

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

1	Abstract	1
2	Introduction	2
2.1	Motivation for Iris	2
2.2	Features List	3
2.2.1	Types of data	3
3	Implementation	4
3.1	Core Framework/Services	4
3.2	Schemas	4
3.3	Dashboards	4
3.4	Aggregation Builder	4
3.5	Data Sources	4
4	Deployment, Testing and Evaluation	5
4.1	Deployment	5
4.1.1	AWS	5
4.1.2	Jetty	5
4.1.3	Gradle	5
4.1.4	WAR	5
4.1.5	MySQL GORM Mapper	5
4.2	Testing and Evaluation - Data Sources	5
4.2.1	Introduction	5
4.2.2	Android App	5
4.2.2.1	Description	5
4.2.2.2	Linkage with Iris	6
4.2.3	Raspberry Pi	8
4.2.3.1	Description	8
4.2.3.2	Linkage with Iris	9
4.2.4	Node.js	11
4.2.4.1	Description	11
4.2.4.2	Linkage with Iris	12
4.2.5	Selenium	14

	4.2.5.1	Description	14
	4.2.5.2	Linkage with Iris	14
4.2.6	MySQL	16
	4.2.6.1	Description	16
	4.2.6.2	Linkage with Iris	17
5	Conclusions		20
	Appendices		22

List of Tables

List of Figures

4.1	Android agent for Iris	6
4.2	Android agent schema in Iris	7
4.3	Android agent dashboard in Iris	8
4.4	Raspberry Pi Agent, iris-crypto-rates.py source code	9
4.5	Raspberry Pi agent schema in Iris	10
4.6	Raspberry Pi agent dashboard in Iris	11
4.7	Node.js agent transformation script	12
4.8	Node.js agent schema in Iris	13
4.9	Node.js agent dashboard in Iris	14
4.10	Selenium agent schema in Iris	15
4.11	Selenium agent dashboard in Iris	16
4.12	MySQL Agent, MySql ‘performance_schema’ table query used in iris-mysql.py source code	17
4.13	MySQL agent schema in Iris	18
4.14	MySQL agent dashboard in Iris	19

CHAPTER 1

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.

2.1 Motivation for Iris

The motivation for this project comes from database and system performance issues that Onaware¹ 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

¹Onaware 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

CHAPTER 3

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 sesmter,. 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 impleneted in semester 2 is fundmental 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

3.2 Schemas

3.3 Dashboards

3.4 Aggregation Builder

3.5 Data Sources

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.

