

# Contents

[Power BI visuals documentation](#)

[Overview](#)

[Visuals in Power BI](#)

[Tutorials](#)

[Develop a Power BI visual](#)

[Create React-based Visuals](#)

[Build a bar chart](#)

[Add formatting options](#)

[Add unit tests for visual project](#)

[Concepts](#)

[Power BI visual project structure](#)

[Visuals concepts](#)

[Visuals for organizations in Power BI](#)

[Guidelines for Power BI visuals](#)

[Performance tips](#)

[Visuals FAQ](#)

[Using capabilities](#)

[Data view mappings](#)

[Objects and properties](#)

[Advanced Edit Mode](#)

[supportsKeyboardFocus feature](#)

[supportsMultiVisualSelection feature](#)

[How to](#)

[Get a Power BI visual certified](#)

[Publish Power BI visuals to AppSource](#)

[Create R-powered visuals](#)

[Test your Power BI visual](#)

[Use R-powered Power BI visuals in Power BI](#)

[Visual interactions](#)

- [Add selections](#)
- [Add visuals tooltips](#)
- [Add analytics pane](#)
- [Filter visuals](#)
- [Fetch more data](#)
- [Add bookmarks support](#)
- [Add context menu support](#)
- [Add Drill-Down support](#)
- [Add colors to your visual](#)
- [Highlight data in visuals](#)
- [Add High-Contrast mode support](#)
- [Enable sync slicers](#)
- [Debug visuals](#)
- [Mobile development](#)
- [Troubleshoot Power BI visuals](#)

## Reference

- [Changelog](#)
- [Landing page](#)
- [Launch URL](#)
- [Visual API](#)
- [Local Storage API](#)
- [Rendering events](#)
- [Sorting](#)
- [Localization](#)
- [Power BI Visual utilities](#)
  - [Adding external libraries](#)
  - [Interactivity utils](#)
  - [Formatting utils](#)
  - [Data view utils](#)
  - [Chart utils](#)
  - [Color utils](#)
  - [SVG utils](#)

[Type utils](#)

[Test utils](#)

[Tooltip utils](#)

[Data view utils](#)

## [Resources](#)

[Dev Center](#)

[Samples of Power BI visuals](#)

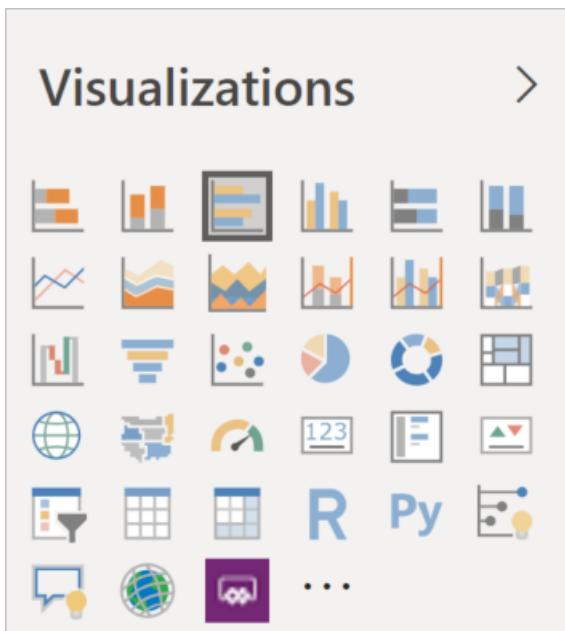
[Custom Visuals Git Repo](#)

[Creating SSL certificate](#)

# Visuals in Power BI

5/18/2020 • 3 minutes to read • [Edit Online](#)

Power BI comes with many out-of-the box Power BI visuals. These visuals are available in the visualization pane of both [Power BI Desktop](#) and [Power BI service](#), and can be used for creating and editing Power BI content.



Many more Power BI visuals are available from the Microsoft [AppSource](#) or through Power BI. These visuals are created by Microsoft and Microsoft partners, and are tested and validated by the AppSource validation team.

You can also develop your own Power BI visual, to be used by you, your organization, or the entire Power BI community.

## Default Power BI visuals

These are the out-of-the-box Power BI visuals available from the visualization pane in *Power BI Desktop* and *Power BI Service*.

To unpin a Power BI visual from the visualization pane, right-click it and select **unpin**.

To restore the default Power BI visuals in the visualization pane, click **Import a custom visual** and select **Restore default visuals**.

## AppSource Power BI visuals

Microsoft and community members contribute Power BI visuals for public benefit, and publish them to the [AppSource](#). You can download these visuals and add them to your Power BI reports. Microsoft has tested and approved these Power BI visuals for functionality and quality.

### What is AppSource?

[AppSource](#) is the place for apps, add-ins, and extensions for your Microsoft software. AppSource connects millions of users of products such as Microsoft 365, Azure, Dynamics 365, Cortana, and Power BI, to solutions that help them get work done more efficiently and insightfully than before.

### Certified Power BI visuals

Certified Power BI visuals are visuals in [AppSource](#) that meet certain specified code requirements that the Microsoft

Power BI team has tested and approved. The tests are designed to check that the visual doesn't access external services or resources.

To view the list of certified Power BI visuals, or to submit your own, see [Certified Power BI visuals](#).

### Samples for Power BI visuals

Each Power BI visual on AppSource has a data sample that illustrates how the visual works. To download the sample, in the [AppSource](#) select a Power BI visual and from the *Try a sample* section, click the **sample report** link.

## Organizational store

Power BI admins approve and deploy Power BI visuals into their organization. This allows report authors to easily discover, update, and use these Power BI visuals. Admins can easily manage these visuals with actions such as updating versions, disabling and enabling Power BI visuals.

To access the organizational store, in the *Visualization* pane click **Import a custom visual**, select **Import from marketplace** and in the top of the *Power BI visuals* window, select the **My organization** tab.

[Read more about organizational visuals](#).

## Visual files

Power BI visuals are packages that include code for rendering the data served to them. Anyone can create a custom visual and package it as a single `.pbviz` file, that can then be imported into a Power BI report.

To import a Power BI visual, in the *Visualization* pane click **Import a custom visual** and select **Import from file**.

If you are you a web developer and are interested in creating your own visual and adding it to AppSource, you can learn how to [develop a Power BI visual](#) and [publish a Power BI visual to AppSource](#).

#### WARNING

A Power BI visual could contain code with security or privacy risks. Make sure you trust the author and Power BI visual source before importing it to your report.

## Next steps

- If you're a developer, start with the [developing a Power BI visual](#) tutorial.
- Learn how a [Power BI visuals project is structured](#).
- Explore the [guidelines for Power BI visuals](#).

More questions? Try the [Frequently asked questions about Power BI visuals](#) page, or the [Power BI Community](#).

# Tutorial: Developing a Power BI visual

12/17/2019 • 11 minutes to read • [Edit Online](#)

We're enabling developers to easily add Power BI visuals into Power BI for use in dashboard and reports. To help you get started, we've published the code for all of our visualizations to GitHub.

Along with the visualization framework, we've provided our test suite and tools to help the community build high-quality Power BI visuals for Power BI.

This tutorial shows you how to develop a Power BI custom visual named Circle Card to display a formatted measure value inside a circle. The Circle Card visual supports customization of fill color and thickness of its outline.

In the Power BI Desktop report, the cards are modified to become Circle Cards.



In this tutorial, you learn how to:

- Create a Power BI custom visual.
- Develop the custom visual with D3 visual elements.
- Configure data binding with the visual elements.
- Format data values.

## Prerequisites

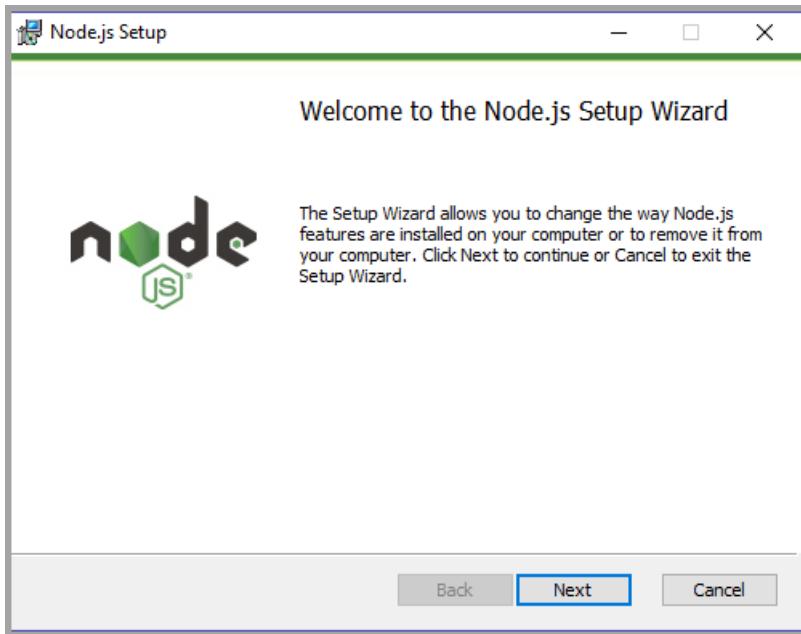
- If you're not signed up for **Power BI Pro**, [sign up for a free trial](#) before you begin.
- You need [Visual Studio Code](#) installed.
- You need [Windows PowerShell](#) version 4 or later for windows users OR the [Terminal](#) for OSX users.

## Setting up the developer environment

In addition to the prerequisites, there are a few more tools you need to install.

### Installing node.js

1. To install Node.js, in a web browser, navigate to [Node.js](#).
2. Download the latest feature MSI installer.
3. Run the installer, and then follow the installation steps. Accept the terms of the license agreement and all defaults.



4. Restart the computer.

## Installing packages

Now you need to install the **pbviz** package.

1. Open Windows PowerShell after the computer has been restarted.
2. To install pbviz, enter the following command.

```
npm i -g powerbi-visuals-tools
```

## Creating and installing a certificate

### Windows

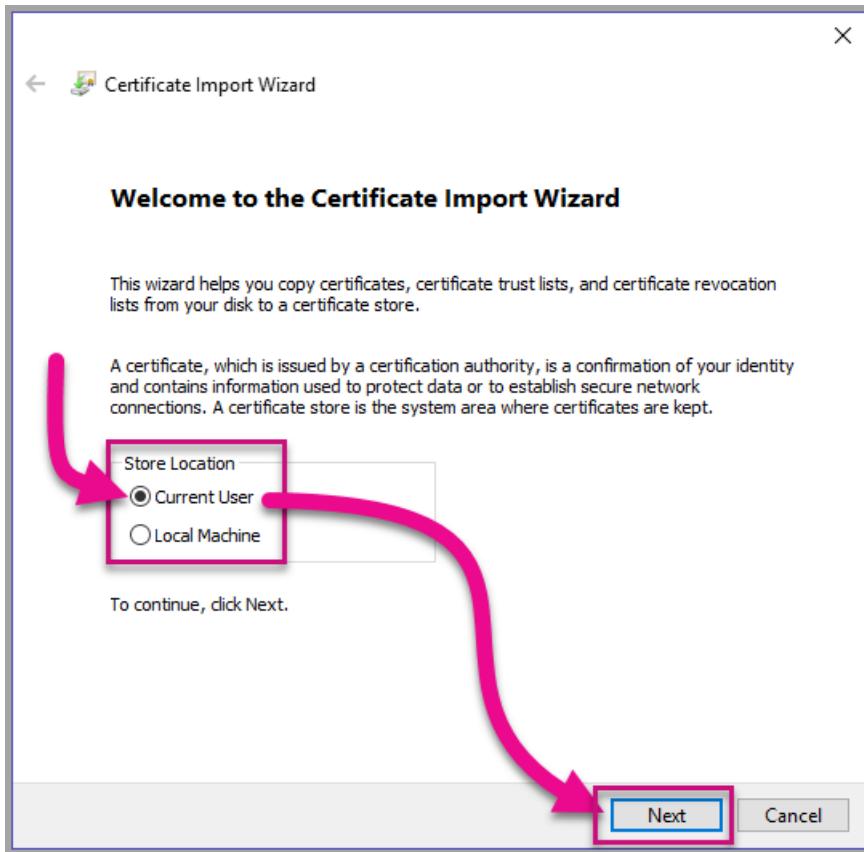
1. To create and install a certificate, enter the following command.

```
pbviz --install-cert
```

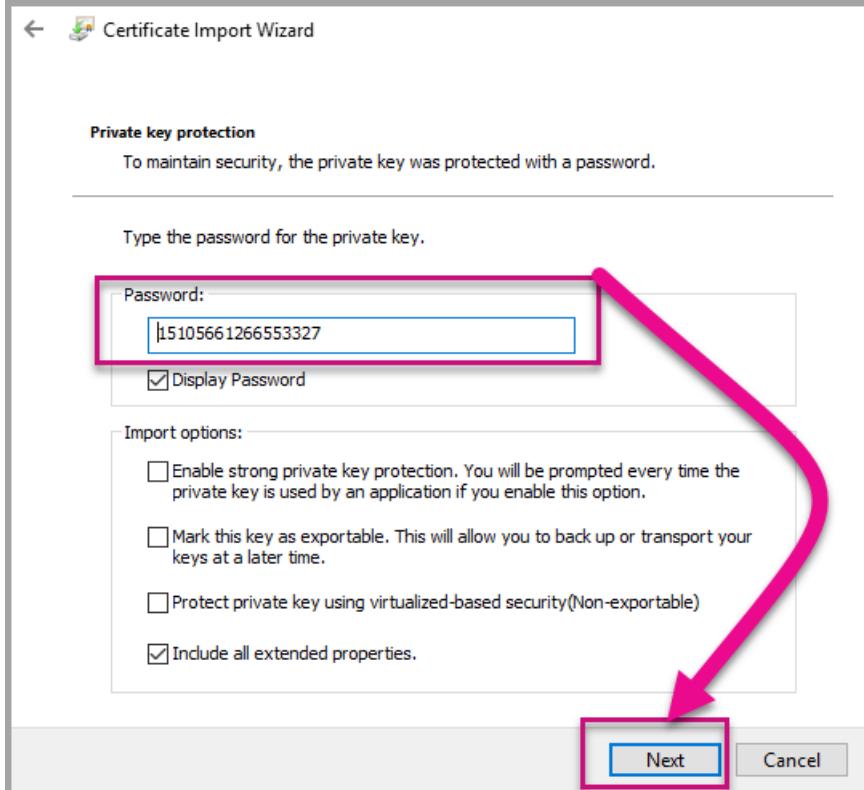
It returns a result that produces a *passphrase*. In this case, the *passphrase* is **15105661266553327**. It also starts the Certificate Import Wizard.

```
PS C:\Users\maghan> pbviz --create-cert
info  Certificate generated. Location is C:\Users\maghan\AppData\Roaming\npm\node_modules\powerbi-visuals-tools\certs\PowerBICustomVisualTest_public.pfx. Passphrase is '15105661266553327'
PS C:\Users\maghan>
```

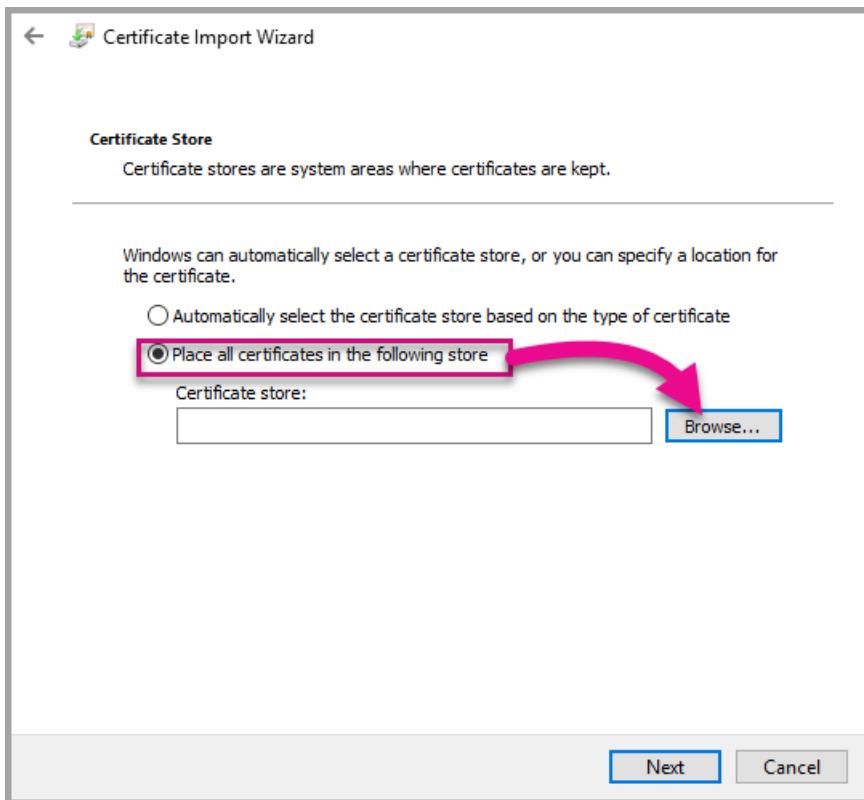
2. In the Certificate Import Wizard, verify that the store location is set to Current User. Then select *Next*.



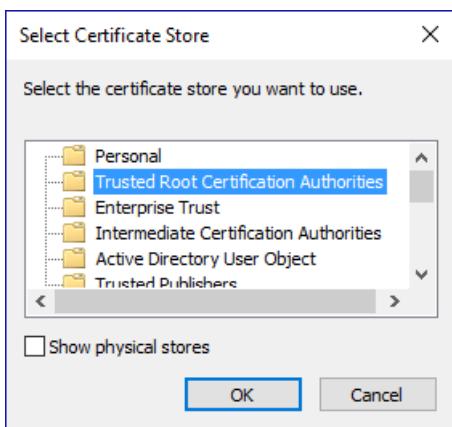
3. At the **File to Import** step, select *Next*.
4. At the **Private Key Protection** step, in the Password box, paste the passphrase you received from creating the cert. Again, in this case it is **15105661266553327**.



5. At the **Certificate Store** step, select the **Place all certificates in the Following store** option. Then select *Browse*.



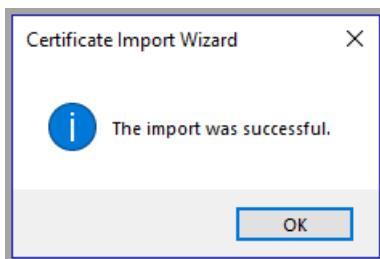
6. In the Select Certificate Store window, select Trusted Root Certification Authorities and then select OK. Then select Next on the Certificate Store screen.



7. To complete the import, select Finish.
8. If you receive a security warning, select Yes.



- When notified that the import was successful, select OK.

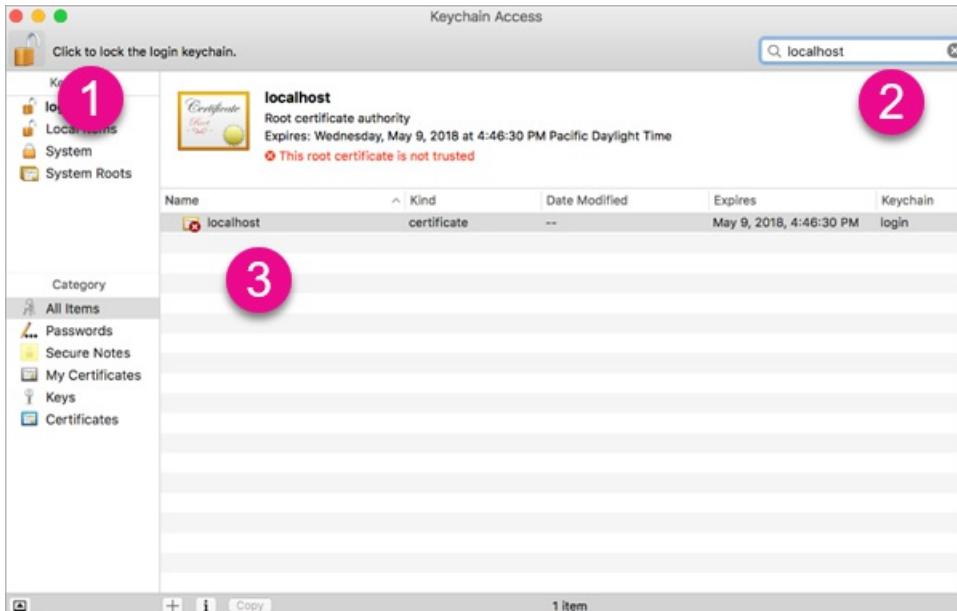


#### IMPORTANT

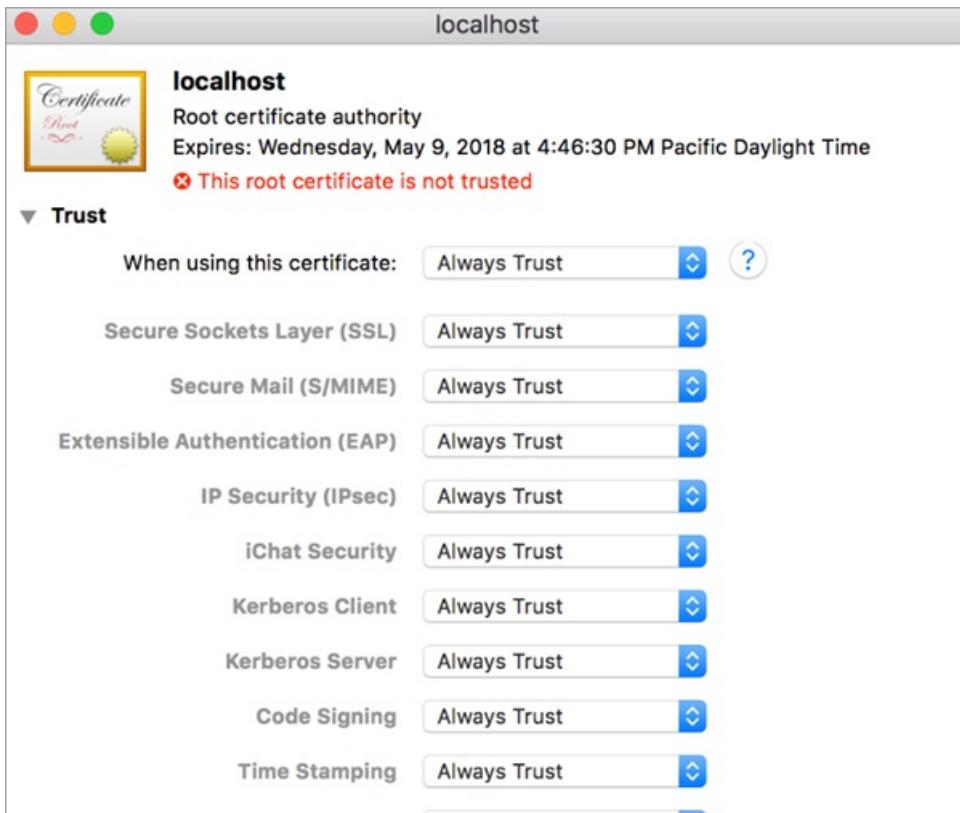
Do not close the Windows PowerShell session.

#### OSX

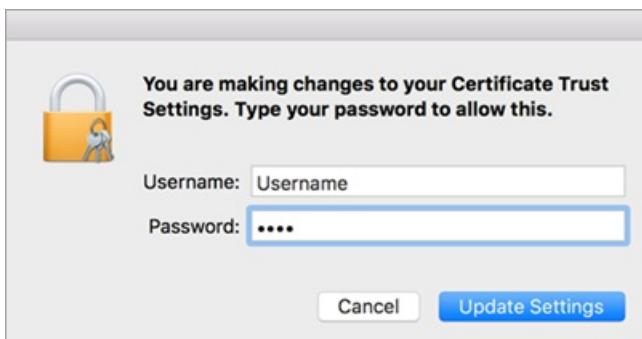
- If the lock in the upper left is locked, select it to unlock. Search for *localhost* and double-click on the certificate.



- Select **Always Trust** and close the window.



3. Enter your username and password. Select **Update Settings**.



4. Close any browsers that you have open.

**NOTE**

If the certificate is not recognized, you may need to restart your computer.

## Creating a custom visual

Now that you have set up your environment, it is time to create your custom visual.

You can [download](#) the full source code for this tutorial.

1. Verify that the Power BI Visual Tools package has been installed.

```
pbviz
```

You should see the help output.

```
+syys0+/  
oms/+osyhdhyo/  
ym/      /+oshddhys+/  
ym/          /+oyhddhyo+/  
ym/              /osyhdho  
ym/                  sm+  
ym/          yddy      om+  
ym/          shho /mmmm/    om+  
/     oys/ +mmmm /mmmm/    om+  
oso  ommmh +mmmm /mmmm/    om+  
ymmyy smmmh +mmmm /mmmm/    om+  
ymmyy smmmh +mmmm /mmmm/    om+  
ymmyy smmmh +mmmm /mmmm/    om+  
+dmd+ smmmh +mmmm /mmmm/    om+  
/hmdo +mmmm /mmmm/ /so+//ym/  
/dmmh /mmmm/ /osyhh/  
//   dmd+  
++
```

#### PowerBI Custom Visual Tool

Usage: pbviz [options] [command]

##### Commands:

new [name]	Create a new visual
info	Display info about the current visual
start	Start the current visual
package	Package the current visual into a pbviz file
update [version]	Updates the api definitions and schemas in the current visual. Changes the version if specified
help [cmd]	display help for [cmd]

##### Options:

-h, --help	output usage information
-V, --version	output the version number
--install-cert	Install localhost certificate

2. Review the output, including the list of supported commands.

```

PS                               pbviz

+syyso+/ 
oms/+osyhdhyso/
ym/      /+oshddhys+/ 
ym/      /+oyhddhyo+/ 
ym/          /osyhdho
ym/          sm+
ym/          yddy   om+
ym/          shho   /mmmm/ om+
/        oys/ +mmm/ /mmmm/ om+
oso  ommmh +mmmm /mmmm/ om+
ymmyy smmmh +mmmm /mmmm/ om+
ymmyy smmmh +mmmm /mmmm/ om+
ymmyy smmmh +mmmm /mmmm/ om+
+dmd+ smmmh +mmmm /mmmm/ om+
/hmdo +mmmm /mmmm/ /so+/ym/
/dmmrh /mmmm/ /osyhhy/
//      dmmrd
++


PowerBI Custom Visual Tool

Usage: pbviz [options] [command]

Commands:

new [name]      Create a new visual
info           Display info about the current visual
start          Start the current visual
package         Package the current visual into a pbviz file
validate [path] Validate pbviz file for submission
update [version] Updates the api definitions and schemas in the current visual. Changes the version if specified
help [cmd]       display help for [cmd]

Options:

-h, --help      output usage information
-V, --version   output the version number
--create-cert  Create new localhost certificate
--install-cert Install localhost certificate

```

- To create a custom visual project, enter the following command. CircleCard is the name of the project.

```
pbviz new CircleCard
```

```

2.1.0
PS C:\> pbviz new CircleCard
info Creating new visual
info Installing packages...
info Installed packages.
done Visual creation complete

```

#### NOTE

You create the new project at the current location of the prompt.

- Navigate to the project folder.

```
cd CircleCard
```

- Start the custom visual. Your CircleCard visual is now running while being hosted on your computer.

```
pbviz start
```

```

PS C:\> cd CircleCard
PS C:\CircleCard> pbviz start
info Building visual...
done build complete

info Starting server...
info Server listening on port 8080.

```

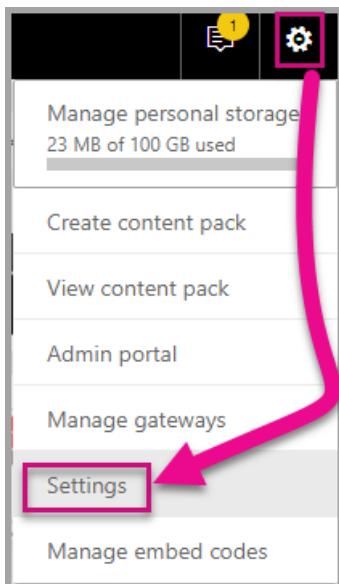
**IMPORTANT**

Do not close the Windows PowerShell session.

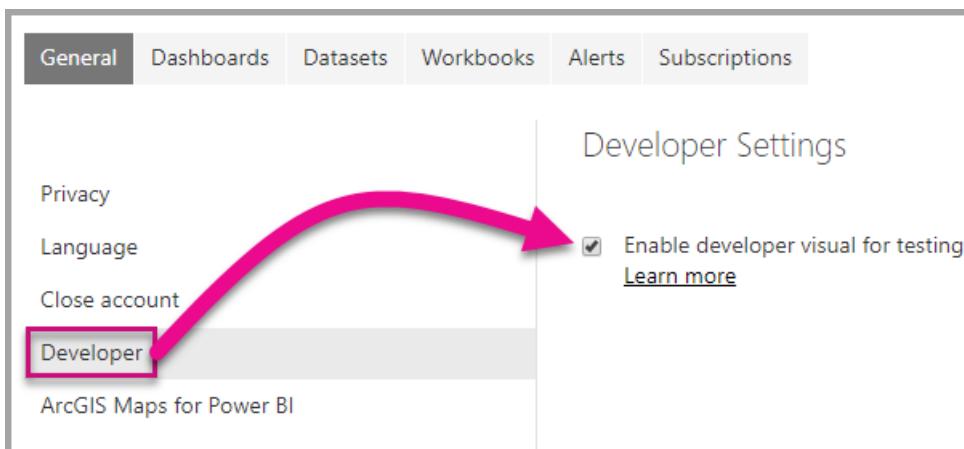
**Testing the custom visual**

In this section, we are going to test the CircleCard custom visual by uploading a Power BI Desktop report and then editing the report to display the custom visual.

1. Sign in to [PowerBI.com](#) > go to the Gear icon > then select **Settings**.



2. Select **Developer** then check the **Enable Developer Visual for testing** checkbox.



3. Upload a Power BI Desktop report.

Get Data > Files > Local File.

You can [download](#) a sample Power BI Desktop report if you do not have one created already.

Shared with me

Workspaces >

Custom visual test ^

DASHBOARDS  
US\_Sales\_Analysis.pbix

REPORTS  
US\_Sales\_Analysis

WORKBOOKS  
You have no workbooks

DATASETS  
US\_Sales\_Analysis

Get Data

Discover content

My organization

Discover apps published by other people in your organization.

Get

Services

Choose apps from online services that you use.

Get

Create new content

Files

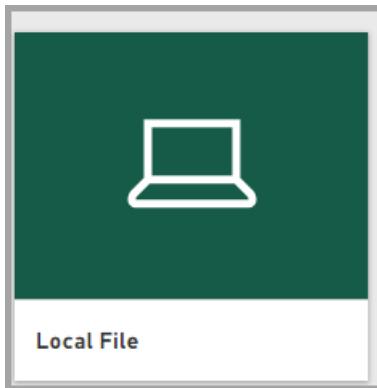
Bring in your reports, workbooks, or data from Excel, Power BI Desktop or CSV files.

Get

Databases

Use Power BI Desktop to connect to data in Azure SQL Database and more.

Get



Now to view the report, select US\_Sales\_Analysis from the Report section in the nav pane on the left.

Power BI Custom visual test > US\_Sales\_Analysis.pbix > US\_Sales\_Analysis

File View Edit report Explore Refresh Pin Live Page

Home (preview)

Favorites >

Recent >

Apps

Shared with me

Workspaces >

Custom visual test ^

DASHBOARDS  
US\_Sales\_Analysis.pbix

REPORTS  
US\_Sales\_Analysis ...

WORKBOOKS  
You have no workbooks

DATASETS  
US\_Sales\_Analysis

DATAFLOWS  
You have no dataflows

Get Data

Regional Sales Analysis

Tailspin Toys

Year  
CY2017

Category  
Select All, Co-Axial, Collective pitch, Fixed pitch, Glider, Trainer, Warbird

256,396  
Quantity

100.00 %  
Sales % All Regions

Sales and Avg Price by Month

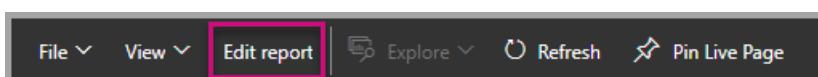
Sales Avg Price

Sales by Demographic

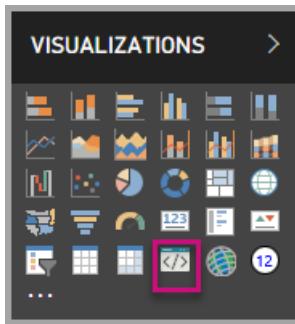
Professional \$54M, Intermediate \$13M, Advanced \$9M, Novice \$1M, Beginner \$1M

4. Now you need to edit the report while in the Power BI service.

Go to **Edit report**.



5. Select the **Developer Visual** from the **Visualizations** pane.



#### NOTE

This visualization represents the custom visual that you started on your computer. It is only available when the developer settings have been enabled.

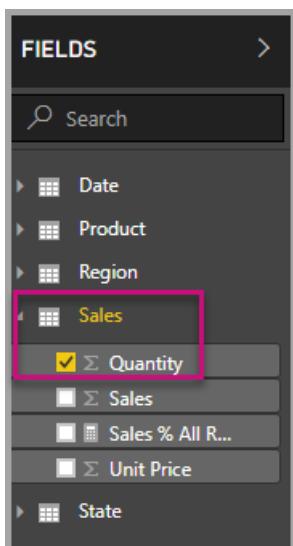
6. Notice that a visualization was added to the report canvas.



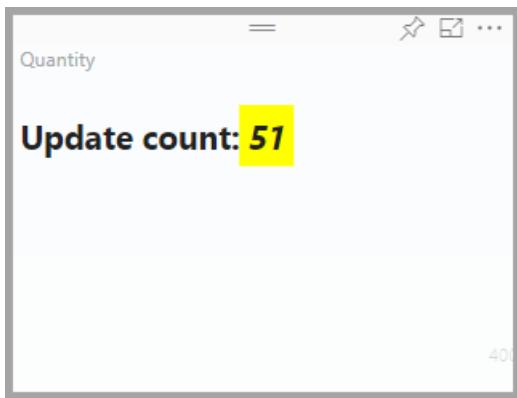
#### NOTE

This is a very simple visual that displays the number of times its Update method has been called. At this stage, the visual does not yet retrieve any data.

7. While selecting the new visual in the report, Go to the Fields Pane > expand Sales > select Quantity.



8. Then to test the new visual, resize the visual and notice the update value increments.



To stop the custom visual running in PowerShell, enter **Ctrl+C**. When prompted to terminate the batch job, enter **Y**, then press **Enter**.

## Adding visual elements

Now you need to install the **D3 JavaScript library**. D3 is a JavaScript library for producing dynamic, interactive data visualizations in web browsers. It makes use of widely implemented SVG HTML5, and CSS standards.

Now you can develop the custom visual to display a circle with text.

### NOTE

Many text entries in this tutorial can be copied from [here](#).

1. To install the **D3 library** in PowerShell, enter the command below.

```
npm i d3@^5.0.0 --save
```

```
PS C:\circlecard>npm i d3@^5.0.0 --save
+ d3@5.11.0
added 179 packages from 169 contributors and audited 306 packages in 33.25s
found 0 vulnerabilities
```

```
PS C:\circlecard>
```

2. To install type definitions for the **D3 library**, enter the command below.

```
npm i @types/d3@^5.0.0 --save
```

```
PS C:\circlecard>npm i @types/d3@^5.0.0 --save
+ @types/d3@5.7.2
updated 1 package and audited 306 packages in 2.217s
found 0 vulnerabilities
```

```
PS C:\circlecard>
```

This command installs TypeScript definitions based on JavaScript files, enabling you to develop the custom visual in TypeScript (which is a superset of JavaScript). Visual Studio Code is an ideal IDE for developing TypeScript applications.

3. To install the **core-js** in PowerShell, enter the command below.

```
npm i core-js@3.2.1 --save
```

```
PS C:\circlecard> npm i core-js@3.2.1 --save

> core-js@3.2.1 postinstall F:\circlecard\node_modules\core-js
> node scripts/postinstall || echo "ignore"

Thank you for using core-js ( https://github.com/zloirock/core-js ) for polyfilling JavaScript standard
library!

The project needs your help! Please consider supporting of core-js on Open Collective or Patreon:
> https://opencollective.com/core-js
> https://www.patreon.com/zloirock

+ core-js@3.2.1
updated 1 package and audited 306 packages in 6.051s
found 0 vulnerabilities

PS C:\circlecard>
```

This command installs modular standard library for JavaScript. It includes polyfills for ECMAScript up to 2019. Read more about [core-js](#)

4. To install the **powerbi-visuals-api** in PowerShell, enter the command below.

```
npm i powerbi-visuals-api --save-dev
```

```
PS C:\circlecard>npm i powerbi-visuals-api --save-dev

+ powerbi-visuals-api@2.6.1
updated 1 package and audited 306 packages in 2.139s
found 0 vulnerabilities

PS C:\circlecard>
```

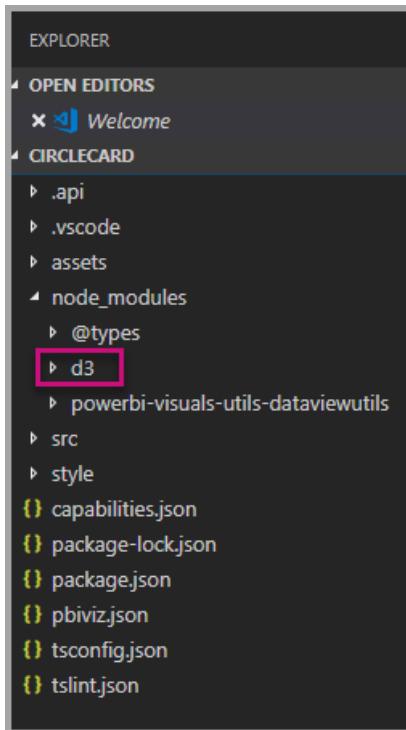
This command installs Power BI Visuals API definitions.

5. Launch **Visual Studio Code**.

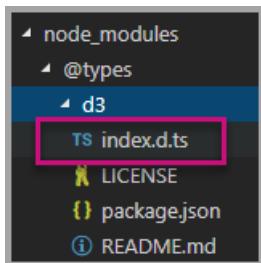
You can launch **Visual Studio Code** from PowerShell by using the following command.

```
code .
```

6. In the **Explorer pane**, expand the **node\_modules** folder to verify that the **d3** library was installed.



7. Make sure that file **index.d.ts** was added, by expanding node\_modules > @types > d3 in the **Explorer pane**.



## Developing the visual elements

Now we can explore how to develop the custom visual to show a circle and sample text.

1. In the **Explorer pane**, expand the **src** folder and then select **visual.ts**.

### NOTE

Notice the comments at the top of the **visual.ts** file. Permission to use the Power BI custom visual packages is granted free of charge under the terms of the MIT License. As part of the agreement, you must leave the comments at the top of the file.

2. Remove the following default custom visual logic from the Visual class.

- The four class-level private variable declarations.
- All lines of code from the constructor.
- All lines of code from the update method.
- All remaining lines within the module, including the parseSettings and enumerateObjectInstances methods.

Verify that the module code looks like the following.

```

"use strict";
import "core-js/stable";
import "../style/visual.less";
import powerbi from "powerbi-visuals-api";
import IVisual = powerbi.extensibility.IVisual;
import VisualConstructorOptions = powerbi.extensibility.visual.VisualConstructorOptions;
import VisualUpdateOptions = powerbi.extensibility.visual.VisualUpdateOptions;
import EnumerateVisualObjectInstancesOptions = powerbi.EnumerateVisualObjectInstancesOptions;
import VisualObjectInstanceEnumeration = powerbi.VisualObjectInstanceEnumeration;
import IVisualHost = powerbi.extensibility.visual.IVisualHost;

import * as d3 from "d3";
type Selection<T extends d3.BaseType> = d3.Selection<T, any, any, any>;

export class Visual implements IVisual {

    constructor(options: VisualConstructorOptions) {

    }

    public update(options: VisualUpdateOptions) {

    }
}

```

3. Beneath the *Visual*/class declaration, insert the following class-level properties.

```

export class Visual implements IVisual {
    // ...
    private host: IVisualHost;
    private svg: Selection<SVGElement>;
    private container: Selection<SVGElement>;
    private circle: Selection<SVGElement>;
    private textValue: Selection<SVGElement>;
    private textLabel: Selection<SVGElement>;
    // ...
}

export class Visual implements IVisual {
    private host: IVisualHost;
    private svg: Selection<SVGElement>;
    private container: Selection<SVGElement>;
    private circle: Selection<SVGElement>;
    private textValue: Selection<SVGElement>;
    private textLabel: Selection<SVGElement>;

    constructor(options: VisualConstructorOptions) {

    }

    public update(options: VisualUpdateOptions) {

    }
}

```

4. Add the following code to the *constructor*.

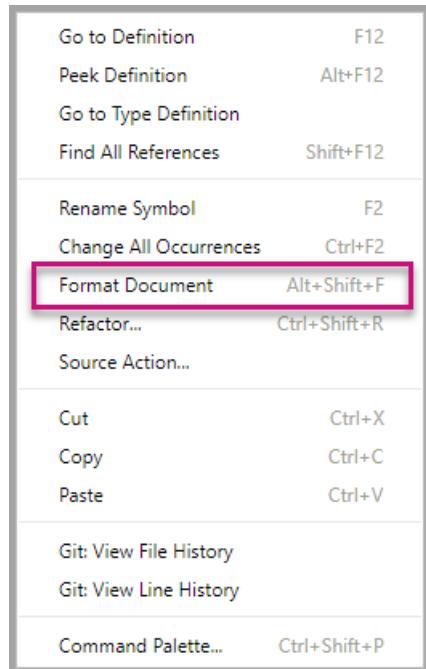
```

this.svg = d3.select(options.element)
  .append('svg')
  .classed('circleCard', true);
this.container = this.svg.append("g")
  .classed('container', true);
this.circle = this.container.append("circle")
  .classed('circle', true);
this.textValue = this.container.append("text")
  .classed("textValue", true);
this.textLabel = this.container.append("text")
  .classed("textLabel", true);

```

This code adds an SVG group inside the visual and then adds three shapes: a circle and two text elements.

To format the code in the document, right-select anywhere in the **Visual Studio Code document**, and then select **Format Document**.



To improve readability, it is recommended that you format the document every time that paste in code snippets.

5. Add the following code to the *update* method.

```

let width: number = options.viewport.width;
let height: number = options.viewport.height;
this.svg.attr("width", width);
this.svg.attr("height", height);
let radius: number = Math.min(width, height) / 2.2;
this.circle
  .style("fill", "white")
  .style("fill-opacity", 0.5)
  .style("stroke", "black")
  .style("stroke-width", 2)
  .attr("r", radius)
  .attr("cx", width / 2)
  .attr("cy", height / 2);
let fontSizeValue: number = Math.min(width, height) / 5;
this.textValue
  .text("Value")
  .attr("x", "50%")
  .attr("y", "50%")
  .attr("dy", "0.35em")
  .attr("text-anchor", "middle")
  .style("font-size", fontSizeValue + "px");
let fontSizeLabel: number = fontSizeValue / 4;
this.textLabel
  .text("Label")
  .attr("x", "50%")
  .attr("y", height / 2)
  .attr("dy", fontSizeValue / 1.2)
  .attr("text-anchor", "middle")
  .style("font-size", fontSizeLabel + "px");

```

*This code sets the width and height of the visual, and then initializes the attributes and styles of the visual elements.*

6. Save the **visual.ts** file.
  7. Select the **capabilities.json** file.
- At line 14, remove the entire objects element (lines 14-60).
8. Save the **capabilities.json** file.
  9. In PowerShell, start the custom visual.

```
pbviz start
```

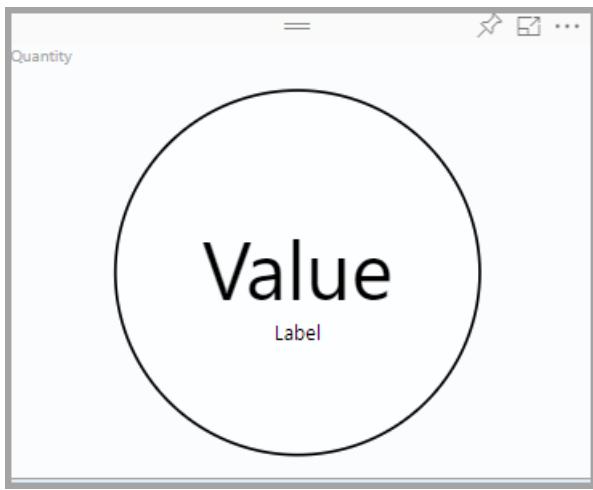
#### Toggle auto reload

1. Navigate back to the Power BI report.
2. In the toolbar floating above the developer visual, select the **Toggle Auto Reload**.



This option ensures that the visual is automatically reloaded each time you save project changes.

3. From the **Fields pane**, drag the **Quantity** field into the developer visual.
4. Verify that the visual looks like the following.



5. Resize the visual.

Notice that the circle and text value scales to fit the available dimension of the visual.

The update method is called continuously with resizing the visual, and it results in the fluid rescaling of the visual elements.

You have now developed the visual elements.

6. Continue running the visual.

## Process data in the visual code

Define the data roles and data view mappings, and then modify the custom visual logic to display the value and display name of a measure.

### Configuring the capabilities

Modify the **capabilities.json** file to define the data role and data view mappings.

1. In Visual Studio code, in the **capabilities.json** file, from inside the **dataRoles** array, remove all content (lines 3-12).
2. Inside the **dataRoles** array, insert the following code.

```
{  
    "displayName": "Measure",  
    "name": "measure",  
    "kind": "Measure"  
}
```

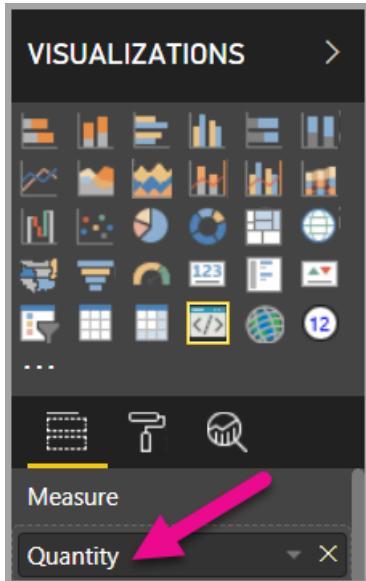
The **dataRoles** array now defines a single data role of type **measure**, that is named **measure**, and displays as **Measure**. This data role allows passing either a measure field, or a field that is summarized.

3. From inside the **dataViewMappings** array, remove all content (lines 10-31).
4. Inside the **dataViewMappings** array, insert the following content.

```
{
  "conditions": [
    { "measure": { "max": 1 } }
  ],
  "single": {
    "role": "measure"
  }
}
```

The **dataViewMappings** array now defines one field can be passed to the data role named **measure**.

5. Save the **capabilities.json** file.
6. In Power BI, notice that the visual now can be configured with **Measure**.



#### NOTE

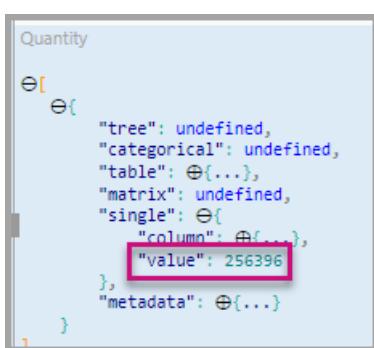
The visual project does not yet include data binding logic.

### Exploring the dataview

1. In the toolbar floating above the visual, select **Show Dataview**.



2. Expand down into **single**, and then notice the value.



3. Expand down into **metadata**, and then into the **columns** array, and in particular notice the **format** and **displayName** values.

```

Quantity
{
  "tree": undefined,
  "categorical": undefined,
  "table": {...},
  "matrix": undefined,
  "single": {
    "column": {
      "roles": {...},
      "type": {...},
      "format": "#,0",
      "displayName": "Quantity",
      "queryName": "Sum(Sales.Quantity)",
      "expr": {...},
      "index": 0,
      "isMeasure": true
    }
  }
}

```

4. To toggle back to the visual, in the toolbar floating above the visual, select Show DataView.



### Consume data in the visual code

1. In Visual Studio Code, in the `visual.ts` file,

import the `DataView` interface from `powerbi` module

```
import DataView = powerbi.DataView;
```

and add the following statement as the first statement of the `update` method.

```
let dataView: DataView = options.dataViews[0];
```

```
public update(options: VisualUpdateOptions) {
  let dataView: DataView = options.dataViews[0];
  let width: number = options.viewport.width;
  let height: number = options.viewport.height;
```

This statement assigns the `dataView` to a variable for easy access, and declares the variable to reference the `dataView` object.

2. In the `update` method, replace `.text("Value")` with the following.

```
.text(<string>dataView.single.value)
```

```
let fontSizeValue: number = Math.min(width, height) / 5;
this.textValue
  .text(dataView.single.value as string)
  .attr({
    x: "50%",
    y: "50%",
    dy: "0.35em",
```

3. In the `update` method, replace `.text("Label")` with the following.

```
.text(dataView.metadata.columns[0].displayName)
```

```
let fontSizeLabel: number = fontSizeValue / 4;
this.textLabel
    .text(dataView.metadata.columns[0].displayName)
    .attr({
        x: "50%",
        y: height / 2,
        dy: fontSizeValue / 1.2,
```

4. Save the `visual.ts` file.
5. In Power BI, review the visual, which now displays the value and the display name.

You have now configured the data roles and bound the visual to the dataview.

In the next tutorial you learn how to add formatting options to the custom visual.

## Debugging

For tips about debugging your custom visual, see the [debugging guide](#).

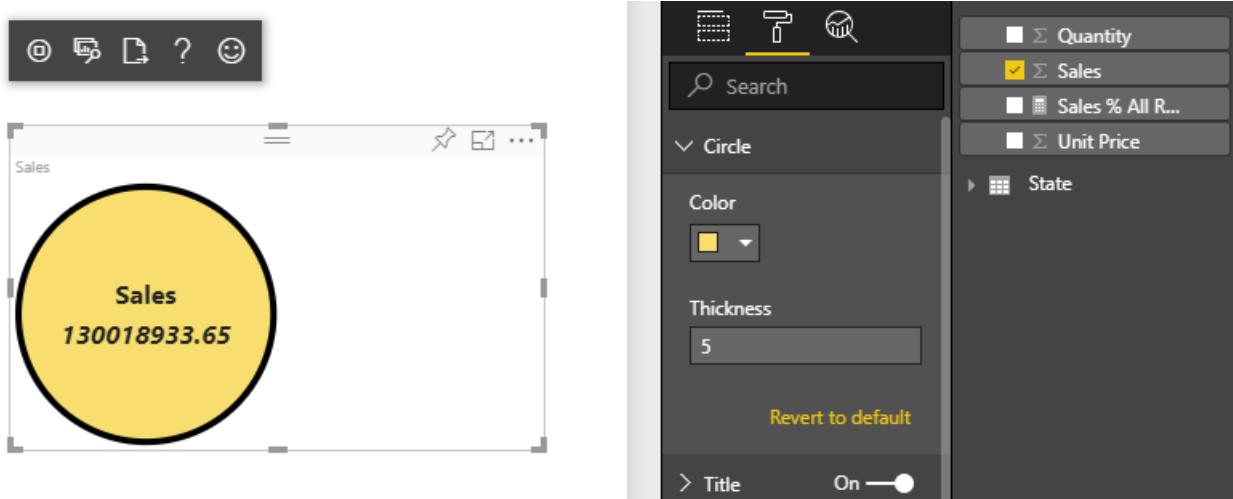
## Next steps

[Adding formatting options](#)

# Tutorial: Create a React-based visual

5/11/2020 • 7 minutes to read • [Edit Online](#)

This tutorial explains how to create a Power BI visual using [React](#). The visual displays a value in a circle. The visual has adaptive size and settings to customize it. With the information in this article, you can create your own Power BI visuals with React.



In this tutorial, you learn how to:

- Set up your development environment
- Create a React visual
- Configure capabilities for the visual
- Render data from Power BI
- Resize the visual
- Make the visual customizable

## Prerequisites

- A Power BI Pro account. [Sign up for a free trial](#) before you begin.
- [Visual Studio Code](#).
- [Windows PowerShell](#) version 4 or later for windows users OR the [Terminal](#) for OSX users.
- An environment as described in [Setting up the developer environment](#).

## Getting started

To begin, create a minimal Power BI visual by using `pbviz`. For more information about projects and project structure, see [Power BI visual project structure](#). For the full source code of this visual, see [Circle Card React Visual](#).

You can clone or download the full source code of the visual from [GitHub](#).

1. Open PowerShell and run the following command:

```
pbviz new ReactCircleCard
```

The command creates a folder called *ReactCircleCard*.

2. Change directories to that folder and open Visual Studio Code.

```
cd ./ReactCircleCard  
code .
```

3. Start the developer server for your visual.

```
pbviz start
```



This basic visual represents updates count. Let's transform it to a circle card at the next step.

## Change the visual to a circle card

This basic visual represents an updates count. Next, transform it to a circle card, which represents a measure and its title.

1. Run the following command to install required dependencies:

```
npm i react react-dom
```

2. Run the following command to install React 16 and corresponding versions of `react-dom` and typings:

```
npm i @types/react @types/react-dom
```

3. Create a React component class. In Visual Studio Code, select **File > New File**. Copy the following code into the file.

```
import * as React from "react";  
  
export class ReactCircleCard extends React.Component<{}>{  
    render(){  
        return (  
            <div className="circleCard">  
                Hello, React!  
            </div>  
        )  
    }  
}  
  
export default ReactCircleCard;
```

4. Select **Save As**. Go to the `src` directory. Enter the name `component`. For **Save as type**, select **TypeScript React**.

5. Open `src/visual.ts`. Replace the current code with the following code:

```

"use strict";
import powerbi from "powerbi-visuals-api";

import DataView = powerbi.DataView;
import VisualConstructorOptions = powerbi.extensibility.visual.VisualConstructorOptions;
import VisualUpdateOptions = powerbi.extensibility.visual.VisualUpdateOptions;
import IVisual = powerbi.extensibility.visual.IVisual;

import "./../style/visual.less";

export class Visual implements IVisual {

    constructor(options: VisualConstructorOptions) {

    }

    public update(options: VisualUpdateOptions) {

    }
}

```

6. Import React dependencies and the component you just added.

```

import * as React from "react";
import * as ReactDOM from "react-dom";
...
import ReactCircleCard from "./component";

```

Default Power BI TypeScript settings don't take React *tsx* files. Visual Studio Code highlights `component` as an error.

7. Open the file *tsconfig.json* and add two lines to the beginning of the `compilerOptions` item.

```
{
  "compilerOptions": {
    "jsx": "react",
    "types": ["react", "react-dom"],
    //...
  }
}
```

The error on `component` should be gone.

To render the component, add the target HTML element. This element is `HTMLElement` in `VisualConstructorOptions`, which is passed into constructor.

8. Modify the `visual` class, as in the following code:

```

private target: HTMLElement;
private reactRoot: React.ComponentElement<any, any>;

constructor(options: VisualConstructorOptions) {
    this.reactRoot = React.createElement(ReactCircleCard, {});
    this.target = options.element;

    ReactDOM.render(this.reactRoot, this.target);
}

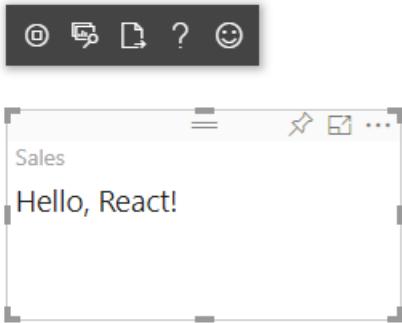
```

9. Save the changes and run the existing code by using this command:

```
pbviz start
```

#### NOTE

If you previously ran `pbviz`, you must restart it to apply changes in `tsconfig.json`.



## Configure capabilities

You can configure the capabilities of the visual.

1. Open `capabilities.json`. Remove the `Category Data` object from `dataRoles`. The `ReactCircleCard` displays a single value, so we need only `Measure Data`. The `dataRoles` key now looks like this:

```
"dataRoles": [  
    {  
        "displayName": "Measure Data",  
        "name": "measure",  
        "kind": "Measure"  
    }  
,
```

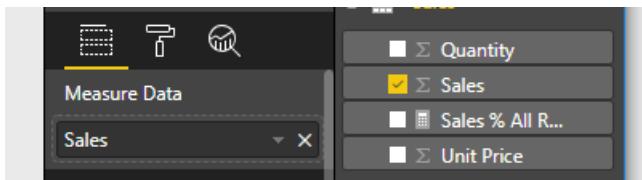
2. Remove all the content of `objects` key. You'll fill it in later.

```
"objects": {},
```

3. Copy the following code of `dataViewMappings` property. The value of `max: 1` means that the only one measure column can be submitted.

```
"dataViewMappings": [  
    {  
        "conditions": [  
            {  
                "measure": {  
                    "max": 1  
                }  
            }  
        ],  
        "single": {  
            "role": "measure"  
        }  
    }  
,
```

Now you can bring data from the `Fields` pane into the visual settings.



## Receive properties from Power BI

You can render data using React. The component can display data from its own state.

1. Modify `src/component.tsx`.

```
export interface State {
    textLabel: string,
    textValue: string
}

export const initialState: State = {
    textLabel: "",
    textValue: ""
}

export class ReactCircleCard extends React.Component<{}, State>{
    constructor(props: any){
        super(props);
        this.state = initialState;
    }

    render(){
        const { textLabel, textValue } = this.state;

        return (
            <div className="circleCard">
                <p>
                    {textLabel}
                    <br/>
                    <em>{textValue}</em>
                </p>
            </div>
        )
    }
}
```

2. Add styles for new markup by editing `styles/visual.less`.

```

.circleCard {
    position: relative;
    box-sizing: border-box;
    border: 1px solid #000;
    border-radius: 50%;
    width: 200px;
    height: 200px;
}

p {
    text-align: center;
    line-height: 30px;
    font-size: 20px;
    font-weight: bold;

    position: relative;
    top: -30px;
    margin: 50% 0 0 0;
}

```

3. Visuals receive current data as an argument of the `update` method. Open `src/visual.ts` and add code to `ReactCircleCard.update`.

```

//...
import { ReactCircleCard, initialState } from "./component";
//...

export class Visual implements IVisual {
    //...
    public update(options: VisualUpdateOptions) {

        if(options.dataViews && options.dataViews[0]){
            const dataView: DataView = options.dataViews[0];

            ReactCircleCard.update({
                textLabel: dataView.metadata.columns[0].displayName,
                textValue: dataView.single.value.toString()
            });
        }
        } else {
            this.clear();
        }
    }

    private clear() {
        ReactCircleCard.update(initialState);
    }
}

```

The code selects `textLabel` and `textValue` from `DataView` and, if the data exists, updates the component state.

4. To send updates to component instance, insert the following code in the `ReactCircleCard` class:

```

private static updateCallback: (data: object) => void = null;

public static update(newState: State) {
    if(typeof ReactCircleCard.updateCallback === 'function'){
        ReactCircleCard.updateCallback(newState);
    }
}

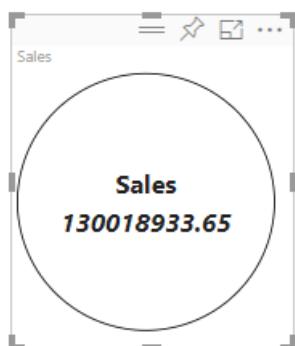
public state: State = initialState;

public componentWillMount() {
    ReactCircleCard.updateCallback = (newState: State): void => { this.setState(newState); };
}

public componentWillUnmount() {
    ReactCircleCard.updateCallback = null;
}

```

5. Test the visual. Make sure that `pbviz start` has been run, and save all files. Refresh the visual.



## Make component resizable

In this section, you make the component resizable. Currently, the component has fixed width and height.

Get the current size of the visual viewport from the `options` object.

1. Open `src/visual.ts`. Import the `IViewport` interface and add the `viewport` property to the `visual` class.

```

import IViewport = powerbi.IViewport;

//...

export class Visual implements IVisual {
    private viewport: IViewport;
    //...
}

```

2. Add the following code to the `update` method of `visual`.

```

if (options.dataViews && options.dataViews[0]) {
    const dataView: DataView = options.dataViews[0];

    this.viewport = options.viewport;
    const { width, height } = this.viewport;
    const size = Math.min(width, height);

    ReactCircleCard.update({
        size,
        //...
    });
}

```

3. Add properties to the `State` interface in `src/component.tsx`.

```

export interface State {
    //...
    size: number
}

const initialState: State = {
    //...
    size: 200
}

```

4. Make the following changes in the `render` method in `src/component.tsx`.

```

render() {
    const { textLabel, minValue, size } = this.state;

    const style: React.CSSProperties = { width: size, height: size };

    return (
        <div className="circleCard" style={style}>
            {/* ... */}
        </div>
    )
}

```

5. Replace `width` and `height` rules in `style/visual.less` with `min-width` and `min-height`.

```

min-width: 200px;
min-height: 200px;

```

Now you can resize the viewport. The circle diameter corresponds to minimal size as width or height.

## Make your Power BI visual customizable

In this section, you make the visual customizable.

1. Open `capabilities.json`. Add the following settings to the `objects` property.

```
//...
{
    "objects": {
        "circle": {
            "displayName": "Circle",
            "properties": {
                "circleColor": {
                    "displayName": "Color",
                    "description": "The fill color of the circle.",
                    "type": {
                        "fill": {
                            "solid": {
                                "color": true
                            }
                        }
                    }
                },
                "circleThickness": {
                    "displayName": "Thickness",
                    "description": "The circle thickness.",
                    "type": {
                        "numeric": true
                    }
                }
            }
        },
        "//...
    }
}
```

2. Replace existing code in *src/settings.ts* with this code:

```
"use strict";

import { DataViewObjectsParser } from "powerbi-visuals-utils-dataviewutils";
import DataViewObjectsParser = DataViewObjectsParser.DataViewObjectsParser;

export class CircleSettings {
    public circleColor: string = "white";
    public circleThickness: number = 2;
}

export class VisualSettings extends DataViewObjectsParser {
    public circle: CircleSettings = new CircleSettings();
}
```

3. Add these `import` statements at the top of *src/visual.ts*:

```
import VisualObjectInstance = powerbi.VisualObjectInstance;
import EnumerateVisualObjectInstancesOptions = powerbi.EnumerateVisualObjectInstancesOptions;
import VisualObjectInstanceEnumerationObject = powerbi.VisualObjectInstanceEnumerationObject;

import { VisualSettings } from "./settings";
```

4. Add the `enumerateObjectInstances` method to *src/visual.ts*. This method is used to apply visual settings.

```

export class Visual implements IVisual {
    private settings: VisualSettings;

    //...

    public enumerateObjectInstances(
        options: EnumerateVisualObjectInstancesOptions
    ): VisualObjectInstance[] | VisualObjectInstanceEnumerationObject {
        return VisualSettings.enumerateObjectInstances(this.settings || VisualSettings.getDefault(),
            options);
    }
}

```

5. Add code so that the `dataView` object can now receive settings.

```

public update(options: VisualUpdateOptions) {
    if(options.dataViews && options.dataViews[0]){
        //...
        this.settings = VisualSettings.parse(dataView) as VisualSettings;
        const object = this.settings.circle;

        ReactCircleCard.update({
            borderWidth: object && object.circleThickness ? object.circleThickness : undefined,
            background: object && object.circleColor ? object.circleColor : undefined,
            //...
        });
    }
}

```

6. Apply the corresponding changes to `src/component.tsx`, first by adding these values to `State`:

```

export interface State {
    //...
    background?: string,
    borderWidth?: number
}

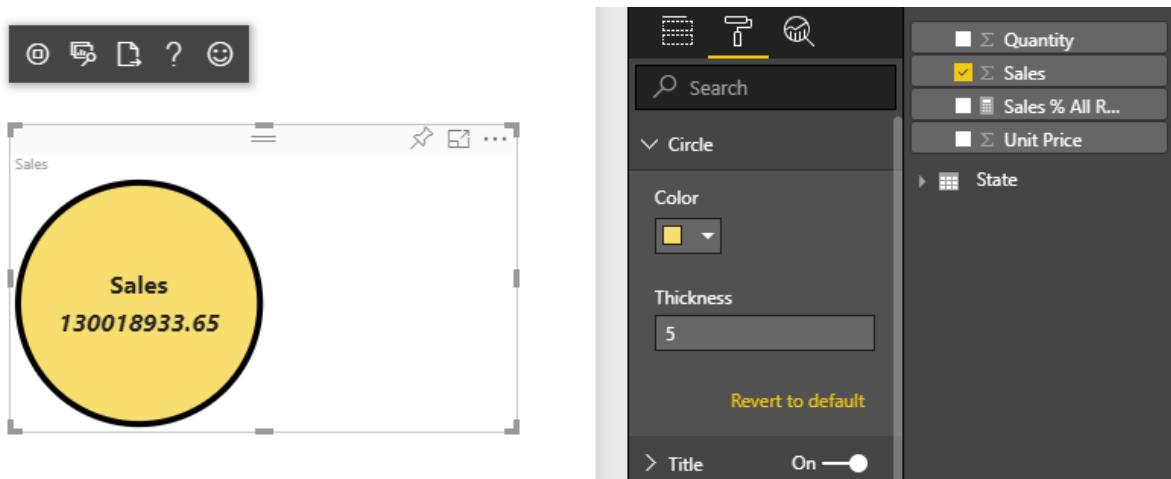
```

7. Then add the following code to the `render` method:

```

const { /*...*/ background, borderWidth } = this.state;
const style: React.CSSProperties = { /*...*/ background, borderWidth };

```



## Next steps

For more about Power BI development, see [Guidelines for Power BI visuals](#) and [Visuals in Power BI](#).

