Part I: Introduction to Developing Visualization Extensions for Qlik Sense

#1: About this Tutorial

The intention of this tutorial is to help you to start developing Extensions in Qlik Sense. If you have already spent some time with the concept of Extensions in QlikView you'll see some similarities in the first chapters, but at the same time you'll recognize how much cleaner and better the Extension API in Qlik Sense is in comparison to the one in QlikView.

Although at this point it's important to mention that the main idea behind this tutorial is to bring Qlik Sense's Visualization Extension concept to everybody, so you **don't need any skills in neither QlikView nor Qlik Sense** to get started.

Prerequisites

To be able to understand this tutorial, it is **not necessary** that you have already developed any extensions in either QlikView or Qlik Sense, but you should have a basic understanding of the following concepts:

- Html
- JavaScript

Qlik Sense

All the concepts explained in this tutorial certainly work for both Qlik Sense Server and Qlik Sense Desktop. At the same time only Qlik Sense Desktop is necessary to follow the examples in this tutorial.

The entire tutorial is written based on Qlik Sense 1.0, published in September 2014. If some features of newer versions of Qlik Sense are discussed you'll find an appropriate inline note.

Code Editor

In addition to Qlik Sense you'll need a code editor, basically every basic editor is sufficient, so something like

- Notepad++,
- Sublime Text or
- Aptana ...

#2: Introduction to Qlik Sense Visualization Extensions

Visualization Extensions are a possibility to extend the visualization capabilities of Qlik Sense by using only standard web-technologies. The concept of extensions in general can also be seen as a plugin-mechanism which allows developers to combine the power of Qlik Sense' APIs with the nearly unlimited capabilities of the web.

Extensions are so powerful in Qlik Sense because they use exactly the same set of technologies as standard objects in Qlik Sense do, so you are basically using the same set of technologies as Qlik's R&D does!
If developed properly, the handling of *Visualization Extensions* is identical to standard objects:

Visualization Extensions can be:

- Added to a sheet via drag'n'drop
- Configured to use the same property panels as standard objects do, therefore you can create visualizations on top of the data of a Qlik Sense app
- Resized, copied, pasted and positioned like any other object
- Added and used in any story
- Responsive, working properly on any device
- And much more …

Nomenclature

If you have a look into the Developer Help of Qlik Sense you'll find both the term "Extension" and also "Visualization". I keep to the term*Visualization Extension* in this tutorial because the main idea of this concept is to create new visualizations for Qlik Sense. Talking only about*Extensions* might be a limitation for the future since there might be additional types of extensions in the future.

Using an existing visualization extension

Before we start developing a visualization extension in Qlik Sense we first should understand how to „install" and use an existing extension.

Installing a visualization extension on Qlik Sense Desktop

To make an existing visualization extension visible for Qlik Sense Desktop you have to make it available at the location where Qlik Sense Desktop is loading extensions from:

`C:\Users\[UserName]\Documents\Qlik\Sense\Extensions\`

If you download a visualization extension you'll typically get a .zip file. Let's say we download a HelloWorld.zip. For getting this visualization working, do the following:

1. Open Windows Explorer and navigate to `C:\Users\[UserName]\Documents\Qlik\Sense\Extensions\`
2. Create a folder named `HelloWorld`
3. Unzip the content of `HelloWorld.zip` to this folder
   - You should then have at least a JavaScript file (probably something like `HelloWorld.js`) and a file with the extension `.qext`(so something like `HelloWorld.qext`)

If you then have a look into the list of visualization objects you should see the one coming from the imported visualization extension, if this is not the case, refresh either Qlik Sense Desktop (press key `F5`) or refresh your browser.

Deploying visualizations in Qlik Sense Desktop in the Qlik Sense Developer help.

Installing a visualization extension on Qlik Sense Server

If you want to make your visualization extension available on Qlik Sense Server you have to import into the repository.

See Deploying visualizations in Qlik Sense in the Qlik Sense Developer help.

Comparison between extensions in QlikView and Qlik Sense

Beside one exception the comparison between the concept of Extensions in QlikView and Qlik Sense is quite easy:

- Whatever you could achieve with Extensions in QlikView, can also be achieved with Qlik Sense plus much more

There is only a single exception to this rule:

- There is no concept of such as a Document Extension in Qlik Sense, all Visualization Extensions are – in the nomenclature of QlikView – Object Extensions.

Where to find Qlik Sense visualization extensions

There are some places where you can find Extensions for Qlik Sense:

- **branch.qlik.com**
  branch.qlik.com is an open source community where you can find a lot of completely new solutions built on top of both QlikView's and Qlik Sense' APIs, certainly there is also a dedicated section on this page only listing visualization extensions for Qlik Sense

- **market.qlik.com**
  Probably quite soon, we'll also see commercial visualization extensions for Qlik Sense on market.qlik.com

- **GitHub**
  Fortunately quite a lot of developers are publishing their extensions to GitHub. Give it a try.

#3: Let's Get Started: Hello World Example

The first example is extremely easy, let's just create a simple „Hello World“ visualization extension. In the following chapters we'll improve this example and extend it with additional functionality.

So what's necessary to create a Qlik Sense Extension? Let's first have a look at the anatomy of an extension:

The anatomy of a Qlik Sense extension

The following files are mandatory when creating a Qlik Sense Extension:

- Main script file
- Extension meta data file (.qext)

In addition to the mandatory files an extension project may certainly have one or more of the following complementary file-types:

- External scripts
- Style sheets (.css files)
- Images, icons and fonts
- and other resources ...

Skeleton of a script file

This is the basic skeleton of a script file for a visualization extension for Qlik Sense:

```
define( [ /* dependencies */ ],

    function ( /* returned dependencies as arguments */ ) {

        'use strict';

        return {



            // Paint resp.Rendering logic

            paint: function ( $element, layout ) {

                // Your rendering code goes here ...

            }

        };
    } );
```

define

`define` is a concept introduced by RequireJS to define dependencies in your JavaScript files. The idea is to load external dependencies before your main script gets executed (read more here).
In our example above we do not load any dependencies, but we'll do in further chapters of this tutorial.

paint

`paint` is the main method to render the visualization.

It will be called every time the visualization should be rendered, either because of new data from the server or because it has been re-sized.
The paint method receives two parameters, `$element` and `layout`.

Parameter	Description
`$element`	jQuery wrapper containing the HTML element where the visualization should be rendered.
`layout`	Data and properties for the visualization.

Structure of the .qext file

The basic structure of a .qext file looks as follows:

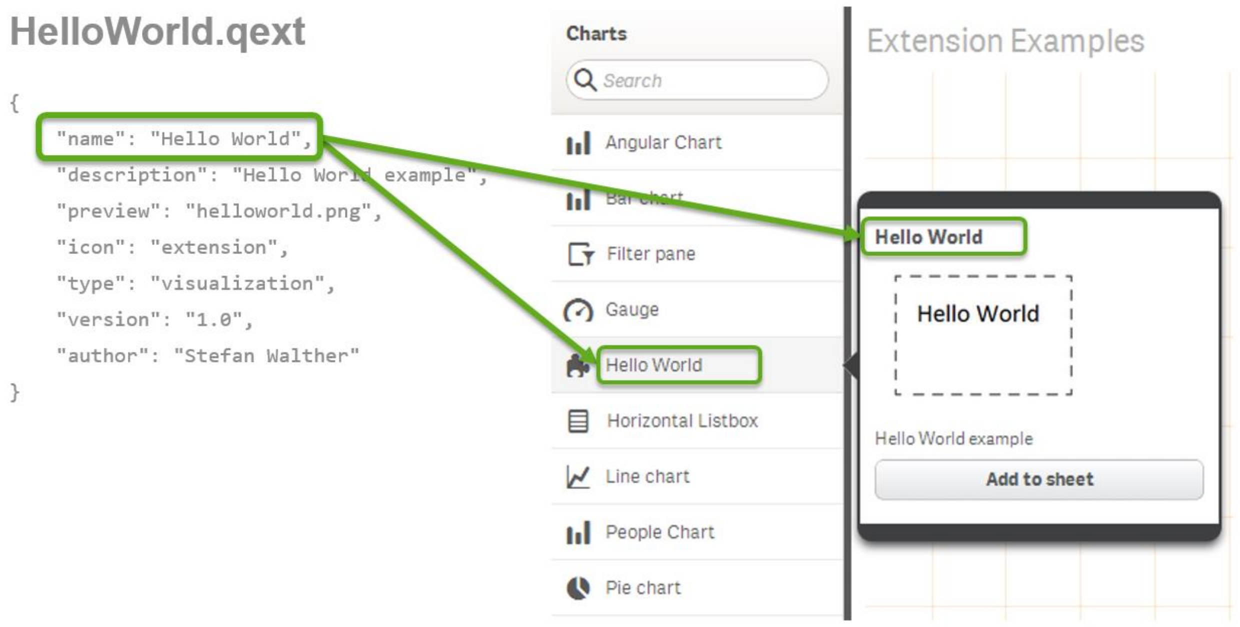
```
{

    "name" : "Extension Tutorial - Hello World",

    "description" : "Extension Tutorial, Chapter 3, Simple Hello World.",

    "icon" : "extension",

    "type" : "visualization",

    "version": "0.1",
    "preview" : "bar",

    "author": "Stefan Walther"

}
```

Hint

When creating your .qext file you should double-check if this file meets the requirements of a valid .json file.

name

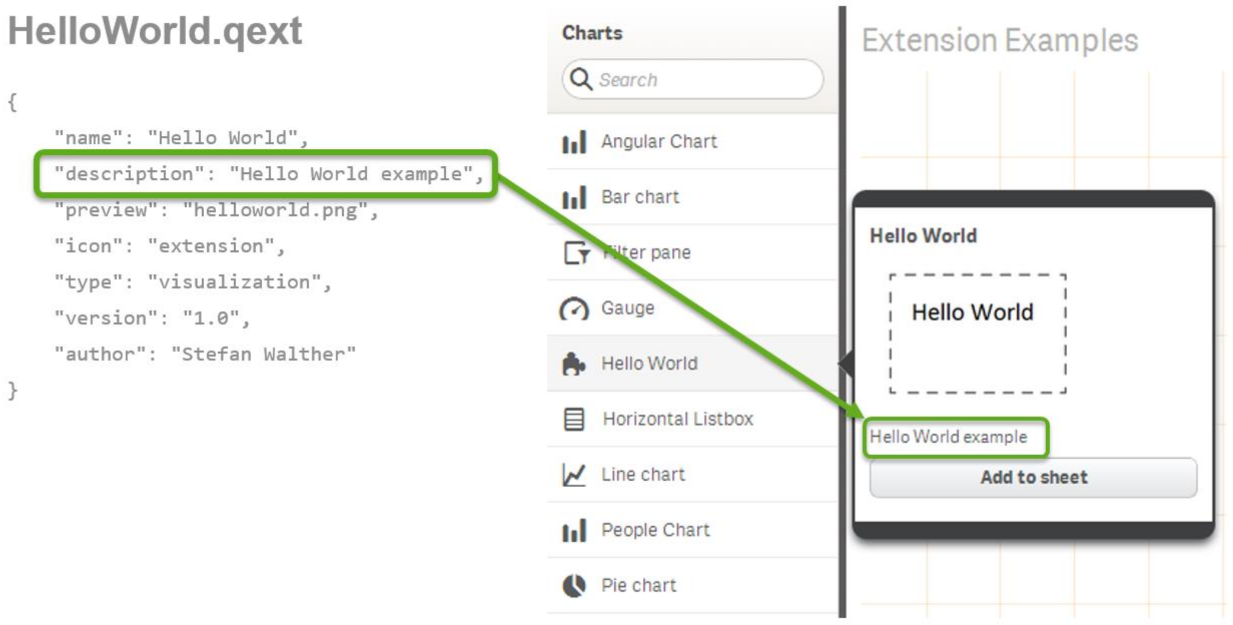
The property `name` will be re-used in the list of visualizations and the preview:


HelloWorld.qext

```
{
    "name": "Hello World",
    "description": "Hello World example",
    "preview": "helloworld.png",
    "icon": "extension",
    "type": "visualization",
    "version": "1.0",
    "author": "Stefan Walther"
}
```

description

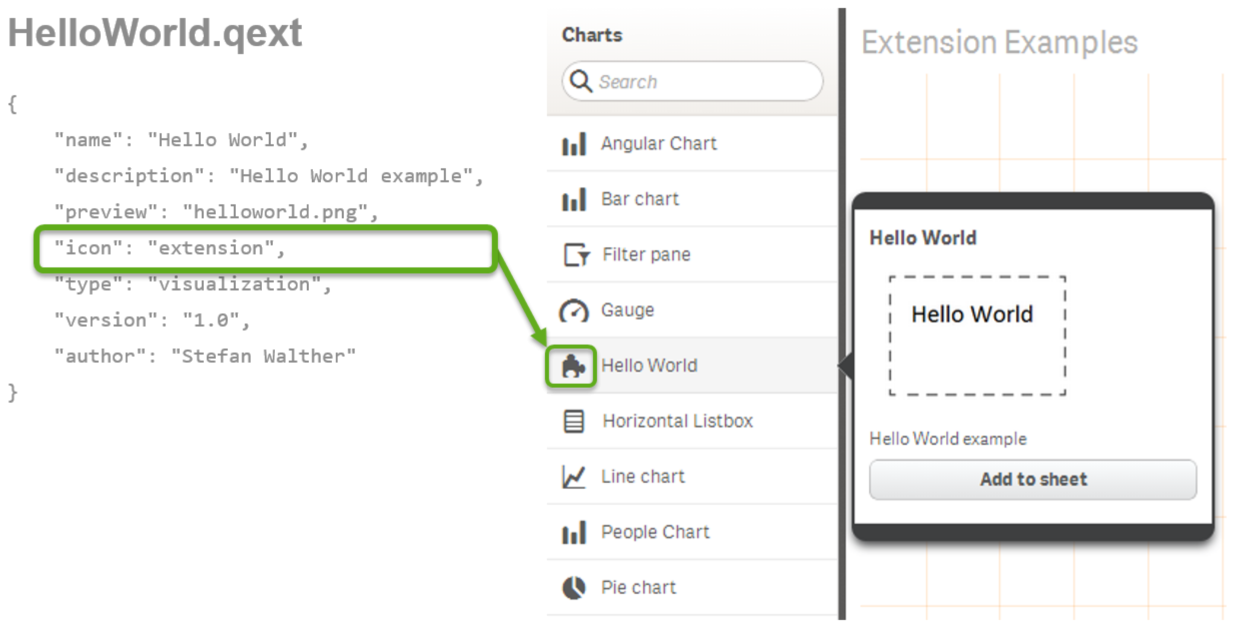
The `description` is visible in the preview:



HelloWorld.qext

```
{
    "name": "Hello World",
    "description": "Hello World example",
    "preview": "helloworld.png",
    "icon": "extension",
    "type": "visualization",
    "version": "1.0",
    "author": "Stefan Walther"
}
```

icon



HelloWorld.qext

```
{
    "name": "Hello World",
    "description": "Hello World example",
    "preview": "helloworld.png",
    "icon": "extension",
    "type": "visualization",
    "version": "1.0",
    "author": "Stefan Walther"
}
```

The following values are possible for setting the icon:

- `bar-chart-vertical`
- `combo-chart`
- `extension`
- `filterpane`

- `gauge-chart`
- `kpi`
- `line-chart`
- `list`
- `map`
- `pie-chart`
- `pivot-table`
- `scatter-chart`
- `table`
- `text-image`
- `treemap`

You can find more about the `icon` property in the official documentation.

type

Defines the type of the extension, should be always `visualization` as of now.

