QlikView

OBJECT EXTENSION BASICS

QlikView Technical Brief

December 2012

QlikView 11

qlikview.com

QlikView

Contents

INTRODUCTION 3

OVERVIEW 4

SET-UP 4

CODE OVERVIEW 6

SUMMARY 9

REFERENCES 10

QlikView

Introduction

Extensions are a very powerful part of QlikView. Unfortunately there is a lot of confusion on some of the more basic set-up concepts. This document will explain how to create a new extension and how to write the basic code to get the extension up and running.

SUPPORTING MATERIALS

- **Where extensions are installed.** Click Here
- **Version 11 SDK** Click Here
- **Definition file set-up** Click Here
- **Converting qv10 extensions** Click Here
- **This document should be packaged with a template extension called template.qar**

QlikView

Overview

In general, the best way to learn about how extensions work would be to open one up and look around at the code inside. There are two main pieces to setting up an extension. First the folder, definition.xml, script.js, and (optionally) CSS and other JS files will need to be created. Usually this is done by copying and pasting another extension, then modifying the files inside. This document focuses on the template extension that can be used which is set up in the most basic way possible to aid in set-up This extension should have been packaged with this document, and can also be found in salesforce and community (soon). Please install this extension as it will provide the most beneficial learning experience with this document.

Secondly, the JavaScript code will need to be written. This code is contained in Script.js. The template extension example will display the following basic concepts:

- General code structure
- Adding external JavaScript and CSS files
- Adding the extension itself to QlikView
- Retrieving data and property settings of the extension
- Looping through the data and displaying it in the extension

Assumption / Requirements

This document is not intended as much to explain the JavaScript behind the extension framework. The reader should have some basic JavaScript and HTML knowledge or else be willing to accept certain concepts and lines of code on faith. The more rudimentary concepts of JavaScript and HTML (setting variables, etc.) will not be covered as much as the basic functions and ideas necessary to get a basic extension up and running.

Set-up

First, it would be ideal to copy and paste the simplest extension you have, changing the name of the folder to whatever extension name you decide on. If you're not sure where to find your extension folder, please visit this link on Qlik Community. Again, I would recommend using the template.qar extension that is packaged with this documentation as I will be using this extension in all further examples. Once this new folder is created, open it and notice that there are only 3 or so files there.

The first file to focus on is Definition.xml. In QlikView 10, the property panel needed to be manually built, and it was a difficult and confusing process. In QlikView 11, however this is done automatically based on the set up of the Definition.xml file. This file defines the properties that are seen when the extension is right clicked, similar to the properties of a normal QlikView object. Using the different types of property elements shown and described below, you should be able to assemble a panel that meets the needs of the extension you're building.

A QlikView extension is, in essence, a straight table which means it must have at least one

dimension and an expression set. The data itself then comes in to the extension in a similar tabular fashion. Thinking of extensions in these terms will be very helpful in the future.

To edit Definition.xml and the other JavaScript and CSS files, it would be very much recommended to obtain a text editing program such as Notepad++. The code should look something like this:

```
<?xml version="1.0" encoding="utf-8"?>

<ExtensionObject Label="template" Description="base template for extension object"
PageHeight="50000">

  <Dimension Label="Dimension 1" Initial="" TargetName="Dimension Name" />

  <Dimension Label="Dimension 2" Initial="" TargetName="Dimension Name" />

  <Measurement Label="Measure 1" Initial=""/>

     <Text Label="Text Box 1" Type="text"/>

   <Text Label="Checkbox 1" Type="checkbox" Initial="" />

   <Initiate Name="Chart.Title" value="Object Template" />

     <Initiate Name="Caption.Text" Value="Object Template" />

 </ExtensionObject>
```

For a full list of different kinds of properties that can be set, please see this post on Qlik Community for a full list. This particular extension uses the most basic properties.

Before thinking about the properties however, some changes need to be made in order for QlikView to know the name of your extension and where to find it. This information is held in the “ExtensionObject” tag at the top. In this tag, the Label is set to be the name of the extension itself as you’ll see it in the list of extensions. In the above code, the extension is labeled “template.”

The description value should be changed to a short description of the extension and what it does. Lastly, the PageHeight defines the number of rows of data the extension will accept. It is set in this example to 50,000 rows.