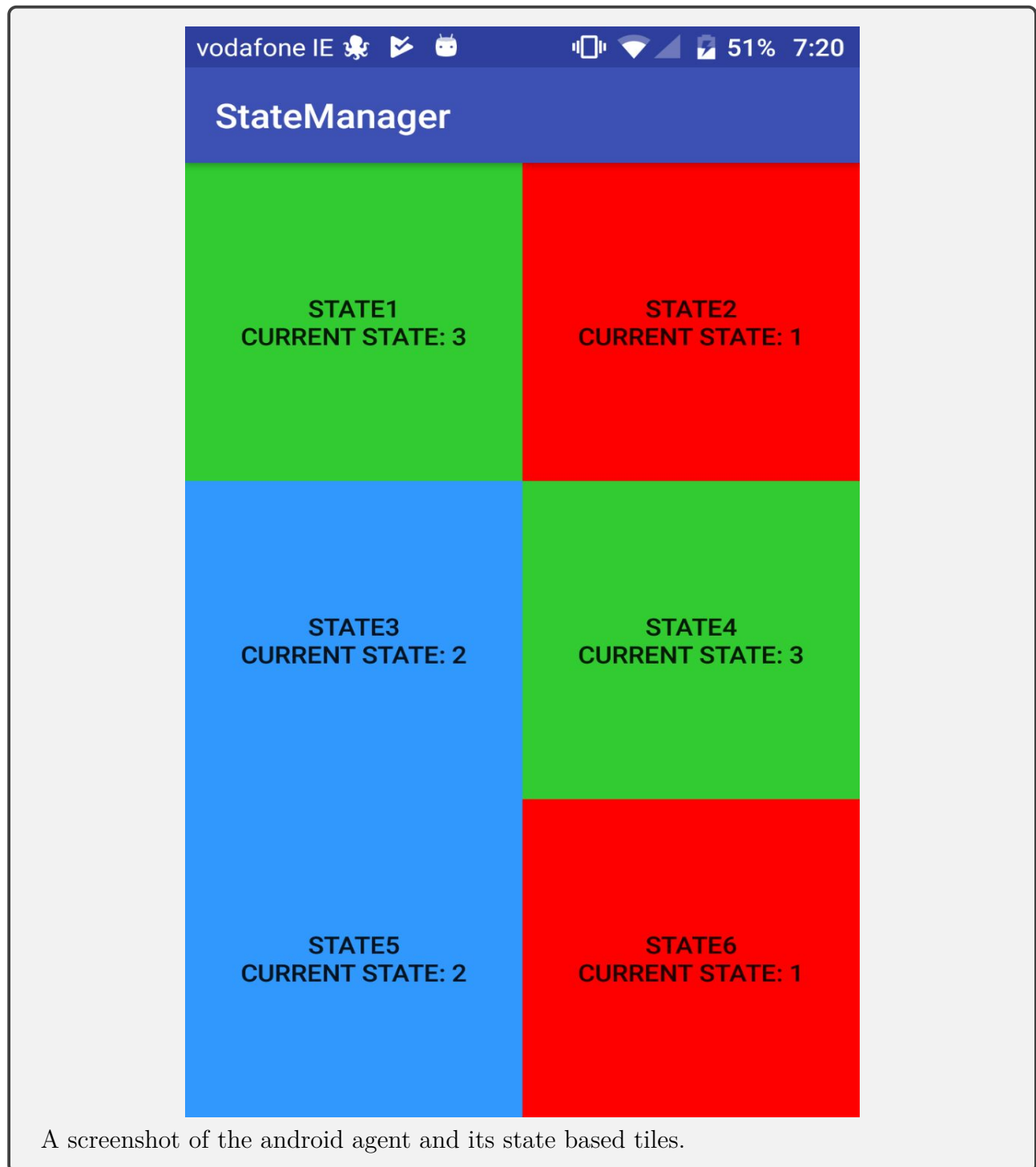


Figure 4.1 – Android agent for Iris

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.



The screenshot displays the configuration for an Android agent schema within the Iris interface. The window is titled "android_agent" and contains several sections:

- Schema Fields (7):** A table listing the fields of the schema.
- URI:** The endpoint for the schema.
- Expected JSON:** The structure of the data received from the agent.
- Schema Rule:** A Groovy script for processing incoming data.

#	Name	Type
1	State1	integer
2	State2	integer
3	State3	integer
4	State4	integer
5	State5	integer
6	State6	integer
7	insertionDate	date

```
http://ec2-52-16-53-220.eu-west-1.compute.amazonaws.com:8080/iris/schema/route/8
```

```
{
  "State1": "YOUR INTEGER",
  "State2": "YOUR INTEGER",
  "State3": "YOUR INTEGER",
  "State4": "YOUR INTEGER",
  "State5": "YOUR INTEGER",
  "State6": "YOUR INTEGER"
}
```

```
1 /*This Groovy script will run every time new data enters associated with this
   schema. You have access to the raw json through a Map object called 'json'*/
```

