**UNIVERSITY OF MORATUWA**

Faculty of Engineering



Non-GPA Module 3992: Industrial Training

**TRAINING REPORT**

**Organization**

Internship Period: 20th of October 2014 to 3rd of April 2015

Name: Dodangoda D.A.P.P

Reg No: 110141R

Department of Computer Science and Engineering

Date of Submission: 20th of April 2015

**Contents**

[1. Introduction to the training establishment 6](#_Toc417320118)

[1.1. Prologue - WSO2 Inc 6](#_Toc417320119)

[1.2. History of WSO2 6](#_Toc417320120)

[1.3. Comparison 6](#_Toc417320121)

[1.4. Products and Services 7](#_Toc417320122)

[1.5. WSO2 Vision 7](#_Toc417320123)

[1.5.1. Reinvent the technology 7](#_Toc417320124)

[1.5.2. Reinvent the Business Relationship 8](#_Toc417320125)

[1.5.3. Reinvent the Support Model 8](#_Toc417320126)

[1.5.4. Create a Great Place to Work 8](#_Toc417320127)

[1.6. WSO2 Support Service Model 9](#_Toc417320128)

[1.6.1. Community Support 9](#_Toc417320129)

[1.6.2. Evaluation Support 9](#_Toc417320130)

[1.6.3. Quick-Start 9](#_Toc417320131)

[1.6.4. Development Support 9](#_Toc417320132)

[1.6.5. Production Support 9](#_Toc417320133)

[1.6.6. Professional Services 10](#_Toc417320134)

[1.6.7. Mailing Lists 10](#_Toc417320135)

[1.7. Teams and Employee Hierarchy at WSO2 10](#_Toc417320136)

[1.8. WSO2 Business Analysis 11](#_Toc417320137)

[1.8.1. Strengths 11](#_Toc417320138)

[2. Training Experience 12](#_Toc417320139)

[2.2. Fast Track Training Project 13](#_Toc417320140)

[2.2.1. Project Description – IgViz (Interactive Generic Visualization Library) 13](#_Toc417320141)

[2.2.2. Getting Started – Inception Phase 13](#_Toc417320142)

[2.2.3. Packing up - Elaboration Phase 14](#_Toc417320143)

[2.2.4. Setting off - Construction Phase 16](#_Toc417320144)

[2.2.5. Winding Up – Transition phase 20](#_Toc417320145)

[2.3. Working with the product teams 21](#_Toc417320146)

[2.3.1. Project Idea - Analytics Dashboard 21](#_Toc417320147)

[2.3.2. Dashboard Architecture & Design 21](#_Toc417320148)

[2.3.3. Dashboard Backend Design as a REST API 22](#_Toc417320149)

[2.3.4. Dashboard Backend REST API Implementation 24](#_Toc417320150)

[2.3.5. Dashboard Backend Design as a SOAP Admin Service 32](#_Toc417320151)

[2.3.6. Dashboard Backend Implementation as a SOAP Admin Service 34](#_Toc417320152)

[3. Annexes 43](#_Toc417320153)

[Figure 1.1 – WSO2 Company Logo 5](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320056)

[Figure 1.2 – WSO2 Employee Hierarchy 9](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320057)

[Figure 2.2 – Samples from d3js.org 13](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320058)

[Figure 2.3 – Modeled dataset sample 13](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320059)

[Figure 2.4 – Milestone plan for the fast-track training project 14](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320060)

[Figure 2.5 – JavaScript Code Segment of *chartConfig* Variable 15](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320061)

[Figure 2.6 – JavaScript Code Segment of *igviz.plot* function 15](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320062)

[Figure 2.7 – Line chart drawn using an unsorted data set 16](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320063)

[Figure 2.8 – JavaScript code segment for sorting a data set 16](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320064)

[Figure 2.9 – JavaScript code segment for appending path to data points 16](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320065)

[Figure 2.11 – Line chart with cardinal interpolation mode 17](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320066)

[Figure 2.12 – Line chart with Step-before interpolation mode 17](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320067)

[Figure 2.10 – Line chart with linear interpolation mode 17](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320068)

[Figure 2.13 – HTML menu for selecting chart configurations 17](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320069)

[Figure 2.14 – JavaScript code segment for loading google geochart 18](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320070)

[Figure 2.15 – JavaScript code segment for drawing the map on an html div 18](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320071)

[Figure 2.16 – JavaScript code segment for setting options for the google chart 18](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320072)

[Figure 2.17 – Regions Chart example 18](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320073)

[Figure 2.19 – Terrain Map example 19](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320074)

[Figure 2.18 – Markers Chart example 19](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320075)

[Figure 2.20 – Miscellaneous chart types provided by IgViz 19](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320076)

[Figure 2.21 – Architecture of IgViz Library 20](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320077)

[Figure 2.22 – Properly configured beans.xml file 22](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320078)

[Figure 2.23 – Properly configured web.xml file 22](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320079)

[Figure 2.24 –Dashboard REST API Package Structure 22](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320080)

[Figure 2.25 – Implementation of pageMetaBean class 23](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320081)

[Figure 2.26 – Implementation of MetaData class 23](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320082)

[Figure 2.27 – High level view of the dashboard service class implementation 24](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320083)

[Figure 2.28 - Method for sending an instance of pageMetaBean as a json response 25](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320084)

[Figure 2.29 – Method for receiving a pageMetaBean instance from a POST request 25](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320085)

[Figure 2.30 – Method for writing objects to the registry in application/json format 26](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320086)

[Figure 2.31 – Method for reading beans from the registry 26](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320087)

[Figure 2.32 – Authentication method in the service class 27](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320088)

[Figure 2.33 – Actual implementation of the authentication function 27](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320089)

[Figure 2.34 – Maven dependency for apache-cxf artifact 28](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320090)

[Figure 2.35 – A snapshot of the terminal after the WSO2 BAM server starts 28](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320091)

[Figure 2.36 – A snapshot of the terminal after the service gets deployed 28](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320092)

[Figure 2.37 – A Snapshot of the Advanced Rest Client 29](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320093)

[Figure 2.38 – Message body of a POST request 29](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320094)

[Figure 2.39 – Response to a successful http request 29](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320095)

[Figure 2.40 – Response for Unsupported Media Type error 30](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320096)

[Figure 2.41 – Unit testing for the REST service using jUnit Test Framework 30](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320097)

[Figure 2.42 – Properly configured services.xml file 31](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320098)

[Figure 2.43 – Package structure of the SOAP admin service 32](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320099)

[Figure 2.44 – High level architecture of the dataView concept 32](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320100)

[Figure 2.45 – Code snippet for the DataViewPrimitive class implementation 33](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320101)

[Figure 2.46 – Code snippet for the DataView child class implementation 33](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320102)

[Figure 2.47 – Code snippet for appending a widget to the existing array of widgets 33](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320103)

[Figure 2.48 – Code snippet for returning a single element from the widget array 34](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320104)

[Figure 2.49 – Code snippet for the widget class implementation 34](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320105)

[Figure 2.50 – Code for reading a dashboard from the registry and returning to the frontend 35](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320106)

[Figure 2.51 – Code for reading a *dataView* from the registry and returning to the frontend 35](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320107)

[Figure 2.52 – Code for adding a *dataView* to the registry 35](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320108)

[Figure 2.53 – Code for returning all the dataViews as an array-list 36](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320109)

[Figure 2.54 – Code for authenticating a user from the SOAP admin service 36](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320110)

[Figure 2.55 – Registry locations allocated for saving dataViews and dashboards 37](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320111)

[Figure 2.56 – Code snippet for reading a resource from the registry 38](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320112)

[Figure 2.57 – Code snippet for closing closeable objects 38](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320113)

[Figure 2.59 – Code snippet for deleting a resource from the registry 38](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320114)

[Figure 2.58 – Code snippet for writing a resource to the registry 38](file:///D:\Academic\Internship%20Aca\Training-Report\Reformat.docx#_Toc417320115)

# Introduction to the training establishment

## Prologue - WSO2 Inc

WSO2 (Web Services Oxygen) is a 100% open source middleware company. As the name implies, WSO2 facilitates varied number of web services which provides enterprise solutions in the web space. Perhaps WSO2 is the only company that provides a comprehensive, open source ecosystem platform for this subject.



##### Figure 1.1 – WSO2 Company Logo

WSO2 has branches in US (Mountain View, CA and Bloomington, IN), UK (London), and Sri Lanka (Colombo 03), Mountain View branch being the Headquarters. Product development is mainly carried out here in Sri Lanka. WSO2 opened two new offices in Jaffna and Maradana Trace Expert City.

## History of WSO2

Initially, the activities of the company focused on providing open source software solutions to the professional software developers. Currently, it focuses on development of software projects based on Service Oriented Architecture which is generally known as SOA.