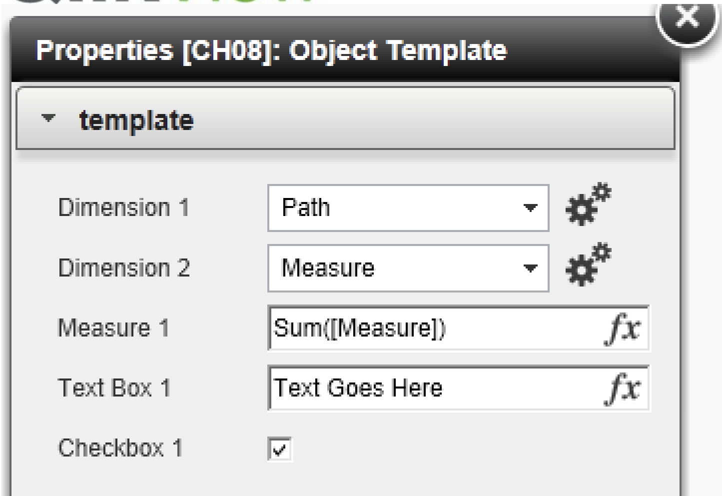
Now that the settings for the Extension itself have been changed, a list of properties is then listed. The “dimension” tagged elements are, simply, the dimensions that you wish to pass into the extension. These can be added or removed to your needs. The value of the “Label” property will be displayed to the left of the dimension selection box.

The “measurement” tagged elements are to place an expression field in the properties panel. This value will also come back in the dataset table that is returned into the extension.

The “text” tagged elements, however, are treated separately from the dimensions and expressions. They return one single value and, as we’ll see later, are only retrieved once, where as a data table needs to be looped through.

Lastly, the initiate tags are used to initiate certain properties of the extension. In the example above the title and caption of the extension are being set up. Since only the basics are being covered here, however, these will not be covered, but much of what would be needed can be found here.

Using the XML code shown above, the following properties box would be produced:



Code Overview

Now that the Definition.xml and file folder is properly set-up, the code for the extension can now be created to display QlikView data with JavaScript. In order to get an extension up and running, the code needs to add the extension itself to QlikView, add any other JavaScript or CSS, and retrieve any values or data that has been passed into the extension.

Looking at the structure of the Script.js file for the template.qar extension will be very helpful. In general, the JavaScript file itself is structured to control the initiation of the extension through functions. So, the code should have several JavaScript functions and, at the bottom of the page, run a function to initiate the extension itself. For example, if you reference the code in the References section below, you can see that there is a function at the bottom of the code called extension_Init(). This function can be found higher in the document and adds a JavaScript file (if necessary), then runs a function called extension_Done. Extension_Done does the actual adding of the extension to the QlikView document. The code is structured this way because there are often dependencies in the code that require the loading of the JavaScript to be segmented.

To summarize, there is a function at the bottom of the code that is run which adds other JavaScript libraries. When this is done, it triggers the final function which finally adds the extension itself.

At this point, it might be beneficial to go line by line through the code of the Script.js file in the template extension example. The variable that is created at the top of the file is a best practice that can be very useful.

```
var template_path = Qva.Remote +
"?public=only&name=Extensions/template/";
```

This variable, template_path, is set to the path of the extension directory itself. This is necessary in order for QlikView to know where to find the code and files for the extension. It can be laborious to type out this full path time and time again, and more importantly, it minimizes the number of times the extension's folder is referenced. When creating a new extension with a new folder name, this variable will of course need to be changed. It's easier to change it in one or two places than searching the whole file.

So, for example, if you were copying this template extension to create your own, and you named the new folder to “superchart,” the paths in the template extension’s Script.js would need to be changed from containing “template” to “superchart.” This template_path variable would need to be changed to:

```
var template_path = Qva.Remote +
"?public=only&name=Extensions/superchart/";
```

The next important piece of code is the extension_Init function. In the template example code below, this code is checking to see if the jQuery JavaScript library is available, and if not, it adds the library to the extension. Basically, this is simply checking to see if the extension is being run in QlikView 10 (which does not contain jQuery by default), and if so, it adds the jQuery file.

Most importantly, this code section shows how to add an external JavaScript file:

```
Qva.LoadScript(template_path + 'jquery.js', extension_Done);
```

The Qva,Loadscript function adds a file based on the path we give (in this case it’s using our template_path variable) and then that JavaScript file we’ve included will be available to us. This is very powerful because there are so many JavaScript libraries out there that visualize data. In most cases, making these libraries available to the extension is as simple as downloading the JavaScript files, then adding them using the Qva.Loadscript function.