A screenshot of the Android agent schema in Iris

Figure 4.2 – Android agent schema in Iris



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 there is a network cable available.

```
#!/usr/bin/python

import requests, json
from coinmarketcap import Market

headers = {'Content-type': 'application/json', 'Accept': 'text/plain'}

agentData = {'name': 'crypto_agent'}

urlResp = requests.post('http://ec2-52-16-53-220.eu-west-1.compute.'+
'amazonaws.com:8080/iris/schema/getAgentUrl',
data=json.dumps(agentData), headers=headers)

endpoint = urlResp.json()['url']

wanted_keys = ['price_usd', 'price_eur', 'name',
               'percent_change_24h', 'rank']

def extract(data):
    return dict((k, data[k]) for k in wanted_keys if k in data)

coinmarketcap = Market()

crypto_currencies = coinmarketcap.ticker(limit=4, convert='EUR')

filterd_currencies = list(map(extract, crypto_currencies))

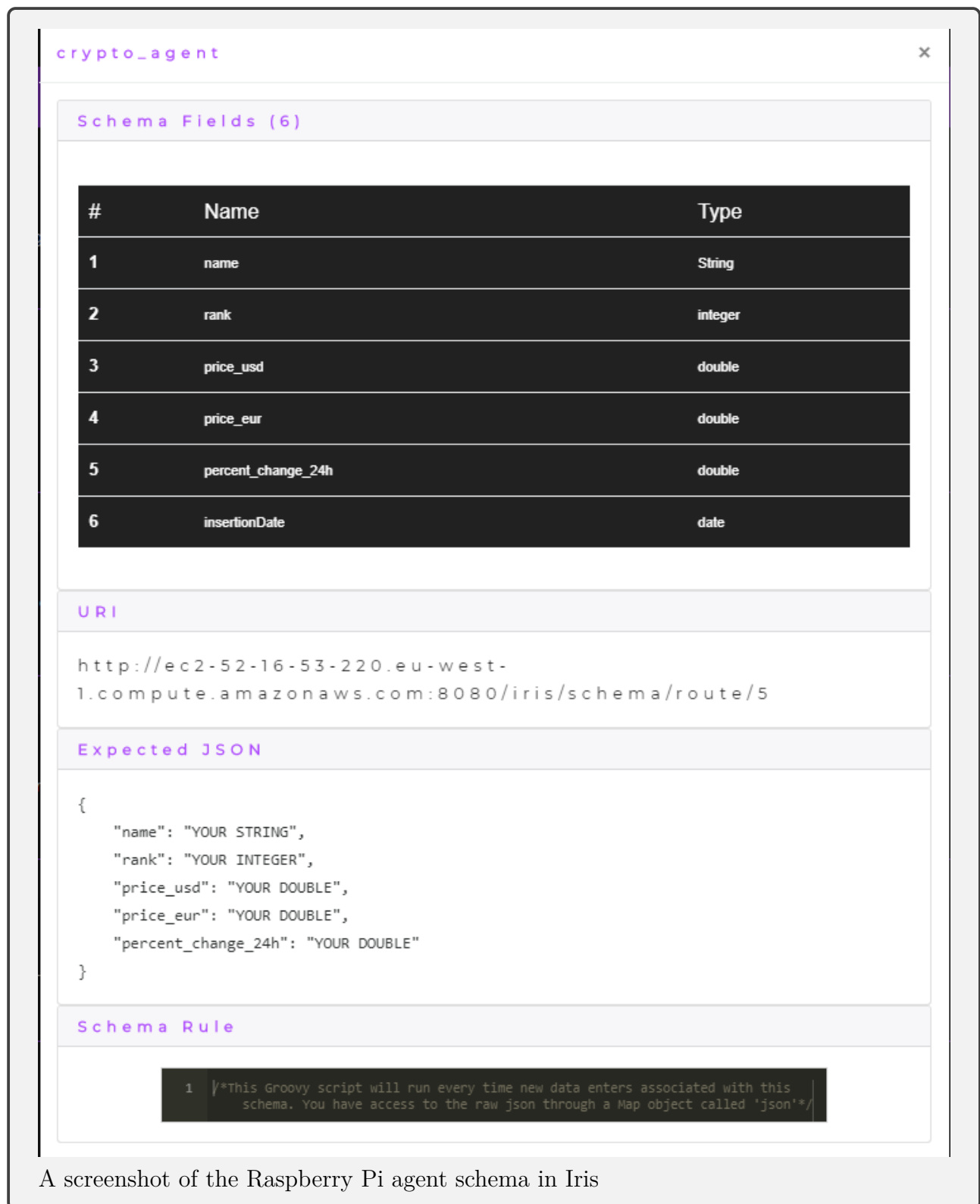
for currency in filterd_currencies:
    resp = requests.post(endpoint,
                        data=json.dumps(currency),
                        headers=headers)
    print resp.json()
```

Figure 4.4 – Raspberry Pi Agent, iris-crypto-rates.py source code

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.



The screenshot shows the 'crypto_agent' configuration window in the Iris interface. It displays the schema fields, URI, expected JSON, and a schema rule.