WSO2 was found by Dr. Sanjiva Weerawarana and Paul Fremantle in 2005. Initially, WSO2 was named as ‘Serendib System’, which later turned into ‘WSO2’ to address a request from an investor.

With huge efforts and dedication, they managed to release the carbon platform which is known as ‘WSO2 Carbon’ by today. WSO2 Carbon is an SOA [*middleware*](http://en.wikipedia.org/wiki/Middleware) platform from WSO2. All the WSO2 products have been built on top of the WSO2 Carbon middleware platform.

## Comparison

Almost all of the IT companies in Sri Lanka, and even in the whole South Asian region, focus on developing software outsourced by other non-IT companies. WSO2, in contrast, is a product development company, similar to the business model of Microsoft or Oracle in that aspect. Nevertheless WSO2 is 100% open source unlike the latter two. In fact WSO2 has only one enterprise ready version for every product, unlike most other open source companies. They have no hidden features or agenda.

## Products and Services

The main advantage, of all WSO2 products, is being built on a common basis - WSO2 Carbon, which is a modular, reconfigurable, flexible OSGi-oriented architecture. This creates a rock solid sound foundation for the construction of large-scale middleware applications.

Currently WSO2 provides 19 products and one development environment plug-in for WSO2 product development, namely

* WSO2 Carbon
* WSO2 API Manager
* WSO2 Application Server
* WSO2 Business Activity Monitor
* WSO2 Business Process Server
* WSO2 Business Rules Server
* WSO2 Cloud Gateway
* WSO2 Complex Event Processor
* WSO2 Data Services Server
* WSO2 Elastic Load Balancer
* WSO2 Enterprise Mobility Manager
* WSO2 Enterprise Service Bus
* WSO2 Enterprise Store
* WSO2 Governance Registry
* WSO2 Identity Server
* WSO2 Message Broker
* WSO2 Storage Server
* WSO2 Task Server
* WSO2 User Engagement Server
* WSO2 Developer Studio

All of these products are free. So from where does all the money come from? The answer is simple, Income comes from support survices. People with a lot of money, but less time, tend to buy WSO2 product related services opening a huge stream of income to the company. (*See* Annex1)

## WSO2 Vision

WSO2s vision is based on *four pillars of innovation,* as they say. The idea is that they are trying to disrupt the competing vendors through a new approach, which is often described by four interesting statements.

### Reinvent the technology

In my words, WSO2 is doing it right. The concept is that, if you do something from scratch, you can make a better version. This is not the same as *reinventing the wheel*. Middleware industry is still not perfect. Therefore doing the same thing from scratch enables WSO2 to find better approaches as well as avoid failure approaches. Final result is a model that fits perfectly to the need, while adapting to changes and opening doors for new and better technologies.

### Reinvent the Business Relationship

Most of the businesses in this world are based on the customers’ assets. The price of a product is usually decided looking at the buyers’ wallet. But this is not what WSO2 does. They have a fixed price for everything. If someone wants to buy their support, the price is fixed. Yet everything is free if the person is unable to pay the price. Still, the free version has all the advanced features as the paid version.

Another point I realized while working for the company is that the company’s profit is not proportional to the number of workers. This is what enables them to expand faster and enable employees to enjoy better salaries and increments.

### Reinvent the Support Model

WSO2 does not have a separate set of support engineers. Support is given by the same people who develop stuff. This ensures that the person who comes for support knows all the underlying connections. WSO2 provides hot fixes, patches, and service packs to keep the installations running smoothly.

In my opinion, what WSO2 believe is that a satisfied customer is way better than a short term profit they might gain by having a cheap support system.

### Create a Great Place to Work

The internal structure of the company is very simple and has a flat hierarchy. A suggestion from the CEO has the same value as one from an intern. This allows all the good choices to get together in a pool from which the best one will pop out eventually. However there are job titles given based on the employees performance.

WSO2 does not force its employees to work until they get exhausted. According to Sanjeeva, the CEO of the company, employees must be there not only physically, but also mentally. Leaves are not recorded. Food, fun activities, leisure time activities and etc are provided without any boundaries. However, at the end of the day, employees do not waste time on just fun activities. They work with their maximum potential with a peace in their minds.

WSO2 makes the employees so good, that they become good enough to leave the company at anytime. On the same time WSO2 treat its employees even better, so they will stay and contribute for its further development.

## WSO2 Support Service Model

WSO2 support model consist of 6 main categories, namely, Community Support, Evaluation Support, Quick-Start, Development Support, Production Support, Professional Services, and Mailing Lists.

Each of these has unique features with the intention of creating a happy and satisfied customer who will stay with WSO2 on the long run.

### Community Support

WSO2 employees will provide answers for any question a user may have via community web sites such as stack-overflow.

### Evaluation Support

WSO2 technology experts guide customers in the early stages of middleware projects in technology selection, product selection/evaluation and migration/integration strategies. The following services are provided free of charge to qualified customers

* Middleware architecture consulting
* Developing Requests For Information/Proposal (RFI/RFP)
* Proof of Concept implementations

### Quick-Start

The Quick-Start is a rapid ramp-up program that brings world class expert developers and architects on-site to work collaboratively with the customer. For this WSO2 provide two on-site engineers and one off-site engineer in one week.

### Development Support

WSO2 does not have separate set of support engineers. The same developers get into the game to provide the customers with a variety of services. Some of the services requested often

* Assistance with product tuning for performance, security and other needs.
* Developing proof-of-concept implementations.
* Access to selected pre-production patches.

### Production Support

WSO2 has designed a support mechanism that guarantees WSO2 middleware infrastructure enables the client application to be available 24x7x365. This is a vital requirement for customers such as *ebay*, who will lose thousands of dollars in a matter of seconds in case of a break down.

### Professional Services

WSO2 provides 3 types of professional services, Consultancy Services, Development Services, and Custom Development and Open Source Sponsorship

### Mailing Lists

WSO2 has several public mailing lists. Out of them the [dev@wso2.org](mailto:dev@wso2.org) mailing list is a very helpful one which helps customers as well as new employees to solve issues and doubts in a matter of minutes. Other mailing lists are also there for *Jira* issues and anything that may come up.

## Teams and Employee Hierarchy at WSO2

The employee is just for Human Resource Management purposes. All other activities such as decision making, salary, increments and promotions are independent of this system. Most of them are done on credit basis. However, for decision making, all the employees including interns and even the cleaning staff can equally contribute.



##### Figure 1.2 – WSO2 Employee Hierarchy

**The finance team** - Finance team takes care of accounts, income, expenditure and budget of the company

**The Engineering team** - Engineering team is the largest team in the WSO2. The engineers who work on research, design, development and testing are in this category. Technical groups (TG) under the Engineering team and the products being developed by them are as shown below.

* **Developing TG**: Application server, Tooling, Jaggery server
* **Solution TG**: App Factory, API Manager
* **Data TG**: Data Service server, Business Activity Monitor, Complex Event Processor,

Business Rules server

* **Integration TG**: Enterprise Service Bus, Identity server, Message Broker, Governance

Registry, Business Process server, Elastic Load balancer

* **Foundation TG**: Carbon, Stratos

**The marketing team** - Marketing team works on marketing the WSO2 products in the software industry by means of sponsorship, advertising campaign, organizing conference, workshop, webinars and etc. Most of the events are organized by the marketing with the help of whole wso2 team.

**The sales team** - Sales team deals with the customer and maintaining the connection between customer and the developers.

**The HR and administration team** - Provides vital administration and human resource work like recruiting employees, salary payments and etc.

## WSO2 Business Analysis

### Strengths

# Training Experience

## 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, did not consist of a killer mechanism to visualize data. This could 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, was 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,

* It provides several reusable and advanced functionalities
* It has gained a good reputation over time 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 . – Samples from d3js.org

During this time, I focused on learning the core technologies that will be used for this project. *The-NewBoston'*s free video tutorial series 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,

* Proposed Core Technologies: JavaScript, d3js, WebStorm IDE, Sublime Text IDE.
* Proposed Core Deliverable(s): A generic JavaScript library

### 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: summarizes the data set into a single number representation.
* Line Chart Diagram: change in one dimension against a unit change in the other
* 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: correlation of data of variables 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 came up with a mechanism to accomplish these using a *JSON* skeleton to which a data set will be modeled into.

"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]

]

}

##### 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.

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 to implement the line chart and the map diagram. I started my work with the line chart diagram. However while working on our individual tasks, we also 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'),

"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 w.r.t X coordinates
* Mapping coordinates according to a scale
* Connecting coordinates and *interpolation*
* Coloring each line chart

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

* 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.