At the end of this function, the “extension_Done” function is triggered. So once a JavaScript file is added, it triggers a function to continue the process. Again, this is in order to ensure our extension code below doesn’t start trying to reference things that are contained in external libraries that might not be loaded yet.

Now that the extension_Done function has been called, this function will add the extension. This code is fairly simple and is done with this code:

```
Qva.AddExtension('template', function(){
```

In this example, the name of the file folder containing the extension is named “template.” If an extension was being created with a folder name of “superchart”, the code would be:

```
Qva.AddExtension('superchart', function(){
```

Again, any reference to the extension’s folder will need to be changed to the folder you have created for your extension.

Now that the extension itself has been created, it might be good to add a CSS stylesheet to change the look and feel of the content of the extension. This is also very simple. Assuming there is a CSS file called style.css in our extension folder, the code to add this file to the extension is:

```
Qva.LoadCSS(template_path + "style.css");
```

This code simply uses the path variable that was created to point to the CSS file in our extension folder.

The next important piece of code is another best practice for developing extensions:

```
var _this = this;
```

The extension object itself can be referenced as a variable called “this” in the extension code, but there are many JavaScript functions that also have elements referred to as “this.” So, in order to avoid conflicts, it is advisable to create a variable called “_this” to contain the extension object.

The next piece of code is our first example of pulling data from our properties panel:

```
//get first text box
var text1 = _this.Layout.Text0.text.toString();
//get check box value
var checkbox1 = _this.Layout.Text1.text.toString();
```

This code pulls the values from the two text properties from the properties panel. Again, for the full formatting of how we're getting these values, please see the JavaScript API on community, but for the purposes of the basics, simply follow this general format for getting text properties. If you added another text box property to your properties panel, for example, you would simply retrieve it this way:

```
var text2 = _this.Layout.Text2.text.toString();
```

The next bit of code is another best practice for extensions. It may be more complicated than is needed for the basics, so rather than explain the code itself, it might be best to simply know what it does:

```
var divName = _this.Layout.ObjectId.replace("\\", "_");
            if(_this.Element.children.length == 0) {//if this div
doesn't already exist, create a unique div with the divName
                  var ui = document.createElement("div");
                  ui.setAttribute("id", divName);
                  _this.Element.appendChild(ui);
            } else {
                  //if it does exist, empty the div so we can fill it
again
                  $("#" + divName).empty();
            }
```

In JavaScript when html elements are created, they're often able to be manipulated by targeting their IDs. If we have several extensions on a sheet (or multiple instances of the same extension), this can cause conflicts in that the code will not know which exact chart it should be running the code on. In order to combat this conflict, the code above simply sets our extension to a unique ID (the QlikView object ID more or less) so the code will only target one specific instance of the extension.

Next, a variable is created to simply hold all of the HTML code that we will inject into the extension:

```
var html = "";
```

The code will continually add code to this variable before finally adding it to the object itself.

Now in the code, the full data set of the extension is set to a variable for easier reference:

```
var td = _this.Data;
```

The extension now wants to write out the values of the text objects we retrieved, as well as the number of rows of data:

```
html += "Text1: " + text1 + "<br /> checkbox1 value: " + checkbox1 +
"<br />Data Length: " + td.Rows.length + " rows<br />";
```

This code is simply stringing together the variables that were set above with the actual html code. The td.Rows.Length code is used to show the total number of rows of data.

Finally, it's time in the code to loop through the data set and display the values of the table. The code that does this is:

```
for(var rowIx = 0; rowIx < td.Rows.length; rowIx++) {
                  //set the current row to a variable
```

```
                    var row = td.Rows[rowIx];
                    //get the value of the first item in the dataset row
                    var val1 = row[0].text;
                    //get the value of the second item in the dataset row
                    var val2 = row[1].text;
                    //get the value of the measurement in the dataset row
                    var m = row[2].text;
                    //add those values to the html variable
                    html += "value 1: " + val1 + " value 2: " + val2 + "
expression value: " + m + "<br />";
            }
```

This code is simply using a for loop to increment through the rows. The data comes in as a multidimensional array, so for each row, the different values can be obtained with this format:

```
var val1 = row[0].text;
```

The number contained in the row array is based on how many columns in the table. For this example, there are three columns in our data table (set up a straight table with the same properties to see this). So, the code is simply taking the value from each column and setting 3 different variables to this value.

