

# Power BI visuals documentation

Develop your own Power BI visuals, to be used by you, your organization, or the entire Power BI community. Our documentation provides the information you need.

## Get started



### GET STARTED

[Develop your own Power BI visual](#)

[Where to find Power BI visuals](#)

[Power BI visual project structure](#)

[Frequently asked questions](#)

## Develop a Power BI visual



### TUTORIAL

[Tutorial: Develop a Power BI visual](#)

[Tutorial: Adding formatting options to a Power BI visual](#)



### CONCEPT

[Guidelines for Power BI visuals](#)

## Publish Power BI visuals



### CONCEPT

[Publish Power BI visuals to Partner Center](#)

[Get a Power BI visual certified](#)

## References and resources



### REFERENCE

[Samples of Power BI visuals ↗](#)

[Git repository ↗](#)

# What are custom visuals in Power BI and where can I get them?

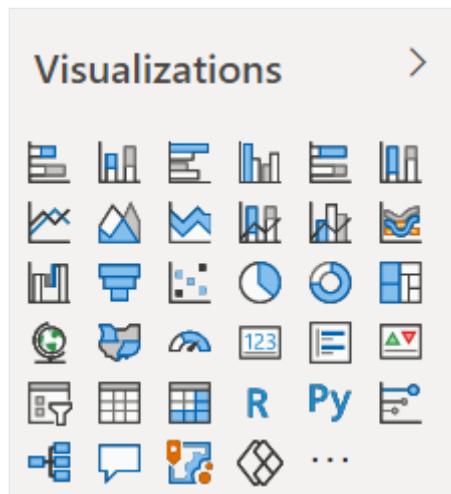
Article • 11/02/2024

Power BI [visuals](#) come from three main sources:

- [Core](#) visuals are readily available on the visualization pane.
- You can [download or import](#) visuals from Microsoft [AppSource](#) or Power BI.
- You can create your own [custom visuals](#).

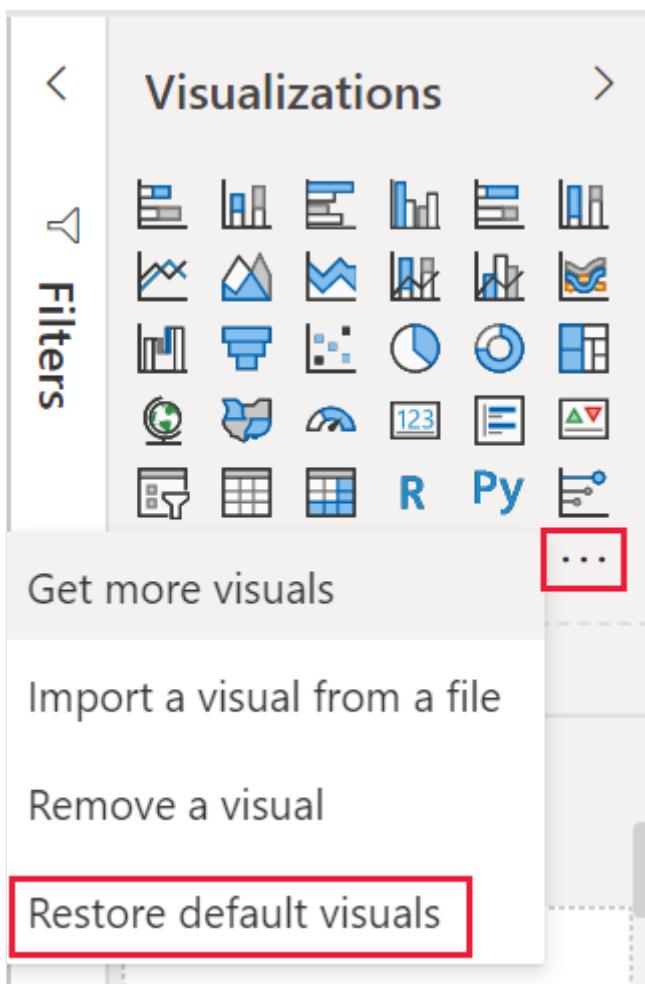
## Core Power BI visuals

Power BI comes with many out-of-the box visuals. These Power BI 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.



To remove 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, select the ellipsis and then select [Restore default visuals](#).



## AppSource Power BI visuals

You can find many more Power BI visuals in [AppSource](#). AppSource is the place to find apps, add-ins, and extensions for your Microsoft software. It connects millions of people who use products such as Microsoft 365, Azure, Dynamics 365, Cortana, and Power BI, to solutions that help them work more efficiently and with more insight than before.

Microsoft and community members develop Power BI visuals for public benefit, and publish them to the AppSource. Microsoft tests and approves these Power BI visuals for functionality and quality. You can download these visuals and add them to your Power BI reports.

### ! Note

- By using Power BI visuals created with our SDK, you may be importing data from, or sending data to, a third party or other service outside of your Power BI tenant's geographic area, compliance boundary, or national/regional cloud instance.

- Once Power BI visuals from AppSource are imported, they may be updated automatically without any additional notice.

## Download from AppSource

Downloading visuals from AppSource is free, but each publisher defines their own business and licensing model for their visual. There are three basic types of payment and licensing plans:

- Free visuals that you can download and use without additional costs. These visuals are labeled as *Free*.
- Licensed visuals managed from the [Microsoft 365 admin center](#). These visuals are available in a limited capacity for free with the option to purchase more features. Transactability happens in [AppSource](#) by clicking on the *Buy now* button.
- Visuals that you can download with basic functionality for free, but have additional features available for pay. These visuals have the *additional purchase may be required* label. You can often get a free trial period to test out the full functionality of the visual before paying for it. Transactability and license management for these visuals happen outside of Microsoft platforms.

Once you select the visual, select the **Plans + Pricing** tab to see the plan for that visual. Pricing information is also shown on the left pane.

Plan	Description	Monthly Price	Annual Price
Power BI Visual Contoso Premium	This plan unlock enhanced feature, which will add additional KPIs to the visual.	\$X.XX/user/month	\$X.XX/user/year
Power BI Visual Contoso Basic	This plan supports X+ hierarchies	First month free, then \$X.XX/user/month	First month free, then \$X.XX/user/year

Select **Add** to download the visual. If a free trial period is offered, it will start when you download the visual.

After you *purchase* the licenses for the visual in AppSource with a credit card, you need to *assign* them to yourself or others in the [Microsoft 365 admin center](#).

For more information on how to assign licenses see [Licensing and transactability enforcement](#).

To see how licenses are enforced, see [custom visual licenses](#)

[More questions about assigning and managing licenses?](#)

## 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.

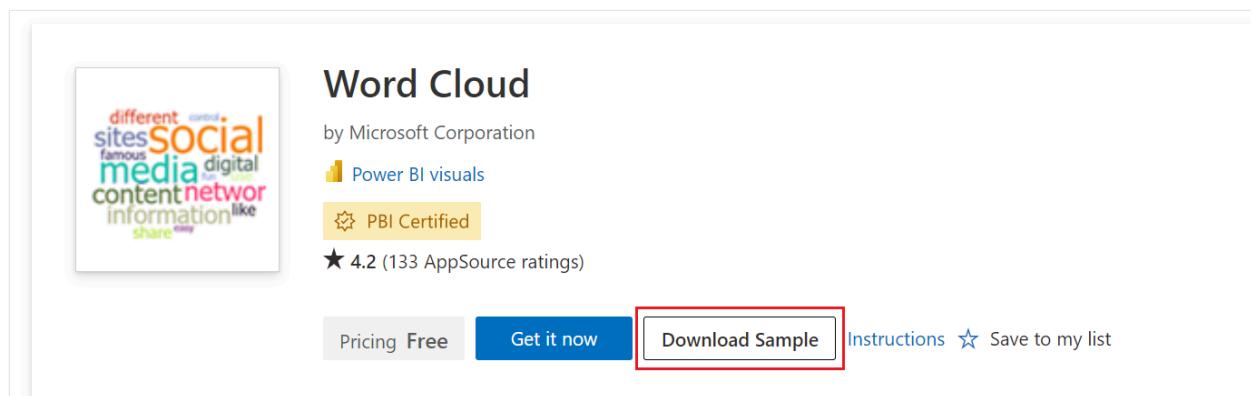
To view the list of certified Power BI visuals, go to [AppSource](#). To submit your own visual for certification, see [Certified Power BI visuals](#).

## Sample reports for Power BI visuals

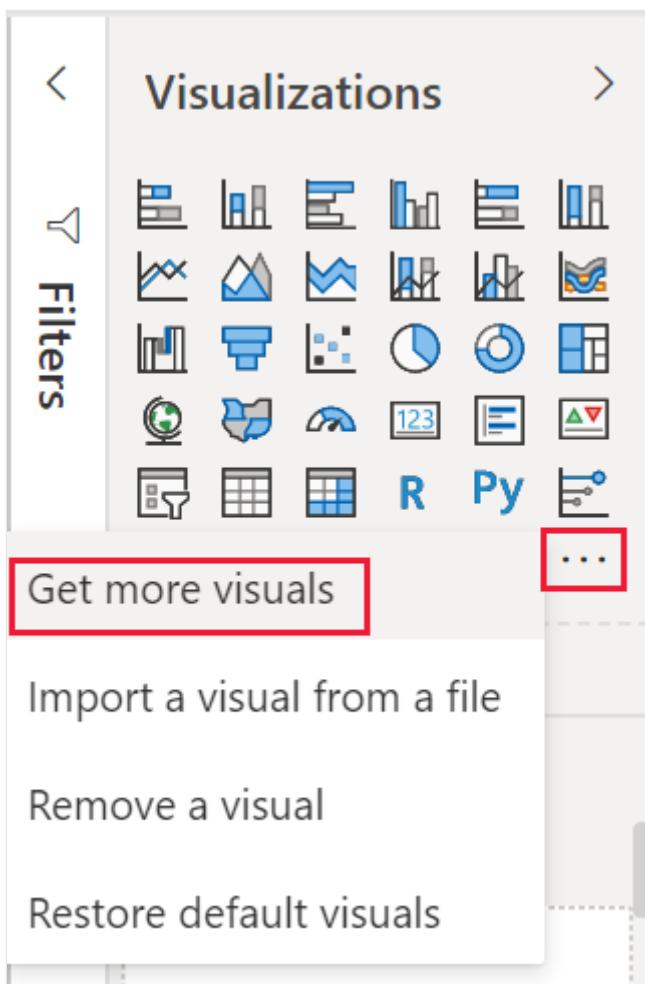
Each Power BI visual on AppSource has a sample report you can download that illustrates how the visual works. To download the sample report, in the [AppSource](#) select a Power BI visual and select the **Download Sample** link.

## Organizational store

Power BI admins can approve and deploy Power BI custom visuals for their organization. Report authors can 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 select the ellipsis, then select **Get more visuals**.



When the *Power BI visuals* window appears, select the **My organization** tab.

[Read more about organizational visuals.](#)

## Custom visual files

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

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

To import a Power BI visual from a file, see [Import a visual file from your local computer into Power BI](#).

If you're a web developer and want to create your own visual and add it to AppSource, you can learn how to [develop a Power BI visual](#) and [publish a custom visual to AppSource](#).

### Warning

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

For some examples of Power BI custom visuals available for downloading on github, see [Examples of Power BI visuals](#).

## Considerations and limitations

Licensed visuals aren't supported in the following environments. Therefore, if licensed visuals are used in these environments, Power BI can't tell the ISV if the user is licensed, nor will it block the visual.

- RS (report server) - no Microsoft Entra ID
- Sovereign or government clouds
- PaaS Power BI embedded App owns data
- Publish to web (P2W)

## Related content

- [Develop a Power BI circle card visual](#)
- [Power BI visuals project structure](#)
- [Guidelines for Power BI visuals](#)
- [Examples of Power BI visuals](#)

More questions? try the [Power BI Community](#) ↗

---

## Feedback

Was this page helpful?



[Provide product feedback](#) ↗ | [Ask the community](#) ↗

# Power BI custom visuals

Article • 11/25/2024

Power BI comes with [core](#) visuals readily available on the visualization pane. You can also [import visuals](#) from Microsoft [AppSource](#) or Power BI.

If none of these visuals meet your specific needs, you can [create your own custom Power BI visual](#) to be used by you, your organization, or the entire Power BI community.

## Power BI visual packages

Power BI visuals are packaged in `.pbviz` files 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 be imported into a Power BI report.

To import a Power BI visual from a file, see [Import a visual file from your local computer into Power BI](#).

## Develop your own custom Power BI visual

If you're a web developer interested in creating your own visual, you can:

- Learn how to set up the necessary [environment](#)
- Learn the basic [structure of Power BI visuals](#)
- Create a simple circle card visual using this [tutorial](#)

To publish the visual you created to AppSource, see [Publish a Power BI visual to AppSource](#).

## Related content

- [Power BI visuals project structure](#)
- [Guidelines for Power BI visuals](#)

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

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Tutorial: Develop a Power BI circle card visual

Article • 02/19/2025

In this tutorial, you develop a Power BI visual named circle card that displays a formatted measure value inside a circle. The circle card visual supports customization of fill color and outline thickness.

In this tutorial, you learn how to:

- [x] Create a development project for your visual.
- [x] Develop your visual with [D3](#) visual elements.
- [x] Configure your visual to process data.
- [x] Configure your visual to adapt to size changes.
- [x] Configure adaptive color and border settings for your visual.

For the full source code of this visual, see [circle card Power BI visual](#).

If you don't have a Power BI account, you can sign up for a free trial on the Power BI website.

## Prerequisites

Before you start developing your Power BI visual, verify that you have everything listed in this section.

- A **Power BI Pro or Premium Per User (PPU)** account. If you don't have one, [sign up for a free trial](#).
- [Visual Studio Code \(VS Code\)](#). VS Code is an ideal Integrated Development Environment (IDE) for developing JavaScript and TypeScript applications.
- [Windows PowerShell](#) version 4 or later (for Windows). Or [Terminal](#) (for Mac).
- An environment ready for developing a Power BI visual. [Set up your environment for developing a Power BI visual](#).
- This tutorial uses the **US Sales Analysis** report. You can [download this report](#) and upload it to Power BI service, or use your own report. If you need more information about Power BI service, and uploading files, refer to the [Get started creating in the Power BI service](#) tutorial.

# Create a development project

In this section, you create a project for the circle card visual.

## ⓘ Note

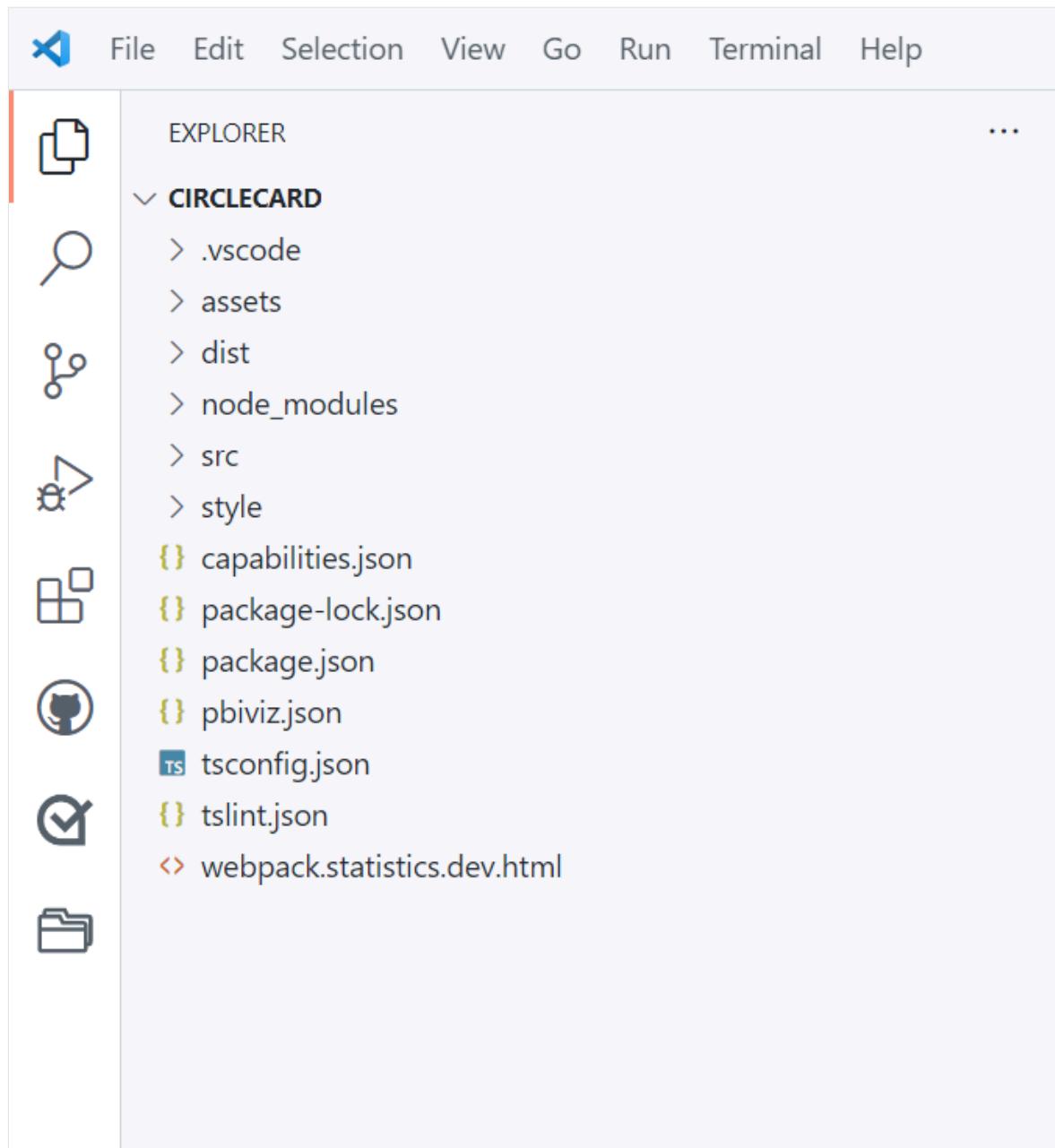
In this tutorial, [Visual Studio Code](#) (VS Code) is used for developing the Power BI visual.

1. Open a new terminal in **VS Code** and navigate to the folder you want to create your project in.
2. Enter the following command in the PowerShell terminal:

```
PowerShell
```

```
pbviz new CircleCard
```

3. Open the *CircleCard* folder in the **VS Code** explorer. (**File > Open Folder**).



For a detailed explanation of the function of each of these files, see [Power BI visual project structure](#).

4. Check the terminal window and confirm that you're in the circleCard directory.

Install the [Power BI visual tools dependencies](#).

```
PowerShell
```

```
npm install
```

### 💡 Tip

To see which dependencies have been installed in your visual, check the `package.json` file.

## 5. Start the circle card visual.

```
PowerShell
```

```
pbviz start
```

Your visual is now running while being hosted on your computer.

### ⓘ Important

Don't close the **PowerShell** window until the end of the tutorial. To stop the visual from running, enter **Ctrl + C** and if prompted to terminate the batch job, enter **Y** and then **Enter**.

## View the visual in the Power BI service

To test the visual in Power BI service, we'll use the **US Sales Analysis** report. You can [download](#) this report and upload it to Power BI service.

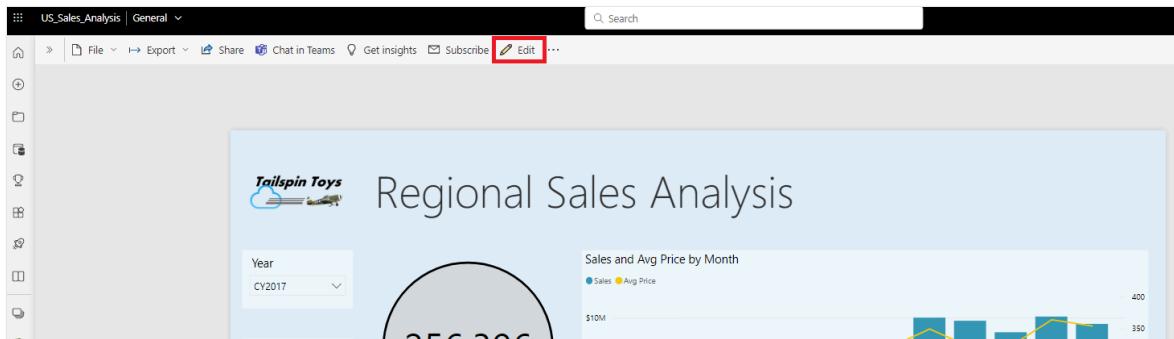
You can also use your own report to test the visual.

### ⓘ Note

Before you continue, verify that you [enabled the visuals developer mode](#).

1. Sign in to [PowerBI.com](#) and open the **US Sales Analysis** report.

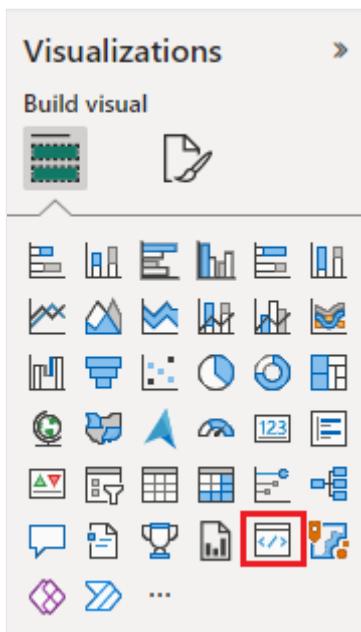
2. Select **Edit**.



3. Create a new page for testing, by clicking on the **New page** button at the bottom of the Power BI service interface.

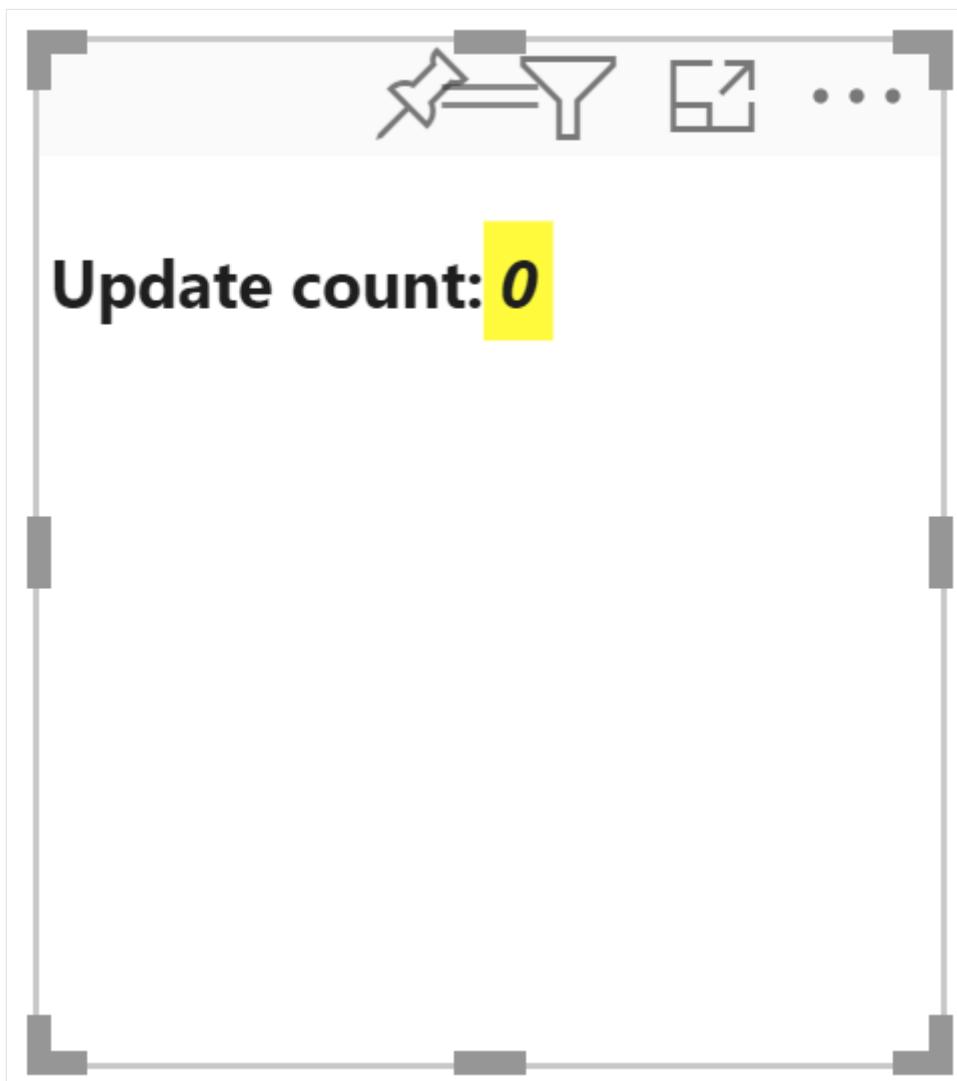


4. From the **Visualizations** pane, select the **Developer Visual**.



This visual represents the custom visual that you're running on your computer. It's only available when the [custom visual debugging](#) setting is enabled.

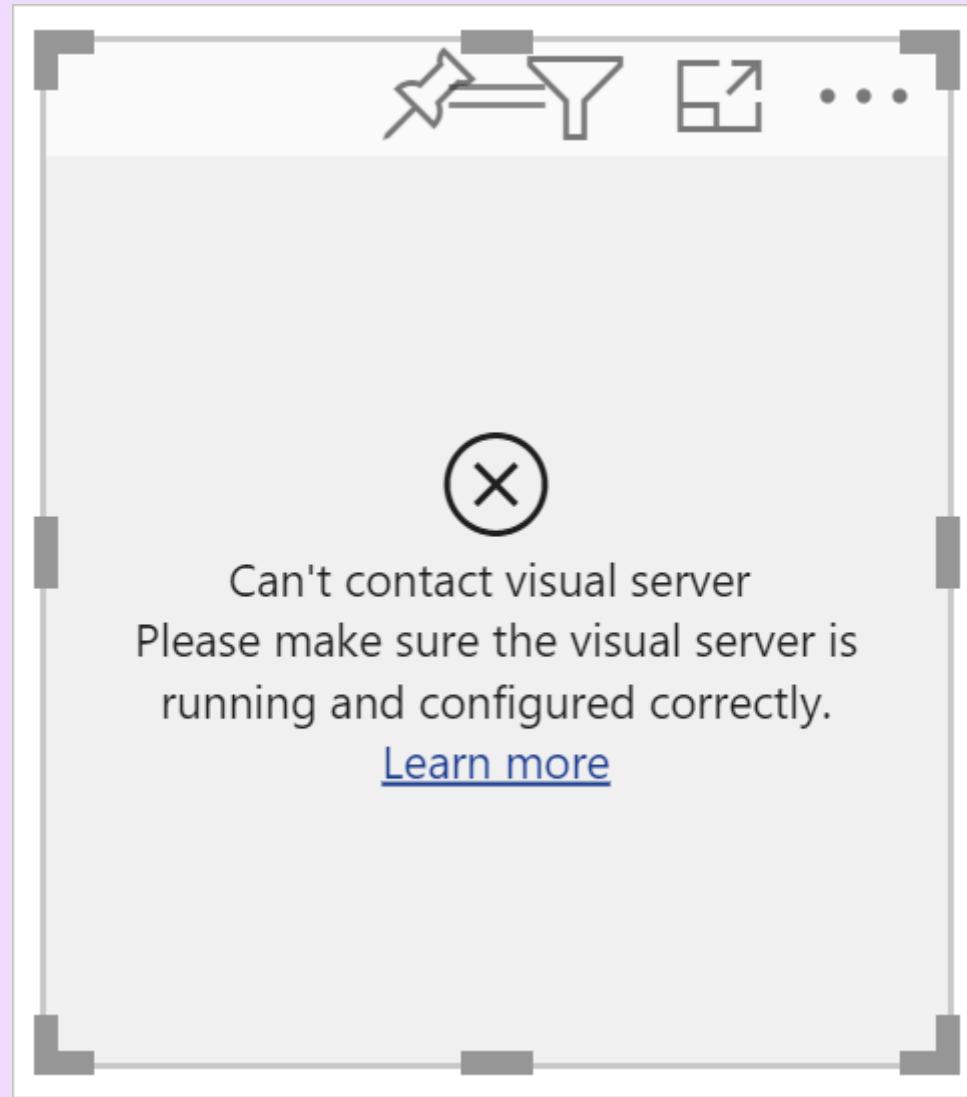
5. Verify that a visual was added to the report canvas.



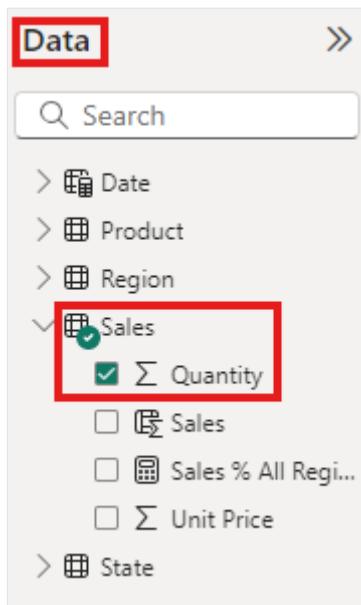
This is a simple visual that displays the number of times its update method has been called. At this stage, the visual does not retrieve any data.

ⓘ Note

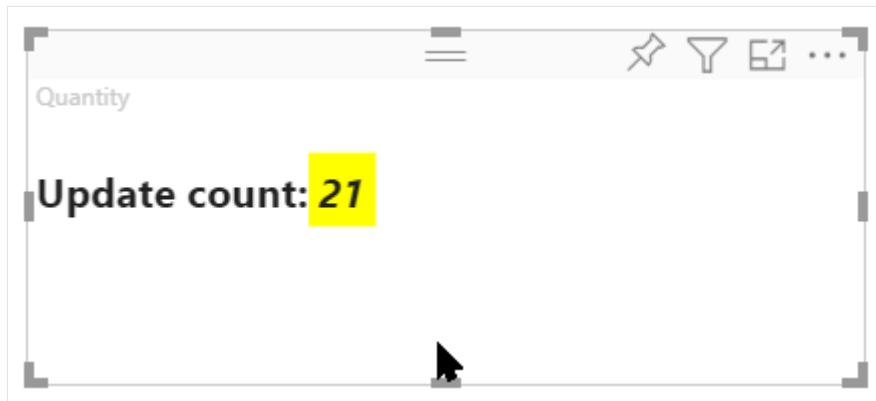
If the visual displays a connection error message, open a new tab in your browser, navigate to <https://localhost:8080/assets>, and authorize your browser to use this address.



6. While the new visual is selected, go to the **Data** pane, expand **Sales**, and select **Quantity**.



7. To test how the visual is responding, resize it and notice that the *Update count* value increments every time you resize the visual.



## Add visual elements and text

In this section you learn how to turn your visual into a circle, and make it display text.

## Modify the visuals file

Set up the `visual.ts` file.

### Tip

To improve readability, it's recommended that you format the document every time you copy code snippets into your project. Right-click anywhere in VS code, and select *Format Document* (or enter `Alt + Shift + F`).

1. In VS Code, in the **Explorer pane**, expand the `src` folder, and select the file `visual.ts`.

The screenshot shows the Visual Studio Code interface. The left sidebar has icons for Explorer, Open Editors, CircleCard, .vscode, assets, dist, node\_modules, and a red-highlighted SRC folder containing settings.ts and visual.ts. The main editor window shows the visual.ts file with the following content:

```
src > TS visual.ts > ...
1  /*
2   *  Power BI Visual CLI
3   *
4   *  Copyright (c) Microsoft Corporation
5   *  All rights reserved.
6   *  MIT License
7   *
8   *  Permission is hereby granted, free of charge, to any person obtaining a copy
9   *  of this software and associated documentation files (the "Software"), to deal
10  *  in the Software without restriction, including without limitation the rights
11  *  to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
12  *  copies of the Software, and to permit persons to whom the Software is
13  *  furnished to do so, subject to the following conditions:
14  *
15  *  The above copyright notice and this permission notice shall be included in
16  *  all copies or substantial portions of the Software.
17  *
18  *  THE SOFTWARE IS PROVIDED *AS IS*, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
19  *  IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
20  *  FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
21  *  AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
22  *  LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
23  *  OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
24  *  THE SOFTWARE.
25 */
```

2. Remove all the code under the MIT License comment.

### Important

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

3. Import the libraries and modules needed, and define the type selection for the [d3 library](#):

### TypeScript

```
"use strict";

import "./../style/visual.less";
import powerbi from "powerbi-visuals-api";
import VisualConstructorOptions =
  powerbi.extensibility.visual.VisualConstructorOptions;
import VisualUpdateOptions =
  powerbi.extensibility.visual.VisualUpdateOptions;
import IVisual = powerbi.extensibility.visual.IVisual;
import DataView = powerbi.DataView;
import IVisualHost = powerbi.extensibility.IVisualHost;
import * as d3 from "d3";
type Selection<T extends d3 BaseType> = d3.Selection<T, any, any, any>;
```

### Note

If the D3 JavaScript library wasn't installed as part of your setup, install it now.

From PowerShell, run `npm i d3@latest --save`

Notice that among the items you imported are:

- *IVisualHost* - A collection of properties and services used to interact with the visual host (Power BI).
- *D3 library* - JavaScript library for creating data driven documents.

4. Below the imports, create an empty *visual* class. The *visual* class implements the *IVisual* interface where all visuals begin:

TypeScript

```
export class Visual implements IVisual {  
}
```

For information about what goes into the visual class, see [Visual API](#). In the next three steps, we define this class.

5. Add class-level *private* methods at the beginning of the *visual* class:

TypeScript

```
private host: IVisualHost;  
private svg: Selection<SVGElement>;  
private container: Selection<SVGElement>;  
private circle: Selection<SVGElement>;  
private textValue: Selection<SVGElement>;  
private textLabel: Selection<SVGElement>;
```

Notice that some of these private methods use the *Selection* type.

6. Define the circle and text elements in the *constructor* method. This method is called when the visual is instantiated. The D3 Scalable Vector Graphics (SVG) enable creating three shapes: a circle, and two text elements:

TypeScript

```
constructor(options: VisualConstructorOptions) {  
    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);
}

```

7. Define the width and height in the update method. This method is called every time there's a change in the data or host environment, such as a new value or resizing.

TypeScript

```

public update(options: VisualUpdateOptions) {
    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");
}

```

8. Save the `visual.ts` file.

## (Optional) Review the code in the `visuals` file

Verify that the final code in the `visual.ts` file looks like this:

## TypeScript

```
/*
 * Power BI Visual CLI
 *
 * Copyright (c) Microsoft Corporation
 * All rights reserved.
 * MIT License
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy
 * of this software and associated documentation files (the ""Software""), to deal
 * in the Software without restriction, including without limitation the
 * rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included
 * in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED *AS IS*, WITHOUT WARRANTY OF ANY KIND, EXPRESS
 * OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
 * THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */
"use strict";

import "./../style/visual.less";
import powerbi from "powerbi-visuals-api";
import VisualConstructorOptions =
  powerbi.extensibility.visual.VisualConstructorOptions;
import VisualUpdateOptions =
  powerbi.extensibility.visual.VisualUpdateOptions;
import IVisual = powerbi.extensibility.visual.IVisual;
import DataView = powerbi.DataView;
import IVisualHost = powerbi.extensibility.IVisualHost;
import * as d3 from "d3";
type Selection<T extends d3.BaseType> = d3.Selection<T, any, any, any>

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) {
    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);
}

public update(options: VisualUpdateOptions) {
    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");
}
}

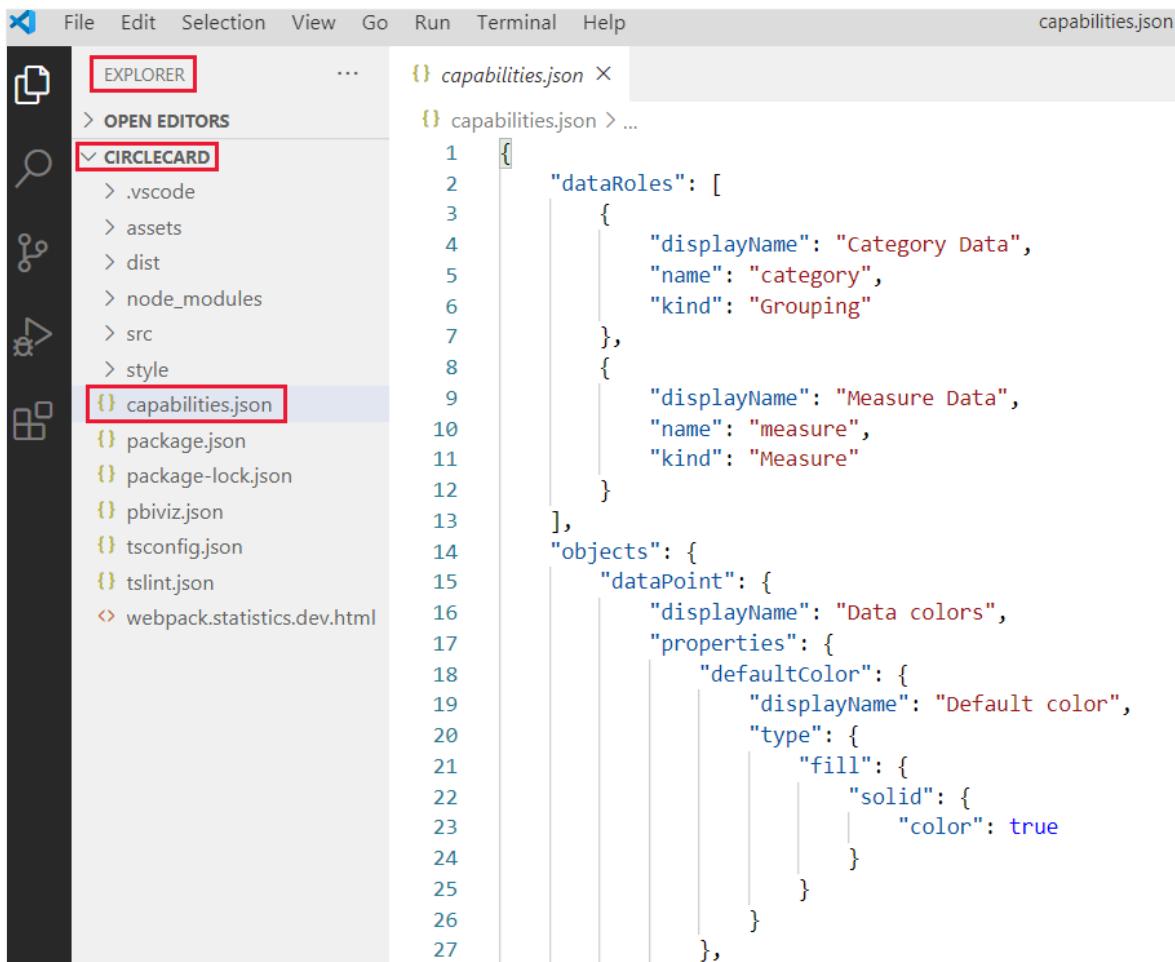
```

## Modify the capabilities file

The circle card visual is a simple visual that doesn't create any objects in the Format pane. Therefore, you can safely remove the *objects* section of the file.

1. Open your project in VS Code (**File > Open Folder**).

2. Select the **capabilities.json** file.



```
1  {
2   "dataRoles": [
3     {
4       "displayName": "Category Data",
5       "name": "category",
6       "kind": "Grouping"
7     },
8     {
9       "displayName": "Measure Data",
10      "name": "measure",
11      "kind": "Measure"
12    }
13  ],
14  "objects": {
15    "dataPoint": {
16      "displayName": "Data colors",
17      "properties": {
18        "defaultColor": {
19          "displayName": "Default color",
20          "type": {
21            "fill": {
22              "solid": {
23                "color": true
24              }
25            }
26          }
27        }
28      }
29    }
30  }
31 }
```

3. Remove the entire *objects* array.

Don't leave any blank lines between *dataRoles* and *dataViewMappings*.

4. Save the **capabilities.json** file.

## Restart the circle card visual

Stop the visual from running and restart it.

1. In the **PowerShell** window where you started the visual, enter **Ctrl + c**. If prompted to terminate the batch job, enter **y** and then **Enter**.

2. In **PowerShell**, start the visual again.

PowerShell

```
pbviz start
```

## Test the visual with the added elements

Verify that the visual displays the newly added elements.

1. In Power BI service, open the *Power BI US Sales Analysis* report. If you're using a different report to develop the circle card visual, navigate to that report.
2. Drag a value into the *Measure* box and make sure that the visual is shaped as a circle.



If the visual isn't displaying anything, from the **Fields** pane, drag the **Quantity** field into the developer visual.

3. Resize the visual.

Notice that the circle and text scale to fit the dimensions of the visual. The update method is called when you resize the visual, and as a result the visual elements get rescaled.

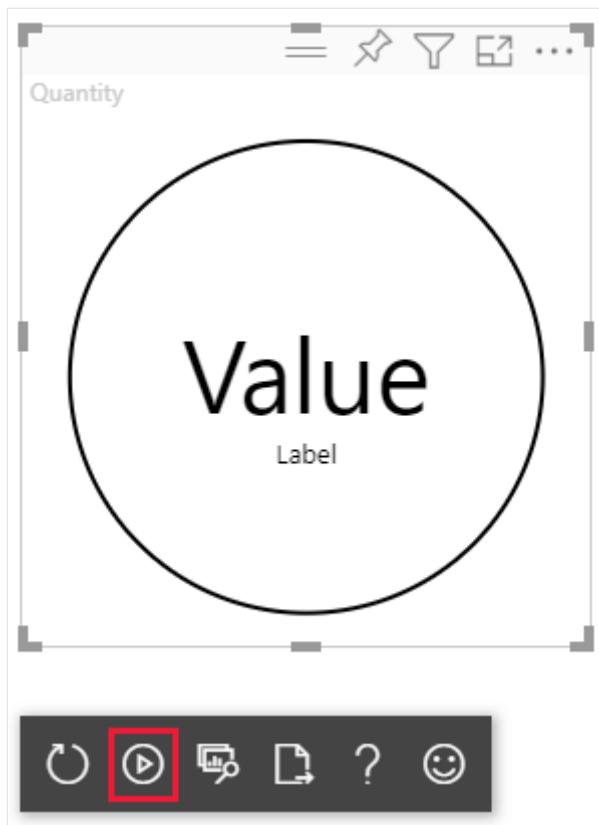
## Enable auto reload

Use this setting to ensure that the visual is automatically reloaded each time you save project changes.

1. Navigate to the *Power BI US Sales Analysis* report (or to the project that has your circle card visual).

2. Select the circle card visual.

3. In the floating toolbar, select **Toggle Auto Reload**.



## Get the visual to process data

In this section, you define data roles and data view mappings. You also modify the visual to display the name of the value it's displaying.

### Configure the capabilities file

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

- **Define the data role**

Define the *dataRoles* array with a single data role of the type *measure*. This data role is called *measure*, and is displayed as *Measure*. It allows passing either a measure field, or a summed up field.

1. Open the **capabilities.json** file in VS Code.
2. Remove all the content inside the *dataRoles* array.
3. Insert the following code to the *dataRoles* array.

JSON

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

4. Save the **capabilities.json** file.

- **Define the data view mapping**

Define a field called *measure* in the *dataViewMappings* array. This field can be passed to the data role.

1. Open the **capabilities.json** file in VS Code.

2. Remove all the content inside the *dataViewMappings* array.

3. Insert the following code to the *dataViewMappings* array.

JSON

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

4. Save the **capabilities.json** file.

Confirm that your *capabilities.json* file looks like this:

TypeScript

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

```
        ],
        "single": {
            "role": "measure"
        }
    ],
    "privileges": []
}
```

## Related content

- Add formatting options to the circle card visual
- Power BI visuals project structure
- Learn how to debug a Power BI visual you created

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Tutorial: Add formatting options to the Circle Card visual

Article • 10/10/2024

When you create a visual, you can add options for customizing its properties. Some of the items that can be customized include:

- Title
- Background
- Border
- Shadow
- Colors

In this tutorial, you learn how to:

- ✓ Add formatting properties to your visual.
- ✓ Package the visual
- ✓ Import the custom visual to a Power BI Desktop or Service report

## Prerequisite

This tutorial explains how to add common formatting properties to a visual. We'll use the [Circle card](#) visual as an example. We'll add the ability to change the color and thickness of the circle. If you don't have the [Circle card](#) project folder that you created in that tutorial, redo the tutorial before continuing.

## Adding formatting options

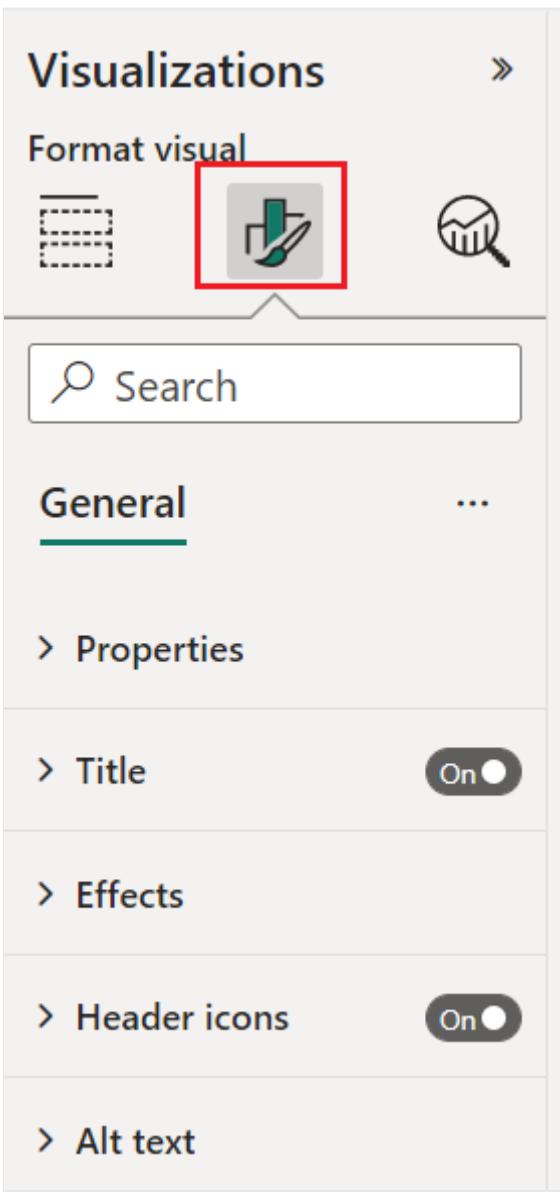
1. In **PowerShell**, Navigate to your circle card project folder and start the circle card visual. Your visual is now running while being hosted on your computer.

```
PowerShell
```

```
pbviz start
```

2. In **Power BI**, select the **Format** panel.

You should see general formatting options, but not any visual formatting options.



3. In **Visual Studio Code**, open the `capabilities.json` file.

4. Before the `dataViewMappings` array, add `objects`.

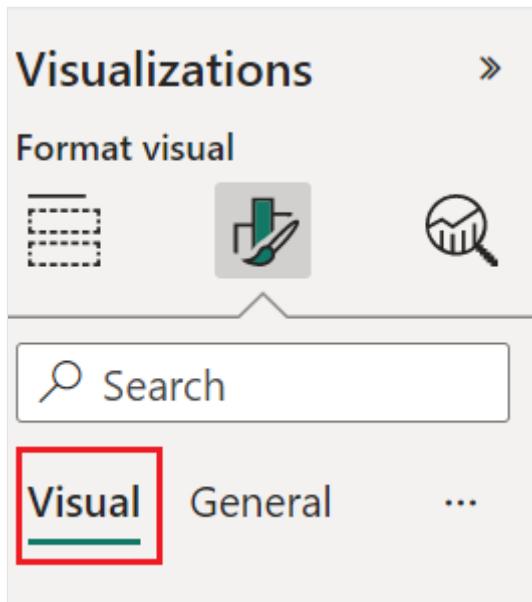
```
JSON
"objects": {},  
  
"dataRoles": [  
    {  
        "displayName": "Measure",  
        "name": "measure",  
        "kind": "Measure"  
    }  
],  
"objects": {},  
"dataViewMappings": []  
{  
    "conditions": [
```

5. Save the `capabilities.json` file.

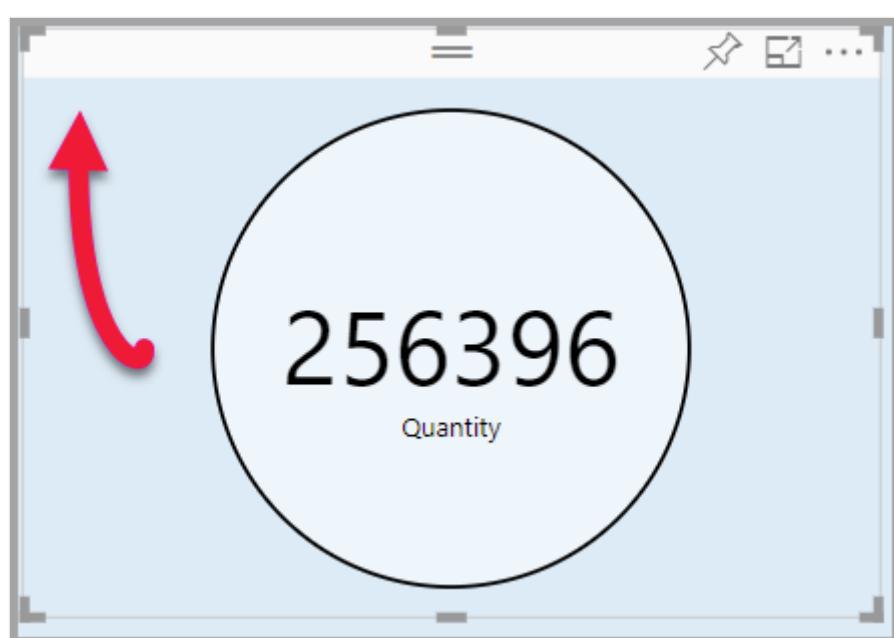
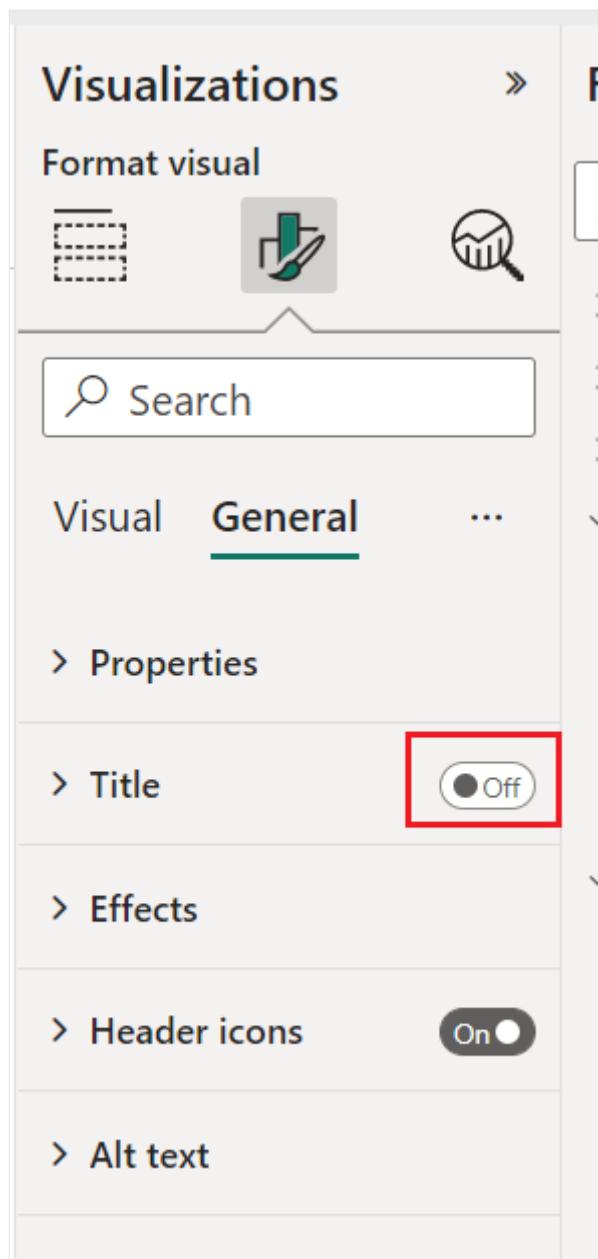
6. In Power BI, review the formatting options again.

ⓘ Note

If you don't see the formatting options change, select **Reload Custom Visual**.



7. 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

Now let's add new group called *color* for configuring the circle color and thickness of the circle's outline.

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

```
JSON

{
    "circle": {
        "properties": {
            "circleColor": {
                "type": {
                    "fill": {
                        "solid": {
                            "color": true
                        }
                    }
                }
            },
            "circleThickness": {
                "type": {
                    "numeric": true
                }
            }
        }
    }
}
```

This JSON fragment describes a group called *circle*, which consists of two variables - *circleColor* and *circleThickness*.

3. Save the `capabilities.json` file.
4. In the **Explorer pane**, go to 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 import lines and two classes with the following code.

```
TypeScript

import { formattingSettings } from "powerbi-visuals-utils-
formattingmodel";

import FormattingSettingsCard = formattingSettings.SimpleCard;
import FormattingSettingsSlice = formattingSettings.Slice;
import FormattingSettingsModel = formattingSettings.Model;
```

```

export class CircleSettings extends FormattingSettingsCard{
    public circleColor = new formattingSettings.ColorPicker({
        name: "circleColor",
        displayName: "Color",
        value: { value: "#ffffff" },
        visible: true
    });

    public circleThickness = new formattingSettings.NumUpDown({
        name: "circleThickness",
        displayName: "Thickness",
        value: 2,
        visible: true
    });

    public name: string = "circle";
    public displayName: string = "Circle";
    public visible: boolean = true;
    public slices: FormattingSettingsSlice[] = [this.circleColor,
    this.circleThickness]
}

export class VisualSettings extends FormattingSettingsModel {
    public circle: CircleSettings = new CircleSettings();
    public cards: FormattingSettingsCard[] = [this.circle];
}

```

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 sets default values. The **VisualSettings** class defines the circle object according to the properties described in the `capabilities.json` file.

6. Save the `settings.ts` file.
7. Open the `visual.ts` file.
8. In the `visual.ts` file, import the :

TypeScript

```

import { VisualSettings } from "./settings";
import { FormattingSettingsService } from "powerbi-visuals-utils-
formattingmodel";

```

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

TypeScript

```
private visualSettings: VisualSettings;
private formattingSettingsService: FormattingSettingsService;
```

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

9. In the **Visual** class, insert the following as the first line of the *constructor*:

TypeScript

```
this.formattingSettingsService = new FormattingSettingsService();
```

10. In the **Visual** class, add the following method after the **update** method.

TypeScript

```
public getFormattingModel(): powerbi.visuals.FormattingModel {
    return
    this.formattingSettingsService.buildFormattingModel(this.visualSettings
);
}
```

This function gets called on every formatting pane render. It allows you to select which of the objects and properties you want to expose to the users in the property pane.

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

TypeScript

```
this.visualSettings =
this.formattingSettingsService.populateFormattingSettingsModel(VisualSe
ttings, options.dataViews[0]);
this.visualSettings.circle.circleThickness.value = Math.max(0,
this.visualSettings.circle.circleThickness.value);
this.visualSettings.circle.circleThickness.value = Math.min(10,
this.visualSettings.circle.circleThickness.value);
```

This code retrieves the format options. It adjusts any value passed into the **circleThickness** property, and converts it to a number between zero and 10.

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

this.visualSettings = this.formattingSettingsService.populateFormattingSettingsModel(VisualSett
this.visualSettings.circle.circleThickness.value = Math.max(0, this.visualSettings.circle.circ
this.visualSettings.circle.circleThickness.value = Math.min(10, this.visualSettings.circle.circ

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

12. In the **circle** element, modify the values passed to the **fill** style and **stroke-width** style as follows:

TypeScript

```
.style("fill", this.visualSettings.circle.circleColor.value.value)
```

TypeScript

```
.style("stroke-width",
this.visualSettings.circle.circleThickness.value)
```

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

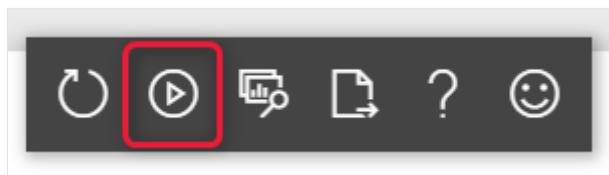
13. Save the `visual.ts` file.

14. In **PowerShell**, start the visual.

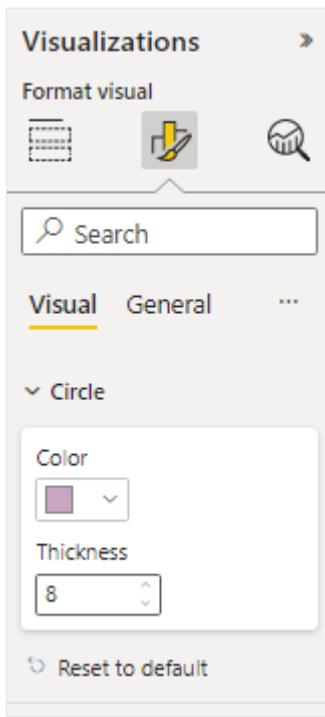
PowerShell

```
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.

## Debugging

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

## Packaging the custom visual

Now that the visual is completed and ready to be used, it's time to package it. A packaged visual can be imported to Power BI reports or service to be used and enjoyed by others.

When your visual is ready, follow the directions in [Package a Power BI visual](#) and then, if you want, share it with others so they can [import](#) and enjoy it.

## Related content

- [Create a Power BI bar chart visual](#)
- [Learn how to debug a Power BI visual you created](#)
- [Power BI visuals project structure](#)

# Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Tutorial: Build a bar chart

Article • 05/13/2024

This tutorial shows you how to develop a Power BI visual that displays data in the form of a simple bar chart. This visual supports a minimal amount of customization. Other pages of this documentation explain how to add further customization like [context menus](#), [tool-tips](#), and more.

In this tutorial, you learn how to:

- ✓ Define the capabilities of your visual
- ✓ Understand the source code used to build a visual
- ✓ Render the visual
- ✓ Add objects to the properties pane
- ✓ Package the visual

## Set up your environment

Before you start developing your Power BI visual, verify that you have everything listed in this section.

- A **Power BI Pro or Premium Per User (PPU)** account. If you don't have one, [sign up for a free trial](#).
- [Visual Studio Code \(VS Code\)](#). VS Code is an ideal Integrated Development Environment (IDE) for developing JavaScript and TypeScript applications.
- [Windows PowerShell](#) version 4 or later (for Windows). Or [Terminal](#) (for OSX).
- An environment ready for developing a Power BI visual. [Set up your environment for developing a Power BI visual](#).
- This tutorial uses the **US Sales Analysis** report. You can [download this report](#) and upload it to Power BI service, or use your own report. If you need more information about Power BI service, and uploading files, refer to the [Get started creating in the Power BI service](#) tutorial.

### Note

If the D3 JavaScript library wasn't installed as part of your setup, install it now. From PowerShell, run `npm i d3@latest --save`

Creating a bar chart visual involves the following steps:

1. [Create a new project](#)
2. [Define the capabilities file - `capabilities.json`](#)
3. Create the [visual API](#)
4. [Package](#) your visual - `pbiviz.json`

## Create a new project

The purpose of this tutorial is to help you understand how a visual is structured and written. You can follow these instructions to create a bar code visual from scratch, or you can [clone the source code repository ↗](#) and use it to follow along without creating your own visual.

Create a new visual

1. Open **PowerShell** and navigate to the folder you want to create your project in.
2. Enter the following command:

```
PowerShell
pbiviz new BarChart
```

You should now have a folder called *BarChart* containing the visual's files.

3. In **VS Code**, open the [`tsconfig.json`] ([visual-project-structure.md#tsconfigjson](#)) file and change the name of "files" to "src/barChart.ts".

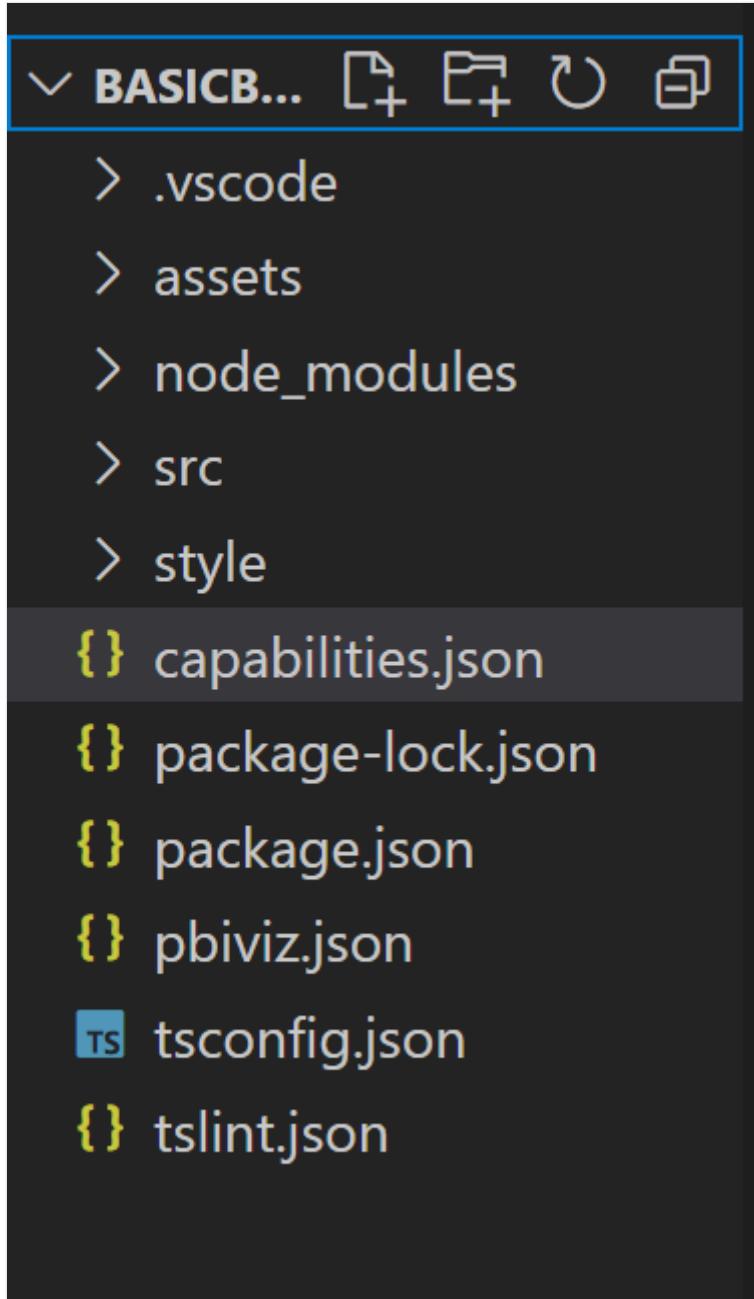
```
TypeScript
{
  "files": [
    "src/barChart.ts"
  ]
}
```

The `tsconfig.json` "files" object points to the file where the main class of the visual is located.

Your final `tsconfig.json` file should look like [this ↗](#).

4. The `package.json` file contains a list of project dependencies. Replace your `package.json` file with [this one](#).

You should now have a new folder for your visual with the following files and folders:



For a detailed explanation of the function of each of these files, see [Power BI visual project structure](#).

The two files we focus on in this tutorial are the `capabilities.json` file, which describes the visual to the host, and the `src/barchart.ts` file, which contains the visual's API.

## Define capabilities

The [capabilities.json](#) file is where we bind data to the host. We describe the kind of data fields it accepts and what features the visual should have.

# Visualizations

Build visual

Category Data

Add data fields here

Measure Data

Add data fields here

## Define data roles

Variables are defined and bound in the [dataRoles](#) section of the capabilities file. We want our bar chart to accept two types of variables:

- **Categorical** data represented by the different bars on the chart
- **Numerical**, or measured data, which is represented by the height of each bar

In **Visual Studio Code**, in the *capabilities.json* file, confirm that the following JSON fragment appears in the object labeled "dataRoles".

```
JSON

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

## Map the data

Next, add [data mapping](#) to tell the host what to do with these variables:

Replace the content of the "dataViewMappings" object with the following code:

```
JSON

"dataViewMappings": [
  {
    "conditions": [
      {
        "category": {
          "max": 1
        },
        "measure": {
          "max": 1
        }
      }
    ],
    "categorical": {
      "categories": {
        "for": {
          "in": "category"
        }
      }
    }
  ]
],
```

```

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

```

The above code creates the "conditions" that each data-role object can hold only one field at a time. Notice that we use the data-role's internal `name` to refer to each field.

It also sets the [categorical data mapping](#) so that each field is mapped to the correct variable.

## Define objects for the properties pane

The "objects" section of the *capabilities* file is where we define the customizable features that should appear on the [format pane](#). These features don't affect the content of the chart but they can change its look and feel.

For more information on objects and how they work, see [Objects](#).

The following objects are optional. Add them if you want to go through the optional sections of this tutorial to add colors and render the X-axis.

Replace the content of the "objects" section with the following code:

JSON

```

"objects": {
    "enableAxis": {
        "properties": {
            "show": {
                "type": {
                    "bool": true
                }
            },
            "fill": {
                "type": {
                    "fill": {
                        "solid": {

```

```
        "color": true
    }
}
},
"colorSelector": {
    "properties": {
        "fill": {
            "type": {
                "fill": {
                    "solid": {
                        "color": true
                    }
                }
            }
        }
    }
},
},
```

Save the *capabilities.json* file.

Your final *capabilities* file should look like [the one in this example ↗](#).

## Visual API

All visuals start with a class that implements the `Ivisual` interface. The `src/visual.ts` file is the default file that contains this class.

In this tutorial, we call our `Ivisual` file *barChart.ts*. [Download the file ↗](#) and save it to the `/src` folder, if you didn't do so already. In this section, we go through this file in detail and describe the various sections.

## Imports

The first section of the file imports the modules that are needed for this visual. Notice that in addition to the Power BI visual modules, we also import the [d3 library ↗](#).

The following modules are imported to your *barChart.ts* file:

TypeScript

```
import {
    BaseType,
    select as d3Select,
    Selection as d3Selection
```

```

} from "d3-selection";
import {
  ScaleBand,
  ScaleLinear,
  scaleBand,
  scaleLinear
} from "d3-scale";
import "./../style/visual.less";

import { Axis, axisBottom } from "d3-axis";

import powerbi from "powerbi-visuals-api";

type Selection<T extends BaseType> = d3Selection<T, any, any, any>

// powerbi.visuals
import DataViewCategoryColumn = powerbi.DataViewCategoryColumn;
import Fill = powerbi.Fill;
import ISandboxExtendedColorPalette =
powerbi.extensibility.ISandboxExtendedColorPalette;
import ISelectionId = powerbi.visuals.ISelectionId;
import IVisual = powerbi.extensibility.IVisual;
import IVisualHost = powerbi.extensibility.visual.IVisualHost;
import PrimitiveValue = powerbi.PrimitiveValue;
import VisualUpdateOptions =
powerbi.extensibility.visual.VisualUpdateOptions;
import VisualConstructorOptions =
powerbi.extensibility.visual.VisualConstructorOptions;
import DataViewObjectPropertyIdentifier =
powerbi.DataViewObjectPropertyIdentifier;

import { textMeasurementService } from "powerbi-visuals-utils-
formattingutils";
import { FormattingSettingsService } from "powerbi-visuals-utils-
formattingmodel";

import { BarChartSettingsModel } from "./barChartSettingsModel";
import { dataViewObjects} from "powerbi-visuals-utils-dataviewutils";

```

## Interfaces

Next, we define the visual [interfaces](#). The following interface is used to describe our bar chart visual:

- BarChartDataPoint

This interface is defined as follows:

TypeScript

```

/**
 * Interface for BarChart data points.
 *
 * @interface
 * @property {PrimitiveValue} value - Data value for point.
 * @property {string} category - Corresponding category of data
 * value.
 * @property {string} color - Color corresponding to data point.
 * @property {string} strokeColor - Stroke color for data point
 * column.
 * @property {number} strokeWidth - Stroke width for data point
 * column.
 * @property {ISelectionId} selectionId - Id assigned to data point for
 * cross filtering
 *
 */
interface BarChartDataPoint {
    value: PrimitiveValue;
    category: string;
    color: string;
    strokeColor: string;
    strokeWidth: number;
    selectionId: ISelectionId;
}

```

## Visual transform

Now that the data structure is defined, we need to map data onto it using the `createSelectorDataPoints` function. This function receives data from the data view and transforms it to a format the visual can use. In this case, it returns the `BarChartDataPoint[]` interface described in the previous section.

The `DataView` contains the data to be visualized. This data can be in different forms, such as categorical or tabular. To build a categorical visual like a bar chart, use the *categorical* property on the `DataView`.

This function is called whenever the visual is updated.

TypeScript

```

/**
 * Function that converts queried data into a.viewmodel 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
*
* @param {IVisualHost} host
which contains services
*/
function createSelectorDataPoints(options: VisualUpdateOptions, host:
IVisualHost): BarChartDataPoint[] {
    const barChartDataPoints: BarChartDataPoint[] = []
    const dataViews = options.dataViews;

    if (!dataViews
        || !dataViews[0]
        || !dataViews[0].categorical
        || !dataViews[0].categorical.categories
        || !dataViews[0].categorical.categories[0].source
        || !dataViews[0].categorical.values
    ) {
        return barChartDataPoints;
    }

    const categorical = dataViews[0].categorical;
    const category = categorical.categories[0];
    const dataValue = categorical.values[0];

    const colorPalette: ISandboxExtendedColorPalette = host.colorPalette;

    const strokeColor: string = getColumnStrokeColor(colorPalette);

    const strokeWidth: number =
getColumnStrokeWidth(colorPalette.isHighContrast);

    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]}`,
        });
    }

    return barChartDataPoints;
}

```

## (!) Note

The next few functions in the `barChart.ts` file deal with color and creating the X axis. Those are optional and are discussed further down in this tutorial. This tutorial will continue from the `IVisual` function.

# Render the visual

Once the data is defined, we render the visual using the `BarChart` class that implements the `IVisual` interface. The `IVisual` interface is described on the [Visual API](#) page. It contains a `constructor` method that creates the visual and an `update` method that is called each time the visual reloads. Before rendering the visual, we have to declare the members of the class:

TypeScript

```
export class BarChart implements IVisual {
    private svg: Selection<SVGSVGElement>;
    private host: IVisualHost;
    private barContainer: Selection<SVGElement>;
    private xAxis: Selection<SVGGElement>;
    private barDataPoints: BarChartDataPoint[];
    private formattingSettings: BarChartSettingsModel;
    private formattingSettingsService: FormattingSettingsService;

    private barSelection: Selection<BaseType>;

    static Config = {
        xScalePadding: 0.1,
        solidOpacity: 1,
        transparentOpacity: 1,
        margins: {
            top: 0,
            right: 0,
            bottom: 25,
            left: 30,
        },
        xAxisFontMultiplier: 0.04,
    };
}
```

# Construct the visual

The [constructor function](#) is called only once, when the visual is rendered for the first time. It creates empty SVG containers for the bar chart and the X-axis. Notice that it uses

the d3 library to render the SVG.

TypeScript

```
/**  
 * Creates instance of BarChart. This method is only called once.  
 *  
 * @constructor  
 * @param {VisualConstructorOptions} options - Contains references to  
 the element that will  
 *                                         contain the visual and a  
 reference to the host  
 *                                         which contains services.  
 */  
constructor(options: VisualConstructorOptions) {  
    this.host = options.host;  
    //Creating the formatting settings service.  
    const localizationManager = this.host.createLocalizationManager();  
    this.formattingSettingsService = new  
FormattingSettingsService(localizationManager);  
  
    this.svg = d3Select(options.element)  
        .append('svg')  
        .classed('barChart', true);  
  
    this.barContainer = this.svg  
        .append('g')  
        .classed('barContainer', true);  
  
    this.xAxis = this.svg  
        .append('g')  
        .classed('xAxis', true);  
}
```

## Update the visual

The [update method](#) is called every time the size of the visual or one of its values changes.

## Scaling

We need to scale the visual so that the number of bars and current values fit into the defined width and height limits of the visual. This is similar to the [update method in the Circle card tutorial](#).

To calculate the scale, we use the `scaleLinear` and `scaleBand` methods that were imported earlier from the `d3-scale` library.

The `options.dataViews[0].categorical.values[0].maxLocal` value holds the largest value of all current data points. This value is used to determine the height of the y axis. The scaling for the width of the x axis is determined by the number of categories bound to the visual in the `barchartdatapoint` interface.

For cases where the X axis is rendered, this visual also handles word breaks in case there isn't enough room to write out the entire name on the X axis.

## Other update features

In addition to scaling, the update method also handles selections and colors. These features are optional and are discussed later:

TypeScript

```
/**  
 * Updates the state of the visual. Every sequential databinding and  
 * resize will call update.  
 *  
 * @function  
 * @param {VisualUpdateOptions} options - Contains references to the  
 * size of the container  
 *                                         and the dataView which  
 * contains all the data  
 *                                         the visual had queried.  
 */  
public update(options: VisualUpdateOptions) {  
    this.formattingSettings =  
this.formattingSettingsService.populateFormattingSettingsModel(BarChartSettingsModel, options.dataViews?[0]);  
    this.barDataPoints = createSelectorDataPoints(options, this.host);  
    this.formattingSettings.populateColorSelector(this.barDataPoints);  
  
    const width = options.viewport.width;  
    let height = options.viewport.height;  
  
    this.svg  
        .attr("width", width)  
        .attr("height", height);  
  
    if (this.formattingSettings.enableAxis.show.value) {  
        const margins = BarChart.Config.margins;  
        height -= margins.bottom;  
    }  
  
    this.xAxis  
        .style("font-size", Math.min(height, width) *  
BarChart.Config.xAxisFontMultiplier)  
        .style("fill",  
this.formattingSettings.enableAxis.fill.value.value);
```

```

    const yScale: ScaleLinear<number, number> = scaleLinear()
      .domain([0,
<number>options.dataViews[0].categorical.values[0].maxLocal])
      .range([height, 0]);

    const xScale: ScaleBand<string> = scaleBand()
      .domain(this.barDataPoints.map(d => d.category))
      .rangeRound([0, width])
      .padding(0.2);

    const xAxis: Axis<string> = axisBottom(xScale);

    this.xAxis.attr('transform', 'translate(0, ' + height + ')')
      .call(xAxis)
      .attr("color",
this.formattingSettings.enableAxis.fill.value.value);

    const textNodes: Selection<SVGElement> =
this.xAxis.selectAll("text");
    BarChart.wordBreak(textNodes, xScale.bandwidth(), height);

    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", (dataPoint: BarChartDataPoint) => height -
yScale(<number>dataPoint.value))
      .attr("y", (dataPoint: BarChartDataPoint) =>
yScale(<number>dataPoint.value))
      .attr("x", (dataPoint: BarChartDataPoint) =>
xScale(dataPoint.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();
}

private static wordBreak(
  textNodes: Selection<SVGElement>,

```

```

        allowedWidth: number,
        maxHeight: number
    ) {
    textNodes.each(function () {
        textMeasurementService.wordBreak(
            this,
            allowedWidth,
            maxHeight);
    });
}

```

## Populate the properties pane using the formatting model Utils

The final method in the `IVisual` function is `getFormattingModel`. This method builds and returns a modern *format pane formatting model* object containing all the `format pane` components and properties. It then places the object inside the `Format` pane. In our case, we create format cards for `enableAxis` and `colorSelector`, including formatting properties for `show` and `fill`, according to the "objects" in the `capabilities.json` file. To add a color picker for each category on the `Property` pane, add a for loop on `barDataPoints` and for each one add a new color picker format property to the formatting model.

To build a formatting model, the developer should be familiar with all its components. Check out the components of the format pane in [Format Pane](#). Check out `getFormattingModel` API of the [FormattingModel utils](#) in the [formatting model utils repository](#).

[Download the file](#) and save it to the `/src` folder. Declare formatting properties and their values in a formatting settings class:

TypeScript

```

import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";
import { BarChartDataPoint } from "./barChart";

import Card = formattingSettings.SimpleCard;
import Model = formattingSettings.Model;
import Slice = formattingSettings.Slice;
import ColorPicker = formattingSettings.ColorPicker;
import ToggleSwitch = formattingSettings.ToggleSwitch;

/**
 * Enable Axis Formatting Card
 */
class EnableAxisCardSettings extends Card {

```

```

show = new ToggleSwitch({
    name: "show",
    displayName: undefined,
    value: false,
});

fill = new ColorPicker({
    name: "fill",
    displayName: "Color",
    value: { value: "#000000" }
});
topLevelSlice: ToggleSwitch = this.show;
name: string = "enableAxis";
displayName: string = "Enable Axis";
slices: Slice[] = [this.fill];
}

/**
 * Color Selector Formatting Card
 */
class ColorSelectorCardSettings extends Card {
    name: string = "colorSelector";
    displayName: string = "Data Colors";

    // slices will be populated in barChart settings model
    `populateColorSelector` method
    slices: Slice[] = [];
}

/**
 * BarChart formatting settings model class
 */
export class BarChartSettingsModel extends Model {
    // Create formatting settings model formatting cards
    enableAxis = new EnableAxisCardSettings();
    colorSelector = new ColorSelectorCardSettings();
    cards: Card[] = [this.enableAxis, this.colorSelector];

    /**
     * populate colorSelector object categories formatting properties
     * @param dataPoints
     */
    populateColorSelector(dataPoints: BarChartDataPoint[]) {
        const slices: Slice[] = this.colorSelector.slices;
        if (dataPoints) {
            dataPoints.forEach(dataPoint => {
                slices.push(new ColorPicker({
                    name: "fill",
                    displayName: dataPoint.category,
                    value: { value: dataPoint.color },
                    selector: dataPoint.selectionId.getSelector(),
                }));
            });
        }
    }
}

```

```
    }  
}
```

Build and create the *formatting settings service* model in the visual's *constructor* method. The *formatting settings service* receives the barChart format settings and converts them into a *FormattingModel* object that's returned in the `getFormattingModel` API.

To use the localization feature, add the localization manager to the formatting settings service.

TypeScript

```
import { FormattingSettingsService } from "powerbi-visuals-utils-formattingmodel";  
  
// ...  
// declare utils formatting settings service  
private formattingSettingsService: FormattingSettingsService;  
//...  
  
constructor(options: VisualConstructorOptions) {  
    this.host = options.host;  
    const localizationManager = this.host.createLocalizationManager();  
    this.formattingSettingsService = new  
FormattingSettingsService(localizationManager);  
  
    // Add here rest of your custom visual constructor code  
}
```

Update the formatting settings model using update API. Call the *Update* API each time a formatting property in the properties pane is changed. Create bar chart selectors data points and populate them in formatting settings model:

TypeScript

```
// declare formatting settings model for bar chart  
private formattingSettings: BarChartSettingsModel;  
  
// ...  
  
public update(options: VisualUpdateOptions) {  
    this.formattingSettings =  
this.formattingSettingsService.populateFormattingSettingsModel(BarChartSettingsModel,  
options.dataViews[0]);  
    this.barDataPoints = createSelectorDataPoints(options, this.host);  
    this.formattingSettings.populateColorSelector(this.barDataPoints);  
  
    // Add the rest of your custom visual update API code here
```

```
}
```

Finally, the new API `getFormattingModel` is a simple line of code using the formatting settings service and current formatting settings model that was created in the *update* API above.

TypeScript

```
public getFormattingModel(): powerbi.visuals.FormattingModel {
    return
>this.formattingSettingsService.buildFormattingModel(this.formattingSettings)
;
}
```

## (Optional) Render the X axis (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.

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

# Visualizations

»

## Format visual



Search

### Visual

### General

...

> Y-axis

On

> X-axis

On

> Legend

On

> Small multiples

> Gridlines

> Bars

> Data labels

Off

> Plot area background

This example renders an X-axis on the bar chart as a static object.

We already added the `enableAxis` property to the *capabilities* file and the `barChartSettings` interface.

## (Optional) Add color (data-bound objects)

Data-bound objects are similar to static objects, but typically deal with data selection. For example, you can use data-bound objects to interactively select the color associated with each data point.

## ▼ Data Colors

Southern



Midwest



Northeast



Pacific Northwest



Southwest



New England



⟲ Reset to default

We already defined the `colorSelector` object in the *capabilities* file.

Each data point is represented by a different color. We include color in the [BarChartDataPoint interface](#), and assign a default color to each data point when it's defined in [IVisualHost](#).

TypeScript

```
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,
        }
    };

    const prop: DataViewObjectPropertyIdentifier = {
        objectName: "colorSelector",
        propertyName: "fill"
    };

    let colorFromObjects: Fill;
    if(category.objects?.[index]){
        colorFromObjects =
        dataViewObjects.getValue(category?.objects[index], prop);
    }

    return colorFromObjects?.solid.color ?? defaultColor.solid.color;
}

function getColumnStrokeColor(colorPalette: ISandboxExtendedColorPalette): string {
    return colorPalette.isHighContrast
        ? colorPalette.foreground.value
        : null;
}

function getColumnStrokeWidth(isHighContrast: boolean): number {
    return isHighContrast
        ? 2
        : 0;
}

```

The `colorPalette` service, in the `createSelectorDataPoints` function, manages these colors. Since `createSelectorDataPoints` iterates through each of the data points, it's an ideal place to assign categorical objects like color.

For more detailed instructions on how to add color to your bar chart go to [Add colors to your Power BI visual](#).

### ⓘ Note

Verify that your final `barChart.ts` file looks like this [barChart.ts source code ↗](#), or download the [barChart.ts source code ↗](#) and use it to replace your file.

# Test the visual

Run the visual in the **Power BI** server to see how it looks:

1. In **PowerShell**, navigate to the project's folder and start the development app.

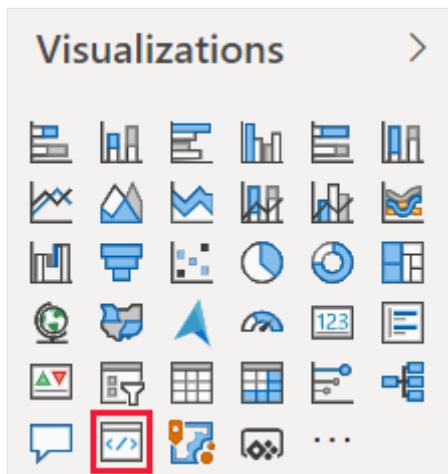
```
PowerShell  
pbviz start
```

Your visual is now running while being hosted on your computer.

 **Important**

Do not close the **PowerShell** window until the end of the tutorial. To stop the visual from running, enter **Ctrl + C**, and if prompted to terminate the batch job, enter **Y**, and then **Enter**.

2. [View the visual in Power BI service](#) by selecting the **Developer visual** from the **Visualization pane**.



3. Add data to the visual

# Visualizations

»

## Build visual



### Category Data

Region ▼ X

### Measure Data

Sales ▼ X

4. Drag the edges of the visual to change the size and notice how the scale adjusts.

5. Toggle the X-axis on and off.

## > Enable Axis

On

6. Change the colors of the different categories.

## Add other features

You can further customize your visual by adding more features. You can add features that increase the visual's functionality, enhance its look and feel, or give the user more control over its appearance. For example, you can:

- Add Selection and Interactions with Other Visuals ↗
- Add a property pane slider that controls opacity ↗
- Add support for tooltips ↗
- Add a landing page
- Add local language support ↗

## Package the visual

Before you can load your visual into [Power BI Desktop](#) ↗ or share it with the community in the [Power BI Visual Gallery](#) ↗, you have to package it.

To prepare the visual for sharing, follow the instructions in [Package a Power BI visual](#).

### ⓘ Note

For the full source code of a bar chart with more features, including [tool-tips](#) and a [context menu](#), see [Power BI visuals sample bar chart](#) ↗.

## Related content

- [Add a context menu to a visual](#)
- [Add a landing page to a visual](#)
- [Add locale support](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Tutorial: Create a Power BI visual using React

Article • 10/12/2024

As a developer you can create your own Power BI visuals. These visuals can be used by you, your organization or by third parties.

In this tutorial, you develop a Power BI visual using [React](#). The visual displays a formatted measure value inside a circle. The visual has adaptive size and allows you to customize its settings.

In this tutorial, you learn how to:

- ✓ Create a development project for your visual.
- ✓ Develop your visual using React.
- ✓ Configure your visual to process data.
- ✓ Configure your visual to adapt to size changes.
- ✓ Configure adaptive color and border settings for your visual.

## ⓘ Note

For the full source code of this visual, see [React circle card Power BI visual](#).

## Prerequisites

Before you start developing your Power BI visual, verify that you have everything listed in this section.

- A **Power BI Pro** or **Premium Per User (PPU)** account. If you don't have one, [sign up for a free trial](#).
- [Visual Studio Code \(VS Code\)](#). VS Code is an ideal Integrated Development Environment (IDE) for developing JavaScript and TypeScript applications.
- [Windows PowerShell](#) version 4 or later (for Windows). Or [Terminal](#) (for Mac).
- An environment ready for developing a Power BI visual. [Set up your environment for developing a Power BI visual](#).
- This tutorial uses the **US Sales Analysis** report. You can [download this report](#) and upload it to Power BI service, or use your own report. If you need more information

about Power BI service, and uploading files, refer to the [Get started creating in the Power BI service](#) tutorial.

## Create a development project

In this section, you create a project for the React circle card visual.

1. Open PowerShell and navigate to the folder you want to create your project in.
2. Enter the following command:

```
PowerShell  
pbviz new ReactCircleCard
```

3. Navigate to the project's folder.

```
PowerShell  
cd ReactCircleCard
```

4. Start the React circle card visual. Your visual is now running while being hosted on your computer.

```
PowerShell  
pbviz start
```

 **Important**

To stop the visual from running, in PowerShell enter **Ctrl+C** and if prompted to terminate the batch job, enter **Y**, and press *Enter*.

## View the React circle card in the Power BI service

To test the visual in Power BI service, we'll use the **US Sales Analysis** report. You can [download](#) this report and upload it to Power BI service.

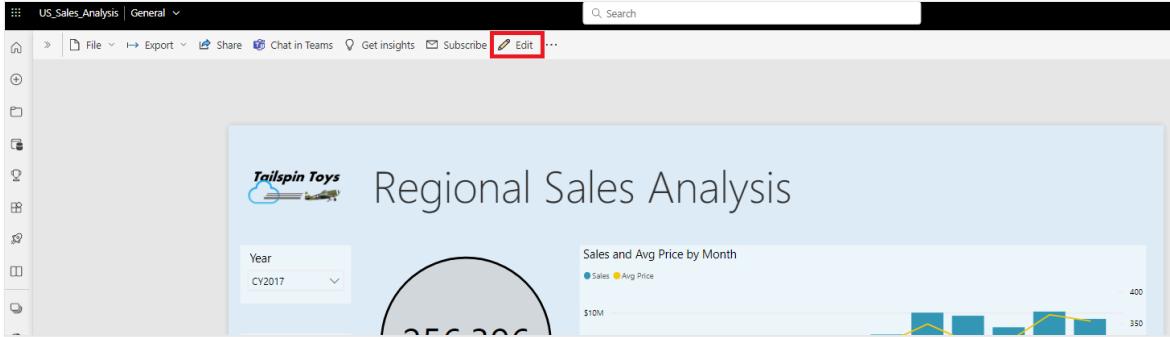
You can also use your own report to test the visual.

## ⚠ Note

Before you continue, verify that you [enabled the visuals developer mode](#).

1. Sign in to [PowerBI.com](#) and open the **US Sales Analysis** report.

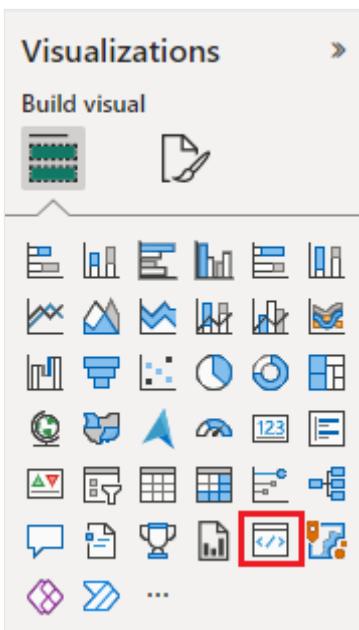
2. Select **Edit**.



3. Create a new page for testing, by clicking on the **New page** button at the bottom of the Power BI service interface.

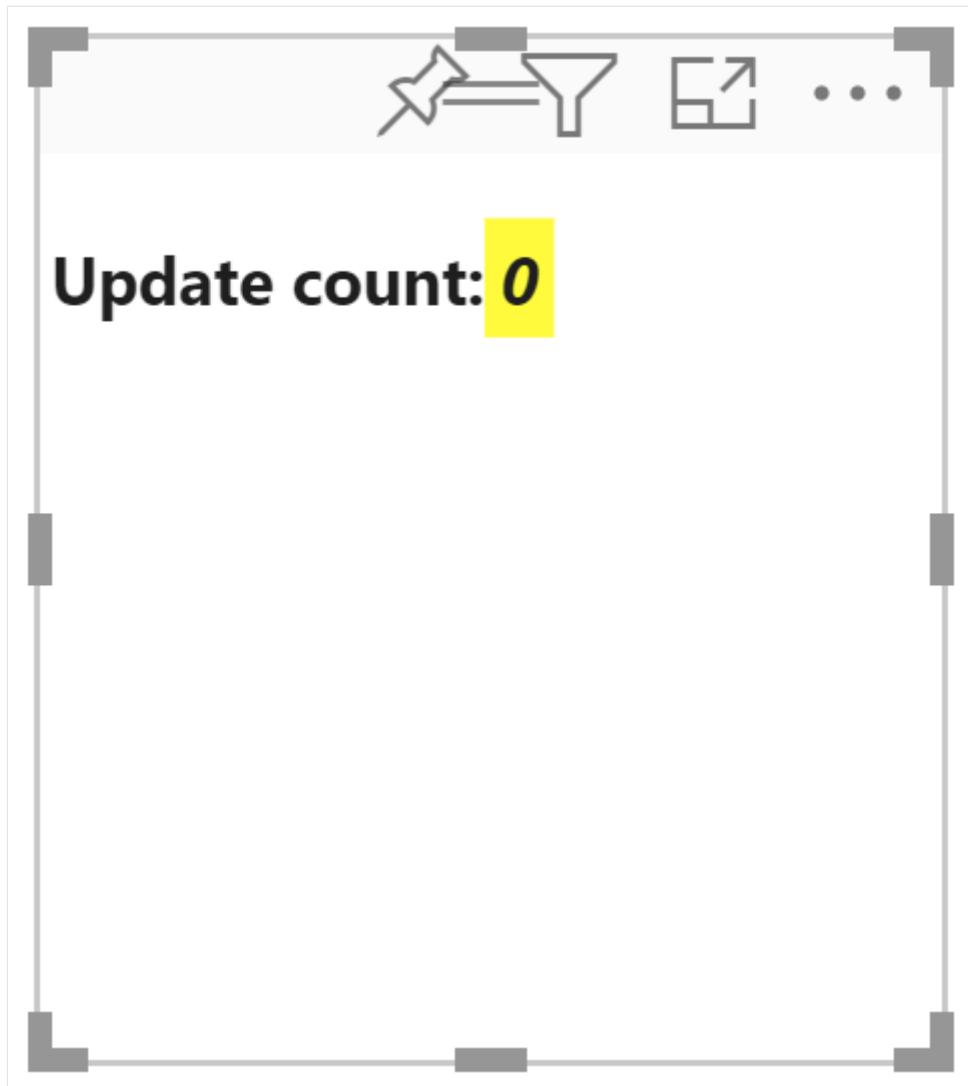


4. From the **Visualizations** pane, select the **Developer Visual**.



This visual represents the custom visual that you're running on your computer. It's only available when the [custom visual debugging](#) setting is enabled.

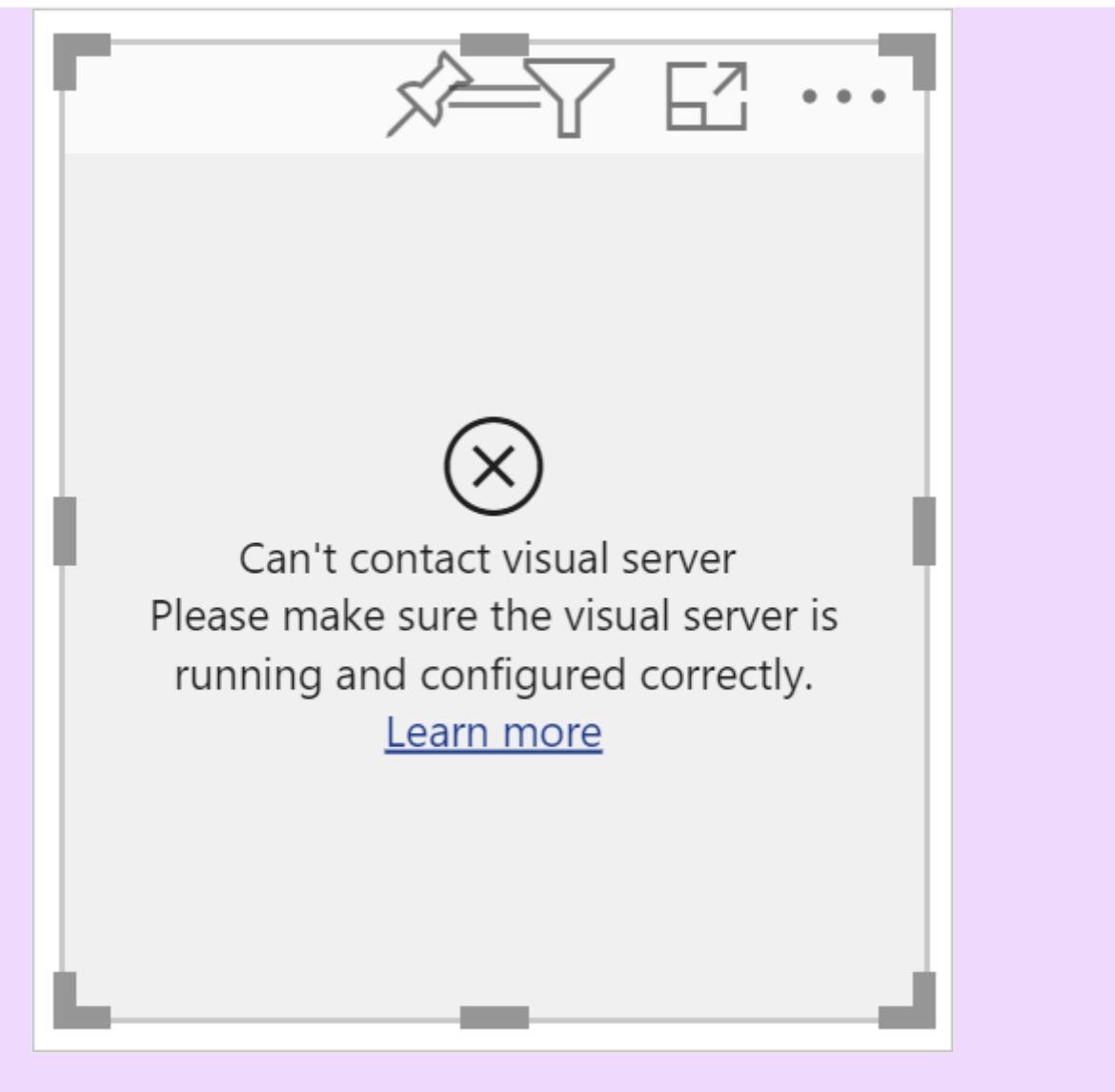
5. Verify that a visual was added to the report canvas.



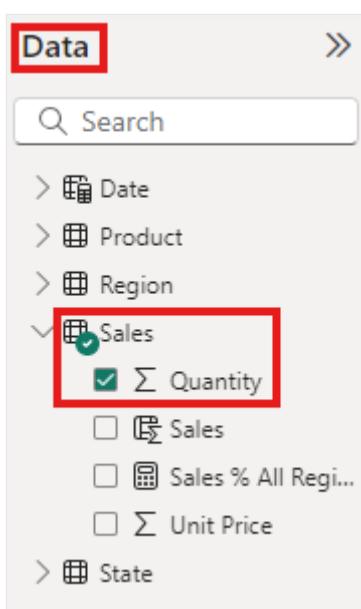
This is a simple visual that displays the number of times its update method has been called. At this stage, the visual does not retrieve any data.

