

IBM Cognos Analytics
Version 12.0.x

Custom Visualizations Developer Guide



Contents

Chapter 1. Introduction.....	1
Developing custom visualizations.....	1
Chapter 2. Step by step guide.....	3
Step-by-step guide for custom visualizations.....	3
Setting up the development environment.....	3
Uninstalling the custom visualizations CLI tools.....	4
Creating a custom visualization.....	4
Using a D3 sample in your custom visualization.....	5
Validating the visualization in a dashboard.....	6
Validating the visualization in a report.....	7
Package and deploy the custom visualization.....	7
How local custom visualizations work.....	8
Chapter 3. Custom visualizations tools and commands.....	9
Custom visualizations tools and commands.....	9
Chapter 4. Customvis library.....	13
Custom visualizations library (customvis-lib).....	13
Rendering.....	13
Data model.....	16
Interactivity.....	18
Legends.....	20
Properties.....	21
Palettes.....	23
Customizing your visualization.....	25
RenderBase.....	26
Chapter 5. Customvis API reference.....	29
Customvis api reference.....	29
Chapter 6. Customvis frequently asked question.....	31
FAQ custom visualizations.....	31
Chapter 7. Visualization definition.....	33
Visualization definition.....	33
Chapter 8. Tutorial.....	37
Custom visualizations - tutorial.....	37
Part 1: Creating a custom visualization.....	38
Part 2: Adding properties to the visualization.....	43
Part 3: Customize legend and properties.....	45
Part 4: Advanced features.....	47
Chapter 9. Manage custom visuals.....	51
Managing custom visuals from the admin UI.....	51
Chapter 10. Improving the performance of custom visualizations.....	53
Improving the performance of custom visualizations.....	53

Chapter 11. Authoring schematics - Tutorial.....	55
SVG definition.....	55
SVG basic mapping.....	55
Rules and restrictions for SVG.....	57
Adding views to schematics.....	59
Highlighting regions in a schematic.....	60
Editing a schematic package.....	62
Distributing a schematic.....	64

Chapter 1. Introduction

Developing custom visualizations

You can create custom visualizations to meet your specific business needs. IBM® Cognos Analytics allows developers to create and test custom visualizations locally before they make them available to other users.

The **Manage Visualizations** capability allows users to control access rights to extensible visualizations for individual users, groups, and roles.

The **Develop Visualizations** capability allows users to develop extensible visualizations.



CAUTION: Be judicious when you assign **Develop Visualizations** access and ensure that you review files that are being uploaded. People who are permitted to upload files may be able to deliver malicious code.

Custom visualization code samples are available in the [Samples Guide](#).

Chapter 2. Step by step guide

Step-by-step guide for custom visualizations

This step-by-step guide explains how to set up the development environment, create a custom visualization from an existing D3 example and how to deploy and authorize the custom visualization in IBM Cognos Analytics.

Setting up the development environment

The IBM Cognos Analytics custom visualizations CLI tools require you to install the NodeJS runtime environment with a NodeJS package manager (NPM).

Before you begin

Make sure that the following software is installed:

- Microsoft Windows PowerShell or CMD on Windows or Terminal on Apple OSX.
- NodeJS version 16. For more information, see <https://nodejs.org/en/>.

Check the setup of your npm registry:

Note: Your npm registry can be public or private.

- If you are using a public npm registry, you must have access to the internet.
- If you are using a private npm registry, refer to the [npm documentation](#) for instructions on pointing your Node.js project at a private registry.

To use Appsody Node.js stacks in an air gap environment, your private npm registry must contain all the dependent npm modules that are used in the stack. You can discover the list of first-level dependencies in the package.json file in the stack definition. Alternatively, you can run `npm ls` or inspect the package-lock.json file in an initialized project to see the full list of npm dependencies. Follow your organization's standard procedures for copying npm modules into your private registry.

Procedure

Installing Node.JS and NPM

1. Download NodeJS version 16 installer from <https://nodejs.org/en/download/>.
2. Run the installer and use the default settings.
3. After the installation, restart your computer.
4. Validate the node and npm installation.
 - a) Open a command-line interface (CLI). Powershell or CMD on Windows or Terminal on OSX.
 - b) Run the following command:

```
node --version
```

The CLI displays the current version of the node. Make sure it meets the prerequisites.

Installing the Cognos Analytics custom visualizations CLI tools

5. Download the Cognos Analytics custom visualizations command-line tools from `[CAserverroot]/bi/js/vida/customvis.tgz`.

For example, `https://CognosAnalytics.mycompany.com/bi/js/vida/customvis.tgz`.

In some browsers, the file extension .tgz is recognized as text. Download the file by right-clicking the link and choose **Save as** and set type to **All Files**. Do not use the Mozilla Firefox browser to download the file.

6. In your file browser, navigate to the directory where the downloaded file is saved to.
7. Open a command-line tool.
 - On Microsoft Windows systems Windows PowerShell or CMD.
 - On Apple OSX systems Terminal.
8. Run the following command:

```
npm install -g customvis.tgz
```

On UNIX and Apple OSX systems, you might get a permission denied error. To solve this error, run the following command as an administrator:

```
sudo npm install -g customvis.tgz
```

The custom visualizations CLI tools are installed on your system.

9. After the process is completed, run the following command to validate the custom visualizations CLI tools.

```
customvis --help
```

The CLI displays the available commands.

10. In your file browser, delete the file `customvis.tgz`.

Uninstalling the custom visualizations CLI tools

If you do not need the IBM Cognos Analytics custom visualizations CLI tools any longer, you can uninstall them.

Procedure

To uninstall the custom visualizations CLI tools and its dependencies, run the following command.

```
npm uninstall -g @businessanalytics/customvis
```

Creating a custom visualization

Before you can implement a custom visualization, you must create a custom visualization.

About this task

The `customvis create` command generates a new custom visualization from a default template. However, you can specify a specific template on which the new custom visualization is based on. For more information, see the [“Create” on page 9](#) command.

Procedure

1. Open a command-line interface (CLI) in one of your chosen directories and run the following command:

```
customvis create CustomVisualization
```

- A directory with the name `CustomVisualization` is created.
- You can choose another name instead of `CustomVisualization`. For example, `Marimekko`, which creates a new directory called `Marimekko`.

2. In the CLI change the directory, by running:

```
cd CustomVisualization
```


3. When you develop a custom visualization, the custom visualizations CLI tools build the sources. The sources for the bundle are located in the `/renderer` directory.
4. You can now start implementing the visualization.

Using a D3 sample in your custom visualization

You use a D3 sample in this step-by-step guide. For more information, see [Collision Detection](https://bl.ocks.org/mbostock/3231298/) at <https://bl.ocks.org/mbostock/3231298/>.

About this task

The example does not support data binding, extra interactivity, and so on. Further information on implementing extra custom visualization capabilities can be found in the API development documentation. Further implementation must be done by a developer with JavaScript & D3 skills.

Procedure

1. Open the `renderer/Main.ts` file in a code editor.
2. The `Main.ts` file is the entry point of the visualization and runs always first. The content of the `Main.ts` file is:

```
import { RenderBase } from "@businessanalytics/customvis-lib";
export default class extends RenderBase
{
    protected create( _node: HTMLElement ): HTMLElement
    {
        _node.textContent = "Basic VizBundle";
        return _node;
    }
};
```

Note: This code is written in TypeScript, which is a super-set of JavaScript. However, for some code areas you can also use JavaScript. This usage is demonstrated in this step-by-step guide.

3. Open a command-line interface (CLI) in the directory that was made in [“Creating a custom visualization”](#) on page 4 and run the following command:

```
customvis start
```

This command builds the sources and starts a local server that hosts the custom visualization. Additionally, the custom visualizations CLI tools start monitoring the source files and rebuild the custom visualization when its source code changes. It is recommended to have this command running during development of the visualization.

To stop the local server that hosts the custom visualization, press Ctrl C.

4. Edit the `Main.ts` file in such a way that the D3 example can be used in IBM Cognos Analytics.
 - a) Determine what version of D3 is used in the online example. The version can be found by searching d3 in the script tag of the example. The online example uses d3.v3. Import this dependency in the `Main.ts` file by adding the following to the top of the file:

```
import * as d3 from "https://d3js.org/d3.v3.min.js";
```

- b) Replace

```
_node.textContent = "Basic VizBundle";
```

with the content in the script tag of the example:

```
{content of the script tag of the example}
```

- c) Ensure that the rendering happens in the container of the widget. Replace

```
d3.select("body").append("svg")
```

with

```
d3.select(_node).append("svg")
```

5. Save the file. If everything works correctly, the CLI displays that the sources finished building.

Results

You can now validate the custom visualization in Cognos Analytics.

Validating the visualization in a dashboard

You must validate a created visualization to ensure that the visualization works in IBM Cognos Analytics. To develop a custom visualization locally, use the Google Chrome browser. However, Mozilla Firefox and Microsoft Edge are also supported.

Before you begin

When you develop a new custom visualization, run the command:

```
customvis start
```

to start the development [on a local server](#).

Note: The Apple Safari browser is not supported for local custom visualization development.


When you validate the visualization, your browser connects to your custom visualization that runs locally. Some browsers can treat this connection as a security threat. For example, the Google Chrome browser might block the connection of a secured Cognos Analytics application (HTTPS) with your insecure local development environment (HTTP). To allow such a connection, in your Chrome browser, set the **Block insecure private network requests** flag to **Disabled** in your Chrome browser, see [the Chrome documentation](#).

Note: Remember to change the flag back to **Enabled** when you finish testing.

About this task

After you upload the custom visualization to Cognos Analytics, you can run it in any supported browser.

Procedure

1. Create a dashboard in Cognos Analytics.
2. Click the **Visualization** tab and then click the **Custom** tab.
3. Under **Developers widgets**, double-click **Test visualization**.
4. To see the changes in the **Page preview** view, click **Refresh visualization** .

Results

Cognos Analytics displays the visualization when you insert the values that the tested visualization requires in the **Fields** and **Properties** panes.

Validating the visualization in a report

It is recommended to use the Google Chrome browser when you develop a custom visualization locally. However, Mozilla Firefox and Edge are also supported.

Before you begin



Note: The Apple Safari browser is not supported for local custom visualization development.

When you validate the visualization, your browser connects to your custom visualization that runs locally. Some browsers might treat the connection as a security threat.

About this task

After the visualization was uploaded to IBM Cognos Analytics, any supported browser can be used to run the uploaded custom visualization.

Procedure

1. Create a report in Cognos Analytics. Make sure that you are in **Page preview** mode.
2. Under **Insertable objects**, click the **Toolbox** icon .
3. Expand the **Advanced** menu and double-click **Custom visual preview**.
The custom visualization is loaded.
4. The custom visualizations CLI tools monitors the source files and rebuilds the custom visualization when its source code changes. To see these changes in the **Page preview** view, click **Refresh visualization** .
5. Run your report as HTML by clicking **Play** and then **Run HTML**.

Results

Cognos Analytics displays the D3 visualization.

Package and deploy the custom visualization

When the custom visualization is ready to be published, you can use the custom visualization command-line interface (CLI) tools to build and pack the visualization. You can upload the packed visualization to IBM Cognos Analytics.

Packaging the custom visualization

Before you can deploy the custom visualization, you must package it.

Procedure

1. Open a command-line interface (CLI) where the custom visualization is located, created in [“Creating a custom visualization”](#) on page 4. Make sure that in the CLI you browse to the directory where the custom visualization is located.
2. Run the following command in the command-line interface:

```
customvis pack
```

The custom visualization is built and packed into a .zip file.

Results

A .zip file is generated in the current directory: {directoryName}.packed.zip. For example: CustomVisualization.packed.zip.


If you have the **Manage Visualizations** capability, you can upload the packaged.zip file to IBM Cognos Analytics.

If you do not have the **Manage Visualizations** capability, you can send the file to someone who has the **Manage Visualizations** capability. If you want to run the custom visualization locally, then extract the packaged.zip file and run the **customvis start** command from a CLI in the extracted folder.

Uploading custom visualizations from the admin UI

Before you can use the custom visualization in a dashboard or report, you must upload it.

Procedure

1. On the home page, click **Manage > Customization > Custom visuals > Upload custom visual** .
2. Select the CustomVisualization.packed.zip file and click **Open**.

Results

The custom visualization is added to the list of visualizations. Optionally you can set the permissions on the visualizations.

Using the uploaded visualization in a dashboard

You can use a custom visualization in an IBM Cognos Analytics dashboard.

Procedure

1. Create a dashboard in Cognos Analytics.
2. Click the **Visualization** tab > **Custom**.

Results

The uploaded custom visualization is selectable from the list of **Custom visuals**.

Using the uploaded visualization in a report

You can use a custom visualization in a IBM Cognos Analytics report.

Procedure

1. Create a report in Cognos Analytics.
2. Create a visualization.

Results

The uploaded custom visualization is selectable from the **Custom** section.

How local custom visualizations work

Using the provided custom visualization command-line (CLI) tools, a local server is started on the customvis start command.

This local server hosts files that are generated by the CLI tools in the build directory. You can access these files over the local network by the browser. IBM Cognos Analytics Reporting & Dashboard includes sandbox widgets that connect to this local server (127.0.0.1) and access the generated files. If these files exist, then these files run in the corresponding widget as a visualization.

Chapter 3. Custom visualizations tools and commands

Custom visualizations tools and commands

With IBM Cognos Analytics custom visualizations tools you can create, compile, run, clean, and pack visualizations from a command-line interface (CLI).

In addition to compiling a visualization, you can use the custom visualizations to host a custom visualization locally to test the visualization in Cognos Analytics before you deploy it.

You use the Cognos Analytics custom visualization tools from commands in a command-line interface (CLI). You run these commands from Microsoft Windows PowerShell or CMD on Windows or from Terminal on Apple OSX.

For installation of the Cognos Analytics custom visualization tools, see [“Setting up the development environment”](#) on page 3.

Help

Prints a help dialogue of all available commands and any additional options that can be used.

```
customvis --help
```

or

```
customvis
```

Create

Creates a visualization within a folder called name.

```
customvis create [options] <name>
```

or

```
customvis c [options] <name>
```

Options:

- **-t, --template <template>** Creates a project from template (default: Basic). This flag determines the initial template that the visualization is generated from. To list all available templates, see the [List](#) command.

```
customvis c -t BarChart <name>
```

- **-f, --force** Forces folder creation with name. Note: If you specify an existing name for the folder, the existing folder is deleted .

```
customvis c -f <name>
```

- **-h, --help** Output usage information

List

Lists all the available templates that can be used for a new custom visualization during the [Create](#) command.

```
customvis list
```

or

```
customvis l
```

Start

Builds and starts a local server that hosts the current visualization.

- The CLI tool builds the sources.
- A local server starts to host the custom visualization.

```
customvis start [options]
```

or

```
customvis s [options]
```

Options:

- `-p, --port`
Starts a server on a specific port (default 8585).

```
customvis start -p 8585
```

- `-h, --help`
Output usage information

```
customvis start -h
```

Pack

Creates a package of the visualization in the current root directory, `<folder-name>.packed.zip`. You can upload this .zip file to Cognos Analytics.

```
customvis pack
```

or

```
customvis p
```

Build

Builds and compiles the visualization in the current directory. Use this to test the deployment version locally.

```
customvis build
```

or

```
customvis b
```

Clean

Cleans the current visualization and removes all the files that are generated by the command-line tool. Use this when you need to share a development version of the visualization.

```
customvis clean
```

or

```
customvis cl
```

Upgrade

Upgrades the customvis-lib in the current visualization. Replaces the customvis-lib dependency with the latest one in the current CLI tool. Also triggers the [Clean](#) command.

```
customvis upgrade
```

or

```
customvis up
```

Info

Get information about the CLI tool and the current visualization. Shows the Cognos Analytics extensible information about the CLI tool and the current visualization.

```
customvis info
```

or

```
customvis i
```

Version

Shows the current Cognos Analytics custom visualization CLI tool version.

```
customvis --version
```

or

```
customvis -V
```

Chapter 4. Customvis library

Custom visualizations library (customvis-lib)

The customvis-lib library is designed to help developers build visualizations quickly by providing documented utility classes and interfaces. This document is intended for developers.