I solved this issue 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) {

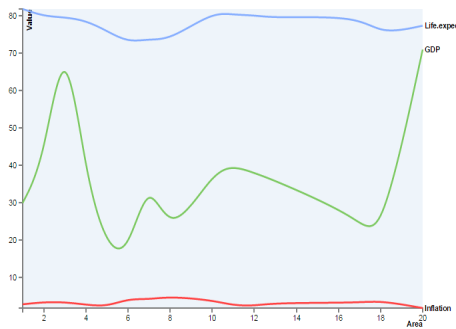
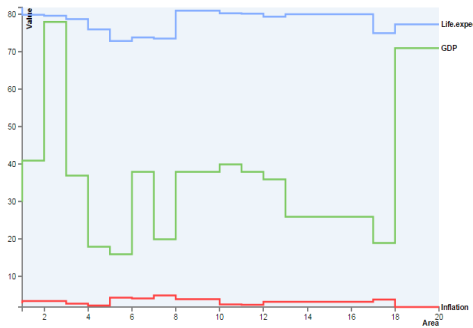
return getColor(i % 3)

});

##### 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 charts drawn in 3 such interpolation modes

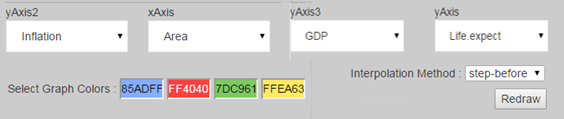


##### Figure 2.11 – Line chart with cardinal interpolation mode

##### Figure 2.12 – Line chart with Step-before interpolation mode

##### Figure 2.10 – Line chart with linear 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

#### Map Diagram Implementation

A major issue I faced while implementing this chart type was d3 not supporting advanced functions for map manipulation. The abstract map provided by d3 is very basic. The max possible zoom-level limited which means that users cannot visualize a data set local to a country like Sri Lanka.

Therefore I had to import another library to lay the foundation for the Map-diagram. After some research, I came across two possibilities

* Google visualization charts
* D3 based Data Maps

Considering the ease of use and the list of supporting functionalities, Google visualization charts library appeared to be a better candidate.

However, one major con of using google-charts was identified as the inability to draw the chart offline. But considering the use case of the IgViz library, this trade off was evaluated as profitable.

**Important code quotations**

The LoadMap() function is responsible for fetching the chart definitions from google. This google.load(…) function can be seen as a common feature in all kinds of google charts. It should be noted that this does not load the map; it is responsible only for loading the chart definitions.

function LoadMap() {

google.load('visualization', '1', {'packages': ['map', 'geochart']});

}

##### Figure 2.14 – JavaScript code segment for loading google geochart

After loading the definitions, the following code segment will draw the map.

var chart = new google.visualization.GeoChart(document.getElementById('chart\_div'));

chart.draw(data, options);

##### Figure 2.15 – JavaScript code segment for drawing the map on an html div

In the above function, the ‘options’ parameter can be modified to set custom map configurations. Shown below is an example that I have set in the IgViz library.

var options = {

region: regionO,

displayMode: mode,

colorAxis: {colors: ['red', 'blue']},

magnifyingGlass: {enable: true, zoomFactor: 3.0},

enableRegionInteractivity: true

};

##### Figure 2.16 – JavaScript code segment for setting options for the google chart

In the above variable, the displayMode can be one of

* Regions Chart
* Markers Chart
* Terrain Map
* Normal Map

Regions Chart

This chart type is used when the data table contains countries in the location field. If this chart type is selected in a case where the location field contains cities, google will map the city name with a country and then draw the chart. Shown below is a *Regions Chart* generated from our IgViz library. The styles are set from the *options* variable (see Fig. 2.16).



##### Figure 2.17 – Regions Chart example

When this chart type is selected, the complete world map will be shown by default. To zoom to a specific region, the region number should be selected from the *ISO 3166 Country Codes* list.

E.g. - 150: Europe, SL: Sri Lanka, US: United States

Markers Chart

This chart type can be drawn using any location on the google map as the location field. The concept is similar to a bubble chart in which we plot 3 dimensions on a two dimensional chart. The three dimensions are identified by location, point color and point size.

I have implemented this chart type in such a way that the user does not have to select the region explicitly. The region covering all the points will be selected internally.

Terrain Map

This is a map rather than a chart. I have added this type for users to visualize their data set on a google map. This enables the users to use any of the general features (panning, zooming, etc) provided by the world famous google maps. Terrain Map and Normal Map are quite similar. The only difference is that normal maps do not consist of the terrains.



##### Figure 2.19 – Terrain Map example

##### Figure 2.18 – Markers Chart example

### Winding Up – Transition phase

In this phase we merged our individual tasks into a single JavaScript file. We also created an html page to show samples of what we have accomplished so far. Types of charts provided by the library, in addition to the ones I created, are shown below.



##### Figure 2.20 – Miscellaneous chart types provided by IgViz

**Product Architecture**

The architecture of the IgViz library we implemented can be illustrated as shown in the diagram below.

Java Script

Google Charts

D3js

IgViz Library

##### Figure 2.21 – Architecture of IgViz Library

## Working with the product teams

After completing the fast track project, I was assigned to the *WSO2-Business-Activity-Monitor* (BAM) team. Then I attended the small introductory session on which we were welcomed and advised on how to work during the coming weeks.

**Team**: WSO2 Business Activity Monitor (BAM)

**Team Lead**: Anjana Fernando (Senior Tech Lead)

**Assigned Supervisor(s**): Anjana Fernando & Dunith Dhanushka(Senior Software Engineer)

### Project Idea - Analytics Dashboard

As advised, I started working on my first project as a BAM team member. The task was to design and implement a dashboard for BAM and *CEP* (WSO2 Complex Event Processor). The existing dashboard was complex and had numerous bugs. So the idea was to create a minimalistic dashboard targeting the two products, BAM and CEP. The dashboard should provide the following functionalities

* Login with credentials
* Create widgets
* Create new sub dashboards
* Add widgets to custom dashboards and display
* Logout

### Dashboard Architecture & Design

We took the following design decisions at the start of this project. These decisions got the approval from the WSO2 Architecture mail thread.

Frontend

Front end of the dashboard will be designed using the following technologies

* HTML/CSS
* JavaScript
* Bootstrap Framework
* GridsterJS
* jQuery

Backend Design and Hosting

The backend of the dashboard will be a Jax-RS web service. This means that there will be a RESTful API to which the frontend developer can send requests for data communication. This will be exported as a web archive file (.war) using maven as the build-tool and will be deployed on the WSO2 BAM server as a web-app.

Work Allocation

I **was assigned to implement the backend** while Dunith and Fawsan proceeded with the frontend development. We shared a git repository for both frontend and backend implementation.

### Dashboard Backend Design as a REST API

The primary functionality expected from the backend is sending and receiving Json/Xml requests and storing them in the WSO2 registry. However support for Json is courage over xml.

I created a new Jax-RS project from WSO2 Developer Studio which is the development tool provided by WSO2 for writing WSO2 related products. WSO2 developer studio is a complete Eclipse-based SOA development environment for the award-winning WSO2 Carbon platform.

When a new Jax-rs service project is created most of the configuration stuff is automatically created. However I had to manually modify the web.xml file and the beans.xml file in order to set the service class and the underlying connections.

Shown below in Fig.2.22 is a screenshot after the configurations were properly done in the web.xml file. The beans.xml file should also be configured to define the service class. This configuration is shown in Fig. 2.23.



##### Figure 2.22 – Properly configured beans.xml file

##### Figure 2.23 – Properly configured web.xml file

After configuring the web.xml file and the beans.xml file, I created a package structure with the required service class and the associated bean classes. In the screenshot given below, the complete package structure of the project is shown. In addition to the classes mentioned above, the screenshot shows the UserAdminClient.java class which was added in a later stage for authenticating a user.

In this package structure, the DashboardConf.java file contains the service class implementation.

The class files that have a name ending with “Bean” are bean classes. A bean class is a simple java class used for data storage purposes. Though not forced, a bean class has a fixed structure in general. There are no complex constructors. Class methods are added only for data manipulation, usually getters and setters will do the job.

Since this is a REST service, these bean classes

##### Figure 2.24 –Dashboard REST API Package Structure

must be annotated with JaxB annotations.

This will be explained in detail later in this document.

### Dashboard Backend REST API Implementation

The implementation process was carried out as two sub processes, service class implementation and bean classes’ implementation. Preceding sections describe these two processes exclusively.

#### Implementing the bean classes

As explained earlier, I designed the bean classes following the general practices. Shown below are the implementations of *pageMetaBean* class and its associated *MetaData* class.





##### Figure .25 – Implementation of pageMetaBean class

##### Figure .26 – Implementation of MetaData class

Methods of these two classes are simply used for setting and getting data. However it can be noted that there are special annotations used above the class as well as its fields.

I added these *JaxB* annotations in order to make them visible from outside via the REST API. The annotations used and the purpose of each is briefly explained below.