# Build a bar chart

5/11/2020 • 12 minutes to read • [Edit Online](#)

This article is a step-by-step guide for building a sample Power BI bar chart visual with code. You can get the complete code example at <https://github.com/Microsoft/PowerBI-visuals-sampleBarChart>.

## View model

It's important to define the bar chart view model first, and iterate on what's exposed to your visual as you build it.

```
/**  
 * Interface for BarCharts viewmodel.  
 *  
 * @interface  
 * @property {BarChartDataPoint[]} dataPoints - Set of data points the visual will render.  
 * @property {number} dataMax - Maximum data value in the set of data points.  
 */  
interface BarChartViewModel {  
    dataPoints: BarChartDataPoint[];  
    dataMax: number;  
};  
  
/**  
 * Interface for BarChart data points.  
 *  
 * @interface  
 * @property {number} value - Data value for the point.  
 * @property {string} category - Corresponding category of the data value.  
 */  
interface BarChartDataPoint {  
    value: number;  
    category: string;  
};
```

## Use static data

Using static data is a great way to test your visual without data binding. Your view model won't change, even after you add data binding in a later step.

```

let testData: BarChartDataPoint[] = [
  {
    value: 10,
    category: 'a'
  },
  {
    value: 20,
    category: 'b'
  },
  {
    value: 1,
    category: 'c'
  },
  {
    value: 100,
    category: 'd'
  },
  {
    value: 500,
    category: 'e'
  }];
};

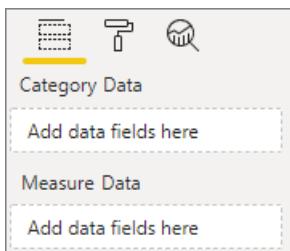
let viewModel: BarChartViewModel = {
  dataPoints: testData,
  dataMax: d3.max(testData.map((dataPoint) => dataPoint.value))
};

```

## Data binding

You add data binding by defining your visual capabilities in *capabilities.json*. The sample code already has a schema for you to use.

Data binding acts on a **Field** well in Power BI.



### Add data roles

The sample code already has data roles, but you can customize them.

- `displayName` is the name shown in the **Field** well.
- `name` is the internal name used to refer to the data role.
- `kind` is for the kind of field. *Grouping* fields (0) have discrete values. *Measure* fields (1) have numeric data values.

```

"dataRoles": [
  {
    "displayName": "Category Data",
    "name": "category",
    "kind": 0
  },
  {
    "displayName": "Measure Data",
    "name": "measure",
    "kind": 1
  }
],

```

For more information, see [Data roles](#).

## Add conditions to DataViewMapping

Define conditions within your `dataViewMappings` to set how many fields each field well can bind. Use the data role's internal `name` to refer to each field.

```

"dataViewMappings": [
  {
    "conditions": [
      {
        "category": {
          "max": 1
        },
        "measure": {
          "max": 1
        }
      }
    ],
  }
]

```

For more information, see [Data view mapping](#).

## Define and use visualTransform

The `DataView` is the structure that Power BI provides to your visual, which contains the queried data to be visualized. However, `DataView` can provide data in different forms, such as categorical and tabular. To build a categorical visual like a bar chart, you only need to use the categorical property on the `DataView`. Defining `visualTransform` lets you convert `DataView` into a view model your visual will use.

To assign colors and select them when defining individual data points, you use `IVisualHost`.

```

/**
 * Function that converts queried data into a view model that will be used by the visual
 *
 * @function
 * @param {VisualUpdateOptions} options - Contains references to the size of the container
 *                                         and the dataView which contains all the data
 *                                         the visual had queried.
 * @param {IVisualHost} host             - Contains references to the host which contains services
 */
function visualTransform(options: VisualUpdateOptions, host: IVisualHost): BarChartViewModel {
  /*Convert dataView to your viewModel*/
}

```

## Color

Color is exposed as one of the services available on `IVisualHost`.

## Add color to data points

Each data point is represented by a different color. You add color to the `BarChartDataPoint` interface.

```
/**  
 * Interface for BarChart data points.  
 *  
 * @interface  
 * @property {number} value - Data value for the point.  
 * @property {string} category - Corresponding category of the data value.  
 * @property {string} color - Color corresponding to the data point.  
 */  
interface BarChartDataPoint {  
    value: number;  
    category: string;  
    color: string;  
};
```

## The `colorPalette` service

The `colorPalette` service manages the colors used in your visual. Its instance is available on `IVisualHost`.

## Assign color to data points

You defined `visualTransform` as a construct to convert `dataView` to a view model that a bar chart can use. Because you iterate through the data points in `visualTransform`, it's also the ideal place to assign colors.

```
let colorPalette: IColorPalette = host.colorPalette; // host: IVisualHost  
for (let i = 0, len = Math.max(category.values.length, dataValue.values.length); i < len; i++) {  
    barChartDataPoints.push({  
        category: category.values[i],  
        value: dataValue.values[i],  
        color: colorPalette.getColor(category.values[i]).value,  
    });  
}
```

## Selection and interactions

Selection lets the user interact both with your visual and other visuals.

## Add selection to each data point

Since each data point is unique, add selection to each data point. You add the `selection` property on the `BarChartDataPoint` interface.

```

/**
 * Interface for BarChart data points.
 *
 * @interface
 * @property {number} value - Data value for the point.
 * @property {string} category - Corresponding category of data value.
 * @property {string} color - Color corresponding to data point.
 * @property {ISelectionId} selectionId - Id assigned to data point for cross filtering
 *                                         and visual interaction.
 */
interface BarChartDataPoint {
    value: number;
    category: string;
    color: string;
    selectionId: ISelectionId;
}

```

## Assign selection IDs to each data point

Since you iterate through the data points in `visualTransform`, it's also the ideal place to create selection IDs. The host variable is an `IVisualHost`, which contains services that the visual may use, such as color and selection builder.

Use the `createSelectionIdBuilder` factory method on `IVisualHost` to create a new selection ID. Create a new selection builder for each data point.

Since you're making selections based only on the category, you only need to define selections `withCategory`.

```

for (let i = 0, len = Math.max(category.values.length, dataValue.values.length); i < len; i++) {
    barChartDataPoints.push({
        category: category.values[i],
        value: dataValue.values[i],
        color: colorPalette.getColor(category.values[i]).value,
        selectionId: host.createSelectionIdBuilder()
            .withCategory(category, i)
            .createSelectionId()
    });
}

```

For more information, see [Create an instance of the selection builder](#).

## Interact with data points

You can interact with each bar of the bar chart once a selection ID is assigned to the data point. The bar chart listens to `click` events.

Use the `selectionManager` factory method on `IVisualHost` to create a selection manager for cross filtering and clearing selections.

```

let selectionManager = this.selectionManager;

//This must be an anonymous function instead of a lambda because
//d3 uses 'this' as the reference to the element that was clicked.
bars.on('click', function(d) {
    selectionManager.select(d.selectionId).then((ids: ISelectionId[]) => {
        bars.attr({
            'fill-opacity': ids.length > 0 ? BarChart.Config.transparentOpacity : BarChart.Config.solidOpacity
        });

        d3.select(this).attr({
            'fill-opacity': BarChart.Config.solidOpacity
        });
    });

    (<Event>d3.event).stopPropagation();
});

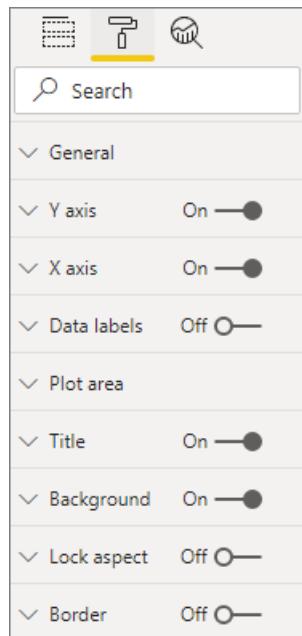
```

For more information, see [How to use SelectionManager](#).

## Static objects

You can add objects to the **Property** pane to further customize the visual. These customizations can be user interface changes, or changes related to the data that was queried. The sample uses static objects to render the X-axis for the bar chart.

You can toggle objects on or off in the **Property** pane.



### Define objects in capabilities

Define an `objects` property inside your `capabilities.json` file for objects to display in the **Property** pane.

- `enableAxis` is the internal name that the `dataView` references.
- `displayName` is the name shown on the **Property** pane.
- `bool` is a primitive value that is typically used with static objects, such as text boxes or switches.
- `show` is a special property on `properties` that enables the `show` switch on the object. Since `show` is a switch, it is typed as a `bool`.



```

"objects": {
    "enableAxis": {
        "displayName": "Enable Axis",
        "properties": {
            "show": {
                "displayName": "Enable Axis",
                "type": { "bool": true }
            }
        }
    }
}

```

For more information, see [Objects](#).

## Define property settings

The following sections describe the basic principles of defining property settings. You can also use the utility classes defined in the `powerbi-visuals-utils-dataviewutils` package for defining property settings. For more information, see the documentation and samples for the [DataViewObjectsParser](#) class.

Although optional, it's best to localize most settings onto a single object for easy reference.

```

/**
 * Interface for BarCharts viewmodel.
 *
 * @interface
 * @property {BarChartDataPoint[]} dataPoints - Set of data points the visual will render.
 * @property {number} dataMax - Maximum data value in the set of data points.
 * @property {BarChartSettings} settings - Object property settings
 */
interface BarChartViewModel {
    dataPoints: BarChartDataPoint[];
    dataMax: number;
    settings: BarChartSettings;
};

/**
 * Interface for BarChart settings.
 *
 * @interface
 * @property "show" enableAxis - Object property that allows axis to be enabled.
 */
interface BarChartSettings {
    enableAxis: {
        show: boolean;
    };
}

```

## Define and use ObjectEnumerationUtility

Object property values are available as metadata on the `dataView`, but there's no service to help retrieve these properties. `ObjectEnumerationUtility` is a set of static functions you can use to retrieve object values from the `dataView`, and for other visual projects. The `ObjectEnumerationUtility` is optional, but is great for iterating through the `dataView` to retrieve object properties.

```

/**
 * Gets property value for a particular object.
 *
 * @function
 * @param {DataViewObjects} objects - Map of defined objects.
 * @param {string} objectName      - Name of desired object.
 * @param {string} propertyName    - Name of desired property.
 * @param {T} defaultValue        - Default value of desired property.
 */
export function getValue<T>(objects: DataViewObjects, objectName: string, propertyName: string, defaultValue: T): T {
    if(objects) {
        let object = objects[objectName];
        if(object) {
            let property: T = object[propertyName];
            if(property !== undefined) {
                return property;
            }
        }
    }
    return defaultValue;
}

```

See [objectEnumerationUtility.ts](#) for source code.

### Retrieve property values from dataView

The `visualTransform` is the ideal place to manipulate the visual's view model. To continue this pattern, retrieve the object properties from the `dataView`.

Define the default state of the property, and use `getValue` to retrieve the property from the `dataView`.

```

let defaultSettings: BarChartSettings = {
    enableAxis: {
        show: false,
    }
};

let barChartSettings: BarChartSettings = {
    enableAxis: {
        show: getValue<boolean>(objects, 'enableAxis', 'show', defaultSettings.enableAxis.show),
    }
}

```

### Populate Property pane with enumerateObjectInstances

The `enumerateObjectInstances` optional method on `IVisual` enumerates through all objects and places them within the **Property** pane. Each object is called with `enumerateObjectInstances`. The object's name is available on `EnumerateVisualObjectInstancesOptions`.

For each object, define the property with its current state.

```

/**
 * Enumerates through the objects defined in the capabilities and adds the properties to the format pane
 *
 * @function
 * @param {EnumerateVisualObjectInstancesOptions} options - Map of defined objects
 */
public enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions):
VisualObjectInstanceEnumeration {
    let objectName = options.objectName;
    let objectEnumeration: VisualObjectInstance[] = [];

    switch(objectName) {
        case 'enableAxis':
            objectEnumeration.push({
                objectName: objectName,
                properties: {
                    show: this.barChartSettings.enableAxis.show,
                },
                selector: null
            });
    };

    return objectEnumeration;
}

```

## Control property update logic

Once an object is added to the **Property** pane, each toggle triggers an update. Add specific object logic in `if` blocks:

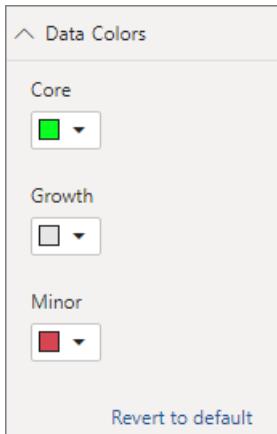
```

if(settings.enableAxis.show) {
    let margins = BarChart.Config.margins;
    height -= margins.bottom;
}

```

## Databound objects

Databound objects are similar to static objects, but typically deal with data selection. For example, you can change the color associated with the data point.



### Define object in capabilities

Similar to static objects, define another object in the `capabilities.json`.

- `colorSelector` is the internal name that the `dataView` references.
- `displayName` is the name shown on the **Property** pane.
- `fill` is a structural object value not associated with a primitive type.

```

"colorSelector": {
    "displayName": "Data Colors",
    "properties": {
        "fill": {
            "displayName": "Color",
            "type": {
                "fill": {
                    "solid": {
                        "color": true
                    }
                }
            }
        }
    }
}

```

For more information, see [Objects](#).

## Use ObjectEnumerationUtility

As with static objects, you need to retrieve object details from the `dataView`. However, instead of the object values being within metadata, the object values are associated with each category.

```

/**
 * Gets property value for a particular object in a category.
 *
 * @function
 * @param {DataViewCategoryColumn} category - List of category objects.
 * @param {number} index                  - Index of category object.
 * @param {string} objectName           - Name of desired object.
 * @param {string} propertyName         - Name of desired property.
 * @param {T} defaultValue             - Default value of desired property.
 */
export function getCategoricalObjectValue<T>(category: DataViewCategoryColumn, index: number, objectName: string, propertyName: string, defaultValue: T): T {
    let categoryObjects = category.objects;

    if(categoryObjects) {
        let categoryObject: DataViewObject = categoryObjects[index];
        if(categoryObject) {
            let object = categoryObject[objectName];
            if(object) {
                let property: T = object[propertyName];
                if(property !== undefined) {
                    return property;
                }
            }
        }
    }
    return defaultValue;
}

```

See [ObjectEnumerationUtility.ts](#) for source code.

## Define default color and retrieve categorical object from dataView

Each color is now associated with each category inside `dataView`. You can set each data point to its corresponding color.

```
for (let i = 0, len = Math.max(category.values.length, dataValue.values.length); i < len; i++) {
    let defaultColor: Fill = {
        solid: {
            color: colorPalette.getColor(category.values[i]).value
        }
    }

    barChartDataPoints.push({
        category: category.values[i],
        value: dataValue.values[i],
        color: getCategoricalObjectValue<Fill>(category, i, 'colorSelector', 'fill', defaultColor).solid.color,
        selectionId: host.createSelectionIdBuilder()
            .withCategory(category, i)
            .createSelectionId()
    });
}
```

### Populate Property pane with enumerateObjectInstances

Use `enumerateObjectInstances` to populate the **Property** pane with objects.

For this instance, add a color picker to render each category on the **Property** pane. To do this, add an additional case to the `switch` statement for `colorSelector`, and iterate through each data point with the associated color.

Selection is required to associate the color with the data point.

```

/**
 * Enumerates through the objects defined in the capabilities and adds the properties to the format pane
 *
 * @function
 * @param {EnumerateVisualObjectInstancesOptions} options - Map of defined objects
 */
public enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions):
VisualObjectInstanceEnumeration {
    let objectName = options.objectName;
    let objectEnumeration: VisualObjectInstance[] = [];

    switch(objectName) {
        case 'enableAxis':
            objectEnumeration.push({
                objectName: objectName,
                properties: {
                    show: this.barChartSettings.enableAxis.show,
                },
                selector: null
            });
            break;
        case 'colorSelector':
            for(let barDataPoint of this.barDataPoints) {
                objectEnumeration.push({
                    objectName: objectName,
                    displayName: barDataPoint.category,
                    properties: {
                        fill: {
                            solid: {
                                color: barDataPoint.color
                            }
                        },
                        selector: barDataPoint.selectionId.getSelector()
                    });
                }
            }
            break;
    };

    return objectEnumeration;
}

```

After providing a selector for each property, you get the following `dataView` object array:

The screenshot shows a portion of the `BarChart.prototype.update` function. A tooltip is displayed over the `dataViews` array, showing its structure:

```

Object
  ▼ dataViews: Array(1)
    ▼ 0: r
      ▼ categorical: r
        ▼ categories: r
          ▼ 0: r
            ▼ objects: Array(21)
              ▼ 0: Object
                ▼ colorSelector: Object
                  ▼ fill: Object
                    ▶ solid: Object
                    ▶ __proto__: Object
                    ▶ __proto__: Object
                    ▶ __proto__: Object
                ▶ 1: Object

```

Each item in the array `dataViews[0].categorical.categories[0].objects` corresponds to the concrete category of the

dataset.

The function `getCategoricalObjectValue` just provides a convenient way of accessing properties by their category index. You must provide an `objectName` and `propertyName` that match the object and property in `capabilities.json`.

## Other features

You can add a slider control or tooltips to the bar chart. For the code to add, see the commits at [Add a property pane slider to control opacity](#) and [Add support for tooltips](#). For more information about tooltips, see [Toolips in Power BI visuals](#).

## Packaging

Before you can load your visual into [Power BI Desktop](#) or share it with the community in the [Power BI Visual Gallery](#), you must package it. Navigate to the root folder of your visual project, which contains the file `pbviz.json`, and use the following command to generate a `pbviz` file:

```
pbviz package
```

This command creates a `pbviz` file in the `dist`/directory of your visual project, and overwrites any `pbviz` file from previous package operations.

## Next steps

You can add the following abilities to your visual:

- [Add a context menu to a visual](#)
- [Landing page](#)
- [Launch URL](#)
- [Locale support](#)

# Tutorial: Adding formatting options to a Power BI visual

3/13/2020 • 5 minutes to read • [Edit Online](#)

In this tutorial, we go through how to add common properties to the visual.

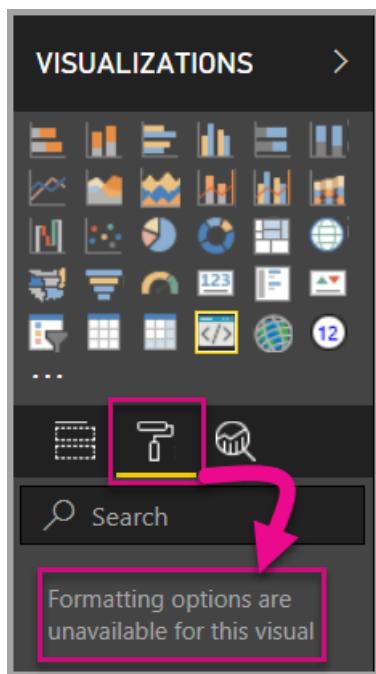
In this tutorial, you learn how to:

- Add visual properties.
- Package the visual.
- Import the custom visual to a Power BI Desktop report.

## Adding formatting options

1. In Power BI, select the Format page.

You should see a message that reads - *Formatting options are unavailable for this visual.*



2. In Visual Studio Code, open the `capabilities.json` file.
3. Before the `dataViewMappings` array, add `objects` (after line 8).

```
"objects": {},
```

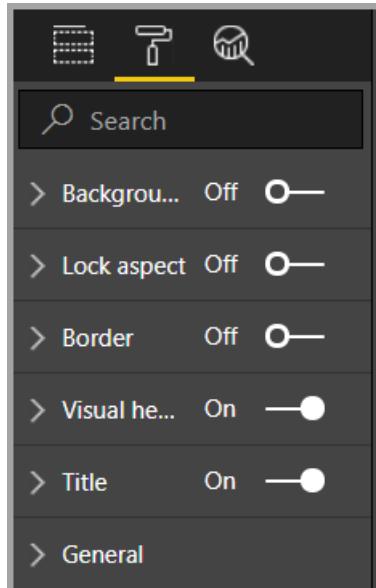
```
"dataRoles": [
    {
        "displayName": "Measure",
        "name": "measure",
        "kind": "Measure"
    }
],
"objects": {},
"dataViewMappings": []
{
    "conditions": [

```

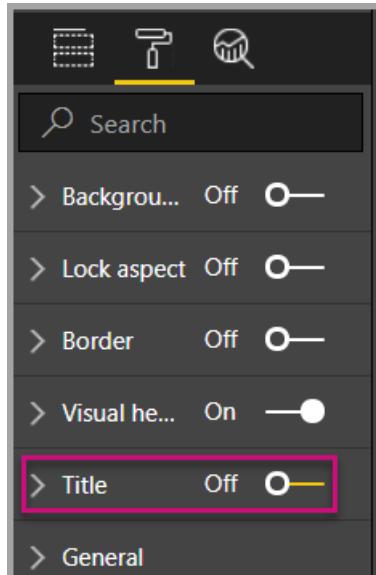
4. Save the **capabilities.json** file.
5. In Power BI, review the formatting options again.

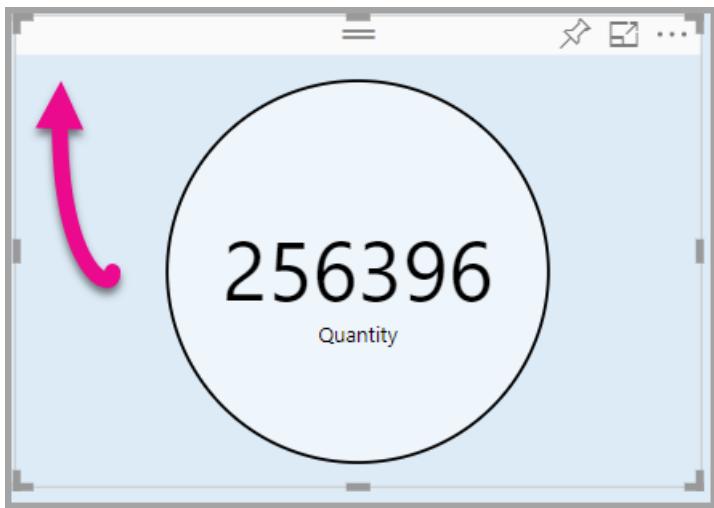
**NOTE**

If you do not see the formatting options change then select **Reload Custom Visual**.



6. Set the **Title** option to *Off*. Notice that the visual no longer displays the measure name at the top-left corner.





## Adding custom formatting options

You can add custom properties to enable configuring the color of the circle, and also the border width.

1. In PowerShell, stop the custom visual.
2. In Visual Studio Code, in the **capabilities.json** file, insert the following JSON fragment into the object labeled **objects**.

```
{
  "circle": {
    "displayName": "Circle",
    "properties": {
      "circleColor": {
        "displayName": "Color",
        "description": "The fill color of the circle.",
        "type": {
          "fill": {
            "solid": {
              "color": true
            }
          }
        }
      },
      "circleThickness": {
        "displayName": "Thickness",
        "description": "The circle thickness.",
        "type": {
          "numeric": true
        }
      }
    }
  }
}
```

The JSON fragment describes a group named **circle**, which consists of two options named **circleColor** and **circleThickness**.

```

"objects": [
    "circle": {
        "displayName": "Circle",
        "properties": {
            "circleColor": {
                "displayName": "Color",
                "description": "The fill color of the circle.",
                "type": {
                    "fill": {
                        "solid": {
                            "color": true
                        }
                    }
                }
            },
            "circleThickness": {
                "displayName": "Thickness",
                "description": "The circle thickness.",
                "type": {
                    "numeric": true
                }
            }
        }
    }
},
"dataViewMappings": [

```

3. Save the **capabilities.json** file.
4. In the **Explorer pane**, from inside the **src** folder, and then select **settings.ts**. *This file represents the settings for the starter visual.*
5. In the **settings.ts** file, replace the two classes with the following code.

```

export class CircleSettings {
    public circleColor: string = "white";
    public circleThickness: number = 2;
}
export class VisualSettings extends DataViewObjectsParser {
    public circle: CircleSettings = new CircleSettings();
}

```

```

"use strict";
import { DataViewObjectsParser } from "powerbi-visuals-utils-dataviewutils"

export class CircleSettings {
    public circleColor: string = "white";
    public circleThickness: number = 2;
}
export class VisualSettings extends DataViewObjectsParser.DataViewObjectsParser {
    public circle: CircleSettings = new CircleSettings();
}

```

This module defines the two classes. The **CircleSettings** class defines two properties with names that match the objects defined in the **capabilities.json** file (**circleColor** and **circleThickness**) and also sets default values. The **VisualSettings** class inherits the **DataViewObjectParser** class and adds a property named **circle**, which matches the object defined in the **capabilities.json** file, and returns an instance of **CircleSettings**.

6. Save the **settings.ts** file.
7. Open the **visual.ts** file.

8. In the **visual.ts** file,

```
import VisualSettings, VisualObjectInstanceEnumeration and EnumerateVisualObjectInstancesOptions :
```

```
import { VisualSettings } from "./settings";
import VisualObjectInstanceEnumeration = powerbi.VisualObjectInstanceEnumeration;
import EnumerateVisualObjectInstancesOptions = powerbi.EnumerateVisualObjectInstancesOptions;
```

and in the **Visual** class add the following property:

```
private visualSettings: VisualSettings;
```

This property stores a reference to the **VisualSettings** object, describing the visual settings.

```
export class Visual implements IVisual {
    private host: IVisualHost;
    private svg: Selection<SVGElement>;
    private container: Selection<SVGElement>;
    private circle: Selection<SVGElement>;
    private textValue: Selection<SVGElement>;
    private textLabel: Selection<SVGElement>;
    private visualSettings: VisualSettings;
```

9. In the **Visual** class, add the following method before the **update** method. This method is used to populate the formatting options.

```
public enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions): VisualObjectInstanceEnumeration {
    const settings: VisualSettings = this.visualSettings || <VisualSettings>VisualSettings.getDefault();
    return VisualSettings.enumerateObjectInstances(settings, options);
}
```

This method is used to populate the formatting options.

```
this.textValue = this.container.append("text")
    .classed("textValue", true);
this.textLabel = this.container.append("text")
    .classed("textLabel", true);
}

public enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions): VisualObjectInstanceEnumeration {
    const settings: VisualSettings = this.visualSettings || <VisualSettings>VisualSettings.getDefault();
    return VisualSettings.enumerateObjectInstances(settings, options);
}

public update(options: VisualUpdateOptions) {
    let dataView: DataView = options.dataViews[0];
    this.visualSettings = VisualSettings.parse<VisualSettings>(dataView);
```

10. In the **update** method, after the declaration of the **radius** variable, add the following code.

```
this.visualSettings = VisualSettings.parse<VisualSettings>(dataView);

this.visualSettings.circle.circleThickness = Math.max(0, this.visualSettings.circle.circleThickness);
this.visualSettings.circle.circleThickness = Math.min(10, this.visualSettings.circle.circleThickness);
```

This code retrieves the format options. It adjusts any value passed into the **circleThickness** property, converting it to 0 if negative, or 10 if it's a value greater than 10.

```
    let radius: number = Math.min(width, height) / 2.2;

    this.visualSettings = VisualSettings.parse<VisualSettings>(dataView);

    this.visualSettings.circle.circleThickness = Math.max(0, this.visualSettings.circle.circleThickness);
    this.visualSettings.circle.circleThickness = Math.min(10, this.visualSettings.circle.circleThickness);

    this.circle
        .style("fill", "white")
        .style("fill-opacity", 0.5)
        .style("stroke", "black")
        .style("stroke-width", 2)
        .attr({
```

11. For the **circle** element, modify the value passed to the **fill** style to the following expression.

```
this.visualSettings.circle.circleColor
```

```
this.circle
    .style("fill", this.visualSettings.circle.circleColor)
    .style("fill-opacity", 0.5)
    .style("stroke", "black")
    .style(["stroke-width", this.visualSettings.circle.circleThickness])
    .attr({
        r: radius,
        cx: width / 2,
        cy: height / 2
    })
);
```

12. For the **circle** element, modify the value passed to the **stroke-width** style to the following expression.

```
this.visualSettings.circle.circleThickness
```

```
this.circle
    .style("fill", this.visualSettings.circle.circleColor)
    .style("fill-opacity", 0.5)
    .style("stroke", "black")
    .style("stroke-width", this.visualSettings.circle.circleThickness)
    .attr({
        r: radius,
        cx: width / 2,
        cy: height / 2
    })
);
```

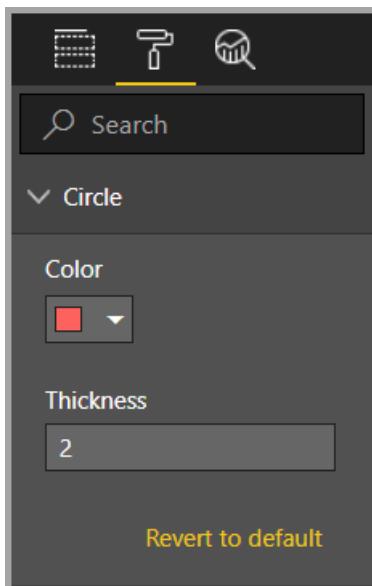
13. Save the visual.ts file.

14. In PowerShell, start the visual.

```
pbviz start
```

15. In Power BI, in the toolbar floating above the visual, select **Toggle Auto Reload**.

16. In the **visual format** options, expand **Circle**.



Modify the **color** and **thickness** option.

Modify the **thickness** option to a value less than zero, and a value higher than 10. Then notice the visual updates the value to a tolerable minimum or maximum.

## Packaging the custom visual

Enter property values for the custom visual project, update the icon file, and then package the custom visual.

1. In **PowerShell**, stop the custom visual.
2. Open the **pbiviz.json** file in **Visual Studio Code**.
3. In the **visual** object, modify the **displayName** property to *Circle Card*.

In the **Visualizations** pane, hovering over the icon reveals the display name.

```
1  {
2    "visual": {
3      "name": "circleCard",
4      "displayName": "Circle Card",
5      "guid": "Visual",
6      "visualClassName": "Visual",
```

4. For the **description** property, enter the following text.

*Displays a formatted measure value inside a circle*

5. Fill **supportUrl** and **gitHubUrl** for the visual.

Example:

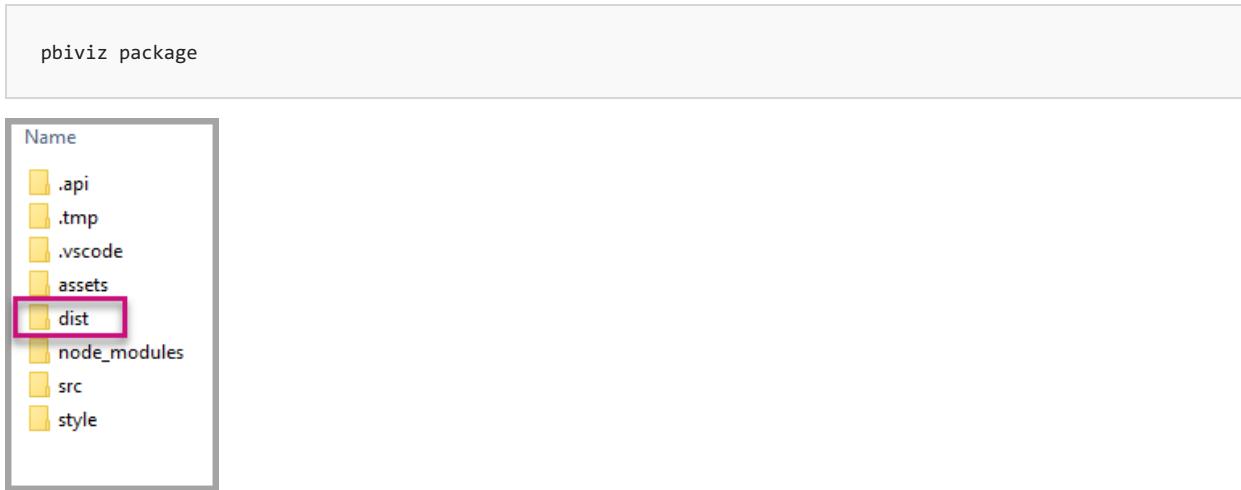
```
{
  "supportUrl": "https://community.powerbi.com",
  "gitHubUrl": "https://github.com/microsoft/PowerBI-visuals-circlecard"
}
```

6. Enter your details in the **author** object.
7. Save the **pbiviz.json** file.
8. In the **assets** object, notice that the document defines a path to an icon. The icon is the image that appears in the **Visualizations** pane. It must be a **PNG** file, *20 pixels by 20 pixels*.

- In Windows Explorer, copy the icon.png file, and then paste it to replace the default file located at assets folder.
- In Visual Studio Code, in the Explorer pane, expand the assets folder, and then select the icon.png file.
- Review the icon.



- In Visual Studio Code, ensure that all files are saved.
- To package the custom visual, in PowerShell, enter the following command.

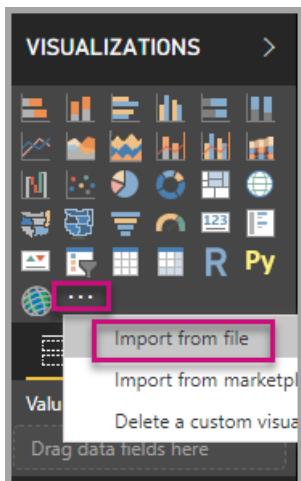


Now the package is output to the **dist** folder of the project. The package contains everything required to import the custom visual into either the Power BI service or a Power BI Desktop report. You have now packaged the custom visual, and it is now ready for use.

## Importing the custom visual

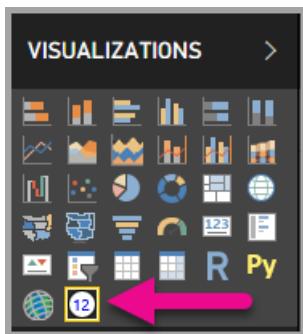
Now you can open the Power BI Desktop report, and import the Circle Card custom visual.

- Open Power BI Desktop, create a new report with any *sample dataset*
- In the **Visualizations** pane, select the **ellipsis**, and then select **Import** from File.



- In the **import window**, select **Import**.
- In the Open window, navigate to the **dist** folder in your project directory.
- Select the **circleCard.pbviz** file, and then select **Open**.

- When the visual has successfully imported, select **OK**.
- Verify that the visual has been added to the **Visualizations** pane.



- Hover over the **Circle Card** icon, and notice the tooltip that appears.

## Debugging

For tips about debugging your custom visual, see the [debugging guide](#).

## Next steps

You can list your newly developed visual for others to use by submitting it to the **AppSource**. For more information on this process reference [Publish Power BI visuals to AppSource](#).

# Tutorial: Add unit tests for Power BI visual projects

3/13/2020 • 9 minutes to read • [Edit Online](#)

This article describes the basics of writing unit tests for your Power BI visuals, including how to:

- Set up the Karma JavaScript test runner testing framework, Jasmine.
- Use the powerbi-visuals-utils-testutils package.
- Use mocks and fakes to help simplify unit testing of Power BI visuals.

## Prerequisites

- An installed Power BI visuals project
- A configured Nodejs environment

## Install and configure the Karma JavaScript test runner and Jasmine

Add the required libraries to the `package.json` file in the `devDependencies` section:

```
"@babel/polyfill": "^7.2.5",
"@types/d3": "5.5.0",
"@types/jasmine": "2.5.37",
"@types/jasmine-jquery": "1.5.28",
"@types/jquery": "2.0.41",
"@types/karma": "3.0.0",
"@types/lodash-es": "4.17.1",
"coveralls": "3.0.2",
"istanbul-instrumenter-loader": "^3.0.1",
"jasmine": "2.5.2",
"jasmine-core": "2.5.2",
"jasmine-jquery": "2.1.1",
"jquery": "3.1.1",
"karma": "3.1.1",
"karma-chrome-launcher": "2.2.0",
"karma-coverage": "1.1.2",
"karma-coverage-istanbul-reporter": "^2.0.4",
"karma-jasmine": "2.0.1",
"karma-junit-reporter": "^1.2.0",
"karma-sourcemap-loader": "^0.3.7",
"karma-typescript": "^3.0.13",
"karma-typescript-preprocessor": "0.4.0",
"karma-webpack": "3.0.5",
"puppeteer": "1.17.0",
"style-loader": "0.23.1",
"ts-loader": "5.3.0",
"ts-node": "7.0.1",
"tslint": "^5.12.0",
"webpack": "4.26.0"
```

To learn more about `package.json`, see the description at [npm-package.json](#).

Save the `package.json` file and, at the `package.json` location, run the following command:

```
npm install
```

The package manager installs all new packages that are added to `package.json`.

To run unit tests, configure the test runner and `webpack` config.

The following code is a sample of the `test.webpack.config.js` file:

```

const path = require('path');
const webpack = require("webpack");

module.exports = {
  devtool: 'source-map',
  mode: 'development',
  optimization : {
    concatenateModules: false,
    minimize: false
  },
  module: [
    rules: [
      {
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/
      },
      {
        test: /\.json$/,
        loader: 'json-loader'
      },
      {
        test: /\.tsx?$/i,
        enforce: 'post',
        include: /(src)/,
        exclude: /(node_modules|resources\js\vendor)/,
        loader: 'istanbul-instrumenter-loader',
        options: { esModules: true }
      },
      {
        test: /\.less$/,
        use: [
          {
            loader: 'style-loader'
          },
          {
            loader: 'css-loader'
          },
          {
            loader: 'less-loader',
            options: {
              paths: [path.resolve(__dirname, 'node_modules')]
            }
          }
        ]
      }
    ],
    externals: {
      "powerbi-visuals-api": '{}'
    },
    resolve: {
      extensions: ['.tsx', '.ts', '.js', '.css']
    },
    output: {
      path: path.resolve(__dirname, ".tmp/test")
    },
    plugins: [
      new webpack.ProvidePlugin({
        'powerbi-visuals-api': null
      })
    ]
  ];
};

```

The following code is a sample of the *karma.conf.ts* file:

```
"use strict";

const webpackConfig = require("./test.webpack.config.js");
const tsconfig = require("./test.tsconfig.json");
const path = require("path");

const testRecursivePath = "test/visualTest.ts";
const srcOriginalRecursivePath = "src/**/*.ts";
const coverageFolder = "coverage";

process.env.CHROME_BIN = require("puppeteer").executablePath();

import { Config, ConfigOptions } from "karma";

module.exports = (config: Config) => {
  config.set(<ConfigOptions>{
    mode: "development",
    browserNoActivityTimeout: 100000,
    browsers: ["ChromeHeadless"], // or Chrome to use locally installed Chrome browser
    colors: true,
    frameworks: ["jasmine"],
    reporters: [
      "progress",
      "junit",
      "coverage-istanbul"
    ],
    junitReporter: {
      outputDir: path.join(__dirname, coverageFolder),
      outputFile: "TESTS-report.xml",
      useBrowserName: false
    },
    singleRun: true,
    plugins: [
      "karma-coverage",
      "karma-typescript",
      "karma-webpack",
      "karma-jasmine",
      "karma-sourcemap-loader",
      "karma-chrome-launcher",
      "karma-junit-reporter",
      "karma-coverage-istanbul-reporter"
    ],
    files: [
      "node_modules/jquery/dist/jquery.min.js",
      "node_modules/jasmine-jquery/lib/jasmine-jquery.js",
      {
        pattern: './capabilities.json',
        watched: false,
        served: true,
        included: false
      },
      testRecursivePath,
      {
        pattern: srcOriginalRecursivePath,
        included: false,
        served: true
      }
    ],
    preprocessors: {
      [testRecursivePath]: ["webpack", "coverage"]
    },
    typescriptPreprocessor: {
      options: tsconfig.compilerOptions
    },
    coverageIstanbulReporter: {
      reports: ["html", "lcovonly", "text-summary", "cobertura"],
      dir: path.join(__dirname, coverageFolder),
      'report-config': {
        html: {

```

```

        subdir: 'html-report'
    }
},
combineBrowserReports: true,
fixWebpackSourcePaths: true,
verbose: false
},
coverageReporter: {
    dir: path.join(__dirname, coverageFolder),
    reporters: [
        // reporters not supporting the `file` property
        { type: 'html', subdir: 'html-report' },
        { type: 'lcov', subdir: 'lcov' },
        // reporters supporting the `file` property, use `subdir` to directly
        // output them in the `dir` directory
        { type: 'cobertura', subdir: '.', file: 'cobertura-coverage.xml' },
        { type: 'lcovonly', subdir: '.', file: 'report-lcovonly.txt' },
        { type: 'text-summary', subdir: '.', file: 'text-summary.txt' },
    ]
},
mime: {
    "text/x-typescript": ["ts", "tsx"]
},
webpack: webpackConfig,
webpackMiddleware: {
    stats: "errors-only"
}
});
};


```

If necessary, you can modify this configuration.

The code in `karma.conf.js` contains the following variable:

- `recursivePathToTests` : Locates the test code
- `srcRecursivePath` : Locates the output JavaScript code after compiling
- `srcCssRecursivePath` : Locates the output CSS after compiling less file with styles
- `srcOriginalRecursivePath` : Locates the source code of your visual
- `coverageFolder` : Determines where the coverage report is to be created

The configuration file includes the following properties:

- `singleRun: true` : Tests are run on a continuous integration (CI) system, or they can be run one time. You can change the setting to `false` for debugging your tests. Karma keeps the browser running so that you can use the console for debugging.
- `files: [...]` : In this array, you can specify the files to load to the browser. Usually, there are source files, test cases, libraries (Jasmine, test utilities). You can add additional files to the list, as necessary.
- `preprocessors` : In this section, you configure actions that run before the unit tests run. They precompile the typescript to JavaScript, prepare source map files, and generate code coverage report. You can disable `coverage` when you debug your tests. Coverage generates additional code for check code for the test coverage, which complicates debugging tests.

For descriptions of all Karma configurations, go to the [Karma Configuration File](#) page.

For your convenience, you can add a test command into `scripts` :

```
{
  "scripts": {
    "pbiviz": "pbiviz",
    "start": "pbiviz start",
    "typings": "node node_modules/typings/dist/bin.js i",
    "lint": "tslint -r \"node_modules/tslint-microsoft-contrib\" \\"+(src|test)/**/*.ts\"",
    "pretest": "pbiviz package --resources --no-minify --no-pbiviz --no-plugin",
    "test": "karma start"
  }
  ...
}
```

You're now ready to begin writing your unit tests.

## Check the DOM element of the visual

To test the visual, first create an instance of visual.

### Create a visual instance builder

Add a *visualBuilder.ts* file to the *test* folder by using the following code:

```
import {
  VisualBuilderBase
} from "powerbi-visuals-utils-testutils";

import {
  BarChart as VisualClass
} from "../src/visual";

import powerbi from "powerbi-visuals-api";
import VisualConstructorOptions = powerbi.extensibility.visual.VisualConstructorOptions;

export class BarChartBuilder extends VisualBuilderBase<VisualClass> {
  constructor(width: number, height: number) {
    super(width, height);
  }

  protected build(options: VisualConstructorOptions) {
    return new VisualClass(options);
  }

  public get mainElement() {
    return this.element.children("svg.barChart");
  }
}
```

There's `build` method for creating an instance of your visual. `mainElement` is a get method, which returns an instance of "root" document object model (DOM) element in your visual. The getter is optional, but it makes writing the unit test easier.

You now have a build of an instance of your visual. Let's write the test case. The test case checks the SVG elements that are created when your visual is displayed.

### Create a typescript file to write test cases

Add a *visualTest.ts* file for the test cases by using the following code:

```

import powerbi from "powerbi-visuals-api";

import { BarChartBuilder } from "./VisualBuilder";

import {
    BarChart as VisualClass
} from "../src/visual";

import VisualBuilder = powerbi.extensibility.visual.test.BarChartBuilder;

describe("BarChart", () => {
    let visualBuilder: VisualBuilder;
    let dataView: DataView;

    beforeEach(() => {
        visualBuilder = new VisualBuilder(500, 500);
    });

    it("root DOM element is created", () => {
        expect(visualBuilder.mainElement).toBeInDOM();
    });
});

```

Several methods are called:

- `describe` : Describes a test case. In the context of the Jasmine framework, it often describes a suite or group of specs.
- `beforeEach` : Is called before each call of the `it` method, which is defined in the `describe` method.
- `it` : Defines a single spec. The `it` method should contain one or more `expectations`.
- `expect` : Creates an expectation for a spec. A spec succeeds if all expectations pass without any failures.
- `toBeInDOM` : One of the *matchers* methods. For more information about matchers, see [Jasmine Namespace: matchers](#).

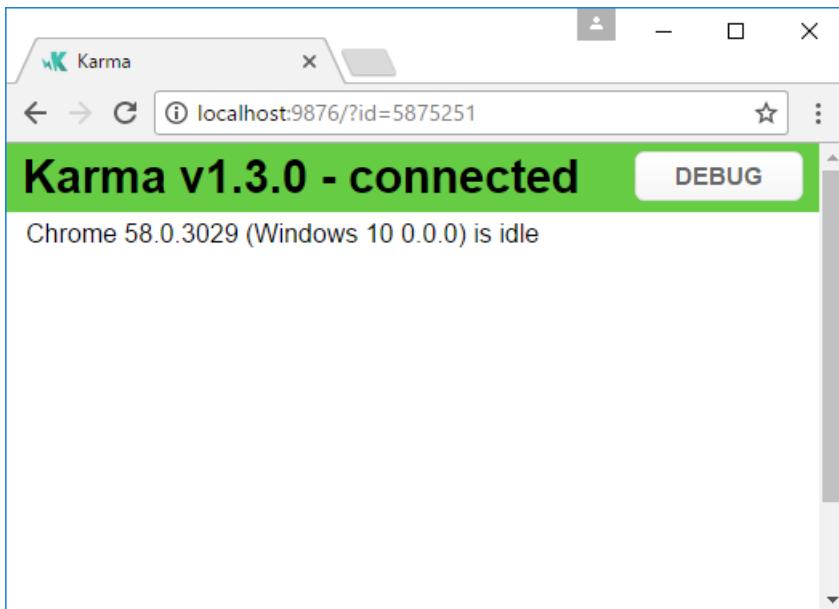
For more information about Jasmine, see the [Jasmine framework documentation](#) page.

## Launch unit tests

This test checks that root SVG element of the visuals is created. To run the unit test, enter the following command in the command-line tool:

```
npm run test
```

`karma.js` runs the test case in the Chrome browser.



#### NOTE

You must install Google Chrome locally.

In the command-line window, you'll get following output:

```
> karma start

23 05 2017 12:24:26.842:WARN [watcher]: Pattern "E:/WORKSPACE/PowerBI/PowerBI-visuals-sampleBarChart/data/*.csv" does not match any file.
23 05 2017 12:24:30.836:WARN [karma]: No captured browser, open https://localhost:9876/
23 05 2017 12:24:30.849:INFO [karma]: Karma v1.3.0 server started at https://localhost:9876/
23 05 2017 12:24:30.850:INFO [launcher]: Launching browser Chrome with unlimited concurrency
23 05 2017 12:24:31.059:INFO [launcher]: Starting browser Chrome
23 05 2017 12:24:33.160:INFO [Chrome 58.0.3029 (Windows 10 0.0.0)]: Connected on socket /#2meR6hjXFmsE_fjiAAAA with id 5875251
Chrome 58.0.3029 (Windows 10 0.0.0): Executed 1 of 1 SUCCESS (0.194 secs / 0.011 secs)

===== Coverage summary =====
Statements : 27.43% ( 65/237 )
Branches   : 19.84% ( 25/126 )
Functions   : 43.86% ( 25/57 )
Lines      : 20.85% ( 44/211 )
=====
```

#### How to add static data for unit tests

Create the *visualData.ts* file in the *test* folder by using the following code:

```

import powerbi from "powerbi-visuals-api";
import DataView = powerbi.DataView;

import {
    testDataViewBuilder,
    getRandomNumbers
} from "powerbi-visuals-utils-testutils";

export class SampleBarChartDataBuilder extends TestDataViewBuilder {
    public static CategoryColumn: string = "category";
    public static MeasureColumn: string = "measure";

    public constructor() {
        super();
        ...
    }

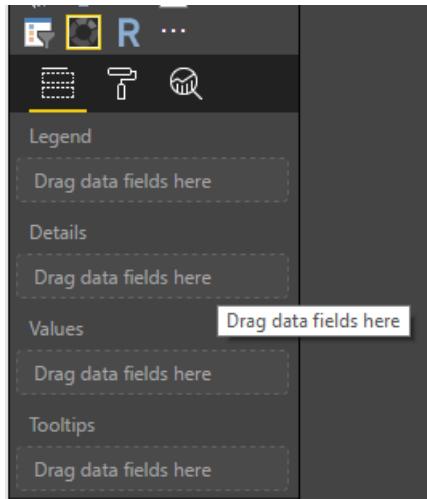
    public getDataView(columnNames?: string[]): DataView {
        let dateView: any = this.createCategoricalDataViewBuilder([
            ...
        ],
        [
            ...
        ],
        columnNames).build();

        // there's client side computed maxValue
        let maxLocal = 0;
        this.valuesMeasure.forEach((item) => {
            if (item > maxLocal) {
                maxLocal = item;
            }
        });
        (<any>dateView).categorical.values[0].maxLocal = maxLocal;
    }
}

```

The `SampleBarChartDataBuilder` class extends `TestDataViewBuilder` and implements the abstract method `getDataView`.

When you put data into data-field buckets, Power BI produces a categorical `dataview` object that's based on your data.



In unit tests, you don't have Power BI core functions to reproduce the data. But you need to map your static data to the categorical `dataview`. The `TestDataViewBuilder` class can help you map it.

For more information about Data View mapping, see [DataViewMappings](#).

In the `getDataView` method, you call the `createCategoricalDataViewBuilder` method with your data.

In `sampleBarChart` visual [capabilities.json](#) file, we have `dataRoles` and `dataViewMappings` objects:

```
"dataRoles": [
  {
    "displayName": "Category Data",
    "name": "category",
    "kind": "Grouping"
  },
  {
    "displayName": "Measure Data",
    "name": "measure",
    "kind": "Measure"
  }
],
"dataViewMappings": [
  {
    "conditions": [
      {
        "category": {
          "max": 1
        },
        "measure": {
          "max": 1
        }
      }
    ],
    "categorical": {
      "categories": {
        "for": {
          "in": "category"
        }
      },
      "values": {
        "select": [
          {
            "bind": {
              "to": "measure"
            }
          }
        ]
      }
    }
  }
],
```

To generate the same mapping, you must set the following params to `createCategoricalDataViewBuilder` method:

```

([
  {
    source: {
      displayName: "Category",
      queryName: SampleBarChartData.ColumnCategory,
      type: ValueType.fromDescriptor({ text: true }),
      roles: {
        Category: true
      },
    },
    values: this.valuesCategory
  }
],
[
  {
    source: {
      displayName: "Measure",
      isMeasure: true,
      queryName: SampleBarChartData.MeasureColumn,
      type: ValueType.fromDescriptor({ numeric: true }),
      roles: {
        Measure: true
      },
    },
    values: this.valuesMeasure
  },
], columnNames)

```

Where `this.valuesCategory` is an array of categories:

```
public valuesCategory: string[] = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"];
```

And `this.valuesMeasure` is an array of measures for each category:

```
public valuesMeasure: number[] = [742731.43, 162066.43, 283085.78, 300263.49, 376074.57, 814724.34, 570921.34];
```

Now, you can use the `SampleBarChartDataBuilder` class in your unit test.

The `ValueType` class is defined in the `powerbi-visuals-utils-testutils` package. And the `createCategoricalDataViewBuilder` method requires the `lodash` library.

Add these packages to the dependencies.

In `package.json` at `devDependencies` section

```
"lodash-es": "4.17.1",
"powerbi-visuals-utils-testutils": "2.2.0"
```

Call

```
npm install
```

to install `lodash-es` library.

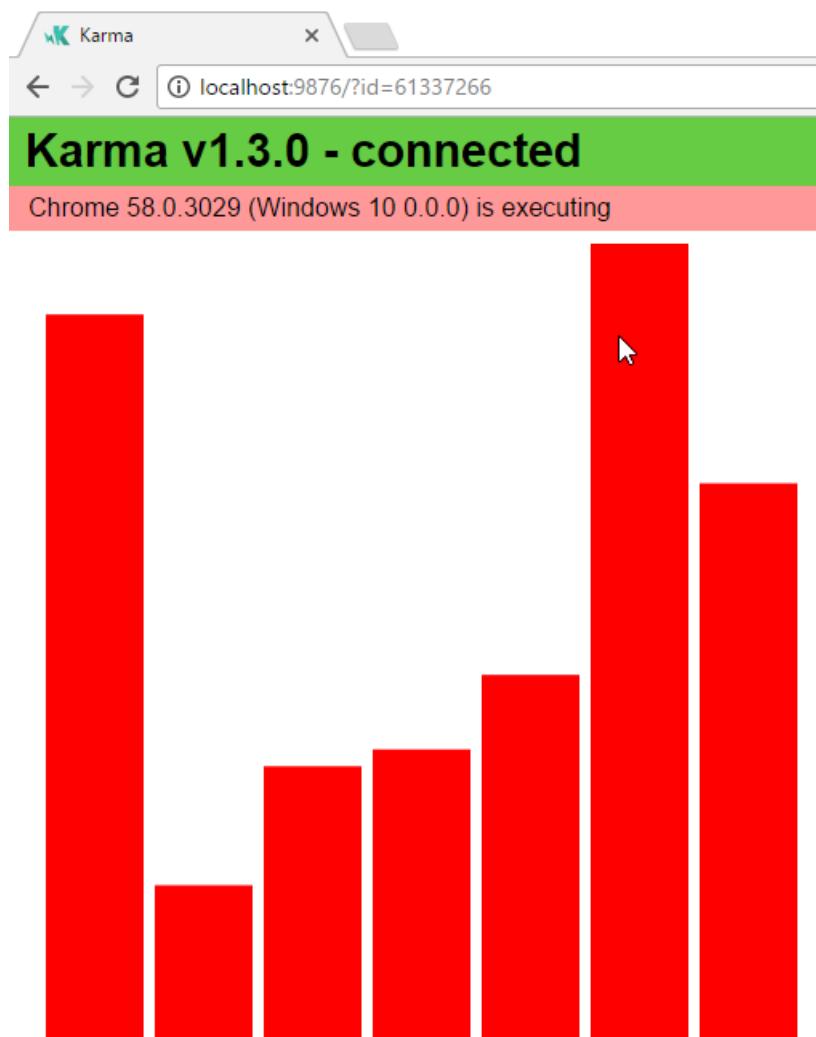
Now, you can run the unit test again. You must get the following output:

```
> karma start

23 05 2017 16:19:54.318:WARN [watcher]: Pattern "E:/WORKSPACE/PowerBI/PowerBI-visuals-sampleBarChart/data/*.csv" does not match any file.
23 05 2017 16:19:58.333:WARN [karma]: No captured browser, open https://localhost:9876/
23 05 2017 16:19:58.346:INFO [karma]: Karma v1.3.0 server started at https://localhost:9876/
23 05 2017 16:19:58.346:INFO [launcher]: Launching browser Chrome with unlimited concurrency
23 05 2017 16:19:58.394:INFO [launcher]: Starting browser Chrome
23 05 2017 16:19:59.873:INFO [Chrome 58.0.3029 (Windows 10 0.0.0)]: Connected on socket /#NcNTAGH9hWfGMCuEAAAA with id 3551106
Chrome 58.0.3029 (Windows 10 0.0.0): Executed 1 of 1 SUCCESS (1.266 secs / 1.052 secs)

=====
===== Coverage summary =====
Statements : 56.72% ( 135/238 )
Branches   : 32.54% ( 41/126 )
Functions   : 66.67% ( 38/57 )
Lines      : 52.83% ( 112/212 )
=====
```

Your visual opens in the Chrome browser, as shown:



The summary shows that coverage has increased. To learn more about current code coverage, open [coverage\index.html](#).

/

89.58% Statements 516/576 69.92% Branches 258/369 83.77% Functions 129/154 89.43% Lines 457/511

File ▲	Statements ▲	Branches ▲	Functions ▲	Lines ▲
src/	<div style="width: 100%;">██████████</div>	89.58%	516/576	69.92% 258/369 83.77% 129/154 89.43% 457/511

Or look at the scope of the `src` folder:

### all files src/

89.58% Statements 516/576 69.92% Branches 258/369 83.77% Functions 129/154 89.43% Lines 457/511

File ▲	Statements ▲	Branches ▲	Functions ▲	Lines ▲
behavior.ts	<div style="width: 95.24%;">███████████</div>	95.24%	20/21	50% 4/8 90% 9/10 93.75% 15/16
columns.ts	<div style="width: 65.31%;">██████████</div>	65.31%	32/49	48.61% 35/72 52% 13/25 65% 26/40
helpers.ts	<div style="width: 75%;">██████████</div>	75%	18/24	58.33% 7/12 75% 9/12 72.22% 13/18
settings.ts	<div style="width: 100%;">██████████</div>	100%	42/42	50% 4/8 100% 9/9 100% 33/33
tooltipBuilder.ts	<div style="width: 72.06%;">██████████</div>	72.06%	49/68	66.1% 39/59 100% 9/9 68.33% 41/60
utils.ts	<div style="width: 100%;">██████████</div>	100%	12/12	87.5% 14/16 100% 6/6 100% 8/8
visual.ts	<div style="width: 96.98%;">███████████</div>	96.98%	289/298	82.61% 133/161 94.34% 50/53 96.85% 277/286
visualLayout.ts	<div style="width: 87.1%;">██████████</div>	87.1%	54/62	66.67% 22/33 80% 24/30 88% 44/50

In the scope of file, you can view the source code. The `Coverage` utilities would highlight the row in red if certain code isn't executed during the unit tests.

```

609 }
610
611 14x     private drawLabels(data: ArcDescriptor<AsterDataPoint>[],
612   context: d3.Selection<AsterArcDescriptor>,
613   layout: ILabelLayout,
614   viewport: IViewport,
615   outlineArc: d3.svg.Arc<AsterArcDescriptor>,
616   labelArc: d3.svg.Arc<AsterArcDescriptor>): void {
617   // Hide and reposition labels that overlap
618   let dataLabelManager = new DataLabelManager();
619   let filteredData = dataLabelManager.hideCollidedLabels(viewport, data, true /* addTransform */);
620
621 14x     if (filteredData.length === 0) {
622       dataLabelUtils.cleanDataLabels(context, true);
623       return;
624     }
625
626   // Draw labels
627   14x     if (context.select(AsterPlot.labelGraphicsContextClass.selector).empty())
628     context.append("g").classed(AsterPlot.labelGraphicsContextClass.class, true);
629
630 14x     let labels = context
631       .select(AsterPlot.labelGraphicsContextClass.selector)
632       .selectAll(".data-labels").data<LabelEnabledDataPoint>(
633         filteredData,
634         (d: ArcDescriptor<AsterDataPoint>) => (d.data.identity as ISelectionId).getKey());
635
636 14x     labels.enter().append("text").classed("data-labels", true);
637
638 14x     if (!labels)
639       return;
640
641 14x     labels
642       .attr({ x: (d: LabelEnabledDataPoint) => d.labelX, y: (d: LabelEnabledDataPoint) => d.labelY, dy: ".35em" })

```

**IMPORTANT**

Code coverage doesn't mean that you have good functionality coverage of the visual. One simple unit test provides over 96 percent coverage in `src\visual.ts`.

## Next steps

When your visual is ready, you can submit it for publication. For more information, see [Publish Power BI visuals to AppSource](#).

# Power BI visual project structure

5/14/2020 • 3 minutes to read • [Edit Online](#)

The best way to start creating a new Power BI visual is to use the Power BI visuals [pbviz](#) tool.

To create a new visual, navigate to the directory you want the Power BI visual to reside in, and run the command:

```
pbviz new <visual project name>
```

Running this command creates a Power BI visual folder that contains the following files:

```
project
├── .vscode
│   ├── launch.json
│   └── settings.json
├── assets
│   └── icon.png
├── node_modules
└── src
    ├── settings.ts
    └── visual.ts
└── style
    └── visual.less
├── capabilities.json
├── package-lock.json
├── package.json
├── pbviz.json
├── tsconfig.json
└── tslint.json
```

## Folder and file description

This section provides information for each folder and file in the directory that the Power BI visuals [pbviz](#) tool creates.

### .vscode

This folder contains the VS code project settings.

To configure your workspace, edit the `.vscode/settings.json` file.

For more information, see [User and Workspace Settings](#)

### assets

This folder contains the `icon.png` file.

The Power BI visuals tool uses this file as the new Power BI visual icon in the Power BI visualization pane.

### src

This folder contains the visual's source code.

In this folder, the Power BI visuals tool creates the following files:

- `visual.ts` - The visual's main source code.
- `settings.ts` - The code of the visual's settings. The classes in the file provide an interface for defining your [visual's properties](#).

## style

This folder contains the `visual.less` file, which holds the visual's styles.

## capabilities.json

This file contains the main properties and settings (or [capabilities](#)) for the visual. It allows the visual to declare supported features, objects, properties, and [data view mapping](#).

## package-lock.json

This file is automatically generated for any operations where *npm* modifies either the `node_modules` tree, or the `package.json` file.

For more information about this file, see the official [npm-package-lock.json](#) documentation.

## package.json

This file describes the project package. It contains information about the project such as authors, description, and project dependencies.

For more information about this file, see the official [npm-package.json](#) documentation.

## pbviz.json

This file contains the visual metadata.

To view an example `pbviz.json` file with comments describing the metadata entries, see the [metadata entries](#) section.

## tsconfig.json

A configuration file for [TypeScript](#).

This file must contain the path to \*.ts file where the main class of the visual is located, as specified in the `visualClassName` property in the `pbviz.json` file.

## tslint.json

This file contains the [TSLint configuration](#).

# Metadata entries

The comments in the following code caption from the `pbviz.json` file, describe the metadata entries.

### NOTE

- From version 3.x.x of the `pbviz` tool, `externalJS` isn't supported.
- For localization support, [add the Power BI locale to your visual](#).

```
{
  "visual": {
    // The visual's internal name.
    "name": "<visual project name>",

    // The visual's display name.
    "displayName": "<visual project name>",

    // The visual's unique ID.
    "guid": "<visual project name>23D8B823CF134D3AA7CC0A5D63B20B7F",

    // The name of the visual's main class. Power BI creates the instance of this class to start using the visual in a Power BI report.
    "visualClassName": "Visual",

    // The visual's version number.
    "version": "1.0.0",

    // The visual's description (optional)
    "description": "",

    // A URL linking to the visual's support page (optional).
    "supportUrl": "",

    // A link to the source code available from GitHub (optional).
    "gitHubUrl": "",

  },
  // The version of the Power BI API the visual is using.
  "apiVersion": "2.6.0",

  // The name of the visual's author and email.
  "author": { "name": "", "email": "" },

  // 'icon' holds the path to the icon file in the assets folder; the visual's display icon.
  "assets": { "icon": "assets/icon.png" },

  // Contains the paths for JS libraries used in the visual.
  // Note: externalJS' isn't used in the Power BI visuals tool version 3.x.x or higher.
  "externalJS": null,

  // The path to the 'visual.less' style file.
  "style": "style/visual.less",

  // The path to the `capabilities.json` file.
  "capabilities": "capabilities.json",

  // The path to the `dependencies.json` file which contains information about R packages used in R based visuals.
  "dependencies": null,

  // An array of paths to files with localizations.
  "stringResources": []
}
}
```

## Next steps

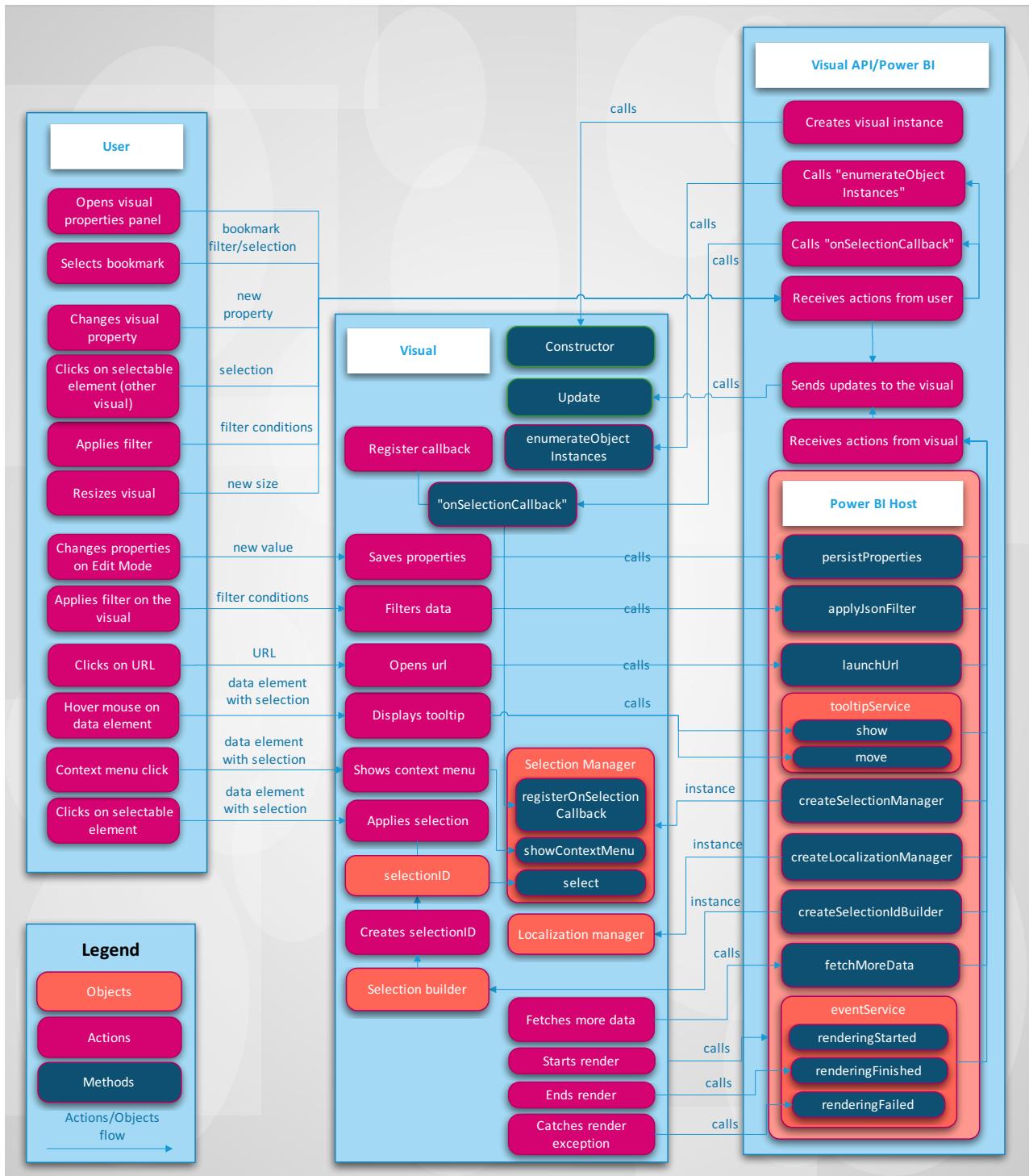
- To understand the interactions between a visual, a user, and Power BI, see [Power BI visual concept](#).
- Start developing your own Power BI visuals from scratch, using the [step by step guide](#).