 **Note**

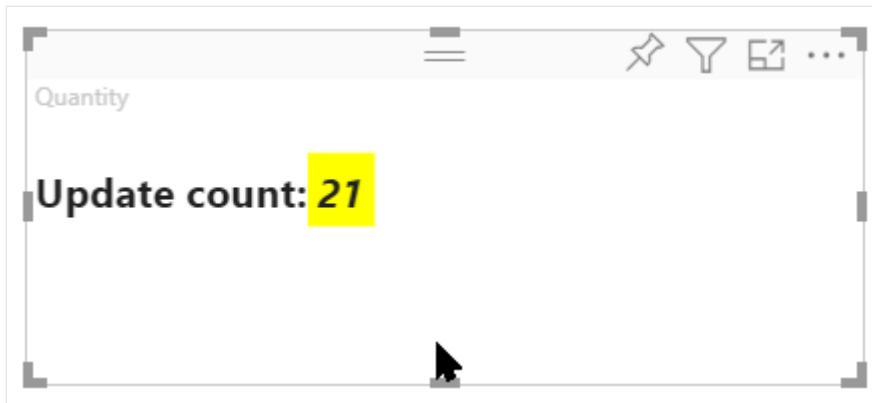
If the visual displays a connection error message, open a new tab in your browser, navigate to <https://localhost:8080/assets>, and authorize your browser to use this address.



6. While the new visual is selected, go to the **Data** pane, expand **Sales**, and select **Quantity**.



7. To test how the visual is responding, resize it and notice that the *Update count* value increments every time you resize the visual.



## Set up React in your project

In this section, you learn how to set up React for your Power BI visual project.

Open PowerShell and stop the visual from running by entering **Ctrl+C**. If prompted to terminate the batch job, enter **Y**, and press **Enter**.

### Install React

To install the required React dependencies, open PowerShell in your *ReactCircleCard* folder, and run the following command:

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

### Install React type definitions

To install type definitions for React, open PowerShell in your *reactCircleCard* folder and run the following command:

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

### Create a React component class

Follow these steps to create a React component class.

1. Open **VS Code** and navigate to the *reactCircleCard* folder.
2. Create a new file by selecting **File > New File**.

3. Copy the following code into the new file.

```
TypeScript

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** and navigate to the **src** folder.

5. Save the file as follows:

- In the *File name* field, enter **component**.
- From the *Save as type* drop-down menu, select **TypeScript React**.

## Add React to the visual file

Replace the code in the **visual.ts** file with code that enables using React.

1. In the **src** folder, open **visual.ts** and replace the code in the file with the following code:

```
TypeScript

"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 React dependencies and the added component
import * as React from "react";
import * as ReactDOM from "react-dom";
import ReactCircleCard from "./component";

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

```
export class Visual implements IVisual {

    constructor(options: VisualConstructorOptions) {

    }

    public update(options: VisualUpdateOptions) {

    }
}
```

### ⓘ Note

As default Power BI TypeScript settings don't recognize React `tsx` files, VS Code highlights `component` as an error.

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

- In the `src` folder, open `visual.ts`.
- Add the following code to the `Visual` class:

#### TypeScript

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

- Add the following lines to the `VisualConstructorOptions` constructor:

#### TypeScript

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

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

Your `visual.ts` file should now look like this:

#### TypeScript

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

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

export class Visual implements IVisual {
    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);
    }

    public update(options: VisualUpdateOptions) {
    }
}

```

### 3. Save `visual.ts`.

## Edit the `tsconfig` file

Edit the `tsconfig.json` to work with React.

1. In the `reactCircleCard` folder, open `tsconfig.json` and add two lines to the beginning of the `compilerOptions` item.

JSON

```

"jsx": "react",
"types": ["react", "react-dom"],

```

Your `tsconfig.json` file should now look like this, and the `component` error in `visual.ts` should be gone.

JSON

```

{
    "compilerOptions": {
        "jsx": "react",

```

```
    "types": ["react", "react-dom"],  
    "allowJs": false,  
    "emitDecoratorMetadata": true,  
    "experimentalDecorators": true,  
    "target": "es6",  
    "sourceMap": true,  
    "outDir": "./.tmp/build/",  
    "moduleResolution": "node",  
    "declaration": true,  
    "lib": [  
        "es2015",  
        "dom"  
    ]  
    "files": [  
        "./src/visual.ts"  
    ]  
}
```

2. Save `tsconfig.json`.

## Test your visual

Open PowerShell in the `CircleCardVisual` folder, and run your project:

Bash

```
pbviz start
```

When you add a new **Developer Visual** to your report in the Power BI service, it looks like this:



## Configure your visual's data field

Configure your visual's capabilities file so that only one data field can be submitted to the visual's *Measure data* field.

1. In VS Code, from the `reactCircleCard` folder, open `capabilities.json`.

2. The `ReactCircleCard` displays a single value, `Measure Data`. Remove the `Category Data` object from `dataRoles`.

After removing the `Category Data` object, the `dataRoles` key looks like this:

JSON

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

3. Remove all the content of the `objects` key (you'll fill it in later).

After you remove its content, the `objects` key looks like this:

JSON

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

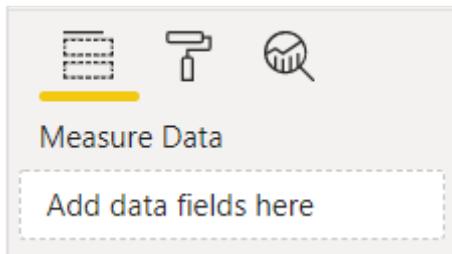
4. Replace the `dataViewMappings` property with the following code. `max: 1` in `measure` specifies that only one data field can be submitted to the visual's *Measure data* field.

JSON

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

5. Save the changes you made to `capabilities.json`.

6. Verify that `pbviz start` is running and in Power BI service, refresh your *React Circle Card* visual. The **Measure data** field can accept only one data field, as specified by `max: 1`.



## Update the visual's style

In this section, you turn the visual's shape into a circle. Use the `visual.less` file to control the style of your visual.

1. From the `style` folder, open `visual.less`.
2. Replace the content of `visual.less` with the following code.

```
css

.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. Save `visual.less`.

## Set your visual to receive properties from Power BI

In this section you configure the visual to receive data from Power BI, and send updates to the instances in the `component.tsx` file.

## Render data using React

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

1. In VS Code, from the `reactCircleCard` folder, open `component.tsx`.

2. Replace the content of `component.tsx` with the following code.

JavaScript

```
import * as React from "react";

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>
        )
    }
}
```

3. Save `component.tsx`.

## Set your visual to receive data

Visuals receive data as an argument of the `update` method. In this section, you update this method to receive data.

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

1. In VS Code, from the `src` folder, open `visual.ts`.
2. Replace the line `import ReactCircleCard from "./component";` with the following code:

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

3. Add the following code to the `update` method.

```
TypeScript  
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();  
}
```

4. Create a `clear` method by adding the following code below the `update` method.

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

5. Save `visual.ts`

## Set your visual to send data

In this section, you update the visual to send updates to instances in the `component` file.

1. In VS Code, from the `src` folder, open `component.tsx`.

2. Add the following code to the `ReactCircleCard` class:

```
TypeScript

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); 
    }
}

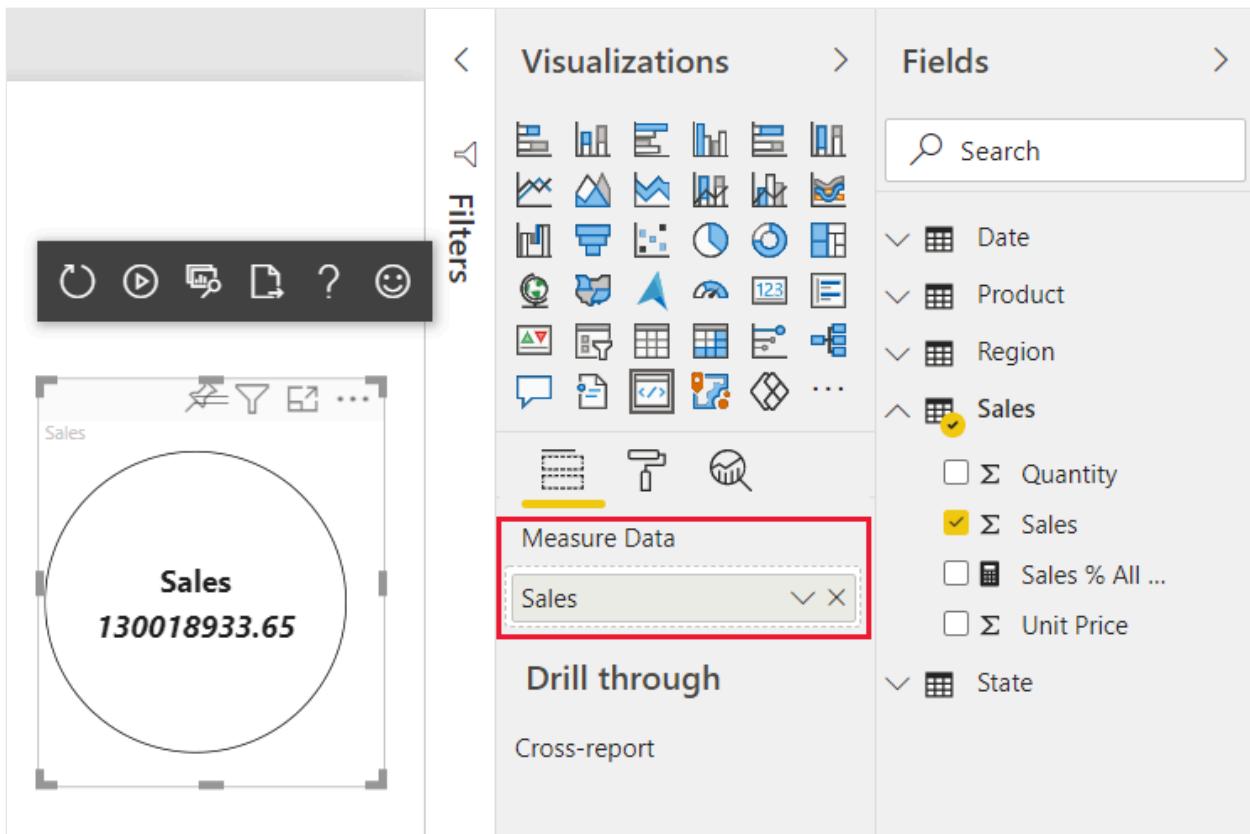
public componentWillUnmount() {
    ReactCircleCard.updateCallback = null;
}
```

3. Save `component.tsx`.

## View the changes to the visual

Test your *React Circle Card* visual to view the changes you made.

1. Verify that `pbviz start` is running, and in the Power BI service, refresh your *React Circle Card* visual.
2. Add **Sales** to the visual's *Measure data* field.



## Make your visual resizable

Currently, your visual has fixed width and height. To make the visual resizable you need to define the `size` variable in both the `visual.ts` and `component.tsx` files. In this section, you make the visual resizable.

After you complete the steps outlined in this section, the circle diameter in your visual will correspond to the minimal width or height size, and you'll be able to resize it in Power BI service.

### Configure the `visual.ts` file

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

1. In VS Code, from the `src` folder, open `visual.ts`.
2. Insert this code to import the `IViewport` interface.

```
TypeScript  
  
import IViewport = powerbi.IViewport;
```

3. Add the `viewport` property to the `visual` class.

TypeScript

```
private viewport: IViewport;
```

4. In the `update` method, before `ReactCircleCard.update`, add the following code.

TypeScript

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

5. In the `update` method, in `ReactCircleCard.update`, add `size`.

TypeScript

```
size,
```

6. Save `visual.ts`.

## Configure the component.tsx file

1. In VS Code, from the `src` folder, open `component.tsx`.

2. Add the following code to `export interface State`.

TypeScript

```
size: number
```

3. Add the following code to `export const initialState: State`.

TypeScript

```
size: 200
```

4. In the `render` method, make the following changes to the code:

- a. Add `size` to `const { labelText, textView, size } = this.state;`. This declaration should now look like this:

TypeScript

```
const { textLabel, textValue, size } = this.state;
```

- b. Add the following code above `return`.

TypeScript

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

- c. Replace the first `return` line `<div className="circleCard">` with:

TypeScript

```
<div className="circleCard" style={style}>
```

5. Save `component.tsx`.

## Configure the visual file

1. In VS Code, from the `style` folder, open `visual.less`.

2. In `.circleCard`, replace `width` and `height` with `min-width` and `min-height`.

css

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

3. Save `visual.less`.

## Make your Power BI visual customizable

In this section, you add the ability to customize your visual, allowing users to make changes to its color and border thickness.

## Add color and thickness to the capabilities file

Add the color and border thickness to the `object` property in `capabilities.json`.

1. In VS Code, from the `reactCircleCard` folder, open `capabilities.json`.

2. Add the following settings to the `objects` property.

### JSON

```
"circle": {
    "properties": {
        "circleColor": {
            "type": {
                "fill": {
                    "solid": {
                        "color": true
                    }
                }
            }
        },
        "circleThickness": {
            "type": {
                "numeric": true
            }
        }
    }
}
```

3. Save `capabilities.json`.

## Add a circle formatting settings class to the settings file

Add the `Circle` formatting settings to `settings.ts`. For more information how to build formatting model settings, see [formatting utils](#).

1. In VS Code, from the `src` folder, open `settings.ts`.
2. Replace the code in `settings.ts` with the following code:

### TypeScript

```
"use strict";

import { formattingSettings } from "powerbi-visuals-utils-
formattingmodel";

import FormattingSettingsCard = formattingSettings.SimpleCard;
import FormattingSettingsSlice = formattingSettings.Slice;
import FormattingSettingsModel = formattingSettings.Model;

/**
 * Circle Formatting Card
 */
class CircleCardSettings extends FormattingSettingsCard {
    circleColor = new formattingSettings.ColorPicker({
        name: "circleColor", // circle color name should match circle
        color property name in capabilities.json
        displayName: "Color",
    })
}
```

```

        description: "The fill color of the circle.",
        show: true,
        value: { value: "white" }
    });

    circleThickness = new formattingSettings.NumUpDown({
        name: "circleThickness", // circle thickness name should match
        circle color property name in capabilities.json
        displayName: "Thickness",
        description: "The circle thickness.",
        show: true,
        value: 2
    });

    name: string = "circle"; // circle card name should match circle
    object name in capabilities.json
    displayName: string = "Circle";
    show: boolean = true;
    slices: Array<FormattingSettingsSlice> = [this.circleColor,
    this.circleThickness];
}

/**
 * visual settings model class
 *
 */
export class VisualFormattingSettingsModel extends
FormattingSettingsModel {
    // Create formatting settings model circle formatting card
    circleCard = new CircleCardSettings();

    cards = [this.circleCard];
}

```

### 3. Save `settings.ts`.

## Add a method to apply visual settings

Add the `getFormattingModel` method used to apply visual settings and required imports to the `visual.ts` file.

1. In VS Code, from the `src` folder, open `visual.ts`.

2. Add these `import` statements at the top of `visual.ts`.

TypeScript

```

import { FormattingSettingsService } from "powerbi-visuals-utils-
formattingmodel";
import { VisualFormattingSettingsModel } from "./settings";

```

3. Add the following declaration to `Visual`.

TypeScript

```
private formattingSettings: VisualFormattingSettingsModel;  
private formattingSettingsService: FormattingSettingsService;
```

4. Add the `getFormattingModel` method to `Visual`.

TypeScript

```
public getFormattingModel(): powerbi.visuals.FormattingModel {  
    return  
    this.formattingSettingsService.buildFormattingModel(this.formattingSettings);  
}
```

5. In the `Visual` class, add the following code line to `constructor` to initialize

`formattingSettingsService`

TypeScript

```
this.formattingSettingsService = new FormattingSettingsService();
```

6. In the `Visual` class, add the following code to `update` to update the visual formatting settings to the latest formatting properties values.

a. Add this code to the `if` statement after `const size = Math.min(width, height);`.

TypeScript

```
this.formattingSettings =  
this.formattingSettingsService.populateFormattingSettingsModel(Visua  
lFormattingSettingsModel, options.dataViews[0]);  
const circleSettings = this.formattingSettings.circleCard;
```

b. Add this code to `ReactCircleCard.update` after `size`:

TypeScript

```
borderWidth: circleSettings.circleThickness.value,  
background: circleSettings.circleColor.value.value,  
}
```

7. Save `visual.ts`.

## Edit the component file

Edit the component file so that it can render the changes to the visual's color and border thickness.

1. In VS Code, from the `src` folder, open `component.tsx`.

2. Add these values to `state`:

```
TypeScript  
  
background?: string,  
borderWidth?: number
```

3. In the `render` method, replace the following code lines:

a. `const { textLabel, minValue, size } = this.state;` with:

```
TypeScript  
  
const { textLabel, minValue, size, background, borderWidth } =  
this.state;
```

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

```
TypeScript  
  
const style: React.CSSProperties = { width: size, height: size,  
background, borderWidth };
```

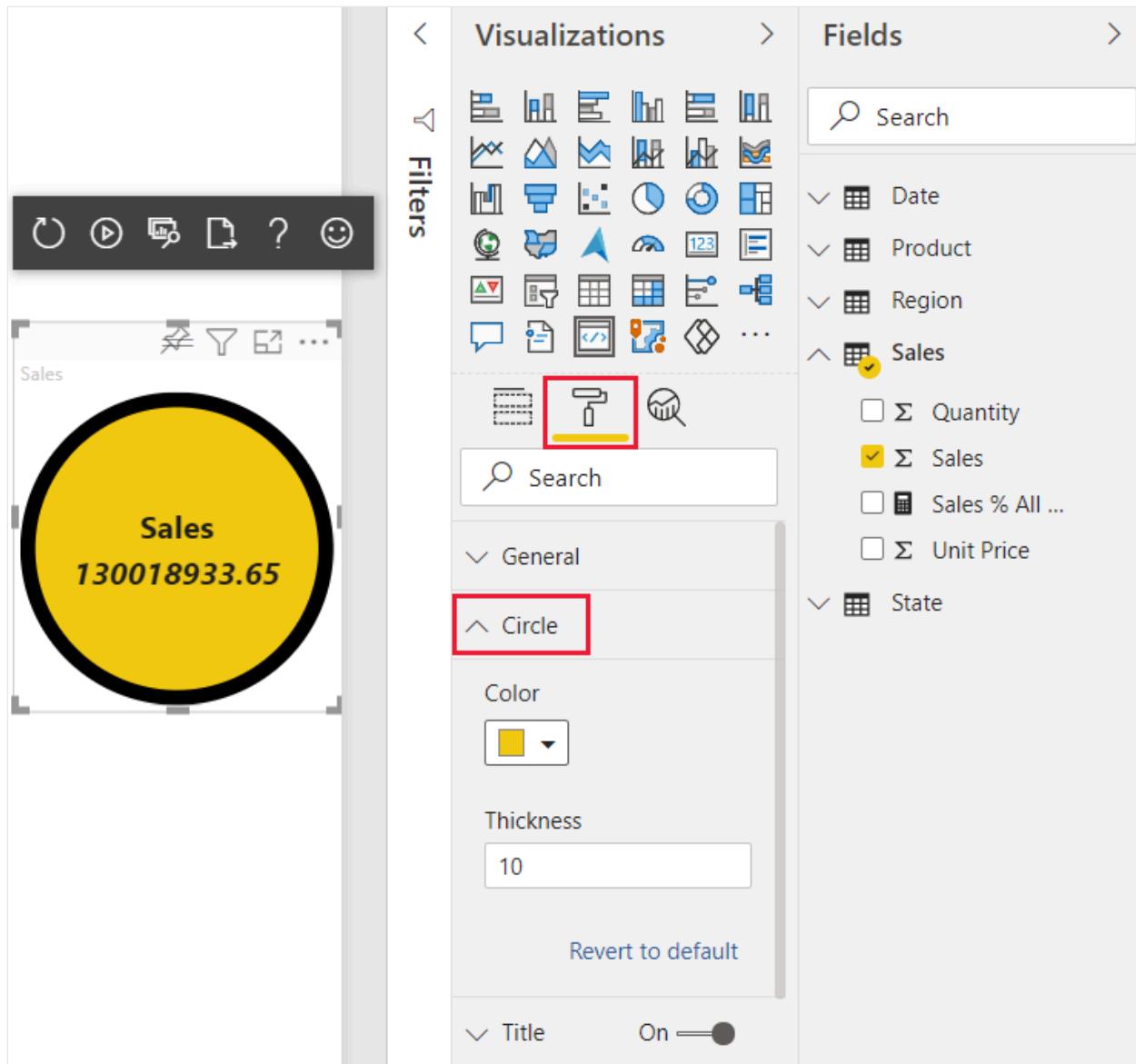
4. Save `component.tsx`.

## Review your changes

Experiment with the visual's color and border thickness, which you can now control.

1. Verify that `pbviz start` is running, and in the Power BI service, refresh your *React Circle Card* visual.
2. Select the **Format** tab and expand **Circle**.

3. Adjust the visual's **Color** and **Thickness** settings, and review their effect on the visual.



## Related content

- Add formatting options to the circle card visual
- Create a Power BI bar chart visual
- Learn how to debug a Power BI visual you created

## Feedback

Was this page helpful?

Yes

No

Provide product feedback | Ask the community

# Tutorial: Create an R-powered Power BI visual

Article • 01/11/2024

As a developer you can create your own Power BI visuals. These visuals can be used by you, your organization or by third parties.

This article is a step by step guide for creating 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 dependencies file of the visual
- ✓ Add a static property

## Prerequisites

- A **Power BI Pro** account. [Sign up for a free trial](#) before you begin.
- An R engine. You can download one free from many locations, including the [Microsoft R 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.

## Get started

1. Prepare some sample data for the visual. You can save these values to an Excel database or a .csv file and import it into [Power BI Desktop](#).

[ ] Expand table

MonthNo	Total Units
1	2303
2	2319
3	1732

MonthNo	Total Units
4	1615
5	1427
6	2253
7	1147
8	1515
9	2516
10	3131
11	3170
12	2762

2. To create a visual, open **PowerShell** or **Terminal**, and run the following command:

Windows Command Prompt

```
pbviz new rVisualSample -t rvisual
```

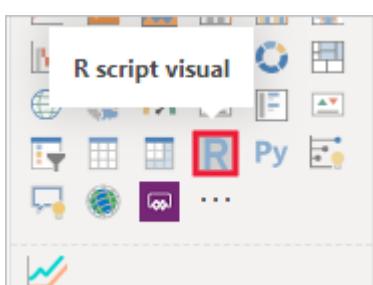
This command creates a new folder for the *rVisualSample* visual. The structure is based on the `rvisual` template. It creates a file called `script.r` in the root folder of the visual. This file holds the R-script that is run to generate the image when the visual is rendered. You can create your R-script in **Power BI Desktop**.

3. From the newly created `rVisualSample` directory run the following command:

Windows Command Prompt

```
pbviz start
```

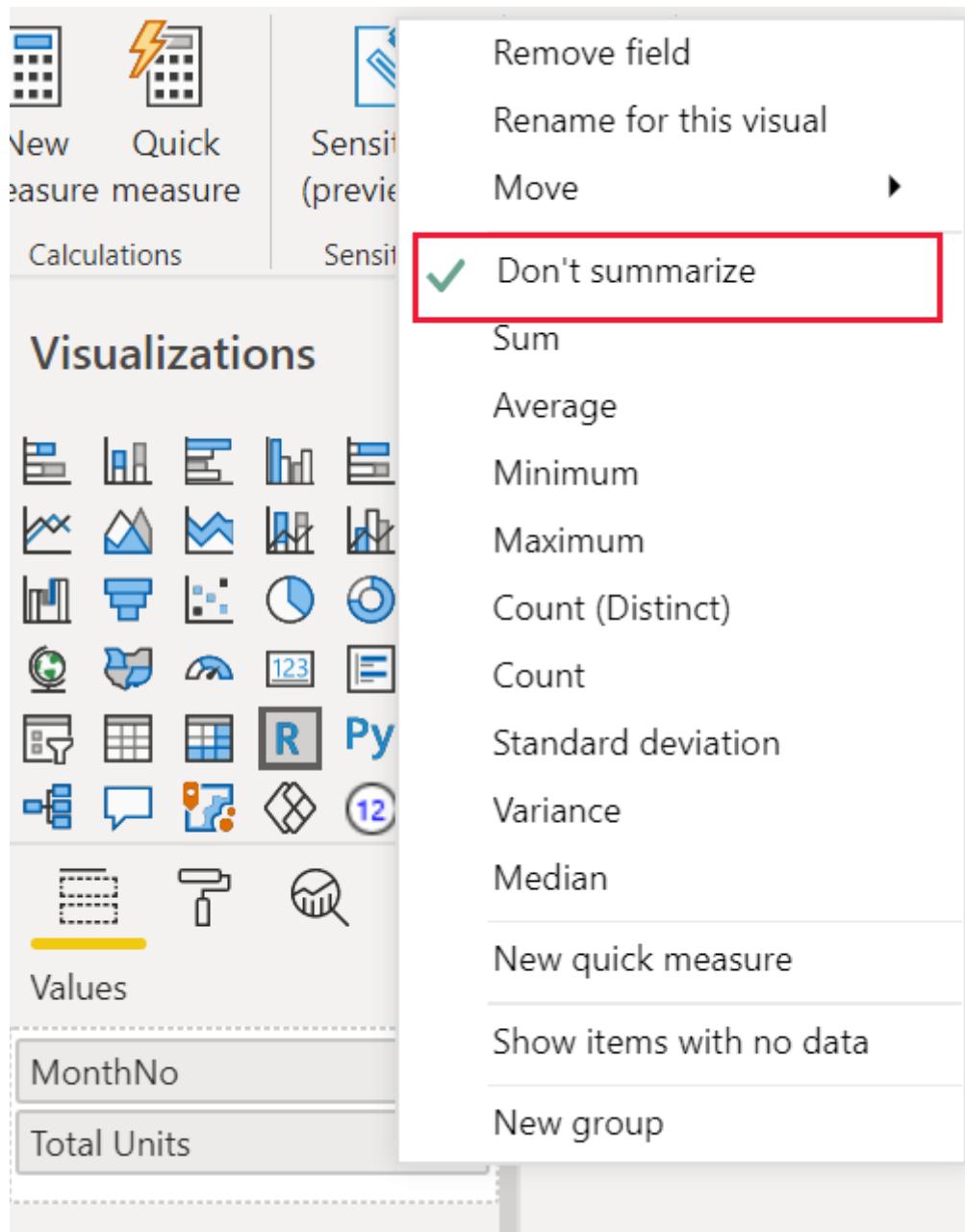
4. In **Power BI Desktop**, select **R script visual**:



5. Assign data to the developer visual by dragging MonthNo and Total units to Values for the visual.

The screenshot shows the Power BI Data view pane. On the left, there is a grid of icons representing different data types and visualizations. Below this grid, there are three rows of icons: the first row contains a bar chart, funnel, scatter plot, donut chart, and a grid icon; the second row contains a globe, map, gauge, numbers, and a chart icon; the third row contains a table, message bubble, location pin, cube, and a circled number 12. Below these rows are three more icons: a table, a paint roller, and a magnifying glass. A yellow horizontal bar is positioned below the first three icons. To the right of the icons, under the heading 'Values', there is a list of two items: 'MonthNo' and 'Total Units'. Both items have a yellow checkmark icon followed by a sigma symbol ( $\Sigma$ ) and their respective names. The entire list is enclosed in a red rectangular box.

6. Set the aggregation type of **Total units** to *Don't summarize*.

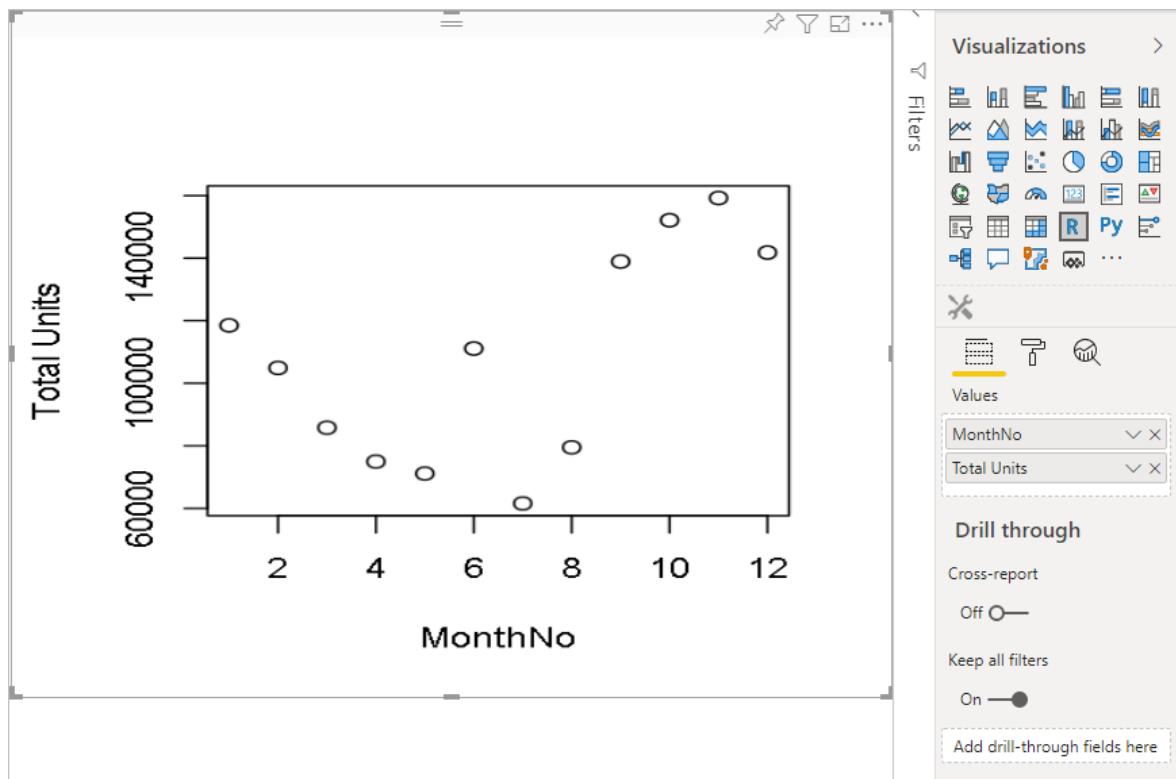


7. From the R-script editor in your **Power BI Desktop**, type the following:

```
R  
plot(dataset)
```

This command creates a scatter chart using the values in the semantic model as input.

8. Select the **Run script** icon to see the result.



## Edit the R Script

The R-script can be modified to create other types of visuals. Let's create a line chart next.

- Paste the following R code into the **R script editor**.

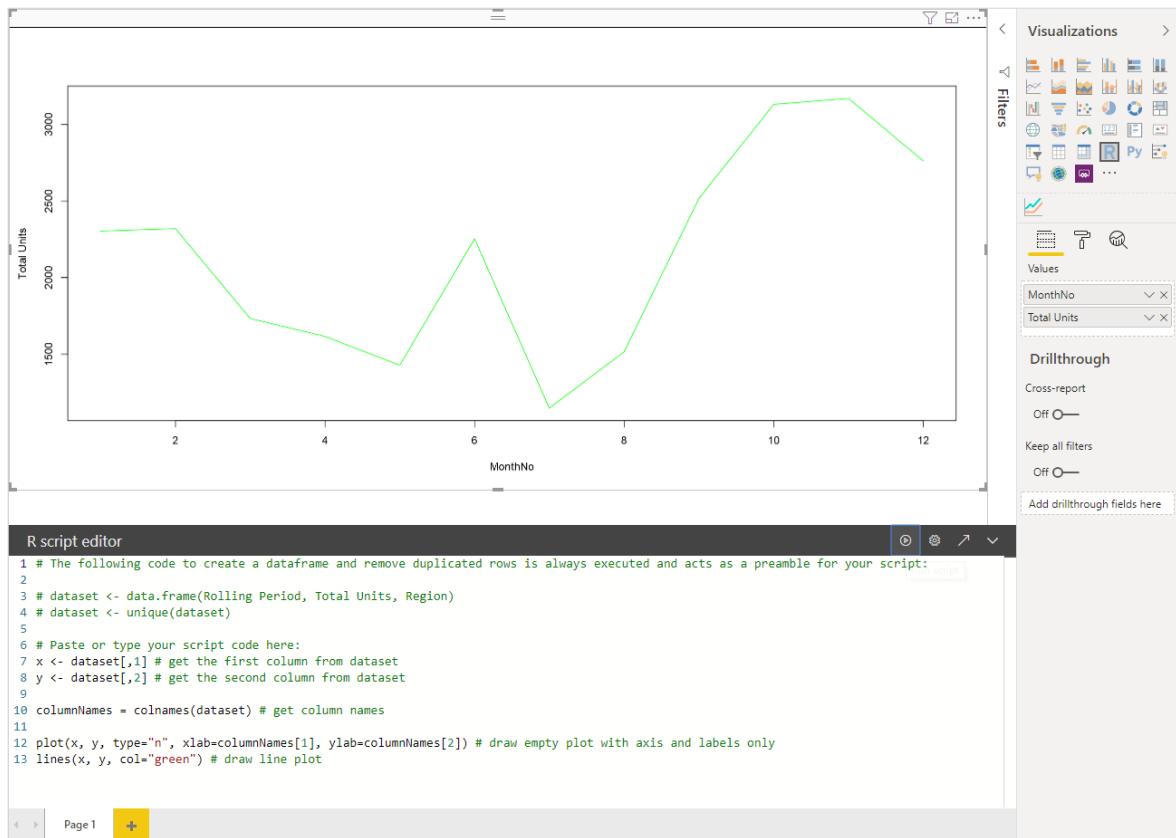
```
R

x <- dataset[,1] # get the first column from semantic model
y <- dataset[,2] # get the second column from semantic model

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
```

- Select the **Run script** icon to see the result.



3. When your R-script is ready, copy it to the `script.r` file located in the root directory of your visual project.
4. In the `capabilities.json` file, change the `dataRoles: name` to `dataset`, and set the `dataViewMappings` input to `dataset`.

JSON

```
{
  "dataRoles": [
    {
      "displayName": "Values",
      "kind": "GroupingOrMeasure",
      "name": "dataset"
    }
  ],
  "dataViewMappings": [
    {
      "scriptResult": {
        "dataInput": {
          "table": {
            "rows": {
              "select": [
                {
                  "for": {
                    "in": "dataset"
                  }
                }
              ],
              "dataReductionAlgorithm": {

```

```
        "top": {}
    }
}
},
...
}
],
}
```

5. Add the following code to support resizing the image in the `src/visual.ts` file.

TypeScript

```
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

The `corrplot` package creates a graphical display of a correlation matrix. For more information about `corrplot`, see [An Introduction to corrplot Package](#).

1. Add the `corrplot` library dependency to the `dependencies.json` file. Here is an example of the file content:

JSON

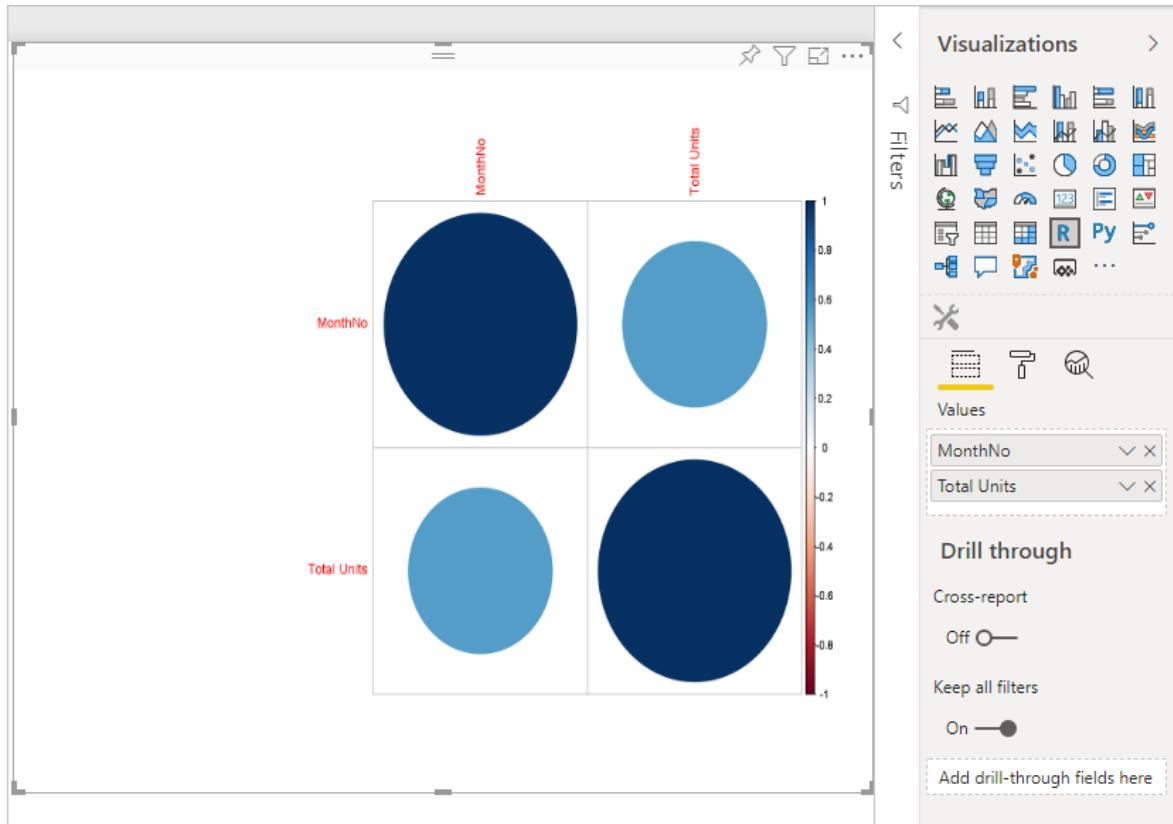
```
{
  "cranPackages": [
    {
      "name": "corrplot",
      "displayName": "corrplot",
      "url": "https://cran.r-project.org/web/packages/corrplot/"
    }
  ]
}
```

2. Now you can start using the `corrplot` package in your `script.r` file.

R

```
library(corrplot)
corr <- cor(dataset)
corrplot(corr, method="circle", order = "hclust")
```

The result of using `corrplot` package looks like this example:



## Add a static property to the property pane

Now that we have a basic `corrplot` visual, let's add properties to the property pane that allow the user to change the look and feel to the visual.

We use the `method` argument to configure the shape of the data points. The default script uses a circle. Modify your visual to let the user choose between several options.

1. Define an `object` called `settings` in the `capabilities.json` file and give it the following properties. Then use this object name in the enumeration method to get the values from the property pane.

JSON

```
{
  "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:

TypeScript

```

"use strict";

import { formattingSettings } from "powerbi-visuals-utils-
formattingmodel";

```

```

import FormattingSettingsCard = formattingSettings.SimpleCard;
import FormattingSettingsSlice = formattingSettings.Slice;
import FormattingSettingsModel = formattingSettings.Model;

/** 

 * RCV Script Formatting Card
 */
class rcvScriptCardSettings extends FormattingSettingsCard {
    provider: FormattingSettingsSlice = undefined;
    source: FormattingSettingsSlice = undefined;

    name: string = "rcv_script";
    displayName: string = "rcv_script";
    slices: Array<FormattingSettingsSlice> = [this.provider,
this.source];
}

/** 

 * visual settings model class
 *

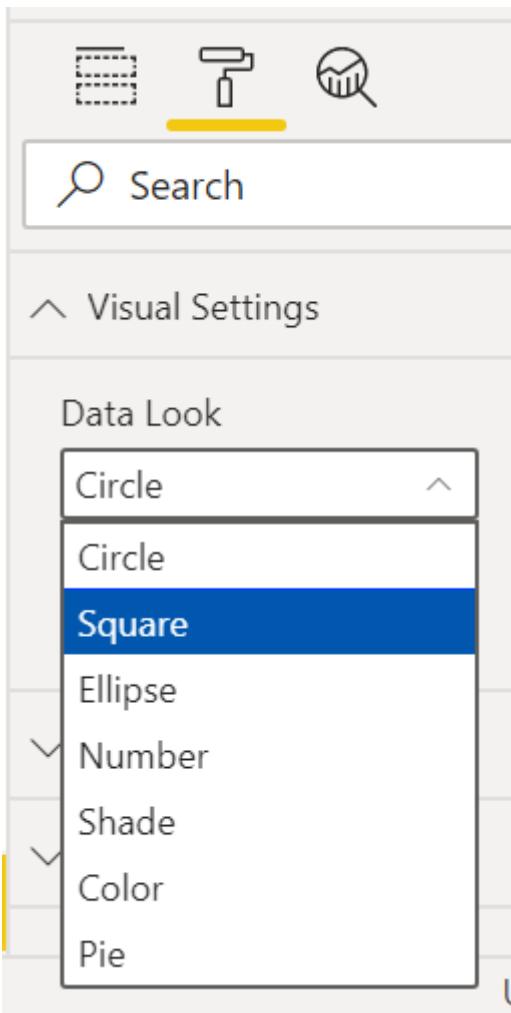
 */

export class VisualFormattingSettingsModel extends
FormattingSettingsModel {
    // Create formatting settings model formatting cards
    rcvScriptCard = new rcvScriptCardSettings();

    cards = [this.rcvScriptCard];
}

```

After these steps, you can change the property of the visual.



Finally, the R-script must have a default property. If the user doesn't change the property value (in this case, the shape setting), the visual uses this value.

For R runtime variables for the properties, the naming convention is

`<objectname>_<propertyname>`, in this case, `settings_method`.

3. Run the following R-script:

```
R

library(corrplot)
corr <- cor(dataset)

if (!exists("settings_method"))
{
    settings_method = "circle";
}

corrplot(corr, method=settings_method, order = "hclust")
```

## Package and import your visual

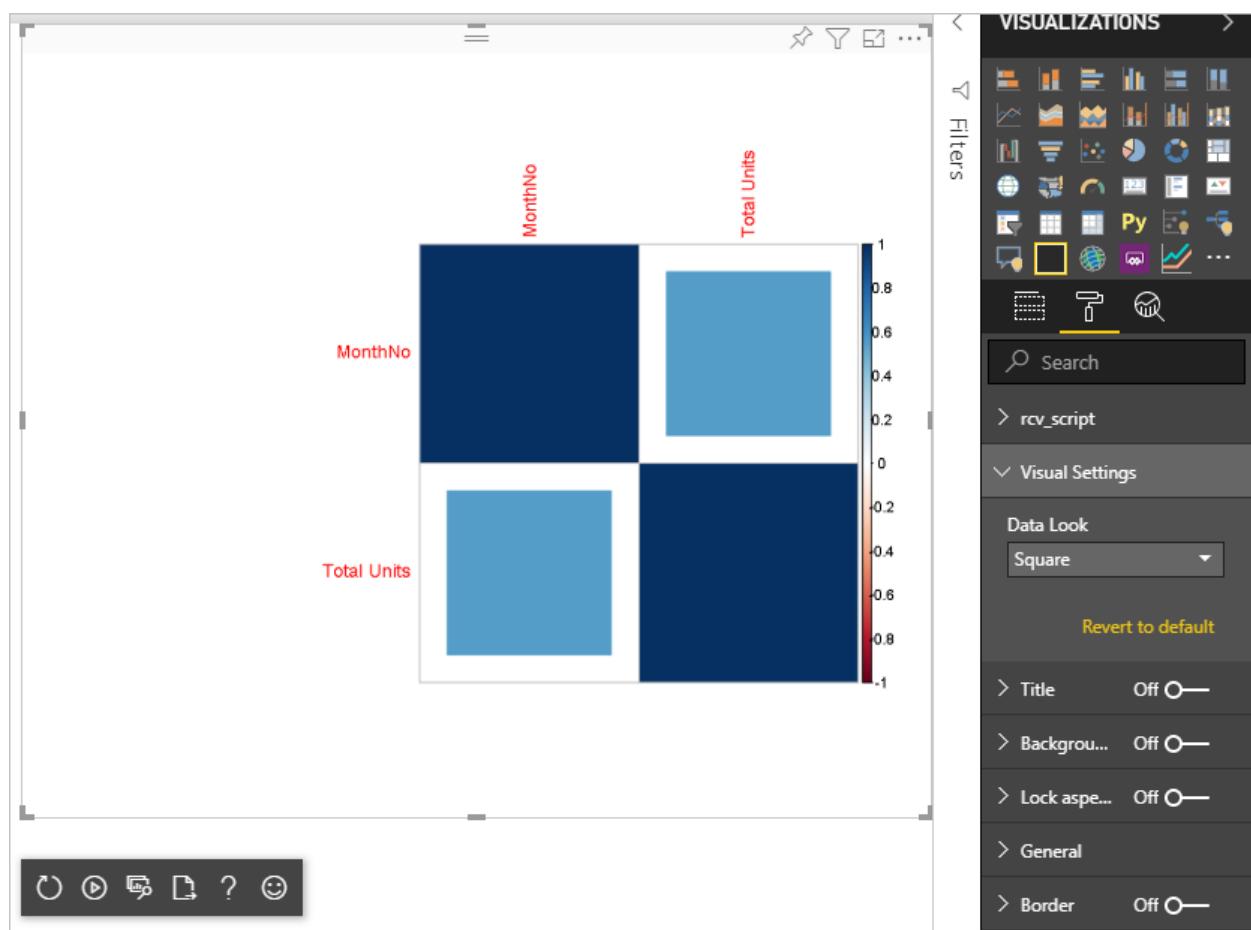
Now you can package the visual and import it to any Power BI report.

1. Fill in the `displayName`, `supportUrl`, `description`, author's `name` and `email`, and any other important information in the `pbviz.json` file.
2. If you want to change the visual's icon on the visualization pane, replace the `icon.png` file in the `assets` folder.
3. From the root directory of your visual run:

```
PowerShell  
pbviz package
```

For more information on packaging your visual see [Packaging the custom visual](#)

4. Import the visual's pbviz file to any Power BI report. See [Import a visual file from your local computer into Power BI](#) for instructions on how to do this.
5. Your final visual looks like the following example:



## Related content

- Use R-powered Power BI visuals in Power BI.
- Build a bar chart

# Tutorial: Build a funnel plot from R script to R visual

Article • 06/17/2024

This article describes how to build a funnel plot using R script in R visual step by step. Source files are available for download under each set of steps.

In this article, you learn how to create:

- ✓ an R-script for RStudio
- ✓ an R-visual in Power BI
- ✓ a PNG-based R-powered Visual in Power BI
- ✓ a HTML-based R-powered Visual in Power BI

The funnel plot provides an easy way to consume, interpret, and show the amount of expected variation. The **funnel** is formed using confidence limits and outliers are shown as dots outside the funnel.

In this example the funnel plot is used to compare and analyze various sets data.

## Prerequisites

- Get a [Microsoft Fabric subscription](#). Or, sign up for a free [Microsoft Fabric trial](#).
- Install [pbviz tools](#).

## Build an R script with semantic model

1. Download a [minimal R script](#) and its data table, [dataset.csv](#).
2. Next, edit the script to mirror [this script](#). This adds input error handling and user parameters to control the plot's appearance.

## Build a report

Next, edit the script to mirror [this script](#). This loads *dataset.csv* instead of *read.csv* into the Power BI desktop workspace and creates a **Cancer Mortality** table. See the results in the following [PBIX file](#).

 Note

The `dataset` is a hard-coded name for the input `data.frame` of any R-visual.

# Create an R-powered visual and package in R code

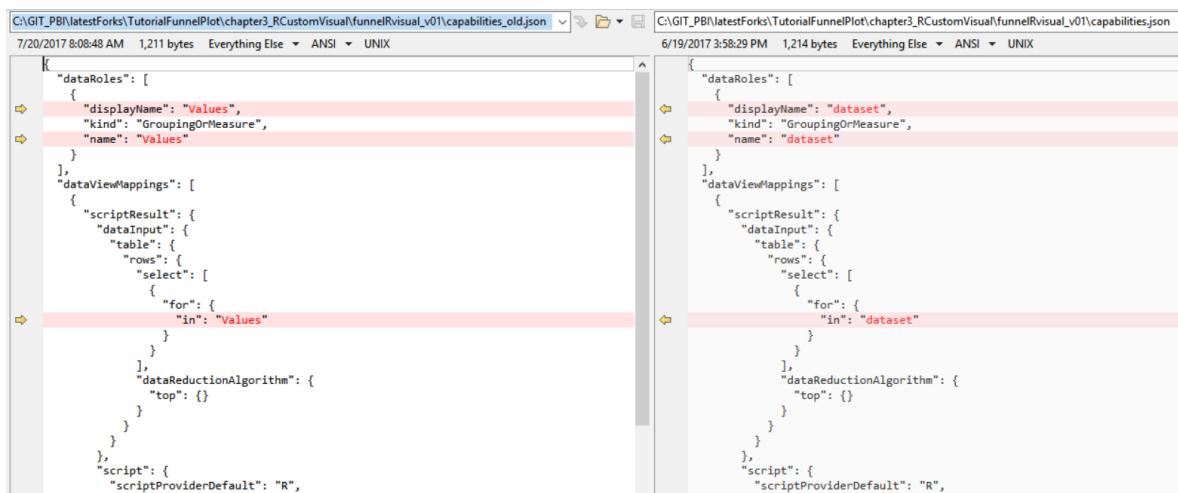
1. Run the following command to create a new R-powered visual:

Bash

```
pbviz new funnel-visual -t rvisual
cd funnel-visual
npm install
pbviz package
```

This command creates the folder `funnel-visual` with initial template visual (`-t` for **template**). The PBIVIZ can be found in the `dist` folder, the R-code inside `script.r` file. Try to import it into Power BI and see what happens.

2. Edit `script.r` file and replace the contents with your previous script.
3. Edit `capabilities.json` and replace the string `Values` with `dataset`. This replaces the name of "Role" in the template to be like in R-code.



4. (optional) Edit `dependencies.json` and add a section for each R package required by the R script. This tells Power BI to automatically import these packages when the visual is loaded for the first time.

```

"cranPackages": [
  {
    "name": "ggplot2",
    "displayName": "GG Plot 2",
    "url": "https://cran.r-project.org/web/packages/ggplot2/index.html"
  },
  {
    "name": "scales",
    "displayName": "scales: Scale Functions for Visualization",
    "url": "https://cran.r-project.org/web/packages/scales/index.html"
  }
]

```

- Repackage the visual using the `pbviz` package command and try to import it into Power BI.

**Note**

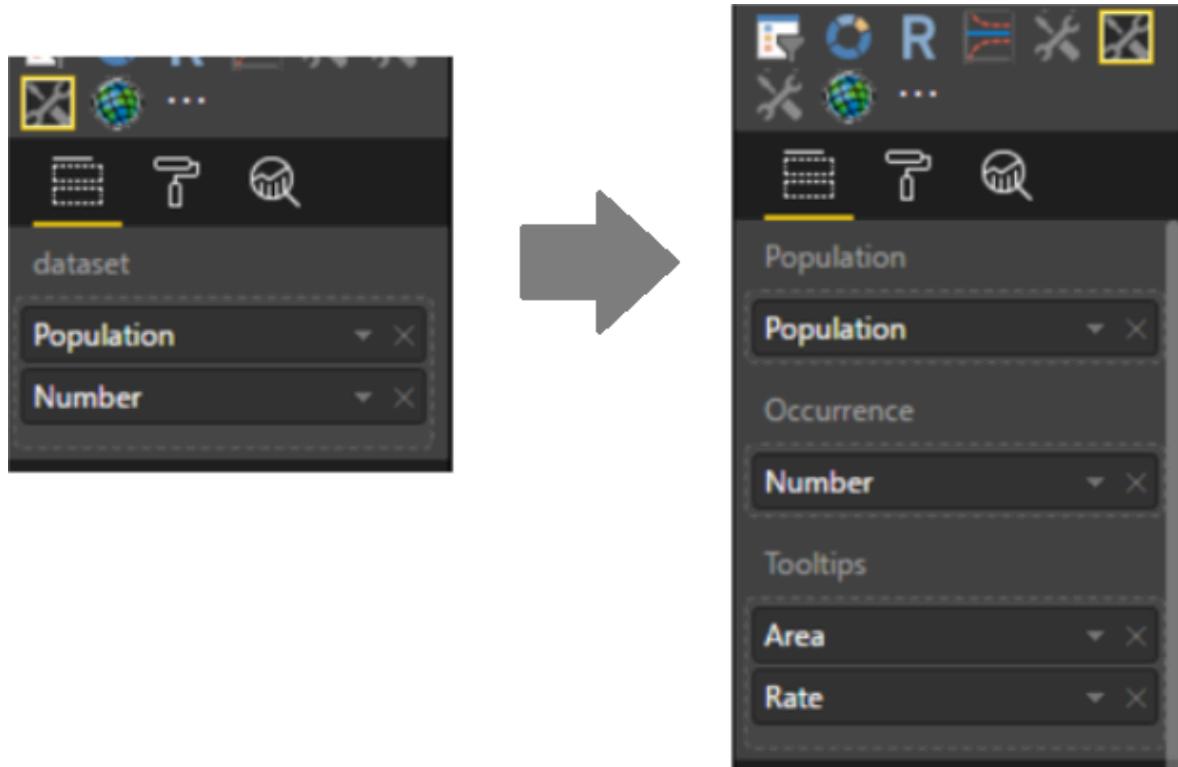
See [PBIX](#) and [source code](#) for download.

## Make R-based visual improvements

The visual isn't yet user-friendly because the user has to know the order of columns in the input table.

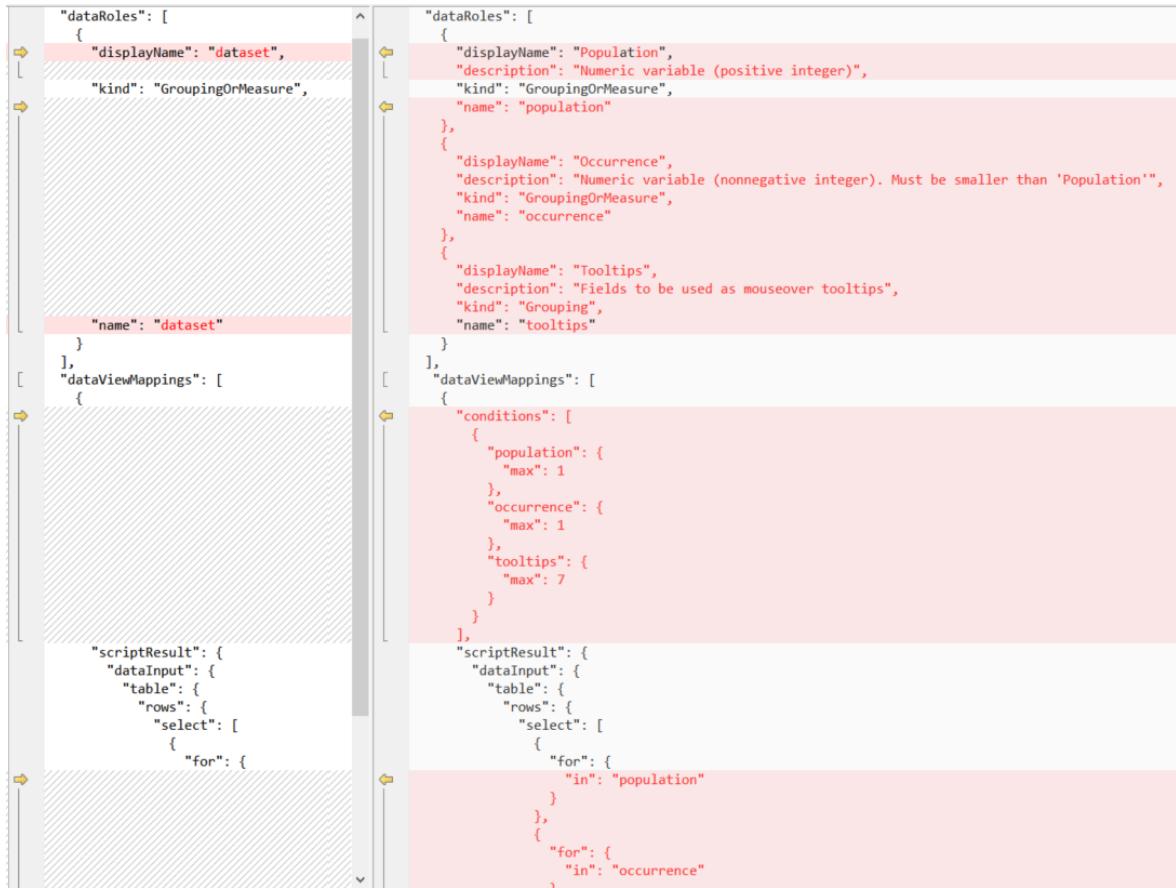
- Divide the input field `dataset` into three fields (roles): `Population`, `Number`, and

`Tooltips`



2. Edit `capabilities.json` and replace the `dataset` role with the three new roles, or download [capabilities.json](#).

You'll need to update sections: `dataRoles` and `dataViewMappings`, which define names, types, tooltips, and maximum columns for each input field.



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

3. Edit `script.r` to support `Population`, `Number` and `Tooltips` as input dataframes instead of `dataset`, or download [script.r](#).

```

        }
        if(validToPlot)
        {
            validData = complete.cases(dataset) & (dataset[,1]>=dataset[,2])
        }

        if(validToPlot && (sum(validData) < minPoints)) # not enough data samples
        {
            validToPlot = FALSE
        }

        ###### Main code #####
        if(validToPlot)
        {
            dataset = dataset[validData,]# keep only valid
            namesDS = names(dataset)
        }

        if(validToPlot)
        {
            # take care of NaN and NULL values
            population[is.na(population)] = 0
            occurrence[is.na(occurrence)] = -1
            population[is.null(population)] = 0
            occurrence[is.null(occurrence)] = -1

            #clean data
            validData = rep(TRUE,nrow(population))
            validData = as.logical(validData & (population > 1) & (occurrence >= 0) & (occurrence <= population ))
        }

        if(validToPlot && (sum(validData) < minPoints)) # not enough data samples
        {
            validToPlot = FALSE
        }

        ###### Main code #####
        if(validToPlot)
        {
            #RVIZ_IN_PBI_GUIDE:BEGIN:Added to enable custom visual fields
            if(exists("tooltips"))
            {
                dataset = cbind(population,occurrence)
            }else{
                dataset = cbind(population,occurrence,tooltips)
            }
            #RVIZ_IN_PBI_GUIDE:END:Added to enable custom visual fields
            dataset = dataset[validData,]# keep only valid
            namesDS = names(dataset)
        }
    }
}

```

### 💡 Tip

To follow the changes in R-script, search for comment blocks:

R

```

#RVIZ_IN_PBI_GUIDE:BEGIN: Added to enable visual fields
...
#RVIZ_IN_PBI_GUIDE:END: Added to enable visual fields

#RVIZ_IN_PBI_GUIDE:BEGIN: Removed to enable visual fields
...
#RVIZ_IN_PBI_GUIDE:BEGIN: Removed to enable visual fields

```

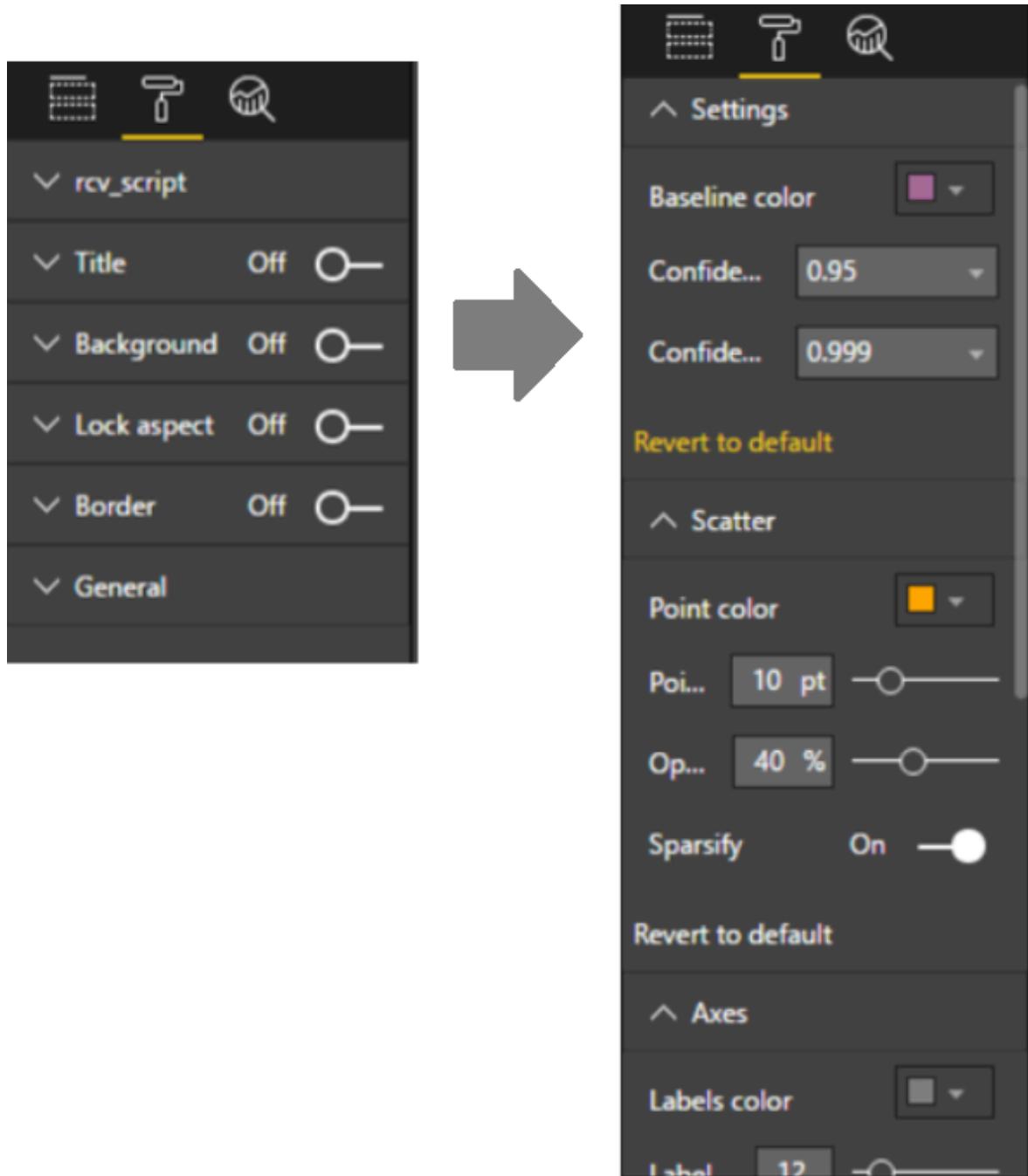
4. Repackage the visual using the `pbviz` package command and try to import it into Power BI.

### ⓘ Note

See [PBIX](#) and [source code](#) for download.

## Add user parameters

1. Add capabilities for the user to control colors and sizes of visual elements including internal parameters from the UI.



2. Edit `capabilities.json` and update the `objects` section. Here we define names, tooltips and types of each parameter, and also decide on the partition of parameters into groups (three groups in this case).

download [capabilities.json](#), see [object properties](#) for more information

```

1  {
2    "dataRoles": [
3      {
4        {
5          {
6            "dataViewMappings": [
7              {
8                "objects": [
9                  {
10                    "rcv_script": {
11                      "settings_funnel_params": {
12                        "display_name": "Funnel Settings",
13                        "description": "Funnel Settings Description"
14                      }
15                    },
16                    "settings_scatter_params": {
17                      "display_name": "Scatter Settings",
18                      "description": "Scatter Settings Description"
19                    },
20                    "settings_axes_params": {
21                      "display_name": "Axes Settings",
22                      "description": "Axes Settings Description"
23                    }
24                  }
25                }
26              }
27            ]
28          }
29        }
30      ]
31    }
32  }

```

### 3. Edit `src/settings.ts` to mirror [this settings.ts](#). This file is written in TypeScript.

Here you'll find two blocks of the code added to:

- Declare new interface to hold the property value
- Define a member property and default values

```

/*
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */
module powerbi.extensibility.visual {
  "use strict";
  import DataViewObjectsParser = powerbi.extensibility.utils.dataview.DataViewObjectsParser;

  export class VisualSettings extends DataViewObjectsParser {
    public rcv_script: rcv_scriptSettings = new rcv_scriptSettings();

    ...
  }

  export class rcv_scriptSettings {
    ...
    public provider; // undefined
    public source; // undefined
  }
}

export class settings_funnel_params {
  public lineColor: string = "blue";
  public conf1: string = "0.95";
  public conf2: string = "0.99";
}

export class settings_scatter_params {
  public pointColor: string = "orange";
  public weight: number = 10;
  public percentile: number = 40;
  public sparsify: boolean = true;
}

export class settings_axes_params {
  public collab1: string = "gray";
  public textSize: number = 12;
  public axisXisPercentage: boolean = true;
  public scaleFormat: string = "comma";
  public scaleFormat: string = "none";
  public sizeTicks: string = "6";
}

```

### 4. Edit `script.r` to mirror [this script.r](#). This adds support for the parameters in the UI by adding `if.exists` calls per user-parameter.



To follow the changes in R-script, search for comments:

R

```
#RVIZ_IN_PBI_GUIDE:BEGIN:Added to enable user parameters
...
#RVIZ_IN_PBI_GUIDE:END:Added to enable user parameters

#RVIZ_IN_PBI_GUIDE:BEGIN:Removed to enable user parameters
...
#RVIZ_IN_PBI_GUIDE:END:Removed to enable user parameters
```

```
C:\GIT_PBI\testForks\TutorialFunnelPlot\chapter3_RCustomVisual\funnelRvisual_v03\script_old.r C:\GIT_PBI\testForks\TutorialFunnelPlot\chapter3_RCustomVisual\funnelRvisual_v03\script.r
6/20/2017 8:05:26 AM 13,026 bytes Everything Else ▾ ANSI ▾ PC
#PBI_PARAM Color of scatterplot points
#Type:string, Default:"orange", Range:NA, PossibleValues:"orange","blue","green","black"
pointsCol = "orange"

#PBI_PARAM Transparency of scatterplot points
#Type:numeric, Default:0.4, Range:[0,1], PossibleValues:NA, Remarks: NA
transparency = 0.4

##PBI_PARAM Color of baseline
#Type:string, Default:"blue", Range:NA, PossibleValues:"orange","blue","green","black"
lineColor = "blue"

#PBI_PARAM Sparsification of scatterplot points
#Type:bool, Default:TRUE, Range:NA, PossibleValues:NA, Remarks: NA
sparsify = TRUE

#PBI_PARAM Size of points on the plot
#Type:numeric, Default: 1 , Range:[0,1,5], PossibleValues:NA, Remarks: NA
pointCex = 1

#PBI_PARAM Confidence level line
#Type:numeric, Default: 0.75 , Range:[0,1], PossibleValues:NA, Remarks: GUI input is predefined
conf1 = 0.95

#PBI_PARAM Confidence level line #2
#Type:numeric, Default: 0.95 , Range:[0,1], PossibleValues:NA, Remarks: NA

6/20/2017 10:09:36 AM 16,066 bytes Everything Else ▾ ANSI ▾ PC
##PBI_PARAM Color of scatterplot points
#Type:string, Default:"orange", Range:NA, PossibleValues:"orange","blue","green","black"
pointsCol = "orange"
if(exists("settings_scatter_params_pointColor")){
  pointsCol = settings_scatter_params_pointColor
}

#PBI_PARAM Transparency of scatterplot points
#Type:numeric, Default:0.4, Range:[0,1], PossibleValues:NA, Remarks: NA
transparency = 0.4
if(exists("settings_scatter_params_percentile")){
  transparency = settings_scatter_params_percentile/100
}

##PBI_PARAM Color of baseline
#Type:string, Default:"blue", Range:NA, PossibleValues:"orange","blue","green","black"
lineColor = "blue"
if(exists("settings_funnel_params_lineColor")){
  lineColor = settings_funnel_params_lineColor
}

#PBI_PARAM Sparsification of scatterplot points
#Type:bool, Default:TRUE, Range:NA, PossibleValues:NA, Remarks: NA
sparsify = TRUE
if(exists("settings_scatter_params_sparsify")){
  sparsify = settings_scatter_params_sparsify
}

#PBI_PARAM Size of points on the plot
#Type:numeric, Default: 1 , Range:[0,1,5], PossibleValues:NA, Remarks: NA
pointCex = 1
if(exists("settings_scatter_params_weight")){
  pointCex = min(50,max(settings_scatter_params_weight,1))/10
}

#PBI_PARAM Confidence level line
#Type:numeric, Default: 0.75 , Range:[0,1], PossibleValues:NA, Remarks: GUI input is predefined set of values
conf1 = 0.95
if(exists("settings_funnel_params_conf1")){
  conf1 = as.numeric(settings_funnel_params_conf1)
}

#PBI_PARAM Confidence level line #2
#Type:numeric, Default: 0.95 , Range:[0,1], PossibleValues:NA, Remarks: NA
```

You can decide not to expose the parameters to the UI, like we did.

5. Repackage the visual using the `pbiviz` package command and try to import it into Power BI.

### ⚠ Note

See [PBIX](#) and [source code](#) for download.

### 💡 Tip

Here we added parameters of several types (boolean, numeric, string, and color) all at once. For a simple case, please see [this example](#) on how to add a single parameter.

## Convert visual to RHTML-based visual

Since the resulting visual is PNG-based, it isn't responsive to mouse hover, can't be zoomed in on, and so on, so we need to convert it to an HTML-based visual. We'll create an empty R-powered HTML-based Visual template, then copy some scripts from the PNG-based project.

1. Run the command:

```
Bash  
  
pbviz new funnel-visual-HTML -t rhtml  
cd funnel-visual-HTML  
npm install  
pbviz package
```

2. Open *capabilities.json* and note the `"scriptOutputType": "html"` line.
3. Open *dependencies.json* and note the names of the listed R-packages.
4. Open *script.r* and note the structure. You can open and run it in RStudio since it doesn't use external input.

This creates and saves *out.html*. This file is self-contained (with no external dependencies) and defines the graphics inside the HTML widget.

### ⓘ Important

For `htmlWidgets` users, R-utilties are provided in the [r\\_files folder](#) to help convert `plotly` or `widget` objects into self-content HTML.

This version of R-powered visual also supports the `source` command (unlike previous types of visuals), to make your code more readable.

5. Replace *capabilities.json* with the *capabilities.json* from the previous step, or download [capabilities.json](#).

Be sure to keep:

```
"scriptOutputType": "html"
```

6. Merge the latest version of *script.r* with the *script.r* from the template, or download [script.r](#).

The new script uses the `plotly` package to convert the `ggplot` object into a `plotly` object, then the `htmlWidgets` package to save it to an HTML file.

Most of the utility functions are moved to [r\\_files/utils.r](#) and the `generateNiceToolips` function is added for the appearance of the `plotly` object.

```

C:\GIT_PBI\latestForks\TutorialFunnelPlot\chapter3_RCustomVisual\funnelRvisual_v03\script.r
6/20/2017 10:09:36 AM 16,066 bytes Everything Else ▾ ANSI ▾ PC

if(!require(packageName, character.only = TRUE))
  warning(paste("The package: '", packageName, "' was not installed ***", sep=""))

libraryRequireInstall("ggplot2")
libraryRequireInstall("scales")

C:\GIT_PBI\latestForks\TutorialFunnelPlot\chapter4_RHTMLCustomVisual\funnelRHTMLvisual_v01\script.r
6/21/2017 11:46:08 AM 16,830 bytes Everything Else ▾ ANSI ▾ PC

if(!require(packageName, character.only = TRUE))
  warning(paste("The package: '", packageName, "' was not installed ***"))

#RVIZ_IN_PBI_GUIDE:BEGIN: Added to create HTML-based
source("./r_files/utils.r")
source("./r_files/flatten_HTML.r")
libraryRequireInstall("plotly")
#RVIZ_IN_PBI_GUIDE:END: Added to create HTML-based

libraryRequireInstall("ggplot2")
libraryRequireInstall("scales")

```

### Tip

To follow the changes in R-script, search for comments:

```
R

#RVIZ_IN_PBI_GUIDE:BEGIN:Added to create HTML-based
...
#RVIZ_IN_PBI_GUIDE:BEGIN:Added to create HTML-based

#RVIZ_IN_PBI_GUIDE:BEGIN:Removed to create HTML-based
...
#RVIZ_IN_PBI_GUIDE:BEGIN:Removed to create HTML-based
```

7. Merge the latest version of `dependencies.json` with the `dependencies.json` from the template, to include new R-package dependencies, or download [dependencies.json](#).

8. Edit `src/settings.ts` the same way from previous steps.

9. Repackage the visual using the `pbiviz package` command and try to import it into Power BI.

 Note

See [PBIX and source code](#) for download.

## Build additional examples

1. Run the following command to create an empty project:

Bash

```
pbiviz new example -t rhtml
cd example
npm install
pbiviz package
```

2. Take code from this [showcase](#) and make the highlighted changes:

```
library(networkD3)
source("r_files/flatten_HTML.r")
data(MisLinks, MisNodes)

p =
  forceNetwork(Links = MisLinks, Nodes = MisNodes, Source = "source",
               Target = "target", Value = "value", NodeID = "name",
               Group = "group", opacity = 0.4)

internalSaveWidget(p, 'out.html')
```

3. Replace your template's `script.r` and run `pbiviz package` again. Now the visual is included in your Power BI report!

## Tips and tricks

- We recommend that developers edit `pbiviz.json` to store correct metadata, such as `version`, `email`, `name`, `license type`, and so on.

 Important

The `guid` field is the unique identifier for a visual. If you create a new project for each visual, the GUID will be also be different. It's only the same when

using an old project copied to a new visual, which you shouldn't do.

- Edit [assets/icon.png](#) to create unique icons for your visual.
- To debug R-code in RStudio using the same data as in your Power BI report, add the following to the beginning of the R-script (edit the `fileRda` variable):

R

```
#DEBUG in RStudio
fileRda = "C:/Users/yourUserName(Temp/tempData.Rda"
if(file.exists(dirname(fileRda)))
{
  if(Sys.getenv("RSTUDIO")!="")
    load(file= fileRda)
  else
    save(list = ls(all.names = TRUE), file=fileRda)
}
```

This saves the environment from a Power BI report and loads it into RStudio.

- You don't need to develop R-powered Visuals from scratch with code available on [GitHub](#). You can select the visual to use as a template and copy the code into a new project.

For example, try using the [spline custom visual](#).

- Each R Visual applies the `unique` operator to its input table. To avoid identical rows being removed, consider adding an extra input field with a unique ID and ignore it in the R code.
- If you have a Power BI account, use the Power BI service to develop a visual [on-the-fly](#) instead of repackaging them with the `pbviz package` command.

## HTML widgets gallery

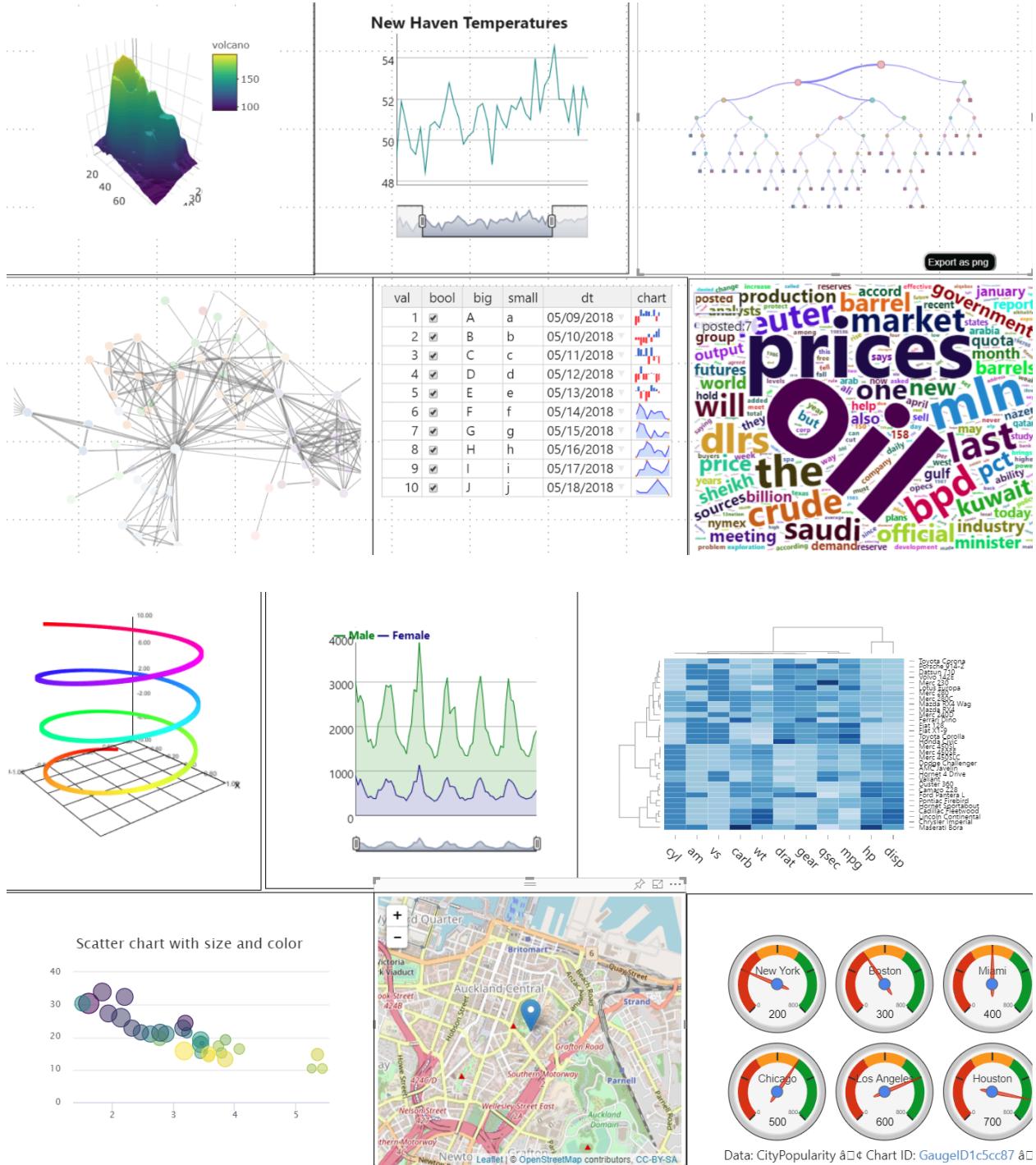
Explore visuals in the [HTML widgets gallery](#) for use in your next visual. To make things easy, we've created a [visuals project repo](#) with over 20 interactive HTML visuals to choose from!

### 💡 Tip

To switch between html widgets use **Format > Settings > Type**. Try it out with [this PBIX file](#).

# To use a sample for your visual

1. Download the entire folder.
2. Edit `script.r` and `dependencies.json` to keep only one widget.
3. Edit `capabilities.json` and `settings.ts` to remove the `Type` selector.
4. Change `const updateHTMLHead: boolean = true;` to `false` in `visual.ts`. (for better performance)
5. Change metadata in `pbviz.json`, most importantly the `guid` field.
6. Repackage and continue to customize the visual as wanted.



Note

Not all widgets in this project are supported by the service.

## Related content

To learn more, see additional Power BI tutorials, [Developing a Power BI circle card visual](#) and [R visuals](#).

Learn how to [develop and submit visuals](#) to the [Office Store \(gallery\)](#), or for further examples, see the [R-script showcase](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

# Import a Power BI visual from AppSource into your workspace

Article • 01/12/2025

Power BI comes with many out-of-the-box visuals that are available in the **Visualizations** pane of both [Power BI Desktop](#) and [Power BI Service](#).

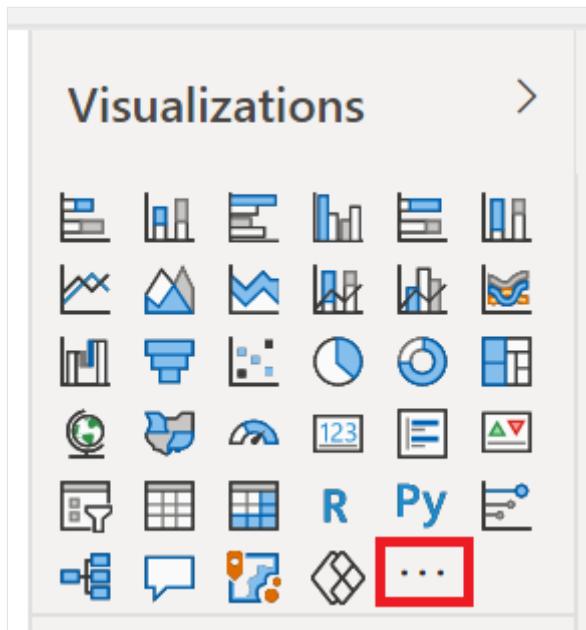
Many more certified Power BI visuals are available from [AppSource](#). These visuals are created by Microsoft and Microsoft partners, and are validated by the AppSource validation team. You can [download these visuals](#) directly to your **Visualizations** pane.

You can also [develop your own Power BI visual](#), or get one from a trusted friend or coworker.

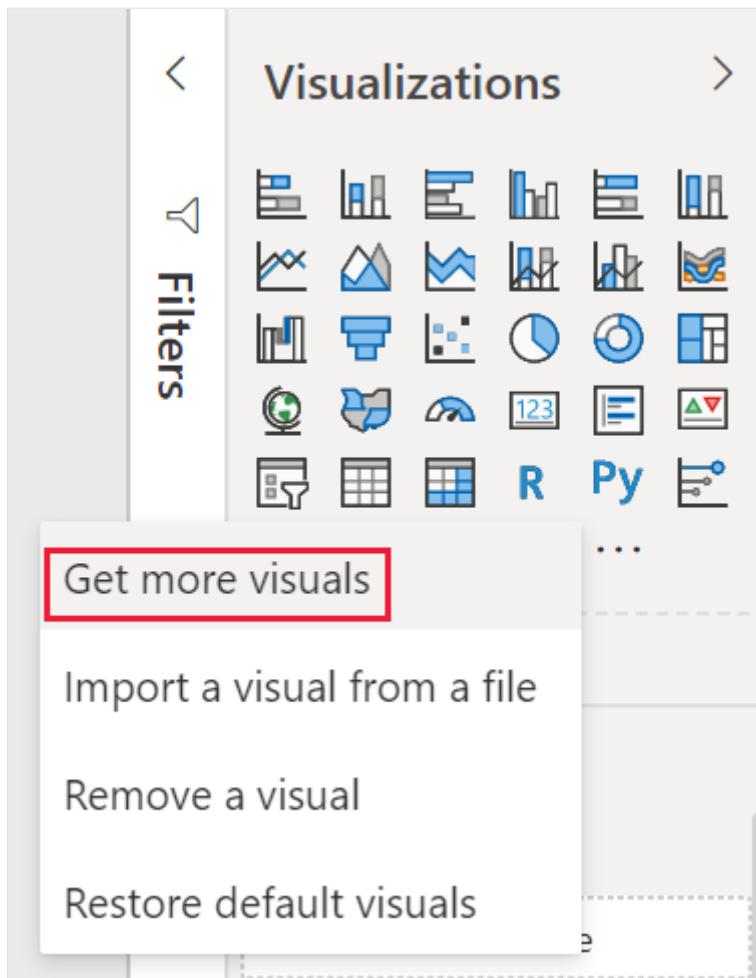
If you download or receive a Power BI visual file, you have to [import](#) it to the **Visualizations** pane before you can use it to create Power BI content.

## Import a Power BI visual directly from AppSource

1. Open your report in [Power BI Desktop](#) or [Power BI service](#).
2. Select the ellipsis from the **Visualizations** pane.



3. Select **Get more visuals** from the menu.



4. Select **AppSource visuals** and choose the visual you want to import.

The screenshot shows the 'Power BI visuals' page in the Microsoft AppSource. At the top, there's a note about agreeing to terms and conditions. Below it, tabs include 'All visuals', 'Organizational visuals', and the selected 'AppSource visuals' tab. A search bar is also present. The page lists several visual extensions, each with an icon, name, developer, rating, and download count:

Visual Extension	Developer	Rating	Downloads
Ultimate Waterfall	dataviz.boutique GmbH	★★★★★ (48)	
Drill Down Pie PRO...	ZoomCharts	★★★★★ (13)	
Journey Chart by M...	MAQ LLC	★★★★★ (14)	
Phrazor	vPhrase Analytics Solution...	★★★★★ (2)	
Floor Plan Visual by...	SIMPSON ASSOCIATES IN...	★★★★★ (2)	
Chiclet Slicer	Microsoft Corporation	★★★★★ (208)	
Text Filter	Microsoft Corporation	★★★★★ (147)	
Infographic Designer	Microsoft Corporation	★★★★★ (91)	
Word Cloud	Microsoft Corporation	★★★★★ (132)	
Advance Card	Bhavesh Jadav	★★★★★ (65)	

5. Select **Add** to add the visual to your report.

AppSource | Apps for Power BI visuals X

< Apps



**Word Cloud**  
Microsoft Corporation  
 4.2 (132)  


**Add**  
**Download Sample**

[Sample Instructions](#)

Pricing  
Free

Products  
Power BI visuals

Publisher  
Microsoft Corporation

Acquire Using  
Work or school account

Version  
2.0.0.0

Updated  
10/18/2021

Categories  
Analytics

Support  
Support  
Help

Legal  
License Agreement  
Privacy Policy

**Overview**   [Ratings + reviews](#)

Create a fun visual from frequent text in your data

Word Cloud is a visual representation of word frequency and value. Use it to get instant insight into the most important terms in your data. With the interactive experience of Word Cloud in Power BI, you no longer have to tediously dig through large volumes of text to find out which terms are prominent or prevalent. You can simply visualize them as Word Cloud and get the big picture instantly and user Power BI's interactivity to slice and dice further to uncover the themes behind the text content. This visual also puts you in control on the appearance of the word cloud, be it the size or usage of space and how to treat the data. You can choose to break the words in the text to look for the frequency word or keep word break off to project a measure as a value of the text. You can also enable stop words to remove the common terms from the word cloud to avoid the clutter. By enabling rotation and playing with the angles allowed, you can become very creative with this visual. Optionally you can also use a measure to provide weightage to the text. If none provided, it will simply use the frequency. Check out the formatting pane for more options. This is an open source visual. Get the code from GitHub: <https://github.com/Microsoft/PowerBI-visuals-wordcloud>

Visual capabilities

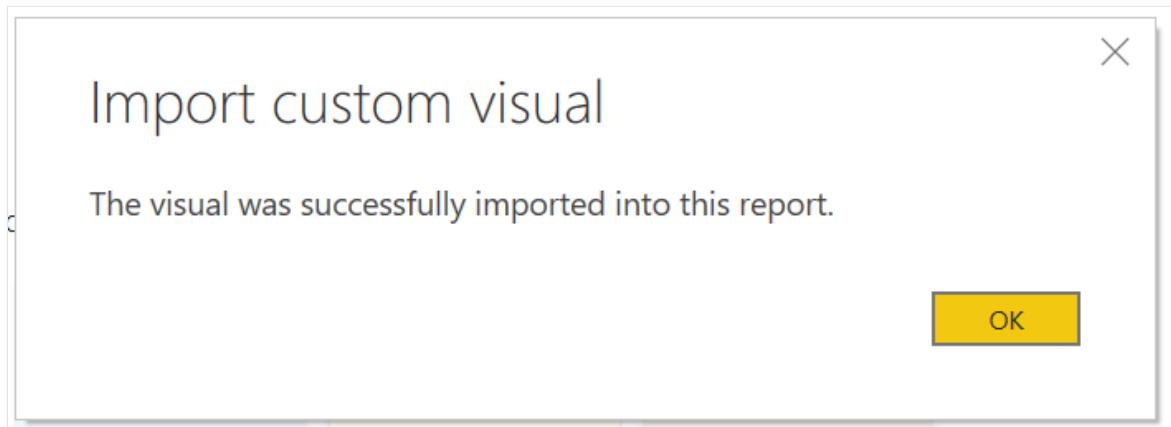
This visual is certified by Power BI  
[Learn more about certified Power BI visuals.](#)



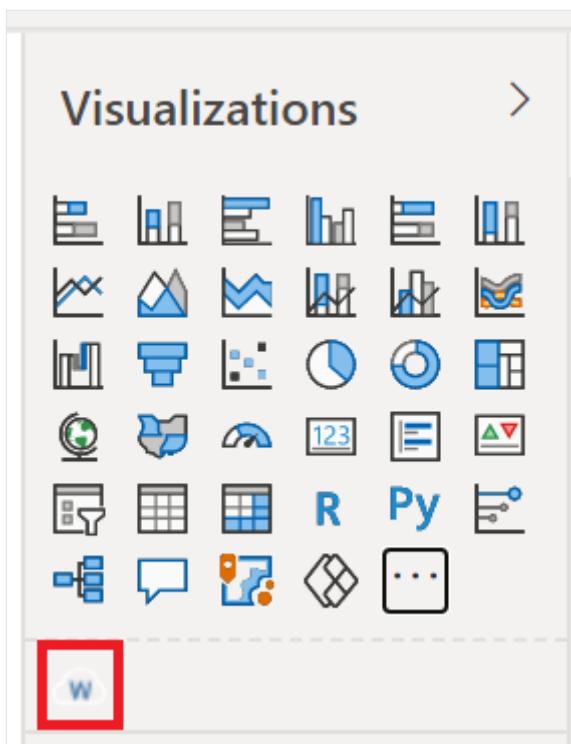
## 💡 Tip

Select **Download Sample** to download a sample Power BI semantic model and report created by the visual publisher. The sample report demonstrates what the visual looks like and how it can be used. It can also include useful comments, tips, and tricks from the publisher.

6. When the visual is successfully imported, select **OK**.



7. The visual now appears as a new icon on the bottom of the visualizations pane of the current report. Select the new visual icon to use this visual in the report.

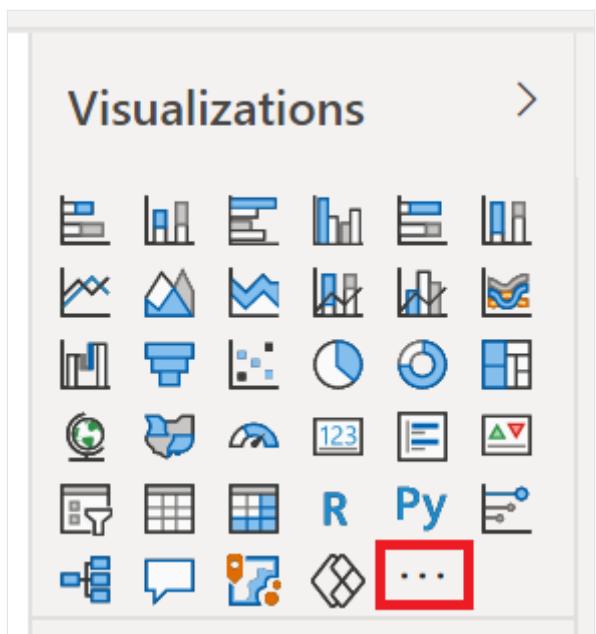


8. If you want the visual to remain on the **Visualizations** pane so you can use it in future reports, right-click the visual's icon and select **Pin to visualization pane**.

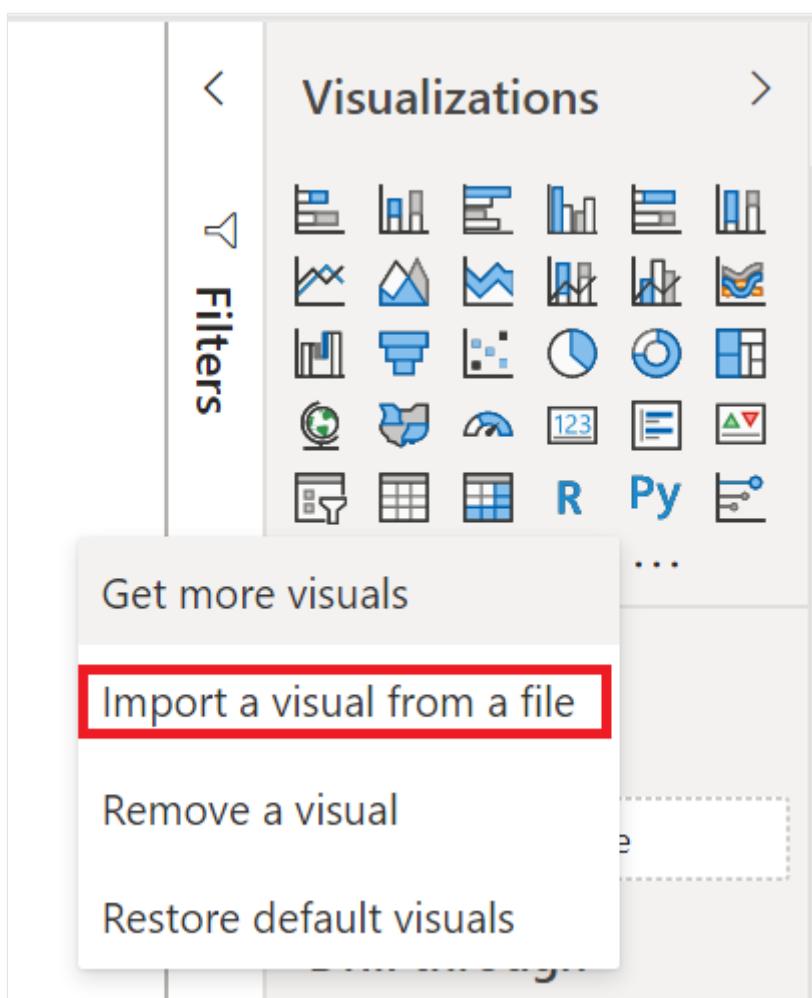
## Import a visual file from your local computer into Power BI

Power BI visuals are packaged as *.pbiviz* files that can be stored on your computer. You can share these files with other Power BI users. You can download visual files from AppSource onto your computer, but you can also get custom visuals from a trusted friend or colleague. Custom visuals that come from sources other than official Microsoft sources should be imported only if you trust the source.

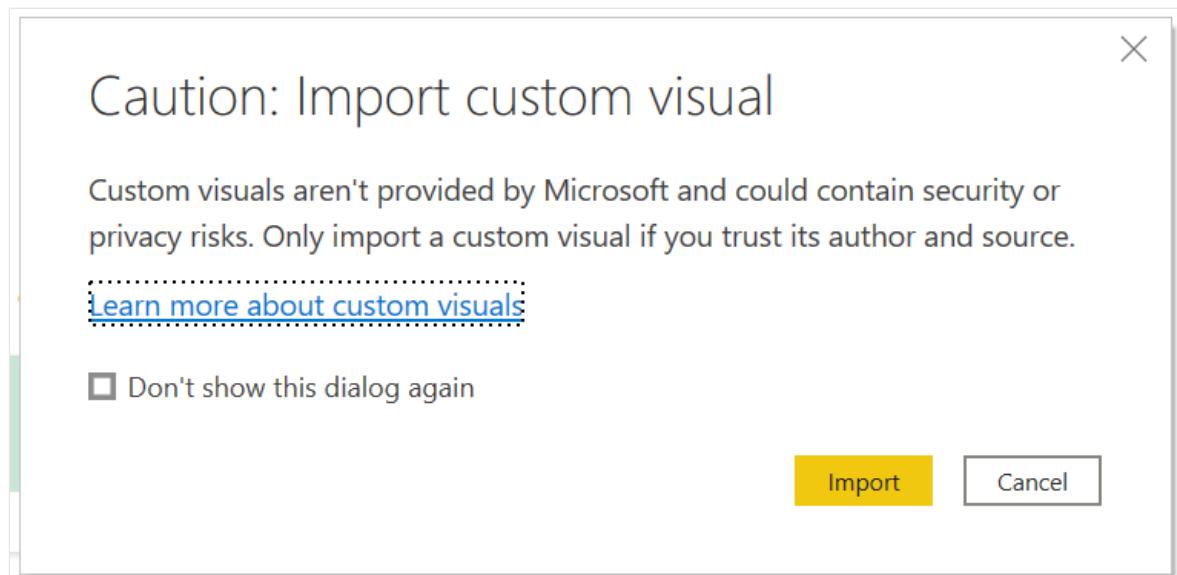
1. Open your report in [Power BI Desktop](#) or [Power BI service](#).
2. Enable [developer mode](#) for Power BI Desktop. This setting stays enabled for the current session only and must be repeated every time you import a visual from a file.
3. Select the ellipsis from the visualizations pane.



