Dynamic Performance Framework

Iris

Status Report (Semester 2)

Dean Gaffney

20067423 Panel 3

Supervisor: Dr. Kieran Murphy

Second Reader: David Drohan

BSc (Hons) in Entertainment Systems

Contents

List of Tables

List of Figures

Abstract

The aim of this project is the design a full implementation of a system for application performance monitoring. The proposed system, has the working title, Iris.

On completion, Iris will provide users with a dynamic performance framework which will allow users to fully customise and centralise their application performance monitoring. This will be achieved through a web interface where a user can specify a schema for a specific application they wish to monitor. Once a schema has been set up, a REST endpoint will be generated for the application. This endpoint will allow a user to send their monitoring data from their desired application to the framework in the form of JSON (matching the specified schema). Iris will also contain features which allow a user to monitor and analyse incoming data, using an intelligent, fully customisable graph and dashboard builder. Iris will then visualise any received data in real time using to the appropriate dashboards using websockets. Iris will come with some out of the box scripts/applications that users can use to monitor typical tasks such as JVM (Java Virtual Machine) performance, Linux OS System Performance.

Introduction

2.1 Motivation for Iris

The motivation for this project comes from database and system performance issues that Onaware[1] has experienced in recent projects. It is often the case that they must deal with large amounts of identity data being aggregated into a third party system called 'IIQ'.

Onaware has faced major issues with aggregating data in the past, in some cases it was taking up to five days, and sometimes they would fail halfway through meaning aggregations would have to be restarted, due to the amount of software involved it is hard to pinpoint what software is causing the issue.

In one such instance of aggregating data issues several attempts were made to rectify the performance issue such as optimising sql queries, increasing ram, multi threading tasks and increasing disk space, none of which worked. Due to the performance issue the IIQ instance became unusable so debugging the issue was not possible from inside the application and log files became so big that text editors would crash when trying to open them. In this case the issue turned out to be a customer putting size constraints on the database storing the aggregated data. While monitoring would not prevent such a mistake it would have reduced the time needed to locate the issue.

In response to difficulties in identifying performance issues Onaware have tried to monitor specific application elements. The aim at the time was to try and combine SQL, JVM and Operating System scripts to track the performance of the tools, however this approach is not very scalable and it would need to be reconfigured for future projects.

Iris is an attempt to solve this problem. Iris will allow a user create a new application monitor with little effort using a web interface, give the user a REST endpoint specific to the application for their scripts to target their data, and allow a user to monitor the data in real time using graphs and dashboards. The aim is to make the framework as flexible as possible and not

---

[1]Onaware is an international company that specialises in IAM (Identity and Access Management) and has offices with 20 staff in Waterford. More information on Onaware can be found at https://onaware.com.

specific to the issue Onaware faced, meaning a user can monitor any data they want from any application they want all they must do is send their data to a REST endpoint.

Users of Iris will consist of Onaware developers who will be monitoring IAM project data and generic tools which may be released to clients at a later time.

2.2 Features List

2.2.1 Types of data

numerical, categorical and textural

Implementation

During semester 1 a number of implementation were designed and tested. As a result the implementation described in semester 1 required not further changes in thus semester. Hence, this section is fundamentally unchanged from that in semester 1 report Distinguish between subparts that were implemented (and so their overview here is similar to that in semester 1 report) and subparts that were implemented during this semester (and so the overview here is new). In this chapter the implementation of Iris is summarised. The components completed in semester were described in the Semester 1 report and that summary is reproduced here (with minor modification) for completeness. The subsections dealing with components implemented in semester 2 is fundamentally new. Table REF lists the components adn when they were completed.

Component	When prototyped	When completed
code framework/services	Semester 1	Semester 1
schemas		
dashboards		
aggregation builder		
data sources		

3.1 Core Framework/Services

In the Semester 1 report the design of Iris is described. Due to the testing of the designs in the first semester the final design needed no further modifications this semester. Hence the description of Iris given here follows closely that given in report 1.

3.1.1 Description

Iris will act as a web interface for a user to create an application monitor and allow the user to query and create personalised dashboards of their data through the use of Elasticsearch. A user may setup an application schema definition within Iris that matches the data they wish

to monitor, a schema will consist of field names and corresponding data types specific to the application. Using the schema Iris will know what data to expect from the user. Once a schema is in place, Iris will generate a unique endpoint associated with the schema, this unique endpoint will be given to the user as a means of sending data to Iris. Data sent to the schema endpoint will be in JSON format and will conform to the schema definition created by the user in Iris.

```
A user creates an application monitor for an SQL database, they may create a schema
like the following:
Schema Name: "SQL Monitor"
Schema Fields:
    -field name: "writeSpeed"
     fieldType: "double"
    -field name: "tableName"
     fieldType: "String"
Iris will then expect a json object to come back in the form:
{
        "writeSpeed": 3000,
        "tableName":  "students"
}
```

Figure 3.1 – Example schema created for Iris.

Iris will take the users data and create data mappings (Elastic.co 2017c) inside Elasticsearch, as well as insert any incoming data into the correct Elasticsearch index (Elastic.co 2017b). With a schema in place a user can route their data through Iris; turning Iris into a centralised area for monitoring application performance data. With Iris being the centralised location to route and view your data a user can write a data transformation script for incoming data. The advantage of this is that it can help reduce the need for applications being redeployed to view new data or to transform data.

The user releases their application, and it is downloaded by 500 people. This data is now being sent from 500 instances of this application. To make any change to this data the developer must add in their desired field and and redeploy the app, those 500 users would then need to download an update for the application in order for it to take effect. The original JSON object passing through Iris looks like the following:

```
{
        "firstName": "Dean",
        "lastName":  "Gaffney"
}
```

In this example let's say the developer prefers to have the data mapped to a field called "fullName" which is all lowercase, the developer wants to avoid having to redeploy the app for one single field, instead the developer goes to Iris and applies a script to the schema. By running the script on incoming data the developer has made their desired change in a central location with no redeploys. The data may look like this after the developer has applied the script to the data:

```
{
        "firstName": "Dean",
        "lastName":  "Gaffney",
        "fullName":  "Dean Gaffney"
}
```

Figure 3.2 – Iris transforming data.