# Visuals in Power BI

5/14/2020 • 4 minutes to read • [Edit Online](#)

The article describes how visuals integrate with Power BI and how a user can interact with a visual in Power BI.

The following figure depicts how common visual-based actions that a user takes, like selecting a bookmark, are processed in Power BI.



## Visuals get updates from Power BI

A visual calls an `update` method to get updates from Power BI. The `update` method usually contains the main logic of the visual and is responsible for rendering a chart or visualizing data.

Updates are triggered when the visual calls the `update` method.

## Action and update patterns

Actions and subsequent updates in Power BI visuals occur in one of these three patterns:

- User interacts with a visual through Power BI.
- User interacts with the visual directly.
- Visual interacts with Power BI.

### User interacts with a visual through Power BI

- A user opens the visual's properties panel.

When a user opens the visual's properties panel, Power BI fetches supported objects and properties from the visual's `capabilities.json` file. To receive actual values of properties, Power BI calls the `enumerateObjectInstances` method of the visual. The visual returns actual values of properties.

For more information, see [Capabilities and properties of Power BI visuals](#).

- A user [changes a property of the visual](#) in the format panel.

When a user changes the value of a property in the format panel, Power BI calls the `update` method of the visual. Power BI passes in the new `options` object to the `update` method. The objects contain the new values.

For more information, see [Objects and properties of Power BI visuals](#).

- A user resizes the visual.

When a user changes the size of a visual, Power BI calls the `update` method with the new `options` object. The `options` objects have nested `viewport` objects that contain the new width and height of the visual.

- A user applies a filter at the report, page, or visual level.

Power BI filters data based on filter conditions. Power BI calls the `update` method of the visual to update the visual with new data.

The visual gets a new update of the `options` objects when there's new data in one of the nested objects. How the update occurs depends on the data view mapping configuration of the visual.

For more information, see [Understand data view mapping in Power BI visuals](#).

- A user selects a data point in another visual in the report.

When a user selects a data point in another visual in the report, Power BI filters or highlights the selected data points and calls the visual's `update` method. The visual gets new filtered data, or it gets the same data with an array of highlights.

For more information, see [Highlight data points in Power BI visuals](#).

- A user selects a bookmark in the bookmarks panel of the report.

When a user selects a bookmark in the report's bookmarks panel, one of two actions can occur:

- Power BI calls a function that's passed and registered by the `registerOnSelectionCallback` method. The callback function gets arrays of selections for the corresponding bookmark.
- Power BI calls the `update` method with a corresponding `filter` object inside the `options` object.

In either case, the visual must change its state according to the received selections or `filter` object.

For more information about bookmarks and filters, see [Visual Filters API in Power BI visuals](#).

### User interacts with the visual directly

- A user hovers the mouse over a data element.

A visual can display more information about a data point through the Power BI Tooltips API. When a user hovers the mouse over a visual element, the visual can handle the event and display data about the associated tooltip element. The visual can display either a standard tooltip or a report page tooltip.

For more information, see [Toolips in Power BI visuals](#).

- A user changes visual properties. (For example, a user expands a tree and the visual saves state in the visual properties.)

A visual can save properties values thought the Power BI API. For example, when a user interacts with the visual and the visual needs to save or update properties values, the visual can call the `persistProperties` method.

- A user selects a URL.

By default, a visual can't open a URL directly. Instead, to open a URL in a new tab, the visual can call the `launchUrl` method and pass the URL as a parameter.

For more information, see [Create a launch URL](#).

- A user applies a filter through the visual.

A visual can call the `applyJsonFilter` method and pass conditions to filter for data in other visuals. Several types of filters are available, including Basic, Advanced, and Tuple filters.

For more information, see [Visual Filters API in Power BI visuals](#).

- A user selects elements in the visual.

For more information about selections in a Power BI visual, see [Add interactivity by using Power BI visual selections](#).

### Visual interacts with Power BI

- A visual requests more data from Power BI.

A visual processes data part by part. The `fetchMoreData` API method requests the next fragment of data in the dataset.

For more information, see [Fetch more data from Power BI](#).

- The event service triggers.

Power BI can export a report to PDF or send a report by e-mail (applies only to certified visuals). To notify Power BI that rendering is finished and that the visual is ready to be captured as PDF or e-mail, the visual should call the Rendering Events API.

For more information, see [Export reports from Power BI to PDF](#).

To learn about the event service, see [Render events in Power BI visuals](#).

## Next steps

Interested in creating visualizations and adding them to Microsoft AppSource? See these articles:

- [Develop a Power BI visual](#)
- [Publish Power BI visuals to Partner Center](#)



# Organizational visuals in Power BI

5/13/2020 • 2 minutes to read • [Edit Online](#)

You can use Power BI visuals in Power BI to create a unique type of visual that's tailored to you. Power BI visuals are created by developers, and they're often created when the multitude of visuals that are included in Power BI don't quite meet their need.

In some organizations, Power BI visuals are even more important – they might be necessary to convey specific data or insights unique to the organization, they may have special data requirements, or they may highlight private business methods. Such organizations need to develop Power BI visuals, share them throughout their organization, and make sure they're properly maintained. Power BI visuals lets organizations do just that.

The following image shows the process by which organization Power BI visuals in Power BI flow from administrator, through development and maintenance, all finally make their way to the data analyst.



Organizational visuals are deployed and managed by the Power BI administrator from the Admin portal. Once deployed into the organizational repository, users in the organization can easily discover them, and import the organizational Power BI visuals into their reports directly from Power BI Desktop.

To learn more about how to use organizational Power BI visuals in the reports that you created, see the following article: [Learn more about importing organizational visuals into your reports](#).

## Administer organizational Power BI visuals

To learn more about how to administer, deploy, and manage organizational Power BI visuals in your organization, see the following article: [Learn more about deployment and management of organization Power BI visuals](#).

### WARNING

A Power BI visual installed from a file, can contain code with security or privacy risks. Make sure you trust the author and the source of the Power BI visual file, before deploying it to the organization repository.

## Considerations and limitations

There are several considerations and limitations that you need to be aware of.

Admin:

- If a Power BI visual from ApSource or a file is deleted from the repository, any existing reports that use the deleted visual will stop rendering. Deleting from the repository isn't reversible. To temporarily disable a Power BI visual from ApSource or a file, use the "Disable" feature.
- Organizational Power BI visuals are not supported in Power BI report server.

End user:

- Organizational Power BI visuals are private visuals imported from the organization repository. As any private visual they can't be [exported to PowerPoint](#) or displayed in emails received when a user [subscribes to report pages](#). Only [certified Power BI visuals](#) imported directly from the marketplace supports these features.
- Visio visual, PowerApps visual, the Map box visual, and the GlobeMap visual from AppSource marketplace don't render if deployed through the organization repository.

## Troubleshoot

For information about troubleshooting, visit [Troubleshooting your Power BI visuals](#).

## FAQ

For more information and answers to questions, visit [Frequently asked questions about Power BI visuals](#).

More questions? [Try the Power BI Community](#).

# Guidelines for Power BI visuals

3/16/2020 • 5 minutes to read • [Edit Online](#)

Before you [publish](#) your Power BI visual to Microsoft AppSource for others to discover and use, make sure that you follow the guidelines to create a great experience for your users.

## Power BI visuals with additional purchases

You can submit Power BI visuals that are free to the Marketplace (Microsoft AppSource). You can also submit to Microsoft AppSource Power BI visuals that have an "additional purchase may be required" price tag. "Additional purchase may be required" Power BI visuals, are similar to in-app purchase (IAP) add-ins in the Office Store.

Similarly to a free Power BI visual, an IAP Power BI visual can also be certified. Before submitting your IAP Power BI visual for certification, make sure it complies with the [certification requirements](#).

### What is a Power BI visual with IAP features?

An IAP Power BI visual is a *free* visual that offers *free features*. It also has some advanced features for which extra charges may be applied. In the Power BI visual's description, developers must notify users about the features that require additional purchases to operate. Currently, Microsoft does not provide native APIs to support the purchase of apps and add-ins.

Developers may use any third-party payment system for these purchases. For more information, see [our store policy](#).

#### IMPORTANT

If you update your Power BI visual from free to "Additional purchase may be required", users must receive the same level of free functionality as before the update. You may add optional advanced paid features in addition to the existing free features.

### Watermarks

You can use watermarks so that customers continue using the IAP advanced features without paying.

Watermarks can be used to showcase the full functionality of the Power BI visual, before a purchase is made.

- Watermarks may only be used on paid features that are used without a valid license.
- Watermarks are not allowed in Power BI visuals with a *free* price tag.
- Watermarks are not allowed in IAP visuals, when the user uses free features.

### Pop-up window

You can use a pop-up window to explain how to purchase a license, when an invalid (or expired) license is used with your Power BI IAP visual.

### Submission process

Follow the [submission process](#) and then navigate to the *Product setup* tab and check the *My product requires the purchase of a service* check box.

After the Power BI visual is validated and approved, the Microsoft AppSource listing for the IAP Power BI visual states, "Additional purchase may be required" under the pricing options.

## Context menu

Context menu is the right-click menu that is displayed when the user is hovering over a visual. All Power BI visuals should enable the context menu to bring a unified experience. Please check [this article](#) to learn how to add a context menu.

## Commercial logo

This section describes the specifications for adding commercial logos in Power BI visuals. Commercial logos are not mandatory. If added they must follow these guidelines.

### NOTE

- In this article, 'commercial logo' refers to any commercial company icon as described in the pictures below.
- The Microsoft commercial logo is used in this article only as an example. Use your own commercial logo with your Power BI visual.

### IMPORTANT

Commercial logos are allowed in *edit mode only*. Commercial logos *can't* be displayed in view mode.

### Commercial logo type

There are three types of commercial logos:

- **Logo** - A logo is comprised of two elements locked together, an icon and a name.



- **Symbol** - A graphic without any text.



- **Logotype** - A logo without an icon, comprised only from text.



### Commercial logo color

When using a commercial logo, the color of the logo must be grey (hex color #C8C8C8). Don't add effects such as gradients to the commercial logo.

- **Logo**



- **Symbol** - A graphic without any text.



- **Logotype** - A logo without an icon, comprised only from text.



### TIP

- If your Power BI visual contains a graphic, consider adding a white background with 10 px margins to your logo.
- Consider adding dropshadow to your logo (30% opacity black).

## Commercial logo size

A Power BI visual requires two commercial logos, one for large tiles and one for small tiles. Place the logo within a bounding box placed at the top or bottom right corner, with 4 px margins.

The following table describes the size considerations for Power BI visuals.

	SMALL POWER BI VISUAL	LARGE POWER BI VISUAL
<i>Logo width</i>	Up to 240 px	Greater than 240 px
<i>Logo height</i>	Up to 160 px	Greater than 160 px
<i>Bounding box size</i>	40 x 15 px	101 x 30 px
<i>Commercial logo example</i>		
<i>Bounding box example</i>		

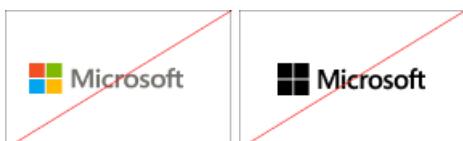
## Commercial logo behavior

Commercial logos are only allowed in edit mode. When clicked, a commercial logo can only include the following functionality:

- Clicking the commercial logo redirects to your website.
- Clicking the commercial logo opens a popup window with additional information. The popup window should be divided into two sections:
  - A marketing area which can include the the commercial logo, a visual and market ratings.
  - An information area which can include information and links.

## Things to avoid

- Commercial logos cannot be displayed in view mode.
- An animated commercial logo can display animation for up to five seconds.
- If your Power BI visual includes informative icons (i) in reading mode, they should comply to the color, size, and location of the commercial logo, as described above.
- Avoid a colorful or a black commercial logo. The commercial logo must be grey (hex color #C8C8C8).



- A commercial logo with effects such as gradients or strong shadows.



## Best practices

When publishing a Power BI visual, consider the following recommendations in order to provide users a great experience.

### **Visual landing page**

Use the landing page to clarify to users how they can use your Power BI visual and where to purchase the license. Don't include videos that are automatically triggered. Add only material that helps improve the user's experience, such as information or links to license purchasing details and how to use IAP features.

### **License key and token**

For the user's convenience, add the license key or token related fields at the top of the format pane.

## FAQ

For more information about Power BI visuals, see [Frequently asked questions about Power BI visuals with additional purchases](#).

## Next steps

Learn how you can publish your Power BI visual to [Microsoft AppSource](#) for others to discover and use.

# How to build a high performance Power BI visual

4/28/2020 • 2 minutes to read • [Edit Online](#)

This article will cover techniques on how a developer can achieve high performance when rendering visuals.

No one wants a visual to take its time when rendering and squeezing every drop of performance you can out of code becomes critical when rendering.

## NOTE

As we continue to improve and enhance the platform, new versions of the API are constantly being released. In order to get the most out of the Power BI visuals' platform and feature set, it's recommended you keep up-to-date with the most recent version.

Since the latest **version 2.1**, Power BI visual load times have improved on average by 20%.

## Power BI visual performance tips

Here are some recommendations on how to achieve optimal visual performance.

### Use User Timing API

Using the **User Timing API** to measure your app's JavaScript performance can help you decide which parts of the script need optimization.

For more information, see the [User Timing API](#).

### Review animation loops

Does the animation loop redraw unchanged elements?

- Problem: It wastes time to draw elements that don't change from frame-to-frame.
- Solution: Update frames selectively.

When the time comes to animate static visualizations, it's tempting to lump draw code into one update function and repeatedly call it with new data for each iteration of the animation loop.

Instead consider the following update pattern, use a visual constructor method to draw everything static, then the update function only needs to draw visualization elements that change.

## TIP

Inefficient animation loops are commonly found in axes and legends.

### Cache DOM Nodes

When a node or list of nodes is retrieved from the DOM, you need to think about whether you can reuse them in later computations (sometimes even the next loop iteration). As long as you don't need to add or delete additional nodes in the relevant area, caching them can improve your application's overall efficiency.

To make sure that your code is fast and doesn't slow down the browser, keep DOM access to a minimum.

- Before:

```
public update(options: VisualUpdateOptions) {
    let axis = $(".axis");
}
```

- After:

```
public constructor(options: VisualConstructorOptions) {
    this.$root = $(options.element);
    this.xAxis = this.$root.find(".xAxis");
}

public update(options: VisualUpdateOptions) {
    let axis = this.axis;
}
```

## Avoid DOM manipulation

Limit DOM manipulation as much as possible. Insert operations like `prepend()`, `append()`, and `after()` are time-consuming and shouldn't be used unless necessary.

For instance:

```
for (let i=0; i<1000; i++) {
    $('#list').append('<li>' + i + '</li>');
}
```

The above example could be quickened using `html()` and building the list beforehand:

```
let list = '';
for (let i=0; i<1000; i++) {
    list += '<li>' + i + '</li>';
}

$('#list').html(list);
```

## Reconsider JQuery

Limiting your JS frameworks and using native JS whenever possible can increase the available bandwidth and lower your processing overhead. This can also limit compatibility issues with older browsers.

For more information, see [youmightnotneedjquery.com](http://youmightnotneedjquery.com) for alternative examples to functions such as JQuery's `show`, `hide`, `addClass`, and more.

## Use canvas or WebGL

For repeated use of animations consider using **Canvas** or **WebGL** instead of SVG. Unlike SVG, with these options performance is determined by size rather than content.

You can read more about the differences in [SVG vs Canvas: How to Choose](#).

## Use `requestAnimationFrame` instead of `setTimeout`

If you use `requestAnimationFrame` to update your on-screen animations, your animation functions are called before the browser calls another repaint.

For more information, see this [sample](#) on smooth animation using `requestAnimationFrame`.

## Next steps

Learn more about optimization techniques in the [Optimization guide for Power BI](#).

# Power BI visuals FAQ

5/11/2020 • 6 minutes to read • [Edit Online](#)

## Organizational Power BI visuals

The admin portal enables managing Power BI visuals for your organization.

### **How can the admin manage organizational Power BI visuals?**

In the Admin portal, under the *Organizational visuals* tab, the admin can see and [manage all the organizational Power BI visuals in the enterprise](#). This includes adding, disabling, enabling, and deleting Power BI visuals.

Users in the organization can easily find Power BI visuals, and import them into their reports directly from Power BI Desktop or Service.

Once the admin uploads a new version of an organizational Power BI visual, everyone in the organization gets the same updated version. All reports using updated Power BI visuals are automatically updated.

Users can find the organizational Power BI visuals in the built-in Power BI Desktop and Power BI service organization store, under the *MY ORGANIZATION* tab.

### **If an admin uploads a Power BI visual from the public marketplace to the organization store, is it automatically updated once a vendor updates the visual in the public marketplace?**

No, there's no automatic update from the public marketplace. It's the Admin's responsibility to update the organizational Power BI visual version.

### **Is there a way to disable the organization store?**

No, users always see the *MY ORGANIZATION* tab in Power BI desktop and Power BI service. If an admin disables or deletes all the organizational Power BI visuals from the admin portal, the organizational store will be empty.

### **If the admin disables Power BI visuals from the Admin portal (tenant settings) do users still have access to the organizational Power BI visuals?**

Yes, if the admin disables the Power BI visuals from the admin portal, it doesn't affect the organizational store.

Some organizations disable Power BI visuals and enable only hand-picked visuals that were imported and uploaded by the Power BI admin to the organizational store.

Disabling the Power BI visuals from the Admin portal isn't enforced in Power BI Desktop. Desktop users can still add and use Power BI visuals from the public marketplace in their reports. However, those public Power BI visuals stop rendering once published to the Power BI Service and issue an appropriate error.

When the Power BI visuals setting in the admin portal, is enforced, users in Power BI service cannot import Power BI visuals from the public marketplace. Only visuals from the organizational store can be imported.

### **What are the advantages of Power BI visuals in the organizational store?**

- Everyone gets the same visual version, which is controlled by the Power BI admin. Once the admin updates the visual's version in the admin portal, all the users in the organization get the updated version automatically.
- No need to share visual files by email or shared folders. The organizational store offers are visible to all members who are logged-in.
- Security and supportability, new versions of organizational Power BI visuals are updated automatically in all reports.

- Admins can control which Power BI visuals are available throughout the organization.
- Admins can enable/disable visuals for testing from the admin portal.

## Certified Power BI visuals

### **What are certified Power BI visuals?**

Certified Power BI visuals are Power BI visuals that meet certain [requirements](#), and are certified by Microsoft.

In the [marketplace](#), certified Power BI visuals have a yellow badge indicating that they're certified.

Microsoft isn't the author of third-party Power BI visuals. We advise customers to contact the author directly to verify the functionality of third-party visuals.

### **What tests are done during the certification process?**

The certification process tests include but are not limited to:

- Code reviews
- Static code analysis
- Data leakage
- Data fuzzing
- Penetration testing
- Access XSS testing
- Malicious data injection
- Input validation
- Functional testing

### **Are certified Power BI visual checked again with every new submission (upgrade)?**

Yes. Every time a new version of certified visual is submitted to the Marketplace, the visual's version update goes under the same certification checks.

The version update certification is automatic. If there's a violation that causes the update to be rejected, an email is sent to the developer to explain what needs to be fixed.

### **Can a certified Power BI visual stop lose its certification after a new update?**

No, this is not possible. A certified visual can't lose its certification with a new update. The update is rejected.

### **Do I need to share my code in a public repository if I'm certifying my Power BI visual?**

No, you don't need to share your code publicly.

Provide read permissions to check the Power BI visual code. For example, by using a private repository in GitHub.

### **Does a certified Power BI visual have to be in the marketplace?**

Yes. Private visuals are not certified.

### **How long does it take to certify my visual?**

Certifying a new Power BI visual (first-time certification) can take up to four weeks.

Certifying an updated version of a Power BI visual, can take up to three weeks.

### **Does the Certification process ensure that there is no data leakage?**

The tests performed are designed to check that the visual does not access external services or resources.

Microsoft is not the author of third-party Power BI visuals. We advise customers to contact the author directly to verify the functionality of third-party Power BI visuals.

### **Are uncertified Power BI visuals safe to use?**

Uncertified Power BI visuals do not necessarily mean unsafe visuals.

Some visuals are not certified because they don't comply with one or more of the [certification requirements](#). For example, connecting to an external service like map visuals, or visuals using commercial libraries.

## Visuals with additional purchases

### What is a visual with additional purchases?

A visual with additional purchases is similar to in-app purchase(IAP) add-ins. These add-ins include an **Additional purchase may be required** price tag.

IAP Power BI visuals are free, downloadable Power BI visuals. Users pay nothing to download those Power BI visuals from the marketplace.

IAP visuals offer optional in-app purchases for advanced features.

### What is changing in the submission process?

The IAP Power BI visuals submission process to the marketplace, is the same process as the one for free Power BI visuals. You can submit a Power BI visual to be certified using [Partner Center](#).

When registering your Power BI visual, navigate to the *Product setup* tab and check the *My product requires the purchase of a service* check box.

### What should I do before submitting my IAP Power BI visual?

If you're working on an IAP Power BI visual, make sure that it complies with the [guidelines](#).

#### NOTE

Power BI free visuals with an added IAP feature, must keep the same free features previously offered. You can add optional advanced paid features on top of the old free features. We recommend submitting the IAP Power BI visual with the advanced features as a new Power BI visual, and not to update the old free one.

### Do IAP Power BI visuals need to be certified?

The [certification](#) process is optional. It is up to the developer to decide whether to certify their IAP Power BI visual or not.

### Can I get my IAP Power BI visual certified?

Yes, once your IAP Power BI visual is approved by the AppSource team, you can submit your Power BI visual to be [certified](#).

Certification is an optional process, it's up to you to decide if you want your IAP visual to be certified.

## Additional questions

### How to get support?

Feel free to contact the Power BI visuals support team at [pbicvssupport@microsoft.com](mailto:pbicvssupport@microsoft.com), with any questions, comments, or issues you have.

## Next steps

For more information, visit [Troubleshooting your Power BI visuals](#).

# Capabilities and properties of Power BI visuals

3/13/2020 • 3 minutes to read • [Edit Online](#)

You use capabilities to provide information to the host about your visual. All properties on the capabilities model are [optional](#).

The root objects of a visual's capabilities are `dataRoles`, `dataViewMappings`, and so on.

```
{  
    "dataRoles": [ ... ],  
    "dataViewMappings": [ ... ],  
    "objects": { ... },  
    "supportsHighlight": true|false,  
    "advancedEditModeSupport": 0|1|2,  
    "sorting": { ... }  
}
```

## Define the data fields that your visual expects: `dataRoles`

To define fields that can be bound to data, you use `dataRoles`. `dataRoles` takes an array of `DataViewRole` objects, which defines all the required properties.

### Properties

- **name:** The internal name of this data field (must be unique).
- **kind:** The kind of field:
  - `Grouping` : Discrete values that are used to group measure fields.
  - `Measure` : Numeric data values.
  - `GroupingOrMeasure` : Values that can be used as either a grouping or a measure.
- **displayName:** The name displayed to the user in the **Properties** pane.
- **description:** A short description of the field (optional).
- **requiredTypes:** The required type of data for this data role. Values that don't match are set to null (optional).
- **preferredTypes:** The preferred type of data for this data role (optional).

### Valid data types in `requiredTypes` and `preferredTypes`

- **bool:** A boolean value
- **integer:** An integer (whole number) value
- **numeric:** A numeric value
- **text:** A text value
- **geography:** A geographic data

### Example

```
"dataRoles": [  
    {  
        "displayName": "My Category Data",  
        "name": "myCategory",  
        "kind": "Grouping",  
        "requiredTypes": [  
            {  
                "text": true  
            }  
        ]  
    }  
]
```

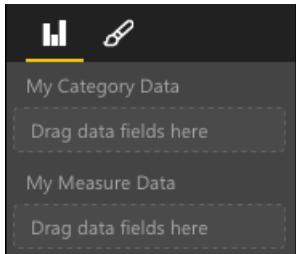
```
        },
        {
            "numeric": true
        },
        {
            "integer": true
        }
    ],
    "preferredTypes": [
        {
            "text": true
        }
    ]
},
{
    "displayName": "My Measure Data",
    "name": "myMeasure",
    "kind": "Measure",
    "requiredTypes": [
        {
            "integer": true
        },
        {
            "numeric": true
        }
    ],
    "preferredTypes": [
        {
            "integer": true
        }
    ]
},
{
    "displayNameKey": "Visual_Location",
    "name": "Locations",
    "kind": "Measure",
    "displayName": "Locations",
    "requiredTypes": [
        {
            "geography": {
                "address": true
            }
        },
        {
            "geography": {
                "city": true
            }
        },
        {
            "geography": {
                "continent": true
            }
        },
        {
            "geography": {
                "country": true
            }
        },
        {
            "geography": {
                "county": true
            }
        },
        {
            "geography": {
                "place": true
            }
        }
    ],
    "preferredTypes": [
        {
            "text": true
        }
    ]
}
```

```

        "geography": {
            "postalCode": true
        }
    },
{
    "geography": {
        "region": true
    }
},
{
    "geography": {
        "stateOrProvince": true
    }
}
]
}
]

```

The preceding data roles would create the fields that are displayed in the following image:



## Define how you want the data mapped: dataViewMappings

A DataViewMappings property describes how the data roles relate to each other and allows you to specify conditional requirements for them.

Most visuals provide a single mapping, but you can provide multiple dataViewMappings. Each valid mapping produces a data view.

```

"dataViewMappings": [
{
    "conditions": [ ... ],
    "categorical": { ... },
    "table": { ... },
    "single": { ... },
    "matrix": { ... }
}
]

```

For more information, see [Understand data view mapping in Power BI visuals](#).

## Define property pane options: objects

Objects describe customizable properties that are associated with the visual. Each object can have multiple properties, and each property has a type that's associated with it. Types refer to what the property will be.

```

"objects": {
    "myCustomObject": {
        "displayName": "My Object Name",
        "properties": { ... }
    }
}

```

For more information, see [Objects and properties of Power BI visuals](#).

## Handle partial highlighting: supportsHighlight

By default, this value is set to `false`, which means that your values are automatically filtered when something on the page is selected. This automatic filtering in turn updates your visual to display only the selected value. If you want to display the full data but highlight only the selected items, you need to set `supportsHighlight` to `true` in your `capabilities.json` file.

For more information, see [Highlight data points in Power BI visuals](#).

## Handle advanced edit mode: advancedEditModeSupport

A visual can declare its support of advanced edit mode. By default, a visual doesn't support advanced edit mode, unless stated otherwise in the `capabilities.json` file.

For more information, see [Advanced edit mode in Power BI visuals](#).

## Data sorting options for visual: sorting

A visual can define its sorting behavior via its capabilities. By default, a visual doesn't support modifying its sorting order, unless stated otherwise in the `capabilities.json` file.

For more information, see [Sorting options for Power BI visuals](#).

# Understand data view mapping in Power BI visuals

3/13/2020 • 12 minutes to read • [Edit Online](#)

This article discusses data view mapping and describes how data roles relate to each other and allow you to specify conditional requirements for them. The article also describes each `dataMappings` type.

Each valid mapping produces a data view, but we currently support performing only one query per visual. You ordinarily get only one data view. However, you can provide multiple data mappings in certain conditions, which allow:

```
"dataViewMappings": [
  {
    "conditions": [ ... ],
    "categorical": { ... },
    "single": { ... },
    "table": { ... },
    "matrix": { ... }
  }
]
```

Power BI creates a mapping to a data view if and only if the valid mapping is filled in `dataViewMappings`.

In other words, `categorical` might be defined in `dataViewMappings` but other mappings, such as `table` or `single`, might not be. For example:

```
"dataViewMappings": [
  {
    "categorical": { ... }
  }
]
```

Power BI produces a data view with a single `categorical` mapping, and `table` and other mappings are undefined:

```
{
  "categorical": {
    "categories": [ ... ],
    "values": [ ... ]
  },
  "metadata": { ... }
}
```

## Conditions

This section describes conditions for a particular data mapping. You can provide multiple sets of conditions and, if the data matches one of the described sets of conditions, the visual accepts the data as valid.

Currently, for each field, you can specify a minimum and maximum value. The value represents the number of fields that can be bound to that data role.

### NOTE

If a data role is omitted in the condition, it can have any number of fields.

## Example 1

You can drag multiple fields into each data role. In this example, you limit the category to one data field and the measure to two data fields.

```
"conditions": [
    { "category": { "max": 1 }, "y": { "max": 2 } },
]
```

## Example 2

In this example, either of two conditions is required:

- Exactly one category data field and exactly two measures
- Exactly two categories and exactly one measure.

```
"conditions": [
    { "category": { "min": 1, "max": 1 }, "measure": { "min": 2, "max": 2 } },
    { "category": { "min": 2, "max": 2 }, "measure": { "min": 1, "max": 1 } }
]
```

## Single data mapping

Single data mapping is the simplest form of data mapping. It accepts a single measure field and gives you the total. If the field is numeric, it gives you the sum. Otherwise, it gives you a count of unique values.

To use single data mapping, you need to define the name of the data role that you want to map. This mapping works only with a single measure field. If a second field is assigned, no data view is generated, so it's also a good practice to include a condition that limits the data to a single field.

### NOTE

This data mapping can't be used in conjunction with any other data mapping. It's meant to reduce data into a single numeric value.

## Example 3

```
{
    "dataRoles": [
        {
            "displayName": "Y",
            "name": "Y",
            "kind": "Measure"
        }
    ],
    "dataViewMappings": [
        {
            "conditions": [
                {
                    "Y": {
                        "max": 1
                    }
                }
            ],
            "single": {
                "role": "Y"
            }
        }
    ]
}
```

The resulting data view still contains the other types (table, categorical, and so on), but each mapping contains only the single value. The best practice is to access the value only in single.

```
{
    "dataView": [
        {
            "metadata": null,
            "categorical": null,
            "matrix": null,
            "table": null,
            "tree": null,
            "single": {
                "value": 94163140.3560001
            }
        }
    ]
}
```

Code sample to process simple data view mapping

```

"use strict";
import powerbi from "powerbi-visuals-api";
import DataView = powerbi.DataView;
import DataViewSingle = powerbi.DataViewSingle;
// standart imports
// ...

export class Visual implements IVisual {
    private target: HTMLElement;
    private host: IVisualHost;
    private valueText: HTMLParagraphElement;

    constructor(options: VisualConstructorOptions) {
        // constructor body
        this.target = options.element;
        this.host = options.host;
        this.valueText = document.createElement("p");
        this.target.appendChild(this.valueText);
        // ...
    }

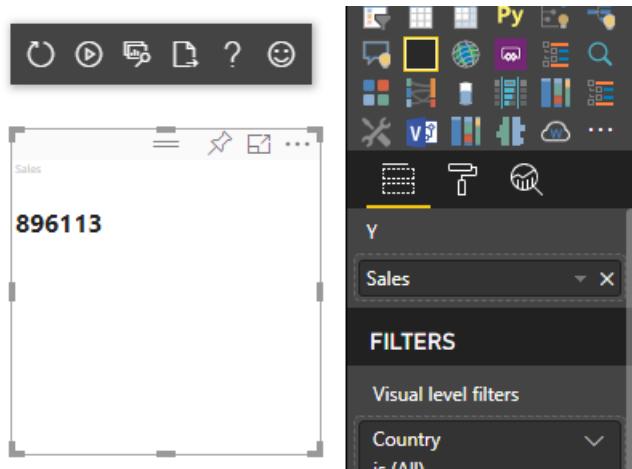
    public update(options: VisualUpdateOptions) {
        const dataView: DataView = options.dataViews[0];
        const singleDataView: DataViewSingle = dataView.single;

        if (!singleDataView ||
            !singleDataView.value ) {
            return
        }

        this.valueText.innerText = singleDataView.value.toString();
    }
}

```

As a result the visual displays a single value from Power BI:



## Categorical data mapping

Categorical data mapping is used to get one or two independent groupings of data.

### Example 4

Here is the definition from the previous example for data roles:

```

"dataRole": [
    {
        "displayName": "Category",
        "name": "category",
        "kind": "Grouping"
    },
    {
        "displayName": "Y Axis",
        "name": "measure",
        "kind": "Measure"
    }
]

```

Here is the mapping:

```

"dataViewMappings": {
    "categorical": {
        "categories": {
            "for": { "in": "category" }

        },
        "values": {
            "select": [
                { "bind": { "to": "measure" } }
            ]
        }
    }
}

```

It's a simple example. It reads "Map my `category` data role so that for every field I drag into `category`, its data is mapped to `categorical.categories`. Also map my `measure` data role to `categorical.values`."

- **for...in:** For all the items in this data role, include them in the data query.
- **bind...to:** Produces the same result as in *for...in*, but expects that the data role will have a condition restricting it to a single field.

### Example 5

This example uses the first two data roles from the previous example and additionally defines `grouping` and `measure2`.

```

"dataRole": [
    {
        "displayName": "Category",
        "name": "category",
        "kind": "Grouping"
    },
    {
        "displayName": "Y Axis",
        "name": "measure",
        "kind": "Measure"
    },
    {
        "displayName": "Grouping with",
        "name": "grouping",
        "kind": "Grouping"
    },
    {
        "displayName": "X Axis",
        "name": "measure2",
        "kind": "Grouping"
    }
]

```

Here is the mapping:

```

"dataViewMappings": {
    "categorical": {
        "categories": {
            "for": { "in": "category" }
        },
        "values": {
            "group": {
                "by": "grouping",
                "select": [
                    { "bind": { "to": "measure" } },
                    { "bind": { "to": "measure2" } }
                ]
            }
        }
    }
}

```

Here the difference is in how we are mapping categorical.values. We are saying that "Map my `measure` and `measure2` data roles to be grouped by the data role `grouping`."

### Example 6

Here are the data roles:

```

"dataRoles": [
    {
        "displayName": "Categories",
        "name": "category",
        "kind": "Grouping"
    },
    {
        "displayName": "Measures",
        "name": "measure",
        "kind": "Measure"
    },
    {
        "displayName": "Series",
        "name": "series",
        "kind": "Measure"
    }
]

```

Here is the data view mapping:

```

"dataViewMappings": [
    {
        "categorical": {
            "categories": {
                "for": {
                    "in": "category"
                }
            },
            "values": {
                "group": {
                    "by": "series",
                    "select": [
                        {
                            "for": {
                                "in": "measure"
                            }
                        }
                    ]
                }
            }
        }
    }
]

```

The categorical data view could be visualized like this:

CATEGORICAL	Year	2013	2014	2015	2016
Country					
USA		x	x	650	350
Canada		x	630	490	x
Mexico		645	x	x	x
UK		x	x	831	x

Power BI produces it as the categorical data view. It's the set of categories.

```
{  
    "categorical": {  
        "categories": [  
            {  
                "source": {...},  
                "values": [  
                    "Canada",  
                    "USA",  
                    "UK",  
                    "Mexico"  
                ],  
                "identity": [...],  
                "identityFields": [...],  
            }  
        ]  
    }  
}
```

Each category maps to a set of values as well. Each of these values is grouped by series, which is expressed as years.

For example, each `values` array represents data for each year. Also each `values` array has 4 values, for Canada, USA, UK and Mexico respectively:

```
{
  "values": [
    // Values for 2013 year
    {
      "source": {...},
      "values": [
        null, // Value for `Canada` category
        null, // Value for `USA` category
        null, // Value for `UK` category
        645 // Value for `Mexico` category
      ],
      "identity": [...],
    },
    // Values for 2014 year
    {
      "source": {...},
      "values": [
        630, // Value for `Canada` category
        null, // Value for `USA` category
        null, // Value for `UK` category
        null // Value for `Mexico` category
      ],
      "identity": [...],
    },
    // Values for 2015 year
    {
      "source": {...},
      "values": [
        490, // Value for `Canada` category
        650, // Value for `USA` category
        831, // Value for `UK` category
        null // Value for `Mexico` category
      ],
      "identity": [...],
    },
    // Values for 2016 year
    {
      "source": {...},
      "values": [
        null, // Value for `Canada` category
        350, // Value for `USA` category
        null, // Value for `UK` category
        null // Value for `Mexico` category
      ],
      "identity": [...],
    }
  ]
}
```

Code sample for processing categorical data view mapping is described below. The sample creates Hierarchical structure `Country => Year => Value`

```

"use strict";
import powerbi from "powerbi-visuals-api";
import DataView = powerbi.DataView;
import DataViewDataViewCategoricalSingle = powerbi.DataViewCategorical;
import DataViewValueColumnGroup = powerbi.DataViewValueColumnGroup;
import PrimitiveValue = powerbi.PrimitiveValue;
// standart imports
// ...

export class Visual implements IVisual {
    private target: HTMLElement;
    private host: IVisualHost;
    private categories: HTMLElement;

    constructor(options: VisualConstructorOptions) {
        // constructor body
        this.target = options.element;
        this.host = options.host;
        this.categories = document.createElement("pre");
        this.target.appendChild(this.categories);
        // ...
    }

    public update(options: VisualUpdateOptions) {
        const dataView: DataView = options.dataViews[0];
        const categoricalDataView: DataViewCategorical = dataView.categorical;

        if (!categoricalDataView ||
            !categoricalDataView.categories ||
            !categoricalDataView.categories[0] ||
            !categoricalDataView.values) {
            return;
        }

        // Categories have only one column in data buckets
        // If you want to support several columns of categories data bucket, you should iterate
        categoricalDataView.categories.array.
        const categoryFieldIndex = 0;
        // Measure has only one column in data buckets.
        // If you want to support several columns on data bucket, you should iterate years.values array in map
        function
            const measureFieldIndex = 0;
            let categories: PrimitiveValue[] = categoricalDataView.categories[categoryFieldIndex].values;
            let values: DataViewValueColumnGroup[] = categoricalDataView.values.grouped();

            let data = {};
            // iterate categories/countries
            categories.map((category: PrimitiveValue, categoryIndex: number) => {
                data[category.toString()] = {};
                // iterate series/years
                values.map((years: DataViewValueColumnGroup) => {
                    if (!data[category.toString()][years.name] &&
years.values[measureFieldIndex].values[categoryIndex]) {
                        data[category.toString()][years.name] = []
                    }
                    if (years.values[0].values[categoryIndex]) {
                        data[category.toString()]
                    }
                    [years.name].push(years.values[measureFieldIndex].values[categoryIndex]);
                })
            });
        });

        this.categories.innerText = JSON.stringify(data, null, 6);
        console.log(data);
    }
}

```

The result of the visual:

The screenshot shows a Power BI visualization titled "Sales and Count of Field by Country and Year". On the left, there is a JSON representation of the data:

```
{  
    "Canada": {  
        "2014": [  
            630  
        ],  
        "2015": [  
            490  
        ]  
    },  
    "USA": {  
        "2015": [  
            650  
        ],  
        "2016": [  
            350  
        ]  
    },  
    "UK": {  
        "2015": [  
            831  
        ]  
    },  
    "Mexico": {  
        "2013": [  
            645  
        ]  
    }  
}
```

On the right, there is a table with three columns: Country, Year, and Sales. The data is as follows:

Country	Year	Sales
Canada	2014	630
Canada	2015	490
Mexico	2013	645
UK	2015	831
USA	2015	650
USA	2016	350
Total		3596

## Table data mapping

The table data view is a simple data mapping. Essentially, it's a list of data points, where numeric data points could be aggregated.

### Example 7

With the given capabilities:

```
"dataRoles": [  
    {  
        "displayName": "Column",  
        "name": "column",  
        "kind": "Grouping"  
    },  
    {  
        "displayName": "Value",  
        "name": "value",  
        "kind": "Measure"  
    }  
]
```

```

"dataViewMappings": [
    {
        "table": {
            "rows": {
                "select": [
                    {
                        "for": {
                            "in": "column"
                        }
                    },
                    {
                        "for": {
                            "in": "value"
                        }
                    }
                ]
            }
        }
    }
]

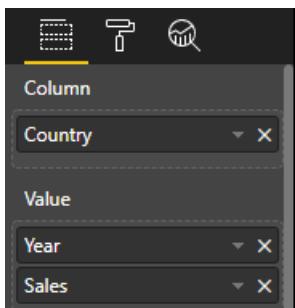
```

You can visualize the table data view as the following:

Data example:

COUNTRY	YEAR	SALES
USA	2016	100
USA	2015	50
Canada	2015	200
Canada	2015	50
Mexico	2013	300
UK	2014	150
USA	2015	75

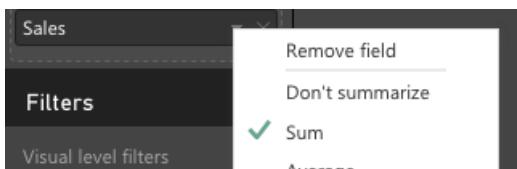
Data binding:



Power BI displays your data as the table data view. You shouldn't assume that the data is ordered.

```
{  
  "table" : {  
    "columns": [...],  
    "rows": [  
      [  
        "Canada",  
        2014,  
        630  
      ],  
      [  
        "Canada",  
        2015,  
        490  
      ],  
      [  
        "Mexico",  
        2013,  
        645  
      ],  
      [  
        "UK",  
        2014,  
        831  
      ],  
      [  
        "USA",  
        2015,  
        650  
      ],  
      [  
        "USA",  
        2016,  
        350  
      ]  
    ]  
  }  
}
```

You can aggregate the data by selecting the desired field and then selecting sum.



Code sample to process table data view mapping.

```

"use strict";
import "../../style/visual.less";
import powerbi from "powerbi-visuals-api";
// ...
import DataViewMetadataColumn = powerbi.DataViewMetadataColumn;
import DataViewTable = powerbi.DataViewTable;
import DataViewTableRow = powerbi.DataViewTableRow;
import PrimitiveValue = powerbi.PrimitiveValue;
// other imports
// ...

export class Visual implements IVisual {
    private target: HTMLElement;
    private host: IVisualHost;
    private table: HTMLParagraphElement;

    constructor(options: VisualConstructorOptions) {
        // constructor body
        this.target = options.element;
        this.host = options.host;
        this.table = document.createElement("table");
        this.target.appendChild(this.table);
        // ...
    }

    public update(options: VisualUpdateOptions) {
        const dataView: DataView = options.dataViews[0];
        const tableDataView: DataViewTable = dataView.table;

        if (!tableDataView) {
            return
        }
        while(this.table.firstChild) {
            this.table.removeChild(this.table.firstChild);
        }

        //draw header
        const tableHeader = document.createElement("thead");
        tableDataView.columns.forEach((column: DataViewMetadataColumn) => {
            const tableHeaderColumn = document.createElement("th");
            tableHeaderColumn.innerText = column.displayName
            tableHeader.appendChild(tableHeaderColumn);
        });
        this.table.appendChild(tableHeader);

        //draw rows
        tableDataView.rows.forEach((row: DataViewTableRow) => {
            const TableRow = document.createElement("tr");
            row.forEach((columnValue: PrimitiveValue) => {
                const cell = document.createElement("td");
                cell.innerText = columnValue.toString();
                TableRow.appendChild(cell);
            })
            this.table.appendChild(TableRow);
        });
    }
}

```

The visual styles file `style/visual.less` contains layout for table:

```

table {
  display: flex;
  flex-direction: column;
}

tr, th {
  display: flex;
  flex: 1;
}

td {
  flex: 1;
  border: 1px solid black;
}

```



The screenshot shows a data visualization tool with two main sections. On the left, there is a table titled "Country, Year and Sales" with three columns: Country, Year, and Sales. The data includes entries for Canada (2014, 630), Canada (2015, 490), Mexico (2013, 645), UK (2015, 831), USA (2015, 650), USA (2016, 350), and a total row for "Total" (3596). On the right, there is a matrix view with columns for "Country", "Year", and "Sales". The matrix shows the same data as the table, with rows for Canada, Mexico, UK, and USA, and a summary row for "Total". Below the matrix, there is a toolbar with various icons for navigation and data manipulation.

Country	Year	Sales
Canada	2014	630
Canada	2015	490
Mexico	2013	645
UK	2015	831
USA	2015	650
USA	2016	350
<b>Total</b>		<b>3596</b>

Country	Year	Sales
Canada	2014	630
Canada	2015	490
Mexico	2013	645
UK	2015	831
USA	2015	650
USA	2016	350
<b>Total</b>		<b>3596</b>

## Matrix data mapping

Matrix data mapping is similar to table data mapping, but the rows are presented hierarchically. Any of the data role values can be used as a column header value.

```
{
    "dataRoles": [
        {
            "name": "Category",
            "displayName": "Category",
            "displayNameKey": "Visual_Category",
            "kind": "Grouping"
        },
        {
            "name": "Column",
            "displayName": "Column",
            "displayNameKey": "Visual_Column",
            "kind": "Grouping"
        },
        {
            "name": "Measure",
            "displayName": "Measure",
            "displayNameKey": "Visual_Values",
            "kind": "Measure"
        }
    ],
    "dataViewMappings": [
        {
            "matrix": {
                "rows": {
                    "for": {
                        "in": "Category"
                    }
                },
                "columns": {
                    "for": {
                        "in": "Column"
                    }
                },
                "values": {
                    "select": [
                        {
                            "for": {
                                "in": "Measure"
                            }
                        }
                    ]
                }
            }
        }
    ]
}
```

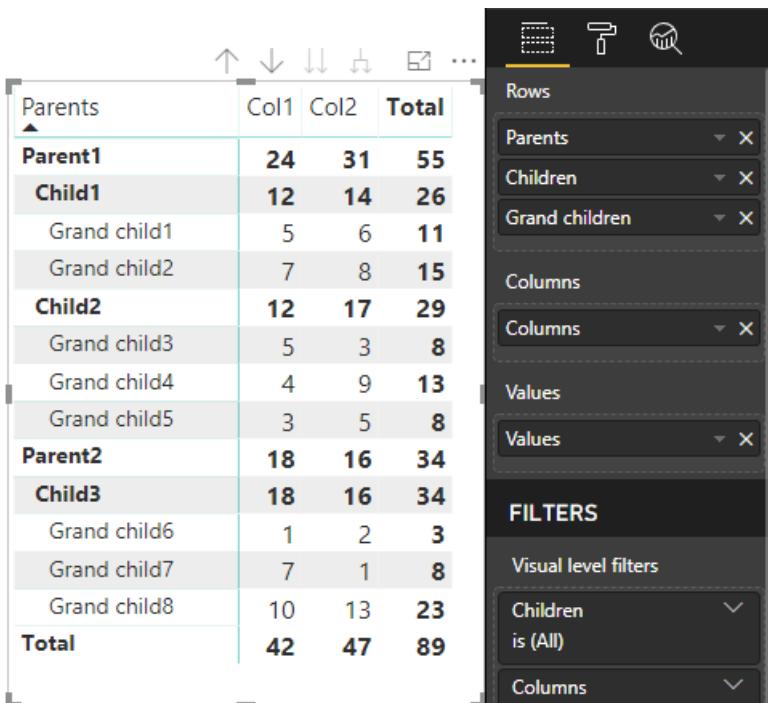
Power BI creates a hierarchical data structure. The root of the tree hierarchy includes the data from the **Parents** column of the **Category** data role, with children from the **Children** column of the data role table.

Dataset:

PARENTS	CHILDREN	GRANDCHILDREN	COLUMNS	VALUES
Parent1	Child1	Grand child1	Col1	5
Parent1	Child1	Grand child1	Col2	6
Parent1	Child1	Grand child2	Col1	7
Parent1	Child1	Grand child2	Col2	8

PARENTS	CHILDREN	GRANDCHILDREN	COLUMNS	VALUES
Parent1	Child2	Grand child3	Col1	5
Parent1	Child2	Grand child3	Col2	3
Parent1	Child2	Grand child4	Col1	4
Parent1	Child2	Grand child4	Col2	9
Parent1	Child2	Grand child5	Col1	3
Parent1	Child2	Grand child5	Col2	5
Parent2	Child3	Grand child6	Col1	1
Parent2	Child3	Grand child6	Col2	2
Parent2	Child3	Grand child7	Col1	7
Parent2	Child3	Grand child7	Col2	1
Parent2	Child3	Grand child8	Col1	10
Parent2	Child3	Grand child8	Col2	13

The core matrix visual of Power BI renders the data as a table.



The visual gets its data structure as described in the following code (only the first two table rows are shown here):

```
{
  "metadata": {...},
  "matrix": {
    "rows": {
      "levels": [...],
      "root": {
        "name": "Parent1",
        "children": [
          {"name": "Child2", "grandchildren": [{"name": "Grand child3", "value": 5}, {"name": "Grand child3", "value": 3}, {"name": "Grand child4", "value": 4}, {"name": "Grand child4", "value": 9}, {"name": "Grand child5", "value": 3}, {"name": "Grand child5", "value": 5}], "value": 5}
        ]
      }
    }
  }
}
```

```

    ...
        "childIdentityFields": [...],
        "children": [
            {
                "level": 0,
                "levelValues": [...],
                "value": "Parent1",
                "identity": {...},
                "childIdentityFields": [...],
                "children": [
                    {
                        "level": 1,
                        "levelValues": [...],
                        "value": "Child1",
                        "identity": {...},
                        "childIdentityFields": [...],
                        "children": [
                            {
                                "level": 2,
                                "levelValues": [...],
                                "value": "Grand child1",
                                "identity": {...},
                                "values": {
                                    "0": {
                                        "value": 5 // value for Col1
                                    },
                                    "1": {
                                        "value": 6 // value for Col2
                                    }
                                }
                            }
                        ],
                        ...
                    ],
                    ...
                ],
                ...
            },
            ...
        ],
        ...
    ],
    "columns": {
        "levels": [...],
        "root": {
            "childIdentityFields": [...],
            "children": [
                {
                    "level": 0,
                    "levelValues": [...],
                    "value": "Col1",
                    "identity": {...}
                },
                {
                    "level": 0,
                    "levelValues": [...],
                    "value": "Col2",
                    "identity": {...}
                },
                ...
            ]
        }
    },
    "valueSources": [...]
}
}

```

## Data reduction algorithm

To control the amount of data to receive in the data view, you can apply a data reduction algorithm.

By default, all Power BI visuals have the top data reduction algorithm applied with the `count` set to 1000 data points. It's the same as setting the following properties in the `capabilities.json` file:

```
"dataReductionAlgorithm": {  
    "top": {  
        "count": 1000  
    }  
}
```

You can modify the `count` value to any integer value up to 30000. R-based Power BI visuals can support up to 150000 rows.

## Data reduction algorithm types

There are four types of data reduction algorithm settings:

- `top` : If you want to limit the data to values taken from the top of the dataset. The top first `count` values will be taken from the dataset.
- `bottom` : If you want to limit the data to values taken from the bottom of the dataset. The last "`count`" values will be taken from the dataset.
- `sample` : Reduce the dataset by a simple sampling algorithm, limited to a `count` number of items. It means that the first and last items are included, and a `count` number of items have equal intervals between them. For example, if you have a dataset [0, 1, 2, ... 100] and a `count` of 9, you'll receive the values [0, 10, 20 ... 100].
- `window` : Loads one `window` of data points at a time containing `count` elements. Currently, `top` and `window` are equivalent. We are working toward fully supporting a windowing setting.

## Data reduction algorithm usage

The data reduction algorithm can be used in categorical, table, or matrix data view mapping.

You can set the algorithm into `categories` and/or group section of `values` for categorical data mapping.

### Example 8

```

"dataViewMappings": {
    "categorical": {
        "categories": {
            "for": { "in": "category" },
            "dataReductionAlgorithm": {
                "window": {
                    "count": 300
                }
            }
        },
        "values": {
            "group": {
                "by": "series",
                "select": [
                    {
                        "for": {
                            "in": "measure"
                        }
                    }
                ],
                "dataReductionAlgorithm": {
                    "top": {
                        "count": 100
                    }
                }
            }
        }
    }
}

```

You can apply the data reduction algorithm to the `rows` section of the Data View mapping table.

### Example 9

```

"dataViewMappings": [
    {
        "table": {
            "rows": {
                "for": {
                    "in": "values"
                },
                "dataReductionAlgorithm": {
                    "top": {
                        "count": 2000
                    }
                }
            }
        }
    }
]

```

You can apply the data reduction algorithm to the `rows` and `columns` sections of the Data View mapping matrix.

## Next steps

Read how to [add Drill-Down support for data view mappings in Power BI visuals](#).

# Objects and properties of Power BI visuals

11/8/2019 • 3 minutes to read • [Edit Online](#)

Objects describe customizable properties that are associated with a visual. An object can have multiple properties, and each property has an associated type that describes what the property will be. This article provides information about objects and property types.

`myCustomObject` is the internal name that's used to reference the object within `dataView` and `enumerateObjectInstances`.

```
"objects": {  
    "myCustomObject": {  
        "displayName": "My Object Name",  
        "properties": { ... }  
    }  
}
```

## Display name

`displayName` is the name that will be shown in the property pane.

## Properties

`properties` is a map of properties that are defined by the developer.

```
"properties": {  
    "myFirstProperty": {  
        "displayName": "firstPropertyName",  
        "type": ValueTypeDescriptor | StructuralTypeDescriptor  
    }  
}
```

### NOTE

`show` is a special property that enables a switch to toggle the object.

Example:

```
"properties": {  
    "show": {  
        "displayName": "My Property Switch",  
        "type": {"bool": true}  
    }  
}
```

## Property types

There are two property types: `ValueTypeDescriptor` and `StructuralTypeDescriptor`.

### Value type descriptor

`ValueTypeDescriptor` types are mostly primitive and are ordinarily used as a static object.

Here are some of the common `ValueTypeDescriptor` elements:

```
export interface ValueTypeDescriptor {
    text?: boolean;
    numeric?: boolean;
    integer?: boolean;
    bool?: boolean;
}
```

#### Structural type descriptor

`StructuralTypeDescriptor` types are mostly used for data-bound objects. The most common `StructuralTypeDescriptor` type is *fill*.

```
export interface StructuralTypeDescriptor {
    fill?: FillTypeDescriptor;
}
```

## Gradient property

The gradient property is a property that can't be set as a standard property. Instead, you need to set a rule for the substitution of the color picker property (*fill* type).

An example is shown in the following code:

```
"properties": {
    "showAllDataPoints": {
        "displayName": "Show all",
        "displayNameKey": "Visual_DataPoint_Show_All",
        "type": {
            "bool": true
        }
    },
    "fill": {
        "displayName": "Fill",
        "displayNameKey": "Visual_Fill",
        "type": {
            "fill": {
                "solid": {
                    "color": true
                }
            }
        }
    },
    "fillRule": {
        "displayName": "Color saturation",
        "displayNameKey": "Visual_ColorSaturation",
        "type": {
            "fillRule": {}
        },
        "rule": {
            "inputRole": "Gradient",
            "output": {
                "property": "fill",
                "selector": [
                    "Category"
                ]
            }
        }
    }
}
```

Pay attention to the *fill* and *fillRule* properties. The first is the color picker, and the second is the substitution rule for the gradient that will replace the *fill property*, *visually*, when the rule conditions are met.

This link between the *fill* property and the substitution rule is set in the `"rule" > "output"` section of the *fillRule* property.

`"Rule" > "InputRole"` property sets which data role triggers the rule (condition). In this example, if data role `"Gradient"` contains data, the rule is applied for the `"fill"` property.

An example of the data role that triggers the fill rule (`the last item`) is shown in the following code:

```
{  
  "dataRoles": [  
    {  
      "name": "Category",  
      "kind": "Grouping",  
      "displayName": "Details",  
      "displayNameKey": "Role_DisplayName_Details"  
    },  
    {  
      "name": "Series",  
      "kind": "Grouping",  
      "displayName": "Legend",  
      "displayNameKey": "Role_DisplayName_Legend"  
    },  
    {  
      "name": "Gradient",  
      "kind": "Measure",  
      "displayName": "Color saturation",  
      "displayNameKey": "Role_DisplayName_Gradient"  
    }  
  ]  
}
```

## The enumerateObjectInstances method

To use objects effectively, you need a function in your custom visual called `enumerateObjectInstances`. This function populates the property pane with objects and also determines where your objects should be bound within the `dataView`.

Here is what a typical setup looks like:

```
public enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions):  
  VisualObjectInstanceEnumeration {  
  let objectName: string = options.objectName;  
  let objectEnumeration: VisualObjectInstance[] = [];  
  
  switch( objectName ) {  
    case 'myCustomObject':  
      objectEnumeration.push({  
        objectName: objectName,  
        properties: { ... },  
        selector: { ... }  
      });  
      break;  
  };  
  
  return objectEnumeration;  
}
```

## Properties

The properties in `enumerateObjectInstances` reflect the properties that you defined in your capabilities. For an example, go to the end of this article.

## Objects selector

The selector in `enumerateObjectInstances` determines where each object is bound in the `dataView`. There are four distinct options.

### static

This object is bound to metadata `dataviews[index].metadata.objects`, as shown here.

```
selector: null
```

### columns

This object is bound to columns with the matching `QueryName`.

```
selector: {
  metadata: 'QueryName'
}
```

### selector

This object is bound to the element that you created a `selectionID` for. In this example, let's assume that we've created `selectionID`s for some data points, and that we're looping through them.

```
for (let dataPoint in dataPoints) {
  ...
  selector: dataPoint.selectionID.getSelector()
}
```

### Scope identity

This object is bound to particular values at the intersection of groups. For example, if you have categories `["Jan", "Feb", "March", ...]` and series `["Small", "Medium", "Large"]`, you might want to have an object at the intersection of values that match `Feb` and `Large`. To accomplish this, you could get the `DataViewScopeIdentity` of both columns, push them to variable `identities`, and use this syntax with the selector.

```
selector: {
  data: <DataViewScopeIdentity[]>identities
}
```

### Example

The following example shows what one objectEnumeration would look like for a `customColor` object with one property, `fill`. We want this object bound statically to `dataViews[index].metadata.objects`, as shown:

```
objectEnumeration.push({
  objectName: "customColor",
  displayName: "Custom Color",
  properties: {
    fill: {
      solid: {
        color: dataPoint.color
      }
    }
  },
  selector: null
});
```

# Advanced edit mode in Power BI visuals

3/13/2020 • 2 minutes to read • [Edit Online](#)

If you require advanced UI controls in your Power BI visual, you can take advantage of advanced edit mode. When you're in report editing mode, you select an **Edit** button to set the edit mode to **Advanced**. The visual can use the `EditMode` flag to determine whether it should display this UI control.

By default, the visual doesn't support advanced edit mode. If a different behavior is required, you can explicitly state this in the visual's `capabilities.json` file by setting the `advancedEditModeSupport` property.

The possible values are:

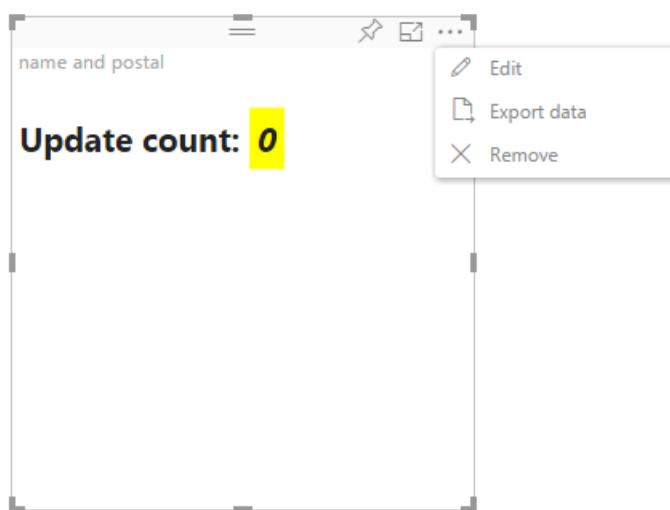
- `0` - NotSupported
- `1` - SupportedNoAction
- `2` - SupportedInFocus

## Enter advanced edit mode

An **Edit** button is displayed if:

- The `advancedEditModeSupport` property is set in the `capabilities.json` file to either `SupportedNoAction` or `SupportedInFocus`.
- The visual is viewed in report editing mode.

If `advancedEditModeSupport` property is missing from the `capabilities.json` file or set to `NotSupported`, the **Edit** button is not displayed.



When you select **Edit**, the visual gets an `update()` call with `EditMode` set to `Advanced`. Depending on the value that's set in the `capabilities.json` file, the following actions occur:

- `SupportedNoAction`: No further action is required by the host.
- `SupportedInFocus`: The host pops out the visual into in focus mode.

## Exit advanced edit mode

The **Back to report** button is displayed if:

- The `advancedEditModeSupport` property is set in the *capabilities.json* file to `SupportedInFocus`.

# Use the supportsKeyboardFocus feature

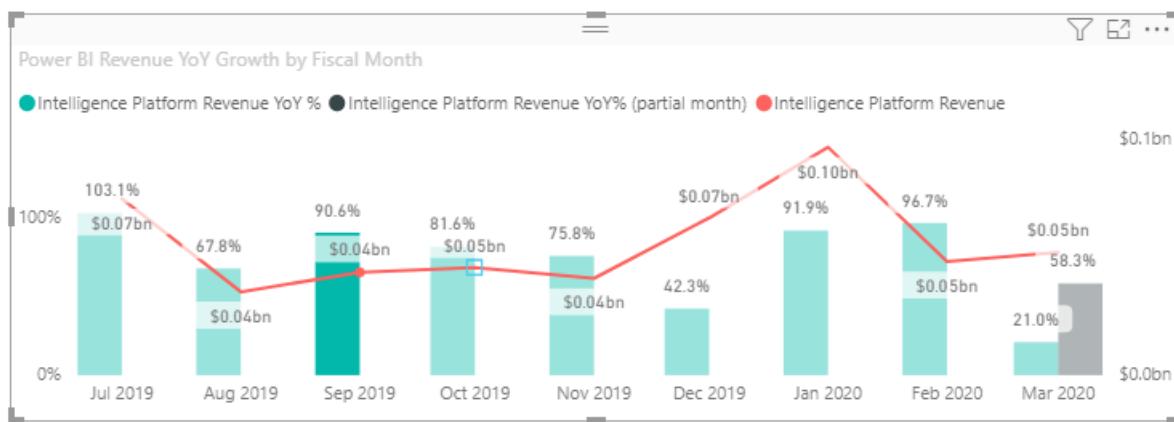
5/13/2020 • 2 minutes to read • [Edit Online](#)

This article describes how to use the `supportsKeyboardFocus` feature in Power BI visuals. The `supportsKeyboardFocus` feature allows navigating the data points of the visual using the keyboard only.

To learn more about keyboard navigation for visuals, see [Keyboard Navigation](#).

## Example

Open a visual that uses the `supportsKeyboardFocus` feature. Select any data point within the visual and select tab. The focus moves to the next data point for each time you select tab. Select enter to select the highlighted data point.



## Requirements

This feature requires API v2.1.0 or higher.

This feature can be applied to all visuals except image visuals.

## Usage

To use the `supportsKeyboardFocus` feature, add the following code to the `capabilities.json` file of your visual. This capability allows the visual to receive focus through keyboard navigation.

```
{  
  ...  
  "supportsKeyboardFocus": true  
  ...  
}
```

## Next steps

To learn more about accessibility features, see [Design Power BI reports for accessibility](#).