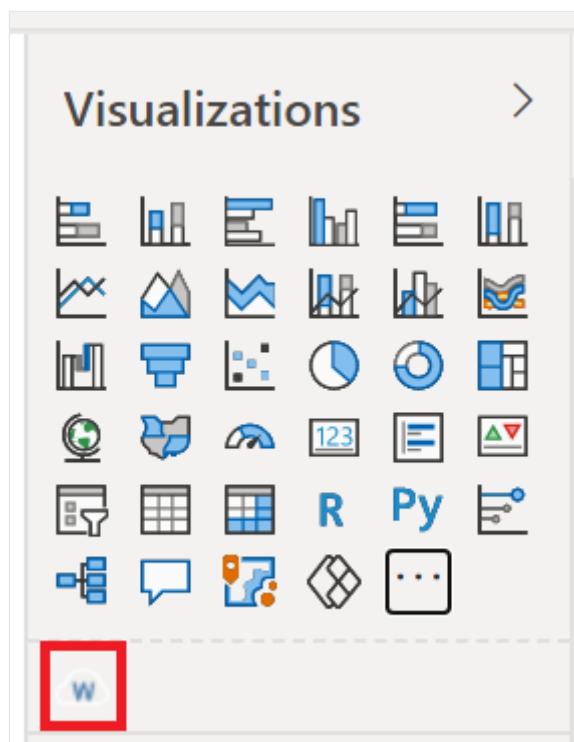
4. Select Import a visual from a file from the menu.



5. If you get a message cautioning you about importing custom files, select Import if you trust the source of the file.



6. Navigate to the folder that has the custom visual file (.pbviz) and open it.
7. When the visual is successfully imported, select **OK**.
8. The visual now appears as a new icon on the bottom of the visualizations pane of the current report. Select the new visual icon to use this visual in the report.



9. If you want the visual to remain on the **Visualizations** pane so you can use it in future reports, right-click the visual's icon and select **Pin to visualization pane**.

## Related content

- [Develop a Power BI circle card visual](#)
- [Visualizations in Power BI](#)

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

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

# Power BI custom visuals FAQ

- If you have other questions, [try asking the Power BI Community](#).
- Still have an issue? See the [Power BI support page](#).

## Organizational visuals

### How can the admin manage access to organizational Power BI custom 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 using "Add visual > from AppSource", is it automatically updated when a vendor updates the visual in the public marketplace?

Yes, the visual is automatically updated from the public marketplace. If the visual is certified, the certification is retained, including extra features such as export to PDF or PowerPoint.

### 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 custom 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 can't 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 aren't 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 visuals checked again with every new submission (upgrade)?

Yes. Every time a new version of a 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 the update is rejected because of a violation, an email is sent to the developer explaining what needs to be fixed.

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

No. A certified visual can't lose its certification with a new update. Instead, 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 aren't certified.

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

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

Getting an update of a Power BI visual certified can take up to three weeks.

## Does the certification process ensure that there's no data leakage?

The tests performed are designed to check that third-party visuals don't 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 don't necessarily mean unsafe visuals.

Some visuals aren't 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) adds-in. These adds-in 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 custom 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's 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, after the AppSource team has approved your IAP Power BI visual, 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.

## More questions

# How to get support?

Contact the Power BI visuals support team [pbicvsupport@microsoft.com](mailto:pbicvsupport@microsoft.com) with any questions, comments, or issues you have. This support channel is for custom visuals developers in the process of developing their own visuals.

For customer experience issues when using custom Power BI visuals, submit a case request via the [Power Platform admin center portal](#).

## Related content

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

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

# Examples of Power BI visuals

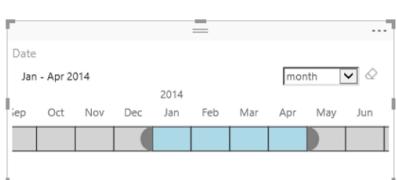
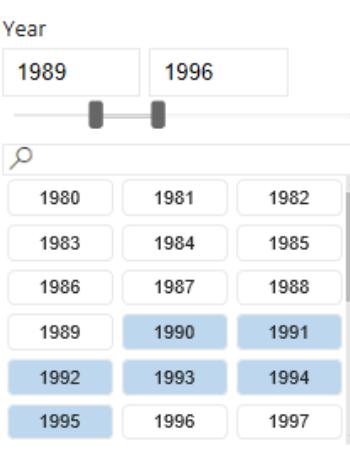
Article • 10/13/2024

This article describes some of the Power BI visuals you can download, use, and modify from GitHub. These sample visuals 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.

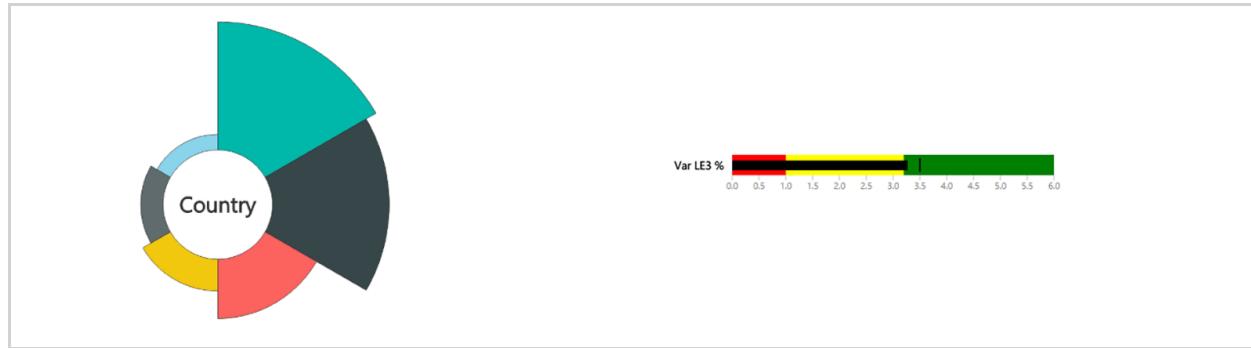
[Expand table](#)

	
<p><a href="#">Chiclet Slicer</a></p> <p>Display image or text buttons that act as an in-canvas filter on other visuals</p>	<p><a href="#">Timeline slicer</a></p> <p>Graphical date range selector that filters by date</p>
	
<p><a href="#">Slicer sample</a></p> <p>Demonstrates the use of the Advanced Filtering API</p>	

## Charts

Get inspired with our gallery of Power BI visuals, including bar charts, pie charts, Word Cloud, and others.

[Expand table](#)

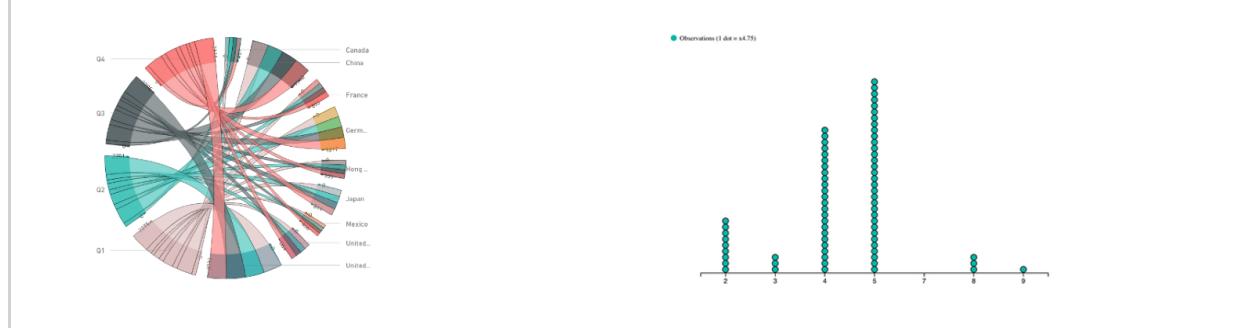


#### Aster Plot

A twist on a standard donut chart that uses a second value to drive sweep angle

#### Bullet chart

A bar chart with extra visual elements that provide context useful for tracking metrics

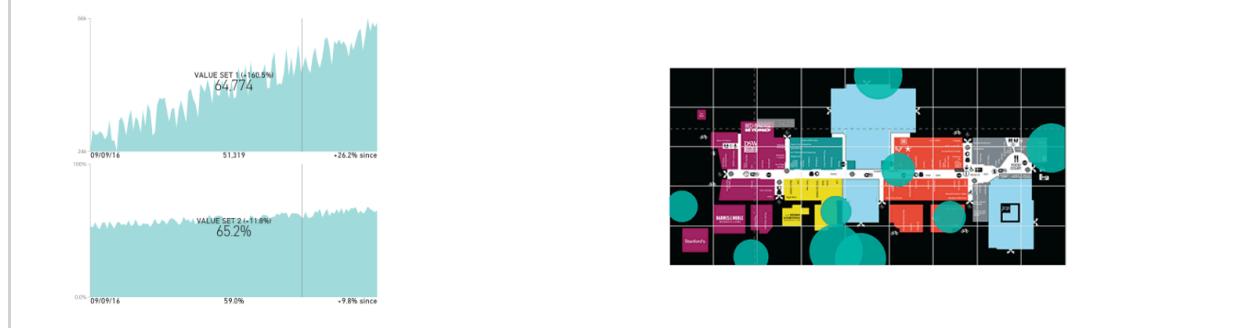


#### Chord

A graphical method that displays the relationships between data in a matrix

#### Dot plot

Shows the distribution of frequencies in a great looking way

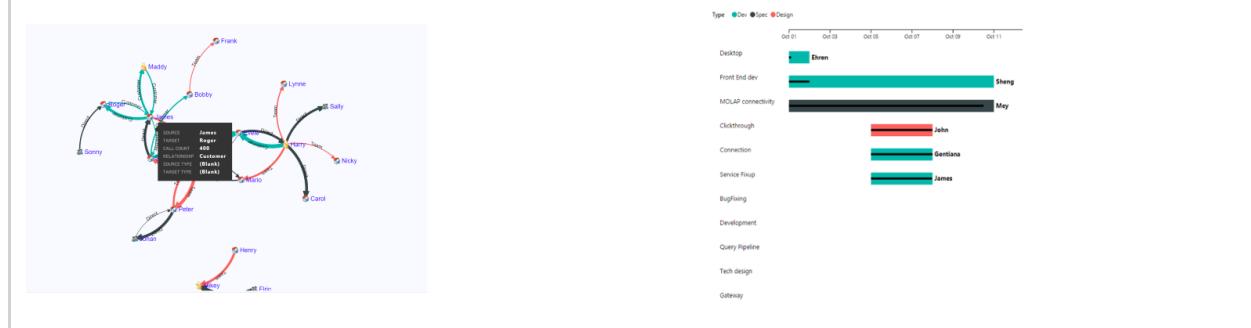


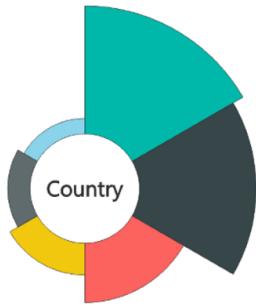
#### Dual KPI

Efficiently visualizes two measures over time, showing their trend on a joint timeline

#### Enhanced Scatter

Improvements on the existing scatter chart





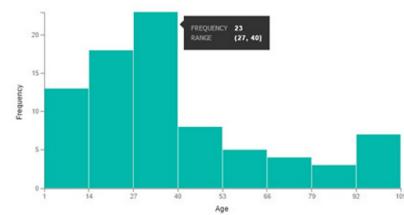
## Force Graph ↗

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

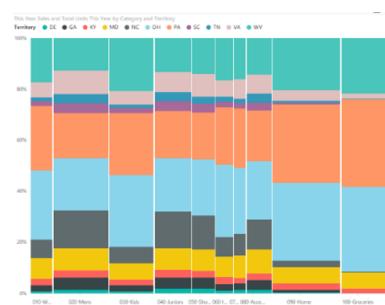
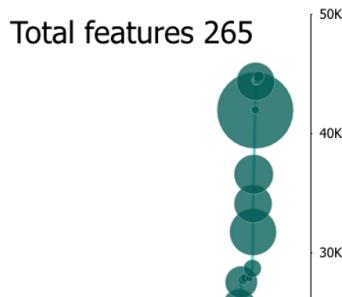


## Table Heatmap ↗

Compare data easily and intuitively using colors in a table

## Histogram chart ↗

Visualizes the distribution of data over a continuous interval or certain time period

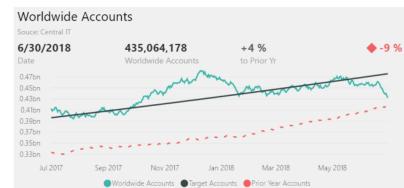
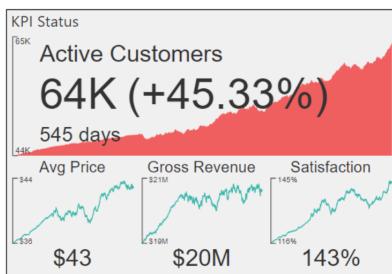


## LineDot chart ↗

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

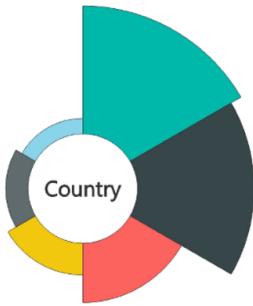


## Multi KPI ↗

A powerful Multi KPI visualization with a key KPI along with multiple sparklines of supporting data

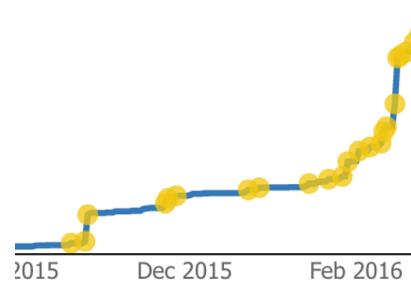
## Power KPI ↗

A powerful KPI Indicator with multi-line chart and labels for current date, value, and variances



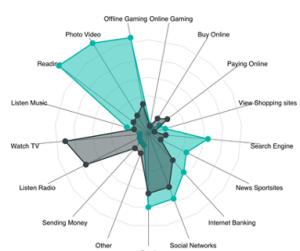
Column-Based Data Model with Metrics as Rows

Category	Sub-Category	Metric Name	Current Value	KPI Status	Last 18 Months
Non-Financial	User	Active Customers	16,191	↑ +4%	
	Transactions	Total Transactions	431,901	↑ +4%	
	Satisfaction	Satisfaction	50.5%	↑ +2%	
	Volume	Total Units	1,124K	↑ +2%	
Financial	Units	Premium Units	30,740	↓ -2%	
	Top Line	Gross Revenue	\$7.97M	↓ -1%	
		Avg Price	\$12.55	↓ -1%	
		Returns	1,572	↑ +7%	



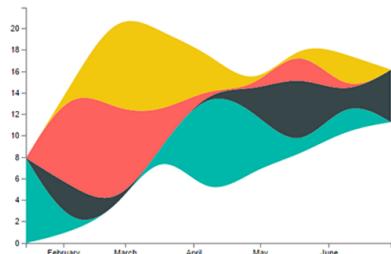
## Power KPI Matrix ↗

Monitor balanced scorecards and unlimited number of metrics and KPIs in a compact, easy to read list



## Radar chart ↗

Presents multiple measures plotted over a categorical axis, which is useful to compare attributes

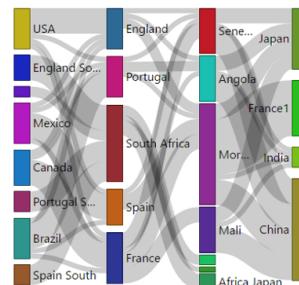


## Stream graph ↗

A stacked area chart with smooth interpolation, which is often used to display values over time

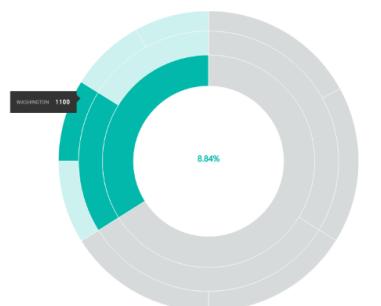
## Pulse chart ↗

This line chart annotated with key events is perfect for telling stories with data



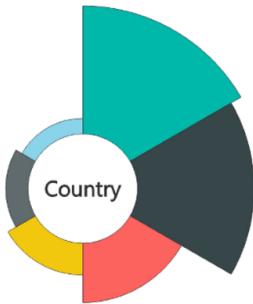
## Sankey chart ↗

Flow diagram where the width of the series is proportional to the quantity of the flow



## Sunburst chart ↗

Multilevel donut chart for visualizing hierarchical data



	Flat	House
Bangkok	0.12M	0.41M
Barcelona	0.16M	0.57M
Beijing	0.25M	1.04M
Berlin	0.22M	0.71M
Cairo	0.02M	0.06M
Cape Town	0.06M	0.22M
Casablanca	0.02M	0.07M
Delhi	0.05M	0.19M
Dubrovnik	0.13M	0.41M
Hong Kong	0.13M	0.41M
Istanbul	0.05M	0.16M
Jakarta	0.02M	0.09M



### Tornado chart ↗

Compare the relative importance of variables between two groups

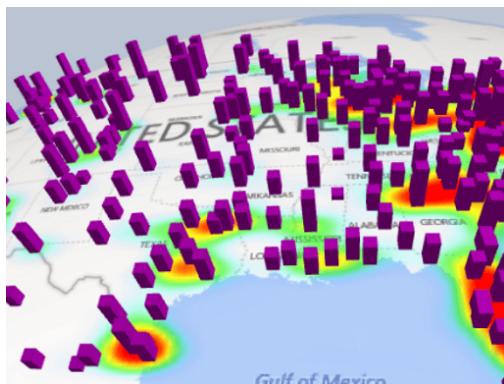
### Word Cloud ↗

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.

[+] Expand table

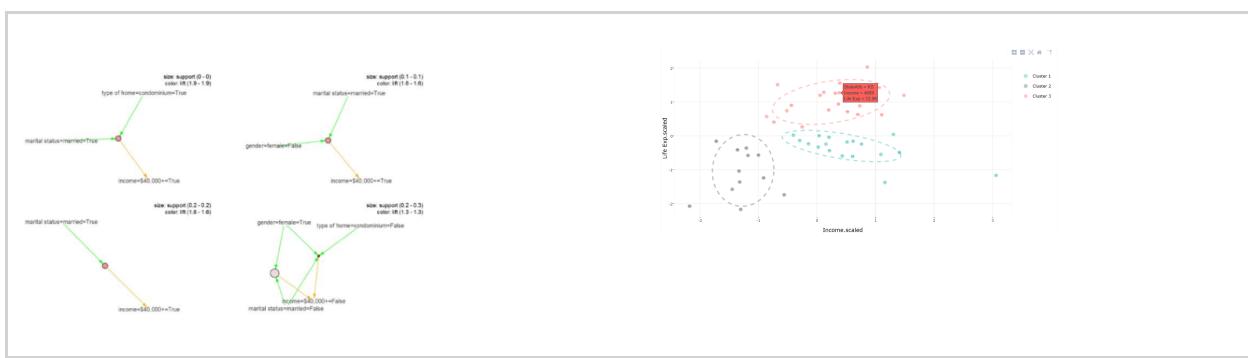


### Globe Map ↗

Plot locations on an interactive 3D map

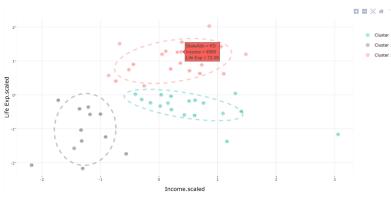
## R visuals

These examples demonstrate how to harness the analytic and visual power of R visuals and R scripts.



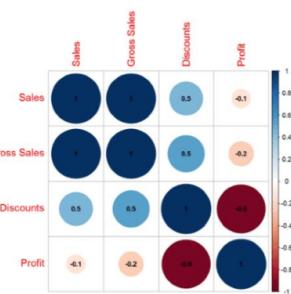
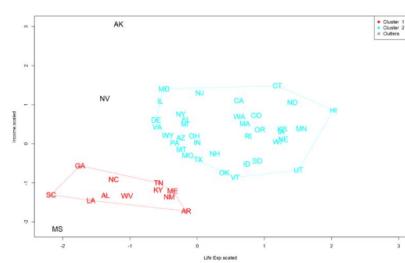
## Association rules ↗

Uncover relationships between seemingly unrelated data using if-then statements



## Clustering ↗

Find similarity groups in your data using k-means algorithm

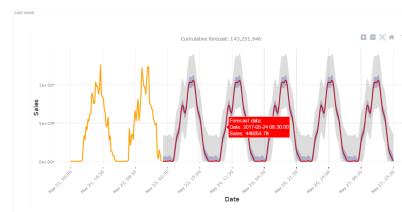
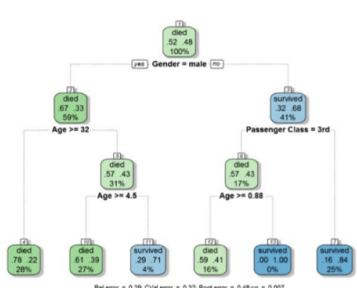


## Clustering with outliers ↗

Find similarity groups and outliers in your data

## Correlation plot ↗

Highlight the most correlated variables in a data table

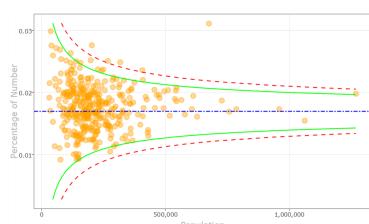


## Decision tree chart ↗

Schematic tree-shaped diagram for determining statistical probability using recursive partitioning

## Forecasting TBATS ↗

Time-series forecasting for series that have multiple seasonalities using the TBATS model



## Forecasting with ARIMA ↗

Predict future values based on historical data using Autoregressive Integrated Moving Avg (ARIMA)

## Funnel plot ↗

Find outliers in your data using a funnel plot

A decision tree diagram illustrating splits based on marital status (Married/Not Married), gender (Female/Male), income (<=40,000/>>40,000), and support (0-0.5/0.1-0.5/0.5-1). The tree has multiple branches and nodes, with specific conditions like 'marital status=married=True' and 'income>40,000>True'.

A scatter plot showing Life expectancy on the Y-axis versus Income scaled on the X-axis. Three distinct clusters of outliers are highlighted: Outlier 1 (blue), Outlier 2 (green), and Outlier 3 (red).

A scatter plot titled "Outliers detection based on LOF (Local Outline Factor)". It shows isolated points (red) and clusters of points (blue) on a 2D coordinate system. A sidebar provides settings for outlier detection, including outlier type (Point and Cluster), algorithm (LOF), threshold (0.3), neighbors (5), scale of app., and reverse to reflect.

A spline chart showing Average Depth on the Y-axis against Date Trade on the X-axis. A red fitted curve represents the trend, and a shaded area indicates the confidence interval. A legend entry for "Spline Fitted" is shown.

**Outliers detection** ↗

Find outliers in your data using the most appropriate method and plot

**Spline chart** ↗

Visualize and understand noisy data

Time Series Decomposition

A time series decomposition chart showing the original data (blue line), trend (red line), seasonal component (yellow line), and residual component (orange line) from 2007 to 2013. The model is additive, with a frequency of 12 per year, seasonal 17%, trend 51%, and remainder 32%.

A time series forecasting chart showing Actual Data (blue line), Forecast (orange line), and Error (grey shaded area) for Calendar Month. The chart includes a legend entry for "Forecast from ETS(A,A,A)".

**Time series decomposition chart** ↗

Understand the time series components using "Seasonal and Trend decomposition using Loess"

**Time series forecasting chart** ↗

Using exponential smoothing model to predict future values based on previously observed values

## Related content

- Import a Power BI visual
- Develop your own Power BI custom visual

## Feedback

Was this page helpful?

Yes

No

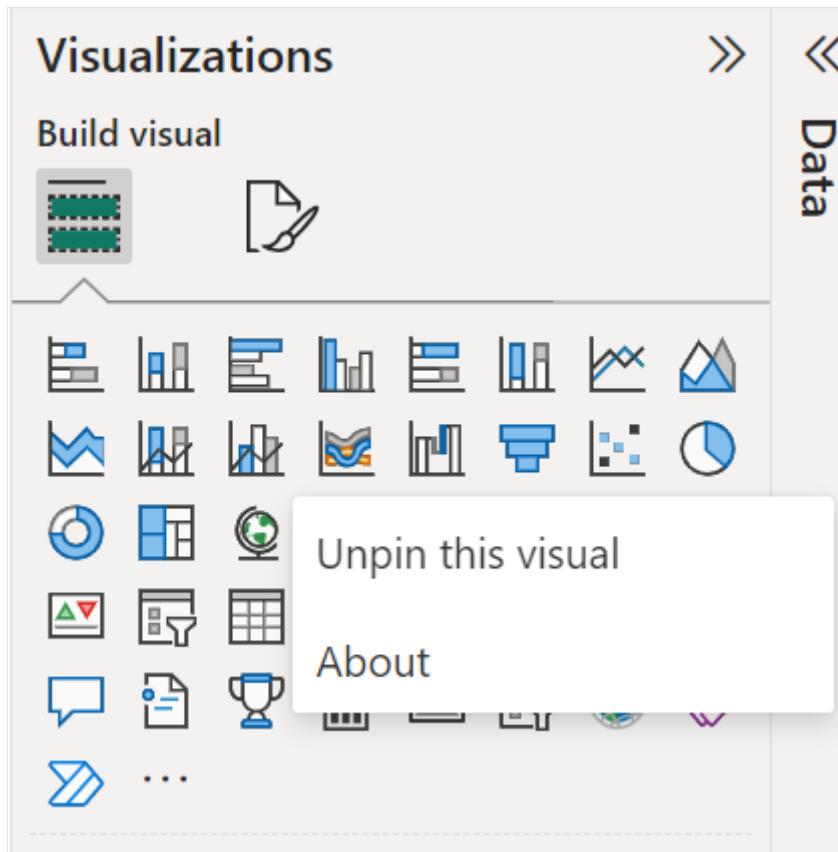
Provide product feedback ↗ | Ask the community ↗

# The Power BI visuals *About* dialog

07/15/2024

Every visual has an *About* dialog that gives you information about the visual, including the version number, who created it, and what permission it requests.

To open the *About* dialog, right-click the visual and select **About**.



The *About* dialog for custom visuals has three tabs that show the following information:

- [About](#)
- [Certification](#)
- [Privileges](#)

## About

The *About* tab provides the following information:

- **Name:** The name of the visual.
- **Publisher:** The name of the publisher of the visual.
- **ID:** The ID of the visual.
- **Version:** The version of the visual.
- **Source:** The source of the visual, if available.
- **Support:** Link to support information, if available.

# Power BI Visual



About

Certification

Privileges

Name: Tachometer

Publisher: Indika Chamara Ranasinghe

Id: Tachometer1474636471549

Version: 3.0.1

Source: [AppSource](#)

Support: <https://www.annik.com/pbisupport/>

Close

## Certification

This section tells you if a visual is certified by Microsoft. Certification means that the visual meets certain specified code requirements and testing by the Power BI team. For more information about certified visuals, see [Certified Power BI visuals](#).

## Privileges

Power BI visuals sometimes require access to data or other resources in your organization. This section describes what permissions a visual might need in order to work properly in the current version. It's important to note that the admin can choose to block a visual from requesting a certain permission through the [Power BI visuals admin settings](#). If a visual requests a permission that's disabled, the visual can't use that permission.

## Power BI Visual

[About](#)[Certification](#)[Privileges](#)

The current version of this visual requests privileges that may be disabled in your organization by your admin.

[Learn more about privileges](#)

This visual would like to:

- Call Power BI local storage APIs
- Access URLs
- Call Power BI Download APIs
- Make calls on behalf of signed-in users

[Close](#)

The following are the permissions a visual might request:

- [Call Power BI local storage APIs](#)
- [Access URLs](#)
- [Call Power BI Download APIs](#)
- [Make calls on behalf of signed in users](#)

Visuals can request permission for any or the following privileges:

## Call Power BI local storage APIs

The visual might want to [store data locally](#) on the user's device to help improve performance. The admin can [choose to block this permission](#) in which case the visual can't store data locally.

## Access URLs

This privilege allows a visual to send HTTP requests to specific URLs that the visual builder declares in advance.

There's no tenant admin switch for this permission. As long as [noncertified visuals are allowed](#), this privilege is granted.

## Call Power BI Download APIs

The [file download API](#) lets users download data from a custom visual into a file on their storage device. Downloading data from a visual requires user consent and admin permission provided in the [Allow downloads from custom visuals tenant switch](#). If the admin doesn't enable this switch, the visual can't download data from Power BI.

If the visual has permission to call the download API, it can export files of the following types:

- .txt
- .csv
- .json
- .tmplt
- .xml
- .pdf
- .xlsx

## Make calls on behalf of signed in users

The visual might need to obtain a Microsoft Entra ID (formerly known as Azure AD) access tokens for signed-in users, to facilitate single sign-on authentication. Permission to access tokens requires user consent and admin permission provided in the [Obtain Microsoft Entra access token tenant switch](#). If the admin doesn't enable this switch, the visual can't access a token or make API calls on behalf of the user.

## Related content

- [About tenant settings](#)
- [Manage Power BI visuals admin settings](#).

# Set up your environment for developing a Power BI visual

Article • 11/25/2024

This article teaches you how to set up your environment for developing a Power BI visual.

Before you start development, you need to install `node.js` and the `pbviz` package. Then, when your local environment is set up, you need to configure Power BI service for developing a Power BI visual.

In this article, you learn how to:

- ✓ [Install `node.js`](#).
- ✓ [Install `pbviz`](#).
- ✓ [Enable Power BI developer mode](#).

## Prerequisites

Before you start developing your Power BI visual, verify that you have everything listed in this section.

- A Power BI Pro or Premium Per User (PPU) account. If you don't have one, [sign up for a free trial](#).
- An integrated development environment (IDE) for developing JavaScript and TypeScript applications. [Visual Studio Code \(VS Code\)](#) is ideal for developing visuals.
- [Windows PowerShell](#) version 4 or later (for Windows). Or [Terminal](#) (for Mac).

## Install `node.js`

`Node.js` is a JavaScript runtime built on Chrome's V8 JavaScript engine. It allows developers to run any apps created on JavaScript.

To install `node.js`:

1. Navigate to [node.js](#), from your web browser.
2. Download the latest recommended MSI installer.

3. Run the installer, and then follow the installation steps. Accept the terms of the license agreement and all defaults.

4. Restart your computer.

## Install pbviz

The *pbviz* tool, which is written using JavaScript, compiles the visual source code of the *pbviz* package.

The *pbviz* package is a zipped Power BI visual project, with all the needed scripts and assets.

To install the latest version of *pbviz*, open Windows PowerShell and enter the following command.

```
PowerShell
```

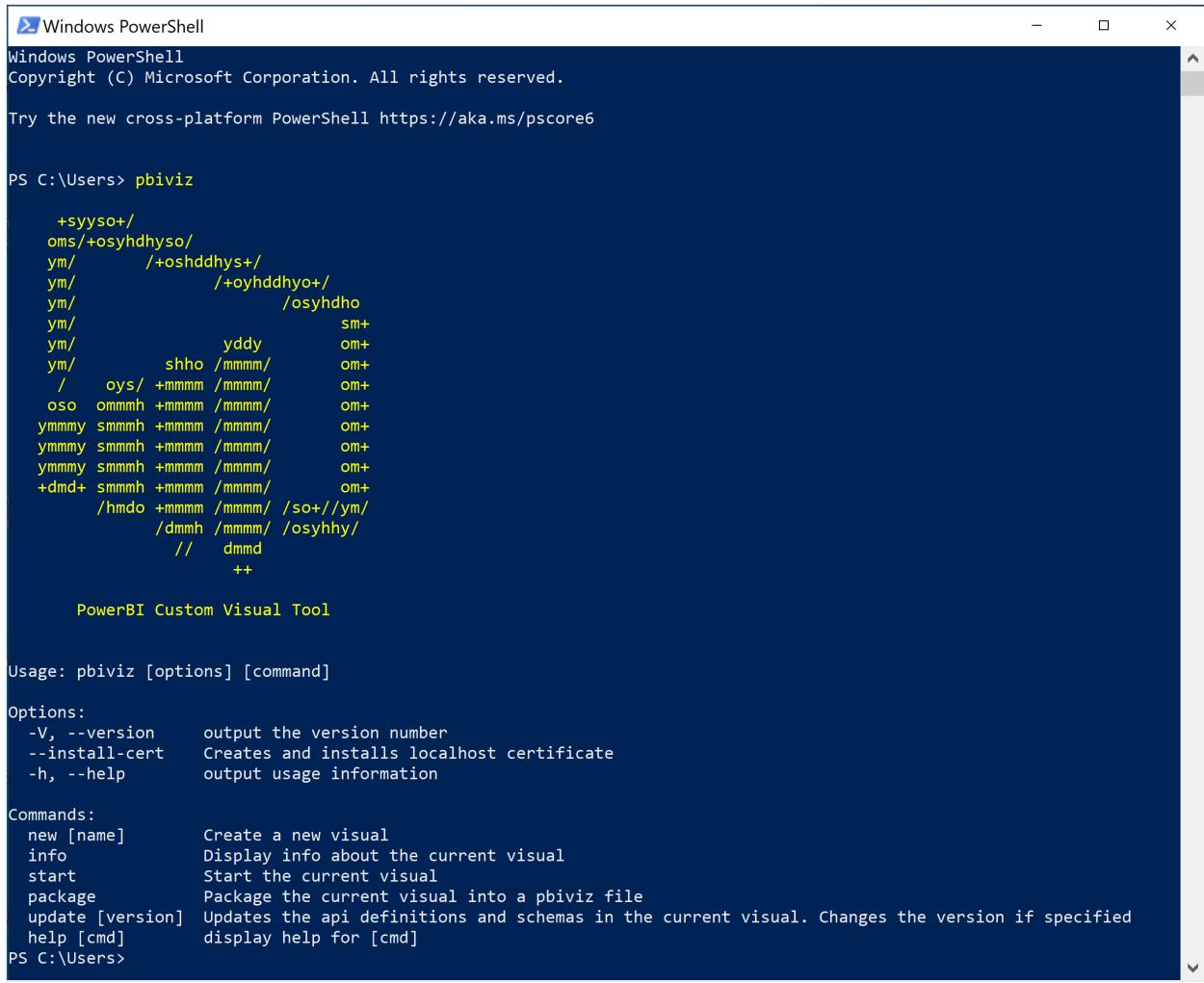
```
npm i -g powerbi-visuals-tools@latest
```

ⓘ Note

You might get some warnings when you run this command. They should not prevent *pbviz* from installing.

## (Optional) Verify that your environment is set up

Confirm that the Power BI visuals tools package is installed. In PowerShell, run the command `pbviz` and review the output, including the list of supported commands.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users> pbviz

+syso+/
oms/+osyhdhyso/
ym/      /+oshddhys+
ym/      /+oyhddhyo+
ym/      /osyhdho
ym/          sm+
ym/      yddy    om+
ym/      shho /mmmm/ om+
/   oys/ +mmmm /mmmm/ om+
oso ommh +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/
// dmmd
++
```

PowerBI Custom Visual Tool

Usage: pbviz [options] [command]

Options:

- V, --version output the version number
- install-cert Creates and installs localhost certificate
- h, --help output usage information

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]

PS C:\Users>

## Enable developer mode

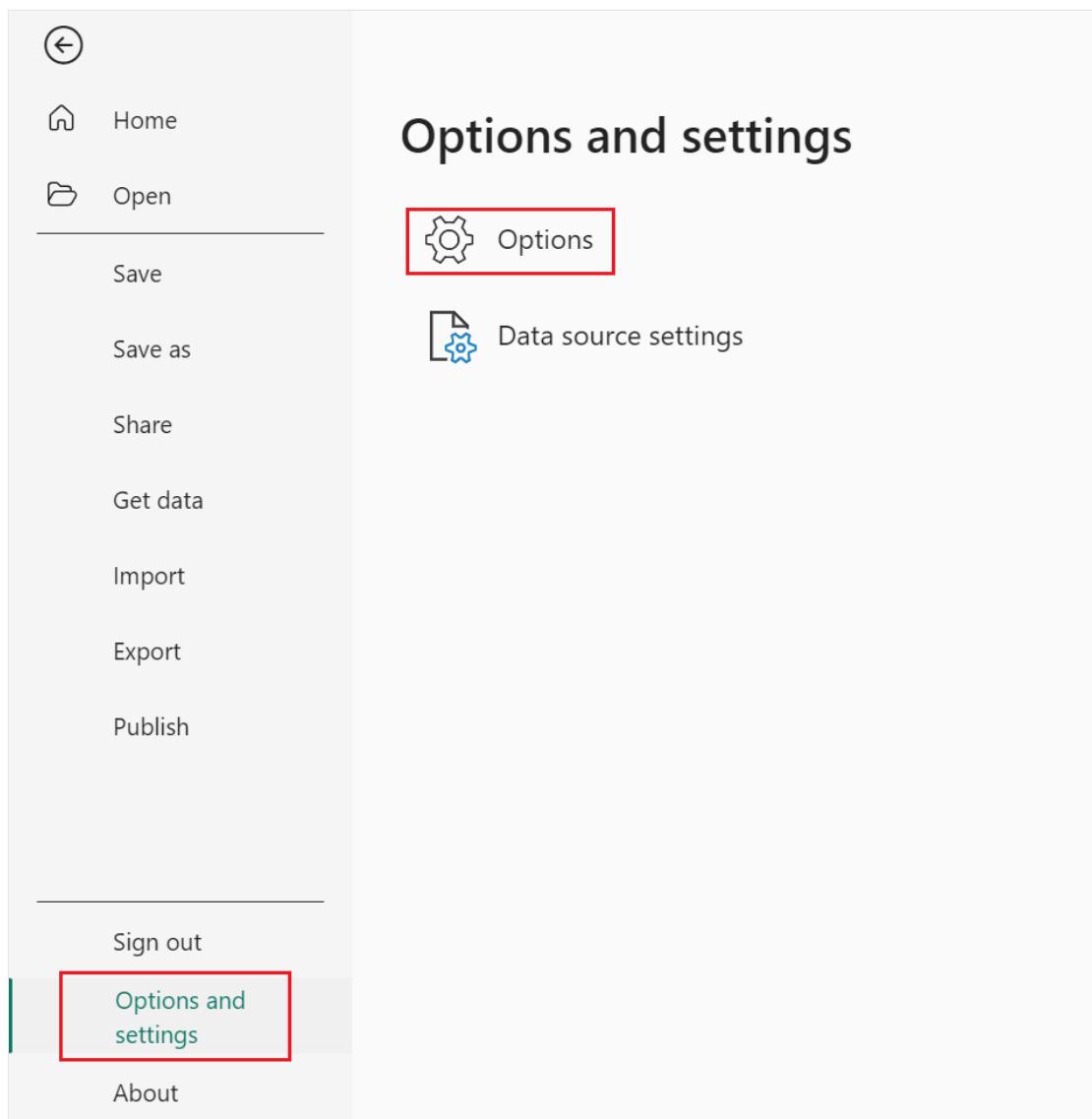
To develop or upload your own Power BI visual in the Desktop or on the web, developer mode must be enabled.

How to enable developer mode in Power BI Desktop

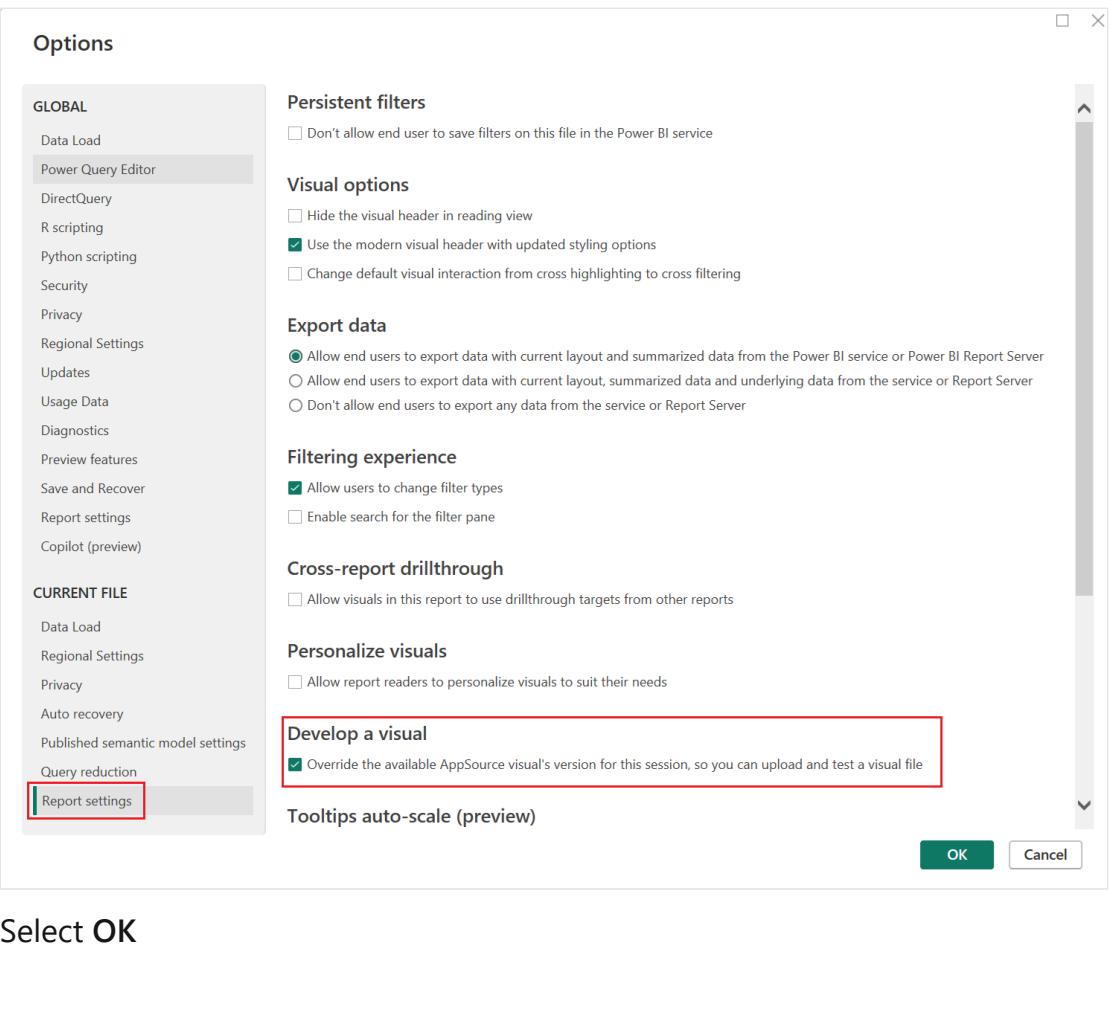
To develop a Power BI visual in the Desktop, enable the *Develop a visual* setting.

This setting only stays enabled for the current session. You must enable this setting in each session where you import a visual from a file.

1. From the Power BI desktop, navigate to **File > Options and settings > Options**



2. Select **Report settings** in the **Current file** section, and enable the *Develop a visual* check box.



3. Select OK

## Related content

- Learn about the Power BI visual project structure
- Create a Power BI circle card visual
- Create an R-powered Power BI visual

## Feedback

Was this page helpful?

Yes

No

Provide product feedback | Ask the community

# Power BI visual project structure

Article • 06/09/2024

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:

```
markdown

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. This icon must be a **PNG** file with dimensions *20 pixels by 20 pixels*.

## 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. Read about the [Visual API](#).
- `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. Certain metadata, like the author's name and email, are required before you can package the visual.

### ⓘ Note

- From version 3.x.x of the `pbviz` tool, `externalJS` isn't supported.
- Version numbers should contain four digits in the following format `x.x.x.x`.
- For localization support, [add the Power BI locale to your visual](#).

### JSON

```
{  
  "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": "1.0.0.0",

// 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": []
}

```

## Related content

- [Power BI visuals system integration](#)

- Develop a Power BI circle card visual
- 

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Power BI visuals system integration

Article • 01/04/2024

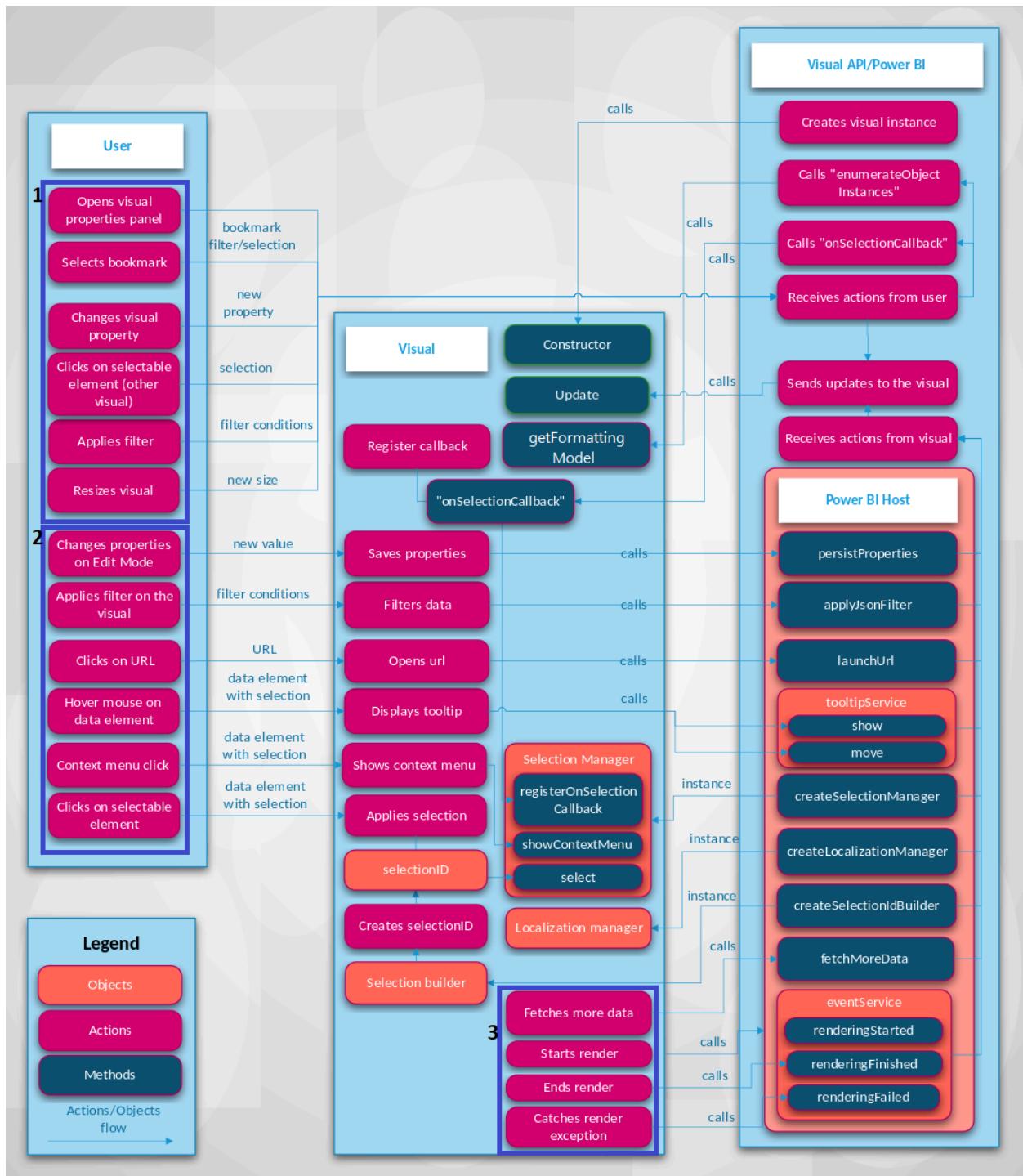
The article describes the [Visual API](#), and how Power BI handles the interactions between you, the visual, and the host.

Actions and subsequent updates in Power BI can be initiated manually or automatically.

Update types:

- [Interact with a visual through Power BI](#).
- [Interact with the visual directly](#).
- [Visual interact with Power BI](#).

The following figure shows how common visual-based actions, like selecting a bookmark, process in Power BI.



## Interact with a visual through Power BI

You can update a visual with Power BI as follows:

- Open the visual's properties panel.

When you open 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 `getFormattingModel` method of the visual (APIs earlier than version 5.0 call `enumerateObjectInstances` instead). The API returns modern format pane model components, properties, and their actual values.

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

- [Customize visualization titles, backgrounds, labels, and legends](#).

When you change the value of a property in the Format panel, Power BI calls the `update` method. Power BI passes in the new `options` object to the `update` method, and the objects contain the new values.

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

- Resize the visual.

When you change the size of a visual, Power BI calls the `update` method and passes in the new `options` object. The `options` objects have nested `viewport` objects that contain the new width and height of the visual.

- Apply 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](#).

- Select a data point in another visual in the report.

When you select 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](#).

- Select a bookmark in the **Bookmarks** panel of the report.

When you select a bookmark in the **Bookmarks** panel, either:

- 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 both cases, the visual changes 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](#).

## Interact with the visual directly

You can also interact directly with the visual to update it:

- Hover over a data element.

A visual can display more information about a data point through the Power BI Tooltips API. When you hover 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 [Add tooltips to your Power BI visuals](#).

- Change visual properties (For example, by expanding a tree) and the visual saves the new state in the visual properties.

A visual can save properties values through the Power BI API. For example, when you interact with the visual and the visual needs to save or update properties values, the visual can call the `persistProperties` method.

- Select a URL.

By default, a visual can't open a URL directly. 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](#).

- Apply 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](#).

- Select elements in the visual.

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

# Visual interacts with Power BI

Sometimes the visual initiates communication with the Power BI host without any input from you:

- 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 semantic model.

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 ["Rendering" events in Power BI visuals](#).

## Related content

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

- [Visual API](#)
- [Develop a Power BI circle card visual](#)
- [Publish Power BI visuals to Microsoft commercial marketplace](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Capabilities and properties of Power BI visuals

Article • 02/19/2024

Every visual has a *capabilities.json* file that is created automatically when you run the `pbviz new <visual project name>` command to [create a new visual](#). The *capabilities.json* file describes the visual to the host.

The *capabilities.json* file tells the host what kind of data the visual accepts, what customizable attributes to put on the properties pane, and other information needed to create the visual. Starting from API v4.6.0, all properties on the capabilities model are *optional* except `privileges`, which are *required*.

The *capabilities.json* file lists the root objects in the following format:

```
JSON

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

When you create a new visual, the default *capabilities.json* file includes the following root objects:

- `privileges`
- `dataRoles`
- `dataViewMappings`
- `objects`

The above objects are the ones needed for data-binding. They can be edited as necessary for your visual.

The following additional root objects are optional and can be added as needed:

- `tooltips`
- `supportsHighlight`
- `sorting`

- `drilldown`
- `expandCollapse`
- `supportsKeyboardFocus`
- `supportsSynchronizingFilterState`
- `advancedEditModeSupport`
- `supportsLandingPage`
- `supportsEmptyDataView`
- `supportsMultiVisualSelection`
- `subtotals`
- `keepAllMetadataColumns`
- `migration`

You can find all these objects and their parameters in the [\*capabilities.json\* schema](#)

## privileges: define the special permissions that your visual requires

Privileges are special operations your visual requires access to in order to operate.

Privileges take an array of `privilege` objects, which defines all privilege properties. The following sections describe the privileges that are available in Power BI.

### Note

From API v4.6.0, privileges **must** be specified in the *capabilities.json* file. In earlier versions, remote access is automatically granted and downloading to files isn't possible. To find out which version you're using, check the `apiVersion` in the *pbviz.json* file.

## Define privileges

A JSON privilege definition contains these components:

- `name` - (string) The name of the privilege.
- `essential` - (Boolean) Indicates whether the visual functionality requires this privilege. A value of `true` means the privilege is required; `false` means the privilege isn't mandatory.
- `parameters` - (string array)(optional) Arguments. If `parameters` is missing, it's considered an empty array.

The following are types of privileges that must be defined:

- Access External resources
- Download to file
- Local storage privileges

### ⓘ Note

Even with these privileges granted in the visual, the admin has to enable the switch in the admin settings to allow people in their organization to benefit from these settings.

## Allow web access

To allow a visual to access an external resource or web site, add that information as a *privilege* in the capabilities section. The privilege definition includes an optional list of URLs the visual is allowed to access in the format `http://xyz.com` or `https://xyz.com`. Each URL can also include a wildcard to specify subdomains.

The following is an example of privileges setting allowing access to external resources:

JSON

```
{  
  "name": "WebAccess",  
  "essential": true,  
  "parameters": [ "https://*.microsoft.com", "http://example.com" ]  
}
```

The preceding `WebAccess` privilege means that the visual needs to access any subdomain of the `microsoft.com` domain via HTTPS protocol only and `example.com` without subdomains via HTTP, and that this access privilege is essential for the visual to work.

## Download to file

To allow the user to export data from a visual into a file, set `ExportContent` to `true`.

This `ExportContent` setting enables the visual to export data to files in the following formats:

- `.txt`
- `.csv`
- `json`

- .tmplt
- .xml
- .pdf
- .xlsx

This setting is separate from and not affected by download restrictions applied in the organization's [export and sharing](#) tenant settings.

The following is an example of a privileges setting that allows downloading to a file:

JSON

```
"privileges": [  
    {  
        "name": "ExportContent",  
        "essential": true  
    }  
]
```

## Local storage privileges

This privilege allows a custom visual to store information on the user's local browser.

The following is an example of a privileges setting that allows use of the local storage:

JSON

```
"privileges": [  
    {  
        "name": "LocalStorage",  
        "essential": true  
    }  
]
```

## No privileges needed

If the visual doesn't require any special permissions, the `privileges` array should be empty:

JSON

```
"privileges": []
```

## Multiple privileges

The following example shows how to set several privileges for a custom visual.

JSON

```
"privileges": [
  {
    "name": "WebAccess",
    "essential": true,
    "parameters": [ "https://*.virtualearth.net" ]
  },
  {
    "name": "ExportContent",
    "essential": false
  }
]
```

## dataroles: define the data fields that your visual expects

To define fields that can be bound to data, you use `dataRoles`. `dataRoles` is an array of `DataViewRole` objects, which defines all the required properties. The `dataRoles` objects are the **fields** that appear on the [Properties pane](#).

The user drags data fields into them to bind data the data fields to the objects.

## DataRole properties

DataRoles are defined by the following properties:

- **name**: The internal name of this data field (must be unique).
- **displayName**: The name displayed to the user in the [Properties pane](#).
- **kind**: The kind of field:
  - `Grouping`: Set of discrete values that are used to group measure fields.
  - `Measure`: Single numeric values.
  - `GroupingOrMeasure`: Values that can be used as either a grouping or a measure.
- **description**: A short text 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 for requiredTypes and preferredTypes

- **bool**: A boolean value
- **integer**: An integer value
- **numeric**: A numeric value
- **text**: A text value
- **geography**: A geographic data

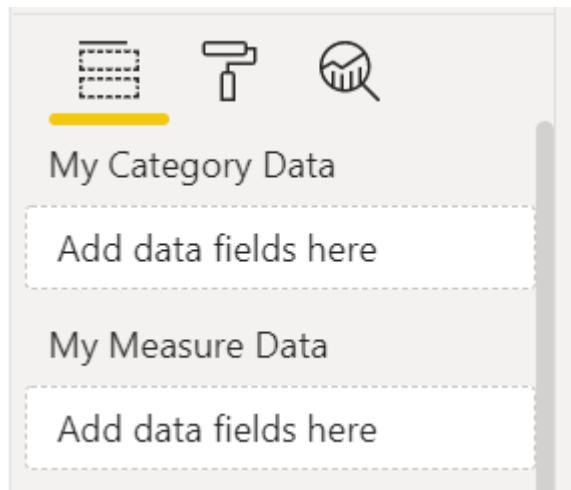
## dataRoles example

JSON

```
"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
      }
    ]
  }
]
```

```
...  
}
```

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



## dataViewMappings: how you want the data mapped

The `dataViewMappings` objects describe how the data roles relate to each other and allow you to specify conditional requirements for the displaying data views.

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

JSON

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

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

## objects: define property pane options

Objects describe customizable properties that are associated with the visual. The objects defined in this section are the objects that appear in the [Format pane](#). Each object can have multiple properties, and each property has a type that's associated with it.

JSON

```
"objects": {  
    "myCustomObject": {  
        "properties": { ... }  
    }  
}
```

For example, to support [dynamic format strings](#) in your custom visual, define the following object:

JSON

```
"objects": {  
    "general": {  
        "properties": {  
            "formatString": {  
                "type": {  
                    "formatting": {  
                        "formatString": true  
                    }  
                }  
            }  
        }  
    },  
},
```

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

## Related content

- [Understand data view mapping in Power BI visuals](#)
- [Objects and properties of Power BI visuals](#)
- [Sorting options for Power BI visuals](#)

# Understand data view mapping in Power BI visuals

Article • 12/12/2024

This article discusses data view mapping and describes how data roles are used to create different types of visuals. It explains how to specify conditional requirements for data roles and the different `dataMappings` types.

Each valid mapping produces a data view. You can provide multiple data mappings under certain conditions. The supported mapping options are:

- [conditions](#)
- [categorical](#)
- [single](#)
- [table](#)
- [matrix](#)

JSON

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

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

In other words, `categorical` might be defined in `dataViewMappings` but other mappings, such as `table` or `single`, might not be. In that case, Power BI produces a data view with a single `categorical` mapping, while `table` and other mappings remain undefined. For example:

JSON

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

```
  },
  "metadata": { ... }
}
]
```

## Conditions

The `conditions` section establishes rules for a particular data mapping. If the data matches one of the described sets of conditions, the visual accepts the data as valid.

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.

In the following example, the `category` is limited to one data field and the `measure` is limited to two data fields.

#### JSON

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

You can also set multiple conditions for a data role. In that case, the data is valid if any one of the conditions is met.

#### JSON

```
"conditions": [
  { "category": { "min": 1, "max": 1 }, "measure": { "min": 0, "max": 2 } },
  { "category": { "min": 2, "max": 2 }, "measure": { "min": 0, "max": 1 } }
]
```

In the previous example, one of the following two conditions is required:

- Exactly one category field and no more than two measures
- Exactly two categories and no more than one measure field

### ⓘ Note

Only one data role can have a minimum value of  $\geq 1$  per condition.

## Single data mapping

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

To use single data mapping, 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 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 to a single numeric value.

For example:

JSON

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

```
    ]  
}
```

The resulting data view can still contain other types of mapping, like table or categorical, but each mapping contains only the single value. The best practice is to access the value only in single mapping.

JSON

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

The following code sample processes simple data views mapping:

TypeScript

```
"use strict";  
import powerbi from "powerbi-visuals-api";  
import DataView = powerbi.DataView;  
import DataViewSingle = powerbi.DataViewSingle;  
// standard 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];
```

```

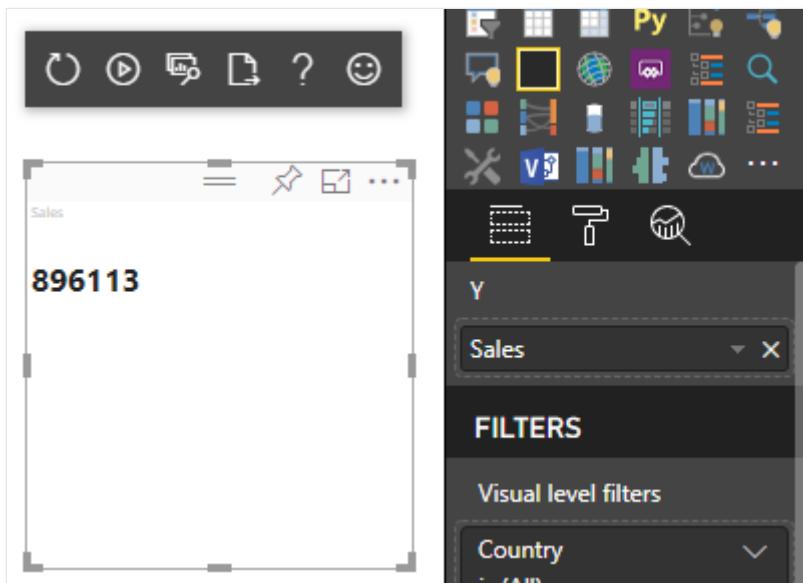
const single DataViewSingle = dataView.single;

if (!single DataView || !single DataView.value) {
    return
}

this.valueText.innerText = single DataView.value.toString();
}
}

```

The previous code sample results in the display of a single value from Power BI:



## Categorical data mapping

Categorical data mapping is used to get independent groupings or categories of data. The categories can also be grouped together by using "group by" in the data mapping.

### Basic categorical data mapping

Consider the following data roles and mappings:

```

JSON

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

```

```

        "kind": "Measure"
    }
],
"dataViewMappings": {
    "categorical": {
        "categories": {
            "for": { "in": "category" }
        },
        "values": {
            "select": [
                { "bind": { "to": "measure" } }
            ]
        }
    }
}
}

```

The previous example 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:** Includes *all* items in this data role in the data query.
- **bind...to:** Produces the same result as *for...in* but expects the data role to have a condition restricting it to a *single* field.

## Group categorical data

The next example uses the same two data roles as the previous example and adds two more data roles named `grouping` and `measure2`.

JSON

```

"dataRoles": [
    {
        "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": "Measure"
    }
]

```

```

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

```

The difference between this mapping and the basic mapping is how `categorical.values` is mapped. When you map the `measure` and `measure2` data roles to the data role `grouping`, the x-axis and y-axis can be scaled appropriately.

## Group hierarchical data

In the next example, the categorical data is used to create a hierarchy, which can be used to support [drill-down actions](#).

The following example shows the data roles and mappings:

JSON

```

"dataRoles": [
{
    "displayName": "Categories",
    "name": "category",
    "kind": "Grouping"
}
]

```

```

},
{
  "displayName": "Measures",
  "name": "measure",
  "kind": "Measure"
},
{
  "displayName": "Series",
  "name": "series",
  "kind": "Measure"
}
],
"dataViewMappings": [
{
  "categorical": {
    "categories": {
      "for": {
        "in": "category"
      }
    },
    "values": {
      "group": {
        "by": "series",
        "select": [
          {
            "for": {
              "in": "measure"
            }
          }
        ]
      }
    }
  }
}
]

```

Consider the following categorical data:

[\[ \] Expand table](#)

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

Power BI produces a categorical data view with the following set of categories.

JSON

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

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

For example, each `values` array represents one year. Also, each `values` array has four values: Canada, USA, UK, and Mexico.

JSON

```
{
  "values": [
    // Values for year 2013
    {
      "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 year 2014
    {
      "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 year 2015
  ]
}
```

```

{
    "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 year 2016
{
    "source": {...},
    "values": [
        null, // Value for `Canada` category
        350, // Value for `USA` category
        null, // Value for `UK` category
        null // Value for `Mexico` category
    ],
    "identity": [...],
}
]
}

```

The following code sample is for processing categorical data view mapping. This sample creates the hierarchical structure **Country/Region > Year > Value**.

TypeScript

```

"use strict";
import powerbi from "powerbi-visuals-api";
import DataView = powerbi.DataView;
import DataViewCategorical = powerbi.DataViewCategorical;
import DataViewValueColumnGroup = powerbi.DataViewValueColumnGroup;
import PrimitiveValue = powerbi.PrimitiveValue;
// standard 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
        // To support several columns of categories data bucket, iterate
categoricalDataView.categories array.
        const categoryFieldIndex = 0;
        // Measure has only one column in data buckets.
        // To support several columns on data bucket, 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-regions
        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);
    }
}

```

Here's the resulting visual:

The screenshot shows the Power BI Data View interface. 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 view with the following data:

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>

## Mapping tables

The *table* data view is essentially a list of data points where numeric data points can be aggregated.

For example, use the [same data in the previous section](#), but with the following capabilities:

JSON

```
"dataRoles": [
  {
    "displayName": "Column",
    "name": "column",
    "kind": "Grouping"
  },
  {
    "displayName": "Value",
    "name": "value",
    "kind": "Measure"
  }
]
```

```

        }
    ],
    "dataViewMappings": [
        {
            "table": {
                "rows": {
                    "select": [
                        {
                            "for": {
                                "in": "column"
                            }
                        },
                        {
                            "for": {
                                "in": "value"
                            }
                        }
                    ]
                }
            }
        }
    ]
}

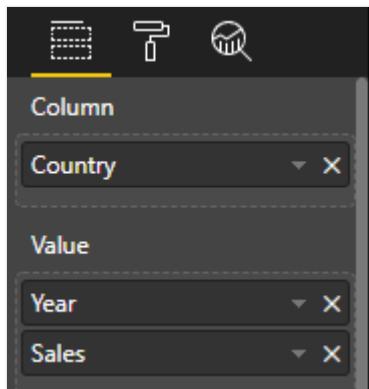
```

Visualize the table data view like this example:

[ ] Expand table

Country/Region	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:

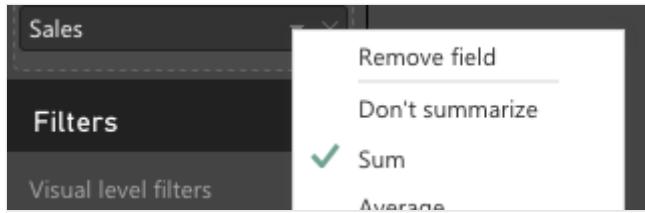


Power BI displays your data as the table data view. *Don't assume that the data is ordered.*

JSON

```
{
  "table" : {
    "columns": [...],
    "rows": [
      [
        "Canada",
        2014,
        630
      ],
      [
        "Canada",
        2015,
        490
      ],
      [
        "Mexico",
        2013,
        645
      ],
      [
        "UK",
        2014,
        831
      ],
      [
        "USA",
        2015,
        650
      ],
      [
        "USA",
        2016,
        350
      ]
    ]
  }
}
```

To aggregate the data, select the desired field and then choose **Sum**.



Code sample to process table data view mapping.

TypeScript

```
"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;
// standard 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("th");
        tableDataView.columns.forEach((column: DataViewMetadataColumn) => {
            const tableHeaderColumn = document.createElement("td");
            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 the layout for the table:

```

less

table {
    display: flex;
    flex-direction: column;
}

tr, th {
    display: flex;
    flex: 1;
}

td {
    flex: 1;
    border: 1px solid black;
}

```

The resulting visual looks like this:

Country, Year and Sales

Country	Year	Sales
Canada	2014	630
Canada	2015	490
Mexico	2013	645
UK	2015	831
USA	2015	650
USA	2016	350
Total		3596

(Circular arrows, magnifying glass, etc.)

# 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.

```
JSON

{
  "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"
              }
            }
          ]
        }
      }
    }
  ]
}
```

```
    ]  
}
```

## Hierarchical structure of matrix data

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.

Semantic model:

 Expand table

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
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 screenshot shows the Power BI Data View interface. On the left is a table with four columns: Parents, Col1, Col2, and Total. The table has rows for Parent1, Child1, Grand child1, Grand child2, Child2, Grand child3, Grand child4, Grand child5, Parent2, Child3, Grand child6, Grand child7, Grand child8, and Total. The 'Total' row is bolded. On the right is the Data Model view, which includes sections for Rows, Columns, and Values. Under Rows, there are three levels: Parents, Children, and Grand children. Under Columns, there is one level: Columns. Under Values, there is one level: Values. Below these are sections for FILTERS and Visual level filters. The Visual level filters section shows a dropdown for 'Children' set to 'is (All)'.

Parents	Col1	Col2	Total
<b>Parent1</b>	<b>24</b>	<b>31</b>	<b>55</b>
Child1	12	14	26
Grand child1	5	6	11
Grand child2	7	8	15
<b>Child2</b>	<b>12</b>	<b>17</b>	<b>29</b>
Grand child3	5	3	8
Grand child4	4	9	13
Grand child5	3	5	8
<b>Parent2</b>	<b>18</b>	<b>16</b>	<b>34</b>
<b>Child3</b>	<b>18</b>	<b>16</b>	<b>34</b>
Grand child6	1	2	3
Grand child7	7	1	8
Grand child8	10	13	23
<b>Total</b>	<b>42</b>	<b>47</b>	<b>89</b>

The visual gets its data structure as described in the following code (only the first two table rows are shown here):

JSON

```
{
  "metadata": {...},
  "matrix": {
    "rows": {
      "levels": [...],
      "root": {
        "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": [...]
}
}

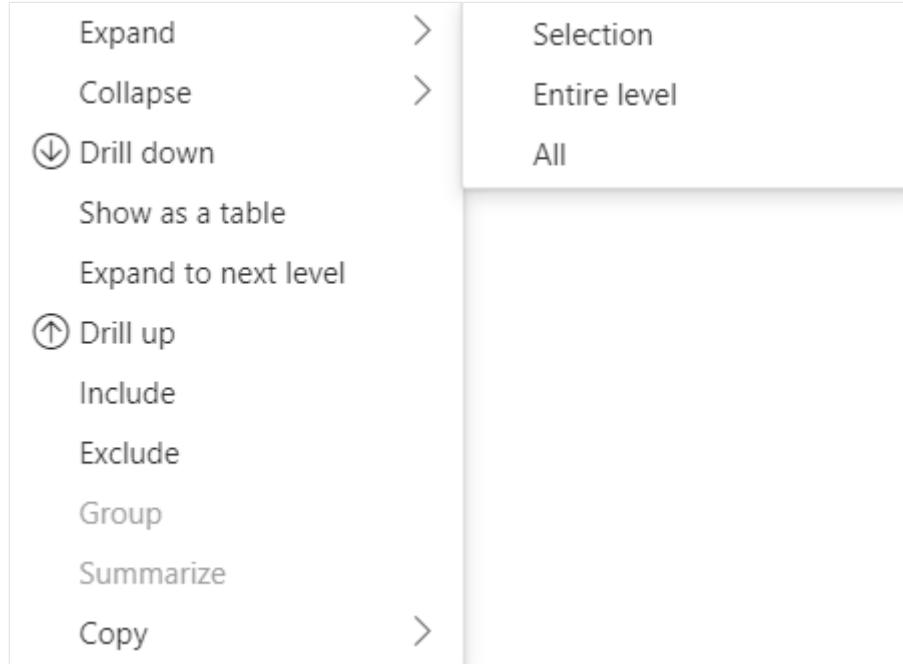
```

## Expand and collapse row headers

For API 4.1.0 or later, matrix data supports [expanding and collapsing row headers](#). From API 4.2 you can expand/collapse entire level programmatically. The expand and collapse feature optimizes fetching data to the dataView by allowing the user to expand or

collapse a row without fetching all the data for the next level. It only fetches the data for the selected row. The row header's expansion state remains consistent across bookmarks and even across saved reports. It's not specific to each visual.

Expand and collapse commands can be added to the context menu by supplying the `dataRoles` parameter to the `showContextMenu` method.



To expand a large number of data points, use the [fetch more data API](#) with the expand/collapse API.

## API features

The following elements have been added to API version 4.1.0 to enable expanding and collapsing row headers:

- The `isCollapsed` flag in the `DataViewTreeNode`:

```
TypeScript

interface DataViewTreeNode {
    //...
    /**
     * TRUE if the node is Collapsed
     * FALSE if it is Expanded
     * Undefined if it cannot be Expanded (e.g. subtotal)
     */
    isCollapsed?: boolean;
}
```

- The `toggleExpandCollapse` method in the `ISelectionManger` interface:

TypeScript

```
interface ISelectionManager {  
    //...  
    showContextMenu(selectionId: ISelectionId, position: IPoint,  
    dataRoles?: string): IPromise<{}>; // dataRoles is the name of the role  
    of the selected data point  
    toggleExpandCollapse(selectionId: ISelectionId, entireLevel?:  
    boolean): IPromise<{}>; // Expand/Collapse an entire level will be  
    available from API 4.2.0  
    //...  
}
```

- The `canBeExpanded` flag in the `DataViewHierarchyLevel`:

TypeScript

```
interface DataViewHierarchyLevel {  
    //...  
    /** If TRUE, this level can be expanded/collapsed */  
    canBeExpanded?: boolean;  
}
```

## Visual requirements

To enable the expand collapse feature on a visual by using the matrix data view:

1. Add the following code to the `capabilities.json` file:

JSON

```
"expandCollapse": {  
    "roles": ["Rows"], // "Rows" is the name of rows data role  
    "addDataViewFlags": {  
        "defaultValue": true // indicates if the DataViewTreeNode will  
        get the isCollapsed flag by default  
    }  
},
```

2. Confirm that the roles are drillable:

JSON

```
"drilldown": {  
    "roles": ["Rows"]  
},
```

3. For each node, create an instance of the selection builder by calling the `withMatrixNode` method in the selected node hierarchy level and creating a `selectionId`. For example:

TypeScript

```
let nodeSelectionBuilder: ISelectionIdBuilder =
visualHost.createSelectionIdBuilder();
// parentNodes is a list of the parents of the selected node.
// node is the current node which the selectionId is created for.
parentNodes.push(node);
for (let i = 0; i < parentNodes.length; i++) {
    nodeSelectionBuilder =
nodeSelectionBuilder.withMatrixNode(parentNodes[i], levels);
}
const nodeSelectionId: ISelectionId =
nodeSelectionBuilder.createSelectionId();
```

4. Create an instance of the selection manager, and use the `selectionManager.toggleExpandCollapse()` method, with the parameter of the `selectionId`, that you created for the selected node. For example:

TypeScript

```
// handle click events to apply expand\collapse action for the
selected node
button.addEventListener("click", () => {
    this.selectionManager.toggleExpandCollapse(nodeSelectionId);
});
```

### ⓘ Note

- If the selected node is not a row node, PowerBI will ignore expand and collapse calls and the expand and collapse commands will be removed from the context menu.
- The `dataRoles` parameter is required for the `showContextMenu` method only if the visual supports `drilldown` or `expandCollapse` features. If the visual supports these features but the `dataRoles` wasn't supplied, an error will output to the console when using the developer visual or if debugging a public visual with debug mode enabled.

## Considerations and limitations

- After you expand a node, new data limits will be applied to the DataView. The new DataView might not include some of the nodes presented in the previous DataView.
- When using expand or collapse, totals are added even if the visual didn't request them.
- Expanding and collapsing columns isn't supported.

## Keep all metadata columns

For API 5.1.0 or later, keeping all metadata columns is supported. This feature allows the visual to receive the metadata for all columns no matter what their active projections are.

Add the following lines to your *capabilities.json* file:

```
JSON

"keepAllMetadataColumns": {
  "type": "boolean",
  "description": "Indicates that visual is going to receive all metadata columns, no matter what the active projections are"
}
```

Setting this property to `true` will result in receiving all the metadata, including from collapsed columns. Setting it to `false` or leaving it undefined will result in receiving metadata only on columns with active projections (expanded, for example).

## Data reduction algorithm

The data reduction algorithm controls which data and how much data is received in the data view.

The *count* is set to the maximum number of values that the data view can accept. If there are more than *count* values, the data reduction algorithm determines which values should be received.

## Data reduction algorithm types

There are four types of data reduction algorithm settings:

- `top`: The first *count* values are taken from the semantic model.
- `bottom`: The last *count* values are taken from the semantic model.

- `sample`: The first and last items are included, and `count` number of items with equal intervals between them. For example, if you have a semantic model [0, 1, 2, ... 100] and a `count` of 9, you 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. In the future, a windowing setting will be fully supported.

By default, all Power BI visuals have the top data reduction algorithm applied with the `count` set to 1000 data points. This default is equivalent to setting the following properties in the `capabilities.json` file:

JSON

```
"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 usage

The data reduction algorithm can be used in categorical, table, or matrix data view mapping.

In categorical data mapping, you can add the algorithm to the "categories" and/or "group" section of `values` for categorical data mapping.

JSON

```
"dataViewMappings": {
    "categorical": {
        "categories": {
            "for": { "in": "category" },
            "dataReductionAlgorithm": {
                "window": {
                    "count": 300
                }
            }
        },
        "values": {
            "group": {
                "by": "series",
                "select": [
                    "for": {
                        "in": "value"
                    }
                ]
            }
        }
    }
}
```

```
        "in": "measure"
    }
],
"dataReductionAlgorithm": {
    "top": {
        "count": 100
    }
}
}
}
```

In table data view mapping, apply the data reduction algorithm to the `rows` section of the Data View mapping table.

JSON

```
"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.

## Related content

- Add drill-down support
- Create custom Power BI visuals without data binding

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Sorting options for Power BI visuals

Article • 10/12/2024

This article describes the different options available for specifying the way a visual sorts items in Power BI.

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

The *sorting* capability requires at least one of the following parameters:

- `default`
- `implicit`
- `custom`

## Default sorting

The `default` option is the simplest form. It allows the user to sort according to any one field and direction (ascending or descending). The user selects the direction and field from the **more options** menu.

A screenshot of a Power BI table visual. The table has two columns: 'City' and 'Total Units This Year'. The 'Total Units This Year' column is sorted in descending order, with Harrisburg, PA at the top and Pittsburgh, PA at the bottom. A context menu is open over the table, with 'Sort by' selected. A dropdown menu shows 'City' and 'Total Units This Year' as options. The 'Sort by' option is highlighted with a yellow bar.

City	Total Units This Year
Harrisburg, PA	\$119,440
Charleston, WV	\$117,185
Huntington, WV	\$107,095
West Mifflin, PA	\$106,941
Abingdon, MD	\$101,385
Morgantown, WV	\$100,957
Boardman, OH	\$100,517
North Canton, OH	\$98,955
Greensburg, PA	\$97,270
Washington, PA	\$96,426
Uniontown, PA	\$95,983
Lavale, MD	\$93,950
Erie, PA	\$92,308
Pittsburgh, PA	\$91,701
<b>Total</b>	<b>\$4,015,391</b>

To enable default sorting, add the following code to your `capabilities.json` file:

JSON

```
"sorting": {  
    "default": {}  
}
```

# Implicit sorting

Implicit sorting allows you to pre-define a sorting array using parameter `clauses`, that describes sorting for each data role. The user can't change the sorting order, so Power BI doesn't display sorting options in the visual's menu. However, Power BI does sort data according to specified settings.

To enable implicit sorting, add the implicit `clauses` to your `capabilities.json` file `clauses` parameters can contain several objects with two parameters each:

- `role`: Determines `DataMapping` for sorting
- `direction`: Determines sort direction (1 = Ascending, 2 = Descending)

```
JSON

"sorting": {
    "implicit": {
        "clauses": [
            {
                "role": "category",
                "direction": 1
            },
            {
                "role": "measure",
                "direction": 2
            }
        ]
    }
}
```

# Custom sorting

Custom sorting gives the developer more flexibility when sorting. The developer can:

- Allow the user to sort by multiple fields at a time.
- Set a default sorting order for the data
- Allow custom sorting operations during runtime

## Enable custom sorting

To enable custom sorting, add the following code to your `capabilities.json` file:

```
TypeScript
```

```
"sorting": {  
    "custom": {}  
}
```

## Example: Custom sort API

TypeScript

```
let queryName1 =  
    this.dataView.matrix.columns.levels[0].sources[0].queryName;  
let queryName2 =  
    this.dataView.matrix.columns.levels[1].sources[0].queryName;  
let args: CustomVisualApplyCustomSortArgs = {  
    sortDescriptors: [  
        {  
            queryName: queryName1,  
            sortDirection: powerbi.SortDirection.Ascending  
        },  
        {  
            queryName: queryName2,  
            sortDirection: powerbi.SortDirection.Descending  
        },  
    ]  
};  
this.host.applyCustomSort(args);
```

## Related content

- Understand data view mapping in Power BI visuals
- Understanding capabilities

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Objects and properties of Power BI visuals

Article • 10/11/2024

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`.

JSON

```
"objects": {  
    "myCustomObject": {  
        "properties": { ... }  
    }  
}
```

## Display name and description

ⓘ Note

Display name and description are deprecated from API version 5.1+. The display name and description are now added in the formatting model instead of the `capabilities.json` file.

`displayName` is the name that will be shown in the property pane. `description` is a description of the formatting property that will be shown to the user as a tooltip.

## Properties

`properties` is a map of properties that are defined by the developer.

JSON

```
"properties": {  
    "myFirstProperty": {  
        "type": ValueTypeDescriptor | StructuralTypeDescriptor  
    }  
}
```

## ⓘ Note

`show` is a special property that enables a switch to toggle the object.

Example:

JSON

```
"properties": {  
    "show": {  
        "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:

TypeScript

```
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*.

TypeScript

```
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:

JSON

```
"properties": {  
    "showAllDataPoints": {  
        "type": {  
            "bool": true  
        }  
    },  
    "fill": {  
        "type": {  
            "fill": {  
                "solid": {  
                    "color": true  
                }  
            }  
        }  
    },  
    "fillRule": {  
        "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:

JSON

```
{  
  "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"  
    }  
  ]  
}
```

## Formatting pane

To customize the properties in the formatting pane, use one of the following methods, depending on what API version you're using.

getFormattingModel API method

### ⓘ Note

The `getFormattingModel` API method is supported from API versions 5.1+. To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.

To use objects effectively in API version 5.1+, you need to implement the `getFormattingModel` method.

This method builds and returns a formatting model that includes full [properties pane](#) hierarchy of formatting cards, formatting groups, Also it contains formatting properties and their values.

## Capabilities objects reflected in formatting model

Each formatting property in the formatting model needs a corresponding object in the `capabilities.json` file. The formatting property should contain a descriptor with an object name and property name that exactly match the corresponding capabilities object (the object and property names are case sensitive).

For example:

For the following formatting property in the formatting model (See the descriptor object content):

TypeScript

```
const myCustomCard: powerbi.visuals.FormattingCard = {
    displayName: "My Custom Object Card",
    uid: "myCustomObjectCard_uid",
    groups: [
        {
            displayName: undefined,
            uid: "myCustomObjectGroup_uid",
            slices: [
                {
                    uid: "myCustomProperty_uid",
                    displayName: "My Custom Property",
                    control: {
                        type:
                            powerbi.visuals.FormattingComponent.ColorPicker,
                        properties: {
                            descriptor: {
                                objectName: "myCustomObject",
                                propertyName: "myCustomProperty",
                                selector: null // selector is
optional
                            },
                            value: { value: "#000000" }
                        }
                    }
                }
            ],
        }],
};
```

The corresponding object from the capabilities `objects` section should be:

## JSON

```
"objects": {  
    "myCustomObject": {  
        "properties": {  
            "myCustomProperty": {  
                "type": {  
                    "fill": {  
                        "solid": {  
                            "color": true  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

## Formatting property selector

The optional selector in formatting properties descriptor determines where each property is bound in the dataView. There are [four distinct options](#).

### Example

The above `myCustomCard` example shows what formatting property in formatting model would look like for an object with one property `myCustomProperty`. This property object bound *statically* to `dataViews[index].metadata.objects`. Selector in descriptor can be changed accordingly to [selector type](#) you choose.

## Objects selectors types

The selector in `enumerateObjectInstances` determines where each object is bound in the dataView. There are four distinct options:

- [static](#)
- [columns](#)
- [selector](#)
- [scope identity](#)

### static

This object is bound to metadata `dataviews[index].metadata.objects`, as shown here.

TypeScript

```
selector: null
```

## columns

This object is bound to columns with the matching `QueryName`.

TypeScript

```
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 created `selectionIDs` for some data points, and we're looping through them.

TypeScript

```
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.

TypeScript

```
selector: {  
    data: <DataViewScopeIdentity[]>identities  
}
```

## Related content

Performance tips

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Performance tips for creating quality Power BI custom visuals

Article • 05/15/2024

This article covers techniques on how a developer can achieve high performance when rendering their custom visuals.

No one wants a visual to take a long time to render. Getting the visual to render as quickly as possible is critical when writing the code.

## 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, we recommend that you keep up-to-date with the most recent version. To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.

Here are some recommendations for achieving optimal performance for your custom visual.

## Reduce plugin size

A smaller custom visual plugin size results in:

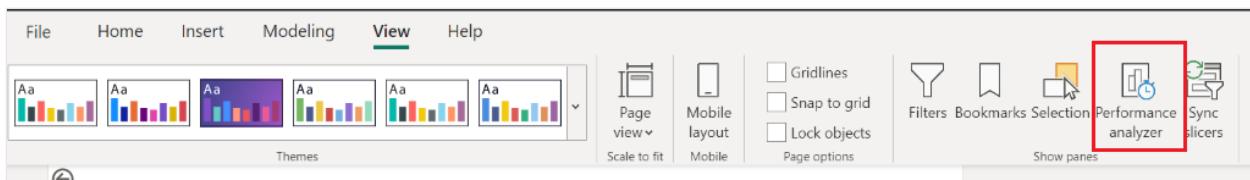
- Faster download time
- Faster installation whenever the visual is run

These third party resources can help you decrease your plugin size, by finding ways for you to [remove unused code](#) or [tree-shaking and code-splitting](#).

## Check render time of the visual

Measure the render time of your visual in various situations to see which, if any, parts of the script need optimization.

## Power BI Desktop performance analyzer



Use the [Power BI Desktop performance analyzer](#) ([View > Performance Analyzer](#)) to check how your visual renders in the following cases:

- First render of the visual
- Thousands of data points
- A single data point/measure (to determine the visual render overhead)
- Filtering
- Slicing
- Resizing (may not work in the performance analyzer)

If possible, compare these measurements with those of a similar core visual to see if there are parts that can be optimized.

## Use the User Timing API

Use the [User Timing API](#) to measure your app's JavaScript performance. This API can also help you decide which parts of the script need optimization.

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

## Other ways to test your custom visual

- Code instrumentation - Use the following console tools to gather data about your custom visual's performance (note that these tools link to external third party tools):
  - [console.log\(\)](#)
  - [console.dir\(\)](#)
  - [console.time\(\)](#)
  - [console.timeEnd\(\)](#)
- The following web developer tools can also help measure your visual's performance, but keep in mind that they profile Power BI as well:
  - [Metrics](#)
  - [JavaScript profiler](#)

Once you determined which parts of your visual need optimization, check out these tips.

# Update messages

When you update the visual:

- Don't rerender the entire visual if only some elements have changed. Render only the necessary elements.
- Store the data view passed on update. Render only the data points that are different from the previous data view. If they haven't changed, there's no need to rerender them.
- Resizing is often done automatically by the browser and doesn't require an update to the visual.

## Cache DOM nodes

When a node or list of nodes is retrieved from the DOM, 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 more nodes in the relevant area, caching them can improve the 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.

For example:

**Instead of:**

```
JavaScript

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

**Try:**

```
JavaScript

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 manipulations as much as possible. *Insert operations* like `prepend()`, `append()`, and `after()` are time-consuming and should only be used when necessary.

For example:

Instead of:

JavaScript

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

Try:

Make the above example faster by using `html()` and building the list beforehand:

JavaScript

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

# Reconsider JQuery

Limit JS frameworks and use native JS whenever possible to increase the available bandwidth and lower your processing overhead. Doing this might also decrease compatibility issues with older browsers.

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

# Animation

## Animation options

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.

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

## Canvas performance tips

Check out the following third party sites for tips on improving canvas performance.

- [Fast load times ↗](#)
- [Improving HTML5 Canvas performance ↗](#)
- [Optimizing canvas ↗](#)

For example, learn how to [avoid unnecessary canvas state changes ↗](#) by rendering by color instead of position.

## Animation functions

Use [requestAnimationFrame ↗](#) to update your on-screen animations, so your animation functions are called **before** the browser calls another repaint.

## Animation loops

Does the animation loop redraw unchanged elements?

If so, it wastes time drawing elements that don't change from frame-to-frame.

Solution: Update the frames selectively.

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

Instead, consider using 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 often found in axes and legends.

## Common issues

- Text size calculation: When there are a lot of data points, don't waste time calculating text size for each point. Calculate a few points and then estimate.

- If some elements of the visual aren't seen in the display, there's no need to render them.

## Related content

[Optimization guide for Power BI](#)

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

---

## Feedback

Was this page helpful?



Yes



No

[Provide product feedback](#) | [Ask the community](#)

# How to create mobile-friendly Power BI visuals

Article • 01/19/2024

Mobile devices allow users to connect to their data anytime and anywhere.

[Power BI apps for Windows, iOS, and Android](#) enable business users to have a comprehensive view of their data that's always at their fingertips.

As a developer creating Power BI visuals, you must address the unique constraints of each mobile device to reach as many users as possible and provide the best mobile experience.

## Required functionality

The following requirements are essential for developing mobile-friendly visuals:

- **Rendering**

A Power BI visual has to render on all [supported mobile devices](#), including browsers and applications. There should be no errors in reports and dashboards, or when visuals run in **Focus** mode.

- **Interactivity**

Mobile devices should have the same interactive functionality as desktop devices. All events handled on desktop browsers must be supported, or have comparable event handlers, on mobile devices.

For example, if a desktop visual supports multi-selection using the `ctrl` key, consider adding a similar event handler for mobile devices.

The following table provides a list of corresponding events on mobile devices.

[Expand table](#)

Mouse event name	Touch event name
<code>click</code>	<code>click</code>
<code>mousemove</code>	<code>touchmove</code>
<code>mousedown</code>	<code>touchstart</code>

Mouse event name	Touch event name
mouseup	touchend
dblclick	external library
contextmenu	external library
mouseover	touchmove
mouseout	touchmove (or external library)
wheel	N/A

➊ Note

Not all mobile or touch screen devices support mouse (or *mouse* prefixed) events. In unsupported cases, handle both *mouse* and *touch* events at the same time.

## Optional functionality

The following functions are optional. The optional functions can be used to create a better end-user experience.

- Recommended rendering

To support smaller visual sizes, add format options that allow the user to adjust the size of each element. For example, add format options to labels to use in reports and dashboards. The format options allow users to customize a visual specifically for their mobile device.

The same settings can be applied to the visuals in desktop browsers and, if needed, be overridden to adapt the visual to smaller screens.

➊ Note

To optimize a visual in **Focus** mode, both portrait and landscape screen size orientations should be considered. See [Display content in Focus mode](#).

- Recommended interactivity

Consider adding mobile-specific event handlers, like dragging and scrolling.

- Failover

If a visual can't render on a mobile device, the visual should show a **descriptive** error.

## Supported browsers and devices

Power BI visuals must render on all devices that support Power BI apps. For more information, see [supported browsers for Power BI](#) and [Power BI mobile apps](#).

## Related content

[Develop a Power BI circle card visual](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

# Debug Power BI custom visuals

Article • 10/14/2024

This article describes some basic debugging procedures that you can use when developing your visual. After you read this article, you should be able to debug your visual by inserting breakpoints and working with exceptions.

## Insert breakpoints

The visual's entire JavaScript is reloaded every time the visual is updated. Any breakpoints that you add will be lost when the debug visual is refreshed.

As a workaround, use `debugger` statements in your code. We recommend that you turn off automatic reload while using `debugger` in your code. Here's an example of how to use a `debugger` statement in your `update` method:

TypeScript

```
public update(options: VisualUpdateOptions) {
    console.log('Visual update', options);
    debugger;
    this.target.innerHTML = `<p>Update count: <em>$ {(this.updateCount</em>
</p>`;
}
```

## Catch exceptions

When you're working on your visual, you notice that the Power BI service "consumes" all errors. This behavior is an intentional feature of Power BI. It prevents 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 with a decorator

To log exceptions in your Power BI visual, you need define an exception-logging decorator. To define the decorator, add the following code to your visual:

TypeScript

```
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;
                }
            }
        }
    }
}
```

You can use this decorator on any function to see error logging as follows:

TypeScript

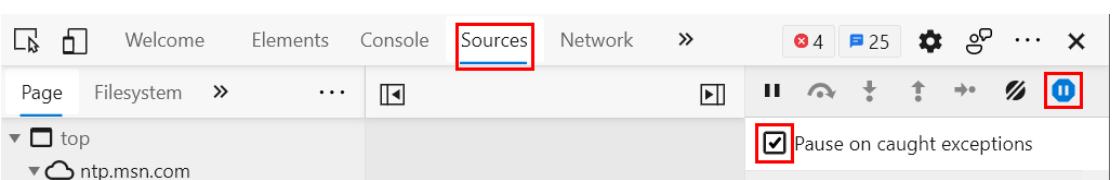
```
@logExceptions()
public update(options: VisualUpdateOptions) {
```

## Break on exceptions

You can set the browser to break on caught exceptions. Breaking stops code execution wherever an error happens, and you can debug from there.

Microsoft Edge

1. Open DevTools (F12).
2. Go to the **Sources** tab.
3. Select the **Pause on exceptions** icon (stop sign with a pause symbol).
4. Select the **Pause on caught exceptions** checkbox.



The screenshot shows the Microsoft Edge DevTools interface. The 'Sources' tab is selected, indicated by a red box around its tab header. In the bottom right corner of the sources panel, there is a checkbox labeled 'Pause on caught exceptions' with a red box around it. The rest of the interface shows the page tree and various developer tools controls.

## Related content

- [Troubleshoot your Power BI developer environment setup](#)
  - [Frequently asked questions about Power BI visuals](#)
- 

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Troubleshoot your Power BI developer environment setup for custom visuals

Article • 12/21/2024

This article explains how to diagnose and fix the following common problems that can occur when setting up your developer environment for creating custom Power BI visuals.

- [Can't start up pbviz](#)
- [Can't connect to the Power BI service](#)
- [Can't see the developer icon on the visualization tab](#)
- [Contacting the support team](#)

## Can't start up pbviz

When your environment isn't set up correctly, you might receive an error such as: *pbviz command not found*

When you run `pbviz` in your terminal's command line, you should see the help screen. If you don't, make sure you have [NodeJS](#) version 4.0 or higher installed. For help with installing `pbviz` or [NodeJS](#) see [Set up your environment for developing a Power BI visual](#).

## Can't connect to the Power BI service

Run the visual server with the command `pbviz start` in your terminal's command line from the root of your visual project.

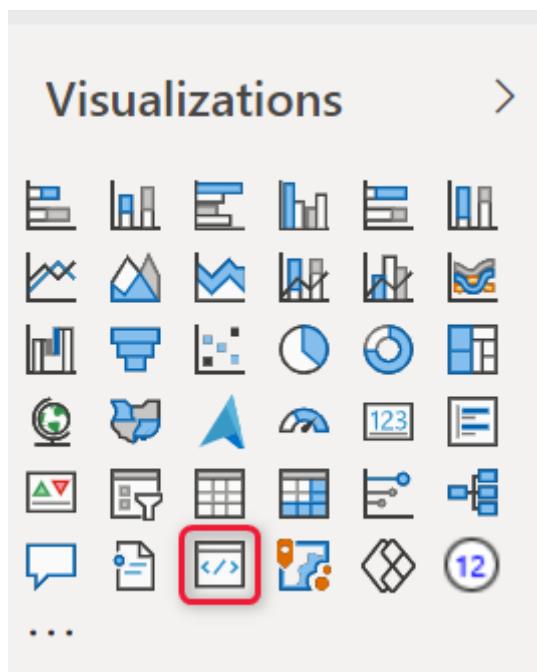
```
D:\src\Visuals\myVisual>pbviz start
info Building visual...
done build complete

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

If the server's not running, your SSL certificates were probably not installed correctly. To install your SSL certificates, see [Create an SSL certificate](#).

## Can't see the developer icon on the Visualizations tab

The developer icon looks like a prompt icon within the **Visualizations** tab.



If you don't see it, make sure you have [enabled developer mode in the Power BI settings](#).

! Note

The developer visual is currently only available in the **Power BI service** and not in Power BI Desktop or the mobile app. The packaged visual will work everywhere.

## Contacting the support team

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.

This support channel is for custom visuals developers in the process of developing their own visuals.

For customer experience issues when using custom Power BI visuals, submit a case request via the [Power Platform admin center portal ↗](#).

## Related content

- [Setting up your Power BI environment](#)
- [Frequently asked questions about Power BI visuals](#)

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

---

# Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Create a dialog box for your Power BI visual

Article • 12/30/2023

When you create a visual, sometimes it's useful to display additional information to the customer in a separate window. For example, you might want to:

- **Show additional information** - Such as a text note or a video
- **Display an input data dialog box** - Such as a date dialog box

For these purposes, you can create a dialog visual pop-up window, called a *dialog box* in this article.

## Dialog box considerations

When creating a dialog box for your visual, keep in mind that:

- During development, you can specify the size and position of the dialog box.
- When the dialog box is triggered, the report background is grayed.
- The dialog header contains the visual's icon and its display name as a title.
- The dialog box can have up to three action buttons. You can choose which buttons to display from a [given selection](#).
- The dialog box uses a rich HTML `iframe`.
- While the dialog box is displayed, no action can be performed in the report until it's dismissed.
- The dialog code can use external npm libraries, just like the visual.

### Important

The dialog box should not be triggered spontaneously. It should be the immediate result of a user action.