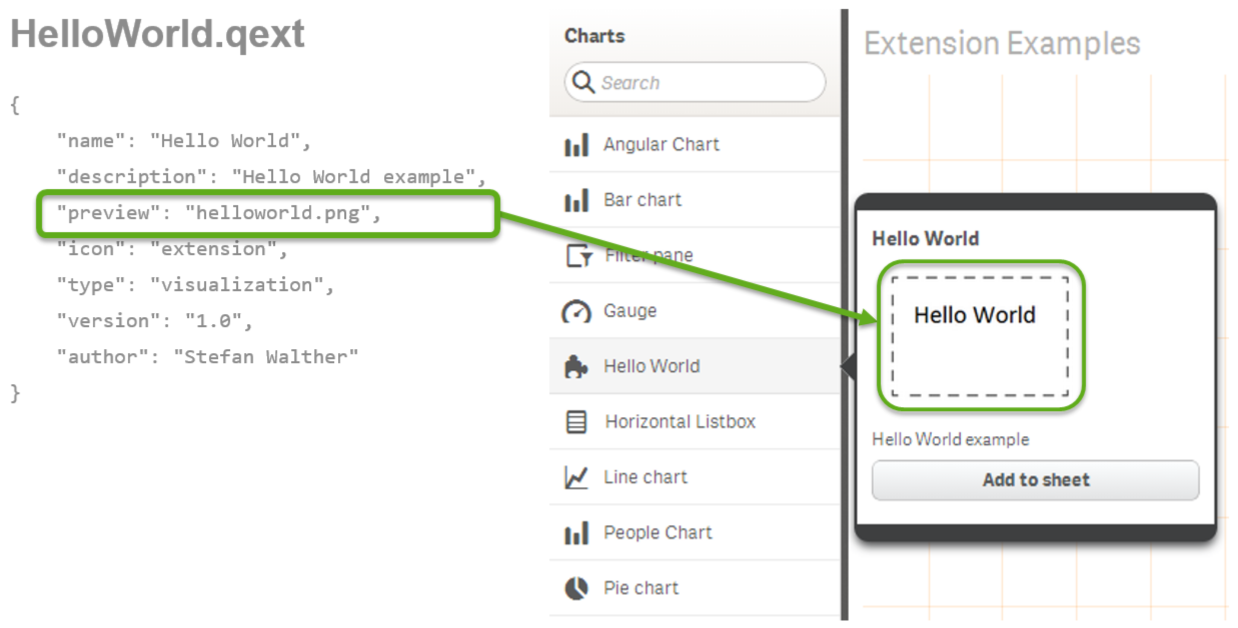
version

Define the version of your visualization extension, I recommend to use Semantic Versioning.

preview

Qlik Sense for Developers states:

You can define a custom preview image that is visible when the visualization has been deployed to Qlik Sense. The preview image is visible when selecting the visualization in the Library or Assets panel. This is done by defining the preview parameter in the qext file. If you do not define the preview parameter in the qext file, the icon definition will be used for rendering the preview image as well.



HelloWorld.qext

```
{
    "name": "Hello World",
    "description": "Hello World example",
    "preview": "helloworld.png",
    "icon": "extension",
    "type": "visualization",
    "version": "1.0",
    "author": "Stefan Walther"
}
```

Hint:

If you want to create a preview image, choose width and height of 140px x 123px.

author

author references the author of the visualization extension, so probably your name.
Note, as of this is not visible in neither the Qlik Sense Desktop nor Qlik Sense Server, so you can only get information about the author if you open the `.qext` file.

Creating the Hello World example

Considering the anatomy of a Qlik Sense Visualization Extension we can now create our "Hello World" example as follows:

Create the Container

The easiest way to get started is to use Qlik Sense Desktop and create a folder containing the two required assets where Qlik Sense Desktop is loading extensions from.

So let's create folder called "Extension Tutorial - Hello World" under

`C:\Users\[UserName]\Documents\Qlik\Sense\Extensions\`

Create a .qext file

Then create a `exttut-03-helloworld.qext` file and paste the following code into it:

```
{
    "name" : "Extension Tutorial - Hello World",

    "description" : "Extension Tutorial, Chapter 3, Simple Hello World.",

    "icon" : "extension",

    "type" : "visualization",

    "version": "0.1",

    "preview" : "bar",

    "author": "Your Name"

}
```

The Script File

Then add the following script to a file called `exttut-03-helloworld.js`:

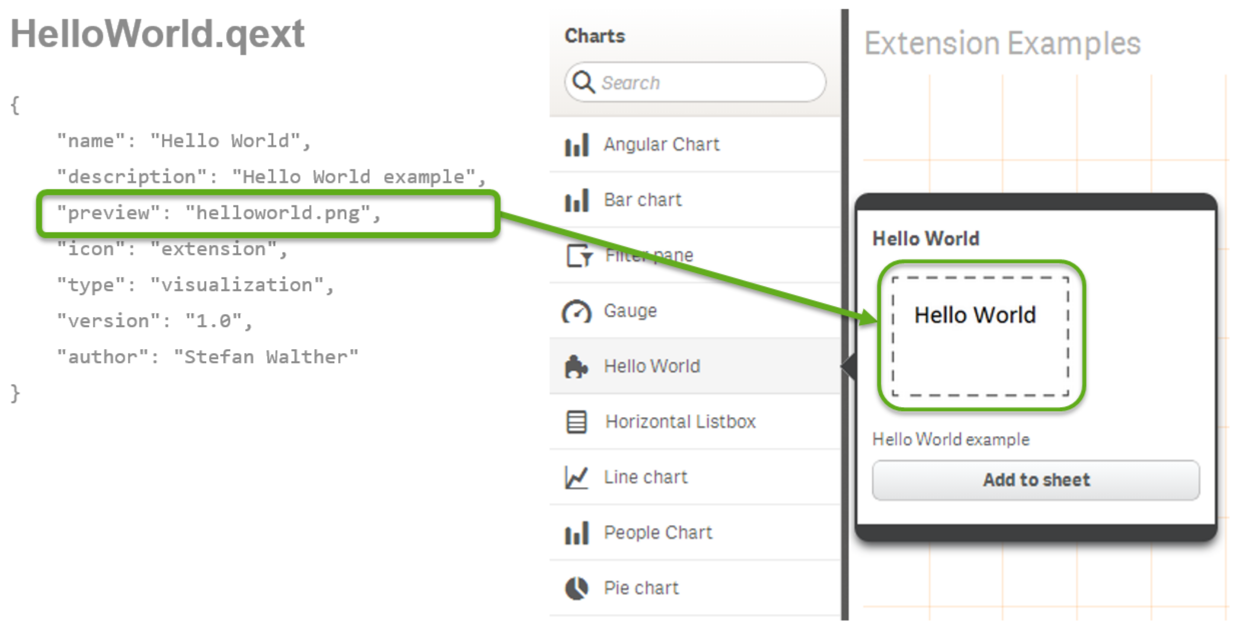
```
define( [

        'jquery'

    ],

    function ( $ ) {

        'use strict';



        return {



            //Paint resp.Rendering logic

            paint: function ( $element, layout ) {
                var $helloWorld = $( document.createElement( 'div' ) );

                $helloWorld.html( 'Hello World from the extension "01-ExtTut-HelloWorld"<br/>'
);

                $element.append( $helloWorld );



            }

        };

    } );
```

Test It

Before we dig into details what this code does, let's just test it.
Just follow these steps:

1. Open Qlik Sense Desktop
2. Open an existing app or create a new one
3. Open an existing sheet or create a new one
4. Go to *Edit mode* and then you should see the extension "Extension Tutorial - Hello World"
5. Drag and Drop it onto the sheet and exit the *Edit mode*

You should see something like that:



HelloWorld.qext

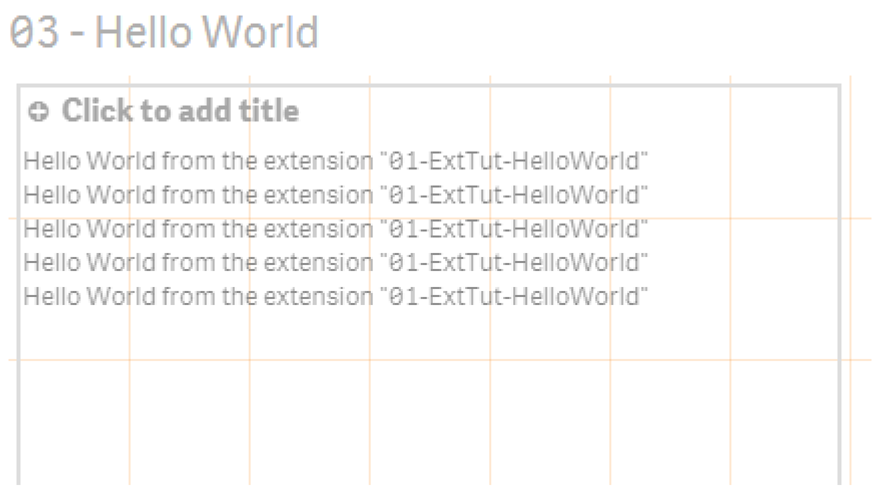
```
{
    "name": "Hello World",
    "description": "Hello World example",
    "preview": "helloworld.png",
    "icon": "extension",
    "type": "visualization",
    "version": "1.0",
    "author": "Stefan Walther"
}
```

Hint:

If this is the first time that you are working with Qlik Sense (Desktop) I've added a step by step guide to the appendix which will guide you through creating your first app and testing this extension.

But wait, something went wrong

Before jumping to the next chapter you'll probably realize that there is something wrong with our solution.
As soon as you resize the (browser-)window you'll recognize that our output get multiplied, so you'll end up into something like that:



```
Hello World from the extension "01-ExtTut-HelloWorld"
Hello World from the extension "01-ExtTut-HelloWorld"
Hello World from the extension "01-ExtTut-HelloWorld"
Hello World from the extension "01-ExtTut-HelloWorld"
Hello World from the extension "01-ExtTut-HelloWorld"
```

So why did this happen?
The answer is quite easy. Have a look above to the description of the `paint`. This method will always be called when the visualization should be rendered, and a resize triggers the paint.
So in fact we are appending a new `$helloWorld` object to `$element` on every resize.

There are several ways how to solve this, I'll introduce two of them:

1) Be lazy, just remove existing content

You can remove all child nodes of `$element` at the beginning for your `paint` method (by using the jQuery empty() method).

Just add the following line at the beginning of `paint`

```
$element.empty();
```

2) Be smarter, detect if the your new node already exists

```
var id = layout.qInfo.qId + '_helloworld';

var $helloWorld = $( '#' + id );

if ( !$helloWorld.length ) {

    console.log( 'No element found with the given Id, so create the element' );

    $helloWorld = $( document.createElement( 'div' ) );

    $helloWorld.attr( 'id', id );

    $helloWorld.html( 'Hello World' );

    $element.append( $helloWorld );

} else {

    console.log( 'Found an element with the given Id, so just change it' );

    $helloWorld.html( 'Hello World' );

}
```

Reviewing this example you'll also recognize something new, the usage of the `layout` object to get the unique Id of the current object (too prevent conflicts if you are using the same object several times on a sheet). We'll talk more about `layout` in upcoming chapters.

Performance Impact ?
Scared about the performance impact of the first approach? I was too, so I did some basic tests: http://jsperf.com/emptyorreuse

#4: Debugging and Web Developer Tools

Before we continue improving our Hello World example I think it's now the right moment to talk a little bit about debugging.

In old days JavaScript developers often used `alert()` to bring some debug messages to the front. Certainly you can still use this approach, but it becomes very soon more than annoying ...
So there is a much better way: Using `console.log()` to send data to the **browser console**.

Web Developer Tools

All modern browser offer some kind of a *Developer Tool*, Chrome is probably one step ahead their competitors, so I'll introduce some basic concepts here based on Chrome.

First of all you'll find a very good introduction to Chrome's *Web DevTools* here.

Developing using Qlik Sense Desktop & Chrome?

It is not in contradiction to using Qlik Sense Desktop to develop your visualization extensions and using the browser's developer tools at the same time.

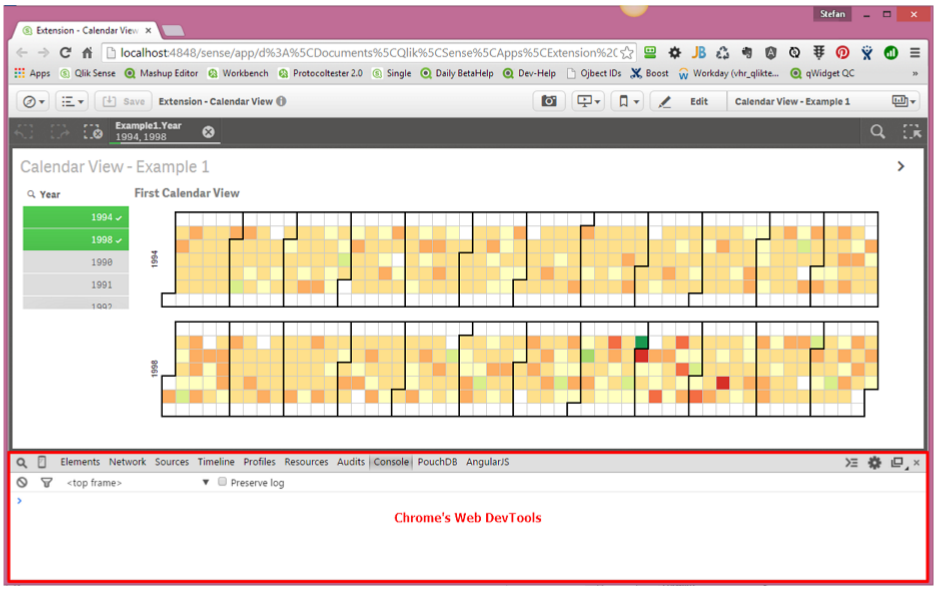
Using your favorite browser

Even when developing using Qlik Sense Desktop you can use your favorite browser for debugging purposes at the same time:

1. Open Qlik Sense Desktop (and leave it opened)

2. Open your favorite browser and open `http://localhost:4848/hub`

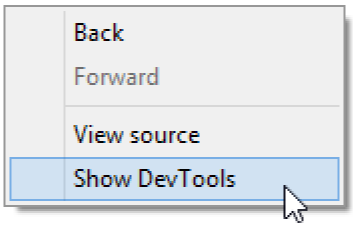
You'll see Qlik Sense' Hub and can open your browser's developer tools (most of the time by pressing `F12`).