To aid performance monitoring, Iris will allow users to create personalised dashboards where they can create charts from their data and place them in the dashboard. This allows each user to have their own set of visualised data relative to them. To help a user see their data and charts rapidly an Elasticsearch aggregation (Elastic.co 2017a) playground will be put into place in order to allow a user create charts and get results back immediately, this will help a user to plan their dashboard charts before setting them up. The playground will also allow a user to chain several aggregations together which will allow them to create complex queries without having to have any prior knowledge of how Elasticsearch works.

3.1.2   Schemas

Schemas in Iris have not changed from semester 1 in terms of their design, they are still used as a representation of a data source in Iris which is created by the user. However there were some improvements made on both the front and backend of Iris to make the creation and interaction of agents with schemas easier for the user. These improvements will be discussed in the following sections.

3.1.2.1   Frontend

This section will discuss the improvements made to the frontend of Iris in regards to the Schema section.

3.1.2.1.1   Schema Fields

Iris now displays the Schema Fields in a table fashion and presents the user with the name and data types of all the fields in the schema.

A screenshot of the schema fields belonging to the Node.js schema in Iris.

Figure 3.3 – Node.js schema fields in Iris.

3.1.2.1.2   Data Source Endpoint

Iris now displays the unique endpoint associated with a data source when you click on the schema in Iris. This allows a user to see what address their data source must use as well as test out the endpoint before committing to writing a data source that uses the endpoint.

A screenshot of the MySQL data source endpoint in Iris.

Figure 3.4 – MySQL data source endpoint in Iris.

3.1.2.1.3   Expected JSON

Iris now displays the expected JSON object from the data source as well as the data types that are associated with each key in the object. This aids users when they are writing code for data sources as they can refer back to Iris to see what format their JSON object needs to conform to.

```
Expected JSON

{
    "name": "YOUR STRING",

    "rank": "YOUR INTEGER",

    "price_usd": "YOUR DOUBLE",

    "price_eur": "YOUR DOUBLE",

    "percent_change_24h": "YOUR DOUBLE"

}
```

A screenshot of the expected JSON for the Raspberry Pi data source in Iris.

Figure 3.5 – Expected JSON for Raspberry Pi data source.

3.1.2.1.4   Transformation/Rule Script editor

Due to Iris' ability to allow users to write transformation scripts that run on incoming data which is discussed in paragraph 3.1.2.2.1, a code editor was embedded into Iris. The code editor being used is the 'Ace' code editor with Groovy syntax highlighting which aims to aid the user in writing small transformation scripts.


```
Schema Rule

1  json.osName = json.osName.toUpperCase()
2  json.memFree = json.memFree / 1024 / 1024 / 1024
3  return json
```

A screenshot of the Transformation Rule script on the Node.js schema written using the 'Ace' code editor.

Figure 3.6 – Node.js Transformation Rule script

More information on the 'Ace' code editor can be found here https://ace.c9.io/

3.1.2.1.5   Schema Overview

The following section ties together the previous front end sections and shows what an entire schema looks like inside Iris in a single image.
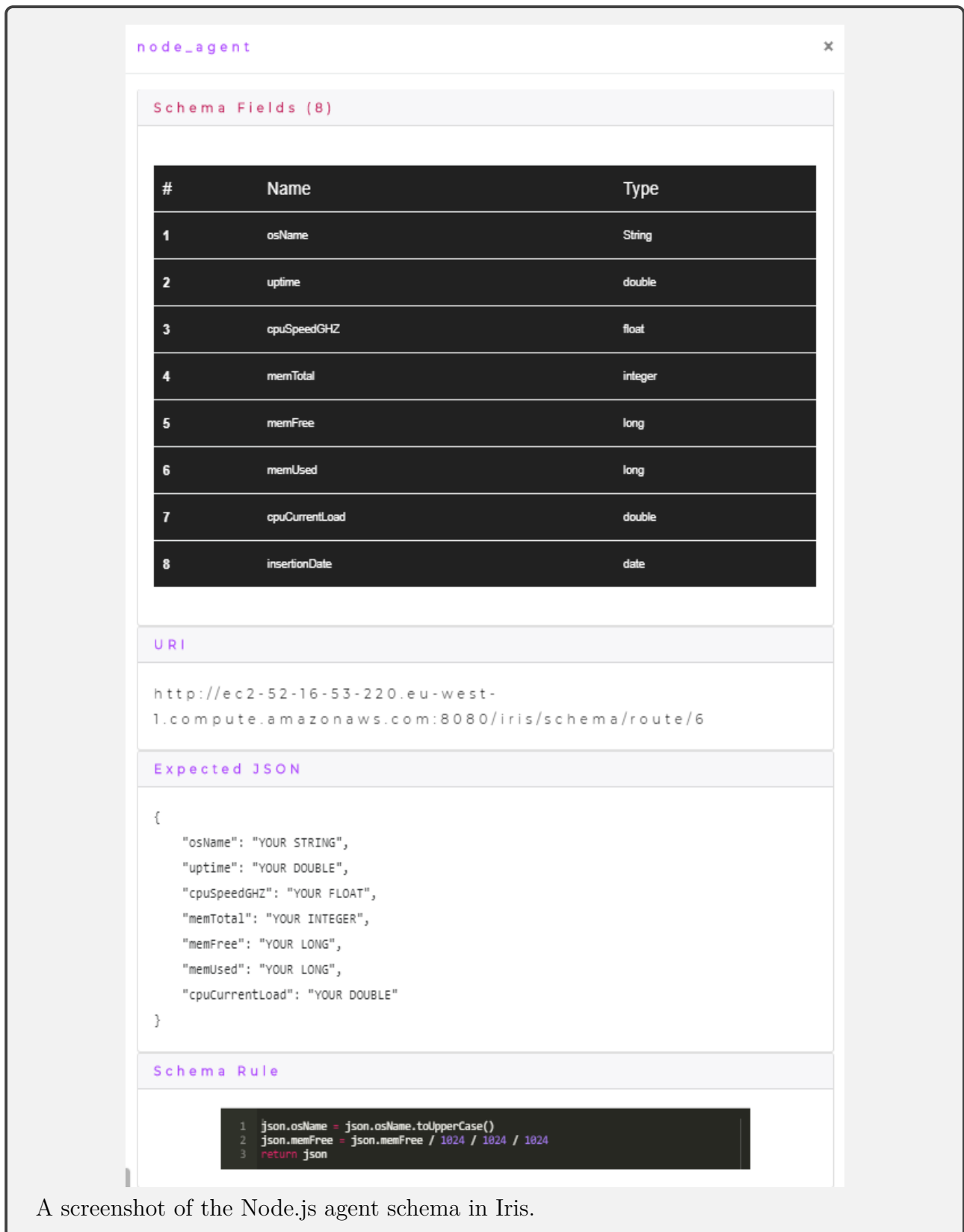
A screenshot of the Node.js agent schema in Iris.