## Design a dialog box for your visual

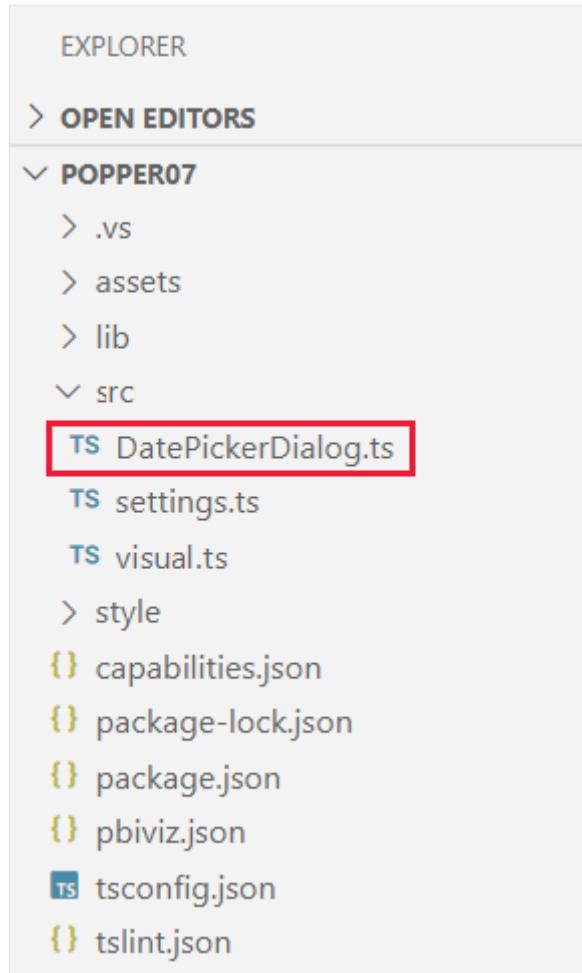
To configure a dialog box, you need to add two components to your code:

- [An implementation file](#) - It's best practice to create an implementation file for each dialog box.

- [Code to invoke your dialog box](#) - To invoke your dialog box, add code to the `visual.ts` file.

## Create the dialog box implementation file

We recommend creating an implementation file for each dialog box you create. Place your dialog box files in the `src` folder:



Each dialog box implementation file should include the following components:

- [A dialog box class](#)
- [A result class](#)
- [Registration of the dialog class](#)

## Create a dialog box class

Create a dialog box class for your dialog box. The `initialState` parameter in `openModalDialog` is passed to the dialog contractor upon its creation. Use the `initialState` object to pass parameters to the dialog box, in order to affect its behavior or appearance.

The dialog code can use these `IDialogHost` methods:

- `IDialogHost.setResult(result:object)` - The dialog code returns a result object that is passed back to its calling visual.
- `IDialogHost.close(actionId: DialogAction, result?:object)` - The dialog code can programmatically close the dialog and provide a result object back to its calling visual.

Imports on top of the file:

TypeScript

```
import powerbi from "powerbi-visuals-api";
import DialogConstructorOptions =
powerbi.extensibility.visual.DialogConstructorOptions;
import DialogAction = powerbi.DialogAction;
// React imports as an example
import * as ReactDOM from 'react-dom';
import * as React from 'react';
import reactDatepicker from 'react-datepicker';
import 'react-datepicker/dist/react-datepicker.css';
```

This example requires to install these packages:

Console

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

Class realization:

TypeScript

```
export class DatePickerDialog {
    static id = "DatePickerDialog";

    constructor(options: DialogConstructorOptions, initialState: object) {
        const host = options.host;
        let pickedDate: Date;
        const startDate = new Date(initialState['startDate']);

        // Dialog rendering implementation
        ReactDOM.render(
            React.createElement(
                reactDatepicker,
                {
                    inline: true,
                    openToDate: startDate,
                    onChange: (date: Date) => {
                        pickedDate = date
                }
            )
        );
    }
}
```

```
        host.setResult({ date: pickedDate })
    }
},
null),
options.element
);

document.addEventListener('keydown', e => {
    if (e.code == 'Enter' && pickedDate) {
        host.close(DialogAction.Close, {date: pickedDate});
    }
});
}
}
```

## Create a result class

Create a class that returns the dialog box result, and then add it to the dialog box implementation file.

In the example below, the `DatePickerDialogResult` class returns a date string.

TypeScript

```
export class DatePickerDialogResult {
    date: string;
}
```

## Add your dialog box to the registry list

Every dialog implementation file needs to include a registry reference. Add the two lines in the example below to the end of your dialog box implementation file. The first line should be identical in every dialog box implementation file. The second line lists your dialog box; modify it according to the name of your dialog box class.

JavaScript

```
globalThis.dialogRegistry = globalThis.dialogRegistry || {};
globalThis.dialogRegistry[DatePickerDialog.id] = DatePickerDialog;
```

## Invoke the dialog box

Before you create a dialog box, you need to decide which buttons it will include. Power BI visuals support the following six dialog box buttons:

TypeScript

```
export enum DialogAction {
    Close = 0,
    OK = 1,
    Cancel = 2,
    Continue = 3,
    No = 4,
    Yes = 5
}
```

Each dialog box you create needs to be invoked in the `visual.ts` file. In this example, the dialog box is defined with two action buttons.

TypeScript

```
import powerbi from "powerbi-visuals-api";
import DialogAction = powerbi.DialogAction;
const dialogActionsButtons = [DialogAction.OK, DialogAction.Cancel];
```

In this example, the dialog box is invoked by clicking a visual button. The visual button is defined as part of the visual constructor in the `visual.ts` file.

## Define the size and position of the dialog box

From API version 4.0 or later, you can define the size and position of the dialog box using the `DialogOpenOptions` parameter of `openModalDialog`. To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.

TypeScript

```
export interface RectSize {
    width: number;
    height: number;
}

export interface DialogOpenOptions {
    title: string;
    size?: RectSize;
    position?: VisualDialogPosition;
    actionButtons: DialogAction[];
}
```

The `position` parameter lets you decide where the dialog box should open on the screen. You can choose to open the dialog box in the center of the screen, or you can

define a different position relative to the visual.

TypeScript

```
const enum VisualDialogPositionType {
    Center = 0,
    RelativeToVisual = 1
}

export interface VisualDialogPosition {
    type: VisualDialogPositionType;
    left?: number;
    top?: number;
}
```

- If no type is specified, the default behavior is to open the dialog box in the center.
- The position is given in pixels relative to the top left corner of the visual.

This example shows a 250 x 300 pixel date selection dialog box 100 pixels to the left and 30 pixels below the top of the visual:

TypeScript

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

    constructor(options: VisualConstructorOptions) {
        this.host = options.host;
        this.target = options.element;
        const dialogActionsButtons = [DialogAction.OK, DialogAction.Cancel];

        const sectionDiv = document.createElement("div");

        const span = document.createElement("span");
        span.id = "datePicker";

        let button = document.createElement("button");
        button.id = "DateButton";
        button.innerText = "Date";

        button.onclick = (event) => {
            const initialDialogState = { startDate: new Date() };
            const position = {
                type: VisualDialogPositionType.RelativeToVisual,
                left: 100,
                top: 30
            };

            const size = {width: 250, height: 300};
            const dialogOptions = {
                actionButtons: dialogActionsButtons,
                ...initialDialogState,
                position
            };
        };
    }
}
```

```

        size: size,
        position: position,
        title: "Select a date"
    );
    this.host.openModalDialog(DatePickerDialog.id, dialogOptions,
initialDialogState).
        then(ret => this.handleDialogResult(ret, span)).
        catch(error => this.handleDialogError(error, span));
}
sectionDiv.appendChild(button);
sectionDiv.appendChild(span);
this.target.appendChild(sectionDiv)
}

// Custom logic to handle dialog results
private handleDialogResult( result: ModalDialogResult, targetElement:
HTMLElement){
    if ( result.actionId === DialogAction.OK || result.actionId ===
DialogAction.Close) {
        const resultState = <DatePickerDialogResult> result.resultState;
        const selectedDate = new Date(resultState.date);
        targetElement.textContent = selectedDate.toDateString();
    }
}

// Custom logic to handle errors in dialog
private handleDialogError( error: any, targetElement: HTMLElement ) {
    targetElement.textContent = "Error: " + JSON.stringify(error);
}
}

```

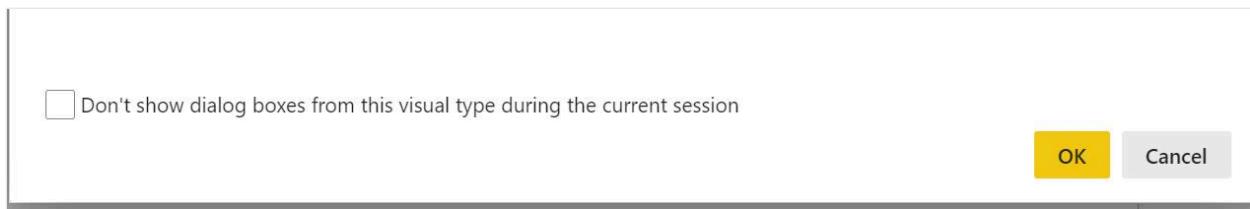
## Define how to close the dialog box

The preferred method for closing the dialog box is by the end-user clicking the [x] button, one of the action buttons, or the report background.

You can also program the dialog box to automatically close by calling the `IDialogHost` close method. This method is blocked for five seconds after the dialog is opened, so that the earliest you can automatically close the dialog box is five seconds after it was initiated.

## Don't show dialog box

The dialog box appears with a checkbox that gives the user the option to block dialog boxes.



This checkbox is a security feature that prevents the visual from creating modal dialogs (either intentionally or not) without the user's consent.

This blocking is in effect only for the current session. So if a user blocks the CV modal dialogs but later changes their mind, they can re-enable the dialogs. To do it they need to start a new session (refresh the reports page in Power BI service, or restart Power BI Desktop).

## Considerations and limitations

- As of powerbi-visuals-API 3.8, the dialog icon and title are determined by the visual's icon and display name and can't be changed.
- The size limitations of the dialog box are described in the table below.

[\[+\] Expand table](#)

Max/min	Width	Height
Maximum	90% of the browser width	90% of the browser height
Minimum	240 px	210 px

- When defining the position of the dialog box, the horizontal position can be a positive or negative integer, depending on which side of the visual you want the box to be. The vertical position can't be negative, as this would place it above the visual.
- The following features don't support the Power BI visuals dialog box:
  - Embedded analytics
  - Publish to web
  - Dashboards

You can program your visual to detect if the current environment allows opening a dialog box, by checking the boolean `this.host.hostCapabilities.allowModalDialog`.

## Related content

[Publish a Power BI custom visual](#)

[Create a Power BI bar chart visual](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

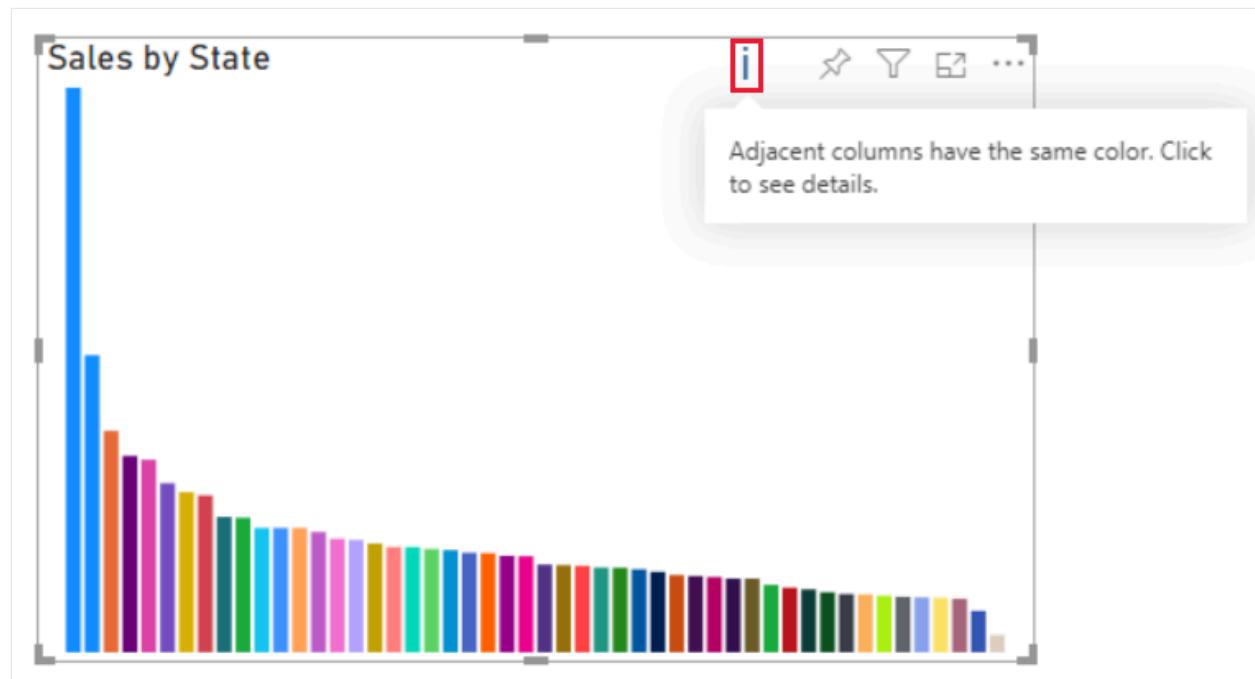
# Add a display warning icon to your visual

Article • 03/16/2024

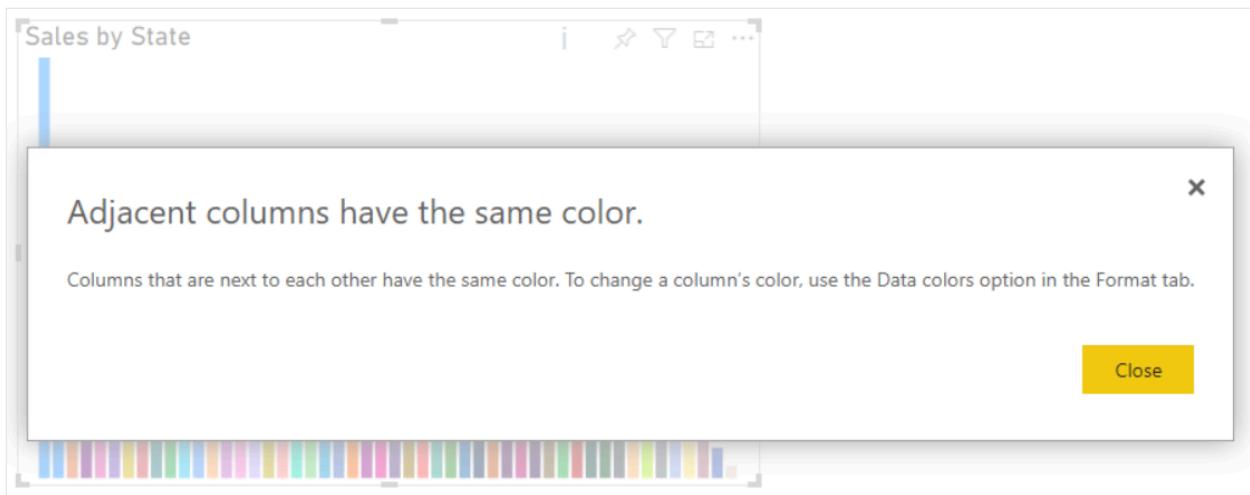
In a Power BI visual, a display warning icon can notify the user of a possible problem or an error. Display warning icons can be useful in many cases, such as:

- A map visual can display a warning icon when values are outside the latitude or longitude valid range.
- A pie chart visual can display a warning icon when it's displaying negative values that are mixed with positive ones.
- A cartesian chart can display a warning icon when infinity values are calculated. For example, if Y is zero, when dividing X by Y, the result is infinity.

When the icon appears, the user can hover over it to see the title of the warning message.



When the user selects the warning icon, a message that describes the problem appears in a pop-up window.



## Create a warning icon

You can create a warning icon with a customized message for a custom visual. The decision whether to raise the warning icon or not is up to you. As these examples demonstrate, the visual continues to function when the warning icon is displayed.

### Add a call to the update method

To add a display warning icon to your visual, invoke the `displayWarningIcon` method. The method is part of `IVisualHost` and is exposed using `powerbi-visuals-api`.

Add the following import to your file:

JavaScript

```
import powerbiVisualsApi from "powerbi-visuals-api";
```

After you add the import, add a condition that determines when to display the warning icon. Use the examples in this article to view two optional conditions.

### Example 1: Check language compatibility

Localization is used to display visuals in the customer's native language, which is determined by the operating system's settings.

In this example, the visual compares the language of the visual, which is set to US English, to the language of the operating system. If the languages don't match, the warning icon is displayed.

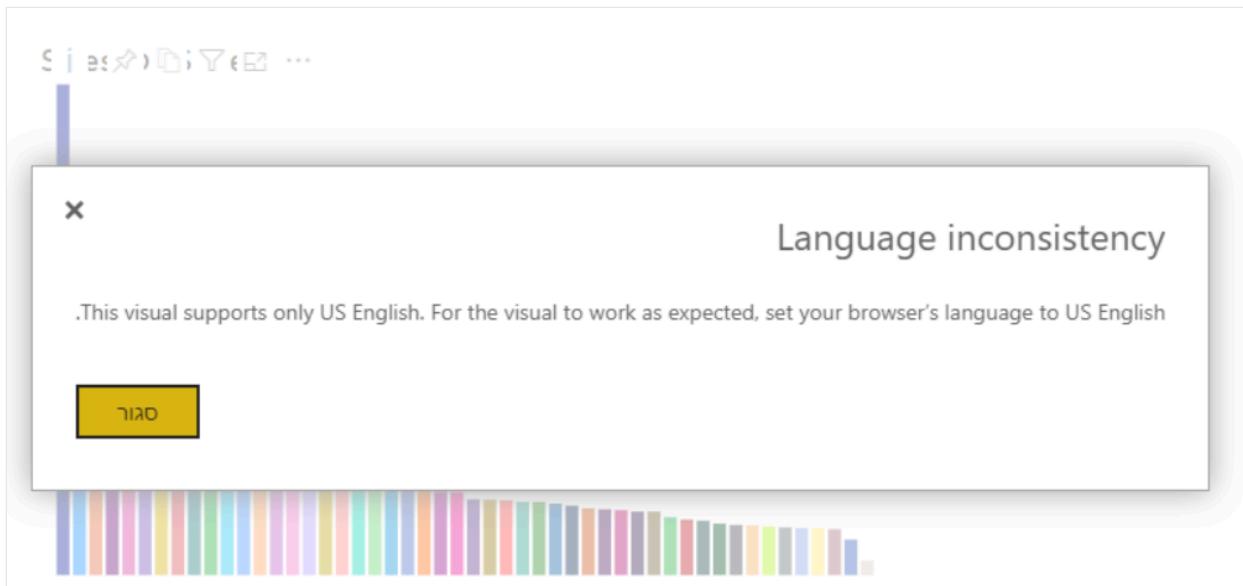
JavaScript

```

if (this.locale != 'en-US') {

    this.host.displayWarningIcon('Language inconsistency', 'This
visual supports only US English. For the visual to work as expected, set
your browser's language to US English.');
}

```



## Example 2: Compare colors in adjacent columns

In this example, the display warning icon appears when two columns that are next to each other have the same color.

The `getColumnColorByIndex` method iterates through all the columns. If two adjacent columns have the same color, a warning icon is displayed.

JavaScript

```

let category = options.dataViews[0].categorical.categories[0];

let dataValue = options.dataViews[0].categorical.values[0];

let colorPalette = this.host.colorPalette;

for (let i = 0, len = Math.max(category.values.length,
dataValue.values.length); i < len-1; i++) {

    const color1: string = getColumnColorByIndex(category, i, colorPalette);

    const color2: string = getColumnColorByIndex(category, i+1,
colorPalette);

    if (color1 == color2) {

```

```
        this.host.displayWarningIcon('Adjacent columns have the same  
color.', 'Columns that are next to each other have the same color. To change  
a column's color, use the data colors option in the Format tab.');
```

}

## Considerations and limitations

- Some errors and warnings aren't caused by the visual. For example, the *Too many values. Not showing all data* error is derived from the Power BI service. Such errors and warnings get propagated before to your visual's calls. They take precedence over errors that originate from your visual's code. If an error occurs while loading data, before the visual's code is run and the display warning icon condition is met, the visual displays that error instead of the display warning icon error.
- The display warning is cleared during each rendering of the visual, for example, when new data is dragged into the visual. The visual's `update()` method is invoked after the visual is rendered. If the visual's warning is raised based on a condition that's checked in the update method, each time the visual is rendered, if the condition is met, the visual displays the warning again.
- Resizing a visual doesn't affect the warning icon.

The following limitations refer to the display icon text:

- The maximum length of the title is 70 characters
- The maximum length of the text is 2,000 characters
- The text is only displayed as plain text

## Related content

DataViewUtils

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Visual interactions in Power BI visuals

Article • 10/22/2024

Sometimes you want to allow the user to interact with the visual by selecting, zooming, or clicking on it. Other times you want the visual to be static, so the user can't interact with the visual.

Visuals can query the value of the `allowInteractions` flag, which indicates if the visual allows visual interactions. For example, visuals can be interactive during [report](#) viewing or editing, but visuals can be non-interactive when they're viewed in a [dashboard](#). These interactions include *click*, *pan*, *zoom*, *selection*, and others.

## ⓘ Note

Best practice is to [enable tooltips](#) in all scenarios, regardless of the indicated flag.

## Set interactive permissions

The `allowInteractions` flag is passed as a boolean value during the initialization of the visual as a member of the `IVisualHost` interface.

For any Power BI scenario that requires non-interactive visuals (for example, dashboard tiles), set the `allowInteractions` flag to `false`. Otherwise (for example, Report), set `allowInteractions` to `true`.

The following code sample shows how to use the `allowInteractions` flag to set interactive permissions.

TypeScript

```
...
let allowInteractions = options.host.hostCapabilities.allowInteractions;
bars.on('click', function(d) {
    if (allowInteractions) {
        selectionManager.select(d.selectionId);
    ...
}
});
```

For more information about using the `allowInteractions` flag, see the [SampleBarChart visual repository](#) ↗.

## Related content

[Visual API](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Add interactivity into visual by Power BI visuals selections

Article • 12/29/2023

Power BI provides two ways to interact with visuals - selecting and filtering. The following example demonstrates how to select an item from one visual and notify the other visuals in the report about the new selection state.

The interface corresponds to a `Selection` object:

TypeScript

```
export interface ISelectionId {
    equals(other: ISelectionId): boolean;
    includes(other: ISelectionId, ignoreHighlight?: boolean): boolean;
    getKey(): string;
    getSelector(): Selector;
    getSelectorsByColumn(): SelectorsByColumn;
    hasIdentity(): boolean;
}
```

## Use the selection manager to select data points

The visual host object provides a method for [creating an instance of the selection manager](#). The selection manager has a corresponding method for each of the following actions:

- Select
- Clear the selection
- Show the context menu
- Store the current selections
- Check the selection state

## Create an instance of the selection manager

To use the selection manager, create the instance of a selection manager. Usually, visuals create a selection manager instance in the `constructor` section of the visual object.

TypeScript

```
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's `createSelectionIdBuilder` method generates a selection for each data point. This method returns an instance of the object with interface `powerbi.visuals.ISelectionIdBuilder`:

TypeScript

```
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 corresponding 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, update to API version 2.5.0 or higher.

## Create selections for categorical data view mapping

Let's review how selections represent categorical data view mapping for a sample semantic model:

[Expand table](#)

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
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

Manufacturer	Type	Value
Toyota	Import Truck	23614

The visual uses the following data view mapping:

JSON

```
{
  "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 preceding example, `Manufacturer` is `columns` and `Type` is `rows`. A series is created by grouping values by `rows` (`Type`).

The visual should be able to slice data by `Manufacturer` or `Type`.

For example, if a user selects `Chrysler` by `Manufacturer`, other visuals should show the following data:

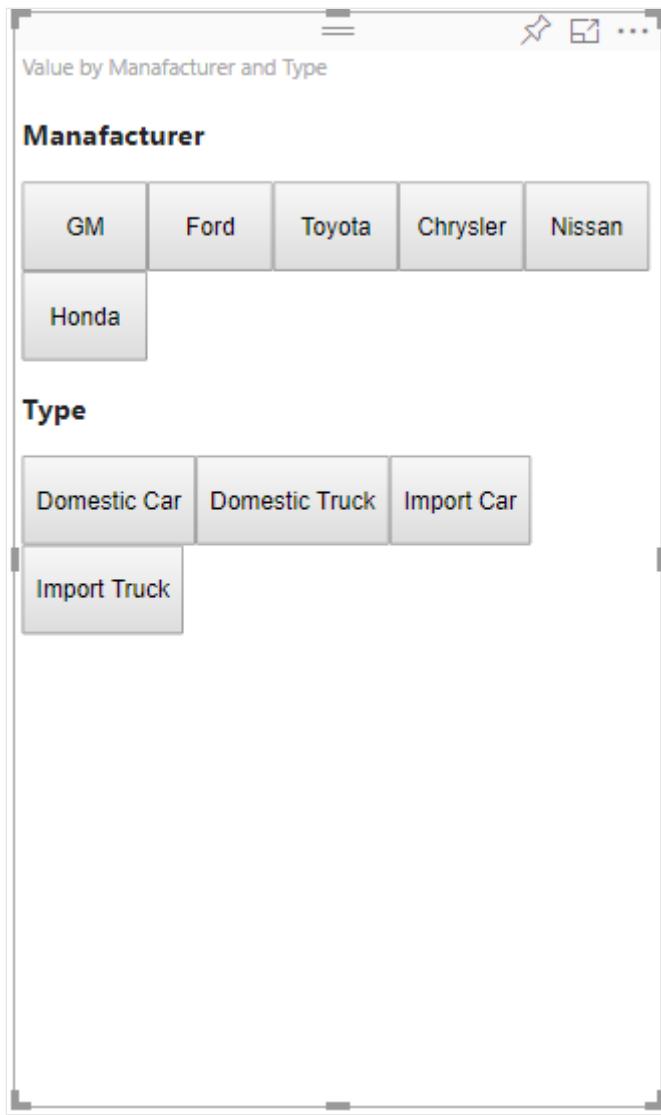
[Expand table](#)

Manufacturer	Type	Value
Chrysler	Domestic Car	28883
Chrysler	Domestic Truck	117131
Chrysler	Import Car	0
Chrysler	Import Truck	6362

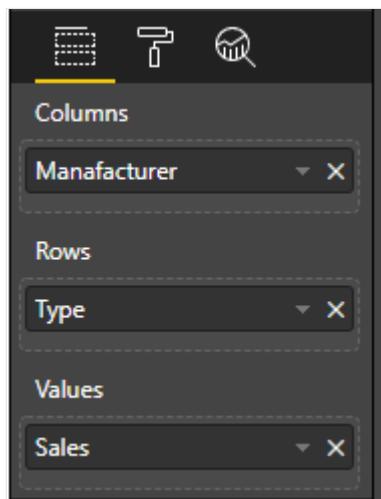
When the user selects `Import Car` by `Type` (selects data by series), the other visuals should show the following data:

[Expand table](#)

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



To display sliced data, fill the visual's data baskets as follows:



In the preceding example, **Manufacturer** is category (columns), **Type** is series (rows), and **Sales** is **Values** for series.

**Note**

Values are required for displaying a series because, according to the data view mapping, Values are grouped by Rows data.

## Create selections for categories

TypeScript

```
// 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 preceding sample code, we iterate through all categories. In each iteration, we call `createSelectionIdBuilder` to create the next selection for each category by calling the

`withCategory` method of the selection builder. The `createSelectionId` method is used as a final method to return the generated `selection` object.

In the `withCategory` method, we pass the column of `category`, in the sample, its `Manufacturer`, and the index of category element.

## Create selections for series

TypeScript

```
// 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

The following example shows table data view mapping:

JSON

```
{
  "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, call the `withTable` method of selection builder.

TypeScript

```
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 the `withTable` table method. Parameters of the `withTable` method are the `table` object and the index of the table row.

## Create selections for matrix data view mapping

TypeScript

```
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` recursively calls each node and child node.

`nodeWalker` creates a `nodeSelection` object on each call. Each `nodeSelection` represents a `selection` of corresponding nodes.

## Select data points to slice other visuals

In this example, we created a click handler for button elements. The handler calls the `select` method of the selection manager and passes the selection object.

TypeScript

```

button.addEventListener("click", () => {
    // handle click event to apply correspond selection
    this.selectionManager.select(categorySelectionId);
});

```

The interface of the `select` method:

TypeScript

```

interface ISelectionManager {
    ...
    select(selectionId: ISelectionId | ISelectionId[], multiSelect?:
boolean): IPromise<ISelectionId[]>;

```

```
// ...  
}
```

The `select` method can accept an array of selections. This allows your visual to have several data points selected at once. The second parameter, `multiSelect`, is responsible for multi-selections. If `multiSelect` is true, Power BI doesn't clear the previous selection state when it applies the current selection. If the value is false, the previous selection is overwritten.

A typical example of using `multiSelect` is handling the `ctrl` button state on a click event. When the `ctrl` button is held down, you can select more than one object.

TypeScript

```
button.addEventListener("click", (mouseEvent) => {  
    const multiSelect = (mouseEvent as MouseEvent).ctrlKey;  
    this.selectionManager.select(seriesSelectionId, multiSelect);  
});
```

## Related content

[Handle selections on bookmarks switching](#)

[Add a context menu for visuals data points](#)

[Add selections into Power BI Visuals](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Add tooltips to your Power BI visuals

Article • 03/29/2025

[ToolTips](#) are an elegant way of providing more contextual information and detail to data points on a visual. The Power BI tooltips API can 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.

You can change the style of your tooltips or add drilling actions by enabling the [modern tooltips](#) feature.

The following image shows a tooltip in a sample bar chart:



The above tooltip image illustrates a single bar category and value. You can extend the tooltip to display multiple values.

# Manage tooltips

You can manage the tooltips in your visual through the `ITooltipService` interface.

`I TooltipService` notifies the host that a tooltip needs to be displayed, removed, or moved.

TypeScript

```
interface ITooltipService {  
    enabled(): boolean;  
    show(options: TooltipShowOptions): void;  
    move(options: TooltipMoveOptions): void;  
    hide(options: TooltipHideOptions): void;  
}
```

Your visual should detect mouse events within the visual and call the `show()`, `move()`, and `hide()` delegates, as needed, with the appropriate content populated in the `Tooltip options` objects. `TooltipShowOptions` and `TooltipHideOptions` in turn define what to display and how to behave in these events.

Calling these methods involves user events such as mouse moves and touch events, so it's a good idea to create listeners for these events, which would in turn invoke the `TooltipService` members. The following example aggregates in a class called `TooltipServiceWrapper`.

## The `TooltipServiceWrapper` class

The basic idea behind this class is to hold the instance of the `TooltipService`, listen for 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.

The example code in this article is based on the [SampleBarChart visual](#). You can examine the source code in [barChart.ts](#).

## Create `TooltipServiceWrapper`

The bar chart constructor now has a `TooltipServiceWrapper` member, which is instantiated in the constructor with the host `tooltipService` instance.

TypeScript

```
    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.

TypeScript

```
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

TypeScript

```
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<Element>:** The d3 elements over which tooltips are handled.
- **getTooltipInfoDelegate: (args: TooltipEventArgs<T>) => VisualTooltipDataItem[]:** The delegate for populating the tooltip content (what to display) per context.
- **getDataPointIdentity: (args: TooltipEventArgs<T>) => 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:

TypeScript

```
...  
...  
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.

TypeScript

```
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 might 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.

TypeScript

```
this.tooltipServiceWrapper.addTooltip(this.barContainer.selectAll('.bar'),
    (tooltipEvent: TooltipEventArgs<number>) =>
BarChart.getTooltipData(tooltipEvent.data),
    (tooltipEvent: TooltipEventArgs<number>) => null);
```

## Add tooltips support to the report page

To add report page tooltips support (the ability to modify tooltips in the format pane of the report page), add a `tooltips` object in the `capabilities.json` file.

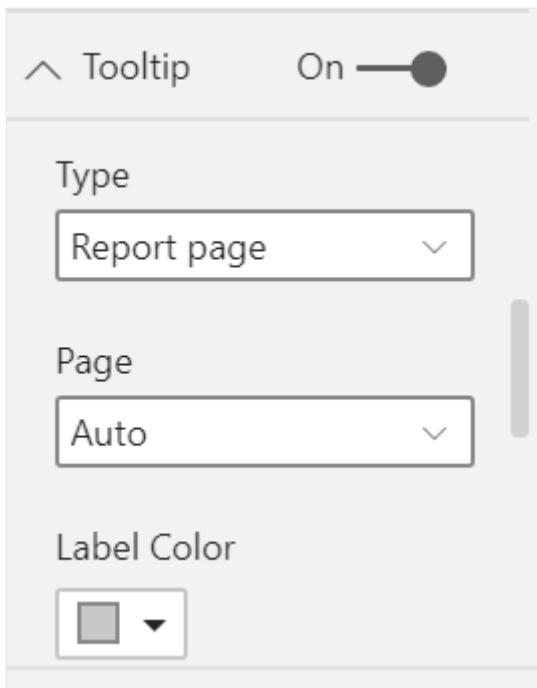
For example:

JSON

```
{
  "tooltips": {
    "supportedTypes": {
      "default": true,
      "canvas": true
    },
    "roles": [
      "tooltips"
    ]
  }
}
```

You can then define the tooltips from the [Formatting pane](#) of the report page.

- `supportedTypes`: The tooltip configuration 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.

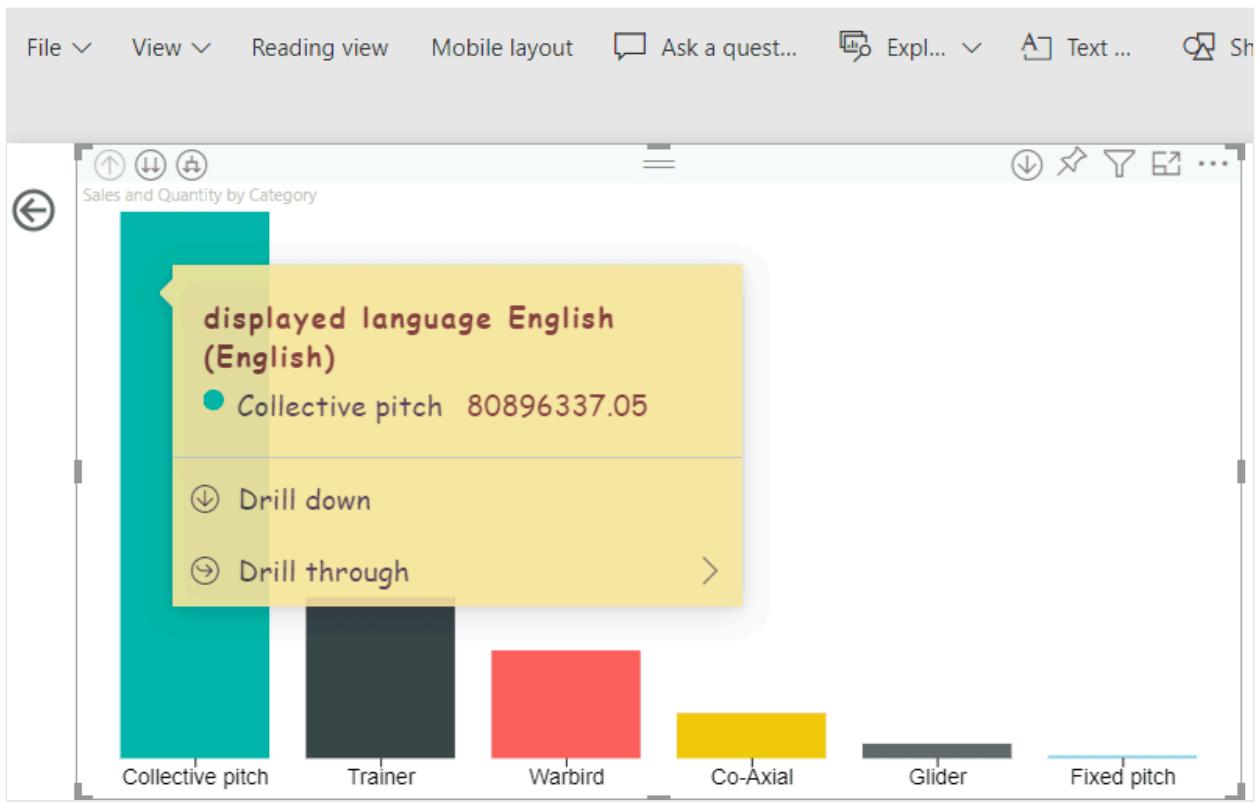
An example of sending the `selectionId` to tooltip display calls is shown in the following code:

TypeScript

```
this.tooltipServiceWrapper.addTooltip(this.barContainer.selectAll('.bar'),
    (tooltipEvent: TooltipEventArgs<number>) =>
    BarChart.getTooltipData(tooltipEvent.data),
    (tooltipEvent: TooltipEventArgs<number>) =>
    tooltipEvent.data.selectionID);
```

## Add modern tooltips support to the report page

From API version 3.8.3 you can also create [modern visual tooltips](#). Modern visual tooltips add data point drill actions to your tooltips, and update the style to match your report theme. To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.



To manage report page modern tooltips support, add the `supportEnhancedTooltips` property to the `tooltips` object in the `capabilities.json` file.

For example:

```
JSON

{
    "tooltips": {
        ...
        "supportEnhancedTooltips": true
    }
}
```

See an example of the modern tooltips feature being used in the [SampleBarChart](#) code.

#### ⓘ Note

Adding this feature to the `capabilities.json` file gives the user the possibility of enabling this feature for the report. Keep in mind that the user will still have to [enable the modern tooltip feature](#) in the report settings.

## Related content

- Tooltip utils
  - Customize tooltips in Power BI
  - Create tooltips based on report pages in Power BI Desktop
- 

## Feedback

Was this page helpful?



Yes



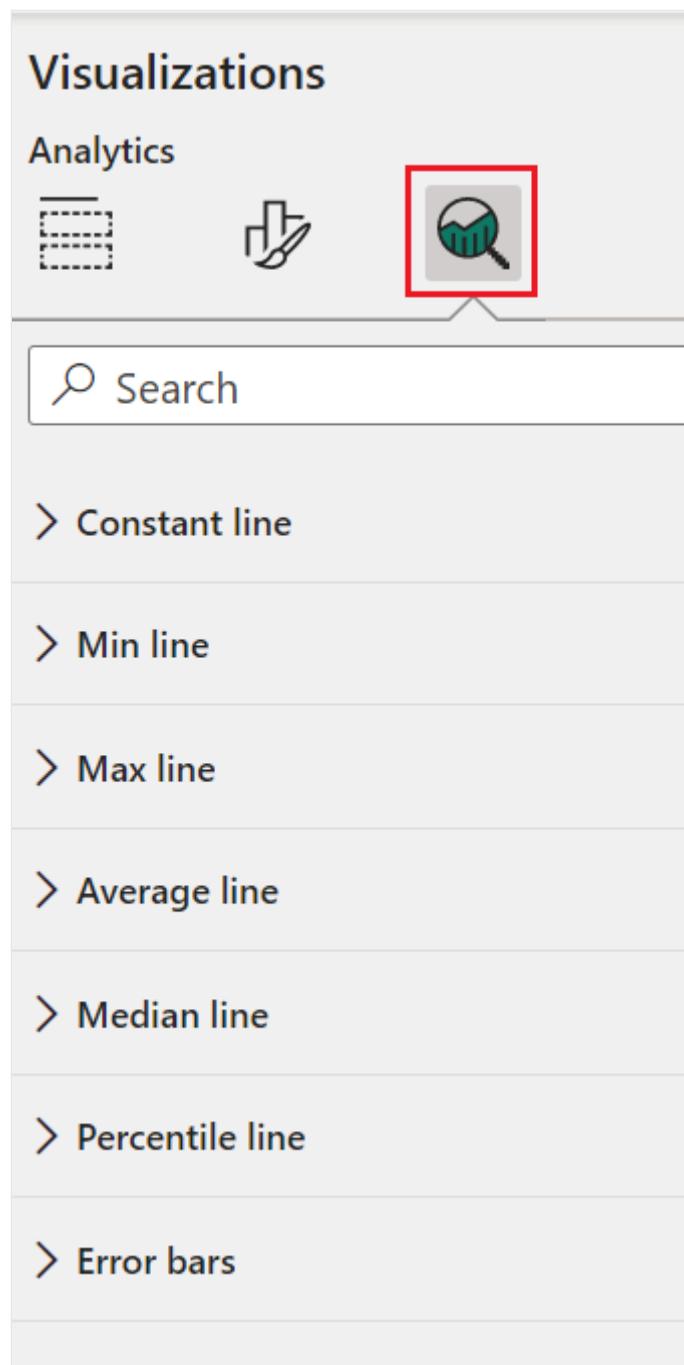
No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# The Analytics pane in Power BI visuals

Article • 01/13/2024

The [Analytics pane](#) allows you to add dynamic reference lines, like min, max, or average, to your visuals. These lines can help you zoom in on important trends or insights. This article discusses how to create Power BI visuals that can present and manage dynamic reference lines in the [Analytics pane](#).



## ① Note

The **Analytics** pane is available in API version 2.5.0. To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.

## Manage the Analytics pane

Managing properties in the **Analytics** pane is similar to the managing properties in the **Format pane**. You define an **object** in the visual's `capabilities.json` file.

For the **Analytics** pane, the object is defined as follows:

API 5.1+

Under the object's definition, add only the object name, property name and type as explained [here](#). Example:

JSON

```
{  
  "objects": {  
    "YourAnalyticsPropertiesCard": {  
      "properties": {  
        "show": {  
          "type": {  
            "bool": true  
          }  
        },  
        "displayName": {  
          "type": {  
            "text": true  
          }  
        },  
        ... //any other properties for your Analytics card  
      }  
    }  
  }  
}
```

In the formatting settings card, specify that this card belongs to the analytics pane by setting the `set card analyticsPane` parameter to `true`. By default, `analyticsPane` parameter is false and the card will be added to formatting pane. See the following implementations:

Using FormattingModel Utils

## TypeScript

```
class YourAnalyticsCardSettings extends FormattingSettingsCard {
    show = new formattingSettings.ToggleSwitch({
        name: "show",
        displayName: undefined,
        value: false,
        topLevelToggle: true
    });

    displayNameProperty = new formattingSettings.TextInput({
        displayName: "displayName",
        name: "displayName",
        placeholder: "",
        value: "Analytics Instance",
    });

    name: string = "YourAnalyticsPropertiesCard";
    displayName: string = "Your analytics properties card's name";
    analyticsPane: boolean = true; // <== Add and set analyticsPane
variable to true
    slices = [this.show, this.displayNameProperty];
}
```

Define other properties the same way that you do for **Format** objects, and enumerate objects just as you do in the **Format** pane.

### ⓘ 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.

## Considerations and limitations

- The **Analytics** pane has no multi-instance support yet. The only **selector** an object can have is *static* (that is, `selector: null`), and Power BI visuals can't have multiple instances of a user-defined card.
- Properties of type `integer` aren't displayed correctly. As a workaround, use type `numeric` instead.

## Related content

[Add conditional formatting](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# The format pane in Power BI custom visuals

Article • 04/29/2025

Using API version 5.1 and later, developers can create visuals with the [Format pane](#). Developers can define the cards and their categories for any property in their custom visual, making it easier for report creators to use these visuals.

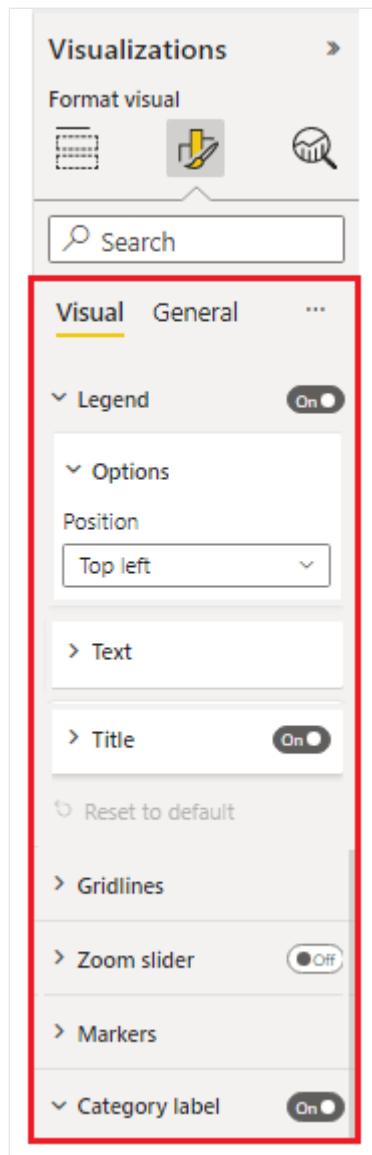
The API uses the **FormattingModel** method to customize parts of the format and analytics panes.

## Tip

The `getFormattingModel` method replaces the `enumerateObjectInstances` method in earlier API versions.

The `getFormattingModel` returns a `FormattingModel` that defines how the visual's formatting and analytics pane look.

In addition to all the old formatting pane capabilities, the [formatting model](#) supports current format pane capabilities, properties, and hierarchies.



## Create a visual that supports the latest Format pane

General steps to add Format pane support to a custom visual:

1. Set the `apiVersion` in your `pbviz.json` file to `5.1` or later.
2. Define all the customizable `objects` in your `capabilities.json` file. These objects are then `mapped` to the properties of the formatting pane. The following properties are required for each object:
  - object name
  - property name
  - property type

All other properties, including `DisplayName` and `description`, are now optional.

3. Build the custom visual `FormattingModel` by doing **one** of the following:

- Use [formattingmodel util](#). (Recommended)
- Without this util, use only APIs.

Define the properties of your custom visual formatting model and build it using code (not JSON).

4. Implement the `getFormattingModel` API in the custom visual class that returns custom visual formatting model. (This API replaces the `enumerateObjectInstances` that was used in previous versions).

## Example of formatting model implementation

- Formatting model using [formattingmodel util example](#). (Recommended)
- Formatting model using only [APIs example](#).

## Map formatting properties

If you have a custom visual created with an older API and you want to update it with the format pane, or if you're creating a new custom visual:

1. Set the `apiVersion` in your `pbviz.json` file to `5.1` or later.
2. For each object name and property name in `capabilities.json`, create a matching formatting property. The formatting property should have a descriptor that contains an `objectName` and `propertyName` that matches the object name and property name in `capabilities.json`.

The `objects` properties in the capabilities file still have the same format and don't need to be changed.

For example, if the `circle` object in your `capabilities.json` file is defined like this:

JSON

```
"objects": {  
    "circle": {  
        "properties": {  
            "circleColor": {  
                "type": {  
                    "fill": {  
                        "solid": {  
                            "color": true  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

```
        },
    }
}
```

The formatting property in your model should be of type `ColorPicker` and look like this:

JavaScript

```
control: {
    type: "ColorPicker",
    properties: {
        descriptor: {
            objectName: "circle",
            propertyName: "circleColor"
        },
        value: {
            value: this.visualSettings.circle.circleColor
        }
    }
}
```

You get an error if one of the following conditions is true:

- The object or property name in the capabilities file doesn't match the one in the formatting model
- The property type in the capabilities file doesn't match the type in formatting model

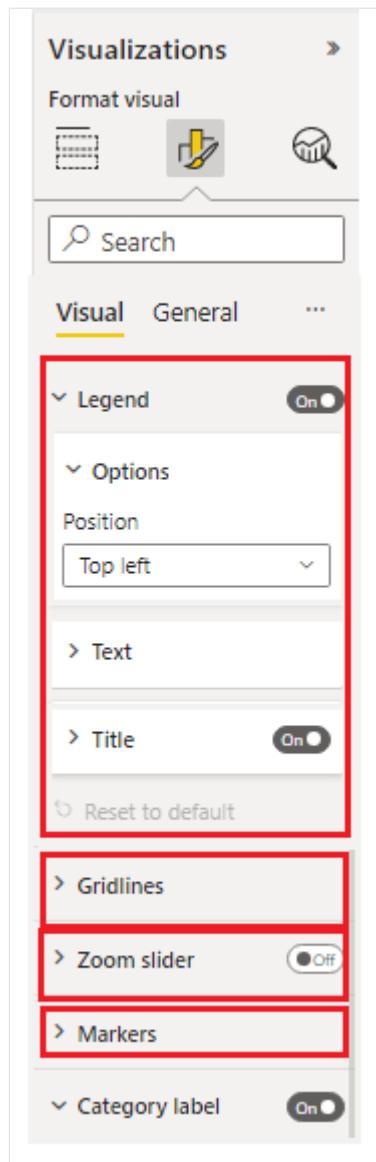
## Formatting model

The formatting model is where you describe and customize all the properties of your format pane.

## Formatting model components

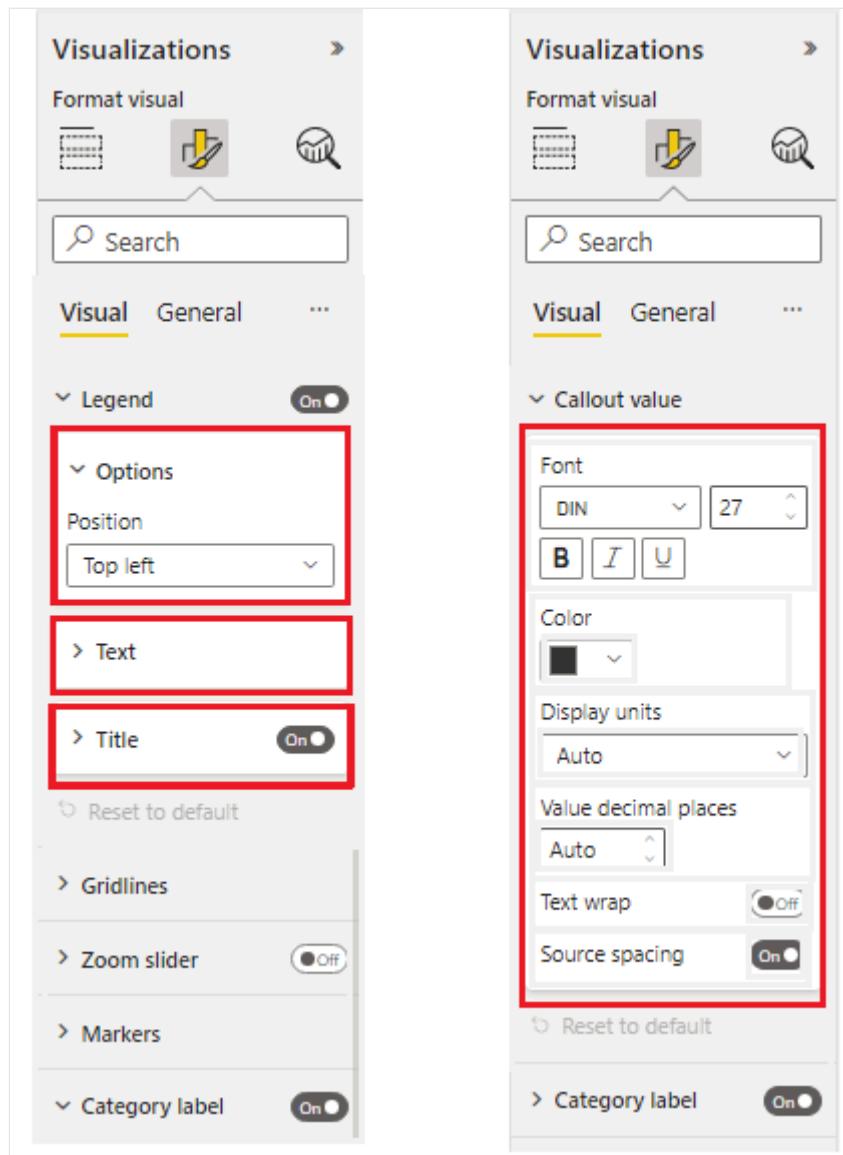
In the formatting model, property components are grouped together in logical categories and subcategories. These groups make the model easier to scan. There are the five basic components, from largest to smallest:

- **Formatting model** The largest pane container, used for formatting the pane's frontal interface. It contains a list of formatting cards.
- **Formatting card** The top level properties grouping container for formatting properties. Each card consists of one or more formatting groups, as shown here.



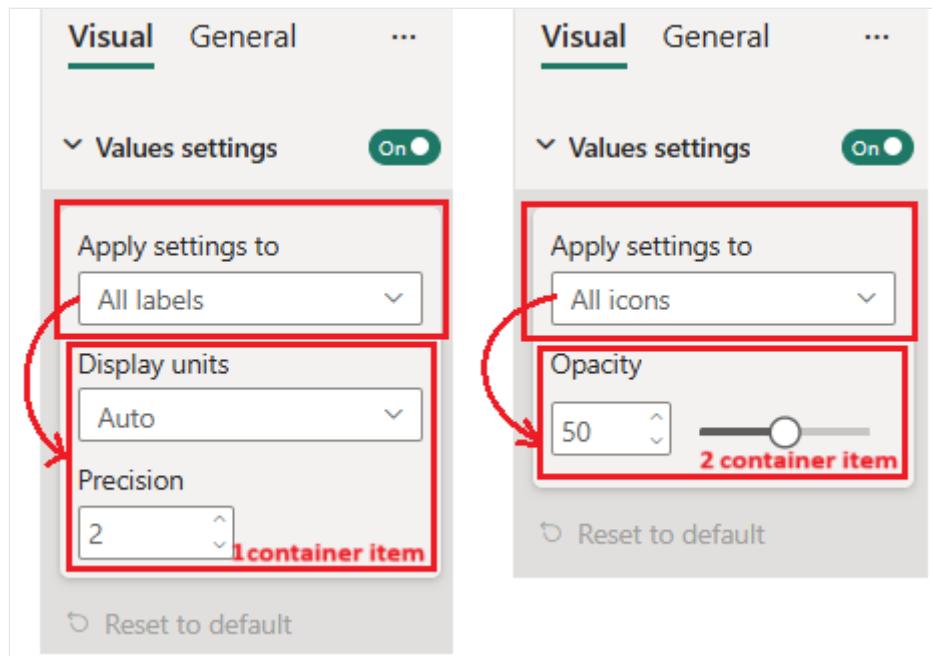
- **Formatting group**

The secondary-level properties grouping container. The formatting group is displayed as a grouping container for formatting slices.



- **Formatting container**

The secondary-level properties grouping container. The formatting container groups formatting slices into separate container items and allows to switch between them using a drop-down list.

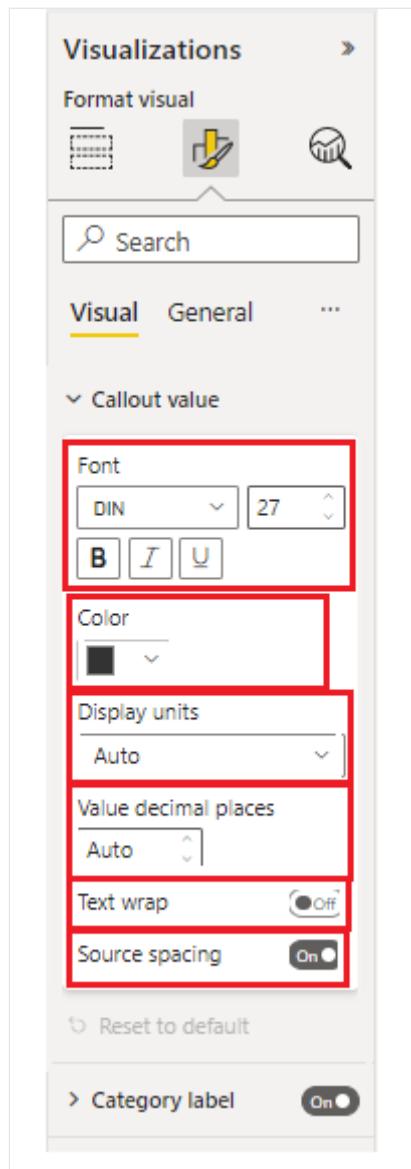


- **Formatting slice**

Property container. There are two types of slices:

- Simple slice: Individual property container
- **Composite slice**: Multiple related property containers grouped into one formatting slice

The following image shows the different types of slices. "Font" is a composite slice consisting of font family, size, and bold, italics, and underline switches. "Color", "display units", and the other slices are simple slices with one component each.



## Visualization pane formatting properties

Every property in the formatting model should match and object type in the *capabilities.json* file.

The following table shows the formatting property types in *capabilities.json* file and their matching type class in modern formatting model properties:

[ ] Expand table

Type	Capabilities Value Type	Formatting Property
Boolean	Bool	ToggleSwitch
Number	<ul style="list-style-type: none"><li>• numeric</li><li>• integer</li></ul>	<ul style="list-style-type: none"><li>• NumUpDown</li><li>• Slider</li></ul>

Type	Capabilities Value Type	Formatting Property
Enumeration list	enumeration:[]	<ul style="list-style-type: none"> <li>• <a href="#">ItemDropdown</a></li> <li>• <a href="#">ItemFlagsSelection</a></li> <li>• <a href="#">AutoDropdown</a></li> <li>• <a href="#">AutoFlagsSelection</a></li> </ul> <p>* See note below</p>
Color	Fill	<a href="#">ColorPicker</a>
Gradient	FillRule	GradientBar: property value should be string consisting of: <code>minValue[,midValue],maxValue</code>
Text	Text	<ul style="list-style-type: none"> <li>• <a href="#">TextInput</a></li> <li>• <a href="#">TextArea</a></li> </ul>

## Capabilities Formatting Objects

 [Expand table](#)

Type	Capabilities Value Type	Formatting Property
Font size	FontSize	NumUpDown
Font family	FontFamily	<a href="#">FontPicker</a>
Line Alignment	Alignment	<a href="#">AlignmentGroup</a>
Label Display Units	LabelDisplayUnits	AutoDropDown

\* The enumeration list formatting property is different in the formatting model and in the capabilities file.

- Declare the following properties in the formatting settings class, including the list of enumeration items:
  - [ItemDropdown](#)
  - [ItemFlagsSelection](#)
- Declare the following properties in the formatting settings class, without the list of enumeration items. Declare their enumeration items list in *capabilities.json* under the appropriate object. (These types are the same as in the previous API versions):
  - [AutoDropdown](#)
  - [AutoFlagSelection](#)

## Composite slice properties

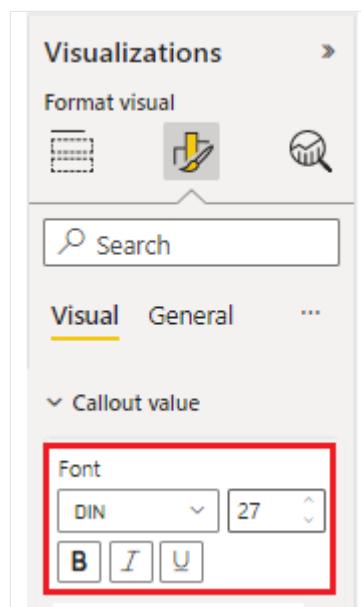
A formatting composite slice is a formatting slice that contains multiple related properties all together.

For now we have two composite slice types:

- [FontControl](#)

This keeps all font related properties together. It consists of the following properties:

- Font Family
- Font Size
- Bold [optional]
- Italic [optional]
- Underline [optional]



Each of these properties should have a corresponding object in capabilities file:

[Expand table](#)

Property	Capabilities Type	Formatting Type
Font Family	Formatting: { fontFamily}	FontPicker
Font Size	Formatting: {fontSize}	NumUpDown
Bold	Bool	ToggleSwitch
Italic	Bool	ToggleSwitch
Underline	Bool	ToggleSwitch

- [MarginPadding](#) Margin padding determines the alignment of the text in the visual. It consists of the following properties:
  - Left

- o Right
- o Top
- o Bottom

Each of these properties should have a corresponding object in capabilities file:

 Expand table

Property	Capabilities Type	Formatting Type
Left	Numeric	NumUpDown
Right	Numeric	NumUpDown
Top	Numeric	NumUpDown
Bottom	Numeric	NumUpDown

## GitHub resources

- All formatting model interfaces can be found in [GitHub - microsoft/powerbi-visuals-api: Power BI custom visuals API ↗](#) in “formatting-model-api.d.ts”
- We recommend using the latest formatting model utils at [GitHub - microsoft/powerbi-visuals-utils-formattingmodel: Power BI visuals formatting model helper utils ↗](#)
- You can find an example of a custom visual *SampleBarChart* that uses API version 5.1.0 and implements `getFormattingModel` using the latest formatting model utils at [GitHub - microsoft/PowerBI-visuals-sampleBarChart: Bar Chart Custom Visual for tutorial ↗](#).

## Related content

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

# Add the new format pane to a custom visual using API

Article • 12/19/2024

## ⓘ Important

To add the new Format pane to a custom visual, we recommend using [formattingmodel utils](#).

Although we recommend using [formattingmodel utils](#) with the new format pane, this example demonstrates how to build a custom visual formatting model with one card using only the API.

## Example: Formatting a data card

The card has two groups:

- **Font control group** with one composite property
  - Font control
- **Data design group** with two simple properties
  - Font color
  - Line alignment

First, add objects to capabilities file:

JSON

```
"objects": {  
    "dataCard": {  
        "properties": {  
            "displayUnitsProperty": {  
                "type": {  
                    "formatting": {  
                        "labelDisplayUnits": true  
                    }  
                }  
            },  
            "fontSize": {  
                "type": {  
                    "formatting": {  
                        "fontSize": true  
                    }  
                }  
            }  
        }  
    }  
}
```

```

        },
        "fontFamily": {
            "type": {
                "formatting": {
                    "fontFamily": true
                }
            }
        },
        "fontBold": {
            "type": {
                "bool": true
            }
        },
        "fontUnderline": {
            "type": {
                "bool": true
            }
        },
        "fontItalic": {
            "type": {
                "bool": true
            }
        },
        "fontColor": {
            "type": {
                "fill": {
                    "solid": {
                        "color": true
                    }
                }
            }
        },
        "lineAlignment": {
            "type": {
                "formatting": {
                    "alignment": true
                }
            }
        }
    }
}

```

Then, create the `getFormattingModel`

TypeScript

```

public getFormattingModel(): powerbi.visuals.FormattingModel {
    // Building data card, We are going to add two formatting groups
    "Font Control Group" and "Data Design Group"
    let dataCard: powerbi.visuals.FormattingCard = {
        description: "Data Card Description",
        displayName: "Data Card",

```

```

        uid: "dataCard_uid",
        groups: []
    }

    // Building formatting group "Font Control Group"
    // Notice that "descriptor" objectName and propertyName should match
capabilities object and property names
    let group1_dataFont: powerbi.visuals.FormattingGroup = {
        displayName: "Font Control Group",
        uid: "dataCard_fontControl_group_uid",
        slices: [
            {
                uid: "dataCard_fontControl_displayUnits_uid",
                displayName: "display units",
                control: {
                    type: powerbi.visuals.FormattingComponent.Dropdown,
                    properties: {
                        descriptor: {
                            objectName: "dataCard",
                            propertyName: "displayUnitsProperty"
                        },
                        value: 0
                    }
                }
            },
            // FontControl slice is composite slice, It means it contain
multiple properties inside it
            {
                uid: "data_font_control_slice_uid",
                displayName: "Font",
                control: {
                    type:
powerbi.visuals.FormattingComponent.FontControl,
                    properties: {
                        fontFamily: {
                            descriptor: {
                                objectName: "dataCard",
                                propertyName: "fontFamily"
                            },
                            value: "wf_standard-font, helvetica, arial,
sans-serif"
                        },
                        fontSize: {
                            descriptor: {
                                objectName: "dataCard",
                                propertyName: "fontSize"
                            },
                            value: 16
                        },
                        bold: {
                            descriptor: {
                                objectName: "dataCard",
                                propertyName: "fontBold"
                            },
                            value: false
                        }
                    }
                }
            }
        ]
    }

```

```

        },
        italic: {
            descriptor: {
                objectName: "dataCard",
                propertyName: "fontItalic"
            },
            value: false
        },
        underline: {
            descriptor: {
                objectName: "dataCard",
                propertyName: "fontUnderline"
            },
            value: false
        }
    }
},
],
};

// Building formatting group "Font Control Group"
// Notice that "descriptor" objectName and propertyName should match
capabilities object and property names
let group2_dataDesign: powerbi.visuals.FormattingGroup = {
    displayName: "Data Design Group",
    uid: "dataCard_dataDesign_group_uid",
    slices: [
        // Adding ColorPicker simple slice for font color
        {
            displayName: "Font Color",
            uid: "dataCard_dataDesign_fontColor_slice",
            control: {
                type:
powerbi.visuals.FormattingComponent.ColorPicker,
                properties: {
                    descriptor:
                    {
                        objectName: "dataCard",
                        propertyName: "fontColor"
                    },
                    value: { value: "#01B8AA" }
                }
            }
        },
        // Adding AlignmentGroup simple slice for line alignment
        {
            displayName: "Line Alignment",
            uid: "dataCard_dataDesign_lineAlignment_slice",
            control: {
                type:
powerbi.visuals.FormattingComponent.AlignmentGroup,
                properties: {
                    descriptor:
                    {
                        objectName: "dataCard",

```

```
        propertyName: "lineAlignment"
    },
    mode:
powerbi.visuals.AlignmentGroupMode.Horizontal,
        value: "right"
    }
},
],
};

// Add formatting groups to data card
dataCard.groups.push(group1_dataFont);
dataCard.groups.push(group2_dataDesign);

// Build and return formatting model with data card
const formattingModel: powerbi.visuals.FormattingModel = { cards:
[dataCard] };
return formattingModel;
}
```

Here's the resulting pane:

# Visualizations

»

## Format visual



Search

### Visual

### General

...

#### ▼ Data Card

##### ▼ Font Control Group

display units

Auto

Font

DIN

▼

16

^

**B**

*I*

U

##### ▼ Data Design Group

Font Color



▼

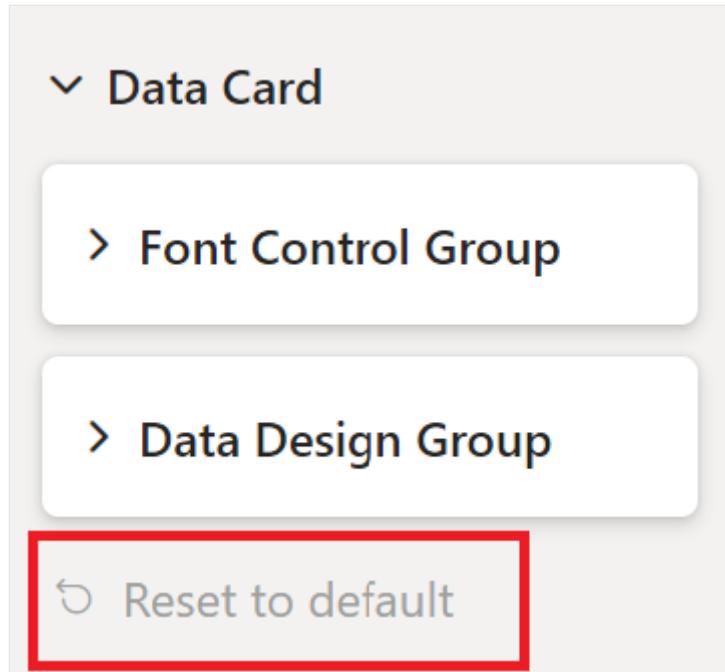
Line Alignment



⟲ Reset to default

# Reset settings to default

The new format pane has the option to reset all formatting card properties values to default by clicking on the *Reset to default* button that appears in the open card.



To enable this feature, add a list of formatting card properties descriptors to formatting card `revertToDefaultDescriptors`. The following example shows how to add the *reset to default* button:

```
TypeScript

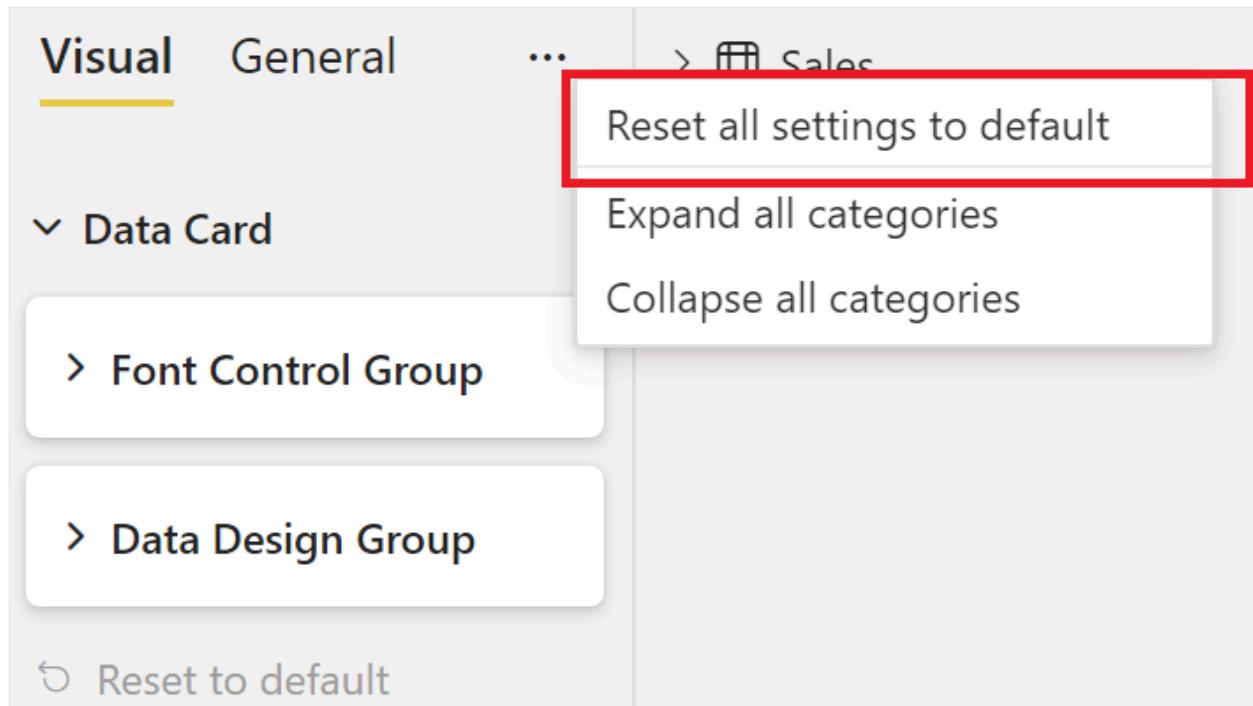
let dataCard: powerbi.visuals.FormattingCard = {
    displayName: "Data Card",
    // ... card parameters and groups list

    revertToDefaultDescriptors: [
        {
            objectName: "dataCard",
            propertyName: "displayUnitsProperty"
        },
        {
            objectName: "dataCard",
            propertyName: "fontFamily"
        },

        // ... the rest of properties descriptors
    ]
};
```

Adding `revertToDefaultDescriptors` to formatting cards also makes it possible to reset all formatting cards properties at once by clicking on the *reset all settings to default*

button in the top bar of the format pane:



## Formatting property selector

The optional selector in formatting properties descriptor determines where each property is bound in the `dataView`. There are four distinct options. Read about them in [objects selectors types](#).

## Localization

For more about the localization feature and to set up a localization environment see [Add the local language to your Power BI visual](#). Use the localization manager to format components that you want to localize:

```
TypeScript

displayName: this.localization.getDisplayName("Font_Color_DisplayNameKey");
description: this.localization.getDisplayName("Font_Color_DescriptionKey");
```

To localize formatting model utils [formatting utils localization](#).

## GitHub Resources

- All formatting model interfaces can be found in [GitHub - microsoft/powerbi-visuals-api: Power BI custom visuals API](#) in "formatting-model-api.d.ts"

- We recommend using the new formatting model utils at GitHub - [microsoft/powerbi-visuals-utils-formattingmodel: Power BI visuals formatting model helper utils](#).
- You can find an example of a custom visual *SampleBarChart* that uses API version 5.1.0 and implements `getFormattingModel` using the new formatting model utils at GitHub - [microsoft/PowerBI-visuals-sampleBarChart: Bar Chart Custom Visual for tutorial](#).

## Related content

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

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

# Formatting settings card

Article • 12/19/2024

A *formatting settings card* specifies a formatting card in the formatting or analytics pane. A formatting settings card can contain multiple formatting slices, containers, groups, and properties.

Adding slices to a formatting settings card puts all of these slices into one formatting card.

Cards, Slices, and Groups can be hidden dynamically by setting the `visible` parameter to *false* (*true* by default).

The card can populate either the formatting pane or analytics pane by setting the `analyticsPane` parameter to *true* or *false*.

## Example: Formatting settings card implementation

### Prerequisites

To build a formatting model with formatting card using `formattingmodel` utils, you need to

- Update `powerbi-visuals-api` version to 5.1 and higher.
- Install `powerbi-visuals-utils-formattingmodel`.
- Initialize [formattingSettingsService](#).
- Initialize [formattingSettingsModel class](#).

#### ⓘ Note

- Card name should match the object name in `capabilities.json`
- Slice name should match the property name in `capabilities.json`

Simple formatting card implementation

In this example, we show how to build a custom visual formatting model with one *simple card* using [formattingmodel util](#). The card has two slices:

- Show property represented by ToggleSwitch slice.
- Display units property represented by AutoDropdown slice.

First, add objects into the `capabilities.json` file:

JSON

```
{
  // ... same level as dataRoles and dataViewMappings
  "objects": {
    "values": {
      "properties": {
        "show": {
          "type": {
            "bool": true
          }
        },
        "displayUnits": {
          "type": {
            "formatting": {
              "labelDisplayUnits": true
            }
          }
        }
      }
    }
  }
}
```

Then, insert the following code fragment into the settings file:

TypeScript

```
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class ValuesCardSetting extends formattingSettings.SimpleCard {
  public show: formattingSettings.ToggleSwitch = new
formattingSettings.ToggleSwitch({
    name: "show",
    value: true
});

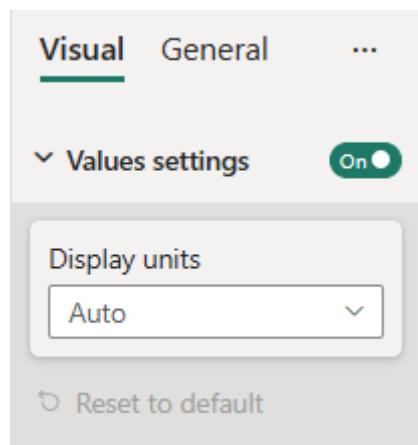
  public displayUnits: formattingSettings.AutoDropdown = new
formattingSettings.AutoDropdown({
    name: "displayUnits",
    displayName: "Display units",
    value: 0
});

  topLevelSlice: formattingSettings.ToggleSwitch = this.show;
  name: string = "values";
}
```

```
    displayName: string = "Values settings";  
  
    public slices: formattingSettings.Slice[] = [ this.displayUnits ];  
}  
  
export class VisualSettingsModel extends formattingSettings.Model {  
    public values: ValuesCardSetting = new ValuesCardSetting();  
    public cards: formattingSettings.SimpleCard[] = [this.values];  
}
```

Follow steps 4 - 8 from the [Build formatting pane](#) tutorial.

Here's the resulting pane:



## Related content

- [Format pane](#)
- [Formatting model utils](#)

## Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Formatting settings group

Article • 12/19/2024

A *formatting settings group* is the secondary-level properties grouping container. Some formatting settings cards can have groups inside. Groups consist of slices and can be expanded/collapsed.

## Formatting settings group implementation

In this example, we show how to build a custom visual formatting model with one [composite card](#) and two *groups* using [formattingmodel util](#). The card has two groups:

- **LabelsSettingsGroup** with two simple properties
  - Precision
  - Display units
- **IconsSettingsGroup** with one simple property
  - Opacity

## Prerequisites

To build a formatting model with composite container using [formattingmodel utils](#), you need to

- Update powerbi-visuals-api version to 5.1 and higher.
- Install powerbi-visuals-utils-formattingmodel.
- Initialize [formattingSettingsService](#).
- Initialize [formattingSettingsModel class](#).

## Example

First, add objects into the `capabilities.json` file:

```
JSON

{
    // ... same level as dataRoles and dataViewMappings
    "objects": {
        "values": {
            "properties": {
                "show": {
                    "type": {
                        "bool": true
                    }
                }
            }
        }
    }
}
```

```

        },
        "displayUnits": {
            "type": {
                "formatting": {
                    "labelDisplayUnits": true
                }
            }
        },
        "precision": {
            "type": {
                "integer": true
            }
        },
        "opacity": {
            "type": {
                "integer": true
            }
        }
    }
},
}
}

```

Then, insert the following code fragment into the settings file:

TypeScript

```

import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class LabelsSettingsGroup extends formattingSettings.SimpleCard {
    public displayUnits: formattingSettings.AutoDropdown = new
formattingSettings.AutoDropdown({
    name: "displayUnits",
    displayName: "Display units",
    value: 0
});

    public precision: formattingSettings.NumUpDown = new
formattingSettings.NumUpDown({
        name: "precision",
        displayName: "Precision",
        value: 2
});

    name: string = "labelsGroup";
    displayName: string = "Labels group";
    collapsible: boolean = false;
    slices: formattingSettings.Slice[] = [this.displayUnits,
this.precision];
}

class IconsSettingsGroup extends formattingSettings.SimpleCard {
    public opacity: formattingSettings.Slider = new

```

```

formattingSettings.Slider({
    name: "opacity",
    displayName: "Opacity",
    value: 50
});

name: string = "iconsGroup";
displayName: string = "Icons group";
slices: formattingSettings.Slice[] = [this.opacity];
}

class ValuesCardSetting extends formattingSettings.CompositeCard {
    public show: formattingSettings.ToggleSwitch = new
formattingSettings.ToggleSwitch({
        name: "show",
        value: true
    });

    public labelsGroup: LabelsSettingsGroup = new LabelsSettingsGroup();
    public iconsGroup: IconsSettingsGroup = new IconsSettingsGroup();

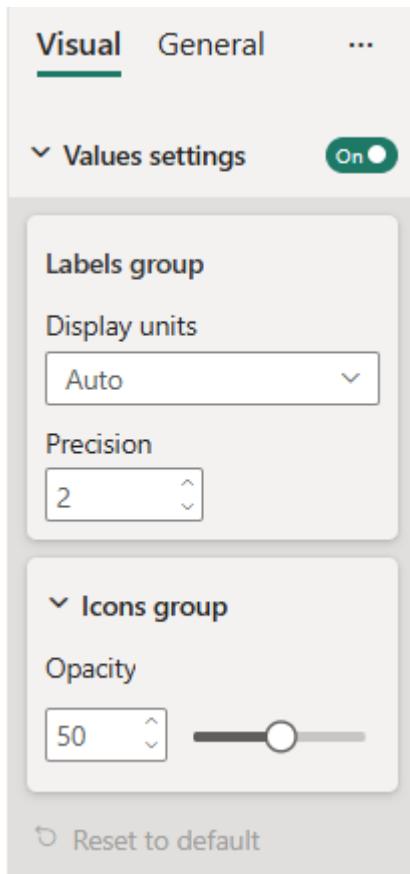
    topLevelSlice: formattingSettings.ToggleSwitch = this.show;
    name: string = "values";
    displayName: string = "Values settings";
    groups: formattingSettings.Group[] = [this.labelsGroup,
this.iconsGroup];
}

export class VisualSettingsModel extends formattingSettings.Model {
    public values: ValuesCardSetting = new ValuesCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.values];
}

```

Follow steps 4 - 8 from the [Build formatting pane](#) tutorial.

Here's the resulting pane:



## Related content

- [Format pane](#)
- [Formatting model utils](#)

## Feedback

Was this page helpful? 👍 Yes 👎 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Formatting settings container

Article • 12/19/2024

A *formatting settings container* is the secondary-level properties grouping container. It groups slices into *container items* and allows users to switch between these items using dropdown element.

## Formatting settings container implementation

In this example, we show how to build a custom visual formatting model with one *container* using [formattingmodel util](#).

The container has two items:

- **LabelsSettingsContainerItem** with two simple properties
  - Precision
  - Display units
- **IconsSettingsContainerItem** with one simple property
  - Opacity

## Prerequisites

To build a formatting model with composite container using [formattingmodel utils](#), you need to

- Update powerbi-visuals-api version to 5.1 and higher.
- Install powerbi-visuals-utils-formattingmodel.
- Initialize [formattingSettingsService](#).
- Initialize [formattingSettingsModel class](#).

## Example

First, add objects into the `capabilities.json` file:

```
JSON

{
    // ... same level as dataRoles and dataViewMappings
    "objects": {
        "values": {
            "properties": {
                "show": {
                    "type": {
                        "label": "Label Settings"
                    }
                }
            }
        }
    }
}
```

```
        "bool": true
    },
},
"displayUnits": {
    "type": {
        "formatting": {
            "labelDisplayUnits": true
        }
    }
},
"precision": {
    "type": {
        "integer": true
    }
},
"opacity": {
    "type": {
        "integer": true
    }
}
}
}
}
```

Then, insert the following code fragment into the settings file:

TypeScript

```
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class LabelsSettingsContainerItem extends formattingSettings.SimpleCard {
    public displayUnits: formattingSettings.AutoDropdown = new
formattingSettings.AutoDropdown({
    name: "displayUnits",
    displayName: "Display units",
    value: 0
});

    public precision: formattingSettings.NumUpDown = new
formattingSettings.NumUpDown({
    name: "precision",
    displayName: "Precision",
    value: 2
});

    name: string = "labelsContainer";
    displayName: string = "All labels";
    slices: formattingSettings.Slice[] = [this.displayUnits,
this.precision];
}

class IconsSettingsContainerItem extends formattingSettings.SimpleCard {
```

```

    public opacity: formattingSettings.Slider = new
formattingSettings.Slider({
    name: "opacity",
    displayName: "Opacity",
    value: 50
});

name: string = "iconsContainer";
displayName: string = "All icons";
slices: formattingSettings.Slice[] = [this.opacity];
}

class ValuesCardSetting extends formattingSettings.SimpleCard {
    public show: formattingSettings.ToggleSwitch = new
formattingSettings.ToggleSwitch({
    name: "show",
    value: true
});

    public labelsContainerItem: LabelsSettingsContainerItem = new
LabelsSettingsContainerItem();
    public iconsContainerItem: IconsSettingsContainerItem = new
IconsSettingsContainerItem();

    public container: formattingSettings.Container = {
        displayName: "Apply settings to",
        containerItems: [this.labelsContainerItem, this.iconsContainerItem]
    };

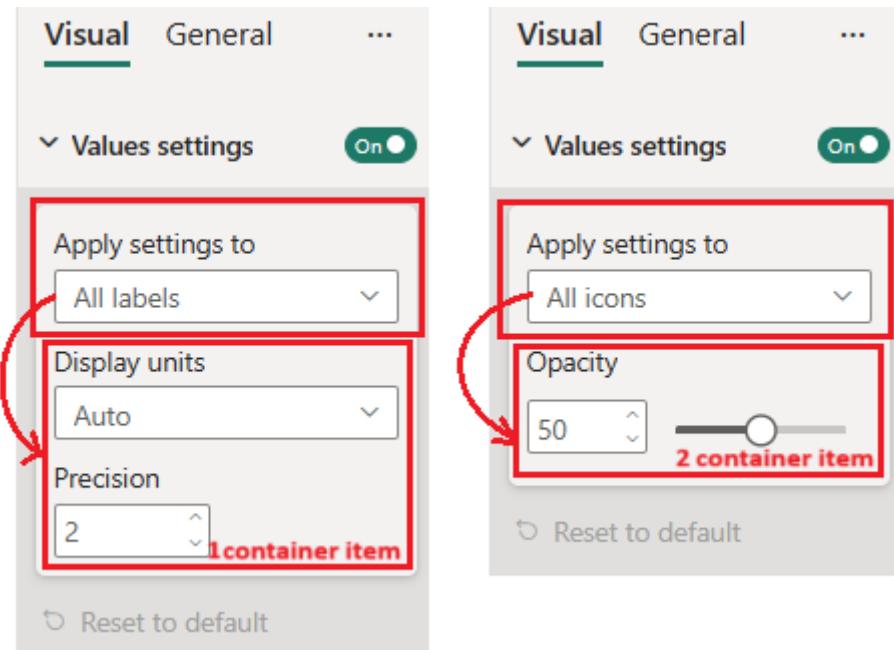
    topLevelSlice: formattingSettings.ToggleSwitch = this.show;
    name: string = "values";
    displayName: string = "Values settings";
}

export class VisualSettingsModel extends formattingSettings.Model {
    public values: ValuesCardSetting = new ValuesCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.values];
}

```

Follow steps 4 - 8 from the [Build formatting pane](#) tutorial.

Here's the resulting pane:



## Related content

- Format pane
- Formatting model utils

## Feedback

Was this page helpful?

Yes

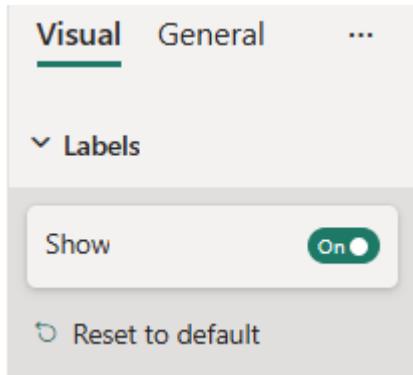
No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# ToggleSwitch formatting slice

Article • 12/19/2024

*ToggleSwitch* is a simple formatting slice which is used to represent *bool* object type from `capabilities.json` file.



## Example: ToggleSwitch implementation

In this example, we show how to build a *ToggleSwitch* slice using formatting model utils.

### Capabilities object

Insert the following JSON fragment into the `capabilities.json` file.

```
JSON

{
    // ... same level as dataRoles and dataViewMappings
    "objects": {
        "labels": {
            "properties": {
                "show": {
                    "type": {
                        "bool": true
                    }
                }
            }
        }
    }
}
```

### Formatting model class

Insert the following code fragment into the settings file.

TypeScript

```
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";

    public showLabels: formattingSettings.ToggleSwitch = new
formattingSettings.ToggleSwitch({
        name: "show", // same as capabilities property name
        displayName: "Show",
        value: true
});

    public slices: formattingSettings.Slice[] = [ this.showLabels ];
}

export class VisualSettings extends formattingSettings.Model {
    public labels: LabelsCardSetting = new LabelsCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.labels];
}
```

## More options

ToggleSwitch slice can also be used as a top-level card toggle.



To make ToggleSwitch top-level, remove slice from the `slices` array and add the following line to the card settings class:

TypeScript

```
topLevelSlice: formattingSettings.ToggleSwitch = this.showLabels;
```

Your final formatting settings file should match this example:

TypeScript

```
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";
```

```
public showLabels: formattingSettings.ToggleSwitch = new
formattingSettings.ToggleSwitch({
    name: "show", // same as capabilities property name
    displayName: "Show",
    value: true
});

topLevelSlice: formattingSettings.ToggleSwitch = this.showLabels;
public slices: formattingSettings.Slice[] = [];
}

export class VisualSettings extends formattingSettings.Model {
    public labels: LabelsCardSetting = new LabelsCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.labels];
}
```

## Related content

- [Format pane](#)
- [Formatting model utils](#)

## Feedback

Was this page helpful?

 Yes

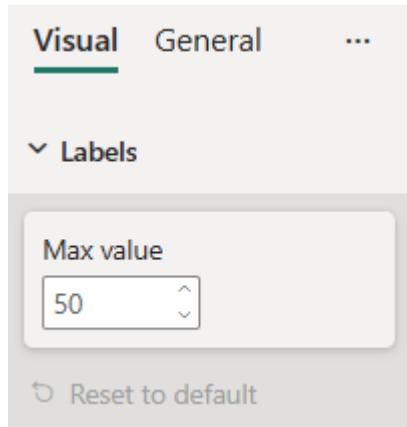
 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# NumUpDown formatting slice

Article • 12/19/2024

*NumUpDown* is a simple formatting slice which is used to represent *numeric* and *integer* object types from `capabilities.json` file.



## Example: NumUpDown implementation

In this example, we show how to build a *NumUpDown* slice using formatting model utils.

### Capabilities object

Insert the following JSON fragment into the `capabilities.json` file.

```
JSON

{
    // ... same level as dataRoles and dataViewMappings
    "objects": {
        "labels": {
            "properties": {
                "max": {
                    "type": {
                        "integer": true
                    }
                }
            }
        }
    }
}
```

### Formatting model class

Insert the following code fragment into the settings file.

```
TypeScript

import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";

    public maxValue : formattingSettings.NumUpDown = new
formattingSettings.NumUpDown({
        name: "max", // same as capabilities property name
        displayName: "Max value",
        value: 50
    });

    public slices: formattingSettings.Slice[] = [ this.maxValue ];
}

export class VisualSettings extends formattingSettings.Model {
    public labels: LabelsCardSetting = new LabelsCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.labels];
}
```

## Validators (optional)

You can validate *NumUpDown* slice input by specifying *options* property as in the example:

```
TypeScript

import powerbi from "powerbi-visuals-api";
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

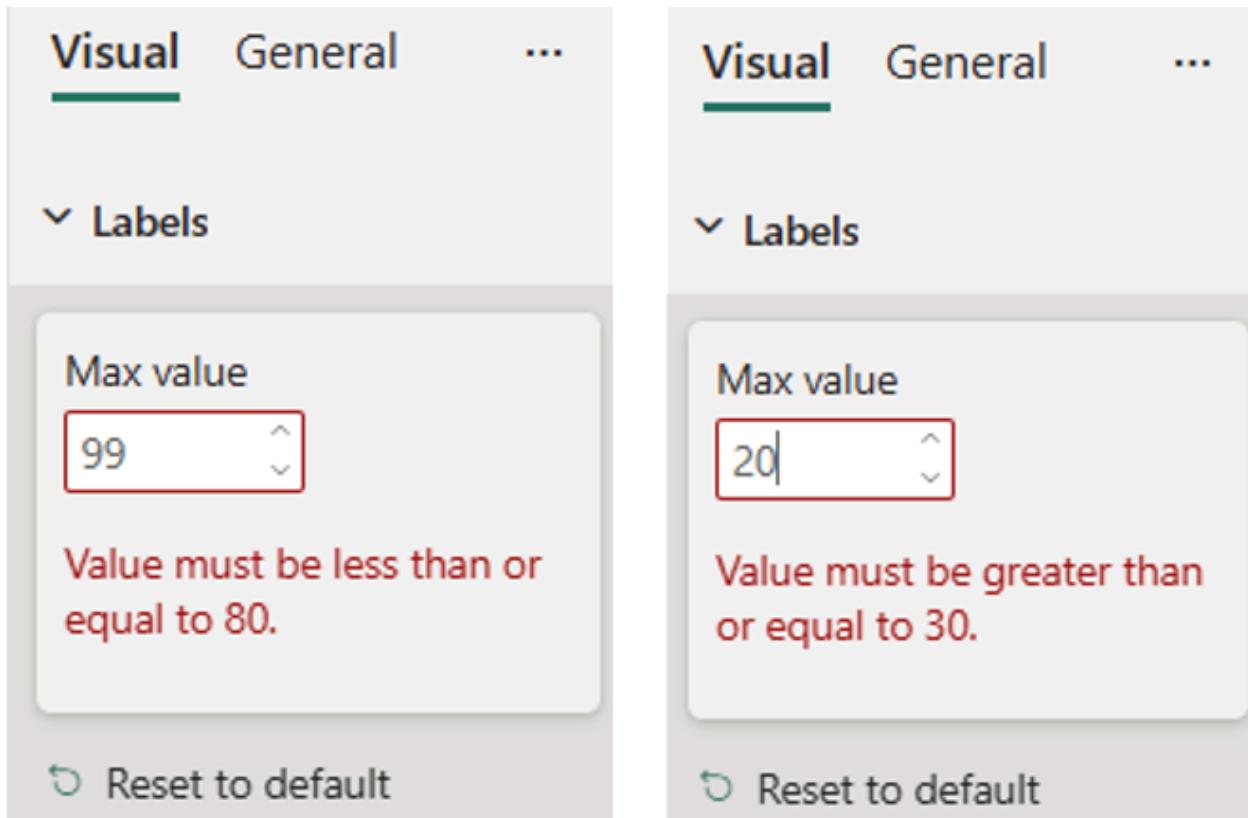
class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";

    public maxValue : formattingSettings.NumUpDown = new
formattingSettings.NumUpDown({
        name: "max", // same as capabilities property name
        displayName: "Max value",
        value: 50, // default slice value
        options: // optional input value validator
        {
            maxValue: {
                type: powerbi.visuals.ValidatorType.Max,
                value: 80
            },
            minValue: {
```

```
        type: powerbi.visuals.ValidatorType.Min,
        value: 30
    }
}

public slices: formattingSettings.Slice[] = [ this maxValue ];
}
```

A warning message is displayed if the passed value is out of the acceptable range.



## Related content

- [Format pane](#)
- [Formatting model utils](#)

## Feedback

Was this page helpful?

Yes

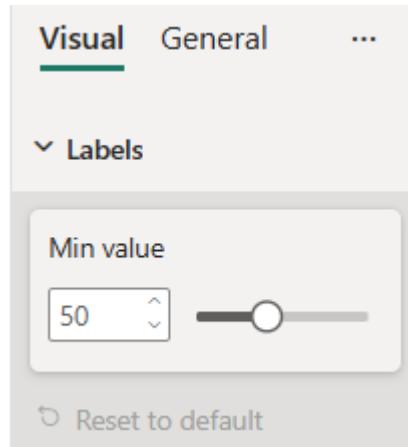
No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Slider formatting slice

Article • 12/19/2024

*Slider* is a simple formatting slice which is used to represent *numeric* and *integer* object types from `capabilities.json` file.



## Example: Slider implementation

In this example, we show how to build a *Slider* slice using formatting model utils.

### Capabilities object

Insert the following JSON fragment into the `capabilities.json` file.

```
JSON

{
    // ... same level as dataRoles and dataViewMappings
    "objects": {
        "labels": {
            "properties": {
                "min": {
                    "type": {
                        "numeric": true
                    }
                }
            }
        }
    }
}
```

### Formatting model class

Insert the following code fragment into the settings file.

```
TypeScript

import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";

    public minValue : formattingSettings.Slider = new
formattingSettings.Slider({
        name: "min", // same as capabilities property name
        displayName: "Min value",
        value: 50
    });

    public slices: formattingSettings.Slice[] = [ this.minValue ];
}

export class VisualSettings extends formattingSettings.Model {
    public labels: LabelsCardSetting = new LabelsCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.labels];
}
```

## Validators (optional)

You can validate *Slider* slice input by specifying *options* property as in the example:

```
TypeScript

import powerbi from "powerbi-visuals-api";
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";

    public minValue : formattingSettings.Slider = new
formattingSettings.Slider({
        name: "min", // same as capabilities property name
        displayName: "Min value",
        value: 50,
        options: // optional input value validator
        {
            maxValue: {
                type: powerbi.visuals.ValidatorType.Max,
                value: 80
            },
            minValue: {
                type: powerbi.visuals.ValidatorType.Min,
```

```
        value: 30
    }
}
});

public slices: formattingSettings.Slice[] = [ this.minLength ];
}
```

## Related content

- [Format pane](#)
  - [Formatting model utils](#)
- 

## Feedback

Was this page helpful?

 Yes

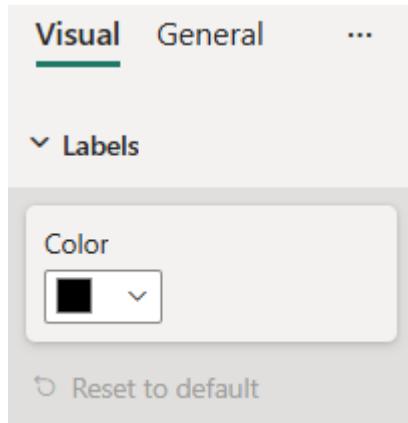
 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# ColorPicker formatting slice

Article • 12/19/2024

*ColorPicker* is a simple formatting slice which is used to represent *fill* object type from `capabilities.json` file.



## Example: ColorPicker implementation

In this example, we show how to build a *ColorPicker* slice using formatting model utils.

### Capabilities object

Insert the following JSON fragment into the `capabilities.json` file.