Using developer tools within Qlik Sense Desktop

Since Qlik Sense Desktop is using Chromium as an embedded browser, you can also open Chrome's Web DevTools within Qlik Sense Desktop:

By using `Ctrl`+`Shift` and right mouse click you'll get the following dialog to activate *DevTools*:



Chrome Web DevTools

There are a bunch of websites out there introducing the capabilities of developer tools, I just want to highlight three areas which are highly relevant when developing visualization extensions in Qlik Sense:

- Using the console
- Inspect elements
- Debugging

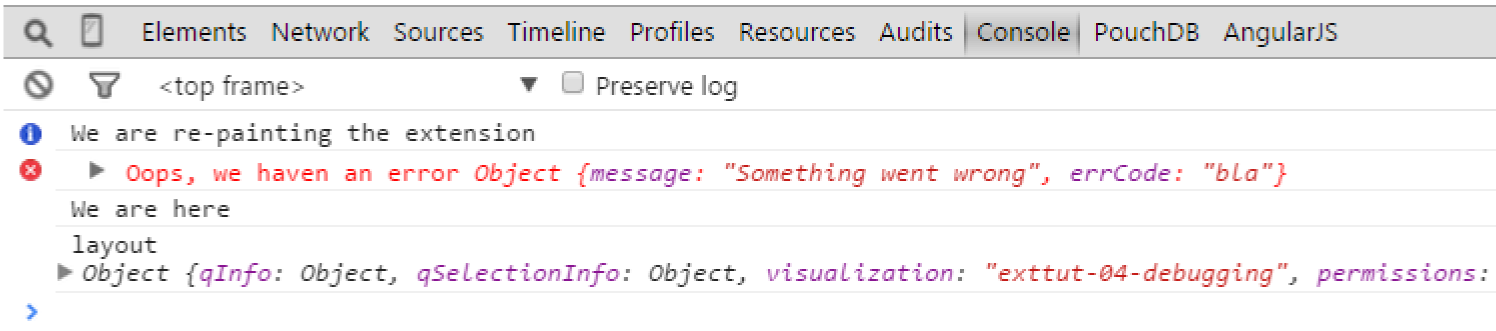
Using the console

As mentioned before, instead of triggering alerts, it's much more efficient and convenient to use the console of *Web DevTools*:

Just use `console.info` or `console.log` to push something to the console:

```
paint: function ( $element, layout ) {



    var err = {

        message: 'Something went wrong',

        errCode: 'bla'

    };



    console.info( 'We are re-painting the extension' );

    console.error( 'Oops, we haven an error', err );

    console.log( 'We are here' );

    console.log( 'layout', layout );



}
```

results into



```
We are re-painting the extension
▶ Oops, we haven an error Object {message: "Something went wrong", errCode: "bla"}
We are here
layout
▶ Object {qInfo: Object, qSelectionInfo: Object, visualization: "exttut-04-debugging", permissions:
```

The console is very powerful, I highly recommend that you spend some time with more "advanced" concepts of the console like:

```
console.assert()
console.group()


console.groupCollapsed()


console.groupEnd()


console.table()
```

Further readings:

- https://developer.chrome.com/devtools/docs/console
- http://anti-code.com/devtools-cheatsheet/

But ...

Unfortunately not all browsers support `console.xxx()` so we'll definitely have to ensure that in a production environment there are no`console.xxx()` left in the code.

There are several strategies for achieving that:

1. Remove all `console.xxx()` manually
   Not really a nice solution, but certainly fine for the beginning

2. Wrap all your calls to `console` that they are only executed if the current browser supports `console` (here's a Gist you can use)

3. Use tools like Grunt or Gulp to create a deployment process where all console statements will be removed automatically.

I personally prefer option 3 because it keeps the deployed code clean (I'll talk about the approach I have chosen and developed over time in one of the later chapters).

Inspect Elements

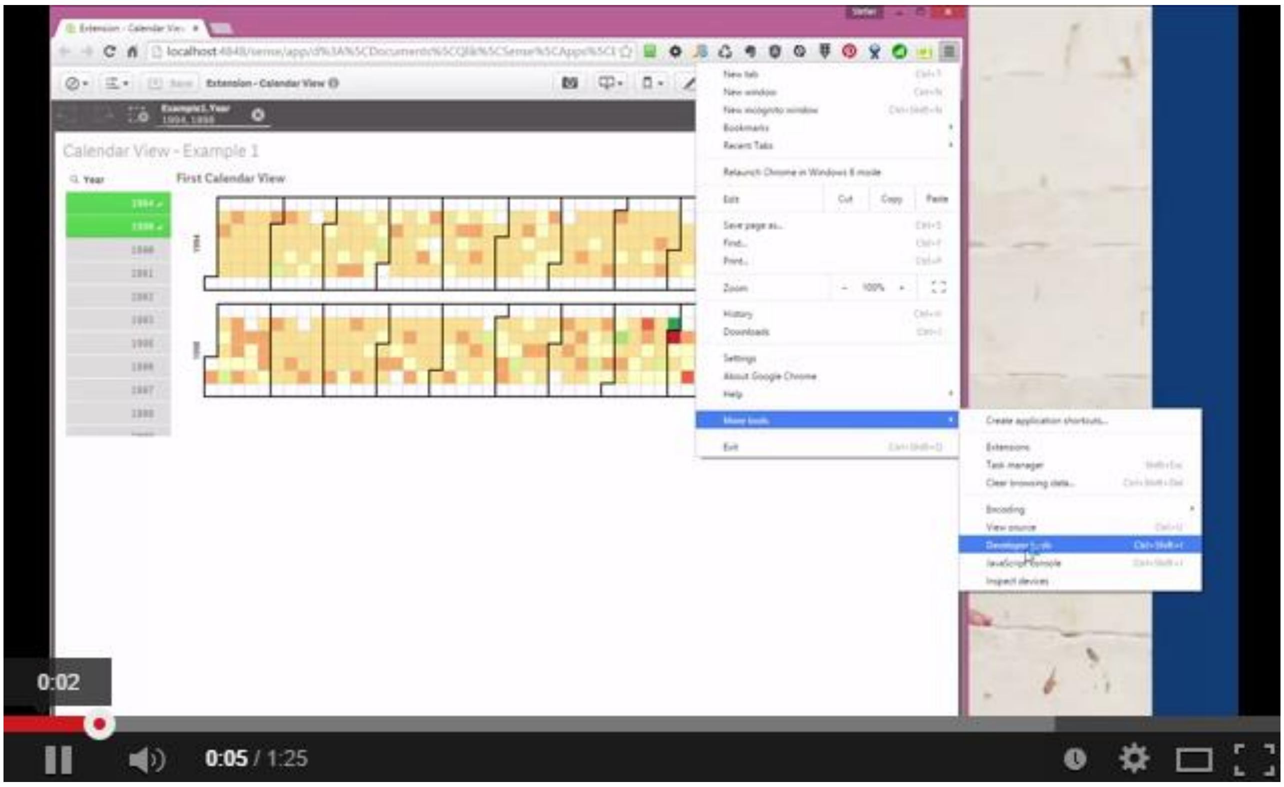
Probably even more important than the console output is the possibility to review how you have manipulated the DOM, so HTML and CSS.

If it is unclear for you what "DOM manipulation" is, here are some good articles:

- Short description on stackoverflow
- Good short tutorial

Having an inspector is a common capability of all Web Developer Tools. In Chrome WebDev Tools just click on the lens:

Have a look at the following short video:



More detailed information: https://developer.chrome.com/devtools/docs/dom-and-styles

Debugging

Finally you can also use Developer Tools for live debugging:

Read more on:

- Debugging with Chrome DevTools site
- Debugging with Firebug
- Debugging with Internet Explorer Developer Tools

Hint:

Very similar to the advice above regarding `console.x()`, do not forget to remove `debugger;` statements if you move your visualization extension to production environment.

#5: Improving the Hello World Experience

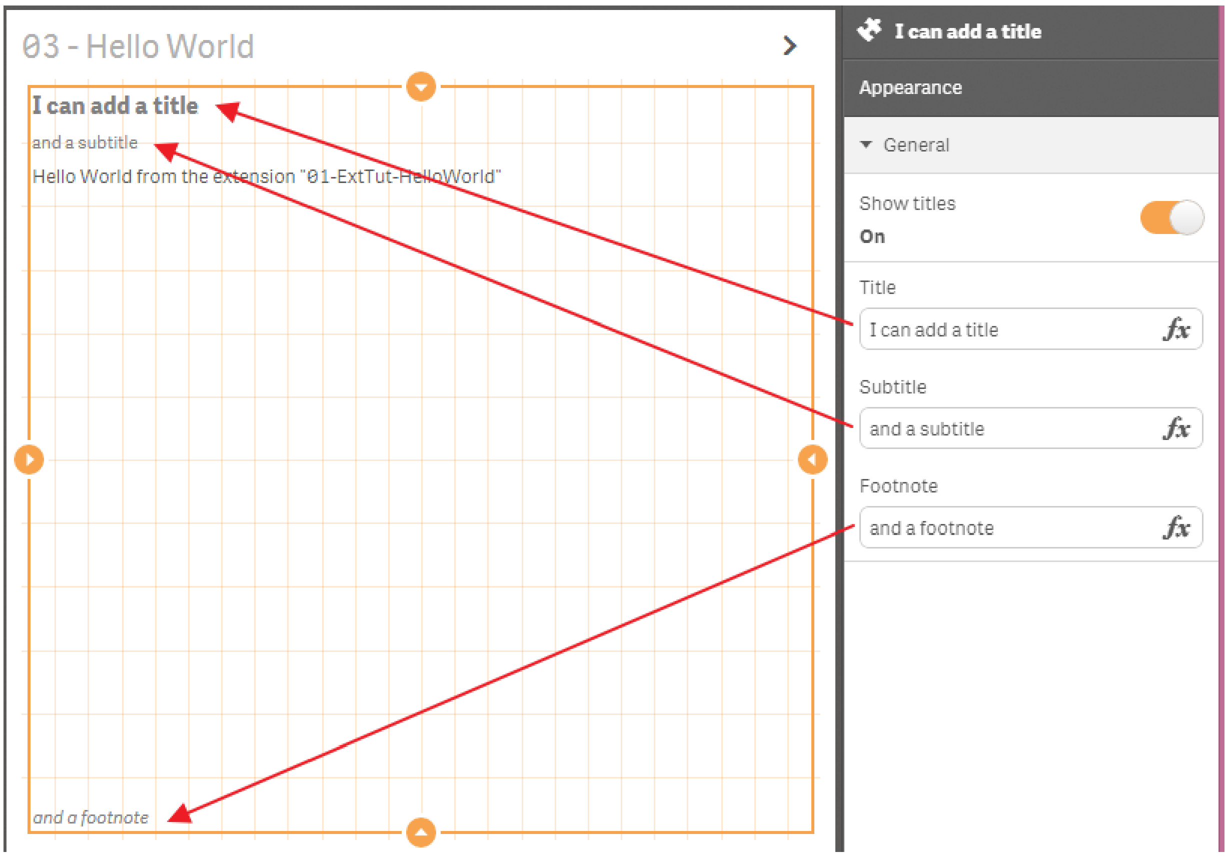
OK, let's improve the *Hello World* experience.
In this chapter we'll make the *Hello World* example a bit more dynamic:

- Changing the behavior by using the property panel:
    - Define the text displayed (not a hard coded "Hello World" anymore)
- Create a preview image to be displayed when clicking on the object in (left) object panel

Making "Hello World" Dynamic

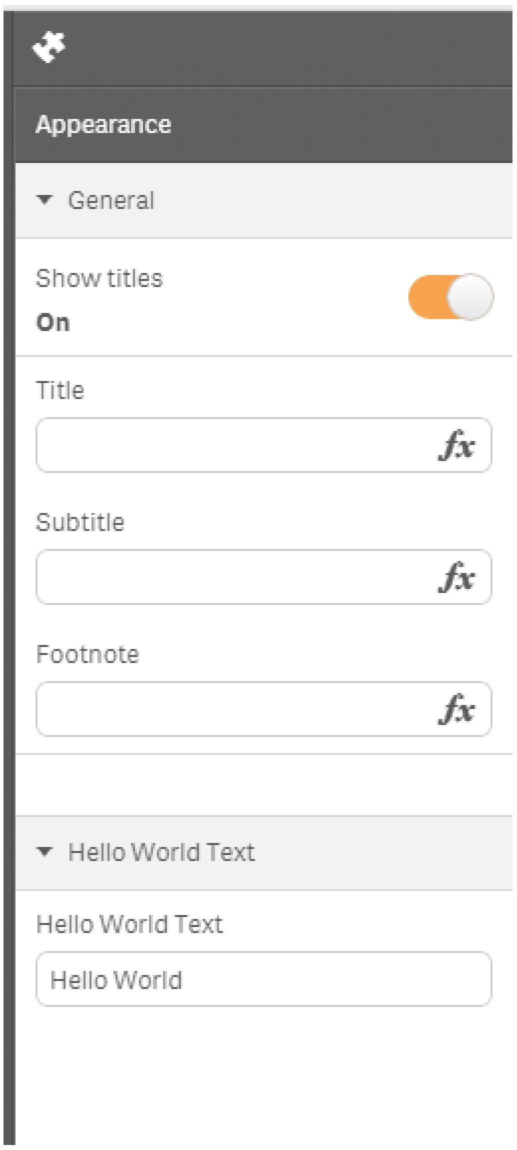
When building *Visualization Extensions* it is essential that a non-developer can use and configure the object like any other native object of Qlik Sense.
Therefore you can define which properties are exposed to the user in the (right) property panel and add additional custom properties which can then be used in your code.

If you have a look at the output of our very first "Hello World" example you'll recognize that the property panel looks as follows:



The "Appearance" accordion is enabled by default and available out of the box without doing anything. Now we'd like to inject just another text box into this accordion where we can define the output rendered:

Desired result:


Appearance

General

Show titles
On

Title

Subtitle

Footnote

Hello World Text

Hello World Text

Hello World

Therefore we add another object to the skeleton of the script file, called `definition`:

```
define( [ /* dependencies */ ],

    function ( /* dependency arguments */ ) {

        'use strict';

        return {

            // Define how our property panel looks like

            definition: {

            },

            // Paint resp.Rendering logic

            paint: function ( $element, layout ) {

                // Your rendering code goes here ...

            }

        };

    } );
```

Inside `definition` you can define how the property panel should look like, basically you can define the accordion, its sections and components to be used.
The default `definition` loads a reusable component called `settings`, therefore we see the "Appearance" accordion:

```
definition: {

    type: "items",

    component: "accordion",

    items: {

        appearancePanel: {

            uses: "settings"

        }

    }

}
```

The line `uses: "settings"` defines here that the re-usable component `settings` should be used. I'll tell you more about other re-usable components in one of the upcoming chapters.