Figure 3.7 – Node.js agent schema in Iris.

3.1.2.2   Backend

This section will discuss the improvements made to the backend of Iris in regards to the Schema section.

3.1.2.2.1 Transformation/Rule Script Execution

In the semester 1 report it was briefly discussed that Iris would allow users to write their own scripts inside Iris to allow a user to modify incoming data, which would avoid a user having to redeploy their data source to make a small change to the data they are pushing ot Iris. A user would mainly use this feature for formatting dates, strings and numerical values before it is inserted into Elasticsearch and rendered on the user's dashboard. This feature is now fully supported in Iris and an image of a transformation script can be found in fig. 3.6

3.1.2.2.2 Data Source Endpoint Retrieval

Iris now accepts 'POST' requests to the endpoint $SERVER_BASE/schema/getAgentUrl and expects a JSON body with the request like the following:

```
{
    "name": "someSchemaName"
}
```

Figure 3.8 – SchemaController getAgentUrl endpoint JSON payload.

Assuming the user is authenticated, Iris will look for a Schema matching this name and return the unique endpoint for this schema. The advantage of this is that a user may create a schema and not be satisfied with it. The user may then decide to delete the schema and create a new schema using the same name with extra attributes attached to it. The issue here is that Iris will give the new schema a new url different from the previous schema. This now means a user must go back to their agent code and change a hardcoded url to a new url and then redeploy the agent. However if a user writes their code to take advantage of the 'getAgentUrl' endpoint in Iris, they will be able to dynamically obtain a new unique endpoint for the new schema as long as the schema name stays the same. This means there is no need to redeploy the agent with a new url as it is obtained dynamically from Iris.
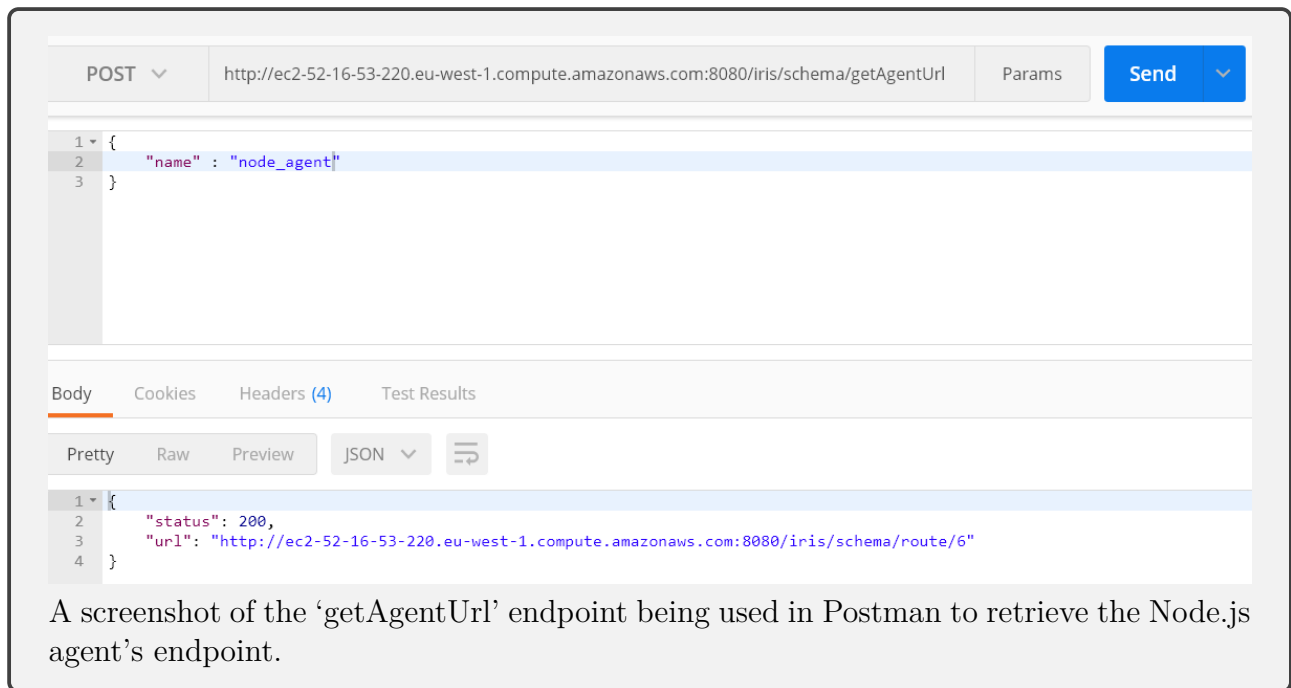
A screenshot of the 'getAgentUrl' endpoint being used in Postman to retrieve the Node.js agent's endpoint.

Figure 3.9 – SchemaController 'getAgentUrl' endpoint in Postman.

3.1.2.2.3   Timestamping Data

Iris now puts a timestamp on all incoming data before it is inserted into Elasticsearch. The timestamp is added as a schema field called 'insertionDate' under the schema to which the data belongs. This field is used in the background of the Aggregation Builder in Iris and will be discussed in that section.

3.1.3   Dashboards

In the semester 1 report the javascript libraries for building Iris' dashboard system were selected and tested on a very small scale. The design for implementing the dashboard and updating the charts in realtime have remained the same throughout the implementation this semester. The dashboard system features and implementation will be discussed in terms of frontend and backend logic in the following sections. A brief overview of how the data flows into the dashboard will also be discussed.

