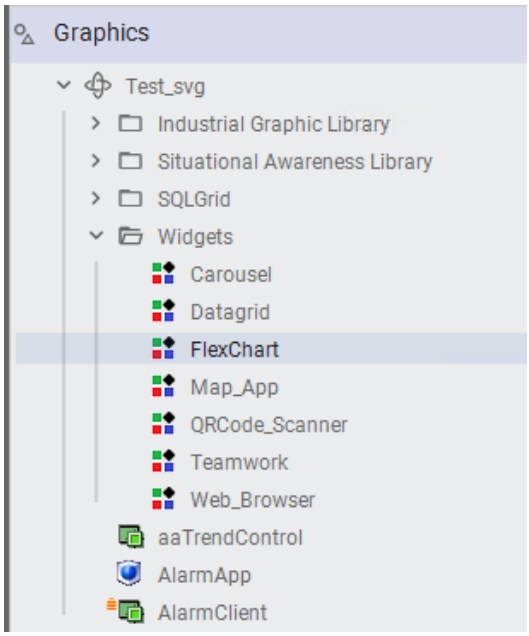


2. The PackageTool will generate the compressed package at the same folder.



3. Open the AVEVA System Platform IDE. Go to Galaxy > Import > Visualization < Web Widget. Then select the .cwp file. The imported web widget will be in the Graphic Toolbox inside the Widgets toolbox and available to be used as any other Graphic.





5. Configure a Web Widget

We can use the imported web widget either inside an Industrial Graphic (embedded) or directly in an OMI Layout. Either way we are going to be able to read and set the properties previously defined in the **.wjson** file.

When embedded inside a Industrial Graphic we can set the properties as constant values or bind them to custom properties or expressions.

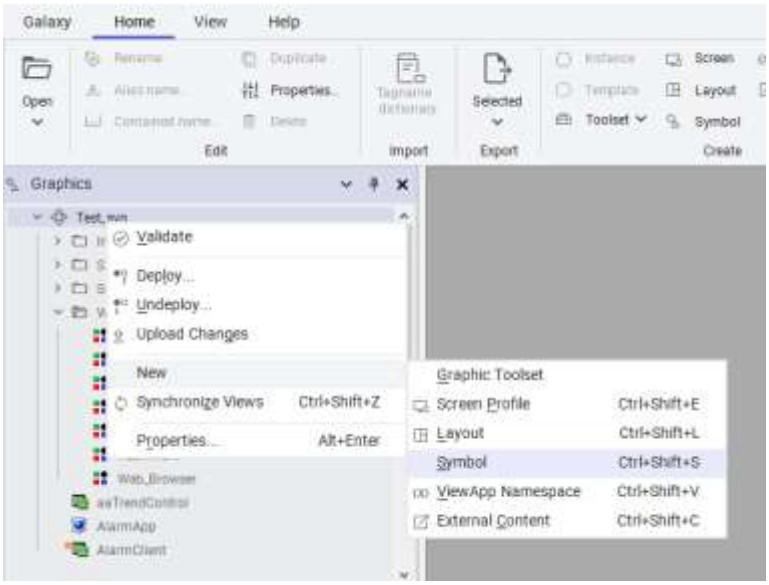
Similarly, when using it directly on a Layout, widget properties can be set as:

- Constant
- In (values will be read from the widget)
- Out (values will be set by the widget)
- In/Out (values can be both read and written from the widget)