Features that are offered by customvis-lib include:

- A managed [“Rendering” on page 13](#) cycle
- A unified [“Data model” on page 16](#)
- Automatic or custom [“Hit testing” on page 18](#)
- Automatic [“Legends” on page 20](#)
- [“Properties” on page 21](#) support
- [“Palettes” on page 23](#) support
- [“Customizing your visualization” on page 25](#)

At the core of customvis-lib, is the [“RenderBase” on page 26](#) class. The RenderBase class is the base class that you can use for every custom visualization. This base class gives you access to all of the features in customvis-lib.

Installation

The library is available as a file called `customvis-lib-x.y.z.tgz` where `x.y.z` is the version of the library. When you create a new visualization that uses the SDK command `customvis create`, the customvis library is copied to the root directory of your new visualization.

Step-by-step visualization

Learn how to build a custom visualization that uses customvis-lib and d3 by following the [“Custom visualizations - tutorial” on page 37](#).

API Reference

Check out the [API reference](#) for detailed information about all the classes and methods that are found in customvis-lib.

Rendering

Rendering makes sure that your data is transformed to pixels that eventually display on your screen.

The result of rendering can be a Document Object Model (DOM) structure, a scalable vector graphic (svg), or pixels on a canvas, depending on which output technology you choose. All code samples use svg rendering in d3. However, in your custom visualization you can choose any other way of rendering.

Within customvis-lib, you must implement a create phase and an update phase:

- The create phase is run exactly once, when the visualization gets initialized.
- The update phase is run each time an aspect of the visualization changes, like the size, data or properties of the visualization.

Implementing these phases is optional. The RenderBase base class provides a default implementation for you.

Create

The create phase is run when the visualization gets initialized. You can implement this phase by overriding the “create()” on [page 26](#) method of `RenderBase`. In this method, you receive the DOM node of the container element for your visualization. You don’t must override create if your visualization does not have initialization code. However, if you do, you can add more DOM or svg nodes to your container and optionally return an `Element` as the new root node of your visualization. An example:

```
protected create( _node: HTMLElement ): void
{
    // Append an svg node that scales to the size of its container.
    d3.select( _node ).append( "svg" )
        .attr( "width", "100%" )
        .attr( "height", "100%" );
    // No return value means _node will be the root of our visualization.
}

protected update( _info: UpdateInfo ): void
{
    // _info.node will be the same '_node' that was passed to 'create'.
}
```

If your initialization requires asynchronous calls, you can return a `Promise` object that resolves with either an `Element` or `void`.

```
protected create( _node: HTMLElement ): void
{
    return new Promise( resolve =>
    {
        // doInit performs async initialization.
        doInit( function cb()
        {
            // Init is done, resolve the Promise. Optionally you can
            // resolve with an 'Element' that will become the root
            // node of your visualization.
            resolve();
        } );
    } );
}
```

Update

The update phase gets called each time something changes in the size, properties, or data of the visualization. You can implement the update phase by overriding the “update()” on [page 27](#) method of `RenderBase`.

Render completion

The implementation of update in its most basic form is the following code:

```
protected update( _info: UpdateInfo ): void
{
    _info.node.textContent = "Hello World: " + Date.now().toString();
}
```

There is no return value, which means that the rendering completed at the end of the update function.

Each visualization has the responsibility to notify its host when it completed rendering in the update phase. Notification is done through the return type: returning `void` implies that rendering is done when the update function ends and returning a `Promise` means that the host should wait for that promise to be fulfilled.

If you want your visualization to show animations or transitions, this is often an asynchronous process. Therefore, the update function can return a `Promise` object that resolves as soon as the transition of the visualization completed. An example illustrates this:

```
protected update( info: UpdateInfo ): Promise<void>
{
    // ...
}
```

```

const t = d3.transition().duration( 300 );

d3.select( _info.node )
  .selectAll( "rect" )
  .data( _info.data.rows )
  .join( "rect" )
    .attr( "x", d => scale( d.tuple( 0 ).key ) )
    .attr( "width", d => scale.bandwidth() )
    .transition( t ) // bind to our transition
      .attr( "height", d => d.values( 0 ) );

return new Promise( resolve => t.on( "end", () => resolve() ) );
}

```

In the `RenderBase` base class, you can see that the update method has the return signature `void | Promise<void>`. However, in your override you can be more explicit and define either `void` or `Promise<void>` as the return type.

If for some reason, you want the host application to know that rendering failed, you can either return a rejected promise or produces a new `Error("msg")`.

UpdateInfo

The one parameter that is passed to update is an object of type `UpdateInfo`. You can see this object as the state that you are about to render. The `UpdateInfo` class holds the following information:

- **reason:** The reason why rendering should take place. Check the flags in this object to see which aspects of the visualization need rendering. Available flags are:
 - **data:** data is changed since last call to update.
 - **size:** size is changed since last call to update.
 - **properties:** properties are changed since last call to update.
 - **decorations:** decorations (selected, highlighted) are changed since last call to update.
- **data:** The data that should be rendered, or null if there is no data. If there is no data to render, it is the responsibility of the visualization to ensure that the visualization is cleared and rendered in an empty state.
- **node:** The HTML element on which rendering should take place. This is the element that was returned previously from the `RenderBase.create` method.
- **properties:** The current set of properties. Properties can also be accessed through the `RenderBase.properties` attribute.
- **locale:** The data locale that should be used for rendering. Do not use for translations, only for custom data formatting where that is necessary.

The following example shows how you can use the fields of `UpdateInfo` to provide an optimized implementation of update:

```

protected update( _info: UpdateInfo ): void
{
  const reason = _info.reason;

  if ( !_info.data )
    return renderEmpty( _info.node );

  // Call expensive data processing function only if data has changed.
  if ( reason.data )
    this.data = processData( _info.data );

  // Perform rendering, based on `this.data` ...
}

```

Data model

Custom visualizations are based on a tabular data model.

Data is organized in rows and columns, where every row corresponds with a data point and every column corresponds with a slot in your visualization definition. For more information, see [“Visualization definition”](#) on page 33. Data is passed to the visualization in the update method, in which it can be used to bind to the elements that make up the visualization. Objects in the data model are also used in hit testing functions to inform a hosting application which data point or tuple was hit at a specific coordinate.

Slots

A slot is the entry point for data that goes into a visualization. For instance, a column visualization might have two slots: one slot that represents the categorical elements that go on the x-axis and one slot that represents the data values that determine the height of a column.

A slot can be defined in the `vizdef.xml` file, as either categorical or continuous:

- The data elements that are found in categorical slots are tuples. For instance, in the column chart above the categories slot might hold the tuples ['summer', 'winter', 'spring', 'autumn']. You might see this list of tuples as the categorical domain of the slot. In a code example further on, you can see that you use the tuples as a domain in a d3 scale object.
- A continuous slot holds data elements that represent values. Instead of a list of tuples, you can retrieve a value domain for this type of slot. A value domain is a minimum value and a maximum value. The domain in the values slot of the column chart example might be something like: [25.4, 136.4].

```
<slotname="categories" type="cat" optional="true" /><slotname="values" type="cont" optional="false" />
```

Slots can be either mapped or not mapped. If a slot is mapped, then some data was assigned to that slot. If a slot is not mapped, then the slot is not associated with data and its tuples attribute are [] and domain are [0, 0].

If a slot is optional, it means that you can, but do not need to associate data with that slot. The visualization is responsible for handling this situation correctly. For example, in a column chart the series slot might be optional. If it is not completed, the visualization can render a simple bar chart with only categories and values. If it is completed, a clustered bar chart can be rendered.

Tuples and domains

Every column in your data corresponds with a slot in your visualization. Slots can be either categorical or continuous. This means that a data column can hold either tuples (for categorical slots) or values (for continuous slots).

The following example shows how you can retrieve the tuples and the value domain for a data set and use them in a d3 scale:

```
protected update( _info: UpdateInfo ): void
{
    const data = _info.data;
    const width = _info.node.clientWidth;
    const height = _info.node.clientHeight;

    // Create a list of tuple keys for categorical column 0 and use it
    // to create a band scale that maps the keys to an axis length.
    const keys = data.cols[ 0 ].tuples.map( _t => _t.key );
    const scaleX = d3.scaleBand().range( [ 0, width ] ).domain( keys );

    // Column 1 has values, so take the domain and use it to create a
    // linear scale that maps the domain to an axis length.
    const domain = data.cols[ 1 ].domain.asArray();
    const scaleY = d3.scaleLinear().range( [ 0, height ] ).domain( domain );

    // ...styling is omitted from this sample...
    d3.select( "svg" ).selectAll( "rect" )
        .data( _info.data.rows )
        .join( "rect" )
```

```

        .attr( "x", d => scaleX( d.tuple( 0 ).key ) )
        .attr( "width", scaleX.bandwidth() )
        .attr( "height", d => scaleY( d.value( 1 ) ) );
    }

```

The example uses the key of every tuple as the unique domain value. In the data accessor function, `d => scaleX(d.tuple(0).key)` this key is used to determine the location of the data point on the x-scale.

The ISlot interface makes no difference between a categorical and a continuous slot. If the slot is continuous, the list of tuples is []. If the slot is categorical, the domain is empty (0, 0).

Data points

Each row in the data set is represented by a data point object. A data point references tuples (one for every categorical slot) and values (one for every continuous slot). If like in the previous example, bind data points to your d3 elements, you can use an accessor function to retrieve the tuple and value information for each data point.

The following example binds each row to a text element and uses an accessor function to retrieve the caption of column 0 as text content. If column 0 is a categorical column, the tuple caption is used. If column 0 is a continuous column, the formatted value is used.

```

d3.select( "svg" ).selectAll( "text" )
  .data( _info.data.rows )
  .join( "text" )
  .text( d => d.caption( 0 ) )

```

Decorations

Each data point and tuple can to get decorated. A decoration is an attribute that holds some meta information about the data point or tuple. Currently, two decoration types are supported: `selected` and `highlighted` (both booleans). If a data point is marked as `selected`, the visualization might want to render the data point slightly different, for instance by using a brighter color.

```

d3.select( "svg" ).selectAll( "rect" )
  .data( _info.data.rows )
  .join( "rect" )
  .attr( "x", d => scaleX( d.tuple( 0 ).key ) )
  .attr( "width", scaleX.bandwidth() )
  .attr( "height", d => scaleY( d.value( 1 ) ) )
  // Selected data points are rendered in the color red.
  .attr( "color", d => d.selected ? "red" : "steelblue" );

```

Alternatively, if the visualization has a `palette` property, you can use the `getFillColor` function of the palette determine the correct color for an element. For example:

```

<!-- vizdef.xml -->
<properties>
  <!-- Categorical palette property -->
  <palette name="color" type="cat" />
</properties>

```

```

protected update( _info: UpdateInfo ): void
{
    // ...
    const palette = _info.props.get( "color" );
    d3.select( "svg" ).selectAll( "rect" )
      .data( _info.data.rows )
      .join( "rect" )
      .attr( "x", d => scaleX( d.tuple( 0 ).key ) )
      .attr( "width", scaleX.bandwidth() )
      .attr( "height", d => scaleY( d.value( 1 ) ) )
      .attr( "color", d => palette.getFillColor( d ) );
}

```

If the palette is bound to a slot, then the color is based on the color of the tuple for that slot. If the palette is not bound to a slot, then the default palette color is used. The `getFillColor` method adjusts the color based on the selection of this and other data points.

There is also a `getOutlineColor` method on a palette that allows you to retrieve a color that can be used for drawing the outline of an element based on the highlight decoration.

The `getFillColor` and `getOutlineColor` are available on all palette types and on the `ContStops` object that is retrieved from a `ContPalette`. For more information, see [“Palettes” on page 23](#).

Handling null data

The update function in your visualization gets a data object passed as part of the `UpdateInfo` object. However, this data object can be `null`. This indicates that the visualization is responsible for clearing all data and show the visualization in an initial state.

```
protected update( _info: UpdateInfo ): void
{
    if ( _info.data === null )
        return this.clearVisualization( _info );

    // Perform rendering here...
}
```

Interactivity

Custom visualizations can use the interactivity capabilities of the hosting application.

Interactivity in this context means:

- **Tooltip:** Provide additional information when you hover over a data element.
- **Selections:** Apply an alternative style to data elements that are tagged as selected.
- **Highlights:** Highlight data elements that are under your mouse cursor.

Many of the interactivity functions can be implemented with little or no coding. If your visualization is based on d3, then `RenderBase` already has a default `hitTest` implementation that can be configured to return the correct data.

For selections and highlights, palettes have a `getFillColor` and `getOutlineColor` method that help you bind the right color to your visual elements.

Hit testing

Hit testing is the process of returning a data element based on a screen coordinate. The hosting application is responsible for calling the `hitTest` method when needed, and the visualization is responsible for returning the correct data element based on the specified coordinate.

The `hitTest` function can be called often (for example on every mouse move), so it is important to make sure that your function runs well.

The following example defines a class `MyData` that stores a reference to the visualization data and is bound to `svg` elements on the screen through the `mydata` field.

```
class MyData
{
    public DataPoint dp;
    // Other MyData members...
}

protected hitTest( _elem: Element | null, _client: Point, _viewport: Point ): DataPoint | null
{
    if ( !_elem || !_elem.hasOwnProperty( "mydata" ) )
        return null; // no element or element has no data
    return _elem.mydata.dp;
}
```

If your visualization is based on d3 and you bound your visualization data to the d3 model, then the default `RenderBase hitTest` function will already run everything for you:

```
protected update( _info: UpdateInfo )
{
    // Render a square for every data point.
    d3.select( "svg" ).selectAll( "rect" )
      .data( _info.data.rows ) // bind data
      .join( "rect" ) // create / update 'rect' elements
        .attr( "x", d => scaleX( d.value( X ) ) )
        .attr( "y", d => scaleY( d.value( Y ) ) )
        .attr( "width", d => scaleSize( d.value( SIZE ) ) )
        .attr( "height", d => scaleSize( d.value( SIZE ) ) );
}

// No hitTest override needed because RenderBase is able to figure out how
// d3 has bound the data to 'rect' elements.
```

Some d3 visualizations encapsulate your `DataPoint` or `Tuple` in another object. In that case, you need to override `hitTest` to return the data object:

```
protected hitTest( _elem: Element | null ): DataPoint | null
{
    const elem = d3.select<Element, any>( _elem );
    const data = elem.empty() ? null : elem.datum();
    return data && data.dp;
}
```

Tooltips

Tooltips are provided for you by the hosting application. To show a tooltip, the host first needs to find out which data element is rendered on a certain screen location. It uses the `hitTest` function to get this information. After that, it is up to the host to show the tooltip with the correct data and styling.