3.1.3.1   Flow of data

3.1.3.2   Frontend

This section will discuss the major factors of the frontend of the Iris dashboard system.

3.1.3.2.1   Gridstack.js

Gridstack.js was briefly discussed in the semester 1 report as being the library that was selected to aid in the dashboard creation. In semester 1 the library was and it's api was researched to ensure it supported the features Iris required.

Due to the extensive research on the library in semester 1 and the active github support from the

Gridstack.js developers there was no major issues during the implementation phase in semester 2. Gridstack.js was used to contain the contents of a dashboard in a grid format. Whenever a chart is created for a dashboard a new widget is created containing the chart and added to the Gridstack grid. This widget automatically becomes resizeable, draggable and serializeable.

One main issue I found with the Gridstack library is that if a chart contains no data it does not scale the chart correctly and makes the chart overflow to one side making it appear compact. Unfortunately this issue is not consistent and as a result was not fixed. Currently the only fix is to click on the resizeable handles of a widget and the chart then scales correctly.

3.1.3.2.2   Billboard.js

3.1.3.2.3   Vue.js

3.1.3.2.4   Chart Types

3.1.3.2.5   Downloadable Chart Data

3.1.3.2.6   Revision History

3.1.3.3   Backend

3.1.3.3.1   Chart Types

3.1.3.3.2   Serilization

3.1.3.3.3   Revision History

3.1.3.3.4   Scalabitility

By end of semester 1 the javascript libraries that will be used were selected but only basic testing of features was carried out. In this semester the dashboards were designed and implemented.

What libraries are used ?

use table, two columns, library name, purpose

Major issues during development include: ans status

- sdf

-

3.1.4   Aggregation Builder

3.1.4.1   Aggregation Types

3.1.4.2   Aggregation Builder Preview

3.1.5   Security

3.1.5.1   Spring Security Core

3.1.5.2   Spring Security REST Plugin

3.1.5.3   Testing Security

3.1.5.4   Data Source Adaptation

3.2   Data Sources

A DataSource can be any process that generates data and pushes it to Iris. and number of examples of data sources are discussed. THe Implementation details of these data sources is not covered in detail because they are not germane to the this project and follow well know patterns.

Deployment, Testing and Evaluation

4.1 Deployment

4.1.1 AWS

4.1.2 Jetty

4.1.3 Gradle

4.1.4 WAR

4.1.5 MySQL GORM Mapper

4.2 Testing and Evaluation — Data Sources

4.2.1 Introduction

A number of data source have been implemented to demonstrate the flexibility of Iris and to test the aggregation. Each of the following sections describe a data source and its unique features.

4.2.2 Android App

4.2.2.1 Description

The android application demonstrates how Iris handles state based data. The application is very simple, but demonstrates how simple it is to monitor an application's state through Iris. The application is simply a screen consisting of six tiles. Each tile represents a different state, each state has three colours and three numerical values linked to the states and colours. The

android application allows the user to change the states of these tiles, which then changes the state of the tile colour and value. When a user is satisfied with the states they wish to send to Iris they simply tap the screen. This results in a JSON object being sent to Iris consisting of the current numerical value for each state tile i.e the current state of the tiles.
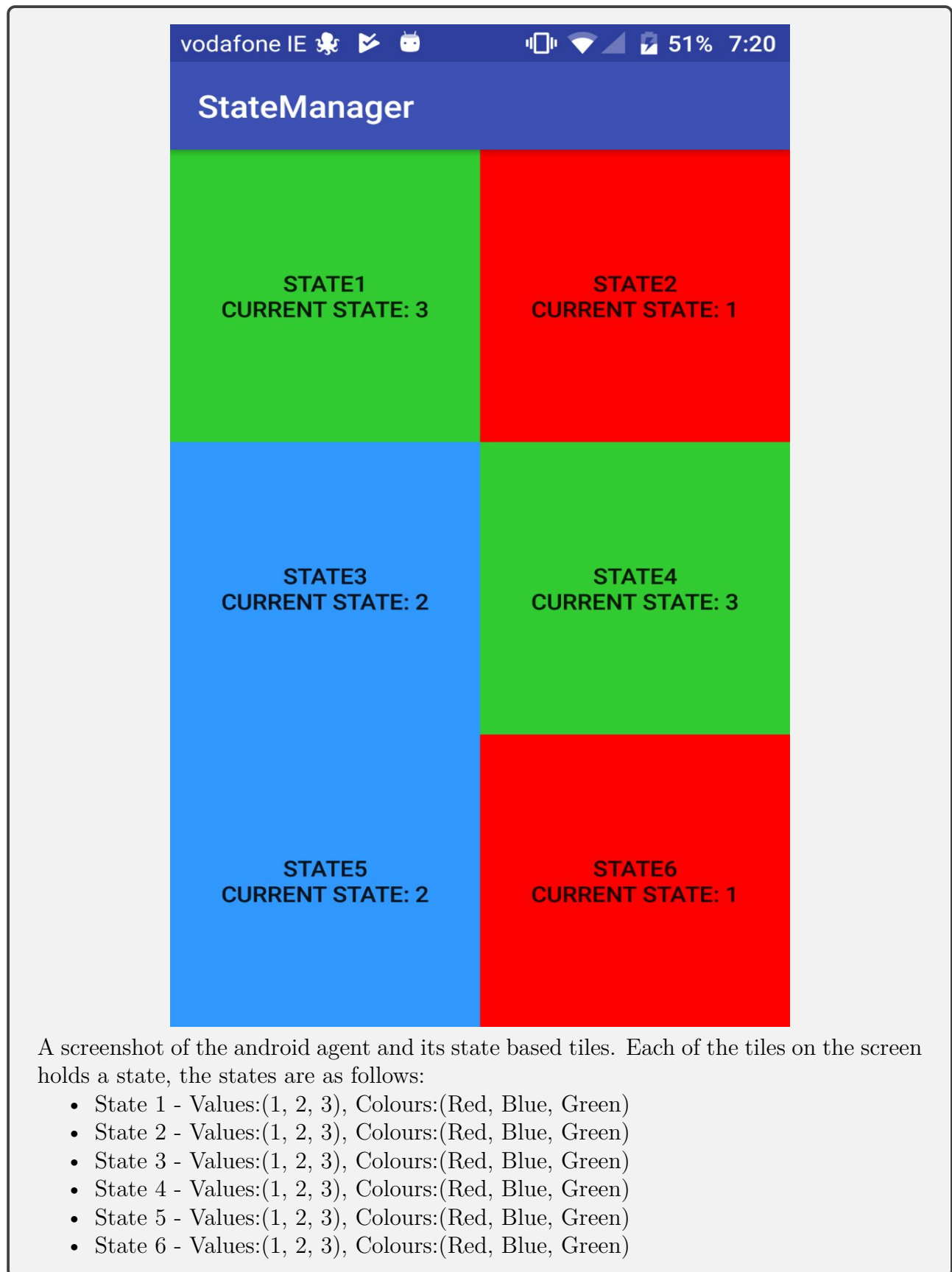
A screenshot of the android agent and its state based tiles. Each of the tiles on the screen holds a state, the states are as follows:

- State 1 - Values:(1, 2, 3), Colours:(Red, Blue, Green)
- State 2 - Values:(1, 2, 3), Colours:(Red, Blue, Green)
- State 3 - Values:(1, 2, 3), Colours:(Red, Blue, Green)
- State 4 - Values:(1, 2, 3), Colours:(Red, Blue, Green)
- State 5 - Values:(1, 2, 3), Colours:(Red, Blue, Green)
- State 6 - Values:(1, 2, 3), Colours:(Red, Blue, Green)

Figure 4.1 – Android agent for Iris.

4.2.2.2   Linkage with Iris

The android application is linked to Iris through a unique REST endpoint specific to the android agent schema. All of the data being sent from the android application is sent to this endpoint. Once the data enters Iris, Iris will run through it's logic for checking for dashboards and charts associated with the schema and send the data through to the charts. In this case the charts are state based and will update according to the state values that are given to them.

A screenshot of the Android agent schema in Iris.

Figure 4.2 – Android agent schema in Iris.

A screenshot of the android agent dashboard in Iris and its state based tiles.

Figure 4.3 – Android agent dashboard in Iris.

4.2.3 Raspberry Pi

The Raspberry Pi agent is used to demonstrate the plug and play ability of a Raspberry Pi combined with the versatility of Iris.

4.2.3.1 Description

The Raspberry Pi agent demonstrates how a user can use a Raspberry Pi to run a script to continuously monitor an API. In the case the Raspberry Pi monitors crypto currency exchange rates using a python script through a library called 'coinmarketcap'. The python script simply retrieves the latest crypto currency data through the 'coinmarketcap' library and sends the data to the corresponding 'crypto_agent' endpoint in Iris. The Raspberry Pi agent is configured to run the script every minute with crontab and is also configured to automatically sign in to the terminal. This results in a plug and play agent wherever their is a network cable available.

For more information on coinmarketcap see their site here https://coinmarketcap.com/

4.2.3.2 Linkage with Iris

The Raspberry Pi agent is linked to Iris through a unique REST endpoint specific to the Raspberry Pi agent schema. All of the data being sent from the Raspberry Pi is sent to this endpoint. Once the data enters Iris, Iris will run through it's logic for checking for dashboards and charts associated with the schema and send the data through to the charts. In this case Iris is monitoring raw data on two charts that are monitoring the current price in euro and usd. The other charts are using Elasticsearch aggregations to monitor the min and max prices reached for currencies in both euro and usd.

A screenshot of the Raspberry Pi agent schema in Iris.

Figure 4.4 – Raspberry Pi agent schema in Iris.

A screenshot of the Raspberry Pi agent dashboard in Iris monitoring crypto currencies.
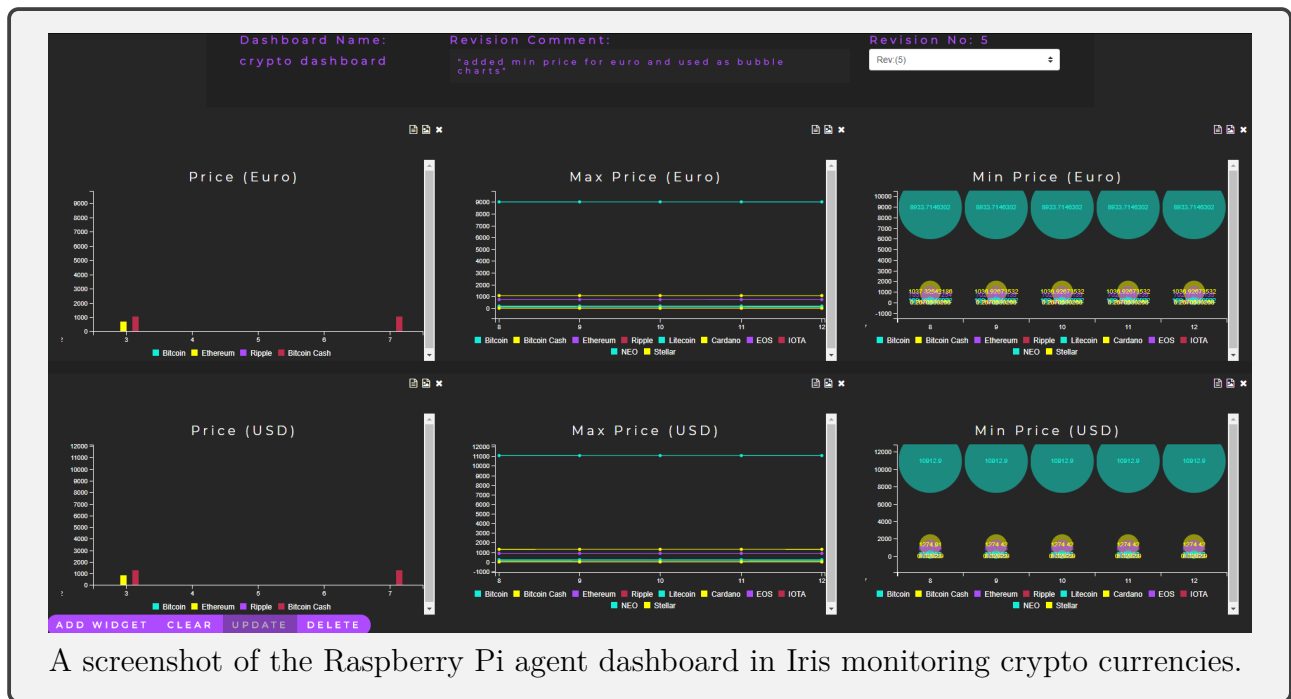
Figure 4.5 – Raspberry Pi agent dashboard in Iris.