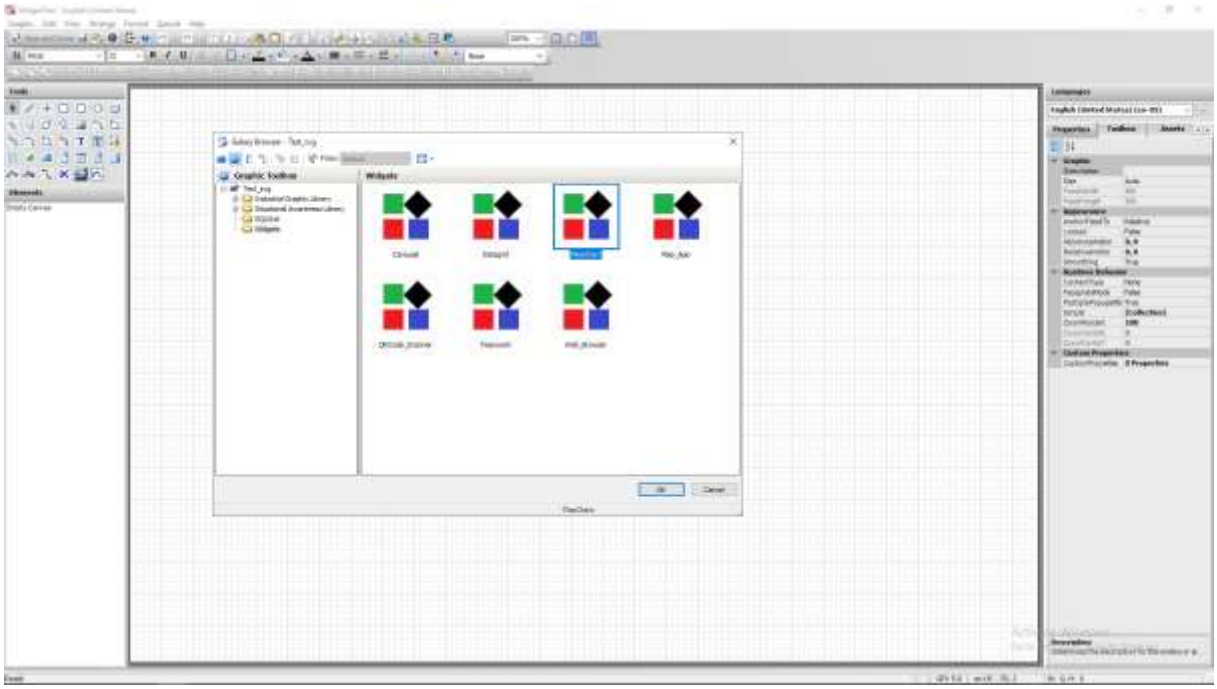
We often use namespaces to move values between different Graphics and Widgets.

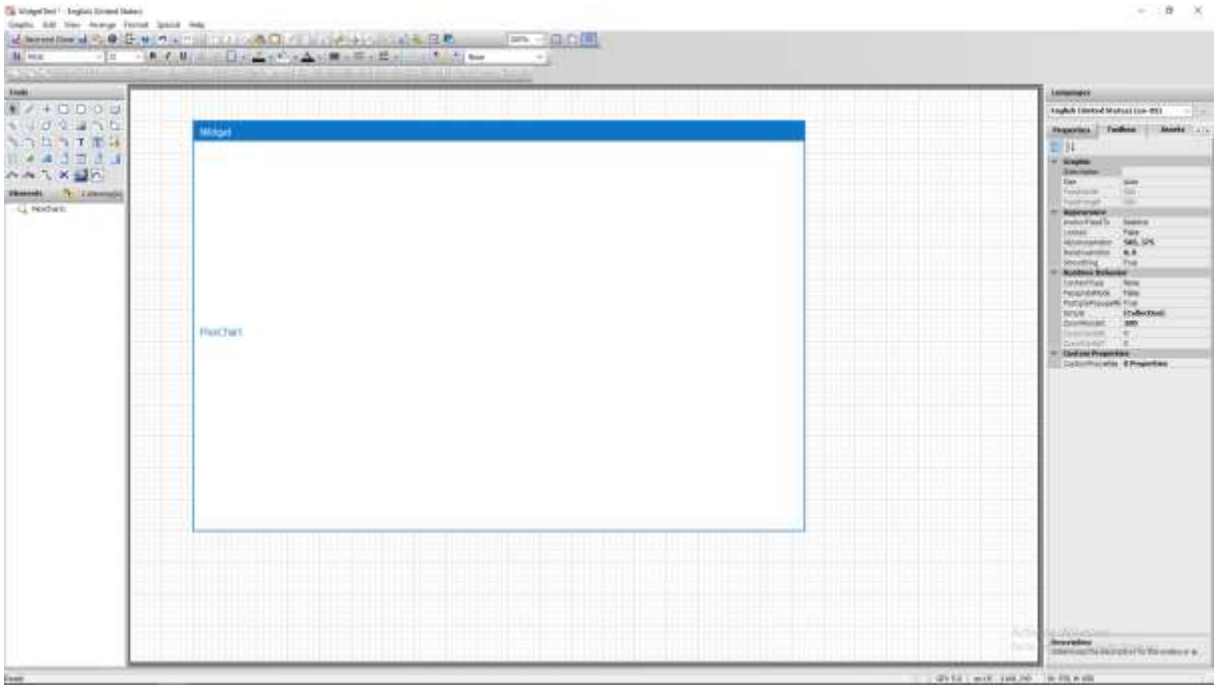
5.1. Hands-On: Part 4

1. In the Graphic Toolset, right click on the Galaxy Name and toggle on the New > Symbol.



2. Rename the symbol to **WidgetTest**. Then, open the symbol to edit it in the Graphic Editor.
3. Click Edit > Embed Industrial Graphic. Go to Widgets and select FlexChart. You can resize the widget to a size that you see fit.