```
JSON

{
    // ... same level as dataRoles and dataViewMappings
    "objects": {
        "labels": {
            "properties": {
                "color": {
                    "type": {
                        "fill": {
                            "solid": {
                                "color": true
                            }
                        }
                    }
                }
            }
        }
    }
}
```

# Formatting model class

Insert the following code fragment into the settings file.

TypeScript

```
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";

    public color: formattingSettings.ColorPicker = new
formattingSettings.ColorPicker({
        name: "color", // same as capabilities property name
        displayName: "Color",
        value: { value: "#000000" }
    });

    public slices: formattingSettings.Slice[] = [ this.color ];
}

export class VisualSettings extends formattingSettings.Model {
    public labels: LabelsCardSetting = new LabelsCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.labels];
}
```

## Related content

- [Format pane](#)
- [Formatting model utils](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Dropdown formatting slices

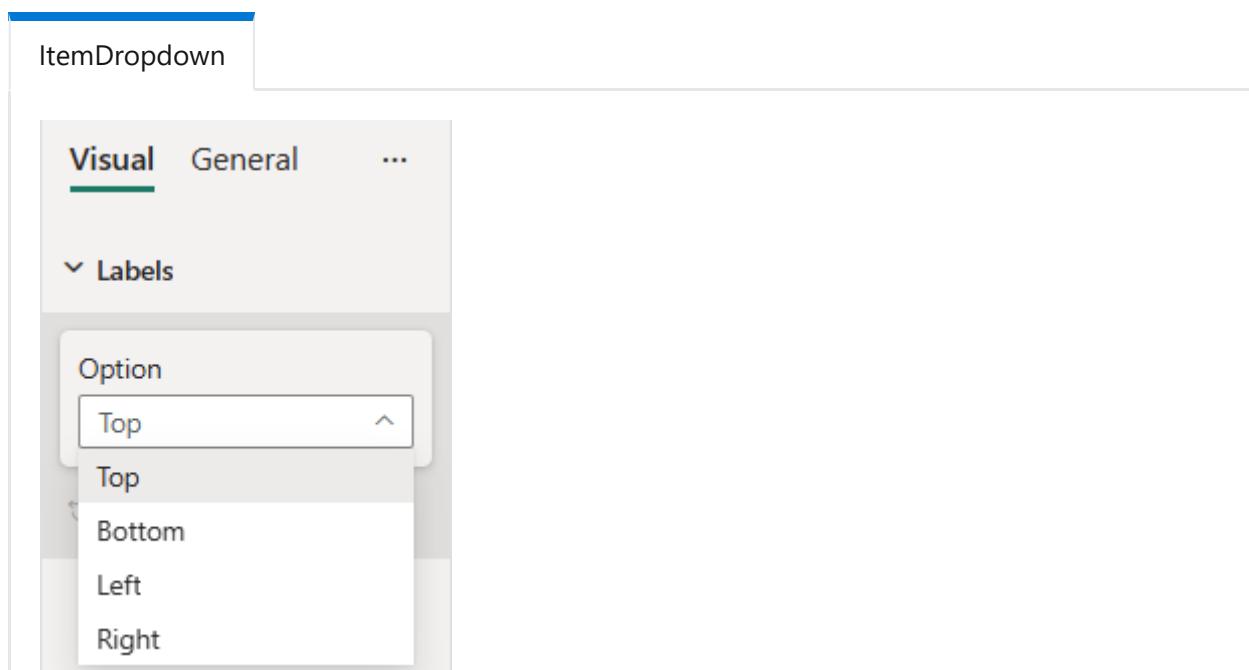
Article • 12/19/2024

*Dropdown* slice is a simple formatting slice which is used to represent *enumeration* object type from `capabilities.json` file. There are two dropdown slices - *ItemDropdown* and *AutoDropdown*.

Their difference is that for *AutoDropdown* slice you need to declare its enumeration items list under the appropriate object in `capabilities.json` file, and for *ItemDropdown* slice in the formatting settings class.

## Example: Dropdown implementation

In this example, we show how to build a *Dropdown* slices using formatting model utils. The following tabs show examples of the *ItemDropdown* and *AutoDropdown* slices.



## Capabilities object

Insert the following JSON fragment into the `capabilities.json` file.

JSON

```
{  
    // ... same level as dataRoles and dataViewMappings  
    "objects": {  
        "labels": {  
            "properties": {
```

```
        "option": {
            "type": {
                "enumeration": []
            }
        },
    }
}
```

## Formatting model class

Insert the following code fragment into the settings file.

TypeScript

```
import powerbi from "powerbi-visuals-api";
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

const positionOptions : powerbi.IEnumMember[] = [
    {value : "top", displayName : "Top"},
    {value : "bottom", displayName : "Bottom"},
    {value : "left", displayName : "Left"},
    {value : "right", displayName : "Right"}
];

class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";

    public option: formattingSettings.ItemDropdown = new
formattingSettings.ItemDropdown({
        name: "option", // same as capabilities property name
        displayName: "Option",
        items: positionOptions,
        value: positionOptions[0]
});

    public slices: formattingSettings.Slice[] = [ this.option ];
}

export class VisualSettings extends formattingSettings.Model {
    public labels: LabelsCardSetting = new LabelsCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.labels];
}
```

## Related content

- Format pane
  - Formatting model utils
- 

## Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Ask the community ↗](#)

# FlagsSelection formatting slices

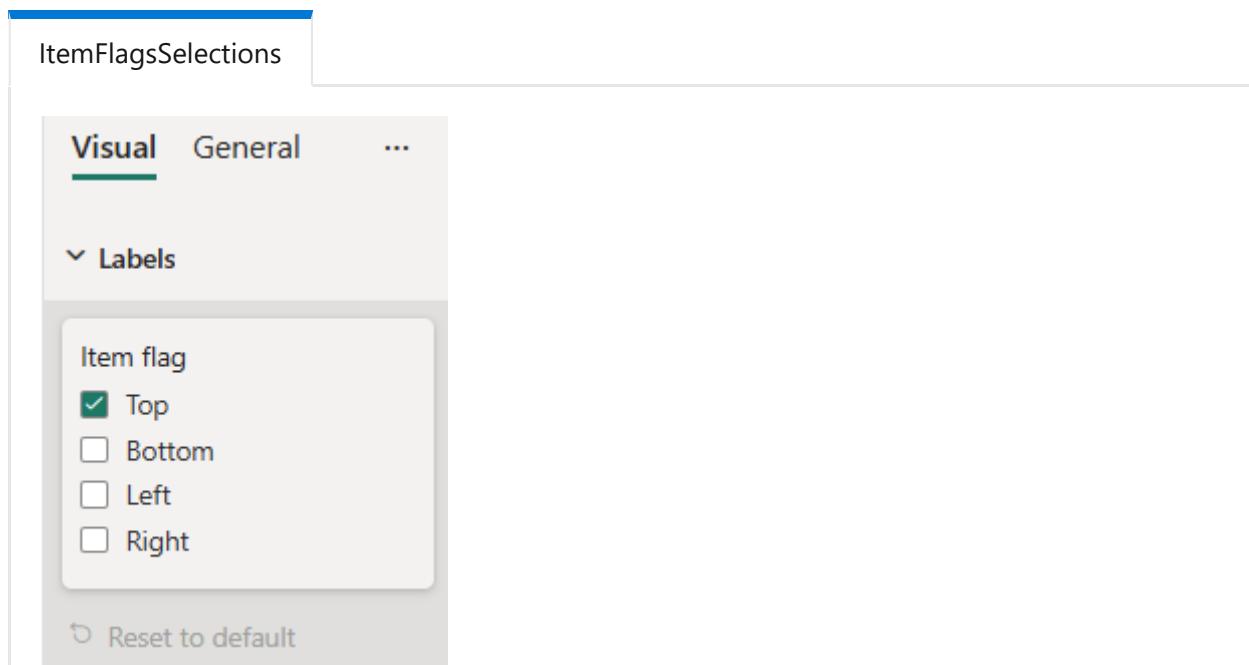
Article • 12/19/2024

*FlagsSelection* slice is a simple formatting slice which is used to represent *enumeration* object type from `capabilities.json` file. There are two FlagSelection slices - `ItemFlagsSelection` and `AutoFlagsSelection`.

Their difference is that for `AutoFlagsSelection` slice you need to declare its enumeration items list under the appropriate object in `capabilities.json` file, and for `ItemFlagsSelection` in the formatting settings class.

## Example: FlagsSelection implementation

In this example, we show how to build a `FlagsSelection` slices using formatting model utils. The following tabs show examples of the `ItemFlagsSelection` and `AutoFlagsSelection` slices.



## Capabilities object

Insert the following JSON fragment into the `capabilities.json` file.

```
JSON

{
    // ... same level as dataRoles and dataViewMappings
    "objects": {
        "labels": {
```

```
        "properties": {
            "itemFlag": {
                "type": {
                    "enumeration": []
                }
            }
        }
    }
}
```

## Formatting model class

Insert the following code fragment into the settings file.

TypeScript

```
import powerbi from "powerbi-visuals-api";
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

const itemFlagOptions : powerbi.IEnumMember[] = [
    {value : "1", displayName : "Top"},
    {value : "2", displayName : "Bottom"},
    {value : "4", displayName : "Left"},
    {value : "8", displayName : "Right"}
];

class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";

    public itemFlag: formattingSettings.ItemFlagsSelection = new
formattingSettings.ItemFlagsSelection({
        name: "itemFlag", // same as capabilities property name
        displayName: "Item flag",
        items: itemFlagOptions,
        value: "1"
});

    public slices: formattingSettings.Slice[] = [ this.itemFlag ];
}

export class VisualSettings extends formattingSettings.Model {
    public labels: LabelsCardSetting = new LabelsCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.labels];
}
```

## Related content

- Format pane
  - Formatting model utils
- 

## Feedback

Was this page helpful?

 Yes

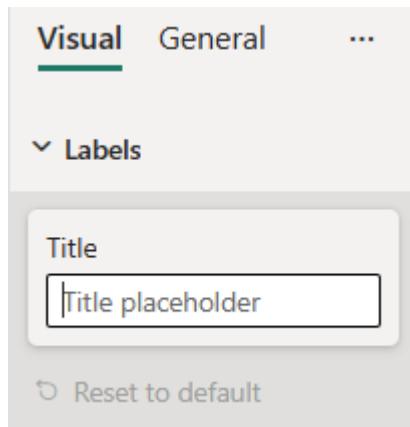
 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# TextInput formatting slice

Article • 12/19/2024

*TextInput* is a simple formatting slice which is used to represent *text* object type from `capabilities.json` file.



## Example: TextInput implementation

In this example, we show how to build a *TextInput* slice using formatting model utils.

### Capabilities object

Insert the following JSON fragment into the `capabilities.json` file.

```
JSON

{
    // ... same level as dataRoles and dataViewMappings
    "objects": {
        "labels": {
            "properties": {
                "title": {
                    "type": {
                        "text": true
                    }
                }
            }
        }
    }
}
```

### Formatting model class

Insert the following code fragment into the settings file.

TypeScript

```
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";

    public title: formattingSettings.TextInput = new
formattingSettings.TextInput({
        name: "title", // same as capabilities property name
        displayName: "Title",
        value: "",
        placeholder: "Title placeholder"
});

    public slices: formattingSettings.Slice[] = [ this.title ];
}

export class VisualSettings extends formattingSettings.Model {
    public labels: LabelsCardSetting = new LabelsCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.labels];
}
```

## Related content

- [Format pane](#)
- [Formatting model utils](#)

## Feedback

Was this page helpful?



Yes



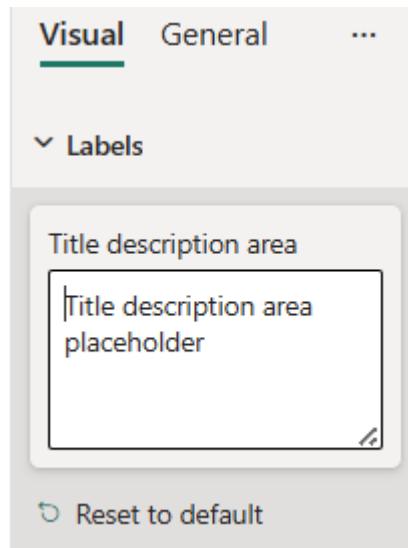
No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# TextArea formatting slice

Article • 12/19/2024

*TextArea* is a simple formatting slice which is used to represent *text* object type from `capabilities.json` file.



## Example: TextArea implementation

In this example, we show how to build a *TextArea* slice using formatting model utils.

## Capabilities object

Insert the following JSON fragment into the `capabilities.json` file.

```
JSON

{
  // ... same level as dataRoles and dataViewMappings
  "objects": {
    "labels": {
      "properties": {
        "titleDescription": {
          "type": {
            "text": true
          }
        }
      }
    }
  }
}
```

# Formatting model class

Insert the following code fragment into the settings file.

TypeScript

```
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";

    public titleDescription: formattingSettings.TextArea = new
formattingSettings.TextArea({
        name: "titleDescription", // same as capabilities property name
        displayName: "Title description area",
        value: "",
        placeholder: "Title description area placeholder"
});

    public slices: formattingSettings.Slice[] = [ this.titleDescription ];
}

export class VisualSettings extends formattingSettings.Model {
    public labels: LabelsCardSetting = new LabelsCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.labels];
}
```

## Related content

- [Format pane](#)
- [Formatting model utils](#)

## Feedback

Was this page helpful?

 Yes

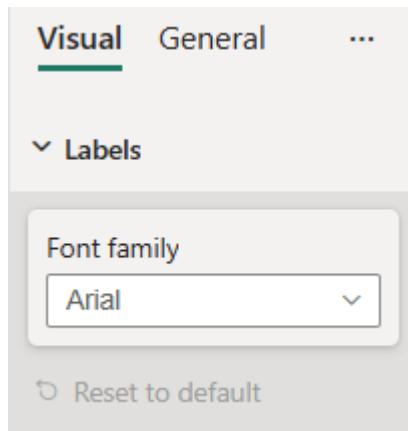
 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# FontPicker formatting slice

Article • 12/19/2024

*FontPicker* is a simple formatting slice which is used to represent *fontFamily* object type from `capabilities.json` file.



## Example: FontPicker implementation

In this example, we show how to build a *FontPicker* slice using formatting model utils.

### Capabilities object

Insert the following JSON fragment into the `capabilities.json` file.

```
JSON

{
    // ... same level as dataRoles and dataViewMappings
    "objects": {
        "labels": {
            "properties": {
                "fontFamily": {
                    "type": {
                        "formatting": {
                            "fontFamily": true
                        }
                    }
                }
            }
        }
    }
}
```

# Formatting model class

Insert the following code fragment into the settings file.

TypeScript

```
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";

    public fontFamily: formattingSettings.FontPicker = new
formattingSettings.FontPicker({
        name: "fontFamily", // same as capabilities property name
        displayName: "Font family",
        value: "Arial, sans-serif"
    });

    public slices: formattingSettings.Slice[] = [ this.fontFamily ];
}

export class VisualSettings extends formattingSettings.Model {
    public labels: LabelsCardSetting = new LabelsCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.labels];
}
```

## Related content

- [Format pane](#)
- [Formatting model utils](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# AlignmentGroup formatting slice

Article • 12/19/2024

*AlignmentGroup* is a simple formatting slice which is used to represent *alignment* object type from `capabilities.json` file.

## Example: AlignmentGroup implementation

In this example, we show how to build a *AlignmentGroup* slice using formatting model utils.

### Capabilities object

Insert the following JSON fragment into the `capabilities.json` file.

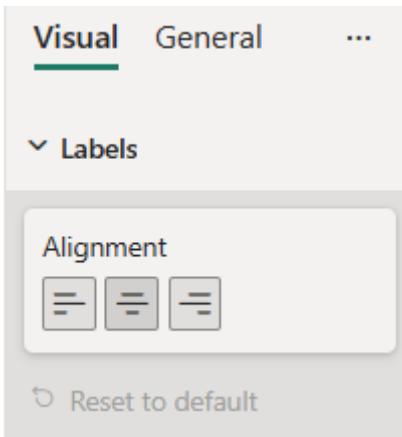
```
JSON

{
    // ... same level as dataRoles and dataViewMappings
    "objects": {
        "labels": {
            "properties": {
                "alignment": {
                    "type": {
                        "formatting": {
                            "alignment": true
                        }
                    }
                }
            }
        }
    }
}
```

### Formatting model class

The following tabs show examples of the same *AlignmentGroup* slice in two available modes.

Horizontal AlignmentGroup



Insert the following code fragment into the settings file.

TypeScript

```
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";

    public labelsAlignment: formattingSettings.AlignmentGroup = new
formattingSettings.AlignmentGroup({
        name: "alignment", // same as capabilities property name
        displayName: "Alignment",
        value: "center", // available values - "center", "left" or
"right"
        mode: powerbi.visuals.AlignmentGroupMode.Horizontal
    });

    public slices: formattingSettings.Slice[] = [ this.labelsAlignment
];
}

export class VisualSettings extends formattingSettings.Model {
    public labels: LabelsCardSetting = new LabelsCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.labels];
}
```

## Related content

- [Format pane](#)
- [Formatting model utils](#)

# Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# FontControl formatting slice

Article • 12/19/2024

*FontControl* is a composite formatting slice that contains font related properties all together. It's used to represent *integer*, *fontControl*, and *bool* object types from `capabilities.json` file.



## Example: FontControl implementation

In this example, we show how to build a *FontControl* slice using formatting model utils.

### Capabilities object

Insert the following JSON fragment into the `capabilities.json` file.

```
JSON

{
    // ... same level as dataRoles and dataViewMappings
    "objects": {
        "labels": {
            "properties": {
                "fontFamily": {
                    "type": {
                        "formatting": {
                            "fontFamily": true
                        }
                    }
                },
                "fontSize": {
                    "type": {
                        "formatting": {
                            "fontSize": true
                        }
                    }
                }
            }
        }
    }
}
```

```
        }
    },
    "bold": {
        "type": {
            "bool": true
        }
    },
    "italic": {
        "type": {
            "bool": true
        }
    },
    "underline": {
        "type": {
            "bool": true
        }
    },
},
}
}
```

## Formatting model class

Insert the following code fragment into the settings file.

TypeScript

```
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";

    public fontFamily: formattingSettings.FontPicker = new
formattingSettings.FontPicker({
        name: "fontFamily", // same as capabilities property name
        value: "Arial, sans-serif"
});

    public fontSize: formattingSettings.NumUpDown = new
formattingSettings.NumUpDown({
        name: "fontSize", // same as capabilities property name
        value: 11
});

    public bold: formattingSettings.ToggleSwitch = new
formattingSettings.ToggleSwitch({
        name: "bold", // same as capabilities property name
        value: false
});
```

```

    public italic: formattingSettings.ToggleSwitch = new
formattingSettings.ToggleSwitch({
    name: "italic", // same as capabilities property name
    value: false
});

    public underline: formattingSettings.ToggleSwitch = new
formattingSettings.ToggleSwitch({
    name: "underline", // same as capabilities property name
    value: false
});

    public font: formattingSettings.FontControl = new
formattingSettings.FontControl({
    name: "font", // must be unique within the same object
    displayName: "Font",
    fontFamily: this.fontFamily,
    fontSize: this.fontSize,
    bold: this.bold, //optional
    italic: this.italic, //optional
    underline: this.underline //optional
});

    public slices: formattingSettings.Slice[] = [ this.font ];
}

export class VisualSettings extends formattingSettings.Model {
    public labels: LabelsCardSetting = new LabelsCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.labels];
}

```

## Related content

- [Format pane](#)
- [Formatting model utils](#)

---

## Feedback

Was this page helpful?

 Yes

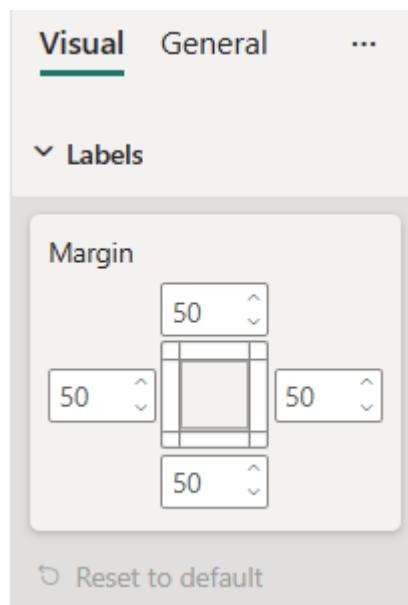
 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# MarginPadding formatting slice

Article • 12/19/2024

*MarginPadding* is a composite formatting slice that contains left, right, top and bottom padding properties all together. It's used to represent *numeric* or *integer* object types from `capabilities.json` file.



## Example: MarginPadding implementation

In this example, we show how to build a *MarginPadding* slice using formatting model utils.

## Capabilities object

Insert the following JSON fragment into the `capabilities.json` file.

```
JSON
{
  // ... same level as dataRoles and dataViewMappings
  "objects": {
    "labels": {
      "properties": {
        "left": {
          "type": {
            "numeric": true
          }
        },
        "right": {
          "type": {
            "numeric": true
          }
        }
      }
    }
  }
}
```

```
        "numeric": true
    }
},
"top": {
    "type": {
        "numeric": true
    }
},
"bottom": {
    "type": {
        "numeric": true
    }
}
}
}
}
```

## Formatting model class

Insert the following code fragment into the settings file.

TypeScript

```
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

class LabelsCardSetting extends formattingSettings.SimpleCard {
    name: string = "labels"; // same as capabilities object name
    displayName: string = "Labels";

    public left : formattingSettings.NumUpDown = new
formattingSettings.NumUpDown({
        name: "left", // same as capabilities property name
        displayName: "Left",
        value: 50
});

    public right : formattingSettings.NumUpDown = new
formattingSettings.NumUpDown({
        name: "right", // same as capabilities property name
        displayName: "Right",
        value: 50
});

    public top : formattingSettings.NumUpDown = new
formattingSettings.NumUpDown({
        name: "top", // same as capabilities property name
        displayName: "Top",
        value: 50
});

    public bottom : formattingSettings.NumUpDown = new
```

```
formattingSettings.NumUpDown({
    name: "bottom", // same as capabilities property name
    displayName: "Bottom",
    value: 50
});

public marginPadding: formattingSettings.MarginPadding = new
formattingSettings.MarginPadding({
    name: "margin", // must be unique within the same object
    displayName: "Margin",
    left: this.left,
    right: this.right,
    top: this.top,
    bottom: this.bottom
});

public slices: formattingSettings.Slice[] = [ this.marginPadding ];
}

export class VisualSettings extends formattingSettings.Model {
    public labels: LabelsCardSetting = new LabelsCardSetting();
    public cards: formattingSettings.SimpleCard[] = [this.labels];
}
```

## Related content

- [Format pane](#)
- [Formatting model utils](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Fetch more data from Power BI

Article • 04/08/2025

The **fetchMoreData** API lets you load data chunks of different sizes as a way of enabling Power BI visuals to bypass the hard limit of a 30K row data view. In addition to the original approach of aggregating all requested chunks, the API now also supports loading data chunks incrementally.

You can configure the number of rows to fetch at a time in advance, or you can use [dataReductionCustomization](#) to allow the report author set the chunk size dynamically.

## (!) Note

The `fetchMoreData` API is available in version 3.4 and above.

The dynamic `dataReductionCustomization` API is available in version 5.2 and above.

To find out which version you're using, check the `apiVersion` in the *pbviz.json* file.

## Enable a segmented fetch of large semantic models

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 you can append to the `dataview` in each update.

For example, add the following code in the *capabilities.json* file to append 100 rows of data at a time:

TypeScript

```
"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.

## Using `fetchMoreData` in the Power BI visual

In Power BI, you can `fetchMoreData` in one of two ways:

- *segments aggregation mode*
- *incremental updates mode*

### Segments aggregation mode (default)

With the segments aggregation mode, the data view that is provided to the visual contains the accumulated data from all previous `fetchMoreData` requests. Therefore, the data view size grows with each update according to the window size. For example, if a total of 100,000 rows are expected, and the window size is set to 10,000, then the first update data view should include 10,000 rows, the second update data view should include 20,000 rows, and so on.

Select the segments aggregation mode by calling `fetchMoreData` with `aggregateSegments = true`.

You can determine whether data exists by checking for the existence of `dataView.metadata.segment`:

TypeScript

```
public update(options: VisualUpdateOptions) {  
    const dataView = options.dataViews[0];  
    console.log(dataView.metadata.segment);  
    // output: __proto__: Object  
}
```

You also can check to see whether the update is 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.

TypeScript

```
// 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 also can invoke the `fetchMoreData` method from a UI event handler:

```
TypeScript

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(true);
        // 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 limits the fetched data total to 100 MB. When this limit is reached, `fetchMoreData()` returns `false`.

## Incremental updates mode

With the incremental updates mode, the data view that is provided to the visual contains only the next set of incremental data. The data view size is equal to the defined window size (or smaller, if the last bit of data is smaller than the window size). For example, if a total of 101,000

rows are expected and the window size is set to 10,000, the visual would get 10 updates with a data view size of 10,000 and one update with a data view of size 1,000.

The incremental updates mode is selected by calling `fetchMoreData` with `aggregateSegments = false`.

You can determine whether data exists by checking for the existence of `dataView.metadata.segment`:

TypeScript

```
public update(options: VisualUpdateOptions) {
    const dataView = options.dataViews[0];
    console.log(dataView.metadata.segment);
    // output: __proto__: Object
}
```

You also can check if the update is the first update or a subsequent update by checking `options.operationKind`. In the following code, `VisualDataChangeOperationKind.Create` refers to the first segment, and `VisualDataChangeOperationKind.Segment` refers to subsequent segments.

TypeScript

```
// 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.Segment) {

    }

    // skip overlapping rows
    const rowOffset = (dataView.table['lastMergeIndex'] === undefined) ? 0 :
dataView.table['lastMergeIndex'] + 1;

    // Process incoming data
    for (var i = rowOffset; i < dataView.table.rows.length; i++) {
        var val = <number>(dataView.table.rows[i][0]); // Pick first column

    }

    // complete update implementation
}
```

You also can invoke the `fetchMoreData` method from a UI event handler:

## TypeScript

```
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(false);
        // 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

Although the data in the different updates of the data views are mostly exclusive, there's some overlap between consecutive data views.

For table and categorical data mapping, the first `N` data view rows can be expected to contain data copied from the previous data view.

```
N is determined by: (dataView.table['lastMergeIndex'] === undefined) ? 0 :
dataView.table['lastMergeIndex'] + 1
```

The visual keeps the data view passed to it so it can access the data without extra communications with Power BI.

## Customized data reduction

Since the developer can't always know in advance what type of data the visual will display, they might want to allow the report author to set the data chunk size dynamically. From API version 5.2, you can allow the report author to set the size of the data chunks that are fetched each time.

To allow the report author to set the count:

1. Define a [formatting object property](#) called `dataReductionCustomization` in your `capabilities.json` file:

JSON

```
"objects": {  
    "dataReductionCustomization": {  
        "properties": {  
            "rowCount": {  
                "type": {  
                    "numeric": true  
                }  
            },  
            "columnCount": {  
                "type": {  
                    "numeric": true  
                }  
            }  
        }  
    },  
},
```

2. Insert the following code fragment into the formatting settings file:

TypeScript

```
class DataReductionCardSettings extends FormattingSettingsCard {  
    rowCount = new formattingSettings.NumUpDown({  
        name: "rowCount",  
        displayName: "Row reduction",  
        description: "Show Reduction for all row groups",  
        value: 100,  
    });  
  
    columnCount = new formattingSettings.NumUpDown({  
        name: "columnCount",  
        displayName: "Column reduction",  
        description: "Show Reduction for all column groups",  
        value: 10,  
    });  
  
    name = "dataReductionCustomization";  
    displayName = "Data Reduction";  
    slices = [this.rowCount, this.columnCount];  
}
```

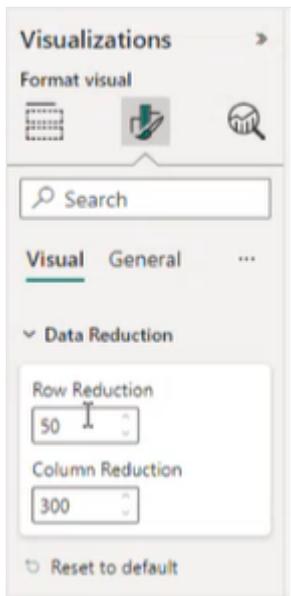
3. After the `dataViewMappings`, define `dataReductionCustomization` as:

JSON

```
"dataReductionCustomization": {  
    "matrix": {  
        "rowCount": {  
            "propertyIdentifier": {
```

```
        "objectName": "dataReductionCustomization",
        "propertyName": "rowCount"
    },
},
"columnCount": {
    "propertyIdentifier": {
        "objectName": "dataReductionCustomization",
        "propertyName": "columnCount"
    }
}
}
```

The data reduction information appears under *visual* in the format pane.



## Considerations and limitations

- The window size is limited to a range of 2-30,000.
- The data view total row count is limited to 1,048,576 rows.
- The data view memory size is limited to 100 MB in segments aggregation mode.
- The *dataReductionCustomization* row count is limited to a range of 15-30,000
- The *dataReductionCustomization* columns count is limited to a range of 1-2,000

## Related content

- [Data view mappings](#)
- [DataViewUtils](#)

# Add bookmark support to visuals in Power BI reports

Article • 04/03/2025

With Power BI report bookmarks, you can capture and save a configured view of a report page. Then you can go back to the saved view quickly and easily whenever you want. The bookmark saves the entire configuration, including selections and filters.

For more information about bookmarks, see [Use bookmarks to share insights and build stories in Power BI](#).

## Visuals that support bookmarks

A Power BI visual that supports bookmarks has to be able to save and provide the correct information when needed. If your visual interacts with other visuals, selects data points, or filters other visuals, you need to save the bookmarked state in the visual's *filterState* properties.

### ⓘ Note

Creating a visual that supports bookmarks requires:

- Visual API version 1.11.0 or later for non-filter visuals that use `SelectionManager`.
- Visual API version 2.6.0 or later for filter visuals.
- To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.

## How Power BI visuals interact with Power BI in report bookmarks

Let's say you want to create several bookmarks on a report page with each bookmark having different data points selected.

First, select one or more data points in your visual. The visual passes your selections to the host. Then select **Add** in the **Bookmark pane**. Power BI saves the current selections for the new bookmark.

Do this several times to create new bookmarks. After you create the bookmarks, you can switch between them.

Each time you select a bookmark, Power BI restores the saved filter or selection state and passes it to the visuals. The visuals in the report are highlighted or filtered according to the state stored in the bookmark. To restore the correct state, your visual must pass the correct selection state to the host (for example, the colors of rendered data points).

The new selection state (or filter) is communicated through the `options.jsonFilters` property in the `update` method. The `jsonFilters` can be either [Advanced Filter](#) or [Tuple Filter](#).

- If your visual contains selected data points, [reset the selection](#) to that of the selected bookmark by using the callback function, `registerOnSelectCallback`, in `ISelectionManager`.
- If your visual uses filters to select data, [reset the filter values](#) to the corresponding values of the selected bookmark.

## Visuals with selection

### ⓘ Note

InteractivityService has been deprecated.

If your visual interacts with other visuals using [Selection](#), you can add bookmark support in one of two ways:

- Through [InteractivityService](#) to manage selections, use the `applySelectionFromFilter`. This is deprecated method.
- Through [SelectionManager](#).

## Use InteractivityService to restore bookmark selections - deprecated

If your visual uses [InteractivityService](#), you don't need any other actions to support the bookmarks in your visual.

When you select a bookmark, the utility automatically handles the visual's selection state.

# Use SelectionManager to restore bookmark selections

You can save and recall bookmark selections using the `ISelectionManager.registerOnSelectCallback` method as follows:

When you select a bookmark, Power BI calls the `callback` method of the visual with the corresponding selections.

TypeScript

```
this.selectionManager.registerOnSelectCallback(  
  (ids: ISelectionId[]) => {  
    //called when a selection was set by Power BI  
  });  
);
```

Let's assume you created a data point in the `visualTransform` method of your visual.

The `datapoints` looks like this:

TypeScript

```
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 then mark the corresponding data points as selected.

TypeScript

```
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 stored in the `filter` object. Then, when the data points are rendered, the custom visual's selection state matches the state of the bookmark.

## 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 used for filtering.

TypeScript

```
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.

In the `update` method, the visual checks the filter in the object:

```
TypeScript

const filter: IAdvancedFilter = FilterManager.restoreFilter(
    && options.jsonFilters
    && options.jsonFilters[0] as any
) as IAdvancedFilter;
```

If the `filter` object isn't null, the visual restores the filter conditions from the object:

```
TypeScript

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 changes its internal state to match the current conditions. The internal state includes the data points and visualization objects (lines, rectangles, and so on).

The [Timeline Slicer](#) visual changes the range selector to the corresponding data ranges.

## Save the filter state of the visual

In addition to saving the conditions of the filter for the bookmark, you also can save other filter aspects.

For example, the [Timeline Slicer](#) stores the `Granularity` property values as a filter state. It allows the granularity of the timeline (days, months, years, etc.) to change as you change bookmarks.

The `filterState` property saves a filter aspect as a property. The visual can store various `filterState` values in bookmarks.

To save a property value as a filter state, set the object property as `"filterState": true` in the `capabilities.json` file.

## Related content

- [What are bookmarks](#)
  - [Create bookmarks in desktop reports](#)
- 

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

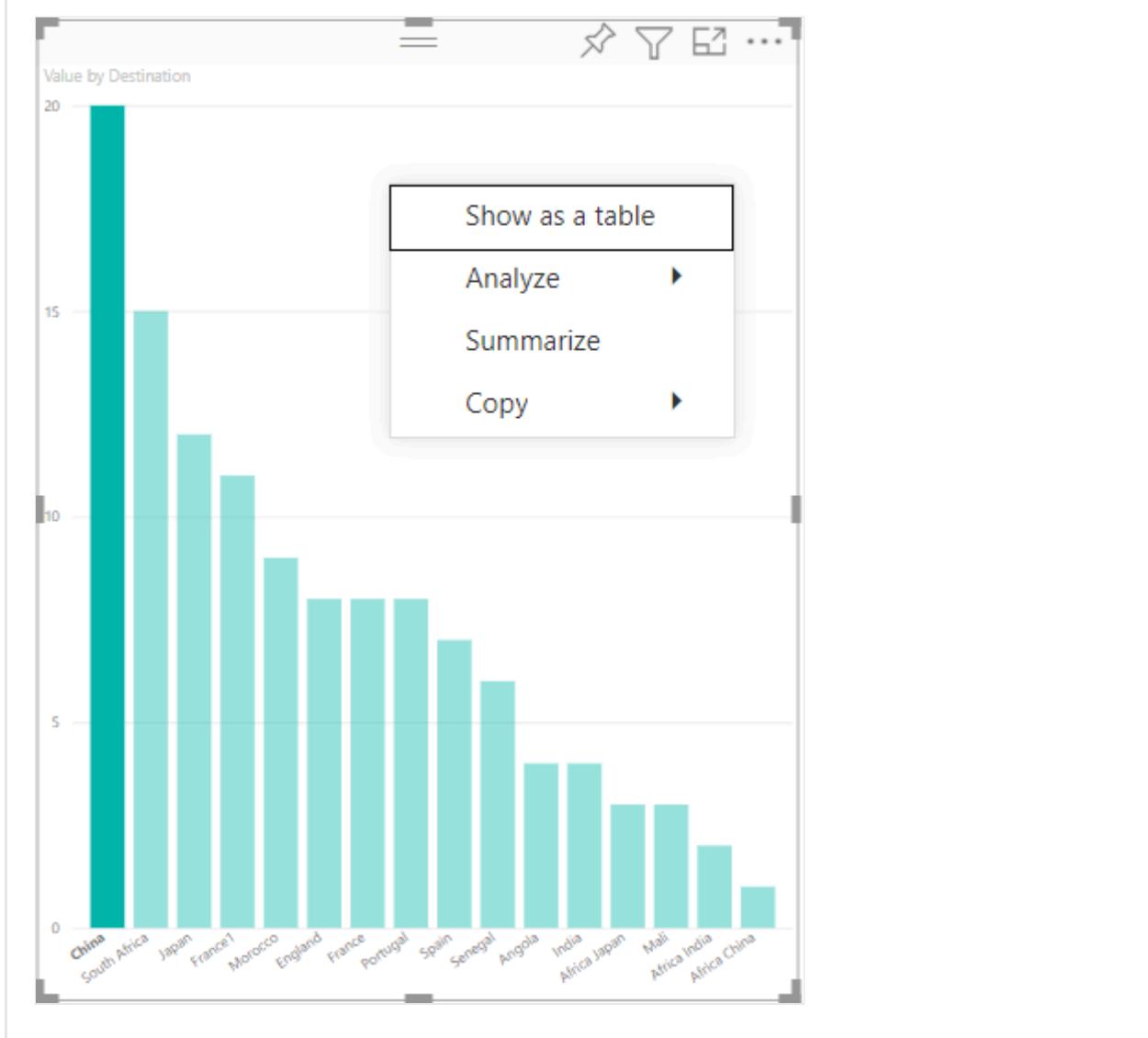
# Add a context menu to your Power BI Visual

Article • 07/22/2024

Every Power BI visual can display a context menu. The context menu allows you to perform various operations on the visual, such as analyzing, summarizing, or copying it. When you right-click anywhere inside a visual's viewport (or long-press for touch devices), the context menu displays. There are two modes of context menus for each visual. The mode that displays depends on where you click inside the visual:

- Call the context menu on **empty space** to see the basic context menu for the visual.
- Call the context menu on a specific **data point** for added options that can be applied to that data point. In this case, the context menu also contains the options *Show data point as a table*, *Include*, and *Exclude*, which will apply the corresponding filter to that data point.

Context menu on empty space



To have Power BI display a context menu for your visual, use

```
selectionManager.showContextMenu() with parameters selectionId and a position (as an {x:, y:} object).
```

### ⓘ Note

- The `selectionManager.showContextMenu()` is available from Visuals API version 2.2.0. To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.
- All visuals published to AppSource must support both `ContextMenu` modes (empty space and data point).

The following example shows how to add a context menu to a visual. The code is taken from the `barChart.ts` file, which is part of the [sample BarChart visual](#):

TypeScript

```
constructor(options: VisualConstructorOptions) {
    ...
    this.handleContextMenu();
}

private handleContextMenu() {
    this.rootSelection.on('contextmenu', (event: PointerEvent,
dataPoint) => {
        this.selectionManager.showContextMenu(dataPoint ? dataPoint: {},
{
            x: mouseEvent.clientX,
            y: mouseEvent.clientY
        });
        mouseEvent.preventDefault();
    });
}
```

## Related content

- Add interactivity into visual by Power BI visuals selections
- Build a bar chart

More questions? [Ask the Power BI Community ↗](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Power BI custom visual dynamic format API

Article • 05/12/2024

From API version 4.2, developers can create reports with [dynamic string formats](#) support.

## Enable the dynamic format support for visual

To allow the visual to operate with dynamic format strings, the following fields should be added in the `capabilities.json`:

```
JSON

{
  "objects": {
    "general": {
      "properties": {
        "formatString": {
          "type": {
            "formatting": {
              "formatString": true
            }
          }
        }
      }
    },
    "type": "string"
  }
}
```

## How to use the dynamic string format

When dynamic string format is enabled, the custom visual receives format strings through the `update()` options as shown in the screenshot.

```

Options...:
▼ {viewport: {...}, dataViews: Array(1), viewMode: 1, editMode: 0, isInFocus: false, ...} ⓘ
  ▼ dataViews: Array(1)
    ▼ 0:
      ▼ categorical:
        ▶ categories: [...]
        ▼ values: Array(1)
          ▼ 0:
            maxLocal: 12519995905.9842
            minLocal: 85266901.4183
            ▼ objects: Array(8)
              ▼ 0:
                ▶ general: {formatString: '¥ #,0'}
                ▶ [[Prototype]]: Object
              ▼ 1:
                ▶ general: {formatString: 'kr#,0'}
                ▶ [[Prototype]]: Object
              ▼ 2:
                ▶ general: {formatString: 'kr#,0'}
                ▶ [[Prototype]]: Object
              ▼ 3:
                ▶ general: {formatString: 'AU$#,0.00'}
                ▶ [[Prototype]]: Object
              ▼ 4:
                ▶ general: {formatString: 'C$#,0.00'}
                ▶ [[Prototype]]: Object
              ▼ 5:
                ▶ general: {formatString: 'CHF#,0.00'}
                ▶ [[Prototype]]: Object
              ▼ 6:
                ▶ general: {formatString: '€ #,0.00'}
                ▶ [[Prototype]]: Object
              ▼ 7:
                ▶ general: {formatString: '£ #,0'}
                ▶ [[Prototype]]: Object
              length: 8
            ▶ [[Prototype]]: Array(0)
        ▶ source: {roles: {...}, type: {...}, displayName: 'Converted Sales Amount', query
        ▶ values: (8) [12519995905.9842, 1005944459.4294, 727632853.9593, 153055826.69

```

There are two arrays inside `options.dataViews[0].categorical.values[0]`:

1. `values` - values from the dataset
2. `objects` - objects with `general.formatString` property

Each `object` corresponds to a `value`. As an example value `12519995905.9842` from the screenshot has the format `¥ #,0`. To apply this format to the value, you can use the [format method](#) from [powerbi-visuals-utils-formattingutils](#).

## Feedback

Was this page helpful?

Yes

No

[Provide product feedback](#) | [Ask the community](#)

# Add drill-down support

Article • 10/12/2024

When a visual has a hierarchy, you can allow users to use the Power BI drill-down feature to reveal more details.

Read more about the Power BI drill-down feature at [Drill mode in the Power BI service](#). To allow the visual to enable or disable the drill feature dynamically, see [Dynamic drill-down control](#).

## Enable drill-down support in the visual

To support drill-down actions in your visual, add a new field to `capabilities.json` named `drill-down`. This field has one property called `roles` that contains the name of the dataRole you want to enable drill-down actions on.

JSON

```
"drilldown": {  
    "roles": [  
        "category"  
    ]  
}
```

### ⓘ 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 the drill-down field, users can drag multiple fields into the data role.

For example:

JSON

```
{  
    "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 a visual with drill-down support

To create a visual with drill-down support, run the following command:

```
Windows Command Prompt
```

```
pbiviz new testDrillDown -t default
```

To create a default sample visual, 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:

```
TypeScript
```

```
"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
        // ...
    }

    /**
     * Returns properties pane formatting model content hierarchies,
     properties and latest formatting values, Then populate properties pane.
     * This method is called once each time we open the properties pane or
     when the user edits any format property.
     */
    public getFormattingModel(): powerbi.visuals.FormattingModel {
        return
    this.formattingSettingsService.buildFormattingModel(this.formattingSettings)
;
    }
}

```

Update the constructor of the visual:

TypeScript

```

export class Visual implements IVisual {
    // visual properties
    // ...
    private div: HTMLDivElement;

    constructor(options: VisualConstructorOptions) {
        console.log('Visual constructor', options);
        this.formattingSettingsService = new FormattingSettingsService();
        this.target = options.element;
        this.updateCount = 0;

        if (document) {
            const new_p: HTMLElement = document.createElement("p");
            new_p.appendChild(document.createTextNode("Update count:"));
            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.div = document.createElement("div"); // <== CREATE DIV
ELEMENT
        this.target.appendChild(new_p);
    }
}
}

```

To create `buttons`, update the `update` visual's method:

TypeScript

```

export class Visual implements IVisual {
    // ...

    public update(options: VisualUpdateOptions) {
        this.formattingSettings =
this.formattingSettingsService.populateFormattingSettingsModel(VisualFormat-
tingSettingsModel, options.dataViews);
        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`:

less

```

button {
    margin: 5px;
    min-width: 50px;
    min-height: 50px;
}

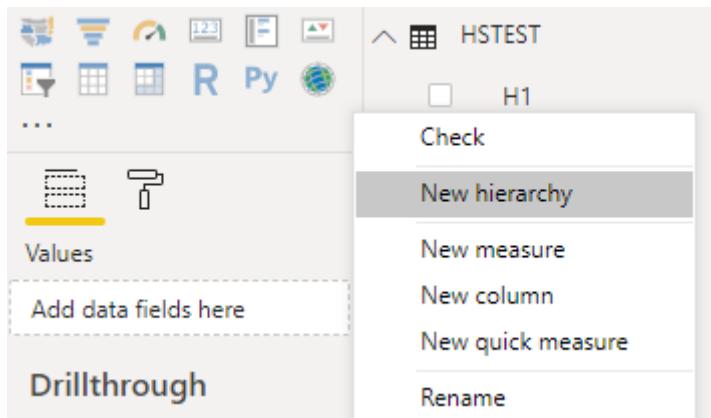
```

Prepare sample data for testing the visual:

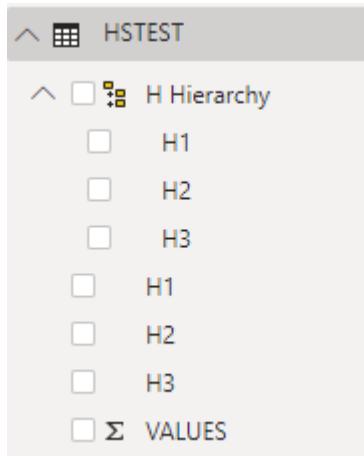
[\[+\]](#) Expand table

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

And create Hierarchy in Power BI Desktop:



Include all category columns (H1, H2, H3) to the new hierarchy:



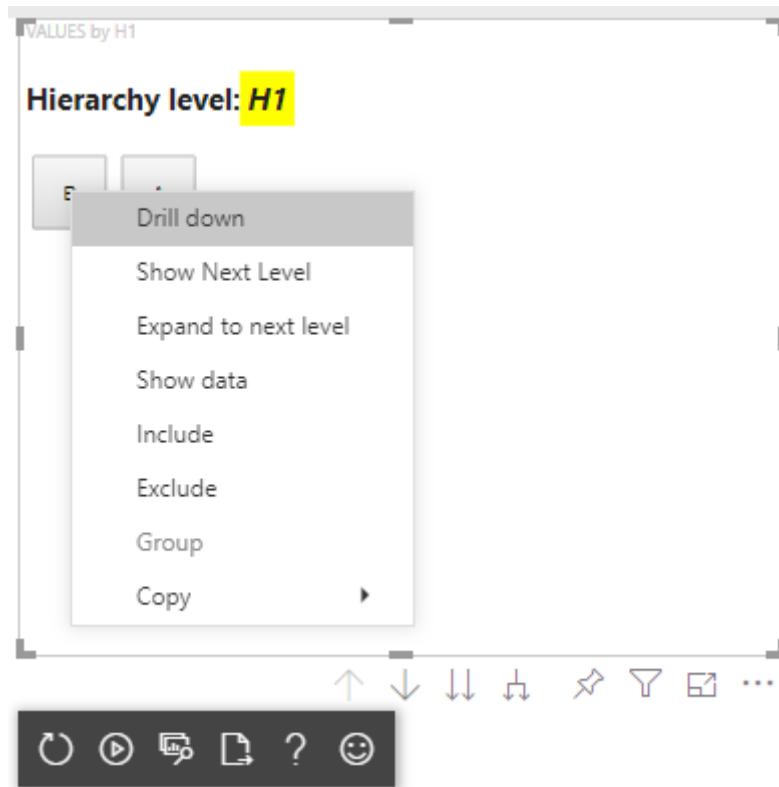
After those steps you should get following visual:

The screenshot shows a Power BI visual titled 'VALUES by H1'. The 'Hierarchy level' is set to 'H1'. There are two buttons labeled 'B' and 'A' on the left. On the right is a data table with four columns: H1, H2, H3, and VALUES. The data is as follows:

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

To add a context menu to the buttons on the visual:



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.

#### TypeScript

```
"use strict";

import "core-js/stable";
import "./../style/visual.less";
// default imports

import IVisualHost = powerbi.extensibility.visual.IVisualHost;
import ISelectionManager = powerbi.extensibility.ISelectionManager;
import ISelectionId = powerbi.visuals.ISelectionId;

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 visual 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:

TypeScript

```

categorical DataView.categories[categorical DataView.categories.length - 1].values.forEach( (category: powerbi.PrimitiveValue, index: number) => {
    // create selectionID for each category value
    let selectionID: ISelectionId = this.host.createSelectionIdBuilder()
        .withCategory(categorical DataView.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 pane. On the left, under 'Category', there is a section for 'H Hierarchy' which contains 'H1', 'H2', and 'H3'. Below it, under 'Value', there is a section for 'VALUES'. On the right, a tree view shows a node 'H Hierarchy' expanded, with its children 'H1', 'H2', and 'H3' listed. A checkmark is next to 'Σ VALUES'.

In the final step you should get visual with selections and context menu:

The screenshot shows a Power BI visual titled 'VALUES by H1'. It is a matrix table with four 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 sums up to 78. The 'H1' column is highlighted with a yellow background. The context menu is open over the 'H1' column header, showing options like 'B' and 'A'. To the right, there are sections for 'Filters' (with 'H1 is (All)', 'H2 is (All)', 'H3 is (All)', and 'VALUES is (All)'), 'Search', and 'Add data fields here'. Below the visual, there are sections for 'Filters on this page' and 'Add data fields here'.

## Add drill-down support for matrix data view mapping

To test the visual with matrix data view mappings, first prepare sample data:

[...] Expand table

Row 1	Row 2	Row 3	Column 1	Column 2	Column 3	Values
R1	R11	R111	C1	C11	C111	1
R1	R11	R112	C1	C11	C112	2

Row 1	Row 2	Row 3	Column 1	Column 2	Column 3	Values
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
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

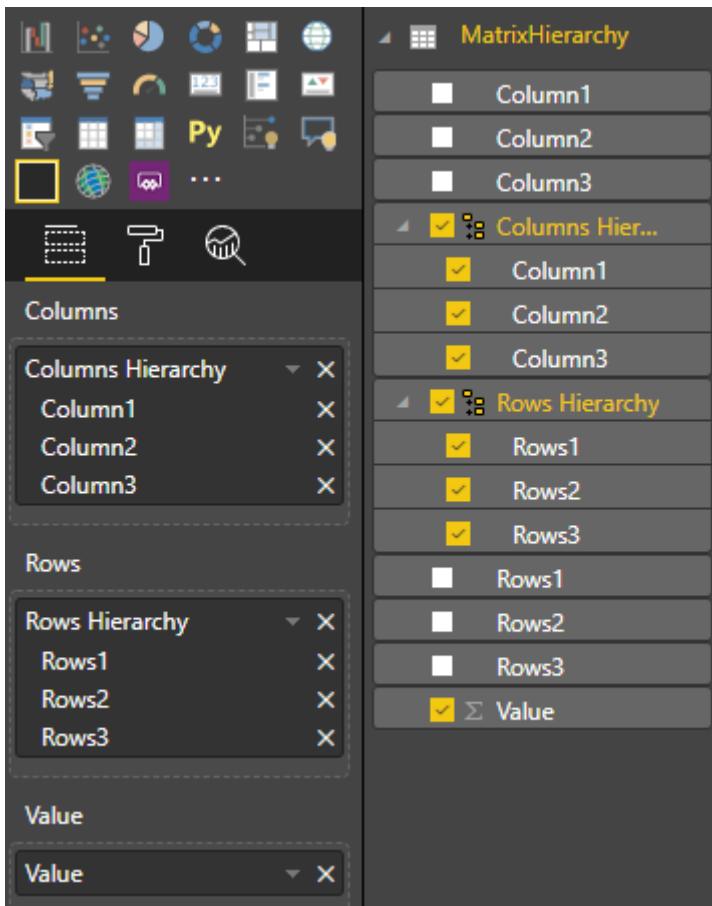
Then apply the following data view mapping to the visual:

JSON

```
{
  "dataRoles": [
    {
      "displayName": "Columns",
      "name": "columns",
      "kind": "Grouping"
    },
    {
      "displayName": "Rows",
      "name": "rows",
      "kind": "Grouping"
    },
    {
      "displayName": "Value",
      "name": "value",
      "kind": "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:

TypeScript

```
// ...
import DataViewMatrix = powerbi.DataViewMatrix;
import DataViewMatrixNode = powerbi.DataViewMatrixNode;
import DataViewHierarchyLevel = powerbi.DataViewHierarchyLevel;
// ...
```

Create two properties for two `div`s of rows and columns elements:

TypeScript

```
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:

TypeScript

```

export class Visual implements IVisual {
    // ...
    constructor(options: VisualConstructorOptions) {
        // constructor body
    }

    public update(options: VisualUpdateOptions) {
        this.formattingSettings =
            this.formattingSettingsService.populateFormattingSettingsModel(VisualFormat-
tingSettingsModel, options.dataViews);
        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:

TypeScript

```

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.

TypeScript

```

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:

## TypeScript

```
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 `buttons` to interact with visual and display context menu for matrix datapoints:

## TypeScript

```
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:

TypeScript

```

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,

```

```
matrix DataView.columns.levels, this.colsDiv));
}
```

Finally, you should get a visual with context menu:

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

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

## Related content

- [How to use the drill down support API](#)
- [Dynamic drill-down control](#)

## Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Add colors to your Power BI visuals

Article • 03/10/2025

This article describes how to add colors to your custom visuals and how to handle data points for a visual that has defined colors.

`IVisualHost`, the collection of properties and services that interact with the visual host, can define colors in custom visuals with the `colorPalette` service. The example code in this article modifies the [SampleBarChart visual](#). For the SampleBarChart visual source code, see [barChart.ts](#).

To get started creating visuals, see [Developing a a Power BI circle card visual](#).

## Add color to data points

To represent each data point in a different color, add the `color` variable to the `BarChartDataPoint` interface, as shown in the following example:

TypeScript

```
/**
 * 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 `colorPalette` service is available on `IVisualHost`.

Define the color palette in the `update` method with the following code:

TypeScript

```

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` has a defined value, category, and color property. `dataPoints` can also include other properties.

In `SampleBarChart`, the `visualTransform` method is a part of the Bar Chart viewmodel. Because the `visualTransform` method iterates through all the `dataPoints` calculations, it's the ideal place to assign colors, as in the following code:

TypeScript

```

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 the data from `dataPoints` to the `d3`-selection `barSelection` inside the `update` method:

TypeScript

```
// This code is 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);

barSelectionMergedclassed('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();
```

## Related content

- Capabilities and properties of Power BI visuals.
- How to debug Power BI visuals

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Add conditional formatting

Article • 08/19/2024

[Conditional formatting](#) lets a report creator specify how colors are displayed in a report, according to a numerical value.

This article describes how to add the conditional formatting functionality to your Power BI visual.

Currently, conditional formatting can only be applied to color.

## Add conditional formatting to your project

This section shows how to add conditional formatting to an existing Power BI visual. The example code in this article is based on the [SampleBarChart](#) visual. You can examine the source code in [barChart.ts](#).

### Add a conditional color formatting entry in the format pane

In this section you learn how to add a conditional color formatting entry, to a data point in format pane.

1. Use the `propertyInstanceKind` array in `VisualObjectInstance`, which is exposed by `powerbi-visuals-api`. Verify that your file includes this import:

TypeScript

```
import powerbiVisualsApi from "powerbi-visuals-api";
```

2. To specify the appropriate type of formatting (*Constant*, *ConstantOrRule*, or *Rule*), use the `VisualEnumerationInstanceKinds` enum. Add the following import to your file:

TypeScript

```
import VisualEnumerationInstanceKinds =  
powerbiVisualsApi.VisualEnumerationInstanceKinds;
```

3. Set formatting property instance kind

## getFormattingModel API method

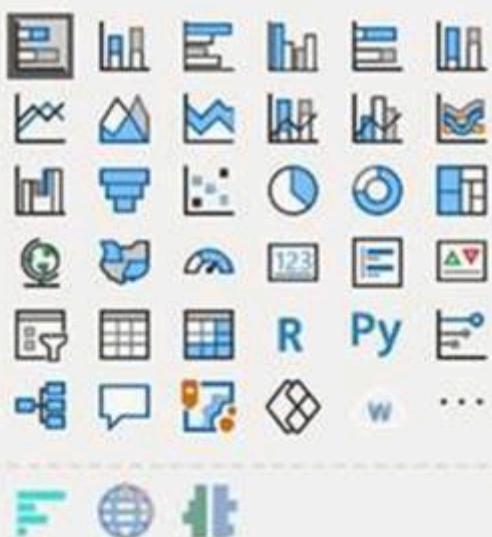
To format properties that support conditional formatting, set the required instance kind in their `descriptor`.

TypeScript

```
public getFormattingModel(): powerbi.visuals.FormattingModel {
    // ...
    formattingGroup.slices.push(
        {
            uid: `colorSelector${barDataPoint_idx}_uid`,
            displayName: barDataPoint.category,
            control: {
                type: powerbi.visuals.FormattingComponent.ColorPicker,
                properties: {
                    descriptor: {
                        objectName: "colorSelector",
                        propertyName: "fill",
                        selector:
                            DataViewWildcard.createDataViewWildcardSelector(DataViewWildcardMatchingOption.InstancesAndTotals),
                        altConstantValueSelector:
                            barDataPoint.selectionId.getSelector(),
                        instanceKind:
                            powerbi.VisualEnumerationInstanceKinds.ConstantOrRule // <== Support
                            conditional formatting
                    },
                    value: { value: barDataPoint.color }
                }
            }
        );
    // ...
}
```

`VisualEnumerationInstanceKinds.ConstantOrRule` creates the conditional formatting UI entry alongside the constant formatting UI element.

## Visualizations >



Search

General

Y axis On

X axis On

Zoom slide... Off

Data colors

Show all

Off

Default color



Revert to default

# Define how conditional formatting behaves

Define how formatting is applied to your data points.

Using `createDataViewWildcardSelector` declared under `powerbi-visuals-utils-dataviewutils`, specify whether to apply conditional formatting to instances, totals, or both. For more information, see [DataViewWildcard](#).

Make the following changes to the properties you want to apply conditional formatting to:

- Replace the `selector` value with the `dataViewWildcard.createDataViewWildcardSelector(dataViewWildcardMatchingOption)` call. `DataViewWildcardMatchingOption` defines whether conditional formatting is applied to instances, totals, or both.
- Add the `altConstantValueSelector` property with the value previously defined for the `selector` property.

getFormattingModel API method

For formatting properties that support conditional formatting, set the required instance kind in their `descriptor`.

TypeScript

```
public getFormattingModel(): powerbi.visuals.FormattingModel {
    // ...

    formattingGroup.slices.push(
        {
            uid: `colorSelector${barDataPoint_idx}_uid`,
            displayName: barDataPoint.category,
            control: {
                type: powerbi.visuals.FormattingComponent.ColorPicker,
                properties: {
                    descriptor: {
                        objectName: "colorSelector",
                        propertyName: "fill",
                        // Define whether the conditional formatting
                        will apply to instances, totals, or both
                        selector:
                            dataViewWildcard.createDataViewWildcardSelector(dataViewWildcard.DataVie
                            wWildcardMatchingOption.InstancesAndTotals),

                        // Add this property with the value previously
                        defined for the selector property
                    }
                }
            }
        }
    )
}
```

```
        altConstantValueSelector:  
        barDataPoint.selectionId.getSelector(),  
  
        instanceKind:  
powerbi.VisualEnumerationInstanceKinds.ConstantOrRule  
    },  
    value: { value: barDataPoint.color }  
}  
}  
};  
  
// ...  
}
```

## Considerations and limitations

Conditional formatting isn't supported for the following visuals:

- Table based visuals
- Matrix based visuals

We recommend that you don't use conditional formatting with series. Instead, you should allow customers to format each series individually, making it easy to visually distinguish between series. Most out-of-the-box visuals with series, share this approach.

## Related content

[DataViewUtils](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

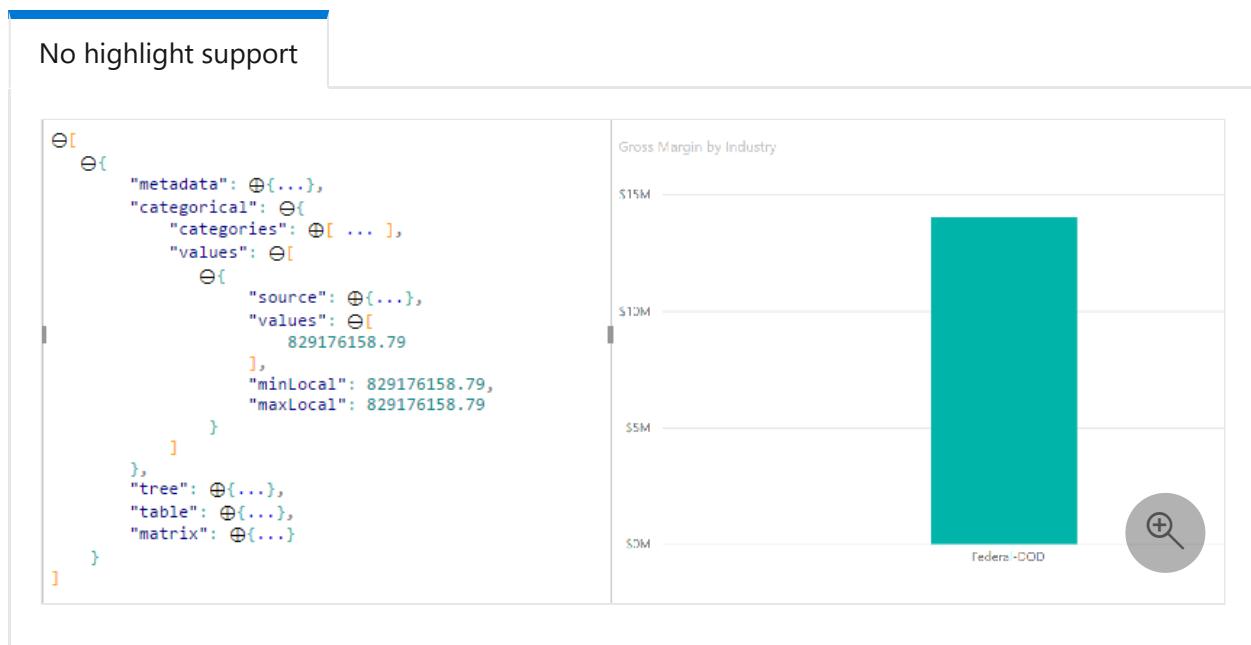
# Highlight data points in Power BI Visuals

Article • 06/17/2024

This article describes how to highlight data on Power BI visuals.

By default, when an element is selected, the `values` array in the `dataView object` is filtered to only show the selected values. When the `values` array is filtered, all other visuals on the page only show the selected data.

If you set the `supportsHighlight` property in your `capabilities.json` file to `true`, it results in the full unfiltered `values` array along with a `highlights` array. The `highlights` array is the same length as the `values` array, and any unselected values are set to `null`. With this property enabled, the appropriate data in the visual is highlighted by comparing the `values` array to the `highlights` array.



In the example, notice that:

- **Without** highlight support, the selection is the only value in the `values` array and the only bar presented in the data view.
- **With** highlight support, all values are in the `values` array. The `highlights` array contains a `null` value for non-highlighted elements. All bars appear in the data view, and the highlighted bar is a different color.

There can also be multiple selections and partial highlights. The highlighted values are presented in the data view.

## ⚠ Note

Table data view mapping doesn't support the highlights feature.

# Highlight data points with categorical data view mapping

For visuals with [categorical data view mapping](#), add `"supportsHighlight": true` to the `capabilities.json` file. For example:

JSON

```
{  
  "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}  
  ]  
}
```

After you remove unnecessary code, the default visual source code looks like the following example:

## TypeScript

```
"use strict";

// ... default imports list

import { FormattingSettingsService } from "powerbi-visuals-utils-
formattingmodel";

import DataViewCategorical = powerbi.DataViewCategorical;
import DataViewCategoryColumn = powerbi.DataViewCategoryColumn;
import PrimitiveValue = powerbi.PrimitiveValue;
import DataViewValueColumn = powerbi.DataViewValueColumn;

import { VisualFormattingSettingsModel } from "./settings";

export class Visual implements IVisual {
    private target: HTMLElement;
    private formattingSettings: VisualFormattingSettingsModel;
    private formattingSettingsService: FormattingSettingsService;

    constructor(options: VisualConstructorOptions) {
        console.log('Visual constructor', options);
        this.formattingSettingsService = new FormattingSettingsService();
        this.target = options.element;
        this.host = options.host;
    }

    public update(options: VisualUpdateOptions) {
        this.formattingSettings =
            this.formattingSettingsService.populateFormattingSettingsModel(VisualFormat-
tingSettingsModel, options.dataViews);
        console.log('Visual update', options);
    }

    // Returns properties pane formatting model content hierarchies,
    properties and latest formatting values, Then populate properties pane.
    // This method is called once every time we open properties pane or when
    the user edit any format property.
    public getFormattingModel(): powerbi.visuals.FormattingModel {
        return
    this.formattingSettingsService.buildFormattingModel(this.formattingSettings)
    ;
    }
}
```

Import required interfaces to process data from Power BI:

## TypeScript

```
import DataViewCategorical = powerbi.DataViewCategorical;
import DataViewCategoryColumn = powerbi.DataViewCategoryColumn;
```

```
import PrimitiveValue = powerbi.PrimitiveValue;
import DataViewValueColumn = powerbi.DataViewValueColumn;
```

Create the root `div` element for category values:

TypeScript

```
export class Visual implements IVisual {
    private target: HTMLElement;
    private formattingSettings: VisualFormattingSettingsModel;
    private formattingSettingsService: FormattingSettingsService;

    private div: HTMLDivElement; // new property

    constructor(options: VisualConstructorOptions) {
        console.log('Visual constructor', options);
        this.formattingSettingsService = new FormattingSettingsService();
        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 the contents of the `div` elements before rendering new data:

TypeScript

```
// ...
public update(options: VisualUpdateOptions) {
    this.formattingSettings =
this.formattingSettingsService.populateFormattingSettingsModel(VisualFormat-
tingSettingsModel, options.dataViews);
    console.log('Visual update', options);

    while (this.div.firstChild) {
        this.div.removeChild(this.div.firstChild);
    }
    // ...
}
```

Get the categories and measure values from the `dataView` object:

TypeScript

```

public update(options: VisualUpdateOptions) {
    this.formattingSettings =
        this.formattingSettingsService.populateFormattingSettingsModel(VisualFormattingSettingsModel, options.dataViews);
    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 the highlighted parts of values.

### ⓘ Note

If values of the `measureHighlights` property are less than values of the `categoryValues` property, then the value was partially highlighted.

Enumerate the `categoryValues` array and get corresponding values and highlights:

TypeScript

```

// ...
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 the visual DOM:

## TypeScript

```
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 the required styles for elements to use `flexbox`, and define colors for div elements:

## css

```
div.vertical {
  display: flex;
  flex-direction: column;
}

div.horizontal {
```

```

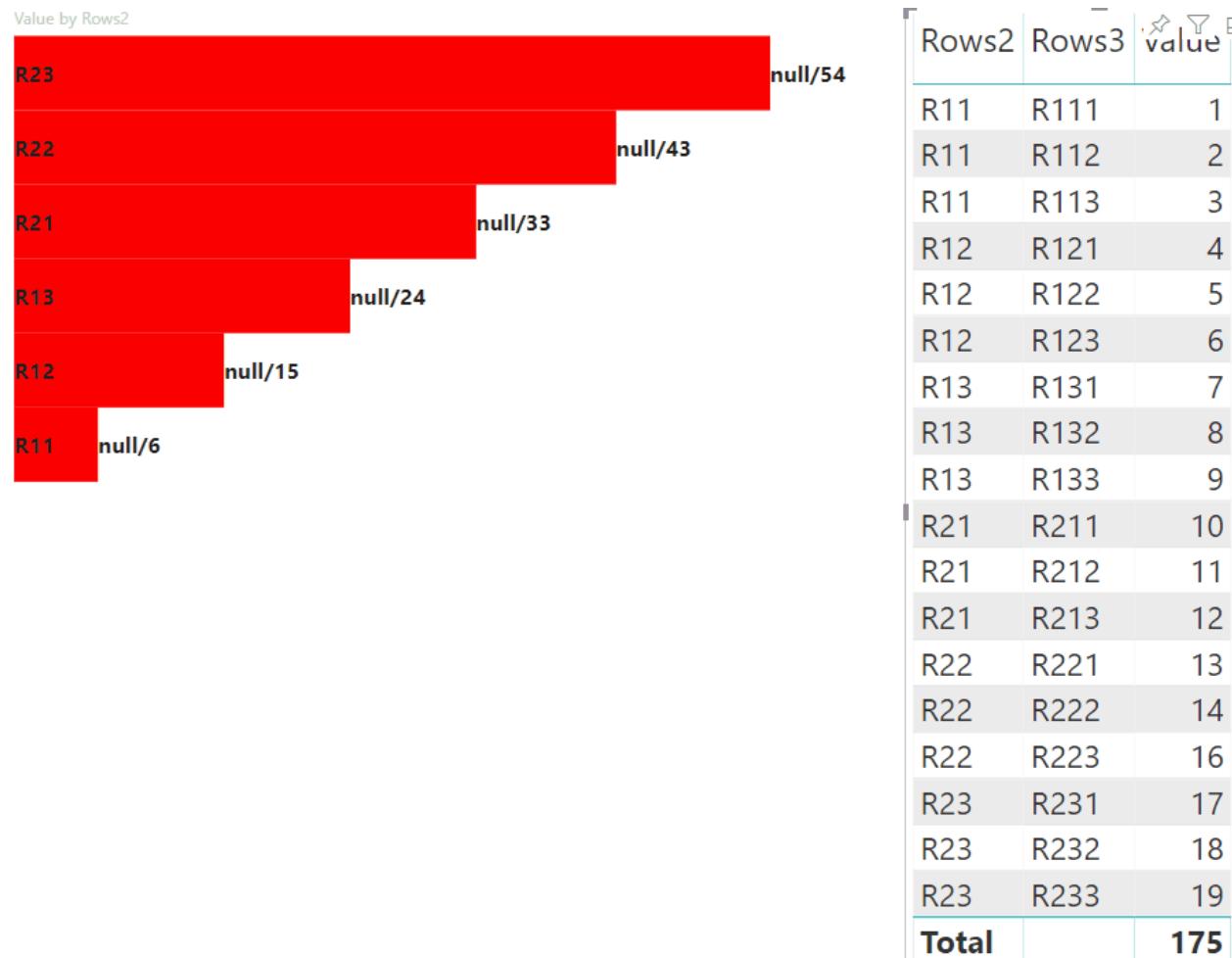
    display: flex;
    flex-direction: row;
}

div.highlight {
    background-color: blue
}

div.value {
    background-color: red;
    display: flex;
}

```

The following view of the visual is the result:



## Highlight data points with matrix data view mapping

For visuals with [matrix data view mapping](#), add `"supportsHighlight": true` to the `capabilities.json` file. For example:

JSON

```
{
  "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 a hierarchy for matrix data view mapping:

[\[\] Expand table](#)

Row1	Row2	Row3	Column1	Column2	Column3	Values
R1	R11	R111	C1	C11	C111	1
R1	R11	R112	C1	C11	C112	2

Row1	Row2	Row3	Column1	Column2	Column3	Values
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
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 the sample of the `capabilities.json` file.

After you remove the unnecessary code, the default visual source code looks like the following example:

```
TypeScript

"use strict";

// ... default imports

import { FormattingSettingsService } from "powerbi-visuals-utils-
formattingmodel";
import { VisualFormattingSettingsModel } from "./settings";

export class Visual implements IVisual {
    private target: HTMLElement;
    private formattingSettings: VisualFormattingSettingsModel;
```

```

private formattingSettingsService: FormattingSettingsService;

constructor(options: VisualConstructorOptions) {
    console.log('Visual constructor', options);
    this.formattingSettingsService = new FormattingSettingsService();
    this.target = options.element;
    this.host = options.host;
}

public update(options: VisualUpdateOptions) {
    this.formattingSettings =
this.formattingSettingsService.populateFormattingSettingsModel(VisualFormat-
ingSettingsModel, options.dataViews);
    console.log('Visual update', options);

}

/**
 * Returns properties pane formatting model content hierarchies,
properties and latest formatting values, Then populate properties pane.
 * This method is called once every time we open properties pane or when
the user edit any format property.
 */
public getFormattingModel(): powerbi.visuals.FormattingModel {
    return
this.formattingSettingsService.buildFormattingModel(this.formattingSettings)
;
}
}

```

Import the required interfaces to process data from Power BI:

TypeScript

```

import DataViewMatrix = powerbi.DataViewMatrix;
import DataViewMatrixNode = powerbi.DataViewMatrixNode;
import DataViewHierarchyLevel = powerbi.DataViewHierarchyLevel;

```

Create two `div` elements for the visual layout:

TypeScript

```

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 the `update` method to ensure that the visual gets data:

TypeScript

```
public update(options: VisualUpdateOptions) {
    this.formattingSettings =
this.formattingSettingsService.populateFormattingSettingsModel(VisualFormat-
ingSettingsModel, options.dataViews);
    console.log('Visual update', options);

    const dataView: DataView = options.dataViews[0];
    const matrixDataView: DataViewMatrix = dataView.matrix;

    if (!matrixDataView ||
        !matrixDataView.columns ||
        !matrixDataView.rows ) {
        return
    }
    // ...
}
```

Clear the contents of the `div` elements before rendering new data:

TypeScript

```
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 the `treeWalker` function to traverse the matrix data structure:

TypeScript

```

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 the recursive function, need to create `div` element and `p` for text as header, and call the function for child elements of node:

TypeScript

```

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 the column and row of the matrix data view structure:

TypeScript

```

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:

TypeScript

```

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 highlighting is to create another array of values.

The object of terminal node has two properties for the values array, value and highlight:

JavaScript

```
JSON.stringify(options.dataViews[0].matrix.rows.root.children[0].children[0]
  .children[0], null, " ");
```

JSON

```
{
  "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` represents the value of the node without applying a selection from the other visual, `highlight` indicates which part of the data was highlighted.

### ⓘ Note

If the value of `highlight` is less than the value of `value`, then `value` was partially highlighted.

Add code to process the `values` array of the node if it's presented:

TypeScript

```
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");

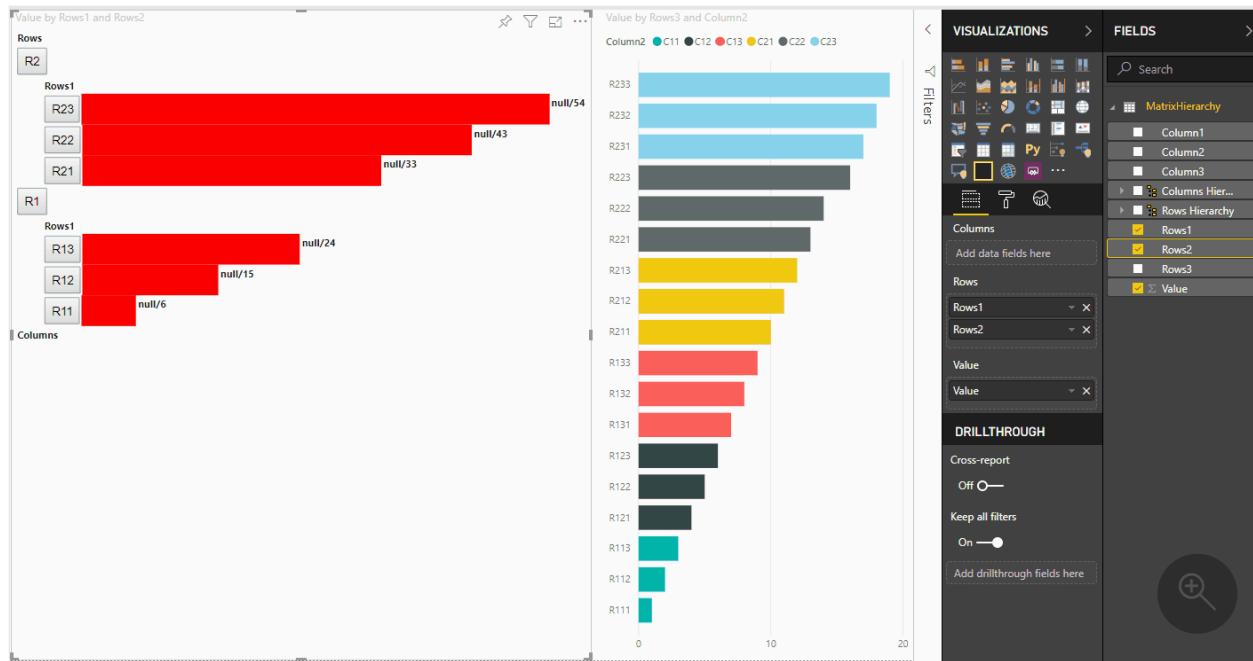
            // create div container for highlighted values
            const highlightsDiv = document.createElement("div");
            highlightsDiv.style.width = sumOfHighlights * 10 + "px";
            highlightsDiv.classList.add("highlight");
            valueDiv.appendChild(highlightsDiv);

            // 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) {
            // ...
        }
    }
}
```

```
    }  
    // ...  
}
```

The result is a visual with buttons and values, like **highlighted value/default value**.



## Related content

- [Matrix data mappings](#)
- [Add interactivity into visual by Power BI visuals selections](#)

## Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

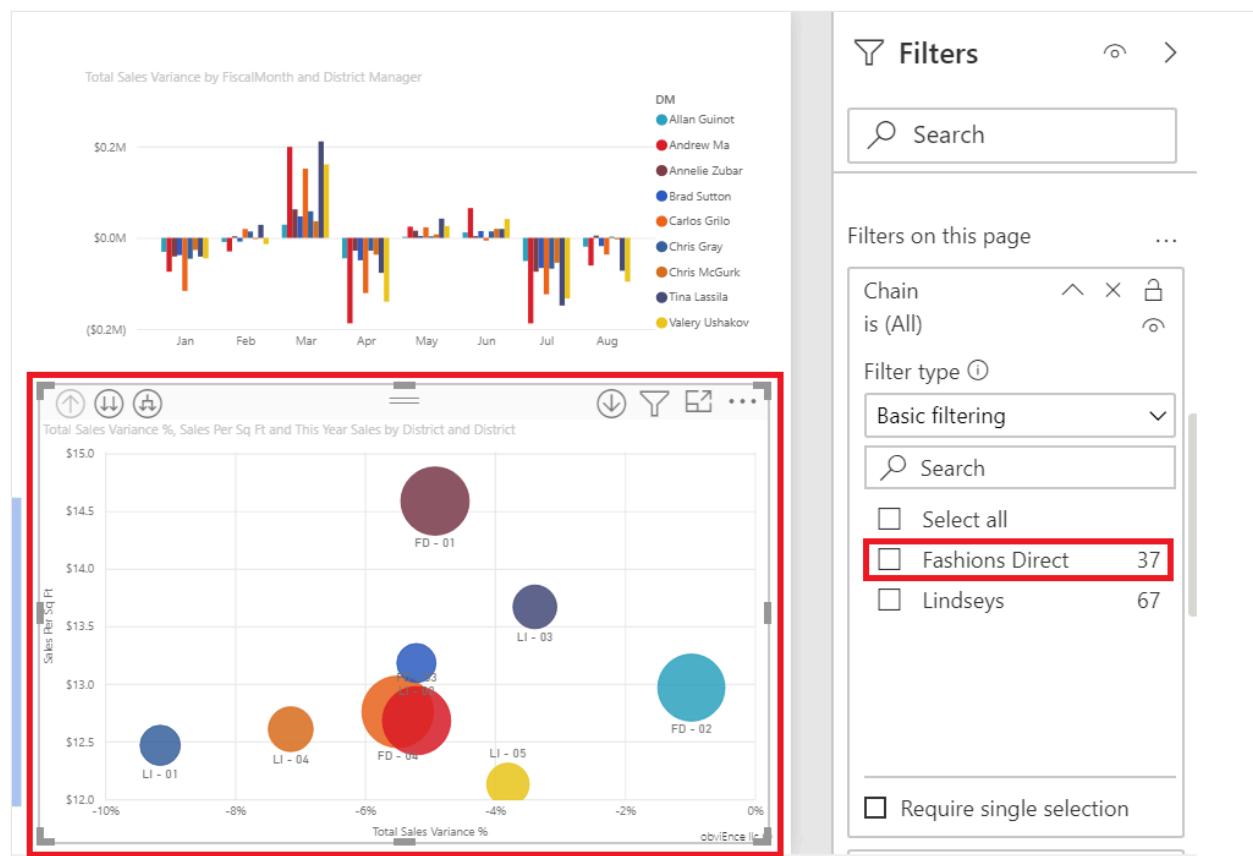
# Apply a selection to multiple visuals in a report

Article • 12/19/2023

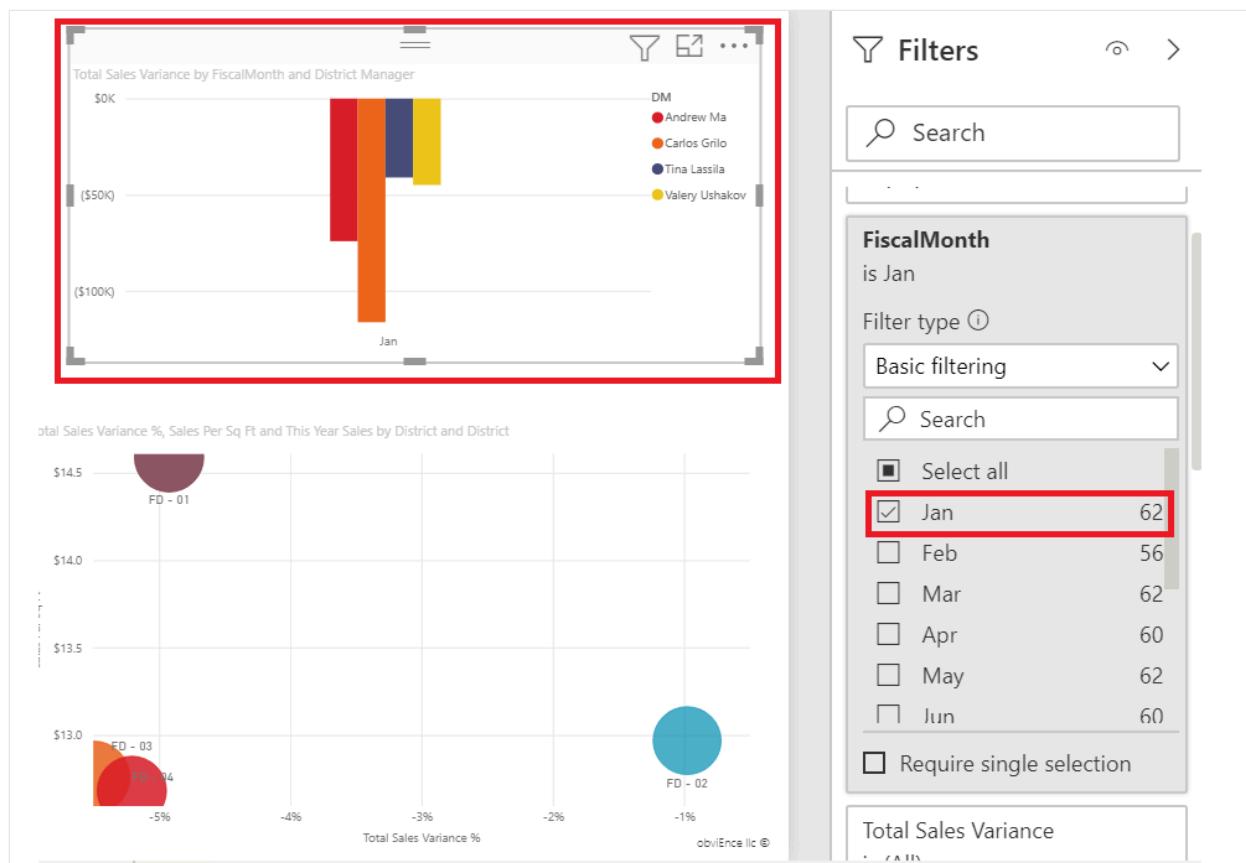
The `supportsMultiVisualSelection` feature enables you to select values from one visual in a Power BI report and apply the selected values to all the visuals in that report.

For example, in the [Overview](#) page of the [Retail Analysis](#) sample:

1. Select the **Total Sales Variance %**, **Sales Per Sq Ft** and **This Year Sales by District and District** visual. In the **Filters** pane that appears, under **Chain**, select **Fashions Direct**.



2. Select the **Total Sales Variance by FiscalMonth and District Manager** visual. In the **Filters** pane that appears, under **FiscalMonth**, select **Jan**.



In the report, these selections apply to all visuals that support this feature. The scope of the visuals is now limited to **Fashions Direct** and **January**.

## Enable the support multiple visual selection feature

To use the support multiple visual selection feature, add the following code to the *capabilities.json* file of your visual:

```
JSON
{
    ...
    "supportsMultiVisualSelection": true
    ...
}
```

## Considerations and limitations

- This feature requires API v3.2.0 or later.
- This feature doesn't apply to image visuals.
- This feature doesn't apply to certain advanced visuals, such as key driver, decomposition tree, Q&A, textbox, and gauge chart visuals.

## Related content

[Visuals in Power BI](#)

[Developing a Power BI circle card](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Support keyboard navigation in a custom visual

Article • 01/20/2024

This article explains how to support navigation through a Power BI visual using the keyboard. Keyboard navigation makes Power BI more accessible to people with disabilities and provides more options for interacting with reports.

The `supportsKeyboardFocus` feature makes it possible to navigate the data points of the visual by using only the keyboard.

## Basic keyboard accessibility features

All visuals come with the following basic keyboard accessibility:

- Press `Esc` to move the focus from inside the visual to the visual container.
- Press `Tab` from inside a custom visual to navigate through tabbable elements in the visual. Pressing `Tab` after the last tabbable element moves the focus back outside of the visual.

## Enhanced keyboard accessibility

To make your custom visual even more accessible, add the `supportsKeyboardFocus` capability to your visual by adding the following line to the "capabilities.json" file:

JSON

```
{  
  ...  
  "supportsKeyboardFocus": true  
  ...  
}
```

This capability adds the following features to your custom visual:

- Press `Enter` when the focus is on the visual container to move the focus to inside the custom visual.
- Press `Tab` from inside the custom visual to navigate through tabbable elements. The focus stays inside the visual until you press `Esc`.

### Note

Not all HTML elements are tabbable by default (for example, div and span). Consider adding the correct attribute (e.g tabindex) to these elements to make them tabbable.

## Considerations and limitations

- This feature requires API v2.1.0 or higher.
- This feature can't be applied to image visuals.
- Pressing `Enter` on the visual container won't always land on the first focusable element of the visual. To be sure to start at the first element, focus it programmatically after the focus goes into the visual.
- After pressing `Enter` on the visual container and after pressing `Tab` on the last focusable element, the user might have to press `Tab` more than once to get to the first element.

## Related content

- [Design Power BI reports for accessibility](#)
- [Developing a Power BI circle card visual](#)

---

## Feedback

Was this page helpful?



Yes



No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# High-contrast mode support in Power BI visuals

Article • 06/17/2024

The Windows *high-contrast* setting makes text and graphics 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](#).

To display a visual in high-contrast mode, you have to:

- Detect high-contrast mode and colors upon [initialization](#).
- Draw the visual correctly on [implementation](#).

## 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

When displaying 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.

## ① Note

If a secondary color is needed, the foreground color can 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 in your `constructor` method:

TypeScript

```
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 an update.

## Implementation

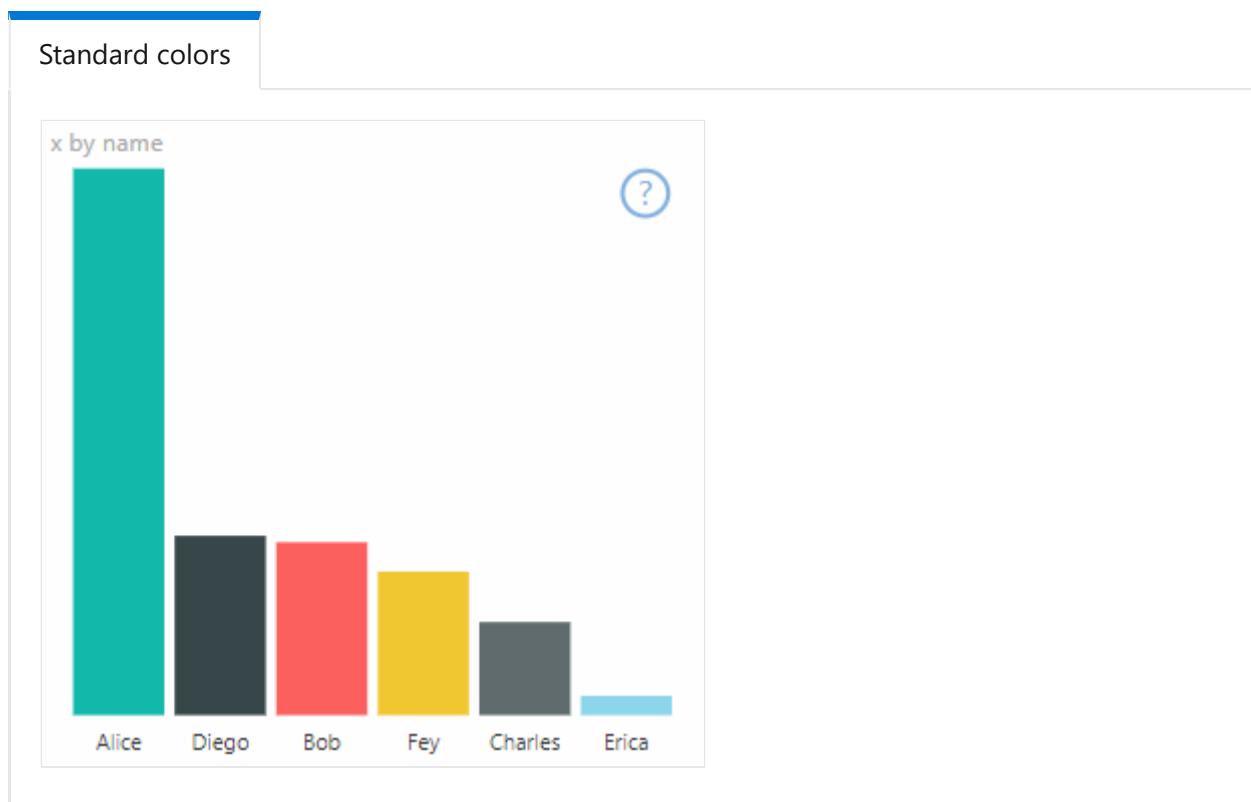
The specific implementations of high-contrast support vary from visual to visual and depend on the details of the graphic design. To keep important details easy to distinguish with limited colors, high-contrast mode ordinarily requires a design that's slightly different from the default mode.

Power BI native visuals follow these guidelines:

- All data points use the same color (foreground).
- All text, axes, arrows, and lines 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 and active filter elements, use the foreground-selected color.

The following sample bar chart is drawn with two pixels of thick foreground outline and background fill. Compare the way it looks with default colors and with the following high-contrast themes:



## Example

The following code shows one place in the `visualTransform` function that was changed to support high-contrast. It's called as part of rendering during the update. For the full implementation of this code, see the `barChart.ts` file in the [PowerBI-visuals-sampleBarChart visual repository](#).

No high-contrast support

```
TypeScript
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: minValue,
    color: getCategoricalObjectValue<Fill>(category, i,
'colorSelector', 'fill', defaultColor).solid.color,
    selectionId: host.createSelectionIdBuilder()
        .withCategory(category, i)
        .createSelectionId()
});
}
```

## Related content

[Design Power BI reports for accessibility](#)

## Feedback

Was this page helpful?

[!\[\]\(3308d3c7c4fd94d0a58c0c47daac2464\_img.jpg\) Yes](#)

[!\[\]\(9cf53cd71d7cf79e447108fd2cdf90a4\_img.jpg\) No](#)

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Sync slicers across pages in Power BI reports

Article • 12/04/2023

[Slicers](#) are a useful way to filter information and focus on a specific portion of the semantic model. They allow you to select exactly which values to display in your Power BI visual.

Sometimes you might want to use a slicer on only one specific page of the report. Other times, you might want to apply the slicer to several pages. By using the *sync slicers* feature, a slicer selection on any page will affect visualizations on all selected pages.

For information about sync slicers and how they work, see [Sync and use slicers on other pages](#).

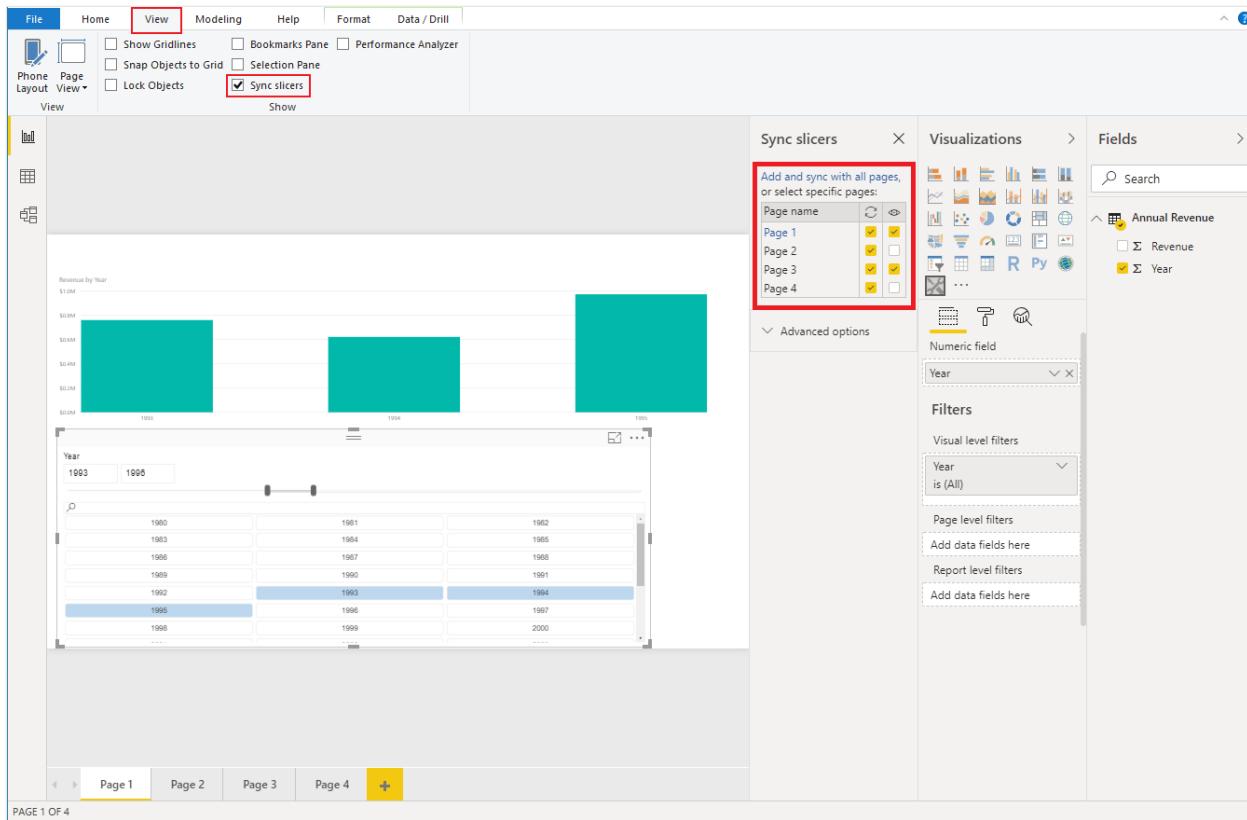
## How to enable the sync slicer feature

In the *capabilities.json* file, set `"supportsSynchronizingFilterState": true`, as shown in the following example:

```
JSON

{
    ...
    "supportsHighlight": true,
    "suppressDefaultTitle": true,
    "supportsSynchronizingFilterState": true,
    "sorting": {
        "default": {}
    }
}
```

After you've updated the *capabilities.json* file, you can view the **Sync slicers** pane when you select your custom slicer visual.



From the **Sync slicers** pane, you can select which report pages the slicer visibility and filtration should apply to.

For more information on how to sync slicers, see [Sync and use slicers on other pages](#).

### ⓘ Note

A report using the sync slicers feature must use API version 1.13.0 or later.

## Considerations and limitations

The sync slicers feature only supports one field at a time. If your slicer has more than one field (**Category** or **Measure**), the sync slicers feature is disabled.

## Related content

[Add a context menu to your Power BI visual](#)

# Request aggregated subtotal data

Article • 11/22/2024

The *Total and Subtotal API* allows custom visuals with a matrix data-view to request aggregated subtotal data from the Power BI host. The subtotals are calculated for the entire matrix semantic model or specified for individual levels of the matrix data hierarchy. See the [sample report](#) for an example of the Total and Subtotal API in a Power BI visual.