Once you implemented the `hitTest` function in `RenderBase`, or rely on the default implementation, you basically get tooltips for free. There is no need to write your own mouse handlers and pop ups to show tooltips. This means that your visualization has tooltips that are consistent with the host application.

Selection & highlighting

A user of the custom visualization can often click a data element to select it, or hover with the mouse cursor over it to highlight it. This selection and highlighting mechanism is implemented by the host and also relies on the `hitTest` method. After the host application determined the data elements that need to be selected or highlighted, the data elements are decorated with **selected** and / or **highlighted** flags and the **update** function of the visualization will be called. It is the responsibility of the update function to do apply the **selected** and **highlighted** flags to the rendered output.

To make visualizations that behave consistently with other visualizations in the hosting application, a `palette` property has a method with which you can retrieve the selection or highlight style of an element based on its **selected** and **highlighted** flags. In code, this might look as follows:

```
{
    d3.select( "svg" ).selectAll( "rect" )
      .data( _info.data.rows )
      .join( "rect" )
        .attr( "x", d => scaleX( d.value( X ) ) )
        .attr( "y", d => scaleY( d.value( Y ) ) )
        .attr( "width", d => scaleSize( d.value( SIZE ) ) )
        .attr( "height", d => scaleSize( d.value( SIZE ) ) )
        // Apply selection and highlight styling to the rect.
        .attr( "fill", d => palette.getFillColor( d ) )
        .attr( "stroke", d => palette.getOutlineColor( d ) );
}
```

Legends

The legend of a visualization can be used to show the user how various channels like color and size are used to represent data.

For example, a grouped bar visualization might want to show the group labels on the x-axis while it shows the colors and data labels of the individual group elements in a legend. Another example is a bubble visualization, where a legend can be used to show the size-range or color-range of the bubbles.

Legends are rendered outside your custom visualization. For a legend to render the correct information, the visualization needs to provide a data structure that tells the legend what it shows. This data structure is generated automatically depending on the channel information that is found in the slots of the [“Visualization definition”](#) on page 33.

Categorical legends

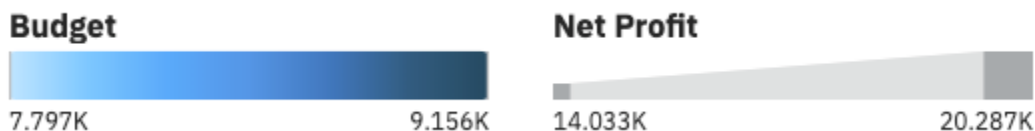
A categorical legend shows a list of swatches that each represent a tuple from your data set. Each swatch can have a colored shape and has a caption. The list of swatches is based on the list of tuples on the slot that is represented by the legend.



The legend can also have a title, as shown in the season example. The title is often derived from the data item that the legend represents.

Continuous legends

A continuous legend shows the range of a value. The range can be expressed as a size range or a color range. The `channel` property of a slot is used to distinguish between the two.



The images above show a color legend and a size legend. The title of these legends is, like the categorical legend, which is derived from the data item that the legend represents.

Enabling legends

In general the only thing you need to do as a bundle developer is provide the [“Visualization definition”](#) on page 33 of your visualization. The `Vizdef.xml` file needs a few entries that allow `RenderBase` to determine if and how a legend should be created. If your [“Visualization definition”](#) on page 33 has the required entries, then `RenderBase` creates the right data structures and passes them to the host application.

Legends are tied to the slot entries in your [“Visualization definition”](#) on page 33. Each slot in the `vizdef` has an optional attribute called `channel`. The channel determines the encoding type of a legend and can have the values `color` and `size`. The `color` type can be used for both categorical (colored swatches) and continuous legends (color range). The `size` type is used for continuous legends only (size range). If a slot has one of the known channel types, a legend for that slot is rendered.

```
<slot name="series" type="cat" channel="color" />
<slot name="values" type="cont" channel="size,color" />
```

For slots with the `color` channel, you define a `palette` property. This allows the host application to assign a custom color scheme to the visualization and legend items. If no palette is available for the slot, then no legend is rendered for that slot.

Note: In the following example, the palette does not have an explicit type attribute. If the palette is linked to a slot, the palette type is determined by the type of linked slot.

```
<slots>
  <slot name="categories" type="cat" />
  <!-- Series tuples will appear as color swatches in legend -->
  <slot name="series" type="cat" channel="color" />

  <!-- Values will appear both as size and color range in legend -->
  <slot name="values" type="cont" channel="size,color" />
</slots>

<properties>
  <!-- Categorical palette linked to 'series' slot -->
  <palette name="series_colors" slot="series" />
  <!-- Continuous palette linked to 'values' slot -->
  <palette name="values_colors" slot="values" />
</properties>
```

Modifying legends

In some specific situations, you want to change the information that is rendered in a legend. For example, you might decide that based on a certain condition a legend might be visible or invisible. RenderBase has a method that is called `updateLegend` that can be overridden to provide such behavior. You can either modify the automatically created legend, or provide a new one.

```
protected updateLegend( _data: DataSet ) : Encoding[]
{
  // Only show legend for 'heat' or 'categories', not both.
  const filterSlot = _data.cols[ HEAT ].mapped ? "heat" : "categories";

  // Only show a legend that matches `filterSlot`.
  return super.updateLegend( _data ).filter( _e => _e.slot === filterSlot );
}
```

The following example shows how to have the items in a categorical legend in reverse order and the legend title in uppercase:

```
public updateLegend( _data: DataSet ) : Encoding[]
{
  // Create default legend information.
  const legend = super.updateLegend( _data );

  // If legend is categorical, make caption upper case and reverse items.
  legend.forEach( _e =>
  {
    if ( _e instanceof CatEncoding )
    {
      _e.caption = _e.caption.toUpperCase();
      _e.entries.reverse();
    }
  } );
  return legend;
}
```

The API for creating and modifying legend needs the following import:

```
import { Encoding, CatEncoding, ContEncoding } from "@businessanalytics/customvis-lib";
```

Properties

A visualization can display one or more properties to allow the user of the visualization to control certain aspect of the visualization.

The [“Visualization definition”](#) on page 33 holds the property definitions for a visualization. Reading property values and active state can be done through the `properties` attribute of `Renderbase`.

Property locking

During rendering, all properties can be considered in a locked state. This means that the host application can still modify property values, but these new values will not be available to the visualization until the next render operation. This ensures that in your visualization you can rely on a property value during for instance a long running animation.

If the visualization needs to have synchronous access to properties, use `properties.peek`. This returns the value of the property that was set by a hosting application without caring about locking.

Another way to get synchronous access to a property value is to override the `updateProperty` on page 27 method. This method is called directly when the property value changes and no locking is applied. A visualization typically uses the `updateProperty` method only if it wants to update the active state of properties based on the value of other properties.

Property values

Each property has a value that is based on one of the predefined property types:

- `number`
- `string`
- `boolean`
- `font`
- `color`
- `palette`
- `length`

A `RenderBase` subclass can access the properties either through `this.properties` or through the `UpdateInfo` object that is passed in the `updateProperty` on page 27 method.

If you want to retrieve the value of a property, you can call the `get` method on the `properties` object. The `get` function returns the property value as either a native type (for number, string and boolean) or an object (for color, font, length, and palette).

```
javascript
var ticks = this.properties.get( "numTicks" );
assert( typeof ticks === "number" );

var title = this.properties.get( "title" );
assert( typeof title === "string" );

var color = this.properties.get( "backgroundColor" );
assert( typeof color === "string" );
```

Dirty properties

When a hosting application changes the value of a property, then that property becomes dirty. The property will remain dirty until after the next render operation. This gives a visualization the opportunity to process dirty properties during update. The dirty state of a property can be retrieved by calling `properties.getDirty()`. In some cases, rendering performance can be optimized by only processing properties that changed since the last update operation.

Active & inactive properties

A property can be marked as active or inactive. The active state of a property is an indication for a host whether it should show the property in the UI as enabled or disabled.

The visualization can control the active state of a property through the `properties.setActive()` method, or retrieve the active state of a property using `properties.isActive()`. Typically, properties should be made active or disabled in the `updateProperty` method of `Renderbase`. This method is called

at initialization time of the visualization and each time a property changes its value. For more information and a code sample, see [“updateProperty” on page 27](#).

Property Groups

Properties can be organized in groups and subgroups. This allows the hosting application to show this group organization in the UI. Any number of groups and subgroups is possible, but it is recommended to stick with a limited grouping depth.

There are two built-in property groups that are called `visualization` and `text`. These groups have a special meaning to the hosting application. If both these groups are available, then a set of automatically generated legend properties, are added to these two groups. If either of these groups is not available, then a new group that is called `legend` is generated and used for the automatically generated legend properties.

The only party that is interested in property groups is the hosting application. Therefore, the `RenderBase` implementation does not have access to property groups. The only thing the visualization developer does is define the property groups in the [“Visualization definition” on page 33](#).

Palettes

A palette is a special type of property that you can use to determine a color based on a tuple or a value.

Palettes come in two different types: categorical palettes and continuous palettes. A categorical palette is used when you want to assign a distinct color to each tuple in a slot. A continuous palette is used if you want to use a color to represent a numerical value.

Categorical palettes

A categorical palette lets you retrieve a color based on a tuple. You define a categorical palette for the visualization in the [“Visualization definition” on page 33](#):

```
<properties>
  <palette name="colors" type="cat" slot="series" />
</properties>
```

When you define a palette property in your `vizdef.xml` file, you can access the palette like any other property:

```
protected update( _info: UpdateInfo ): void
{
  // Retrieve the palette and cast to CatPalette.
  const palette = _info.props.get( "colors" ) as CatPalette;

  // Get the series tuples from column 2 in the data set.
  const seriesTuples = _info.cols[ 2 ].tuples;

  // Retrieve a color for each tuple in the list.
  const colors = seriesTuples.map( _tuple => palette.getColor( _tuple ) );
}
```

A common example of using categorical palettes is when you want to set the color of an element that is bound to data. For example, the following code shows all tuples of the series slot in a list and assigns a different color to each element:

```
protected update( _info: UpdateInfo ): void
{
  // Retrieve the tuples from slot 0. If the data was cleared, then
  // use an empty list to indicate there are no tuples.
  const tuples = _info.data ? _info.data.cols[ 0 ].tuples : [];

  // Create a scale based on visualization height and tuple keys.
  const scale = d3.scaleBand()
    .range( [ 0, _info.node.clientHeight ] )
    .domain( tuples.map( _t => _t.key ) );

  const palette = _info.props.get( "color" ) as CatPalette;
```

```

const svg = d3.select( _info.node );

// For each tuple, create an svg text element and assign a color
// from the palette.
svg.selectAll( "text" ).data( tuples, ( t: any ) => t.key )
  .join( "text" )
  .attr( "y", t => scale( t.key ) )
  .attr( "fill", t => palette.getFillColor( t ) )
  .text( t => t.caption );
}

```

Notice the use of `getFillColor` in the code above. The difference between `getColor` and `getFillColor` is that the latter will look at the selected state of the tuple and adjust the color to make it lighter if the tuple should appear as deselected. Since you are binding actual tuples to our text elements, you get hit testing for free. This means that the user can click a tuple to select it and see the color of the other tuples change automatically.

Continuous palettes

A continuous palette can return a color based on a numerical value. You define a palette for a visualization in the “Visualization definition” on page 33:

```

<properties>
  <palette name="colors" type="cont" slot="values" />
</properties>

```

Retrieving colors from a continuous palette is a two-step process:

1. Get the `ColorStops` of the palette by providing a slot instance. The slot instance should match the slot name that was specified in the vizdef.
2. On the `ColorStops` instance, call `getColor` passing a numerical value.

Both the `ContPalette` and the `ColorStops` class have a `getFillColor` and `getOutlineColor` method that lets you retrieve a color based on a data point. This can be useful if your visualization needs to render elements that should reflect the selection or highlight state of the data. An example of a scatter chart visualization:

```

protected create( _node: HTMLElement ): void
{
  // Create an svg node that resizes to the visualization.
  d3.select( _node ).append( "svg" ).attr( "width", "100%" ).attr( "height", "100%" );
}

protected update( _info: UpdateInfo ): void
{
  const data = _info.data;

  // Determine the size of the visualization
  const width = _info.node.clientWidth;
  const height = _info.node.clientHeight;

  // Determine the value domain for x and y.
  const domainX = data ? data.cols[ 1 ].domain.asArray() : [ 0, 0 ];
  const domainY = data ? data.cols[ 2 ].domain.asArray() : [ 0, 0 ];

  // Create linear scales for positioning the data points.
  const scaleX = d3.scaleLinear().range( [ 0, width ] ).domain( domainX );
  const scaleY = d3.scaleLinear().range( [ height, 0 ] ).domain( domainY );

  const palette = _info.props.get( "color" ) as ContPalette;

  // Render a circle with fixed radius for each data point.
  const svg = d3.select( _info.node ).select( "svg" );
  svg.selectAll( "circle" ).data( data ? data.rows : [] )
    .join( "circle" )
    .attr( "cx", d => scaleX( Number( d.value( 1 ) ) ) )
    .attr( "cy", d => scaleY( Number( d.value( 2 ) ) ) )
    .attr( "r", 12 )
    .attr( "stroke-width", 2 )
    // Stroke and fill color are based on selected and highlighted state.
    .attr( "stroke", d => palette.getOutlineColor( d ) )

```

```

        .attr( "fill", d => palette.getFillColor( d ) );
    }

```

For completeness of the example, this is the `vizdef.xml` file that was used:

```

<?xml version="1.0" encoding="UTF-8"?>
<visualizationDefinition version="3.1" xmlns="http://www.ibm.com/xmlns/prod/ba/vipr/vizBundle/
vizdef.3">
  <slots>
    <slot name="categories" type="cat" optional="false" />
    <slot name="x" type="cont" optional="false" />
    <slot name="y" type="cont" optional="false" />
    <slot name="color" type="cont" optional="true" channel="color" />
  </slots>

  <dataSets>
    <dataSet name="data">
      <ref slot="categories" />
      <ref slot="x" />
      <ref slot="y" />
      <ref slot="color" />
    </dataSet>
  </dataSets>

  <properties>
    <group name="general">
      <palette name="color" type="cont" slot="color" />
    </group>
  </properties>

  <capabilities>
    <decorations>
      <decoration name="selected" type="boolean" target="datapoint, tuple" />
      <decoration name="hasSelection" type="boolean" target="dataset" />
      <decoration name="highlighted" type="boolean" target="datapoint, tuple" />
    </decorations>
  </capabilities>
</visualizationDefinition>

```

Palettes and slots

In the above examples, the palette definition in the `vizdef.xml` file was always linked to a slot. This allows the palette to determine the color of a data point and it allows `RenderBase` to generate a legend for that slot based on the palette.

In some situations, you don't know in advance to which slot a palette are linked. For instance, if you have two categorical slots and the palette might be linked to one of the two slots. In that case, you can omit the `slot` attribute in the palette definition and override the `RenderBase.getSlotForPalette` method. This method takes a palette name and returns its corresponding slot name that you want the palette to be linked to. The method is called only for palettes that have no slot that is associated with them. An example:

```

protected getSlotForPalette( _data: DataSet, _palette: string ): string
{
    // Link the 'catColors' palette to the series slot if that slot is
    // mapped. Otherwise link it to the categories slot.
    if ( _palette === "catColors" )
        return _data.cols[ SERIES ].mapped ? "series" : "categories";
    return null;
}

```

Customizing your visualization

`RenderBase` offers a number of customizable aspects of a visualization.