To try out Power BI development, see [Developing a Power BI visual](#).

# Use the `supportsMultiVisualSelection` feature

5/13/2020 • 2 minutes to read • [Edit Online](#)

The `supportsMultiVisualSelection` feature enables you to use selection in multiple visuals in a report.

## Example

In a report with more than one visual, select two values to have those values apply to other visuals. For instance, in [Retail Analysis sample](#), select **Fashions Direct** in one visual. Select ctrl and select **Jan** in another visual. In the report, your selections apply to the other visuals that support this feature usage. Other visuals now scope to **Fashions Direct** and **Jan**.

## Requirements

This feature requires API v3.2.0 or higher.

You can't apply this feature to image visuals. You can't apply it to some advanced visuals such as key driver, decomposition tree, Q&A visuals, textbox, and gauge charts.

## Usage

To use the `supportsMultiVisualSelection` feature, add the following code to the `capabilities.json` file of your visual.

```
{  
    ...  
    "supportsMultiVisualSelection": true  
    ...  
}
```

## Next steps

To learn about Power BI concepts, see [Visuals in Power BI](#).

To try out Power BI development, see [Developing a Power BI visual](#).

# Get a Power BI visual certified

5/1/2020 • 6 minutes to read • [Edit Online](#)

Certified Power BI visuals are Power BI visuals in [AppSource](#) that meet the Microsoft Power BI team [code requirements](#). These visuals are tested to verify that they don't access external services or resources, and that they follow secure coding patterns and guidelines.

Once a Power BI visual is certified, it offers more features. For example, you can [export to PowerPoint](#), or display the visual in received emails, when a user [subscribes to report pages](#).

The certification process is optional. Power BI visuals that are not certified, are not necessarily unsafe Power BI visuals. Some Power BI visuals aren't certified because they don't comply with one or more of the [certification requirements](#). For example, a map Power BI visual connecting to an external service, or a Power BI visual using commercial libraries.

## NOTE

Microsoft is not the author of third-party Power BI visuals. To verify the functionality of third-party visuals, contact the author of the visual directly.

## Certification requirements

To get your Power BI visual [certified](#), your Power BI visual must comply with the requirements listed in this section.

### General requirements

Your Power BI visual has to be approved by Partner Center. We recommend that your Power BI visual is already in [AppSource](#). To learn how to publish a Power BI visual to AppSource, see [Publish Power BI visuals to Partner Center](#).

Before submitting your Power BI visual to be certified, verify that it complies with the [guidelines for Power BI visuals](#).

When submitting the Power BI visual, make sure that the compiled package exactly matches the submitted package.

### Code repository requirements

Although you don't have to publicly share your code in GitHub, the code repository has to be available for a review by the Power BI team. The best way to do this, is by providing the source code (JavaScript or TypeScript) in GitHub.

The repository must contain the following:

- Code for only one Power BI visual. It can't contain code for multiple Power BI visuals, or unrelated code.
- A branch named **certification** (lowercase required). The source code in this branch has to match the submitted package. This code can only be updated during the next submission process, if you're resubmitting your Power BI visual.

If your Power BI visual uses private npm packages, or git submodules, you must provide access to the additional repositories containing this code.

To understand how a Power BI visual repository looks, review the GitHub repository for the [Power BI visuals sample bar chart](#).

### File requirements

Use the latest version of the API to write the Power BI visual.

The repository must include the following files:

- **.gitignore** - Add `node_modules`, `.tmp` and `dist` to this file. The code cannot include the `node_modules`, `.tmp` or `dist` folders.
- **capabilities.json** - If you are submitting newer version of your Power BI visual with changes to the properties in this file, verify that they do not break reports for existing users.
- **pbviz.json**
- **package.json**. The visual must have the following package installed:
  - `"tslint"` - Version 5.18.0 or higher
  - `"typescript"` - Version 3.0.0 or higher
  - `"tslint-microsoftcontrib"` - Version 6.2.0 or higher
  - The file must contain a command for running linter - `"lint": "tslint -c tslint.json -p tsconfig.json"`
- **package-lock.json**
- **tsconfig.json**

## Command requirements

Make sure that the following commands don't return any errors.

- `npm install`
- `pbviz package`
- `npm audit` - Must not return any warnings with high or moderate level.
- `TSlint from Microsoft` with [the required configuration](#). This command must not return any lint errors.

## Compiling requirements

Use the latest version of [powerbi-visuals-tools](#) to write the Power BI visual.

You must compile your Power BI visual with `pbviz package`. If you're using your own build scripts, provide a `npm run package` custom build command.

## Source code requirements

Verify that you follow the [Power BI visuals additional certification](#) policy list. If your submission doesn't follow these guidelines, the rejection email from Partner Center will include the policy numbers listed in this link.

Follow the code requirements listed below to make sure that your code is in line with the Power BI certification policies.

### Required

- Only use public reviewable OSS components such as public JavaScript or TypeScript libraries.
- The code must support the [Rendering Events API](#).
- Ensure DOM is manipulated safely. Use sanitization for user input or user data, before adding it to DOM.
- Use the [sample report](#) as a test dataset.

### Not allowed

- Accessing external services or resources. For example, no HTTP/S or WebSocket requests can go out of Power BI to any services.
- Using `innerHTML`, or `D3.html(user data or user input)`.
- JavaScript errors or exceptions in the browser console, for any input data.
- Arbitrary or dynamic code such as `eval()`, unsafe use of `settimeout()`, `requestAnimationFrame()`, `setinterval(user input function)`, and user input or user data.
- Minified JavaScript files or projects.

# Submitting a Power BI visual for certification

You can request to have your Power BI visual certified by the Power BI team via Partner Center.

## TIP

The Power BI certification process might take time. If you're creating a new Power BI visual, we recommend that you publish your Power BI visual via the Partner Center before you request Power BI certification. This ensures that the publishing of your visual is not delayed.

To request Power BI certification:

1. Sign in to Partner Center.
2. On the **Overview** page, choose your Power BI visual, and go to the **Product** setup page.
3. Select the **Request Power BI certification** check box.
4. On the **Review and publish** page, in the **Notes for certification** text box, provide a link to the source code and the credentials required to access it.

## Private repository submission process

If you're using a private repository such as GitHub to submit your Power BI visual for certification, follow the instructions in this section.

1. Create a new account for the validation team.
2. Configure [two-factor authentication](#) for your account.
3. [Generate a new set of recovery codes](#).
4. When submitting your Power BI visual, provide the following:
  - A link to the repository
  - Login credentials (including a password)
  - Recovery codes
  - Read-only permissions to our account ([pbicvsupport](#))

# Certified Power BI visual badges

Once a Power BI visual is certified, it gets a designated badge that indicates that it's certified.

## Certified Power BI visuals in AppSource

- When searching online for [Power BI visuals in AppSource](#), a small yellow badge on the visual's card indicates that it's a certified Power BI visual.



- After clicking the Power BI visual card in AppSource, a yellow badge titled *PBI Certified* indicates that this Power BI visual is certified.



## Certified Power BI visuals in the Power BI interface

- When importing a Power BI visual from within Power BI (Desktop or service), a blue badge indicates that the Power BI visual is certified.



- You can display only certified Power BI visuals, by selecting the *Power BI Certified* filter option.

## Publication timeline

Deploying to AppSource is a process that may take some time. Your Power BI visual will be available to download from AppSource when this process is complete.

### When will users be able to download my visual?

- If you submitted a Power BI visual for the first time, users will be able to download it a few hours after you receive an email from AppSource.
- If you submitted an update to an existing Power BI visual, users will be able to download it within a month of your submission.

#### NOTE

The *version* field in AppSource will be updated with the day your Power BI was approved by AppSource, approximately a week after you submitted your visual. Users will be able to download the updated visual but the updated capabilities will not take effect. Your visual's new capabilities will affect the user's reports after about a month.

### When will my Power BI visual display a certification badge?

- If you submitted a Power BI visual for the first time, the certification badge will appear within a day of receiving the approval email from AppSource.
- If you're requesting certification for an existing Power BI visual, the certification badge will be visible within a month of your submission.

## Next steps

- If you're a web developer interested in creating your own Power BI visuals and adding them to the [Microsoft AppSource](#), start with the [Developing a Power BI visual](#) tutorial.
- For more information about visuals, see [Frequently asked questions about certified visuals](#).
- [Developing a Power BI visual](#)
- [Microsoft's Power BI visual playlist on YouTube](#)
- [Visuals in Power BI](#)
- [Publish Power BI visuals to Microsoft AppSource](#)
- More questions? [Try the Power BI Community](#)

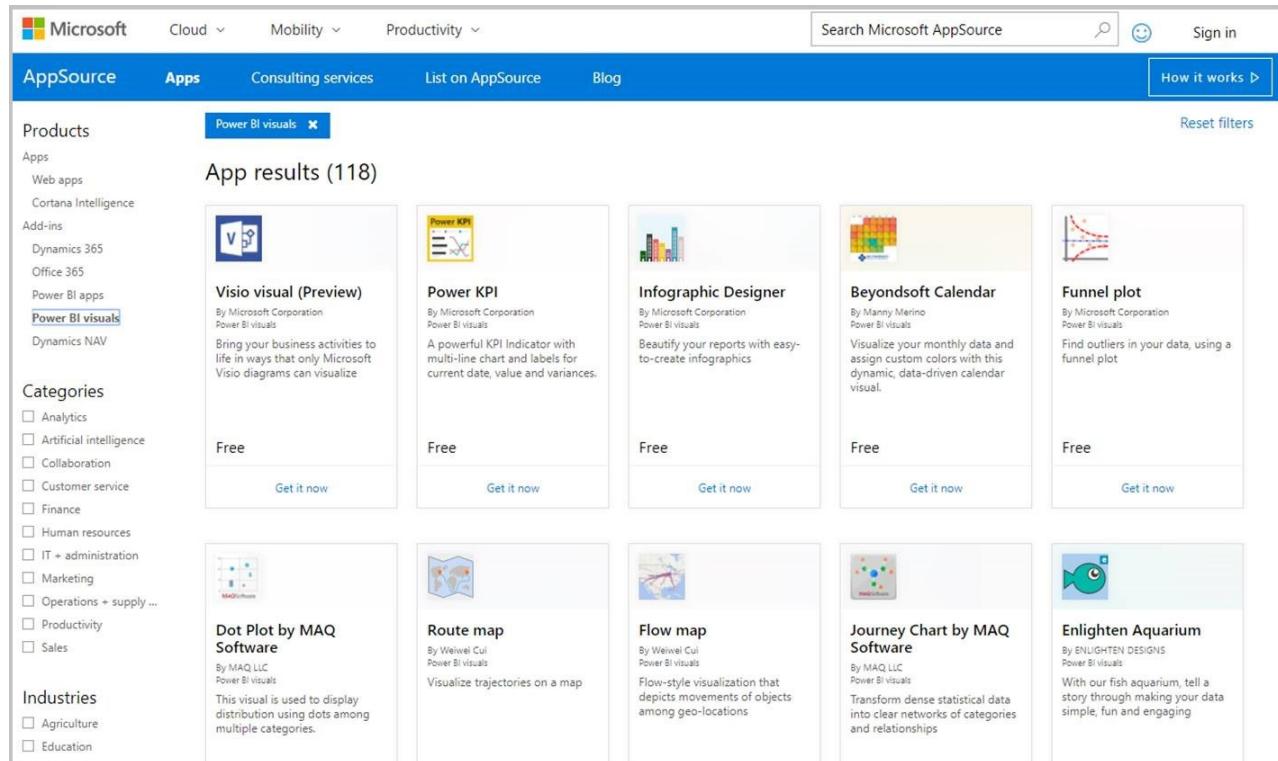
# Publish Power BI visuals to Partner Center

5/11/2020 • 5 minutes to read • [Edit Online](#)

Once you have created your Power BI visual, you may want to publish it to the AppSource for others to discover and use. For more information about creating a Power BI visual, see [Developing a Power BI visual](#).

## What is AppSource?

[AppSource](#) is the place to find SaaS apps and add-ins for your Microsoft products and services.



The screenshot shows the Microsoft AppSource homepage with a search bar at the top. The search term "Power BI visuals" is entered. Below the search bar, there are navigation links for "AppSource", "Apps", "Consulting services", "List on AppSource", and "Blog". A "How it works" button is also visible. On the left, there's a sidebar with categories like "Products", "Apps", "Add-ins", "Categories", and "Industries". The main content area displays "App results (118)" for "Power BI visuals". There are ten cards, each representing a different Power BI visual. Each card includes a thumbnail, the name of the visual, the developer, a brief description, and a "Get it now" button. The cards are:

- Visio visual (Preview)** by Microsoft Corporation: Bring your business activities to life in ways that only Microsoft Visio diagrams can visualize. Free. Get it now.
- Power KPI** by Microsoft Corporation: A powerful KPI Indicator with multi-line chart and labels for current date, value and variances. Free. Get it now.
- Infographic Designer** by Microsoft Corporation: Beautify your reports with easy-to-create infographics. Free. Get it now.
- Beyondsoft Calendar** by Manay Merino: Visualize your monthly data and assign custom colors with this dynamic, data-driven calendar visual. Free. Get it now.
- Funnel plot** by Microsoft Corporation: Find outliers in your data, using a funnel plot. Free. Get it now.
- Dot Plot by MAQ Software** by MAQ LLC: This visual is used to display distribution using dots among multiple categories. Free. Get it now.
- Route map** by Weivel Cul: Visualize trajectories on a map. Free. Get it now.
- Flow map** by Weivel Cul: Flow-style visualization that depicts movements of objects among geo-locations. Free. Get it now.
- Journey Chart by MAQ Software** by MAQ LLC: Transform dense statistical data into clear networks of categories and relationships. Free. Get it now.
- Enlighten Aquarium** by ENLIGHTEN DESIGNS: With our fish aquarium, tell a story through making your data simple, fun and engaging. Free. Get it now.

## Preparing to submit your Power BI visual

Before submitting a Power BI visual to AppSource, make sure you've read the [Power BI visuals guidelines](#) and [tested your custom visual](#).

When you are ready to submit your Power BI visual, verify that your visual meets all the requirements listed below.

ITEM	REQUIRED	DESCRIPTION
Pbviz package	Yes	Pack your Power BI visual into a Pbviz package containing all the required metadata. Visual name Display name GUID Version Description Author name and email

ITEM	REQUIRED	DESCRIPTION
Sample .pbix report file	Yes	To showcase your visual, you should help users to get familiar with the visual. Highlight the value that the visual brings to the user and give examples of usage and formatting options. You can also add a " <i>hints</i> " page at the end with some tips and tricks and things to avoid. The sample .pbix report file must work offline, without any external connections.
Icon	Yes	You should include the custom visual logo that will appear in the store front. The format can be .png, .jpg, jpeg or .gif. It must be exactly 300 px (width) x 300 px (height). <b>Important!</b> Please review the <a href="#">AppSource store images guide</a> carefully, before submitting the icon.
Screenshots	Yes	Provide at least one screenshot. The format can be .png, .jpg, jpeg or .gif. The dimensions must be exactly 1366 px (width) by 768 px (height). The size of the file can't be larger than 1024 kb. For greater usage, add text bubbles to articulate the value proposition of key features shown in each screenshot.
Support download link	Yes	Provide a support URL for your customers. This link is entered as part of your Partner Center listing, and is visible to users when they access your visual's listing on AppSource. The format of your URL should include https:// or https://.
Privacy document link	Yes	Provide a link to the visual's privacy policy. This link is entered as part of your Partner Center listing, and is visible to users when they access your visual's listing on AppSource. The format of your link should include https:// or https://.
End-user license agreement (EULA)	Yes	You must provide an EULA file for your Power BI visual. You can use the <a href="#">standard contract</a> , <a href="#">Power BI visuals contract</a> , or your own EULA.
Video link	No	To increase the interest of users for your custom visual, provide a link to a video about your visual. The format of your URL should include https:// or https://.

ITEM	REQUIRED	DESCRIPTION
GitHub repository	No	Share a public link to a <a href="#">GitHub</a> repository with sources of your Power BI visual and sample data. This allows other developers an opportunity to provide feedback and propose improvements to your code.

## Getting an app package XML

To submit a Power BI visual you need an app package XML from the Power BI team. To get the app package XML, send an email to the Power BI visuals submission team ([pbivizsubmit@microsoft.com](mailto:pbivizsubmit@microsoft.com)).

Before you create the **pbiviz** package, you must fill the following fields in the **pbiviz.json** file:

- description
- supportUrl
- author
- name
- email

Attach the **pbiviz** file and the **sample report pbix** file to your email. The Power BI team will reply back with instructions and an app package XML file to upload. This XML app package is required in order to submit your visual through the Office developer center.

### NOTE

To improve quality and assure that existing reports are not breaking, updates to existing visuals will take an additional two weeks to reach production environment after approval in the store.

## Submitting to AppSource

To submit your Power BI visual to AppSource, you need to get an app package from the Power BI team, and then submit it to Partner Center.

### Getting the app package

You must send an email with the **pbiviz** file and the **pbix** file to the Power BI team before submitting to AppSource. This allows the Power BI team to upload the files to the public share server. Otherwise, the store will not be able to retrieve the files.

The Power BI team has to check files for new Power BI visual submissions, updates to existing Power BI visuals, and fixes to rejected submissions.

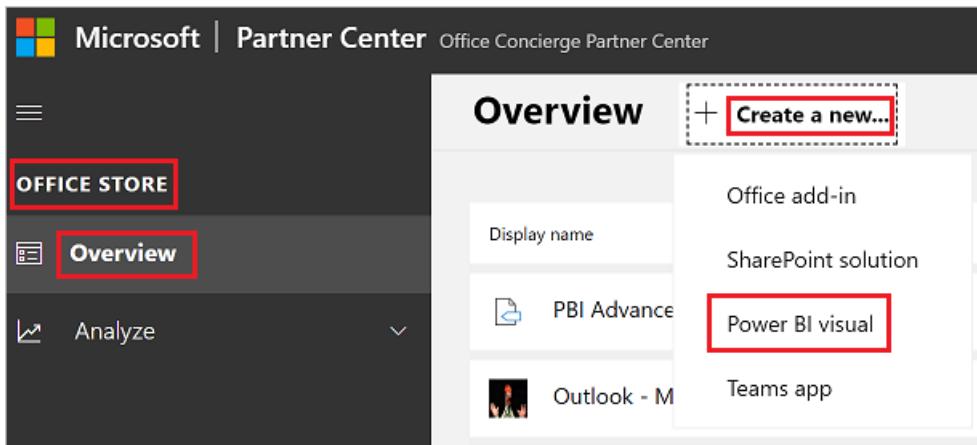
### Submitting to Partner Center

To submit your Power BI visual to Partner Center, you have to be registered with Partner Center. If you're not yet registered, [Open a developer account in Partner Center](#).

Follow the steps below to submit your Power BI visual to Partner Center. For more information about the submission process, see [Submit your Office solution to AppSource via Partner Center](#).

1. Log into Partner Center.
2. On the left pane, select OFFICE STORE.
3. Select Overview.

4. Select **Create a new** and from the drop-down menu, select **Power BI visual**.



5. In the **Create a new Power BI visual** window, enter a name for your Power BI visual and select **Create**.
6. Select **Packages** and upload your Power BI visual XML app package.
7. Select **Properties** and provide the required information.
8. If your product requires additional purchase, select **Product setup** and check the **Associated service purchase** check box.
9. (Optional) If you want to **certify** your visual, select **Product setup** and check the **Power BI certification** check box.

**TIP**

The Power BI certification process might take time. If you're creating a new Power BI visual, we recommend that you publish your Power BI visual via the Partner Center before you request Power BI certification. This ensures that the publishing of your visual is not delayed.

10. Select **Product setup** and click **Review and publish**.

## Tracking submission status and usage

You can review the [validation policies](#).

- After submission, you will be able to view the submission status in the [app dashboard](#).
- To understand when your Power BI visual will be available to download from AppSource, review the Power BI visuals [publication timeline](#).

## Certify your visual

Once your visual is created, if you want you can get your visual [certified](#).

## Next steps

- [Developing a Power BI custom visual](#)
- [Visualizations in Power BI](#)
- [Visuals in Power BI](#)
- [Getting a Power BI visual certified](#)

- More questions? [Try asking the Power BI Community](#)

# Tutorial: Create an R-powered Power BI visual

5/28/2020 • 4 minutes to read • [Edit Online](#)

This tutorial describes how to create an R-powered visual for Power BI.

In this tutorial, you learn how to:

- Create an R-powered visual
- Edit the R script in Power BI Desktop
- Add libraries to the visual
- Add a static property

## Prerequisites

- A Power BI Pro account. [Sign up for a free trial](#) before you begin.
- The R engine. You can download it free from many locations, including the [Revolution Open download page](#) and the [CRAN Repository](#). For more information, see [Create Power BI visuals using R](#).
- [Power BI Desktop](#).
- [Windows PowerShell](#) version 4 or later for Windows users OR the [Terminal](#) for OSX users.

## Getting started

1. Prepare sample data for the visual. You can save these values to an Excel database or .csv file and import it into Power BI Desktop.

MONTHNO	TOTAL UNITS
1	2303
2	2319
3	1732
4	1615
5	1427
6	2253
7	1147
8	1515
9	2516
10	3131
11	3170

MONTHNO	TOTAL UNITS
12	2762

2. To create a visual, open PowerShell or Terminal, and run the following command:

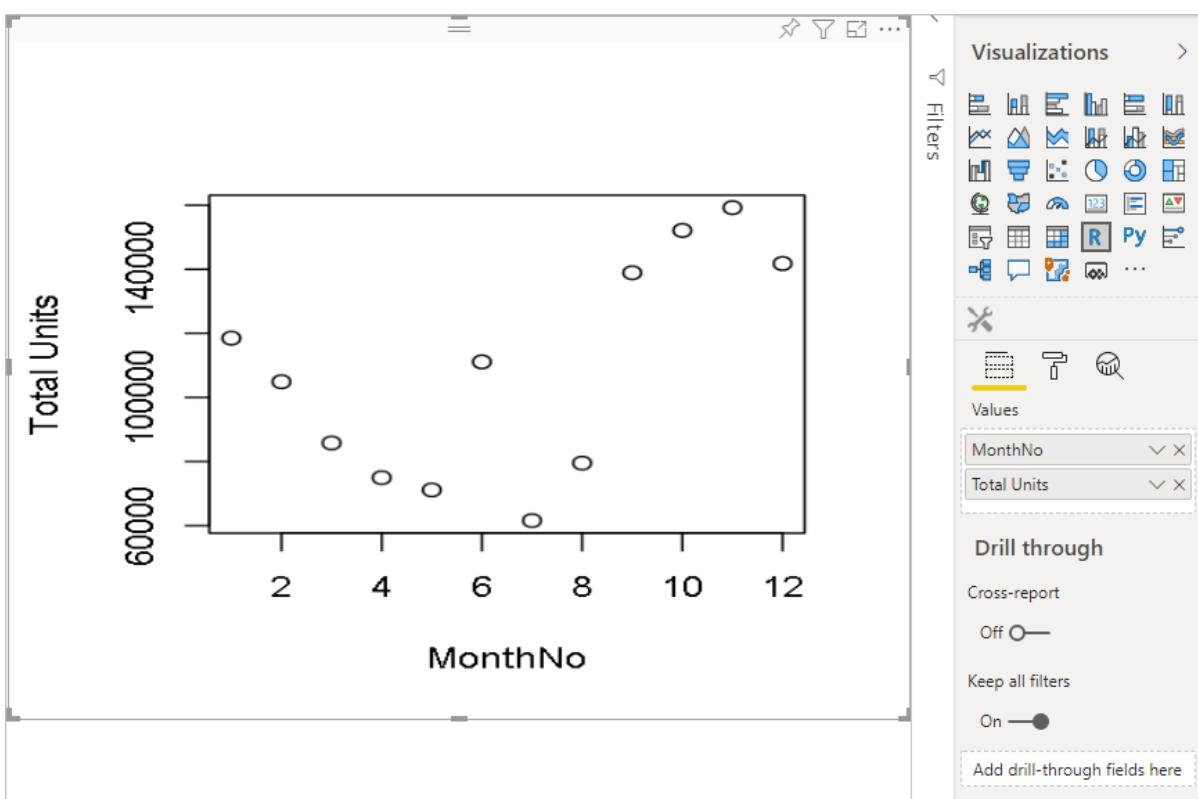
```
pbviz new rVisualSample -t rvisual
```

This command creates a new folder structure based on the `rvisual1` template. This template includes a basic, ready-to-run R-powered visual that runs the following R script:

```
plot(Values)
```

The `Values` data frame will contain columns in `Values` data role.

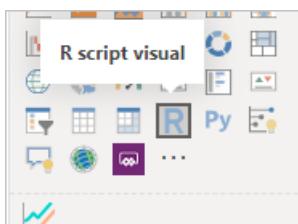
3. Assign data to the developer visual by adding `MonthNo` and `Total units` to `Values` for the visual.



## Editing the R Script

When you use `pbviz` to create the R-powered visual based on the `rvisual1` template, it creates a file called `script.r` in the root folder of the visual. This file holds the R script that runs to generate the image for a user. You can create your R script in Power BI Desktop.

1. In Power BI Desktop, select **R script visual**:



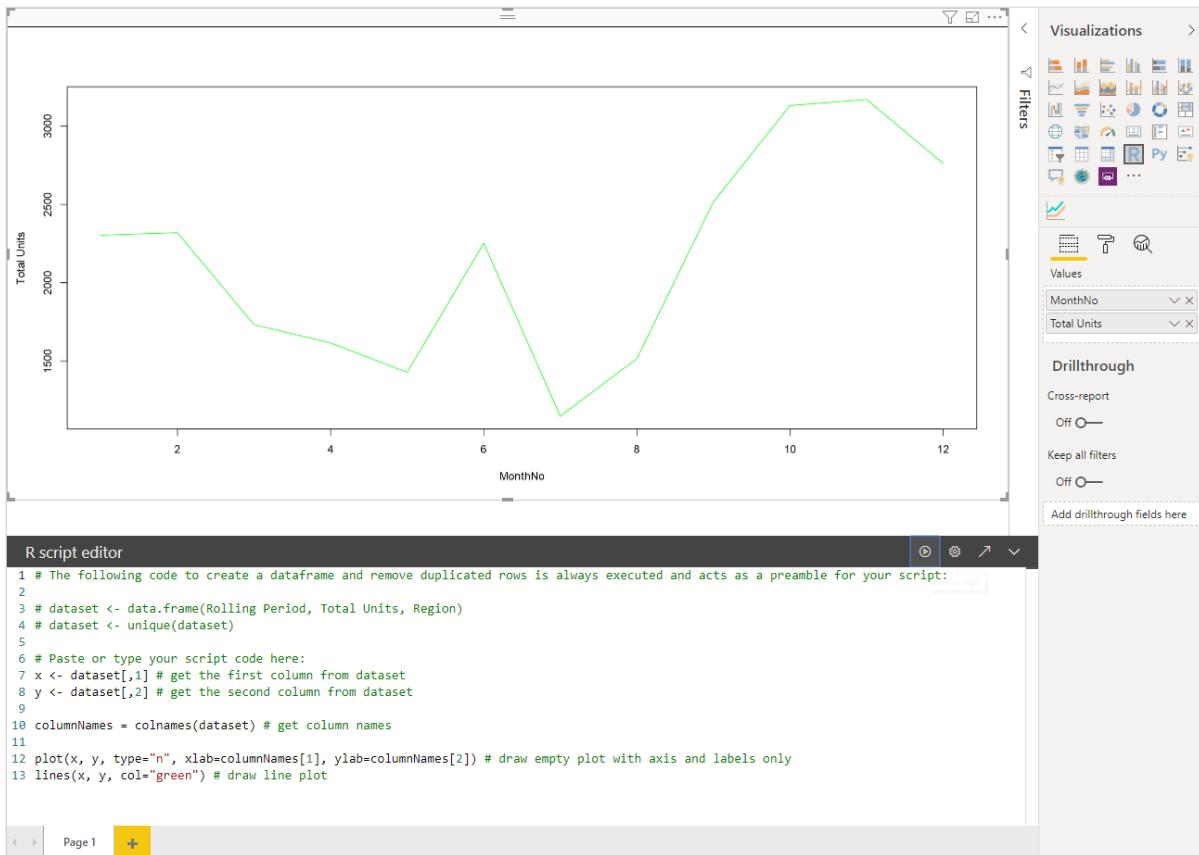
2. Paste this R code into the R script editor:

```
x <- dataset[,1] # get the first column from dataset
y <- dataset[,2] # get the second column from dataset

columnNames = colnames(dataset) # get column names

plot(x, y, type="n", xlab=columnNames[1], ylab=columnNames[2]) # draw empty plot with axis and labels only
lines(x, y, col="green") # draw line plot
```

3. Select the Run script icon to see the result.



4. When your R script is ready, copy it to the `script.r` file in your visual project created at one of the previous steps.

5. Change the `name` of `dataRoles` in `capabilities.json` to `dataRoles`. Power BI passes data as the `dataset` data frame object for the R script visual, but the R visual gets the data frame name according to `dataRoles` names.

```
{
  "dataRoles": [
    {
      "displayName": "Values",
      "kind": "GroupingOrMeasure",
      "name": "dataRoles"
    }
  ],
  "dataViewMappings": [
    {
      "scriptResult": {
        "dataInput": {
          "table": {
            "rows": {
              "select": [
                {
                  "for": {
                    "in": "dataset"
                  }
                }
              ],
              "dataReductionAlgorithm": {
                "top": {}
              }
            }
          }
        }
      }
    },
    ...
  ]
}
```

- Add the following code to support resizing the image in the `src/visual.ts` file.

```
public onResizing(finalViewport: IViewport): void {
  this.imageDiv.style.height = finalViewport.height + "px";
  this.imageDiv.style.width = finalViewport.width + "px";
  this.imageElement.style.height = finalViewport.height + "px";
  this.imageElement.style.width = finalViewport.width + "px";
}
```

## Add libraries to visual package

This procedure allows your visual to use the `corrplot` package.

- Add the library dependency for your visual to `dependencies.json`. Here is an example of the file content:

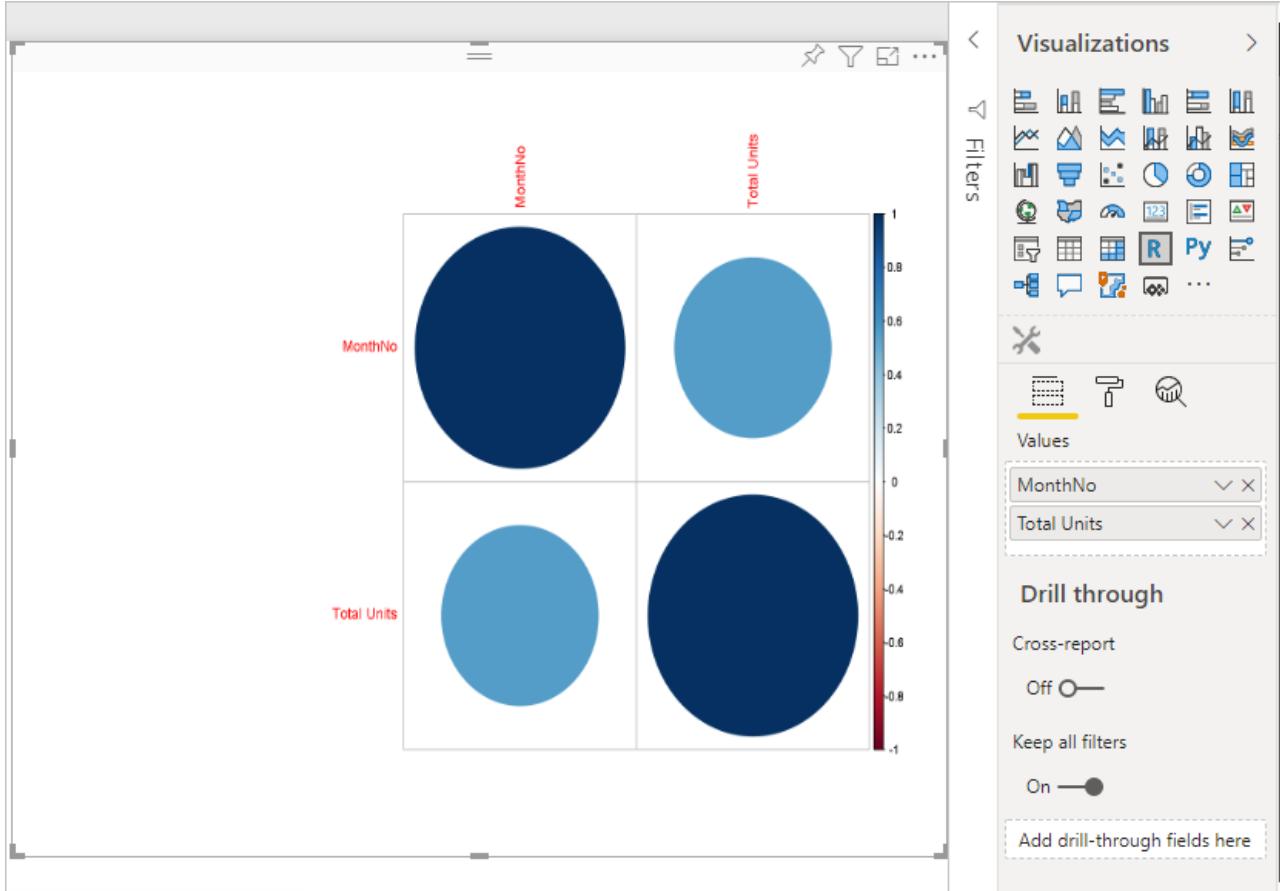
```
{
  "cranPackages": [
    {
      "name": "corrplot",
      "displayName": "corrplot",
      "url": "https://cran.r-project.org/web/packages/corrplot/"
    }
  ]
}
```

The `corrplot` package is a graphical display of a correlation matrix. For more information about `corrplot`, see [An Introduction to corrplot Package](#).

2. After you make these changes, start using the package in your `script.r` file.

```
library(corrplot)
corr <- cor(dataset)
corrplot(corr, method="circle", order = "hclust")
```

The result of using `corrplot` package looks like this example:



## Adding a static property to the property pane

Enable users to change UI settings. To do this, add properties to the property pane that change the user-input based behavior of the R script.

You can configure `corrplot` by using the `method` argument for the `corrplot` function. The default script uses a circle. Modify your visual to let the user choose between several options.

1. Define the object and property in the `capabilities.json` file. Then use this object name in the enumeration `method` to get those values from the property pane.

```
{
  "settings": {
    "displayName": "Visual Settings",
    "description": "Settings to control the look and feel of the visual",
    "properties": {
      "method": {
        "displayName": "Data Look",
        "description": "Control the look and feel of the data points in the visual",
        "type": {
          "enumeration": [
            {
              "displayName": "Circle",
              "value": "circle"
            },
            {
              "displayName": "Square",
              "value": "square"
            },
            {
              "displayName": "Ellipse",
              "value": "ellipse"
            },
            {
              "displayName": "Number",
              "value": "number"
            },
            {
              "displayName": "Shade",
              "value": "shade"
            },
            {
              "displayName": "Color",
              "value": "color"
            },
            {
              "displayName": "Pie",
              "value": "pie"
            }
          ]
        }
      }
    }
  }
}
```

2. Open the `src/settings.ts` file. Create a `CorrPlotSettings` class with the public property `method`. The type is `string` and the default value is `circle`. Add the `settings` property to the `VisualSettings` class with the default value:

```

"use strict";

import { DataViewObjectsParser } from "powerbi-visuals-utils-dataviewutils";
import DataViewObjectsParser = DataViewObjectsParser.DataViewObjectsParser;

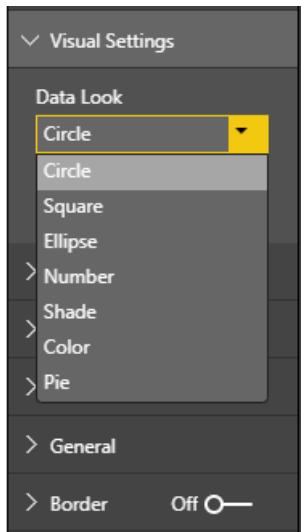
export class VisualSettings extends DataViewObjectsParser {
    public rcv_script: rcv_scriptSettings = new rcv_scriptSettings();
    public settings: CorrPlotSettings = new CorrPlotSettings();
}

export class CorrPlotSettings {
    public method: string = "circle";
}

export class rcv_scriptSettings {
    public provider;
    public source;
}

```

After these steps, you can change the property of the visual.



Finally, the R script must start with a property. If the user doesn't change the property, the visual doesn't get any value for this property.

For R runtime variables for the properties, the naming convention is `<objectname>_<propertyname>`, in this case, `settings_method`.

3. Change the R script in your visual to match the following code:

```

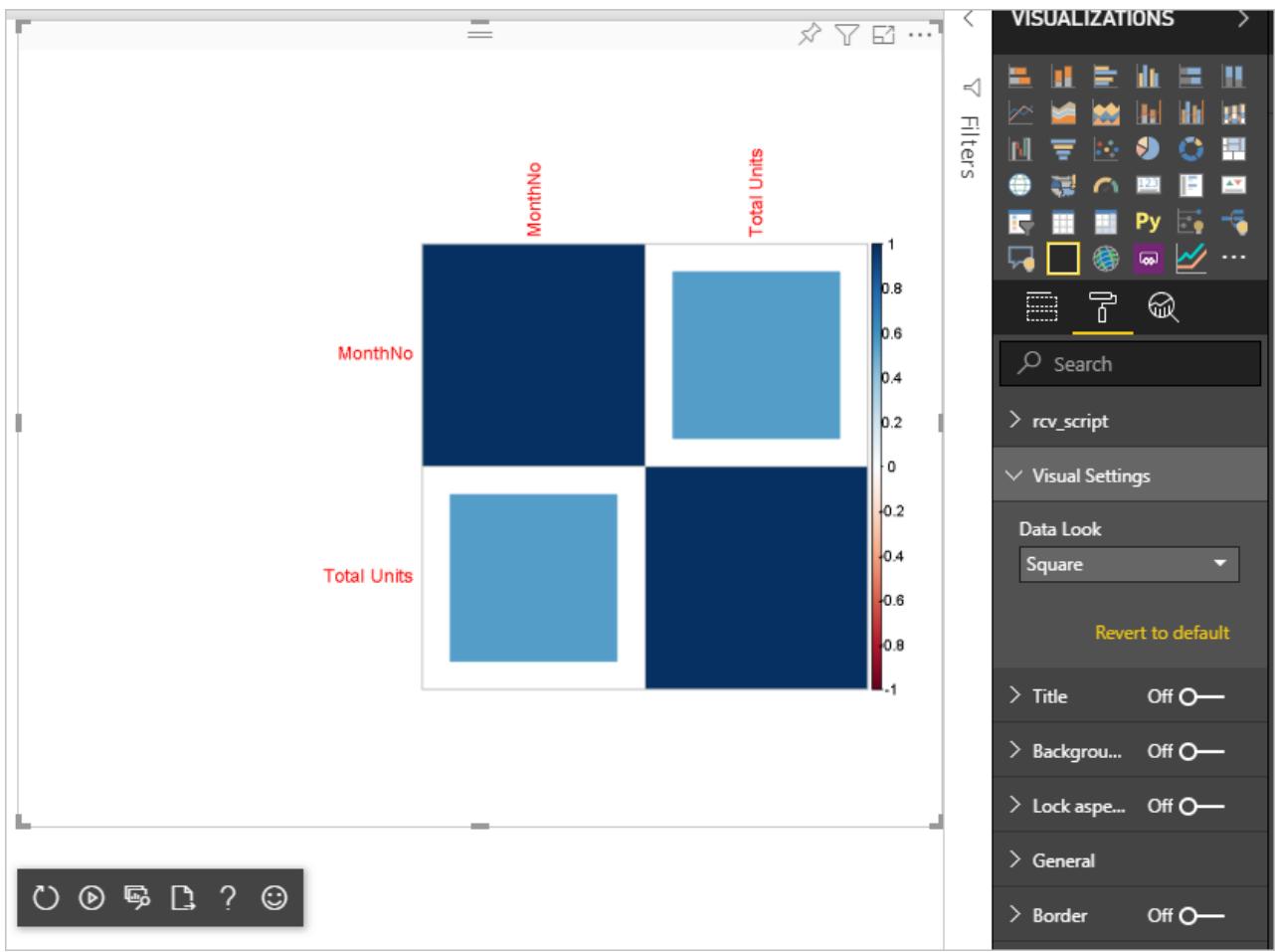
library(corrplot)
corr <- cor(dataset)

if (!exists("settings_method"))
{
    settings_method = "circle";
}

corrplot(corr, method=settings_method, order = "hclust")

```

Your final visual looks like the following example:



## Next steps

To learn more about R-powered visuals, see [Use R-powered Power BI visuals in Power BI](#).

For more information about R-powered visuals in Power BI Desktop, see [Create Power BI visuals using R](#).

# Submission testing of a Power BI visual

5/11/2020 • 5 minutes to read • [Edit Online](#)

Before you publish your visual to [AppSource](#), it must pass these test cases. Test your visual before you submit it. If your visual doesn't pass required test cases, it will be rejected.

For more information about the publishing process, see [Publish Power BI visuals to Partner Center](#).

## General test cases

TEST CASE	EXPECTED RESULTS
Create a <b>Stacked column chart</b> with <b>Category</b> and <b>Value</b> . Convert it to your visual and then back to column chart.	No error appears after these conversions.
Create a <b>Gauge</b> with three measures. Convert it to your visual and then back to <b>Gauge</b> .	No error appears after these conversions.
Make selections in your visual.	Other visuals reflect the selections.
Select elements in other visuals.	Your visual shows filtered data according to selection in other visuals.
Check min/max <b>dataViewMapping</b> conditions.	Field buckets can accept multiple fields, a single field, or are determined by other buckets. The min/max <b>dataViewMapping</b> conditions must be correctly set up in the capabilities of your visual.
Remove all fields in different orders.	Visual cleans up properly as fields are removed in arbitrary order. There are no errors in the console or the browser.
Open the <b>Format</b> pane with each possible bucket configuration.	This test doesn't trigger null reference exceptions.
Filter data using the <b>Filter</b> pane at the visual, page, and report level.	Tooltips are correct after applying filters. Tooltips show the filtered value.
Filter data using a <b>Slicer</b> .	Tooltips are correct after applying filters. Tooltips show the filtered value.
Filter data using a published visual. For instance, select a pie slice or a column.	Tooltips are correct after applying filters. Tooltips show the filtered value.
If cross-filtering is supported, verify that filters work correctly.	Applied selection filters other visuals on this page of the report.
Select with Ctrl, Alt, and Shift keys.	No unexpected behaviors appear.
Change the <b>View Mode</b> to <b>Actual size</b> , <b>Fit to page</b> , and <b>Fit to width</b> .	Mouse coordinates are accurate.

TEST CASE	EXPECTED RESULTS
Resize your visual.	Visual reacts correctly to resizing.
Set the report size to the minimum.	There are no display errors.
Ensure scroll bars work correctly.	Scroll bars should exist, if required. Check scroll bar sizes. Scroll bars shouldn't be too wide or tall. Position and size of scroll bars must be in accord with other elements of your visual. Verify that scroll bars are needed for different sizes of the visual.
Pin your visual to a <b>Dashboard</b> .	The visual should be displayed properly.
Add multiple versions of your visual to a single report page.	All versions of the visual be displayed and operate properly.
Add multiple versions of your visual to multiple report pages.	All versions of the visual be displayed and operate properly.
Switch between report pages.	The visual displays correctly.
Test Reading view and Edit view for your visual.	All functions work correctly.
If your visual uses animations, add, change, and delete elements of your visual.	Animation of visual elements works correctly.
Open the <b>Property</b> pane. Turn properties on and off, enter custom text, stress the options available, and input bad data.	The visual responds correctly.
Save the report and reopen it.	All properties settings persist.
Switch pages in the report and then switch back.	All properties settings persist.
Test all functionality of your visual, including different options that the visual provides.	All displays and features work correctly.
Test all numeric, date, and character data types, as in the following tests.	All data is formatted properly.
Review formatting of tooltip values, axis labels, data labels, and other visual elements with formatting.	All elements are formatted correctly.
Verify that data labels use the format string.	All data labels are formatted correctly.
Switch automatic formatting on and off for numeric values in Tooltips.	Tooltips display values correctly.
Test data entries with different types of data, including numeric, text, date-time, and different format strings from the model. Test different data volumes, such as thousands of rows, one row, and two rows.	All displays and features work correctly.
Provide bad data to your visual, such as null, infinity, negative values, and wrong value types.	All displays and features work correctly.

## Optional browser testing

The AppSource team validates visual on the most current Windows versions of Google Chrome, Microsoft Edge, and Mozilla Firefox browsers. Optionally, test your visual in the following browsers.

TEST CASE	EXPECTED RESULTS
<b>Windows</b>	
Google Chrome (previous version)	All displays and features work correctly.
Mozilla Firefox (previous version)	All displays and features work correctly.
Microsoft Edge (previous version)	All displays and features work correctly.
Microsoft Internet Explorer 11 (optional)	All displays and features work correctly.
<b>macOS</b>	
Chrome (previous version)	All displays and features work correctly.
Firefox (previous version)	All displays and features work correctly.
Safari (previous version)	All displays and features work correctly.
<b>Linux</b>	
Firefox (latest and previous versions)	All displays and features work correctly.
<b>Mobile iOS</b>	
Apple Safari iPad (previous Safari version)	All displays and features work correctly.
Chrome iPad (latest Safari version)	All displays and features work correctly.
<b>Mobile Android</b>	
Chrome (latest and previous versions)	All displays and features work correctly.

## Desktop testing

Test your visual in the current version of [Power BI Desktop](#).

TEST CASE	EXPECTED RESULTS
Test all features of your visual.	All displays and features work correctly.
Import, save, open a file, and publish to the Power BI web service by using the <b>Publish</b> button in Power BI Desktop.	All displays and features work correctly.
Change the numeric format string to have zero decimal places or three decimal places by increasing or decreasing the precision.	The visual displays correctly.

## Performance testing

Your visual should perform at an acceptable level. Use developer tools to validate performance. Don't rely on visual cues and the console time logs.

TEST CASE	EXPECTED RESULTS
Create a visual with many visual elements.	The visual should perform well and not freeze the application. There should be no performance issues with elements such as animation speed, resizing, filtering, and selecting.

## Next steps

For more information about the publishing process, see [Publish Power BI visuals to Partner Center](#).

More questions? [Ask the Power BI Community](#).

# Visual interactions in Power BI visuals

3/19/2020 • 2 minutes to read • [Edit Online](#)

Visuals can query the value of the `allowInteractions` flag, which indicates whether the visual should allow visual interactions. For example, visuals are interactive during report viewing or editing, but not interactive when they're viewed in a dashboard. These interactions are *click, pan, zoom, selection*, and others.

## NOTE

You should enable tooltips in all scenarios, regardless of which flag is indicated.

The `allowInteractions` flag is passed as a Boolean during the initialization of the visual, as a member of the `IVisualHost` interface.

In any Power BI scenario that requires that the visual not be interactive (for example, dashboard tiles), the `allowInteractions` flag is set to `false`. Otherwise (for example, Report), `allowInteractions` is set to `true`.

For more information, see the [SampleBarChart visual repository](#).

```
...
let allowInteractions = options.host.allowInteractions;
bars.on('click', function(d) {
    if (allowInteractions) {
        selectionManager.select(d.selectionId);
        ...
    }
});
```

# Add interactivity into visual by Power BI visuals selections

3/19/2020 • 6 minutes to read • [Edit Online](#)

Power BI provides two ways of interaction between visuals - selection and filtering. The sample below demonstrates how to select any items in one visual and notify other visuals in the report about new selection state.

`Selection` object corresponds to the interface:

```
export interface ISelectionId {
    equals(other: ISelectionId): boolean;
    includes(other: ISelectionId, ignoreHighlight?: boolean): boolean;
    getKey(): string;
    getSelector(): Selector;
    getSelectorsByColumn(): SelectorsByColumn;
    hasIdentity(): boolean;
}
```

## How to use SelectionManager to select data points

The visual host object provides the method for creating an instance of selection manager. The selection manager responsible to select, to clear selection, to show the context menu, to store current selections and check selection state. And the selection manager has corresponded methods for those actions.

### Create an instance of the selection manager

For using the selection manager, you need to create the instance of a selection manager. Usually, visuals create a selection manager instance in the `constructor` of the visual object.

```
export class Visual implements IVisual {
    private target: HTMLElement;
    private host: IVisualHost;
    private selectionManager: ISelectionManager;
    // ...
    constructor(options: VisualConstructorOptions) {
        this.host = options.host;
        // ...
        this.selectionManager = this.host.createSelectionManager();
    }
    // ...
}
```

### Create an instance of the selection builder

When the selection manager instance is created, you need to create `selections` for each data point of the visual. The visual host object provides `createSelectionIdBuilder` method to generate selection for each data point. This method return instance of the object with interface `powerbi.visuals.ISelectionIdBuilder`:

```

export interface ISelectionIdBuilder {
    withCategory(categoryColumn: DataViewCategoryColumn, index: number): this;
    withSeries(seriesColumn: DataViewValueColumns, valueColumn: DataViewValueColumn | DataViewValueColumnGroup): this;
    withMeasure(measureId: string): this;
    withMatrixNode(matrixNode: DataViewMatrixNode, levels: DataViewHierarchyLevel[]): this;
    withTable(table: DataViewTable, rowIndex: number): this;
    createSelectionId(): ISelectionId;
}

```

This object has corresponded methods to create `selections` for different types of data view mappings.

#### **NOTE**

The methods `withTable` and `withMatrixNode` were introduced on API 2.5.0 of the Power BI visuals. If you need to use selections for table or matrix data view mappings you need to update API version to 2.5.0 or higher.

### Create selections for categorical data view mapping

Let's review how selections represent on categorical data view mapping for sample dataset:

MANUFACTURER	TYPE	VALUE
Chrysler	Domestic Car	28883
Chrysler	Domestic Truck	117131
Chrysler	Import Car	0
Chrysler	Import Truck	6362
Ford	Domestic Car	50032
Ford	Domestic Truck	122446
Ford	Import Car	0
Ford	Import Truck	0
GM	Domestic Car	65426
GM	Domestic Truck	138122
GM	Import Car	197
GM	Import Truck	0
Honda	Domestic Car	51450
Honda	Domestic Truck	46115
Honda	Import Car	2932
Honda	Import Truck	0

MANUFACTURER	TYPE	VALUE
Nissan	Domestic Car	51476
Nissan	Domestic Truck	47343
Nissan	Import Car	5485
Nissan	Import Truck	1430
Toyota	Domestic Car	55643
Toyota	Domestic Truck	61227
Toyota	Import Car	20799
Toyota	Import Truck	23614

And the visual uses the following data view mapping:

```
{
  "dataRoles": [
    {
      "displayName": "Columns",
      "name": "columns",
      "kind": "Grouping"
    },
    {
      "displayName": "Rows",
      "name": "rows",
      "kind": "Grouping"
    },
    {
      "displayName": "Values",
      "name": "values",
      "kind": "Measure"
    }
  ],
  "dataViewMappings": [
    {
      "categorical": {
        "categories": {
          "for": {
            "in": "columns"
          }
        },
        "values": {
          "group": {
            "by": "rows",
            "select": [
              {
                "for": {
                  "in": "values"
                }
              }
            ]
          }
        }
      }
    }
  ]
}
```

In the sample, `Manufacturer` is `columns` and `Type` is `rows`. There's series created by groupings values by `rows` (`Type`).

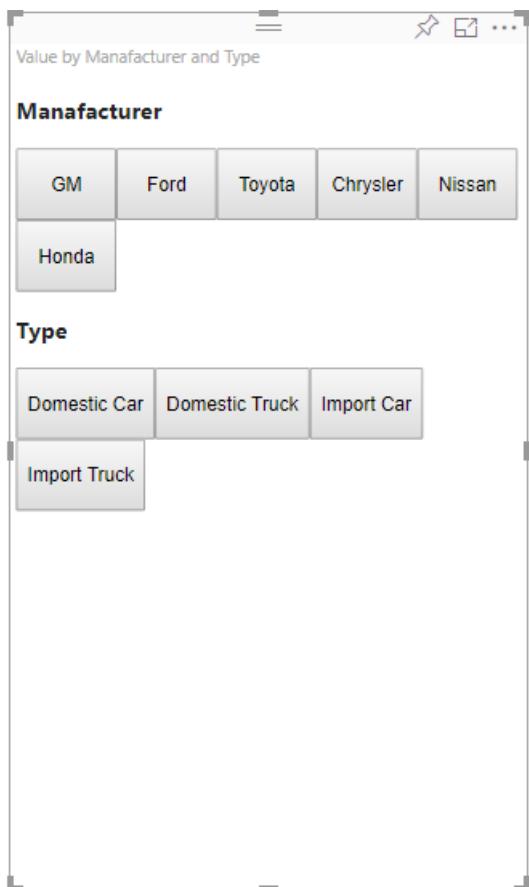
And visual should able to slice data by `Manufacturer` and `Type` too.

For example, when user selects `Chrysler` by `Manufacturer`, other visuals should show following data:

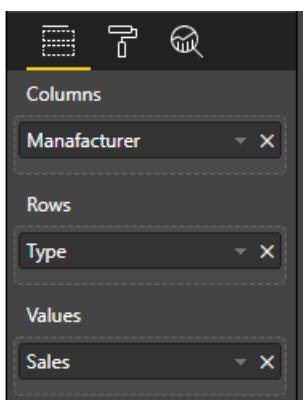
MANUFACTURER	TYPE	VALUE
Chrysler	Domestic Car	28883
Chrysler	Domestic Truck	117131
Chrysler	Import Car	0
Chrysler	Import Truck	6362

When user selects `Import Car` by `Type` (selects data by series), other visuals should show following data:

MANUFACTURER	TYPE	VALUE
Chrysler	Import Car	0
Ford	Import Car	0
GM	Import Car	197
Honda	Import Car	2932
Nissan	Import Car	5485
Toyota	Import Car	20799



Need to fill the visual data baskets.



There are **Manufacturer** as category (columns), **Type** as series (rows) and **Sales** as **values** for series.

## NOTE

The `values` are required for series because according to data view mapping the visual expects that `Values` will be groped by `Rows` data.

### Create selections for categories

```
// categories
const categories = dataView.categorical.categories;

// create label for 'Manufacturer' column
const p = document.createElement("p") as HTMLParagraphElement;
p.innerText = categories[0].source.displayName.toString();
this.target.appendChild(p);

// get count of category elements
const categoriesCount = categories[0].values.length;

// iterate all categories to generate selection and create button elements to use selections
for (let categoryIndex = 0; categoryIndex < categoriesCount; categoryIndex++) {
    const categoryValue: powerbi.PrimitiveValue = categories[0].values[categoryIndex];

    const categorySelectionId = this.host.createSelectionIdBuilder()
        .withCategory(categories[0], categoryIndex) // we have only one category (only one `Manufacturer` column)
        .createSelectionId();
    this.dataPoints.push({
        value: categoryValue,
        selection: categorySelectionId
    });
    console.log(categorySelectionId);

    // create button element to apply selection on click
    const button = document.createElement("button") as HTMLButtonElement;
    button.value = categoryValue.toString();
    button.innerText = categoryValue.toString();
    button.addEventListener("click", () => {
        // handle click event to apply correspond selection
        this.selectionManager.select(categorySelectionId);
    });
    this.target.appendChild(button);
}
```

In the sample code, you can see that we iterate all categories. And in each iteration, we call

`createSelectionIdBuilder` to create the next selection for each category by calling `withCategory` method of the selection builder. The method `createSelectionId` is used as a final method to return the generated `selection` object.

In `withCategory` method, we pass the column of `category`, in the sample, it's `Manufacturer` and index of category element.

### Create selections for series

```

// get grouped values for series
const series: powerbi.DataViewValueColumnGroup[] = dataView.categorical.values.grouped();

// create label for 'Type' column
const p2 = document.createElement("p") as HTMLParagraphElement;
p2.innerText = dataView.categorical.values.source.displayName;
this.target.appendChild(p2);

// iterate all series to generate selection and create button elements to use selections
series.forEach( (ser: powerbi.DataViewValueColumnGroup) => {
    // create selection id for series
    const seriesSelectionId = this.host.createSelectionIdBuilder()
        .withSeries(dataView.categorical.values, ser)
        .createSelectionId();

    this.dataPoints.push({
        value: ser.name,
        selection: seriesSelectionId
    });

    // create button element to apply selection on click
    const button = document.createElement("button") as HTMLButtonElement;
    button.value = ser.name.toString();
    button.innerText = ser.name.toString();
    button.addEventListener("click", () => {
        // handle click event to apply correspond selection
        this.selectionManager.select(seriesSelectionId);
    });
    this.target.appendChild(button);
});

```

## Create selections for table data view mapping

Sample of table data views mapping

```
{
  "dataRoles": [
    {
      "displayName": "Values",
      "name": "values",
      "kind": "GroupingOrMeasure"
    }
  ],
  "dataViewMappings": [
    {
      "table": {
        "rows": {
          "for": {
            "in": "values"
          }
        }
      }
    }
  ]
}
```

To create a selection for each row of table data view mapping, you need to call `withTable` method of selection builder.

```

public update(options: VisualUpdateOptions) {
    const dataView = options.dataViews[0];
    dataView.table.rows.forEach((row: DataViewTableRow, rowIndex: number) => {
        this.target.appendChild(rowDiv);
        const selection: ISelectionId = this.host.createSelectionIdBuilder()
            .withTable(dataView.table, rowIndex)
            .createSelectionId();
    });
}

```

The visual code iterates the rows of the table and each row calls `withTable` table method. Parameters of `withTable` method are `table` object and index of the table row.

### Create selections for matrix data view mapping

```

public update(options: VisualUpdateOptions) {
    const host = this.host;
    const rowLevels: powerbi.DataViewHierarchyLevel[] = dataView.matrix.rows.levels;
    const columnLevels: powerbi.DataViewHierarchyLevel[] = dataView.matrix.rows.levels;

    // iterate rows hierarchy
    nodeWalker(dataView.matrix.rows.root, rowLevels);
    // iterate columns hierarchy
    nodeWalker(dataView.matrix.columns.root, columnLevels);

    function nodeWalker(node: powerbi.DataViewMatrixNode, levels: powerbi.DataViewHierarchyLevel[]) {
        const nodeSelection = host.createSelectionIdBuilder().withMatrixNode(node, levels);

        if (node.children && node.children.length) {
            node.children.forEach(child => {
                nodeWalker(child, levels);
            });
        }
    }
}

```

In the sample, `nodeWalker` calls recursively for each node and child nodes.

`nodeWalker` creates `nodeSelection` object on each call. And each `nodeSelection` represent `selection` of correspond nodes.

### Select datapoints to slice other visuals

In the sample, codes of selections for categorical data view mapping, you saw that we created a click handler for button elements. The handler calls `select` method of the selection manager and passes the selection object.

```

button.addEventListener("click", () => {
    // handle click event to apply correspond selection
    this.selectionManager.select(categorySelectionId);
});

```

The interface of `select` method is

```

interface ISelectionManager {
    // ...
    select(selectionId: ISelectionId | ISelectionId[], multiSelect?: boolean): IPromise<ISelectionId[]>;
    // ...
}

```

You can see `select` can accept an array of selections. It means your visual can select several datapoints. The second parameter `multiSelect` responsible for multi-select. If the value is true, Power BI doesn't clear the previous selection state and apply current selection otherwise previous selection will reset.

The typical scenario of using `multiSelect` handling CTRL button state on click event.

```
button.addEventListener("click", (mouseEvent) => {
    const multiSelect = (mouseEvent as MouseEvent).ctrlKey;
    this.selectionManager.select(seriesSelectionId, multiSelect);
});
```

## Next steps

- [Read how to use selections for binding visual properties to data points](#)
- [Read how to handle selections on bookmarks switching](#)
- [Read how to add context menu for visuals data points](#)
- [Read how to use InteractivityUtils to add selections into Power BI Visuals](#)

# Tooltips in Power BI visuals

4/9/2020 • 5 minutes to read • [Edit Online](#)

Visuals can now make use of Power BI tooltip support. Power BI tooltips handle the following interactions:

- Show a tooltip.
- Hide a tooltip.
- Move a tooltip.

Tooltips can display a textual element with a title, a value in a given color, and opacity at a specified set of coordinates. This data is provided to the API, and the Power BI host renders it the same way it renders tooltips for native visuals.

A tooltip in a sample bar chart is shown in the following image:



The preceding tooltip image illustrates a single bar category and value. You can extend a single tooltip to display multiple values.

## Manage tooltips

The interface through which you manage tooltips is the "ITooltipService." It's used to notify the host that a tooltip needs to be displayed, removed, or moved.

```
interface ITooltipService {  
    enabled(): boolean;  
    show(options: TooltipShowOptions): void;  
    move(options: TooltipMoveOptions): void;  
    hide(options: TooltipHideOptions): void;  
}
```

Your visual needs to listen to the mouse events within your visual and call the `show()`, `move()`, and `hide()`

delegates, as needed, with the appropriate content populated in the `Tooltip****Options` objects.

`TooltipShowOptions` and `TooltipHideOptions` would in turn define what to display and how to behave in these events.

Because calling these methods involves user events such as mouse moves and touch events, it's a good idea to create listeners for these events, which would in turn invoke the `TooltipService` members. Our sample aggregates in a class called `TooltipServiceWrapper`.

### The `TooltipServiceWrapper` class

The basic idea behind this class is to hold the instance of the `TooltipService`, listen to D3 mouse events over relevant elements, and then make the calls to `show()` and `hide()` the elements when needed.

The class holds and manages any relevant state and logic for these events, which are mostly geared at interfacing with the underlying D3 code. The D3 interfacing and conversion is out of scope for this article.

You can find the full sample code in [SampleBarChart visual repository](#).

### Create `TooltipServiceWrapper`

The bar chart constructor now has a `TooltipServiceWrapper` member, which is instantiated in the constructor with the host `tooltipService` instance.

```
private tooltipServiceWrapper: ITooltipServiceWrapper;  
  
this.tooltipServiceWrapper = createTooltipServiceWrapper(this.host.tooltipService, options.element);
```

The `TooltipServiceWrapper` class holds the `tooltipService` instance, also as the root D3 element of the visual and touch parameters.

```
class TooltipServiceWrapper implements ITooltipServiceWrapper {  
    private handleTouchTimeoutId: number;  
    private visualHostTooltipService: ITooltipService;  
    private rootElement: Element;  
    private handleTouchDelay: number;  
  
    constructor(tooltipService: ITooltipService, rootElement: Element, handleTouchDelay: number) {  
        this.visualHostTooltipService = tooltipService;  
        this.handleTouchDelay = handleTouchDelay;  
        this.rootElement = rootElement;  
    }  
    .  
    .  
    .  
}
```

The single entry point for this class to register event listeners is the `addTooltip` method.

### The `addTooltip` method

```

public addTooltip<T>(
    selection: d3.Selection<Element>,
    getTooltipInfoDelegate: (args: TooltipEventArgs<T>) => VisualTooltipDataItem[],
    getDataPointIdentity: (args: TooltipEventArgs<T>) => ISelectionId,
    reloadTooltipDataOnMouseMove?: boolean): void {

    if (!selection || !this.visualHostTooltipService.enabled()) {
        return;
    }
    ...
    ...
}

```

- **selection: d3.Selection**: The d3 elements over which tooltips are handled.
- **getTooltipInfoDelegate: (args: TooltipEventArgs) => VisualTooltipDataItem[]**: The delegate for populating the tooltip content (what to display) per context.
- **getDataPointIdentity: (args: TooltipEventArgs) => ISelectionId**: The delegate for retrieving the data point ID (unused in this sample).
- **reloadTooltipDataOnMouseMove? boolean**: A Boolean that indicates whether to refresh the tooltip data during a MouseMove event (unused in this sample).

As you can see, `addTooltip` exits with no action if the `tooltipService` is disabled or there's no real selection.

### Call the show method to display a tooltip

The `addTooltip` method next listens to the D3 `mouseover` event, as shown in the following code:

```

...
...
selection.on("mouseover.tooltip", () => {
    // Ignore mouseover while handling touch events
    if (!this.canDisplayTooltip(d3.event))
        return;

    let tooltipEventArgs = this.makeTooltipEventArgs<T>(rootNode, true, false);
    if (!tooltipEventArgs)
        return;

    let tooltipInfo = getTooltipInfoDelegate(tooltipEventArgs);
    if (tooltipInfo == null)
        return;

    let selectionId = getDataPointIdentity(tooltipEventArgs);

    this.visualHostTooltipService.show({
        coordinates: tooltipEventArgs.coordinates,
        isTouchEvent: false,
        dataItems: tooltipInfo,
        identities: selectionId ? [selectionId] : []
    });
});

```

- **makeTooltipEventArgs**: Extracts the context from the D3 selected elements into a tooltipEventArgs. It calculates the coordinates as well.
- **getTooltipInfoDelegate**: It then builds the tooltip content from the tooltipEventArgs. It's a callback to the BarChart class, because it is the visual's logic. It's the actual text content to display in the tooltip.
- **getDataPointIdentity**: Unused in this sample.

- `this.visualHostTooltipService.show`: The call to display the tooltip.

Additional handling can be found in the sample for `mouseout` and `mousemove` events.