Schema Fields (6)

#	Name	Type
1	name	String
2	rank	integer
3	price_usd	double
4	price_eur	double
5	percent_change_24h	double
6	insertionDate	date

URI

```
http://ec2-52-16-53-220.eu-west-1.compute.amazonaws.com:8080/iris/schema/route/5
```

Expected JSON

```
{
  "name": "YOUR STRING",
  "rank": "YOUR INTEGER",
  "price_usd": "YOUR DOUBLE",
  "price_eur": "YOUR DOUBLE",
  "percent_change_24h": "YOUR DOUBLE"
}
```

Schema Rule

```
1 /*This Groovy script will run every time new data enters associated with this
   schema. You have access to the raw json through a Map object called 'json'*/
```

A screenshot of the Raspberry Pi agent schema in Iris

Figure 4.5 – Raspberry Pi agent schema in Iris

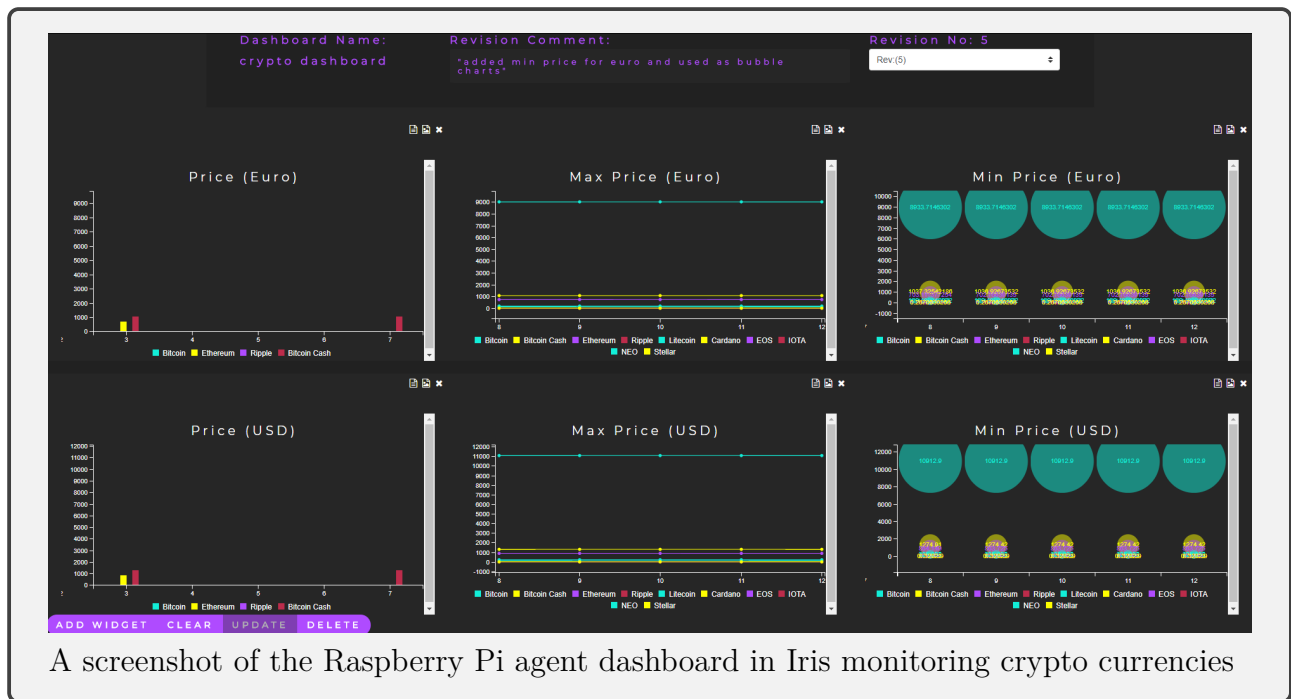


Figure 4.6 – Raspberry Pi agent dashboard in Iris

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.