A custom string property

To add a text box into the "Appearance" accordion, use the following code:

```
...

definition: {

    type: "items",

    component: "accordion",

    items: {

        appearancePanel: {

            uses: "settings",

            items: {

                MyStringProp: {

                    ref: "myDynamicOutput",

                    type: "string",

                    label: "Hello World Text"

                }

            }

        }

    }

},

...
```

Code Explanation:

- `MyStringProp` is the representation of our new custom object
  - `ref` defines the name to reference the new property in the code
  - `type` defines the type definition, in our case we want to have a string returned
  - `label` is used for displaying the label above the text box

Good, we have now added a new property to the property panel where we can enter a value for our new, improved "Hello World" example:

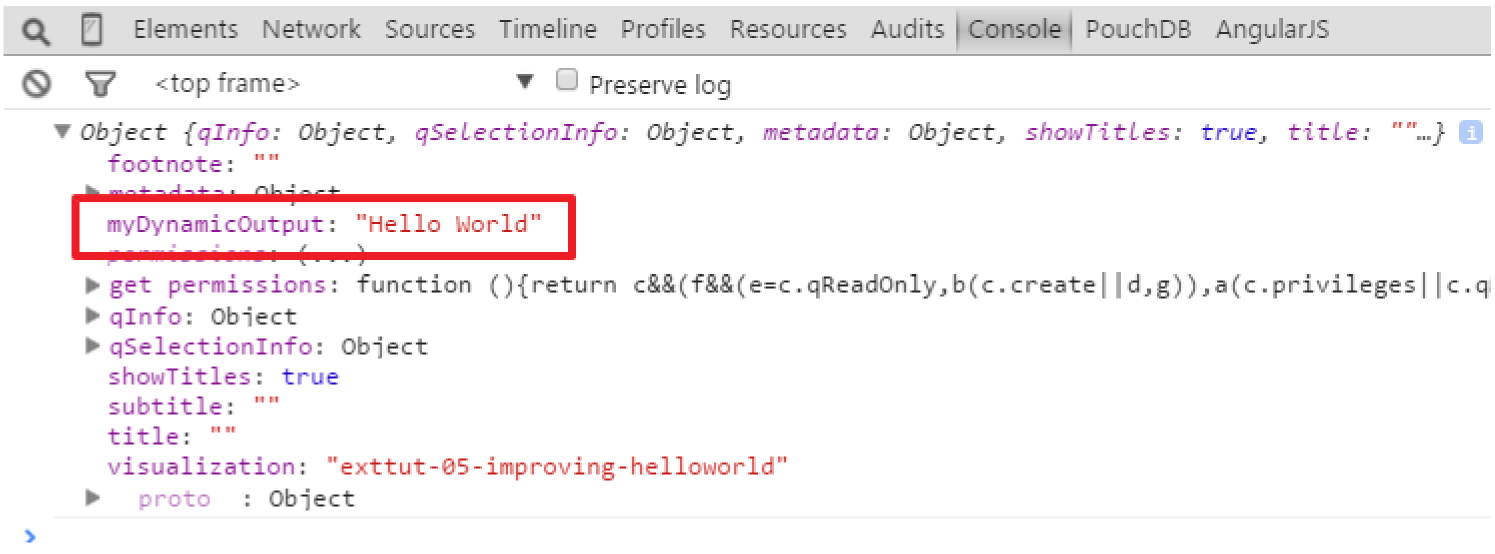
Hint:

If you are making changes in your script file and then testing it either in your browser or within Qlik Sense Desktop, do not forget to re-load the page. (In Qlik Sense Desktop and most browsers just by hitting the key `F5`)

Use the custom string property

Now let's modify the code to render what we enter in our text box. Before doing so let's go back to our code and add a console output to double-check where we can find the object to reference in our code:

```
// put this inside at the beginning of your paint method


console.log(layout);
```



```
▼ Object {qInfo: Object, qSelectionInfo: Object, metadata: Object, showTitles: true, title: ""…}
    footnote: ""
  ▶ metadata: Object
    myDynamicOutput: "Hello World"
    permissions: (...)
  ▶ get permissions: function (){return c&&(f&&(e=c.qReadOnly,b(c.create||d,g)),a(c.privileges||c.q
  ▶ qInfo: Object
  ▶ qSelectionInfo: Object
    showTitles: true
    subtitle: ""
    title: ""
    visualization: "exttut-05-improving-helloworld"
  ▶ __proto__: Object
>
```

The rest of this exercise is easy, instead of hard-coding the result

```
$element.empty();


var $helloWorld = $( document.createElement( 'div' ) );
```

```
$helloWorld.html( 'Hello World from the extension "05-ExtTut-HelloWorld"<br/>' );

$element.append( $helloWorld );
```

we make it dynamic by using `layout.myDynamicOutput`

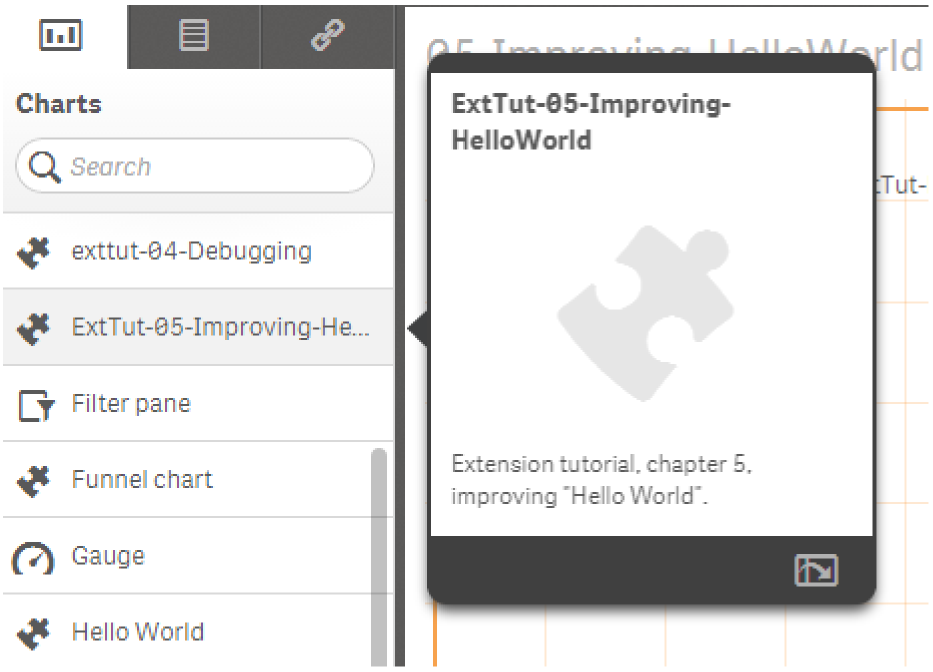
```
$element.empty();

var $helloWorld = $( document.createElement( 'div' ) );

$helloWorld.html( layout.myDynamicOutput );

$element.append( $helloWorld );
```

Hint:

If you are making changes to `ref`, refreshing the browser does not reflect the changes. You have to delete an existing object and re-add it to the sheet. Therefore double-checking the returned properties using `console.log(...)` is always a good advice.

Adding a preview image

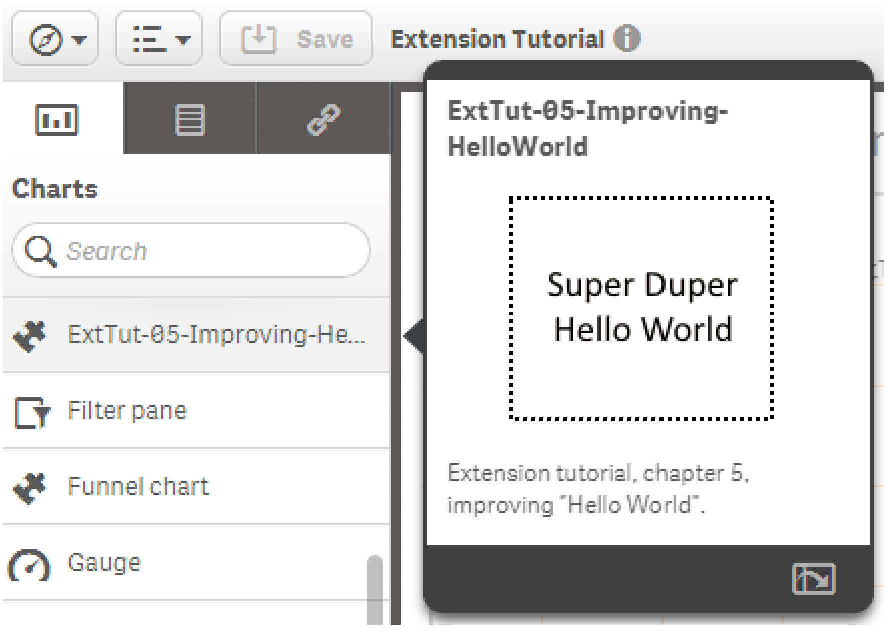
The last exercise is easy, we want to add a good looking preview image if you click on the object in the Library or Assets panel.


Charts
Search
exttut-04-Debugging
ExtTut-05-Improving-He...
Filter pane
Funnel chart
Gauge
Hello World
05-Improving-HelloWorld
ExtTut-05-Improving-HelloWorld
Extension tutorial, chapter 5, improving "Hello World".

For modifying the displayed preview image, do the following:

- Create a new image and save it as .png file in the folder of your extension
- Open the .qext file and change it by adding the line preview as shown below

Our final result:


Save
Extension Tutorial
Charts
Search
ExtTut-05-Improving-He...
Filter pane
Funnel chart
Gauge
ExtTut-05-Improving-HelloWorld
Super Duper Hello World
Extension tutorial, chapter 5, improving "Hello World".

Hint:

As of version 1.0 or 1.1 of Qlik Sense the expected dimensions of the image are not clearly defined. It works best if you choose 140x123 pixels or a multiple of this ratio.

Congrats, you have now created an **improved version** of your "Hello World" example, although there is certainly still room for improvement ;-)

#6: Introduction to Using Properties

Before we start with the real fun stuff, let's spend a chapter or some minutes to understand the basic principles behind defining and using properties in Qlik Sense *Visualization Extensions*.

This chapter only focuses on the basic concept of properties, a detailed overview of all possibilities will be added later on in one of the advanced chapters.

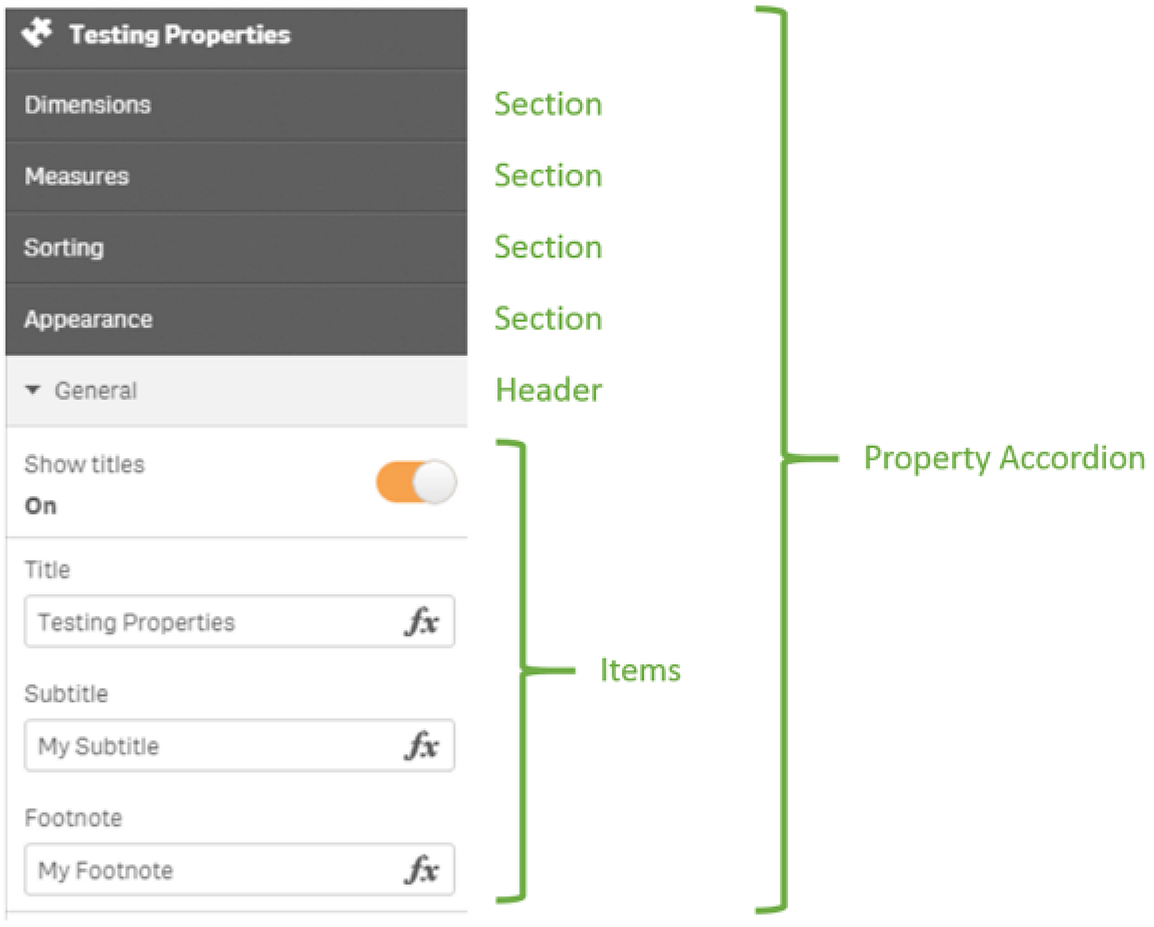
The Idea Behind Properties

The main idea behind properties is to offer users a way to customize the behavior of *visualization extensions* in the same way as they would control Qlik Sense' native objects. Therefore an extension developer can use a programmatic interface to define properties which are then displayed in the right property panel the exactly same way as it is the case with native objects.

The property panel

If you have a look at the property panel accordion you'll recognize that properties are structured in a hierarchy as follows:

- **Accordion**
  - Accordion **sections**
    - Section **headers**
      - Property **items**



Show titles
On

Title: Testing Properties

Subtitle: My Subtitle

Footnote: My Footnote

Property panel definition in your JavaScript file

As already mentioned in the previous chapter, properties can be added by defining them in the `definition` property of the extension's JavaScript file:

Built-in vs. custom properties

There are two main approaches how you can define properties:

1. **Re-use existing** (built-in) **properties**
2. Create **custom properties**

Re-use existing properties

If you have spent some time with Qlik Sense, you already know the concept of typical properties used for the most common native objects:

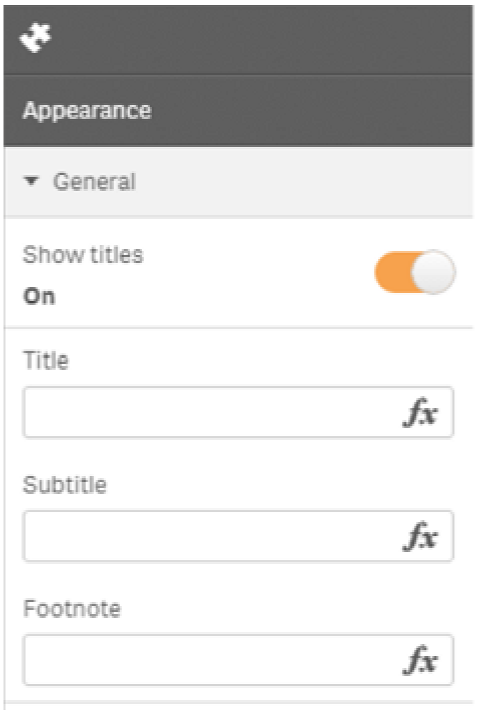
- Every native object has a section called "Appearance" and a section header called "General"
- Most of the native objects allow "Dimensions" and "Measures" to be defined

If you create a visualization extension without specifying any specific properties to be shown in the property panel you'll get the following:

Code:

```
define( [

        'jquery'

    ],

    function ( $ ) {

        'use strict';



        return {



            paint: function ( $element /*, layout*/ ) {
                $element.empty();

                var $msg = $( document.createElement( 'div' ) );

                $msg.html( 'Just demonstrating default behavior of the property panel"' );

                $element.append( $msg );



            }

        };



    } );
```

Result:



So getting the "Appearance" section is the default behavior.

