|  |
| --- |
| [Type the company name] |
| [Type the document title] |
| [Type the document subtitle] |
|  |
| **[Type the author name]** |
| **[Pick the date]** |

|  |
| --- |
| [Type the abstract of the document here. The abstract is typically a short summary of the contents of the document. Type the abstract of the document here. The abstract is typically a short summary of the contents of the document.] |

Contents

[2.2. Fast Track Training Project 3](#_Toc417123965)

[2.2.1. Project Description 3](#_Toc417123966)

[2.2.2. Getting Started – Inception Phase 3](#_Toc417123967)

[2.2.3. Packing up - Elaboration Phase 4](#_Toc417123968)

[2.2.4. Setting off - Construction Phase 6](#_Toc417123969)

## Fast Track Training Project

After completing the *Fast-Track* training period successfully, we made up groups to proceed with our fast track training project. Me, Fawsan, and Fasna made up a group of three members and selected a project from the “WSO2-Redmine”. We scheduled a meeting with Mr. Srinath Perera to discuss about the project details. He inspired us with the project idea and gave the approval for us to continue working on the project.

### Project Description – IgViz (Interactive Generic Visualization Library)

WSO2-Products, despite of having very powerful and efficient functionalities, do not consist of a killer mechanism to visualize data. This can lead potential customers to underestimate the eminent productivity of their products. Therefore a generic web tool (a *widget* as we may address) for visualizing data, without expertise knowledge on the subject, is a vital requirement. Our task, therefore, was to create a generic JavaScript library which can be used by open-source developers to draw charts with minimal effort.

### Getting Started – Inception Phase

The library will be a simple JavaScript library from a high level point of view. However doing this without the aid of external libraries is discouraged due to 3 *mundane* facts,

* Limited time frame
* Having to *reinvent the wheel*
* Neglecting the best open-source practices.

Nevertheless we have to be extra careful when selecting external libraries for a software project. We must ensure that the resource is trustworthy and able to perform the expected task. Considering all above mentioned facts, we agreed upon using **d3js** to lay the foundation for our tool.

#### Why *we* chose d3js over other similar libraries?

The term 'd3' stands for *data driven documents*. It is a kind of low level JavaScript library for manipulating documents based on data. So to answer the question, 'why d3?',

* It provides several reusable and advanced functionalities, which would have consumed a lot of time, if attempted to write from scratch.
* It has gained a good reputation over time, even from experts, for its strongly written code base.
* A lot of samples are available for new developers to grasp the d3 environment.
* Neatly written documentations are also available.



**Figure 2.1 - d3js Logo**



**Figure 2.2 – Samples from d3js.org**

During this time period, I mainly focused on learning the core technologies that will be used for this project. *The-NewBoston'*s free video tutorial series (www.thenewboston.com) on JavaScript was pretty helpful. Additionally I followed few more videos to learn the d3 basics for svg manipulation.

Adapting to the JavaScript development environment was not much of a challenge as it is a scripting language used for web development which does not require any additional configurations to be installed. The only requirements were a *text editor* and a *web browser*.

The plan we made during the **Inception phase** of the fast track project can be summarized as follows,

* Project Idea : Build a generic tool for chart visualization
* Proposed Core Technologies: JavaScript, d3js, WebStorm IDE, Sublime Text IDE.
* Expected Delivery Date: 23/01/2015
* Proposed Core Deliverable(s): A generic JavaScript library
* Version Controlling: git via github

### Packing up - Elaboration Phase

After the initial stage, we started designing the tool. Considering the requirement explained in an earlier discussion, we identified the need to implement the following chart types in the initial version of the tool.

* Single Number diagram: A chart which summarizes the data set into a single number representation.

In our case, we will show the average, maximum, and minimum value.

* **Line Chart Diagram:** A two dimensional chart which shows the change in one dimension against a

unit change in the other as a continuous line.

* Bubble Chart: A chart that displays three dimensions of data in a two dimensional plain.
* Table: A Straight forward representation of the complete data set with custom styles.
* **Map Diagram:** A diagram which shows data specific to a location on the world map.
* Bar Chart: A chart that represents the correlation of data of one discrete variable against another

discrete/continuous variable using rectangle bars.

We discussed and analyzed above mentioned chart types giving thought to how each would be implemented without losing consistency and extensibility. We had to come up with some mechanism to accomplish these as well as other common non functional requirements.

Therefore we planned to create a *JSON* skeleton to which a data set will be modeled into, before sending to draw a chart.

"dataTable": {

"metadata":{

"names":["Country","Area","GDP","Inflation","Life.expect","Military","Pop.growth"],

"types":['C', 'N', 'N', 'N', 'N', 'N', 'N','N']

},

"data": [

["Austria", 83871, 41600, 3.5, 79.91, 0.8, 0.03],

["Belgium", 30528, 37800, 3.5, 79.65, 1.3, 0.06],

["Bulgaria", 110879, 13800, 4.2, 73.84, 2.6, -0.8],

["Croatia", 56594, 18000, 2.3, 75.99, 2.39, -0.09],

["Czech Republic", 78867, 27100, 1.9, 77.38, 1.15, -0.13]

]

}

**Figure 2.3 – Modeled dataset sample**

Fig 2.3 shows how the ‘types’ section labels the columns defined by the ‘names’. ‘C’ stands for *categorical* and ‘N’ stands for *numerical*. This categorization will come in handy when we have to suggest suitable chart types according to the selected columns. The ‘data’ section contains a matching set of data to the metadata section defined earlier.