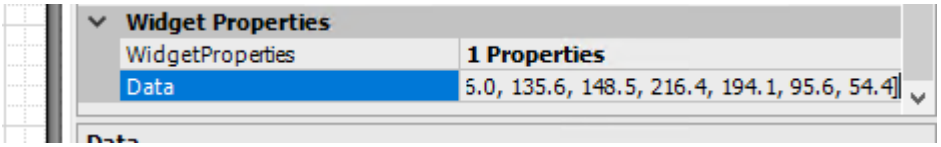




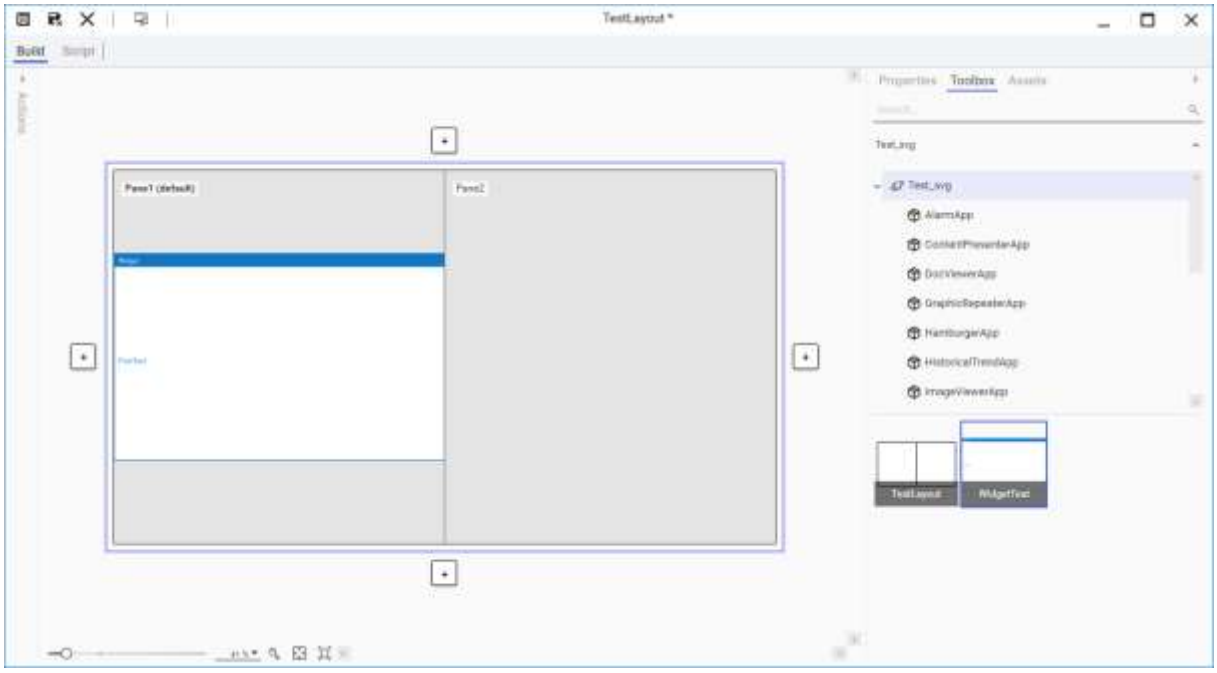
4. In the app.js file, copy the Data string. (the line highlighted below)

```
58 })(window.cwidget || {
59     Data: "[49.9, 71.5, 106.4, 129.2, 144.0, 176.0, 135.6, 148.5, 216.4, 194.1, 95.6, 54.4]"
60 });
```