For example, if your visualization has a certain limit on the number of data points it supports, you can set the `RenderBase.meta.dataLimit` variable to that limit. The `meta` attribute is an instance of the `MetaInfo` class. This class holds all aspects that you can customize.

MetaInfo

The MetaInfo class holds the aspects of a visualization that you can customize. Typically, you set these aspects during the initialization of the visualization (the `create` function):

```
protected create( _node: HTMLElement ): void
{
    // Categorical legends should use a circle symbol.
    this.meta.legendShape = "circle";

    // Limit the categories slot to 250 elements.
    this.meta.slotLimits.set( "categories", 250 );

    // Limit the number of data points to 1000.
    this.meta.dataLimit = 1000;
}
```

Here is an overview of the supported aspects:

- **legendShape** - Defines the shape of elements in a categorical (swatch) legend. Possible values are `circle`, `rectangle`, `square`, `triangle`, `cross`, `donut`, `line`, `diamond`, `star`, `triangle-down`, `triangle-up`, `triangle-left` or `triangle-right`. It can also be `null` to indicate that the host should use a default shape or the string `none` to indicate that no shape should be displayed.
- **slotLimits** - Slot limits define the number of supported tuples in a slot. It is implemented as a map from slot name to limit value. If you don't want to limit the slot, remove the slot name from the map or set its value to `-1`.
- **dataLimit** - Defines the maximum number of data points the visualizations supports. Set to `-1` to indicate that there is no limit.

RenderBase

RenderBase is the base class for all custom visualization implementations.

It provides the visualization developer with some basic functionality like

- [“Rendering” on page 26](#)
- [“Properties” on page 27](#)
- [“Hit testing” on page 28](#)
- [“nls” on page 28 resources](#)
- [“meta” on page 28 information](#)

Each of these functions is described in more detail in the following sections.

Rendering

Rendering your visualization is done through the `create` and `update` methods of `RenderBase`. These methods implement the initialization and update phase of a render cycle. For more information, see [“Rendering” on page 13](#). You can override these methods to provide your own render initialization and updates. Overriding these methods is optional, if you do not override them, a default implementation is provided in `RenderBase`.

create()

syntax:

```
create( _node: HTMLElement ): Element | void | Promise<Element | void>
```

The `create` method is called exactly once, when the visualization is created. You basically put all your initialization code here. Or you set up your drawing context (maybe an `svg` node with some fixed child elements or groups). The return value becomes the root element of your visualization and is passed to you in the `update` method. If you do not return a value from this function, or choose not to implement it at all, then the passed `_node` becomes the root element for your visualization.

update()

syntax:

```
update( _info: UpdateInfo ): void | Promise<void>
```

The update method is called each time that the visualization needs to be redrawn. Reasons for redrawing the visualization are new data, modified properties, or a modified visualization size. A parameter holding UpdateInfo is passed to the update method, allowing you to access the latest data, properties and a set of flags that tell you the reason for the update call.

Properties

Your visualization will probably expose one or more properties that allow the user to control specific behaviors of the visualization. RenderBase gives you access to these properties using the `properties` attribute. In addition to retrieving property values, RenderBase also lets you monitor changes to properties. This is useful if you need to activate or deactivate properties based on the values of other properties. For instance, a bar chart might have a `stacked` and a `stackPercent` property. If `stacked` is not set, then `stackPercent` might be disabled.

syntax:

```
properties: Properties
```

This attribute holds the properties that are available in the visualization. You can access the value of a property by calling `properties.get("propname")`. Getting and setting the active state of a property can be done by calling `isActive` and `setActive` on the properties attribute respectively. The properties are also passed to you in the UpdateInfo object. For more information, see [“update\(\)” on page 27](#).

updateProperty

syntax:

```
updateProperty( _name: string, _value: any ): void
```

Called immediately if the host assigns a new value to a property. You can override this method to process property changes. For instance, they can set the active state of properties based on the value of another property.

```
protected updateProperty( _name: string, _value: any ): void
{
    if ( _name === "stacked" )
        this.properties.setActive( "stackedPercent", _value );
}
```

The updateProperty method is also called at initialization time for each property in the visualization. You can set the correct initial active state for all properties.

If you want to access the value of a property in this method, use `properties.peek` instead of `properties.value`. The former gives synchronous access to the property value meaning it is not influenced by locking. For more information, see [“Property locking” on page 22](#).

Localization

If your visualization needs to expose translated text or messages, then you can set up an `nls` directory structure and add your text to the `resources.js` file in the root of the `nls` directories. You can create sub directories and add translated `resources.js` files for each language you want to support. When your visualization is loaded, your text resources are loaded and made available in your RenderBase subclass through the `nls` attribute.

The RequireJS `nls` plugin is used internally to load and process `nls` files.

nls

syntax:

```
nls( _name: string ): string
```

Returns a translated string given a unique name.

```
protected update( _info: UpdateInfo ): void
{
    if ( _info.data === null )
    {
        _info.node.textContent = this.nls( "no_data" );
        return;
    }
}
```

Hit testing

Your visualization can provide custom hit testing functions, that allow the hosting application to show tooltips, selections and highlights. A default hit testing function is already provided for you in `RenderBase`. This function checks for the existence of d3 data binding and use that if available.

hitTest

syntax:

```
hitTest( _elem: Element | null, _client: Point, _viewport: Point ): DataPoint | Tuple | null
```

Returns a data element based on an element and coordinate. The coordinate is passed both as client point (relative to browser window) and viewport point (relative to visualization). You are supposed to return a data object that was passed to you in the [“update\(\)” on page 27](#) method. For more information, see [“Interactivity” on page 18](#).

meta

syntax:

```
meta: MetaInfo
```

Currently, the following fields are available:

- `legendShape` - Defines the shape of elements in a categorical legend. The default value is `null`, indicating that the host can determine a default shape.
- `slotLimits` - This is a map from slot name to a number of supported tuples in that slot. The information in this map is passed to the host as an indication of how much data can be handled by the visualization. Slots that are not in this map, or have a limit of `-1`, are not limited to a maximum number of tuples.
- `dataLimit` - The maximum number of data points that the visualization supports, or `-1` if there is no limit.

Chapter 5. Customvis API reference

Customvis api reference

An API reference for Customvis-lib classes and methods is available.

Check out the [API reference](#) for detailed information about all the classes and methods that are found in customvis-lib.

Chapter 6. Customvis frequently asked question

FAQ custom visualizations

An FAQ for custom visualizations is available.

Check out the [FAQ](#) for frequently asked questions on custom visualizations.

Chapter 7. Visualization definition

Visualization definition

The VizDef is an XML file that contains the definition of the visualization that you are building.

The VizDef file tells a hosting application something about the visualization such as the way data is organized and the properties of the visualization.

Every custom visualization that you create must have a `vizdef.xml` file in the root directory. If you create a new visualization with `customvis start`, then an initial `vizdef.xml` is created for you.

Slots

Slots define the aspects in the visualization that can hold your data. For example, in a bar visualization you might have two slots, one for the categorical tuples and one for values. A slot must at least have a name (for internal reference) and a type (categorical or continuous). A typical slots definition might look as follows:

```
<slots>
  <slot name="categories" caption="Categories" type="cat" optional="false" />
  <slot name="values" caption="Values" type="cont" optional="false" />
  <slot name="series" caption="Color" type="cat" optional="true" channel="color" />
</slots>
```

The name and type attributes are mandatory and hold the internal name and type of the slot.

The caption attribute contains the name that can be used in an application's UI. You can either specify a literal string here, or a reference to a localized string such as `$nls.categories`. For more information, see [“Localization” on page 35](#).

The optional attribute indicates whether the visualization is able to render anything meaningful without any data mapped to that slot. For instance, in the previous example the `series` slot is marked optional. This means that if the user does not map data into this slot, the visualization renders only one default series.

Finally, the channel attribute indicates that the hosting application can use a specific encoding to represent a legend for that slot. Possible values for channel are `color` and `size`. For more information, see [“Legends” on page 20](#).

DataSets

A DataSet defines the collection of slots that form the edges of your data. Currently, only one data set is supported by RenderBase, so your typical DataSets entry in `VizDef.xml` looks as follows:

```
<dataSets>
  <dataSet name="data">
    <ref slot="categories" />
    <ref slot="series" />
    <ref slot="values" />
  </dataSet>
</dataSets>
```

Each slot that you defined in the slots section must be listed in a data set.

Properties

A visualization can expose properties that allow the user of the visualization to control various aspects of the behavior of the visualization. Each property is defined at least by a name and a type. Depending on the

property type, other attributes can be applied. A visualization can organize properties into logical groups. It is up to the host application if it wants to show the property grouping in the UI. An example:

```
<properties>
  <group name="general">
    <!-- Boolean property with default value = true -->
    <boolean name="labels.visible" defaultValue="true" />
    <!-- Positive numeric value; allow null to indicate 'automatic' tick count -->
    <number name="ticks.count" minValue="0" allowNull="true" />
    <!-- String property; allow null to indicate 'automatic' title -->
    <string name="title.text" allowNull="true" />
    <!-- Font property; host receives a Font structure to change individual font aspects -->
    <font name="title.font" defaultValue="10px arial" />
  </group>
</properties>
```

Note: Using `allowNull` requires the visualization code to correctly handle the null case. Setting this flag means that null becomes a valid value of the property. If `allowNull` is `false`, then the visualization can assume that the value of the property will never be null and there is no need for valid checks in the code.

The following attributes can be applied to a property:

- `name`: The name of the property. Used to refer to the property in the rendering service code.
- `caption`: The label that appears in the UI for the property. Can be a literal string, or a localization ID: a string prefixed with `$nls..`
- `defaultValue`: The default value that a property gets when it is initialized. If the `defaultValue` is not specified and `allowNull` is `true`, then the default value will be `null`.
- `allowNull`: Flag that indicates whether a host application is allowed to set a `null` value. Often used to indicate automatic values. For example, for a scale.
- `minValue`: Only for number properties, defines the minimum allowed value.
- `maxValue`: Only for number properties, defines the maximum allowed value.
- `slot`: Only for palette properties, links a palette to a slot. The palette type and slot type must match. If you omit the palette type, the slot type is taken.
- `type`: Only for palette properties, defines the type of the palette: `cat` or `cont`.

The following property types are available:

- `string`: Holds a textual value.
- `number`: Holds a numerical value. Minimum and maximum value can optionally be specified through the `minValue` and `maxValue` attributes.
- `boolean`: Holds a Boolean value.
- `enum`: Holds a static list of values from which a user of the visualization can choose one. The list of possible values can be specified in `<possibleValue>` elements. See the example code further down.
- `color`: Holds a color value. The `defaultValue` of a color can be a cascading stylesheet (css) representation of a color, including named colors ('red', 'green').
- `font`: Holds a css font definition.
- `palette`: holds a palette reference. A palette is defined by a name and possibly a `defaultValue`. If no `defaultValue` is specified, a built-in palette is assigned. Default values for a palette are defined by a semi-color separated list of colors.

```
<!-- font property example -->
<font name="axis.title" defaultValue="bold 16px Arial" />

<!-- enum property example -->
<enum name="shape" caption="Shape">
  <possibleValue caption="Circle">circle</possibleValue>
  <possibleValue caption="Square">square</possibleValue>
</enum>

<!-- palette property example -->
```

```
<palette name="color" type="cat" defaultValue="red;green;blue" />
<palette name="range" type="cont" defaultValue="red 0%;green 50%;blue 100%" />
```

Capabilities

If your visualization must be able to respond to highlights and selections, then specify the following section in your `vizdef.xml` file:

```
<capabilities>
  <decorations>
    <decoration name="selected" type="boolean" target="datapoint" />
    <decoration name="hasSelection" type="boolean" target="dataset" />
    <decoration name="highlighted" type="boolean" target="datapoint" />
  </decorations>
</capabilities>
```

This section tells the host application that data points can be selected and highlighted. If you do not want your visualization to use the built-in support for highlighting and selection, then you can choose not to include this section in your `vizdef.xml` file.

Localization

The `vizdef.xml` file holds information that a host application can use to render a user interface. For example, the `vizdef.xml` file holds all properties that a visualization exposes and a host application can read these properties and show them to the end in a properties panel.

To allow localization of property names and descriptions, a special notation was introduced that maps the value of an attribute to an index of a resource table. For example, look at the following property definition:

```
<boolean name="labels.visible" caption="$nls.labels.visible" defaultValue="true" />
```

The caption is user-facing and can be localized. Instead of hardcoding a value in this attribute, use an ID: `$nls.labels.visible`. A resource table is then used to look up this ID and find the actual caption in the correct language.

The resource table that contains these localized strings is formatted according to the `requirejs i18n.js` plug-in. For `customvis` to correctly find and process your localizations, you must have a file and folder structure similar to the following example:

```
./nls/de/Resources.js
./nls/nl/Resources.js
./nls/Resources.js
```

This example contains a base, or root, language in `./nls` and German and Dutch translations in `./nls/de` and `./nls/nl`. The root `Resources.js` file looks as follows:

```
define(
{
  "root":
  {
    "labels.visible" : "Show Labels",
    "labels.font"    : "Label Font"
  },
  "nl": true,
  "de": true
} );
```

Each of the language-specific files has the following content:

```
define(
{
  "labels.visible" : "Toon Labels",
  "labels.font"    : "Label Lettertype"
} );
```

The resource entries found here are not limited to strings used in the VizDef. You can also include entries that are used in your rendering code. For more information, see [“Localization” on page 27](#).

For more information on REQUIREJS API, see the [requirejs documentation](#).