For more information, see the [SampleBarChart visual repository](#).

### Populate the tooltip content by the `getTooltipData` method

The BarChart class was added with a `getTooltipData` member, which simply extracts the `category`, `value`, and `color` of the data point into a `VisualTooltipDataItem[]` element.

```
private static getTooltipData(value: any): VisualTooltipDataItem[] {
    return [
        {
            displayName: value.category,
            value: value.value.toString(),
            color: value.color,
            header: 'ToolTip Title'
        }
    ];
}
```

In the preceding implementation, the `header` member is constant, but you can use it for more complex implementations, which require dynamic values. You can populate the `VisualTooltipDataItem[]` with more than one element, which adds multiple lines to the tooltip. It can be useful in visuals such as stacked bar charts where the tooltip may display data from more than a single data point.

### Call the `addTooltip` method

The final step is to call the `addTooltip` method when the actual data might change. This call takes place in the `BarChart.update()` method. A call is made to monitor the selection of all the 'bar' elements, passing only the `BarChart.getTooltipData()`, as mentioned previously.

```
this.tooltipServiceWrapper.addTooltip(this.barContainer.selectAll('.bar'),
    (tooltipEvent: TooltipEventArgs<number>) => BarChart.getTooltipData(tooltipEvent.data),
    (tooltipEvent: TooltipEventArgs<number>) => null);
```

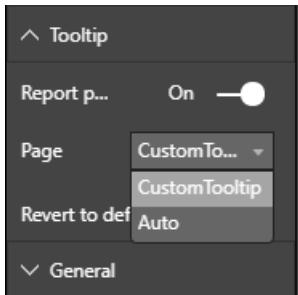
## Add report page tooltips

To add report page tooltips support, you'll find most changes in the `capabilities.json` file.

A sample schema is

```
{
  "tooltips": {
    "supportedTypes": {
      "default": true,
      "canvas": true
    },
    "roles": [
      "tooltips"
    ]
  }
}
```

You can define report page tooltips in the **Format** pane.



- `supportedTypes` : The tooltip configuration that's supported by the visual and reflected in the fields well.
  - `default` : Specifies whether the "automatic" tooltips binding via the data field is supported.
  - `canvas` : Specifies whether the report page tooltips are supported.
- `roles` : (Optional) After it's defined, it instructs what data roles are bound to the selected tooltip option in the fields well.

For more information, see [Report page tooltips usage guidelines](#).

To display the report page tooltip, after the Power BI host calls `ITooltipService.Show(options: TooltipShowOptions)` or `ITooltipService.Move(options: TooltipMoveOptions)`, it consumes the `selectionId` (`identities` property of the preceding `options` argument). To be retrieved by the tooltip, `SelectionId` should represent the selected data (category, series, and so on) of the item you hovered over.

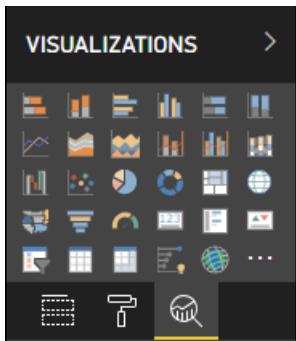
An example of sending the `selectionId` to tooltip display calls is shown in the following code:

```
this.tooltipServiceWrapper.addTooltip(this.barContainer.selectAll('.bar'),
    (tooltipEvent: TooltipEventArgs<number>) => BarChart.getTooltipData(tooltipEvent.data),
    (tooltipEvent: TooltipEventArgs<number>) => tooltipEvent.data.selectionID);
```

# The Analytics pane in Power BI visuals

3/13/2020 • 2 minutes to read • [Edit Online](#)

The **Analytics** pane was introduced for [native visuals](#) in November 2018. This article discusses how Power BI visuals with API v2.5.0 can present and manage their properties in the **Analytics** pane.



## Manage the Analytics pane

Just as you'd manage properties in the [Format pane](#), you manage the **Analytics** pane by defining an object in the visual's *capabilities.json* file.

For the **Analytics** pane, the differences are as follows:

- Under the object's definition, you add an **objectCategory** field with a value of 2.

### NOTE

The optional `objectCategory` field was introduced in API 2.5.0. It defines the aspect of the visual that the object controls (1 = Formatting, 2 = Analytics). `Formatting` is used for such elements as look and feel, colors, axes, and labels. `Analytics` is used for such elements as forecasts, trendlines, reference lines, and shapes.

If the value isn't specified, `objectCategory` defaults to "Formatting."

- The object must have the following two properties:

- `show` of type `bool`, with a default value of `false`.
- `displayName` of type `text`. The default value that you choose becomes the instance's initial display name.

```
{
  "objects": {
    "YourAnalyticsPropertiesCard": {
      "displayName": "Your analytics properties card's name",
      "objectCategory": 2,
      "properties": {
        "show": {
          "type": {
            "bool": true
          }
        },
        "displayName": {
          "type": {
            "text": true
          }
        },
        ... //any other properties for your Analytics card
      }
    }
  ...
}
}
```

You can define other properties in the same way that you do for **Format** objects. And you can enumerate objects just as you do in the **Format** pane.

## Known limitations and issues of the Analytics pane

- The **Analytics** pane has no multi-instance support yet. Objects can't have a **selector** other than static (that is, "selector": null), and Power BI visuals can't have user-defined multiple instances of a card.
- Properties of type `integer` aren't displayed correctly. As a workaround, use type `numeric` instead.

### NOTE

- Use the **Analytics** pane only for objects that add new information or shed new light on the presented information (for example, dynamic reference lines that illustrate important trends).
- Any options that control the look and feel of the visual (that is, formatting) should be limited to the **Formatting** pane.

# The Visual Filters API in Power BI visuals

3/19/2020 • 4 minutes to read • [Edit Online](#)

The Visual Filters API allows you to filter data in Power BI visuals. The main difference from other selections is that other visuals will be filtered in any way, despite highlight support by other visual.

To enable filtering for the visual, it should contain a `filter` object in the `general` section of `capabilities.json` code.

```
"objects": {  
    "general": {  
        "displayName": "General",  
        "displayNameKey": "formattingGeneral",  
        "properties": {  
            "filter": {  
                "type": {  
                    "filter": true  
                }  
            }  
        }  
    }  
}
```

Visual Filters API interfaces are available in the [powerbi-models](#) package. The package also contains classes to create filter instances.

```
npm install powerbi-models --save
```

If you use an older (earlier than 3.x.x) version of the tools, you should include `powerbi-models` in the visuals package. For more information, see the short guide, [Add the Advanced Filter API to the custom visual](#).

All filters extend the `IFilter` interface, as shown in the following code:

```
export interface IFilter {  
    $schema: string;  
    target: IFilterTarget;  
}
```

Where:

- `target` is the table column on the data source.

## The Basic Filter API

Basic filter interface is shown in the following code:

```
export interface IBasicFilter extends IFilter {  
    operator: BasicFilterOperators;  
    values: (string | number | boolean)[];  
}
```

Where:

- `operator` is an enumeration with the values `In`, `NotIn`, and `All`.

- `values` are values for the condition.

Example of a basic filter:

```
let basicFilter = {
    target: {
        column: "Col1"
    },
    operator: "In",
    values: [1,2,3]
}
```

The filter means, "Give me all rows where `col1` equals the value 1, 2, or 3."

The SQL equivalent is:

```
SELECT * FROM table WHERE col1 IN ( 1 , 2 , 3 )
```

To create a filter, you can use the `BasicFilter` class in `powerbi-models`.

If you use an older version of the tool, you should get an instance of models in the window object by using `window['powerbi-models']`, as shown in the following code:

```
let categories: DataViewCategoricalColumn = this.dataView.categorical.categories[0];

let target: IFilterColumnTarget = {
    table: categories.source.queryName.substr(0, categories.source.queryName.indexOf('.')),
    column: categories.source.displayName
};

let values = [ 1, 2, 3 ];

let filter: IBasicFilter = new window['powerbi-models'].BasicFilter(target, "In", values);
```

The visual invokes the filter by using the `applyJsonFilter()` method on the host interface, `IVisualHost`, which is provided to the visual in the constructor.

```
visualHost.applyJsonFilter(filter, "general", "filter", FilterAction.merge);
```

## The Advanced Filter API

The [Advanced Filter API](#) enables complex cross-visual data-point selection and filtering queries that are based on multiple criteria, such as *LessThan*, *Contains*, *Is*, *IsBlank*, and so on).

The filter was introduced in Visuals API 1.7.0.

The Advanced Filter API also requires `target` with a `table` and `column` name. But the Advanced Filter API operators are *And* and *Or*.

Additionally, the filter uses conditions instead of values with the interface:

```
interface IAdvancedFilterCondition {
    value: (string | number | boolean);
    operator: AdvancedFilterConditionOperators;
}
```

Condition operators for the `operator` parameter are `None`, `LessThan`, `LessThanOrEqualTo`, `GreaterThanOrEqual`, `GreaterThanOrEqualTo`, `Contains`, `DoesNotContain`, `StartsWith`, `DoesNotStartWith`, `Is`,  `IsNot`, `IsBlank`, and `"IsNotBlank"`.

```
let categories: DataViewCategoricalColumn = this.dataView.categorical.categories[0];

let target: IFilterColumnTarget = {
    table: categories.source.queryName.substr(0, categories.source.queryName.indexOf('.')), // table
    column: categories.source.displayName // col1
};

let conditions: IAdvancedFilterCondition[] = [];

conditions.push({
    operator: "LessThan",
    value: 0
});

let filter: IAdvancedFilter = new window['powerbi-models'].AdvancedFilter(target, "And", conditions);

// invoke the filter
visualHost.applyJsonFilter(filter, "general", "filter", FilterAction.merge);
```

The SQL equivalent is:

```
SELECT * FROM table WHERE col1 < 0;
```

For the complete sample code for using the Advanced Filter API, go to the [Sampleslicer visual repository](#).

## The Tuple Filter API (multi-column filter)

The Tuple Filter API was introduced in Visuals API 2.3.0. It is similar to the Basic Filter API, but it allows you to define conditions for several columns and tables.

The filter interface is shown in the following code:

```
interface ITupleFilter extends IFilter {
    $schema: string;
    filterType: FilterType;
    operator: TupleFilterOperators;
    target: ITupleFilterTarget;
    values: TupleValueType[];
}
```

Where:

- `target` is an array of columns with table names:

```
declare type ITupleFilterTarget = IFilterTarget[];
```

The filter can address columns from various tables.

- `$schema` is <https://powerbi.com/product/schema#tuple>.
- `filterType` is `FilterType.Tuple`.
- `operator` allows use only in the `/n` operator.
- `values` is an array of value tuples, and each tuple represents one permitted combination of the target

column values.

```
declare type TupleValueType = ITupleElementValue[];  
  
interface ITupleElementValue {  
    value: PrimitiveValueType  
}
```

Complete example:

```
let target: ITupleFilterTarget = [  
    {  
        table: "DataTable",  
        column: "Team"  
    },  
    {  
        table: "DataTable",  
        column: "Value"  
    }  
];  
  
let values = [  
    [  
        // the first column combination value (or the column tuple/vector value) that the filter will pass  
        through  
        {  
            value: "Team1" // the value for the `Team` column of the `DataTable` table  
        },  
        {  
            value: 5 // the value for the `Value` column of the `DataTable` table  
        }  
    ],  
    [  
        // the second column combination value (or the column tuple/vector value) that the filter will pass  
        through  
        {  
            value: "Team2" // the value for `Team` column of `DataTable` table  
        },  
        {  
            value: 6 // the value for `Value` column of `DataTable` table  
        }  
    ]  
];  
  
let filter: ITupleFilter = {  
    $schema: "https://powerbi.com/product/schema#tuple",  
    filterType: FilterType.Tuple,  
    operator: "In",  
    target: target,  
    values: values  
}  
  
// invoke the filter  
visualHost.applyJsonFilter(filter, "general", "filter", FilterAction.merge);
```

#### IMPORTANT

The order of the column names and condition values is sensitive.

The SQL equivalent is:

```
SELECT * FROM DataTable WHERE ( Team = "Team1" AND Value = 5 ) OR ( Team = "Team2" AND Value = 6 );
```

## Restore the JSON filter from the data view

Starting with API version 2.2, you can restore the JSON filter from *VisualUpdateOptions*, as shown in the following code:

```
export interface VisualUpdateOptions extends extensibility.VisualUpdateOptions {
    viewport: IViewport;
    dataViews: DataView[];
    type: VisualUpdateType;
    viewMode?: ViewMode;
    editMode?:EditMode;
    operationKind?: VisualDataChangeOperationKind;
    jsonFilters?: IFilter[];
}
```

When you switch, bookmarks, Power BI calls the `update` method of the visual, and the visual gets a corresponding `filter` object. For more information, see [Add bookmark support for Power BI visuals](#).

### Sample JSON filter

Some sample JSON filter code is shown in the following image:

```
< ▾ {viewport: {...}, dataViews: Array(1), viewMode: 1, editMode: 0, operationKind: 0, ...} ⓘ
  ▶ dataViews: [{}]
  ▶ editMode: 0
  ▾ jsonFilters: Array(1)
    ▾ 0:
      $schema: "http://powerbi.com/product/schema#advanced"
      ▾ conditions: Array(2)
        ▶ 0: {operator: "GreaterThanOrEqualTo", value: "2018-02-28T17:00:00.000Z"}
        ▶ 1: {operator: "LessThan", value: "2018-03-30T17:00:00.000Z"}
        length: 2
      ▶ __proto__: Array(0)
      filterType: 0
      logicalOperator: "And"
      ▶ target: {table: "DateCalendar", column: "Date"}
      ▶ __proto__: Object
      length: 1
    ▶ __proto__: Array(0)
    operationKind: 0
    type: 62
    viewMode: 1
  ▶ viewport: {width: 847.8769356153219, height: 242.02363488182556, scale: 0.36015625}
  ▶ __proto__: Object
```

### Clear the JSON filter

The Filter API accepts the `null` value of the filter as *reset* or *clear*.

```
// invoke the filter
visualHost.applyJsonFilter(null, "general", "filter", FilterAction.merge);
```

# Fetch more data from Power BI

3/19/2020 • 2 minutes to read • [Edit Online](#)

This article discusses how to load more data to bypass the hard limit of a 30-KB data point. This approach provides data in chunks. To improve performance, you can configure the chunk size to accommodate your use case.

## Enable a segmented fetch of large datasets

For the `dataview` segment mode, you define a window size for `dataReductionAlgorithm` in the visual's `capabilities.json` file for the required `dataViewMapping`. The `count` determines the window size, which limits the number of new data rows that can be appended to the `dataview` in each update.

Add the following code in the `capabilities.json` file:

```
"dataViewMappings": [
  {
    "table": {
      "rows": {
        "for": {
          "in": "values"
        },
        "dataReductionAlgorithm": {
          "window": {
            "count": 100
          }
        }
      }
    }
]
```

New segments are appended to the existing `dataview` and provided to the visual as an `update` call.

## Usage in the Power BI visual

You can determine whether data exists by checking the existence of `dataView.metadata.segment`:

```
public update(options: VisualUpdateOptions) {
  const dataView = options.dataViews[0];
  console.log(dataView.metadata.segment);
  // output: __proto__: Object
}
```

You can also check to see whether it's the first update or a subsequent update by checking `options.operationKind`. In the following code, `VisualDataChangeOperationKind.Create` refers to the first segment, and `VisualDataChangeOperationKind.Append` refers to subsequent segments.

For a sample implementation, see the following code snippet:

```
// CV update implementation
public update(options: VisualUpdateOptions) {
    // indicates this is the first segment of new data.
    if (options.operationKind == VisualDataChangeOperationKind.Create) {

    }

    // on second or subsequent segments:
    if (options.operationKind == VisualDataChangeOperationKind.Append) {

    }

    // complete update implementation
}
```

You can also invoke the `fetchMoreData` method from a UI event handler, as shown here:

```
btn_click(){
{
    // check if more data is expected for the current data view
    if (dataView.metadata.segment) {
        //request for more data if available; as a response, Power BI will call update method
        let request_accepted: bool = this.host.fetchMoreData();
        // handle rejection
        if (!request_accepted) {
            // for example, when the 100 MB limit has been reached
        }
    }
}
```

As a response to calling the `this.host.fetchMoreData` method, Power BI calls the `update` method of the visual with a new segment of data.

#### NOTE

To avoid client memory constraints, Power BI currently limits the fetched data total to 100 MB. You can see that the limit has been reached when `fetchMoreData()` returns `false`.

# Add bookmark support for Power BI visuals

3/19/2020 • 5 minutes to read • [Edit Online](#)

With Power BI report bookmarks, you can capture a configured view of a report page, the selection state, and the filtering state of the visual. But it requires additional action from the Power BI visuals side to support the bookmark and react correctly to changes.

For more information about bookmarks, see [Use bookmarks to share insights and build stories in Power BI](#).

## Report bookmarks support in your visual

If your visual interacts with other visuals, selects data points, or filters other visuals, you need to restore the state from properties.

## Add report bookmarks support

1. Install (or update) the required utility, [powerbi-visuals-utils-interactivityutils](#) version 3.0.0 or later. It contains additional classes to manipulate with the state selection or filter. It's required for filter visuals and any visual that uses `InteractivityService`.
2. Update the visual API to version 1.11.0 to use `registerOnSelectCallback` in an instance of `SelectionManager`. It's required for non-filter visuals that use the plain `SelectionManager` rather than `InteractivityService`.

### How Power BI visuals interact with Power BI in report bookmarks

Let's consider the following scenario: you want to create several bookmarks on the report page, with a different selection state in each bookmark.

First, you select a data point in your visual. The visual interacts with Power BI and other visuals by passing selections to the host. You then select **Add** in the **Bookmark** pane, and Power BI saves the current selections for the new bookmark.

It happens repeatedly as you change the selection and add new bookmarks. After you create the bookmarks, you can switch between them.

When you select a bookmark, Power BI restores the saved filter or selection state and passes it to the visuals. Other visuals are highlighted or filtered according to the state that's stored in the bookmark. The Power BI host is responsible for the actions. Your visual is responsible for correctly reflecting the new selection state (for example, for changing the colors of rendered data points).

The new selection state is communicated to the visual through the `update` method. The `options` argument contains a special property, `options.jsonFilters`. Its `JSONFilter` property can contain `Advanced Filter` and `Tuple Filter`.

The visual should restore the filter values to display the corresponding state of the visual for the selected bookmark. Or, if the visual uses selections only, you can use the special callback function call that's registered as the `registerOnSelectCallback` method of `ISelectionManager`.

### Visuals with Selection

If your visual interacts with other visuals by using `Selection`, you can add bookmarks in either of two ways:

- If the visual hasn't already used `InteractivityService`, you can use the `FilterManager.restoreSelectionIds` method.

- If the visual already uses `InteractivityService` to manage selections, you should use the `applySelectionFromFilter` method in the instance of `InteractivityService`.

#### Use `ISelectionManager.registerOnSelectCallback`

When you select a bookmark, Power BI calls the `callback` method of the visual with the corresponding selections.

```
this.selectionManager.registerOnSelectCallback(
  (ids: ISelectionId[]) => {
    //called when a selection was set by Power BI
  });
);
```

Let's assume that you have a data point in your visual that was created in the `visualTransform` method.

And `datapoints` looks like:

```
visualDataPoints.push({
  category: categorical.categories[0].values[i],
  color: getCategoricalObjectValue<Fill>(categorical.categories[0], i, 'colorSelector', 'fill',
  defaultColor).solid.color,
  selectionId: host.createSelectionIdBuilder()
    .withCategory(categorical.categories[0], i)
    .createSelectionId(),
  selected: false
});
```

You now have `visualDataPoints` as your data points and the `ids` array passed to the `callback` function.

At this point, the visual should compare the `ISelectionId[]` array with the selections in your `visualDataPoints` array and mark the corresponding data points as selected.

```
this.selectionManager.registerOnSelectCallback(
  (ids: ISelectionId[]) => {
    visualDataPoints.forEach(dataPoint => {
      ids.forEach(bookmarkSelection => {
        if (bookmarkSelection.equals(dataPoint.selectionId)) {
          dataPoint.selected = true;
        }
      });
    });
  });
);
```

After you update the data points, they'll reflect the current selection state that's stored in the `filter` object. Then, when the data points are rendered, the custom visual's selection state will match the state of the bookmark.

#### Use `InteractivityService` for control selections in the visual

If your visual uses `InteractivityService`, you don't need any additional actions to support the bookmarks in your visual.

When you select bookmarks, the utility handles the visual's selection state.

#### Visuals with a filter

Let's assume that the visual creates a filter of data by date range. You have `startDate` and `endDate` as the start and end dates of the range.

The visual creates an advanced filter and calls the host method `applyJsonFilter` to filter data by the relevant conditions.

The target is the table that's used for filtering.

```
import { AdvancedFilter } from "powerbi-models";

const filter: IAdvancedFilter = new AdvancedFilter(
    target,
    "And",
    {
        operator: "GreaterThanOrEqualTo",
        value: startDate
            ? startDate.toJSON()
            : null
    },
    {
        operator: "LessThanOrEqualTo",
        value: endDate
            ? endDate.toJSON()
            : null
    });
);

this.host.applyJsonFilter(
    filter,
    "general",
    "filter",
    (startDate && endDate)
        ? FilterAction.merge
        : FilterAction.remove
);
```

Each time you select a bookmark, the custom visual gets an `update` call.

The custom visual should check the filter in the object:

```
const filter: IAdvancedFilter = FilterManager.restoreFilter(
    && options.jsonFilters
    && options.jsonFilters[0] as any
) as IAdvancedFilter;
```

If the `filter` object isn't null, the visual should restore the filter conditions from the object:

```
const jsonFilters: AdvancedFilter = this.options.jsonFilters as AdvancedFilter[];

if (jsonFilters
    && jsonFilters[0]
    && jsonFilters[0].conditions
    && jsonFilters[0].conditions[0]
    && jsonFilters[0].conditions[1]
) {
    const startDate: Date = new Date(`${jsonFilters[0].conditions[0].value}`);
    const endDate: Date = new Date(`${jsonFilters[0].conditions[1].value}`);

    // apply restored conditions
} else {
    // apply default settings
}
```

After that, the visual should change its internal state to reflect the current conditions. The internal state includes the data points and visualization objects (lines, rectangles, and so on).

## **IMPORTANT**

In the report bookmarks scenario, the visual shouldn't call `applyJsonFilter` to filter the other visuals. They will already be filtered by Power BI.

The Timeline Slicer visual changes the range selector to the corresponding data ranges.

For more information, see the [Timeline Slicer repository](#).

## **Filter the state to save visual properties in bookmarks**

The `filterState` property makes a property of a part of filtering. The visual can store a variety of values in bookmarks.

To save the property value as a filter state, mark the object property as `"filterState": true` in the *capabilities.json* file.

For example, the Timeline Slicer stores the `Granularity` property values in a filter. It allows the current granularity to change as you change the bookmarks.

For more information, see the [Timeline Slicer repository](#).

# Add context menu to Power BI Visual

3/19/2020 • 2 minutes to read • [Edit Online](#)

You can use `selectionManager.showContextMenu()` with parameters `selectionId` and a position (as an `{x:, y:}` object) to have Power BI display a context menu for your visual.

## IMPORTANT

The `selectionManager.showContextMenu()` was introduced in Visuals API 2.2.0.

Typically it's added as a right-click event (or long-press for touch devices) Context-Menu was added to the sample BarChart for reference:

```
public update(options: VisualUpdateOptions) {
    //...
    //handle context menu
    this.svg.on('contextmenu', () => {
        const mouseEvent: MouseEvent = d3.event as MouseEvent;
        const eventTarget: EventTarget = mouseEvent.target;
        let dataPoint = d3.select(eventTarget).datum();
        this.selectionManager.showContextMenu(dataPoint? dataPoint.selectionId : {}, {
            x: mouseEvent.clientX,
            y: mouseEvent.clientY
        });
        mouseEvent.preventDefault();
    });
}
```

# Add Drill-Down support

3/13/2020 • 8 minutes to read • [Edit Online](#)

Power BI visuals can use Power BI's drill-down.

Read more about Power BI drill-down [here](#)

## Enable drill-down support in the visual

To support drill down in your visual, add a new field to `capabilities.json` named "drill-down", which has one property:

```
*roles - the name of the dataRole you want to enable drill-down on.
```

### NOTE

The drill-down dataRole must be of `Grouping` type. `max` property in the dataRole conditions must be set to 1.

Once you add the role to drill-down, users can drag multiple fields into the data role.

example:

```
{
  "dataRoles": [
    {
      "displayName": "Category",
      "name": "category",
      "kind": "Grouping"
    },
    {
      "displayName": "Value",
      "name": "value",
      "kind": "Measure"
    }
  ],
  "drilldown": {
    "roles": [
      "category"
    ]
  },
  "dataViewMappings": [
    {
      "categorical": {
        "categories": {
          "for": {
            "in": "category"
          }
        },
        "values": {
          "select": [
            {
              "bind": {
                "to": "value"
              }
            }
          ]
        }
      }
    }
  ]
}
```

## Create the visual with drill-down support

Run

```
pbviz new testDrillDown -t default
```

to create a default sample visual. And apply the above sample of `capabilities.json` to the newly created visual.

Create the property for `div` container to hold HTML elements of the visual:

```

"use strict";

import "core-js/stable";
import "../../style/visual.less";
// imports

export class Visual implements IVisual {
    // visual properties
    // ...
    private div: HTMLDivElement; // <== NEW PROPERTY

    constructor(options: VisualConstructorOptions) {
        // constructor body
        // ...
    }

    public update(options: VisualUpdateOptions) {
        // update method body
        // ...
    }

    private static parseSettings(dataView: DataView): VisualSettings {
        return <VisualSettings>VisualSettings.parse(dataView);
    }

    public enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions): VisualObjectInstance[] | VisualObjectInstanceEnumerationObject {
        return VisualSettings.enumerateObjectInstances(this.settings || VisualSettings.getDefault(), options);
    }
}

```

Update the constructor of the visual:

```

export class Visual implements IVisual {
    // visual properties
    // ...
    private div: HTMLDivElement;

    constructor(options: VisualConstructorOptions) {
        console.log('Visual constructor', options);
        this.target = options.element;
        this.updateCount = 0;

        const new_p: HTMLElement = document.createElement("p");
        new_p.appendChild(document.createTextNode("Hierarchy level:"));
        const new_em: HTMLElement = document.createElement("em");
        this.textNode = document.createTextNode(this.updateCount.toString());
        new_em.appendChild(this.textNode);
        new_p.appendChild(new_em);
        this.target.appendChild(new_p);

        this.div = document.createElement("div"); // <== CREATE DIV ELEMENT
        this.target.appendChild(this.div);
    }
}

```

Update the `update` method of the visual to create `button`s:

```

export class Visual implements IVisual {
    // ...

    public update(options: VisualUpdateOptions) {
        this.settings = Visual.parseSettings(options && options.dataViews && options.dataViews[0]);
        console.log('Visual update', options);

        const dataView: DataView = options.dataViews[0];
        const categoricalDataView: DataViewCategorical = dataView.categorical;

        // don't create elements if no data
        if (!options.dataViews[0].categorical ||
            !options.dataViews[0].categorical.categories) {
            return
        }

        // to display current level of hierarchy
        if (typeof this.textNode !== undefined) {
            this.textNode.textContent = categoricalDataView.categories[categoricalDataView.categories.length - 1].source.displayName.toString();
        }

        // remove old elements
        // for better performance use D3js pattern:
        // https://d3js.org/#enter-exit
        while (this.div.firstChild) {
            this.div.removeChild(this.div.firstChild);
        }

        // create buttons for each category value
        categoricalDataView.categories[categoricalDataView.categories.length - 1].values.forEach( (category: powerbi.PrimitiveValue, index: number) => {
            let button = document.createElement("button");
            button.innerText = category.toString();

            this.div.appendChild(button);
        })
    }
    // ...
}

```

Apply simple styles in `.\style\visual.less`:

```

button {
    margin: 5px;
    min-width: 50px;
    min-height: 50px;
}

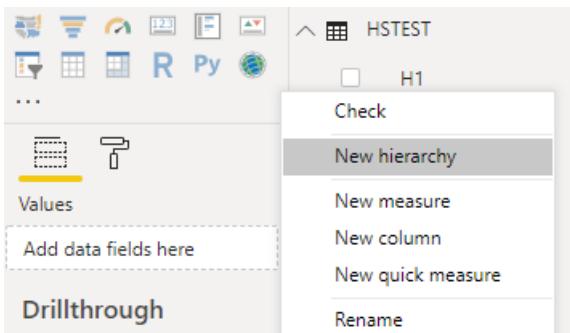
```

Prepare sample data to test the visual:

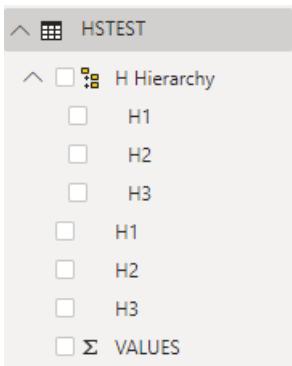
H1	H2	H3	VALUES
A	A1	A11	1
A	A1	A12	2
A	A2	A21	3
A	A2	A22	4

H1	H2	H3	VALUES
A	A3	A31	5
A	A3	A32	6
B	B1	B11	7
B	B1	B12	8
B	B2	B21	9
B	B2	B22	10
B	B3	B31	11
B	B3	B32	12

And create Hierarchy in Power BI Desktop:



Include all category columns (H1, H2, H3) to the new hierarchy:



After those steps you should get following visual:

VALUES by H1

Hierarchy level: **H1**

H1	H2	H3	VALUES
A	A1	A11	1
A	A1	A12	2
A	A2	A21	3
A	A2	A22	4
A	A3	A31	5
A	A3	A32	6
B	B1	B11	7
B	B1	B12	8
B	B2	B21	9
B	B2	B22	10
B	B3	B31	11
B	B3	B32	12
Total			78

↑ ↓ ⇠ ⇢ ⚡ ⚢ ⚤ ...

↻ ⏴ ⏵ ⏶ ? ☺

# Add context menu to visual elements

In this step you'll add context menu to the button's on the visual:

A screenshot of the Qlik Sense interface. At the top left, it says "VALUES by H1". Below this, a bolded text "Hierarchy level: H1" is highlighted with a yellow box. A context menu is open over this text, listing the following options from top to bottom: "Drill down", "Show Next Level", "Expand to next level", "Show data", "Include", "Exclude", "Group", and "Copy". The "Copy" option has a small arrow pointing to the right next to it. At the bottom of the screen, there is a toolbar with various icons: a circular arrow, a play button, a magnifying glass, a document, a question mark, and a smiley face.

To create context menu, save `host` object in the properties of the visual and call `createSelectionManager` method to the create selection manager to display a context menu by using Power BI Visuals API.

```

"use strict";

import "core-js/stable";
import "../../style/visual.less";
// imports

export class Visual implements IVisual {
    // visual properties
    // ...
    private div: HTMLDivElement;
    private host: IVisualHost; // <== NEW PROPERTY
    private selectionManager: ISelectionManager; // <== NEW PROPERTY

    constructor(options: VisualConstructorOptions) {
        // constructor body
        // save the host in the visuals properties
        this.host = options.host;
        // create selection manager
        this.selectionManager = this.host.createSelectionManager();
        // ...
    }

    public update(options: VisualUpdateOptions) {
        // update method body
        // ...
    }

    // ...
}

```

Change the body of `forEach` function callback to:

```

categoricalDataView.categories[categoricalDataView.categories.length - 1].values.forEach( (category: powerbi.PrimitiveValue, index: number) => {
    // create selectionID for each category value
    let selectionID: ISelectionID = this.host.createSelectionIdBuilder()
        .withCategory(categoricalDataView.categories[0], index)
        .createSelectionId();

    let button = document.createElement("button");
    button.innerText = category.toString();

    // add event listener to click event
    button.addEventListener("click", (event) => {
        // call select method in the selection manager
        this.selectionManager.select(selectionID);
    });

    button.addEventListener("contextmenu", (event) => {
        // call showContextMenu method to display context menu on the visual
        this.selectionManager.showContextMenu(selectionID, {
            x: event.clientX,
            y: event.clientY
        });
        event.preventDefault();
    });

    this.div.appendChild(button);
});

```

Apply data to the visual:

The screenshot shows the Power BI Data view ribbon. In the 'Category' section, there is a 'H Hierarchy' dropdown menu with items 'H1', 'H2', and 'H3'. Below it is a 'Value' section with a 'VALUES' dropdown menu. The 'H Hierarchy' and 'VALUES' sections are highlighted with yellow boxes.

In the final step you should get visual with selections and context menu:

The screenshot shows the Power BI visual interface. On the left is a matrix table with columns 'H1', 'H2', 'H3', and 'VALUES'. The data rows are: A, A1, A11, 1; A, A1, A12, 2; A, A2, A21, 3; A, A2, A22, 4; A, A3, A31, 5; A, A3, A32, 6; B, B1, B11, 7; B, B1, B12, 8; B, B2, B21, 9; B, B2, B22, 10; B, B3, B31, 11; B, B3, B32, 12. A total row at the bottom shows 'Total' and '78'. To the right is a 'Filters' pane with sections for 'Filters on this visual' (H1 is (All), H2 is (All), H3 is (All), VALUES is (All)) and 'Filters on this page' (Add data fields here).

## Add drill-down support for matrix data view mapping

Prepare sample data to test the visual with matrix data view mappings:

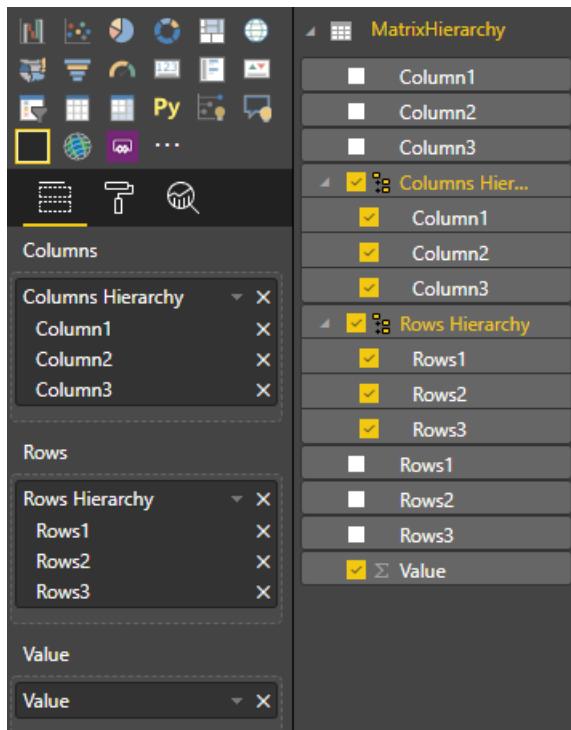
ROW1	ROW2	ROW3	COLUMN1	COLUMN2	COLUMN3	VALUES
R1	R11	R111	C1	C11	C111	1
R1	R11	R112	C1	C11	C112	2
R1	R11	R113	C1	C11	C113	3
R1	R12	R121	C1	C12	C121	4
R1	R12	R122	C1	C12	C122	5
R1	R12	R123	C1	C12	C123	6

ROW1	ROW2	ROW3	COLUMN1	COLUMN2	COLUMN3	VALUES
R1	R13	R131	C1	C13	C131	7
R1	R13	R132	C1	C13	C132	8
R1	R13	R133	C1	C13	C133	9
R2	R21	R211	C2	C21	C211	10
R2	R21	R212	C2	C21	C212	11
R2	R21	R213	C2	C21	C213	12
R2	R22	R221	C2	C22	C221	13
R2	R22	R222	C2	C22	C222	14
R2	R22	R223	C2	C22	C223	16
R2	R23	R231	C2	C23	C231	17
R2	R23	R232	C2	C23	C232	18
R2	R23	R233	C2	C23	C233	19

Apply following dataview mapping for the visual:

```
{  
    "dataRoles": [  
        {  
            "displayName": "Columns",  
            "name": "columns",  
            "kind": "Grouping"  
        },  
        {  
            "displayName": "Rows",  
            "name": "rows",  
            "kind": "Grouping"  
        },  
        {  
            "displayName": "Value",  
            "name": "value",  
            "kind": "Measure"  
        }  
    ],  
    "drilldown": {  
        "roles": [  
            "columns",  
            "rows"  
        ]  
    },  
    "dataViewMappings": [  
        {  
            "matrix": {  
                "columns": {  
                    "for": {  
                        "in": "columns"  
                    }  
                },  
                "rows": {  
                    "for": {  
                        "in": "rows"  
                    }  
                },  
                "values": {  
                    "for": {  
                        "in": "value"  
                    }  
                }  
            }  
        }  
    ]  
}
```

Apply data to the visual:



Import required interfaces to process matrix data view mappings:

```
// ...
import DataViewMatrix = powerbi.DataViewMatrix;
import DataViewMatrixNode = powerbi.DataViewMatrixNode;
import DataViewHierarchyLevel = powerbi.DataViewHierarchyLevel;
// ...
```

Create two properties for two `div`s of rows and columns elements:

```
export class Visual implements IVisual {
    // ...
    private rowsDiv: HTMLDivElement;
    private colsDiv: HTMLDivElement;
    // ...
    constructor(options: VisualConstructorOptions) {
        // constructor body
        // ...
        // Create div elements and append to main div of the visual
        this.rowsDiv = document.createElement("div");
        this.target.appendChild(this.rowsDiv);

        this.colsDiv = document.createElement("div");
        this.target.appendChild(this.colsDiv);
    }
    // ...
}
```

Check the data before rendering elements and display the current level of hierarchy:

```
export class Visual implements IVisual {
    // ...
    constructor(options: VisualConstructorOptions) {
        // constructor body
    }

    public update(options: VisualUpdateOptions) {
        this.settings = Visual.parseSettings(options && options.dataViews && options.dataViews[0]);
        console.log('Visual update', options);

        const dataView: DataView = options.dataViews[0];
        const matrixDataView: DataViewMatrix = dataView.matrix;

        // if the visual doesn't receive the data no reason to continue rendering
        if (!matrixDataView ||
            !matrixDataView.columns ||
            !matrixDataView.rows ) {
            return
        }

        // to display current level of hierarchy
        if (typeof this.textNode !== undefined) {
            this.textNode.textContent = categoricalDataView.categories[categoricalDataView.categories.length - 1].source.displayName.toString();
        }
        // ...
    }
    // ...
}
```

Create function `treeWalker` for traverse the hierarchy:

```

export class Visual implements IVisual {
    // ...
    public update(options: VisualUpdateOptions) {
        // ...

        // if the visual doesn't receive the data no reason to continue rendering
        if (!matrix DataView || !matrix DataView.columns || !matrix DataView.rows ) {
            return
        }

        const treeWalker = (matrixNode: DataViewMatrixNode, index: number, levels: DataViewHierarchyLevel[], div: HTMLDivElement) => {
            // ...
            if (matrixNode.children) {
                // ...
                // traversing child nodes
                matrixNode.children.forEach((node, index) => treeWalker(node, index, levels, childDiv));
            }
        }

        // traversing rows
        const rowRoot: DataViewMatrixNode = matrix DataView.rows.root;
        rowRoot.children.forEach((node, index) => treeWalker(node, index, matrix DataView.rows.levels, this.rowsDiv));

        // traversing columns
        const colRoot = matrix DataView.columns.root;
        colRoot.children.forEach((node, index) => treeWalker(node, index, matrix DataView.columns.levels, this.colsDiv));
    }
    // ...
}

```

Generate the selections for datapoints.

```

const treeWalker = (matrixNode: DataViewMatrixNode, index: number, levels: DataViewHierarchyLevel[], div: HTMLDivElement) => {
    // generate selectionID for each node of matrix
    const selectionID: ISelectionID = this.host.createSelectionIdBuilder()
        .withMatrixNode(matrixNode, levels)
        .createSelectionId();
    // ...
    if (matrixNode.children) {
        // ...
        // traversing child nodes
        matrixNode.children.forEach((node, index) => treeWalker(node, index, levels, childDiv));
    }
}

```

Create `div` for each level of hierarchy:

```

const treeWalker = (matrixNode: DataViewMatrixNode, index: number, levels: DataViewHierarchyLevel[], div: HTMLDivElement) => {
    // generate selectionID for each node of matrix
    const selectionID: ISelectionID = this.host.createSelectionIdBuilder()
        .withMatrixNode(matrixNode, levels)
        .createSelectionId();
    // ...
    if (matrixNode.children) {
        // create div element for level
        const childDiv = document.createElement("div");
        // add to current div
        div.appendChild(childDiv);
        // create paragraph element to display next
        const p = document.createElement("p");
        // display level name on paragraph element
        const level = levels[matrixNode.level];
        p.innerText = level.sources[level.sources.length - 1].displayName;
        // add paragraph element to created child div
        childDiv.appendChild(p);
        // traversing child nodes
        matrixNode.children.forEach((node, index) => treeWalker(node, index, levels, childDiv));
    }
}

```

Create `button` to interact with visual and display context menu for matrix datapoints:

```

const treeWalker = (matrixNode: DataViewMatrixNode, index: number, levels: DataViewHierarchyLevel[], div: HTMLDivElement) => {
    // generate selectionID for each node of matrix
    const selectionID: ISelectionID = this.host.createSelectionIdBuilder()
        .withMatrixNode(matrixNode, levels)
        .createSelectionId();

    // create button element
    let button = document.createElement("button");
    // display node value/name of the button's text
    button.innerText = matrixNode.value.toString();

    // add event listener on click
    button.addEventListener("click", (event) => {
        // call select method in the selection manager
        this.selectionManager.select(selectionID);
    });

    // display context menu on click
    button.addEventListener("contextmenu", (event) => {
        // call showContextMenu method to display context menu on the visual
        this.selectionManager.showContextMenu(selectionID, {
            x: event.clientX,
            y: event.clientY
        });
        event.preventDefault();
    });

    div.appendChild(button);

    if (matrixNode.children) {
        // ...
    }
}

```

Clear `div` elements before render elements again:

```
public update(options: VisualUpdateOptions) {
    // ...
    const treeWalker = (matrixNode: DataViewMatrixNode, index: number, levels: DataViewHierarchyLevel[], div: HTMLDivElement) => {
        // ...
    }

    // remove old elements
    // to better performance use D3js pattern:
    // https://d3js.org/#enter-exit
    while (this.rowsDiv.firstChild) {
        this.rowsDiv.removeChild(this.rowsDiv.firstChild);
    }
    // create label for row elements
    const prow = document.createElement("p");
    prow.innerText = "Rows";
    this.rowsDiv.appendChild(prow);

    while (this.colsDiv.firstChild) {
        this.colsDiv.removeChild(this.colsDiv.firstChild);
    }
    // create label for columns elements
    const pcol = document.createElement("p");
    pcol.innerText = "Columns";
    this.colsDiv.appendChild(pcol);

    // render elements for rows
    const rowRoot: DataViewMatrixNode = matrixDataView.rows.root;
    rowRoot.children.forEach((node, index) => treeWalker(node, index, matrixDataView.rows.levels,
    this.rowsDiv));

    // render elements for columns
    const colRoot = matrixDataView.columns.root;
    colRoot.children.forEach((node, index) => treeWalker(node, index, matrixDataView.columns.levels,
    this.colsDiv));
}
```

At the final step you should get visual with context menu:

VALUES by H1

Hierarchy level: **H1**

B A

↑ ↓ ↻ ⌂ ⌂ ...

H1	H2	H3	VALUES
A	A1	A11	1
A	A1	A12	2
A	A2	A21	3
A	A2	A22	4
A	A3	A31	5
A	A3	A32	6
B	B1	B11	7
B	B1	B12	8
B	B2	B21	9
B	B2	B22	10
B	B3	B31	11
B	B3	B32	12
Total			78

Filters

Search

Filters on this visual ...

H1  
is (All)

H2  
is (All)

H3  
is (All)

VALUES  
is (All)

Add data fields here

Filters on this page ...

Add data fields here

## Next steps

For more information, see [Understand data view mapping in Power BI visuals](#).

# Add colors to your Power BI visuals

5/11/2020 • 2 minutes to read • [Edit Online](#)

This article describes how to add colors to your visuals and how to handle data points for a color visual.

`IVisualHost` exposes color as one of its services. The example code in this article modifies the [SampleBarChart visual](#). For source code, see [barChart.ts](#).

To get started creating visuals, see [Develop a Power BI visual](#).

## Add color to data points

A different color represents each data point. Add the color to the `BarChartDataPoint` interface, as in the following example:

```
/**  
 * Interface for BarChart data points.  
 *  
 * @interface  
 * @property {number} value - Data value for point.  
 * @property {string} category - Corresponding category of data value.  
 * @property {string} color - Color corresponding to data point.  
 */  
interface BarChartDataPoint {  
    value: number;  
    category: string;  
    color: string;  
};
```

## Use the color palette service

The `colorPalette` service manages the colors used in your visual. An instance of the service is available on `IVisualHost`.

Define it in the `update` method.

```
constructor(options: VisualConstructorOptions) {  
    this.host = options.host; // host: IVisualHost  
    ...  
}  
  
public update(options: VisualUpdateOptions) {  
  
    let colorPalette: IColorPalette = host.colorPalette;  
    ...  
}
```

## Assigning color to data points

Next, specify `dataPoints`. In this example, each of the `dataPoints` includes value, category, and color. `dataPoints` can also include other properties.

In `SampleBarChart`, the `visualTransform` method encapsulates the `dataPoints` calculation. That method is a part of the Bar Chart.viewmodel. Because the method iterates through the `dataPoints` calculation in `visualTransform`, it's

the ideal place to assign colors, as in the following code:

```
public update(options: VisualUpdateOptions) {
    ....
    let viewModel: BarChartViewModel = visualTransform(options, this.host);
    ....
}

function visualTransform(options: VisualUpdateOptions, host: IVisualHost): BarChartViewModel {
    let colorPalette: IColorPalette = host.colorPalette; // host: IVisualHost
    for (let i = 0, len = Math.max(category.values.length, dataValue.values.length); i < len; i++) {
        barChartDataPoints.push({
            category: category.values[i],
            value: dataValue.values[i],
            color: colorPalette.getColor(category.values[i]).value,
        });
    }
}
```

Then apply data from `dataPoints` on the `d3-selection` `barSelection` inside the `update` method:

```
// This code is actual for d3 v5
// in d3 v5 for this case we should use merge() after enter() and apply changes on barSelectionMerged
this.barSelection = this.barContainer
    .selectAll('.bar')
    .data(this.barDataPoints);

const barSelectionMerged = this.barSelection
    .enter()
    .append('rect')
    .merge(<any>this.barSelection);

barSelectionMerged.classed('bar', true);

barSelectionMerged
    .attr("width", xScale.bandwidth())
    .attr("height", d => height - yScale(<number>d.value))
    .attr("y", d => yScale(<number>d.value))
    .attr("x", d => xScale(d.category))
    .style("fill", (dataPoint: BarChartDataPoint) => dataPoint.color)
    .style("stroke", (dataPoint: BarChartDataPoint) => dataPoint.strokeColor)
    .style("stroke-width", (dataPoint: BarChartDataPoint) => `${dataPoint.strokeWidth}px`);

this.barSelection
    .exit()
    .remove();
```

## Next steps

To learn more about Power BI visuals, see [Capabilities and properties of Power BI visuals](#).

To learn more about developing Power BI visuals, see [How to debug Power BI visuals](#) and [Troubleshoot Power BI visuals](#).

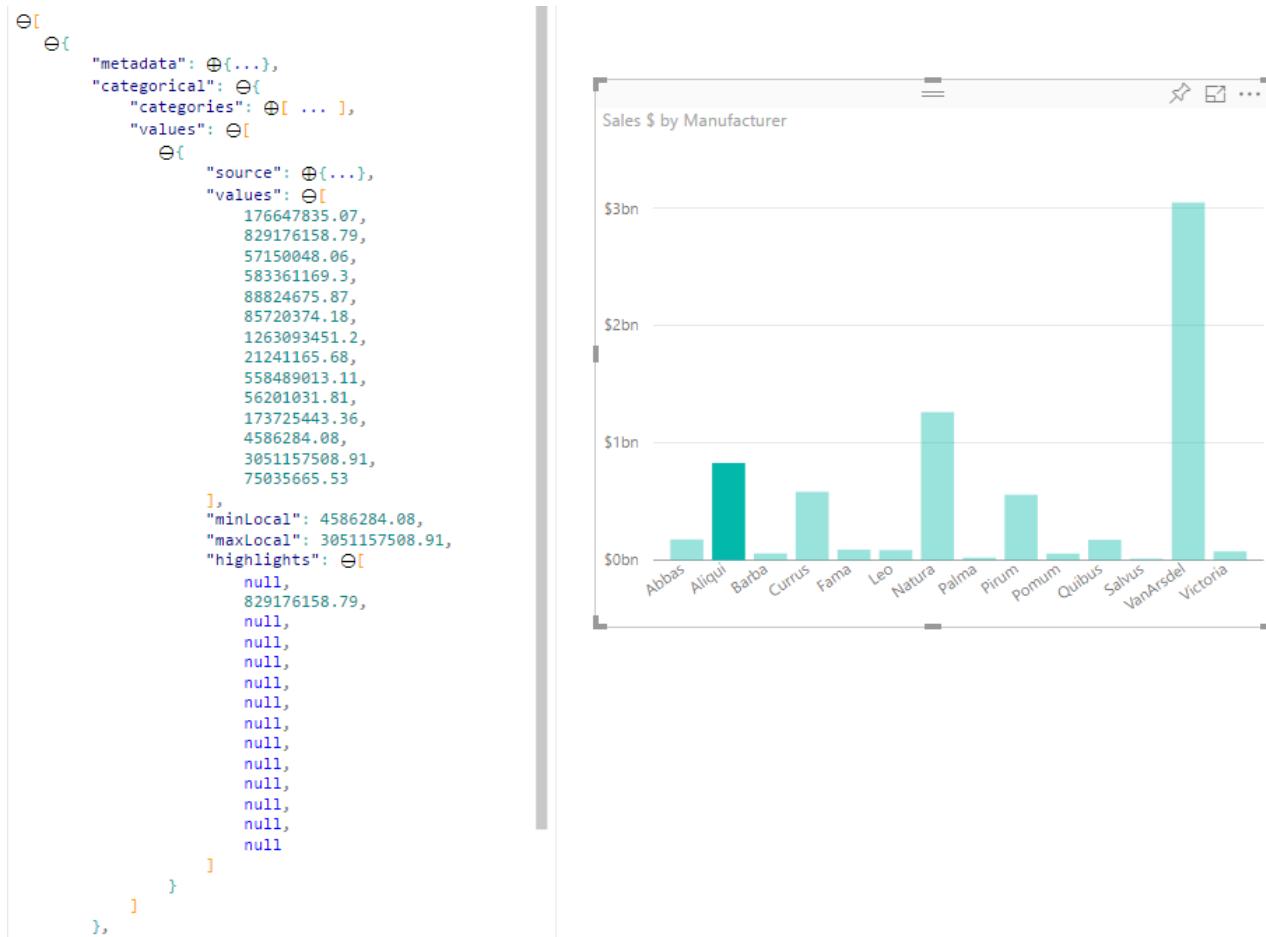
# Highlight data points in Power BI Visuals

3/19/2020 • 9 minutes to read • [Edit Online](#)

By default whenever an element is selected the `values` array in the `dataView` object will be filtered to just the selected values. It will cause all other visuals on the page to display just the selected data.



If you set the `supportsHighlight` property in your `capabilities.json` to `true`, you'll receive the full unfiltered `values` array along with a `highlights` array. The `highlights` array will be the same length as the `values` array and any non-selected values will be set to `null`. With this property enabled it's the visual's responsibility to highlight the appropriate data by comparing the `values` array to the `highlights` array.



In the example, you'll notice that 1 bar is selected. And it's the only value in the `highlights` array. It's also important

to note that there could be multiple selections and partial highlights. The highlighted values will be presented in the data view.

#### NOTE

Table data view mapping doesn't support the highlights feature.

## Highlight data points with categorical data view mapping

The visuals with categorical data view mapping have `capabilities.json` with `"supportsHighlight": true` parameter. For example:

```
{
  "dataRoles": [
    {
      "displayName": "Category",
      "name": "category",
      "kind": "Grouping"
    },
    {
      "displayName": "Value",
      "name": "value",
      "kind": "Measure"
    }
  ],
  "dataViewMappings": [
    {
      "categorical": {
        "categories": {
          "for": {
            "in": "category"
          }
        },
        "values": {
          "for": {
            "in": "value"
          }
        }
      }
    }
  ],
  "supportsHighlight": true
}
```

The default visual source code after removing unnecessary code will look like this:

```

"use strict";

// ... default imports list

import DataViewCategorical = powerbi.DataViewCategorical;
import DataViewCategoryColumn = powerbi.DataViewCategoryColumn;
import PrimitiveValue = powerbi.PrimitiveValue;
import DataViewValueColumn = powerbi.DataViewValueColumn;

import { VisualSettings } from "./settings";

export class Visual implements IVisual {
    private target: HTMLElement;
    private settings: VisualSettings;

    constructor(options: VisualConstructorOptions) {
        console.log('Visual constructor', options);
        this.target = options.element;
        this.host = options.host;
    }

    public update(options: VisualUpdateOptions) {
        this.settings = Visual.parseSettings(options && options.dataViews && options.dataViews[0]);
        console.log('Visual update', options);
    }

    private static parseSettings(dataView: DataView): VisualSettings {
        return <VisualSettings>VisualSettings.parse(dataView);
    }

    /**
     * This function gets called for each of the objects defined in the capabilities files and allows you to
     * select which of the
     *   * objects and properties you want to expose to the users in the property pane.
     *   *
     */
    public enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions): VisualObjectInstance[] | VisualObjectInstanceEnumerationObject {
        return VisualSettings.enumerateObjectInstances(this.settings || VisualSettings.getDefault(), options);
    }
}

```

Import required interfaces to process data from Power BI:

```

import DataViewCategorical = powerbi.DataViewCategorical;
import DataViewCategoryColumn = powerbi.DataViewCategoryColumn;
import PrimitiveValue = powerbi.PrimitiveValue;
import DataViewValueColumn = powerbi.DataViewValueColumn;

```

Create root `div` element for category values:

```

export class Visual implements IVisual {
    private target: HTMLElement;
    private settings: VisualSettings;

    private div: HTMLDivElement; // new property

    constructor(options: VisualConstructorOptions) {
        console.log('Visual constructor', options);
        this.target = options.element;
        this.host = options.host;

        // create div element
        this.div = document.createElement("div");
        this.div.classList.add("vertical");
        this.target.appendChild(this.div);

    }
    // ...
}

```

Clear content of div elements before rendering new data:

```

// ...

public update(options: VisualUpdateOptions) {
    this.settings = Visual.parseSettings(options && options.dataViews && options.dataViews[0]);
    console.log('Visual update', options);

    while (this.div.firstChild) {
        this.div.removeChild(this.div.firstChild);
    }
    // ...
}

```

Get categories and measure values from `dataView` object:

```

public update(options: VisualUpdateOptions) {
    this.settings = Visual.parseSettings(options && options.dataViews && options.dataViews[0]);
    console.log('Visual update', options);

    while (this.div.firstChild) {
        this.div.removeChild(this.div.firstChild);
    }

    const dataView: DataView = options.dataViews[0];
    const categoricalDataView: DataViewCategorical = dataView.categorical;
    const categories: DataViewCategoryColumn = categoricalDataView.categories[0];
    const categoryValues = categories.values;

    const measures: DataViewValueColumn = categoricalDataView.values[0];
    const measureValues = measures.values;
    const measureHighlights = measures.highlights;
    // ...
}

```

Where `categoryValues` is an array of category values, `measureValues` is an array of measures, and `measureHighlights` is highlighted parts of values.

## NOTE

Values of `measureHighlights` property can be less than values of `categoryValues` property. It means that value was highlighted partially.

Enumerate `categoryValues` array and get corresponding values and highlights:

```
// ...
const measureHighlights = measures.highlights;

categoryValues.forEach((category: PrimitiveValue, index: number) => {
  const measureValue = measureValues[index];
  const measureHighlight = measureHighlights && measureHighlights[index] ? measureHighlights[index] : null;
  console.log(category, measureValue, measureHighlight);

});
```

Create `div` and `p` elements to display and visualize data view values in visual DOM:

```
categoryValues.forEach((category: PrimitiveValue, index: number) => {
  const measureValue = measureValues[index];
  const measureHighlight = measureHighlights && measureHighlights[index] ? measureHighlights[index] : null;
  console.log(category, measureValue, measureHighlight);

  // div element. it contains elements to display values and visualize value as progress bar
  let div = document.createElement("div");
  div.classList.add("horizontal");
  this.div.appendChild(div);

  // div element to visualize value of measure
  let barValue = document.createElement("div");
  barValue.style.width = +measureValue * 10 + "px";
  barValue.style.display = "flex";
  barValue.classList.add("value");

  // element to display category value
  let bp = document.createElement("p");
  bp.innerText = category.toString();

  // div element to visualize highlight of measure
  let barHighlight = document.createElement("div");
  barHighlight.classList.add("highlight")
  barHighlight.style.backgroundColor = "blue";
  barHighlight.style.width = +measureHighlight * 10 + "px";

  // element to display highlighted value of measure
  let p = document.createElement("p");
  p.innerText = `${measureHighlight}/${measureValue}`;
  barHighlight.appendChild(bp);

  div.appendChild(barValue);

  barValue.appendChild(barHighlight);
  div.appendChild(p);
});
```

Apply required styles for elements to use `flex box` and define colors for div elements:

```

div.vertical {
    display: flex;
    flex-direction: column;
}

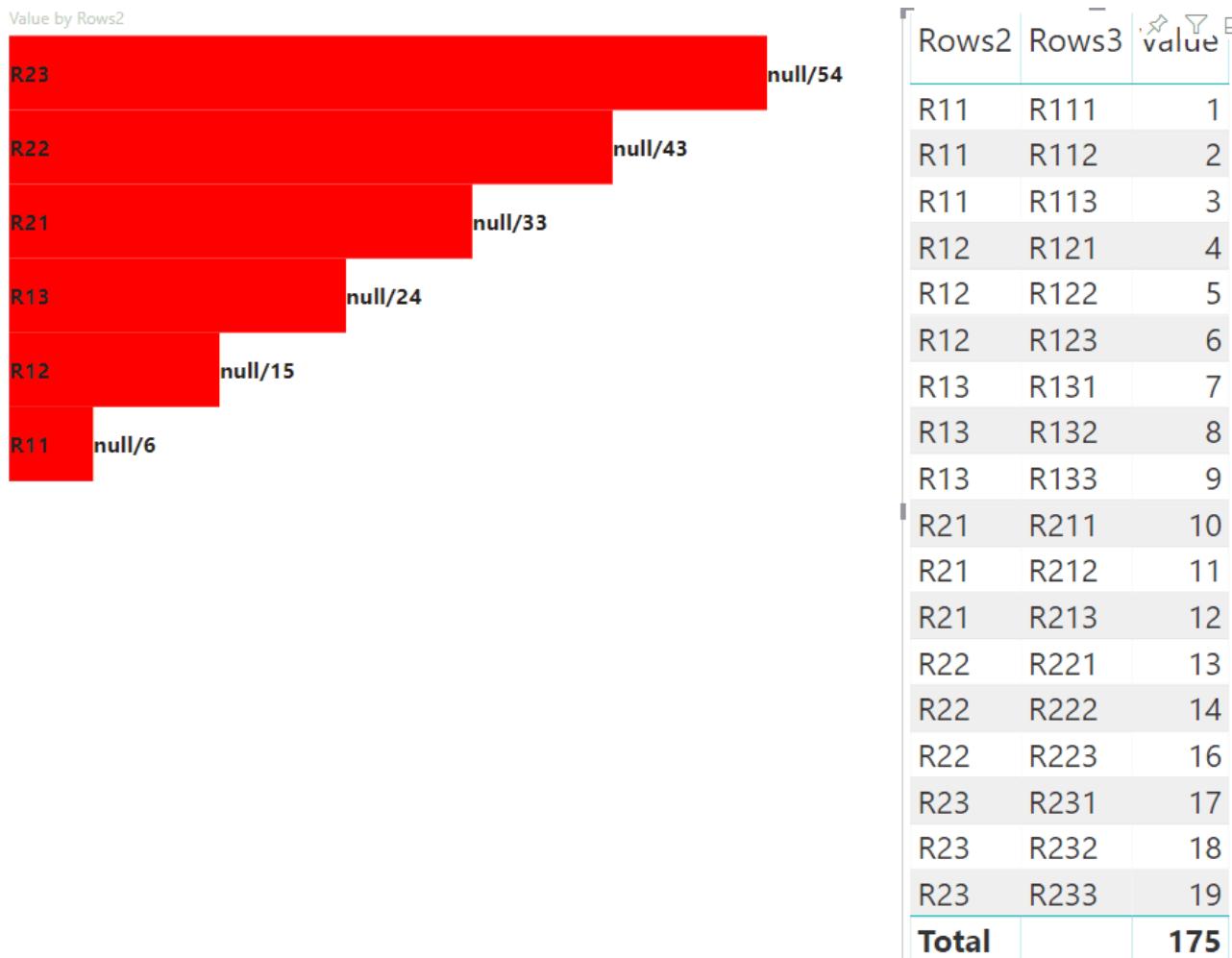
div.horizontal {
    display: flex;
    flex-direction: row;
}

div.highlight {
    background-color: blue
}

div.value {
    background-color: red;
    display: flex;
}

```

In the result, you should have the following view of the visual.



## Highlight data points with matrix data view mapping

The visuals with matrix data view mapping have `capabilities.json` with `"supportsHighlight": true` parameter. For example:

```
{
  "dataRoles": [
    {
      "displayName": "Columns",
      "name": "columns",
      "kind": "Grouping"
    },
    {
      "displayName": "Rows",
      "name": "rows",
      "kind": "Grouping"
    },
    {
      "displayName": "Value",
      "name": "value",
      "kind": "Measure"
    }
  ],
  "dataViewMappings": [
    {
      "matrix": {
        "columns": {
          "for": {
            "in": "columns"
          }
        },
        "rows": {
          "for": {
            "in": "rows"
          }
        },
        "values": {
          "for": {
            "in": "value"
          }
        }
      }
    }
  ],
  "supportsHighlight": true
}
```

The sample data to create hierarchy for matrix data view mapping:

ROW1	ROW2	ROW3	COLUMN1	COLUMN2	COLUMN3	VALUES
R1	R11	R111	C1	C11	C111	1
R1	R11	R112	C1	C11	C112	2
R1	R11	R113	C1	C11	C113	3
R1	R12	R121	C1	C12	C121	4
R1	R12	R122	C1	C12	C122	5
R1	R12	R123	C1	C12	C123	6
R1	R13	R131	C1	C13	C131	7
R1	R13	R132	C1	C13	C132	8

ROW1	ROW2	ROW3	COLUMN1	COLUMN2	COLUMN3	VALUES
R1	R13	R133	C1	C13	C133	9
R2	R21	R211	C2	C21	C211	10
R2	R21	R212	C2	C21	C212	11
R2	R21	R213	C2	C21	C213	12
R2	R22	R221	C2	C22	C221	13
R2	R22	R222	C2	C22	C222	14
R2	R22	R223	C2	C22	C223	16
R2	R23	R231	C2	C23	C231	17
R2	R23	R232	C2	C23	C232	18
R2	R23	R233	C2	C23	C233	19

Create the default visual project and apply sample of `capabilities.json`.

Default visual source code after removing unessesray code will look:

```

"use strict";

// ... default imports

import { VisualSettings } from "./settings";

export class Visual implements IVisual {
    private target: HTMLElement;
    private settings: VisualSettings;

    constructor(options: VisualConstructorOptions) {
        console.log('Visual constructor', options);
        this.target = options.element;
        this.host = options.host;
    }

    public update(options: VisualUpdateOptions) {
        this.settings = Visual.parseSettings(options && options.dataViews && options.dataViews[0]);
        console.log('Visual update', options);
    }

    private static parseSettings(dataView: DataView): VisualSettings {
        return <VisualSettings>VisualSettings.parse(dataView);
    }

    /**
     * This function gets called for each of the objects defined in the capabilities files and allows you to
     * select which of the
     *   * objects and properties you want to expose to the users in the property pane.
     *   *
     */
    public enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions): VisualObjectInstance[] | VisualObjectInstanceEnumerationObject {
        return VisualSettings.enumerateObjectInstances(this.settings || VisualSettings.getDefault(), options);
    }
}

```

Import required interfaces to process data from Power BI:

```

import DataViewMatrix = powerbi.DataViewMatrix;
import DataViewMatrixNode = powerbi.DataViewMatrixNode;
import DataViewHierarchyLevel = powerbi.DataViewHierarchyLevel;

```

Create two `div` elements for visual layout:

```

constructor(options: VisualConstructorOptions) {
    // ...
    this.rowsDiv = document.createElement("div");
    this.target.appendChild(this.rowsDiv);

    this.colsDiv = document.createElement("div");
    this.target.appendChild(this.colsDiv);
    this.target.style.overflowY = "auto";
}

```

Check the data in `update` method, to ensure that visual gets data:

```

public update(options: VisualUpdateOptions) {
    this.settings = Visual.parseSettings(options && options.dataViews && options.dataViews[0]);
    console.log('Visual update', options);

    const dataView: DataView = options.dataViews[0];
    const matrixDataView: DataViewMatrix = dataView.matrix;

    if (!matrixDataView ||
        !matrixDataView.columns ||
        !matrixDataView.rows ) {
        return
    }
    // ...
}

```

Clear content of `div` elements before render new data:

```

public update(options: VisualUpdateOptions) {
    // ...

    // remove old elements
    // to better performance use D3js pattern:
    // https://d3js.org/#enter-exit
    while (this.rowsDiv.firstChild) {
        this.rowsDiv.removeChild(this.rowsDiv.firstChild);
    }
    const prow = document.createElement("p");
    prow.innerText = "Rows";
    this.rowsDiv.appendChild(prow);

    while (this.colsDiv.firstChild) {
        this.colsDiv.removeChild(this.colsDiv.firstChild);
    }
    const pcol = document.createElement("p");
    pcol.innerText = "Columns";
    this.colsDiv.appendChild(pcol);
    // ...
}

```

Create `treeWalker` function to traverse matrix data structure:

```

public update(options: VisualUpdateOptions) {
    // ...
    const treeWalker = (matrixNode: DataViewMatrixNode, index: number, levels: DataViewHierarchyLevel[], div: HTMLDivElement) => {

    }
    // ...
}

```

Where `matrixNode` is the current node, `levels` is metadata columns of this hierarchy level, `div` - parent element for child HTML elements.