* **@XmlAccessorType** : I have set this to *XmlAccessType.Field*. This makes sure that every field defined in the class is bound to xml unless specified exclusively.
* **@XmlType** : I used this annotation to specify the proper order of the fields.
* **@XmlRootElement** : This annotation is used to bind the class to xml.
* **@XmlElement** : I added this as an extra option just in case someone wanted to

change the name of the field without disrupting the xml.

#### Implementing the service class

The figure below shows the high level implementation of the service class. Important methods of the class will be explained later. It is to be noted that I have set the return type of several methods as “Response”. These are the methods which will be visible to the outside via the REST API after deployment.



**Logger**: The logger is used to log exceptions before they are thrown. Any other important information can also be logged as info or debug. I added this logger as a supplementary mechanism to trace errors or exception which might be raised after the service is deployed.

All other fields are instances of the bean classes defined earlier which will be converted to and back from json/xml strings.

##### Figure .27 – High level view of the dashboard service class implementation

**Annotating the service class**

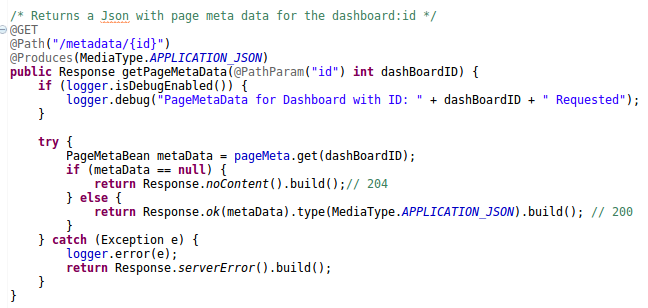
I have used 3 types of annotations in addition to the ones that are used for identifying the http request type. Brief descriptions of the annotations that I have used are given below.

* **@GET**: Http request type for receiving data by the client side.
* **@POST**: Http request type for receiving data by the server side.
* **@DELETE**: Http request type for deleting data stored in the server side.
* **@Path**: Used for identifying the service endpoint of the desired method.
* **@Produces**: Format of the response sent from the server side.
* **@Consumes**: Format of the request expected by the server side.
* **@PathParam**: Used for extracting method parameters from the service URL

The latter two annotations are used to decide the content type among the three possibilities, *application/json*, *application/xml* and *plain/text*.

**Important service methods**

I wrote the following method for sending an instance of the pageMetaBean to the frontend client as a response with a json string as the response body. The content type is set to application/json.

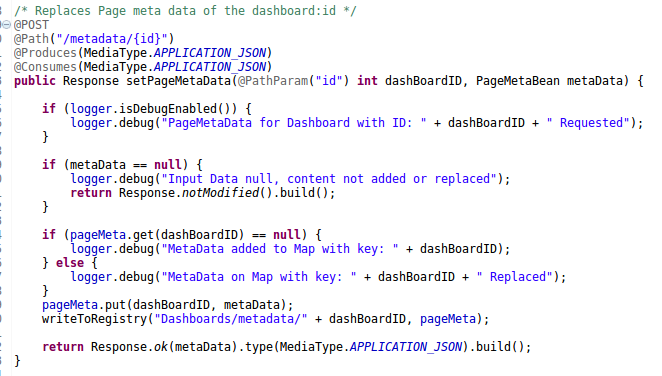


##### Figure .28 - Method for sending an instance of pageMetaBean as a json response

If examined carefully, it can be seen that there are response codes which will be sent along with the response. I have used standard response codes for the ease of use from the frontend. Some of the well known response codes which I used and encountered are,

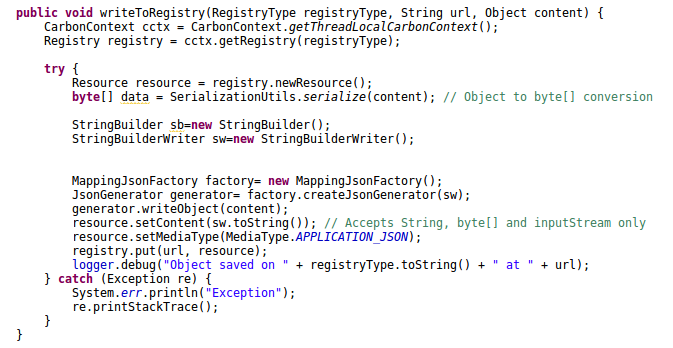
* 200 – OK
* 204 – No Content
* 304 – Not Modified
* 403 – Forbidden
* 404 – End Point Not Found
* 415 – Media Type Unsupported
* 500 – Internal server error

The code segment of a method used for handling POST requests is similar to this. However the accepting request content type should be defined exquisitely using the @Consumes annotation.



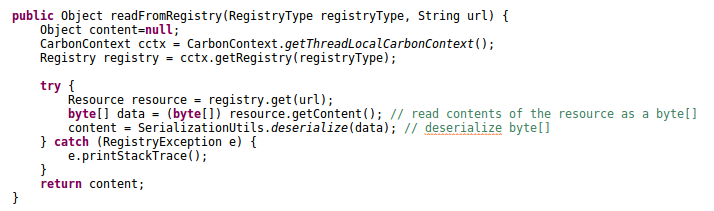
##### Figure .29 – Method for receiving a pageMetaBean instance from a POST request

For the registry access purposes, I created two methods, *writeToRegistry* and *readFromRegistry* respectively for writing to and reading from the WSO2 registry. The most straight forward solution for registry access would have been implementing a separate method for each class. But that approach would be a hindrance to the code extensibility.

After a long effort, I found a mechanism to do these two tasks using two generic methods. I have also added code segments in these methods for the type conversion between bean objects and json strings.

##### Figure .30 – Method for writing objects to the registry in application/json format

There are several external libraries which can be used for the conversion between beans and json strings. For this REST service I used the *MappingJsonFactory* provided by the Jackson dependency to do the parsing. However, it is to be noticed that there are other similar libraries such as google gson which can be used for the same purpose.



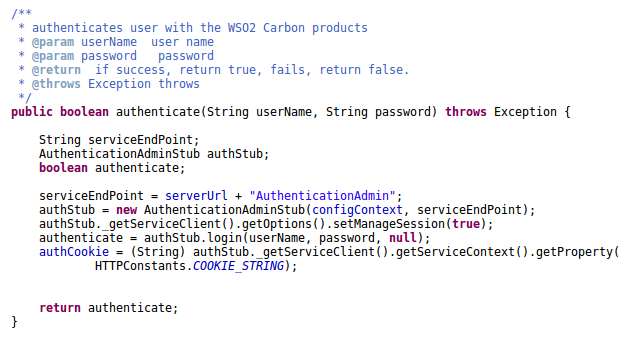
##### Figure .31 – Method for reading beans from the registry

Apart from the ones similar to the above described methods, one other important method was implemented in the service class. I wrote this method for the purpose of authenticating a valid user. The method written in the service class does not do any major tasks. The authentication is done by actually done by the *UserAdminClient* class. Nevertheless, the method calls must be organized correctly in order to accomplish the authentication correctly.



##### Figure .32 – Authentication method in the service class

I wrote the code segment responsible for the actual authentication inside the *UserAdminClient* Class. The method containing the specific code segment is shown below.



##### Figure .33 – Actual implementation of the authentication function

#### Building the web archive for deployment

I have used maven as the build tool for this service. A Project Object Model or POM is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the project. It contains default values for most projects.

Configurations that can be specified in the POM are the project dependencies, the plug-ins or goals that can be executed, the build profiles, and so on. Other information such as the project version, description, developers, mailing lists and such can also be specified.

The following xml code snippet was added to this project to import the libraries related to the apache-cxf artifact which is a fundamental requirement when writing a Jax-RS service.



##### Figure .34 – Maven dependency for apache-cxf artifact

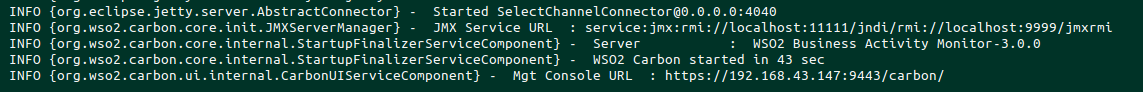
All other dependencies and plug-ins which are required for the build were added to the POM in the same format.

Running “maven clean install” command, from either the terminal or via eclipse, downloads and installs all the specified dependencies and plug-ins to the development environment.

When the POM is configured properly, a web archive file can be exported directly from the Eclipse IDE. The “.war” file is saved inside the BAM product under the *web-apps* directory.

#### Jax-RS service deployment

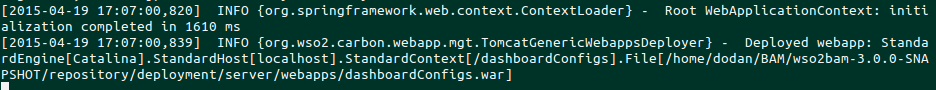
After the web archive is placed inside the web-apps directory, I started the WSO2 BAM server. The server was up and running after few seconds.