Chapter 8. Tutorial

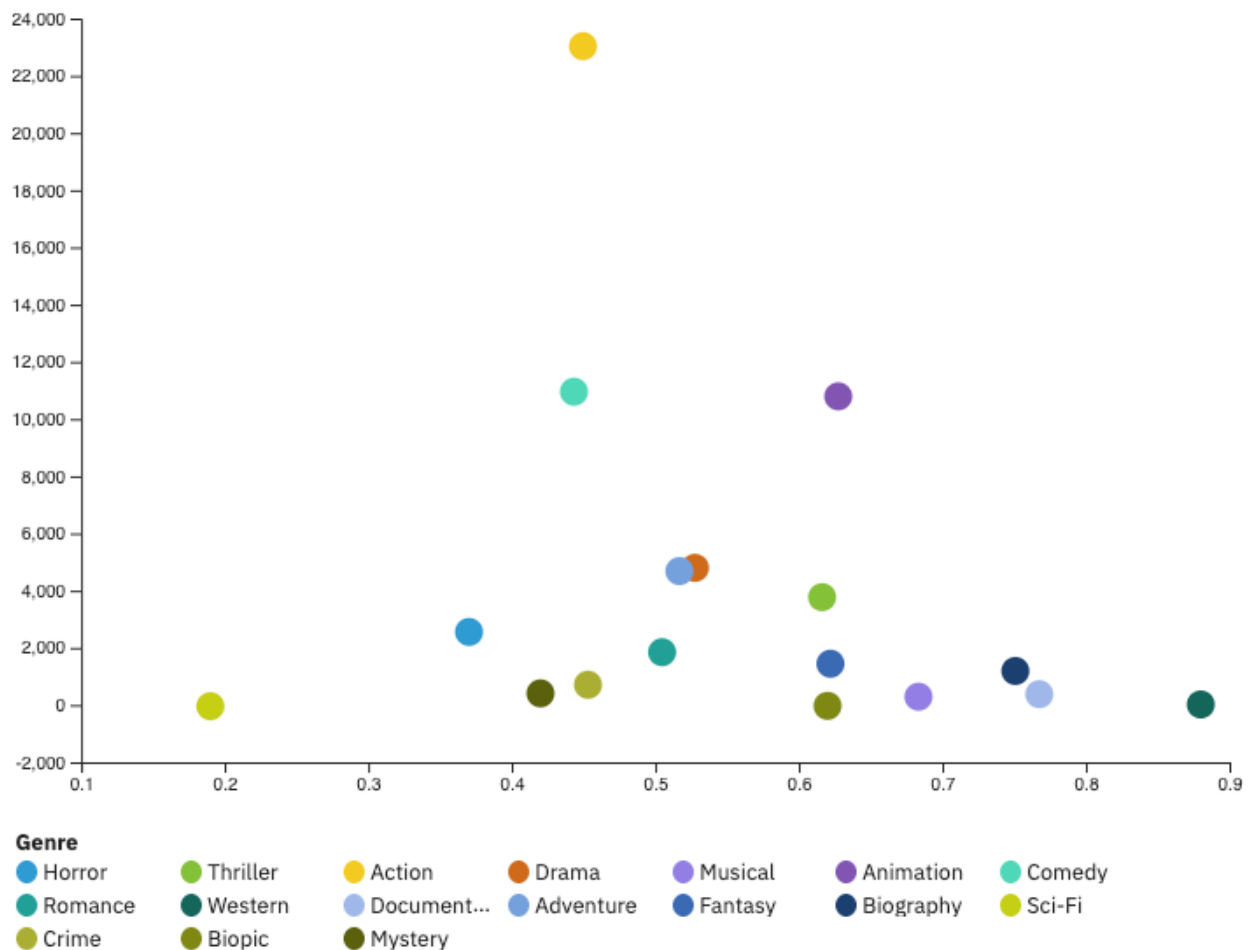
Custom visualizations - tutorial

In this tutorial you learn how to create a custom visualization, from scratch, using the customvis command line interface (CLI) tool.

Before you start

1. Make sure that you have setup your development environment and installed the custom visualizations CLI tools as explained in [“Setting up the development environment”](#) on page 3.
2. This tutorial assumes a basic knowledge about d3. It makes following the steps in this document much easier and allows you to make your own modifications after you have completed the tutorial.
3. Basic knowledge about TypeScript is recommended since the samples in this tutorial are TypeScript based.
4. There is [API reference](#) documentation available on all classes and functions in customvis-lib.

At the end of this tutorial you have created a visualization that looks as follows:



The visualization has three data slots and a customizable palette. In addition, the visualization supports tooltips, selection, highlighting, and shows a product-based legend when used in IBM Cognos Analytics dashboard or reporting.

Part 1 will setup the data model and implement initialization and rendering code. At the end of this part you will have a working, data bound, scatter visualization with support for highlights, selections and tooltips.

Part 2 will be adding more properties to the visualization, like a point shape, title, and maximum values for the x- and y axis. You will use different property types and see how to use the 'defaultValue' and 'allowNull' attributes of a property.

Part 3 will further improve the visualization by adapting the legend to the scatter symbols, setting the active state of the title property based on the value of the showTitle property, and providing property captions.

Part 4 is the last part of this tutorial. It shows how to define optional slots, change the legend order, and provide localization. Localization allows you to provide translations in various languages for the user interface elements, for example slot- and property captions, of your visualization.

At the end of each part there is a link to the source code used in that part. Every part builds upon the previous one, so if you want to skip a part you can download the source code of the preceding part and continue with that.

- [Download](#) the complete source code for the tutorial.

Part 1: Creating a custom visualization

In this first part, you will create a new visualization using one of the built-in templates and add a data definition. You will also provide an initial implementation that renders an x-axis, y-axis and, scatter points for each row in the provided data. At the end of this part you have a working sample of a (very basic) scatter visualization.

Before you start

Make sure that you have setup your development environment and installed the custom visualizations CLI tools as explained in [“Setting up the development environment”](#) on page 3.

Step 1: Creating the visualization

Open a command terminal, navigate to the directory where you create your visualizations, and create a new visualization by running the following commands:

Line	Code
1	customvis create MyScatter
2	cd MyScatter
3	customvis start

```
customvis create MyScatter
cd MyScatter
customvis start
```

The first command creates a template visualization in a new directory named MyScatter. The second command takes you to that directory where the following files are created:

- package.json

This is where you can change the name and icon of the bundle as it appears in IBM Cognos Analytics.

```
"meta": {
  "name": "Fred",
  "icon": "bundle.svg"
```

A name and icon are already filled in for you, but you can choose a different name or different icon file format (.png, .jpg).

- vizdef.xml

[“Visualization definition” on page 33](#) file describing aspects of your visualization like slots, data organization, and external properties. In step 2 you will edit this file and add slots and properties for the new visualization.

- `externals.d.ts`

If your visualization needs to reference external libraries, you can add them here. In this example however you do not need to make changes to this file.

- `customvis-lib.tgz`

This file contains the APIs that you will use to develop your visualization, including a base class called [“RenderBase” on page 26](#) from which you will derive your visualization class.

- `renderer/Main.ts`

The main implementation file. The `.ts` extension tells you that this is a typescript file. In step 3 you will start adding some code in this file.

The third and last command starts the visualization and makes it available for use in IBM Cognos Analytics. The visualization does not have much functionality yet, so in the next steps you will start making changes to `vizdef.xml` file and `Main.ts` file and add some features. All changes you make will be picked up continuously by the CLI tool and compiled into a working visualization. More on that in step 4.

Note: After running `customvis start` you will see that a few extra files and folders are created. This is build output, needed to run the visualization. If you want to start with a clean folder again, run `customvis clean`. This will remove these generated files.

Step 2: Edit `vizdef.xml`

The two most important files for your visualization are `vizdef.xml` and `Main.ts`. In this step we will focus on [“Visualization definition” on page 33](#). The `vizdef` is a formal xml definition of the characteristics of your visualization. This definition can be used by a host application to, for example, create a user interface for managing properties, implement highlighting and selections, and perform efficient data queries.

You start by editing the already created `vizdef.xml` in your text editor. You will see the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<visualizationDefinition version="3.1" xmlns="http://www.ibm.com/xmlns/prod/ba/vipr/vizBundle/
vizdef.3">
</visualizationDefinition>
```

For the scatter visualization, you will need to add slots, a data set, and capabilities.

Slots and data sets

The first thing to add is the slots and a `DataSet` that define the data used within the visualization. The scatter visualization will have three slots:

- A categorical slot that holds a tuple for every point we are going to plot.
- A continuous slot that holds the value for the x-position of the point.
- A continuous slot that holds the value for the y-position of the point.

For more information about slots and data sets, see [“Data model” on page 16](#).

In this example, all slots are considered mandatory. If there is no data for any of the slots, the visualization will not render. Given these requirements, edit `vizdef.xml` so it looks as follows:

Line	Code
1	<?xml version="1.0" encoding="UTF-8"?>
2	<visualizationDefinition version="3.1" xmlns="http://www.ibm.com/xmlns/prod/ba/vipr/vizBundle/vizdef.3">
3	<slots>
4	<slot name="categories" type="cat" optional="false" channel="color" />
5	<slot name="xpos" type="cont" optional="false" />
6	<slot name="ypos" type="cont" optional="false" />
7	</slots>
8	
9	<dataSets>
10	<dataSet>
11	<ref slot="categories" />
12	<ref slot="xpos" />
13	<ref slot="ypos" />
14	</dataSet>
15	</dataSets>
16	</visualizationDefinition>

It is important that you add a single dataSet entry that references each slot.

The channel="color" attribute in the categories slot on line 4 is needed to draw an Cognos Analytics product-based legend based on the data of that slot. See [“Legends” on page 20](#) for more information.

Properties

Your visualization can expose public properties that allow the authors to modify within Cognos Analytics how the visualization looks. You can think of font properties, colors, margins etc. For now we you will add only one property: a color palette.

A palette can be used by a visualization to determine the colors used for the data elements. For the scatter visualization, we want to show a different color for each rendered circle. Here's the xml that is needed in vizdef:

```

...
<properties>
  <palette name="color" slot="categories" />
</properties>
</visualizationDefinition>

```

Selection and highlighting capabilities

If you want the visualization data elements to indicate selections and highlights, you need to add a few capabilities to the vizdef. These capabilities ensure that the host application provides us with information about the highlight and selection status of elements in the data:

```

...
<capabilities>
  <decorations>
    <decoration name="selected" type="boolean" target="datapoint" />
    <decoration name="hasSelection" type="boolean" target="dataset" />
    <decoration name="highlighted" type="boolean" target="datapoint" />
  </decorations>
</capabilities>
</visualizationDefinition>

```

The target field indicates that we are only interested in data point decorations. In other visualizations you might also specify tuple in addition to datapoint (comma-separated). But for now we will stick with the definition shown above. More information about tuples and data points can be found in the [“Data model” on page 16](#) documentation.

Step 3: Start coding in Main.ts

Now that the `vizdef.xml` file is ready, you can start coding the behavior of the visualization. To start, open the `Main.ts` file, located in the `renderer` directory. The file looks as follows:

```
import { RenderBase } from "@businessanalytics/customvis-lib";

export default class extends RenderBase
{
  protected create( _node: HTMLElement ): HTMLElement
  {
    _node.textContent = "Basic VizBundle";
    return _node;
  }
}
```

Next, you will learn how to add a d3 reference, perform initial setup of the visualization and handle updates.

Reference d3

Since the visualization will be based on d3, you need to import the d3 types. You will also be handling render updates, so `UpdateInfo` is also required. Finally, you define a few constants that are useful later on for referencing slots:

```
import { RenderBase, UpdateInfo } from "@businessanalytics/customvis-lib";
import * as d3 from "d3";

const CAT = 0, XPOS = 1, YPOS = 2; // slot indices
...
```

Perform initialization

Every visualization has a `create` method that is called one time, during the creation of the visualization. You can add code that performs initial setup here, so let's create an `svg` canvas that we will use to render on. Change the existing `create` function as follows:

```
protected create( _node: HTMLElement ): Element
{
  // Create an svg canvas that sizes to its parent.
  const svg = d3.select( _node ).append( "svg" )
    .attr( "width", "100%" )
    .attr( "height", "100%" );

  // Create groups for axes and elements.
  const chart = svg.append( "g" ).attr( "class", "chart" );
  chart.append( "g" ).attr( "class", "xaxis data" );
  chart.append( "g" ).attr( "class", "yaxis data" );
  chart.append( "g" ).attr( "class", "elem data" );

  // Return the svg node as the visualization root node.
  return svg.node();
}
```

Note: The return value of the `create` function has changed from `HTMLElement` to `Element`!

The `create` function has one parameter: `_node`. This is the HTML element that is the container element for your visualization. You render everything either directly on this node or on children of this node.

The last command `return svg.node();` makes the `svg` node the root node of the visualization. The root node is passed to the update function that you will implement in the next step.

Handle updates

The majority of the rendering is done in the update function. This function will be called in case:

- New data arrives.
- Property values have changed.
- The size of the visualization has changed.
- Selections or highlights have changed.

Here is the complete code for update. You can paste it directly after the create function that you just modified. The majority of the code is d3-related. Detailed explanations follows below.

Line	Code
1	protected update(_info: UpdateInfo): void
2	{
3	// Get data, properties and svg node.
4	const data = _info.data;
5	const props = _info.props;
6	const svg = d3.select(_info.node);
7	
8	// If there is no data, remove all axes and elements.
9	if (!data)
10	{
11	svg.selectAll(".data>*").remove(); // remove children of '.data' elements
12	return;
13	}
14	
15	const margin = 20; // use a 20-pixel margin
16	const xHeight = 20; // assume an x-axis height of 20px.
17	
18	// Create the y-axis.
19	const yHeight = _info.node.clientHeight - 2 * margin - xHeight;
20	const yDomain = data.cols[YPOS].domain.asArray(); // [min, max]
21	const yScale = d3.scaleLinear().range([yHeight,
22	0]).domain(yDomain).nice();
23	const yAxis = svg.select<SVGGElement>(".yaxis").call(d3.axisLeft(yScale));
24	const yWidth = yAxis.node().getBBox().width;
25	
26	// Create the x-axis and position at bottom of y-axis.
27	const xWidth = _info.node.clientWidth - yWidth - 2 * margin;
28	const xDomain = data.cols[XPOS].domain.asArray(); // [min, max]
29	const xScale = d3.scaleLinear().range([0, xWidth]).domain(xDomain).nice();
30	const xAxis = svg.select(".xaxis").call(d3.axisBottom(xScale));
31	xAxis.attr("transform", `translate(0,\${yHeight})`);
32	
33	// Position the chart (axes and elements).
34	svg.select(".chart").attr("transform", `translate(\${yWidth+margin}, \$
35	{margin})`);
36	
37	// Determine color palette from the properties and create a Circle shape.
38	const palette = props.get("color");
39	const shape = d3.symbol().size(256).type(d3.symbolCircle);
40	
41	// For every row in the data, create and position a circle element.
42	svg.select(".elem")
43	.selectAll("path")
44	.data(data.rows, (row: any) => row.key)
45	.join("path")
46	.attr("d", shape)
47	.attr("transform", row => `translate(\${xScale(row.value(XPOS))}, \$
48	{yScale(row.value(YPOS))})`)
49	.attr("stroke-width", 3)
	.attr("stroke", row => palette.getOutlineColor(row))
	.attr("fill", row => palette.getFillColor(row));
	}

Lines 9-13 Handle the situation when there is no data. This means that `_info.data` is null and the only thing to do is remove all elements in any of the three g groups from the screen.

Lines 15-16 Define a few constants for margin, axis height, and column indices. These are used for readability of the code to follow.

Lines 19-30 Create a y-axis and x-axis. For an axis you need to have a scale that is based on the domain of the data values (min and max) and the range (in pixels). The transform attribute that is added to the axis ensures the axis is positioned correctly.

Line 36 Determines the palette, that you defined in Step 2. You will need this palette in lines 48 and 49 to determine the fill and outline color of the circles that are rendered.

Lines 40-48 Perform the creating, updating, and removing of points for the scatter. The d3 data is based on `data.rows`. A key function ensures that a unique id is assigned to each symbol. Notice the use of the palette on **line 47 and 48**. You can query the palette for a specific color given a data point `d`. Effectively

these two lines make the circles in the visualization respond to highlights (mouse over) and selections (mouse click). The actual positioning of each circle is done on **line 43**, using a `translate` transformation.

Note: You may have noticed that there is no code for handling mouse events, tooltips and drawing legends. Although in very specific cases you can add custom code for this, the visualization that you just created already supports highlights, selections, filtering, and an interactive legend! All of this is provided to you by the `RenderBase` base class and the environment in which the visualization runs. This allows you to create powerful and consistent visualizations with minimal effort.

Step 4: Run and test the visualization

In step 1 you already started the visualization by running `customvis start`. This means you can now start Cognos Analytics and try out your visualization in either a dashboard or a reporting. For details about this, see [“Step-by-step guide for custom visualizations”](#) on page 3.

Links

- [Download](#) the source code for part 1.

Part 2: Adding properties to the visualization

In part 1 of this tutorial, we added a palette property for the colors of the scatter points. In part 2 we will add some more properties to the visualization like a title, point shape, x- and y maximum, and show title flag. We will see various options for defining a property, like a default value and the `allowNull` flag, and property groups.

Before you start

Make sure you have completed [“Part 1: Creating a custom visualization”](#) on page 38 or you have downloaded the [source code](#) for part 1.

Step 1: Modify the `vizdef.xml` file

We already saw that property definitions are part of the `vizdef.xml` file. In this step we will not only add properties, but we also organise the properties in property groups.

Property groups are a hierarchical organisation of properties. The host application will often show property groups as separate tabs or panels in the UI. Placing a property of your custom visualization in a group will make the property appear in the corresponding place in the UI.

For the scatter visualization, we want to define the following properties structure:

- **visualization** (group)
 - **general** (group)
 - `color` - palette property that we added in step 1 of part 1.
 - `pointShape` - shape of a scatter point, ("circle" or "square").
 - `background` - url to a background image.
 - `showBackground` - flag that indicates whether the background should be displayed.
 - **axis** (group)
 - `xmax` - maximum value on x-axis, or null for automatic.
 - `ymax` - maximum value on y-axis, or null for automatic.

The `vizdef.xml` entries for these properties look as follows:

```
<properties>
  <group name="general">
    <palette name="color" slot="categories" />
    <enum name="pointShape" defaultValue="circle">
      <possibleValue>circle</possibleValue>
      <possibleValue>square</possibleValue>
    </enum>
  </group>
  <group name="axis">
    <property name="xmax" />
    <property name="ymax" />
  </group>
</properties>
```

```

    </enum>
    <string name="background" defaultValue="" />
    <boolean name="showBackground" defaultValue="true" />
  </group>
  <group name="axis">
    <number name="xmax" allowNull="true" />
    <number name="ymax" allowNull="true" />
  </group>
</properties>

```

Two attributes to notice here are `defaultValue` and `allowNull`. The `defaultValue` attribute represents the initial value that the property will get. The `allowNull` attribute indicate if null values for the property are allowed. By default, null values are not allowed. For your custom visualization code this means in the default situation you can always rely on a non-null value when calling `Properties.get`.

Step 2: Add code in Main.ts

pointShape

Changing the shape of a point is easy enough, since for each possible enum value there is a corresponding d3 symbol type available. Replace the current code that determines the palette and shape to:

```

// Determine color palette and shape from the properties.
const palette = props.get( "color" );
const shape = d3.symbol().size( 256 );
if ( props.get( "pointShape" ) === "square" )
  shape.type( d3.symbolSquare );
else
  shape.type( d3.symbolCircle );

```

If you want to support more than just these two shape types, feel free to extend this code a bit more. Maybe you want to use a switch statement to select the correct symbol type, or use a lookup map.

Notice that the value of an enum property is always a string. We can safely compare the property value with the literal string "square", or we can use the value of the property in a switch statement.

Note: It is good practice to choose enum values that are not likely to change over time. In Part 3 of this tutorial we will learn how to add a localized 'caption' for an enum value, that is used for showing in the UI.

xmax and ymax

In part 1 we already defined the domain for our axes by calling `getDomain` on the data. If `xmax` or `ymax` are non-null, we can use these property values instead. So the code to create the y-axis should be changed as follows:

Line	Code
1	// Create the y-axis.
2	const yHeight = _info.node.clientHeight - 2 * margin - xHeight;
3	const yDomain = data.cols[YPOS].domain.asArray(); // [min, max]
4	const yMax = props.get("ymax");
5	if (yMax !== null)
6	yDomain[1] = yMax;
7	const yScale = d3.scaleLinear().range([yHeight, 0]).domain(yDomain);
8	if (yMax === null)
9	yScale.nice(); // apply nice scale only if scale is automatic.
10	const yAxis = svg.select<SVGGElement>(".yaxis").call(d3.axisLeft(yScale));
11	const yWidth = yAxis.node().getBBox().width;

The call to `props.get("ymax")` on line 4 will return either null or a valid number. This is because `allowNull = true` was set on the property.

The code change for the x-axis is now trivial:

Line	Code
1	// Create the x-axis and position at bottom of y-axis.
2	const xWidth = _info.node.clientWidth - yWidth - 2 * margin;
3	const xDomain = data.cols[XPOS].domain.asArray(); // [min, max]
4	const xMax = props.get("xmax");
5	if (xMax !== null)
6	xDomain[1] = xMax;
7	const xScale = d3.scaleLinear().range([0, xWidth]).domain(xDomain);
8	if (xMax === null)
9	xScale.nice(); // apply nice scale only if scale is automatic.
10	const xAxis = svg.select<SVGElement>(".xaxis").call(d3.axisBottom(xScale));
11	xAxis.attr("transform", `translate(0,\${yHeight})`);

background and showBackground

To add a background to the scatter visualization, we add the following code to the `updateMethods`, just before you create the y-axis:

Line	Code
1	// Set the background image.
2	let urlImage = null;
3	if (props.get("showBackground"))
4	{
5	const image = props.get("background");
6	if (image !== "")
7	urlImage = `url(\${image})`;
8	}
9	_info.node.parentElement.style.backgroundImage = urlImage;
10	
11	// Create the y-axis.
12	...

This code changes the `backgroundImage` style of the parent element of the `svg`. The parent of the `svg` is actually the `_node` that was passed to us in `create`.

Step 3: Run and test the visualization

Create a new dashboard or report in IBM Cognos Analytics and add the 'Preview Visualization'. After adding some data you should see the scatter visualization in action. Try changing some of the properties that we have just added. You will see that the properties have been added to the **visualization** tab and the **text** tab respectively.

Links

- [Download](#) the source code for part 2.

Part 3: Customize legend and properties

In this part of the tutorial, we will improve the code in `main.ts` file. We start with adapting the symbols used in the legend to the symbols used for the scatter points. The `title` and `showTitle` properties from part 2 are related, so we will add code that ensures that title gets disabled when `showTitle` is switched off in the user interface. Finally, we will add captions to some of the properties.

Before you start

Make sure you have completed “Part 1: Creating a custom visualization” on page 38 and “Part 2: Adding properties to the visualization” on page 43 or you have [downloaded](#) the source code for part 2.

Step 1: Change the legend

In part 2 you added a property that allows the user to change the shape of the scatter points. In this step you will see how you can have the legend show the same shape as used in the visualization.

The `RenderBase` class allows you to configure various aspects of your visualization through the `meta` attribute. This attribute allows you to set things like maximum number of data points and legend shape. For instance if we add the following line of code inside the 'create' method, then categorical legends always render a 'circle' symbol:

```
this.meta.legendShape = "circle";
```

Often the `meta` attribute is used during initialization of the visualization. However, for your scatter visualization you want to have the legend update as soon as our `pointShape` property changes.

Add the following code to the custom visualization class in `Main.ts` file:

```
protected updateProperty( _name: string, _value: any )
{
    switch( _name )
    {
        case "pointShape":
            this.meta.legendShape = _value;
            break;
    }
}
```

The `updateProperty` method is called immediately when a property changes. This means that next time when the legend needs to redraw, it retrieves the correct value from `meta.legendShape`. If we would have updated this value in the `update` method, then we would have been too late because the legend might have rendered its contents before the visualization.

Note: In [“Part 2: Adding properties to the visualization” on page 43](#) you saw that the value of an enum property is always a string. In the code above you can assign the string value to `meta.legendShape` since you are sure the enum value is a valid legend shape.

Step 2: Active & inactive properties

The same `updateProperty` function that was defined in the previous step can be used to implement so-called 'active state' for the properties. The active state of a property determines how a property is rendered in the UI. If a property is inactive, it is often rendered as 'disabled'. In our scatter visualization we want to disable the 'background' property if 'showBackground' is set to `false`. If 'showBackground' is set to `true`, then the 'background' should be enabled again. To do this, we change `updateProperty` and handle the property change for the 'showBackground' property:

```
protected updateProperty( _name: string, _value: any )
{
    switch( _name )
    {
        case "pointShape":
            this.meta.legendShape = _value;
            break;

        case "showBackground":
            // Active state for 'background' depends on 'showBackground'.
            this.properties.setActive( "background", _value );
            break;
    }
}
```

Step 3: Property captions

In the previous parts you have seen that each property in `vizdef.xml` file has a name to which can refer in code. These names are normally not suited to be shown to the end user in the UI. For this reason, every property can have a `caption` attribute that defines the name that should be shown in the UI. Change the `vizdef.xml` file as follows:

```
<properties>
  <group name="general">
    <palette name="color" caption="Colors" slot="categories" />
    <enum name="pointShape" caption="Shape" defaultValue="circle">
      <possibleValue caption="Circle">circle</possibleValue>
      <possibleValue caption="Square">square</possibleValue>
    </enum>
  </group>
</properties>
```

```

        </enum>
        <string name="background" caption="Background" defaultValue="" />
        <boolean name="showBackground" caption="Show Background" defaultValue="true" />
    </group>
    <group name="axis" caption="Axis">
        <number name="xmax" caption="X-axis max" allowNull="true" />
        <number name="ymax" caption="Y-axis max" allowNull="true" />
    </group>
</properties>

```

Note: The property group `general` does not have a caption because that group is already provided by the host application and therefore already has a (localized) caption. If you define your own property groups, you can give them a caption.

In “Part 4: Advanced features” on page 47 you will see how to make the captions adapt to the user interface language of IBM Cognos Analytics.

Step 4: Run and test the visualization

Create a new dashboard or report in Cognos Analytics and add the 'Preview Visualization'. After adding some data you should see the scatter visualization in action. Try changing some of the properties that we have just added.

This code changes the `backgroundImage` style of the parent element of the `svg`. The parent of the `svg` is actually the `_node` that was passed to us in `create`.

Links

- [Download](#) the source code for part 3.

Part 4: Advanced features

In the last part of this tutorial, you will see the use of optional slots and show how the automatically generated legend can be customized. You will also see how to setup localization for your custom visualization.

Step 1: Optional slots

In “Part 1: Creating a custom visualization” on page 38 you added entries in the `vizdef.xml` file to define the slots of the visualization. All slots were marked as mandatory by setting the `optional` attribute to `false`. This means that the user has to provide data in all of the slots before the visualization is able to render data.

In this step you will add an optional `color` slot. The slot type is `continuous`, which means that the user can map numerical data to the slot. If the slot is mapped, then the values in the slot determine the point color. If the slot is not mapped, then the scatter chart works like it does now, using the `color` palette.

Here is the `vizdef.xml` file change that is needed. Notice the a change is needed in both the `slots` section (line 3) and the `dataSet` section (line 9).

Line	Code
1	<slots>
2	...
3	<slot name="color" type="cont" optional="true" channel="color" />
4	</slots>
5	
6	<dataSets>
7	<dataSet>
8	...
9	<ref slot="color" />
10	</dataSet>
11	</dataSets>

In addition to adding the slot, you also need to add a continuous palette that will be used in case the new slot is mapped to data:

Line	Code
1	<pre> <properties> <group name="general"> <palette name="color_cat" slot="categories" /> <palette name="color_cont" slot="color" /> ... </pre>
2	
3	
4	
5	

See line 3 and 4. Notice that you renamed the existing property "color" to "color_cat" to make a clearer distinction between the two palette properties.

In the `Main.ts` file add a constant that identifies the color slot:

```
const CAT = 0, XPOS = 1, YPOS = 2, COLOR = 3; // slot indices
```

Finally modify the code in `Main.ts` file to pick up the colors and data from the new slot:

Line	Code
1	<pre> <!-- Determine color palette and shape from the properties. const hasColor = data.cols[COLOR].mapped; const palette = props.get(hasColor ? "color_cont" : "color_cat"); const shape = d3.symbol().size(256); ... </pre>
2	
3	
4	
5	

On line 3 you can see that depending on the mapped state of the column, the continuous or the categorical palette is retrieved for determining the color.

That's all it takes, the user of the custom visualization can now map data onto the new slot. That data will determine the color of a scatter point. If there is no data, then the color will be from the categorical slot.

Step 2: Customizing the legend

When you have completed the previous step and tested it, you might have noticed that in case there was data mapped to the new 'color' slot, the categorical legend still was visible. This is because both the 'categories' slot and 'color' slot have a 'color' channel and both are mapped. The automatic legend algorithm cannot determine which of the two palettes is actually in use.

To override the behavior of the automatic legend, you can override the `updateLegend` function. In this function, you can add, remove or modify individual legends and legend items.

To start, update the 'imports' section as follows:

```
import { RenderBase, UpdateInfo } from "@businessanalytics/customvis-lib";
import { Encoding, DataSet } from "@businessanalytics/customvis-lib";
import * as d3 from "d3";
```

Then add an `updateLegend` method to the visualization class:

Line	Code
1	<pre> protected updateLegend(_data: DataSet): Encoding[] { // Call base class implementation to setup initial encodings. const encodings = super.updateLegend(_data); const hasColor = _data.cols[COLOR].mapped; // Filter encodings: if color slot is mapped, skip categorical legend. return encodings.filter(_encoding => { if (hasColor && _encoding.type === "cat") return false; return true; }); } </pre>
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

The `RenderBase.updateLegend` implementation creates an array holding legend Encodings (line 4). This array holds all the information needed for the legend to draw. Overriding `updateLegend` gives you

the ability to make changes to the encodings created in `RenderBase`. The `filter` function that is called in line 8 will return the encodings that satisfy the condition defined in the function body.

Step 3: Localization

In “Part 3: Customize legend and properties” on page 45 you added captions to the `vizdef.xml` file, so the user interface shows proper labels for the properties and slots. In this step you will see how you can localize these captions, so they adapt to the UI language that you have chosen in IBM Cognos Analytics.

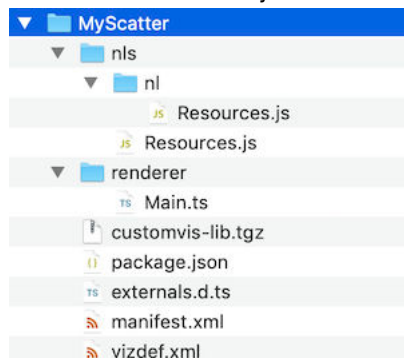
To have the visualization support translated strings, add a new directory called `nls` in the root directory where the visualization is located (as a sibling of `vizdef.xml` file). Then add a file `Resources.js` in this `nls` directory, with the following contents:

```
define(
{
  "root": {
    "prop_color_cat":      "Point Color",
    "prop_color_cont":     "Color Range",
    "prop_shape":          "Shape",
    "prop_shape_circle":   "Circle",
    "prop_shape_square":   "Square",
    "prop_background":     "Background",
    "prop_show_background": "Show Background",
    "prop_xmax":           "X-axis max",
    "prop_ymax":           "Y-axis max",
    "prop_axis":           "Axis"
  },
  "nl": true
} );
```

The last entry `"nl": true` indicates that there is a Dutch version of this file in the `nl` subfolder. So, let's create a subfolder `nl` inside the `nls` folder. Then add another `Resources.js` file in this `nl` subfolder with the following contents:

```
define(
{
  "prop_color_cat":      "Puntkleur",
  "prop_color_cont":     "Kleurbereik",
  "prop_shape":          "Vorm",
  "prop_shape_circle":   "Cirkel",
  "prop_shape_square":   "Vierkant",
  "prop_background":     "Achtergrond",
  "prop_show_background": "Toon Achtergrond",
  "prop_xmax":           "X-as max",
  "prop_ymax":           "Y-as max",
  "prop_axis":           "As"
} );
```

Your file and directory structure should now look as follows:



Finally, you replace the hard-coded captions in `vizdef.xml` file with the variables you just defined in the `nls` files (prefixed with `$nls.`):