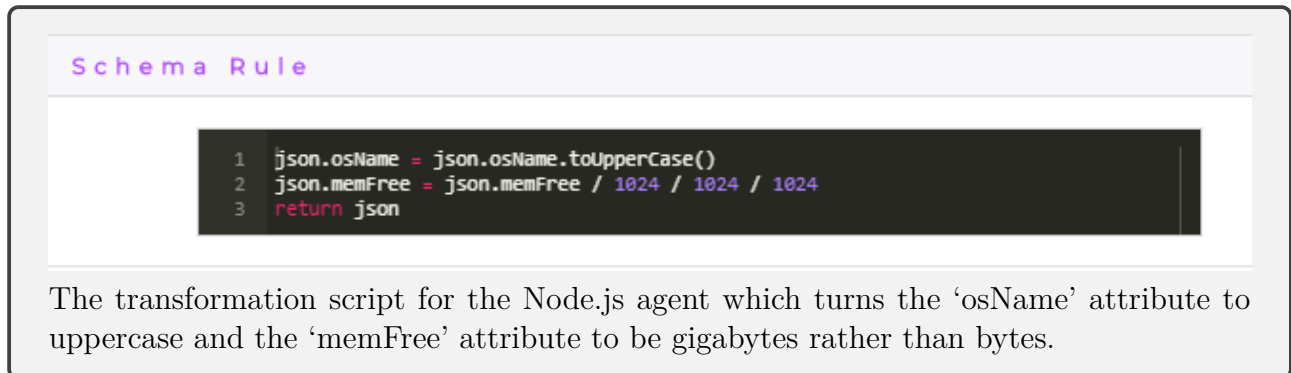


Figure 4.7 – Node.js agent transformation script

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.

node_agent

Schema Fields (8)

#	Name	Type
1	osName	String
2	uptime	double
3	cpuSpeedGHZ	float
4	memTotal	integer
5	memFree	long
6	memUsed	long
7	cpuCurrentLoad	double
8	insertionDate	date

URI

http://ec2-52-16-53-220.eu-west-1.compute.amazonaws.com:8080/iris/schema/route/6

Expected JSON

```
{
  "osName": "YOUR STRING",
  "uptime": "YOUR DOUBLE",
  "cpuSpeedGHZ": "YOUR FLOAT",
  "memTotal": "YOUR INTEGER",
  "memFree": "YOUR LONG",
  "memUsed": "YOUR LONG",
  "cpuCurrentLoad": "YOUR DOUBLE"
}
```

Schema Rule

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

A screenshot of the Node.js agent schema in Iris

Figure 4.8 – Node.js agent schema in Iris

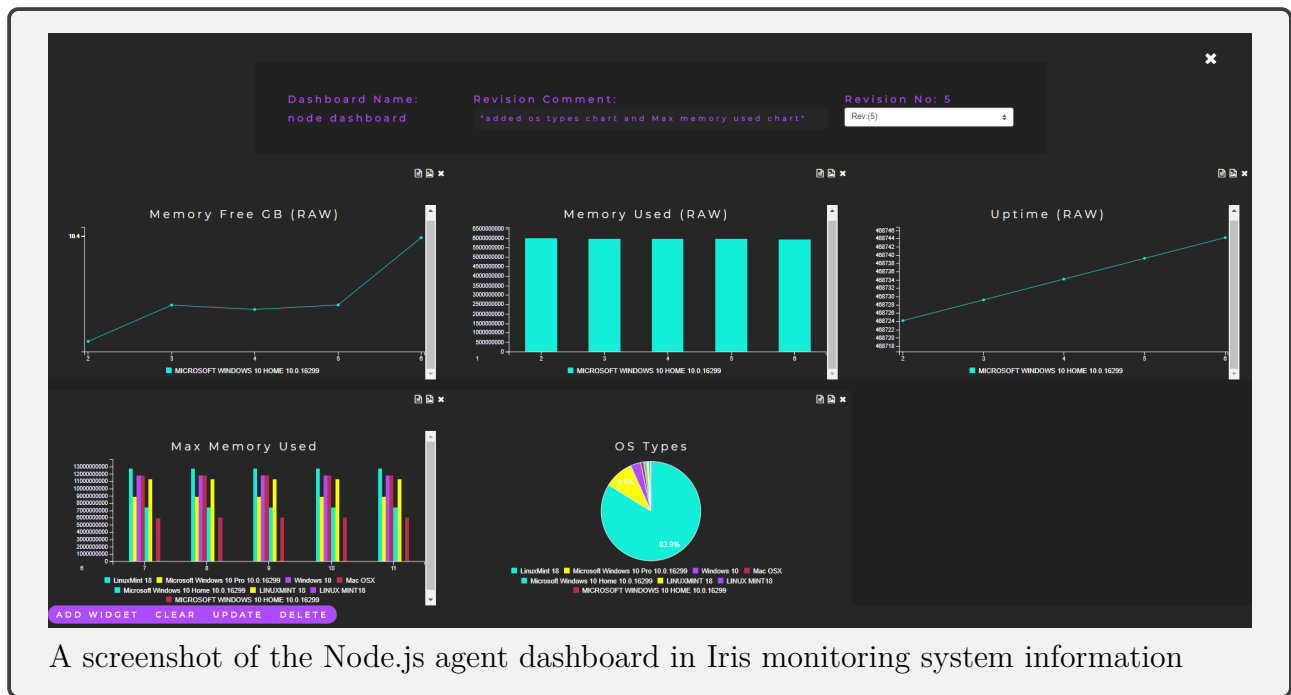


Figure 4.9 – Node.js agent dashboard in Iris

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.