## ⓘ Note

Requesting subtotal data is supported in version 2.6.0 and later. The `rowSubtotalType` property is available in version 5.1.0 and later. To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.

Every time a visual refreshes its data, the visual issues a [data fetch request](#) to the Power BI backend. These data requests are usually for values of the fields that the user dragged into the field wells of the visual. Sometimes the visual needs other aggregations or subtotals (for example, sum or count) applied to the field wells. The Total and Subtotal API lets you customize the outgoing data query to request more aggregation or subtotal data.

A screenshot of a Power BI matrix visual titled "SalesQuantity by ContinentName, RegionCountryName, QuarterOfYear and MonthName". The visual displays sales data across four continents: North America, Asia, Europe, and South America. The data is organized into four columns representing Quarters (1, 2, 3, 4) and twelve rows representing months. The "SalesQuantity" column header has a context menu open, showing options for aggregation: Sum (selected), Average, Minimum, Maximum, Count (Distinct), Count, Standard deviation, Variance, and Median. The menu also includes options to Remove field, Rename, and Move to.

1				2				3				4			
	February	January	March	Totals	April	June	May	Totals	August	July	September	Totals	October	November	December
North America															
United States	1343256	1388890	1293376	4025522	1745192	1723105	1773585	5241882	1881114	1869757	1802851	5553722	1945992	1939825	1867363
Canada	48552	47207	44803	140562	65610	65420	65955	196985	64878	70068	64512	199458	1945992	1939825	1867363
<b>Totals</b>	<b>1391808</b>	<b>1436097</b>	<b>1338179</b>	<b>4166084</b>	<b>1810802</b>	<b>1788525</b>	<b>1839540</b>	<b>5438867</b>	<b>1945992</b>	<b>1939825</b>	<b>1867363</b>	<b>5753180</b>			
Asia															
China	316813	431083	316798	1064494	351680	440802	459714	1252196	402220	473934	383635	1259789			
Japan	53407	61485	47064	161956	49277	70200	68298	187775	60790	66474	66315	193579			
Australia	24659	32950	24890	82499	27668	35178	38430	101276	28075	30384	29140	87599			
India	25030	30933	25753	81716	26387	33462	33192	93041	28095	32856	26290	87241			
Turkmenistan	16837	21497	15583	53917	17684	18996	21714	58394	19808	21228	16580	57613			
Iran	14887	20567	13980	49434	18442	21564	20388	60394	20285	21906	17495	59886			
Syria	14631	17621	14766	47018	16241	20982	20958	58181	17145	17208	15930	50283			
Pakistan	12233	18912	12789	43934	15971	16512	20166	52649	14900	19146	14540	48586			
South Korea	11977	15280	12747	40004	12644	13962	17394	44000	15685	17688	15630	49003			
Thailand	12466	16706	9771	38943	11622	16392	16914	44928	12530	13236	13820	39586			

## The subtotals API

The API offers the following customization for each data-view type (currently, only matrix data-views).

- `rowSubtotals`: (boolean) Indicates if the subtotal data should be requested for all fields in the rows field well.
- `rowSubtotalsPerLevel`: (boolean) Indicates if the subtotal data can be toggled for individual fields in the row's field well.
- `columnSubtotals`: (boolean) Indicates if the *subtotal* data should be requested for all fields in the columns field well.
- `columnSubtotalsPerLevel`: (boolean) Indicates if the *subtotal* data can be toggled for individual fields in the columns field well.
- `levelSubtotalEnabled`: (boolean) Indicates if the subtotals are requested for the row or column. Unlike all the other properties, this property is applied to individual rows or columns.
- `rowSubtotalsType`: ("Top" or "Bottom") Indicates if the row with the *total* data should be retrieved before (`top`) or after (`bottom`) the rest of the data. If this property is set to `bottom`, the total can only be displayed after all the data is fetched. The default is `bottom`.

Each of these switches is assigned a value based on the related properties in the property pane and the defaults.

## How to use the subtotal API

The visual's *capabilities.json* file has to:

- specify the property that each `switch` maps to.
- provide the default value to be used if the property is undefined.

The switches use a format like the following example:

```
JSON

"rowSubtotals": {
    "propertyIdentifier": {
        "objectName": "subTotals",
        "propertyName": "rowSubtotals"
    },
    "defaultValue": true
},
```

The preceding code indicates that the row subtotals are enabled by the property `rowSubtotals` in the `subTotals` object. The property has a default value of `true`.

The API is automatically enabled for a visual whenever the subtotals structure and all switch mappings are defined in the *capabilities.json* file.

The following code is an example of the complete API configuration in the *capabilities.json* file (copied from the API sample visual):

JSON

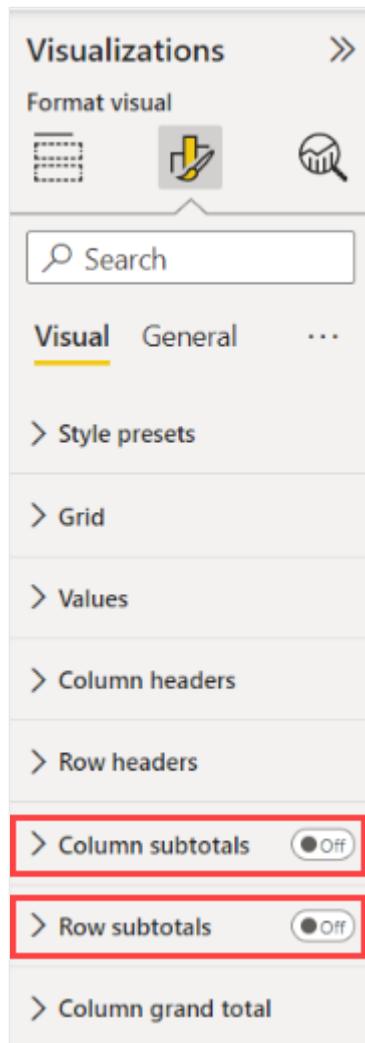
```
"subtotals": {
    "matrix": {
        "rowSubtotals": {
            "propertyIdentifier": {
                "objectName": "subTotals",
                "propertyName": "rowSubtotals"
            },
            "defaultValue": true
        },
        "rowSubtotalsPerLevel": {
            "propertyIdentifier": {
                "objectName": "subTotals",
                "propertyName": "perRowLevel"
            },
            "defaultValue": false
        },
        "columnSubtotals": {
            "propertyIdentifier": {
                "objectName": "subTotals",
                "propertyName": "columnSubtotals"
            },
            "defaultValue": true
        },
        "columnSubtotalsPerLevel": {
            "propertyIdentifier": {
                "objectName": "subTotals",
                "propertyName": "perColumnLevel"
            },
            "defaultValue": false
        },
        "levelSubtotalEnabled": {
            "propertyIdentifier": {
                "objectName": "subTotals",
                "propertyName": "levelSubtotalEnabled"
            },
            "defaultValue": true
        },
        "rowSubtotalsType": {
            "propertyIdentifier": {
                "objectName": "subtotals",
                "propertyName": "rowSubtotalsType"
            },
            "defaultValue": "Bottom"
        }
    }
}
```

It's important that the `enumerateProperties()` function of the visual aligns with the defaults specified in the `capabilities.json` file. The customization logic operates according to the specified defaults. If the `enumerateProperties()` function and the defaults aren't aligned, the actual subtotal customizations might differ from user expectations.

```
TypeScript

enum RowSubtotalType {
    Top = "Top",
    Bottom = "Bottom",
}
```

To review the available customizations, expand the **Subtotals** drop-down menu in the **Format** property pane. Modify the subtotals settings and track the changes to the subtotals presentation (named *Totals*) in the Visualizations pane.



## Considerations and limitations

- The `rowSubtotalsType` property is only available for rows. You can't set column subtotals to the beginning of a column.

- The [expand and collapse](#) feature overrides `rowSubtotals`. Subtotals display when the rows are expanded, even if `rowSubtotals` is set to `false`.

## Related content

[Add interactivity to visual using Power BI visuals selections](#)

---

## Feedback

Was this page helpful?

 Yes

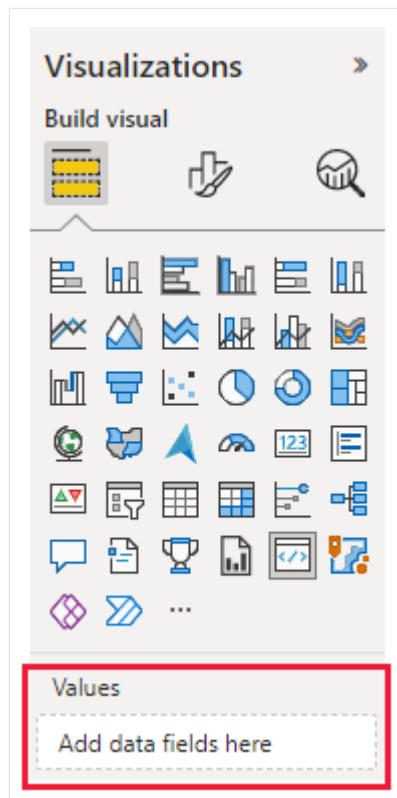
 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Create custom Power BI visuals without data binding

Article • 01/31/2024

This article explains how to use the *No data binding feature* to create Power BI custom visuals without data roles. Ordinarily, when you create a visual in a Power BI report, the values are defined interactively by adding data fields to the **Values** well on the **Visualizations** pane.



By default, if no values are defined, the format settings are disabled, and you can't update the visual's formatting.

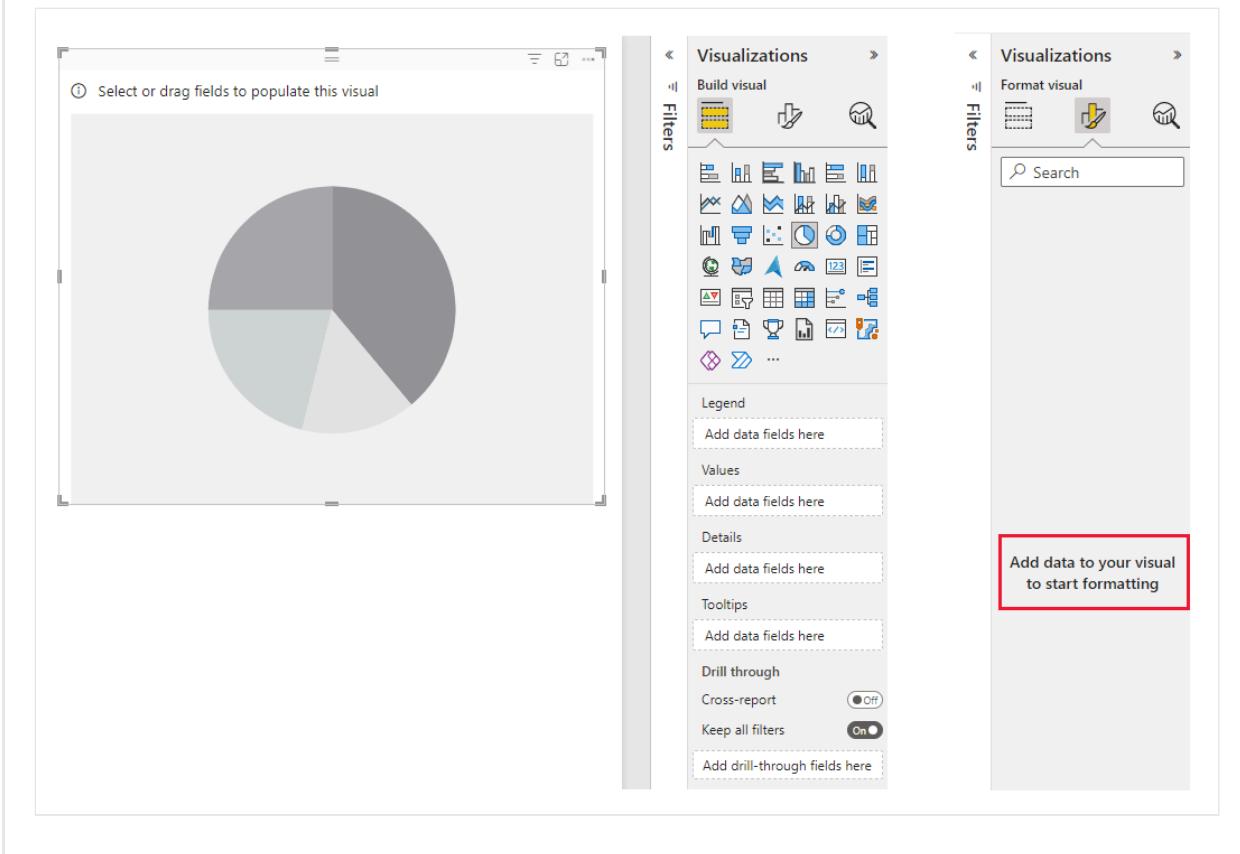
The `dataRoles` property of the [capabilities model](#) allows you to format graphics in Power BI without binding data.

Using the `dataRoles` capabilities property, you can render a visual and use the `update` method to change the format settings. You can change settings even if the data buckets are empty, or if your visual doesn't use any data roles.

The following tabs show two examples of a Power BI visual. One visual requires binding data, and the other uses the *no data roles* feature and doesn't require binding data.

Binding data required

When binding data is required, the formatting settings are disabled if there are no data roles or the data wells are empty.



## How to create a visual that doesn't require data binding

### ⓘ Note

This feature is available from [API version 3.6.0](#) and above. To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.

To enable the No data binding feature, set the following two parameters in the `capabilities.json` file to `true`.

- `supportsLandingPage` allows you to display information on the Power BI card before it's loaded with data.
- `supportsEmpty DataView` allows Power BI updates when the values field is empty.

### JSON

```
{  
    "supportsLandingPage": true,
```

```
        "supportsEmptyDataView": true,  
    }
```

## Related content

- [Using capabilities](#)
- [Add a landing page](#)

---

## Feedback

Was this page helpful?



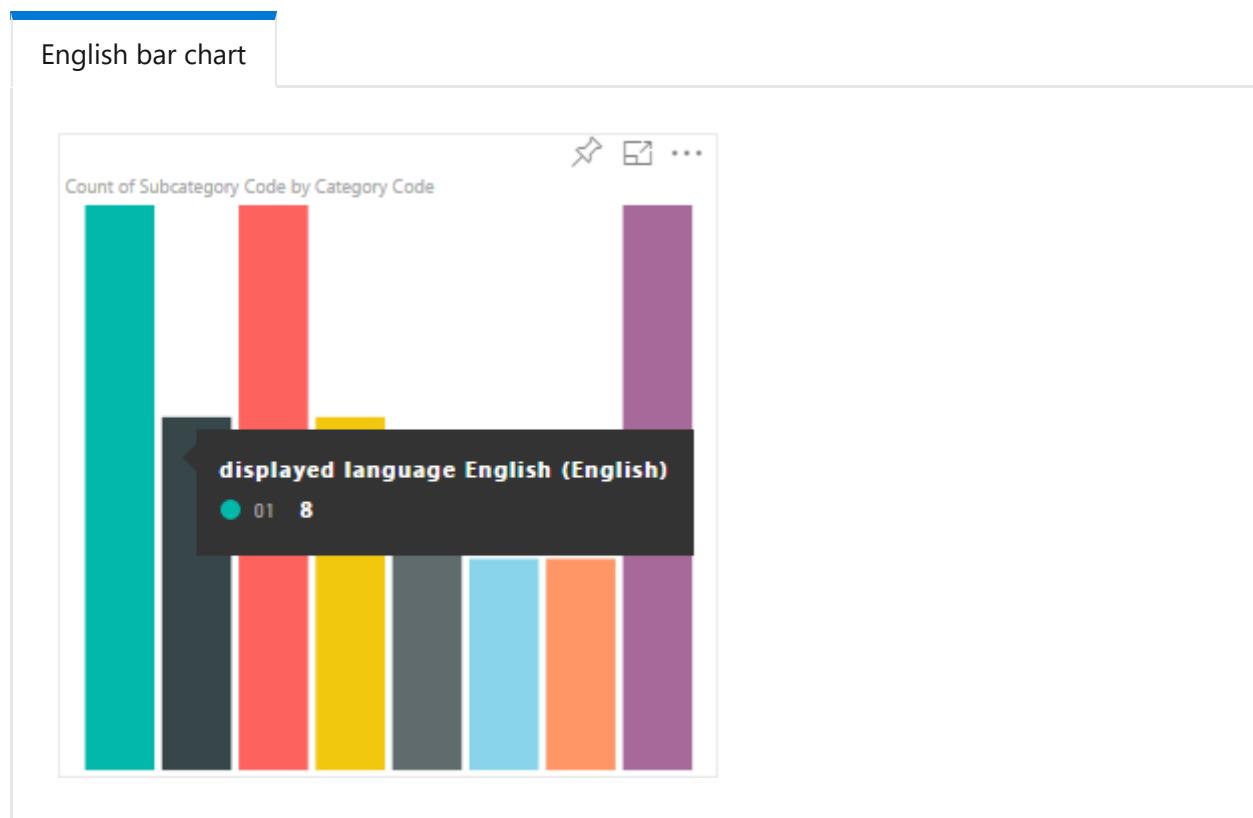
[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Add the local language to your Power BI visual

Article • 10/10/2024

Power BI [supports a range of local languages](#). You can retrieve the Power BI locale language, and use it to display content in your visual.

The following tabs show examples of the same *sample bar chart* visual displaying content in different languages. Each of these bar charts was created using a different locale language (English, Basque, and Hindi) which is displayed in the tooltip.



## ⓘ Note

- The localization manager in the visual's code is supported from API 1.10.0 and higher.
- Localization is not supported for debugging the visual while in development.

## How to add the local Power BI language to your visual

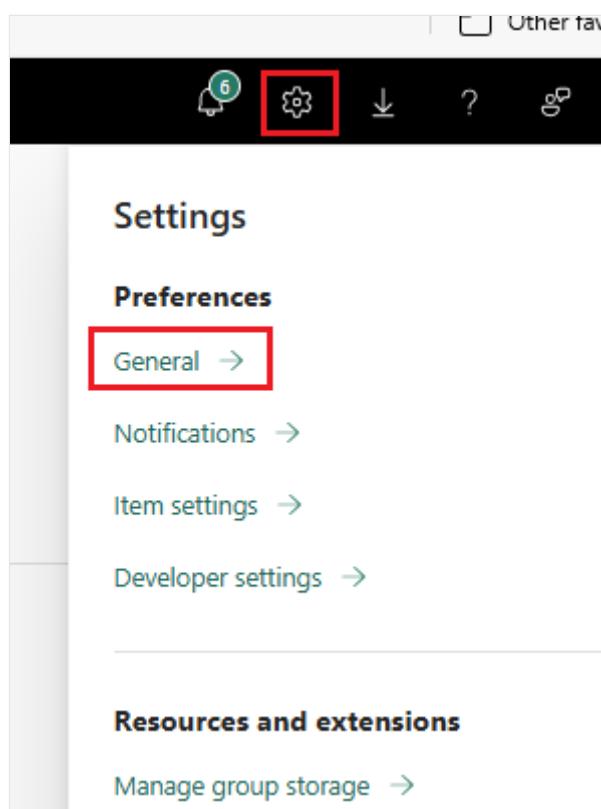
To add the local Power BI language to your visual, follow these steps:

1. Set up your environment to display a language that isn't English.
2. Get the local Power BI language.
3. Set the visual display names
4. Create a language folder.
5. Add a resources file for each language.
6. Create a new localizationManager instance.
7. Call the getDisplayName function.

## Step 1 - Set up your environment to display a language that isn't English

To test your visual, set Power BI to a language that isn't English. This section shows how to change the settings of Power BI Desktop and Power BI service, so that they use a local language that isn't English.

- **Power BI Desktop** - Download the localized version of Power BI desktop from <https://powerbi.microsoft.com>
- **Power BI service** - If you're using Power BI service (web portal), change your language in settings:
  1. Sign in to [PowerBI.com](#).
  2. Navigate to **Settings > General**.



3. Select **Select display language** to select the language you want Power BI to use.

The screenshot shows the 'Language' section of the Power BI settings. The 'Display language' dropdown is set to 'Default (browser language)'. A note says 'The language you select will appear in the interface and parts of the visuals.' A 'Select display language' button is highlighted with a red box. In the background, there's a 'Privacy' section with 'Q&A questions' turned on. A 'Close account' button is also visible. A modal window titled 'Languages' lists various languages: Português (Brasil), Português (Portugal), română, Русский, српски, srpski, slovenčina, slovenski, **✓ español** (highlighted with a red box), svenska, ไทย, Türkçe, and others. The 'Select' button at the bottom of the modal is also highlighted with a red box.

>

## Step 2 - Get the locale Power BI language

The local Power BI language is passed as a string called `locale` during the initialization of the visual. If a locale language is changed in Power BI, the visual is generated again in the new language.

TypeScript

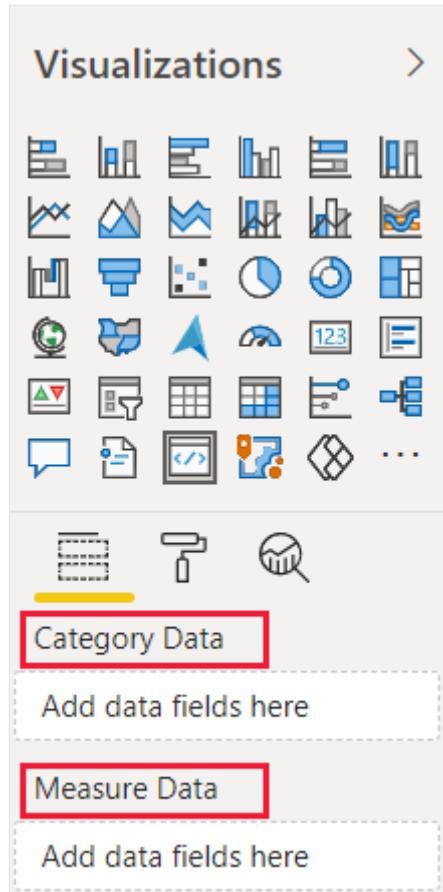
```
private locale: string;  
...  
this.locale = options.host.locale;
```

ⓘ Note

In Power BI Desktop, the `locale` property contains the language of the installed Power BI Desktop.

## Step 3 - Set the visual display names

Every visual displays information in the property pane. For example, a nonlocalized custom visual created by using the `pbviz new` command shows the *Category Data* and *Measure Data* fields in the property pane.



The property pane display fields are defined in the `capabilities.json` file. Every display field is defined using a `displayName` property. Add a `displayNameKey` to every display name you want to localize.

```
JSON

{
  "dataRoles": [
    {
      "displayName": "Category Data",
      "displayNameKey": "VisualCategoryDataNameKey1",
      "name": "category",
      "kind": "Grouping"
    },
    {
      "displayName": "Measure Data",
      "displayNameKey": "VisualMeasureDataNameKey2",
      "name": "measure",
      "kind": "Measure"
    }
  ]
}
```

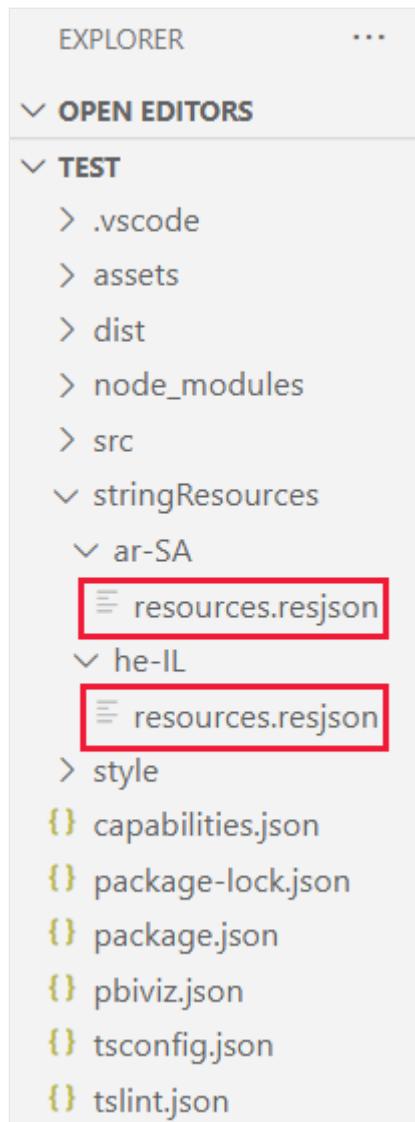
## Step 4 - Create a language folder

To create localized visuals, your project needs to have a language folder. In your project, create a folder called `stringResources`. The folder contains one sub folder for each local language you want your visual to support. For example, to support Arabic and Hebrew, add two folders in the following way:



## Step 5 - Add a resources file for each language

For each language you want your visual to support, add a `resources.resjson` JSON file in the appropriate `stringResources` sub folder. These files contain the locale language information, and the localized string values for every `displayNameKey` you want to replace.



Every JSON file defines a single [supported locale language](#). Add all the localization strings you're going to use into each `resources.resjson` file.

## Examples

- `resources.resjson` file with *Russian* strings for each `displayNameKey`.

```
JSON

{
  ...
  "Role_Legend": "Обозначения",
  "Role_task": "Задача",
  "Role_StartDate": "Дата начала",
  "Role_Duration": "Длительность"
  ...
}
```

- `resources.resjson` file with *Hebrew* strings for each `displayNameKey`.

## JSON

```
{  
    ...  
    "Role_Legend": "אקורדיון",  
    "Role_task": "משימה",  
    "Role_StartDate": "תאריך תחילת",  
    "Role_Duration": "משך זמן"  
    ...  
}
```

## Step 6 - Create a new localizationManager instance

Create a new `localizationManager` instance in your visual's code.

### TypeScript

```
private localizationManager: ILocalizationManager;  
  
constructor(options: VisualConstructorOptions) {  
    this.localizationManager = options.host.createLocalizationManager();  
}
```

## Step 7 - Call the getDisplayName function

After creating a new `localizationManager` instance, you can call the localization manager's `getDisplayName` function with the string key argument you defined in `resources.resjson`.

For example, the following code returns *Legend* for `en-US`, and *Обозначения* for `ru-RU`.

### TypeScript

```
let legend: string = this.localizationManager.getDisplayName("Role_Legend");
```

## Format pane and analytics pane localization

### ⓘ Note

Relevant to API version 5.1+

To support localization on format pane and analytics pane components, set localized string as follows:

#### TypeScript

```
displayName: this.localization.getDisplayName("Font_Color_DisplayNameKey");  
description: this.localization.getDisplayName("Font_Color_DescriptionKey");
```

For localize formatting model see [format pane localization](#).

For localize formatting model utils see [formatting model utils - localization](#).

## Supported languages

The following table contains a list of all the languages supported in Power BI, and the string that the `locale` variable returns for each one.

[\[+\] Expand table](#)

Locale string	Language
ar-SA	العربية (Arabic)
bg-BG	български (Bulgarian)
ca-ES	català (Catalan)
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)

<b>Locale string</b>	<b>Language</b>
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)
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	slovenški (Slovenian)
sr-Cyrl-RS	српски (Serbian)
sr-Latn-RS	srpski (Serbian)
sv-SE	svenska (Swedish)
th-TH	ไทย (Thai)
tr-TR	Türk (Turkish)

Locale string	Language
uk-UA	український (Ukrainian)
vi-VN	tiếng Việt (Vietnamese)
zh-CN	中国 (Chinese-Simplified)
zh-TW	中國 (Chinese-Traditional)

## Related content

[Formatting utils](#)

Questions? Ask the Power BI community 

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)  | [Ask the community](#) 

# Tutorial: Add unit tests for Power BI visual projects

Article • 12/05/2024

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 Node.js environment

The examples in this article use the [bar chart](#) visual for testing.

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

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

JSON

```
"@types/jasmine": "^5.1.5",
"@types/karma": "^6.3.9",
"coverage-istanbul-loader": "^3.0.5",
"jasmine": "^5.5.0",
"karma": "^6.4.4",
"karma-chrome-launcher": "^3.2.0",
"karma-coverage": "^2.2.1",
"karma-coverage-istanbul-reporter": "^3.0.3",
"karma-jasmine": "^5.1.0",
"karma-junit-reporter": "^2.0.1",
"karma-sourcemap-loader": "^0.4.0",
"karma-typescript": "^5.5.4",
"karma-typescript-preprocessor": "^0.4.0",
"karma-webpack": "^5.0.1",
"playwright-chromium": "^1.49.0",
"powerbi-visuals-api": "~5.11.0",
"powerbi-visuals-tools": "^5.6.0",
"powerbi-visuals-utils-testutils": "6.1.1",
"powerbi-visuals-utils-typeutils": "6.0.3",
```

```
"style-loader": "^4.0.0",
"ts-loader": "~9.5.1"
```

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

Save the *package.json* file and run the following command at the location of the *package.json* file:

Windows Command Prompt

```
npm install
```

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

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

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

TypeScript

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

module.exports = {
  devtool: 'source-map',
  mode: 'development',
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/
      },
      {
        test: /\.json$/,
        loader: 'json-loader'
      },
      {
        test: /\.tsx?$/i,
        enforce: 'post',
        include: path.resolve(__dirname, 'src'),
        exclude: /(node_modules|resources\js\vendor)/,
        loader: 'coverage-istanbul-loader',
        options: { esModules: true }
      },
      {
        test: /\.less$/,
        use: [
          {
            loader: 'style-loader'
          },
        ]
      }
    ]
  }
};
```

```

        {
            loader: 'css-loader'
        },
        {
            loader: 'less-loader',
            options: {
                lessOptions: {
                    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 *test.tsconfig.json* file:

JSON

```
{
  "compilerOptions": {
    "allowJs": false,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "target": "es2022",
    "sourceMap": true,
    "outDir": "./.tmp/build/",
    "sourceRoot": "../../src/",
    "moduleResolution": "node",
    "declaration": true,
    "lib": [
      "es2022",
      "dom"
    ],
  },
  "files": [

```

```
    "./test/visualTest.ts"
  ],
  "include": [
    "src/*.ts"
  ]
}
```

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

TypeScript

```
"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("playwright-
chromium").chromium.executablePath();

module.exports = (config) => {
  config.set({
    mode: "development",
    browserNoActivityTimeout: 100000,
    browsers: ["ChromeHeadless"], // or specify Chrome to use the
locally installed Chrome browser
    colors: true,
    frameworks: ["jasmine", "webpack"],
    reporters: [
      "progress",
      "junit",
      "coverage",
      "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: [
    testRecursivePath,
    {
      pattern: srcOriginalRecursivePath,
      included: false,
      served: true
    },
    {
      pattern: './capabilities.json',
      watched: false,
      served: true,
      included: false
    }
  ],
  preprocessors: {
    [testRecursivePath]: ["webpack"]
  },
  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: {
    type: "html",
    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 variables:

- `testRecursivePath`: Locates the test code.
- `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. The Karma framework 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. The files you load are typically source files, test cases, and libraries (such as Jasmine or test utilities). You can add more files as necessary.
- `preprocessors`: In this section, you configure actions that run before the unit tests run. The actions can precompile TypeScript to JavaScript, prepare source map files, and generate a code coverage report. You can disable `coverage` when you debug your tests. `coverage` generates more code for code coverage testing, 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` in `package.json`:

JSON

```
{  
    "scripts": {  
        "pbiviz": "pbiviz",  
        "start": "pbiviz start",  
        "package": "pbiviz package",  
        "pretest": "pbiviz package --resources --no-minify --no-pbiviz",  
        "test": "karma start",  
        "debug": "karma start --single-run=false --browsers=Chrome"  
    }  
}
```

```
    }  
    ...  
}
```

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 the visual.

### Create a visual instance builder

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

TypeScript

```
import { VisualBuilderBase } from "powerbi-visuals-utils-testutils";  
  
import { BarChart as VisualClass } from "../src/barChart";  
  
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(): SVGELEMENT | null {  
        return this.element.querySelector("svg.barChart");  
    }  
}
```

The `build` method creates an instance of your visual. `mainElement` is a get method, which returns an instance of a 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 example 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:

TypeScript

```
import powerbi from "powerbi-visuals-api";

import { BarChartBuilder } from "./visualBuilder";
import { SampleBarChartDataBuilder } from "./visualData";

import DataView = powerbi.DataView;

describe("BarChart", () => {
  let visualBuilder: BarChartBuilder;
  let dataView: DataView;
  let defaultDataViewBuilder: SampleBarChartDataBuilder;

  beforeEach(() => {
    visualBuilder = new BarChartBuilder(500, 500);
    defaultDataViewBuilder = new SampleBarChartDataBuilder();
    dataView = defaultDataViewBuilder.getDataView();
  });

  it("root DOM element is created", () => {
    visualBuilder.updateRenderTimeout(dataView, () => {
      expect(document.body.contains(visualBuilder.mainElement)).toBeTruthy();
    });
  });
});
```

Several Jasmine methods are called:

- `describe`: Describes a test case. In the context of the Jasmine framework, `describe` 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`: Is 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.

# How to add static data for unit tests

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

TypeScript

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

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

import TestDataViewBuilder = testDataViewBuilder.TestDataViewBuilder;

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

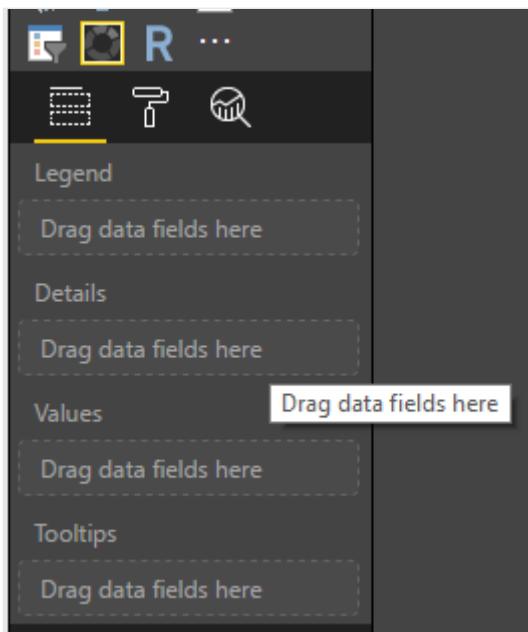
    public getDataView(columnNames?: string[]): DataView {
        let dataView: any = this.createCategoricalDataViewBuilder(
            [
                ...
            ],
            [
                ...
            ],
            columnNames
        ).build();

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

        return dataView;
    }
}
```

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 access to Power BI core functions that you normally use to reproduce the data. But you need to map your static data to the categorical `dataView`.

Use the `TestDataViewBuilder` class to map your static data.

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

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

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

```
JSON

"dataRoles": [
    {
        "displayName": "Category Data",
        "name": "category",
        "kind": "Grouping"
    },
    {
        "displayName": "Measure Data",
        "name": "measure",
        "kind": "Measure"
    },
    {
        "displayName": "ToolTips",
        "name": "ToolTips",
        "kind": "Measure"
    }
],
"dataViewMappings": [
    {
        "conditions": [
            ...
        ],
        "view": {
            "categorical": [
                ...
            ],
            "dataRoles": [
                ...
            ]
        }
    }
]
```

```

        {
            "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 parameters to the `createCategoricalDataViewBuilder` method:

TypeScript

```

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

```

```

        roles: {
            Measure: true
        },
    },
    values: this.valuesMeasure
},
], columnNames)

```

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

ts

```

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

```

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

ts

```

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

```

The final version of `visualData.ts` contains the following code:

ts

```

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

import { testDataViewBuilder } from "powerbi-visuals-utils-testutils";
import { valueType } from "powerbi-visuals-utils-typeutils";
import ValueType = valueType.ValueType;

import TestDataViewBuilder = testDataViewBuilder.TestDataViewBuilder;

export class SampleBarChartDataBuilder extends TestDataViewBuilder {
    public static CategoryColumn: string = "category";
    public static MeasureColumn: string = "measure";
    public valuesCategory: string[] = [
        "Monday",
        "Tuesday",
        "Wednesday",
        "Thursday",
        "Friday",
        "Saturday",
        "Sunday",
    ];
    public valuesMeasure: number[] = [
        742731.43, 162066.43, 283085.78, 300263.49, 376074.57, 814724.34,
        570921.34,
    ];
}

```

```

];
public getDataView(columnNames?: string[]): DataView {
  let dataView: any = this.createCategoricalDataViewBuilder(
    [
      {
        source: {
          displayName: "Category",
          queryName: SampleBarChartDataBuilder.CategoryColumn,
          type: ValueType.fromDescriptor({ text: true }),
          roles: {
            category: true,
          },
        },
        values: this.valuesCategory,
      },
    ],
    [
      {
        source: {
          displayName: "Measure",
          isMeasure: true,
          queryName: SampleBarChartDataBuilder.MeasureColumn,
          type: ValueType.fromDescriptor({ numeric: true }),
          roles: {
            measure: true,
          },
        },
        values: this.valuesMeasure,
      },
    ],
    columnNames
  ).build();

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

  return dataView;
}
}

```

The `ValueType` class is defined in the `powerbi-visuals-utils-typeutils` package.

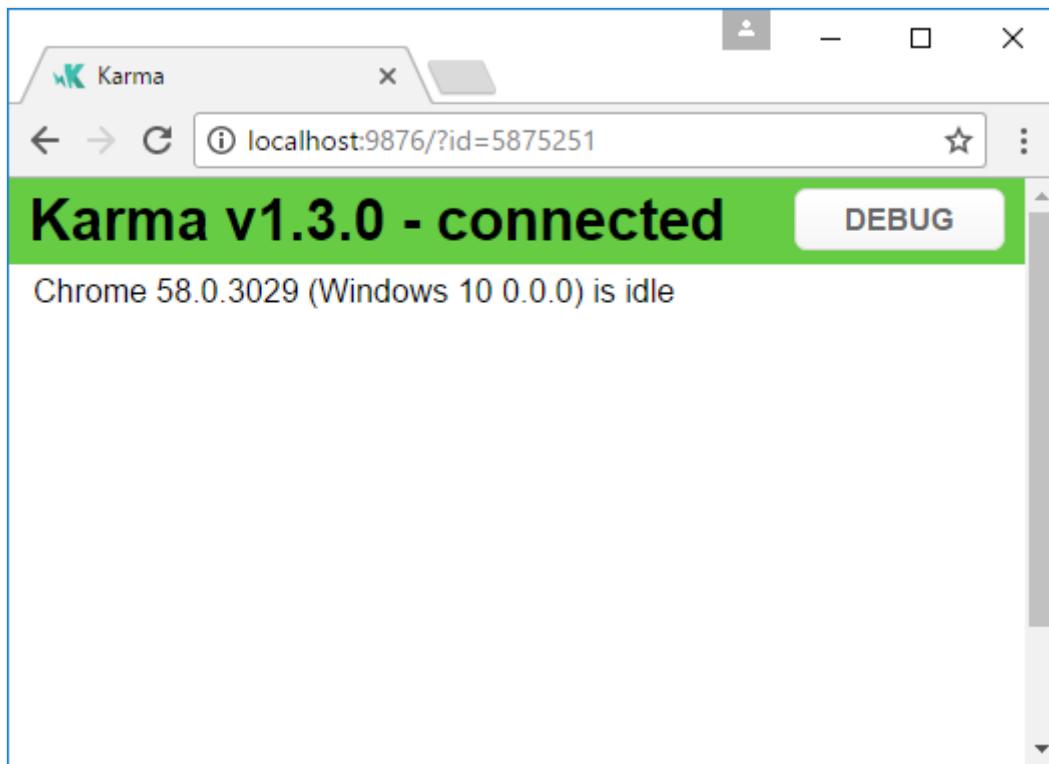
Now, you can run the unit test.

## Launch unit tests

This test checks that the root SVG element for your visual exists when the visual runs. To run the unit test, enter the following command in the command-line tool:

```
Windows Command Prompt
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:

```
Windows Command Prompt
> karma start

Webpack bundling...
assets by status 8.31 KiB [compared for emit]
  assets by path ..../build/test/*.ts 1020 bytes
    asset ..../build/test/visualData.d.ts 512 bytes [compared for emit]
    asset ..../build/test/visualBuilder.d.ts 499 bytes [compared for emit]
    asset ..../build/test/visualTest.d.ts 11 bytes [compared for emit]
  assets by path ..../build/src/*.* 6.67 KiB
    asset ..../build/src/barChart.d.ts 4.49 KiB [compared for emit]
    asset ..../build/src/barChartSettingsModel.d.ts 2.18 KiB [compared for
```

```

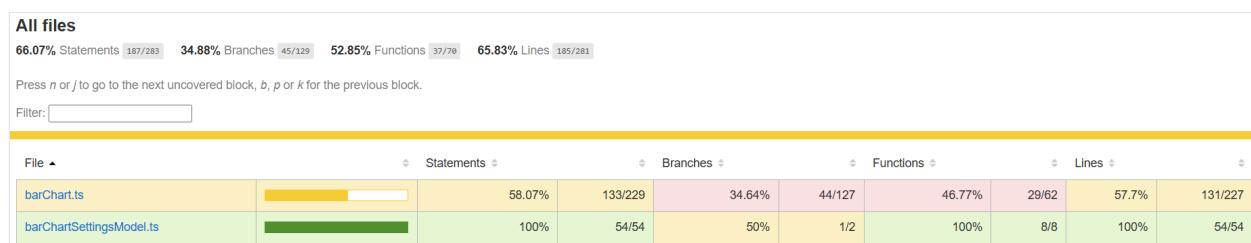
emit]
  asset visualTest.3941401795.js 662 bytes [compared for emit] (name:
visualTest.3941401795) 1 related asset
assets by status 2.48 MiB [emitted]
  asset commons.js 2.48 MiB [emitted] (name: commons) (id hint: commons) 1
related asset
  asset runtime.js 6.48 KiB [emitted] (name: runtime) 1 related asset
Entrypoint visualTest.3941401795 2.48 MiB (2.34 MiB) = runtime.js 6.48 KiB
commons.js 2.48 MiB visualTest.3941401795.js 662 bytes 3 auxiliary assets
webpack 5.97.0 compiled successfully in 3847 ms
04 12 2024 11:01:19.255:INFO [karma-server]: Karma v6.4.4 server started at
http://localhost:9876/
04 12 2024 11:01:19.257:INFO [launcher]: Launching browsers ChromeHeadless
with concurrency unlimited
04 12 2024 11:01:19.277:INFO [launcher]: Starting browser ChromeHeadless
04 12 2024 11:01:20.634:INFO [Chrome Headless 131.0.0.0 (Windows 10)]:Connected on socket QYSj9NyHQ14QjFBoAAAB with id 9616879
Chrome Headless 131.0.0.0 (Windows 10): Executed 1 of 1 SUCCESS (0.016 secs
/ 0.025 secs)
TOTAL: 1 SUCCESS
TOTAL: 1 SUCCESS

```

#### ===== Coverage summary =====

<b>Statements</b>	: 66.07% ( 187/283 )
<b>Branches</b>	: 34.88% ( 45/129 )
<b>Functions</b>	: 52.85% ( 37/70 )
<b>Lines</b>	: 65.83% ( 185/281 )

To learn more about current code coverage, open the [coverage/html-report/index.html](#) file.



In the file scope, you can view the source code. The `coverage` utilities highlight the row in red if certain lines of code don't run during the unit tests.

```

609 }
610
611 14×     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           14×       let dataLabelManager = new DataLabelManager();
619           14×       let filteredData = dataLabelManager.hideCollidedLabels(viewport, data, layout, true /* addTransform */);
620
621           14×       [I] if (filteredData.length === 0) {
622           14×           dataLabelUtils.cleanDataLabels(context, true);
623           14×           return;
624       }
625
626           // Draw labels
627           14×       [E] if (context.select(AsterPlot.labelGraphicsContextClass.selector).empty())
628           14×           context.append("g").classed(AsterPlot.labelGraphicsContextClass.class, true);
629
630           14×       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           14×       labels.enter().append("text").classed("data-labels", true);
637
638           14×       [I] if (!labels)
639           14×           return;
640
641           14×       labels
642           92×           .attr({x: (d: LabelEnabledDataPoint) => d.labelX, y: (d: LabelEnabledDataPoint) => d.labelY, dy: ".35em"})

```

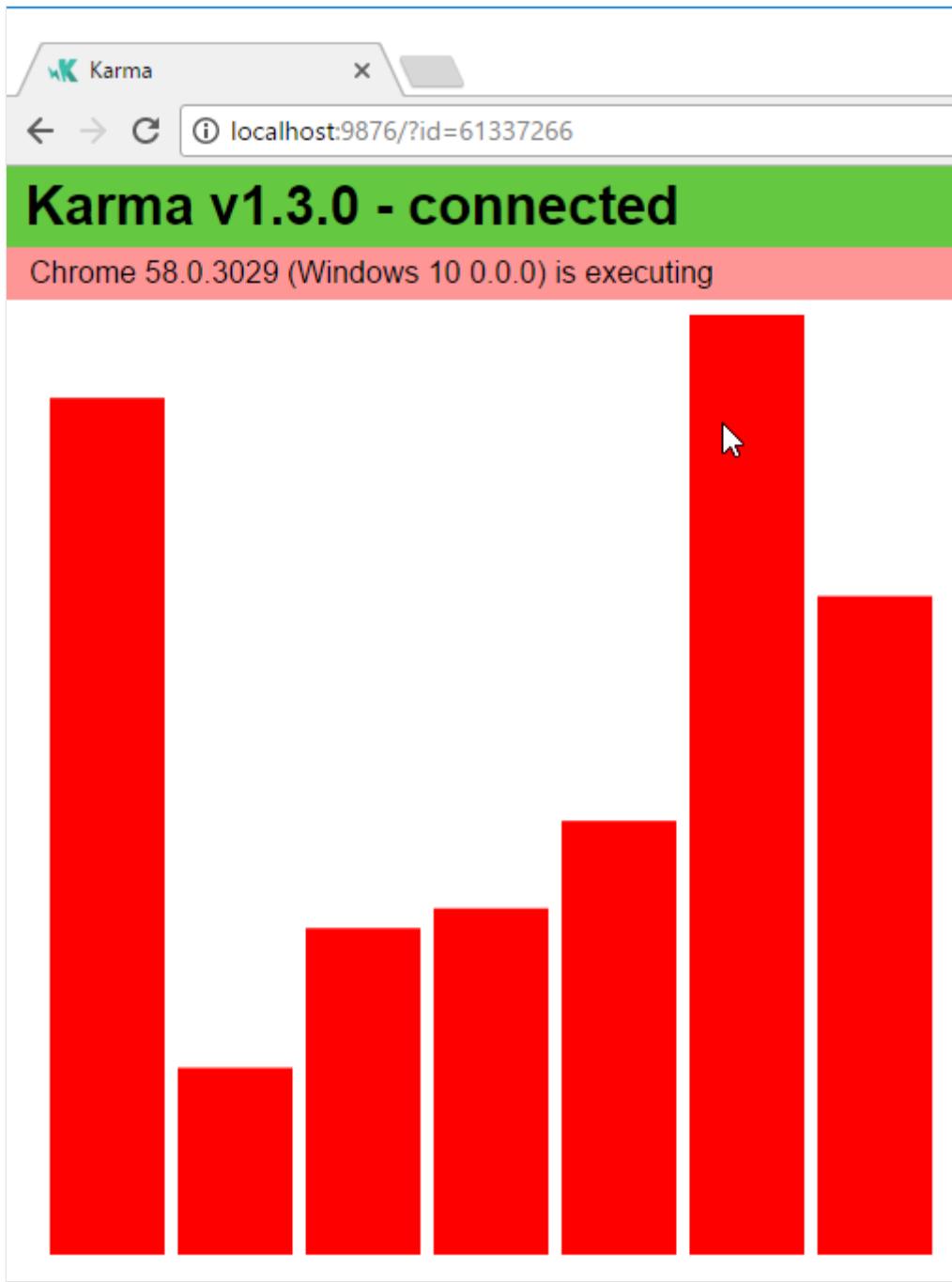
## Important

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/barChart.ts`.

## Debugging

To debug your tests via browser console, change the `singleRun` value in `karma.conf.ts` to `false`. This setting will keep your browser running when the browser launches after the tests run.

Your visual opens in the Chrome browser.



## Related content

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

## Feedback

Was this page helpful?

Yes

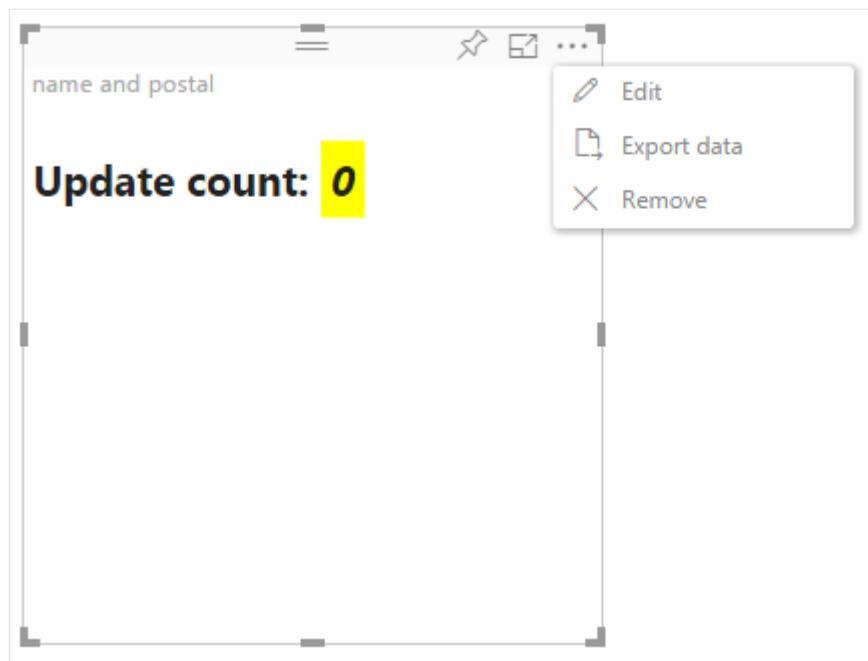
No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Advanced edit mode in Power BI visuals

Article • 02/05/2025

Advanced edit mode enables you to use advanced UI controls in your Power BI visual. From report editing mode, select the **Edit** button on a visual and set the edit mode to **Advanced**. The visual uses the `EditMode` flag to determine if it should display this UI control.



By default, the visual doesn't support advanced edit mode (`"advancedEditModeSupport: 0"`). To enable Advanced edit mode, add a line to the visual's `capabilities.json` file by setting the `advancedEditModeSupport` property.

The possible values are:

- `0` - **NotSupported**. The visual doesn't support Advanced edit mode. The **Edit** button is not displayed on this visual.
- `1` - **SupportedNoAction**. The visual supports Advanced edit mode and doesn't require any further changes. Power BI doesn't switch the visual to *Focus Mode*. Developers can use this setting as an external button to run several processes in the same viewport.
- `2` - **SupportedInFocus**. The visual supports Advanced edit mode and requires the host to enter Focus mode when entering Advanced edit mode.

## 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 doesn't appear.

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`: The host doesn't require further action.
- `SupportedInFocus`: The host pops out the visual in Focus mode.

You can read more about how to configure the `capabilities.json` file in [Capabilities and properties of Power BI visuals](#).

## Exit advanced edit mode

The **Back to report** button appears if the `advancedEditModeSupport` property is set in the `capabilities.json` file to `SupportedInFocus`.

## Related content

[Add conditional formatting](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Package a Power BI visual

Article • 06/03/2024

Before you can load your custom visual into [Power BI Desktop](#) or share it with the community in the [Power BI Visual Gallery](#), you need to package it. Using this tutorial, you will:

- Provide [property values](#) and metadata for the visual.
- Update the icon.
- [Package](#) the custom visual.

## Enter property values

1. In **PowerShell**, stop the visual if it's running.
2. In **VS Code**, navigate to the root folder of your visual project and open the `pbviz.json` file.
3. In the `visual` object, set the `displayName` value to what you want to be your visual's display name.

```
1  {
2    "visual": {
3      "name": "circleCard",
4      "displayName": "Circle Card",
5      "guid": "4a2a2a2a-2a2a-4a2a-a2a2-2a2a2a2a2a2a",
6      "visualClassName": "Visual",
```

The visual's display name appears in the **Visualizations** pane of Power BI when you hover the cursor over the visual icon.

4. Fill in or modify the following fields in the `pbviz.json` file:

- `visualClassName`
- `description`

`visualClassName` is optional, but `description` must be filled in for the package command to run.

5. Fill in `supportUrl` and `gitHubUrl` with the URLs that a user can visit to get support and view your visual's GitHub project.

The following code shows `supportUrl` and `gitHubUrl` examples:

## JSON

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

6. Enter your name and email in the `author` object.

7. Save the `pbviz.json` file.

## Update the icon (optional)

1. In the `pbviz.json` file, notice that the document defines a path to an icon in the `assets` object. The icon is the image that appears in the **Visualizations** pane in Power BI. It must be a PNG format file and 20 x 20 pixels.
2. In **Windows Explorer**, copy the `icon.png` file you want to use, and then paste it to replace the default `icon.png` file located in the `assets` folder.
3. In **VS Code**, in the Explorer pane, expand the `assets` folder, and then select the `icon.png` file.
4. Review the icon.



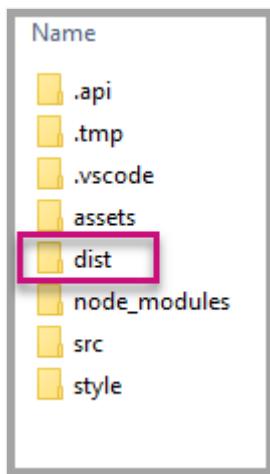
## Package the visual

1. In **VS Code**, ensure that all files are saved.
2. In **PowerShell**, enter the following command to generate a `pbviz` file:

## PowerShell

```
pbviz package
```

This command creates a `pbviz` file in the `/dist/` directory of your visual project, and overwrites any previous `pbviz` file that might exist.



The package outputs 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 packaged the custom visual, and it's ready for use.

## Related content

[Publish Power BI visuals to Partner Center](#)

[Import a custom visual](#)

---

## Feedback

Was this page helpful?

[Yes](#)

[No](#)

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Test a Power BI custom visual before submitting it for publication

Article • 06/09/2024

Before you publish your visual to [AppSource](#), it must pass the tests listed in this article. It's important to test your visual before you submit it. If your visual doesn't pass the required test cases, it will be rejected.

For more information about the publishing process, see [Publish Power BI visuals to Partner Center](#).

## Testing a new version of a published visual

By default, Power BI loads the latest published version of the visual from AppSource, even if you import the visual from a local file. Version numbers consist of four digits in the following format: x.x.x.x.

When testing or updating a visual that's already published, make sure you're using the correct version of the visual **without changing the GUID**. To override the AppSource version with a local file version, enable Developer mode in either Power BI Desktop or Power BI service.

### Important

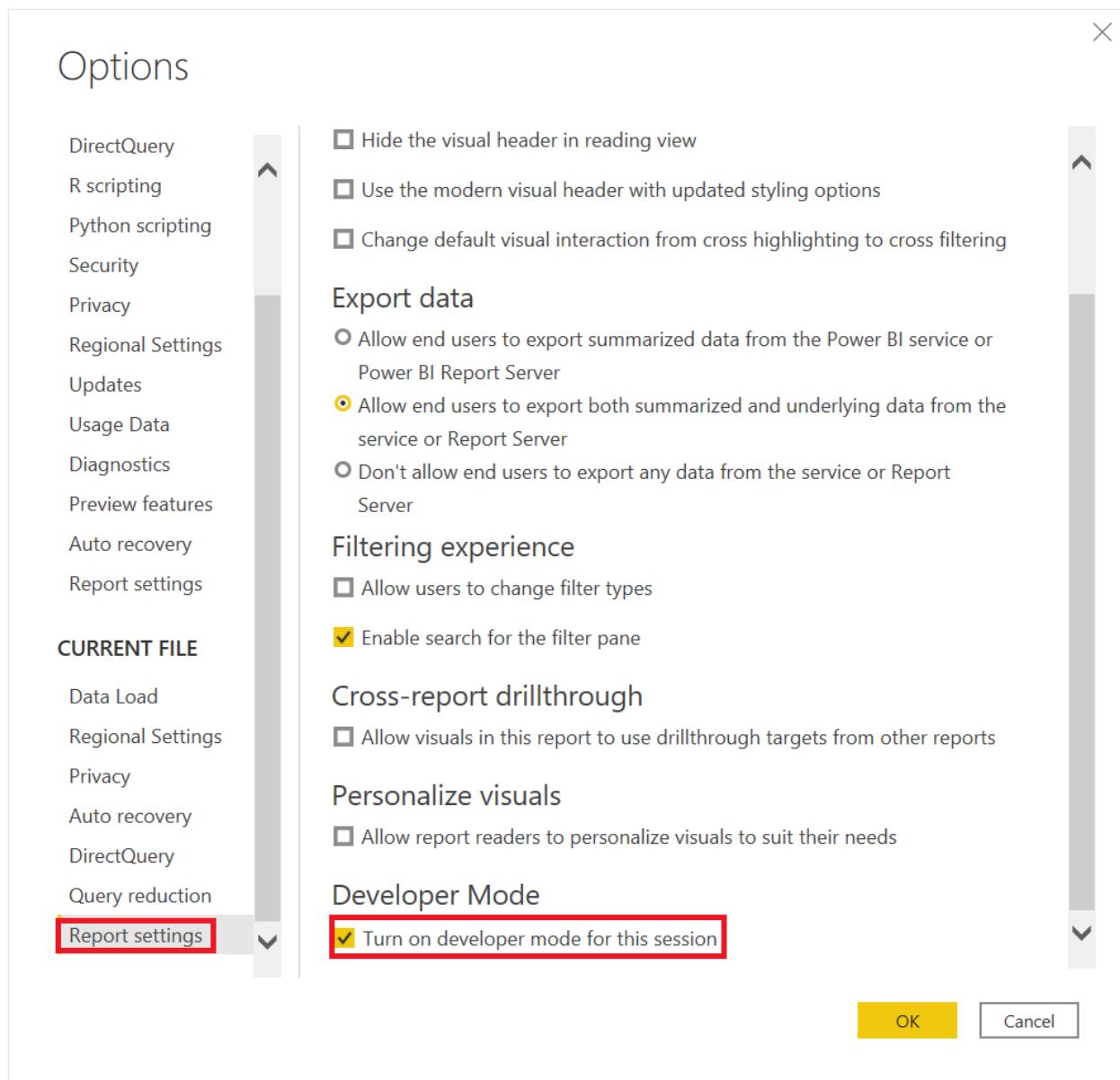
When testing or debugging a new version of a visual that's available in AppSource, **do not change the GUID of the visual**. Use Developer mode instead.

## Enable Developer mode in Power BI Desktop

In Power BI Desktop, Developer mode is valid for only one session. If you open a new Power BI Desktop instance for testing, you need to enable Developer mode again.

To enable Developer mode, follow these steps:

1. Open Power BI Desktop.
2. Select **File > Options and settings**.
3. Select **Options**.
4. In the Options window, from the CURRENT FILE list, select **Report settings**.
5. In Developer Mode, select the **Turn on developer mode for this session** option.



## Enable Developer mode in Power BI service

In Power BI service, Developer mode is kept per user account. Whenever a user loads the package from the local file, Power BI will ignore the AppSource version of the visual.

To enable Developer mode in Power BI service, follow the instructions in [Set up Power BI service for developing a visual](#).

## General test cases

Verify that your visual passes the general test cases.

[ ] [Expand table](#)

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 <b>Ctrl</b> , <b>Alt</b> , and <b>Shift</b> 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.
Resize your visual.	Visual reacts correctly to resizing.
Set the report size to the minimum.	There's no display errors.
Ensure scroll bars work correctly.	Scroll bars should exist, if necessary. 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

Test case	Expected results
	scroll bars are needed for different sizes of the visual.
Pin your visual to a <b>Dashboard</b> .	The visual displays properly.
Add multiple versions of your visual to a single report page.	All versions of the visual display and operate properly.
Add multiple versions of your visual to multiple report pages.	All versions of the visual display and operate properly.
Switch between report pages.	The visual displays properly.
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.	Toolips 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	All displays and features work correctly.

Test case	Expected results
types.	

## Optional browser testing

The AppSource team validates a visual on the most current Windows versions of Google Chrome, Microsoft Edge, and Mozilla Firefox browsers. Optionally, test your visual in the following browsers.

[\[+\] Expand table](#)

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](#).

[Expand table](#)

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 its performance. Don't rely on visual cues and the console time logs.

[Expand table](#)

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. Check out these <a href="#">tips for optimal performance</a> .

## Related content

- [Publish Power BI visuals to Partner Center](#)
- [Get a Power BI visual certified](#)
- [Performance tips for creating quality Power BI custom visuals](#)

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

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

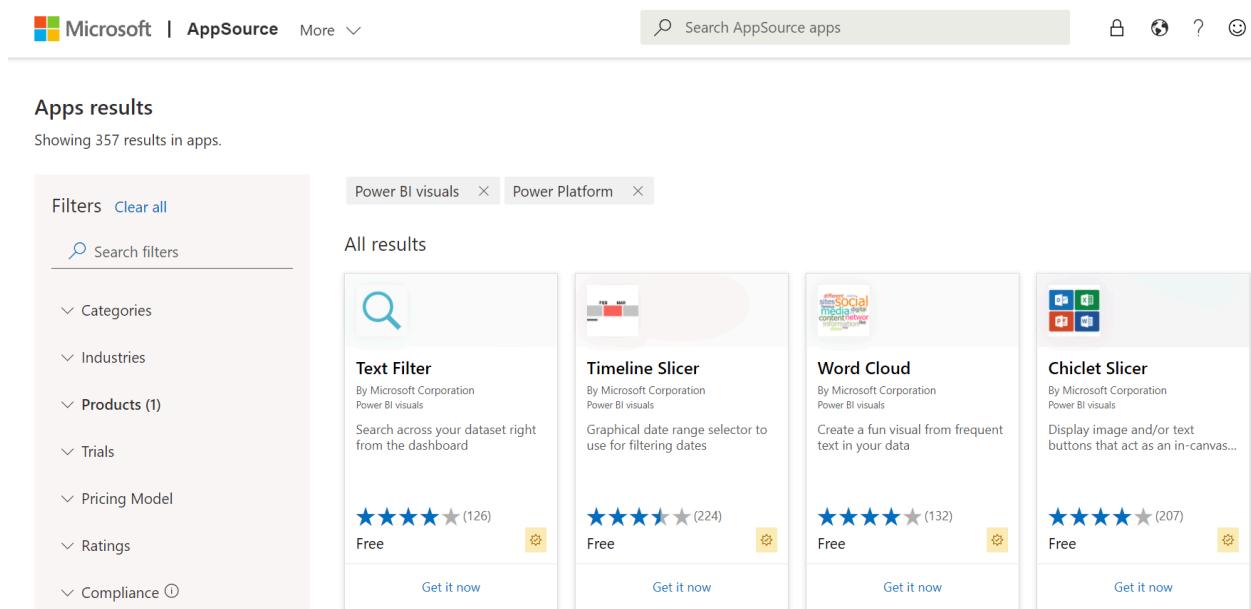
# Publish Power BI visuals to the Microsoft commercial marketplace

Article • 06/09/2024

Once you have created your Power BI visual, you may want to publish it to AppSource for others to discover and use. For a detailed explanation of how to create a Power BI visual offer, see [Plan a Power BI visual offer](#).

## What is AppSource?

[AppSource](#) is the place to find SaaS apps and add-ins for your Microsoft products and services. You can find many Power BI visuals here.



The screenshot shows the Microsoft AppSource interface. At the top, there's a navigation bar with the Microsoft logo, a search bar labeled "Search AppSource apps", and icons for account, globe, help, and smiley face. Below the navigation is a section titled "Apps results" with the subtext "Showing 357 results in apps." On the left, a sidebar titled "Filters" includes "Clear all" and a "Search filters" input. It lists categories like "Categories", "Industries", "Products (1)", "Trials", "Pricing Model", "Ratings", and "Compliance". In the center, there's a search bar with filters "Power BI visuals" and "Power Platform". Below this is a grid of four app cards under "All results": "Text Filter" (by Microsoft Corporation, Power BI visuals, 126 reviews, free, Get it now), "Timeline Slicer" (by Microsoft Corporation, Power BI visuals, 224 reviews, free, Get it now), "Word Cloud" (by Microsoft Corporation, Power BI visuals, 132 reviews, free, Get it now), and "Chiclet Slicer" (by Microsoft Corporation, Power BI visuals, 207 reviews, free, Get it now).

## Prerequisite

To submit your Power BI visual, you must be enrolled with [Partner Center](#). If you're not yet enrolled, [Open a developer account in Partner Center](#).

## Prepare to submit your Power BI visual

Before submitting a Power BI visual to AppSource, ensure that it complies with the [Power BI visuals guidelines](#).

 **Important**

If you're resubmitting or updating a visual, **do not change its GUID**. Follow these instructions to [test a new version of a visual](#).

When you're ready to submit your Power BI visual, verify that your visual meets all the following requirements.

[+] [Expand table](#)

Item	Required	Description
Pbiviz package	Yes	<p><a href="#">Pack your Power BI visual into a .pbiviz package</a>. Ensure the <i>pbiviz.json</i> file contains all the required metadata:</p> <ul style="list-style-type: none"><li>- Visual name</li><li>- Display name</li><li>- GUID</li><li>- Version (four digits: <code>x.x.x.x</code>)</li><li>- Description</li><li>- Support URL</li><li>- Author name and email</li></ul>
Sample .pbix report file	Yes	<p>To help users become familiar with the visual, highlight the value that the visual brings to the user and give examples of usage and formatting. You can also add a "<i>hints</i>" page at the end with some tips and tricks and things to avoid.</p> <p>The sample .pbix report file must work offline, without any external connections.</p>
Logo	Yes	<p>Include the custom visual logo that will appear in the Marketplace listing. It should be in PNG format and exactly 300 x 300 px.</p> <p><b>Important!</b> Review the <a href="#">AppSource store images guide</a> carefully, before submitting the logo.</p>
Screenshots	Yes	<p>Provide at least one screenshot, and up to five, in PNG format. The dimensions must be exactly 1366 px (width) by 768 px (height), and the size not larger than 1024 kb.</p> <p>Add text bubbles to explain the value of key features shown in each screenshot.</p>
Support download link	Yes	<p>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 URL should start with <code>https://</code>.</p>
Privacy document link	Yes	<p>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 URL should start with <code>https://</code>.</p>

Item	Required	Description
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 URL should start with <a href="https://">https://</a> .

## Submit or update your custom visual to AppSource

To submit a Power BI visual to AppSource, upload a `.pbviz` package and `.pbix` file to Partner Center.

Before you create the `.pbviz` package, complete the following fields in the `pbviz.json` file:

- description
- supportUrl
- author
- name
- email

### Note

**Individual publishers** can use one of these methods to submit a Power BI visual:

- If you have an old Seller Dashboard account, you can continue using this account's credentials to sign into Partner Center.
- If you don't have an old Seller Dashboard account, and are not registered to Partner Center, you'll need to [Open a developer account in Partner Center](#) using your work email.

### Important

Before you submit your visual, make sure it passes all the [requirements](#). A visual that doesn't pass the requirements will be rejected.

When you're ready to create or update your offer, follow the instructions in [Create a Power BI app offer](#).

## Track submission status and usage

- Review the [validation policies](#).
- To understand when your Power BI visual will be available to download from AppSource, review the Power BI visuals [publication timeline](#).

## Certify your visual

Certified Power BI visuals are visuals in the Marketplace that meet certain specified code [requirements](#) that the Microsoft Power BI team has tested and approved. To request certification, select the Request Power BI certification check box. We recommend that you submit and publish your Power BI visual before you request certification, because the certification process can take time. When you request certification, be sure to provide all required certification information in the Notes for certification box on the Review and publish page. All certified visuals must pass all the [certification requirements](#).

## Related content

- [Performance tips for creating quality Power BI custom visuals](#)
- [Guidelines for publishing Power BI visuals](#)
- [Test your Power BI visual before submitting for certification](#)

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

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

# Certified Power BI visuals

06/23/2025

Certified Power BI visuals are Power BI visuals in [AppSource](#) that meet the Microsoft Power BI team [code requirements](#) and testing. The tests performed are designed to check that the visual doesn't access external services or resources. However, Microsoft isn't the author of third-party custom visuals, and we advise customers to contact the author directly to verify the functionality of these visuals.

Certified Power BI visuals can be used like any other Power BI visual. They offer more features than noncertified visuals. For example, you can [export them to PowerPoint](#), or display the visual in received emails when a user [subscribes to report pages](#).

The certification process is optional. It's up to the developers to decide if they want their visual certified. Power BI visuals that aren't certified, aren't necessarily unsafe. 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 can't be certified.

## Removal of certification

Microsoft reserves the right to remove a visual from the certified list, at its discretion.

## Certification requirements

To get your Power BI visual certified, it must meet the requirements listed in this section.

### General requirements

Your Power BI visual has to be approved by Partner Center. Before requesting certification, we recommend that you publish your Power BI visual 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 for certification, verify that:

- The visual isn't an R-visual
- The visual complies with the [guidelines for Power BI visuals](#)
- The visual passes all the [required tests](#)
- 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:

- 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 also provide access to the 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 can't include the `node_modules`, `.tmp`, or `dist` folders.
- **capabilities.json** - If you're submitting a newer version of an existing Power BI visual with changes to the properties in this file, verify that they don't break reports for existing users.
- **pbviz.json**
- **package.json**. The visual must have the following package installed:
  - `"typescript"`
  - `"eslint"`
  - `"eslint-plugin-powerbi-visuals"`
  - The file must contain a command for running linter - `"eslint": "npx eslint . --ext .js,.jsx,.ts,.tsx"`
- **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.
- `ESlint` 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.

Compile your Power BI visual with `pbviz package`. If you're using your own build scripts, provide a `npm run package` custom build command.

### Tip

Starting from powerbi-visuals-tools version 6.1.0, you can check your visual for unsafe calls to `fetch`, `XMLHttpRequest`, and `eval` using the following command: `pbviz package --certification-audit`. If any unsafe code is detected during the audit, you can automatically build a package with the necessary fixes by running: `pbviz package --certification-fix`. This flag removes all forbidden calls. You need to thoroughly test your visual to ensure it works as expected. Also, don't forget to update `npm run package` script in `package.json` to avoid hash mismatch during certification review.

## Source code requirements

Make sure you follow the [Power BI visuals additional certification](#) policy list. If your submission doesn't follow these guidelines, you'll get a rejection email from Partner Center with the policy numbers listed in this link.

Follow the code requirements listed here 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. Therefore, [WebAccess privileges](#) should be empty, or omitted, in the capabilities settings.
- Using `XMLHttpRequest`, or `fetch`.
- 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.

## Submit a Power BI visual for certification

Now you're ready to submit a request to have your Power BI visual certified by the Power BI team.

### 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, make sure you provide the following details:

- A link to the repository
- Sign in 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 indicating that it's a certified Power BI visual.

### Certified Power BI visuals in AppSource

- When someone searches 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.



- When the Power BI visual card is clicked 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 a Power BI visual is imported 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

The process of deploying to AppSource can take 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 new Power BI visual, it will be available for download from the AppSource link within a few hours. However, it takes an extra 10–14 days to reach production and become available in Power BI Desktop/Service.
- If you submitted an update to an existing Power BI visual, the new version will also appear on AppSource but will take up to two weeks to be deployed to the production environment.

### 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 won't take effect. Your visual's new capabilities will affect the user's reports after about a two weeks.

## When will my Power BI visual display a certification badge?

The certification badge should be visible within three weeks after your submission is approved.

## Related content

- [Frequently asked questions about certified visuals.](#)
- [Guidelines for publishing Power BI visuals](#)

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

# Guidelines for publishing Power BI custom visuals

Article • 07/02/2024

Before you publish Power BI custom visuals to Microsoft commercial marketplace for others to discover and use, follow these guidelines to create a great experience for your users.

## Power BI visuals: Free or for purchase

Power BI visuals submitted to the [Commercial Marketplace](#) (Microsoft AppSource) can be made available for free, or you can add the tag *additional purchase may be required*. Visuals with the *additional purchase may be required* designation are similar to in-app purchase (IAP) add-ins.

Like the free Power BI visuals, 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](#).

## Power BI visuals with IAP features

An IAP Power BI visual is a *free* visual that offers *free features*. It also has some advanced features available for a fee. In the Power BI visual's description, developers must notify users about features that require additional purchases to operate them.

The transactability and license management of these visuals are the responsibility of the ISV. They can be managed using any platform, but we recommend Microsoft's [Licensing API](#). With the Licensing API, licensed visuals are purchased through [AppSource](#) and customers can manage licenses in the [M365 admin portal](#) ↗. For more information about license management, see [Frequently asked questions about Custom visual license management](#).

For more information, see [our Commercial Marketplace certification policy](#).

### Important

If you update your Power BI visual from *free* to *additional purchase may be required*, customers must receive the same level of free functionality as before the update.

You can add optional advanced paid features in addition to the existing free features.

## Watermarks

You can use watermarks to allow customers to continue using the IAP advanced features without paying.

Watermarks let you showcase the full functionality of the Power BI visual to customers before they make a purchase.

Only use watermarks on **paid** features used without a valid license. Watermarks aren't allowed on free features of IAP visuals.

## Pop-up window

Use a pop-up window to explain how to purchase a license when customers use an invalid (or expired) license with your Power BI IAP visual.

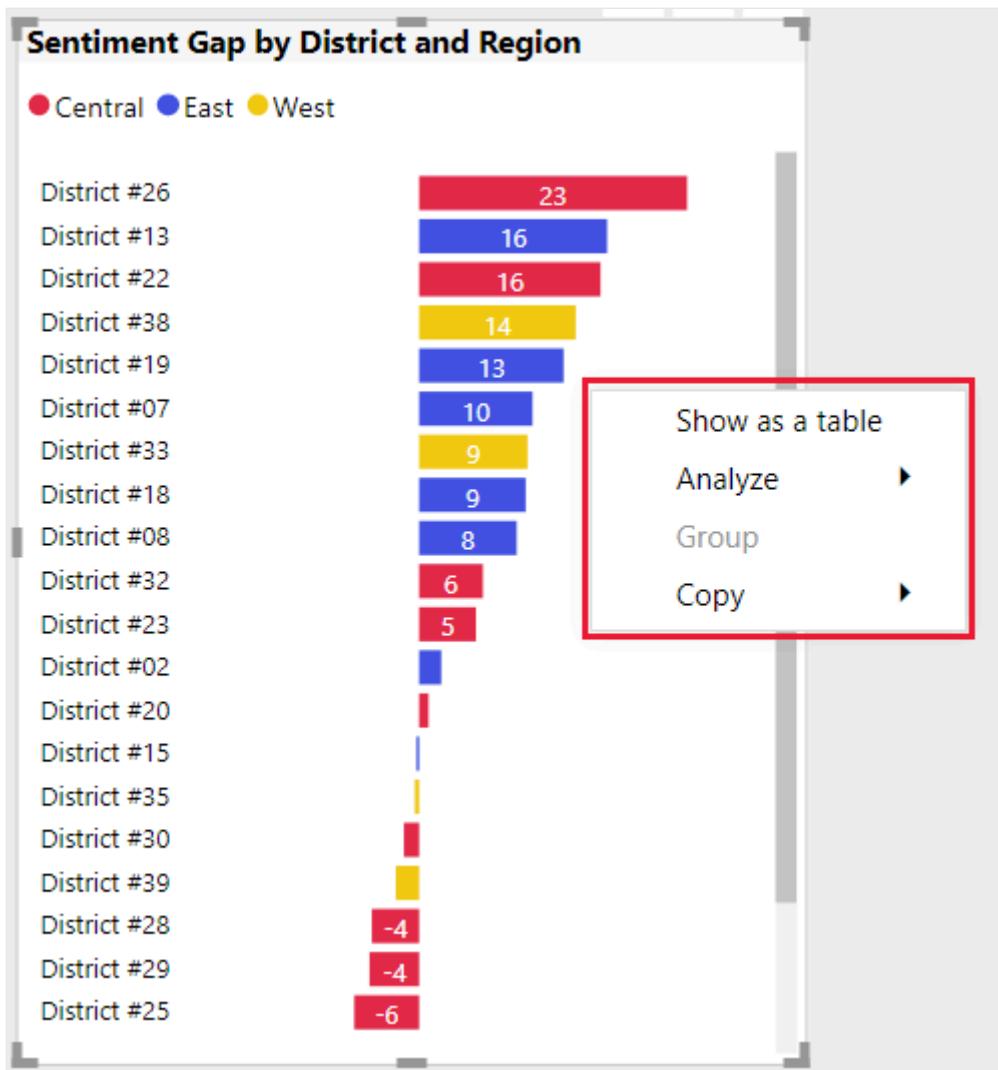
## Submission process

Follow the [submission process](#), then navigate to the **Offer setup** tab. Select 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 displays that *more purchase may be required* under the pricing options.

## Context menu

The context menu is the menu that displays when the user right-clicks inside a visual. All Power BI visuals should enable the context menu to give users a unified experience. For more information, see [Add a context menu to your Power BI visual](#).



## Commercial logo

This section describes the specifications for adding commercial logos in Power BI visuals. Commercial logos aren't required. If you add them, make sure they follow these guidelines.

### Note

- In this article, the term commercial logo refers to any commercial company icon. See the following pictures.
- 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:** Two elements locked together, an icon and a name.



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



- **Logotype:** A text logo without an icon.



## 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**



- **Logotype**



### 💡 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 uses two commercial logos—one for the offer details page and one for the search page. Provide the large logo in PNG format at 300 x 300 px. The Partner Center uses this logo to generate a smaller logo for the search page. You can optionally replace this logo with a different image later.

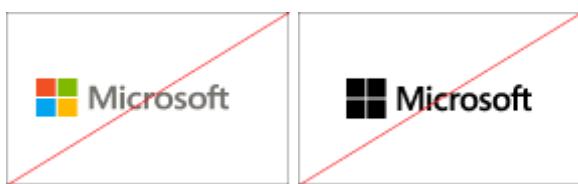
## Commercial logo behavior

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

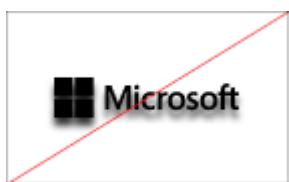
- Selecting the commercial logo redirects customers to your website.
- Selecting the commercial logo opens a pop-up window with additional information divided into two sections.
  - A marketing area for the commercial logo, a visual, and market ratings.
  - An information area for information and links.

## Limitations regarding logos

- Commercial logos can't be displayed in view mode.
- If your Power BI visual includes informative icons (i) in reading mode, they should comply with the color, size, and location of the commercial logo, as described earlier.
- Avoid a colorful or a black commercial logo. The commercial logo must be grey (hex color #C8C8C8).



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



## Best practices

When publishing a Power BI visual, consider the following recommendations to give customers the best possible experience.

- Create a [landing page](#) that provides information about your Power BI visual. Include details like how to use the visual and where to purchase the license. A meaningful landing page helps report creators use the visual correctly and easily.
- Don't include videos that are automatically triggered.
- Add only material that improves the customer's experience, such as information or links to license purchasing details and how to use IAP features.
- For the customer's convenience, add the license key or token related fields at the top of the format pane.
- Submit a short screen-recording video that shows how to use the visual.
- Submit a detailed description of the visual's functionality. Include information about supported features such as [high contrast](#), [report page tooltip](#), and [drill down](#).
- Check the quality of your code. Make sure it's up to standard, including unhandled exceptions.
- Update your visual using the [latest API ↗](#).

## FAQ

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

## Related content

Learn how you can publish your Power BI visual to for others to discover and use.

[Publish Power BI visuals](#)

[Get a Power BI visual certified](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

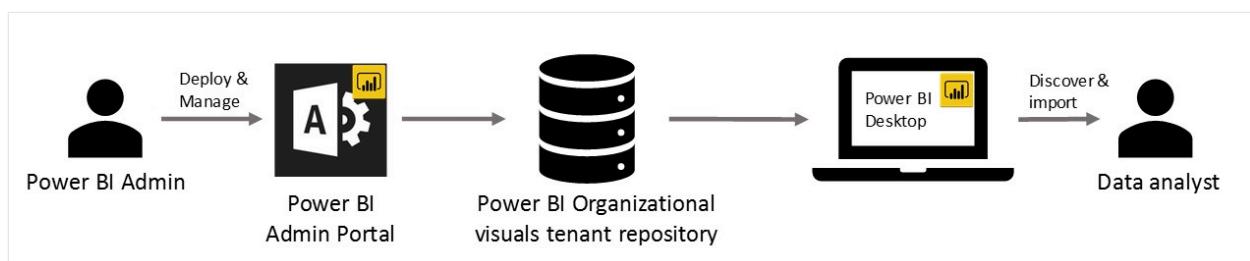
# Power BI organizational visuals

Article • 02/27/2024

You can develop your own custom Power BI visuals tailored to your own or your organization's specific needs. Usually, organizations develop their own custom visuals when none of the visuals included with Power BI meet their exact needs.

Some organizations might have unique requirements for their Power BI visuals. They might need visuals that can convey specific data or insights unique to their organization. They might have special data requirements, or they might highlight private business methods. These organizations can develop and maintain their own Power BI visuals that they can share throughout their organization.

The Power BI administrator uses the Admin portal to deploy and manage organizational visuals. After the visuals are deployed to the organizational repository, users in the organization can easily discover and import them into their reports directly from Power BI Desktop.



## Administer organizational Power BI visuals

To learn more about how to administer, deploy, and manage organizational Power BI visuals, see [manage Power BI visuals admin settings](#).

### ⚠ Warning

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

## Considerations and limitations

Be aware of the following [admin](#) and [user](#) considerations when using custom Power BI visuals in your organization.

## Admin considerations

- If a Power BI visual from AppSource or a file is deleted from the repository, any reports that use the deleted visual will stop rendering. Deleting from the repository isn't reversible. To temporarily disable a Power BI visual from AppSource or a file, use the **Disable** feature.
- If you remove a visual from AppSource, the visual will continue to render as it did before, as long as it's not removed from the repository.

## User considerations

- Organizational Power BI visuals 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 support these features.
- Organizational Power BI visuals aren't supported in Power BI report server.
- Certain visuals won't render if deployed through the organization's repository. Use the **Add from AppSource** option in the Admin portal to manage the following visuals:
  - Visio
  - Mapbox
  - Power Automate
  - Charticulator

## Related content

- [Get a Power BI visual certified](#)
- [Publish Power BI visuals to Partner Center](#)
- [Frequently asked questions about Power BI visuals](#)

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

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) ↗ | [Ask the community](#) ↗

# Power BI AppSource visual license models

Article • 06/05/2024

When you buy a custom visual from [AppSource](#), there are several business and licensing models (pricing, free trials etc.) available. Some visuals have free trial versions, while others have a basic version available for free with extra functionality available for purchase.

## ⓘ Note

Licensed visuals are supported from the July 2022 desktop.

## No license or partial license

If you try to render an unlicensed visual or a visual that you only have a partial license for, you might see one of the following icons:

- If you have an unlicensed visual in your report, the visual renders with an icon in the corner. Hover over the icon for a link to that visual on AppSource.com.

**Title of visual**

Subhead to visual



- If you have a free version of the visual, a banner appears with a link to upgrade your license. This banner will disappear after a while.

**Title of visual**

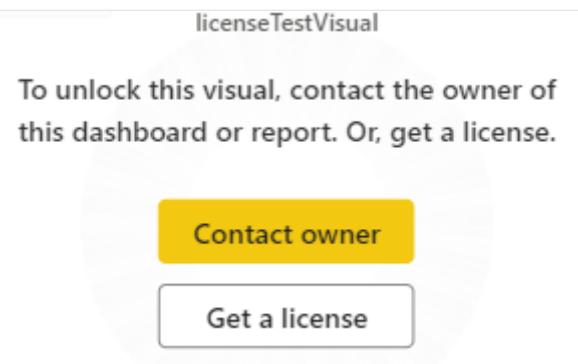
Subhead to visual

Upgrade visual

- If you try using a feature in the visual that you don't have a license to use, a banner appears with a link to upgrade your license.

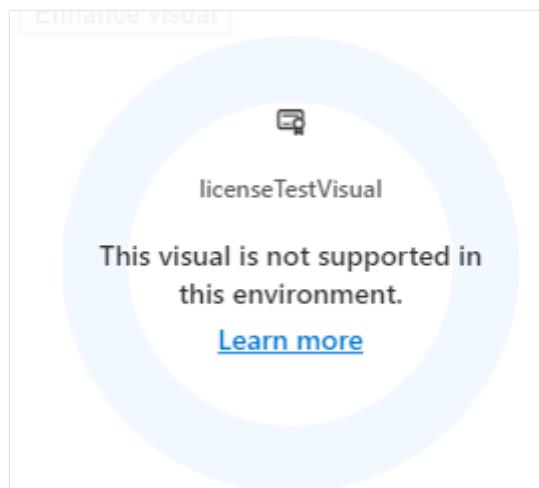
Upgrade to access this feature ⓘ Upgrade

- If you have an unlicensed visual in your report, the visual doesn't render, and a button appears enabling you to get a license or contact the report owner.



## Unsupported environment

If your report or dashboard contains a visual that isn't supported in your environment, the visual doesn't render and a notification appears.



For a list of unsupported environments see [limitations](#).

## Solution

Select **Upgrade visual** to go to the AppSource page where you can buy the license or upgrade to the paid version.

After you purchase and assign the license, it can take up to an hour for the license to be recognized. Wait one hour and then refresh your Power BI session (restart Desktop or refresh your web browser).

## Considerations and limitations

Currently, the following Power BI environments don't support license management or license enforcement:

- Embedded - Publish To Web, PaaS embed

- National/Regional clouds (Depends on general support for transactability in national/regional clouds)
- RS Server (No planned support)
- Exporting (PDF\PPT) using REST API

## Related content

[Import a Power BI visual](#)

---

## Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Licensing and transactability enforcement (Public preview)

Article • 11/04/2024

When you create Power BI visuals for download on AppSource, you can now manage and enforce their licenses using Microsoft systems. The end-user assigns and manages licenses using familiar tools like [Microsoft 365 admin center](#), and the [licensing API](#) lets you enforce these licenses and ensure that only licensed users can render the visuals.

## License enforcement process

The following table illustrates the steps involved in managing your visual licenses through Microsoft:

[\[+\] Expand table](#)

Step	Details
<a href="#">Create an offer in Partner Center</a>	Choose to transact through the Microsoft commerce system. Enable Microsoft to manage licenses. Set pricing and availability.
<a href="#">Add license enforcement to your Power BI visual package</a>	Create or reconfigure your package to use the Power BI runtime license, which enforces licensing according to each user's access.
<a href="#">Customers discover your offer in AppSource and purchase a subscription</a>	When customers purchase your offer in <a href="#">AppSource</a> , they also get licenses for the Power BI Visual.
<a href="#">Customers manage their subscriptions and assign/unassign user licenses</a>	Customers manage subscriptions and assign licenses for these Visuals and offers in the <a href="#">Microsoft 365 admin center</a> , just like they do for any of their other subscriptions like Office or Power BI.
<a href="#">Enforce runtime checks</a>	Give your customers a uniform experience by using our out-of-the-box APIs to enforce runtime license checks.
<a href="#">View reports to fuel growth</a>	Gain insight into revenue, payout information, and order and license details. View information about licenses and orders purchased, renewed, and canceled over time and by geography.

# Licensing API

The **Licensing API** allows Power BI visual developers to enforce Power BI visual licenses. The API supports retrieving the information on Power BI visual licenses that are assigned to the Power BI user. It also enables triggering the licensing related notifications that appear on the Power BI visual and inform the user that they need to purchase the missing licenses. The visual shouldn't display its own licensing UX, instead use one of Power BI supported predefined notifications as detailed in the following sections.

Learn more about Power BI licensing see [license enforcement](#).

## ⓘ Note

The **Licensing API** is available from version 4.7. To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.

## Retrieve visual's service plans that are assigned to the active user

To get service plans assigned, add a call to `getAvailableServicePlans` (available via `IVisualLicenseManager`). From performance perspective, attempt to fetch the licenses once, preferably in the `constructor` or the `init` calls, and save the result. Once licenses are retrieved, they're cached on Power BI host side during the Power BI session and any further calls to the same return the cached data.

### TypeScript

```
export interface IVisualLicenseManager {
    getAvailableServicePlans(): IPromise<powerbi.extensibility.visual.LicenseInfoResult>;
}
```

Retrieving the licenses might be a long operation, thus the `getAvailableServicePlans` call is an asynchronous call, and should be handled as such in your code. As a response to calling the method, `LicenseInfoResult` object is returned.

### TypeScript

```
export interface LicenseInfoResult {
    plans: ServicePlan[] | undefined;
    isLicenseUnsupportedEnv: boolean;
```

```
    isLicenseInfoAvailable: boolean;  
}
```

- `plans` - an array of Service Plans purchased by the active user for *this* visual.  
(Licenses purchased for any other visuals aren't included in the response.)  
A ServicePlan contains the service identifier (`splIdentifier`) and its state  
(`ServicePlanState`).
  - `splIdentifier`: the string value of the Service ID generated when you configure your offer's plans in Partner Center (see the following example)

**Power BI Visual Transact Test | Plan overview** + Create new plan

Name
Power BI Visual Contoso Basic1 Service ID: test_isvconnect159909224747.powerbivisualtransact.plan1

- `state` – enum (`ServicePlanState`) that represents the state of the plans assigned.  
Supported service plan states:

[+] Expand table

State	Description
Inactive	Indicates that the license isn't active and shouldn't be used for provisioning benefits.
Active	Indicates that the license is active and can be used for provisioning benefits.
Warning	Indicates that the license is in grace period likely due to payment violation.
Suspended	Indicates that the license is suspended likely due to payment violation.
Unknown	Microsoft Sentinel value.

Only the **active** and **warning** states represent a usable license. All other states should be treated as not resulting in a usable license.

- `isLicenseUnsupportedEnv` - indicates that the visual is being rendered in a Power BI environment that doesn't support licenses management or enforcement. Currently, the following Power BI environments don't support license management or license enforcement:
  - Embedded - Publish To Web, PaaS embed

- National/Regional clouds (Depends on general support for transactability in national/regional clouds)
- RS Server (No planned support)
- Exporting (PDF\PPT) using [REST API](#)
- `isLicenseInfoAvailable` - Indicates whether the licenses info could be retrieved. Failure in licenses retrieval can occur in case Power BI Desktop user isn't signed in or isn't connected to the internet (offline). For web, licenses retrieval can fail due to a temporary service outage.

Example of calling `getAvailableServicePlans` (using the service ID from the previous image):

TypeScript

```
private currentUserValidPlans: ServicePlan[] | undefined;
private hasServicePlans: boolean | undefined;
private isLicenseUnsupportedEnv: boolean | undefined;

this.licenseManager.getAvailableServicePlans()
.then(({ plans, isLicenseUnsupportedEnv, isLicenseInfoAvailable }: LicenseInfoResult) => {
  if (isLicenseInfoAvailable && !isLicenseUnsupportedEnv) {
    this.currentUserValidPlans = plans?.filter(({ spIdentifier, state }) =>
      (state === powerbi.ServicePlanState.Active || state ===
powerbi.ServicePlanState.Warning)
  );
  this.hasServicePlans = !!currentUserValidPlans?.length;
}
  this.isLicenseUnsupportedEnv = isLicenseUnsupportedEnv;
}).catch((err) => {
  this.currentUserValidPlans = undefined;
  this.hasServicePlans = undefined;
  console.log(err);
});
```

## Notify the user that the required licenses are missing

Power BI platform provides several out of the box experiences that can be used to notify:

- Licenses should be purchased in order to enjoy full visual's capabilities
- Particular visual's feature is blocked due to missing licenses
- Entire visual is blocked due to missing licenses
- Entire visual is blocked because the Power BI environment in use doesn't support license management\enforcement

TypeScript

```
export interface IVisualLicenseManager {
    notifyLicenseRequired(notificationType: LicenseNotificationType): IPromise<boolean>;
    notifyFeatureBlocked(tooltip: string): IPromise<boolean>;
    clearLicenseNotification(): IPromise<boolean>;
}
```

Example of calling `notifyLicenseRequired`:

TypeScript

```
private defaultNotificationType: powerbi.LicenseNotificationType =
powerbi.LicenseNotificationType.General;
private isNotificaitonDisplayed: boolean = false;

if (!this.isNotificaitonDisplayed) {
    const notificationType = this.isLicenseUnsupportedEnv ?
powerbi.LicenseNotificationType.UnsupportedEnv :
this.defaultNotificationType
    this.licenseManager.notifyLicenseRequired(this.getNotificationType())
        .then((value) => {
            this.isNotificaitonDisplayed = value;
        }).catch((err) => {
            console.log(err);
        });
}
```

## General icon indicating a required license is missing

Use `notifyLicenseRequired` call with `LicenseNotificationType.General` to display an icon as part of the visual's container.

Once triggered, the icon is preserved throughout the visual's lifetime until `clearLicenseNotification` or `notifyLicenseRequired` are called.

### ⓘ Note

The `LicenseNotificationType.General` notification is only enforced from an environment that supports licensing management and for Power BI Edit scenarios. Calling this in an unsupported environment or when the report is in Read mode or in dashboard doesn't apply the icon and returns `false` in the call's response.

Example of the visual display containing the "licenses are required" general icon:

## Title of visual

Subhead to visual



## Title of visual

Subhead to visual



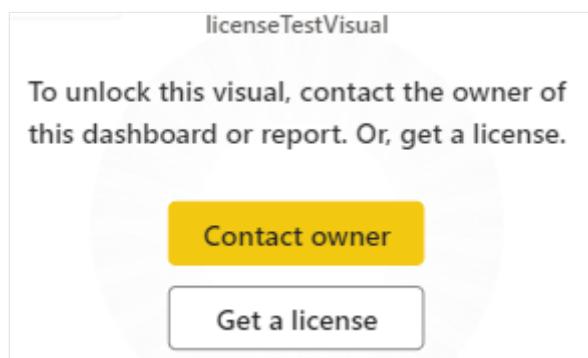
Upgrade visual

## Overlay the visual's display with a *missing license* notification

Use `notifyLicenseRequired` call with `LicenseNotificationType.VisualIsBlocked` to overlay the visual's display with a notification that visual is blocked since required licenses were found missing.

Once triggered, this notification is preserved throughout the visual's lifetime until `clearLicenseNotification` or `notifyLicenseRequired` are called.

Example of the visual display containing the *visual blocked* notification. Power BI Desktop only displays the *Get a license* option:



## Overlay the visual's display with an *unsupported environment* notification

Use `notifyLicenseRequired` call with `LicenseNotificationType.UnsupportedEnv` to overlay the visual's display with a notification that visual is blocked since the Power BI in use doesn't support licenses management\enforcement.

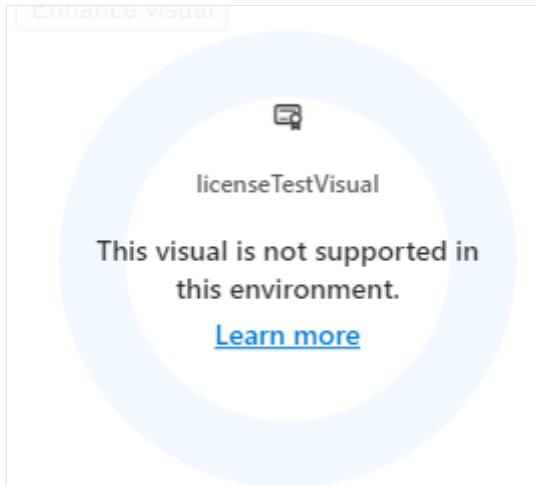
Once triggered, the icon is preserved throughout the visual's lifetime until `clearLicenseNotification` or `notifyLicenseRequired` are called.

### ⓘ Note

The `LicenseNotificationType.UnsupportedEnv` notification is only enforced when called in context of unsupported for licensing environment. Calling this in any other

environment doesn't apply the notification and returns `false` in the call's response.

Example of the visual display containing the "Unsupported Environment" notification:



## Display a banner notifying that a specific visual's functionality couldn't be applied

When applying a specific visual's functionality requires licenses that were found missing, you can use the `notifyFeatureBlocked` call that displays a pop-up banner as part of the visual's container. The banner also supports a custom tooltip that you can set and use to provide additional information on the feature that triggered the notification.

### ⓘ Note

The *feature is blocked* notification is only enforced when both the following conditions apply:

- It's called from a supported licensing environment
- Blocking overlays aren't applied (`LicenseNotificationType.UnsupportedEnv`, `LicenseNotificationType.VisualIsBlocked`).

Calling this notification in an unsupported environment doesn't apply the notification and returns `false` in the call's response.