```
<properties>
  <group name="general">
    <palette name="color_cat" caption="$nls.prop_color_cat" slot="categories" />
```

```

    <palette name="color_cont" caption="$nls.prop_color_cont" slot="color" />
    <enum name="pointShape" caption="$nls.prop_shape" defaultValue="circle">
      <possibleValue caption="$nls.prop_shape_circle">circle</possibleValue>
      <possibleValue caption="$nls.prop_shape_square">square</possibleValue>
    </enum>
    <string name="background" caption="$nls.prop_background" defaultValue="" />
    <boolean name="showBackground" caption="$nls.prop_show_background"
defaultValue="true" />
  </group>
  <group name="axis" caption="$nls.prop_axis">
    <number name="xmax" caption="$nls.prop_xmax" allowNull="true" />
    <number name="ymax" caption="$nls.prop_ymax" allowNull="true" />
  </group>
</properties>

```

Note: Not only properties can have captions. You can also define captions for the slots and either hard code it in the vizdef.xml file or provide a localization id like you did for the properties.

Step 4: Run and test the visualization

Create a new dashboard or report in Cognos Analytics and add the 'Preview Visualization'. After adding some data you should see the scatter visualization in action. After adding some data you should see the scatter visualization in action. Try adding data to the 'color' slot and removing it again. You will notice the difference in the legend that appears for your visualization.

Links

- [Download](#) the source code for part 4.

Chapter 9. Manage custom visuals


Managing custom visuals from the admin UI





Use the manage UI to delete, download, update, or set the properties of custom visuals

About this task

You can manage custom visuals from the IBM Cognos Analytics admin UI.

Procedure

1. On the home page, click **Manage > Customization > Custom visuals**.
2. Next to the custom visual, click the **More** icon  and select one or the following options:

Option	Description
Delete 	If you no longer need the custom visual, delete it.
Download 	Download the custom visual. Download the custom visual to, for example, send the custom visual to a support organization.
Update 	If you have a new version available, update the custom visual. You need to select the updated custom visual.
Properties 	Set the permissions for the custom visual. For more information, see Access Permissions and Disabling Packages and Folders .

Results

The custom visualization is added to the list of visualizations. Optionally you can set the permissions on the visualizations.

Chapter 10. Improving the performance of custom visualizations

Improving the performance of custom visualizations

If your custom visualization needs to display many data points, you might experience decreased performance of the visualization.

You can apply several techniques that limit the amount of time that is spent in rendering your visualization. The `customvis` API provides you with information that can be used in optimizing your custom visualization.

Render performance

Rendering a custom visualization with many data points can take a hit on the performance. If you know what triggers the render action, then you can decide what needs to be rendered again.

Use the `reason` field in the `UpdateInfo` class to determine what triggered the render and then determine what items you render to minimize the performance decrease. An instance of `UpdateInfo` is passed to you every time update is called.

Updating the document object model (DOM) and recalculating are often expensive operations. Whenever possible, prevent these updates from being performed too often. For instance, if the reason for rendering is the change of the background color of your custom visualization, then you might want to change the color of the corresponding UI element and leave the rest of the visualization untouched.

Also, be aware that highlights and selections can cause the update function to be called many times. If your custom visualization supports highlighting elements, then each mouse move operation over the visualization might trigger an update. You find that for rendering highlights and selections the `decorations` field in the `reason` is flagged. Depending on how you would like to show highlighted and selected elements, you can consider an alternative, high performance, render function that is dedicated to rendering selections and highlights.

For more information, see [Class RenderReason](#) and [Class UpdateInfo](#).

Responsiveness of the custom visualization

The displayed size of your custom visualizations can introduce a decrease in performance. It is not necessary to always show everything. You can tweak the following items of your custom visualization to both increase the performance and show a meaningful visualization in limited screen space:

- Labels on axis
- Titles
- Text labels

Chapter 11. Authoring schematics - Tutorial

SVG definition

You must annotate SVG files to link SVG elements to a dataset.

The mapping is visually represented in the rendered SVG by changing the fill (color and opacity) of the SVG elements. When you create an SVG document, elements that have no visible fill area do not appear as visibly changed.

Note: You can also use third-party tools, such as Inkscape, to annotate SVGs.

For more information, see [Using Inkscape to annotate schematics for use in IBM Cognos Analytics](https://community.ibm.com/community/user/businessanalytics/blogs/marco-maas1/2019/12/20/using-inkscape-to-annotate-schematics) at <https://community.ibm.com/community/user/businessanalytics/blogs/marco-maas1/2019/12/20/using-inkscape-to-annotate-schematics>.

SVG basic mapping

You map between an element in an SVG and a value in a dataset by adding the value from the dataset as a new attribute called data-cv-key for the corresponding region in the SVG.

Note: You can also use third-party tools, such as Inkscape, to annotate SVGs.

For more information, see [Using Inkscape to annotate schematics for use in IBM Cognos Analytics](https://community.ibm.com/community/user/businessanalytics/blogs/marco-maas1/2019/12/20/using-inkscape-to-annotate-schematics) at <https://community.ibm.com/community/user/businessanalytics/blogs/marco-maas1/2019/12/20/using-inkscape-to-annotate-schematics>.

Mapping is supported on the following elements:

- Circle
- Ellipse
- Line
- Path
- Polygon
- Polyline
- Rect
- Text
- SVG
- G

Note: The G element is non-visual and is designed to group visual elements together to create a composite element.

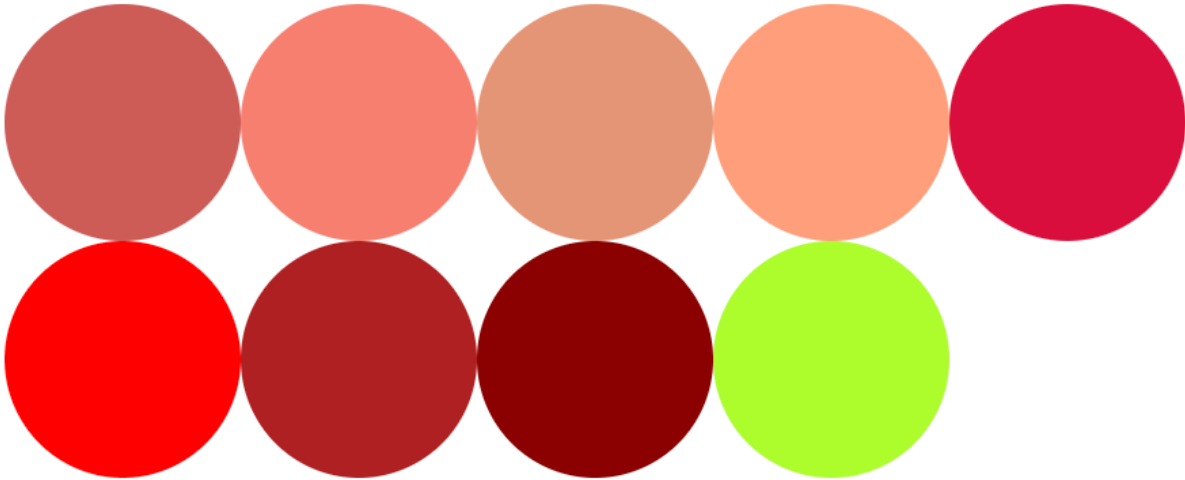
The behavior for other SVG elements is undefined.

The following SVG file does not have the data-cv-key attribute set for the circles:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg height="40" width="100" xmlns="http://www.w3.org/2000/svg">
  <g>
    <circle cx="10" cy="10" fill="#cd5c5c" r="10"/>
    <circle cx="30" cy="10" fill="#fa8072" r="10"/>
    <circle cx="50" cy="10" fill="#e9967a" r="10"/>
    <circle cx="70" cy="10" fill="#ffa07a" r="10"/>
    <circle cx="90" cy="10" fill="#dc143c" r="10"/>
  </g>
  <g>
    <circle cx="70" cy="30" fill="#adff2f" r="10"/>
    <circle cx="10" cy="30" fill="#ff0000" r="10"/>
    <circle cx="30" cy="30" fill="#b22222" r="10"/>
    <circle cx="50" cy="30" fill="#8b0000" r="10"/>
  </g>
</svg>
```

```
</g>
</svg>
```

If you would load this visualization, it would display as follows. Mapping to data is not possible.



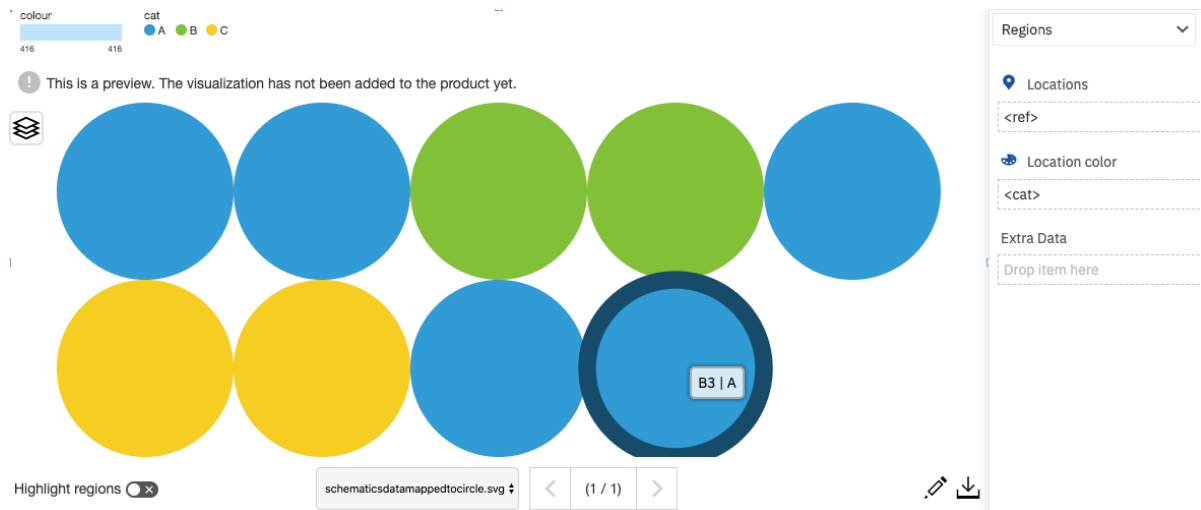
If you add the data-cv-key attribute to the SVG, then you can map data to the circles in the SVG.

```
<?xml version="1.0" encoding="UTF-8"?>
<svg height="40" width="100" xmlns="http://www.w3.org/2000/svg">
  <g>
    <circle data-cv-key="A2" cx="10" cy="10" fill="#cd5c5c" r="10"/>
    <circle data-cv-key="A3" cx="30" cy="10" fill="#fa8072" r="10"/>
    <circle data-cv-key="A4" cx="50" cy="10" fill="#e9967a" r="10"/>
    <circle data-cv-key="A5" cx="70" cy="10" fill="#ffa07a" r="10"/>
    <circle data-cv-key="B2" cx="90" cy="10" fill="#dc143c" r="10"/>
  </g>
  <g>
    <circle data-cv-key="B3" cx="70" cy="30" fill="#adff2f" r="10"/>
    <circle data-cv-key="B4" cx="10" cy="30" fill="#ff0000" r="10"/>
    <circle data-cv-key="B5" cx="30" cy="30" fill="#b22222" r="10"/>
    <circle data-cv-key="D1" cx="50" cy="30" fill="#8b0000" r="10"/>
  </g>
</svg>
```

And the following data:

ref	cat	color	occupied	size
A2	A	72	no	157
A3	A	36	no	684
A4	B	12	yes	325
A5	B	22	no	186
B2	A	16	yes	487
B3	A	78	yes	560
B4	C	74	yes	871
B5	C	87	no	980
D1	A	19	yes	574

If you use this SVG and drag *ref* to the **Locations** data slot and *cat* to the **Location color** data slot, then the color of the circles is mapped with the cat data.



Rules and restrictions for SVG

There are some considerations that you must be aware of when you author schematics for IBM Cognos Analytics.

View-mapping rules and restrictions

Supported visual elements:

- Circle
- Ellipse
- Line
- Path
- Polygon
- Polyline
- Rect
- Text

Supported non-visual elements:

- svg
- g

An SVG element with an attribute of `data-cv-view` can also have an attribute of `data-cv-key`. For example,:

```
<circle x=10 y=10 data-cv-view="circles" data-cv-key="A1"/>
```

Only a single `data-cv-view` attribute is supported.

An SVG element can exist in a single view only. Multiple views are not supported.

Hierarchical views are not supported. For example, the following example is not supported:

```
<g data-cv-view="circles">
  <circle x=10 y=10 data-cv-view="child_circle" data-cv-key="A1"/>
  <circle x=10 y=10 data-cv-view="child_circle" data-cv-key="A2"/>
</g>
```

Composite items are supported. For example, removing the child data-cv-view attributes from the previous example:

```
<g data-cv-view="circles">
  <circle x=10 y=10 data-cv-key="A1"/>
  <circle x=10 y=10 data-cv-key="A2"/>
</g>
```

The same value is supported on multiple elements:

```
<circle x=10 y=10 data-cv-view="circles" data-cv-key="A1"/>
<circle x=10 y=10 data-cv-view="circles" data-cv-key="A2"/>
```

There is no limit on the number of data-cv-view attribute occurrences within a document.

Key mapping rules and restrictions

Supported visual elements:

- Circle
- Ellipse
- Line
- Path
- Polygon
- Polyline
- Rect
- Text

Supported non-visual elements:

- svg
- g

Only a single data-cv-key attribute is supported:

```
<circle x=10 y=10 data-cv-key="A1"/>
```

Hierarchical or multi-part keys are not supported:

```
<g data-cv-key="A">
  <circle x=10 y=10 data-cv-key="1"/>
  <circle x=10 y=10 data-cv-key="2"/>
</g>
```

Composite items are supported:

```
<g data-cv-key="A1">
  <circle x=10 y=10/>
  <circle x=10 y=10/>
</g>
```

In a composite item, multiple data-cv-key attributes are not supported:

```
<g data-cv-key="A1">
  <circle x=10 y=10 data-cv-key="A2"/>
  <circle x=10 y=10 data-cv-key="A3"/>
</g>
```

Elements that do not have a closed path are not supported. For example, the following path is not closed with z or Z.

```
<path d="M8 0C3.582 0 0 3.582"/>
```

Supported SVG features

Use control structures and other features when you author your SVG documents. The following control structures are supported when you author schematics:

- use
- defs
- href

The following control structures are not supported when you author schematics:

- Embedded JavaScript
- Include tags
- Use of Entity defined in DOCTYPE section
- Animation events
- Document events
- Document element events
- Graphical events
- Any other global events
- References to anything external, including URIs to external fonts, and the use of the HTML anchor <a> element.

Saving and exporting as SVG options

- Choose **Presentation Attributes** for styling if you require compatibility with Microsoft Edge. Choose styling that uses element attribute or style attribute, and avoid CSS style through internal header or external CSS style.
- Use embedded images if necessary.
- Avoid options like preserving editor-specific capability that might inject unnecessary XML code.

Adding views to schematics

Schematics support **Views** that are defined in the SVG. Views are conceptually different than layers.


A layer is created when data is mapped to either the **Region > Locations** or **Points > Locations** slots. You can toggle layers on or off, which shows or hides the results of the data mapping.