Then, the code is simply adding this variables and their values to some html code, and adding it to our html variable:

```
html += "value 1: " + val1 + " value 2: " + val2 + " expression value: "
+ m + "<br />";
```

This loop continues to run until it's gone through all the rows. Once the loop is finished, the extension has a variable called html which contains all of the html that should be displayed. To add the html to the extension finally, the following code is used:

```
_this.Element.innerHTML = html;
```

The example extension would then display some ugly HTML similar to the following:

Object Template
Text1: Text Box Value
checkbox1 value: 1
Data Length: 9 rows
value 1: BO value 2: Barack Obama expression value: 1
value 1: BR value 2: Buddy Roemer expression value: 1
value 1: FK value 2: Fred Karger expression value: 1
value 1: GJ value 2: Gary Johnson expression value: 1
value 1: MR value 2: Mitt Romney expression value: 1
value 1: NG value 2: Newt Gingrich expression value: 1
value 1: RP value 2: Ron Paul expression value: 1
value 1: RS value 2: Rick Santorum expression value: 1
value 1: RT value 2: Randall Terry expression value: 1

Properties [CH46]: Object Template
template
Dimension 1 Initials
Dimension 2 Candidate
Measure 1 =1
Text Box 1 Text Box Value
Checkbox 1

Summary

JavaScript is a very complex and robust coding language, however, the basics for getting QlikView extensions up and running should be fairly easy to understand. The best way to get started is to get your hands dirty. Open up the template example extension and start toying around with the code and properties to get a better understanding of extensions. Extensions are a very powerful tool and great way for a QlikView developer to increase the value of their skillset.

The following is the code for the Script.js file of the template extension:

```
var template_path = Qva.Remote +
"?public=only&name=Extensions/template/";
function extension_Init()
{
      // Use QlikView's method of loading other files needed by an
extension. These files should be added to your extension .zip file
(.qar)
      if (typeof jQuery == 'undefined') {
          Qva.LoadScript(template_path + 'jquery.js', extension_Done);
      }
      else {
          extension_Done();
      }

    //If more than one script is needed you can nest the calls to get
them loaded in the correct order
    //Qva.LoadScript(template_path + "file1.js", function() {
    //Qva.LoadScript(template_path + "file2.js", extension_Done);
    //});

}

function extension_Done(){
      //Add extension
      Qva.AddExtension('template', function(){
            //Load a CSS style sheet
            Qva.LoadCSS(template_path + "style.css");
            var _this = this;
            //get first text box
            var text1 = _this.Layout.Text0.text.toString();
            //get check box value
            var checkbox1 = _this.Layout.Text1.text.toString();
            //add a unique name to the extension in order to prevent
conflicts with other extensions.
            //basically, take the object ID and add it to a DIV
            var divName = _this.Layout.ObjectId.replace("\\", "_");
            if(_this.Element.children.length == 0) {//if this div
doesn't already exist, create a unique div with the divName
                  var ui = document.createElement("div");
                  ui.setAttribute("id", divName);
                  _this.Element.appendChild(ui);
            } else {
                  //if it does exist, empty the div so we can fill it
again
                  $("#" + divName).empty();
            }
            //create a variable to put the html into
            var html = "";
            //set a variable to the dataset to make things easier
            var td = _this.Data;
            //add the text variables to the html variable
            html += "Text1: " + text1 + "<br /> checkbox1 value: " +
checkbox1 + "<br />Data Length: " + td.Rows.length + " rows<br />";
```

```
		//loop through the data set and add the values to the html
variable
		for(var rowIx = 0; rowIx < td.Rows.length; rowIx++) {
			//set the current row to a variable
			var row = td.Rows[rowIx];
			//get the value of the first item in the dataset row
			var val1 = row[0].text;
			//get the value of the second item in the dataset row
			var val2 = row[1].text;
			//get the value of the measurement in the dataset row
			var m = row[2].text;
			//add those values to the html variable
			html += "value 1: " + val1 + " value 2: " + val2 + "
expression value: " + m + "<br />";
		}
		//insert the html from the html variable into the extension.
		_this.Element.innerHTML = html;
	});
}
//Initiate extension

extension_Init();
```