4.2.4   Node.js

The Node.js agent shows how Iris can integrate with one of the most popular server-side frameworks and how it can monitor the status of the system running Node.js through the npm (node package manager) package 'systeminformation'.

4.2.4.1   Description

Due to Node.js' rise in popularity in recent years this agent demonstrates how Iris can easily monitor a system using the 'systeminformation' package from npm. The 'systeminformation' package offers various levels of detail about the system running Node. Iris' schema for the Node.js agent measures the following attributes:

- Operating System Name
- Current Uptime of the Operating System
- Current CPU Speed
- Total Memory Available
- Total Memory Free
- Total Memory Used
- Current CPU Load

A full list of all the obtainable attributes through 'systeminformation' can be found here https://github.com/sebhildebrandt/systeminformation.

An important thing to note about this agent is that it takes use of Iris' ability to transform incoming data through scripting. Many of the memory attributes retrieved from 'systeminformation' are represented in bytes, this agent uses Iris' centralised transformation abilities to

transform the 'memFree' attribute to be represented as gigabytes rather than bytes. This script also transforms all the 'osName' attributes to be uppercase. Transforming these attributes in Iris through scripting removed the need for a redeployment of the Node.js application.
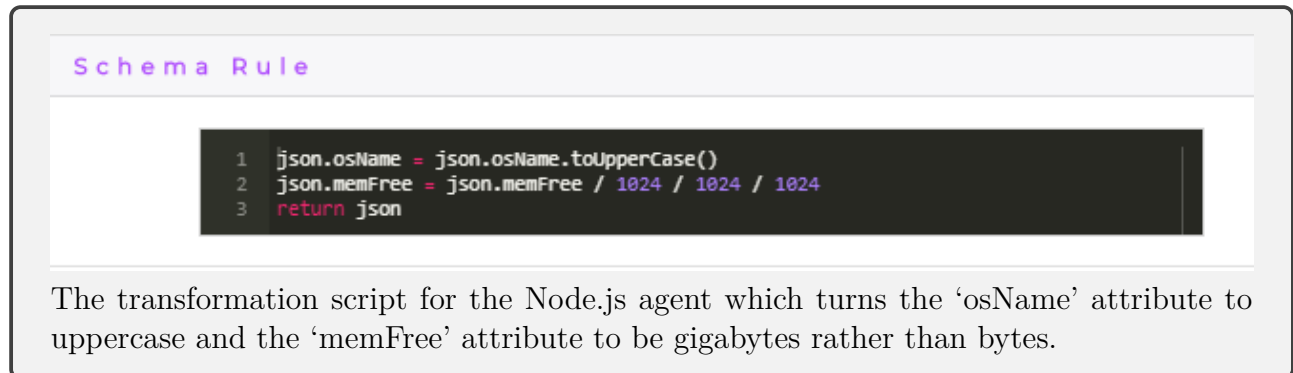

Schema Rule

```
1  json.osName = json.osName.toUpperCase()
2  json.memFree = json.memFree / 1024 / 1024 / 1024
3  return json
```

The transformation script for the Node.js agent which turns the 'osName' attribute to uppercase and the 'memFree' attribute to be gigabytes rather than bytes.

Figure 4.6 – Node.js agent transformation script.

4.2.4.2   Linkage with Iris

The Node.js agent is linked to Iris through a unique REST endpoint specific to the Node.js agent schema. All of the data being sent from the Node.js agent is sent to this endpoint. Once the data enters Iris, Iris will run through it's logic for checking for dashboards and charts associated with the schema and send the data through to the charts. In this case Iris is monitoring raw data on three charts that are monitoring the current memory free in gigabytes, the current memory used and the current uptime of the system. The other charts display the max memory used over all operating systems and the pie chart shows all the types of operating systems which have run the agent, both of these use Elasticsearch aggregations to calculate the data for the charts.

A screenshot of the Node.js agent schema in Iris.

Figure 4.7 – Node.js agent schema in Iris.