The `treeWalker` is recursive function, need to create `div` element and `p` for text as header, and call the function for child elements of node:

```

public update(options: VisualUpdateOptions) {
    // ...
    const treeWalker = (matrixNode: DataViewMatrixNode, index: number, levels: DataViewHierarchyLevel[], div: HTMLDivElement) => {
        // ...

        if (matrixNode.children) {
            const childDiv = document.createElement("div");
            childDiv.classList.add("vertical");
            div.appendChild(childDiv);

            const p = document.createElement("p");
            const level = levels[matrixNode.level]; // get current level column metadata from current node
            p.innerText = level.sources[level.sources.length - 1].displayName; // get column name from metadata

            childDiv.appendChild(p); // add paragraph element to div element
            matrixNode.children.forEach((node, index) => treeWalker(node, levels, childDiv, ++levelIndex));
        }
    }
    // ...
}

```

Call the function for root elements of column and row of matrix data view structure:

```

public update(options: VisualUpdateOptions) {
    // ...
    const treeWalker = (matrixNode: DataViewMatrixNode, index: number, levels: DataViewHierarchyLevel[], div: HTMLDivElement) => {
        // ...
    }
    // ...
    // remove old elements
    // ...

    // ...
    const rowRoot: DataViewMatrixNode = matrixDataView.rows.root;
    rowRoot.children.forEach((node) => treeWalker(node, matrixDataView.rows.levels, this.rowsDiv));

    const colRoot = matrixDataView.columns.root;
    colRoot.children.forEach((node) => treeWalker(node, matrixDataView.columns.levels, this.colsDiv));
}

```

Generate selectionID for nodes and Create buttons to display nodes:

```

public update(options: VisualUpdateOptions) {
    // ...
    const treeWalker = (matrixNode: DataViewMatrixNode, index: number, levels: DataViewHierarchyLevel[], div: HTMLDivElement) => {
        const selectionID: ISelectionID = this.host.createSelectionIdBuilder()
            .withMatrixNode(matrixNode, levels)
            .createSelectionId();

        let nodeBlock = document.createElement("button");
        nodeBlock.innerText = matrixNode.value.toString();

        nodeBlock.addEventListener("click", (event) => {
            // call select method in the selection manager
            this.selectionManager.select(selectionID);
        });

        nodeBlock.addEventListener("contextmenu", (event) => {
            // call showContextMenu method to display context menu on the visual
            this.selectionManager.showContextMenu(selectionID, {
                x: event.clientX,
                y: event.clientY
            });
            event.preventDefault();
        });
        // ...
    }
    // ...
}

```

The main step of using highlighting is to process additional array of values.

If you inspect the object of terminal node, you can see that the values array has two properties - value and highlight:

```
JSON.stringify(options.dataViews[0].matrix.rows.root.children[0].children[0].children[0], null, " ")
```

```
{
    "level": 2,
    "levelValues": [
        {
            "value": "R233",
            "levelSourceIndex": 0
        }
    ],
    "value": "R233",
    "identity": {
        "identityIndex": 2
    },
    "values": {
        "0": {
            "value": null,
            "highlight": null
        },
        "1": {
            "value": 19,
            "highlight": 19
        }
    }
}
```

Where `value` property represents value of node without applying a selection from other visual, and `highlight` property indicates which part of data was highlighted.

## NOTE

Value of `highlight` property can be less than value of `value` property. It means that value was highlighted partially.

Add the code to process the `values` array of node if it is presented:

```
public update(options: VisualUpdateOptions) {
    // ...
    const treeWalker = (matrixNode: DataViewMatrixNode, index: number, levels: DataViewHierarchyLevel[], div: HTMLDivElement) => {
        // ...

        if (matrixNode.values) {
            const sumOfValues = Object.keys(matrixNode.values) // get key property of object (value are 0 to N)
                .map(key => +matrixNode.values[key].value) // convert key property to number
                .reduce((prev, curr) => prev + curr) // sum of values

            let sumOfHighlights = sumOfValues;
            sumOfHighlights = Object.keys(matrixNode.values) // get key property of object (value are 0 to N)
                .map(key => matrixNode.values[key].highlight ? +matrixNode.values[key].highlight : null) // convert key property to number if it exists
                .reduce((prev, curr) => curr ? prev + curr : null) // convert key property to number

            // create div container for value and highlighted value
            const vals = document.createElement("div");
            vals.classList.add("vertical")
            vals.classList.replace("vertical", "horizontal");
            // create paragraph element for label
            const highlighted = document.createElement("p");
            // Display complete value and highlighted value
            highlighted.innerText = `${sumOfHighlights}/${sumOfValues}`;

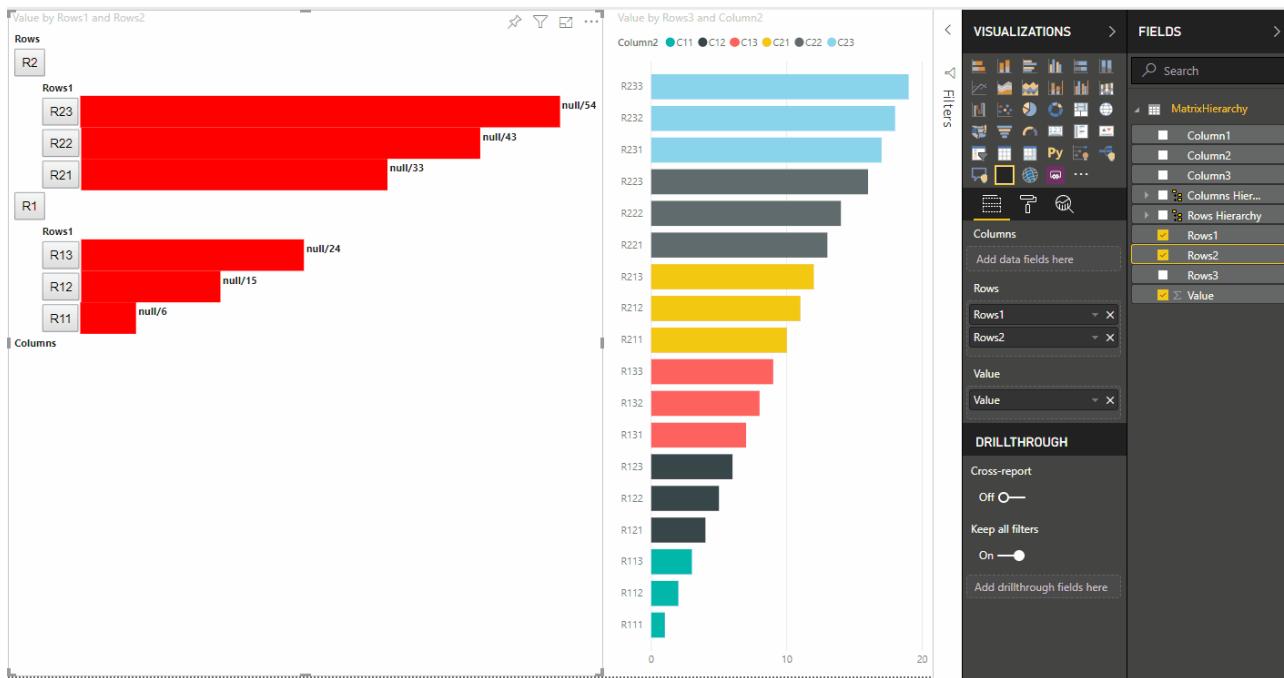
            // create div container for value
            const valueDiv = document.createElement("div");
            valueDiv.style.width = sumOfValues * 10 + "px";
            valueDiv.classList.add("value");

            // create div container for highlighted values
            const highlightsDiv = document.createElement("div");
            highlightsDiv.style.width = sumOfHighlights * 10 + "px";
            highlightsDiv.classList.add("highlight");
            valueDiv.appendChild(highlightsDiv);

            // append button and paragraph to div containers to parent div
            vals.appendChild(nodeBlock);
            vals.appendChild(valueDiv);
            vals.appendChild(highlighted);
            div.appendChild(vals);
        } else {
            div.appendChild(nodeBlock);
        }

        if (matrixNode.children) {
            // ...
        }
    }
    // ...
}
```

As the result you'll get the visual with buttons and values `highlighted value/default value`



## Next steps

- [Read about matrix data view mappings](#)
- [Read about capabilities of the visual](#)

# High-contrast mode support in Power BI visuals

3/19/2020 • 2 minutes to read • [Edit Online](#)

The Windows *high contrast* setting makes text and apps easier to see by displaying more distinct colors. This article discusses how to add high-contrast mode support to Power BI visuals. For more information, see [high-contrast support in Power BI](#).

To view an implementation of high-contrast support, go to the [PowerBI-visuals-sampleBarChart visual repository](#).

## On initialization

The `colorPalette` member of `options.host` has several properties for high-contrast mode. Use these properties to determine whether high-contrast mode is active and, if it is, what colors to use.

### Detect that Power BI is in high-contrast mode

If `host.colorPalette.isHighContrast` is `true`, high-contrast mode is active and the visual should draw itself accordingly.

### Get high-contrast colors

In high-contrast mode, your visual should limit itself to the following settings:

- **Foreground** color is used to draw any lines, icons, text, and outline or fill of shapes.
- **Background** color is used for background, and as the fill color of outlined shapes.
- **Foreground - selected** color is used to indicate a selected or active element.
- **Hyperlink** color is used only for hyperlink text.

#### NOTE

If a secondary color is needed, foreground color may be used with some opacity (Power BI native visuals use 40% opacity). Use this sparingly to keep the visual details easy to see.

During initialization, you can store the following values:

```
private isHighContrast: boolean;

private foregroundColor: string;
private backgroundColor: string;
private foregroundSelectedColor: string;
private hyperlinkColor: string;
//...

constructor(options: VisualConstructorOptions) {
    this.host = options.host;
    let colorPalette: ISandboxExtendedColorPalette = host.colorPalette;
    //...
    this.isHighContrast = colorPalette.isHighContrast;
    if (this.isHighContrast) {
        this.foregroundColor = colorPalette.foreground.value;
        this.backgroundColor = colorPalette.background.value;
        this.foregroundSelectedColor = colorPalette.foregroundSelected.value;
        this.hyperlinkColor = colorPalette.hyperlink.value;
    }
}
```

Or you can store the `host` object during initialization and access the relevant `colorPalette` properties during update.

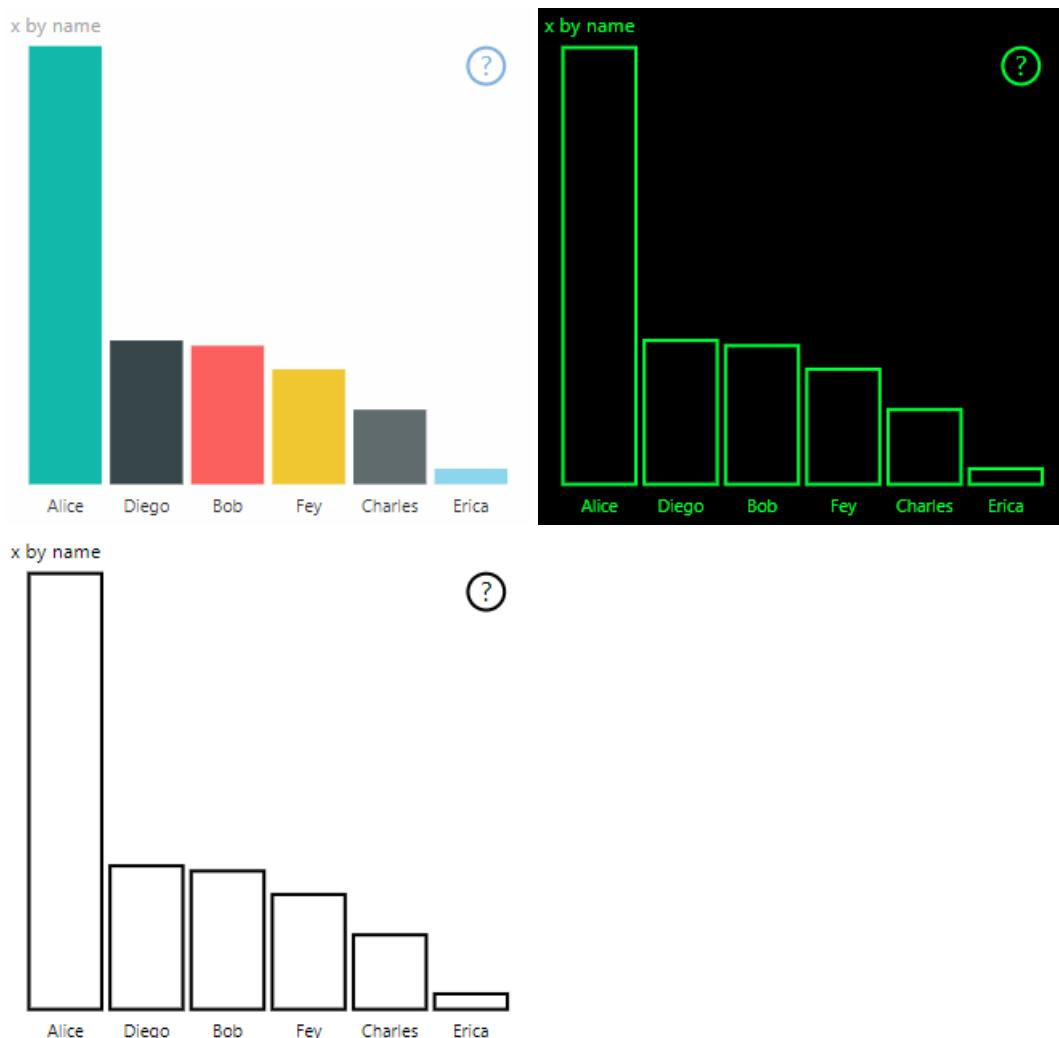
## On update

The specific implementations of high-contrast support vary from visual to visual and depend on the details of the graphic design. To keep important details easy to distinguish with the limited colors, high-contrast mode ordinarily requires a design that's slightly different from the default mode.

Power BI native visuals follow these guidelines:

- All data points use the same color (foreground).
- All text, axes, arrows, lines, and so on use the foreground color.
- Thick shapes are drawn as outlines, with thick strokes (at least two pixels) and background color fill.
- When data points are relevant, they're distinguished by different marker shapes, and data lines are distinguished by different dashing.
- When a data element is highlighted, all other elements change their opacity to 40%.
- For slicers, active filter elements use foreground-selected color.

In the following sample bar chart, for example, all bars are drawn with two pixels of thick foreground outline and background fill. Compare the way it looks with default colors and with a couple of high-contrast themes:



The next section shows one place in the `visualTransform` function that was changed to support high contrast. It's called as part of rendering during the update.

### Before

```

for (let i = 0, len = Math.max(category.values.length, dataValue.values.length); i < len; i++) {
    let defaultColor: Fill = {
        solid: {
            color: colorPalette.getColor(category.values[i] + '').value
        }
    };

    barChartDataPoints.push({
        category: category.values[i] + '',
        value: dataValue.values[i],
        color: getCategoricalObjectValue<Fill>(category, i, 'colorSelector', 'fill', defaultColor).solid.color,
        selectionId: host.createSelectionIdBuilder()
            .withCategory(category, i)
            .createSelectionId()
    });
}

```

## After

```

for (let i = 0, len = Math.max(category.values.length, dataValue.values.length); i < len; i++) {
    const color: string = getColumnColorByIndex(category, i, colorPalette);

    const selectionId: ISelectionId = host.createSelectionIdBuilder()
        .withCategory(category, i)
        .createSelectionId();

    barChartDataPoints.push({
        color,
        strokeColor,
        strokeWidth,
        selectionId,
        value: dataValue.values[i],
        category: `${category.values[i]} `,
    });
}

//...

function getColumnColorByIndex(
    category: DataViewCategoryColumn,
    index: number,
    colorPalette: ISandboxExtendedColorPalette,
): string {
    if (colorPalette.isHighContrast) {
        return colorPalette.background.value;
    }

    const defaultColor: Fill = {
        solid: {
            color: colorPalette.getColor(` ${category.values[index]} `).value,
        }
    };

    return getCategoricalObjectValue<Fill>(category, index, 'colorSelector', 'fill', defaultColor).solid.color;
}

```

# Sync slicers in Power BI visuals

3/19/2020 • 2 minutes to read • [Edit Online](#)

To support the **Sync Slicers** feature, your custom slicer visual must use API version 1.13 or later.

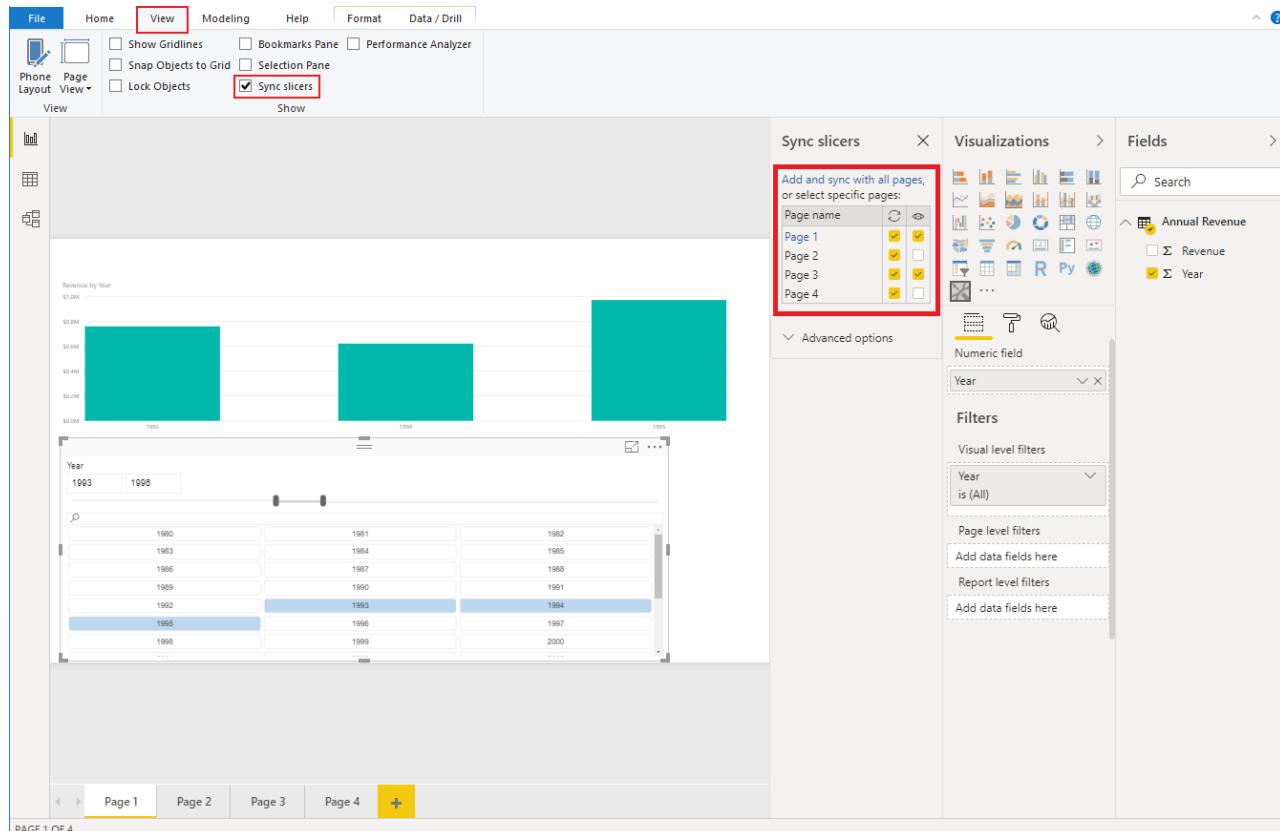
Additionally, you need to enable the option in the *capabilities.json* file, as shown in the following code:

```
{  
    ...  
    "supportsHighlight": true,  
    "suppressDefaultTitle": true,  
    "supportsSynchronizingFilterState": true,  
    "sorting": {  
        "default": {}  
    }  
}
```

After you've updated the *capabilities.json* file, you can view the **Sync slicers** options pane when you select your custom slicer visual.

## NOTE

The Sync Slicers feature doesn't support more than one field. If your slicer has more than one field (**Category** or **Measure**), the feature is disabled.



In the **Sync slicers** pane, you can see that your slicer visibility and its filtration can be applied to several report pages.

# How to debug Power BI visuals

4/20/2020 • 2 minutes to read • [Edit Online](#)

This page shows some tips for debugging while building your visual. It includes basic steps and shows differences between standard frontend applications and Power BI visual's debugging. After reading the article you will be able to debug Power BI visuals using breakpoints, log exceptions, and catch exceptions in Chrome and Edge.

## Using breakpoints

As the visual's JavaScript is entirely reloaded every time the visual is updated, any breakpoints you add will be lost when the debug visual is refreshed. As a workaround, use `debugger` statements in your code. It's recommended to turn off auto reload while using `debugger` in your code.

```
public update(options: VisualUpdateOptions) {
    console.log('Visual update', options);
    debugger;
    this.target.innerHTML = `<p>Update count: <em>${(this.updateCount}</em></p>`;
}
```

## Showing exceptions

When working on your visual, you'll notice that all errors are 'consumed' by the Power BI service. This is an intentional feature of Power BI to prevent misbehaving visuals from causing the entire app to become unstable.

As a workaround, add code to catch and log your exceptions, or set your debugger to break on caught exceptions.

## Log exceptions

To log exceptions in your Power BI visual, add the following code to your visual, to define an exception logging decorator.

```
export function logExceptions(): MethodDecorator {
    return function (target: Object, propertyKey: string, descriptor: TypedPropertyDescriptor<any>):
        TypedPropertyDescriptor<any> {
        return {
            value: function () {
                try {
                    return descriptor.value.apply(this, arguments);
                } catch (e) {
                    console.error(e);
                    throw e;
                }
            }
        }
    }
}
```

Then, you can use this decorator on any function to see error logging.

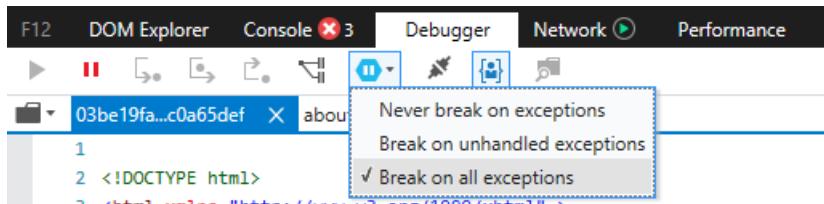
```
@logExceptions()
public update(options: VisualUpdateOptions) {
```

## Break on exceptions

You can also set the browser to break on caught exceptions. This'll stop code execution wherever an error happens, and allow you to debug from there.

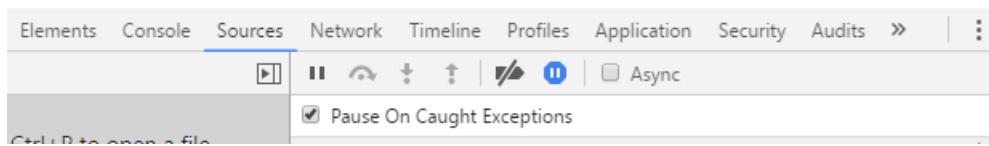
### Edge

1. Open developer tools (F12).
2. Go to the **Debugger** tab.
3. Click the **break on exceptions** icon (hexagon with a pause symbol).
4. Select **Break on all exceptions**.



### Chrome

1. Open developer tools (F12).
2. Go to the **Sources** tab.
3. Click the **break on exceptions** icon (stop sign with a pause symbol).
4. Select the **Pause On Caught Exceptions** check box.



## Next steps

- [Troubleshoot Power BI visuals](#)
- For more information and answers to questions, visit [Frequently asked questions about Power BI visuals](#)

# How to create mobile-friendly Power BI visuals

4/28/2020 • 2 minutes to read • [Edit Online](#)

Mobile consumption has a major role in Power BI. One of its strengths is staying connected to your data anytime, anywhere.

As a developer creating Power BI visuals, the unique constraints of each mobile device must be addressed to reach as many users as possible, and provide the best mobile experience.

Use the [Power BI app for Windows, iOS, and Android](#) to give your business users a comprehensive view of their data on the go, at the touch of their fingertips.

## Requirements

The following requirements are essential for mobile friendly visual development:

- Render

Your Power BI visual has to render on all supported mobile devices, including supported browsers and applications, with no errors in reports, dashboards, or when running in **Focus** mode.

- Interactive

Interactive functionality must be provided in the same way as it's provided for desktop devices. All events handled on desktop browsers must be supported, or have comparable event handlers for mobile devices.

For example, basic navigation and the selection of data points, should have the same functionality as in desktop browsers. If a visual supports multi-select using **Ctrl**, the developer needs to consider adding a similar event handler for mobile devices.

The following table provides a list of corresponding events on mobile devices.

MOUSE EVENT NAME	TOUCH EVENT NAME
click	click
mousemove	touchmove
mousedown	touchstart
mouseup	touchend
dblclick	use external library
contextmenu	use external library
mouseover	touchmove
mouseout	touchmove (or external library)
wheel	Nan

#### NOTE

Not all mobile or touch screen devices support mouse (or *mouse* prefixed) events. In such cases, handle both *mouse* and *touch* events at the same time.

## Optional

The following are considered optional and used to create a better end-user experience.

- Render

To support smaller visual sizes, try adding format options that the user can change to adjust the size of each element. For example, add format options to labels, for use in reports and dashboards. This allows users to customize a visual specifically for their mobile device.

The same settings can also be applied to the visuals in desktop browsers, and if needed, be overridden to adapt the visual to smaller screens.

#### NOTE

To optimize a visual in **Focus** mode, both portrait and landscape screen size orientations need consideration, see [Display content in Focus mode](#).

- Interactive

Consider the addition of mobile specific event handlers, such as dragging and scrolling.

- Failover

A visual should show a descriptive error if it cannot render on the mobile device.

## Supported browsers and devices

The Power BI visual must render on all devices supporting Power BI Apps, for more information see [supported browsers for Power BI](#) and [Power BI mobile apps](#).

When testing against the latest models of Windows, iOS, and Android devices, the developer needs to consider most of these quality aspects.

## Next steps

To get started, see [Tutorial: Developing a Power BI visual](#).

# Troubleshoot Power BI visuals

3/19/2020 • 2 minutes to read • [Edit Online](#)

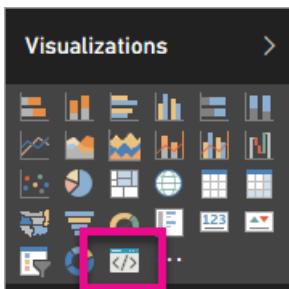
## Debug

### Pbiviz command not found (or similar errors)

When you run `pbiviz` in your terminal's command line, you should see the help screen. If not, then it is not installed correctly. Make sure you have at least the 4.0 version of NodeJS installed.

### Can't find the debug visual in the Visualizations tab

The debug visual looks like a prompt icon within the **Visualizations** tab.



If you don't see it, make sure you have enabled it within the Power BI settings.

#### NOTE

The debug visual is currently only available in the Power BI service and not in Power BI Desktop or the mobile app. The packaged visual will still work everywhere.

### Can't contact visual server

Run the visual server with the command `pbiviz start` in your terminal's command line from the root of your visual project. If the server is not running, it is likely that your SSL certificates weren't installed correctly.

Feel free to contact the Power BI visuals support team([pbicvsupport@microsoft.com](mailto:pbicvsupport@microsoft.com))with any questions, comments, or issues you have.

## Next steps

For more information, visit [Frequently asked questions about Power BI visuals](#).

# Power BI visuals API changelog

5/11/2020 • 2 minutes to read • [Edit Online](#)

This page contains a quick summary of the API versions. Versions listed here are considered stable and will not change.

## API v2.6

- Adds `isInFocus` to update option and `switchFocusModeState` method to visual host
- Supports `subtotals` customization

## API v2.5

- Supports [Analytics Pane](#)
- Supports `SelectionIdBuilder` `withMatrixNode` and `withTable` methods
- No longer supports `DataRepetitionSelector` interface, replaced with `data.CustomVisualOpaqueIdentity` interface

## API v2.3

- [Landing Page API](#)
- [Local Storage API](#)
- [Tuple filter API \(multi-column filter\)](#)
- [Rendering Events API](#)

## API v2.2

- Supports [restoring JSON Filter from DataView](#)
- [ContextMenu API](#)

## API v2.1

- Performance enhancements:
  - Faster load times
  - Smaller memory footprint
  - Optimized data and event transactions

### Release notes

- Refactored filtering APIs will be available in API 2.2 and are not supported in API 2.1.
- Visuals will only receive the `dataView` type that was declared in their capabilities. Visuals that used multiple `dataView` types will break as a result of this update.
- No longer supports `DataViewScopeIdentity` interface, replaced with `data.DataRepetitionSelector` interface. If you used key property of the `DataViewScopeIdentity` interface, you can replace it with `JSON.stringify(identity)`
- `undefined` is replaced with `null` inside the `dataView`. When iterating over an array using `var item in myArray` it skips on `undefined`, but doesn't skip on `null`. Visuals that use this pattern may be broken by this update. Make sure to check for `null` in arrays:

```
for (var item in myArray) {  
    if (!item) {  
        continue;  
    }  
    console.log(item);  
}
```

- The `proto` property no longer stores hidden metadata\data inside the dataView. Visuals that access properties via `proto` may be broken by this update.

## API v1.13

- Supports [Sync Slicers](#), note this only works for single field slicers due to PBI current code state, [read more](#).
- Accessibility: [High-contrast support](#)
- Accessibility: Allow Keyboard Focus flag

## API v1.12

- Supports Themes
- Supports [fetchMoreData](#), note the **Fetch More Data API** overcomes the hard limit of 30K data points
- [Canvas Tooltips API](#)

## API v1.11

- [FilterManager API](#)
- Supports [Bookmarks](#)

## API v1.10

- Adds `ILocalizationManager`
- [Authentication API](#)

## API v1.9

- [launchUrl API](#)

## API v1.8

- Supports new type `fillRule` (gradient) in capabilities schema
- Supports `rule` property in capabilities schema for object properties

## API v1.7

- Supports [RESJSON](#)

## API v1.6.2

- Supports [Edit mode](#) for visual to enter in-visual edit mode
- Supports [Interactive \(html\) R Power BI visuals](#), based on html

## API v1.5.0

- Supports [Allow interactions](#) for visual interactivity

## API v1.4.0

- Supports [Localization](#)

## API v1.3.0

- Supports [Tooltips](#)

## API v1.2.0

- Adds **colorPalette** to manage the colors used on your visual.
- Supports **Multiple selection** - selectionManager can accept an array of `SelectionId`.
- Supports [R visuals](#) using R scripts

## API v1.1.0

- Supports debug visual in iFrame
- Adds light weight sandbox with faster initialization of the iFrame
- Fixes [Capabilities.objects does not support "text" type](#) issue
- Supports `pbviz update` to update visual API type definitions and schema
- Supports `--api-version` flag in `pbviz new` to create visuals with a specific api version
- Supports alpha release of API v1.2.0

### Visual Host

- Adds `createSelectionIdBuilder` to create unique identifiers used for data selection
- Adds `createSelectionManager` to manage the selection state of the visual and communicates changes to the visual host
- Adds an array of default `colors` to use in visuals

## API v1.0.0

- Initial API release

# Add a landing page to your Power BI visuals

3/13/2020 • 2 minutes to read • [Edit Online](#)

With API 2.3.0, you can add a landing page to your Power BI visuals. To do so, add `supportsLandingPage` to the capabilities, and set it to true. This action initializes and updates your visual before you add data to it. Because the visual no longer shows a watermark, you can design your own landing page to be displayed in the visual as long as it has no data.

```
export class BarChart implements IVisual {
    //...
    private element: HTMLElement;
    private isLandingPageOn: boolean;
    private LandingPageRemoved: boolean;
    private LandingPage: d3.Selection<any>;

    constructor(options: VisualConstructorOptions) {
        //...
        this.element = options.element;
        //...
    }

    public update(options: VisualUpdateOptions) {
        //...
        this.HandleLandingPage(options);
    }

    private HandleLandingPage(options: VisualUpdateOptions) {
        if(!options.dataViews || !options.dataViews.length) {
            if(!this.isLandingPageOn) {
                this.isLandingPageOn = true;
                const SampleLandingPage: Element = this.createSampleLandingPage(); //create a landing page
                this.element.appendChild(SampleLandingPage);
                this.LandingPage = d3.select(SampleLandingPage);
            }
        } else {
            if(this.isLandingPageOn && !this.LandingPageRemoved){
                this.LandingPageRemoved = true;
                this.LandingPage.remove();
            }
        }
    }
}
```

An example landing page is shown in the following image:



...



## Sample Bar Chart Landing Page

[Learn more about Landing.page](#)

# Create a launch URL

3/13/2020 • 2 minutes to read • [Edit Online](#)

By creating a launch URL, you can open a new browser tab (or window) by delegating the actual work to Power BI.

## IMPORTANT

The `host.launchUrl()` was introduced in Visuals API 1.9.0.

## Sample

Import `IVisualHost` interface and save link to `host` object in the constructor of the visual.

```
import powerbi from "powerbi-visuals-api";
import IVisualHost = powerbi.extensibility.visual.IVisualHost;

export class Visual implements IVisual {
    private host: IVisualHost;
    // ...
    constructor(options: VisualConstructorOptions) {
        // ...
        this.host = options.host;
        // ...
    }
    // ...
}
```

## Usage

Use the `host.launchUrl()` API call, passing your destination URL as a string argument:

```
this.host.launchUrl('https://some.link.net');
```

## Restrictions

- Use only absolute paths, not relative paths. For example, use an absolute path such as `https://some.link.net/subfolder/page.html`. The relative path, `/page.html`, won't be opened.
- Currently, only *HTTP* and *HTTPS* protocols are supported. Avoid *FTP*, *MAILTO*, and so on.

## Best practices

- Usually, it's best to open a link only as a response to a user's explicit action. Make it easy for the user to understand that clicking the link or button will result in opening a new tab. Triggering a `launchUrl()` call without a user's action, or as a side effect of a different action can be confusing or frustrating for the user.
- If the link isn't essential for the proper functioning of the visual, we recommend that you give the report's author a way to disable and hide the link. This recommendation is especially relevant for special Power BI use cases, such as embedding a report in a third-party application or publishing it to the web.

- Avoid triggering a `launchUrl()` call from inside a loop, the visual's `update` function, or any other frequently recurring code.

## A step-by-step example

### Add a link-launching element

The following lines were added to the visual's `constructor` function:

```
this.helpLinkElement = this.createHelpLinkElement();
options.element.appendChild(this.helpLinkElement);
```

A private function that creates and attaches the anchor element was added:

```
private createHelpLinkElement(): Element {
    let linkElement = document.createElement("a");
    linkElement.textContent = "?";
    linkElement.setAttribute("title", "Open documentation");
    linkElement.setAttribute("class", "helpLink");
    linkElement.addEventListener("click", () => {
        this.host.launchUrl("https://docs.microsoft.com/power-bi/developer/visuals/custom-visual-develop-tutorial");
    });
    return linkElement;
};
```

Finally, an entry in the `visual.less` file defines the style for the link element:

```
.helpLink {
    position: absolute;
    top: 0px;
    right: 12px;
    display: block;
    width: 20px;
    height: 20px;
    border: 2px solid #80B0E0;
    border-radius: 20px;
    color: #80B0E0;
    text-align: center;
    font-size: 16px;
    line-height: 20px;
    background-color: #FFFFFF;
    transition: all 900ms ease;

    &:hover {
        background-color: #DDEEFF;
        color: #5080B0;
        border-color: #5080B0;
        transition: all 250ms ease;
    }

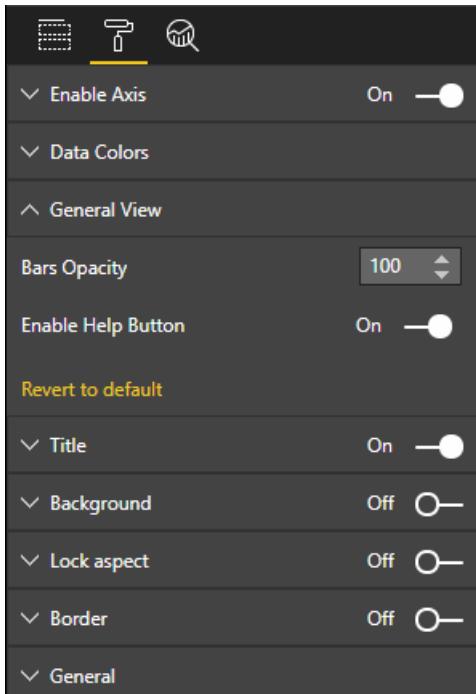
    &.hidden {
        display: none;
    }
}
```

### Add a toggling mechanism

To add a toggling mechanism, you need to add a static object so that the report's author can toggle the visibility of the link element. (The default is set to `hidden`.) For more information, see the [static object tutorial](#).

A `showHelpLink` Boolean static object was added to the `capabilities.json` file's objects entry, as shown in the following code:

```
"objects": {  
    "generalView": {  
        "displayName": "General View",  
        "properties": {  
            "showHelpLink": {  
                "displayName": "Show Help Button",  
                "type": {  
                    "bool": true  
                }  
            }  
        }  
    }  
}
```



And, in the visual's `update` function, the following lines were added:

```
if (settings.generalView.showHelpLink) {  
    this.helpLinkElement.classList.remove("hidden");  
} else {  
    this.helpLinkElement.classList.add("hidden");  
}
```

The `hidden` class is defined in the `visual.less` file to control the display of the element.

# Visual API

5/13/2020 • 3 minutes to read • [Edit Online](#)

All visuals start with a class that implements the `IVisual` interface. You can name the class anything as long as there's exactly one class that implements the `IVisual` interface.

## NOTE

The visual class name must match what's defined in the `pbviz.json` file.

See the sample visual class with the following methods that should be implemented:

- `constructor`, a standard constructor to initialize the visual's state
- `update`, to update the visual's data
- `enumerateObjectInstances`, to return objects to populate the property pane (formatting options) where you can modify them as needed
- `destroy`, a standard destructor for cleanup

```
class MyVisual implements IVisual {

    constructor(options: VisualConstructorOptions) {
        //one time setup code goes here (called once)
    }

    public update(options: VisualUpdateOptions): void {
        //code to update your visual goes here (called on all view or data changes)
    }

    public enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions):
    VisualObjectInstanceEnumeration {
        //returns objects to populate the property pane (called for each object defined in capabilities)
    }

    public destroy(): void {
        //one time cleanup code goes here (called once)
    }
}
```

## constructor

The constructor of the visual class is called when the visual is instantiated. It can be used for any set up operations needed by the visual.

```
constructor(options: VisualConstructorOptions)
```

## VisualConstructorOptions

- `element: HTMLElement`, a reference to the DOM element that will contain your visual
- `host: IVisualHost`, a collection of properties and services that can be used to interact with the visual host (Power BI)

`IWebHost` contains the following services:

```
export interface IWebHost extends extensibility.IWebHost {
    createSelectionIdBuilder: () => visuals.ISelectionIdBuilder;
    : () => ISelectionManager;
    colorPalette: ISandboxExtendedColorPalette;
    persistProperties: (changes: VisualObjectInstancesToPersist) => void;
    applyJsonFilter: (filter: IFilter[] | IFilter, objectName: string, propertyName: string, action: FilterAction) => void;
    tooltipService: ITooltipService;
    telemetry: ITlemetryService;
    authenticationService: IAuthenticationService;
    locale: string;
    allowInteractions: boolean;
    launchUrl: (url: string) => void;
    fetchMoreData: () => boolean;
    instanceId: string;
    refreshHostData: () => void;
    createLocalizationManager: () => ILocalizationManager;
    storageService: ILocalVisualStorageService;
    eventService: IVisualEventService;
    switchFocusModeState: (on: boolean) => void;
}
```

- `createSelectionIdBuilder`, generates and stores metadata for selectable items in your visual
- `createSelectionManager`, creates the communication bridge used to notify the visual's host on changes in the selection state, see [Selection API](#).
- `createLocalizationManager`, generates a manager to help with [Localization](#)
- `allowInteractions: boolean`, a boolean flag which hints whether or not the visual should be interactive
- `applyJsonFilter`, applies specific filter types, see [Filter API](#)
- `persistProperties`, allows users to persist properties and save them along with the visual definition, so they're available on the next reload
- `eventService`, returns an [event service](#) to support [Render](#) events
- `storageService`, returns a service to help use [local storage](#) in the visual
- `authenticationService`, generates a service to help with user authentication
- `tooltipService`, returns a [tooltip service](#) to help use tooltips in the visual
- `launchUrl`, helps to [launch URL](#) in next tab
- `locale`, returns a locale string, see [Localization](#)
- `instanceId`, returns a string to identify the current visual instance
- `colorPalette`, returns the colorPalette required to apply colors to your data
- `fetchMoreData`, supports using more data than the standard limit (1K rows)
- `switchFocusModeState`, helps to change the focus mode state

## update

All visuals must implement a public update method that's called whenever there's a change in the data or host environment.

```
public update(options: VisualUpdateOptions): void
```

### VisualUpdateOptions

- `viewport: IViewport`, dimensions of the viewport that the visual should be rendered within
- `dataViews: DataView[]`, the dataview object which contains all data needed to render your visual

will typically use the categorical property under DataView)

- `type: VisualUpdateType`, flags to indicate the type(s) of this update (Data | Resize | ViewMode | Style | ResizeEnd)
- `viewMode: ViewMode`, flags to indicate the view mode of the visual (View | Edit | InFocusEdit)
- `editMode:EditMode`, flag to indicate the edit mode of the visual (Default | Advanced) (if the visual supports AdvancedEditMode, it should render its advanced UI controls only when editMode is set to Advanced, see AdvancedEditMode)
- `operationKind?: VisualDataChangeOperationKind`, flag to indicate type of data change (Create | Append)
- `jsonFilters?: IFilter[]`, collection of applied json filters
- `isInFocus?: boolean`, flag to indicate if the visual is in focus mode or not

## enumerateObjectInstances optional

This method is called for every object listed in the capabilities. Using the options (currently just the name) you return a `VisualObjectInstanceEnumeration` with information about how to display this property.

```
enumerateObjectInstances(options:EnumerateVisualObjectInstancesOptions):VisualObjectInstanceEnumeration
```

## EnumerateVisualObjectInstancesOptions

- `objectName: string`, name of the object

## destroy optional

The destroy function is called when your visual is unloaded and can be used for clean up tasks such as removing event listeners.

```
public destroy(): void
```

### NOTE

Power BI generally doesn't call `destroy` since it's faster to remove the entire IFrame that contains the visual.

# Local Storage API

3/13/2020 • 2 minutes to read • [Edit Online](#)

The Local Storage API is an API a custom visual can use to request the host to save or load data from the device's storage. It's isolated in the sense that there's separation of storage access between different visual types.

## Sample

If the custom visual should increase some counter every time the update method is called, but the counter value should also be preserved and not reset on every visual start:

```
export class Visual implements IVisual {
    // ...
    private updateCountName: string = 'updateCount';
    private updateCount: number;
    private storage: ILocalVisualStorageService;
    // ...

    constructor(options: VisualConstructorOptions) {
        // ...
        this.storage = options.host.storageService;
        // ...

        this.storage.get(this.updateCountName).then(count =>
        {
            this.updateCount = +count;
        })
        .catch(() =>
        {
            this.updateCount = 0;
            this.storage.set(this.updateCountName, this.updateCount.toString());
        });
        // ...
    }

    public update(options: VisualUpdateOptions) {
        // ...
        this.updateCount++;
        this.storage.set(this.updateCountName, this.updateCount.toString());
        // ...
    }
}
```

## Known limitations and issues

Local Storage API isn't activated for Power BI visuals by default. If you want to activate it for your Power BI visual, send a request to Power BI visuals Support [pbicvsupport@microsoft.com](mailto:pbicvsupport@microsoft.com).

Please note that your visual should be available in [AppSource](#) and be certified.

# Render events in Power BI visuals

3/13/2020 • 2 minutes to read • [Edit Online](#)

The new API consists of three methods (`started`, `finished`, or `failed`) that should be called during rendering.

When rendering starts, the Power BI visual code calls the `renderingStarted` method to indicate that the rendering process has started.

If rendering is completed successfully, the Power BI visual code immediately calls the `renderingFinished` method, notifying the listeners (primarily, *export to PDF* and *export to PowerPoint*) that the visual's image is ready for export.

If a problem occurs during the process, the Power BI visual is prevented from being rendered successfully. To notify the listeners that the rendering process hasn't been completed, the Power BI visual code should call the `renderingFailed` method. This method also provides an optional string to provide a reason for the failure.

## Usage

```
export interface IVisualHost extends extensibility.IVisualHost {
    eventService: IVisualEventService ;
}

/**
 * An interface for reporting rendering events
 */
export interface IVisualEventService {
    /**
     * Should be called just before the actual rendering starts,
     * usually at the start of the update method
     *
     * @param options - the visual update options received as an update parameter
     */
    renderingStarted(options: VisualUpdateOptions): void;

    /**
     * Should be called immediately after rendering finishes successfully
     *
     * @param options - the visual update options received as an update parameter
     */
    renderingFinished(options: VisualUpdateOptions): void;

    /**
     * Called when rendering fails, with an optional reason string
     *
     * @param options - the visual update options received as an update parameter
     * @param reason - the optional failure reason string
     */
    renderingFailed(options: VisualUpdateOptions, reason?: string): void;
}
```

**Sample: The visual displays no animations**

```

export class Visual implements IVisual {
    ...
    private events: IVisualEventService;
    ...

    constructor(options: VisualConstructorOptions) {
        ...
        this.events = options.host.eventService;
        ...
    }

    public update(options: VisualUpdateOptions) {
        this.events.renderingStarted(options);
        ...
        this.events.renderingFinished(options);
    }
}

```

### Sample: The visual displays animations

If the visual has animations or async functions for rendering, the `renderingFinished` method should be called after the animation or inside async function.

```

export class Visual implements IVisual {
    ...
    private events: IVisualEventService;
    private element: HTMLElement;
    ...

    constructor(options: VisualConstructorOptions) {
        ...
        this.events = options.host.eventService;
        this.element = options.element;
        ...
    }

    public update(options: VisualUpdateOptions) {
        this.events.renderingStarted(options);
        ...
        // Learn more at https://github.com/d3/d3-transition/blob/master/README.md#transition\_end
        d3.select(this.element).transition().duration(100).style("opacity","0").end().then(() => {
            // renderingFinished called after transition end
            this.events.renderingFinished(options);
        });
    }
}

```

## Rendering events for visual certification

One requirement of visuals certification is the support of rendering events by the visual. For more information, see [certification requirements](#).

# Sorting options for Power BI visuals

3/13/2020 • 2 minutes to read • [Edit Online](#)

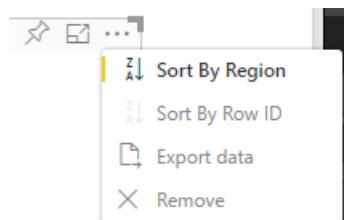
This article describes how *sorting* options specify the sorting behavior for Power BI visuals.

The sorting capability requires one of the following parameters.

## Default sorting

The `default` option is the simplest form. It allows sorting the data presented in the 'DataMappings' section. The option enables sorting of the data mappings by the user and specifies the sorting direction.

```
"sorting": {  
    "default": {}  
}
```



## Implicit sorting

Implicit sorting is sorting with the array parameter `clauses`, which describes sorting for each data role. `implicit` means that the visual's user can't change the sorting order. Power BI doesn't display sorting options in the visual's menu. However, Power BI does sort data according to specified settings.

`clauses` parameters can contain several objects with two parameters:

- `role` : Determines `DataMapping` for sorting
- `direction` : Determines sort direction (1 = Ascending, 2 = Descending)

```
"sorting": {  
    "implicit": {  
        "clauses": [  
            {  
                "role": "category",  
                "direction": 1  
            },  
            {  
                "role": "measure",  
                "direction": 2  
            }  
        ]  
    }  
}
```

## Custom sorting

Custom sorting means that the sorting is managed by the developer in the visual's code.

# Add the locale in Power BI for Power BI visuals

5/13/2020 • 3 minutes to read • [Edit Online](#)

Visuals can retrieve the Power BI locale to localize their content to the relevant language.

Read more about [Supported languages and countries/regions for Power BI](#)

For example, getting locale in the Sample Bar Chart visual.



Each of these bar charts was created under a different locale (English, Basque, and Hindi), and it's displayed in the tooltip.

## NOTE

The localization manager in the visual's code is supported from API 1.10.0 and higher.

## Get the locale

The `locale` is passed as a string during the initialization of the visual. If a locale is changed in Power BI, the visual will be generated again with the new locale. You can find the full sample code at [SampleBarChart with Locale](#).

The BarChart constructor now has a `locale` member, which is instantiated in the constructor with the host locale instance.

```
private locale: string;  
...  
this.locale = options.host.locale;
```

Supported locales:

LOCALE STRING	LANGUAGE
ar-SA	العربية (Arabic)
bg-BG	български (Bulgarian)
ca-ES	català (Catalan)

LOCALE STRING	LANGUAGE
cs-CZ	čeština (Czech)
da-DK	dansk (Danish)
de-DE	Deutsche (German)
el-GR	ελληνικά (Greek)
en-US	English (English)
es-ES	español service (Spanish)
et-EE	eesti (Estonian)
eU-ES	Euskal (Basque)
fi-FI	suomi (Finnish)
fr-FR	français (French)
gl-ES	galego (Galician)
he-IL	עברית (Hebrew)
hi-IN	हिन्दी (Hindi)
hr-HR	hrvatski (Croatian)
hu-HU	magyar (Hungarian)
id-ID	Bahasa Indonesia (Indonesian)
it-IT	italiano (Italian)
ja-JP	日本の (Japanese)
kk-KZ	Қазақ (Kazakh)
ko-KR	한국의 (Korean)
lt-LT	Lietuvos (Lithuanian)
lv-LV	Latvijas (Latvian)
ms-MY	Bahasa Melayu (Malay)
nb-NO	norsk (Norwegian)
nl-NL	Nederlands (Dutch)

LOCALE STRING	LANGUAGE
pl-PL	polski (Polish)
pt-BR	português (Portuguese)
pt-PT	português (Portuguese)
ro-RO	românesc (Romanian)
ru-RU	русский (Russian)
sk-SK	slovenský (Slovak)
sl-SI	slovenski (Slovenian)
sr-Cyrl-RS	српски (Serbian)
sr-Latn-RS	srpski (Serbian)
sv-SE	svenska (Swedish)
th-TH	ไทย (Thai)
tr-TR	Türk (Turkish)
uk-UA	український (Ukrainian)
vi-VN	tiếng Việt (Vietnamese)
zh-CN	中国 (Chinese-Simplified)
zh-TW	中國 (Chinese-Traditional)

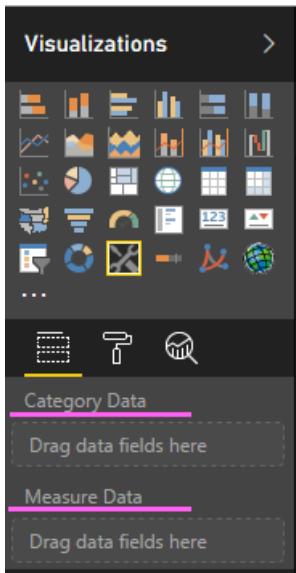
#### NOTE

In the PowerBI Desktop the locale property will contain the language of the PowerBI Desktop installed.

## Localizing the property pane for Power BI visuals

Fields in the property pane can be localized to provide more integrated and coherent experience. It makes your custom visual behave like any other Power BI core visual.

For example, a non-localized custom visual created by using the `pbviz new` command, will show the following fields in the property pane:



both the Category Data and the Measure Data are defined in the capabilities.json file as `displayName`.

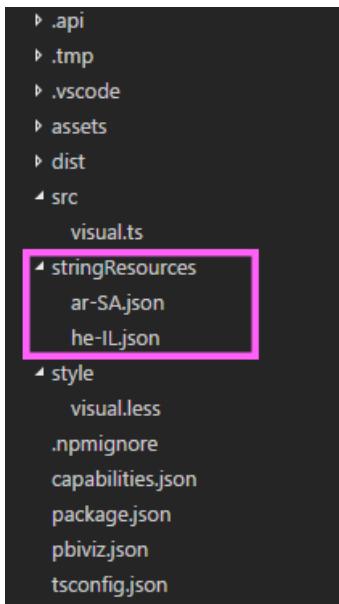
## How to localize capabilities

First add a display name key to every display name you want to localize in your capabilities. In this example:

```
{
  "dataRoles": [
    {
      "displayName": "Category Data",
      "displayNameKey": "VisualCategoryDataNameKey1",
      "name": "category",
      "kind": "Grouping"
    },
    {
      "displayName": "Measure Data",
      "displayNameKey": "VisualMeasureDataNameKey2",
      "name": "measure",
      "kind": "Measure"
    }
  ]
}
```

Then add a directory called `stringResources`. The directory will contain all your different string resource files based on the locales you want your visual to support. Under this directory, you'll need to add a JSON file for every locale you want to support. Those files contain the locale information and the localized strings values for every `displayNameKey` you want to replace.

In our example, lets say we want to support Arabic and Hebrew. We will need to add two JSON files in the following way:



Every JSON file defines a single locale (this file has to be one of the locales from the supported list above), with the string values for the desired display name keys. In our example the Hebrew string resource file will look as follows:

```
{  
  "locale": "he-IL",  
  "values": {  
    "VisualCategoryDataNameKey1": "קטגוריה",  
    "VisualMeasureDataNameKey2": "ýchidot midah"  
  }  
}
```

All the required steps to use the localization manager are described below.

#### NOTE

Currently, localization is not supported for debugging the dev visual

## Setup environment

### Desktop

For desktop usage, download the localized version of Power BI desktop from <https://powerbi.microsoft.com>.

### Web service

If you use the web client (browser) in the service, then change your language in settings:

The screenshot shows the Power BI settings interface. On the left is a sidebar with navigation links: General, Dashboards, Datasets, Workbooks, Alerts, and Subscriptions. The 'General' tab is selected. On the right, under 'Language Settings', it says 'Select the language for Power BI. The language you select will appear in the interface and parts of the visuals.' A dropdown menu shows 'Русский' (Russian) as the selected language. Below the dropdown are 'Apply' and 'Discard' buttons.

## Resource file

Add a resources.resjson file to a folder named as the locale you're going to use inside of the stringResources folder. It is en-US and ru-RU in our example.

```
stringResources
  en-US
    resources.resjson
  ru-RU
    resources.resjson
```

After that, add all the localization strings you are going to use into the resources.resjson file you've added in the previous step.

```
{
  ...
  "Role_Legend": "Обозначения",
  "Role_task": "Задача",
  "Role_StartDate": "Дата начала",
  "Role_Duration": "Длительность"
  ...
}
```

This sample is the en-US version of resources.resjson file:

```
{
  ...
  "Role_Legend": "Legend",
  "Role_task": "Task",
  "Role_StartDate": "Start date",
  "Role_Duration": "Duration"
  ...
}
```

New localizationManager instance Create an instance of localizationManager in your visual's code as follows

```
private localizationManager: ILocalizationManager;

constructor(options: VisualConstructorOptions) {
  this.localizationManager = options.host.createLocalizationManager();
}
```

## localizationManager usage sample

Now you can call the `getDisplayName` function of the localization manager with the string key argument you defined in `resources.resjson` to get the required string anywhere inside of your code:

```
let legend: string = this.localization.getDisplayName("Role_Legend");
```

It returns "Legend" for en-US and "Обозначения" for ru-RU

## Next steps

- [Read how to use formatting utils to provide localized formats](#)

# Adding external libraries

3/19/2020 • 2 minutes to read • [Edit Online](#)

This article describes how to use external libraries in Power BI visuals. It includes information about installing, importing, and calling external libraries from the Power BI visual's code.

## JavaScript libraries

1. Install an external JavaScript library by using any package manager such as *npm* or *yarn*.
2. Import the required modules into the source files using the external library.

### NOTE

If you'd like to add typings to your JavaScript library, and get intellisense and compile-time safety, make sure that you install the appropriate package.

### Installing the *d3* library

This is an example of installing the [d3 library](#) and the [@types/d3](#) package, using [npm](#).

For a full example, see the [Power BI visualizations](#) code.

1. Install the *d3* package and the *d3 types* package.

```
npm install d3@5 --save
npm install @types/d3@5 --save
```

2. Import the *d3* library in the files that use it, such as `visual.ts`.

```
import * as d3 from "d3";
```

## CSS framework

1. Install an external CSS framework by using any package manager such as *npm* or *yarn*.
2. In the `.less` file of the visual, include the `import` statement.

### Installing bootstrap

This is an example of installing [bootstrap](#) using [npm](#).

For a full example, see the [Power BI visualizations](#) code.

1. Install the *bootstrap* package.

```
npm install bootstrap --save
```

2. Include the `import` statement in `visual.less`.

```
@import (less) "node_modules/bootstrap/dist/css/bootstrap.css";
```

# Power BI visuals interactivity utils

3/13/2020 • 4 minutes to read • [Edit Online](#)

Interactivity utils (`InteractivityUtils`) is a set of functions and classes that can be used to simplify the implementation of cross-selection and cross-filtering.

## NOTE

The new updates of interactivity utils support only the latest version of tools (3.x.x and above).

## Installation

1. To install the package, run the following command in the directory with your current Power BI visual project.

```
npm install powerbi-visuals-utils-interactivityutils --save
```

2. If you're using version 3.0 or later or the tool, install `powerbi-models` to resolve dependencies.

```
npm install powerbi-models --save
```

3. To use interactivity utils, import the required component in the source code of the Power BI visual.

```
import { interactivitySelectionService } from "powerbi-visuals-utils-interactivityutils";
```

## Including the CSS files

To use the package with your Power BI visual, import the following CSS file to your `.less` file.

```
node_modules/powerbi-visuals-utils-interactivityutils/lib/index.css
```

Import the CSS file as a `.less` file because the Power BI visuals tool wraps external CSS rules.

```
@import (less) "node_modules/powerbi-visuals-utils-interactivityutils/lib/index.css";
```

## SelectableDataPoint properties

Usually, data points contain selections and values. The interface extends the `SelectableDataPoint` interface.

`SelectableDataPoint` already contains properties as described below.

```

/** Flag for identifying that a data point was selected */
selected: boolean;

/** Identity for identifying the selectable data point for selection purposes */
identity: powerbi.extensibility.ISelectionId;

/*
 * A specific identity for when data points exist at a finer granularity than
 * selection is performed. For example, if your data points select based
 * only on series, even if they exist as category/series intersections.
 */

specificIdentity?: powerbi.extensibility.ISelectionId;

```

## Defining an interface for data points

1. Create an instance of interactivity utils and save the object as a property of the visual

```

export class Visual implements IVisual {
    // ...
    private interactivity: interactivityBaseService.IInteractivityService<VisualDataPoint>;
    // ...
    constructor(options: VisualConstructorOptions) {
        // ...
        this.interactivity = interactivitySelectionService.createInteractivitySelectionService(this.host);
        // ...
    }
}

```

```

import { interactivitySelectionService } from "powerbi-visuals-utils-interactivityutils";

export interface VisualDataPoint extends interactivitySelectionService.SelectableDataPoint {
    value: powerbi.PrimitiveValue;
}

```

2. Extend the base behavior class.

**NOTE**

`BaseBehavior` was introduced in the [5.6.x version of interactivity utils](#). If you use an older version, create a behavior class from the sample below.

3. Define the interface for the behavior class options.

```

import { SelectableDataPoint } from "./interactivitySelectionService";

import {
  IBehaviorOptions,
  BaseDataPoint
} from "./interactivityBaseService";

export interface BaseBehaviorOptions<SelectableDataPointType extends BaseDataPoint> extends
IBehaviorOptions<SelectableDataPointType> {

  /** d3 selection object of the main elements on the chart */
  elementsSelection: Selection<any, SelectableDataPoint, any, any>;

  /** d3 selection object of some elements on backgroup, to hadle click of reset selection */
  clearCatcherSelection: d3.Selection<any, any, any, any>;
}

```

4. Define a class for `visual behavior`. Or, extend the `BaseBehavior` class.

#### Defining a class for `visual behavior`

The class is responsible to handle `click` `contextmenu` mouse events.

When a user clicks on data elements, the visual calls the selection handler to select data points. if the user clicks on the background element of the visual, it calls the clear selection handler.

The class has the following correspond methods:

- `bindClick`
- `bindClearCatcher`
- `bindContextMenu`.

```

export class Behavior<SelectableDataPointType extends BaseDataPoint> implements IInteractiveBehavior {

  /** d3 selection object of main elements in the chart */
  protected options: BaseBehaviorOptions<SelectableDataPointType>;
  protected selectionHandler: ISelectionHandler;

  protected bindClick() {
    // ...
  }

  protected bindClearCatcher() {
    // ...
  }

  protected bindContextMenu() {
    // ...
  }

  public bindEvents(
    options: BaseBehaviorOptions<SelectableDataPointType>,
    selectionHandler: ISelectionHandler): void {
    // ...
  }

  public renderSelection(hasSelection: boolean): void {
    // ...
  }
}

```

#### Extending the `BaseBehavior` class

```

import powerbi from "powerbi-visuals-api";
import { interactivitySelectionService, baseBehavior } from "powerbi-visuals-utils-interactivityutils";

export interface VisualDataPoint extends interactivitySelectionService.SelectableDataPoint {
    value: powerbi.PrimitiveValue;
}

export class Behavior extends baseBehavior.BaseBehavior<VisualDataPoint> {
    // ...
}

```

5. To handle click on elements, call the `d3` selection object `on` method. This also applies for `elementsSelection` and `clearCatcherSelection`.

```

protected bindClick() {
    const {
        elementsSelection
    } = this.options;

    elementsSelection.on("click", (datum) => {
        const mouseEvent: MouseEvent = getEvent() as MouseEvent || window.event as MouseEvent;
        mouseEvent && this.selectionHandler.handleSelection(
            datum,
            mouseEvent.ctrlKey);
    });
}

```

6. Add a similar handler for the `contextmenu` event, to call the selection manager's `showContextMenu` method.

```

protected bindContextMenu() {
    const {
        elementsSelection
    } = this.options;

    elementsSelection.on("contextmenu", (datum) => {
        const event: MouseEvent = (getEvent() as MouseEvent) || window.event as MouseEvent;
        if (event) {
            this.selectionHandler.showContextMenu(
                datum,
                {
                    x: event.clientX,
                    y: event.clientY
                });
            event.preventDefault();
        }
    });
}

```

7. To assign functions to handlers, the interactivity utils calls the `bindEvents` method. Add the following calls to the `bindEvents` method:

- `bindClick`
- `bindClearCatcher`
- `bindContextMenu`

```

public bindEvents(
    options: BaseBehaviorOptions<SelectableDataPointType>,
    selectionHandler: ISelectionHandler): void {

    this.options = options;
    this.selectionHandler = selectionHandler;

    this.bindClick();
    this.bindClearCatcher();
    this.bindContextMenu();
}

```

8. The `renderSelection` method is responsible for updating the visual state of elements in the chart. Here's a sample implementation of `renderSelection`.

```

public renderSelection(hasSelection: boolean): void {
    this.options.elementsSelection.style("opacity", (category: any) => {
        if (category.selected) {
            return 0.5;
        } else {
            return 1;
        }
    });
}

```

9. The last step is creating an instance of `visual behavior`, and calling the `bind` method of the interactivity utils instance.

```

this.interactivity.bind(<BaseBehaviorOptions<VisualDataPoint>>{
    behavior: this.behavior,
    dataPoints: this.categories,
    clearCatcherSelection: select(this.target),
    elementsSelection: selectionMerge
});

```

- `selectionMerge` is the *d3* selection object, which represents all the visual's selectable elements.
- `select(this.target)` is the *d3* selection object, which represents the visual's main DOM elements.
- `this.categories` are data points with elements, where the interface is `visualDataPoint` or `categories: VisualDataPoint[];`.
- `this.behavior` is a new instance of `visual behavior` created in the constructor of the visual, as shown below.

```

export class Visual implements IVisual {
    // ...
    constructor(options: VisualConstructorOptions) {
        // ...
        this.behavior = new Behavior();
    }
    // ...
}

```

## Next steps

- [Read how to handle selections on bookmarks switching](#)

- [Read how to add context menu for visuals data points](#)
- [Read how to use selection manager to add selections into Power BI Visuals](#)

# Formatting utils

3/13/2020 • 6 minutes to read • [Edit Online](#)

Formatting utils contains the classes, interfaces, and methods to format values. It also contains extender methods to process strings, and measure text size in SVG/HTML document.