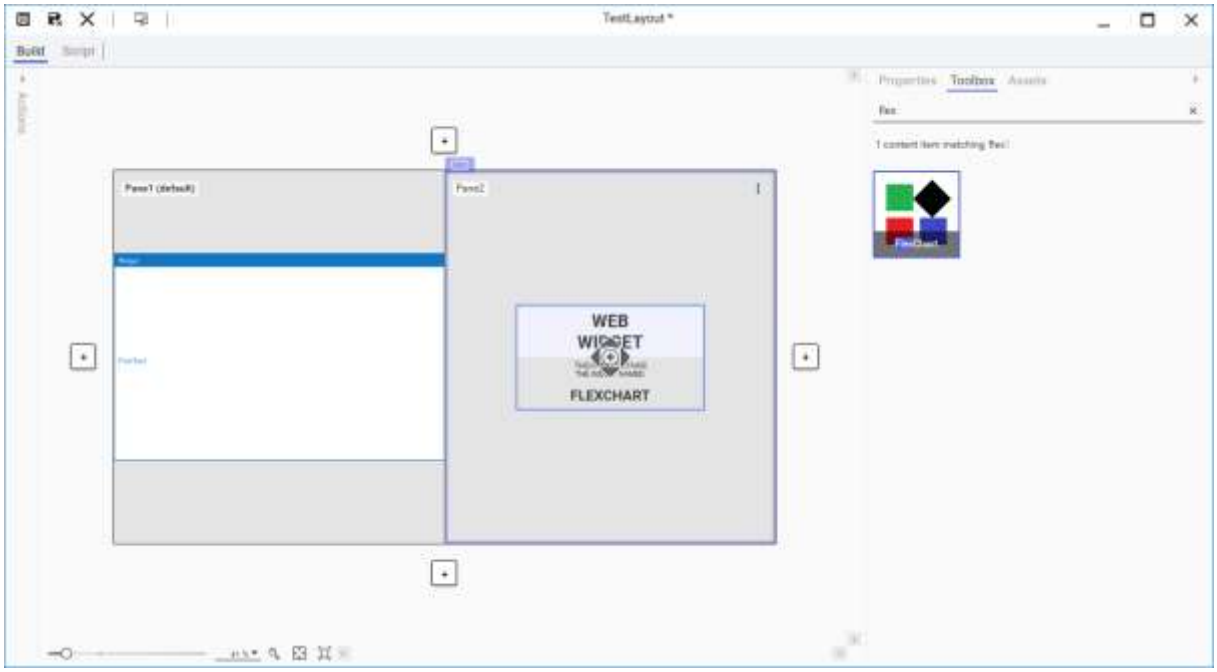
5. In System Platform IDE, under the Properties pane of the widget, you will see Widget Properties. In the Data property, paste the data stream.



6. Save and close the Graphic Editor.
7. In the Graphic Toolbox, create a new ScreenProfile and name it **TestScreenProfile**.
8. In the Graphic Toolbox, create a new layout and name it **TestLayout**. Double click it to open it in the Graphic Editor.
9. Add a new pane to the right side of Pane1. Then, in the Toolbox section search for WidgetTest. Drag WidgetTest on Pane1.