A screenshot of the Node.js agent dashboard in Iris monitoring system information.
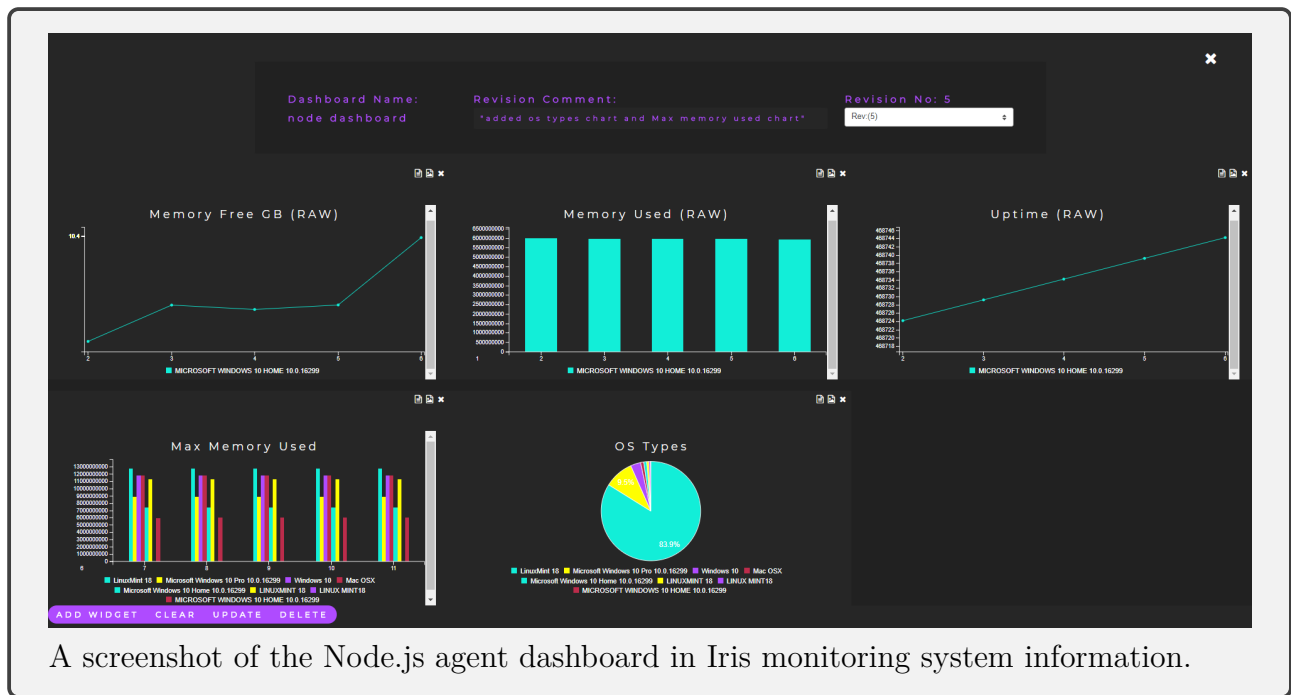
Figure 4.8 – Node.js agent dashboard in Iris.

4.2.5 Selenium

The Selenium agent demonstrates the versatility of Iris by monitoring the price of a guitar by scraping music store websites and monitoring the price of the guitar. This agent focuses on versatility over complexity.

4.2.5.1 Description

The Selenium agent demonstrates Iris' versatility through the Selenium web driver which is commonly used for web automation and functional testing. In this case Selenium is used in conjunction with a Selenium wrapper called 'Selenide' to automate the scraping of music store websites for a specific guitar. On each site the guitar's price is taken from the webpage as well as the name of the music store, the price of the guitar and the name of the music store is then sent to Iris so that it may be monitored.

For more information on Selenide see their site here http://selenide.org/

4.2.5.2 Linkage with Iris

The Selenium agent is linked to Iris through a unique REST endpoint specific to the Selenium agent schema. All of the data being sent from the Selenium agent is sent to this endpoint. Once the data enters Iris, Iris will run through it's logic for checking for dashboards and charts associated with the schema and send the data through to the charts. In this case Iris is monitoring raw data on a bar chart. The bar chart monitors the price of the guitar in euro and labels the chart based on the music store website that the price has been scraped from. In this case two sites have been scraped 'Thomann' and 'MusicStore.de', both sites are competitors and this reflects in the chart as both sites have matched the price of the guitar at the same price.

Figure 4.9 – Selenium agent schema in Iris.
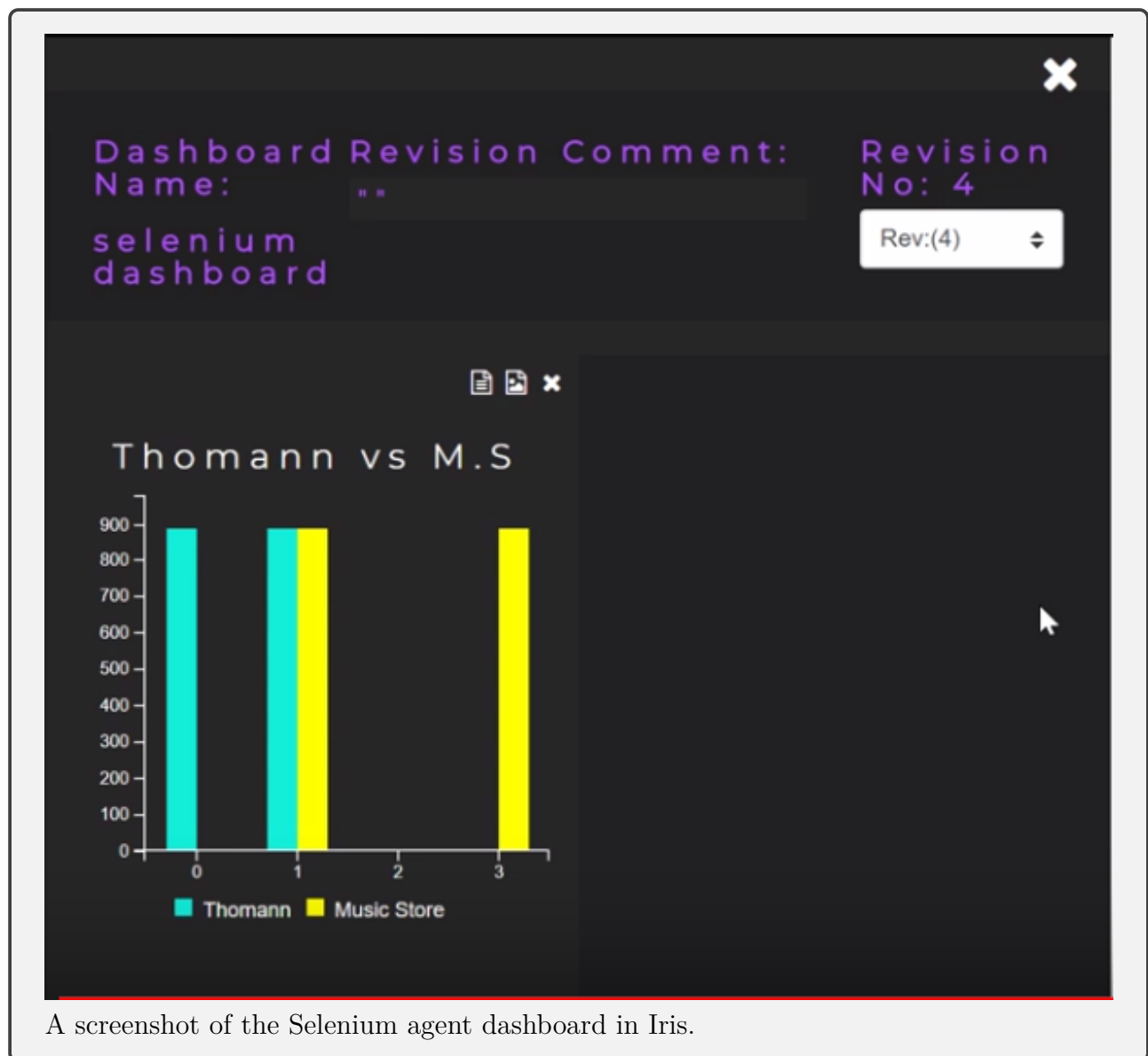
A screenshot of the Selenium agent dashboard in Iris.