You'll get the same result by telling Qlik Sense to re-use the section "settings" (which is the internal name for the "Appearance" section.

```
define( [

        'jquery'

    ],

    function ( $ ) {

        'use strict';



        return {



            definition: {

                type: "items",

                component: "accordion",

                items: {

                    appearance: {

                        uses: "settings"

                    }

                }

            },



            paint: function ( $element /*, layout*/ ) {
                $element.empty();

                var $msg = $( document.createElement( 'div' ) );

                $msg.html( 'Just demonstrating default behavior of the property panel"' );

                $element.append( $msg );



            }

        };



    } );
```

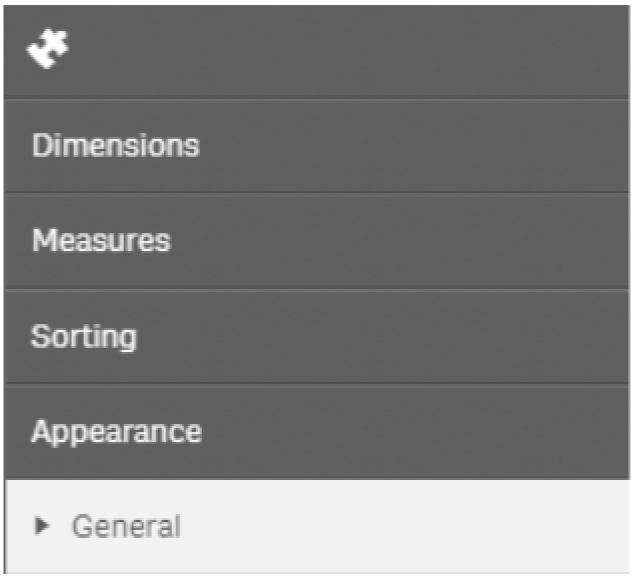
Based on this example we can now extend `definition` to re-use other built-in sections.

Code:

```
// --

// Reuse the following sections

//   - dimensions

//   - measures

//   - sorting

//   - settings ("Appearance" section)

// --
```

```
definition: {

    type: "items",

    component: "accordion",

    items: {

        dimensions: {

            uses: "dimensions"

        },

        measures: {

            uses: "measures"

        },

        sorting: {

            uses: "sorting"

        },

        appearance: {

            uses: "settings"

        }

    }

}
```

Result:



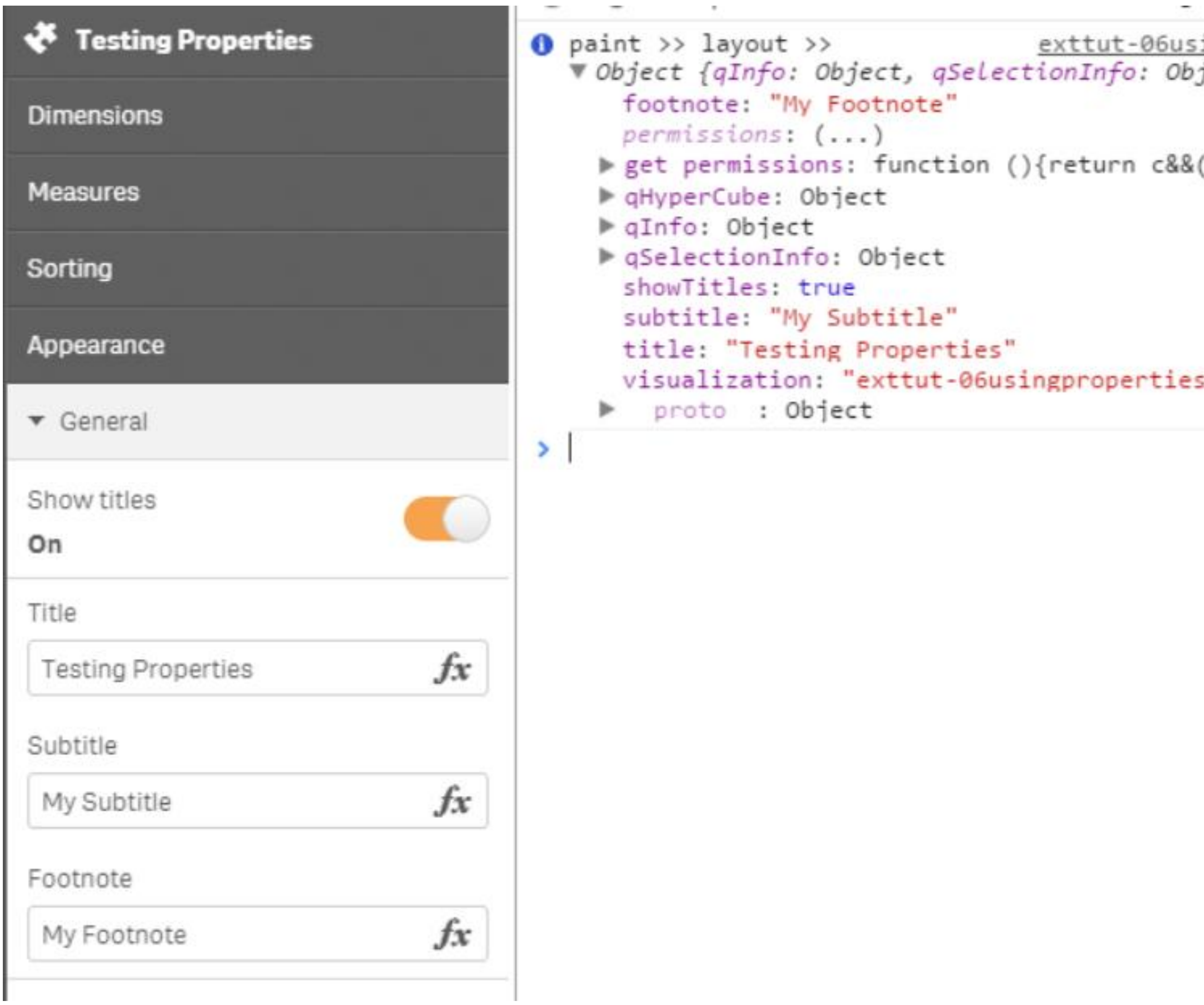
Referencing properties

As we have now defined some properties to use in our extension, let's have a look how to reference the properties in our code.

We haven't covered this so far: A second parameter called `layout` will be passed to the `paint` method, which includes the current scope the extension object, also "holding" the properties we have defined.

```
// Make sure to add some dimensions and measures to your extension first

// (This will give you a better picture what layout can contain ...)

paint: function ( $element, layout ) {



    console.info('paint >> layout >> ', layout);



}
```

If you have a look at the console output of e.g. Chrome's Dev Tools, you can easily find what you are looking for:



```
paint >> layout >>                                   exttut-06us
▼ Object {qInfo: Object, qSelectionInfo: Obj
    footnote: "My Footnote"
    permissions: (...)
  ▶ get permissions: function (){return c&&(
  ▶ qHyperCube: Object
  ▶ qInfo: Object
  ▶ qSelectionInfo: Object
    showTitles: true
    subtitle: "My Subtitle"
    title: "Testing Properties"
    visualization: "exttut-06usingproperties
  ▶   proto   : Object
>
```

So let's output some values in our `paint` method:

```
paint: function ( $element, layout ) {



    // Output values from the property panel

    $element.empty();



    // Our output container
    var $msg = $( document.createElement( 'div' ) );



    // Variable holding the output

    var html = '<b>Property values:</b><br/>';

    html += 'Title: ' + layout.title + '<br/>';

    html += 'SubTitle: ' + layout.subtitle + '<br/>';



    // Assigning the variable to our output container

    $msg.html( html );



    // Adding the output container to the current element

    $element.append( $msg );



}
```

Great, you now can use built-in properties in your *visualization extension* in the next chapter we'll explore how to customize the property panel to get our very custom properties into there to allow users to configure the behavior of your *visualization extension*.

#7: Custom Properties

In the previous chapter you have learned how to re-use built-in properties in Qlik Sense' property panel. In this chapter we'll cover how we can extend the property panel to custom needs.

Introduction to custom properties

If you need other properties than the predefined ones you can create them as desired by adding *custom properties*.

As of Qlik Sense 1.1/2.0 you can use a collection of different UI components to display your custom properties in the property panel:

- Check box
- Input box / Text box
- Drop down list
- Radio button
- Button group
- Switch
- Slider
- Range-slider

These UI components can be grouped into sections and headers in the property panel.
Each of them exposes different configuration options to manipulate their behavior (I recommend to look into the official "Qlik Sense for Developers" documentation for more details on each of the components).

Improving readability & maintainability of properties

In previous chapters we have defined properties in the main script's `definition` property. That's fine, but if you have a lot of different properties and you are defining them in your main JavaScript file you'll recognize that your script file becomes quite big and hard to read and maintain.

There are several approaches to improve this situation:

- Split the definition into several JavaScript variables
- Break out the definition into a separate file and load this file in your extension's main file

The online help "Qlik Sense for Developer" nearly always uses the approach of splitting logical pieces into separate variables, so it's good to understand how this works.
I personally recommend to combine both approaches, which helps me a lot to make all my extensions easier to maintain.

Initial situation:

```
define( [],

    function ( ) {

        'use strict';



        return {

            definition: {

                type: "items",

                component: "accordion",

                items: {

                    dimensions: {

                        uses: "dimensions"

                    },
                    measures: {

                        uses: "measures"

                    },

                    appearance: {

                        uses: "settings"

                    }

            },

            paint: function ( $element , layout ) {

                // Your main rendering logic here

            }

        }

    }

);
```

So let's start refactoring:

Separate properties file:

Create a file called "properties.js" and put it into the same folder where the main file of your visualization extension is located. In this very simple example the file will look as follows:

```
define( [], function () {

    'use strict';
```

```
// *****************************************************************************

// Dimensions & Measures

// *****************************************************************************

var dimensions = {

    uses: "dimensions",

    min: 0,

    max: 1

};



var measures = {

    uses: "measures",

    min: 0,

    max: 1

};



// *****************************************************************************

// Appearance Section

// *****************************************************************************

var appearanceSection = {

    uses: "settings"
    };



    // ****************************************************************************

    // Main property panel definition

    // ~~

    // Only what's defined here will be returned from properties.js

    // ****************************************************************************



    return {

        type: "items",

        component: "accordion",

        items: {

            dimensions: dimensions,

            measures: measures,

            appearance: appearanceSection



        }

    };


} );
```

Your main script:
In your main script load the external definition of properties and assign it to the properties' `definition`:

```
define( [

        // Load the properties.js file using requireJS

        // Note: If you load .js files, omit the file extension, otherwhise

        // requireJS will not load it correctly

        './properties'

    ],

    function ( props ) {

        'use strict';



        return {



            definition: props,

            paint: function ( $element , layout ) {



            }

        }

    }
);
```

Comparing these two examples it's not quite obvious why this is actually an improvement, but as soon as we add more to the property panel definition, you'll recognize the advantages of this approach.

Basic custom property - a string input box

Let's have a look at the most basic but at the same time also most common component, the text box and how to define it:

```
// Text box definition

var myTextBox = {

    ref: "props.myTextBox",

    label: "My Text Box",

    type: "string"

};
```

In the definition of every UI component there are some common properties which can be set:

Property	Description
`type`	Used for all custom property type definitions. Can be either `string`, `integer`, `number` or `boolean`.
`ref`	Name or Id used to reference a property.
`label`	Used for defining the label that is displayed in the property panel.
`component`	Used for defining how the property is visualized in the property panel.

Hint:

You may be wondering yourself why we haven't defined the component in the example above: In case of a UI component of type string or integer without defining a specific component Qlik Sense defaults automatically to a text box.

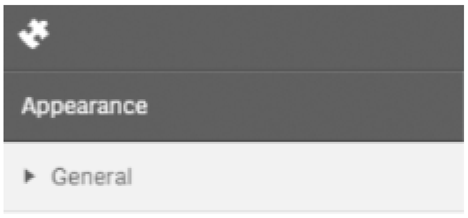
Adding the component definition to the property panel

As we have now defined the custom property - in our case a string based text box - we now have to add it to the accordion. The easiest approach to achieve that is to add the new custom property to the built-in "Appearance" section:

Code before adding `myTextBox`:

```
// Re-using the appearance section

var appearanceSection = {

    uses: "settings"

};



// Return overall definition of the property accordion

return {

    type: "items",

    component: "accordion",

    items: {

        appearance: appearanceSection

    }

};
```

Result:

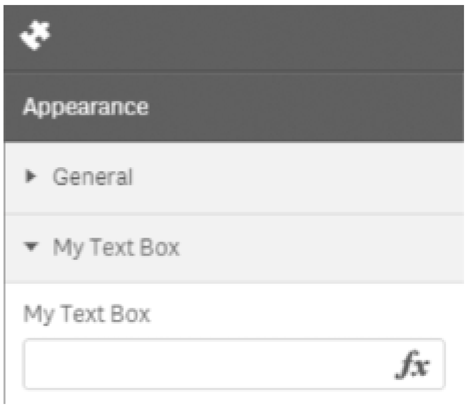


Code after adding `myTextBox`:

```
// Text box definition

var myTextBox = {

    ref: "props.myTextBox",

    label: "My Text Box",

    type: "string"

};



// Appearance section, now re-using the appearance section + injecting our myTextBox component

var appearanceSection = {

    uses: "settings",

    items: {

        myTextBox: myTextBox

    }

};



// Return overall definition of the property accordion
```

```
return {

    type: "items",

    component: "accordion",

    items: {

        appearance: appearanceSection

    }

};
```

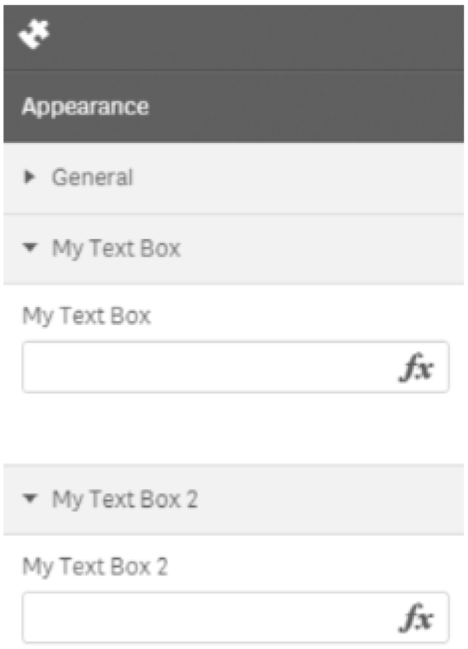
Result:



So what happened here?
You'll realize that not only our text box component was add but also a header was created automatically re-using the label of our component. Let's verify this by just adding another text box component:

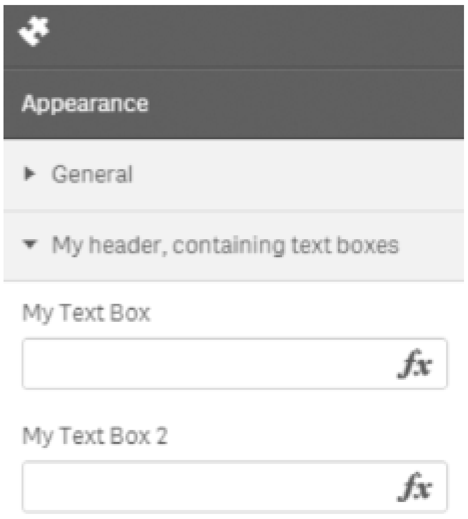
```
var myTextBox = {

    ref: "props.myTextBox",

    label: "My Text Box",

    type: "string",

    expression: "optional"

};
```

```
var myTextBox2 = {

    ref: "props.myTextBox2",

    label: "My Text Box 2",

    type: "string",

    expression: "optional"

};



// Appearance sectiion, now with two text boxes

var appearanceSection = {

    uses: "settings",

    items: {

        myTextBox: myTextBox,

        myTextBox2: myTextBox2

    }

};



// Return overall definition of the property accordion

return {

    type: "items",

    component: "accordion",
    items: {

        appearance: appearanceSection

    }

};
```



Adding a custom section header

That's fine, but in many case that's probably not the result we'd like to achieve, it's probably more common, that we want to create a new header within a section containing several components, so something like that:



To achieve this result, we have to create a new header (`myNewHeader`) into the existing section "Appearance" - which is loaded by `uses: "settings"` - and add the items there:

```
var appearanceSection = {

    uses: "settings",

    items: {

        // Here the magic happens ...

        myNewHeader: {

            type: "items",

            label: "My header, containing text boxes",

            items: {

                myTextBox: myTextBox,

                myTextBox2: myTextBox2

            }

        }

    }
};



// NOTHING CHANGED HERE ...

// Return overall definition of the property accordion

return {

    type: "items",

    component: "accordion",

    items: {

        appearance: appearanceSection

    }

};
```

Adding a custom section

Until now we have

- added custom items to the built-in section "appearance"
- added a new section header to the built-in section "appearance"

But we haven't covered so far how to create a new accordion section, so let's create one:

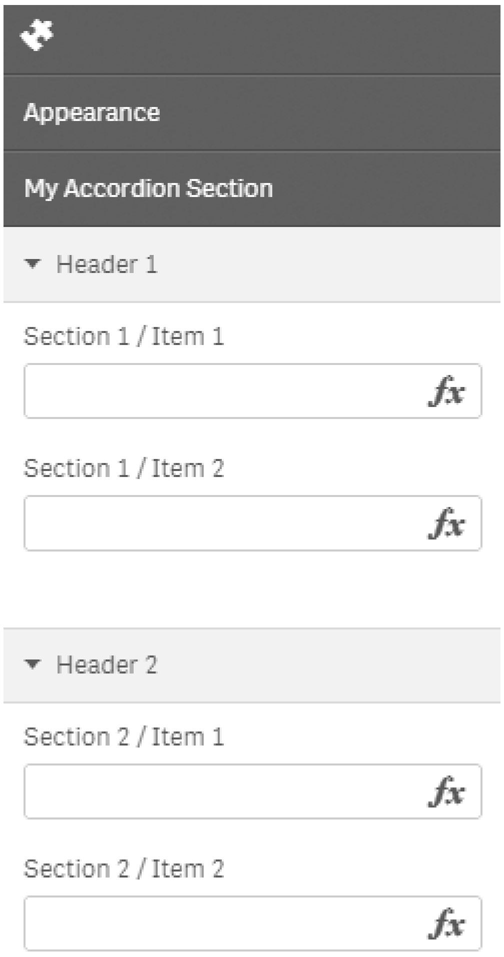
```


// Some components

var header1_item1 = {

    ref: "props.section1.item1",
    label: "Section 1 / Item 1",

    type: "string",

    expression: "optional"

};



...

...



// Define a custom section

var myCustomSection = {

    // not necessary to define the type, component "expandable-items" will automatically

    // default to "items"

    // type: "items"

    component: "expandable-items",

    label: "My Accordion Section",

    items: {

        header1: {

            type: "items",

            label: "Header 1",

            items: {
                header1_item1: header1_item1,

                header1_item2: header1_item2

            }

        },

        header2: {

            type: "items",

            label: "Header 2",

            items: {

                header2_item1: header2_item2,

                header2_item2: header2_item2

            }

        }



    }

}
```

The key in the code above is that you add the component `expandable-items` the rest of the code works as all the other examples.

Then let's again just use it:

```
return {

    type: "items",

    component: "accordion",
    items: {

        appearance: appearanceSection,

        customSection: myCustomSection

    }

};
```

The result is a custom accordion section with section headers and items:



Display & persistence

Once the custom properties are defined, Qlik Sense takes care of the rest:

- Showing the custom properties together with the built-in ones

- Persistence of property values, so if a user changes the value of a property, you don't have to take care of persisting (saving and loading) the property value

Referencing property values

Referencing the property values of *custom properties* is not very much different from referencing values from built-in properties, with one exception:

By using `ref` you can define how the property value is exposed in the object tree. This principle applies to all *custom property* items.

Two examples:

By defining a text box using

```
var myTextBox = {

    ref: "myTextBox",

    ...

};
```

the value can then be referenced in your script

```
console.log( layout.myTextBox );
```

whereas

```
var myTextBox = {

    ref: "prop.myTextBox",

    ...

};
```

will then be called using

```
console.log( layout.prop.myTextBox );
```

Grouping properties in the source code

I personally prefer to prefix all properties with `props`. Firstly this doesn't messy up the root of `layout`, secondly this approach allows me to easily iterate through all custom properties and thirdly ensures that there are not naming conflicts with the standard object of Qlik Sense (even in future versions of Qlik Sense).

Troubleshooting

Changes are not reflected

If you are changing the properties of an existing visualization extension you might run into the issue that changes in your extension's code are not reflected in visualizations based on this specific extension.

In this case remove the existing object and re-create it and you should see the changes made to the property panel.

This behavior only applies if you are making changes to the `definition` property of a visualization extension, not if you are e.g. making changes in the `paint` implementation.

#8: Hello Data

OK, now we are done with the very, very basics. Let's move on to the interesting stuff, bringing data into our*visualization extension*.

Create your Hello-Data visualization extension

For this chapter create a new visualization extension, let's call it "Hello-Data".

Hello-Data.qext

```
{

    "name" : "Hello Data",

    "description" : "Examples how to use data in visualization extensions.",

    "icon" : "extension",

    "type" : "visualization",

    "version": "0.1.0",

    "preview" : "08-hellodata.png",

    "author": "Stefan Walther"
}
```

Hello-Data.js

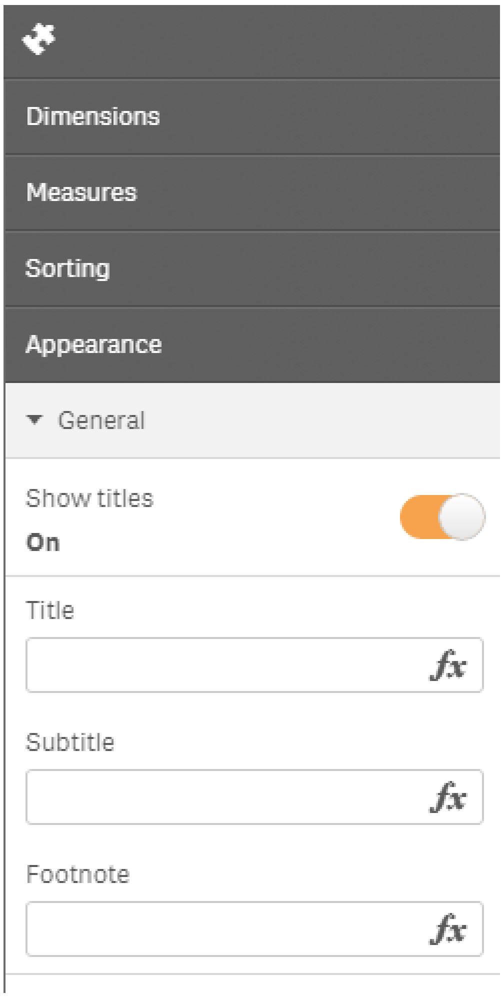
```
define( [],

    function ( ) {

        'use strict';



        return {

            definition: {},

            initialProperties: {},

            paint: function ( $element, layout ) {



                // Your code comes here



            }

        };

    } );
```

Getting data into visualization extension

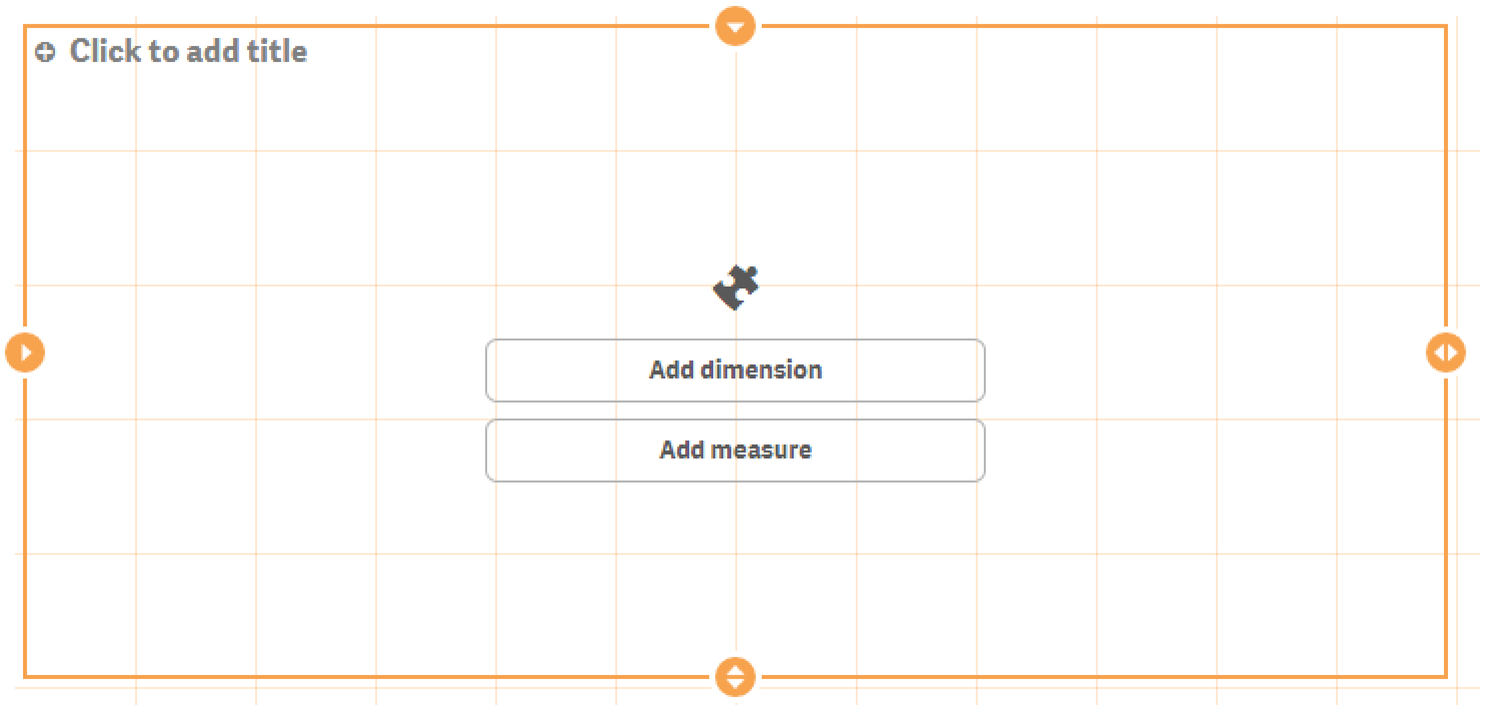
In one of the previous chapters we have already looked into the solution of re-using the built-in capability to let the user define dimensions and measures in a Qlilk Sense Visualization Extension. Therefore add the following code to the `definition` object of your extension:

```
definition: {

    type: "items",
    component: "accordion",

    items: {

        dimensions: {

            uses: "dimensions"

        },

        measures: {

            uses: "measures"

        },

        sorting: {

            uses: "sorting"

        },

        appearance: {

            uses: "settings"

        }

    }

}
```

This results into the following property panel:



If you add the visualization onto a sheet you'll furthermore realize that in *Edit Mode* the user can now define the dimensions and measures directly in the object (without using the property panel). In addition to that also drag and drop of previously defined dimensiosn and measures from the Master items is possible:

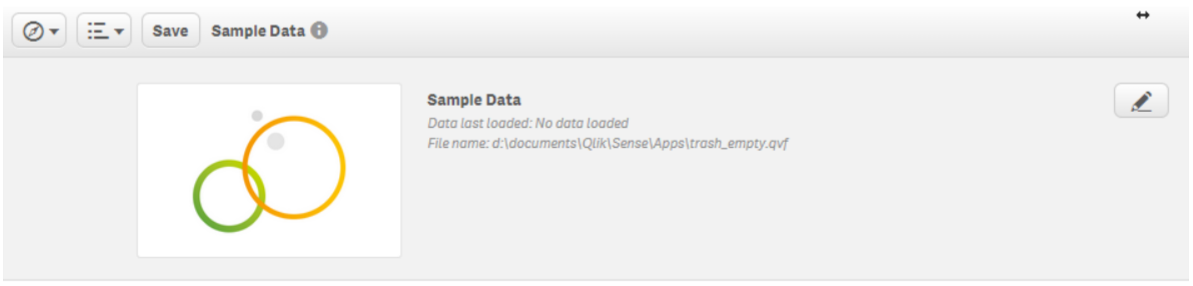


Testing your extensions with data

If you want to test your visualization extension in a blank, new Qlik Sense application you'll realize that Qlik Sense requires some data to be able to create a new sheet.