selenium_agent

Schema Fields (3)

#	Name	Type
1	site	String
2	price	double
3	insertionDate	date

URI

http://ec2-52-16-53-220.eu-west-1.compute.amazonaws.com:8080/iris/schema/route/7

Expected JSON

{
 "site": "YOUR STRING",
 "price": "YOUR DOUBLE"
}

Schema Rule

1

/*This Groovy script will run every time new data enters associated with this schema. You have access to the raw json through a Map object called 'json'*/

A screenshot of the Selenium agent schema in Iris

Figure 4.10 – Selenium agent schema in Iris

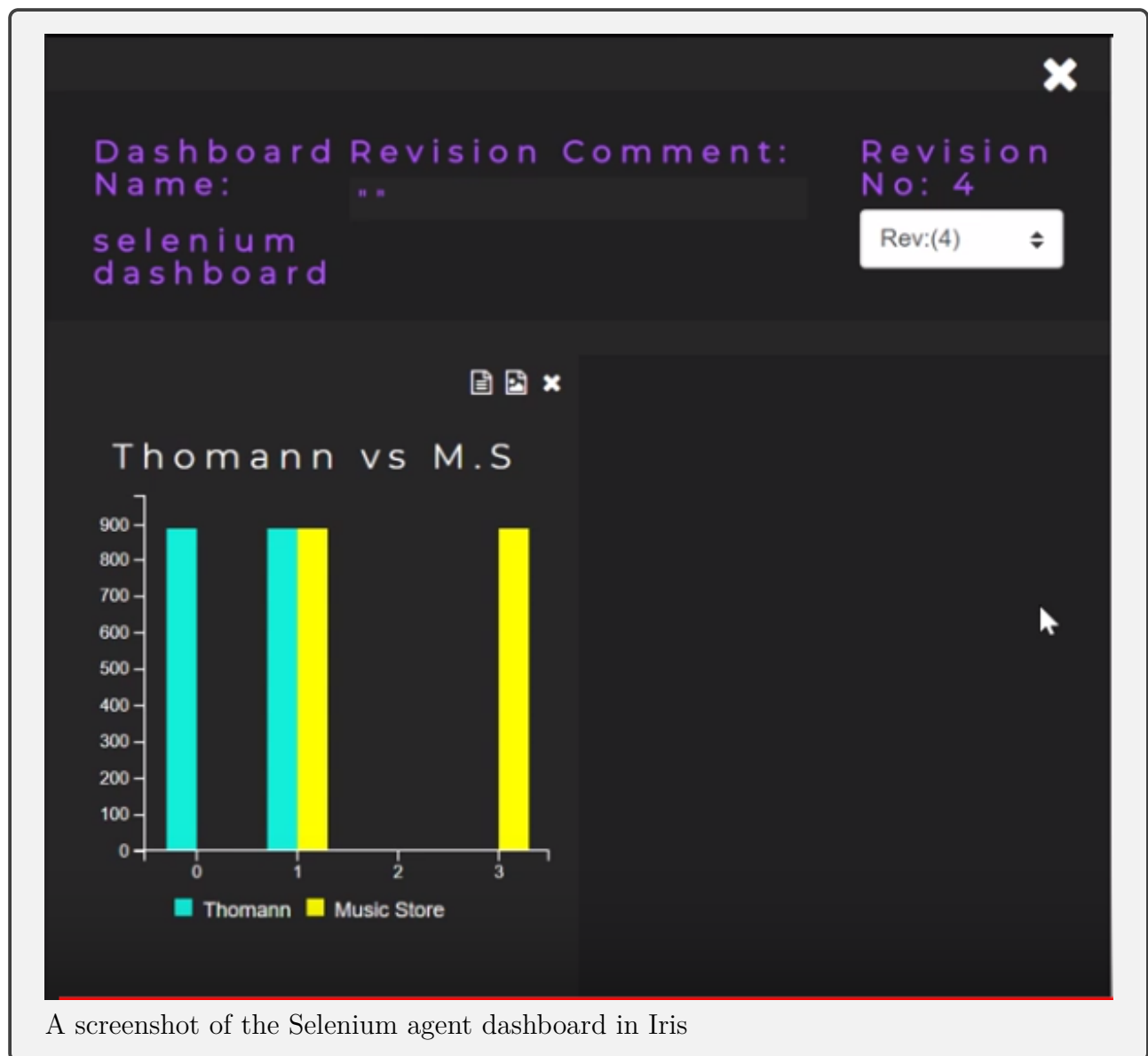


Figure 4.11 – Selenium agent dashboard in Iris

4.2.6 MySQL

The MySQL agent demonstrates how a user can monitor their database by converting the MySQL 'performance_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 4.12 – MySQL Agent, MySql ‘performance_schema’ table query used in iris-mysql.py source code

More information on the MySQL ‘performance_schema’ table can be found here <https://dev.mysql.com/doc/refman/5.6/en/performance-schema-quick-start.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.