## Text measurement service

The module provides the following functions and interfaces:

### TextProperties interface

This interface describes common properties of the text.

```
interface TextProperties {  
    text?: string;  
    fontFamily: string;  
    fontSize: string;  
    fontWeight?: string;  
    fontStyle?: string;  
    fontVariant?: string;  
    whiteSpace?: string;  
}
```

### measureSvgTextWidth

This function measures the width of the text with the given SVG text properties.

```
function measureSvgTextWidth(textProperties: TextProperties, text?: string): number;
```

Example of using `measureSvgTextWidth`:

```
import { textMeasurementService } from "powerbi-visuals-utils-formattingutils";  
import TextProperties = textMeasurementService.TextProperties;  
// ...  
  
let textProperties: TextProperties = {  
    text: "Microsoft PowerBI",  
    fontFamily: "sans-serif",  
    fontSize: "24px"  
};  
  
textMeasurementService.measureSvgTextWidth(textProperties);  
  
// returns: 194.71875
```

### measureSvgTextRect

This function returns a rect with the given SVG text properties.

```
function measureSvgTextRect(textProperties: TextProperties, text?: string): SVGRect;
```

Example of using `measureSvgTextRect`:

```

import { textMeasurementService } from "powerbi-visuals-utils-formattingutils";
import TextProperties = textMeasurementService.TextProperties;
// ...

let textProperties: TextProperties = {
    text: "Microsoft PowerBI",
    fontFamily: "sans-serif",
    fontSize: "24px"
};

textMeasurementService.measureSvgTextRect(textProperties);

// returns: { x: 0, y: -22, width: 194.71875, height: 27 }

```

## measureSvgTextHeight

This function measures the height of the text with the given SVG text properties.

```
function measureSvgTextHeight(textProperties: TextProperties, text?: string): number;
```

Example of using `measureSvgTextHeight`:

```

import { textMeasurementService } from "powerbi-visuals-utils-formattingutils";
import TextProperties = textMeasurementService.TextProperties;
// ...

let textProperties: TextProperties = {
    text: "Microsoft PowerBI",
    fontFamily: "sans-serif",
    fontSize: "24px"
};

textMeasurementService.measureSvgTextHeight(textProperties);

// returns: 27

```

## estimateSvgTextBaselineDelta

This function returns a baseline of the given SVG text properties.

```
function estimateSvgTextBaselineDelta(textProperties: TextProperties): number;
```

Example:

```

import { textMeasurementService } from "powerbi-visuals-utils-formattingutils";
import TextProperties = textMeasurementService.TextProperties;
// ...

let textProperties: TextProperties = {
    text: "Microsoft PowerBI",
    fontFamily: "sans-serif",
    fontSize: "24px"
};

textMeasurementService.estimateSvgTextBaselineDelta(textProperties);

// returns: 5

```

## estimateSvgTextHeight

This function estimates the height of the text with the given SVG text properties.

```
function estimateSvgTextHeight(textProperties: TextProperties, tightFitForNumeric?: boolean): number;
```

Example of using `estimateSvgTextHeight`:

```
import { textMeasurementService } from "powerbi-visuals-utils-formattingutils";
import TextProperties = textMeasurementService.TextProperties;
// ...

let textProperties: TextProperties = {
    text: "Microsoft PowerBI",
    fontFamily: "sans-serif",
    fontSize: "24px"
};

textMeasurementService.estimateSvgTextHeight(textProperties);

// returns: 27
```

You can take a look at the example code of the custom visual [here](#).

### **measureSvgTextElementWidth**

This function measures the width of the svgElement.

```
function measureSvgTextElementWidth(svgElement: SVGTextElement): number;
```

Example of using `measureSvgTextElementWidth`:

```
import { textMeasurementService } from "powerbi-visuals-utils-formattingutils";
// ...

let svg: D3.Selection = d3.select("body").append("svg");

svg.append("text")
    .text("Microsoft PowerBI")
    .attr({
        "x": 25,
        "y": 25
    })
    .style({
        "font-family": "sans-serif",
        "font-size": "24px"
    });

let textElement: D3.Selection = svg.select("text");

textMeasurementService.measureSvgTextElementWidth(textElement.node());

// returns: 194.71875
```

### **getMeasurementProperties**

This function fetches the text measurement properties of the given DOM element.

```
function getMeasurementProperties(element: Element): TextProperties;
```

Example of using `getMeasurementProperties`:

```

import { textMeasurementService } from "powerbi-visuals-utils-formattingutils";
// ...

let element: JQuery = $(document.createElement("div"));

element.text("Microsoft PowerBI");

element.css({
    "width": 500,
    "height": 500,
    "font-family": "sans-serif",
    "font-size": "32em",
    "font-weight": "bold",
    "font-style": "italic",
    "white-space": "nowrap"
});

textMeasurementService.getMeasurementProperties(element.get(0));

/* returns: {
    fontFamily:"sans-serif",
    fontSize: "32em",
    fontStyle: "italic",
    fontVariant: "",
    fontWeight: "bold",
    text: "Microsoft PowerBI",
    whiteSpace: "nowrap"
}*/

```

## getSvgMeasurementProperties

This function fetches the text measurement properties of the given SVG text element.

```
function getSvgMeasurementProperties(svgElement: SVGTextElement): TextProperties;
```

Example of using `getSvgMeasurementProperties`:

```

import { textMeasurementService } from "powerbi-visuals-utils-formattingutils";
// ...

let svg: D3.Selection = d3.select("body").append("svg");

let textElement: D3.Selection = svg.append("text")
    .text("Microsoft PowerBI")
    .attr({
        "x": 25,
        "y": 25
    })
    .style({
        "font-family": "sans-serif",
        "font-size": "24px"
    });
}

textMeasurementService.getSvgMeasurementProperties(textElement.node());

/* returns: {
    "text": "Microsoft PowerBI",
    "fontFamily": "sans-serif",
    "fontSize": "24px",
    "fontWeight": "normal",
    "fontStyle": "normal",
    "fontVariant": "normal",
    "whiteSpace": "nowrap"
}*/

```

## getDivElementWidth

This function returns the width of a div element.

```
function getDivElementWidth(element: JQuery): string;
```

Example of using `getDivElementWidth`:

```

import { textMeasurementService } from "powerbi-visuals-utils-formattingutils";
// ...

let svg: Element = d3.select("body")
    .append("div")
    .style({
        "width": "150px",
        "height": "150px"
    })
    .node();

textMeasurementService.getDivElementWidth(svg)

// returns: 150px

```

## getTailoredTextOrDefault

Compares labels text size to the available size and renders ellipses when the available size is smaller.

```
function getTailoredTextOrDefault(textProperties: TextProperties, maxWidth: number): string;
```

Example of using `getTailoredTextOrDefault`:

```
import { textMeasurementService } from "powerbi-visuals-utils-formattingutils";
import TextProperties = textMeasurementService.TextProperties;
// ...

let textProperties: TextProperties = {
    text: "Microsoft PowerBI!",
    fontFamily: "sans-serif",
    fontSize: "24px"
};

textMeasurementService.getTailoredTextOrDefault(textProperties, 100);

// returns: Micros...
```

## String extensions

The module provides the following functions:

### endsWith

This function checks if a string ends with a substring.

```
function endsWith(str: string, suffix: string): boolean;
```

Example of using `endsWith`:

```
import { stringExtensions } from "powerbi-visuals-utils-formattingutils";
// ...

stringExtensions.endsWith("Power BI", "BI");

// returns: true
```

### equalIgnoreCase

This function compares strings, ignoring case.

```
function equalIgnoreCase(a: string, b: string): boolean;
```

Example of using `equalIgnoreCase`:

```
import { stringExtensions } from "powerbi-visuals-utils-formattingutils";
// ...

stringExtensions.equalIgnoreCase("Power BI", "power bi");

// returns: true
```

### startsWith

This function checks if a string starts with a substring;

```
function startsWith(a: string, b: string): boolean;
```

Example of using `startsWith`:

```
import { stringExtensions } from "powerbi-visuals-utils-formattingutils";
// ...

stringExtensions.startsWith("Power BI", "Power");

// returns: true
```

## contains

This function checks if a string contains a specified substring.

```
function contains(source: string, substring: string): boolean;
```

Example of using `contains` method:

```
import { stringExtensions } from "powerbi-visuals-utils-formattingutils";
// ...

stringExtensions.contains("Microsoft Power BI Visuals", "Power BI");

// returns: true
```

## isNullOrEmpty

Checks if a string is null or undefined or empty.

```
function isNullOrEmpty(value: string): boolean;
```

Example of `isNullOrEmpty` method:

```
import { stringExtensions } from "powerbi-visuals-utils-formattingutils";
// ...

stringExtensions.isNullOrEmpty(null);

// returns: true
```

# Value formatter

The module provides the following functions, interfaces, and classes:

## IValueFormatter

This interface describes public methods and properties of the formatter.

```
interface IValueFormatter {
    format(value: any): string;
    displayUnit?: DisplayUnit;
    options?: ValueFormatterOptions;
}
```

## IValueFormatter.format

This method formats the given value.

```
function format(value: any, format?: string, allowFormatBeautification?: boolean): string;
```

Examples for `IValueFormatter.format`:

#### The thousand formats

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({ value: 1001 });

iValueFormatter.format(5678);

// returns: "5.68K"
```

#### The million formats

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({ value: 1e6 });

iValueFormatter.format(1234567890);

// returns: "1234.57M"
```

#### The billion formats

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({ value: 1e9 });

iValueFormatter.format(1234567891236);

// returns: 1234.57bn
```

#### The trillion format

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({ value: 1e12 });

iValueFormatter.format(1234567891236);

// returns: 1.23T
```

#### The exponent format

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({ format: "E" });

iValueFormatter.format(1234567891236);

// returns: 1.234568E+012
```

#### The culture selector

```

import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let valueFormatterUK = valueFormatter.create({ cultureSelector: "en-GB" });

valueFormatterUK.format(new Date(2007, 2, 3, 17, 42, 42));

// returns: 03/03/2007 17:42:42

let valueFormatterUSA = valueFormatter.create({ cultureSelector: "en-US" });

valueFormatterUSA.format(new Date(2007, 2, 3, 17, 42, 42));

// returns: 3/3/2007 5:42:42 PM

```

### The percentage format

```

import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({ format: "0.00 %;-0.00 %;0.00 %" });

iValueFormatter.format(0.54);

// returns: 54.00 %

```

### The dates format

```

import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let date = new Date(2016, 10, 28, 15, 36, 0),
    iValueFormatter = valueFormatter.create({});

iValueFormatter.format(date);

// returns: 11/28/2016 3:36:00 PM

```

### The boolean format

```

import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({});

iValueFormatter.format(true);

// returns: True

```

### The customized precision

```

import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({ value: 0, precision: 3 });

iValueFormatter.format(3.141592653589793);

// returns: 3.142

```

You can take a look at the example code of the custom visual [here](#).

## ValueFormatterOptions

This interface describes `options` of the `IValueFormatter` and options of 'create' function.

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";
import ValueFormatterOptions = valueFormatter.ValueFormatterOptions;

interface ValueFormatterOptions {
    /** The format string to use. */
    format?: string;
    /** The data value. */
    value?: any;
    /** The data value. */
    value2?: any;
    /** The number of ticks. */
    tickCount?: any;
    /** The display unit system to use */
    displayUnitSystemType?: DisplayUnitSystemType;
    /** True if we are formatting single values in isolation (e.g. card), as opposed to multiple values with a common base (e.g. chart axes) */
    formatSingleValues?: boolean;
    /** True if we want to trim off unnecessary zeroes after the decimal and remove a space before the % symbol */
    *
    allowFormatBeautification?: boolean;
    /** Specifies the maximum number of decimal places to show*/
    precision?: number;
    /** Detect axis precision based on value */
    detectAxisPrecision?: boolean;
    /** Specifies the column type of the data value */
    columnType?: ValueTypeDescriptor;
    /** Specifies the culture */
    cultureSelector?: string;
}
}
```

## create

This method creates an instance of IValueFormatter.

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";
import create = valueFormatter.create;

function create(options: ValueFormatterOptions): IValueFormatter;
```

### Example of using create

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

valueFormatter.create({});

// returns: an instance of IValueFormatter.
```

## Next steps

[Add localizations to a Power BI custom visual](#)

# DataViewUtils

3/13/2020 • 8 minutes to read • [Edit Online](#)

The `DataViewUtils` is a set of functions and classes to simplify parsing of the `DataView` object for Power BI visuals

## Installation

To install the package, you should run the following command in the directory with your current custom visual:

```
npm install powerbi-visuals-utils-dataviewutils --save
```

This command installs the package and adds a package as a dependency to your `package.json`

## DataRoleHelper

The `DataRoleHelper` provides functions to check roles of the `dataView` object.

The module provides the following functions:

### `getMeasureIndexOfRole`

This function finds the measure by role name and returns its index.

```
function getMeasureIndexOfRole(grouped: DataViewValueColumnGroup[], roleName: string): number;
```

Example:

```

import powerbi from "powerbi-visuals-api";
import DataViewValueColumnGroup = powerbi.DataViewValueColumnGroup;
import { dataRoleHelper } from "powerbi-visuals-utils-dataviewutils";
// ...

// This object is actually a part of the dataView object.
let columnGroup: DataViewValueColumnGroup[] = [{{
    values: [
        {
            source: {
                displayName: "Microsoft",
                roles: {
                    "company": true
                }
            },
            values: []
        },
        {
            source: {
                displayName: "Power BI",
                roles: {
                    "product": true
                }
            },
            values: []
        }
    ]
}];

dataRoleHelper.getMeasureIndexOfRole(columnGroup, "product");

// returns: 1

```

## getCategoryIndexOfRole

This function finds the category by role name and returns its index.

```
function getCategoryIndexOfRole(categories: DataViewCategoryColumn[], roleName: string): number;
```

Example:

```

import powerbi from "powerbi-visuals-api";
import DataViewCategoryColumn = powerbi.DataViewCategoryColumn;
import { dataRoleHelper } from "powerbi-visuals-utils-dataviewutils";
// ...

// This object is actually a part of the dataView object.
let categoryGroup: DataViewCategoryColumn[] = [
  {
    source: {
      displayName: "Microsoft",
      roles: {
        "company": true
      }
    },
    values: []
  },
  {
    source: {
      displayName: "Power BI",
      roles: {
        "product": true
      }
    },
    values: []
  }
];
dataRoleHelper.getCategoryIndexOfRole(categoryGroup, "product");

// returns: 1

```

## hasRole

This function checks if the provided role is defined in the metadata.

```
function hasRole(column: DataViewMetadataColumn, name: string): boolean;
```

Example:

```

import powerbi from "powerbi-visuals-api";
import DataViewMetadataColumn = powerbi.DataViewMetadataColumn;
import { dataRoleHelper } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let metadata: DataViewMetadataColumn = {
  displayName: "Microsoft",
  roles: {
    "company": true
  }
};

DataRoleHelper.hasRole(metadata, "company");

// returns: true

```

## hasRoleInDataView

This function checks if the provided role is defined in the dataView.

```
function hasRoleInDataView(dataView: DataView, name: string): boolean;
```

Example:

```

import powerbi from "powerbi-visuals-api";
import DataView = powerbi.DataView;
import { dataRoleHelper } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let dataView: DataView = {
    metadata: {
        columns: [
            {
                displayName: "Microsoft",
                roles: {
                    "company": true
                }
            },
            {
                displayName: "Power BI",
                roles: {
                    "product": true
                }
            }
        ]
    }
};

DataRoleHelper.hasRoleInDataView(dataView, "product");

// returns: true

```

### hasRoleInValueColumn

This function checks if the provided role is defined in the value column.

```
function hasRoleInValueColumn(valueColumn: DataViewValueColumn, name: string): boolean;
```

Example:

```

import powerbi from "powerbi-visuals-api";
import DataViewValueColumn = powerbi.DataViewValueColumn;
import { dataRoleHelper } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let valueColumn: DataViewValueColumn = {
    source: {
        displayName: "Microsoft",
        roles: {
            "company": true
        }
    },
    values: []
};

dataRoleHelper.hasRoleInValueColumn(valueColumn, "company");

// returns: true

```

## DataViewObjects

The `DataViewObjects` provides functions to extract values of the objects.

The module provides the following functions:

### getValue

This function returns the value of the given object.

```
function getValue<T>(objects: DataViewObjects, propertyId: DataViewObjectPropertyIdentifier, defaultValue?: T): T;
```

Example:

```
import powerbi from "powerbi-visuals-api";
import DataViewObjectPropertyIdentifier = powerbi.DataViewObjectPropertyIdentifier;
import { dataViewObjects } from "powerbi-visuals-utils-dataviewutils";

let property: DataViewObjectPropertyIdentifier = {
    objectName: "microsoft",
    propertyName: "bi"
};

// This object is actually a part of the dataView object.
let objects: powerbi.DataViewObjects = {
    "microsoft": {
        "windows": 5,
        "bi": "Power"
    }
};

dataViewObjects.getValue(objects, property);

// returns: Power
```

## getObject

This function returns an object of the given object.

```
function getObject(objects: DataViewObjects, objectName: string, defaultValue?: IDataViewObject): IDataViewObject;
```

Example:

```
import { dataViewObjects } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let objects: powerbi.DataViewObjects = {
    "microsoft": {
        "windows": 5,
        "bi": "Power"
    }
};

dataViewObjects.getObject(objects, "microsoft");

/* returns: {
    "bi": "Power",
    "windows": 5
}*/
```

## getFillColor

This function returns a solid color of the objects.

```
function getFillColor(objects: DataViewObjects, propertyId: DataViewObjectPropertyIdentifier, defaultColor?: string): string;
```

Example:

```
import powerbi from "powerbi-visuals-api";
import DataViewObjectPropertyIdentifier = powerbi.DataViewObjectPropertyIdentifier;
import { dataViewObjects } from "powerbi-visuals-utils-dataviewutils";

let property: DataViewObjectPropertyIdentifier = {
    objectName: "power",
    propertyName: "fillColor"
};

// This object is actually part of the dataView object.
let objects: powerbi.DataViewObjects = {
    "power": {
        "fillColor": {
            "solid": {
                "color": "yellow"
            }
        },
        "bi": "Power"
    }
};

dataViewObjects.getFillColor(objects, property);

// returns: yellow
```

## getCommonValue

This function is a universal function for retrieving the color or value of a given object.

```
function getCommonValue(objects: DataViewObjects, propertyId: DataViewObjectPropertyIdentifier, defaultValue?: any): any;
```

Example:

```

import powerbi from "powerbi-visuals-api";
import DataViewObjectPropertyIdentifier = powerbi.DataViewObjectPropertyIdentifier;
import { dataViewObjects } from "powerbi-visuals-utils-dataviewutils";

let colorProperty: DataViewObjectPropertyIdentifier = {
    objectName: "power",
    propertyName: "fillColor"
};

let biProperty: DataViewObjectPropertyIdentifier = {
    objectName: "power",
    propertyName: "bi"
};

// This object is actually part of the dataView object.
let objects: powerbi.DataViewObjects = {
    "power": {
        "fillColor": {
            "solid": {
                "color": "yellow"
            }
        },
        "bi": "Power"
    }
};

dataViewObjects.getCommonValue(objects, colorProperty); // returns: yellow
dataViewObjects.getCommonValue(objects, biProperty); // returns: Power

```

## DataViewObject

The `DataViewObject` provides functions to extract value of the object.

The module provides the following functions:

### **getValue**

This function returns a value of the object by property name.

```
function getValue<T>(object: IDataViewObject, propertyName: string, defaultValue?: T): T;
```

Example:

```

import { dataViewObject } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let object: powerbi.DataViewObject = {
    "windows": 5,
    "microsoft": "Power BI"
};

dataViewObject.getValue(object, "microsoft");

// returns: Power BI

```

### **getFillColorByPropertyName**

This function returns a solid color of the object by property name.

```
function getFillColorByPropertyName(object: IDataViewObject, propertyName: string, defaultColor?: string): string;
```

Example:

```
import { dataViewObject } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let object: powerbi.DataViewObject = {
    "windows": 5,
    "fillColor": {
        "solid": {
            "color": "green"
        }
    }
};

dataViewObject.getFillColorByPropertyName(object, "fillColor");

// returns: green
```

## converterHelper

The `converterHelper` provides functions to check properties of the dataView.

The module provides the following functions:

### categoryIsAlsoSeriesRole

This function checks if the category is also series.

```
function categoryIsAlsoSeriesRole(dataView: DataViewCategorical, seriesRoleName: string, categoryRoleName: string): boolean;
```

Example:

```
import powerbi from "powerbi-visuals-api";
import DataViewCategorical = powerbi.DataViewCategorical;
import { converterHelper } from "powerbi-visuals-utils-dataviewutils";
// ...

// This object is actually part of the dataView object.
let categorical: DataViewCategorical = {
    categories: [
        {
            source: {
                displayName: "Microsoft",
                roles: {
                    "power": true,
                    "bi": true
                }
            },
            values: []
        }
    ];
};

converterHelper.categoryIsAlsoSeriesRole(categorical, "power", "bi");

// returns: true
```

## getSeriesName

This function returns a name of the series.

```
function getSeriesName(source: DataViewMetadataColumn): PrimitiveValue;
```

Example:

```
import powerbi from "powerbi-visuals-api";
import DataViewMetadataColumn = powerbi.DataViewMetadataColumn;
import { converterHelper } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let metadata: DataViewMetadataColumn = {
    displayName: "Microsoft",
    roles: {
        "power": true,
        "bi": true
    },
    groupName: "Power BI"
};

converterHelper.getSeriesName(metadata);

// returns: Power BI
```

### **isImageUrlColumn**

This function checks if the column contains an image url.

```
function isImageUrlColumn(column: DataViewMetadataColumn): boolean;
```

Example:

```
import powerbi from "powerbi-visuals-api";
import DataViewMetadataColumn = powerbi.DataViewMetadataColumn;
import { converterHelper } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let metadata: DataViewMetadataColumn = {
    displayName: "Microsoft",
    type: {
        misc: {
            imageUrl: true
        }
    }
};

converterHelper.isImageUrlColumn(metadata);

// returns: true
```

### **isWebUrlColumn**

This function checks if the column contains a web url.

```
function isWebUrlColumn(column: DataViewMetadataColumn): boolean;
```

Example:

```

import powerbi from "powerbi-visuals-api";
import DataViewMetadataColumn = powerbi.DataViewMetadataColumn;
import { converterHelper } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let metadata: DataViewMetadataColumn = {
    displayName: "Microsoft",
    type: {
        misc: {
            webUrl: true
        }
    }
};

converterHelper.isWebUrlColumn(metadata);

// returns: true

```

## hasImageUrlColumn

This function checks if the dataView has a column with image url.

```
function hasImageUrlColumn(dataView: DataView): boolean;
```

Example:

```

import DataView = powerbi.DataView;
import converterHelper = powerbi.extensibility.utils.dataview.converterHelper;

// This object is actually part of the dataView object.
let dataView: DataView = {
    metadata: {
        columns: [
            {
                displayName: "Microsoft"
            },
            {
                displayName: "Power BI",
                type: {
                    misc: {
                        imageUrl: true
                    }
                }
            }
        ]
    }
};

converterHelper.hasImageUrlColumn(dataView);

// returns: true

```

## DataViewObjectsParser

The `DataViewObjectsParser` provides the simplest way to parse properties of the formatting panel.

The class provides the following methods:

### getDefault

This static method returns an instance of `DataViewObjectsParser`.

```
static getDefault(): DataViewObjectsParser;
```

Example:

```
import { dataViewObjectsParser } from "powerbi-visuals-utils-dataviewutils";
// ...

dataViewObjectsParser.getDefault();

// returns: an instance of the DataViewObjectsParser
```

## parse

This method parses properties of the formatting panel and returns an instance of `DataViewObjectsParser`.

```
static parse<T extends DataViewObjectsParser>(dataView: DataView): T;
```

Example:

```
import powerbi from "powerbi-visuals-api";
import IVisual = powerbi.extensibility.IVisual;
import VisualUpdateOptions = powerbi.extensibility.visual.VisualUpdateOptions;
import { dataViewObjectsParser } from "powerbi-visuals-utils-dataviewutils";

/**
 * This class describes formatting panel properties.
 * Name of the property should match its name described in the capabilities.
 */
class DataPointProperties {
    public fillColor: string = "red"; // This value is a default value of the property.
}

class PropertiesParser extends dataViewObjectsParser.DataViewObjectsParser {
    /**
     * This property describes a group of properties.
     */
    public dataPoint: DataPointProperties = new DataPointProperties();
}

export class YourVisual extends IVisual {
    // implementation of the IVisual.

    private propertiesParser: PropertiesParser;

    public update(options: VisualUpdateOptions): void {
        // Parses properties.
        this.propertiesParser = PropertiesParser.parse<PropertiesParser>(options.dataViews[0]);

        // You can use the properties after parsing
        console.log(this.propertiesParser.dataPoint.fillColor); // returns "red" as default value, it will be
        updated automatically after any change of the formatting panel.
    }
}
```

## enumerateObjectInstances

This static method enumerates properties and returns an instance of `VisualObjectInstanceEnumeration`.

Execute it in `enumerateObjectInstances` method of the visual.

```
static enumerateObjectInstances(dataViewObjectParser: dataViewObjectsParser.DataViewObjectsParser, options: EnumerateVisualObjectInstancesOptions): VisualObjectInstanceEnumeration;
```

Example:

```
import powerbi from "powerbi-visuals-api";
import IVisual = powerbi.extensibility.IVisual;
import EnumerateVisualObjectInstancesOptions = powerbi.EnumerateVisualObjectInstancesOptions;
import VisualObjectInstanceEnumeration = powerbi.VisualObjectInstanceEnumeration;
import VisualUpdateOptions = powerbi.extensibility.visual.VisualUpdateOptions;
import { dataViewObjectsParser } from "powerbi-visuals-utils-dataviewutils";

/**
 * This class describes formatting panel properties.
 * Name of the property should match its name described in the capabilities.
 */
class DataPointProperties {
    public fillColor: string = "red";
}

class PropertiesParser extends dataViewObjectsParser.DataViewObjectsParser {
    /**
     * This property describes a group of properties.
     */
    public dataPoint: DataPointProperties = new DataPointProperties();
}

export class YourVisual extends IVisual {
    // implementation of the IVisual.

    private propertiesParser: PropertiesParser;

    public update(options: VisualUpdateOptions): void {
        // Parses properties.
        this.propertiesParser = PropertiesParser.parse<PropertiesParser>(options.dataViews[0]);
    }

    /**
     * This method will be executed only if the formatting panel is open.
     */
    public enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions):
    VisualObjectInstanceEnumeration {
        return PropertiesParser.enumerateObjectInstances(this.propertiesParser, options);
    }
}
```

# Chart utils

3/13/2020 • 8 minutes to read • [Edit Online](#)

ChartUtils is a set of interfaces and methods for creating axis, data labels, and legends in Power BI Visuals.

## Installation

To install the package, you should run the following command in the directory with your current visual:

```
npm install powerbi-visuals-utils-chartutils --save
```

## Axis Helper

The axis helper (`axis` object in `utils`) provides functions to simplify manipulations with axis.

The module provides the following functions:

### **getRecommendedNumberOfTicksForXAxis**

This function returns recommended number of ticks according to width of chart.

```
function getRecommendedNumberOfTicksForXAxis(availableWidth: number): number;
```

Example:

```
import { axis } from "powerbi-visuals-utils-chartutils";
// ...
axis.getRecommendedNumberOfTicksForXAxis(1024);

// returns: 8
```

### **getRecommendedNumberOfTicksForYAxis**

This function returns recommended number of ticks according to height of chart.

```
function getRecommendedNumberOfTicksForYAxis(availableWidth: number);
```

Example:

```
import { axis } from "powerbi-visuals-utils-chartutils";
// ...
axis.getRecommendedNumberOfTicksForYAxis(100);

// returns: 3
```

### **getBestNumberOfTicks**

Gets the optimal number of ticks based on minimum value, maximum value, and measure metadata and max tick count;

```
function getBestNumberOfTicks(  
    min: number,  
    max: number,  
    valuesMetadata: DataViewMetadataColumn[],  
    maxTickCount: number,  
    isDateTime?: boolean  
) : number;
```

Example:

```
import { axis } from "powerbi-visuals-utils-chartutils";  
// ...  
var dataViewMetadataColumnWithIntegersOnly: powerbi.DataViewMetadataColumn[] = [  
    {  
        displayName: "col1",  
        isMeasure: true,  
        type: ValueType.fromDescriptor({ integer: true })  
    },  
    {  
        displayName: "col2",  
        isMeasure: true,  
        type: ValueType.fromDescriptor({ integer: true })  
    }  
];  
var actual = axis.getBestNumberOfTicks(  
    0,  
    3,  
    dataViewMetadataColumnWithIntegersOnly,  
    6  
);  
// returns: 4
```

## getTickLabelMargins

This function returns the margins for tick labels.

```
function getTickLabelMargins(  
    viewport: IViewport,  
    yMarginLimit: number,  
    textWidthMeasurer: ITextAsSVGMeasurer,  
    textHeightMeasurer: ITextAsSVGMeasurer,  
    axes: CartesianAxisProperties,  
    bottomMarginLimit: number,  
    properties: TextProperties,  
    scrollbarVisible?: boolean,  
    showOnRight?: boolean,  
    renderXAxis?: boolean,  
    renderY1Axis?: boolean,  
    renderY2Axis?: boolean  
) : TickLabelMargins;
```

Example:

```

import { axis } from "powerbi-visuals-utils-chartutils";
// ...

axis.getTickLabelMargins(
  plotArea,
  marginLimits.left,
  TextMeasurementService.measureSvgTextWidth,
  TextMeasurementService.estimateSvgTextHeight,
  axes,
  marginLimits.bottom,
  textProperties,
  /*scrolling*/ false,
  showY1OnRight,
  renderXAxis,
  renderY1Axis,
  renderY2Axis
);

// returns: xMax, yLeft, yRight, stackHeigh;

```

## isOrdinal

Checks if a string is null or undefined or empty.

```
function isOrdinal(type: ValueTypeDescriptor): boolean;
```

Example:

```

import { axis } from "powerbi-visuals-utils-chartutils";
// ...
let type = ValueType.fromDescriptor({ misc: { barcode: true } });
axis.isOrdinal(type);

// returns: true

```

## isDateTime

Checks if value is of DateTime type.

```
function isDateTime(type: ValueTypeDescriptor): boolean;
```

Example:

```

import { axis } from "powerbi-visuals-utils-chartutils";
// ...

axis.isDateTime(ValueType.fromDescriptor({ dateTime: true }));

// returns: true

```

## getCategoryThickness

Uses the D3 scale to get the actual category thickness.

```
function getCategoryThickness(scale: any): number;
```

Example:

```

import { axis } from "powerbi-visuals-utils-chartutils";
// ...

let range = [0, 100];
let domain = [0, 10];
let scale = d3.scale
  .linear()
  .domain(domain)
  .range(range);
let actualThickness = axis.getCategoryThickness(scale);

```

## **invertOrdinalScale**

This function inverts the ordinal scale. If  $x < \text{scale.range()[0]}$ , then  $\text{scale.domain()[0]}$  is returned. Otherwise, it returns the greatest item in  $\text{scale.domain()}$  that's  $\leq x$ .

```
function invertOrdinalScale(scale: d3.scale.Ordinal<any, any>, x: number);
```

Example:

```

import { axis } from "powerbi-visuals-utils-chartutils";
// ...

let domain: number[] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
  pixelSpan: number = 100,
  ordinalScale: d3.scale.ordinal = axis.createOrdinalScale(
    pixelSpan,
    domain,
    0.4
  );
axis.invertOrdinalScale(ordinalScale, 49);
// returns: 4

```

## **findClosestXAxisIndex**

This function finds and returns the closest x-axis index.

```
function findClosestXAxisIndex(
  categoryValue: number,
  categoryAxisValues: AxisHelperCategoryDataPoint[]
): number;
```

Example:

```

import { axis } from "powerbi-visuals-utils-chartutils";
// ...

/**
 * Finds the index of the category of the given x coordinate given.
 * pointX is in non-scaled screen-space, and offsetX is in render-space.
 * offsetX does not need any scaling adjustment.
 * @param {number} pointX The mouse coordinate in screen-space, without scaling applied
 * @param {number} offsetX Any left offset in d3.scale render-space
 * @return {number}
 */
private findIndex(pointX: number, offsetX?: number): number {
    // we are using mouse coordinates that do not know about any potential CSS transform scale
    let xScale = this.scaleDetector.getScale().x;
    if (!Double.equalWithPrecision(xScale, 1.0, 0.00001)) {
        pointX = pointX / xScale;
    }
    if (offsetX) {
        pointX += offsetX;
    }

    let index = axis.invertScale(this.xAxisProperties.scale, pointX);
    if (this.data.isScalar) {
        // When we have scalar data the inverted scale produces a category value, so we need to search for the
        closest index.
        index = axis.findClosestXAxisIndex(index, this.data.categoryData);
    }

    return index;
}

```

## diffScaled

This function computes and returns a diff of values in the scale.

```

function diffScaled(
    scale: d3.scale.Linear<any, any>,
    value1: any,
    value2: any
): number;

```

Example:

```

import { axis } from "powerbi-visuals-utils-chartutils";
// ...

var scale: d3.scale.Linear<number, number>,
    range = [0, 999],
    domain = [0, 1, 2, 3, 4, 5, 6, 7, 8, 999];

scale = d3.scale.linear()
    .range(range)
    .domain(domain);

return axis.diffScaled(scale, 0, 0));

// returns: 0

```

## createDomain

This function creates a domain of values for axis.

```
function createDomain(
  data: any[],
  axisType: ValueTypeDescriptor,
  isScalar: boolean,
  forcedScalarDomain: any[],
  ensureDomain?: NumberRange
): number[];
```

Example:

```
import { axis } from "powerbi-visuals-utils-chartutils";
// ...

var cartesianSeries = [
  {
    data: [
      { categoryValue: 7, value: 11, categoryIndex: 0, seriesIndex: 0 },
      {
        categoryValue: 9,
        value: 9,
        categoryIndex: 1,
        seriesIndex: 0
      },
      {
        categoryValue: 15,
        value: 6,
        categoryIndex: 2,
        seriesIndex: 0
      },
      { categoryValue: 22, value: 7, categoryIndex: 3, seriesIndex: 0 }
    ]
  }
];
var domain = axis.createDomain(
  cartesianSeries,
  ValueType.fromDescriptor({ text: true }),
  false,
  []
);
// returns: [0, 1, 2, 3]
```

## getCategoryValueType

This function gets the `ValueType` of a category column. Default is `Text` if the type isn't present.

```
function getCategoryValueType(
  data: any[],
  axisType: ValueTypeDescriptor,
  isScalar: boolean,
  forcedScalarDomain: any[],
  ensureDomain?: NumberRange
): number[];
```

Example:

```

import { axis } from "powerbi-visuals-utils-chartutils";
// ...

var cartesianSeries = [
  {
    data: [
      { categoryValue: 7, value: 11, categoryIndex: 0, seriesIndex: 0 },
      {
        categoryValue: 9,
        value: 9,
        categoryIndex: 1,
        seriesIndex: 0
      },
      {
        categoryValue: 15,
        value: 6,
        categoryIndex: 2,
        seriesIndex: 0
      },
      { categoryValue: 22, value: 7, categoryIndex: 3, seriesIndex: 0 }
    ]
  }
];
axis.getCategoryValueType(
  cartesianSeries,
  ValueType.fromDescriptor({ text: true }),
  false,
  []
);
// returns: [0, 1, 2, 3]

```

## createAxis

This function creates a D3 axis including scale. Can be vertical or horizontal, and either datetime, numeric, or text.

```
function createAxis(options: CreateAxisOptions): IAxisProperties;
```

Example:

```

import { axis } from "powerbi-visuals-utils-chartutils";
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";
// ...

var dataPercent = [0.0, 0.33, 0.49];

var formatStringProp: powerbi.DataViewObjectPropertyIdentifier = {
    objectName: "general",
    propertyName: "formatString"
};
let metaDataColumnPercent: powerbi.DataViewMetadataColumn = {
    displayName: "Column",
    type: ValueType.fromDescriptor({ numeric: true }),
    objects: {
        general: {
            formatString: "0 %"
        }
    }
};

var os = axis.createAxis({
    pixelSpan: 100,
    dataDomain: [dataPercent[0], dataPercent[2]],
    metaDataColumn: metaDataColumnPercent,
    formatString: valueFormatter.getFormatString(
        metaDataColumnPercent,
        formatStringProp
    ),
    outerPadding: 0.5,
    isScalar: true,
    isVertical: true
});

```

## applyCustomizedDomain

This function sets customized domain, but don't change when nothing is set.

```
function applyCustomizedDomain(customizedDomain, forcedDomain: any[]): any[];
```

Example:

```

import { axis } from "powerbi-visuals-utils-chartutils";
// ...

let customizedDomain = [undefined, 20],
    existingDomain = [0, 10];

axis.applyCustomizedDomain(customizedDomain, existingDomain);

// returns: {0:0, 1:20}

```

## combineDomain

This function combines the forced domain with the actual domain if one of the values was set. The forcedDomain is in first priority. Extends the domain if any reference point requires it.

```

function combineDomain(
    forcedDomain: any[],
    domain: any[],
    ensureDomain?: NumberRange
): any[];

```

Example:

```
import { axis } from "powerbi-visuals-utils-chartutils";
// ...

let forcedYDomain = this.valueAxisProperties
? [this.valueAxisProperties["secStart"], this.valueAxisProperties["secEnd"]]
: null;

let xDomain = [minX, maxX];

axis.combineDomain(forcedYDomain, xDomain, ensureXDomain);
```

## powerOfTen

This function indicates whether the number is power of 10.

```
function powerOfTen(d: any): boolean;
```

Example:

```
import { axis } from "powerbi-visuals-utils-chartutils";
// ...

axis.powerOfTen(10);

// returns: true
```

# DataLabelManager

The `DataLabelManager` helps to create and maintain labels. It arranges label elements using the anchor point or rectangle. Collisions can be automatically detected to reposition or hide elements.

The `DataLabelManager` class provides the following methods:

## hideCollidedLabels

This method arranges the labels position and visibility on the canvas according to labels sizes and overlapping.

```
function hideCollidedLabels(
  viewport: IViewport,
  data: any[],
  layout: any,
  addTransform: boolean = false
): LabelEnabledDataPoint[];
```

Example:

```
let dataLabelManager = new DataLabelManager();
let filteredData = dataLabelManager.hideCollidedLabels(
  this.viewport,
  values,
  labelLayout,
  true
);
```

## IsValid

This static method checks if provided rectangle is valid(has positive width and height).

```
function isValid(rect: IRect): boolean;
```

Example:

```
let rectangle = {  
    left: 150,  
    top: 130,  
    width: 120,  
    height: 110  
};  
  
DataLabelManager.isValid(rectangle);  
  
// returns: true
```

## DataLabelUtils

The `DataLabelUtils` provides utils to manipulate data labels.

The module provides the following functions, interfaces, and classes:

### **getLabelPrecision**

This function calculates precision from given format.

```
function getLabelPrecision(precision: number, format: string): number;
```

### **getLabelFormattedText**

This function returns format precision from given format.

```
function getLabelFormattedText(options: LabelFormattedTextOptions): string;
```

Example:

```
import { dataLabelUtils } from "powerbi-visuals-utils-chartutils";  
// ...  
  
let options: LabelFormattedTextOptions = {  
    text: "some text",  
    fontFamily: "sans",  
    fontSize: "15",  
    fontWeight: "normal"  
};  
  
dataLabelUtils.getLabelFormattedText(options);
```

### **enumerateDataLabels**

This function returns VisualObjectInstance for data labels.

```
function enumerateDataLabels(  
    options: VisualDataLabelsSettingsOptions  
) : VisualObjectInstance;
```

### enumerateCategoryLabels

This function adds VisualObjectInstance for Category data labels to enumeration object.

```
function enumerateCategoryLabels(  
    enumeration: VisualObjectInstanceEnumerationObject,  
    dataLabelsSettings: VisualDataLabelsSettings,  
    withFill: boolean,  
    isShowCategory: boolean = false,  
    fontSize?: number  
) : void;
```

### createColumnFormatterCacheManager

This function returns Cache Manager that provides quick access to formatted labels

```
function createColumnFormatterCacheManager(): IColumnFormatterCacheManager;
```

Example:

```
import { dataLabelUtils } from "powerbi-visuals-utils-chartutils";  
// ...  
  
let value: number = 200000;  
  
labelSettings.displayUnits = 1000000;  
labelSettings.precision = 1;  
  
let formattersCache = DataLabelUtils.createColumnFormatterCacheManager();  
let formatter = formattersCache.getOrCreate(null, labelSettings);  
let formattedValue = formatter.format(value);  
  
// formattedValue == "0.2M"
```

## Legend service

The `Legend` service provides helper interfaces for creating and managing PBI legends for Power BI visuals

The module provides the following functions and interfaces:

### createLegend

This helper function simplifies Power BI Custom Visual legends creation.

```
function createLegend(  
    legendParentElement: HTMLElement, // top visual element, container in which legend will be created  
    interactive: boolean, // indicates that legend should be interactive  
    interactivityService: IIInteractivityService, // reference to IIInteractivityService interface which need to  
    // create legend click events  
    isScrollable: boolean = false, // indicates that legend could be scrollable or not  
    legendPosition: LegendPosition = LegendPosition.Top // Position of the legend inside of legendParentElement  
    container  
) : ILegend;
```

Example:

```

public constructor(options: VisualConstructorOptions) {
    this.visualInitOptions = options;
    this.layers = [];

    var element = this.element = options.element;
    var viewport = this.currentViewport = options.viewport;
    var hostServices = options.host;

    //... some other init calls

    if (this.behavior) {
        this.interactivityService = createInteractivityService(hostServices);
    }
    this.legend = createLegend(
        element,
        options.interactivity && options.interactivity.isInteractiveLegend,
        this.interactivityService,
        true);
}

```

## ILegend

This Interface implements all methods necessary for legend creation

```

export interface ILegend {
    getMargins(): IViewport;
    isVisible(): boolean;
    changeOrientation(orientation: LegendPosition): void; // processing legend orientation
    getOrientation(): LegendPosition; // get information about current legend orientation
    drawLegend(data: LegendData, viewport: IViewport); // all legend rendering code is placing here
    /**
     * Reset the legend by clearing it
     */
    reset(): void;
}

```

## drawLegend

This function measures the height of the text with the given SVG text properties.

```
function drawLegend(data: LegendData, viewport: IViewport): void;
```

Example:

```

private renderLegend(): void {
    if (!this.isInteractive) {
        let legendObjectProperties = this.data.legendObjectProperties;
        if (legendObjectProperties) {
            let legendData = this.data.legendData;
            LegendData.update(legendData, legendObjectProperties);
            let position = <string>legendObjectProperties[legendProps.position];
            if (position)
                this.legend.changeOrientation(LegendPosition[position]);

            this.legend.drawLegend(legendData, this.parentViewport);
        } else {
            this.legend.changeOrientation(LegendPosition.Top);
            this.legend.drawLegend({ dataPoints: [] }, this.parentViewport);
        }
    }
}

```

## Next steps

- [Read how to use InteractivityUtils to add selections into Power BI Visuals](#)

# Color utils

3/13/2020 • 5 minutes to read • [Edit Online](#)

This article will help you to install, import, and use color utils. This article describes how to use color utils simplify applying of themes and palettes on visual's data points on Power BI visuals.

## Requirements

To use the package, you should have the following things:

- [node.js](#) (we recommend the latest LTS version)
- [npm](#) (the minimal supported version is 3.0.0)
- The custom visual created by [PowerBI-visuals-tools](#)

## Installation

To install the package, you should run the following command in the directory with your current visual:

```
npm install powerbi-visuals-utils-colorutils --save
```

This command installs the package and adds a package as a dependency to your `package.json`

## Usage

To user interactivity utils, you have to import the required component in the source code of the visual.

```
import { ColorHelper } from "powerbi-visuals-utils-colorutils";
```

Learn how to install and use the ColorUtils in your Power BI visuals:

- [Usage Guide] The Usage Guide describes a public API of the package. You will find a description and a few examples for each public interface of the package.

This package contains the following classes and modules:

- [ColorHelper](#) - helps to generate different colors for your chart values
- [colorUtils](#) - helps to convert color formats

### ColorHelper

The `ColorHelper` class provides the following functions and methods:

```
getColorForSeriesValue
```

This method gets the color for the given series value. If no explicit color or default color has been set, then the color is allocated from the color scale for this series.

```
getColorForSeriesValue(objects: IDataViewObjects, value: PrimitiveValue, themeColorName?: ThemeColorName): string;
```

### Example

```

import powerbi from "powerbi-visuals-api";
import { ColorHelper } from "powerbi-visuals-utils-colorutils";

import DataViewObjects = powerbi.DataViewObjects;

import DataViewValueColumns = powerbi.DataViewValueColumns;
import DataViewValueColumnGroup = powerbi.DataViewValueColumnGroup;
import DataViewObjectPropertyIdentifier = powerbi.DataViewObjectPropertyIdentifier;

import IVisual = powerbi.extensibility.visual.IVisual;
import IColorPalette = powerbi.extensibilityIColorPalette;
import VisualUpdateOptions = powerbi.extensibility.visual.VisualUpdateOptions;
import VisualConstructorOptions = powerbi.extensibility.visual.VisualConstructorOptions;

export class YourVisual implements IVisual {
    // Implementation of IVisual

    private colorPalette: IColorPalette;

    constructor(options: VisualConstructorOptions) {
        this.colorPalette = options.host.colorPalette;
    }

    public update(visualUpdateOptions: VisualUpdateOptions): void {
        const valueColumns: DataViewValueColumns = visualUpdateOptions.dataViews[0].categorical.values,
            grouped: DataViewValueColumnGroup[] = valueColumns.grouped(),
            defaultDataPointColor: string = "green",
            fillProp: DataViewObjectPropertyIdentifier = {
                objectName: "objectName",
                propertyName: "propertyName"
            };

        let colorHelper: ColorHelper = new ColorHelper(
            this.colorPalette,
            fillProp,
            defaultDataPointColor);

        for (let i = 0; i < grouped.length; i++) {
            let grouping: DataViewValueColumnGroup = grouped[i];

            let color = colorHelper.getColorForSeriesValue(grouping.objects, grouping.name); // returns a color
            of the series
        }
    }
}

```

#### getColorForMeasure

This method gets the color for the given measure.

```
getColorForMeasure(objects: IDataViewObjects, measureKey: any, themeColorName?: ThemeColorName): string;
```

#### Example

```

import powerbi from "powerbi-visuals-api";
import { ColorHelper } from "powerbi-visuals-utils-colorutils";

import DataViewObjects = powerbi.DataViewObjects;
import IVisual = powerbi.extensibility.visual.IVisual;
import IColorPalette = powerbi.extensibility.IColorPalette;
import VisualUpdateOptions = powerbi.extensibility.visual.VisualUpdateOptions;
import DataViewObjectPropertyIdentifier = powerbi.DataViewObjectPropertyIdentifier;
import VisualConstructorOptions = powerbi.extensibility.visual.VisualConstructorOptions;

export class YourVisual implements IVisual {
    // Implementation of IVisual

    private colorPalette: IColorPalette;

    constructor(options: VisualConstructorOptions) {
        this.colorPalette = options.host.colorPalette;
    }

    public update(visualUpdateOptions: VisualUpdateOptions): void {
        const objects: DataViewObjects = visualUpdateOptions.dataViews[0].categorical.categories[0].objects[0],
            defaultDataPointColor: string = "green",
            fillProp: DataViewObjectPropertyIdentifier = {
                objectName: "objectName",
                propertyName: "propertyName"
            };

        let colorHelper: ColorHelper = new ColorHelper(
            this.colorPalette,
            fillProp,
            defaultDataPointColor);

        let color = colorHelper.getColorForMeasure(objects, ""); // returns a color
    }
}

```

### **static** `normalizeSelector`

This method returns the normalized selector

```
static normalizeSelector(selector: Selector, isSingleSeries?: boolean): Selector;
```

### **Example**

```

import ISelectionId = powerbi.visuals.ISelectionId;
import { ColorHelper } from "powerbi-visuals-utils-colorutils";

let selectionId: ISelectionId = ...;
let selector = ColorHelper.normalizeSelector(selectionId.getSelector(), false);

```

Methods `getThemeColor` and `getHighContrastColor` are both related to color theme colors. Also `ColorHelper` has `isHighContrast` property that should be useful for check.

### `getThemeColor`

This method returns theme color

```
getThemeColor(themeColorName?: ThemeColorName): string;
```

### `getHighContrastColor`

This method return color for high contrast mode

```
getHighContrastColor(themeColorName?: ThemeColorName, defaultColor?: string): string;
```

#### Example related to high contrast mode usage:

```
import { ColorHelper } from "powerbi-visuals-utils-colorutils";

export class MyVisual implements IVisual {
    private colorHelper: ColorHelper;

    private init(options: VisualConstructorOptions): void {
        this.colors = options.host.colorPalette;
        this.colorHelper = new ColorHelper(this.colors);
    }

    private createViewport(element: HTMLElement): void {
        const fontColor: string = "#131aea";
        const axisBackgroundColor: string = this.colorHelper.getThemeColor();

        // some d3 code before
        d3ElementName.attr("fill", colorHelper.getHighContrastColor("foreground", fontColor));
    }

    public static parseSettings(dataView: DataView, colorHelper: ColorHelper): VisualSettings {
        // some code that should be applied on formatting settings
        if (colorHelper.isHighContrast) {
            this.settings.fontColor = colorHelper.getHighContrastColor("foreground",
this.settings.fontColor);
        }
    }
}
```

## ColorUtils

The module provides the following functions:

- [hexToRGBString](#)
- [rotate](#)
- [parseColorString](#)
- [calculateHighlightColor](#)
- [createLinearColorScale](#)
- [shadeColor](#)
- [rgbBlend](#)
- [channelBlend](#)
- [hexBlend](#)

### hexToRGBString

Converts hex color to RGB string.

```
function hexToRGBString(hex: string, transparency?: number): string
```

#### Example

```
import { hexToRGBString } from "powerbi-visuals-utils-colorutils";

hexToRGBString('#112233');

// returns: "rgb(17,34,51)"
```

## rotate

Rotates RGB color.

```
function rotate(rgbString: string, rotateFactor: number): string
```

### Example

```
import { rotate } from "powerbi-visuals-utils-colorutils";

rotate("#CC0000", 0.25); // returns: #66CC00
```

## parseColorString

Parses any color string to RGB format.

```
function parseColorString(color: string): RgbColor
```

### Example

```
import { parseColorString } from "powerbi-visuals-utils-colorutils";

parseColorString('#09f');
// returns: {R: 0, G: 153, B: 255 }

parseColorString('rgba(1, 2, 3, 1.0)');
// returns: {R: 1, G: 2, B: 3, A: 1.0 }
```

## calculateHighlightColor

Calculate the highlight color from the rgbColor based on the lumianceThreshold and delta.

```
function calculateHighlightColor(rgbColor: RgbColor, lumianceThreshold: number, delta: number): string
```

### Example

```
import { calculateHighlightColor } from "powerbi-visuals-utils-colorutils";

let yellow = "#FFFF00",
    yellowRGB = parseColorString(yellow);

calculateHighlightColor(yellowRGB, 0.8, 0.2);

// returns: '#CCCC00'
```

## createLinearColorScale

Returns a linear color scale for given domain of numbers.

```
function createLinearColorScale(domain: number[], range: string[], clamp: boolean): LinearColorScale
```

## Example

```
import { createLinearColorScale } from "powerbi-visuals-utils-colorutils";

let scale = ColorUtility.createLinearColorScale(
    [0, 1, 2, 3, 4],
    ["red", "green", "blue", "black", "yellow"],
    true);

scale(1); // returns: green
scale(10); // returns: yellow
```

## shadeColor

Converts string hex expression to number, calculate percentage and R, G, B channels. Applies percentage for each channel and returns back hex value as string with pound sign.

```
function shadeColor(color: string, percent: number): string
```

## Example

```
import { shadeColor } from "powerbi-visuals-utils-colorutils";

shadeColor('#000000', 0.1); // returns '#1a1a1a'
shadeColor('#FFFFFF', -0.5); // returns '#808080'
shadeColor('#00B8AA', -0.25); // returns '#008a80'
shadeColor('#00B8AA', 0); // returns '#00b8aa'
```

## rgbBlend

Overlays a color with opacity over a background color. Any alpha-channel is ignored.

```
function rgbBlend(foreColor: RgbColor, opacity: number, backColor: RgbColor): RgbColor
```

## Example

```
import { rgbBlend} from "powerbi-visuals-utils-colorutils";

rgbBlend({R: 100, G: 100, B: 100}, 0.5, {R: 200, G: 200, B: 200});

// returns: {R: 150, G: 150, B: 150}
```

## channelBlend

Blends a single channel for two colors.

```
function channelBlend(foreChannel: number, opacity: number, backChannel: number): number
```

## Example

```
import { channelBlend} from "powerbi-visuals-utils-colorutils";

channelBlend(0, 1, 255); // returns: 0
channelBlend(128, 1, 255); // returns: 128
channelBlend(255, 0, 0); // returns: 0
channelBlend(88, 0, 88); // returns: 88
```

## hexBlend

Overlays a color with opacity over a background color.

```
function hexBlend(foreColor: string, opacity: number, backColor: string): string
```

### Example

```
import { hexBlend} from "powerbi-visuals-utils-colorutils";

let yellow = "#FFFF00",
    black = "#000000",
    white = "#FFFFFF";

hexBlend(yellow, 0.5, white); // returns: "#FFFF80"
hexBlend(white, 0.5, yellow); // returns: "#FFFF80"

hexBlend(yellow, 0.5, black); // returns: "#808000"
hexBlend(black, 0.5, yellow); // returns: "#808000"
```

# SVG utils

3/13/2020 • 8 minutes to read • [Edit Online](#)

SVGUtils is a set of functions and classes to simplify SVG manipulations for Power BI visuals

## Installation

To install the package, you should run the following command in the directory with your current visual:

```
npm install powerbi-visuals-utils-svgutils --save
```

## CssConstants

The `CssConstants` module provides the special function and interface to work with class selectors.

The `powerbi.extensibility.utils.svg.CssConstants` module provides the following function and interface:

## ClassAndSelector

This interface describes common properties of the class selector.

```
interface ClassAndSelector {  
  class: string;  
  selector: string;  
}
```

### createClassAndSelector

This function creates an instance of `ClassAndSelector` with the given name of the class.

```
function createClassAndSelector(className: string): ClassAndSelector;
```

Example:

```
import { CssConstants } from "powerbi-visuals-utils-svgutils";  
import createClassAndSelector = CssConstants.createClassAndSelector;  
import ClassAndSelector = CssConstants.ClassAndSelector;  
  
let divSelector: ClassAndSelector = createClassAndSelector("sample-block");  
  
divSelector.selector === ".sample-block"; // returns: true  
divSelector.class === "sample-block"; // returns: true
```

## manipulation

The `manipulation` provides some special functions to generate strings for using with SVG transform property.

The module provides the following functions:

### translate

This function creates a translate string for using with the SVG transform property.

```
function translate(x: number, y: number): string;
```

Example:

```
import { manipulation } from "powerbi-visuals-utils-svgutils";
// ...

manipulation.translate(100, 100);

// returns: translate(100,100)
```

### **translateXWithPixels**

This function creates a translateX string for using with the SVG transform property.

```
function translateXWithPixels(x: number): string;
```

Example:

```
import { manipulation } from "powerbi-visuals-utils-svgutils";
// ...

manipulation.translateXWithPixels(100);

// returns: translateX(100px)
```

### **translateWithPixels**

This function creates a translate string for using with the SVG transform property.

```
function translateWithPixels(x: number, y: number): string;
```

Example:

```
import { manipulation } from "powerbi-visuals-utils-svgutils";
// ...

manipulation.translateWithPixels(100, 100);

// returns: translate(100px,100px)
```

### **translateAndRotate**

This function creates a translate-rotate string for using with the SVG transform property.

```
function translateAndRotate(
  x: number,
  y: number,
  px: number,
  py: number,
  angle: number
): string;
```

Example:

```
import { manipulation } from "powerbi-visuals-utils-svgutils";
// ...

manipulation.translateAndRotate(100, 100, 50, 50, 35);

// returns: translate(100,100) rotate(35,50,50)
```

## scale

This function creates a scale string for using in a CSS transform property.

```
function scale(scale: number): string;
```

Example:

```
import { manipulation } from "powerbi-visuals-utils-svgutils";
// ...

manipulation.scale(50);

// returns: scale(50)
```

## transformOrigin

This function creates a transform-origin string for using in a CSS transform-origin property.

```
function transformOrigin(xOffset: string, yOffset: string): string;
```

Example:

```
import { manipulation } from "powerbi-visuals-utils-svgutils";
// ...

manipulation.transformOrigin(5, 5);

// returns: 5 5
```

## flushAllD3Transitions

This function forces every transition of D3 to complete.

```
function flushAllD3Transitions(): void;
```

Example:

```
import { manipulation } from "powerbi-visuals-utils-svgutils";
// ...

manipulation.flushAllD3Transitions();

// forces every transition of D3 to complete
```

## parseTranslateTransform

This function parses the transform string with value "translate(x,y)".

```
function parseTranslateTransform(input: string): { x: string; y: string };
```

Example:

```
import { manipulation } from "powerbi-visuals-utils-svgutils";
// ...

manipulation.parseTranslateTransform("translate(100px,100px)");

// returns: { "x": "100px", "y": "100px" }
```

## createArrow

This function creates an arrow.

```
function createArrow(
  width: number,
  height: number,
  rotate: number
): { path: string; transform: string };
```

Example:

```
import { manipulation } from "powerbi-visuals-utils-svgutils";
// ...

manipulation.createArrow(10, 20, 5);

/* returns: {
  "path": "M0 0L0 20L10 10 Z",
  "transform": "rotate(5 5 10)"
}*/
```

# Rect

The `Rect` module provides some special functions to manipulate rectangles.

The module provides the following functions:

## getOffset

This function returns an offset of the rectangle.

```
function getOffset(rect: IRect): IPoint;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.getOffset({ left: 25, top: 25, width: 100, height: 100 });

/* returns: {
  x: 25,
  y: 25
}*/
```

## **getSize**

This function returns size of the rectangle.

```
function getSize(rect: IRect): ISize;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.getSize({ left: 25, top: 25, width: 100, height: 100 });

/* returns: {
    width: 100,
    height: 100
}*/
```

## **setSize**

This function modifies size of the rectangle.

```
function setSize(rect: IRect, value: ISize): void;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

let rectangle = { left: 25, top: 25, width: 100, height: 100 };

Rect.setSize(rectangle, { width: 250, height: 250 });

// rectangle === { left: 25, top: 25, width: 250, height: 250 }
```

## **right**

This function returns a right position of the rectangle.

```
function right(rect: IRect): number;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.right({ left: 25, top: 25, width: 100, height: 100 });

// returns: 125
```

## **bottom**

This function returns a bottom position of the rectangle.

```
function bottom(rect: IRect): number;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.bottom({ left: 25, top: 25, width: 100, height: 100 });

// returns: 125
```

## topLeft

This function returns a top-left position of the rectangle.

```
function topLeft(rect: IRect): IPo
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.topLeft({ left: 25, top: 25, width: 100, height: 100 });

// returns: { x: 25, y: 25 }
```

## topRight

This function returns a top-right position of the rectangle.

```
function topRight(rect: IRect): IPo
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.topRight({ left: 25, top: 25, width: 100, height: 100 });

// returns: { x: 125, y: 25 }
```

## bottomLeft

This function returns a bottom-left position of the rectangle.

```
function bottomLeft(rect: IRect): IPo
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.bottomLeft({ left: 25, top: 25, width: 100, height: 100 });

// returns: { x: 25, y: 125 }
```

## bottomRight

This function returns a bottom-right position of the rectangle.

```
function bottomRight(rect: IRect): IPoint;
```

## Example

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.bottomRight({ left: 25, top: 25, width: 100, height: 100 });

// returns: { x: 125, y: 125 }
```

## clone

This function creates a copy of the rectangle.

```
function clone(rect: IRect): IRect;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.clone({ left: 25, top: 25, width: 100, height: 100 });

/* returns: {
    left: 25, top: 25, width: 100, height: 100}
 */
```

## toString

This function converts rectangle to string.

```
function toString(rect: IRect): string;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.toString({ left: 25, top: 25, width: 100, height: 100 });

// returns: {left:25, top:25, width:100, height:100}
```

## offset

This function applies given offset to the rectangle.

```
function offset(rect: IRect, offsetX: number, offsetY: number): IRect;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.offset({ left: 25, top: 25, width: 100, height: 100 }, 50, 50);

/* returns: {
    left: 75,
    top: 75,
    width: 100,
    height: 100
}*/
```

## add

This function adds the first rectangle to the second rectangle.

```
function add(rect: IRect, rect2: IRect): IRect;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.add(
  { left: 25, top: 25, width: 100, height: 100 },
  { left: 50, top: 50, width: 75, height: 75 }
);

/* returns: {
    left: 75,
    top: 75,
    height: 175,
    width: 175
}*/
```

## getClosestPoint

This function returns the closest point on the rect to the given point.

```
function getClosestPoint(rect: IRect, x: number, y: number): IPoint;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.getClosestPoint({ left: 0, top: 0, width: 100, height: 100 }, 50, 50);

/* returns: {
  x: 50,
  y: 50
}*/
```

## equal

This function compares rectangles and returns true if they're the same.

```
function equal(rect1: IRect, rect2: IRect): boolean;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.equal(
  { left: 0, top: 0, width: 100, height: 100 },
  { left: 50, top: 50, width: 100, height: 100 }
);

// returns: false
```

## equalWithPrecision

This function compares rectangles by considering precision of the values.

```
function equalWithPrecision(rect1: IRect, rect2: IRect): boolean;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.equalWithPrecision(
  { left: 0, top: 0, width: 100, height: 100 },
  { left: 50, top: 50, width: 100, height: 100 }
);

// returns: false
```

## isEmpty

This function checks if rectangle is empty.

```
function isEmpty(rect: IRect): boolean;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.isEmpty({ left: 0, top: 0, width: 0, height: 0 });

// returns: true
```

## containsPoint

This function checks if rectangle contains the point.

```
function containsPoint(rect: IRect, point: IPoint): boolean;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.containsPoint(
  { left: 0, top: 0, width: 100, height: 100 },
  { x: 50, y: 50 }
);

// returns: true
```

## isIntersecting

This function checks if rectangles are intersecting.

```
function isIntersecting(rect1: IRect, rect2: IRect): boolean;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.isIntersecting(
  { left: 0, top: 0, width: 100, height: 100 },
  { left: 0, top: 0, width: 50, height: 50 }
);

// returns: true
```

## intersect

This function returns an intersection of rectangles.

```
function intersect(rect1: IRect, rect2: IRect): IRect;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.intersect(
  { left: 0, top: 0, width: 100, height: 100 },
  { left: 0, top: 0, width: 50, height: 50 }
);

/* returns: {
  left: 0,
  top: 0,
  width: 50,
  height: 50
}*/
```

## combine

This function combines rectangles.

```
function combine(rect1: IRect, rect2: IRect): IRect;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.combine(
  { left: 0, top: 0, width: 100, height: 100 },
  { left: 0, top: 0, width: 50, height: 120 }
);

/* returns: {
  left: 0,
  top: 0,
  width: 100,
  height: 120
}*/
```

## getCentroid

This function returns a center point of the rectangle.

```
function getCentroid(rect: IRect): IPPoint;
```

Example:

```
import { shapes } from "powerbi-visuals-utils-svgutils";
import Rect = shapes.Rect;
// ...

Rect.getCentroid({ left: 0, top: 0, width: 100, height: 100 });

/* returns: {
  x: 50,
  y: 50
}*/
```

# pointer

The `pointer` module provides a special function to get position of the pointer.

The module provides the following function:

## getCoordinates

This function returns position of the pointer.

```
function getCoordinates(rootNode: Element, isPointerEvent: boolean): number[];
```

Example:

```
import { pointer } from "powerbi-visuals-utils-svgutils";

let bodySelection = d3.select("body");

bodySelection
  .append("div")
  .style({
    width: "100px",
    height: "100px",
    "background-color": "green"
  })
  .on("click", () => {
    pointer.getCoordinates(bodySelection.node(), true);
  });

// click element, after that you will get position of the pointer
```

# Type utils

3/13/2020 • 4 minutes to read • [Edit Online](#)

TypeUtils is a set of functions and classes to extend the basic types for Power BI visuals.

## Installation

To install the package, you should run the following command in the directory with your current custom visual:

```
npm install powerbi-visuals-utils-typeutils --save
```

This command installs the package and adds a package as a dependency to your package.json

## Double

The `Double` provides abilities to manipulate precision of the numbers.

The module provides the following functions:

### **pow10**

This function returns power of 10.