The easiest way to achieve that - if you haven't some good test data - is the following:

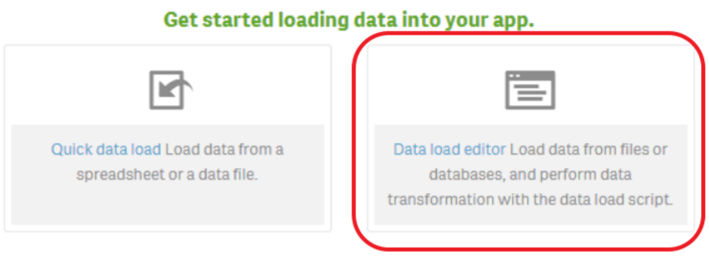
Go to the ***Data load editor***:



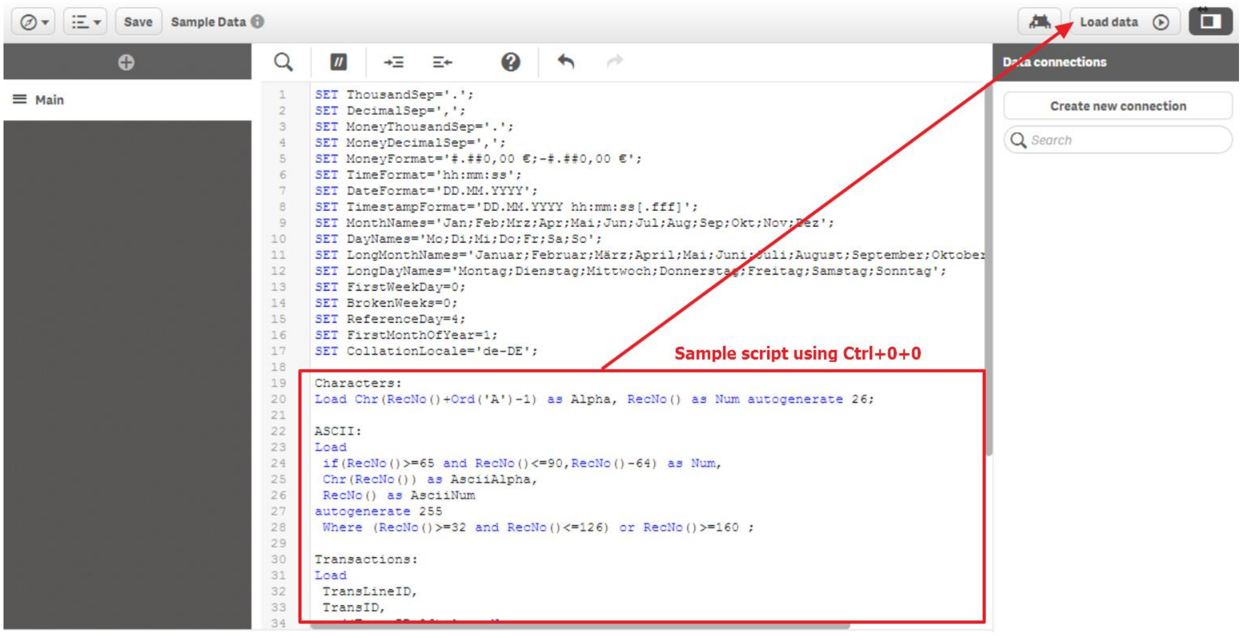
Sample Data

Data last loaded: No data loaded

File name: d:\documents\Qlik\Sense\Apps\trash_empty.qvf


Get started loading data into your app.
Quick data load Load data from a spreadsheet or a data file.
Data load editor Load data from files or databases, and perform data transformation with the data load script.

At the end of the script of the *Main* tab add the sample script by using the keyboard shortcut "Ctrl+0+0", load the data, go back to the "App overview" and then you can create your sheet to test your visualization extension.



```
SET ThousandSep='.';
SET DecimalSep=',';
SET MoneyThousandSep='.';
SET MoneyDecimalSep=',';
SET MoneyFormat='#.##0,00 €;-#.##0,00 €';
SET TimeFormat='hh:mm:ss';
SET DateFormat='DD.MM.YYYY';
SET TimestampFormat='DD.MM.YYYY hh:mm:ss[.fff]';
SET MonthNames='Jan;Feb;Mrz;Apr;Mai;Jun;Jul;Aug;Sep;Okt;Nov;Dez';
SET DayNames='Mo;Di;Mi;Do;Fr;Sa;So';
SET LongMonthNames='Januar;Februar;März;April;Mai;Juni;Juli;August;September;Oktober
SET LongDayNames='Montag;Dienstag;Mittwoch;Donnerstag;Freitag;Samstag;Sonntag';
SET FirstWeekDay=0;
SET BrokenWeeks=0;
SET ReferenceDay=4;
SET FirstMonthOfYear=1;
SET CollationLocale='de-DE';

Characters:
Load Chr(RecNo()+Ord('A')-1) as Alpha, RecNo() as Num autogenerate 26;

ASCII:
Load
 if(RecNo()>=65 and RecNo()<=90,RecNo()-64) as Num,
 Chr(RecNo()) as AsciiAlpha,
 RecNo() as AsciiNum
autogenerate 255
 Where (RecNo()>=32 and RecNo()<=126) or RecNo()>=160 ;

Transactions:
Load
 TransLineID,
 TransID,
```

Testing with more data

If you want to test with more data, here a modified sample script you can use:

```
// Change the amount here to create more records
SET vAmountTransactions=10000;

Characters:
Load Chr(RecNo()+Ord('A')-1) as Alpha, RecNo() as Num autogenerate 26;

ASCII:
Load
if(RecNo()>=65 and RecNo()<=90,RecNo()-64) as Num,
Chr(RecNo()) as AsciiAlpha,
RecNo() as AsciiNum
autogenerate 255
Where (RecNo()>=32 and RecNo()<=126) or RecNo()>=160 ;

Transactions:
Load
TransLineID,
TransID,
mod(TransID,26)+1 as Num,
Pick(Ceil(3Rand1),'A','B','C') as Dim1,
Pick(Ceil(6Rand1),'a','b','c','d','e','f') as Dim2,
Pick(Ceil(3Rand()),'X','Y','Z') as Dim3,
Round($(#vAmountTransactions)Rand()Rand()Rand1) as Expression1,
Round( 10Rand()Rand()Rand1) as Expression2,
Round(Rand()Rand1,0.00001) as Expression3;
Load
Rand() as Rand1,
IterNo() as TransLineID,
RecNo() as TransID
```

```
Autogenerate $(#vAmountTransactions)
While Rand()<=0.5 or IterNo()=1;
```

Visualizing the data returned from the Qlik Engine

If you start working with data returned from the Qlik Engine, I recommend that you first visualize your data in a native *Table* object side by side with your visualization extension. You'll recognize in a few minutes why this is quite useful.

1. Create a new sheet
2. Add your visualization to the sheet
3. Add a table object to the sheet
4. Then add the same dimensions and extensions to both, your extension and the table object

Based on the sample code/data above I have chosen the following data:

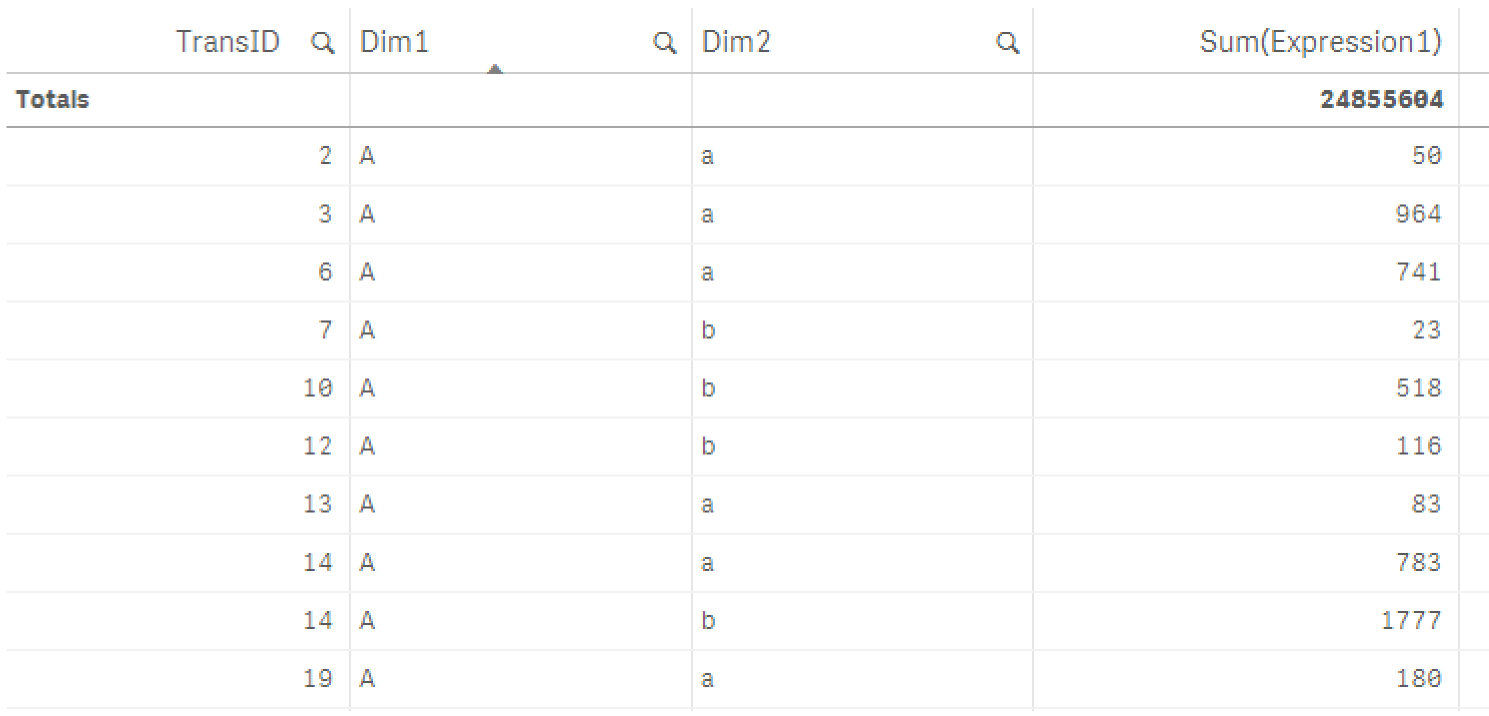
Dimensions:

- `TransId`
- `Dim1`
- `Dim2`

Measures:

- `Sum(Expression1)`
- `Sum(Expression2)`

In the native table object this will result into something like this:



TransID	Dim1	Dim2	Sum(Expression1)
Totals			24855604
2	A	a	50
3	A	a	964
6	A	a	741
7	A	b	23
10	A	b	518
12	A	b	116
13	A	a	83
14	A	a	783
14	A	b	1777
19	A	a	180

How to retrieve the data in your extension

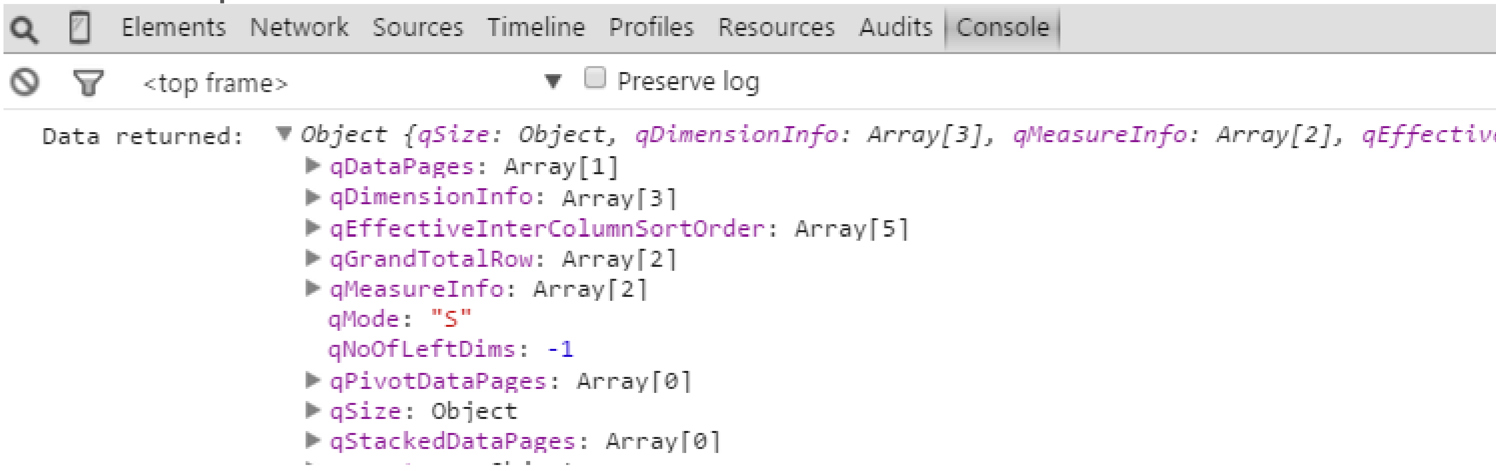
As soon as you add dimensions and measures to a visualization extension the Qlik Engine will return a so called HyperCube. As of now, don't be confused by this term, just think of table returned from the Engine (altough a HyperCube is much more).
(*You'll learn more about the construct of a HyperCube in later chapters.*)

To be able to look under the hood, let's again just use Chrome DevTools to output the HyperCube to DevTool's console.

Therefore change the paint section as follows:

```
paint: function ( $element, layout ) {



    console.log('Data returned: ', layout.qHyperCube);



}
```

The console output:



```
Data returned:  ▼ Object {qSize: Object, qDimensionInfo: Array[3], qMeasureInfo: Array[2], qEffectiv
                  ▶ qDataPages: Array[1]
                  ▶ qDimensionInfo: Array[3]
                  ▶ qEffectiveInterColumnSortOrder: Array[5]
                  ▶ qGrandTotalRow: Array[2]
                  ▶ qMeasureInfo: Array[2]
                    qMode: "S"
                    qNoOfLeftDims: -1
                  ▶ qPivotDataPages: Array[0]
                  ▶ qSize: Object
                  ▶ qStackedDataPages: Array[0]
```

You will immediately realize, that this is not *just* a normal data table, yes, it's the structure of a HyperCube. For the beginning three different objects are interesting (expand these nodes to review):

- `layout.qHyperCube.qDimensionInfo` - used dimensions
- `layout.qHyperCube.qMeasureInfo` - used measures
- `layout.qHyperCube.qDataPages` - the result

Let's create an HTML table

As we now know how the underlying data structure looks like, let's create a very simple HTML table to display the data. The HTML table should contain a header (including the labels for the dimensions measures) and certainly a body containing the data.