##### 

##### Figure .35 – A snapshot of the terminal after the WSO2 BAM server starts

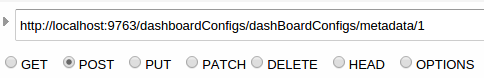
If there are any web archives inside the web-apps directory already, they will also be deployed. Moreover, even if a web archive is placed inside the directory after the server starts, they will also get deployed dynamically.



##### 

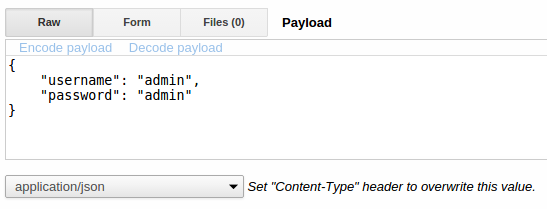
##### Figure .36 – A snapshot of the terminal after the service gets deployed

I used the chrome extension, Advanced-Rest-Client to for testing the service with mock requests. This extension is also free and open source.

Advanced-Rest-Client provides options to configure everything that is needed by an http request. The menu snippet for specifying the service endpoint and the request type is shown in the following figure.

##### Figure .37 – A Snapshot of the Advanced Rest Client

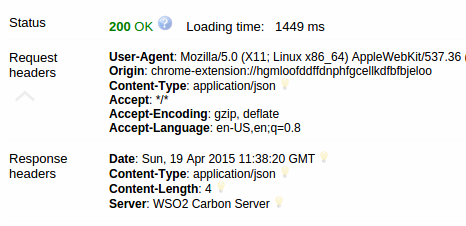
When the request type is selected as POST, PUT or PATCH, the request body can be used to add the data to be sent. Content-Type of the body also can be specified before sending the request.



##### Figure .38 – Message body of a POST request

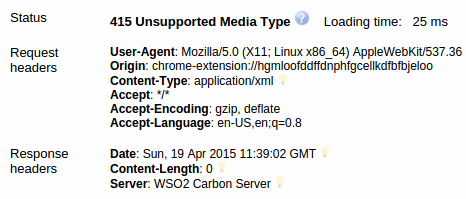
If the service has no bugs and the request is configured properly and sent, a response will be received with a status code ‘200’.

##### Figure .39 – Response to a successful http request



However, I had to try numerous times to get this response. During this mock testing period, I received a lot of error messages, exceptions and logic errors etc, which were debugged and fixed to get this expected result.

Shown in the next diagram is one of such issues I faced, the Unsupported-Media-Type message.



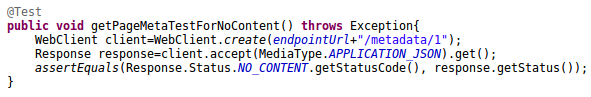
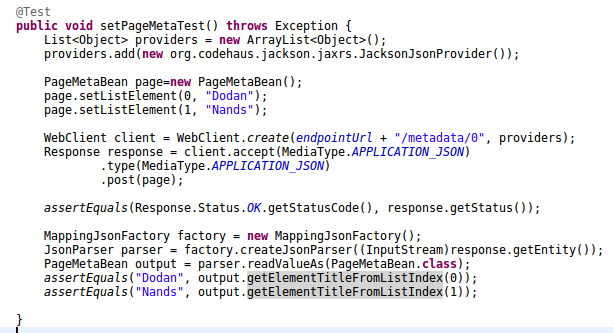
##### Figure .40 – Response for Unsupported Media Type error

This issue was caused due to a mistake I had done while annotating the bean classes. After annotating the beans correctly, this issue was fixed.

Another issue was found when trying to authenticate a valid user via the WSO2 user authentication stub. This SSL error was caused due to an issue in the path given for the java-key-store. Removing this key-store solved the problem. The concept behind the solution is that the service is deployed local to the BAM server. Therefore there was no need to define the key-store explicitly.

#### JUnit Testing for the service

I wrote several JUnit test cases for this service. Unit testing was done in a similar manner to that of a standard JSE project. The only difference is that I had to deploy the service first inside the BAM server. Then I ran the server. While the server is running, I executed the unit test cases from the eclipse IDE as a separate project.



##### Figure .41 – Unit testing for the REST service using jUnit Test Framework

According to the current architecture, the frontend of the dashboard is hosted outside the BAM server. After a long discussion with the stakeholders and other related personnel, it was decided to change this architecture and move the frontend to inside of the server.

Then the dashboard will be viewed as a new component in the WSO2 BAM Admin Console. Therefore the REST API service had to be replaced by a new *SOAP Admin Service*.

The backend admin service, frontend *UI component* and the *service stub* can now be bundled as an *OSGI bundle* which will eventually be deployed as *a Carbon Component*.

Therefore my next task was to implement the backend of the analytics dashboard as a SOAP Admin service component.

### Dashboard Backend as a SOAP Admin Service

High level architecture of the dashboard backend remains almost the same as the earlier case. Therefore what I had to do is porting the REST service into a SOAP Admin Service. I created a new project from IntelliJ IDEA IDE and did the fundamental configurations to make the project into an *Admin service*.

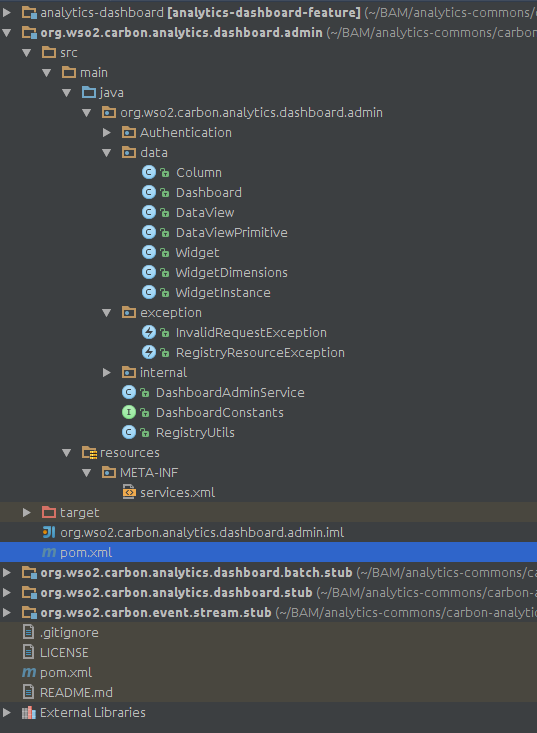
First I configured the *services.xml* file as shown below. This is the place where the service class is specified. The name given inside the <service> tag will be displayed when the service is deployed on the carbon server.

##### Figure 2.42 – Properly configured services.xml file

* **<transport>https</transport>** : makes the service secure.
* **parameter name="adminService"** : makes the service to an ***admin service***.
* **parameter name="hiddenService**" : makes the service hidden to public.
* **parameter name="AuthorizationAction"** : makes the service authorized.

Next I created a better package structure, allocating a separate *util-class* for registry related operations.

Methods inside the util class are decorated with the *public* access modifier. This is to enable the util class to be used by any outside or inside package.

Moreover those methods have the *static* keyword in their method declarations. This makes sure that the methods can be called without instantiating the RegistryUtils Class.

Method calls to a util class are in the format, *RegistryUtils.read( ).*

I modified the structure of the data classes and their connections to match the new dashboard architecture. The main reason behind this major change is the invention of the *dataView* concept. Unlike in the previous case, widgets are now tightly bound to the corresponding data source. The data source and the related characteristics such as columns, filters etc are now bundled into a *dataView* object. Widgets can be defined inside these *dataViews* which can later be instantiated and placed inside a dashboard.

widget1

widget2

widget N

id

name

type

dataSource

filter

column111

column2

column N

##### Figure 2.43 – Package structure of the SOAP admin service

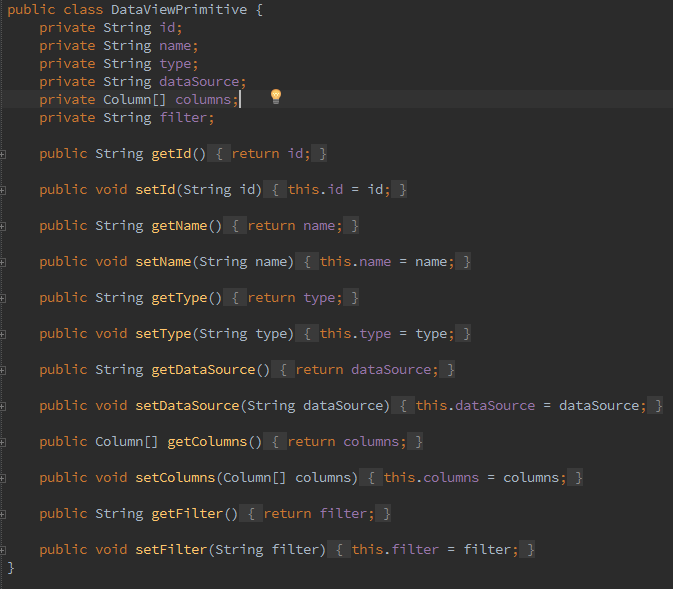
It is to be highlighted again that widgets will not be defined inside the dashboard. An instance of a widget defined inside a *dataView* will be shown which will be coupled by the dimensions and the relative position on the respective dashboard.