Figure 4.10 – Selenium agent dashboard in Iris.

4.2.6   MySQL

The MySQL agent demonstrates how a user can monitor their database by converting the MySQL 'information_schema' results into JSON and sending them to Iris.

4.2.6.1   Description

The MySQL agent is significant for Iris as this agent was the inspiration for Iris in the beginning, as Onaware developers wanted a way to monitor a remote MySQL database with scripts and be able to monitor the database memory locally with a web application. The MySQL agent is a simple python script that connects to the MySQL instance and monitors all the databases as well as the amount of memory they use. The database name and the amount of memory it currently uses is then sent to Iris to be monitored. The python script is run every day at 8:00pm on the same AWS server that Iris is running on using a cron job.

```
SELECT table_schema AS "databaseName",
ROUND(SUM(data_length + index_length) / 1024 /  1024, 2) as "memoryMB"
FROM information_schema.TABLES
GROUP BY table_schema;
```

Figure 4.11 – MySQL Agent, MySQL 'information_schema' table query used in iris-mysql.py source code.

More information on the MySQL 'information_schema' table can be found here https://dev.mysql.com/doc/refman/5.7/en/information-schema.html

4.2.6.2   Linkage with Iris

The MySQL agent is linked to Iris through a unique REST endpoint specific to the MySQL agent schema. All of the data being sent from the MySQL agent is sent to this endpoint. Once the data enters Iris, Iris will run through it's logic for checking for dashboards and charts associated with the schema and send the data through to the charts. The MySQL agent dashboard monitors the average memory taken up by each database in the MySQL instance and displays the results as a line chart in Iris.

A screenshot of the MySQL agent schema in Iris.

Figure 4.12 – MySQL agent schema in Iris.

A screenshot of the MySQL agent dashboard in Iris.
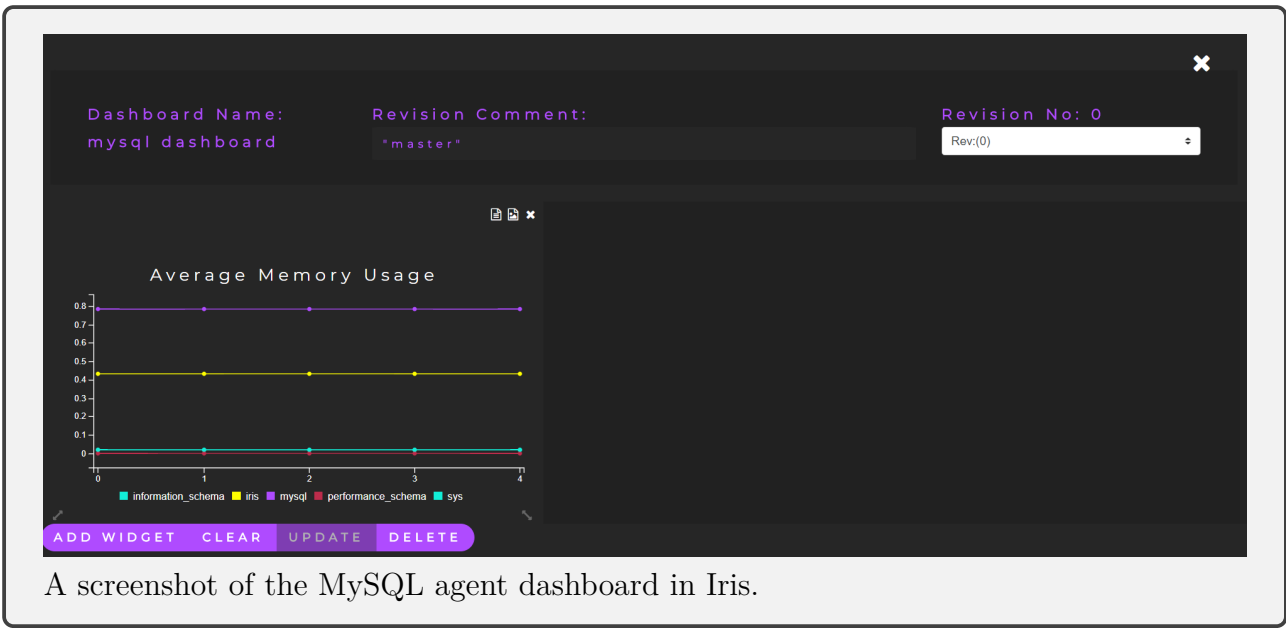
Figure 4.13 – MySQL agent dashboard in Iris.

Conclusions

Docs.spring.io (2017). *STOMP Support*. URL: https://docs.spring.io/spring-integration/reference/html/stomp.html (visited on 11/14/2017).

Elastic.co (2017a). *Aggregations*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations.html (visited on 11/04/2017) (cit. on p. 6).

– (2017b). *Basic Concepts*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html (visited on 11/04/2017) (cit. on p. 5).

– (2017c). *Mapping*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html (visited on 11/04/2017) (cit. on p. 5).

Logz.io (2017). *Grafana vs. Kibana: The Key Differences to Know*. URL: https://logz.io/blog/grafana-vs-kibana/ (visited on 11/16/2017).

Niederwieser, Peter T. (2017). *Introduction*. URL: http://spockframework.org/spock/docs/1.1/introduction.html (visited on 11/11/2017).

Appendices