Skeleton

```
paint: function ( $element, layout ) {



    var hc = layout.qHyperCube;

    console.log( 'Data returned: ', hc );



    $element.empty();

    var table = '<table border="1">';

```

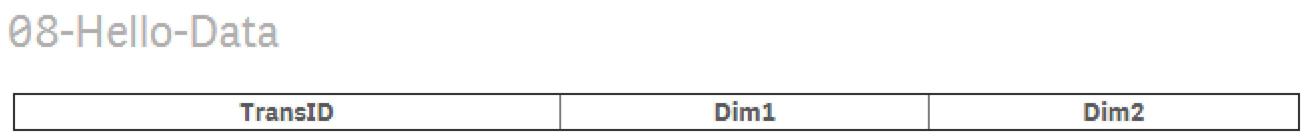
```
        table += '<thead>';

        table += '</thead>';



        table += '<tbody>';

        table += '</tbody>';

    table += '</table>';

    $element.append( table );

}
```

Table header: dimensions & measures

To get first all dimensions let's iterate through all existing dimensions using the `hc.DimensionInfo` array, then use the property`qFallbackTitle` which holds the label of the dimension:

```
...

table += '<thead>';

    table += '<tr>';

            table += '<th>' + hc.qDimensionInfo[i].qFallbackTitle + '</th>';

        }

    table += '</tr>';

table += '</thead>';

...
```

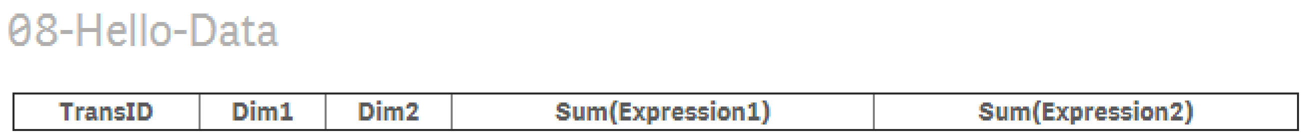
The result so far:



08-Hello-Data

TransID	Dim1	Dim2

Now let's do the same for measures (by adding additional table headings) and you should come up with:



08-Hello-Data

TransID	Dim1	Dim2	Sum(Expression1)	Sum(Expression2)

Hint:

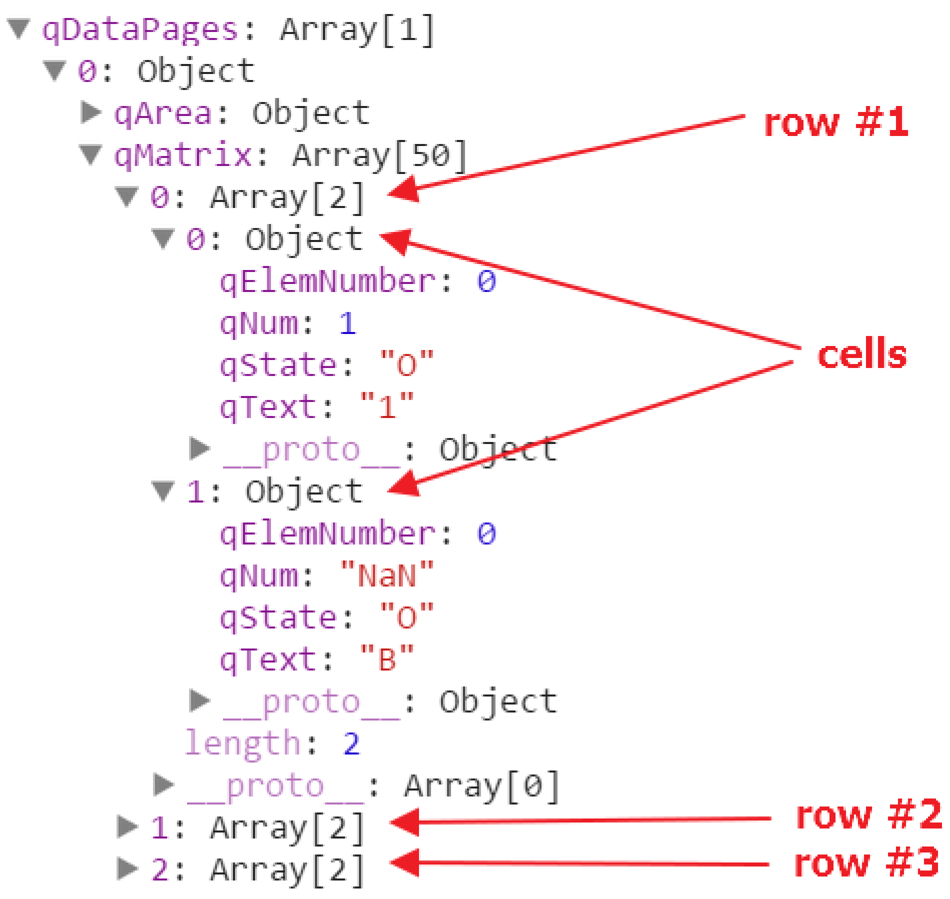
At this stage I highly recommend that you follow the creation of this simple table-output step by step and make yourself familiar with the concept. Sure, creating a table might not be a very useful and common requirement, but for the majority of visualizations extensions it is essential do deal with data and understanding the underlying data-structure - the structure of the HyperCube.

Table data

Before we add the table data, let's have a look again the the console output and especially at `qDataPages`:
If you expand the qDataPages node we recognize that

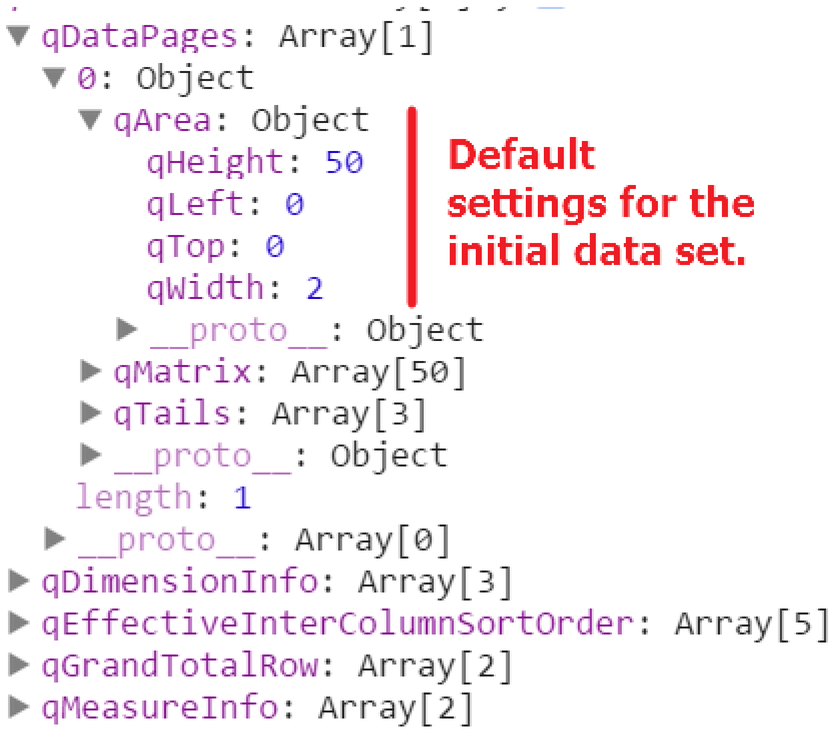
- qDataPages is an array
- and that the data is held with qDataPages[0].qDataPages.qMatrix,
- which is again an array of objects (the rows),
- each again holding an array of some other objects (the cells)

But wait, something is wrong, isn't it?
As we have defined 3 measures and 2 dimensions we should see 5 cell, but instead there are only two:



```
▼ qDataPages: Array[1]
  ▼ 0: Object
    ▶ qArea: Object
    ▼ qMatrix: Array[50]
      ▼ 0: Array[2]
        ▼ 0: Object
            qElemNumber: 0
            qNum: 1
            qState: "O"
            qText: "1"
          ▶ __proto__: Object
        ▼ 1: Object
            qElemNumber: 0
            qNum: "NaN"
            qState: "O"
            qText: "B"
          ▶ __proto__: Object
          length: 2
        ▶ __proto__: Array[0]
      ▶ 1: Array[2]
      ▶ 2: Array[2]
```

The explanation can be found if you expand `qDataPages[0].qArea`, which shows us the default settings how many data the Engine will return:



```
▼ qDataPages: Array[1]
  ▼ 0: Object
    ▼ qArea: Object
        qHeight: 50
        qLeft: 0
        qTop: 0
        qWidth: 2
      ▶ __proto__: Object
    ▶ qMatrix: Array[50]
    ▶ qTails: Array[3]
    ▶ __proto__: Object
    length: 1
  ▶ __proto__: Array[0]
▶ qDimensionInfo: Array[3]
▶ qEffectiveInterColumnSortOrder: Array[5]
▶ qGrandTotalRow: Array[2]
▶ qMeasureInfo: Array[2]
```

By default these are:

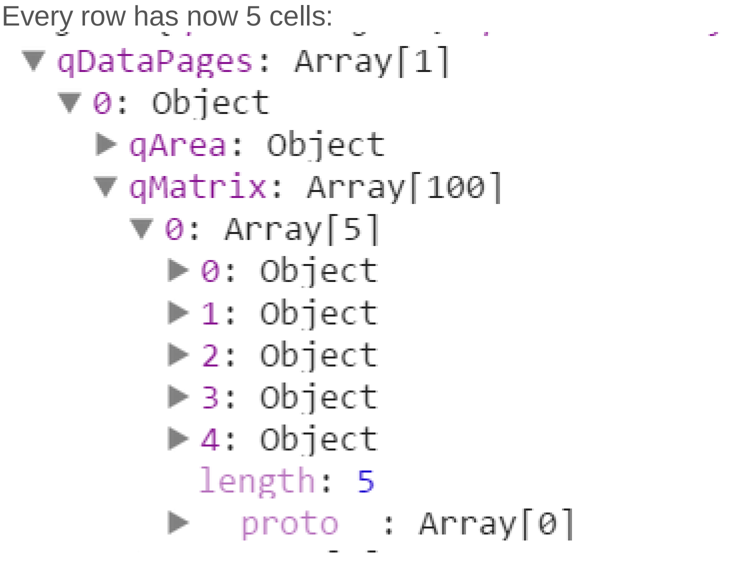
- 50 rows (`qArea.qHeight`)
- 2 columns (`qArea.qWidth`)

Changing the initial properties

We can change this behavior by overruling the default properties in initialProperties, which is already part of our script file:

```
initialProperties: {

    qHyperCubeDef: {

        qDimensions: [],

        qMeasures: [],

        qInitialDataFetch: [

            {

                qWidth: 10,

                qHeight: 100

            }

        ]

    }

},
```

Every row has now 5 cells:



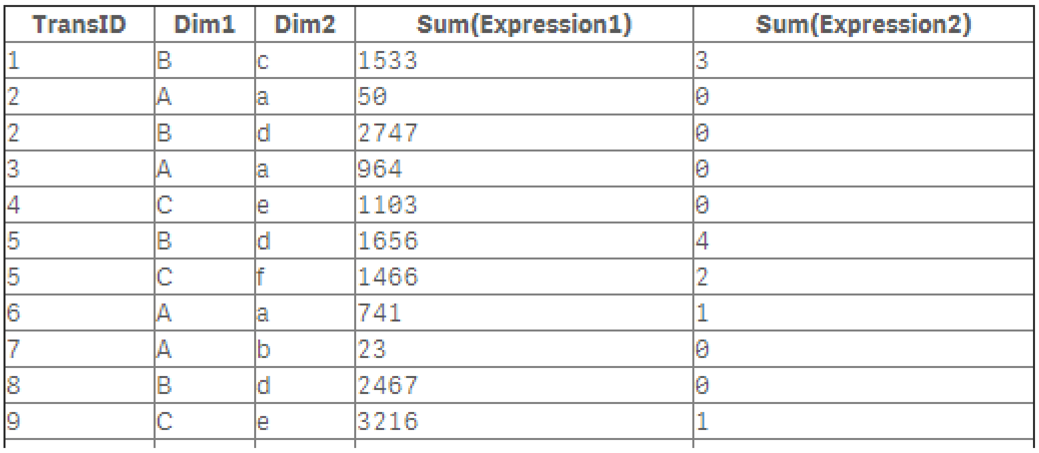
```
▼ qDataPages: Array[1]
  ▼ 0: Object
    ▶ qArea: Object
    ▼ qMatrix: Array[100]
      ▼ 0: Array[5]
        ▶ 0: Object
        ▶ 1: Object
        ▶ 2: Object
        ▶ 3: Object
        ▶ 4: Object
          length: 5
        ▶   proto  : Array[0]
```

Rendering table data

Very similar to the code above we now have to iterate over rows and then cells to render all rows and columns:

```
table += '<tbody>';



    // iterate over all rows

    for (var r = 0; r < hc.qDataPages[0].qMatrix.length; r++) {

        table += '<tr>';



        // iterate over all cells within a row

        for (var c = 0; c < hc.qDataPages[0].qMatrix[r].length; c++) {

            table += '<td>';

                table += hc.qDataPages[0].qMatrix[r][c].qText;

            table += '</td>';

        }
        table += '</tr>';


    }


table += '</tbody>';
```

The result:



TransID	Dim1	Dim2	Sum(Expression1)	Sum(Expression2)
1	B	c	1533	3
2	A	a	50	0
2	B	d	2747	0
3	A	a	964	0
4	C	e	1103	0
5	B	d	1656	4
5	C	f	1466	2
6	A	a	741	1
7	A	b	23	0
8	B	d	2467	0
9	C	e	3216	1

Hurray, we are done, we have rendered all the data, so we are fine, are we?
Unfortunately not really. There are still some issues we have to solve:

Issues with the current implementation

If you have a deeper look at our current solution (and especially if you compare it with the native *Table*, you'll recognize that:

- **Table is not scrollable**
  A user is now not able to scroll vertically, only the rows fitting into the current screen will be shown.

- **All data?**
  Since we have defined qHeight: 100 only 100 rows will be rendered, but the native *Table* returns in my case 10.000 records. This is where the `qDataPages` come into play.

- **Making selections**
  In the current implementation a user cannot make selections, e.g. clicking into the a cell in column 2 (Dim1) and select a value in the field `Dim1`.

- **The table is ugly**
  Yes, it's more or less just the default style. We could certainly add now inline styles to the table but that's not really a best practice. Instead it would be much better to add an external style sheet which we can refer to. (We'll look into that in the next chapter).

We'll tackle these problems in Part I / chapter 10, first let's have a look into another topic, thus how to **load external resources** to be able to style our table.

Some golden rules

When working with data the following best practices might be helpful for you:

- Prove the data returned from the Qlik Engine by using the same dimensions and measures in a native *Table* object
- Don't guess, use console.log wisely to understand the underlying data-structure
- Always add `initialProperties` to define how many rows/columns are available in your JavaScript object
- When changing the `initialProperties`, remove and re-add your visualization extension.

Final code

```
define( [],

    function ( ) {

        'use strict';



        return {



            definition: {

                type: "items",

                component: "accordion",

                items: {

                    dimensions: {

                        uses: "dimensions"

                    },

                    measures: {
                        uses: "measures"

                    },

                    sorting: {

                        uses: "sorting"

                    },

                    appearance: {

                        uses: "settings"

                    }

                }

            },

            initialProperties: {

                qHyperCubeDef: {

                    qDimensions: [],

                    qMeasures: [],

                    qInitialDataFetch: [

                        {

                            qWidth: 10,

                            qHeight: 100

                        }

                    ]

                }
            },

            paint: function ( $element, layout ) {



                var hc = layout.qHyperCube;

                //console.log( 'Data returned: ', hc );



                // Default rendering with HTML injection

                $element.empty();

                var table = '<table border="1">';



                    table += '<thead>';

                        table += '<tr>';

                        for (var i = 0; i < hc.qDimensionInfo.length; i++) {

                            table += '<th>' + hc.qDimensionInfo[i].qFallbackTitle + '</th>';

                        }

                        for (var i = 0; i < hc.qMeasureInfo.length; i++) {

                            table += '<th>' + hc.qMeasureInfo[i].qFallbackTitle + '</th>';

                        }

                    table += '</tr>';

                    table += '</thead>';
                    table += '<tbody>';

                        for (var r = 0; r < hc.qDataPages[0].qMatrix.length; r++) {

                            table += '<tr>';

                            for (var c = 0; c < hc.qDataPages[0].qMatrix[r].length; c++) {

                                table += '<td>';

                                    table += hc.qDataPages[0].qMatrix[r][c].qText;

                                table += '</td>';

                            }

                            table += '</tr>';

                        }

                    table += '</tbody>';

                table += '</table>';

                $element.append( table );



            }

        };



    } );
```

Reference Links

http://qliksite.io/tutorials/qliksense-visualization-extensions/

https://help.qlik.com/en-US/sense-developer/3.0/Subsystems/APIs/Content/TableAPI/exportData-method.htm