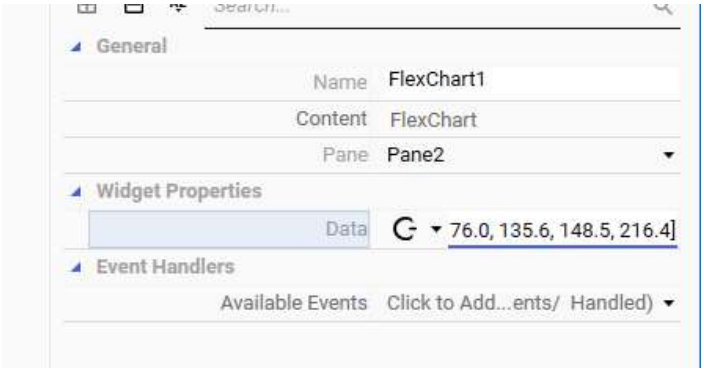


10. In the Toolbox section, search for FlexChart. Drag and drop it on Pane2.

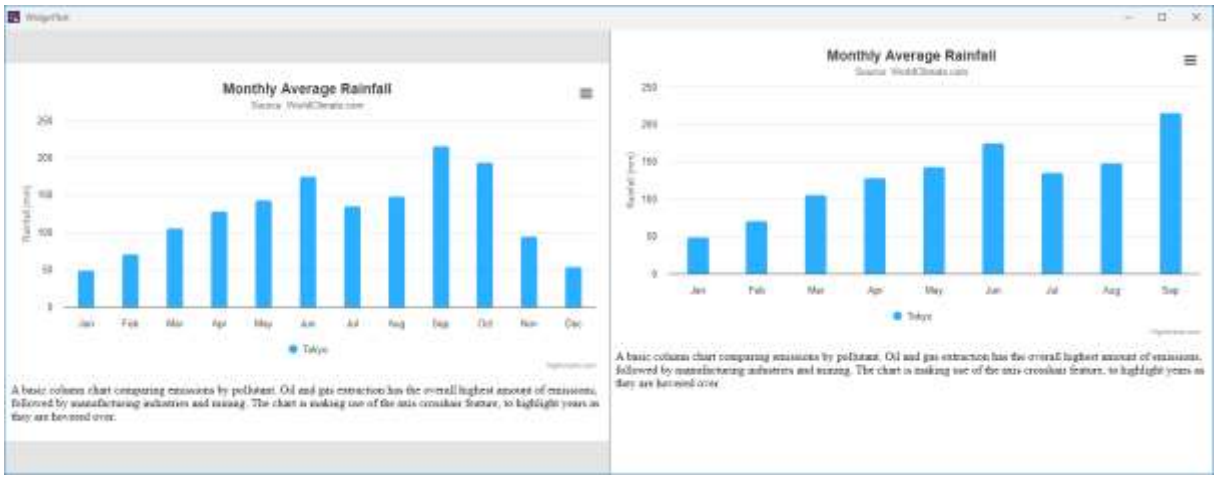


11. Go to the properties section of the FlexChart widget and paste the Data Stream under the Widget Properties section.

12. Remove the last three data of the data stream list.



13. Save and Close the Editor.
14. In the Template Toolbox, in the System Folder, right click on $ViewApp. Create a New > Derived Template. Rename it as **$WidgetTest**.
15. Doble click to $WidgetTest to configure the OMI App.
16. Select one ScreenProfile and click on Next.
17. Select the TestLayout and click Finish.
18. Click on the Preview Button. You should see the web widgets that you just imported.



6. Tips and Tricks

- Make sure that the web widget code works before integrating into the System Platform.
- You can set the property type as string and serialize the data as JSON to pass complex data types.
- You can validate JSON strings through: *https://jsonlint.com*
- You can issue REST calls directly inside the widget
- If you need to react to an event inside the widget, set a property with the event data and subscribe to changes on that property.
- You can debug the widget using browser developer tools while running the ViewApp in OMI Web.
- Also, adding this line to OnShow layout script, you will be able to open developer tools in OMI Desktop too:

  ```
  MyViewApp.ViewApp.AllowBrowserDevTools = true;
  ```

- If you are using external libraries that are hosted in the cloud, it is recommended to download them to later be able to use the web widget in environments without an internet connection. Libraries must locate in *resources/libs*.
- In order to use the PackageTool, the path must not contain spaces.
- To import a new version of the web widget:
  1. Close any industrial graphic or layout where the web widget is used.
  2. Close ViewApp editor.
  3. Delete the web widget.
  4. Import the new web widget version.

7. Security

- Make sure that the web widget comes from a website or source that is trusted.
- Do not use functions in JavaScript that allows to execute any code coming, especially with the properties that the clients can set. If not, malicious users will be able to write strings in such a way that they could steal user information.
- Abide by the same security measures and precautions that you would take when building a web application.
- Be careful to not put any sensitive information inside the widget (in the code or through passing as an attribute/property) since it is very easy to inspect. Any sensitive information should be kept on the backend (System Platform).

8. Enhancing the Web Widget

8.1. Hands-On: Part 5

In the index.html file, there are references to external libraries, so if the machine doesn't have internet connection the web widget will not work. A good practice is to use these libraries locally.