### ⓘ Note

To support localized Power BI environment, we recommend maintaining localized versions of the tooltips in use. Please use [Localization API](#) to retrieve the Power BI locale language.

Once triggered, the banner is displayed for 10 seconds, or until other "feature blocked" banner is triggered, or until `clearLicenseNotification` is called (whatever comes first).

Example of the visual display containing the "feature blocked" banner notification:



## Test a licensed visual

To test a licensed visual end to end before making it publicly available:

- If you're creating a brand new offer, add the visual as a private plan for a test customer account. The offer is only visible to this test account for purchasing. Use this account to validate the offer before making it public.
- If your visual is already available in AppSource and you want to upgrade it to a licensed visual, you *can't make it a private plan* because that hides the visual from AppSource, and your existing users won't have access to it. There's currently no way to test a published visual end to end. Test it the same way you tested the original visual to AppSource, by mocking the licensing API value to check the different possibilities.

## Considerations and limitations

- Tooltip for feature banner is limited by 500 chars.
- Tooltip for feature banner requires localization.
- License bundling (that is, one license that covers multiple offers from the same publisher) isn't yet supported.

## Related content

[Publish a Power BI custom visual](#)

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

---

## Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Frequently asked questions about Custom visual license management and transactability

This article answers some of the questions users often have about how to set up, pay for, and manage licenses for custom Power BI visuals that they purchase from [AppSource](#).

## ! Note

If you just purchased a licensed visual, you are assigned a license automatically. [Just refresh your report.](#)

To assign licenses to other users in your organization, see [How do we assign licenses](#).

## Purchasing

### What are licensed visuals?

Licensed visuals are Power BI visuals available from the [Microsoft commercial marketplace](#).

All licensed visuals have a limited basic version available for free, and a version with more functionality available for purchase. You can try out a free version of the visual before you need to purchase a license. You can also assign licenses to other users in your organization.

### Do I have to pay when I install a visual?

You can install the visual for free from [AppSource](#) by selecting **Install free** and following the instructions there, or by [embedding it directly into your report](#). This allows you to use the free features provided. For the full experience, however, you need to purchase the visual from [AppSource](#). The pricing for each licensed visual is described in the *Plans + Pricing* tab.

**Gantt Chart by MAQ Software**

by MAQ LLC

Power BI visuals

PBI Certified

★ 4.0 (91 ratings)

Install free Buy now Download Sample Instructions

Overview	Plans + Pricing	Ratings + reviews	Details + support
Plan	Description	Monthly Price	Annual Price
Gantt Chart by MAQ Software (1 - 99 users)	This plan unlock below additional features <ul style="list-style-type: none"><li>Progress Bar: Show progress bar on tasks based on a measure value</li><li>Task Status: Show custom colored task status flags based on a categorical field</li><li>Task Milestones: Plot date fields as milestones on task bar with custom colored shapes</li><li>Row Configuration: Customize the row height and task bar radius</li></ul>	First month free, then \$XX.XX/user/month	First month free, then \$XX.XX/user/year
Gantt Chart by MAQ Software (100 - 499 users)	This plan unlocks below additional features <ul style="list-style-type: none"><li>Progress Bar: Show progress bar on tasks based on a measure value</li><li>Task Status: Show custom colored task status flags based on a categorical field</li><li>Task Milestones: Plot date fields as milestones on task bar with custom colored shapes</li><li>Row Configuration: Customize the row height and task bar radius</li></ul>	First month free, then \$XX.XX/user/month	First month free, then \$XX.XX/user/year

## Who can purchase a visual license?

Anyone can purchase a license, and assign the license to themselves or others.

## How can I purchase and pay for the visuals?

To purchase the visual, select **Buy now** and complete the checkout path in [AppSource](#).

**Gantt Chart by MAQ Software**

by MAQ LLC

Power BI visuals

PBI Certified

★ 4.0 (91 ratings)

Install free Buy now Download Sample Instructions

You can pay for the licenses with a credit card.

Prepaid cards are not supported.

Billing admins can also pay by [invoice](#) if that's set up for your company.

Read more about [setting payment methods](#). Follow [these steps](#) to see your billing profile.

**Bill to**

Select the billing profile you want to use for this purchase. You can also edit an existing profile. Learn more about billing profiles

**Billing profile**  
FL test account - Invoice

[Edit](#)    [Add new](#)

For troubleshooting payment and billing issues, see [Troubleshoot Azure payment issues](#) and [Troubleshoot a declined card](#).

## Do we need to have Power BI Premium to purchase Licenses?

No. any user can purchase visual licenses in AppSource.

## We have a Power BI Enterprise agreement with Microsoft. Is this included under that agreement?

Not yet.

## How do we negotiate site licensing or private custom offers?

Private offers and site licensing aren't supported yet. You can, however create Private Plans. [Learn more about private plans](#).

## Can we buy in local currency?

Yes.

## Can we do multi-year license purchase instead of one year at a time?

Not yet.

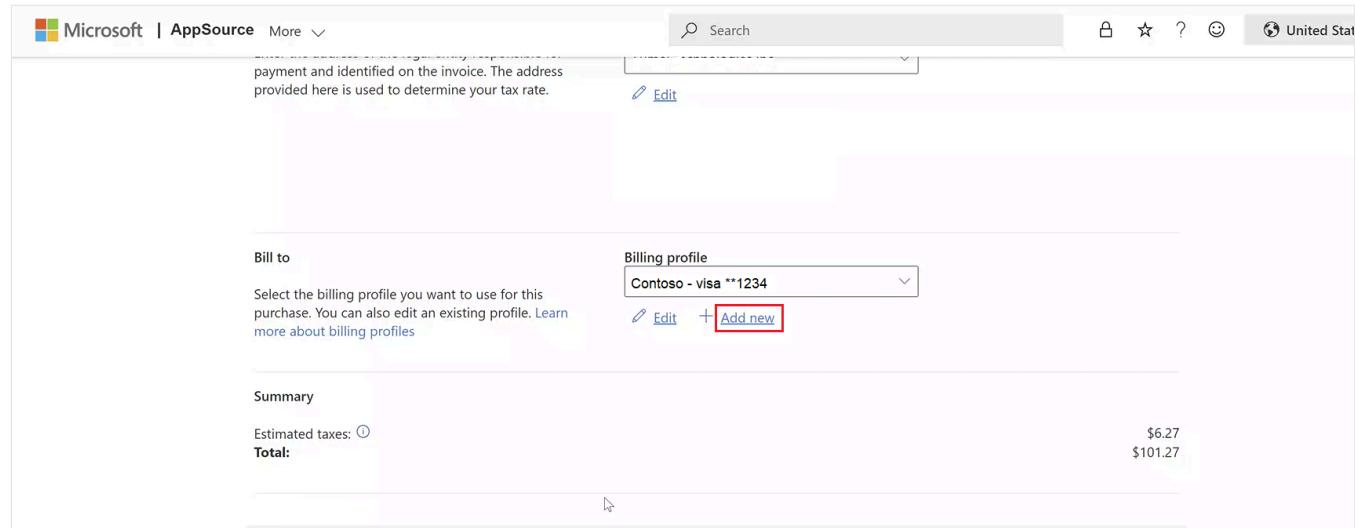
## How do I know if someone in my company has already purchased this same visual?

If you're a company administrator, sign in to the [Microsoft 365 admin center](#) to see all subscriptions and to whom they're assigned.

If you aren't an administrator, speak to the administrator in your company.

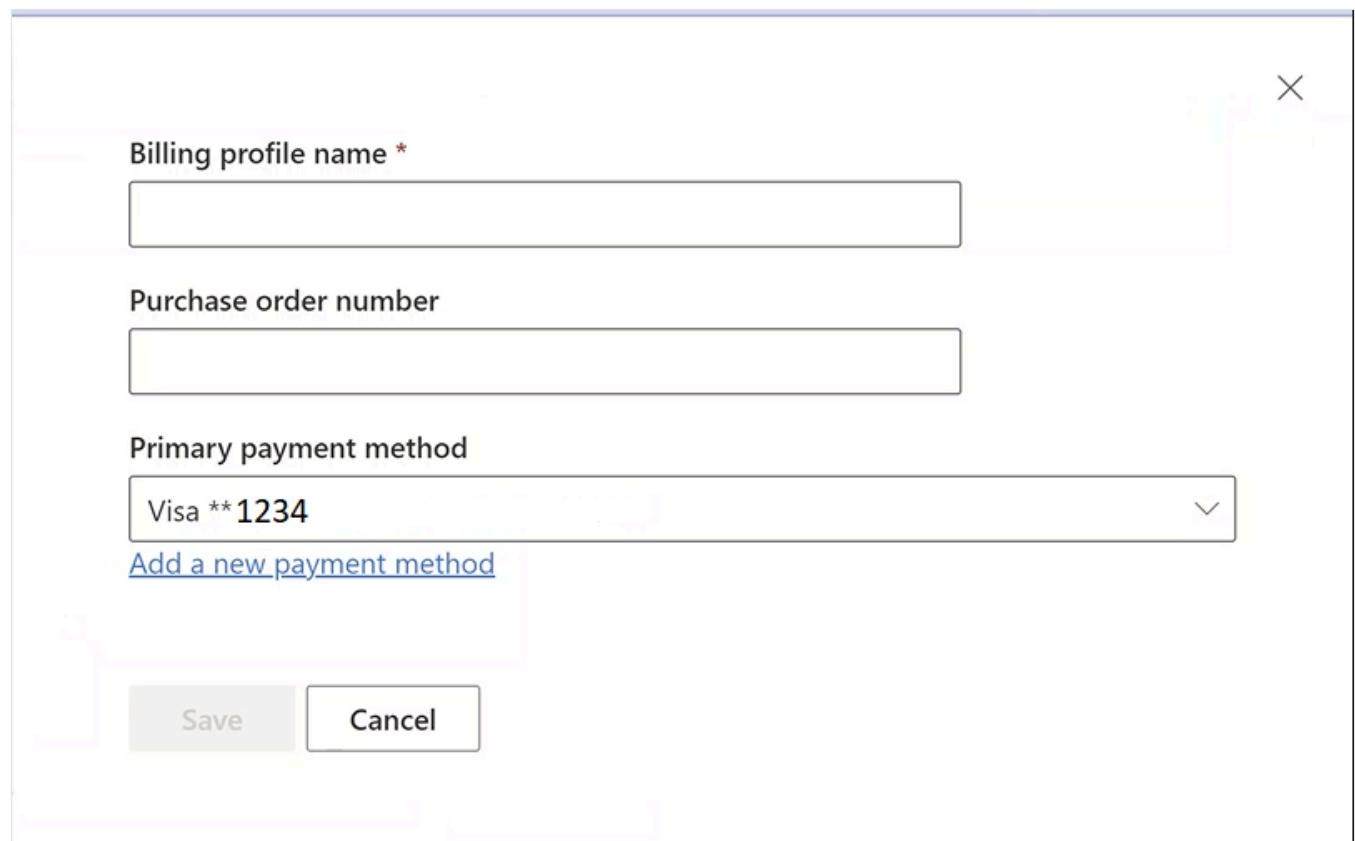
## How can I add a purchase order number to the transaction?

You can add a purchase number to the billing profile of your order. At the final stage of your checkout, select **Add new** under the billing profile.



The screenshot shows a Microsoft AppSource checkout interface. At the top, there's a navigation bar with the Microsoft logo, 'AppSource', and a search bar. Below the navigation, there's a note about identifying payment and address information. On the right side of the main area, there's a 'Billing profile' dropdown set to 'Contoso - visa \*\*\*1234', with an 'Edit' link next to it. A red box highlights the '+ Add new' button. Below this, there's a 'Summary' section showing 'Estimated taxes: ⓘ' and 'Total: \$101.27'. The total amount is also shown as '\$6.27'.

Add the purchase order number in the appropriate space.



The screenshot shows a modal dialog for adding a new billing profile. It has fields for 'Billing profile name \*' (empty), 'Purchase order number' (empty), and 'Primary payment method' (set to 'Visa \*\*1234'). Below the payment method is a link 'Add a new payment method'. At the bottom are 'Save' and 'Cancel' buttons.

# How can I download my receipt after purchase?

After you complete your purchase, you can find your receipt and download it as PDF from the Microsoft 365 admin center under Bills & payments - Invoices.

The screenshot shows the Microsoft 365 Admin Center interface. The left sidebar is collapsed. The main content area displays the 'Invoice summary' for invoice T0013503. At the top, there's a breadcrumb navigation: Home > Bills & payments - Invoices > T0013503. Below the breadcrumb, the 'Billing period' is listed as February 1, 2023–February 28, 2023. The 'Invoice date' is March 5, 2023, and the 'Billing profile' is Customer. To the right, it shows 'Amount due: \$0.00' and 'Last payment received on' (which is not visible). A red box highlights the 'Download PDF' button. The main body shows a breakdown of the invoice by service: FirstResource (\$0.00) and SaaS, which further breaks down into Contoso SaaS Offer-site. The bottom right corner of the screen has a small toolbar with a magnifying glass icon and a message icon.

## Can we use a visual without having to upload it to organization visual?

Yes. You can download a visual [directly to your reports](#).

## How do we get upgrades of the visuals?

The AppSource visuals are updated automatically when a new version is available.

## What is the refund policy?

You're eligible for a full refund if you cancel your subscription within seven days of purchase. Refunds aren't available for subscriptions canceled after that time period.

## How can I get more help buying a visual?

For help with buying a visual, [contact support](#).

# License assignment

## Who can assign licenses after the purchase is complete?

The person who buys the visual is the owner, and only the owner can assign licenses.

### Note

After buying the visual, the buyer (owner) is automatically assigned a license. Their license is available after [refreshing the report](#). It might take a few minutes.

If the tenant admin owns the licenses (organizational licenses), then any of the license admins (License Admin or User Admin) can manage the licenses.

However, if the license owner isn't an admin, then only the purchaser/owner can manage the subscription and licenses.

## How do we assign the licenses?

Only the buyer (owner) can assign a license. You receive a license automatically within five minutes of purchasing the visual. If you want to assign a license to other users or update your subscription, go to [admin.microsoft.com](#) and select the **License** link under the **Billing** node from the menu. A list of visuals appears.

The sidebar menu includes:

- Home
- Users
- Teams & groups
- Billing**
- Purchase services
- Your products
- Licenses**
- Bills & payments
- Billing accounts
- Payment methods
- Billing notifications
- Setup

... Show all

Home &gt; Licenses

## Licenses

[Subscriptions](#)   [Requests](#)   [Auto-claim policy](#)

Select a product to view and assign licenses.

[Go to Your products](#) to manage billing or buy more licenses.[Export](#)   [Refresh](#)

Name ↑



Select the Visual you want to assign licenses for. In the next page, select the user(s) you want to assign the licenses to.

## Once I assign a license, how long does it take until I can use it?

### ! Note

After you assign a license in the Microsoft 365 Admin Center, the license should be available within a few minutes.

If you were just assigned a license, refresh the report to activate it by doing one of the following:

- If you're using the licensed visual in the Power BI Service, refresh the report by hitting **F5**.
- If you're using the licensed visual in the Power BI Desktop, close and reopen it.

If, after you refresh, the license is still not available, wait a while and try again. It could take up to five minutes.

## **Can I sign in to Microsoft 365 admin center even if I'm not an admin?**

Yes. When you purchase a Power BI Visual subscription, you also get access to the Microsoft 365 admin center. You can manage your own license subscriptions from there, but can't see or access any other subscriptions unless you're an admin.

## **Do I have to assign a license to myself even if I'm the owner?**

No. Buyers get a license assigned to them automatically. It can take up to five minutes from the time of purchase [for the license to take effect](#). If you want to add more users or update your subscription, go to the [admin center](#).

## **As an admin can I see the subscriptions that were purchased in my organization?**

A tenant admin can see all subscriptions purchased under the tenant account, including the subscriptions purchased by non-admins.

## **What happens if the subscription owner leaves the company?**

Even if the owner of the subscription leaves the company, tenant admin purchases (typically known as organizational purchases) continue to work as-is. If the owner was an admin, any license admin (License Admin, User Admin) can continue to manage the licenses. The subscriptions can be managed by any Billing Admin with access to the billing account used for purchase.

If the owner wasn't an admin, the license will continue to work as long as the subscription is active. However, the subscription can't be managed by anyone else.

## **We're a global team. Can we assign licenses to someone working in another country/region?**

You can assign licenses to users in the same tenant that you purchased subscription under regardless of the location.

# As an admin, can I cancel offer purchase from AppSource?

As a billing admin, you can see all subscriptions in the Microsoft 365 admin center, and you can cancel them.

## How can I get more help or support for managing licenses?

For help managing licenses, [contact support](#)

## Related content

[Licensing models](#)

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

# Power BI visuals API changelog

Article • 07/22/2024

This page contains a short summary of the existing API versions and what to expect in the upcoming version. Versions listed here are considered stable and don't change.

## API v5.10.0

- **DataViewMetadataColumn** has a new property called `sourceFieldParameters`. This property indicates if the current field is the result of a field parameter. If a single field can originate from multiple field parameters, this property lists all the related field parameters.
- Supports Desktop June 2024

## API v5.9.1

- [acquireAADTokenService](#): Enhanced to support the following clouds.
  - Commercial Cloud
  - China Cloud
  - US Government Community Cloud
  - US Government Community Cloud High
  - US Department of Defense Cloud

## API v5.9.0

- [Hierachial identity filter API](#): Allows you to create a visual that uses Matrix DataView Mapping to filter data based on data points that use a hierachal structure. This is useful for custom visuals that leverage group-on keys semantic models and want to filter hierarchies based on data points.
- [acquireAADTokenService](#): Extended with additional properties
- Supports Desktop March 2024

## API v5.8.0

- [Local storage API](#): A new version of local storage API available for all custom visuals and controlled by a global admin setting that is *on* by default. The admin can Turn off the global setting to disable both the legacy API and the new version of the API.

- **On-object support for custom visuals:** On object support for custom visuals to optimize the user experience and provide a unified authoring experience on par with out of the box visuals.
- Supports Desktop February 2024

## API v5.7.0

- **Power BI Custom Visuals Authentication API:** Allows Custom Visuals to obtain Microsoft Entra access tokens through single sign-on (SSO), facilitating secure and efficient user-contextual operations.
- **Dynamic drill control:** Allows the visual to enable or disable the drill feature dynamically using an API call.
  - When the drill feature is enabled, all the functionalities of drilldown and expand/collapse features are available. These functionalities include API calls, context menu commands, header drill buttons, and support for hierarchy data.
  - When the drill feature is disabled, these functionalities aren't available.
- Supports Desktop December 2023

## API v5.4.0

- **Improved keyboard navigation:** Improves accessibility and usability of your visuals by providing more options for interacting with visual using the keyboard.
- **Detect filter use in reports:** Detect if there are any filters applied to a report.
- Supports Desktop May 2023

## API v5.3.0

- SelectionId's update-fix for `matrix` `dataView`.

 **Note**

The selectionId's core data might change. Therefore, a persisted selectionId/identityIndex using an older API version might not be relevant in matrix visuals.

- **downloadService:** Adds a new method `exportVisualsContentExtended` that returns expanded result information of the download.
- Supports Desktop March 2023

## API v5.2.0

- **Customized data reduction** - This feature added to *capabilities.json* schema allows the [data fetch window](#) to be modified dynamically by the custom visual code the report author.
- Supports Desktop December 2022

## API v5.1.0

- **Custom sorting** - improved custom sorting for tables
- **Subtotals** - new *Subtotals Type* indicates if totals should be retrieved before or after the rest of the data
- **Identity filter** - filter categorical data
- **New format pane** - design a custom visual that supports the new format pane design
- Supports Desktop October 2022

## API v4.7.0

- **Licensing API** - Sell, manage, and enforce licenses directly through the commercial marketplace.
- **Drilldown API** - Create a visual that can trigger a drilldown operation on its own, without user interaction.
- Supports Desktop July 2022

## API v4.6.0

- New capabilities property: [privileges](#) and two privileges:
  - web access
  - download file from custom visual
- Added two corresponding [tenant admin switches](#) ↗
- **Download API** to allow downloading visual to file
- Supports Desktop June 2022

## API v4.2.0

- New flags to [expand and collapse row headers](#)
- Supports Desktop February 2022

## API v3.8.0

- Supports Desktop May 2021 and later.

## API v3.7.0

- Supports Desktop April 2021 and later.

## API v3.6.0

- Visual can receive updates from Power BI [without the need to bind any data](#).
- Supports Desktop 2021 February and later.
- Supports Desktop RS May 2021 and later.

## API v3.4.0

- `fetchMoreData` : new `aggregateSegments` parameter (default true), for supporting no-aggregation fetchMoreData
- Supports Desktop 2020 November and later.
- Supports Desktop RS January 2021 and later.

## API v3.2.0

- Supports [supportsMultiVisualSelection](#)
- Supports Desktop 2019 September and later.
- Supports Desktop RS January 2020 and later.

## API v2.6.0

- Adds `isInFocus` to update option and `switchFocusModeState` method to visual host
- Supports `subtotals` customization
- Supports Desktop 2019 June and later.
- Supports Desktop RS May 2019 and later.

## API v2.5.0

- Supports [Analytics Pane](#)

- Supports `SelectionIdBuilder` `withMatrixNode` and `withTable` methods
- No longer supports `DataRepetitionSelector` interface, replaced with `data.CustomVisualOpaqueIdentity` interface

## API v2.3.0

- [Landing Page API](#)
- [LocalStorage API](#)
- [Tuple filter API \(multi-column filter\)](#)
- [Rendering Events API](#)

## API v2.2.0

- Supports [restoring JSON Filter from DataView](#)
- [ContextMenu API](#)
- Supports [Drillthrough](#) feature

## API v2.1.0

- 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:

TypeScript

```
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 v1.13.0

- 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 v1.12.0

- Supports Themes
- Supports [fetchMoreData](#), note the [Fetch More Data API](#) overcomes the hard limit of 30K data points
- [Canvas Tooltips API](#)

## API v1.11.0

- [FilterManager API](#)
- Supports [Bookmarks](#)

## API v1.10.0

- Adds `ILocalizationManager`
- [Authentication API](#)

## API v1.9.0

- [launchUrl API](#)

## API v1.8.0

- Supports new type **fillRule** (gradient) in capabilities schema
- Supports **rule** property in capabilities schema for object properties

## API v1.7.0

- 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 [Toolips](#)

## 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 `pbiviz update` to update visual API type definitions and schema
- Supports `--api-version` flag in `pbiviz 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
- 

## Feedback

Was this page helpful?

 Yes

 No

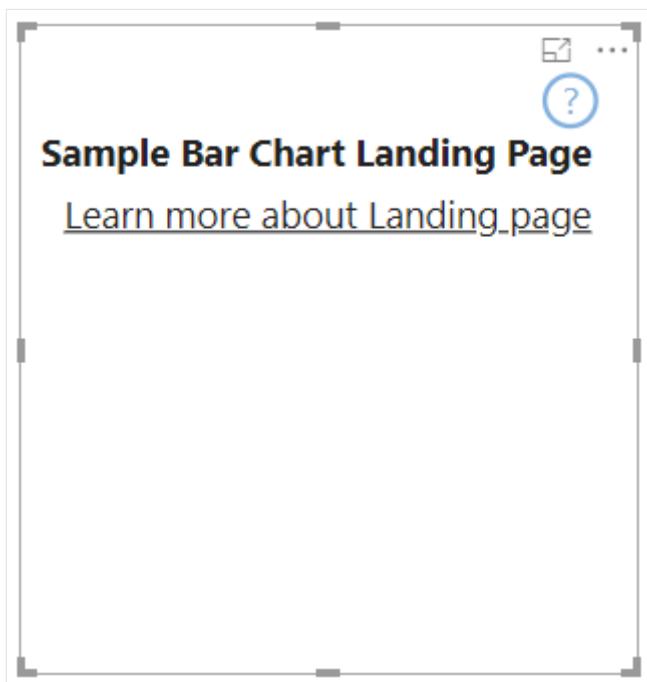
[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Add a landing page to your Power BI visual

Article • 06/17/2024

A Power BI visual's landing page can display information in your Power BI visual card before the card gets data. A visual's landing page can display:

- Text that explains how to use the visual.
- A link to your website.
- A link to a video.



This article explains how to design a landing page for your visual. The landing page appears whenever the visual has no data in it.

## ⓘ Note

Designing a Power BI visual landing page is supported in API version 2.3.0 and above. To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.

## Create a landing page

To create a landing page, two capabilities must be set in the `capabilities.json` file.

- For the landing page to work, enable `supportsLandingPage`.

- For the landing page to be displayed in view mode, or for the visual to be interactive even when in **no data-roles mode**, enable `supportsEmptyDataView`.

JSON

```
{
  "supportsLandingPage": true,
  "supportsEmptyDataView": true,
}
```

## Example of a visual with a landing page

The following code shows how a landing page can be added to a bar chart visual.

TypeScript

```
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[0]?metadata?.columns?.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();
      }
    }
  }
}
```

```
    }  
}
```

## Related content

Formatting utils

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Create a launch URL

Article • 06/17/2024

A launch URL lets you open a new browser tab or window by adding the `host.launchUrl()` API call to the code of a Power BI visual.

## ⓘ Note

The `host.launchUrl()` method was introduced in Visuals API 1.9.0.

## Sample

Import the `IVisualHost` interface and save the link to the `host` object in the constructor of the visual.

TypeScript

```
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 and pass your destination URL as a string argument:

TypeScript

```
this.host.launchUrl('https://some.link.net');
```

## 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 the button results in opening a new tab. It can be confusing or frustrating for the user if a `launchUrl()` call triggers without a user's action or as a side effect of a different action.
- If the link isn't essential for the visual to function properly, we recommend that you give the report's author a way to disable and hide the link. Special Power BI use cases, such as embedding a report in a third-party application or publishing it to the web, might require disabling and hiding the link.
- 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

Add the following lines to the visual's `constructor` function:

TypeScript

```
this.helpLinkElement = this.createHelpLinkElement();
options.element.appendChild(this.helpLinkElement);
```

Add a private function that creates and attaches the anchor element:

TypeScript

```
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://learn.microsoft.com/power-
bi/developer/visuals/custom-visual-develop-tutorial");
    });
    return linkElement;
};
```

Define the style for the link element with an entry in the `visual.less` file:

less

```
.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

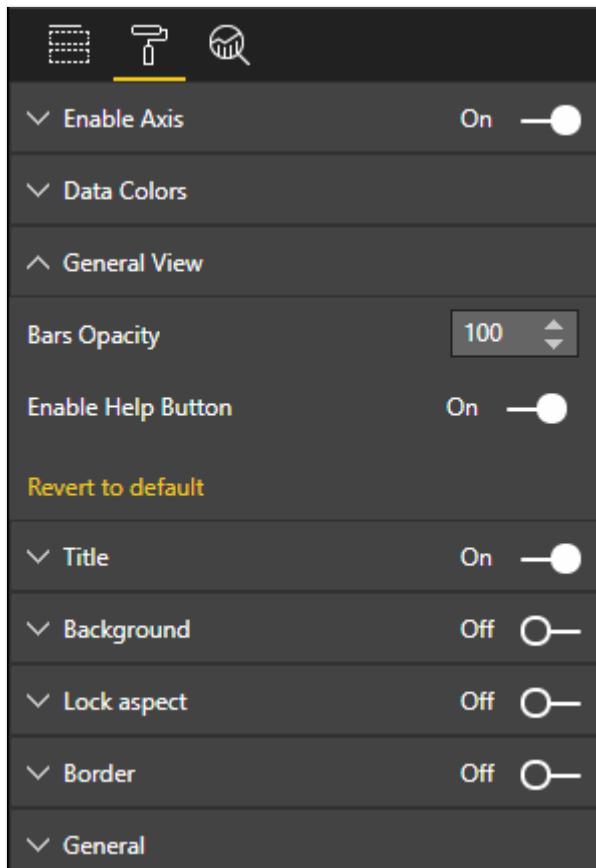
To add a toggling mechanism, you need to add a static object so that the report's author can toggle the visibility of the link element. (The default is set to `hidden`.) For more information, see the [static object tutorial](#).

Add the `showHelpLink` Boolean static object to the `capabilities.json` file's objects entry:

TypeScript

```
"objects": {
  "generalView": {
    "displayName": "General View",
    "properties": {
      "showHelpLink": {
        "displayName": "Show Help Button",
        "type": {
          "bool": true
        }
      }
    }
}
```

```
        }  
    }
```



Add the following lines in the visual's `update` function:

TypeScript

```
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.

## Considerations and limitations

- 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 other protocols.

# Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Visual API

Article • 03/11/2025

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 be the same as the `visualClassName` in the `pbviz.json` file.

The visual class should implement the following methods as shown in the following sample:

- `constructor` - a standard constructor that initializes the visual's state
- `update` - updates the visual's data
- `getFormattingModel`, returns a formatting model that populates the property pane (formatting options) where you can modify properties as needed
- `destroy` - a standard destructor for cleanup

TypeScript

```
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 getFormattingModel(): FormattingModel {
        // returns modern format pane formatting model that contain all
        format pane components and properties (called on opening format and
        analytics pane or on editing format properties)
    }

    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 the visual needs.

TypeScript

```
constructor(options: VisualConstructorOptions)
```

## VisualConstructorOptions

These interfaces get updated with each new API version. For the most updated interface format, go to our [GitHub repo ↗](#).

The following list describes some of the properties of the `VisualConstructorOptions` interface:

- `element: HTMLElement` - a reference to the DOM element that contains your visual
- `host: IVisualHost` - a collection of properties and services that can be used to interact with the visual host (Power BI)

`IVisualHost` contains the following services:

- `createSelectionIdBuilder` - generates and stores metadata for selectable items in your visual
- `createSelectionManager` - creates the communication bridge used to notify the visual's host about changes in the selection state, see [Selection API](#).
- `hostCapabilities`
- `refreshHostData`
- `downloadService` - returns expanded result information of the [download](#).
- `eventService` - returns information about [rendering events](#).
- `hostEnv`
- `displayWarningIcon` - returns [error or warning message](#).
- `licenseManager` - returns [license information](#).
- `createLocalizationManager` - generates a manager to help with [localization](#)
- `applyJsonFilter` - applies specific filter types. See [Filter API](#)
- `applyCustomSort` - allows [custom sorting options](#).
- `acquireAADTokenService` - returns Microsoft Entra ID [authentication information](#).
- `webAccessService` - returns permission status for [accessing remote resources](#).
- `openModalDialog` - returns a [dialog box](#).

- `persistProperties` - allows users to create persistent settings 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
- `storageV2Service` - returns a service to help use [local storage](#) version 2 in the visual
- `tooltipService` - returns a [tooltip service](#) to help use tooltips in the visual
- `telemetry`
- `drill`
- `launchUrl` - helps to [launch URL](#) in next tab
- `authenticationService` - returns a Microsoft Entra ID token.
- `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 (1,000 rows).  
See [Fetch more data](#)
- `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.

TypeScript

```
public update(options: VisualUpdateOptions): void
```

## VisualUpdateOptions

- `viewport: IViewport` - dimensions of the viewport that the visual should be rendered within
- `dataViews: DataView[]` - the data view object that contains all data needed to render your visual (a visual generally uses the categorical property under `DataView`)
- `type: VisualUpdateType` - flags indicating the type of data being updated (`Data` | `Resize` | `ViewMode` | `Style` | `ResizeEnd`)
- `viewMode: ViewMode` - flags indicating the view mode of the visual (`View` | `Edit` | `InFocusEdit`)

- `editMode:EditMode` - flag indicating 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 indicating 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
- `pendingChanges?: PendingChanges` - flag to indicate that local filter changes are made but yet applied to the report, usually triggered when **apply all slicers** button exists | `pendingChanges[Filter]: boolean`

## getFormattingModel (*optional*)

This method is called once each time we open the properties pane or the user edits any of the properties in the pane. It returns [FormattingModel](#) with all information on the properties pane design, hierarchy, properties, and latest formatting values.

TypeScript

```
getFormattingModel(): visuals.FormattingModel;
```

## 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.

TypeScript

```
public destroy(): void
```

### 💡 Tip

Power BI generally doesn't call `destroy` since it's faster to remove the entire IFrame that contains the visual.

## Related content

- Visual project structure
  - Local storage API
- 

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Local Storage API

Article • 04/30/2025

With the local storage API, you can store data in the browser's local storage. To use the local storage API, the customer's [local storage admin switch](#) has to be enabled.

Local storage is isolated so that each type of visual has its own separate storage access.

## ! Note

It's developer's responsibility to ensure that the stored data conforms to the consumer's organizational policies, and to inform users about what information is stored, if the sensitivity of the data requires it. In particular, custom visual developers should encrypt the data if business goals or scenarios expect it.

## How to use local storage

Version 1

This version of the *local storage API* is scheduled for deprecation. We're not accepting any more requests. When possible, use Version 2.

In the following example, a counter is increased whenever the *update* method is called. The counter value is saved locally and called each time the visual starts. This way, the counter continues counting from where it left off instead of starting over each time the visual is started:

TypeScript

```
export class Visual implements IVisual {
    // ...
    private updateCountName: string = 'updateCount';
    private updateCount: number;
    private storage: ILocalStorageService;
    // ...

    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());
// ...
}
}
```

## Considerations and limitations

Considerations and limitations Version 1

- The local storage limit is 1 mb per GUID.
- Data can be shared between visuals with the same GUID only.
- Data can't be shared with another instance of Power BI Desktop.
- The local storage API isn't activated by default. To activate it for your Power BI visual, send a request to Power BI visuals support, [pbicvssupport@microsoft.com](mailto:pbicvssupport@microsoft.com).
- The local storage API doesn't support `await` constructions. Only `then` and `catch` methods are allowed.

Your visual should be available in [AppSource](#) and be [certified](#).

## Related content

- [Power BI custom visual API](#)

# Authentication API

Article • 06/26/2024

The Authentication API enables visuals to obtain Microsoft Entra ID (formerly known as Azure AD) access tokens for signed-in users, facilitating single sign-on authentication.

Power BI administrators can enable or disable the API through a [global switch](#). The default setting blocks (disables) the API.

The API is applicable only for AppSource visuals, and not for private visuals. Visuals that are under development can be tested in debug mode before they're published.

## Supported environments

The following environments are supported:

- Web
- Desktop
- RS Desktop
- Mobile

## Unsupported environments

The following environments aren't yet supported:

- RS Service
- Embedded analytics
- Teams

## How to use the Authentication API

In the *capabilities.json* file, add the "AADAuthentication" privilege with a Microsoft Entra ID registered application URI for each supported cloud. Fabric generates a token according to the audience configured for the current cloud, and delivers it to the visual. The visual can then utilize the token to authenticate against the respective audience, representing its backend service:

JSON

```
"privileges": [  
    {
```

```

        "name": "AADAuthentication",
        "parameters": {
            "COM": "https://contoso.com",
            "CN": "https://contoso.cn"
        }
    }
]

```

In the `pbviz.json` file, set the API version to **5.9.1** or higher:

The newly exposed **AcquireAADTokenService** contains two methods:

- **acquireAADToken**: Returns an authentication token payload of type `AcquireAADTokenResult` for the visual or null if it can't be fetched.

TypeScript

```

/**
 * Enum representing the various clouds supported by the Authentication
API.
 */
export const enum CloudName {
    COM = "COM",           // Commercial Cloud
    CN = "CN",             // China Cloud
    GCC = "GCC",            // US Government Community Cloud
    GCCHIGH = "GCCHIGH",   // US Government Community Cloud High
    DOD = "DOD",            // US Department of Defense Cloud
}

/**
 * Interface representing information about the user associated with
the token.
 */
export interface AcquireAADTokenUserInfo {
    userId?: string; // Unique identifier for the user
    tenantId?: string; // Unique identifier for the tenant
}

/**
 * Interface representing information about the fabric environment.
 */
export interface AcquireAADTokenFabricInfo {
    cloudName?: CloudName; // Name of the cloud environment
}

/**
 * Interface representing the result of acquiring a Microsoft Entra ID
token.
 */
export interface AcquireAADTokenResult {
    accessToken?: string; // Access token issued by Microsoft
    Entra ID
}

```

```

    expiresOn?: number;           // Expiration time of the access token
    userInfo?: AcquireAADTokenUserInfo; // Information about the
    user associated with the token
    fabricInfo?: AcquireAADTokenFabricInfo; // Information about the
    fabric environment
}

```

- **acquireAADTokenstatus:** Returns one of the following privilege statuses associated with acquiring the token.
  - **Allowed:** The privilege is allowed in the current environment.
  - **NotDeclared:** The privilege declaration is missing in visual capabilities section.
  - **NotSupported:** The privilege isn't supported in the current environment.
  - **DisabledByAdmin:** The Fabric administrator denied privilege usage.

The following sample code demonstrates how to acquire a Microsoft Entra ID token using the API:

TypeScript

```

// Step 1: Check the status of AAD token acquisition
const acquireTokenStatus = await
this.acquireAADTokenService.acquireAADTokenstatus();

// Step 2: Verify if acquiring the token is allowed
if (acquireTokenStatus === PrivilegeStatus.Allowed) {

    // Step 3: Acquire the Microsoft Entra ID token
    const acquireAADTokenResult: AcquireAADTokenResult = await
this.acquireAADTokenService.acquireAADToken();

    // Step 4: Confirm successful acquisition of the access token
    if (acquireAADTokenResult.accessToken) {

        // Step 5: Call your backend API with the obtained token
    }
}

// Step 6: Handle unsuccessful AAD token acquisition

```

## Considerations and limitations

Token acquisition is blocked if any of the following conditions apply:

- The tenant switch is turned off.
- The user isn't signed in (in Desktop).
- The ISV didn't preauthorize the Power BI application.

- The format of the AADAuthentication privilege parameter is invalid.
- The visual isn't publicly approved or isn't a debug visual.
- The visual's backend service, configured as the audience by the visual, doesn't have appropriate consents for the Graph API in the consumer tenant using the visual.  
For more about consent, see [tenant administrator consent](#).

## Related content

[Microsoft Entra ID application setup](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Microsoft Entra ID application setup

Article • 07/15/2024

To use the Authentication API, the ISV must first register an application in Microsoft Entra ID for each cloud to be supported, and preauthorize the Power BI applications with a dedicated scope for each visual. The tenant administrator then needs to grant consent. This article outlines all of these essential steps.

The Authentication API is supported in the following clouds:

- **COM (Required)** - Commercial Cloud
- **CN** - China Cloud
- **GCC** - US Government Community Cloud
- **GCCHIGH** - US Government Community Cloud High
- **DOD** - US Department of Defense Cloud

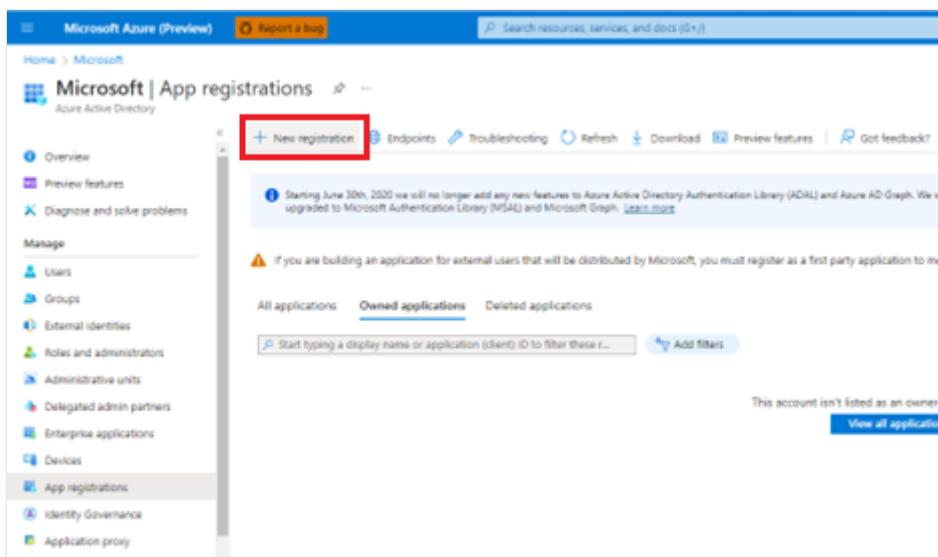
## Register the app in Microsoft Entra ID

For each cloud the visual is intended to support, follow these steps:

1. Navigate to the respective Azure portal and go to **App registrations**.

- [COM \(required\) and GCC ↗](#)
- [CN ↗](#)
- [GCCHIGH and DOD ↗](#)

2. Select + New Registration



3. On the Register an application page, do the following:

- a. Enter your desired application name in the **Name** section.

- b. Select **Accounts in any organizational directory (Any Microsoft Entra ID tenant - Multitenant)** in the **Supported account types** section.
- c. Select **Register**.

Home > App registrations >

## Register an application

**⚠️** If you are building an application for external users that will be distributed by Microsoft, you must register as a first party application to meet all security, privacy, and compliance policies. [Read our decision guide](#)

\* Name  
The user-facing display name for this application (this can be changed later).  
 ✓

Supported account types  
Who can use this application or access this API?  
 Accounts in this organizational directory only (Microsoft only - Single tenant)  
 Accounts in any organizational directory (Any Microsoft Entra ID tenant - Multitenant)  
 Accounts in any organizational directory (Any Microsoft Entra ID tenant - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)  
 Personal Microsoft accounts only  
[Help me choose...](#)

By proceeding, you agree to the Microsoft Platform Policies

**Register**

4. Once your application is successfully registered, select **Expose an API** on the left side menu.

Home > App registrations >

**contoso-app** ⚡ ...

Search

Overview Quickstart Integration assistant

Manage

- Branding & properties
- Authentication
- Certificates & secrets
- Token configuration
- API permissions
- Expose an API**

Delete Endpoints Preview features

Essentials

Display name	:	<a href="#">contoso-app</a>
Application (client) ID	:	
Object ID	:	
Directory (tenant) ID	:	

Supported account types : [Multiple organizations](#)

**Info** Welcome to the new and improved App registrations. Looking to learn how it's ch

**Info** Starting June 30th, 2020 we will no longer add any new features to Azure Active D security updates but we will no longer provide feature updates. Applications will n

5. In the **Application ID URI** field, select **Add**.

The screenshot shows the Azure portal interface for managing app registrations. On the left, a sidebar menu lists various management options: Overview, Quickstart, Integration assistant, Manage (with sub-options: Branding & properties, Authentication, Certificates & secrets, Token configuration, API permissions, Expose an API, and App roles). The 'Expose an API' option is currently selected and highlighted with a grey background. The main content area is titled 'contoso-app | Expose an API'. At the top right of this area, there is a button labeled 'Add' with a red box drawn around it. Below this, a section titled 'Scopes defined by this API' contains a note about defining custom scopes to restrict access. A link 'Go to App roles' is present. A large 'Add a scope' button is visible. A table titled 'Scopes' shows one entry: 'No scopes have been defined'. To the right of the table, a column header 'Who can consent' is shown.

6. In the **Edit Application ID URI** field, enter your [Verified Custom Domain](#), ensuring that it begins with "<https://>" and doesn't contain "[onmicrosoft.com](https://onmicrosoft.com)", and select **Save**.

To add a custom domain:

- Navigate to Microsoft Entra ID Custom domain names.
- Add your custom domain.

This screenshot shows a modal dialog box titled 'Edit application ID URI' overlaid on the main Azure portal page. The dialog has a single input field labeled 'Application ID URI' containing the value 'https://contoso.com'. Below the input field, a descriptive note explains that the globally unique URI is used to identify the web API and serves as a prefix for scopes and access tokens. At the bottom of the dialog are two buttons: 'Save' and 'Discard'. The background of the portal page shows the same 'Expose an API' blade as the previous screenshot, with the 'Manage' sidebar visible on the left.

### ⚠ Note

The application URI can be manually added to the application manifest under the “`identifierUris`” array.

The screenshot shows the Microsoft Azure App registrations page. The URL is [https://apps.dev.microsoft.com/#/appRegistrations/contoso-app](#). The page title is "contoso-app | Manifest". On the left, there's a sidebar with links like Overview, Quickstart, Integration assistant, Manage (Branding & properties, Authentication, Certificates & secrets, Token configuration, API permissions, Expose an API, App roles, Owners, Roles and administrators), and Manifest. The "Manifest" link is highlighted. At the top right, there are Save, Discard, Upload, Download, and Got feedback? buttons. Below the sidebar, a search bar and a JSON code editor are present. The JSON code is as follows:

```

11     "description": null,
12     "certification": null,
13     "disabledByMicrosoftStatus": null,
14     "groupMembershipClaims": null,
15     "identifierUris": [
16       "https://contoso.com",
17       "https://contoso.net"
18     ],
19     "informationalUrls": {
20       "termsOfService": null,
21       "support": null,
22       "privacy": null,
23       "marketing": null
24     },
25     "keyCredentials": [],
26     "knownClientApplications": [],
27     "logoUrl": null,
28     "logoutUrl": null,
29     "name": "contoso-app",
30     "notes": null,
31     "oauth2AllowIdTokenImplicitFlow": false,

```

## 7. Select + Add a scope.

8. In the Scope name field, enter `<visual_guid>_CV_ForPBI` and add the required information. Fill in the Admin consent fields. Then select Add scope button. (There's a 40 characters scope length limitation, but you can manually modify the scope name in the registered application manifest to manage this limitation).

The screenshot shows the Microsoft Azure App registrations page for the "contoso-app" application. The URL is [https://apps.dev.microsoft.com/#/appRegistrations/contoso-app](#). The sidebar shows the "Expose an API" section selected. The main content area shows the "Expose an API" page with the "Application ID URI" set to `https://contoso.com`. On the right, a modal window titled "Edit a scope" is open. The modal has fields for "Scope name" (set to `AuthApiTestVisual_CV_ForPBI`), "Who can consent" (set to "Admins and users"), "Admin consent display name" (set to "Admins only"), "Admin consent description" (set to "Admin consent description"), "User consent display name" (set to "e.g. Read your files"), "User consent description" (set to "e.g. Allows the app to read your files"), and "State" (set to "Enabled"). There are also "Save", "Discard", and "Delete" buttons at the top of the modal.

## 9. To preauthorize Power BI applications:

- a. Select + Add a client application.

b. Enter the **Power BI WFE** application appId in the **Client ID** field of the right-hand window.

- **COM (required)** and **CN**: "871c010f-5e61-4fb1-83ac-98610a7e9110".
- **GCC, GCCHIGH**, and **DOD**: "ec04d7d8-0476-4acd-bce4-81f438363d37".

c. Select your desired scope.

d. Select **Add application**.

e. Repeat this process with:

- **Power BI Desktop**:
  - **COM (required)** and **CN**: "7f67af8a-fedc-4b08-8b4e-37c4d127b6cf".
  - **GCC, GCCHIGH**, and **DOD**: "6807062e-abc9-480a-ae93-9f7deee6b470".
- **Power BI Mobile**:
  - **COM (required)** and **CN**: "c0d2a505-13b8-4ae0-aa9e-cddd5eab0b12".
  - **GCC, GCCHIGH** and **DOD**: "ce76e270-35f5-4bea-94ff-eab975103dc6".

# ISV consent

The tenant administrator can determine whether or not users are allowed to consent for themselves. This consent process takes place outside of Power BI.

ISV backend application (for example, <https://contoso.com>) should be consented to Graph API and other dependencies (by users or tenant administrators) according to standard AAD rules:

If the ISV application is running on a different tenant than the visual consumer's tenant, grant consent for the ISV's application in one of the following ways:

- Administrator preconsent:

Follow the instructions in [Grant tenant-wide admin consent to an application](#).

Replace the **tenant-wide admin consent URL** with the respective link for each cloud:

- COM and GCC:

```
https://login.microsoftonline.com/{organization}/adminconsent?client_id={clientId}
```

- CN: [https://login.partner.microsoftonline.cn/{organization}/adminconsent?client\\_id={clientId}](https://login.partner.microsoftonline.cn/{organization}/adminconsent?client_id={clientId})

- GCCHIGH and DOD:

```
https://login.microsoftonline.us/{organization}/adminconsent?client_id={clientId}
```

- Interactive consent:

If the tenant administrator didn't preconsent, any user that uses a visual that triggers the API receives a one-time consent prompt when rendering the visual.

See [Application consent experience](#) for more information.

## Related content

- [Authentication API](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

# On-object formatting API (preview)

Article • 02/19/2024

[On-object formatting](#) allows users to quickly and easily modify the format of visuals by directly selecting the elements they want to modify. When an element is selected, the format pane automatically navigates and expands the specific formatting setting for the selected element. For more information about on-object formatting, see [On-object formatting in Power BI Desktop](#).

To add these functionalities to your visual, each visual needs to provide a subselection style option and shortcut for each subselectable region.

## ⓘ Note

- Visuals that support on-object formatting need to implement the [getFormattingModel API](#) which is available from API version 5.1.
- If you are using powerbi-visuals-utils-formattingmodel, use version 6.0.0 at least.

## Create an on-object experience

Use the subselection service when the user selects a subselectable element to send Power BI the subselection. Provide the subselection styles and shortcuts using the [subselection API](#). The [subselection helper](#) can be used to simplify the process.

## Format mode

Format mode is a new mode where the user can turn `onObject` formatting on and off when in authoring mode. The visual is updated with the status of the format mode in the update options. The update options also include the currently subselected subSelection as `CustomVisualSubSelection`.

## How to implement the on-object formatting API

### Capabilities file

In the `capabilities.json` file, add the following properties to declare that the visual supports on-object formatting:

#### JSON

```
{  
  "supportsOnObjectFormatting": true,  
  "enablePointerEventsFormatMode": true,  
}
```

## IVisual interface

The visual needs to implement the `VisualOnObjectFormatting` interface as part of the `IVisual` interface.

`VisualOnObjectFormatting` contains three methods:

- `getSubSelectionStyles`
- `getSubSelectionShortcuts`
- `getSubSelectables`

### getSubSelectionStyles

Each visual is required to implement a `getSubSelectionStyles` method, which is called when a subselectable element is subselected. The `getSubSelectionStyles` method is provided with the current subselected elements as a `CustomVisualSubSelection` array and is expected to return either a `SubSelectionStyles` object or `undefined`.

There are three categories of subselection styles that cover most scenarios:

- Text
- Numeric Text
- Shape

Each `SubSelectionStyles` object provides a different experience for the user for modifying the style of an element.

### getSubSelectionShortcuts

To provide more options for the user, the visual must implement the `getSubSelectionShortcuts` method. This method returns either `VisualSubSelectionShortcuts` or `undefined`. Additionally, if `SubSelectionShortcuts` are

provided, a `VisualNavigateSubSelectionShortcut` must also be provided so that when a user subselects an element and the format pane is open, the pane automatically scrolls to the appropriate card.

There are several subselection shortcuts to modify the visual state. Each one defines a menu item in the context menu with the appropriate label.

**Sub-Selection Disambiguation Menu:** The On-Object disambiguation menu provides a method for users to select their desired subselection when it's not clear which visual element is being subselected. This often happens when the user subselects the background of the visual. For the disambiguous menu to present more subselections, the visual must provide all subselections via the `getSubSelectables` method.

## getSubSelectables

To provide subselections to the disambiguation menu, the visual needs to implement the `getSubSelectables` method. This method is provided an optional `filterType` argument, of type `SubSelectionStylesType` and returns an array of `CustomVisualSubSelection` or `undefined`. If the `HTMLSubSelectionHelper` is being utilized to create a subselection, the `HTMLSubSelectionHelper.getSubSelectables()` method can be used to gather subselectable elements from the DOM.

**Sub-Selection Direct Text Editing:** With On-Object formatting, you can double-click the text of a subs-electable element to directly edit it. To provide direct-edit capability, you need to provide a `RectangleSubSelectionOutline` with the appropriate `cVDirectEdit` Property populated with a `SubSelectableDirectEdit` object. The outline can either be provided as a custom outline or, if you're using the `HTMLSubSelectionHelper` you can use the `SubSelectableDirectEdit` attribute. (See the attributes provided by the `HTMLSubSelectionHelper`)

Adding a direct edit for a specific datapoint (using selectors) isn't yet supported.

## FormattingId interface

The following interface is used to reference the `subSelection` shortcuts and styles.

TypeScript

```
interface FormattingId {
    objectName: string;
    propertyName: string;
```

```
        selector?: powerbi.data.Selector;
    }
```

- `objectName`: the object name as declared in the `capabilities.json`.
- `propertyName`: the property name of an object as declared in the `capabilities.json`.
- `selector`: if the datapoint has a `selectionId`, use `selectionId.getSelector()`, this selector must be the same as provided for the formatting model slice.

## Examples

In this example, we build a custom visual that has two objects, `colorSelector` and `directEdit`. We use the `HTMLSubSelectionHelper` from the `onobjectFormatting` utils, to handle most of the subSelection job. For more information, see [on-object utils](#).

First, we build cards for the formatting pane and provide `subSelectionShortcuts` and `styles` for each subselectable.

## Define the objects

Define the objects and declare that the visual is supporting OnObject Formatting in the `capabilities.json`:

JSON

```
"objects": {
    "directEdit": {
        "properties": {
            "show": {
                "displayName": "Show",
                "type": {
                    "bool": true
                }
            },
            "textProperty": {
                "displayName": "Text",
                "type": {
                    "text": true
                }
            },
            "fontFamily": {
                "type": {
                    "formatting": {
                        "fontFamily": true
                    }
                }
            },
            "fontSize": {
```

```
"type": {
    "formatting": {
        "fontSize": true
    }
},
"bold": {
    "type": {
        "bool": true
    }
},
"italic": {
    "type": {
        "bool": true
    }
},
"underline": {
    "type": {
        "bool": true
    }
},
"fontColor": {
    "displayName": "Font Color",
    "type": {
        "fill": {
            "solid": {
                "color": true
            }
        }
    }
},
"background": {
    "displayName": "Background",
    "type": {
        "fill": {
            "solid": {
                "color": true
            }
        }
    }
},
"position": {
    "displayName": "Position",
    "type": {
        "enumeration": [
            { "displayName": "Left", "value": "Left" }, { "displayName": "Right", "value": "Right" }
        ]
    }
},
"colorSelector": {
    "displayName": "Data Colors",
    "properties": {
```

```
"fill": {
    "displayName": "Color",
    "type": {
        "fill": {
            "solid": {
                "color": true
            }
        }
    }
},
},
"supportsOnObjectFormatting": true,
"enablePointerEventsFormatMode": true,
```

## Build the formatting cards

Build their formatting cards using the [formattingModel utils](#).

### Color selector card settings

TypeScript

```
class ColorSelectorCardSettings extends Card {
    name: string = "colorSelector";
    displayName: string = "Data Colors";
    slices = [];
}
```

Add a method to the formattingSetting so we can populate the slices dynamically for the colorSelector object (our datapoints).

TypeScript

```
populateColorSelector(dataPoints: BarChartDataPoint[]) {
    let slices: formattingSettings.ColorPicker[] =
this.colorSelector.slices;
    if (dataPoints) {
        dataPoints.forEach(dataPoint => {
            slices.push(new formattingSettings.ColorPicker({
                name: "fill",
                displayName: dataPoint.category,
                value: { value: dataPoint.color },
                selector: dataPoint.selectionId.getSelector(),
            }));
    });
}
```

```
    }
}
```

We pass the selector of the specific datapoint in the selector field. This selector is the one used when implementing the get APIs of the OnObject.

## Direct edit card settings

TypeScript

```
class DirectEditSettings extends Card {
    displayName = 'Direct Edit';
    name = 'directEdit';
    private minFontSize: number = 8;
    private defaultFontSize: number = 11;
    show = new formattingSettings.ToggleSwitch({
        name: "show",
        displayName: undefined,
        value: true,
    });
    topLevelSlice = this.show;
    textProperty = new formattingSettings.TextInput({
        displayName: "Text Property",
        name: "textProperty",
        value: "What is your quest?",
        placeholder: ""
    });
    position = new formattingSettings.ItemDropdown({
        name: 'position',
        items: [{ displayName: 'Left', value: 'Left' }, { displayName: 'Right', value: 'Right' }],
        value: { displayName: 'Right', value: 'Right' }
    });
    font = new formattingSettings.FontControl({
        name: "font",
        displayName: 'Font',
        fontFamily: new formattingSettings.FontPicker({
            name: "fontFamily",
            displayName: "Font Family",
            value: "Segoe UI, wf_segoe-ui_normal, helvetica, arial, sans-serif"
        }),
        fontSize: new formattingSettings.NumUpDown({
            name: "fontSize",
            displayName: "Font Size",
            value: this.defaultFontSize,
            options: {
                minValue: {
                    type: powerbi.visuals.ValidatorType.Min,
                    value: this.minFontSize,
                }
            }
        })
    });
}
```

```

        }),
        bold: new formattingSettings.ToggleSwitch({
            name: 'bold',
            displayName: "Font Size",
            value: true
        }),
        italic: new formattingSettings.ToggleSwitch({
            name: 'italic',
            displayName: "Font Size",
            value: true
        }),
        underline: new formattingSettings.ToggleSwitch({
            name: 'underline',
            displayName: "Font Size",
            value: true
        })
    });
fontColor = new formattingSettings.ColorPicker({
    name: "fontColor",
    displayName: "Color",
    value: { value: "#000000" }
});
background = new formattingSettings.ColorPicker({
    name: "background",
    displayName: "Color",
    value: { value: "#FFFFFF" }
});
slices = [this.show, this.textProperty, this.font, this.fontColor,
this.background, this.position];
}

```

## Use subselection helper attributes

Add the `HTMLSubSelectionHelper` attributes to our objects. To see which attributes the `HTMLSubSelectionHelper` provide, check the on [object utils documentation](#).

- For the directEdit attribute:

TypeScript

```

import {
    HtmlSubSelectableClass, HtmlSubSelectionHelper,
    SubSelectableDirectEdit as SubSelectableDirectEditAttr,
    SubSelectableDisplayNameAttribute, SubSelectableObjectNameAttribute,
    SubSelectableTypeAttribute
} from 'powerbi-visuals-utils-onobjectutils';

const DirectEdit: powerbi.visuals.SubSelectableDirectEdit = {
    reference: {
        objectName: 'directEdit',
        propertyName: 'textProperty'
    }
}

```

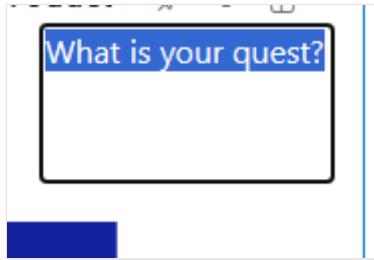
```

        },
        style: SubSelectableDirectEditStyle.Outline,
    };
    private visualDirectEditSubSelection = JSON.stringify(DirectEdit);

    this.directEditElement
        .classed('direct-edit', true)
        .classed('hidden',
    !this.formattingSettings.directEditSettings.show.value)
        .classed(HtmlSubSelectableClass, options.formatMode &&
    this.formattingSettings.directEditSettings.show.value)
        .attr(SubSelectableObjectNameAttribute, 'directEdit')
        .attr(SubSelectableDisplayNameAttribute, 'Direct Edit')
        .attr(SubSelectableDirectEditAttr,
    this.visualDirectEditSubSelection)

```

The `HTMLSubSelectionHelper` uses the `SubSelectableDirectEditAttr` attribute to provide the `directEdit` reference of the `directEdit` outline, so a direct edit starts when a user double clicks on the element.



- For the colorSelector:

#### TypeScript

```

barSelectionMerged
    .attr(SubSelectableObjectNameAttribute, 'colorSelector')
    .attr(SubSelectableDisplayNameAttribute, (dataPoint:
BarChartDataPoint) =>
    this.formattingSettings.colorSelector.slices[dataPoint.index].displayName)
    .attr(SubSelectableTypeAttribute,
powerbi.visuals.SubSelectionStylesType.Shape)
    .classed(HtmlSubSelectableClass, options.formatMode)

```

## Define references

Define the following interface to simplify the examples:

### ⓘ Note

The `cardUid` you provide should be the same as the one provided for the `getFormattingModel` API. For example, if you're using `powerbi-visuals-utils-formattingmodel`, provide the `cardUid` as *Visual-cardName-card*, where the `cardName` is the name you assigned to this card in the formatting model settings. Otherwise, provide it as the `Visual-cardUid` you assigned to this card.

TypeScript

```
interface References {
    cardUid?: string;
    groupUid?: string;
    fill?: FormattingId;
    font?: FormattingId;
    fontColor?: FormattingId;
    show?: FormattingId;
    fontFamily?: FormattingId;
    bold?: FormattingId;
    italic?: FormattingId;
    underline?: FormattingId;
    fontSize?: FormattingId;
    position?: FormattingId;
    textProperty?: FormattingId;
}
```

For the purpose of this example, create an enum for the objects names:

TypeScript

```
const enum BarChartObjectNames {
    ColorSelector = 'colorSelector',
    DirectEdit = 'directEdit'
}
```

- References for the `directEdit` object:

TypeScript

```
const directEditReferences: References = {
    cardUid: 'Visual-directEdit-card',
    groupUid: 'directEdit-group',
    fontFamily: {
        objectName: BarChartObjectNames.DirectEdit,
        propertyName: 'fontFamily'
    },
    bold: {
        objectName: BarChartObjectNames.DirectEdit,
        propertyName: 'bold'
    },
}
```

```

italic: {
    objectName: BarChartObjectNames.DirectEdit,
    propertyName: 'italic'
},
underline: {
    objectName: BarChartObjectNames.DirectEdit,
    propertyName: 'underline'
},
fontSize: {
    objectName: BarChartObjectNames.DirectEdit,
    propertyName: 'fontSize'
},
fontColor: {
    objectName: BarChartObjectNames.DirectEdit,
    propertyName: 'fontColor'
},
show: {
    objectName: BarChartObjectNames.DirectEdit,
    propertyName: 'show'
},
position: {
    objectName: BarChartObjectNames.DirectEdit,
    propertyName: 'position'
},
textProperty: {
    objectName: BarChartObjectNames.DirectEdit,
    propertyName: 'textProperty'
}
};


```

- For `colorSelector`:

TypeScript

```

const colorSelectorReferences: References = {
    cardUid: 'Visual-colorSelector-card',
    groupUid: 'colorSelector-group',
    fill: {
        objectName: BarChartObjectNames.ColorSelector,
        propertyName: 'fill'
    }
};

```

## Implement APIs

Now let's implement the get APIs for the `onObject` formatting and provide them in the `visualOnObjectFormatting`:

1. In the constructor code, provide the get methods in the `visualOnObjectFormatting`:

## TypeScript

```
public visualOnObjectFormatting:  
powerbi.extensibility.visual.VisualOnObjectFormatting;  
constructor(options: VisualConstructorOptions) {  
    this.subSelectionHelper =  
    HtmlSubSelectionHelper.createHtmlSubselectionHelper({  
        hostElement: options.element,  
        subSelectionService: options.host.subSelectionService,  
        selectionIdCallback: (e) =>  
    this.selectionIdCallback(e),  
    });  
  
    this.visualOnObjectFormatting = {  
        getSubSelectionStyles: (subSelections) =>  
    this.getSubSelectionStyles(subSelections),  
        getSubSelectionShortcuts: (subSelections, filter) =>  
    this.getSubSelectionShortcuts(subSelections, filter),  
        getSubSelectables: (filter) => this.  
getSubSelectables(filter)  
    }  
}  
  
private getSubSelectionStyles(subSelections:  
CustomVisualSubSelection[]): powerbi.visuals.SubSelectionStyles |  
undefined {  
    const visualObject = subSelections[0]?.customVisualObjects[0];  
    if (visualObject) {  
        switch (visualObject.objectName) {  
            case BarChartObjectName.ColorSelector:  
                return this.getColorSelectorStyles(subSelections);  
            case BarChartObjectName.DirectEdit:  
                return this.getDirectEditStyles();  
        }  
    }  
}  
  
private getSubSelectionShortcuts(subSelections:  
CustomVisualSubSelection[], filter: SubSelectionShortcutsKey |  
undefined): VisualSubSelectionShortcuts | undefined {  
    const visualObject = subSelections[0]?.  
customVisualObjects[0];  
    if (visualObject) {  
        switch (visualObject.objectName) {  
            case BarChartObjectName.ColorSelector:  
                return  
this.getColorSelectorShortcuts(subSelections);  
            case BarChartObjectName.DirectEdit:  
                return this.getDirectEditShortcuts();  
        }  
    }  
}
```

## 2. Implement the getSubSelection shortcuts and style for the colorSelector:

TypeScript

```
private getColorSelectorShortcuts(subSelections:  
CustomVisualSubSelection[]): VisualSubSelectionShortcuts {  
    const selector =  
subSelections[0].customVisualObjects[0].selectionId?.getSelector();  
    return [  
        {  
            type: VisualShortcutType.Reset,  
            relatedResetFormattingIds: [  
                ...colorSelectorReferences.fill,  
                selector  
            ],  
        },  
        {  
            type: VisualShortcutType.Navigate,  
            destinationInfo: { cardUid:  
colorSelectorReferences.cardUid },  
            label: 'Color'  
        }  
    ];  
}
```

The above shortcut returns relevant menu item in the context menu and adds the following functionalities:

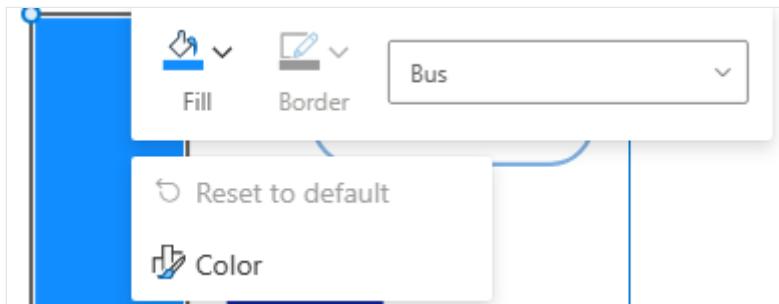
- VisualShortcutType.Navigate: when a user selects on one of the bars (data point), and the formatting pane is open, the format pane scrolls to the color selector card and open it
- VisualShortcutType.Reset: adds a reset shortcut to the context menu. It's enabled if the fill color was changed.

TypeScript

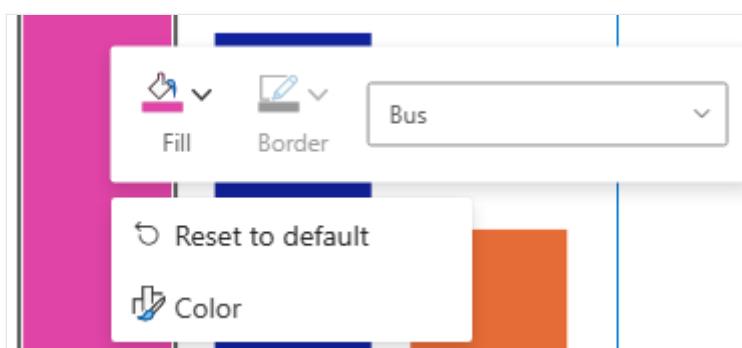
```
private getColorSelectorStyles(subSelections:  
CustomVisualSubSelection[]): SubSelectionStyles {  
    const selector =  
subSelections[0].customVisualObjects[0].selectionId?.getSelector();  
    return {  
        type: SubSelectionStylesType.Shape,  
        fill: {  
            label: 'Fill',  
            reference: {  
                ...colorSelectorReferences.fill,  
                selector  
            },  
        },  
    },
```

```
    };
}
```

When a user right-clicks on a bar, the following appears:



When changing the color:



## Subsection shortcuts

To implement the subSelection shortcuts and styles for the directEdit:

TypeScript

```
private getDirectEditShortcuts(): VisualSubSelectionShortcuts {
    return [
        {
            type: VisualShortcutType.Reset,
            relatedResetFormattingIds: [
                directEditReferences.bold,
                directEditReferences.fontFamily,
                directEditReferences.fontSize,
                directEditReferences.italic,
                directEditReferences.underline,
                directEditReferences.textColor,
                directEditReferences.textProperty
            ]
        },
        {
            type: VisualShortcutType.Toggle,
            relatedToggledFormattingIds: [
                ...directEditReferences.show,
            ],
        }
    ];
}
```

```

        ...directEditReferences.show,
        disabledLabel: 'Delete',
    },
    {
        type: VisualShortcutType.Picker,
        ...directEditReferences.position,
        label: 'Position'
    },
    {
        type: VisualShortcutType.Navigate,
        destinationInfo: { cardUid: directEditReferences.cardUid },
        label: 'Direct edit'
    }
];
}

```

This shortcut adds a relevant menu item in the context menu and adds the following functionalities:

- VisualShortcutType.Reset: adds a reset to the default item to the context menu, when one of the properties provided in relatedResetFormattingIds array changes.
- VisualShortcutType.Toggle: adds a Delete options to the context menu. When clicked, the toggle switch for the *directEdit* card is turned off.
- VisualShortcutType.Picker: Adds an option in the context menu to pick between Right and Left, since we added the position slice in the formatting card for the *directEdit*.
- VisualShortcutType.Navigate: When the format pane is open and the user selects the *directEdit* element, the format pane scrolls and opens the *directEdit* card.

#### TypeScript

```

private getDirectEditStyles(): SubSelectionStyles {
    return {
        type: powerbi.visuals.SubSelectionStylesType.Text,
        fontFamily: {
            reference: {
                ...directEditReferences.fontFamily
            },
            label: 'font family'
        },
        bold: {
            reference: {
                ...directEditReferences.bold
            },
            label: 'bold'
        },
        italic: {
            reference: {
                ...directEditReferences.italic
            },

```

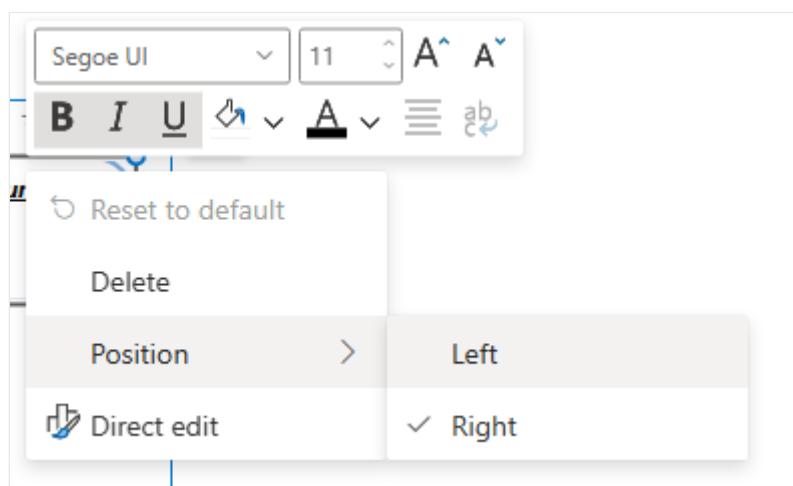
```

        label: 'italic'
    },
    underline: {
        reference: {
            ...directEditReferences.underline
        },
        label: 'underline'
    },
    fontSize: {
        reference: {
            ...directEditReferences.fontSize
        },
        label: 'font size'
    },
    fontColor: {
        reference: {
            ...directEditReferences.fontColor
        },
        label: 'font color'
    },
    background: {
        reference: {
            objectName: 'directEdit',
            propertyName: 'background'
        },
        label: 'background'
    }
}
}

```

We provided the relevant properties as we added them in the `formattingSettings`.

The following image illustrates how the UI looks when right-clicking on the `directEdit` element:



## Localization

The visual should handle the localization and provide localized strings.

## GitHub resources

- All on object formatting interfaces can be found in (link to be provided once the API is released) in on-object-formatting-api.d.ts
- We recommend using the [on object utils], which include the [HTMLSubSelectionHelper](link to be provided once the API is released)
- You can find an example of a custom visual [SampleBarChart](#) that uses API version 5.8.0 and implements the support for the on object formatting using the on object utils at (link to be provided once the API is released)

## Related content

- [Subselection API](#)
- [On-object utils](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Subselection API (preview)

Article • 02/19/2024

[On-object formatting](#) allows users to quickly and easily modify the format of visuals by directly selecting the elements they want to modify. When an element is selected, the format pane automatically navigates and expands the specific formatting setting for the selected element. As part of [on-object formatting](#), the subselection service is used to send subselections and outlines to Power BI.

## How to use the Subselection API

The SubSelection Service provides two methods:

- [subSelect](#)
- [updateRegionOutlines](#)

### subSelect

Sends the subselection for Power BI to use when a user selects an element that allows subselections.

TypeScript

```
subSelect(args: visuals.CustomVisualSubSelection | undefined): void

CustomVisualSubSelection
interface CustomVisualSubSelection {
    customVisualObjects: CustomVisualObject[];
    displayName: string;
    subSelectionType: SubSelectionStylesType;
    selectionOrigin: SubSelectionOrigin;
    /** Whether to show the UI for this sub-selection, like
formatting context menus and toolbar */
    showUI: boolean;
    /** If immediate direct edit should be triggered, the ID of the
sub-selection outline to edit */
    immediateDirectEdit?: string;
    metadata?: unknown;
}

interface CustomVisualObject {
    objectName: string;
    selectionId: powerbi.visuals.ISelectionId | undefined;
}
```

This method has the following parameters:

- customVisualObjects: an array that contains `customVisualObjects`, the `objectName` of the object should be the same as the one declared in the `capabilities.json`, and the `selectionId` for the selected data point, if it exists.
- displayName: the display name should be localized if the visual supports localization.
- subSelectionType: the type of the subselection (shape, text, or Numeric text).
- selectionOrigin: the coordinates of the subselected element.
- showUI: Whether to show the UI for this subselection, like formatting context menus and toolbar.
- immediateDirectEdit: If immediate direct edit should be triggered, the ID of the subselection outline to edit.

If you don't use the [HTMLSubSelectionHelper](#), you need to manage the subselections.

## Subselection example

In this example, we add an event listener to the host element, for the right-click, context menu events.

TypeScript

```
constructor(options: VisualConstructorOptions) {
    this.hostElement = options.element;
    this.subSelectionService = options.host.subSelectionService;
    ...
}

public update(options: VisualUpdateOptions) {
    if (options.formatMode) {
        // remove event listeners which are irrelevant for format mode.
        ...
        this.hostElement.addEventListener('click',
            this.handleFormatModeClick);
        this.hostElement.addEventListener('contextmenu',
            this.handleFormatModeContextMenu);
    } else {
        this.hostElement.removeEventListener('click',
            this.handleFormatModeClick);
        this.hostElement.removeEventListener('contextmenu',
            this.handleFormatModeContextMenu);
        ...
        // add event listeners which are irrelevant for format mode
    }
}

private handleFormatModeClick(event: MouseEvent): void {
```

```

        this.subSelectFromEvent(event, true /**showUI */);
    }

private handleFormatModeContextMenu(event: MouseEvent): void {
    this.subSelectFromEvent(event, false);
}

private subSelectFromEvent(event: MouseEvent, showUI: boolean): void {
    //find the element which was selected and fill the needed fields
    const cVObject: powerbi.visuals.CustomVisualObject = {
        objectName: 'myObject',//the object name that is relevant to the
    clicked element
        selectionId: undefined
    };
    const subSelection: CustomVisualSubSelection = {
        customVisualObjects: [cVObject],
        displayName: 'myObject',
        selectionOrigin: {
            x: event.clientX,
            y: event.clientY
        },
        subSelectionType: SubSelectionStylesType.Shape,// choose the
    relevant type
        showUI
    };
    this.subSelectionService.subSelect(subSelection);
}

```

## updateRegionOutlines

This method sends outlines to Power BI to render. Use it in the `update` method of the visual since that's where Power BI sends the subselection that the visual sent previously. You can also use it when you want to render an outline for a hovered element.

typescript

```

updateRegionOutlines(outlines: visuals.SubSelectionRegionOutline[]): void

SubSelectionRegionOutline
interface SubSelectionRegionOutline {
    id: string;
    visibility: SubSelectionOutlineVisibility; // controls
visibility for outlines
    outline: SubSelectionOutline;
}

```

If you don't use the [HTMLSubSelectionHelper](#), you have to manually manage the outlines and their state (if they're active, hovered or not visible).

## Update region outlines example

In this example we assume that we have an object called `myObject`, and we want to render a rectangle outline when the relevant element is hovered. We use the code in the previous example for `subSelect`.

In the update, we also need to add an event listener for the `pointerover` event.

We want to manage our outlines using a Record.

TypeScript

```
private subSelectionRegionOutlines: Record<string, SubSelectionRegionOutline> = {};  
  
public update(options: VisualUpdateOptions) {  
    if (options.formatMode) {  
        // remove event listeners which are irrelevant for format mode.  
        ...  
        this.hostElement.addEventListener('click',  
this.handleFormatModeClick);  
        this.hostElement.addEventListener('contextmenu',  
this.handleFormatModeContextMenu);  
        this.hostElement.addEventListener('pointerover',  
this.handleFormatModePointerOver);  
    } else {  
        this.hostElement.removeEventListener('click',  
this.handleFormatModeClick);  
        this.hostElement.removeEventListener('contextmenu',  
this.handleFormatModeContextMenu);  
        this.hostElement.removeEventListener('pointerover',  
this.handleFormatModePointerOver);  
    }  
    ...  
    // add event listeners which are irrelevant for format mode  
}  
}  
  
private handleFormatModePointerOver(event: MouseEvent): void {  
    // use the event to extract the element that was hovered.  
    // in this example we assume that we found the element and it is  
    related to object called myObject.  
    // we need to clear previously hovered outlines before rendering  
    const regionOutlines = getValues(this.subSelectionRegionOutlines);  
    const hoveredOutline = regionOutlines.find(outline =>  
outline.visibility === SubSelectionOutlineVisibility.Hover);  
    if (hoveredOutline) {  
        this.subSelectionRegionOutlines[hoveredOutline.id] = {  
            ...this.subSelectionRegionOutlines[hoveredOutline.id],  
            visibility:  
powerbi.visuals.SubSelectionOutlineVisibility.None  
        };  
    }  
}
```

```
// now we will build the outline for myObject relevant element.
let element: HTMLElement;// assume we found the relevant element.
const domRect = element.getBoundingClientRect();
const { x, y, width, height } = domRect;
const outline: powerbi.visuals.RectangleSubSelectionOutline = {
    height,
    width,
    x,
    y,
    type: powerbi.visuals.SubSelectionOutlineType.Rectangle,
};

const regionOutline: powerbi.visuals.SubSelectionRegionOutline = {
    id: 'myObject',
    visibility:
powerbi.visuals.SubSelectionOutlineVisibility.Hover,
    outline
};
this.subSelectionRegionOutlines[regionOutline.id] = regionOutline;
this.renderOutlines();
// you need to remove the hovered outline when the element is not
hovered anymore
}
private renderOutlines(): void {
    const regionOutlines = getValues(this.subSelectionRegionOutlines);
    this.subSelectionService.updateRegionOutlines(regionOutlines);
}
```

## Related content

- [On object utils](#)
- [On-object formatting API](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# The Visual Filters API in Power BI visuals

Article • 06/06/2024

The Visual Filters API allows you to filter data in Power BI visuals. The main difference between the filter API and other ways of selecting data is how it affects other visuals in the report. When a filter is applied to a visual, only the filtered data will be visible in all visuals, despite highlight support by other visuals.

To enable filtering for the visual, the `capabilities.json` file should contain a `filter` object in the `general` section.

JSON

```
"objects": {  
    "general": {  
        "displayName": "General",  
        "displayNameKey": "formattingGeneral",  
        "properties": {  
            "filter": {  
                "type": {  
                    "filter": true  
                }  
            }  
        }  
    }  
}
```

## ⓘ Note

- Visual filters API interfaces are available in the [powerbi-models](#) package. This package also contains classes to create filter instances.

Windows Command Prompt

```
npm install powerbi-models --save
```

- If you're using an older (earlier than 3.x.x) version of the tools, include `powerbi-models` in the `visuals` package. For more information, see the short guide, [Add the Advanced Filter API to the custom visual](#). To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.

All filters use the [IFilter interface](#), as shown in the following code:

TypeScript

```
export interface IFilter {
    $schema: string;
    target: IFilterTarget;
}
```

Where `target` is a table column in the data source.

There are three filter APIs:

- [Basic filter API](#)
- [Advanced filter API](#)
- [Tuple \(multi-column\) filter API](#)

## The Basic Filter API

Basic filter interface is shown in the following code:

TypeScript

```
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

The following example returns all rows where `col1` equals the value 1, 2, or 3.

TypeScript

```
let basicFilter = {
    target: {
        column: "Col1"
    },
    operator: "In",
    values: [1,2,3]
}
```

The SQL equivalent of the above example is:

SQL

```
SELECT * FROM table WHERE col1 IN ( 1 , 2 , 3 )
```

To create a filter, you can use the `BasicFilter` class in `powerbi-models`.

If you are using 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:

JavaScript

```
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 calling the `applyJsonFilter()` method on the host interface, `IVisualHost`, which is provided to the visual in the constructor method.

TypeScript

```
IVisualHost.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).

This filter was introduced in the Visuals API version 1.7.0.

As opposed to the *Basic API*, in the *Advanced Filter API*:

- The `target` requires both a `table` and `column` name (the *Basic API* just had `column`).
- Operators are *And* and *Or* (as opposed to *In*).
- The filter uses conditions (*less than*, *greater than* etc.) instead of values with the interface:

TypeScript

```
interface IAdvancedFilterCondition {
    value: (string | number | boolean);
    operator: AdvancedFilterConditionOperators;
}
```

Condition operators for the `operator` parameter are: *None*, *LessThan*, *LessThanOrEqual*, *Greater Than*, *GreaterThanOrEqual*, *Contains*, *DoesNotContain*, *StartsWith*, *DoesNotStartWith*, *Is*,  *IsNot*, *IsBlank*, and "IsNotBlank"

JavaScript

```
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:

SQL

```
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:

TypeScript

```
interface ITupleFilter extends IFilter {  
    $schema: string;  
    filterType: FilterType;  
    operator: TupleFilterOperators;  
    target: ITupleFilterTarget;  
    values: TupleValueType[];  
}
```

Where

- `target` is an array of columns with table names:

TypeScript

```
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 *In* operator.
- `values` is an array of value tuples. Each tuple represents one permitted combination of the target column values.

TypeScript

```
declare type TupleValueType = ITupleElementValue[];  
  
interface ITupleElementValue {
```

```
    value: PrimitiveValueType  
}
```

Complete example:

TypeScript

```
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 important.

The SQL equivalent of the above code is:

SQL

```
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 2.2.0, you can restore the JSON filter from *VisualUpdateOptions*, as shown in the following code:

TypeScript

```
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

To reset or clear the filter, pass a `null` value to the filter API.

TypeScript

```
// invoke the filter
visualHost.applyJsonFilter(null, "general", "filter", FilterAction.merge);
```

## Related content

[Use Power BI visuals selections to add interactivity to a visual](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Identity filter API

Article • 06/17/2024

The **Identity filter API** allows you to create a visual that can filter categorical data using a semantic query.

It filters the data by **data points** rather than mathematical expressions.

The API keeps track of user selections and which data points to display. The data points are saved in an array and referenced by their position in the array.

This API is useful in the following scenarios:

- For custom visuals that use semantic models with group on keys
- Migrating visuals that used an older API (earlier than 2.2) to a newer API
- Allow selections using identifying index arrays

## ⓘ Note

The **Identity filter API** is available from API version 5.1 To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.

The Identity filter model is based on the [IIdentityFilter](#) interface.

### TypeScript

```
private filter: IIdentityFilter = {
    $schema: "",
    filterType: FilterType.Identity,
    operator: "In",
    target: []
}
```

For example, if the visual received the following data view update:

```

▼ {viewport: {...}, dataViews: Array(1), viewMode: 1, editMode: 0, isInFocus: false, ...} ⓘ
  ▼ dataViews: Array(1)
    ▼ 0:
      ▼ categorical:
        ▼ categories: Array(2)
          ▼ 0:
            ▼ identity: Array(280)
              ▼ [0 ... 99]
                ▶ 0: {identityIndex: 0}
                ▶ 1: {identityIndex: 1}
                ▶ 2: {identityIndex: 2}
                ▶ 3: {identityIndex: 3}
                ▶ 4: {identityIndex: 4}
                ▶ 5: {identityIndex: 5}
                ▶ 6: {identityIndex: 6}
                ▶ 7: {identityIndex: 7}

```

The array is of type number[] and it contains the identity fields of the items that the user selected.

The identityIndex corresponds to the index of the value in the semantic model's value array (see the following example).

```

▼ dataViews: Array [ {...} ]
  ▼ 0: Object { tree: null, single: null, matrix: null, ... }
    ▼ categorical: Object { categories: (1) [...], values: (1) [...] }
      ▼ categories: Array [ {...} ]
        ▼ 0: Object { source: {...}, values: (280) [...], identity: (280) [...] , ... }
          ▶ identity: Array(280) [ ..., ..., ..., ... ]
          ▶ identityFields: Array [ {...} ]
          ▶ source: Object { displayName: "Name", queryName: "Table1.Name", index: 0, ... }
          ▶ values: Array(280) [ "Aaliyah", "Aaliyah", "Aaliyah", ... ]
          ▶ <prototype>: Object { ... }
        length: 1

```

In the above example: {identityIndex: 0} = "Aaliyah" {identityIndex: 1} = "Aaliyah" {identityIndex: 02 = "Aaliyah" etc.

## How to use the Identity filter API

To use the Identity filter API, your powerbi-models version needs to be 1.9.1 or higher.

- Add the following property as a member of the visual.ts class:

TypeScript

```

private filter: IIIdentityFilter = {
  $schema: "",
  filterType: FilterType.Identity,
  operator: "In",
  target: []
}

```

- To handle Power BI updates, read the *target* array from the 'jsonFilters' in the 'VisualUpdateOptions' and translate it to the corresponding values. These values are the ones that were selected. In the previous example, a target array of [0,10] corresponds to the values of *Aliyah* and *Abigail*.
- To handle user selections in the previous example, click on the first *Abigail* to add the value 8 to the filter target array and send it using the following command:

TypeScript

```
this.visualHost.applyJsonFilter(this.filter, "general", "filter",
powerbi.FilterAction.merge);
```

## Migrating visuals with old API

Starting from API 5.1.0, to support the identity filter on visuals that were created using a version earlier than 2.2, add the following lines to your *capabilities.json* file:

JSON

```
"migration": {
  "filter": {
    "shouldUseIdentityFilter": true
  }
}
```

These lines convert the selections to identity filters.

### ⓘ Note

This step is only necessary for existing visuals created with older APIs. Newer visuals don't need to add this.

## Related content

[Use Power BI visuals selections to add interactivity to a visual](#)

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

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# The hierarchical identity filters API in Power BI visuals

Article • 06/17/2024

The **Hierarchy Identity filter API** enables visuals that use [Matrix DataView Mapping](#) to filter data on multiple fields at a time based on data points that use a [hierarchy structure](#).

This API is useful in the following scenarios:

- Filtering hierarchies based on data points
- Custom visuals that use semantic models with group on keys

## ⓘ Note

The Hierarchy Identity filter API is available from API version **5.9.0**

The filter interface is shown in the following code:

TypeScript

```
interface IHierarchyIdentityFilter<IdentityType> extends IFilter {
    target: IHierarchyIdentityFilterTarget;
    hierarchyData: IHierarchyIdentityFilterNode<IdentityType>[];
}
```

- *\$schema*: `https://powerbi.com/product/schema#hierarchyIdentity` (inherited from `IFilter`)
- *filterType*: `FilterType.HierarchyIdentity` (inherited from `IFilter`)
- *target*: Array of relevant columns in the query. Currently only a single role is supported; therefore, the target isn't required and should be empty.
- *hierarchyData*: the selected and unselected items in a hierarchy tree where each `IHierarchyIdentityFilterNode<IdentityType>` represents a single value selection.

TypeScript

```
type IHierarchyIdentityFilterTarget = IQueryNameTarget[]

interface IQueryNameTarget {
```

```
    queryName: string;  
}
```

- *queryName*: query name of the source column in the query. It comes from the `DataViewMetadataColumn`

TypeScript

```
interface IHierarchyIdentityFilterNode<IdentityType> {  
    identity: IdentityType;  
    children?: IHierarchyIdentityFilterNode<IdentityType>[];  
    operator: HierarchyFilterNodeOperators;  
}
```

- *identity*: The Node identity in `DataView`. The `IdentityType` Should be `CustomVisualOpaqueIdentity`
- *children*: List of node children relevant to the current selection
- *operator*: The operator for single objects in the tree. The operator can be one of the following three options:

TypeScript

```
type HierarchyFilterNodeOperators = "Selected" | "NotSelected" |  
"Inherited";
```

- *Selected*: value is explicitly selected.
- *NotSelected*: value is explicitly not selected.
- *Inherited*: value selection is according to the parent value in the hierarchy, or default if it's the root value.

Keep the following rules in mind when defining your hierarchy identity filter:

- Take the identities from the `DataView`.
- Each *identity* path should be a valid path in the `DataView`.
- Every leaf should have an operator of *Selected* or *NotSelected*.
- To compare identities, use the `ICustomVisualsOpaqueUtils.compareCustomVisualOpaqueIdentities` function.
- The identities might change following fields changes (for example, adding or removing fields). Power BI assigns the updated identities to the existing `filter.hierarchyData`.

# How to use the Hierarchy identity filter API

The following code is an example of how to use the hierarchy identity filter API in a custom visual:

TypeScript

```
import { IHierarchyIdentityFilterTarget, IHierarchyIdentityFilterNode,
HierarchyIdentityFilter } from "powerbi-models"

const target: IHierarchyIdentityFilterTarget = [];

const hierarchyData:
IHierarchyIdentityFilterNode<CustomVisualOpaqueIdentity>[] = [
{
    identity: {...},
    operator: "Selected",
    children: [
        {
            identity: {...},
            operator: "NotSelected"
        }
    ]
},
{
    identity: {...},
    operator: "Inherited",
    children: [
        {
            identity: {...},
            operator: "Selected"
        }
    ]
}
];
};

const filter = new HierarchyIdentityFilter(target, hierarchyData).toJSON();
```

To apply the filter, use the `applyJsonFilter` API call:

TypeScript

```
this.host.applyJsonFilter(filter, "general", "filter", action);
```

To restore the active JSON filter, use the `jsonFilters` property found in the "VisualUpdateOptions":

TypeScript

```
export interface VisualUpdateOptions extends extensibility.VisualUpdateOptions {
    //...
    jsonFilters?: IFilter[];
}
```

## Hierarchy related fields validation (optional)

The `HierarchyIdentity` filter is supported only for hierarchically related fields. By default, Power BI doesn't validate if the fields are hierarchically related.

To activate hierarchically related validation, add the 'areHierarchicallyRelated' property to the relevant role condition in the capabilities.json file:

TypeScript

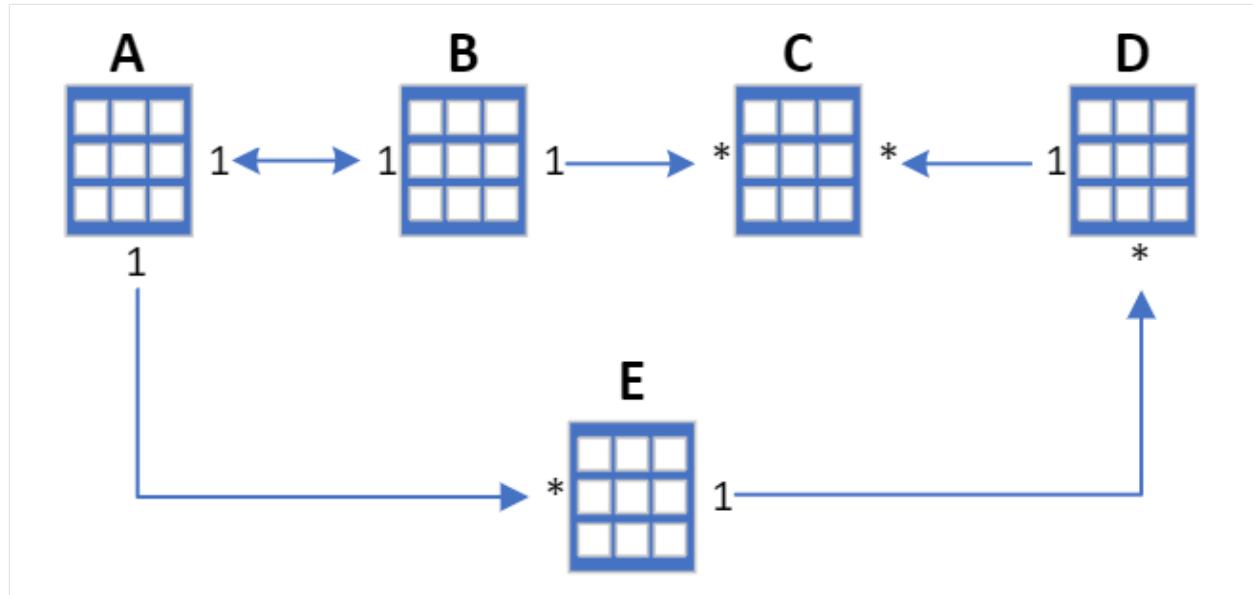
```
"dataViewMappings": [
    {
        "conditions": [
            {
                "Rows": {
                    "min": 1,
                    "areHierarchicallyRelated": true <----- NEW ----->
                },
                "Value": {
                    "min": 0
                }
            }
        ],
        ...
    }
]
```

Fields are hierarchically related if the following conditions are met:

- No included relationship edge is many to many cardinality, nor `ConceptualNavigationBehavior.Weak`.
- All fields in the filter exist in the path.
- Every relationship in the path has the same direction or bidirectional.
- The relationship direction matches the cardinality for one to many or bidirectional.

## Example of hierarchy relationships

For example, given the following entity relationship:



- A, B are hierarchically related: true
- B, C are hierarchically related: true
- A, B, C are hierarchically related: true
- A, C, E are hierarchically related: true (A --> E --> C)
- A, B, E are hierarchically related: true (B --> A --> E)
- A, B, C, E are hierarchically related: true (B --> A --> E --> C)
- A, B, C, D are hierarchically related: false (violated rule #3)
- C, D are hierarchically related: true
- B, C, D are hierarchically related: false (violated rule #3)
- A, C, D, E are hierarchically related: false (violated rule #3)

#### ⓘ Note

- When these validations are enabled, and the fields are not hierarchically related, the visual won't render, and an error message will be displayed:



You are using fields that don't have a supported set of relationships. [See details](#)

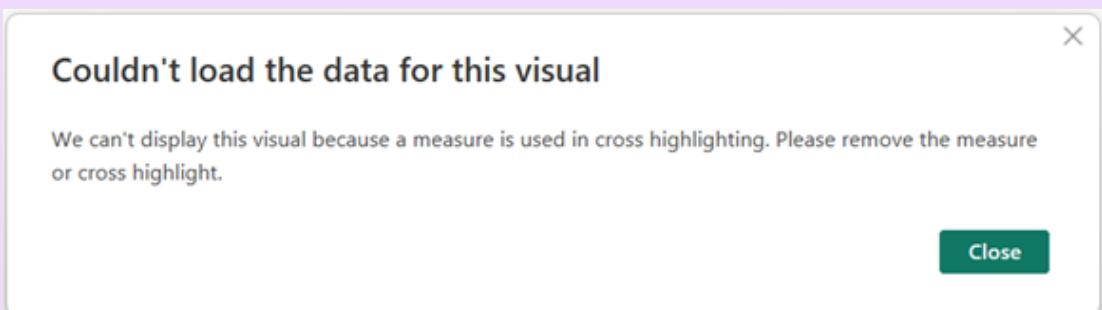
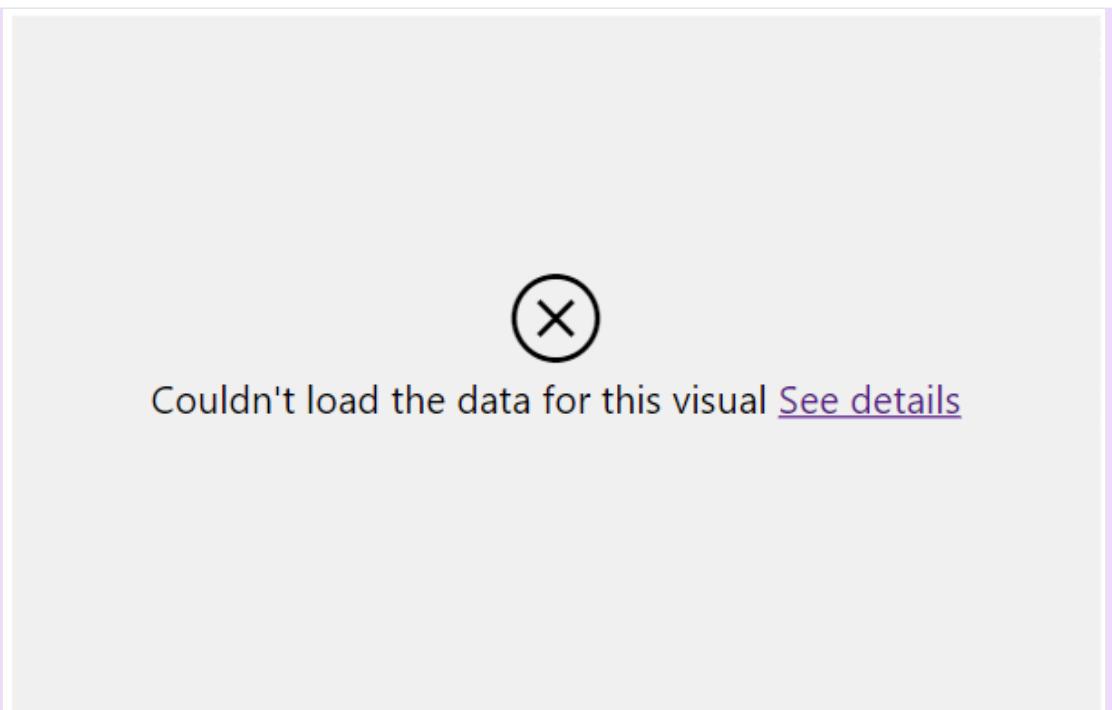


## Can't display this visual.

You are using fields that don't have a supported set of relationships.