mysql_agent

Schema Fields (3)

#	Name	Type
1	databaseName	String
2	memoryMB	double
3	insertionDate	date

URI

http://ec2-52-16-53-220.eu-west-1.compute.amazonaws.com:8080/iris/schema/route/4

Expected JSON

```
{
  "databaseName": "YOUR STRING",
  "memoryMB": "YOUR DOUBLE"
}
```

Schema Rule

```
1 /*This Groovy script will run every time new data enters associated with this
   schema. You have access to the raw json through a Map object called 'json'*/
```

Figure 4.13 – MySQL agent schema in Iris

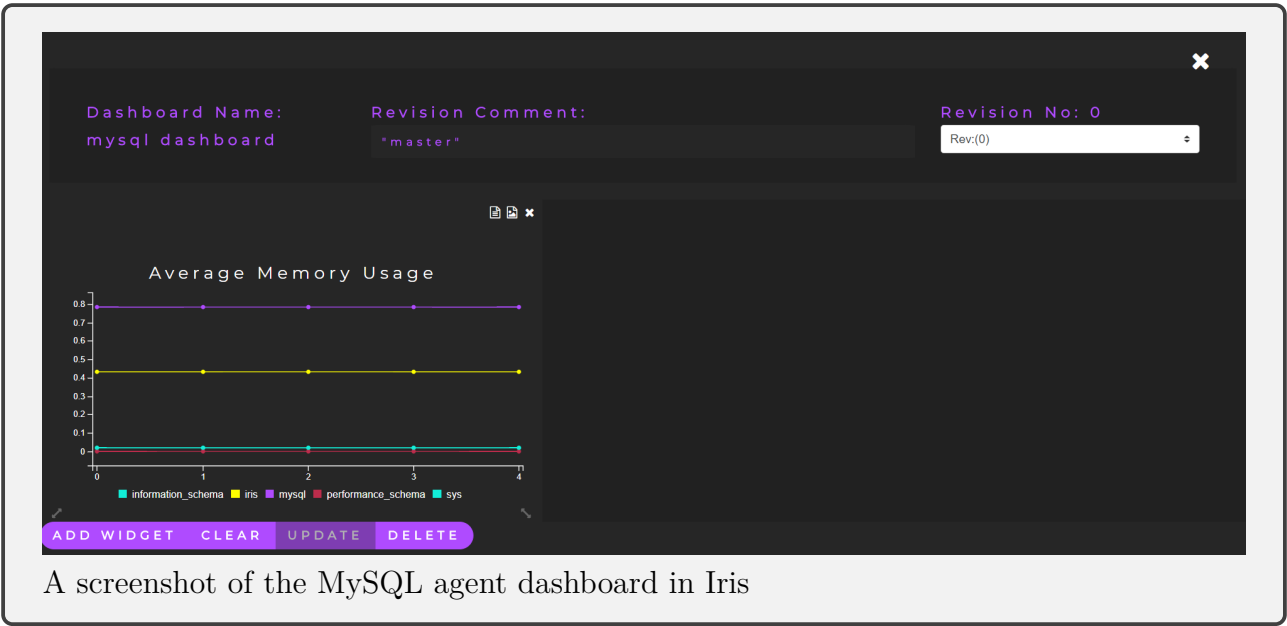


Figure 4.14 – MySQL agent dashboard in Iris

CHAPTER 5

Conclusions

Bibliography

- 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).
- (2017b). *Basic Concepts*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html (visited on 11/04/2017).
 - (2017c). *Mapping*. URL: <https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html> (visited on 11/04/2017).
- 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