```
function pow10(exp: number): number;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.pow10(25);

// returns: 1e+25
```

### **log10**

This function returns a 10 base logarithm of the number.

```
function log10(val: number): number;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.log10(25);

// returns: 1
```

## getPrecision

This function returns a power of 10 representing precision of the number.

```
function getPrecision(x: number, decimalDigits?: number): number;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.getPrecision(562344, 6);

// returns: 0.1
```

### **equalWithPrecision**

This function checks if a delta between two numbers is less than provided precision.

```
function equalWithPrecision(x: number, y: number, precision?: number): boolean;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.equalWithPrecision(1, 1.005, 0.01);

// returns: true
```

### **lessWithPrecision**

This function checks if the first value is less than the second value.

```
function lessWithPrecision(x: number, y: number, precision?: number): boolean;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.lessWithPrecision(0.995, 1, 0.001);

// returns: true
```

### **lessOrEqualWithPrecision**

This function checks if the first value is less or equal than the second value.

```
function lessOrEqualWithPrecision(x: number, y: number, precision?: number): boolean;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.lessOrEqualWithPrecision(1.005, 1, 0.01);

// returns: true
```

## greaterWithPrecision

This function checks if the first value is greater than the second value.

```
function greaterWithPrecision(x: number, y: number, precision?: number): boolean;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.greaterWithPrecision(1, 0.995, 0.01);

// returns: false
```

## greaterOrEqualWithPrecision

This function checks if the first value is greater or equal to the second value.

```
function greaterOrEqualWithPrecision(x: number, y: number, precision?: number): boolean;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.greaterOrEqualWithPrecision(1, 1.005, 0.01);

// returns: true
```

## floorWithPrecision

This function floors the number with the provided precision.

```
function floorWithPrecision(x: number, precision?: number): number;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.floorWithPrecision(5.96, 0.001);

// returns: 5
```

## ceilWithPrecision

This function `ceils` the number with the provided precision.

```
function ceilWithPrecision(x: number, precision?: number): number;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.ceilWithPrecision(5.06, 0.001);

// returns: 6
```

## **floorToPrecision**

This function floors the number to the provided precision.

```
function floorToPrecision(x: number, precision?: number): number;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.floorToPrecision(5.96, 0.1);

// returns: 5.9
```

## **ceilToPrecision**

This function `ceils` the number to the provided precision.

```
function ceilToPrecision(x: number, precision?: number): number;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.ceilToPrecision(-506, 10);

// returns: -500
```

## **roundToPrecision**

This function rounds the number to the provided precision.

```
function roundToPrecision(x: number, precision?: number): number;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.roundToPrecision(596, 10);

// returns: 600
```

## **ensureInRange**

This function returns a number that is between min and max.

```
function ensureInRange(x: number, min: number, max: number): number;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.ensureInRange(-27.2, -10, -5);

// returns: -10
```

## round

This function rounds the number.

```
function round(x: number): number;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.round(27.45);

// returns: 27
```

## removeDecimalNoise

This function removes the decimal noise.

```
function removeDecimalNoise(value: number): number;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.removeDecimalNoise(21.49300000000002);

// returns: 21.493
```

## isInteger

This function checks if the number is integer.

```
function isInteger(value: number): boolean;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.isInteger(21.49300000000002);

// returns: false
```

## toIncrement

This function increments the number by the provided number and returns the rounded number.

```
function toIncrement(value: number, increment: number): number;
```

Example:

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.toIncrement(0.6383723, 0.05);

// returns: 0.65
```

## Prototype

The `Prototype` provides abilities to inherit objects.

The module provides the following functions:

### inherit

This function returns a new object with the provided object as its prototype.

```
function inherit<T>(obj: T, extension?: (inherited: T) => void): T;
```

Example:

```
import { prototype } from "powerbi-visuals-utils-typeutils";
// ...

let base = { Microsoft: "Power BI" };

prototype.inherit(base);

/* returns: {
  __proto__: {
    Microsoft: "Power BI"
  }
}*/
```

### inheritSingle

This function returns a new object with the provided object as its prototype if, and only if, the prototype hasn't been set.

```
function inheritSingle<T>(obj: T): T;
```

Example:

```
import { prototype } from "powerbi-visuals-utils-typeutils";
// ...

let base = { Microsoft: "Power BI" };

prototype.inheritSingle(base);

/* returns: {
  __proto__: {
    Microsoft: "Power BI"
  }
}*/

```

## PixelConverter

The `PixelConverter` provides an ability to convert pixels to points, and points to pixels.

The module provides the following functions:

### toString

This function converts the pixel value to a string.

```
function toString(px: number): string;
```

Example:

```
import { pixelConverter } from "powerbi-visuals-utils-typeutils";
// ...

pixelConverter.toString(25);

// returns: 25px
```

### fromPoint

This function converts the provided point value to the pixel value and returns the string interpretation.

```
function fromPoint(pt: number): string;
```

Example:

```
import { pixelConverter } from "powerbi-visuals-utils-typeutils";
// ...

pixelConverter.fromPoint(8);

// returns: 33.333333333333px
```

### fromPointToPixel

This function converts the provided point value to the pixel value.

```
function fromPointToPixel(pt: number): number;
```

Example:

```
import { pixelConverter } from "powerbi-visuals-utils-typeutils";
// ...

pixelConverter.fromPointToPixel(8);

// returns: 10.66666666666666
```

## toPoint

This function converts the pixel value to the point value.

```
function toPoint(px: number): number;
```

Example:

```
import { pixelConverter } from "powerbi-visuals-utils-typeutils";
// ...

pixelConverter.toPoint(8);

// returns: 6
```

# Power BI visuals test utils

4/28/2020 • 10 minutes to read • [Edit Online](#)

This article helps you install, import, and use the Power BI visuals test utils. These test utilities can be used for unit testing and include mocks and methods for elements such as data views, selections and color schemas.

## Requirements

To use this package, you'll need to install the following:

- [node.js](#), it's recommended to use the LTS version
- [npm](#), version 3.0.0 or higher
- The [PowerBI-visuals-tools](#) package

## Installation

To install test utils and add its dependency to your *package.json*, run the following command from your Power BI visuals directory:

```
npm install powerbi-visuals-utils-testutils --save
```

The following provide descriptions and examples on the test utils public API.

## VisualBuilderBase

Used by [VisualBuilder](#) in unit-tests with the most frequently used methods `build`, `update`, and `updateRenderTimeout`.

The `build` method returns a created instance of the visual.

The `enumerateObjectInstances` and `updateEnumerateObjectInstancesRenderTimeout` methods are required to check changes on the bucket and formatting options.

```
abstract class VisualBuilderBase<T extends IVisual> {
    element: JQuery;
    viewport: IViewport;
    visualHost: IVisualHost;
    protected visual: T;
    constructor(width?: number, height?: number, guid?: string, element?: JQuery);
    protected abstract build(options: VisualConstructorOptions): T;
    init(): void;
    destroy(): void;
    update(dataView: DataView[] | DataView): void;
    updateRenderTimeout(dataViews: DataView[] | DataView, fn: Function, timeout?: number): number;
    updateEnumerateObjectInstancesRenderTimeout(dataViews: DataView[] | DataView, options: EnumerateVisualObjectInstancesOptions, fn: (enumeration: VisualObjectInstance[]) => void, timeout?: number): number;
    updateFlushAllD3Transitions(dataViews: DataView[] | DataView): void;
    updateflushAllD3TransitionsRenderTimeout(dataViews: DataView[] | DataView, fn: Function, timeout?: number): number;
    enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions): VisualObjectInstance[];
}
```

## NOTE

For further examples, see [writing VisualBuilderBase unit tests](#) and a [real usage VisualBuilderBase scenario](#).

## DataViewBuilder

Used by **TestDataViewBuilder**, this module provides a **CategoricalDataViewBuilder** class used in the `createCategoricalDataViewBuilder` method. It also specifies interfaces and methods required for working with mocked **DataView** in unit-tests.

- `withValues` adds static series columns and `withGroupedValues` adds dynamic series columns

Don't apply both dynamic series and static series in a visual **DataViewCategorical**. You can only use them both in the **DataViewCategorical** query, where **DataViewTransform** is expected to split them into separate visual **DataViewCategorical** objects.

- `build` returns the **DataView** with metadata and **DataViewCategorical**  
`build` returns **Undefined** if the combination of parameters is illegal, such as including both dynamic and static series when building the visual **DataView**.

```
class CategoricalDataViewBuilder implements IDataViewBuilderCategorical {
    withCategory(options: DataViewBuilderCategoryColumnOptions): IDataViewBuilderCategorical;
    withCategories(categories: DataViewCategoryColumn[]): IDataViewBuilderCategorical;
    withValues(options: DataViewBuilderValuesOptions): IDataViewBuilderCategorical;
    withGroupedValues(options: DataViewBuilderGroupedValuesOptions): IDataViewBuilderCategorical;
    build(): DataView;
}

function createCategoricalDataViewBuilder(): IDataViewBuilderCategorical;
```

## TestDataViewBuilder

Used for **VisualData** creation in unit tests. When data is placed in data-field buckets, Power BI produces a categorical **DataView** object based on the data. The **TestDataViewBuilder** helps simulate categorical **DataView** creation.

```
abstract class TestDataViewBuilder {
    static DataViewName: string;
    private aggregateFunction;
    static setDefaultQueryName(source: DataViewMetadataColumn): DataViewMetadataColumn;
    static getDataViewBuilderColumnIdentitySources(options: TestDataViewBuilderColumnOptions[] | TestDataViewBuilderColumnOptions): DataViewBuilderColumnIdentitySource[];
    static getValuesTable(categories?: DataViewCategoryColumn[], values?: DataViewValueColumn[]): any[][];
    static createDataViewBuilderColumnOptions(categoriesColumns: (TestDataViewBuilderCategoryColumnOptions | TestDataViewBuilderCategoryColumnOptions[])[], valuesColumns: (DataViewBuilderValuesColumnOptions | DataViewBuilderValuesColumnOptions[])[], filter?: (options: TestDataViewBuilderColumnOptions) => boolean, customizeColumns?: CustomizeColumnFn): DataViewBuilderAllColumnOptions;
    static setUpDataViewBuilderColumnOptions(options: DataViewBuilderAllColumnOptions, aggregateFunction: (array: number[]) => number): DataViewBuilderAllColumnOptions;
    static setUpDataView(dataView: DataView, options: DataViewBuilderAllColumnOptions): DataView;
    protected createCategoricalDataViewBuilder(categoriesColumns: (TestDataViewBuilderCategoryColumnOptions | TestDataViewBuilderCategoryColumnOptions[])[], valuesColumns: (DataViewBuilderValuesColumnOptions | DataViewBuilderValuesColumnOptions[])[], columnNames: string[], customizeColumns?: CustomizeColumnFn): IDataViewBuilderCategorical;
    abstract getDataView(columnNames?: string[]): DataView;
}
```

The following lists the most frequently used interfaces when creating a `test DataView`:

```
interface TestDataViewBuilderColumnOptions extends DataViewBuilderColumnOptions {
    values: any[];
}

interface TestDataViewBuilderCategoryColumnOptions extends TestDataViewBuilderColumnOptions {
    objects?: DataViewObjects[];
    isGroup?: boolean;
}

interface DataViewBuilderColumnOptions {
    source: DataViewMetadataColumn;
}

interface DataViewBuilderSeriesData {
    values: PrimitiveValue[];
    highlights?: PrimitiveValue[];
    /** Client-computed maximum value for a column. */
    maxLocal?: any;
    /** Client-computed minimum value for a column. */
    minLocal?: any;
}

interface DataViewBuilderColumnIdentitySource {
    fields: any[];
    identities?: CustomVisualOpaqueIdentity[];
}
```

#### NOTE

For further examples, see [writing TestDataViewBuilder unit tests](#) and a [real usage TestDataViewBuilder scenario](#).

## Mocks

### MockIVisualHost

Implements `IVisualHost` to test Power BI visuals without external dependencies, such as the Power BI framework.

Useful methods include `createSelectionIdBuilder`, `createSelectionManager`, `createLocalizationManager`, and getter properties.

```

import powerbi from "powerbi-visuals-api";

import VisualObjectInstancesToPersist = powerbi.VisualObjectInstancesToPersist;
import ISelectionIdBuilder = powerbi.visuals.ISelectionIdBuilder;
import ISelectionManager = powerbi.extensibility.ISelectionManager;
import IColorPalette = powerbi.extensibility.IColorPalette;
import IVisualEventService = powerbi.extensibility.IVisualEventService;
import ITooltipService = powerbi.extensibility.ITooltipService;
import IVisualHost = powerbi.extensibility.visual.IVisualHost;

class MockIVisualHost implements IVisualHost {
    constructor(
        colorPalette?: IColorPalette,
        selectionManager?: ISelectionManager,
        tooltipServiceInstance?: ITooltipService,
        localeInstance?: MockILocale,
        allowInteractionsInstance?: MockIAllowInteractions,
        localizationManager?: powerbi.extensibility.ILocalizationManager,
        telemetryService?: powerbi.extensibility.ITelemetryService,
        authService?: powerbi.extensibility.IAuthenticationService,
        storageService?: ILocalVisualStorageService,
        eventService?: IVisualEventService);
    createSelectionIdBuilder(): ISelectionIdBuilder;
    createSelectionManager(): ISelectionManager;
    createLocalizationManager(): ILocalizationManager;
    colorPalette: IColorPalette;
    locale: string;
    telemetry: ITelemetryService;
    tooltipService: ITooltipService;
    allowInteractios: boolean;
    storageService: ILocalVisualStorageService;
    eventService: IVisualEventService;
    persistProperties(changes: VisualObjectInstancesToPersist): void;
}

```

- `createVisualHost` creates and returns instance of `IVisualHost`, actually `MockIVisualHost`

```

function createVisualHost(locale?: Object, allowInteractions?: boolean, colors?: IColorInfo[],
isEnabled?: boolean, displayNames?: any, token?: string): IVisualHost;

```

#### Example

```

import { createVisualHost } from "powerbi-visuals-utils-testutils"

let host: IVisualHost = createVisualHost();

```

#### IMPORTANT

`MockIVisualHost` is a fake implementation of `IVisualHost` and should only be used with unit-tests.

#### MockIColorPalette

Implements `IColorPalette` to test Power BI visuals without external dependencies, such as the Power BI Framework.

`MockIColorPalette` provides useful properties for checking color schema or high-contrast mode in unit-tests.

```

import powerbi from "powerbi-visuals-api";
import IColorPalette = powerbi.extensibility.ISandboxExtendedColorPalette;
import IColorInfo = powerbi.IColorInfo;

class MockIColorPalette implements IColorPalette {
    constructor(colors?: IColorInfo[]);
    getColor(key: string): IColorInfo;
    reset(): IColorPalette;
    isHighContrastMode: boolean;
    foreground: {value: string};
    foregroundLight: {value: string};
    ...
    background: {value: string};
    backgroundLight: {value: string};
    ...
    shapeStroke: {value: string};
}

```

- `createColorPalette` creates and returns an instance of `IColorPalette`, actually `MockIColorPalette`

```
function createColorPalette(colors?: IColorInfo[]): IColorPalette;
```

#### Example

```

import { createColorPalette } from "powerbi-visuals-utils-testutils"

let colorPalette: IColorPalette = createColorPalette();

```

#### IMPORTANT

`MockIColorPalette` is a fake implementation of `IColorPalette` and should only be used with unit-tests.

## MockISelectionId

Implements `ISelectionId` to test Power BI visuals without external dependencies, such as the Power BI Framework.

```

import powerbi from "powerbi-visuals-api";
import Selector = powerbi.data.Selector;
import ISelectionId = powerbi.visuals.ISelectionId;

class MockISelectionId implements ISelectionId {
    constructor(key: string);
    equals(other: ISelectionId): boolean;
    includes(other: ISelectionId, ignoreHighlight?: boolean): boolean;
    getKey(): string;
    getSelector(): Selector;
    getSelectorsByColumn(): Selector;
    hasIdentity(): boolean;
}

```

- `createSelectionId` creates and returns an instance of `ISelectionId`, actually `MockISelectionId`

```
function createSelectionId(key?: string): ISelectionId;
```

#### Example

```
import { createColorPalette } from "powerbi-visuals-utils-testutils"

let selectionId: ISelectionId = createSelectionId();
```

#### NOTE

**MockISelectionId** is a fake implementation of **ISelectionId** and should only be used with unit-tests.

### MockISelectionIdBuilder

Implements **ISelectionIdBuilder** to test Power BI visuals without external dependencies, such as the Power BI Framework.

```
import DataViewCategoryColumn = powerbi.DataViewCategoryColumn;
import DataViewValueColumn = powerbi.DataViewValueColumn;
import DataViewValueColumnGroup = powerbi.DataViewValueColumnGroup;
import DataViewValueColumns = powerbi.DataViewValueColumns;
import ISelectionIdBuilder = powerbi.visuals.ISelectionIdBuilder;
import ISelectionId = powerbi.visuals.ISelectionId;

class MockISelectionIdBuilder implements ISelectionIdBuilder {
    withCategory(categoryColumn: DataViewCategoryColumn, index: number): this;
    withSeries(seriesColumn: DataViewValueColumns, valueColumn: DataViewValueColumn | DataViewValueColumnGroup): this;
    withMeasure(measureId: string): this;
    createSelectionId(): ISelectionId;
    withMatrixNode(matrixNode: DataViewMatrixNode, levels: DataViewHierarchyLevel[]): this;
    withTable(table: DataViewTable, rowIndex: number): this;
}
```

- `createSelectionIdBuilder` creates and returns an instance of **ISelectionIdBuilder**, actually **MockISelectionIdBuilder**

```
function createSelectionIdBuilder(): ISelectionIdBuilder;
```

#### Example

```
import { selectionIdBuilder } from "powerbi-visuals-utils-testutils";

let selectionIdBuilder = createSelectionIdBuilder();
```

#### NOTE

**MockISelectionIdBuilder** is a fake implementation of **ISelectionIdBuilder** and should only be used with unit-tests.

### MockISelectionManager

Implements **ISelectionManager** to test Power BI visuals without external dependencies, such as the Power BI Framework.

```

import powerbi from "powerbi-visuals-api";
import IPromise = powerbi.IPromise;
import ISelectionId = powerbi.visuals.ISelectionId;
import ISelectionManager = powerbi.extensibility.ISelectionManager;

class MockISelectionManager implements ISelectionManager {
    select(selectionId: ISelectionId | ISelectionId[], multiSelect?: boolean): IPromise<ISelectionId[]>;
    hasSelection(): boolean;
    clear(): IPromise<{}>;
    getSelectionIds(): ISelectionId[];
    containsSelection(id: ISelectionId): boolean;
    showContextMenu(selectionId: ISelectionId, position: IPoint): IPromise<{}>;
    registerOnSelectCallback(callback: (ids: ISelectionId[]) => void): void;
    simulateSelection(selections: ISelectionId[]): void;
}

```

- `createSelectionManager` creates and returns an instance of **ISelectionManager**, actually **MockISelectionManager**

```
function createSelectionManager(): ISelectionManager
```

#### Example

```

import { createSelectionManager } from "powerbi-visuals-utils-testutils";

let selectionManager: ISelectionManager = createSelectionManager();

```

#### NOTE

**MockISelectionManager** is a fake implementation of **ISelectionManager** and should only be used with unit-tests.

## MockILocale

Sets locale and changes it for your needs during unit-testing process.

```

class MockILocale {
    constructor(locales?: Object): void; // Default locales are en-US and ru-RU
    locale(key: string): void; // setter property
    locale(): string; // getter property
}

```

- `createLocale` creates and returns instance of **MockILocale**

```
funciton createLocale(locales?: Object): MockILocale;
```

## MockITooltipService

Simulates `TooltipService` and calls it for your needs during unit-testing process.

```

class MockITooltipService implements ITooltipService {
    constructor(isEnabled: boolean = true);
    enabled(): boolean;
    show(options: TooltipShowOptions): void;
    move(options: TooltipMoveOptions): void;
    hide(options: TooltipHideOptions): void;
}

```

- `createTooltipService` creates and returns instance of `MockITooltipService`

```
function createTooltipService(isEnabled?: boolean): ITooltipService;
```

## MockIAccountInteractions

```
export class MockIAccountInteractions {
    constructor(public isEnabled?: boolean); // false by default
}
```

- `createAccountInteractions` creates and returns instance of `MockIAccountInteractions`

```
function createAccountInteractions(isEnabled?: boolean): MockIAccountInteractions;
```

## MockILocalizationManager

Provides basic abilities of `LocalizationManager` which are needed for unit-testing.

```
class MockILocalizationManager implements ILocalizationManager {
    constructor(displayNames: {[key: string]: string});
    getDisplayName(key: string): string; // returns default or setted displayNames for localized elements
}
```

- `createLocalizationManager` creates and returns an instance of `ILocalizationManager`, actually `MockILocalizationManager`

```
function createLocalizationManager(displayNames?: any): ILocalizationManager;
```

### Example

```
import { createLocalizationManager } from "powerbi-visuals-utils-testutils";
let localizationManagerMock: ILocalizationManager = createLocalizationManager();
```

## MockITelemetryService

Simulates `TelemetryService` usage.

```
class MockITelemetryService implements ITelemetryService {
    instanceId: string;
    trace(veType: powerbi.VisualEventType, payload?: string) {
    }
}
```

Creation of `MockITelemetryService` `typescript function createTelemetryService(): ITelemetryService;`

## MockIAuthenticationService

Simulates the work of `AuthenticationService` by providing a mocked AAD token.

```
class MockIAuthenticationService implements IAuthenticationService {
    constructor(token: string);
    getAADToken(visualId?: string): powerbi.IPromise<string>
}
```

- `createAuthenticationService` creates and returns an instance of `IAuthenticationService`, actually

## MockIAuthenticationService

```
function createAuthenticationService(token?: string): IAuthenticationService;
```

## MockIStorageService

Allows you to use **ILocalVisualStorageService** with the same behavior as **LocalStorage**.

```
class MockIStorageService implements ILocalVisualStorageService {
    get(key: string): IPromise<string>;
    set(key: string, data: string): IPromise<number>;
    remove(key: string): void;
}
```

- `createStorageService` creates and returns an instance of **ILocalVisualStorageService**, actually **MockIStorageService**

```
function createStorageService(): ILocalVisualStorageService;
```

## MockIEventService

```
import powerbi from "powerbi-visuals-api";
import IVisualEventService = powerbi.extensibility.IVisualEventService;
import VisualUpdateOptions = powerbi.extensibility.VisualUpdateOptions;

class MockIEventService implements IVisualEventService {
    renderingStarted(options: VisualUpdateOptions): void;
    renderingFinished(options: VisualUpdateOptions): void;
    renderingFailed(options: VisualUpdateOptions, reason?: string): void;
}
```

- `createEventService` creates and returns an instance of **IVisualEventService**, actually **MockIEventService**

```
function createEventService(): IVisualEventService;
```

## Utils

Utils include helper methods for Power BI visuals' unit testing, including helpers related to colors, numbers, and events.

- `renderTimeout` returns timeout

```
function renderTimeout(fn: Function, timeout: number = DefaultWaitForRender): number
```

- `testDom` helps set fixture in unit tests

```
function testDom(height: number | string, width: number | string): JQuery
```

Example

```

import { testDom } from "powerbi-visuals-utils-testutils";
describe("testDom", () => {
    it("should return an element", () => {
        let element: JQuery = testDom(500, 500);
        expect(element.get(0)).toBeDefined();
    });
});

```

## Color-related helper methods

- `getSolidColorStructuralObject`

```
function getSolidColorStructuralObject(color: string): any
```

Returns the following structure:

```
{ solid: { color: color } }
```

- `assertColorsMatch` compares `RgbColor` objects parsed from input strings

```
function assertColorsMatch(actual: string, expected: string, invert: boolean = false): boolean
```

- `parseColorString` parses color from the input string and returns it in specified interface `RgbColor`

```
function parseColorString(color: string): RgbColor
```

## Number-related helper methods

- `getRandomNumbers` generates a random number using min and max values. You can specify `exceptionList` and provide a function for result change.

```

function getRandomNumber(
    min: number,
    max: number,
    exceptionList?: number[],
    changeResult: (value: any) => number = x => x): number

```

- `getRandomNumbers` provides an array of random numbers generated by the `getRandomNumber` method with specified min and max values

```
function getRandomNumbers(count: number, min: number = 0, max: number = 1): number[]
```

## Event-related helper methods

The following methods are written for web page event simulation in unit tests.

- `clickElement` simulates a click on the specified element

```
function clickElement(element: JQuery, ctrlKey: boolean = false): void
```

- `createTouch` returns a `Touch` object to help simulate a touch event

```
function createTouch(x: number, y: number, element: JQuery, id: number = 0): Touch
```

- `createTouchesList` returns a list of simulated **Touch** events

```
function createTouchesList(touches: Touch[]): TouchList
```

- `createContextMenuEvent` returns **MouseEvent**

```
function createContextMenuEvent(x: number, y: number): MouseEvent
```

- `createMouseEvent` creates and returns **MouseEvent**

```
function createMouseEvent(  
    mouseEventType: MouseEventType,  
    eventType: ClickEventType,  
    x: number,  
    y: number,  
    button: number = 0): MouseEvent
```

- `createTouchEndEvent`

```
function createTouchEndEvent(touchList?: TouchList): UIEvent
```

- `createTouchMoveEvent`

```
function createTouchMoveEvent(touchList?: TouchList): UIEvent
```

- `createTouchStartEvent`

```
function createTouchStartEvent(touchList?: TouchList): UIEvent
```

## D3 event-related helper methods

The following methods are used to simulate D3 events in unit tests.

- `flushAllD3Transitions` forces all D3 transitions to complete

```
function flushAllD3Transitions()
```

### NOTE

Normally, zero-delay transitions are executed after an instantaneous delay (<10 ms), but this can cause a brief flicker if the browser renders the page twice. Once at the end of the first event loop, then again immediately on the first timer callback.

These flickers are more noticeable on IE and with a large number of webviews and are not recommended for iOS.

By flushing the timer queue at the end of the first event loop you can run any zero-delay transitions immediately and avoid the flicker.

The following methods are also included:

```
function d3Click(element: JQuery, x: number, y: number, eventType?: ClickEventType, button?: number): void
function d3MouseUp(element: JQuery, x: number, y: number, eventType?: ClickEventType, button?: number): void
function d3MouseDown(element: JQuery, x: number, y: number, eventType?: ClickEventType, button?: number): void
function d3MouseOver(element: JQuery, x: number, y: number, eventType?: ClickEventType, button?: number): void
function d3MouseMove(element: JQuery, x: number, y: number, eventType?: ClickEventType, button?: number): void
function d3MouseOut(element: JQuery, x: number, y: number, eventType?: ClickEventType, button?: number): void
function d3KeyEvent(element: JQuery, typeArg: string, keyArg: string, keyCode: number): void
function d3TouchStart(element: JQuery, touchlist?: TouchList): void
function d3TouchMove(element: JQuery, touchList?: TouchList): void
function d3TouchEnd(element: JQuery, touchList?: TouchList): void
function d3ContextMenu(element: JQuery, x: number, y: number): void
```

## Helper interfaces

The following interface and enumerations are used in the helper function.

```
interface RgbColor {
    R: number;
    G: number;
    B: number;
    A?: number;
}

enum ClickEventType {
    Default = 0,
    CtrlKey = 1,
    AltKey = 2,
    ShiftKey = 4,
    MetaKey = 8,
}

enum MouseEvent {
    click,
    mousedown,
    mouseup,
    mouseover,
    mousemove,
    mouseout,
}
```

## Next steps

To write unit tests for webpack-based Power BI visuals, and unit test with `karma` and `jasmine`, see for example [Tutorial: Add unit tests for Power BI visual projects](#).

# Tooltip utils

3/13/2020 • 2 minutes to read • [Edit Online](#)

This article will help you to install, import, and use tooltip utils. This util useful for any tooltip customization in Power BI visuals.

## Requirements

To use the package, you should have the following things:

- [node.js](#) (we recommend the latest LTS version)
- [npm](#) (the minimal supported version is 3.0.0)
- The custom visual created by [PowerBI-visuals-tools](#)

## Installation

To install the package, you should run the following command in the directory with your current visual:

```
npm install powerbi-visuals-utils-colorutils --save
```

This command installs the package and adds a package as a dependency to your `package.json`

## Usage

The Usage Guide describes a public API of the package. You will find a description and a few examples for each public interface of the package.

This package contains provide you the way to create `TooltipServiceWrapper` and methods to help handle tooltip actions. It uses tooltip interfaces - `ITooltipServiceWrapper`, `TooltipEventArgs`, `TooltipEnabledDataPoint`.

Also it has specific methods (touch events handlers) related to mobile development: `touchEndEventName`, `touchStartEventName`, `usePointerEvents`.

`TooltipServiceWrapper` provides the simplest way in order to manipulate tooltips.

This module provides the following interface and function:

- [ITooltipServiceWrapper](#)
  - [addTooltip](#)
  - [hide](#)
- [Interfaces](#)
  - [TooltipEventArgs](#)
  - [TooltipEnabledDataPoint](#)
  - [TooltipServiceWrapperOptions](#)
- [Touch events](#)

```
createTooltipServiceWrapper
```

This function creates an instance of `ITooltipServiceWrapper`.

```
function createTooltipServiceWrapper(tooltipService: ITooltipService, rootElement: Element, handleTouchDelay?: number, getEventMethod?: () => MouseEvent): ITooltipServiceWrapper;
```

The `ITooltipService` is available in [IVisualHost](#).

## Example

```
import { createTooltipServiceWrapper } from "powerbi-visuals-utils-tooltiputils";

export class YourVisual implements IVisual {
    // implementation of IVisual.

    constructor(options: VisualConstructorOptions) {
        createTooltipServiceWrapper(
            options.host.tooltipService,
            options.element);

        // returns: an instance of ITooltipServiceWrapper.
    }
}
```

You can take a look at the example code of the custom visual [here](#).

`ITooltipServiceWrapper`

This interface describes public methods of the TooltipService.

```
interface ITooltipServiceWrapper {
    addTooltip<T>(selection: d3.Selection<any, any, any, any>, getTooltipInfoDelegate: (args: TooltipEventArgs<T>) => powerbi.extensibility.VisualTooltipDataItem[], getDataPointIdentity?: (args: TooltipEventArgs<T>) => powerbi.visuals.ISelectionId, reloadTooltipDataOnMouseMove?: boolean): void;
    hide(): void;
}
```

`ITooltipServiceWrapper.addTooltip`

This method adds tooltips to the current selection.

```
addTooltip<T>(selection: d3.Selection<any>, getTooltipInfoDelegate: (args: TooltipEventArgs<T>) => VisualTooltipDataItem[], getDataPointIdentity?: (args: TooltipEventArgs<T>) => ISelectionId, reloadTooltipDataOnMouseMove?: boolean): void;
```

## Example

```

import { createTooltipServiceWrapper, TooltipEventArgs, ITooltipServiceWrapper, TooltipEnabledDataPoint } from
"powerbi-visuals-utils-tooltiputils";

let bodyElement = d3.select("body");

let element = bodyElement
    .append("div")
    .style({
        "background-color": "green",
        "width": "150px",
        "height": "150px"
    })
    .classed("visual", true)
    .data([{
        tooltipInfo: [
            {
                displayName: "Power BI",
                value: 2016
            }
        ]
    }]);
}

let tooltipServiceWrapper: ITooltipServiceWrapper = createTooltipServiceWrapper(tooltipService,
bodyElement.get(0)); // tooltipService is from the IVisualHost.

tooltipServiceWrapper.addTooltip<TooltipEnabledDataPoint>(element, (eventArgs:
TooltipEventArgs<TooltipEnabledDataPoint>) => {
    return eventArgs.data.tooltipInfo;
});

// You will see a tooltip if you mouseover the element.

```

You can take a look at the example code of the custom visual [here](#).

Also pay attention at following example of tooltip customization in Gantt custom visual [here](#)

`ITooltipServiceWrapper.hide`

This method hides the tooltip.

`hide(): void;`

## Example

```

import {createTooltipServiceWrapper} from "powerbi-visuals-utils-tooltiputils";

let tooltipServiceWrapper = createTooltipServiceWrapper(options.host.tooltipService, options.element); // 
options are from the VisualConstructorOptions.

tooltipServiceWrapper.hide();

```

`Interfaces`

This interfaces are used during TooltipServiceWrapper creation and it's usage. Also they were mentioned in examples from previous topic [here](#).

`TooltipEventArgs`

```
interface TooltipEventArgs<TData> {
    data: TData;
    coordinates: number[];
    elementCoordinates: number[];
    context: HTMLElement;
    isTouchEvent: boolean;
}
```

TooltipEnabledDataPoint

```
interface TooltipEnabledDataPoint {
    tooltipInfo?: powerbi.extensibility.VisualTooltipDataItem[];
}
```

TooltipServiceWrapperOptions

```
interface TooltipServiceWrapperOptions {
    tooltipService: ITooltipService;
    rootElement: Element;
    handleTouchDelay: number;
    getEventMethod?: () => MouseEvent;
```

Touch events

Now tooltip utils has ability to handle several touch events useful for mobile development.

touchStartEventName

```
function touchStartEventName(): string
```

This method returns touch start event name.

touchEndEventName

```
function touchEndEventName(): string
```

This method returns touch start event name.

usePointerEvents

```
function usePointerEvents(): boolean
```

This method returns is current touchStart event related to pointer or not.

# DataViewUtils

3/13/2020 • 8 minutes to read • [Edit Online](#)

The `DataViewUtils` is a set of functions and classes to simplify parsing of the `DataView` object for Power BI visuals

## Installation

To install the package, you should run the following command in the directory with your current custom visual:

```
npm install powerbi-visuals-utils-dataviewutils --save
```

This command installs the package and adds a package as a dependency to your `package.json`

## DataRoleHelper

The `DataRoleHelper` provides functions to check roles of the `dataView` object.

The module provides the following functions:

### `getMeasureIndexOfRole`

This function finds the measure by role name and returns its index.

```
function getMeasureIndexOfRole(grouped: DataViewValueColumnGroup[], roleName: string): number;
```

Example:

```

import powerbi from "powerbi-visuals-api";
import DataViewValueColumnGroup = powerbi.DataViewValueColumnGroup;
import { dataRoleHelper } from "powerbi-visuals-utils-dataviewutils";
// ...

// This object is actually a part of the dataView object.
let columnGroup: DataViewValueColumnGroup[] = [{{
    values: [
        {
            source: {
                displayName: "Microsoft",
                roles: {
                    "company": true
                }
            },
            values: []
        },
        {
            source: {
                displayName: "Power BI",
                roles: {
                    "product": true
                }
            },
            values: []
        }
    ]
}];

dataRoleHelper.getMeasureIndexOfRole(columnGroup, "product");

// returns: 1

```

## getCategoryIndexOfRole

This function finds the category by role name and returns its index.

```
function getCategoryIndexOfRole(categories: DataViewCategoryColumn[], roleName: string): number;
```

Example:

```

import powerbi from "powerbi-visuals-api";
import DataViewCategoryColumn = powerbi.DataViewCategoryColumn;
import { dataRoleHelper } from "powerbi-visuals-utils-dataviewutils";
// ...

// This object is actually a part of the dataView object.
let categoryGroup: DataViewCategoryColumn[] = [
  {
    source: {
      displayName: "Microsoft",
      roles: {
        "company": true
      }
    },
    values: []
  },
  {
    source: {
      displayName: "Power BI",
      roles: {
        "product": true
      }
    },
    values: []
  }
];
dataRoleHelper.getCategoryIndexOfRole(categoryGroup, "product");
// returns: 1

```

## hasRole

This function checks if the provided role is defined in the metadata.

```
function hasRole(column: DataViewMetadataColumn, name: string): boolean;
```

Example:

```

import powerbi from "powerbi-visuals-api";
import DataViewMetadataColumn = powerbi.DataViewMetadataColumn;
import { dataRoleHelper } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let metadata: DataViewMetadataColumn = {
  displayName: "Microsoft",
  roles: {
    "company": true
  }
};

DataRoleHelper.hasRole(metadata, "company");
// returns: true

```

## hasRoleInDataView

This function checks if the provided role is defined in the dataView.

```
function hasRoleInDataView(dataView: DataView, name: string): boolean;
```

Example:

```

import powerbi from "powerbi-visuals-api";
import DataView = powerbi.DataView;
import { dataRoleHelper } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let dataView: DataView = {
    metadata: {
        columns: [
            {
                displayName: "Microsoft",
                roles: {
                    "company": true
                }
            },
            {
                displayName: "Power BI",
                roles: {
                    "product": true
                }
            }
        ]
    }
};

DataRoleHelper.hasRoleInDataView(dataView, "product");

// returns: true

```

## hasRoleInValueColumn

This function checks if the provided role is defined in the value column.

```
function hasRoleInValueColumn(valueColumn: DataViewValueColumn, name: string): boolean;
```

Example:

```

import powerbi from "powerbi-visuals-api";
import DataViewValueColumn = powerbi.DataViewValueColumn;
import { dataRoleHelper } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let valueColumn: DataViewValueColumn = {
    source: {
        displayName: "Microsoft",
        roles: {
            "company": true
        }
    },
    values: []
};

dataRoleHelper.hasRoleInValueColumn(valueColumn, "company");

// returns: true

```

# DataViewObjects

The `DataViewObjects` provides functions to extract values of the objects.

The module provides the following functions:

## getValue

This function returns the value of the given object.

```
function getValue<T>(objects: DataViewObjects, propertyId: DataViewObjectPropertyIdentifier, defaultValue?: T): T;
```

Example:

```
import powerbi from "powerbi-visuals-api";
import DataViewObjectPropertyIdentifier = powerbi.DataViewObjectPropertyIdentifier;
import { dataViewObjects } from "powerbi-visuals-utils-dataviewutils";

let property: DataViewObjectPropertyIdentifier = {
    objectName: "microsoft",
    propertyName: "bi"
};

// This object is actually a part of the dataView object.
let objects: powerbi.DataViewObjects = {
    "microsoft": {
        "windows": 5,
        "bi": "Power"
    }
};

dataViewObjects.getValue(objects, property);

// returns: Power
```

## getObject

This function returns an object of the given object.

```
function getObject(objects: DataViewObjects, objectName: string, defaultValue?: IDataViewObject): IDataViewObject;
```

Example:

```
import { dataViewObjects } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let objects: powerbi.DataViewObjects = {
    "microsoft": {
        "windows": 5,
        "bi": "Power"
    }
};

dataViewObjects.getObject(objects, "microsoft");

/* returns: {
    "bi": "Power",
    "windows": 5
}*/
```

## getFillColor

This function returns a solid color of the objects.

```
function getFillColor(objects: DataViewObjects, propertyId: DataViewObjectPropertyIdentifier, defaultColor?: string): string;
```

Example:

```
import powerbi from "powerbi-visuals-api";
import DataViewObjectPropertyIdentifier = powerbi.DataViewObjectPropertyIdentifier;
import { dataViewObjects } from "powerbi-visuals-utils-dataviewutils";

let property: DataViewObjectPropertyIdentifier = {
    objectName: "power",
    propertyName: "fillColor"
};

// This object is actually part of the dataView object.
let objects: powerbi.DataViewObjects = {
    "power": {
        "fillColor": {
            "solid": {
                "color": "yellow"
            }
        },
        "bi": "Power"
    }
};

dataViewObjects.getFillColor(objects, property);

// returns: yellow
```

## getCommonValue

This function is a universal function for retrieving the color or value of a given object.

```
function getCommonValue(objects: DataViewObjects, propertyId: DataViewObjectPropertyIdentifier, defaultValue?: any): any;
```

Example:

```

import powerbi from "powerbi-visuals-api";
import DataViewObjectPropertyIdentifier = powerbi.DataViewObjectPropertyIdentifier;
import { dataViewObjects } from "powerbi-visuals-utils-dataviewutils";

let colorProperty: DataViewObjectPropertyIdentifier = {
    objectName: "power",
    propertyName: "fillColor"
};

let biProperty: DataViewObjectPropertyIdentifier = {
    objectName: "power",
    propertyName: "bi"
};

// This object is actually part of the dataView object.
let objects: powerbi.DataViewObjects = {
    "power": {
        "fillColor": {
            "solid": {
                "color": "yellow"
            }
        },
        "bi": "Power"
    }
};

dataViewObjects.getCommonValue(objects, colorProperty); // returns: yellow
dataViewObjects.getCommonValue(objects, biProperty); // returns: Power

```

## DataViewObject

The `DataViewObject` provides functions to extract value of the object.

The module provides the following functions:

### **getValue**

This function returns a value of the object by property name.

```
function getValue<T>(object: IDataViewObject, propertyName: string, defaultValue?: T): T;
```

Example:

```

import { dataViewObject } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let object: powerbi.DataViewObject = {
    "windows": 5,
    "microsoft": "Power BI"
};

dataViewObject.getValue(object, "microsoft");

// returns: Power BI

```

### **getFillColorByPropertyName**

This function returns a solid color of the object by property name.

```
function getFillColorByPropertyName(object: IDataViewObject, propertyName: string, defaultColor?: string): string;
```

Example:

```
import { dataViewObject } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let object: powerbi.DataViewObject = {
    "Windows": 5,
    "fillColor": {
        "solid": {
            "color": "green"
        }
    }
};

dataViewObject.getFillColorByPropertyName(object, "fillColor");

// returns: green
```

## converterHelper

The `converterHelper` provides functions to check properties of the dataView.

The module provides the following functions:

### categoryIsAlsoSeriesRole

This function checks if the category is also series.

```
function categoryIsAlsoSeriesRole(dataView: DataViewCategorical, seriesRoleName: string, categoryRoleName: string): boolean;
```

Example:

```
import powerbi from "powerbi-visuals-api";
import DataViewCategorical = powerbi.DataViewCategorical;
import { converterHelper } from "powerbi-visuals-utils-dataviewutils";
// ...

// This object is actually part of the dataView object.
let categorical: DataViewCategorical = {
    categories: [
        {
            source: {
                displayName: "Microsoft",
                roles: {
                    "power": true,
                    "bi": true
                }
            },
            values: []
        }
    ];
};

converterHelper.categoryIsAlsoSeriesRole(categorical, "power", "bi");

// returns: true
```

## getSeriesName

This function returns a name of the series.

```
function getSeriesName(source: DataViewMetadataColumn): PrimitiveValue;
```

Example:

```
import powerbi from "powerbi-visuals-api";
import DataViewMetadataColumn = powerbi.DataViewMetadataColumn;
import { converterHelper } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let metadata: DataViewMetadataColumn = {
    displayName: "Microsoft",
    roles: {
        "power": true,
        "bi": true
    },
    groupName: "Power BI"
};

converterHelper.getSeriesName(metadata);

// returns: Power BI
```

### isImageUrlColumn

This function checks if the column contains an image url.

```
function isImageUrlColumn(column: DataViewMetadataColumn): boolean;
```

Example:

```
import powerbi from "powerbi-visuals-api";
import DataViewMetadataColumn = powerbi.DataViewMetadataColumn;
import { converterHelper } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let metadata: DataViewMetadataColumn = {
    displayName: "Microsoft",
    type: {
        misc: {
            imageUrl: true
        }
    }
};

converterHelper.isImageUrlColumn(metadata);

// returns: true
```

### isWebUrlColumn

This function checks if the column contains a web url.

```
function isWebUrlColumn(column: DataViewMetadataColumn): boolean;
```

Example:

```

import powerbi from "powerbi-visuals-api";
import DataViewMetadataColumn = powerbi.DataViewMetadataColumn;
import { converterHelper } from "powerbi-visuals-utils-dataviewutils";

// This object is actually a part of the dataView object.
let metadata: DataViewMetadataColumn = {
    displayName: "Microsoft",
    type: {
        misc: {
            webUrl: true
        }
    }
};

converterHelper.isWebUrlColumn(metadata);

// returns: true

```

## hasImageUrlColumn

This function checks if the dataView has a column with image url.

```
function hasImageUrlColumn(dataView: DataView): boolean;
```

Example:

```

import DataView = powerbi.DataView;
import converterHelper = powerbi.extensibility.utils.dataview.converterHelper;

// This object is actually part of the dataView object.
let dataView: DataView = {
    metadata: {
        columns: [
            {
                displayName: "Microsoft"
            },
            {
                displayName: "Power BI",
                type: {
                    misc: {
                        imageUrl: true
                    }
                }
            }
        ]
    }
};

converterHelper.hasImageUrlColumn(dataView);

// returns: true

```

## DataViewObjectsParser

The `DataViewObjectsParser` provides the simplest way to parse properties of the formatting panel.

The class provides the following methods:

### getDefault

This static method returns an instance of `DataViewObjectsParser`.

```
static getDefault(): DataViewObjectsParser;
```

Example:

```
import { dataViewObjectsParser } from "powerbi-visuals-utils-dataviewutils";
// ...

dataViewObjectsParser.getDefault();

// returns: an instance of the DataViewObjectsParser
```

## parse

This method parses properties of the formatting panel and returns an instance of `DataViewObjectsParser`.

```
static parse<T extends DataViewObjectsParser>(dataView: DataView): T;
```

Example:

```
import powerbi from "powerbi-visuals-api";
import IVisual = powerbi.extensibility.IVisual;
import VisualUpdateOptions = powerbi.extensibility.visual.VisualUpdateOptions;
import { dataViewObjectsParser } from "powerbi-visuals-utils-dataviewutils";

/**
 * This class describes formatting panel properties.
 * Name of the property should match its name described in the capabilities.
 */
class DataPointProperties {
    public fillColor: string = "red"; // This value is a default value of the property.
}

class PropertiesParser extends dataViewObjectsParser.DataViewObjectsParser {
    /**
     * This property describes a group of properties.
     */
    public dataPoint: DataPointProperties = new DataPointProperties();
}

export class YourVisual extends IVisual {
    // implementation of the IVisual.

    private propertiesParser: PropertiesParser;

    public update(options: VisualUpdateOptions): void {
        // Parses properties.
        this.propertiesParser = PropertiesParser.parse<PropertiesParser>(options.dataViews[0]);

        // You can use the properties after parsing
        console.log(this.propertiesParser.dataPoint.fillColor); // returns "red" as default value, it will be
        updated automatically after any change of the formatting panel.
    }
}
```

## enumerateObjectInstances

This static method enumerates properties and returns an instance of `VisualObjectInstanceEnumeration`.

Execute it in `enumerateObjectInstances` method of the visual.

```
static enumerateObjectInstances(dataViewObjectParser: dataViewObjectsParser.DataViewObjectsParser, options: EnumerateVisualObjectInstancesOptions): VisualObjectInstanceEnumeration;
```

Example:

```
import powerbi from "powerbi-visuals-api";
import IVisual = powerbi.extensibility.IVisual;
import EnumerateVisualObjectInstancesOptions = powerbi.EnumerateVisualObjectInstancesOptions;
import VisualObjectInstanceEnumeration = powerbi.VisualObjectInstanceEnumeration;
import VisualUpdateOptions = powerbi.extensibility.visual.VisualUpdateOptions;
import { dataViewObjectsParser } from "powerbi-visuals-utils-dataviewutils";

/**
 * This class describes formatting panel properties.
 * Name of the property should match its name described in the capabilities.
 */
class DataPointProperties {
    public fillColor: string = "red";
}

class PropertiesParser extends dataViewObjectsParser.DataViewObjectsParser {
    /**
     * This property describes a group of properties.
     */
    public dataPoint: DataPointProperties = new DataPointProperties();
}

export class YourVisual extends IVisual {
    // implementation of the IVisual.

    private propertiesParser: PropertiesParser;

    public update(options: VisualUpdateOptions): void {
        // Parses properties.
        this.propertiesParser = PropertiesParser.parse<PropertiesParser>(options.dataViews[0]);
    }

    /**
     * This method will be executed only if the formatting panel is open.
     */
    public enumerateObjectInstances(options: EnumerateVisualObjectInstancesOptions):
    VisualObjectInstanceEnumeration {
        return PropertiesParser.enumerateObjectInstances(this.propertiesParser, options);
    }
}
```

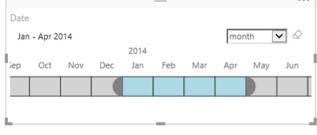
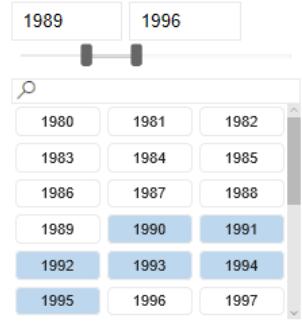
# Samples of Power BI visuals

5/11/2020 • 3 minutes to read • [Edit Online](#)

You can download, use, and modify these Power BI visuals from GitHub. These samples illustrate how to handle common situations when developing with Power BI.

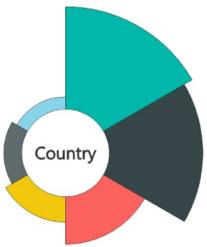
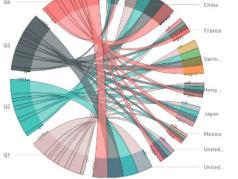
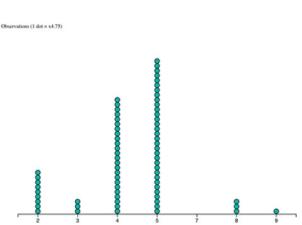
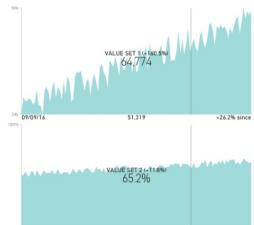
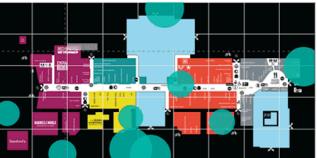
## Slicers

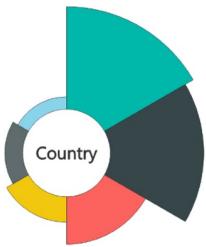
A slicer narrows the portion of data shown in other visualizations in a report. Slicers are one of several ways to filter data in Power BI.

		
<p><b>Chidet Slicer</b> Display image or text buttons that act as an in-canvas filter on other visuals</p>	<p><b>Timeline slicer</b> Graphical date range selector that filters by date</p>	<p><b>Slicer sample</b> Demonstrates the use of the Advanced Filtering API</p>

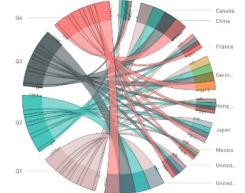
## Charts

Be inspired with our gallery, including bar charts, pie charts, Word Cloud, and others.

		
<p><b>Aster Plot</b> A twist on a standard donut chart that uses a second value to drive sweep angle</p>	<p><b>Bullet chart</b> A bar chart with extra visual elements that provide context useful for tracking goals</p>	<p><b>Chord</b> A graphical method that displays the relationships between data in a matrix</p>
		



Var LE3 %



### Dot plot

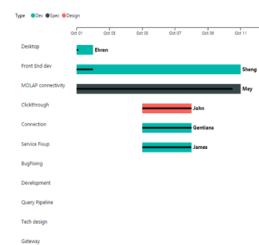
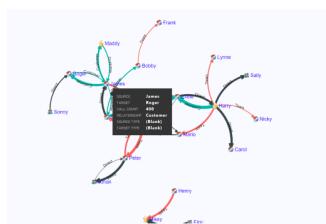
Shows the distribution of frequencies in a great looking way

### Dual KPI

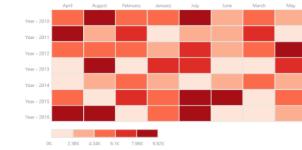
Efficiently visualizes two measures over time, showing their trend on a joint timeline

### Enhanced Scatter

Improvements on the existing scatter chart



Product sales by year/month



### Force Graph

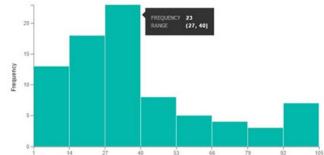
Force layout diagram with curved path, which is useful to show connections between entities

### Gantt

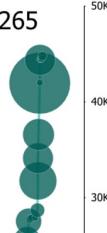
A bar chart that illustrates a project timeline or schedule with resources

### Table Heatmap

Compare data easily and intuitively using colors in a table



Total features 265



### Histogram chart

Visualizes the distribution of data over a continuous interval or certain time period

### LineDot chart

An animated line chart with animated dots that engage an audience with data

### Mekko chart

A mix of 100% stacked column chart and 100% stacked bar chart combined into one view



Column-Based Data Model with Metrics as Rows



### Multi KPI

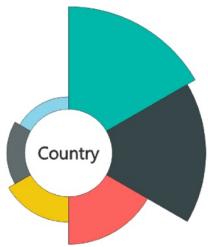
A powerful Multi KPI visualization with a key KPI along with multiple sparklines of supporting data

### Power KPI

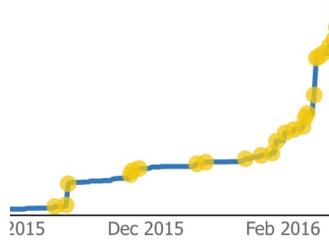
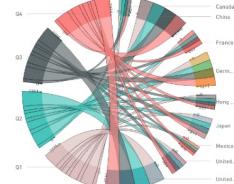
A powerful KPI Indicator with multi-line chart and labels for current date, value, and variances

### Power KPI Matrix

Monitor balanced scorecards and unlimited number of metrics and KPIs in a compact, easy to read list

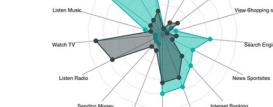


Var LE3 %



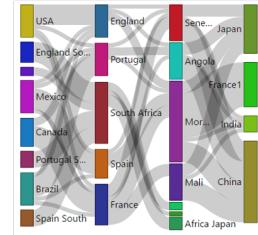
### Pulse chart

This line chart annotated with key events is perfect for telling stories with data



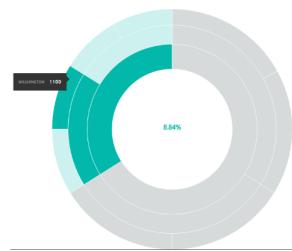
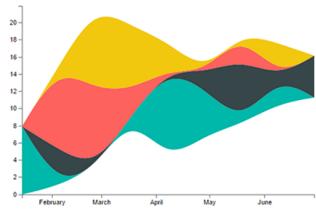
### Radar chart

Presents multiple measures plotted over a categorical axis, which is useful to compare attributes



### Sankey chart

Flow diagram where the width of the series is proportional to the quantity of the flow



### Stream graph

A stacked area chart with smooth interpolation, which is often used to display values over time

### Sunburst chart

Multilevel donut chart for visualizing hierarchical data

### Tornado chart

Compare the relative importance of variables between two groups

### Tornado chart

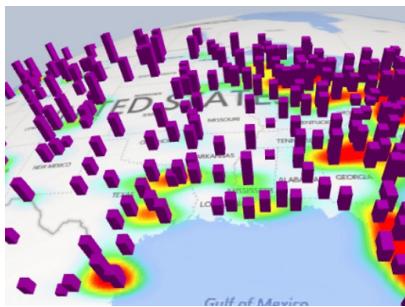


### Word Cloud

Create a fun visual from frequent text in your data

## WebGL

WebGL lets web content use an API based on OpenGL ES 2.0 to do 2D and 3D rendering in an HTML canvas.



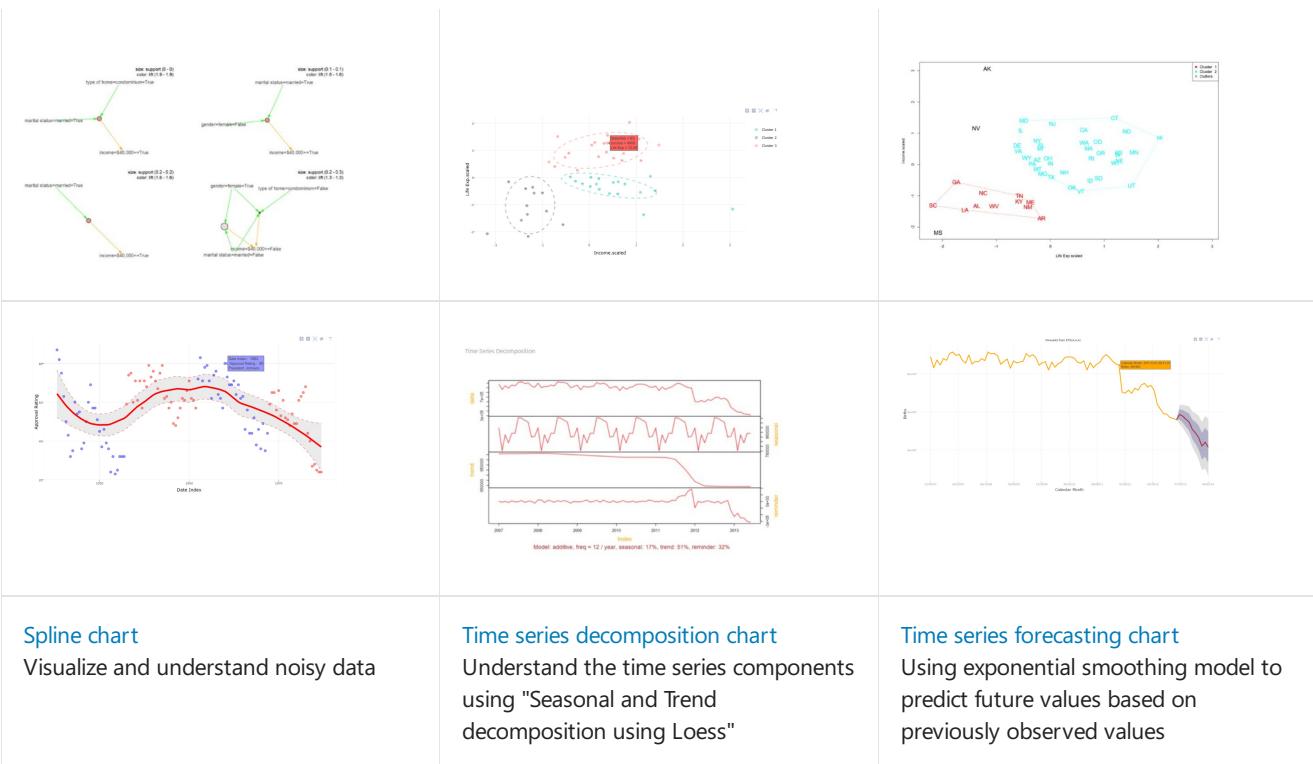
## Globe Map

Plot locations on an interactive 3D map

# R visuals

These samples demonstrate how to harness the analytic and visual power of R visuals and R scripts.

<h3>Association rules</h3> <p>Uncover relationships between seemingly unrelated data using if-then statements</p>	<h3>Clustering</h3> <p>Find similarity groups in your data using k-means algorithm</p>	<h3>Clustering with outliers</h3> <p>Find similarity groups and outliers in your data</p>
<h3>Correlation plot</h3> <p>Highlight the most correlated variables in a data table</p>	<h3>Decision tree chart</h3> <p>Schematic tree-shaped diagram for determining statistical probability using recursive partitioning</p>	<h3>Forecasting TBATS</h3> <p>Time-series forecasting for series that have multiple seasonalities using the TBATS model</p>
<h3>Forecasting with ARIMA</h3> <p>Predict future values based on historical data using Autoregressive Integrated Moving Avg (ARIMA)</p>	<h3>Funnel plot</h3> <p>Find outliers in your data using a funnel plot</p>	<h3>Outliers detection</h3> <p>Find outliers in your data using the most appropriate method and plot</p>



## Next steps

To try out creating Power BI visuals, see [Tutorial: Developing a Power BI visual](#).

# Create an SSL certificate

5/13/2020 • 3 minutes to read • [Edit Online](#)

This article describes how to generate and install Secure Sockets Layer (SSL) certificates for Power BI visuals.

For the Windows, macOS X, and Linux procedures, you must have the Power BI Visual Tools **pbviz** package installed. For more information, see [Set up the developer environment](#).

## Create a certificate on Windows

To generate a certificate by using the PowerShell cmdlet `New-SelfSignedCertificate` on Windows 8 and later, run the following command:

```
pbviz --install-cert
```

For Windows 7, the `pbviz` tool requires the OpenSSL utility to be available from the command line. To install OpenSSL, go to [OpenSSL](#) or [OpenSSL Binaries](#).

For more information and instructions for installing a certificate, see [Create and install a certificate for Windows](#).

## Create a certificate on macOS X

The OpenSSL utility is usually available in the macOS X operating system.

You can also install the OpenSSL utility by running either of the following commands:

- From the *Brew* package manager:

```
brew install openssl  
brew link openssl --force
```

- By using *MacPorts*:

```
sudo port install openssl
```

After you install the OpenSSL utility, run the following command to generate a new certificate:

```
pbviz --install-cert
```

For more information and instructions, see [Create and install a certificate for OS X](#).

## Create a certificate on Linux

The OpenSSL utility is usually available in the Linux operating system.

Before you begin, run the following commands to make sure `openssl` and `certutil` are installed:

```
which openssl  
which certutil
```

If `openssl` and `certutil` aren't installed, install the `openssl` and `libnss3` utilities.

## Create the SSL configuration file

Create a file called `/tmp/openssl.cnf` that contains the following text:

```
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, nonRepudiation, keyEncipherment, dataEncipherment
subjectAltName = @alt_names

[ alt_names ]
DNS.1=localhost
```

## Generate root certificate authority

To generate root certificate authority (CA) to sign local certificates, run the following commands:

```
touch $HOME/.rnd
openssl req -x509 -nodes -new -sha256 -days 1024 -newkey rsa:2048 -keyout /tmp/local-root-ca.key -out
/tmp/local-root-ca.pem -subj "/C=US/CN=Local Root CA/O=Local Root CA"
openssl x509 -outform pem -in /tmp/local-root-ca.pem -out /tmp/local-root-ca.crt
```

## Generate a certificate for localhost

To generate a certificate for `localhost` using the generated CA and `openssl.cnf`, run the following commands:

```
PBIVIZ=`which pbviz`
PBIVIZ=`dirname $PBIVIZ`
PBIVIZ="$PBIVIZ/../lib/node_modules/powerbi-visuals-tools/certs"
# Make sure that $PBIVIZ contains the correct certificate directory path. ls $PBIVIZ should list 'blank' file.
openssl req -new -nodes -newkey rsa:2048 -keyout $PBIVIZ/PowerBIVisualTest_private.key -out
$PBIVIZ/PowerBIVisualTest.csr -subj "/C=US/O=PowerBI Visuals/CN=localhost"
openssl x509 -req -sha256 -days 1024 -in $PBIVIZ/PowerBIVisualTest.csr -CA /tmp/local-root-ca.pem -CAkey
/tmp/local-root-ca.key -CAcreateserial -extfile /tmp/openssl.cnf -out $PBIVIZ/PowerBIVisualTest_public.crt
```

## Add root certificates

To add a root certificate to the Chrome browser's database, run:

```
certutil -A -n "Local Root CA" -t "CT,C,C" -i /tmp/local-root-ca.pem -d sql:$HOME/.pki/nssdb
```

To add a root certificate to the Mozilla Firefox browser's database, run:

```
for certDB in $(find $HOME/.mozilla* -name "cert*.db")
do
certDir=$(dirname ${certDB});
certutil -A -n "Local Root CA" -t "CT,C,C" -i /tmp/local-root-ca.pem -d sql:${certDir}
done
```

To add a system-wide root certificate, run:

```
sudo cp /tmp/local-root-ca.pem /usr/local/share/ca-certificates/
sudo update-ca-certificates
```

## Remove root certificates

To remove a root certificate, run:

```
sudo rm /usr/local/share/ca-certificates/local-root-ca.pem  
sudo update-ca-certificates --fresh
```

## Generate a certificate manually

You can also generate an SSL certificate manually using OpenSSL. You can specify any tools to generate your certificates.

If the OpenSSL utility is already installed, generate a new certificate by running:

```
openssl req -x509 -newkey rsa:4096 -keyout PowerBIVisualTest_private.key -out PowerBIVisualTest_public.crt -  
days 365
```

You can usually find the `PowerBI-visuals-tools` web server certificates by running one of the following commands:

- For the global instance of the tools:

```
%appdata%\npm\node_modules\PowerBI-visuals-tools\certs
```

- For the local instance of the tools:

```
<Power BI visual project root>\node_modules\PowerBI-visuals-tools\certs
```

### PEM format

If you use the Privacy Enhanced Mail (PEM) certificate format, save the certificate file as `PowerBIVisualTest_public.crt`, and save the private key as `PowerBIVisualTest_private.key`.

### PFX format

If you use the Personal Information Exchange (PFX) certificate format, save the certificate file as `PowerBIVisualTest_public.pfx`.

If your PFX certificate file requires a passphrase:

- In the config file, specify:

```
\PowerBI-visuals-tools\config.json
```

- In the `server` section, specify the passphrase by replacing the <YOUR PASSPHRASE> placeholder:

```
"server":{  
    "root":"webRoot",  
    "assetsRoute":"/assets",  
    "privateKey":"certs/PowerBIVisualTest_private.key",  
    "certificate":"certs/PowerBIVisualTest_public.crt",  
    "pfx":"certs/PowerBIVisualTest_public.pfx",  
    "port":"8080",  
    "passphrase":"<YOUR PASSPHRASE>"  
}
```

## Next steps

- [Develop a Power BI visual](#)

- [Power BI visuals samples](#)
- [Publish a Power BI visual to AppSource](#)