##### Figure 2.44 – High level architecture of the dataView concept

### Dashboard Backend Implementation as a SOAP Admin Service

#### Implementing the bean classes

I did the java implementation of the above design decision in such a way that a parent class will hold all the primitive fields while an extended child class will hold the widget array.

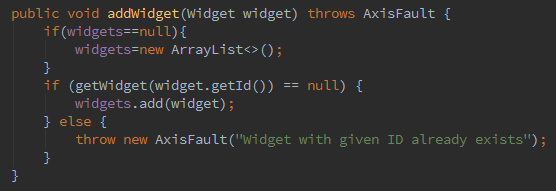


##### Figure 2.45 – Code snippet for the DataViewPrimitive class implementation

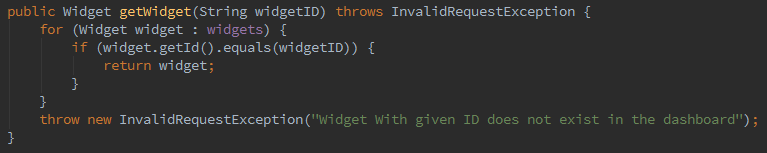
The DataView child class which inherits the DataViewPrimitive class has nothing else but an array of widgets and the associated getters and setters.

D:\Academic\Internship Aca\Training-Report\Selection_035.png

##### Figure 2.46 – Code snippet for the DataView child class implementation

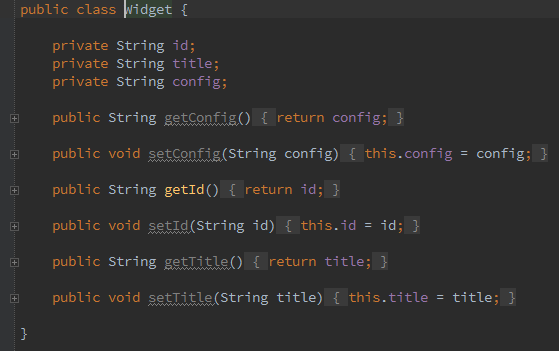
In addition to the auto-generated getters and setters, I added few more methods to get and set values of array fields element by element. The *addWidget* method appends a new element at the end of the existing array-list of *widgets* while the *getWidget* method returns a single *widget* using the array-list index as the method parameter.

##### Figure 2.47 – Code snippet for appending a widget to the existing array of widgets



##### Figure 2.48 – Code snippet for returning a single element from the widget array

A widget in this case is defined as a simple java class with 3 string fields. A widget has no meaning when viewed as a solitary class. In other words, widgets are tightly coupled with *dataViews.*



##### Figure 2.49 – Code snippet for the widget class implementation

#### Implementing the Service Class

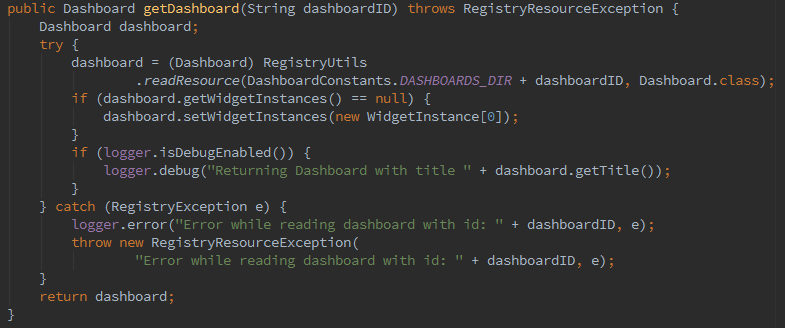
The service class implementation was done by giving special attention to several functional and non functional requirements.

I implemented service class methods to accomplish the following tasks

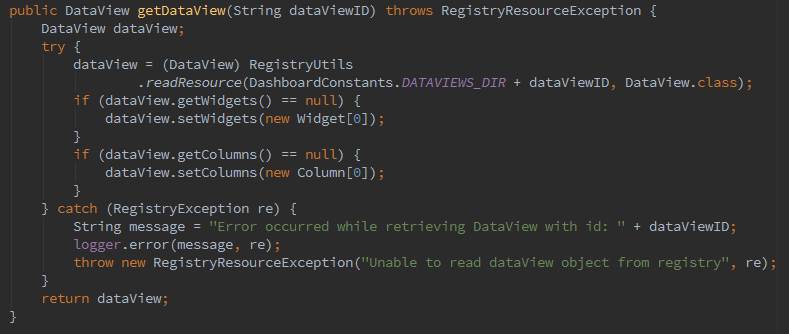
* Get *dataView* from *dataViewID*
* Get all existing *dataViews* as a list
* Add new *dataView*
* Update/Delete Existing *dataView*
* Add/Update/delete a widget to/from an existing *dataView*
* Get dashboard from *dashboardID*
* Get all existing dashboards as a list
* Add new dashboard
* Update/Delete Existing dashboard
* Add/Update/Delete a *widgetInstance* to/from an existing dashboard
* Authenticate a Valid User

Note: All the registry related methods were implemented in a separate *util* class.(*see* section 2.3.6.3)

The two methods for returning a dashboard and a *dataView* respectively are similar in structure. I have carefully implemented the casting process so that consistency and impeccability shall be conserved.

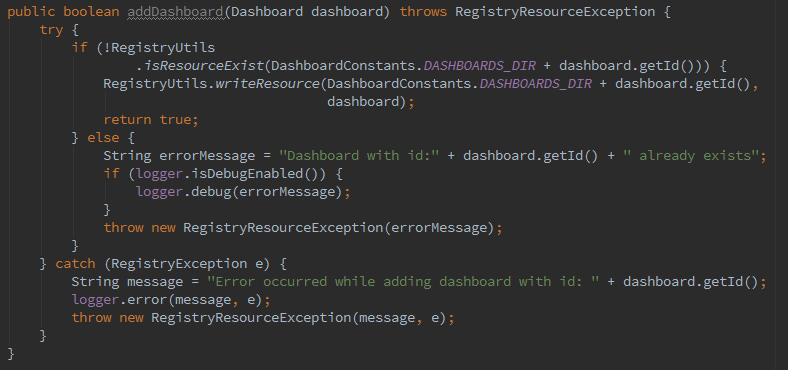


##### Figure 2.50 – Code for reading a dashboard from the registry and returning to the frontend



##### Figure 2.51 – Code for reading a *dataView* from the registry and returning to the frontend

I gave extra effort to find and fix any possibilities that a null pointer may be thrown. An unexpected null pointer at runtime is considered as a ferocious weakness of a programmer.

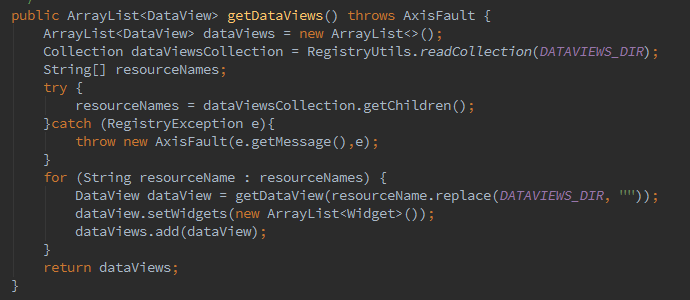


##### Figure 2.52 – Code for adding a *dataView* to the registry

However the code for reading and returning all the *dataViews* as a list is complex. This method is executed as three main sub tasks.

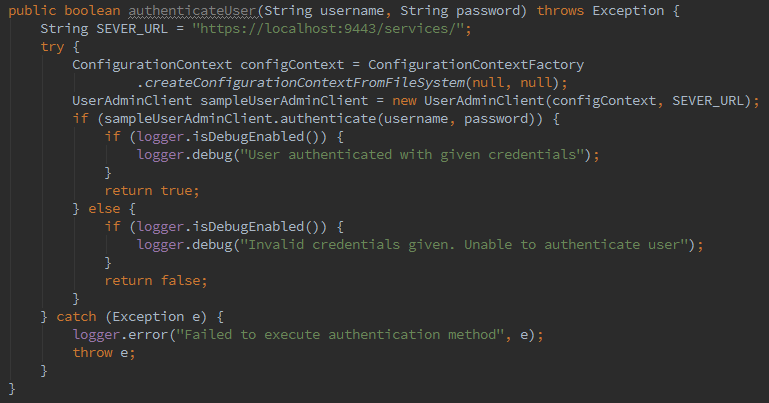
* Getting the name list of *dataViews*
* Reading each *dataView* from the registry
* Creating a list with each *dataView* read and returning as an array-list

There are few advanced concepts associated with registry access. These concepts are explained in the section 2.3.6.3.