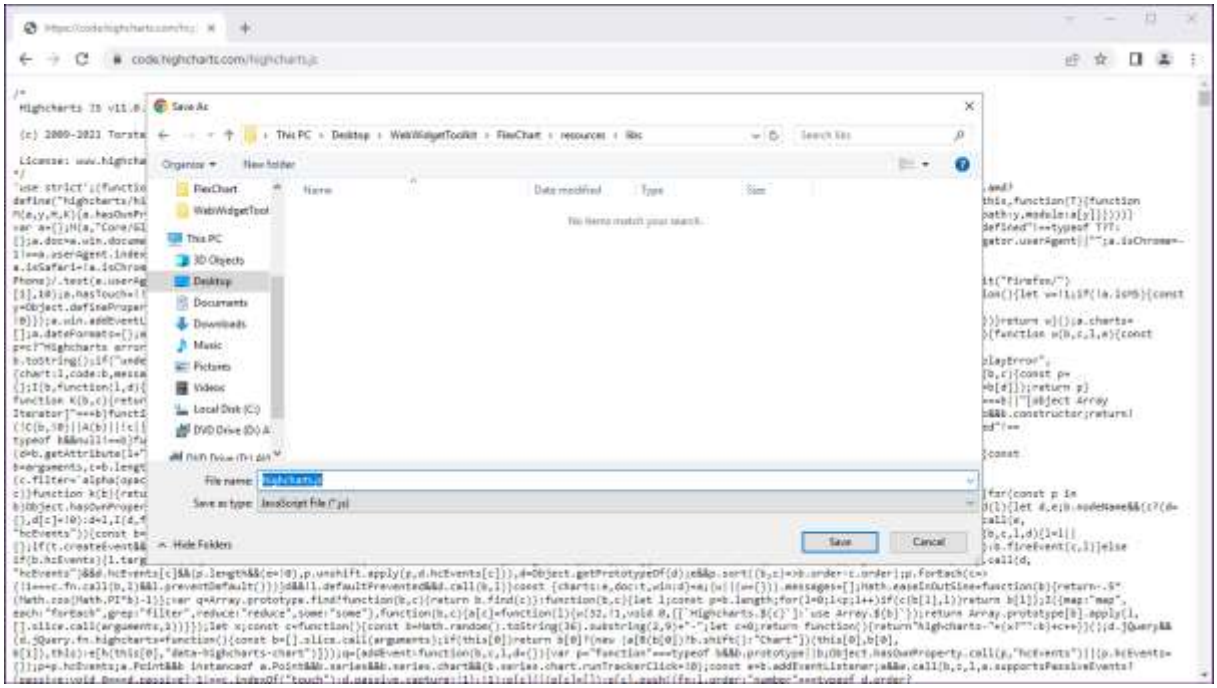
1. In the **index.html**, delete the last 3 referentes to the modules of highcharts. These 3 libraries are for advanced features as exporting data, so for this example it's not need it.

```
7   <body>
8     <!-- Put here HTML body elements -->
9     <script src="https://code.highcharts.com/highcharts.js"></script>
10    <script src="https://code.highcharts.com/modules/exporting.js"></script>
11    <script src="https://code.highcharts.com/modules/export-data.js"></script>
12    <script src="https://code.highcharts.com/modules/accessibility.js"></script>
13
14    <figure class="highcharts-figure">
15        <div id="container"></div>
```

2. Copy the only reference to highcharts below line: <!-- Include here any other JS library -->

```
21    <script src="../resources/apis/proxy.js" cwidget="widget" autoResize="disable"></script>
22    <!-- Include here any other JS library -->
23    <script src="https://code.highcharts.com/highcharts.js"></script>
24    <script src="app.js"></script>
25  </body>
26  </html>
```

3. Click to the url reference or copy the url and paste it into a web browser. The library will appear inside the web browser.
4. Righ-click and save as.



5. Save it in the following location: *WebWidgetToolkit\FlexChart\resources\libs*.
6. Modify the reference for the local library instead of the cloud one.

```
<script src="../resources/libs/highcharts.js"></script>
```

7. Save the file.
8. Delete the previous package and create a new one. Remember, drag and drop the folder FlexChart into PackageIntoCWP.cmd
9. Open the System Platform IDE, if any industrial graphic or layout is open, close it. Also close the ViewApp Editor if it's opened.
10. Go to Graphics->Widgets and delete the FlexChart.
11. Import the new version.
12. Open the ViewApp editor and click Preview to test the new version.

8.2. Hands-On: Part 6

This lab is an example of how to pass data from the web widget to System Platform. Clicking on a bar generates an event and at that moment a set of data is captured, which is passed in json format to the System Platform through a string-type property.