Considering the above design constraints and the complexity of each chart type, we divided the work among ourselves. Shown below is the milestone plan we created to present to the project stake holders.

#### Milestone Plan for the project

We prepared an on-line milestone plan for the project according to design considerations and sent it via email to Mr. Samissa for getting the approval. Shown below is the accepted milestone plan according t which we continued the project.



**Figure 2.4 – Milestone plan for the fast-track training project**

### Setting off - Construction Phase

We created a repository for the project in *git* as planned and each member forked a copy from the main repository. I was assigned earlier to implement the line chart and the map diagram. I started my task with the line chart diagram. However while working on our individual tasks, we had to implement the frame for the library to which individual charts would be *plugged in* after they are implemented.

#### Skeleton JavaScript file with the shared primitive functions

Seven functions were implemented in this file to accomplish the following sub tasks

* Plot Chart
* Create Scales
* Create Axes
* Configure Points
* Configure Point Labels
* Re Draw Clicked
* Create Form

If we examine the js functions written for above, the one for “Re-Draw-Clicked” plays a special role. This contains a variable called *chartConfigs* which defines the dimensions needed to draw the charts. The code segment for *chartConfigs* is shown below,

var chartConfig = {

"title": "Title",

"xLog": false,

"yLog": false,

"xAxisData": getValue('xAxis'),

"yAxisData": getValue('yAxis'),

"yAxis2Data": getValue('yAxis2'),

"yAxis3Data": getValue('yAxis3'),

"mapLocation": getValue('mapLocation'),

"pointColor": getValue('pointColor'),

"pointSize": getValue('pointSize'),

"pointLabel": 0,

"chartWidth": 600,

"chartHight": 400,

"padding": 60,

"chartType": targetChartId.replace("#", "")

}

**Figure 2.5 – JavaScript Code Segment of *chartConfig* Variable**

The plot function is implemented to call the correct function to plot a chart. The key parameter for selecting the chart type is extracted from the variable mentioned above which will be passed to the plot function as a parameter.

igViz.plot = function (divId, chartConfig) {

if ("scatter" == chartConfig.chartType) {

drawScatterPlot(divId, chartConfig, this.dataTable)

} else if ("bar" == chartConfig.chartType) {

drawBarChart(divId, chartConfig, this.dataTable)

} else if ("singleNumber" == chartConfig.chartType) {

drawSingleNumberDiagram(divId, chartConfig, this.dataTable)

} else if ("map" == chartConfig.chartType) {

drawMapDiagram(divId, chartConfig, this.dataTable)

} else if ("lineChart" == chartConfig.chartType) {

drawLineChart(divId, chartConfig, this.dataTable)

}

else {console.error("Unknown chart type " + chartConfig.chartType);return;}

}

**Figure 2.6 – JavaScript Code Segment of *igviz.plot* function**

#### Line Chart Diagram Implementation

The main concepts, sub tasks, and challenges I came across while this implementation phase can be listed as follows,

* Sorting the data set with respect to the X-axis coordinates
* Mapping coordinates according to a scale
* Connecting coordinates and *interpolation*
* Coloring each line chart
* Appending labels

Sorting the data set is a vital requirement. This makes sure that the coordinates are connected in correct order. Unlike in a situation where we would connect the dots with our hand, this causes the path to be a scribble if unsorted. Shown below is a case in which the X coordinates are not sorted properly.

****

**Figure 2.7 – Line chart drawn using an unsorted data set**

This issue was solved using the following handy function to sort the data set.

dataSet.sort(function (a, b) {

return a.data[xAxisID] - b.data[xAxisID];

});

**Figure 2.8 – JavaScript code segment for sorting a data set**

This *algorithm* sorts the data set *in place***.** The logic underneath is similar to *bubble sort*. Comparison is done internally by evaluating the sign of the subtraction. JavaScript language provides several such functions which can be used to replace complex code segments.

When everything else is setup correctly, the code segment for appending the path to the svg can be executed. Interpolation and line coloring can be done in the same code segment if the programmer knows how to play with his stuff. The code segment I wrote is shown below.

graph.append("path")

.attr("class", "line")

.attr("d", function (d) {

return line.interpolate(mode)(d.values);

})

.style("stroke", function (d, i) {

return getColor(i % 3)

});

**Figure 2.9 – JavaScript code segment for appending path to data points**

The interpolation *mode* in the above code segment can be one of linear, basis, step-before, step-after, cardinal or monotonous.

Shown below are chartsdrawn in 3 such interpolation modes



**Figure 2.10 – Line chart with linear**

**interpolation mode**

**Figure 2.11 – Line chart with cardinal**

**interpolation mode**



**Figure 2.12 – Line chart with Step-before**

**interpolation mode**

I created a sub menu to select chart configurations (i.e. - columns, colors and interpolation mode) for drawing the chart. Shown below is a snapshot of the menu.



**Figure 2.13 – HTML menu for selecting**

**chart configurations**

Terms used: javascript , d3js, svg, version controlling, git, github, webstorm, sublime text, WSo2, IDE, text editor, web browser, widget,json,igViz, bubble sort, sort, algorithm