[Close](#)

- When these validations are disabled, and the filter visual applies a filter that contains nodes related to non-hierarchically related fields, other visuals might not render properly when measures are in use:



## Code example for updating the hierarchy data tree after new selection

The following code shows how to update the `hierarchyData` tree after new a selection:

TypeScript

```
type CompareIdentitiesFunc = (id1: CustomVisualOpaqueIdentity, id2: CustomVisualOpaqueIdentity) => boolean;
/**
 * Updates the filter tree following a new node selection.
 * Prunes irrelevant branches after node insertion/removal if necessary.
 * @param path Identities path to the selected node.
 * @param treeNodes Array of IHierarchyIdentityTreeNode representing a valid filter tree.
 * @param compareIdentities Compare function for CustomVisualOpaqueIdentity to determine equality. Pass the ICustomVisualsOpaqueUtils.compareCustomVisualOpaqueIdentities function.
 * @returns A valid filter tree after the update
 */
```

```

function updateFilterTreeOnNodeSelection(
    path: CustomVisualOpaqueIdentity[],
    treeNodes: IHierarchyIdentityFilterNode<CustomVisualOpaqueIdentity>[],
    compareIdentities: CompareIdentitiesFunc
): IHierarchyIdentityFilterNode<CustomVisualOpaqueIdentity>[] {
    if (!path) return treeNodes;
    const root: IHierarchyIdentityFilterNode<CustomVisualOpaqueIdentity> = {
        identity: null,
        children: treeNodes || [],
        operator: 'Inherited',
    };
    let currentNodesLevel = root.children;
    let isClosestSelectedParentSelected = root.operator === 'Selected';
    let parents: { node:
        IHierarchyIdentityFilterNode<CustomVisualOpaqueIdentity>, index: number }[]
    = [{ node: root, index: -1 }];
    let shouldFixTree = false;
    path.forEach((identity, level) => {
        const index = currentNodesLevel.findIndex((node) =>
            compareIdentities(node.identity, identity));
        const isLastNodeInPath = level === path.length - 1
        if (index === -1) {
            const newNode:
                IHierarchyIdentityFilterNode<CustomVisualOpaqueIdentity> = {
                    identity,
                    children: [],
                    operator: isLastNodeInPath ? (isClosestSelectedParentSelected
? 'NotSelected' : 'Selected') : 'Inherited',
                };
            currentNodesLevel.push(newNode);
            currentNodesLevel = newNode.children;
            if (newNode.operator !== 'Inherited') {
                isClosestSelectedParentSelected = newNode.operator ===
'Selected';
            }
        } else {
            const currentNode = currentNodesLevel[index];
            if (isLastNodeInPath) {
                const partial = currentNode.children &&
currentNode.children.length;
                if (partial) {
                    /**
                     * The selected node has subtree.
                     * Therefore, selecting this node should lead to one of
                     the following scenarios:
                     * 1. The node should have Selected operator and its
                     subtree should be pruned.
                     * 2. The node and its subtree should be pruned from the
                     tree and the tree should be fixed.
                    */
                    // The subtree should be always pruned.
                    currentNode.children = [];
                    if (currentNode.operator === 'NotSelected' ||
(currentNode.operator === 'Inherited' && isClosestSelectedParentSelected ))
{

```

```

        /**
         * 1. The selected node has NotSelected operator.
         * 2. The selected node has Inherited operator, and
         its parent has Slected operator.
         * In both cases the node should be pruned from the
         tree and the tree shoud be fixed.
        */
        currentNode.operator = 'Inherited'; // to ensure it
will be pruned
        parents.push({ node: currentNode, index });
        shouldFixTree = true;
    } else {
        /**
         * 1. The selected node has Selected operator.
         * 2. The selected node has Inherited operator, but its
         parent doesn't have Selected operator.
         * In both cases the node should stay with Selected
         operator pruned from the tree and the tree should be fixed.
         * Note that, node with Selected oprator and parent
         with Selector operator is not valid state.
        */
        currentNode.operator = 'Selected';
    }
} else {
    // Leaf node. The node should be pruned from the tree and
    the tree should be fixed.
    currentNode.operator = 'Inherited'; // to ensure it will
be pruned
    parents.push({ node: currentNode, index });
    shouldFixTree = true;
}
} else {
    // If it's not the last noded in path we just continue
traversing the tree
    currentNode.children = currentNode.children || [];
    currentNodesLevel = currentNode.children
    if (currentNode.operator !== 'Inherited') {
        isClosestSelectedParentSelected = currentNode.operator
        === 'Selected';
        // We only care about the closet parent with
        Selected/NotSelected operator and its children
        parents = [];
    }
    parents.push({ node: currentNode, index });
}
}
});

// Prune brnaches with Inherited leaf
if (shouldFixTree) {
    for (let i = parents.length - 1; i >= 1; i--) {
        // Normalize to empty array
        parents[i].node.children = parents[i].node.children || [];
        if (!parents[i].node.children.length && (parents[i].node.operator
        === 'Inherited')) {
            // Remove the node from its parent children array
        }
    }
}

```

```

        removeElement(parents[i - 1].node.children, parents[i].index);
    } else {
        // Node has children or Selected/NotSelected operator
        break;
    }
}
return root.children;
}
/** 
 * Removes an element from the array without preserving order.
 * @param arr - The array from which to remove the element.
 * @param index - The index of the element to be removed.
 */
function removeElement(arr: any[], index: number): void {
    if (!arr || !arr.length || index < 0 || index >= arr.length) return;
    arr[index] = arr[arr.length - 1];
    arr.pop();
}

```

## Considerations and limitations

- This filter is supported only for matrix dataView mapping.
- The visual should contain only one *grouping* data role.
- A visual that uses the Hierarchy identity filter type should apply only a single filter of this type.

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Detect filter state

Article • 06/06/2024

## ⓘ Note

This feature is available from API version 5.4

The `isDataFilterApplied` boolean parameter in the `DataViewMetadata` object notes if the rendered visual or report has any filters applied to it. The developer can then adjust the display accordingly (for example, by adding or removing text depending on if there's a filter). This feature applies whether the filter is applied to the entire report, page, or specific visual.

To use the `isDataFilterApplied` parameter:

TypeScript

```
public update(options: VisualUpdateOptions) {
    const dataView = options?.dataViews[0];
    if (dataView && dataView?.metadata?.isDataFilterApplied) {
        ...
    }
}
```

## Related content

[Use Power BI visuals selections to add interactivity to a visual](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# "Rendering" events in Power BI visuals

Article • 04/30/2024

In order to get a [visual certified](#), it must include **rendering events**. These events let listeners (primarily, *export to PDF* and *export to PowerPoint*) know when the visual is being rendered and when it's ready for export.

## ⓘ Important

Any visual that exports data (for example, to a PowerPoint or *.pdf* file) must contain rendering events to ensure that the export doesn't begin before the visual finished rendering.

The **rendering events API** consists of three methods that should be called during rendering:

- `renderingStarted`: The Power BI visual code calls the `renderingStarted` method to indicate that the rendering process started. **This method should always be the first line of the *update* method** since that is where the rendering process begins.
- `renderingFinished`: When rendering is completed successfully, the Power BI visual code calls the `renderingFinished` method to notify the listeners that the visual's image is ready for export. This method should be the **last line of code executed** when the visual updates. It's usually, but not always, the last line of the update method.
- `renderingFailed`: If a problem occurs during the rendering process, the Power BI visual doesn't render successfully. To notify the listeners that the rendering process wasn't 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.

## ! Note

*Rendering events* are a requirement for visuals certification. Without them your visual won't be approved by the Partner Center for publication. For more information, see [certification requirements](#).

## How to use the rendering events API

To call the rendering methods, you have to first import them from the `IVisualEventService`.

1. In your `visual.ts` file, include the line:

```
TypeScript
```

```
import IVisualEventService = powerbi.extensibility.IVisualEventService;
```

2. In the `IVisual` class include the line:

```
TypeScript
```

```
private events: IVisualEventService;
```

3. In the `constructor` method of the `IVisual` class

```
TypeScript
```

```
this.events = options.host.eventService;
```

You can now call the methods `this.events.renderingStarted(options)`, `this.events.renderingFinished(options)`, and `this.events.renderingFailed(options)`, where appropriate in your *update* method.

## Example 1: Visual without animations

Here's an example of a simple visual that uses the *render events* API.

```
TypeScript
```

```
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);
    }
```

## Example 2: Visual with animations

If the visual has animations or asynchronous functions for rendering, the `renderingFinished` method should be called after the animation or inside `async` function, even if it's not the last line of the `update` method.

TypeScript

```
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);
            });
    }
}
```

## Related content

[Visual API](#)

[Get a Power BI visual certified](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Drilldown API

Article • 01/19/2024

The **Drilldown API** allows you to create a visual that can trigger a drilldown operation on its own, without user interaction.

The API enables the visual to show next level, expand to next level, or drill up based on the parameters passed to the API. For more information about drilling down, see [Drill down support](#).

## How to use the drilldown API

### Note

The **Drilldown API** is available from API version 4.7.0 To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.

Add the following to the `capabilities.json` file:

JSON

```
"drilldown": {  
    "roles": ["Rows", "Columns"]  
}
```

## Example: Drilldown API

The following example shows how the visual call a drilldown operation.

TypeScript

```
public update(options: VisualUpdateOptions) {  
    if  
    ((options.dataViews[0].metadata.dataRoles.drillableRoles['Columns']).indexOf  
    (powerbi.DrillType.Down) >= 0) {  
        let args: powerbi.DrillDownArgs = {  
            roleName: "Columns",  
            drillType: powerbi.DrillType.Down  
        };  
        this.host.drill(args);  
    }  
}
```

---

# Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Dynamic drill control

Article • 06/06/2024

## ⓘ Note

This feature is available from API version 5.7.0.

The dynamic drill control feature allows the visual to enable or disable the [drill feature](#) dynamically using an [API call](#). When the drill feature is enabled, all drilldown functionalities and [expand/collapse features](#) are available, including API calls, context menu commands, header drill buttons, and support for hierarchy data. When disabled, these functionalities aren't available.

The following images show an example of a visual with the dynamic drill control feature enabled and disabled:

Drill enabled

Sum of pop by continent	
<input type="button" value="enableDrillState"/>	<input type="button" value="disableDrillState"/>
Totals	6251013179
<input type="checkbox"/> Asia	3811953827
<input type="checkbox"/> Africa	929539692
<input type="checkbox"/> Americas	898871184
<input type="checkbox"/> Europe	586098529
<input checked="" type="checkbox"/> Oceania	
Australia	20434176
New Zealand	4115771

The dynamic drill control feature includes the following API elements:

- The `isDrillDisabled` flag in the `DataRolesInfo`:

TypeScript

```
export interface DataRolesInfo {  
    //...  
    isDrillDisabled?: boolean; // ----- NEW -----  
}
```

- The `setCanDrill` method in the `IVisualHost` interface:

TypeScript

```
export interface IVisualHost extends extensibility.IVisualHost {  
    //...  
    setCanDrill: (drillAllowed: boolean) => void; // ----- NEW -----  
}
```

To identify whether the drill is disabled, use the `isDrillDisabled` property in the update method:

TypeScript

```
private update(options: VisualUpdateOptions) {  
    //...  
    const isDrillDisabled =  
options.dataViews[0].metadata.dataRoles.isDrillDisabled;  
    //...  
}
```

Then use the API call to enable or disable the drill as needed:

- To enable: `this.host.setCanDrill(true /* drillAllowed */);`
- To disable: `this.host.setCanDrill(false /* drillAllowed */);`

## Dynamic drill control requirements

Drilling is enabled by default, but the dynamic drill control feature allows the visual to enable or disable drilling using an API call.

A visual with the dynamic drill control feature, has the following code in the `capabilities.json` file:

- With drill disabled by default:

JSON

```
"drilldown": {  
    "roles": [  
        "Rows",  
        "Columns"  
    ],  
    "canDisableDrill": {  
        "disabledByDefault": true  
    }  
},
```

- With drill enabled by default:

JSON

```
"drilldown": {  
    "roles": [  
        "Rows",  
        "Columns"  
    ],  
    "canDisableDrill": {}  
},
```

The `canDisableDrill` property indicates that the visual supports this feature. Without this property, the API call isn't respected.

The `disabledByDefault` property indicates whether or not to disable the drill feature by default.

#### ① Note

The `disabledByDefault` property takes effect when you do one of the following actions:

- Add a new visual to the canvas
- Convert a visual from one that doesn't support this feature.

For example, if you convert a *sourceVisual* to *targetVisual*, the *targetVisual's* `disabledByDefault` property is considered only if the *sourceVisual* doesn't support this feature. If *sourceVisual* does support this feature, the *targetVisual* keeps the *sourceVisual's* state and not the default.

## Adding drill-down support to a new version of an existing visual

Using the drilldown feature represents a breaking change. Therefore, for the smoothest transition, we recommend that you use a new visual GUID for the new version.

If, however, you want to keep the same GUID, keep in mind the following points:

- When you migrate from a nondrillable version to a new drillable version, some data might not be provided in the `dataView` due to the hierarchical data support introduced as part of the drill feature. The dynamic drill control feature doesn't offer automatic support for this issue but can be used to manage the migration process.
- For self-migration of the visual, the visual should take the following actions:
  - Identify the first time the new version is loaded instead of the older version, and apply the `persistProperties` API.
  - Disable the drill to receive all the data, using the `setCanDrill` API.

The following example shows how to self-migrate an older visual to one that uses dynamic drill control:

1. Add the following object to the `capabilities.json` file:

JSON

```
"DrillMigration": {  
    "displayName": "Drill Migration",  
    "properties": {  
        "isMigrated": {  
            "displayName": "Is Drill Migrated",  
            "type": {  
                "bool": true  
            }  
        }  
    },  
},
```

2. Add the following to the `visual.ts` file:

TypeScript

```
export class Visual implements IVisual {  
    //...  
    private isCalledToDisableDrillInMigrationScenario = false;  
    private drillMigration = { disabledByDefault: true };  
    constructor(options: VisualConstructorOptions) {  
        //...  
        this.host = options.host;
```

```

//...
}

private update(options: VisualUpdateOptions) {
    this.handleSelfDrillMigration(options);
    //...
}

private handleSelfDrillMigration(options: VisualUpdateOptions):
void {
    if (options && options.dataViews && options.dataViews[0] &&
options.dataViews[0].metadata) {
        const metadata = options.dataViews[0].metadata;
        if (metadata && metadata.dataRoles) {
            const isDrillDisabled =
metadata.dataRoles.isDrillDisabled;
            if (isDrillDisabled === undefined) {
                return;
            }
            // Continue in case the visual is already migrated
            if (!metadata.objects?.DrillMigration?.isMigrated) {
                // Persist the isMigrated property when the drill
has the correct state
                if (this.drillMigration.disabledByDefault ===
isDrillDisabled) {
                    this.persistMigrationProperty();
                } else if
(!this.isCalledToDisableDrillInMigrationScenario) {
                    // Use the API call only once
                    this.host.setCanDrill(!this.drillMigration.disabledByDefault);
                    this.isCalledToDisableDrillInMigrationScenario = true;
                }
            }
        }
    }
    private persistMigrationProperty(): void {
        let property = {
            merge: [
                {
                    objectName: "DrillMigration",
                    properties: {
                        isMigrated: true
                    },
                    selector: null
                }
            ]
        };
        this.host.persistProperties(property);
    }
}

```

The first time the visual is opened after adding this code, the DrillMigration variable is set to true and the visual opens in the default state.

# Considerations and limitations

- The drill state isn't saved after disabling the drill. If you reenable the drill after disabling it, only the first level is displayed regardless of what was displayed before it was disabled.
- The expand/collapse state isn't saved after disabling the drill. All the rows are collapsed once the drill is reenabled.
- The API call isn't supported for dashboards.
- Data view mapping conditions: Use `"max": 1` for all conditions for the drillable role to limit the visual to showing only one field when drill is disabled. For example:
  - For categorical data view:

JSON

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

- For matrix data view:

JSON

```
"conditions": [
    { "Rows": { "max": 0 }, "Columns": { "max": 0 }, "Value": {
        "min": 1 } },
    { "Rows": { "min": 1 }, "Columns": { "min": 0 }, "Value": {
        "min": 0 } },
    { "Rows": { "min": 0 }, "Columns": { "min": 1 }, "Value": {
        "min": 0 } },
]
```

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# File download API

Article • 04/22/2024

The **file download API** lets users download data from a custom visual into a file on their storage device. Downloading a visual requires user consent and admin permission provided in the admin global switch. This setting is separate from and not affected by download restrictions applied in your organization's [export and sharing](#) tenant settings.

## Allow downloads from custom visuals

Enabling this setting will let custom visuals download any information available to the visual (such as summarized data and visual configuration) upon user consent. It is not affected by download restrictions applied in your organization's Export and sharing settings. [Learn more](#)



! If the report or its underlying dataset has an applied sensitivity label, the label and its protection settings (such as encryption) won't be applied to the exported .csv file. [Learn more](#)

Apply to:

- The entire organization  
 Specific security groups  
 Except specific security groups

**Apply**

**Cancel**

## Note

The **file download API** has three methods:

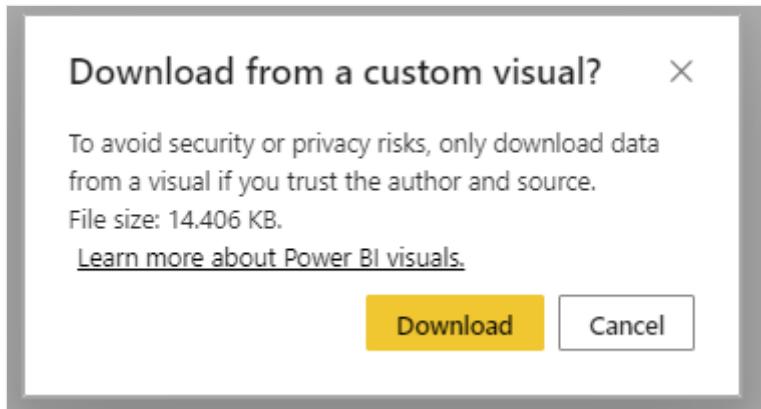
- [exportVisualsContent](#) is available from API version 4.5
- [status](#) is available from API version 4.6.
- [exportVisualsContentExtended](#) is available from API version 5.3.
- To find out which version you're using, check the `apiVersion` in the `pbviz.json` file.

Use the **file download API** to export to files of the following types:

- .txt

- .CSV
- .json
- .tmplt
- .xml
- .pdf
- .xlsx

Before the download begins, a window appears asking to confirm that the visual is from a trusted source.



## How to use the file download API

To use the file download API, add a declaration to the [privileges array in visual capabilities](#).

The **file download API** has three methods:

- [status](#): available from API version 4.6
- [exportVisualsContent](#): available from API version 4.5
- [exportVisualsContentExtended](#): available from API version 5.3.

The difference between the two methods is the return value.

### The `status` method

The status method returns the status of the file download API:

- *PrivilegeStatus.DisabledByAdmin*: the tenant admin switch is off
- *PrivilegeStatus.NotDeclared*: the visual has no declaration for the local storage in the privileges array
- *PrivilegeStatus.NotSupported*: the API isn't supported. See [limitations](#) for more information.
- *PrivilegeStatus.Allowed*: the API is supported and allowed.

## The `exportVisualsContent` method

The `exportVisualsContent` method has four parameters:

- `content`: string
- `filename`: string
- `fileType`: string - When exporting to a `.pdf` or `.xlsx` file, the `fileType` parameter should be `base64`
- `fileDescription`: string

This method returns a promise that will be resolved for a Boolean value.

## The `exportVisualsContentExtended` method

The `exportVisualsContentExtended` method also has four parameters:

- `content`: string
- `filename`: string
- `fileType`: string - When exporting to a `.pdf` or `.xlsx` file, the `fileType` parameter should be `base64`
- `fileDescription`: string

This method returns a promise, which will be resolved with a result of type

`ExportContentResultInfo` that contains the following parameters:

- `downloadCompleted` – if the download completed successfully.
- `filename` – the exported file name.

## Example: file download API

Here's an example of how to download the content of a custom visual into an excel file and a text file.

TypeScript

```
import IDownloadService = powerbi.extensibility.IDownloadService;  
...  
  
export class Visual implements IVisual {  
    ...  
    private downloadService: IDownloadService;  
    ...  
  
    constructor(options: VisualConstructorOptions) {  
        this.downloadService = options.host.downloadService;
```

```

    ...

    const downloadBtn: HTMLElement = document.createElement("button");
    downloadBtn.onclick = () => {
        let contentXlsx: string = ...;//content in base64
        let contentTxt: string = ...;
        this.downloadService.exportVisualsContent(contentTxt,
"mytxt.txt", "txt", "txt file").then((result) => {
            if (result) {
                //do something
            }
        }).catch(() => {
            //handle error
        });
    };

    this.downloadService.exportVisualsContent(contentXlsx,
"myfile.xlsx", "base64", "xlsx file").then((result) => {
        if (result) {
            //do something
        }
    }).catch(() => {
        //handle error
    });
}

this.downloadService.exportVisualsContentExtended(contentXlsx,
"myfile.xlsx", "base64", "xlsx file").then((result) => {
    if (result.downloadCompleted) {
        //do something
        console.log(result.fileName);
    }
}).catch(() => {
    //handle error
});
};

// if you are using API version > 4.6.0
downloadBtn.onclick = async () => {
    try {
        const status: powerbi.PrivilegeStatus = await
this.downloadService.exportStatus();
        if (status === powerbi.PrivilegeStatus.Allowed) {
            const result = await
this.downloadService.exportVisualsContent('aaaaa','a.txt', 'text/plain',
'aa');
            // handle result
        } else {
            // handle if the API is not allowed
        }
    } catch (err) {
        //handle error
    }
}
}
}

```

# Considerations and limitations

- The API is supported only in the Power BI service and Power BI desktop
- The size limit for a downloaded file is 30 MB.
- This API is a [privileged API](#).

## Related content

- [Learn about the Visual API](#)
- [Get a Power BI visual certified](#)

---

## Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Check permissions API

Article • 06/28/2024

As a developer of Power BI visuals, you can develop visuals that need permission to access various resources. You request these permissions in the privileges section of the [capabilities.json](#) file. These privileges include the ability to access:

- remote resources or web sites
- local storage for downloading data

Each organization's admin can allow or block these permissions. The *check permissions API* allows you to query the host at runtime to determine which permissions are granted. You can use this information to design a visual that will work with various permission settings.

The *check permissions API* returns the status of each permission query function:

TypeScript

```
/**  
 * Represents a return type for privilege status query methods  
 */  
export const enum PrivilegeStatus {  
    /**  
     * The privilege is allowed in the current environment  
     */  
    Allowed,  
  
    /**  
     * The privilege declaration is missing in visual capabilities section  
     */  
    NotDeclared,  
  
    /**  
     * The privilege is not supported in the current environment  
     */  
    NotSupported,  
  
    /**  
     * The privilege usage was denied by tenant administrator  
     */  
    DisabledByAdmin,  
}
```

## How to use the check permissions API

Every privilege API has its own query method to check for the permission status. The permission status can be one of the following:

- Allowed
- Not declared
- Not supported
- Disabled by Admin

## Web access

TypeScript

```
export interface IWebAccessService {  
    /**  
     * Returns the availability status of the service for specified url.  
     *  
     * @param url - the URL to check status for  
     * @returns the promise that resolves to privilege status of the service  
     */  
    webAccessStatus(url: string): IPromise<PrivilegeStatus>;  
}
```

## Export content

TypeScript

```
export interface IDownloadService {  
    /**  
     * Returns the availability status of the service.  
     *  
     * @returns the promise that resolves to privilege status of the service  
     */  
    exportStatus(): IPromise<PrivilegeStatus>;  
}
```

## Related content

[Power BI custom visual API](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# How to add external libraries

Article • 03/09/2025

This article describes how to use external libraries in Power BI visuals. This article also describes how to install, import, and call external libraries from a Power BI visual's code.

## JavaScript libraries

1. Install an external JavaScript library by using a package manager, such as *npm* or *yarn*.
2. Import the required modules into the source files using the external library.

### Note

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 section provides an example of installing the [D3 library](#) and the [@types/d3](#) package using [npm](#) in the code of a Power BI visual.

For a full example, see the [Power BI visualizations](#) code.

1. Install the *d3* package and the *@types/d3* package.

PowerShell

```
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*.

TypeScript

```
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 section provides an example of installing [bootstrap](#) using [npm](#).

For a full example, see the [Power BI visualizations](#) code.

1. Install the *bootstrap* package.

PowerShell

```
npm install bootstrap --save
```

2. Include the `import` statement in *visual.less*.

less

```
@import (less) "node_modules/bootstrap/dist/css/bootstrap.css";
```

## Related content

[Set up your environment for developing a Power BI visual](#)

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

# Power BI visuals interactivity utils

Article • 01/10/2024

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 latest 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.

Bash

```
npm install powerbi-visuals-utils-interactivityutils --save
```

2. If you're using version 3.0 or later of the tool, install `powerbi-models` to resolve dependencies.

Bash

```
npm install powerbi-models --save
```

3. To use interactivity utils, import the required component in the source code of the Power BI visual.

TypeScript

```
import { interactivitySelectionService } from "powerbi-visuals-utils-interactivityutils";
```

## Include 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.

```
less
```

```
@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 follows:

```
TypeScript
```

```
/** 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.

```
TypeScript
```

```
export class Visual implements IVisual {  
    // ...  
    private interactivity:  
        interactivityBaseService.IInteractivityService<VisualDataPoint>;
```

```
// ...
constructor(options: VisualConstructorOptions) {
    // ...
    this.interactivity =
interactivitySelectionService.createInteractivitySelectionService(this.
host);
    // ...
}
}
```

TypeScript

```
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 following sample.

## 3. Define the interface for the behavior class options.

TypeScript

```
import { SelectableDataPoint } from "./interactivitySelectionService";

import {
    IBehaviorOptions,
    BaseDataPoint
} from "./interactivity BaseService";

export interface BaseBehaviorOptions<SelectableDataPointType extends
BaseDataPoint> extends IBBehaviorOptions<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.

### Define a class for `visual behavior`

The class is responsible for `click` and `contextmenu` mouse events.

When a user clicks data elements, the visual calls the selection handler to select data points. If the user clicks the background element of the visual, it calls the clear selection handler.

The class has the following correspond methods:

- `bindClick`
- `bindClearCatcher`
- `bindContextMenu`.

TypeScript

```
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 {
    // ...
  }
}
```

### Extend the `BaseBehavior` class

### TypeScript

```
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 a click on elements, call the `d3` selection object `on` method. This also applies for `elementsSelection` and `clearCatcherSelection`.

### TypeScript

```
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.

### TypeScript

```
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(
                event,
                datum);
        }
    });
}
```

```

        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`

TypeScript

```

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. A sample implementation of `renderSelection` follows.

TypeScript

```

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.

```
TypeScript
```

```
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:

```
TypeScript
```

```
export class Visual implements IVisual {
  // ...
  constructor(options: VisualConstructorOptions) {
    // ...
    this.behavior = new Behavior();
  }
  // ...
}
```

## Related content

- [Visuals with selection](#)
- [Add a context menu to your Power BI visual](#)
- [Add interactivity into visual by Power BI visuals selection](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Formatting utils

Article • 10/10/2024

Formatting utils contain classes, interfaces, and methods to format values. It also contains extender methods to process strings and measure text size in an SVG/HTML document.

## Text measurement service

The module provides the following functions and interfaces:

### TextProperties interface

This interface describes common properties of the text.

TypeScript

```
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 specific SVG text properties.

TypeScript

```
function measureSvgTextWidth(textProperties: TextProperties, text?: string): number;
```

Example of using `measureSvgTextWidth`:

TypeScript

```
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.

TypeScript

```
function measureSvgTextRect(textProperties: TextProperties, text?: string):  
    SVGRect;
```

Example of using `measureSvgTextRect`:

TypeScript

```
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 specific SVG text properties.

TypeScript

```
function measureSvgTextHeight(textProperties: TextProperties, text?: string): number;
```

Example of using `measureSvgTextHeight`:

TypeScript

```
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 specific SVG text properties.

TypeScript

```
function estimateSvgTextBaselineDelta(textProperties: TextProperties): number;
```

Example:

TypeScript

```
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 specific SVG text properties.

TypeScript

```
function estimateSvgTextHeight(textProperties: TextProperties,  
tightFitForNumeric?: boolean): number;
```

Example of using `estimateSvgTextHeight`:

TypeScript

```
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
```

For an example, see [custom visual code ↗](#).

## measureSvgTextElementWidth

This function measures the width of the `svgElement`.

TypeScript

```
function measureSvgTextElementWidth(svgElement: SVGTextElement): number;
```

Example of using `measureSvgTextElementWidth`:

TypeScript

```

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.

TypeScript

```
function getMeasurementProperties(element: Element): TextProperties;
```

Example of using `getMeasurementProperties`:

TypeScript

```

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",
  "color": "#0072bc"
});

textMeasurementService.getMeasurementProperties(element);

```

```
        "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.

TypeScript

```
function getSvgMeasurementProperties(svgElement: SVGTextElement):
TextProperties;
```

Example of using `getSvgMeasurementProperties`:

TypeScript

```
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.

TypeScript

```
function getDivElementWidth(element: JQuery): string;
```

Example of using `getDivElementWidth`:

TypeScript

```
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 a label's text size to the available size, and renders ellipses when the available size is smaller.

TypeScript

```
function getTailoredTextOrDefault(textProperties: TextProperties, maxWidth:
number): string;
```

Example of using `getTailoredTextOrDefault`:

```
TypeScript

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.

```
TypeScript

function endsWith(str: string, suffix: string): boolean;
```

Example of using `endsWith`:

```
TypeScript

import { stringExtensions } from "powerbi-visuals-utils-formattingutils";
// ...

stringExtensions.endsWith("Power BI", "BI");

// returns: true
```

### equalIgnoreCase

This function compares strings, ignoring case.

TypeScript

```
function equalIgnoreCase(a: string, b: string): boolean;
```

Example of using `equalIgnoreCase`:

TypeScript

```
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.

TypeScript

```
function startsWith(a: string, b: string): boolean;
```

Example of using `startsWith`:

TypeScript

```
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.

TypeScript

```
function contains(source: string, substring: string): boolean;
```

Example of using `contains` method:

TypeScript

```
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.

TypeScript

```
function isNullOrEmpty(value: string): boolean;
```

Example of `isNullOrEmpty` method:

TypeScript

```
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.

TypeScript

```
interface IValueFormatter {
    format(value: any): string;
    displayUnit?: DisplayUnit;
    options?: ValueFormatterOptions;
}
```

## IValueFormatter.format

This method formats the specified value.

TypeScript

```
function format(value: any, format?: string, allowFormatBeautification?: boolean): string;
```

Examples for `IValueFormatter.format`:

### The thousand formats

TypeScript

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({ value: 1001 });

iValueFormatter.format(5678);

// returns: "5.68K"
```

### The million formats

TypeScript

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({ value: 1e6 });

iValueFormatter.format(1234567890);

// returns: "1234.57M"
```

### The billion formats

TypeScript

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({ value: 1e9 });

iValueFormatter.format(1234567891236);
```

```
// returns: 1234.57bn
```

## The trillion format

TypeScript

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({ value: 1e12 });

iValueFormatter.format(1234567891236);

// returns: 1.23T
```

## The exponent format

TypeScript

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({ format: "E" });

iValueFormatter.format(1234567891236);

// returns: 1.234568E+012
```

## The culture selector

TypeScript

```
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: 02/03/2007 17:42:42

let valueFormatterUSA = valueFormatter.create({ cultureSelector: "en-US" });

valueFormatterUSA.format(new Date(2007, 2, 3, 17, 42, 42));

// returns: 2/3/2007 5:42:42 PM
```

## The percentage format

TypeScript

```
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

TypeScript

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let date = new Date(2016, 10, 28, 15, 36, 0),
    iValueFormatter = valueFormatter.create({});

iValueFormatter.format(date);

// returns: 10/28/2016 3:36:00 PM
```

## The boolean format

TypeScript

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({});

iValueFormatter.format(true);

// returns: True
```

## The customized precision

TypeScript

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

let iValueFormatter = valueFormatter.create({ value: 0, precision: 3 });
```

```
iValueFormatter.format(3.141592653589793);  
// returns: 3.142
```

For an example, see [custom visual code ↗](#).

## ValueFormatterOptions

This interface describes `options` of the `IValueFormatter` and options of `create` function.

TypeScript

```
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 */  
    valueTypeDescriptor?: ValueTypeDescriptor;  
    /** Specifies the culture */  
    cultureSelector?: string;  
}
```

## create

This method creates an instance of `IValueFormatter`.

TypeScript

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";
import create = valueFormatter.create;

function create(options: ValueFormatterOptions): IValueFormatter;
```

## Example of using create

TypeScript

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

valueFormatter.create({});

// returns: an instance of IValueFormatter.
```

## format

Alternative way to format the value without creating `IValueFormatter`. Useful for cases with [dynamic formats string](#)

TypeScript

```
import { format } from "powerbi-visuals-utils-formattingutils";
import format = valueFormatter.format;

function format(value: any, format?: string, allowFormatBeautification?: boolean, cultureSelector?: string): string;
```

## Example of using format

TypeScript

```
import { valueFormatter } from "powerbi-visuals-utils-formattingutils";

const value = 12
const format = '¥ #,0'
valueFormatter.format(value, format);

// returns: formatted value as string (¥ 12)
```

## Related content

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# FormattingModel utils

Article • 10/23/2024

*Formatting model utils* contains the classes, interfaces, and methods needed to build a formatting settings model to populate the property panes (format and analytics panes) of your Power BI custom visual.

## Formatting settings service

The *formatting settings service* receives a formatting settings model, and turns it into a formatting model that populates the formatting pane. The formatting model service also supports string localizations.

Initializing formatting settings service:

TypeScript

```
import { FormattingSettingsService } from "powerbi-visuals-utils-formattingmodel";

export class MyVisual implements IVisual {
    // declaring formatting settings service
    private formattingSettingsService: FormattingSettingsService;

    constructor(options: VisualConstructorOptions) {

        this.formattingSettingsService = new FormattingSettingsService();

        // ...
    }
}
```

Formatting settings service interface **IFormattingSettingsService** has two main methods:

TypeScript

```
/**
 * Build visual formatting settings model from metadata dataView
 *
 * @param dataViews metadata dataView object
 * @returns visual formatting settings model
 */
populateFormattingSettingsModel<T extends Model>(typeClass: new () => T,
dataViews: powerbi.DataView[]): T;
```

```
/**  
 * Build formatting model by parsing formatting settings model object  
 *  
 * @returns powerbi visual formatting model  
 */  
buildFormattingModel(formattingSettingsModel: Model):  
visuals.FormattingModel;
```

## Formatting settings model

The *settings model* contains and wraps all formatting cards for the formatting pane and analytics pane.

TypeScript

```
export class Model {  
    cards: Array<Cards>;  
}
```

This example declares a new formatting settings model:

TypeScript

```
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";  
  
import FormattingSettingsCompositeCard = formattingSettings.CompositeCard;  
import FormattingSettingsCards = formattingSettings.Cards;  
import FormattingSettingsModel = formattingSettings.Model;  
  
export class VisualSettingsModel extends FormattingSettingsModel {  
    // Building my visual formatting settings card  
    myVisualCard: FormattingSettingsCompositeCard = new  
myVisualCardSettings();  
  
    // Add formatting settings card to cards list in model  
    cards: FormattingSettingsCards[] = [this.myVisualCard];  
}
```

## Formatting settings card

A *formatting settings card* specifies a formatting card in the formatting or analytics pane. A formatting settings card can contain multiple formatting slices, containers, groups, and properties.

Adding slices to a formatting settings card puts all of these slices into one formatting card.

Cards, Slices, and Groups can be hidden dynamically by setting the `visible` parameter to `false` (`true` by default).

The card can populate either the formatting pane or analytics pane by setting the `analyticsPane` parameter to `true` or `false`.

Example declaring formatting settings card, including one formatting settings group and slice:

- Card name should match the object name in `capabilities.json`
- Slice name should match the property name in `capabilities.json`

TypeScript

```
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

import FormattingSettingsCompositeCard = formattingSettings.CompositeCard;
import FormattingSettingsGroup = formattingSettings.Group;
import FormattingSettingsSlice = formattingSettings.Slice;

// Formatting settings group
class myVisualGroupSettings extends FormattingSettingsGroup {
    // Formatting settings slice
    myAnotherNumericSlice = new formattingSettings.NumUpDown({
        name: "myAnotherNumericSlice",
        displayName: "My Formatting Numeric Slice in group",
        value: 15,
        visible: true,
    });

    name: string = "myVisualCard";
    displayName: string = "My Formatting Card";
    analyticsPane: boolean = false;
    visible: boolean = true;
    slices: Array<FormattingSettingsSlice> = [this.myNumericSlice];
}

// Formatting settings card
class myVisualCardSettings extends FormattingSettingsCompositeCard {
    // Formatting settings slice
    myNumericSlice = new formattingSettings.NumUpDown({
        name: "myNumericSlice",
        displayName: "My Formatting Numeric Slice",
        value: 50,
        visible: true,
        options: {
            minValue: {
                type: powerbi.visuals.ValidatorType.Min,
                value: 0,
            },
            maxValue: {
                type: powerbi.visuals.ValidatorType.Max,
            }
        }
    });
}
```

```

        value: 100,
    },
}
});

name: string = "myVisualCard";
displayName: string = "My Formatting Card";
analyticsPane: boolean = false;
visible: boolean = true;

groupSetting = new myVisualGroupSettings(Object())
groups: Array<FormattingSettingsGroup> = [this.groupSetting]
slices: Array<FormattingSettingsSlice> = [this.myNumericSlice];
}

```

The *capabilities.json* property declaration should be:

JSON

```

"objects": {
    "myVisualCard": {
        "properties": {
            "myNumericSlice": {
                "type": {
                    "numeric": true
                }
            },
            "myAnotherNumericSlice": {
                "type": {
                    "numeric": true
                }
            },
        }
    }
}

```

## Formatting settings group

Some formatting settings cards can have groups inside. Groups consist of slices and can be expanded/collapsed.

Example declaring formatting settings group with one slice:

TypeScript

```

import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

import FormattingSettingsGroup = formattingSettings.Group;
import FormattingSettingsSlice = formattingSettings.Slice;

```

```

class myVisualGroupSettings extends FormattingSettingsGroup {
    myAnotherNumericSlice = new formattingSettings.NumUpDown({
        name: "myAnotherNumericSlice",
        displayName: "My Formatting Numeric Slice in group",
        value: 15,
        visible: true
    });

    name: string = "myVisualCard";
    displayName: string = "My Formatting Card";
    analyticsPane: boolean = false;
    visible: boolean = true;
    slices: Array<FormattingSettingsSlice> = [this.myNumericSlice];
}

```

## Formatting settings slice

The formatting settings slice type consists of two types of slices - [simple](#) and [composite](#).

Each slice contains formatting properties. There's a long list of available [formatting properties types](#).

Example declaring formatting settings slice of type `NumUpDown` with limitations:

The slice name should match property name from `capabilities.json`.

TypeScript

```

import { formattingSettings } from "powerbi-visuals-utils-
formattingmodel";

myNumericSlice = new formattingSettings.NumUpDown({
    name: "myNumericSlice",
    displayName: "My Formatting Numeric Slice",
    value: 50,
    visible: true,
    options: {
        minValue: {
            type: powerbi.visuals.ValidatorType.Min,
            value: 0,
        },
        maxValue: {
            type: powerbi.visuals.ValidatorType.Max,
            value: 100,
        },
    }
});

```

# Build formatting pane model using FormattingModel Utils

1. Open your `settings.ts` file.
2. Build your own formatting settings model with all its components (cards, groups, slices, properties ...), and name it `VisualFormattingSettings`. Replace your settings code with the following:

TypeScript

```
import { formattingSettings } from "powerbi-visuals-utils-formattingmodel";

import FormattingSettingsCompositeCard = formattingSettings.CompositeCard;
import FormattingSettingsSlice = formattingSettings.Slice;
import FormattingSettingsModel = formattingSettings.Model;

export class VisualSettingsModel extends FormattingSettingsModel {
    // Building my visual formatting settings card
    myVisualCard: FormattingSettingsCard = new myVisualCardSettings();

    // Add formatting settings card to cards list in model
    cards: Array<FormattingSettingsCompositeCard> = [this.myVisualCard];
}

class myVisualCardSettings extends FormattingSettingsCompositeCard {
    myNumericSlice = new formattingSettings.NumUpDown({
        name: "myNumericSlice",
        displayName: "My Formatting Numeric Slice",
        value: 100,
    });

    name: string = "myVisualCard";
    displayName: string = "My Formatting Card";
    analyticsPane: boolean = false;
    slices: Array<FormattingSettingsSlice> = [this.myNumericSlice];
}
```

3. In your capabilities file, add your formatting objects and properties

JSON

```
"objects": {
    "myVisualCard": {
        "properties": {
            "myNumericSlice": {
                "type": {
                    "numeric": true
                }
            }
        }
    }
}
```

```
        }
    }
}
```

4. In your visual class, import the following:

TypeScript

```
import { FormattingSettingsService } from "powerbi-visuals-utils-
formattingmodel";
import { VisualFormattingSettingsModel } from "./settings";
```

5. Declare formatting settings and formatting settings service

TypeScript

```
private formattingSettings: VisualFormattingSettingsModel;
private formattingSettingsService: FormattingSettingsService;
```

6. Initialize the formatting settings service in constructor

TypeScript

```
constructor(options: VisualConstructorOptions) {
    this.formattingSettingsService = new FormattingSettingsService();

    // ...
}
```

7. Build formatting settings in update API using formatting settings service

```
populateFormattingSettingsModel
```

TypeScript

```
public update(options: VisualUpdateOptions) {
    this.formattingSettings =
this.formattingSettingsService.populateFormattingSettingsModel(VisualFormat-
tingSettingsModel, options.dataViews);
    // ...
}
```

8. Build formatting model and return it in `getFormattingModel` API

TypeScript

```
public getFormattingModel(): powerbi.visuals.FormattingModel {
    return
    this.formattingSettingsService.buildFormattingModel(this.formattingSettings)
;
}
```

## Formatting property selector

The optional selector in the formatting properties descriptor determines where each property is bound in the dataView. There are [four distinct options](#).

You can add selector to formatting property in its descriptor object. This example is taken from the [SampleBarChart](#) for color custom visual data points using property selectors:

TypeScript

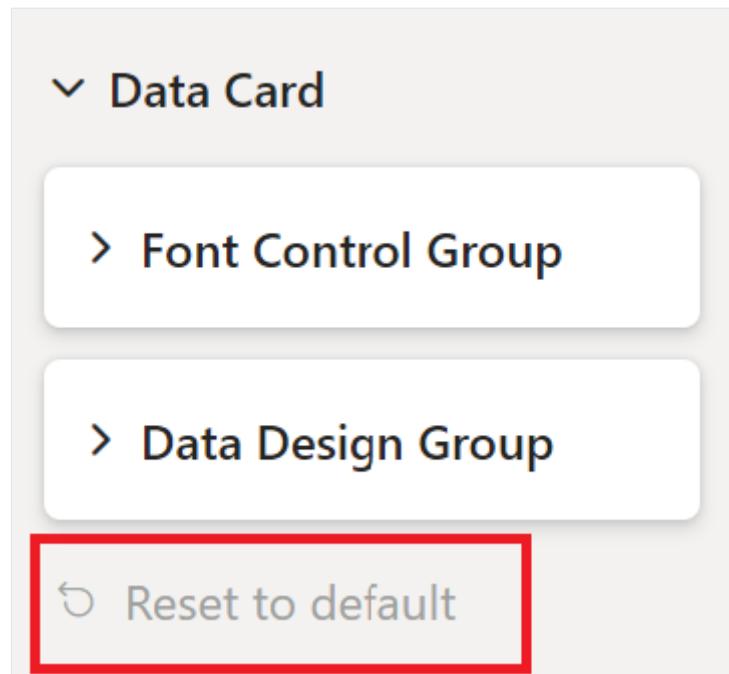
```
new formattingSettings.ColorPicker({
    name: "fill",
    displayName: dataPoint.category,
    value: { value: dataPoint.color },
    selector:
        dataViewWildcard.createDataViewWildcardSelector(dataViewWildcard.DataViewWildcardMatchingOption.InstancesAndTotals),
        altConstantSelector: dataPoint.selectionId.getSelector(),
        instanceKind: powerbi.VisualEnumerationInstanceKinds.ConstantOrRule
})
```

## Reset settings to default

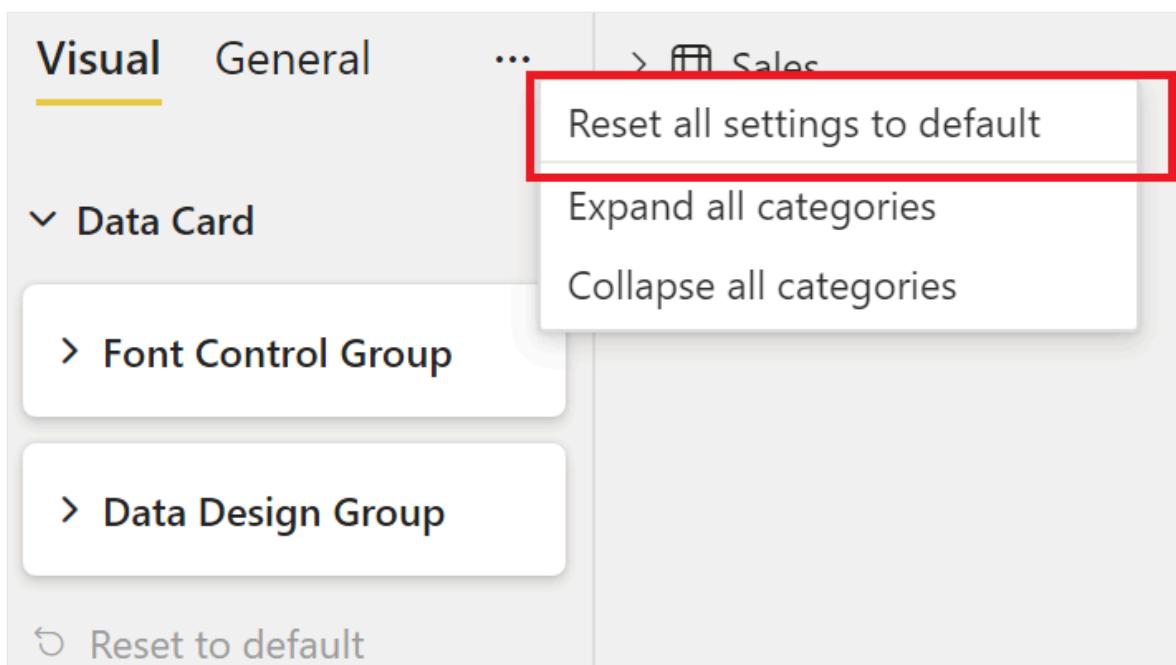
Formatting model utils enable you to [reset settings to default](#) by automatically adding all the formatting properties descriptors to the formatting card list of features to revert to default descriptors `revertToDefaultDescriptors`.

You can enable resetting formatting settings from:

- The formatting card *reset to default* button



- The formatting pane top bar *reset all settings to default* button



## Localization

For more about the localization feature and to set up localization environment, see [Add the local language to your Power BI visual](#).

Init formatting settings service with localization manager in case localization is required in your custom visual:

```
TypeScript  
  
constructor(options: VisualConstructorOptions) {
```

```
const localizationManager = options.host.createLocalizationManager();
this.formattingSettingsService = new
FormattingSettingsService(localizationManager);

// ...
}
```

Add `displayNameKey` or `descriptionKey` instead of `displayName` and `description` in the appropriate formatting component whenever you want a string to be localized. Example for building a formatting slice with localized display name and description

TypeScript

```
myFormattingSlice = new formattingSettings.NumUpDown({
    name: "myFormattingSlice",
    displayNameKey: "myFormattingSlice_Key",
    descriptionKey: "myFormattingSlice_DescriptionKey",
    value: 100
});
```

`displayNameKey` and `descriptionKey` values should be added to `resources.json` files.

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# On-object utils - subselection helper (preview)

Article • 02/19/2024

The `HTMLSubSelectionHelper` provides an easy way for your Power BI custom visual to emit subselections to Power BI, get and render outlines.

`HTMLSubSelectionHelper` is a helper class intended to facilitate the creation and subsequent management of subselection outlines. It contains methods for finding subselectable elements.

The utils exports CSS classes and attributes making it easier for the visual to define and maintain subselections.

## ① Note

Use version 6.0.1 or higher of the utils.

To define subselectable elements, we also need to add a class to each desired element.

[Expand table](#)

CSS Class	Purpose	Required
subselectable	Provides a selector for the <code>HTMLSubSelectionHelper</code> to find possible subselections	yes

To define subselections for the visual, there are a few attributes that need to be added to the desired elements.

[Expand table](#)

Attribute Name	Attribute	Purpose	Required	Type	Example
SubSelectableDisplayNameAttribute	data-sub-selection-display-name	Provide a localized string for display name of the subselected element	yes	string	data-sub-selection-display-name="Visual Title"
SubSelectableObjectNameAttribute	data-sub-selection-object-name	Provide an object name to associate with subselection shortcuts and style	yes	string	data-sub-selection-object-name="title"
SubSelectableTypeAttribute	data-sub-selection-type	Provide the type of the subselected style	yes	SubSelectionStylesType	data-sub-selection-type="1"
SubSelectableDirectEdit	data-sub-selection-direct-edit	Provide direct text edit references, including the CardUID, GroupUID, and the orientation of the text box	no	SubSelectableDirectEdit should be provided as a string	data-sub-selection-direct-edit={"reference": {"cardUid": "Visual-title", "groupUid": "title-text", "sliceUid": "title-text-text"}, "style": 0}

Attribute Name	Attribute	Purpose	Required	Type	Example
SubSelectableHideOutlineAttribute	data-sub-selection-hide-outline	Provide a boolean value for subselectable elements that shouldn't have an outline shown	no	boolean	data-sub-selection-hide-outline="true"
SubSelectableRestrictingElementAttribute	data-sub-selection-restricting-element	Used to indicate the element that will be restricted, the outlines, and the type of restriction (clamp or clip)	no	SubSelectionOutlineRestrictionType	data-sub-selection-restricting-element="1"
SubSelectableSubSelectedAttribute	data-sub-selection-sub-selected	Indicate whether the subselectable is selected or not	No, the helper assigns it to the elements when needed	Boolean	data-subselection-sub-selected="true"

## Format mode

When the visual enters format mode, You need to disable interactivity for the visual, as we expect the user to select the visual and visual element to initiate formatting.

## HTMLSubSelectionHelper public methods

The `HTMLSubSelectionHelper` has some public methods that you can use, but there are two main methods and the helper does the rest.

The two methods are `setFormatMode` and `updateOutlinesFromSubSelections`.

The public methods of the helper include:

- `createHtmlSubselectionHelper(args: HtmlSubselectionHelperArgs): HtmlSubSelectionHelper` - This is a static method that takes args of type `HtmlSubselectionHelperArgs` and returns an instance of `HTMLSubSelectionHelper`.
- `setFormatMode(isFormatMode: boolean): void` - This method sets the format mode for the `HTMLSubSelectionHelper`, If `isFormatMode` is true, the helper attaches relevant event listeners to the host element to enable format mode functionality (subselecting, rendering outlines).
- `getSubSelectionSourceFromEvent(event: PointerEvent): HtmlSubSelectionSource or undefined` - returns an `HtmlSubSelectionSource` object that is built according to the event parameter.
- `onVisualScroll(): void` - Indicates to the `HTMLSubSelectionHelper` that scrolling is currently occurring. Scrolling should remove outlines until scrolling is finished.
- `updateElementOutlines(elements: HTMLElement[], visibility: SubSelectionOutlineVisibility, suppressRender: boolean = false): SubSelectionRegionOutlineId[]` - update the outlines (and emits them to Power BI to be rendered) of the elements.

- `clearHoveredOutline(): void` - This method clears hovered outlines if they exist.
- `updateRegionOutlines(regionOutlines: HelperSubSelectionRegionOutline[], suppressRender: boolean = false): void` - update and emits the given outlines to get rendered.
- `getElementsFromSubSelections(subSelections: CustomVisualSubSelection[]): HTMLElement[]` - given subSelections, this method returns the relevant HTMLElements.
- `updateOutlinesFromSubSelections(subSelections: CustomVisualSubSelection[], clearExistingOutlines?: boolean, suppressRender?: boolean): void` - updates and renders the outlines for the given subSelection with respect to suppressRender and clearExistingOutlines.
- `hideAllOutlines(suppressRender: boolean = false): void` - hide all the outlines with respect to suppressRender.
- `getAllSubSelectables(filterType?: SubSelectionStylesType): CustomVisualSubSelection[]` - returns all the subSelectables according to the filterType.
- `createVisualSubSelectionForSingleObject(createVisualSubSelectionArgs: CreateVisualSubSelectionFromObjectArgs): CustomVisualSubSelection` - create CustomVisualSubSelection object from the createVisualSubSelectionArgs.
- `setDataForElement(el: HTMLElement | SVGElement, data: SubSelectionElementData): void` - a static method that sets data for the elements.
- `getDataForElement(el: HTMLElement | SVGElement): SubSelectionElementData` - a static method that gets the associated previously assigned using setDataForElement.

## HtmlSubselectionHelperArgs

TypeScript

```
interface HtmlSubselectionHelperArgs {
    /** Element which contains the items that can be sub-selected */
    hostElement: HTMLElement; // The host element, the helper will attach the listeners to this element
    subSelectionService: powerbi.extensibility.IVisualSubSelectionService; // subSelectionService which is provided in powerbi-visuals-api
    selectionIdCallback?: ((e: HTMLElement) => ISelectionId); // a callback that gets the selectionId for the specific element
    customOutlineCallback?: ((subSelection: CustomVisualSubSelection) =>
        SubSelectionRegionOutlineFragment[]); // a callback to get custom outline for the specific subSelection
    customElementCallback?: ((subSelection: CustomVisualSubSelection) => HTMLElement[]);
    subSelectionMetadataCallback?: ((subSelectionElement: HTMLElement) => unknown); // a callback to attach any meta data to the subSelection.
}
```

## SubSelectionStylesType

TypeScript

```
const enum SubSelectionStylesType {
    None = 0,
    Text = 1,
    NumericText = 2,
    Shape = 3,
}
```

## SubSelectableDirectEdit

TypeScript

```
interface SubSelectableDirectEdit {
    reference: SliceFormattingModelReference;
    style: SubSelectableDirectEditText;
```

```
        displayValue?: string;
    }
```

## SubSelectionOutlineRestrictionType

TypeScript

```
const enum SubSelectionOutlineRestrictionType {
    /**
     * A clamping element will adjust the outline to prevent it from extending beyond
     * the restricting element.
     */
    Clamp,
    /**
     * A clipping element will make parts of the outline not visible if the outline extends beyond
     * the
     * restricting element's bounds.
     */
    Clip
}
```

To add restriction options to a specific element use the `HTMLSubSelectionHelper` `setDataForElement` with this data type, the helper uses the data to update the outlines:

TypeScript

```
interface SubSelectionElementData {
    outlineRestrictionOptions?: SubSelectionOutlineRestrictionOptions;
}

/** Options used to indicate if a restricting element should allow outlines more space to
 * generate by adding padding or if the restricting element should constrict the outline more
 * by adding a margin.
 */
export interface SubSelectionOutlineRestrictionOptions {
    padding?: IOffset;
    margin?: IOffset;
}
```

## Example

In this example, we implement `customOutlineCallback` and `selectionIdCallback`. The following code is in Visual Code. We have an object in the visual called `arcElement`. We want to render the outline when the element is hovered or subselected.

TypeScript

```
import ISelectionId = powerbi.visuals.ISelectionId;

const enum BarChartObjectNames {
    ArcElement = 'arcElement',
    ColorSelector = 'colorSelector',
    ...
}

private ArcOutlines(subSelection: CustomVisualSubSelection):
powerbi.visuals.SubSelectionRegionOutlineFragment[] {
    const outlines: powerbi.visuals.SubSelectionRegionOutlineFragment[] = []
    if (subSelection?.customVisualObjects[0].objectName === BarChartObjectNames.ArcElement) {
        const outline: powerbi.visuals.ArcSubSelectionOutline = {
            type: powerbi.visuals.SubSelectionOutlineType.Arc,
            center: { x: this.arcCenterX, y: this.arcCenterY },
            startAngle: this.arcStartAngle,
            endAngle: this.arcEndAngle,
            innerRadius: this.arcInnerRadius,
            outerRadius: this.arcOuterRadius
        }
        outlines.push(outline)
    }
    return outlines
}
```

```

    };
    outlines.push({
        id: BarChartObjectNames.ArcElement,
        outline
    });
    return outlines;
}
}

public selectionIdCallback(e: Element): ISelectionId {
    const elementType: string = d3.select(e).attr(SubSelectableObjectNameAttribute);
    switch (elementType) {
        case BarChartObjectNames.ColorSelector:
            const datum = d3.select<Element, BarChartDataPoint>(e).datum();
            return datum.selectionId;
        default:
            return undefined;
    }
}
}

```

Import the `HtmlSubSelectionHelper`:

```
import { HtmlSubSelectionHelper } from 'powerbi-visuals-utils-onobjectutils';
```

In the constructor code, create the `HTMLSubSelectionHelper`:

TypeScript

```

constructor(options: VisualConstructorOptions) {
    ...
    this.subSelectionHelper = HtmlSubSelectionHelper.createHtmlSubselectionHelper({
        hostElement: options.element,
        subSelectionService: options.host.subSelectionService,
        selectionIdCallback: (e) => this.selectionIdCallback(e),
        customOutlineCallback: (e) => this.ArcOutlines(e),
    });
    ...
}

```

In update method of the visual add the following code to update the outlines of the subSelection, update the state of the format mode for the `HTMLSubSelectionHelper` and disable interactions that aren't for format mode if format mode is on:

TypeScript

```

public update(options: VisualUpdateOptions) {
    ...

    if (this.formatMode) // disabling interaction with the visual data in format mode
        barSelectionMerged.on('click', null);
        this.svg.on('click', null);
        this.svg.on('contextmenu', null);
    } else {
        this.handleBarClick(barSelectionMerged);
        this.handleClick(barSelectionMerged);
        this.handleContextMenu();
    }
    this.subSelectionHelper.setFormatMode(options.formatMode);
    const shouldUpdateSubSelection = options.type & powerbi.VisualUpdateType.Data
        || options.type & powerbi.VisualUpdateType.Resize
        || options.type & powerbi.VisualUpdateType.FormattingSubSelectionChange;
    if (this.formatMode && shouldUpdateSubSelection) {
        this.subSelectionHelper.updateOutlinesFromSubSelections(options.subSelections, true);
    }
    ...
}

```

## Related content

- Subselection API
  - On-object formatting API
- 

## Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Ask the community ↗](#)

# DataViewUtils

Article • 01/13/2024

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, 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` file.

## DataViewWildcard

`DataViewWildcard` provides the `createDataViewWildcardSelector` function to support a property's [conditional formatting](#).

`createDataViewWildcardSelector` returns a selector that's required for defining how the conditional formatting entry in the format pane will be applied, based on `dataviewWildcardMatchingOption (InstancesAndTotals (default), InstancesOnly, TotalsOnly)`.

Example:

TypeScript

```
import { dataViewWildcard } from "powerbi-visuals-utils-dataviewutils";

let selector =
  dataViewWildcard.createDataViewWildcardSelector(dataViewWildcard.DataViewWil
dcardMatchingOption.InstancesAndTotals);
// returns {data: [{dataViewWildcard:{matchingOption: 0}}]};
```

## 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 the role name and returns its index.

TypeScript

```
function getMeasureIndexOfRole(grouped: DataViewValueColumnGroup[],  
roleName: string): number;
```

Example:

TypeScript

```
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 the role name and returns its index.

TypeScript

```
function getCategoryIndexOfRole(categories: DataViewCategoryColumn[],  
roleName: string): number;
```

Example:

TypeScript

```
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.

TypeScript

```
function hasRole(column: DataViewMetadataColumn, name: string): boolean;
```

Example:

TypeScript

```
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.

TypeScript

```
function hasRoleInDataView(dataView: DataView, name: string): boolean;
```

Example:

TypeScript

```
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.

TypeScript

```
function hasRoleInValueColumn(valueColumn: DataViewValueColumn, name: string): boolean;
```

Example:

TypeScript

```

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 the values of the objects.

The module provides the following functions:

## getValue

This function returns the value of the specific object.

TypeScript

```
function getValue<T>(objects: DataViewObjects, propertyId: DataViewObjectPropertyIdentifier, defaultValue?: T): T;
```

Example:

TypeScript

```
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 from specified objects.

TypeScript

```
function getObject(objects: DataViewObjects, objectName: string, defaultValue?: IDataViewObject): IDataViewObject;
```

Example:

TypeScript

```

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.

TypeScript

```

function getFillColor(objects: DataViewObjects, propertyId:
DataViewObjectPropertyIdentifier, defaultColor?: string): string;

```

Example:

TypeScript

```

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 universal function retrieves the color or value of a specific object.

TypeScript

```
function getCommonValue(objects: DataViewObjects, propertyId: DataViewObjectPropertyIdentifier, defaultValue?: any): any;
```

Example:

TypeScript

```
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 the value of the object.

The module provides the following functions:

## getValue

This function returns a value of the object by the property name.

TypeScript

```
function getValue<T>(object: IDataViewObject, propertyName: string,  
defaultValue?: T): T;
```

Example:

TypeScript

```
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 the property name.

TypeScript

```
function getFillColorByPropertyName(object: IDataViewObject, propertyName:  
string, defaultColor?: string): string;
```

Example:

TypeScript

```
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 a series.

TypeScript

```
function categoryIsAlsoSeriesRole(dataView: DataViewCategorical,
seriesRoleName: string, categoryRoleName: string): boolean;
```

Example:

TypeScript

```
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.

TypeScript

```
function getSeriesName(source: DataViewMetadataColumn): PrimitiveValue;
```

Example:

TypeScript

```
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.

TypeScript

```
function isImageUrlColumn(column: DataViewMetadataColumn): boolean;
```

Example:

TypeScript

```
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.

TypeScript

```
function isWebUrlColumn(column: DataViewMetadataColumn): boolean;
```

Example:

TypeScript

```
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 an image URL.

TypeScript

```
function hasImageUrlColumn(dataView: DataView): boolean;
```

Example:

TypeScript

```
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 the properties of the formatting panel.

The class provides the following methods:

## getDefault

This static method returns an instance of `DataViewObjectsParser`.

TypeScript

```
static getDefault(): DataViewObjectsParser;
```

Example:

TypeScript

```
import { dataViewObjectsParser } from "powerbi-visuals-utils-dataviewutils";
// ...

dataViewObjectsParser.getDefault();

// returns: an instance of the DataViewObjectsParser
```

## parse

This method parses the properties of the formatting panel and returns an instance of `DataViewObjectsParser`.

TypeScript

```
static parse<T extends DataViewObjectsParser>(dataView: DataView): T;
```

Example:

TypeScript

```
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

**ⓘ Important**

`enumerateObjectInstances` was deprecated in API version 5.1. It was replaced by [getFormattingModel](#). Also, see [FormattingModel utils](#).

This static method enumerates properties and returns an instance of `VisualObjectInstanceEnumeration`.

Execute it in the `enumerateObjectInstances` method of the visual.

TypeScript

```

static enumerateObjectInstances(dataViewObjectParser:
    dataViewObjectsParser.DataViewObjectsParser, options:

```

```
EnumerateVisualObjectInstancesOptions): VisualObjectInstanceEnumeration;
```

Example:

TypeScript

```
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);
    }
}
```

# Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Chart utils

Article • 01/11/2024

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:

Bash

```
npm install powerbi-visuals-utils-chartutils --save
```

## Axis Helper

The axis helper (`axis` object in `utils`) provides functions to simplify manipulations that have an axis.

The module provides the following functions:

### getRecommendedNumberOfTicksForXAxis

This function returns the recommended number of ticks according to the width of chart.

TypeScript

```
function getRecommendedNumberOfTicksForXAxis(availableWidth: number): number;
```

Example:

TypeScript

```
import { axis } from "powerbi-visuals-utils-chartutils";
// ...
axis.getRecommendedNumberOfTicksForXAxis(1024);

// returns: 8
```

## getRecommendedNumberOfTicksForYAxis

This function returns the recommended number of ticks according to the height of chart.

TypeScript

```
function getRecommendedNumberOfTicksForYAxis(availableWidth: number);
```

Example:

TypeScript

```
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, measure metadata, and max tick count.

TypeScript

```
function getBestNumberOfTicks(
    min: number,
    max: number,
    valuesMetadata: DataViewMetadataColumn[],
    maxTickCount: number,
    isDateTime?: boolean
): number;
```

Example:

TypeScript

```
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.

TypeScript

```
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:

TypeScript

```
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, undefined or empty.

TypeScript

```
function isOrdinal(type: ValueTypeDescriptor): boolean;
```

Example:

TypeScript

```
import { axis } from "powerbi-visuals-utils-chartutils";
// ...
let type = ValueType.fromDescriptor({ misc: { barcode: true } });
axis.isOrdinal(type);

// returns: true
```

## isDateTime

Checks if a value is of the DateTime type.

TypeScript

```
function isDateTime(type: ValueTypeDescriptor): boolean;
```

Example:

TypeScript

```
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.

TypeScript

```
function getCategoryThickness(scale: any): number;
```

Example:

TypeScript

```
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  $\leq x$ .

TypeScript

```
function invertOrdinalScale(scale: d3.scale.Ordinal<any, any>, x: number);
```

Example:

TypeScript

```
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.

TypeScript

```
function findClosestXAxisIndex(  
    categoryValue: number,  
    categoryAxisValues: AxisHelperCategoryDataPoint[]  
): number;
```

Example:

TypeScript

```
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.

TypeScript

```
function diffScaled(  
    scale: d3.scale.Linear<any, any>,  
    value1: any,  
    value2: any  
): number;
```

Example:

TypeScript

```
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 an axis.

TypeScript

```
function createDomain(  
    data: any[],  
    axisType: ValueTypeDescriptor,  
    isScalar: boolean,
```

```
    forcedScalarDomain: any[],
    ensureDomain?: NumberRange
): number[];
```

Example:

TypeScript

```
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.

TypeScript

```
function getCategoryValueType(
  data: any[],
```

```
axisType: ValueTypeDescriptor,
isScalar: boolean,
forcedScalarDomain: any[],
ensureDomain?: NumberRange
): number[];
```

Example:

TypeScript

```
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.

TypeScript

```
function createAxis(options: CreateAxisOptions): IAxisProperties;
```

Example:

TypeScript

```
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 a customized domain, but it doesn't change when nothing is set.

TypeScript

```
function applyCustomizedDomain(customizedDomain: any[], forcedDomain: any[]): any[];
```

Example:

TypeScript

```
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.

TypeScript

```
function combineDomain(
    forcedDomain: any[],
    domain: any[],
    ensureDomain?: NumberRange
): any[];
```

Example:

TypeScript

```
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.

TypeScript

```
function powerOfTen(d: any): boolean;
```

Example:

TypeScript

```
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 label sizes and overlapping.

TypeScript

```
function hideCollidedLabels(
    viewport: IViewport,
    data: any[],
    layout: any,
    addTransform: boolean = false
    hideCollidedLabels?: boolean
): LabelEnabledDataPoint[];
```

Example:

TypeScript

```
let dataLabelManager = new DataLabelManager();
let filteredData = dataLabelManager.hideCollidedLabels(
  this.viewport,
  values,
  labelLayout,
  true,
  true
);
```

## IsValid

This static method checks if the provided rectangle is valid, that is, it has positive width and height.

TypeScript

```
function isValid(rect: IRect): boolean;
```

Example:

TypeScript

```
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 method provides the following functions, interfaces, and classes:

### getLabelPrecision

This function calculates precision from a provided format.

TypeScript

```
function getLabelPrecision(precision: number, format: string): number;
```

## getLabelFormattedText

This function returns format precision from the provided format.

TypeScript

```
function getLabelFormattedText(options: LabelFormattedTextOptions): string;
```

Example:

TypeScript

```
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.

TypeScript

```
function enumerateDataLabels(
    options: VisualDataLabelsSettingsOptions
): VisualObjectInstance;
```

## enumerateCategoryLabels

This function adds VisualObjectInstance for Category data labels to an enumeration object.

TypeScript

```
function enumerateCategoryLabels(
    enumeration: VisualObjectInstanceEnumerationObject,
    dataLabelsSettings: VisualDataLabelsSettings,
    withFill: boolean,
    isShowCategory: boolean = false,
    font_size?: number
): void;
```

## createColumnFormatterCacheManager

This function returns the Cache Manager that provides quick access to formatted labels.

TypeScript

```
function createColumnFormatterCacheManager(): IColumnFormatterCacheManager;
```

Example:

TypeScript

```
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 Power BI legends for Power BI visuals.

The module provides the following functions and interfaces:

### createLegend

This helper function simplifies Power BI Custom Visual legend creation.

TypeScript

```
function createLegend(  
    legendParentElement: HTMLElement, // top visual element, container in  
    which legend will be created  
    interactive: boolean, // indicates that legend should be interactive  
    interactivityService: IInteractivityService, // reference to  
    IInteractivityService 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:

TypeScript

```
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.

TypeScript

```
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.

TypeScript

```
function drawLegend(data: LegendData, viewport: IViewport): void;
```

Example:

TypeScript

```
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);
        }
    }
}
```

## Related content

- [Power BI visuals interactivity utils](#)

# Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Color utils

Article • 01/13/2024

This article will help you to install, import, and use color utils. Learn how to use color utils to apply themes and palettes on a Power BI visual's data points.

## Prerequisites

To use the package, install:

- [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:

Bash

```
npm install powerbi-visuals-utils-colorutils --save
```

This command installs the package and adds a package as a dependency to your `package.json` file.

## Usage

To use interactivity utils, you have to import the required component in the source code of the visual.

TypeScript

```
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. It provides a description and examples for each public interface.

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 a specific series value. If no explicit color or default color has been set, then the color is allocated from the color scale for this series.

TypeScript

```
getColorForSeriesValue(objects: IDataViewObjects, value: PrimitiveValue,  
themeColorName?: ThemeColorName): string;
```

### getColorForSeriesValue example

TypeScript

```
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 a specific measure.

TypeScript

```
getColorForMeasure(objects: IDataViewObjects, measureKey: any,
themeColorName?: ThemeColorName): string;
```

## Get color for measure example

TypeScript

```

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.

TypeScript

```

static normalizeSelector(selector: Selector, isSingleSeries?: boolean):
Selector;

```

## static normalizeSelector example

TypeScript

```

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. `ColorHelper` has the `isHighContrast` property.

## getThemeColor

This method returns the theme color.

TypeScript

```
getThemeColor(themeColorName?: ThemeColorName): string;
```

## getHighContrastColor

This method returns the color for high-contrast mode.

TypeScript

```
getHighContrastColor(themeColorName?: ThemeColorName, defaultColor?:  
    string): string;
```

## High-contrast mode example

TypeScript

```
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 a hex color to an RGB string.

TypeScript

```
function hexToRGBString(hex: string, transparency?: number): string
```

## hexToRGBString example

TypeScript

```
import { hexToRGBString } from "powerbi-visuals-utils-colorutils";

hexToRGBString('#112233');

// returns: "rgb(17,34,51)"
```

## rotate

Rotates RGB color.

TypeScript

```
function rotate(rgbString: string, rotateFactor: number): string
```

## rotate example

TypeScript

```
import { rotate } from "powerbi-visuals-utils-colorutils";

rotate("#CC0000", 0.25); // returns: #66CC00
```

## parseColorString

Parses any color string to RGB format.

TypeScript

```
function parseColorString(color: string): RgbColor
```

## parseColorString example

TypeScript

```
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

Calculates the highlight color from the rgbColor based on the lumianceThreshold and delta.

TypeScript

```
function calculateHighlightColor(rgbColor: RgbColor, lumianceThreshold: number, delta: number): string
```

## calculateHighlightColor example

TypeScript

```
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 a specific domain of numbers.

TypeScript

```
function createLinearColorScale(domain: number[], range: string[], clamp: boolean): LinearColorScale
```

## createLinearColorScale example

TypeScript

```
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 a string hex expression to a number, and calculates the percentage and R, G, B channels. Applies the percentage for each channel and returns the hex value as a string

with a pound sign.

TypeScript

```
function shadeColor(color: string, percent: number): string
```

## shadeColor example

TypeScript

```
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.

TypeScript

```
function rgbBlend(foreColor: RgbColor, opacity: number, backColor: RgbColor): RgbColor
```

## rgbBlend example

TypeScript

```
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.

TypeScript

```
function channelBlend(foreChannel: number, opacity: number, backChannel: number): number
```

## channelBlend example

TypeScript

```
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.

TypeScript

```
function hexBlend(foreColor: string, opacity: number, backColor: string): string
```

## hexBlend example

TypeScript

```
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"
```

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# SVG utils

Article • 01/10/2024

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:

Bash

```
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.

TypeScript

```
interface ClassAndSelector {
  class: string;
  selector: string;
}
```

## createClassAndSelector

This function creates an instance of ClassAndSelector with the name of the class.

TypeScript

```
function createClassAndSelector(className: string): ClassAndSelector;
```

Example:

TypeScript

```
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` method provides some special functions to generate strings that you can use with the SVG transform property.

The module provides the following functions:

### translate

This function creates a translate string for use with the SVG transform property.

TypeScript

```
function translate(x: number, y: number): string;
```

Example:

TypeScript

```
import { manipulation } from "powerbi-visuals-utils-svgutils";
// ...

manipulation.translate(100, 100);

// returns: translate(100,100)
```

### translateXWithPixels

This function creates a translateX string for use with the SVG transform property.

TypeScript

```
function translateXWithPixels(x: number): string;
```

Example:

TypeScript

```
import { manipulation } from "powerbi-visuals-utils-svgutils";
// ...

manipulation.translateXWithPixels(100);

// returns: translateX(100px)
```

## translateWithPixels

This function creates a translate string for use with the SVG transform property.

TypeScript

```
function translateWithPixels(x: number, y: number): string;
```

Example:

TypeScript

```
import { manipulation } from "powerbi-visuals-utils-svgutils";
// ...

manipulation.translateWithPixels(100, 100);

// returns: translate(100px,100px)
```

## translateAndRotate

This function creates a translate-rotate string for use with the SVG transform property.

TypeScript

```
function translateAndRotate(
  x: number,
  y: number,
```

```
px: number,  
py: number,  
angle: number  
): string;
```

Example:

TypeScript

```
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 use in a CSS transform property.

TypeScript

```
function scale(scale: number): string;
```

Example:

TypeScript

```
import { manipulation } from "powerbi-visuals-utils-svgutils";  
// ...  
  
manipulation.scale(50);  
  
// returns: scale(50)
```

## transformOrigin

This function creates a transform-origin string for use in a CSS transform-origin property.

TypeScript

```
function transformOrigin(xOffset: string, yOffset: string): string;
```

Example:

TypeScript

```
import { manipulation } from "powerbi-visuals-utils-svgutils";
// ...

manipulation.transformOrigin(5, 5);

// returns: 5 5
```

## flushAllD3Transitions

This function forces every transition of D3 to complete.

TypeScript

```
function flushAllD3Transitions(): void;
```

Example:

TypeScript

```
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)".

TypeScript

```
function parseTranslateTransform(input: string): { x: string; y: string };
```

Example:

TypeScript

```
import { manipulation } from "powerbi-visuals-utils-svgutils";
// ...
```

```
manipulation.parseTranslateTransform("translate(100px,100px)");

// returns: { "x":"100px", "y":"100px" }
```

## createArrow

This function creates an arrow.

TypeScript

```
function createArrow(
    width: number,
    height: number,
    rotate: number
): { path: string; transform: string };
```

Example:

TypeScript

```
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.

TypeScript

```
function getOffset(rect: IRect): IPPoint;
```

Example:

```
TypeScript

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 the size of the rectangle.

```
TypeScript

function getSize(rect: IRect): ISize;
```

Example:

```
TypeScript

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 the size of the rectangle.

```
TypeScript

function setSize(rect: IRect, value: ISize): void;
```

Example:

```
TypeScript

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.

```
TypeScript

function right(rect: IRect): number;
```

Example:

```
TypeScript

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.

```
TypeScript

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.

TypeScript

```
function topLeft(rect: IRect): IPPoint;
```

Example:

TypeScript

```
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.

TypeScript

```
function topRight(rect: IRect): IPPoint;
```

Example:

TypeScript

```
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.

TypeScript

```
function bottomLeft(rect: IRect): IPoint;
```

Example:

TypeScript

```
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.

TypeScript

```
function bottomRight(rect: IRect): IPoint;
```

Example:

TypeScript

```
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.

TypeScript

```
function clone(rect: IRect): IRect;
```

Example:

TypeScript

```
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 the rectangle to a string.

TypeScript

```
function toString(rect: IRect): string;
```

Example:

TypeScript

```
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 an offset to the rectangle.

TypeScript

```
function offset(rect: IRect, offsetX: number, offsetY: number): IRect;
```

Example:

TypeScript

```
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.

TypeScript

```
function add(rect: IRect, rect2: IRect): IRect;
```

Example:

TypeScript

```
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 rectangle to a specific point.

TypeScript

```
function getClosestPoint(rect: IRect, x: number, y: number): IPoint;
```

Example:

TypeScript

```
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.

TypeScript

```
function equal(rect1: IRect, rect2: IRect): boolean;
```

Example:

TypeScript

```
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.

TypeScript

```
function equalWithPrecision(rect1: IRect, rect2: IRect): boolean;
```

Example:

TypeScript

```
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 a rectangle is empty.

TypeScript

```
function isEmpty(rect: IRect): boolean;
```

Example:

TypeScript

```
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 a rectangle contains a specific point.

TypeScript

```
function containsPoint(rect: IRect, point: IPoint): boolean;
```

Example:

TypeScript

```
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.

TypeScript

```
function isIntersecting(rect1: IRect, rect2: IRect): boolean;
```

Example:

TypeScript

```
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.

TypeScript

```
function intersect(rect1: IRect, rect2: IRect): IRect;
```

Example:

TypeScript

```
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.

TypeScript

```
function combine(rect1: IRect, rect2: IRect): IRect;
```

Example:

TypeScript

```
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.

TypeScript

```
function getCentroid(rect: IRect): IPoint;
```

Example:

TypeScript

```
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
}*/
```

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Ask the community](#)

# Type utils

Article • 01/04/2025

TypeUtils is a set of functions and classes to extend the basic types for Power BI visuals.

## Installation

To install the package, 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` file.

## Double

The `Double` module provides abilities to manipulate precision of the numbers.

It provides the following functions:

### pow10

This function returns power of 10.

TypeScript

```
function pow10(exp: number): number;
```

Example:

TypeScript

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.pow10(25);

// returns: 1e+25
```

### log10

This function returns a 10 base logarithm of the number.

TypeScript

```
function log10(val: number): number;
```

Example:

TypeScript

```
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.

TypeScript

```
function getPrecision(x: number, decimalDigits?: number): number;
```

Example:

TypeScript

```
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.

TypeScript

```
function equalWithPrecision(x: number, y: number, precision?: number):
boolean;
```

Example:

TypeScript

```
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.

TypeScript

```
function lessWithPrecision(x: number, y: number, precision?: number):
boolean;
```

Example:

TypeScript

```
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 than or equal to the second value.

TypeScript

```
function lessOrEqualWithPrecision(x: number, y: number, precision?: number):
boolean;
```

Example:

TypeScript

```
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.

TypeScript

```
function greaterWithPrecision(x: number, y: number, precision?: number): boolean;
```

Example:

TypeScript

```
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 than or equal to the second value.

TypeScript

```
function greaterOrEqualWithPrecision(x: number, y: number, precision?: number): boolean;
```

Example:

TypeScript

```
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.

TypeScript

```
function floorWithPrecision(x: number, precision?: number): number;
```

Example:

TypeScript

```
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.

TypeScript

```
function ceilWithPrecision(x: number, precision?: number): number;
```

Example:

TypeScript

```
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.

TypeScript

```
function floorToPrecision(x: number, precision?: number): number;
```

Example:

TypeScript

```
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.

TypeScript

```
function ceilToPrecision(x: number, precision?: number): number;
```

Example:

TypeScript

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.ceilToPrecision(-506, 10);

// returns: -500
```

## roundToPrecision

This function rounds the number to the provided precision.

TypeScript

```
function roundToPrecision(x: number, precision?: number): number;
```

Example:

TypeScript

```
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.

TypeScript

```
function ensureInRange(x: number, min: number, max: number): number;
```

Example:

TypeScript

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.ensureInRange(-27.2, -10, -5);

// returns: -10
```

## round

This function rounds the number.

TypeScript

```
function round(x: number): number;
```

Example:

TypeScript

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...
```

```
double.round(27.45);  
// returns: 27
```

## removeDecimalNoise

This function rounds the number to eliminate some decimal spaces.

TypeScript

```
function removeDecimalNoise(value: number): number;
```

Example:

TypeScript

```
import { double } from "powerbi-visuals-utils-typeutils";  
// ...  
  
double.removeDecimalNoise(21.493000000000002);  
  
// returns: 21.493
```

## isInteger

This function checks if the number is an integer.

TypeScript

```
function isInteger(value: number): boolean;
```

Example:

TypeScript

```
import { double } from "powerbi-visuals-utils-typeutils";  
// ...  
  
double.isInteger(21.493000000000002);  
  
// returns: false
```

## toIncrement

This function increments the number by the provided number and returns the rounded number.

```
TypeScript
```

```
function toIncrement(value: number, increment: number): number;
```

Example:

```
TypeScript
```

```
import { double } from "powerbi-visuals-utils-typeutils";
// ...

double.toIncrement(0.6383723, 0.05);

// returns: 0.65
```

## Prototype

The `Prototype` module provides the ability to inherit objects.

It provides the following functions:

### inherit

This function returns a new object with the provided object as its prototype.

```
TypeScript
```

```
function inherit<T>(obj: T, extension?: (inherited: T) => void): T;
```

Example:

```
TypeScript
```

```
import { prototype } from "powerbi-visuals-utils-typeutils";
// ...

let base = { Microsoft: "Power BI" };

prototype.inherit(base);

/* returns: {
  __proto__: {
    Microsoft: "Power BI"
  }
}
```

```
        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.

TypeScript

```
function inheritSingle<T>(obj: T): T;
```

Example:

TypeScript

```
import { prototype } from "powerbi-visuals-utils-typeutils";  
// ...  
  
let base = { Microsoft: "Power BI" };  
  
prototype.inheritSingle(base);  
  
/* returns: {  
    __proto__: {  
        Microsoft: "Power BI"  
    }  
}*/
```

## PixelConverter

The `PixelConverter` module provides the ability to convert pixels to points, and points to pixels.

It provides the following functions:

### toString

This function converts the pixel value to a string.

TypeScript

```
function toString(px: number): string;
```

Example:

TypeScript

```
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.

TypeScript

```
function fromPoint(pt: number): string;
```

Example:

TypeScript

```
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.

TypeScript

```
function fromPointToPixel(pt: number): number;
```

Example:

TypeScript

```
import { pixelConverter } from "powerbi-visuals-utils-typeutils";
// ...

pixelConverter.fromPointToPixel(8);

// returns: 10.666666666666666
```

## toPoint

This function converts the pixel value to the point value.

TypeScript

```
function toPoint(px: number): number;
```

Example:

TypeScript

```
import { pixelConverter } from "powerbi-visuals-utils-typeutils";
// ...

pixelConverter.toPoint(8);

// returns: 6
```

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Power BI visuals test utils

Article • 01/12/2024

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.

## Prerequisites

To use this package, install:

- [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:

Bash

```
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.

TypeScript

```
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;
nit(): 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 more 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.

TypeScript

```

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 `testDataView`:

TypeScript

```
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 more 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.

TypeScript

```
import powerbi from "powerbi-visuals-api";

import VisualObjectInstancesToPersist =
powerbi.VisualObjectInstancesToPersist;
import ISelectionIdBuilder = powerbi.visuals.ISelectionIdBuilder;
import ISelectionManager = powerbi.extensibility.ISelectionManager;
import IColorPalette = powerbi.extensibilityIColorPalette;
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?: MockIAccountInteractions,
        localizationManager?: powerbi.extensibility.ILocalizationManager,
        telemetryService?: powerbi.extensibility.ILoggerService,
        authService?: powerbi.extensibility.IAuthenticationService,
        storageService?: ILocalVisualStorageService,
        eventService?: IVisualEventService);
    createSelectionIdBuilder(): ISelectionIdBuilder;
    createSelectionManager(): ISelectionManager;
    createLocalizationManager(): ILocalizationManager;
    colorPalette: IColorPalette;
    locale: string;
    telemetry: ITelemetryService;
    tooltipService: ITooltipService;
    allowInteractions: boolean;
    storageService: ILocalVisualStorageService;
    eventService: IVisualEventService;
    persistProperties(changes: VisualObjectInstancesToPersist): void;
}
```

- `createVisualHost` creates and returns an instance of **IVisualHost**, actually **MockIVisualHost**.

TypeScript

```
function createVisualHost(locale?: Object, allowInteractions?: boolean,
colors?: IColorInfo[], isEnabled?: boolean, displayNames?: any, token?:
```

```
string): IVisualHost;
```

Example:

TypeScript

```
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.

TypeScript

```
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**.

TypeScript

```
function createColorPalette(colors?: IColorInfo[]): IColorPalette;
```

Example:

TypeScript

```
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.

TypeScript

```
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**.

TypeScript

```
function createSelectionId(key?: string): ISelectionId;
```

Example:

TypeScript

```
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.

TypeScript

```
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**.

TypeScript

```
function createSelectionIdBuilder(): ISelectionIdBuilder;
```

Example:

TypeScript

```
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.

TypeScript

```
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**.

TypeScript

```
function createSelectionManager(): ISelectionManager
```

Example:

TypeScript

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

let selectionManager: ISelectionManager = createSelectionManager();
```

① Note

**MockISelectionManager** is a fake implementation of **ISelectionManager** and should only be used with unit tests.

## MockILocale

Sets the locale and changes it for your needs during a unit testing process.

TypeScript

```
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 an instance of **MockILocale**.

TypeScript

```
funciton createLocale(locales?: Object): MockILocale;
```

## MockITooltipService

Simulates `TooltipService` and calls it for your needs during a unit testing process.

TypeScript

```
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 an instance of **MockITooltipService**.

TypeScript

```
function createTooltipService(isEnabled?: boolean): ITooltipService;
```

## MockIAccountInteractions

TypeScript

```
export class MockIAccountInteractions {  
    constructor(public isEnabled?: boolean); // false by default  
}
```

- `createAccountInteractions` creates and returns an instance of **MockIAccountInteractions**.

TypeScript

```
function createAccountInteractions(isEnabled?: boolean):  
    MockIAccountInteractions;
```

## MockILocalizationManager

Provides basic abilities of **LocalizationManager**, which are needed for unit testing.

TypeScript

```
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**.

TypeScript

```
function createLocalizationManager(displayNames?: any):
```

```
ILocalizationManager;
```

Example:

TypeScript

```
import { createLocalizationManager } from "powerbi-visuals-utils-testutils";
let localizationManagerMock: ILocalizationManager =
createLocalizationManager();
```

## MockITelemetryService

Simulates **TelemetryService** usage.

TypeScript

```
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 Microsoft Entra token.

TypeScript

```
class MockIAuthenticationService implements IAuthenticationService {
    constructor(token: string);
    getAADToken(visualId?: string): powerbi.IPromise<string>
}
```

- `createAuthenticationService` creates and returns an instance of **IAuthenticationService**, actually **MockIAuthenticationService**.

TypeScript

```
function createAuthenticationService(token?: string):  
IAuthenticationService;
```

## MockIStorageService

Allows you to use **ILocalVisualStorageService** with the same behavior as **LocalStorage**.

TypeScript

```
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**.

TypeScript

```
function createStorageService(): ILocalVisualStorageService;
```

## MockIEventService

TypeScript

```
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**.

TypeScript

```
function createEventService(): IVisualEventService;
```

# Utils

Utils include helper methods for Power BI visuals' unit testing, including helpers related to colors, numbers, and events.

- `renderTimeout` returns a timeout.

TypeScript

```
function renderTimeout(fn: Function, timeout: number = DefaultWaitForRender): number
```

- `testDom` helps set a fixture in unit tests.

TypeScript

```
function testDom(height: number | string, width: number | string): JQuery
```

Example:

TypeScript

```
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`

TypeScript

```
function getSolidColorStructuralObject(color: string): any
```

Returns the following structure:

JSON

```
{ solid: { color: color } }
```

- `assertColorsMatch` compares `RgbColor` objects parsed from input strings.

TypeScript

```
function assertColorsMatch(actual: string, expected: string, invert: boolean = false): boolean
```

- `parseColorString` parses color from the input string and returns it in specified interface `RgbColor`.

TypeScript

```
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.

TypeScript

```
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.

TypeScript

```
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.

TypeScript

```
function clickElement(element: JQuery, ctrlKey: boolean = false): void
```

- `createTouch` returns a **Touch** object to help simulate a touch event.

TypeScript

```
function createTouch(x: number, y: number, element: JQuery, id: number = 0): Touch
```

- `createTouchesList` returns a list of simulated **Touch** events.

TypeScript

```
function createTouchesList(touches: Touch[]): TouchList
```

- `createContextMenuEvent` returns **MouseEvent**.

TypeScript

```
function createContextMenuEvent(x: number, y: number): MouseEvent
```

- `createMouseEvent` creates and returns **MouseEvent**.

TypeScript

```
function createMouseEvent(  
    mouseEventType: MouseEventType,  
    eventType: ClickEventType,  
    x: number,  
    y: number,  
    button: number = 0): MouseEvent
```

- `createTouchEndEvent`

TypeScript

```
function createTouchEndEvent(touchList?: TouchList): UIEvent
```

- `createTouchMoveEvent`

TypeScript

```
function createTouchMoveEvent(touchList?: TouchList): UIEvent
```

- `createTouchStartEvent`

```
TypeScript
```

```
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.

```
TypeScript
```

```
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 aren't 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:

```
TypeScript
```

```
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.

TypeScript

```
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,
}
```

## Related content

To write unit tests for webpack-based Power BI visuals, and unit test with `karma` and `jasmine`, see [Tutorial: Add unit tests for Power BI visual projects](#).

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Tooltip utils

Article • 01/10/2024

This article will help you to install, import, and use tooltip utils. This util is useful for tooltip customizations in Power BI visuals.

## Prerequisites

To use the package, you need:

- [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:

Bash

```
npm install powerbi-visuals-utils-tooltiputils --save
```

This command installs the package and adds a package as a dependency to your `package.json` file.

## Usage

The Usage Guide describes a public API of the package. You will find a description and some examples for each public interface of the package.

This package provides you with a way to create `TooltipServiceWrapper` and methods to help handle tooltip actions. It uses tooltip interfaces - `ITooltipServiceWrapper`, `TooltipEventArgs`, `TooltipEnabledDataPoint`.

It has specific methods (touch events handlers) related to mobile development:

`touchEndEventName`, `touchStartEventName`, `usePointerEvents`.

`TooltipServiceWrapper` provides the simplest way 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`.

TypeScript

```
function createTooltipServiceWrapper(tooltipService: ITooltipService,
rootElement: Element, handleTouchDelay?: number, getEventMethod?: () =>
MouseEvent): ITooltipServiceWrapper;
```

The `ITooltipService` is available in `IVisualHost`.

Example:

TypeScript

```
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.
    }
}
```

See an example of the custom visual [here ↗](#).

## I TooltipServiceWrapper

This interface describes public methods of the TooltipService.

TypeScript

```
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;  
}
```

### I TooltipServiceWrapper.addTooltip

This method adds tooltips to the current selection.

TypeScript

```
addTooltip<T>(selection: d3.Selection<any>, getTooltipInfoDelegate: (args:  
    TooltipEventArgs<T>) => VisualTooltipDataItem[], getDataPointIdentity?:  
    (args: TooltipEventArgs<T>) => ISelectionId, reloadTooltipDataOnMouseMove?:  
    boolean): void;
```

Example:

TypeScript

```
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.
```

See an example of the custom visual [here](#).

See an example of tooltip customization in a Gantt custom visual [here](#).

## ITooltipServiceWrapper.hide

This method hides the tooltip.

TypeScript

```
hide(): void;
```

Example:

TypeScript

```
import {createTooltipServiceWrapper} from "powerbi-visuals-utils-  
tooltiputils";  
  
let tooltipServiceWrapper =  
createTooltipServiceWrapper(options.host.tooltipService, options.element);  
// options are from the VisualConstructorOptions.  
  
tooltipServiceWrapper.hide();
```

## Interfaces

Interfaces are used during TooltipServiceWrapper creation and when it's used. They were mentioned in examples from previous articles [here](#).

## TooltipEventArgs

## TypeScript

```
interface TooltipEventArgs<TData> {
    data: TData;
    coordinates: number[];
    elementCoordinates: number[];
    context: HTMLElement;
    isTouchEvent: boolean;
}
```

## TooltipEnabledDataPoint

### TypeScript

```
interface TooltipEnabledDataPoint {
    tooltipInfo?: powerbi.extensibility.VisualTooltipDataItem[];
}
```

## TooltipServiceWrapperOptions

### TypeScript

```
interface TooltipServiceWrapperOptions {
    tooltipService: ITooltipService;
    rootElement: Element;
    handleTouchDelay: number;
```

## Touch events

Now tooltip utils can handle several touch events that are useful for mobile development.

## touchStartEventName

### TypeScript

```
function touchStartEventName(): string
```

This method returns a touch start event name.

## touchEndEventName

TypeScript

```
function touchEndEventName(): string
```

This method returns a touch end event name.

## usePointerEvents

TypeScript

```
function usePointerEvents(): boolean
```

This method returns the current touchStart event related to a pointer or not.

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)

# Create an SSL certificate

Article • 12/08/2023

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 your environment for developing a Power BI visual](#).

## 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:

```
PowerShell  
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](#).

## 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:

```
Windows Command Prompt  
brew install openssl  
brew link openssl --force
```

- By using *MacPorts*:

```
Windows Command Prompt  
sudo port install openssl
```

After you install the OpenSSL utility, run the following command to generate a new certificate:

```
Windows Command Prompt
```

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

## 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:

```
sh
```

```
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:

```
sh
```

```
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:

```
sh

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:

```
sh

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:

```
sh

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:

```
sh
```

```
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:

```
sh
```

```
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:

Windows Command Prompt

```
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:

Windows Command Prompt

```
%appdata%\npm\node_modules\PowerBI-visuals-tools\certs
```

- For the local instance of the tools:

Windows Command Prompt

```
<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:

1. In the config file, specify:

```
Windows Command Prompt  
\PowerBI-visuals-tools\config.json
```

2. In the `server` section, specify the passphrase by replacing the <YOUR PASSPHRASE> placeholder:

```
Windows Command Prompt  
  
"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>  
}
```

## Related content

- [Develop a Power circle card BI visual](#)
- [Power BI visuals samples](#)
- [Publish a Power BI visual to AppSource](#)

---

## Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Ask the community ↗](#)