##### Figure 2.53 – Code for returning all the dataViews as an array-list

Authenticating a user is done in a similar manner to that of the REST service. However this method was modified with added loggers and exceptions to maintain good programming practices.



##### Figure 2.54 – Code for authenticating a user from the SOAP admin service

The *userAdminClient* was directly ported to this service and no any changes were done.

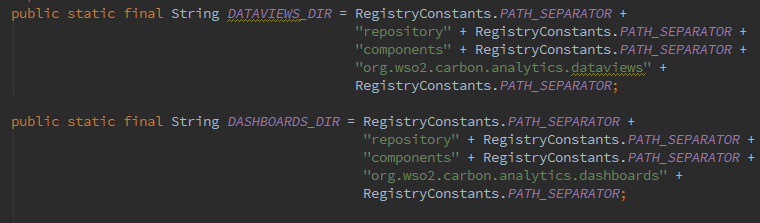
One important thing I learned while working for WSO2 is the importance of maintaining code quality. Though it is possible to implement all the functional requirements without worrying about code quality, it hinders further extension and maintenance.

One practice I followed to maintain code quality is implementing codes according to a model. The CRUD model which stands for Create-Read-Update-Delete. I implemented codes according to this model for all the fields that will be sent and received via requests.

#### Working with the WSO2 Registry

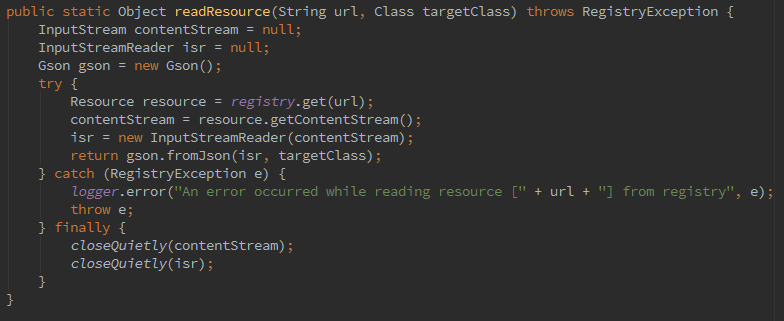
Working with the registry is a vital skill needed when writing a service for the WSO2 carbon.WSO2 registry is consist of three components, namely, *Governance*, *Local* and *Configuration*. As emphasized earlier, I organized everything related to registry access inside a separate *util* class, *RegistryUtils*.

I created two new directories under the *system governance* registry for saving *dataViews* and dashboards separately. These objects are saved in the registry as *resources*. A directory inside the registry is collection of resources.



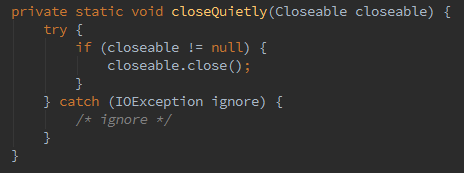
##### Figure 2.55 – Registry locations allocated for saving dataViews and dashboards

Two of the most straight forward methods implemented in the *RegistryUtils* class are *readResource* and *writeResource*. I had to modify the two methods when porting from them REST service since the *JacksonJson* parser is not supported by SOAP. I replaced the Jackson parser with *gson*.

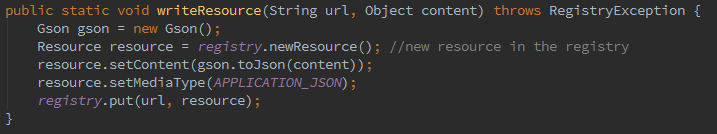
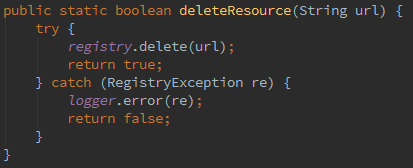


##### Figure 2.56 – Code snippet for reading a resource from the registry

Reading the resource was done using an *InputStreamReader*. Whenever an *InputStream* is declared and used, it should be closed. Programmers usually write a code snippet inside the *finally* clause directly. As it seemed odd, I wrote a separate method to close closeable objects. Only the method call will be placed inside the *finally* clause



##### Figure 2.57 – Code snippet for closing closeable objects



##### Figure 2.59 – Code snippet for deleting a resource from the registry

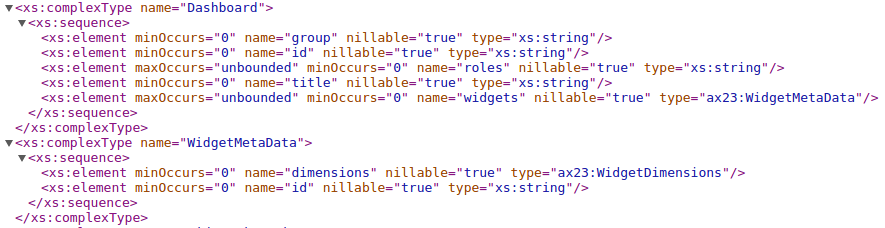
##### Figure 2.58 – Code snippet for writing a resource to the registry

#### Service deployment

After writing the code to a considerable level, I focused on the service deployment process. The most important element that comes to play when implementing a SOAP web service is the service WSDL file. WSDL stands for Web Service Definition Language. This file usually specifies the service location and the associated operations which are exposed by the service.

To obtain the WSDL for this service, the service should be deployed first. I exported the service as an OSGI bundle and placed inside the *dropins* directory of the BAM server. When the server starts, the OSGI bundle also gets activated along with the server.

To discover the service end point of this service, I started the WSO2 BAM server with OSGI console enabled and then used the OSGI command, “listAdminServices” from the terminal. The WSDL of the service is exposed from this ***epr****.* Then I changed the <HideAdminServiceWSDLs> parameter from true to false from <CARBON\_HOME>/repository/conf/carbon.xml. This was done because the WSDL will not be exposed when the connection is secured (as in by default).



D:\Academic\Internship Aca\Training-Report\Selection_065.png

##### Figure 2.60 – A Code segment of the WSDL exposed from the Admin service

This service can be tested using the SOAP-UI application by uploading or referencing the service wsdl file.

#### Problems faced and solutions found

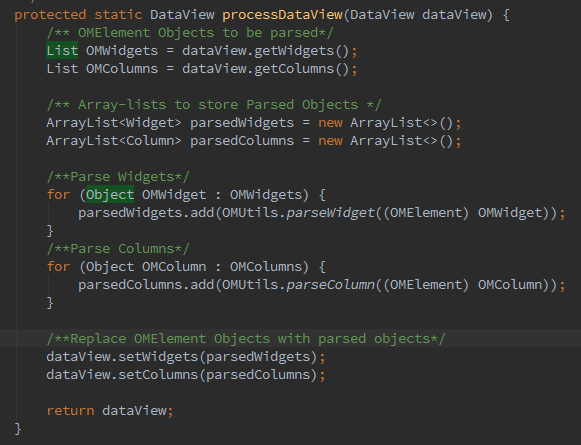
Problem 01:

When the service wsdl was given to the SOAP-UI, it gave an error related to the use of *enums*. After some research, I got to know that it is not a good practice to have enums in the service implementation code.

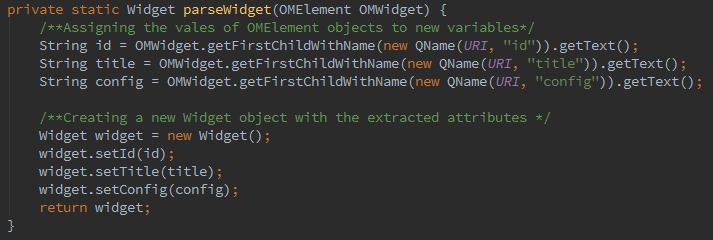
Solution: I replaced all the enums defined or used in the service implementation with other alternatives such as strings. Then the wsdl was accepted by the SOAP-UI.

Problem 02:

When a request is sent from the SOAP-UI (*see* section 2.3.6.6.), automatic parsing was done for all types of data types except for the use cases of array-lists. This issue was discovered using **remote debugging**. The variables, which are ought to hold a specific data-type, were holding OMElements instead, when array-lists were involved.

Solution A: The first solution I came up with is to parse these array-lists manually by writing a separate utils class. I wrote a method to process each object which recursively calls few other methods to parse its attributes.