Here is the documentation of the function used in this lab:
https://api.highcharts.com/highcharts/plotOptions.series.events.click

1. With **app.js** file opened, add the following code inside plotOptions.

```
,
series: { events: { click: function(event) {
      cwidget.Click = JSON.stringify({
            seriesIndex: event.point.series.index,
             seriesName: event.point.series.name,
             pointIndex: event.point.index,
             pointName: event.point.name,
             category: event.point.category,
             x: event.point.x,
             y: event.point.y
      });
} } }
```

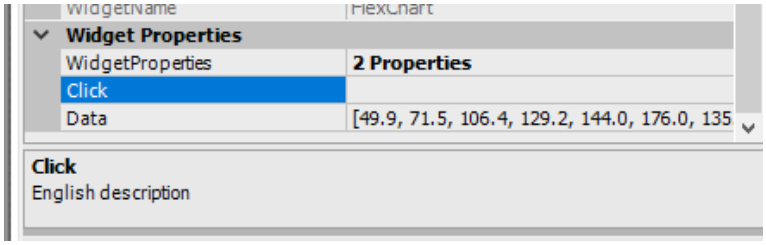
```
43          plotOptions: {
44              column: {
45                  pointPadding: 0.2,
46                  borderWidth: 0
47              },
48              series: { events: { click: function(event) {
49                  cwidget.Click = JSON.stringify({
50                      seriesIndex: event.point.series.index,
51                      seriesName: event.point.series.name,
52                      pointIndex: event.point.index,
53                      pointName: event.point.name,
54                      category: event.point.category,
55                      x: event.point.x,
56                      y: event.point.y
57                  });
58              } } }
59          },
60          series: [{
61              name: 'Tokyo',
62              data: cwidget.Data ? JSON.parse(cwidget.Data) : []
63
64          }]
```

2. The new property must be defined in the **widget.wjson**. Additionally, we are going to add a multilingual description.

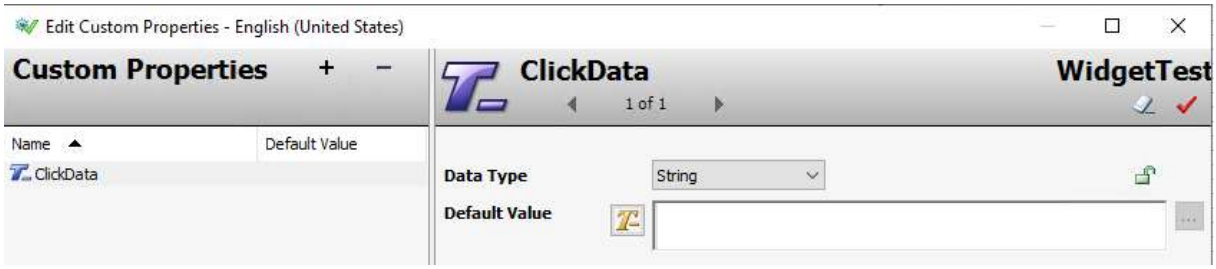
```
,
       "1": {
          "name": "Click",
           "type": "String",
           "value": "",
           "desc": {
               "1033": "English description",
               "1036": "French description",
               "1031": "English description",
               "1041": "Japanese description",
               "2052": "Chinese description"
           }
}
```

```
            "value": ""
        },
        "1": {
            "name": "Click",
            "type": "String",
            "value": "",
            "desc": {
                "1033": "English description",
                "1036": "French description",
                "1031": "English description",
                "1041": "Japanese description",
                "2052": "Chinese description"
            }
        }
    }
}
```

3. Save the file.
4. Delete the previous package, drag and drop the folder FlexChart into PackageIntoCWP.cmd to create a new one.
5. Open the System Platform IDE, if any industrial graphic or layout is open, close it. Also close the ViewApp Editor if it's opened.
6. Go to Graphics->Widgets and delete the FlexChart.
7. Import the new version.
8. Go to Graphics and open WidgetTest graphic.
9. Verify that the new property appears and has the description.