When you author an SVG, you can create views. Views define groups of SVG elements where these elements are logically related to each other. For example, if you create a floor plan of a hospital, then you define **Views** for each hospital ward in the SVG. These views allow the user of the schematic to show or hide the views that they are interested in.

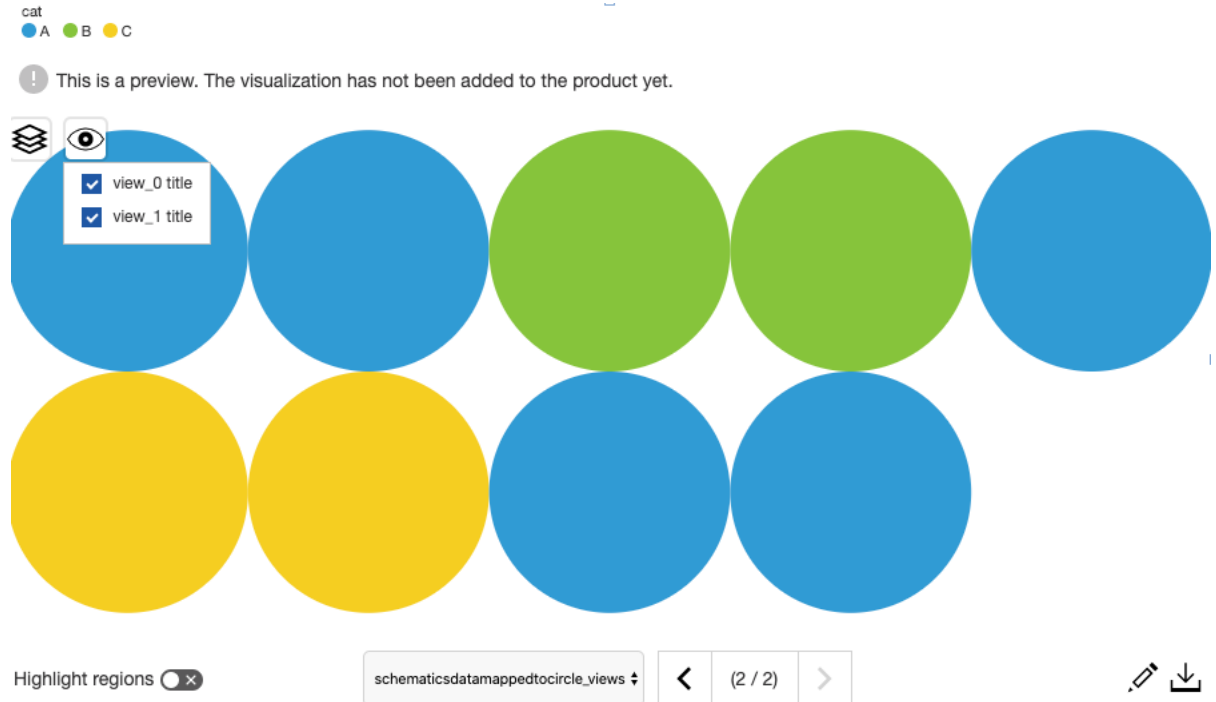
Annotating an SVG is done in a similar way to the data mapping that uses the data attribute data-cv-view, along with an optional title and description for each view. The description is rendered as a tooltip in the schematic. See the following SVG code sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg height="40" width="100" xmlns="http://www.w3.org/2000/svg">
  <g data-cv-view="view_0" desc="view_0 desc" title="view_0 title">
    <circle data-cv-key="A2" cx="10" cy="10" fill="#cd5c5c" r="10"/>
    <circle data-cv-key="A3" cx="30" cy="10" fill="#fa8072" r="10"/>
    <circle data-cv-key="A4" cx="50" cy="10" fill="#e9967a" r="10"/>
    <circle data-cv-key="A5" cx="70" cy="10" fill="#ffa07a" r="10"/>
    <circle data-cv-key="B2" cx="90" cy="10" fill="#dc143c" r="10"/>
  </g>
  <g data-cv-view="view_1" desc="view_1 desc" title="view_1 title">
    <circle data-cv-key="B3" cx="70" cy="30" fill="#adff2f" r="10"/>
    <circle data-cv-key="B4" cx="10" cy="30" fill="#ff0000" r="10"/>
    <circle data-cv-key="B5" cx="30" cy="30" fill="#b22222" r="10"/>
    <circle data-cv-key="D1" cx="50" cy="30" fill="#8b0000" r="10"/>
  </g>
</svg>
```

```
</g>  
</svg>
```

If you use this SVG and drag *ref* to the **Locations** data slot and *cat* to the **Location color** data slot, then the color of the circles is mapped with the cat data. To select a view, click the **Views**  icon.

The title is displayed in the list of available views. If you did not define a title, then the data-cv-view attribute value is used. By default all views are selected.

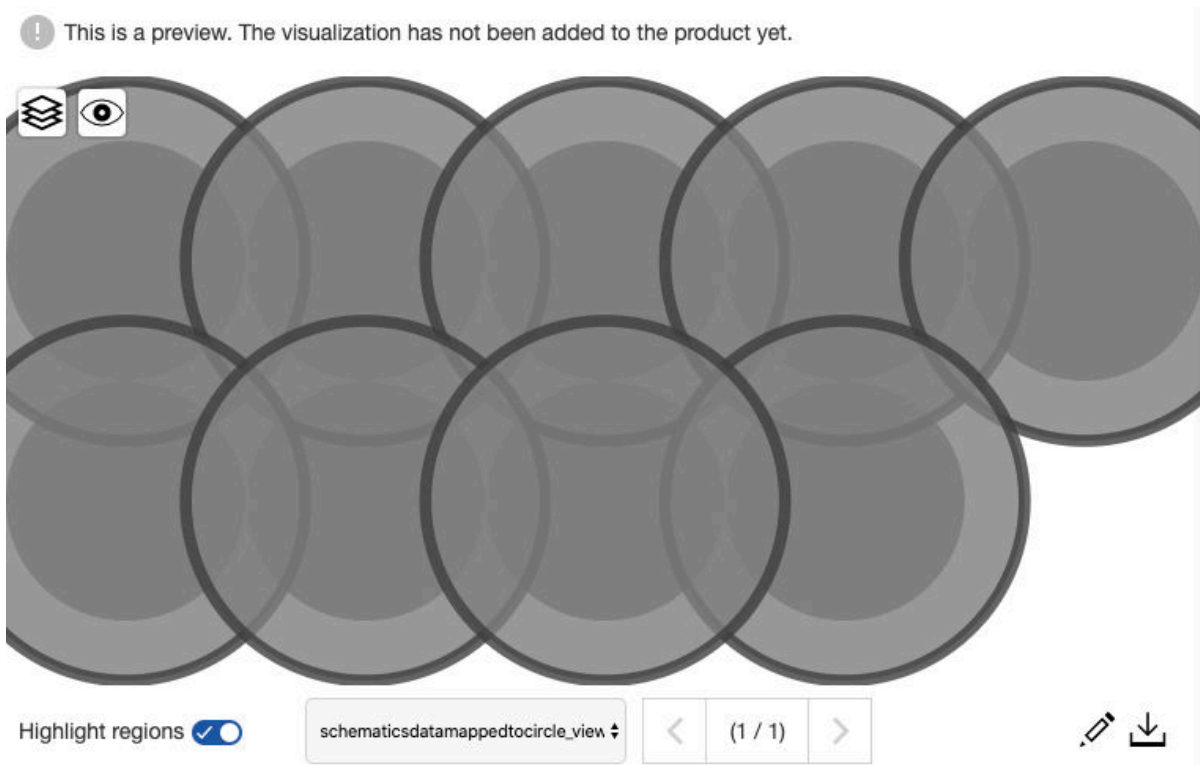


Highlighting regions in a schematic

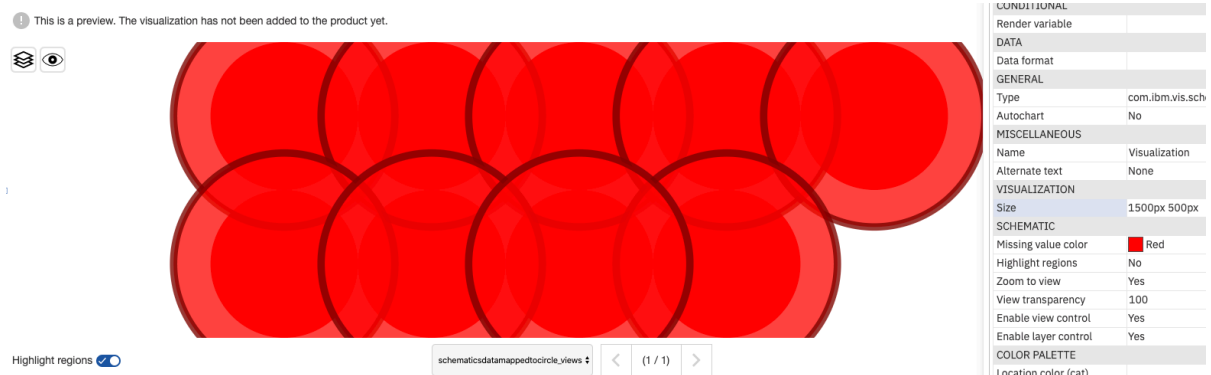
To know which parts of the SVG can be mapped, you can highlight those parts in IBM Cognos Analytics.

About this task

Use the **Highlight region** option to display the parts of an SVG that can be mapped. An element in the SVG can be mapped when the element has an attribute `data-cv-key` with a value that contains a key.



When turned on, all mappable elements receive a fill with a color as defined in the property **Missing value color**. All non-mappable elements have an opacity that is applied.



For more information, see [Properties for individual 11.1 visualizations in Reporting](#).

Procedure

1. In a report on the **Toolbox** tab of the content pane, select the **Test schematic** object and drag it onto the workspace.
In a dashboard on the **Custom** tab of the **Visualizations** pane, select the **Test schematic** object and drag it onto the workspace.
2. Toggle the **Highlight region** switch.

Editing a schematic package

Before you make the schematic, that you authored, available to the users, you can edit the schematic package.

About this task

You can edit the properties and the content of the schematic package.

Edit package

Package

Content

Name

Schematic

Description

Icon

Drop files here

or

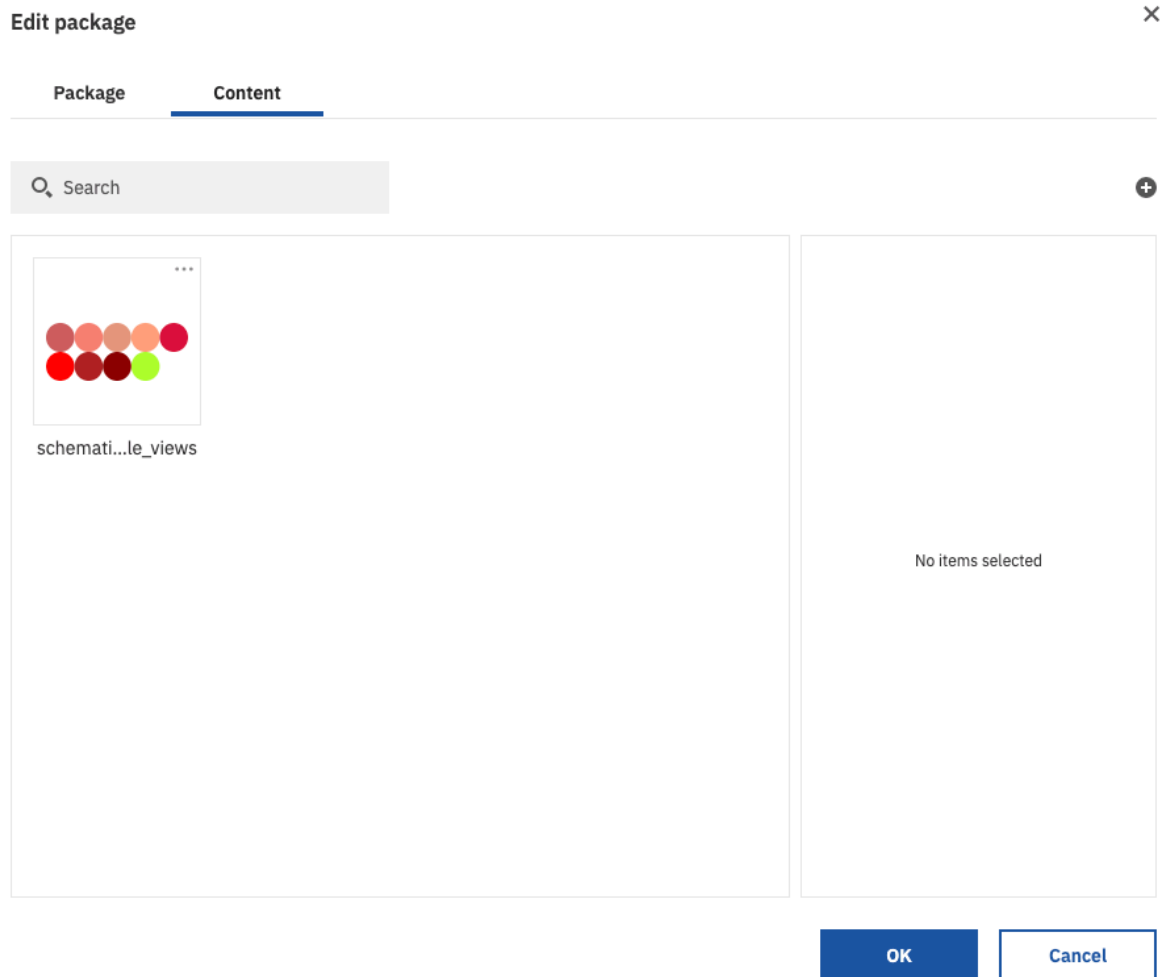
Choose a file

Only .jpg, .png, and .svg files.

500MB max file size.

OK



Cancel



Note:

You can upload the schematic package, that you downloaded, to the **Schematic preview** widget, for further editing.

Procedure

1. In a report on the **Toolbox** tab of the content pane, select the **Schematic preview** object and drag it onto the workspace.
2. Click the **Edit package**  icon.
3. On the **Package** tab, you can change the following properties:
 - **Name**
 - **Description**
 - **Icon**: The icon can be of the .jpg, .png, or .svg type with a maximum file size of 500 MB.
4. On the **Content** tab, you can change the content of the schematic package.
 - Click the **Upload file**  icon and select the SVG files that you want to include in the schematic package.
 - Click an SVG to change its **Name** and **Caption**.
 - On an SVG, click the **More** *** icon to **Replace** or **Delete** the SVG.

Distributing a schematic



If you want to make the authored schematic available to the users in IBM Cognos Analytics, distribute it. Distribution is done by first downloading the packaged schematic from the **Schematic preview** and then uploading the packaged schematic as a **Custom visual**.

About this task

The **Manage Visualizations** capability allows users to control access rights to schematics for individual users, groups, and roles.

The **Develop Visualizations** capability allows users to develop and distribute schematics.

Procedure

1. In a report on the **Toolbox** tab of the content pane, select the **Schematic preview** object and drag it onto the workspace.
2. Author the schematic.
3. Click the **Download**  icon.
A .zip file is downloaded on your file system.
4. On the home page, click **Manage > Customization > Custom visuals > Upload custom visual** .
5. Select the `yourschematic.zip` file and click **Open**.

Results

The schematic is added to the list of visualizations. Optionally you can set the permissions on the visualizations.