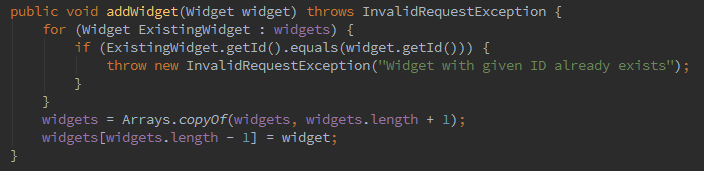
##### Figure 2.61 – Method which calls other methods recursively to parse the OM Element



##### Figure 2.61 – Method which parses a Widget from an OM Element

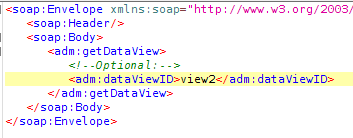
Solution B: Above described solution-A solved the problem. However Anjana advised me not to use array-lists in the service class implementation. So I replaced all the array-lists by arrays. Methods which used the language specific feature were also replaced by simpler methods.

This came out to be a better solution than the first as it does not require any extra methods to be written and also adds extensibility to the code.

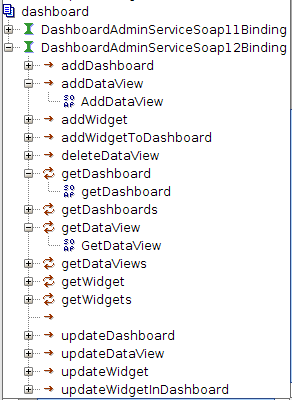


##### Figure 2.62 – The addWidget method after array-lists are replaced by arrays

#### Testing the service with SOAP-UI

When the WSDL file was given to the SOAP-UI, it automatically generates mock soap requests which can be modified with custom values. A mock soap request is an xml type message. I created and sent mock requests for all the methods and got responses. The responses are also in xml format. I had to fix few errors in some cases.

##### Figure 2.63 – Mock SOAP request for getDataView Method



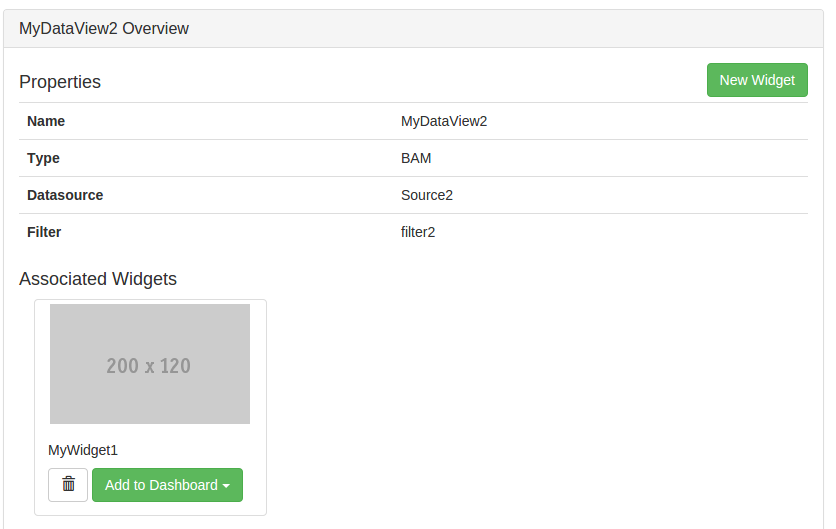
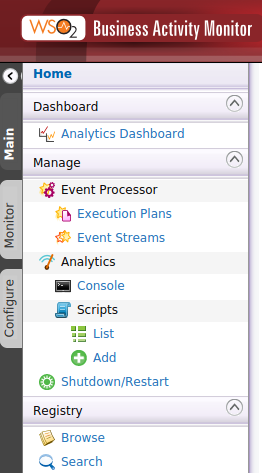
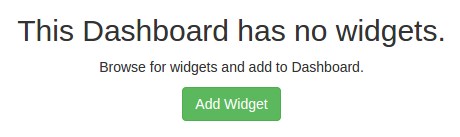
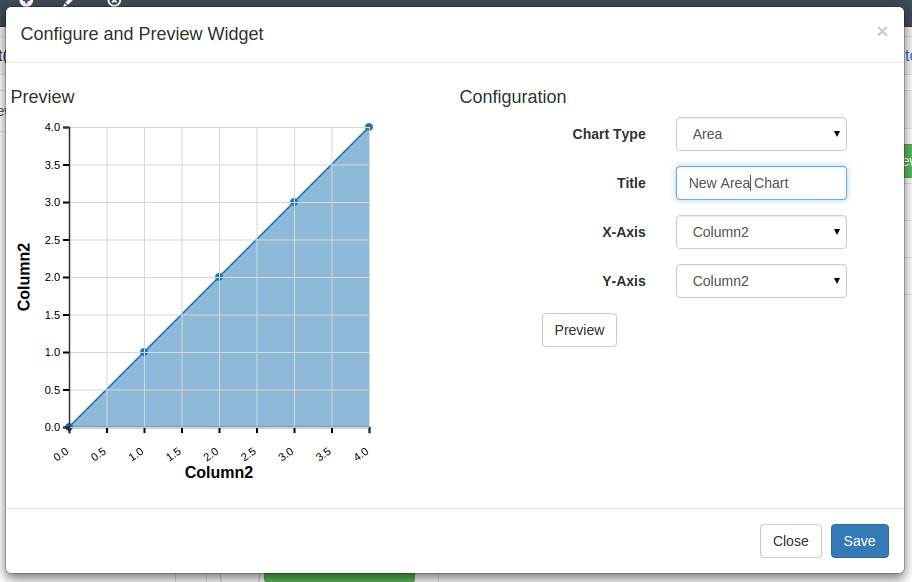
##### Figure 2.65 – Response SOAP message for the request in Fig. 2.63

##### Figure 2.64 –Auto generated SOAP request skeletons for the Admin Service

#### Frontend implementation

I generated the service stub of the service to be used by the frontend component to send and receive requests from the backend. Using wsdl2java along with maven-antrun-plugin, I generated the Java classes from the existing WSDL document. If the dependencies and the plug-ins are set correctly, running a maven build alone is enough to accomplish this task.

Dunith implemented the frontend UI component using the backend service and the service stub I generated. After completing the frontend work, the dashboard could hold widgets generated from the IgViz library we created. (*see* section 2.2.1)

D:\Academic\Internship Aca\Training-Report\Selection_069.png

##### Figure 2.66 – a collection of screen shots of the Analytics-Dashboard Implementation

### Back to the IgViz

Last few days of my internship was dedicated back to the IgViz library. By this time, the library had been improved by Tharindu, using Vega, a declarative format for creating, saving and sharing visualization designs. Most of the chart types we initially designed had better styling and functionalities.

One of the key challenges that Tharindu had been facing was the large code base which is difficult to maintain. Therefore I and Tharindu decided to *refactor* the code. Tharindu started splitting the code into several smaller JS files while I started on *GruntJS*, a *task runner* for JavaScript.

#### Repetitive task automation with GruntJS

The main reason for using GruntJS in this project is to provide extensibility. When we split the code into several smaller JS files, we will have to include the references of each of them in our HTML page which is a very inefficient task. Therefore we have to *concatenate* the code into a single JS file before distribution. It is possible to do this for one time, without a task runner. But then again, extensibility is lost.

Another reason is for *minifying*. Minifying a .js file can reduce its size nearly by a factor of 2 or 3 or even more, which can result in faster loading speeds, when used in an HTML design. Although we have implemented only these two automations, it is easy to automate many other features easily, as the basic ground work is already done by us.

**The Procedure**

Grunt and Grunt plugins are installed and managed via npm, the Node.js package manager. Therefore first I installed npm. Then I installed the Grunt-Comman-Line-Interface using npm. Then I configured the source folder with a *package.json* file. This file is used by npm to store metadata for projects published as npm modules.

After laying the foundation as mentioned above, I started writing the *GruntFile.js* file. This file is used to configure and define tasks and load Grunt plug-ins. For this project, I wrote codes for concatenating several files into one and *uglifying the* big file created. I automated both these tasks in the same GruntFile.js file.

With this project, I marked the closure of my Internship at WSO2.

# Annexes

Annex. 1: WSO 2 Products Overview

Annex.2: WSO2 support model overview



Annex. 3: A short listed list of WSO2 customers based on their popularity



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, google, html, BAM

Gridster, bootstrap, Ajax, Rest, jax-rs, api, web service, maven, war, wso2 server, web app, Jackson, JaxB, gson, carbon, middleware,SOA, jira, annotate, JSE JEE, deployment, wsdl,epr,enums,remote debugging,refactor,Vega,GruntJs, Task Runner, concatenate, Uglify,minify.