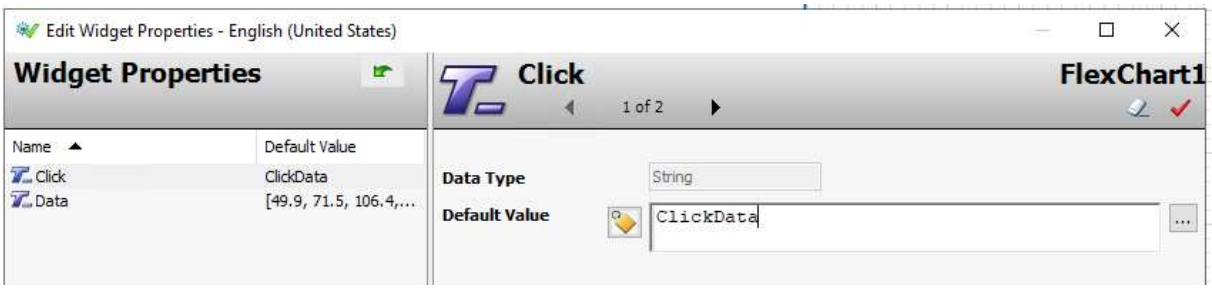


10. Create a new string Custom Property called ClickData.

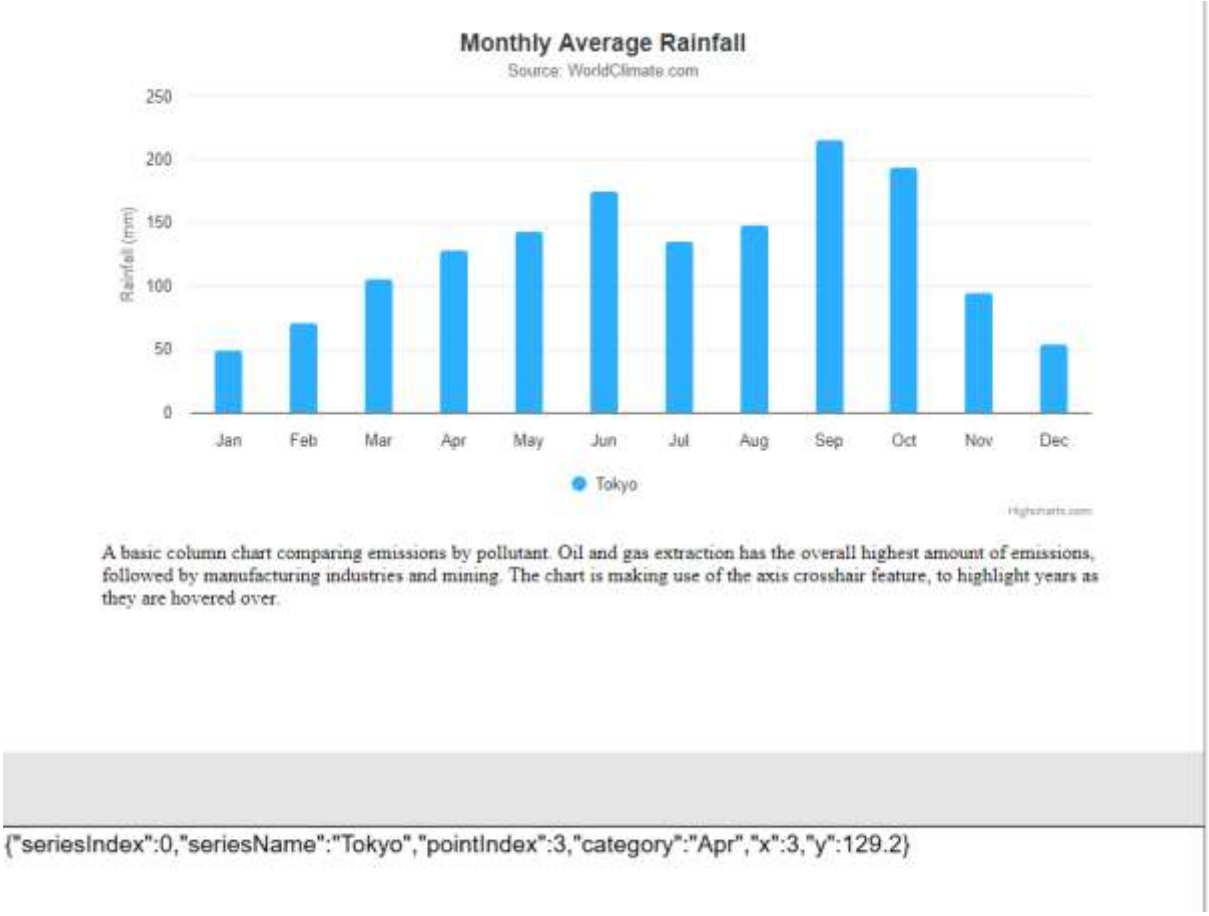


11. Add a new TextBox below the FlexChart widget.
12. Add Value Display animation to the TextBox1 to display string data. On the reference box enter ClickData.

13. Open the FlexChart widget properties. Click the 3 dots button on WidgetProperties.
14. Modify the Click property to Reference.
15. Write ClickData as reference.



16. Close and save the graphic.
17. Open the ViewApp editor and click Preview to test the new version.
18. Click on any column of the left chart and the json data should appear on the text box.



8.3. Hands-On: Part 7

Modify the web widget code to change the chart type in runtime. Configure these 5 types: area, bar, column, line and pie. You should achieve something like: