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

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.

CHAPTER 2

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

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 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.

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

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.

URI

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

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.

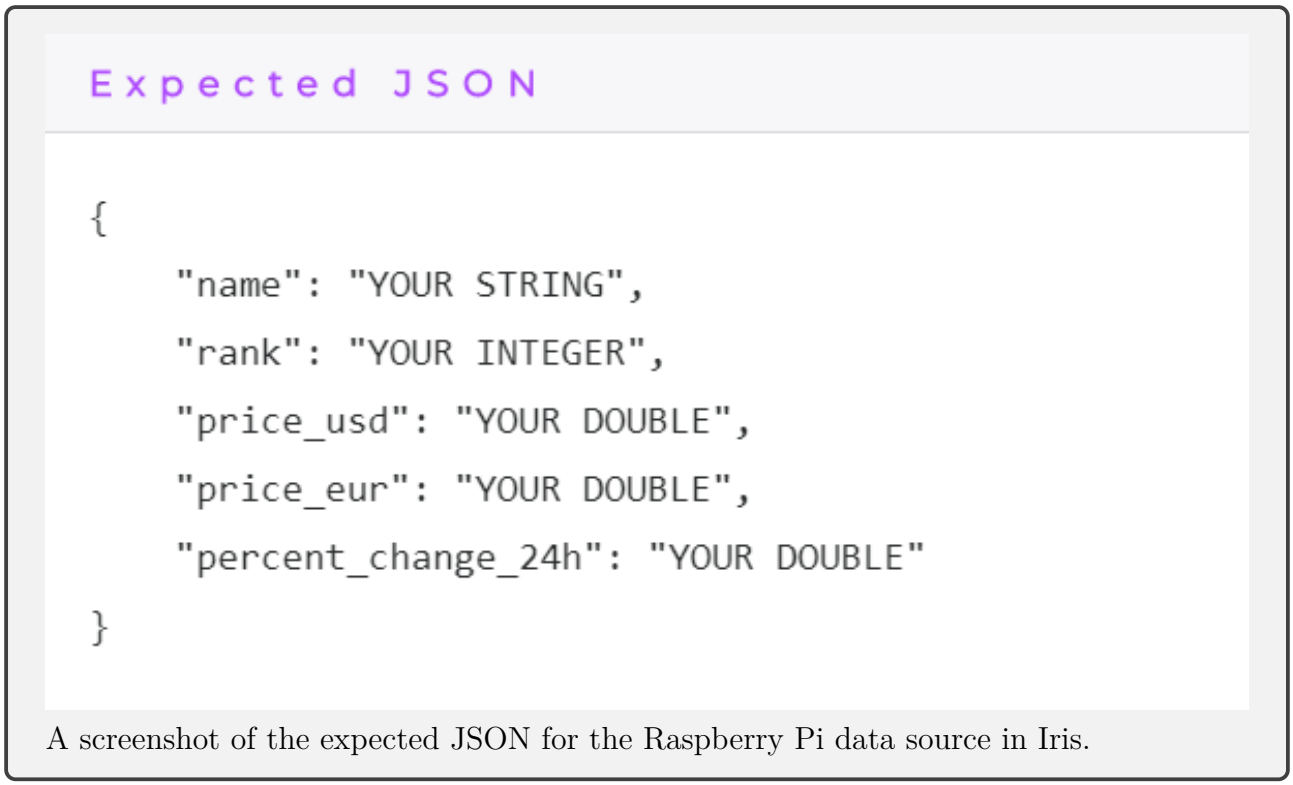


Figure 3.5 – Expected JSON for Raspberry Pi data source.

3.1.2.1.4 Transformation/Rule Script editor

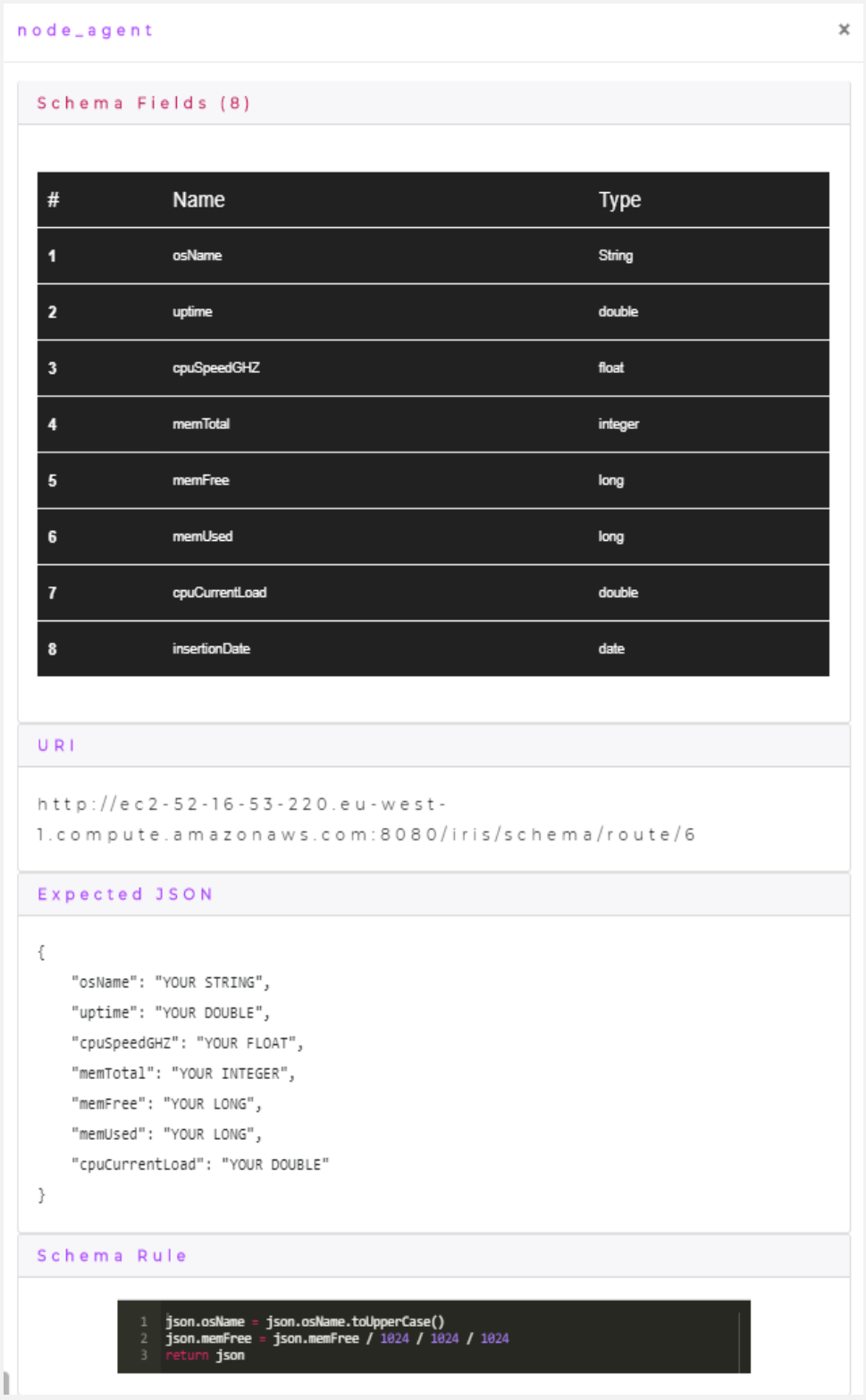
Due to Iris' ability to allow users to write transformation scripts that run on incoming data which is discussed in paragraph 3.1.2.2.1, a code editor was embedded into Iris. The code editor being used is the 'Ace' code editor with Groovy syntax highlighting which aims to aid the user in writing small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

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

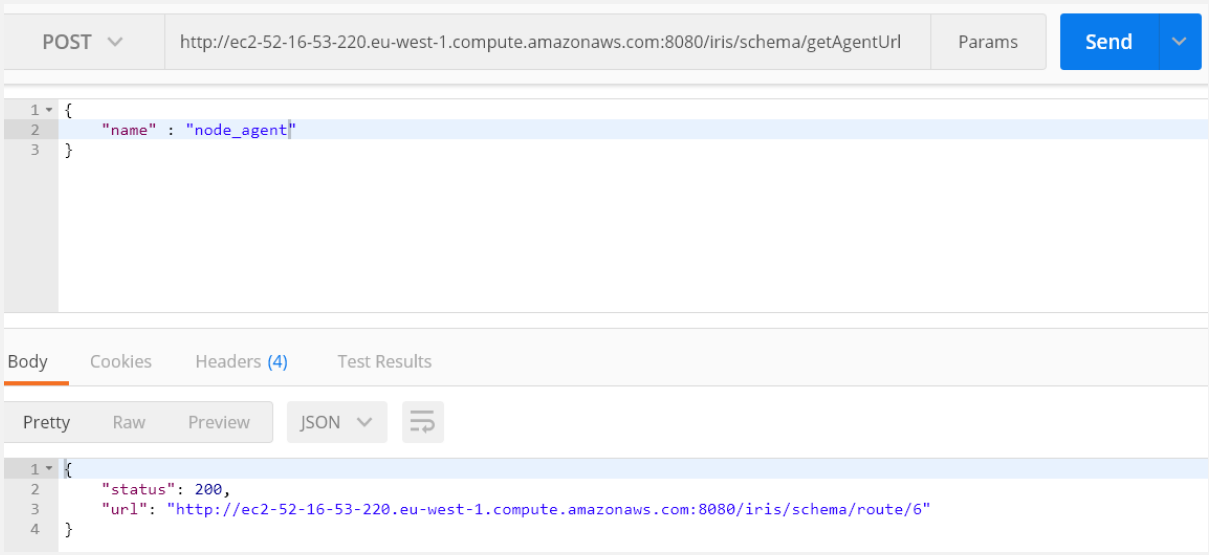
3.1.2.2.2 Data Source Endpoint Retrieval

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

During the Autumn Semester the javascript libraries for building the Iris dashboard system were selected and tested in detail. As a result, the design for implementing the dashboard and updating the charts in realtime has remained essentially unchanged during the implementation this Semester. The dashboard system features and implementation is discussed in terms of frontend and backend logic in the following sections. A brief overview of how the data flows into the dashboard is also be discussed.

3.1.3.1 Frontend

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

The frontend of the Iris system, is concerned with the charting and dashboard configuration components, which have been built using libraries discussed in the Semester 1 report.

3.1.3.1.1 Gridstack.js

Gridstack.js is a jQuery plugin that provides a drag-and-drop multi-column grid for generic widgets. Its features and the basis for selecting gridstack.js is covered in the Semester 1 report.

Due to the extensive research and testing on this library in Semester 1 and the active github

support from Gridstack.js developers there was no major issues during the implementation phase in Semester 2. Gridstack.js is used to contain the components (charts) of the dashboard in a grid format. Whenever a chart is created for a dashboard a new container widget is created containing the chart and added to the gridstack grid. This widget automatically becomes resizable, draggable and serialisable.

The logic used to format the gridstack grid into a serialisable object was retrieved from the Gridstack.js serialise demo[1]. Using the basic structure supplied by the gridstack.js developers, the Iris client code was developed and expanded the existing functionality to support chart types and aggregation objects. (see Figure 3.10a)

```
this.serializedData = _.map($('.grid-stack >
↪  .grid-stack-item:visible'), function (el) {
    el = $(el);
    var node = el.data('_gridstack_node');
    return {
        x: node.x,
        y: node.y,
        width: node.width,
        height: node.height
    };
}, this);
```

(a) Gridstack.js Developer's Serialization logic

```
serializedData = _.map($('.grid-stack >
↪  .grid-stack-item:visible'), function (el) {
el = $(el);
var node = el.data('_gridstack_node');
var widgetInfo = el.data();
return {
    x: node.x,
    y: node.y,
    width: node.width,
    height: node.height,
    id: node.el[0].id,
    schemaId: widgetInfo.schemaid,
    chartName: widgetInfo.chartname,
    chartType: widgetInfo.charttype,
    data: JSON.parse(localStorage.getItem(node.el[0].id))
};
}, this);
```

(b) Iris Dashboard Serialization logic

Figure 3.10 – Iris extension of Gridstack.js Serialization

[1] https://dsmorse.github.io/gridster.js/demos/serialize.html

One issue with gridstack.js, that was found during the implementation phase, is that the width of a widget can be incorrectly calculated at the upon loading a dashboard. Unfortunately this issue is not consistent and has proven difficult to resolve. Currently the only fix is for the user to click on the resizable handles of a widget and the chart then scales correctly.

For a basic demo of Gridstack.js refer to this url http://gridstackjs.com/demo/

3.1.3.1.2 Billboard.js

Billboard.js is a charting library based on D3.js. Billboard.js was researched and tested in detail during Semester 1 but was not integrated within Iris until Semester 2.

Integrating Billboard.js into Iris, and more specifically with Gridstack.js, was relatively straight forward, the only issue was sizing issue for loading charts mentioned above. Dynamically adding a Billboard.js chart to an existing Gridstack.js grid required an extra parent container to be wrapped around the chart element, effectively converting it to a Gridstack.js widget.

The chart type support will be discussed in paragraph 3.1.3.1.3 and the chart subscriptions will be discussed in paragraph 3.1.3.1.4.

For examples of charts created with Billboard.js see this url https://naver.github.io/billboard.js/demo/

3.1.3.1.3 Chart Types

Iris supports the following chart types:

- Bar Chart
- Bubble Chart
- Pie Chart
- Line Chart
- State Disc Chart (A chart used for monitoring state based data)

3.1.3.1.4 Chart Subscriptions

The design for how Iris handles sending incoming data to the correct charts is documented in the Semester 1 report. However the design was not implemented until Semester 2. The libraries chosen in Semester 1 for web socket functionality remained the same. Iris uses the Grails Web Socket plugin to add socket communication between dashboard charts and the server, allowing charts to subscribe to incoming data.

Each client chart on an Iris dashboard subscribes to messages from the server on a unique socket endpoint, which ensures the correct data gets sent to the correct chart. This design is discussed in the Semester 1 report and has remained the same during the implementation of the chart subscription logic in Iris.

For more information on he Grails 3 'grails-spring-websocket' plugin refer to https://github.com/zyro23/grails-spring-websocket.

```
function setChartSubscription(subscriptionId, chart, chartType,
 ↪  schemaId){
    //this subscription is for updates being sent to the chart
    client.subscribe("/topic/" + schemaId + "/" + chartType + "/" +
     ↪  subscriptionId, function(message) {
        var parsedMsg = JSON.parse(message.body);
        //update the chart
        if(chartType != chartTypes.StateDisc){  //REGULAR CHARTS
            updateBasicCharts(chart, parsedMsg);
        }else{  //STATE BASED CHARTS
            updateStateDiscChart(chart, parsedMsg);
        }
    });

    //this subscription is for initial loading data for the chart
    client.subscribe("/topic/load/" + schemaId + "/" + chartType + "/" +
     ↪  subscriptionId, function(message){
        var parsedMsg = JSON.parse(message.body);
    // update the chart
        chart.instance.load({
            columns: parsedMsg.data.columns,
            length: 0
        });
    });
}
```

Figure 3.11 – Iris Client Side Chart Subscription logic.

3.1.3.1.5 Downloadable Chart Data

Iris supports the ability to download a JSON file consisting of the current data being displayed on any chart. This functionality was built by using the Billboard.js API and retrieving the chart data, and adding custom logic to format that data into a file to be downloaded upon request.

A screenshot of a downloadable chart in Iris, taken from the 'selenium agent' dashboard in Iris. The file icon in the top right is clicked to start the download.

Figure 3.12 – Downloadable Chart Example


```
[{"id":"Music Store","id_org":"Music
↪  Store","values":[{"x":0,"value":888,"id":"Music
↪  Store","index":0},{"x":1,"value":888,"id":"Music
↪  Store","index":1,"name":"Music
↪  Store"},{"x":2,"value":888,"id":"Music
↪  Store","index":2},{"x":3,"value":888,"id":"Music
↪  Store","index":3},{"x":4,"value":888,"id":"Music
↪  Store","index":4}]},{"id":"Thomann","id_org":"Thomann",
"values":[{"x":0,"value":888,"id":"Thomann","index":0},
{"x":1,"value":888,"id":"Thomann","index":1,"name":"Thomann"},
{"x":2,"value":888,"id":"Thomann","index":2},
{"x":3,"value":888,"id":"Thomann","index":3},
{"x":4,"value":888,"id":"Thomann","index":4}]}]
```


Figure 3.13 – JSON file data downloaded from a chart on the 'selenium agent' dashboard as shown in fig. 3.12.

3.1.3.1.6 Revision History

Iris now supports revision history for all dashboards, meaning a user can navigate between previous versions of a dashboard by simply using a select box on the dashboard page which allows the user select the revision they wish to view. Each revision commit allows a user to

write a comment about what changed for the new revision. A user may also delete revisions they no longer wish to keep, once all revisions are deleted then a dashboard is deleted.

A screenshot of the Node.js dashboard in Iris capturing its fourth revision.

Figure 3.14 – Node.js Dashboard Revision No. 4

A screenshot of the Node.js dashboard in Iris capturing its fifth revision. Note the extra chart 'OS Types' that was added to this revision compared to the fourth revision show in fig. 3.14.

Figure 3.15 – Node.js Dashboard Revision No. 5

3.1.3.2 Backend

This section will focus on the major factors behind the backend implementation of the Iris dashboard system.

Within this section designs from the Semester 1 report will be discussed in terms of their implementation phase during Semester 2 as well as some new features that were designed and added within Semester 2.

3.1.3.2.1 Chart Types

The Iris chart types are handled by using enums in Iris. This allows for the code in Iris to be easily expanded upon when a new chart type is needed, and the use of enums allows for more readable code when dealing with different types of charts.

```
enum ChartType {

    BAR("Bar"),
    BUBBLE("Bubble"),
    PIE("Pie"),
    LINE("Line"),
    STATE_LIST("StateList"),
    STATE_DISC("StateDisc");

    private String value;

    private ChartType(String value){
        this.value = value;
    }

    String getValue(){ return this.value; }
}
```

Figure 3.16 – ChartType.groovy enum from Iris.

Due to Iris supporting multiple chart types as well as different types of data, each chart has optional attributes. Some of these optional attributes consist of an Elasticsearch aggregation if the user created the chart using the aggregation builder, and a raw attribute in the case of a user wishing to just send raw data to a chart and not have Elasticsearch perform any aggregations over the incoming data.

```
class Chart {

    String name
    String chartType
    String subscriptionId
    Aggregation aggregation
    IrisSchema schema
    boolean isRaw = false
    boolean archived = false

    static constraints = {
        name(nullable: false, blank: false)
        chartType(nullable: false, inList:
        ↪  ChartType.values()*.getValue())
        aggregation(nullable: false)
        schema(nullable: false)
        archived(nullable: true)
        subscriptionId(nullable: true)
        isRaw(nullable: true)
    }

    static belongsTo = [grid: Grid]
}
```

Figure 3.17 – Chart.groovy domain for representing Charts in Iris.

3.1.3.2.2 Chart Subscriptions

When data enters Iris, Iris uses the chart type to figure out how a chart should be updated. For all charts the flow of data is the same, a data source will send data to Iris and Iris will send the data to the chart using a Grails Service which which comes as part of the 'grails-spring-websocket' plugin. If a chart is marked as raw or of type 'State Disc' then the data being sent to the chart is not modified, if a chart instance has an Elasticsearch aggregation attached to it the aggregation is executed in Elasticsearch and the result is parsed and formatted to suit the chart type and then sent to the chart to be updated on the frontend.

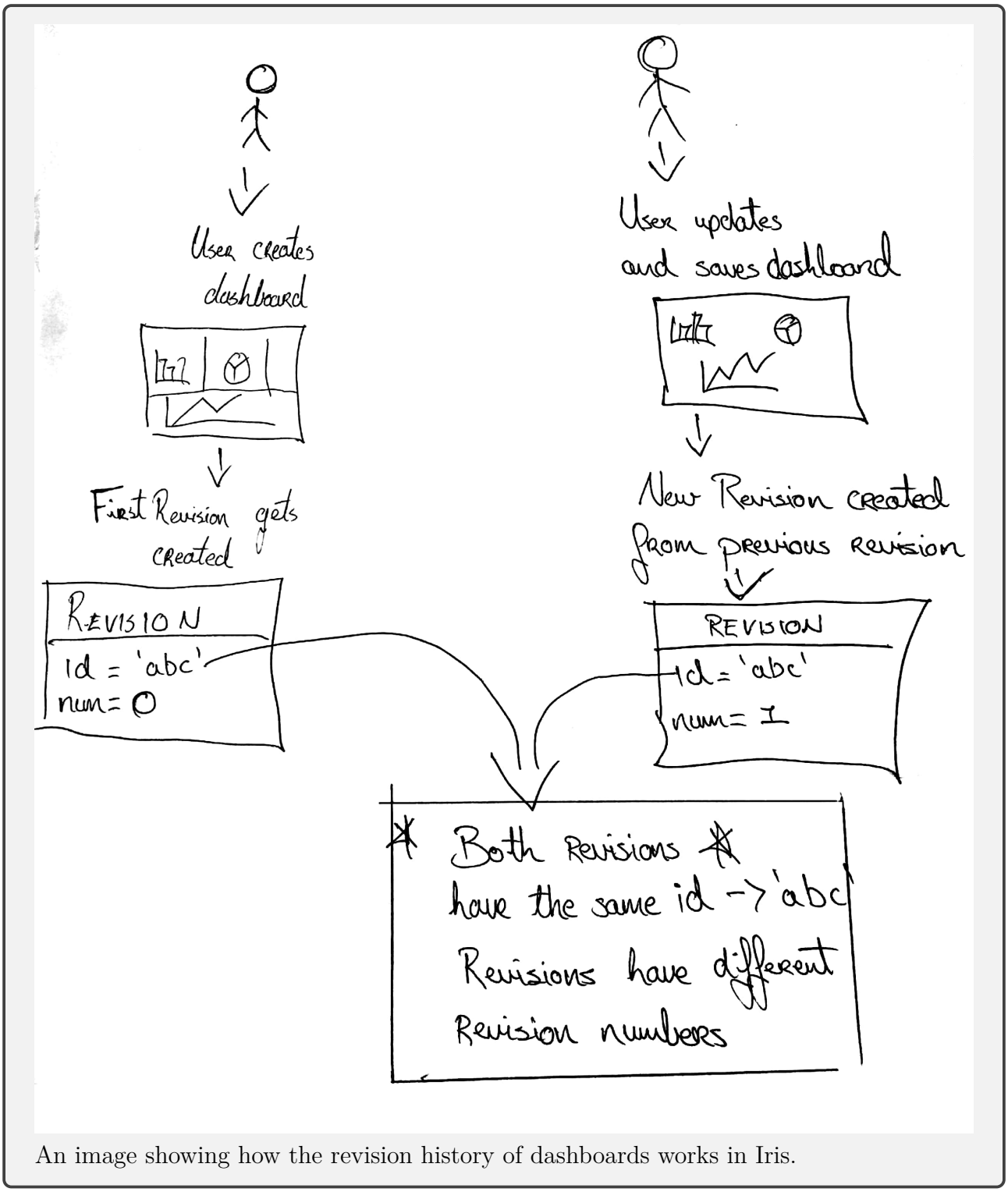
3.1.3.2.3 Serilization

Iris serializes Dashboard domains through composition. Each Dashboard domain has a Grid domain object which stores the entire dashboard in JSON. This means that serializing and loading dashboards is simply done with a string of JSON. This JSON is passed to the frontend where it is parsed by the client to create the dashboard.

Due to Iris storing the dashboard grid as a JSON string, it allows for Iris to extend its dashboard system in the future by allowing users to build dashboards offline or locally and upload a JSON file which represents their dashboard. Iris will be able to save and load the dashboard as long as the user conforms to the existing dashboard JSON structure which is in place.

3.1.3.2.4 Revision History

Iris supports revision history for its dashboard system. When a dashboard is initially created Iris will create a new Revision domain object to associate with the dashboard. This revision object has a unique id which is referred to as the 'revisionId', this id is specific to each dashboard. Along with the revision id, a revision number is also stored, meaning all revisions specific to a dashboard have the same revision id and different revision numbers. Whenever a dashboard is updated and the changes are committed, Iris will create a new revision for the dashboard by coping the revision id of the most recent revision and then incrementing the revision number of the most recent revision. This results in a brand new revision being created. When a dashboard is rendered on the frontend of Iris the backend sends down all the revisions associated with the dashboard which allows the user to select what version of the dashboard they wish to view.



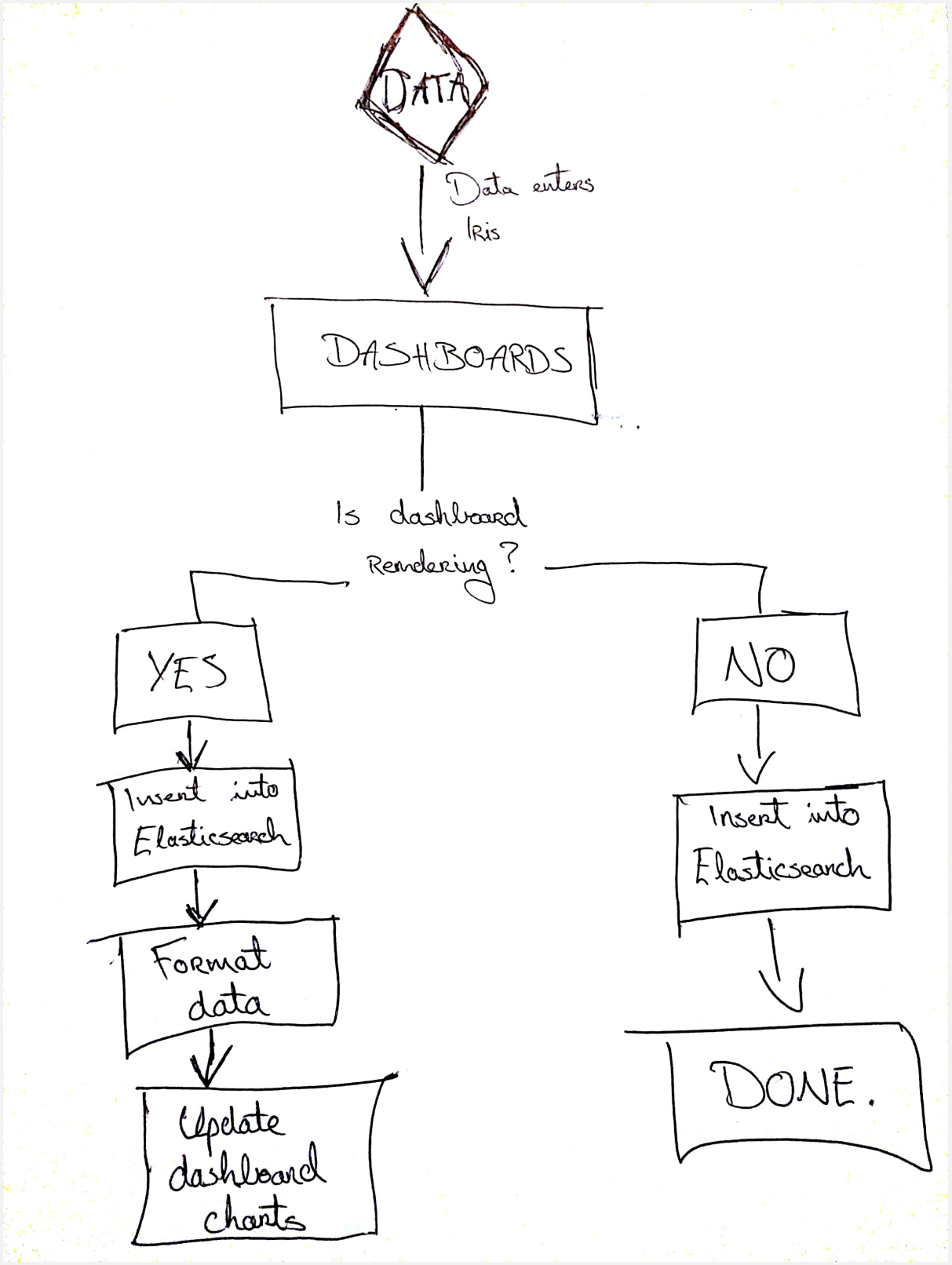
An image showing how the revision history of dashboards works in Iris.

Figure 3.18 – Iris Dashboard Revision Diagram.

3.1.3.2.5 Scalability

A big concern with Iris in the beginning was to see how scalable the dashboard system would be if a lot of data was being sent to Iris. In an effort to make the dashboard system more efficient Iris dashboards have a boolean attribute called 'isRendering'. When a dashboard is being viewed by a user the backend toggles the 'isRendering' state of the dashboard to true, this is switched to false once a dashboard is no longer being viewed. The reason for this is to prevent data being modified and formatted for a dashboard which is not being rendered to the

user. By having this state attached to a dashboard, a dashboard will only be sent updates if it is being viewed by a user, otherwise the data will simply be entered into Elasticsearch and no more code will be executed. This makes the dashboards much more efficient and scalable as Iris is only dealing with dashboards that are being viewed.



A flow chart showing how Iris handles dashboards that are rendering versus dashboards that are not rendering.

Figure 3.19 – Iris Dashboard Scalability.

3.1.3.2.6 User Specific

In the Semester 1 report similar dashboard tools were compared with the design for Iris. One of the most obvious tools to compare against Iris was Kibana due to it being apart of the ELK (Elasticsearch, Logstash, Kibana) stack. In the Semester 1 report it was discussed that Kibana does not support user specific dashboards, meaning all users can see all dashboards. Iris treats both dashboards and schemas as being user specific, meaning each user has their own personal set of dashboards in comparison to Kibana which does not.

3.1.4 Aggregation Builder

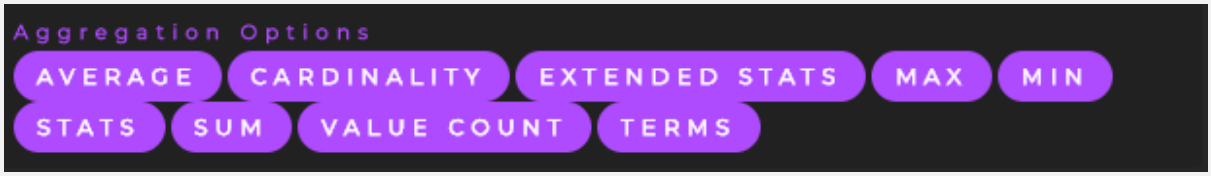
The aggregation builder was partially implemented in Semester 1, the aggregation types were in place and the logic for building an Elasticsearch aggregation through the Iris UI were in place. In Semester 2 some more features were added to allow for more fine grained data monitoring. In the following sections the current aggregation types that Iris supports are listed and the new aggregation builder previewer are discussed.

3.1.4.1 Aggregation Types

In this section the aggregations which Iris currently supports are discussed. The following list contains the aggregations that Iris can currently build through the aggregation builder.

- Average (Calculates the average of a numerical field).
- Cardinality (Calculates the approximate count of distinct values).
- Stats (Returns basic stats on a numerical field such as count, min, max, avg, sum).
- Extended Stats (Returns a grouping of stats on a numerical field such as max, min, count, avg, sum, sum of squares, variance, standard deviation).
- Max (Calculates the max of a numerical field).
- Min (Calculates the min of a numerical field).
- Sum (Calculates the sum of a numerical field).
- Value Count (Calculates the number of values extracted from aggregated documents).

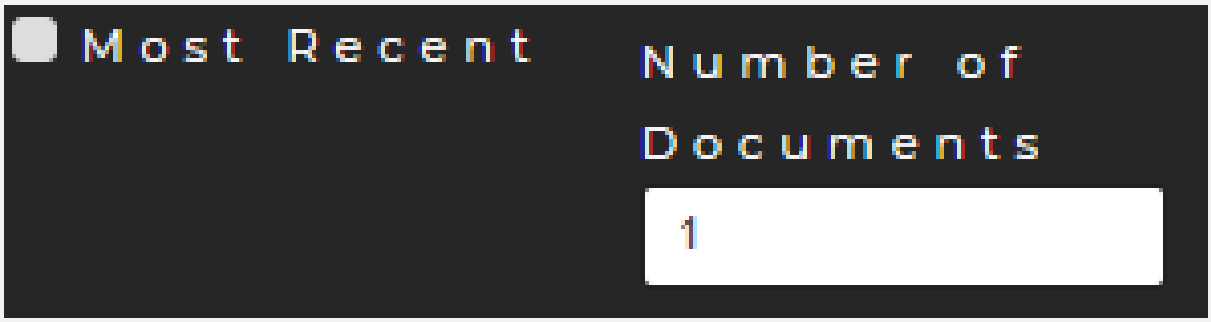
Some of the aggregations listed above are not suited for use on charts as the data they return are not suited for charts, for example the stats and extended stats aggregations would not be ideal for fitting to a chart. However they are very useful to get an overall view of the data quickly without having to worry about fitting the data to a chart.



A screenshot showing the aggregation options available inside the Iris aggregation builder.

Figure 3.20 – Iris Aggregation Builder Options.

In Semester 1 the aggregations were running over all documents in the Elasticsearch index, this was fine for demonstrating the aggregation builder's ability to build aggregations, but for realtime data this would be an expensive way to run aggregations, especially if the Elasticsearch index contained a large amount of data. To tackle this issue there was an option to run aggregations over the most recent entries in the Elasticsearch index and to specify how many entries you want to run the aggregation over. This is a powerful modification to the aggregation builder as now users can take advantage of creating charts that display moving averages, minimums and maximums, as well as avoid running an aggregation over all of the entries in the Elasticsearch index which would be slow.



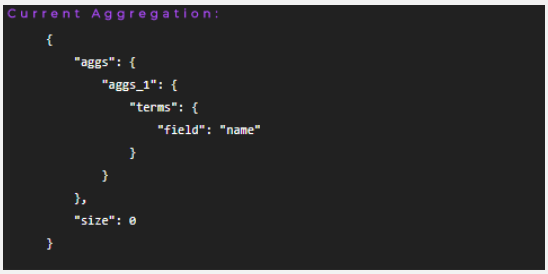
A screenshot showing the aggregation most recent option.

Figure 3.21 – Iris Aggregation Most Recent Option.

It is not advisable to create aggregations for charts which are not run over the most recent entries, especially if the data in the Elasticsearch index is large. If a user wishes to run these aggregations they should do so in the aggregation builder area, where no dashboards need to be updated and time is not a concern.

3.1.4.2 Aggregation Builder Preview

Iris now supports a preview of the aggregation JSON object being built as the user interacts with the aggregation builder UI. This gives the user a look into how the aggregation objects are constructed and allows them to copy the aggregation object in case they wish to execute the aggregation outside of Iris and don't know how to construct an Elasticsearch aggregation.



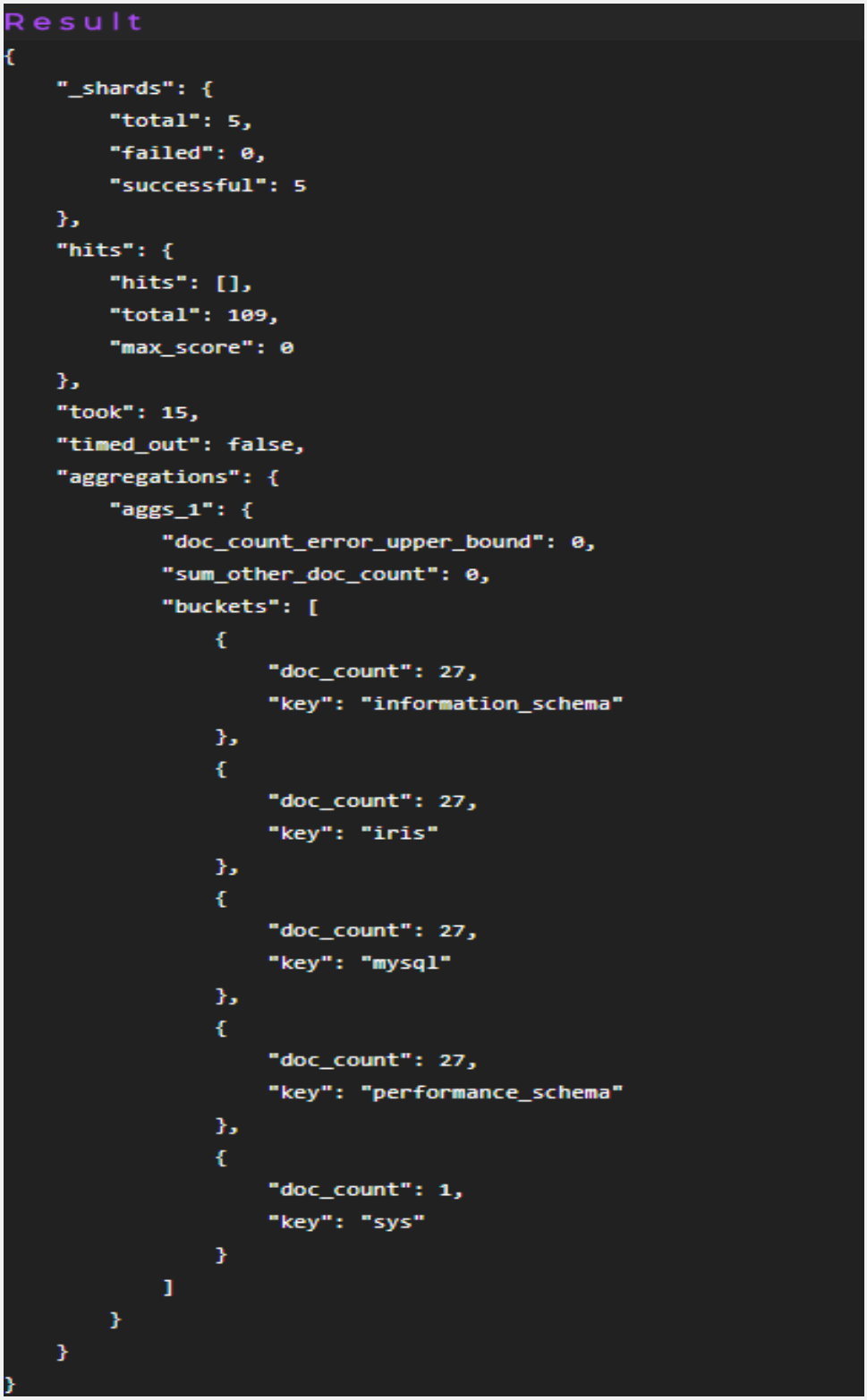
(a) Current Aggregation with one level.

(b) Current Aggregation after nesting an aggregation inside the previous aggregation.

Figure 3.22 – Iris aggregation builder Preview.

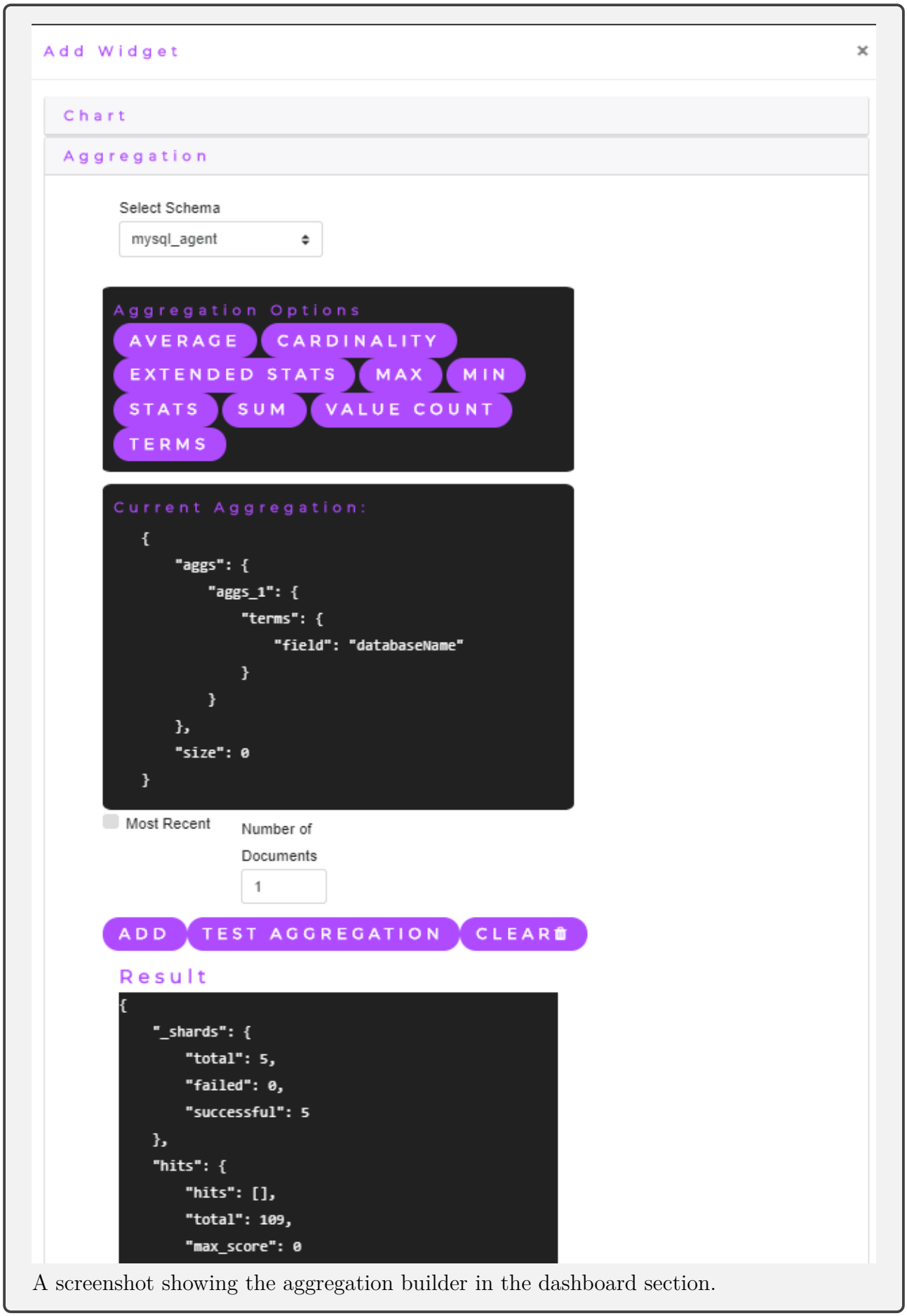
3.1.4.3 Testing Aggregations

In Semester 1 it was demonstrated that Iris test users could use the aggregation builder area to test their aggregations before putting using the aggregation for a chart. However the aggregation builder was on a different page to that of the dashboards. In Semester 2 the aggregation builder was added to the dashboard creation area, giving users the full functionality of the aggregation builder as they make a chart for a dashboard. This allows a user to test their aggregations before saving a chart to a dashboard to make sure the data being returned is satisfactory.



A screenshot showing the aggregation result returned from testing an aggregation inside the aggregation builder area.

Figure 3.23 – Iris Aggregation Test Result (aggregation builder area).



A screenshot showing the aggregation builder in the dashboard section.

Figure 3.24 – Iris Aggregation Test Result (Dashboard area).

3.1.5 Security

In Semester 1 it was mentioned that the Grails Spring Security plugin would be used for authentication in Iris. The following sections will discuss the security improvements made from Semester 1 to Semester 2.

3.1.5.1 Spring Security Core Plugin

In Semester 1 Iris used the Spring Security Core plugin to create Users, Roles and User Roles. The plugin allowed Iris to authenticate all endpoints by not allowing a user access to the web application unless they are a valid user who has been fully authenticated i.e logged in.

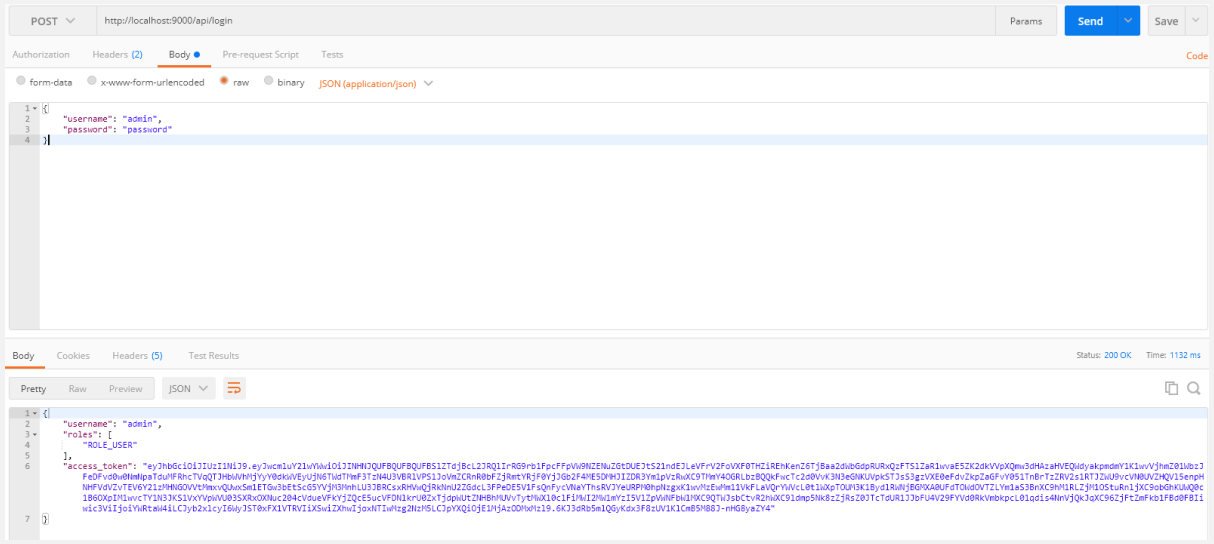
A basic role association was created for the user called 'ROLE_USER'. This tells Iris what access a user with this specific role has within the web application.

Spring Security also automatically deals with hashing passwords for users which adds an extra layer of security for users.

3.1.5.2 Spring Security REST Plugin

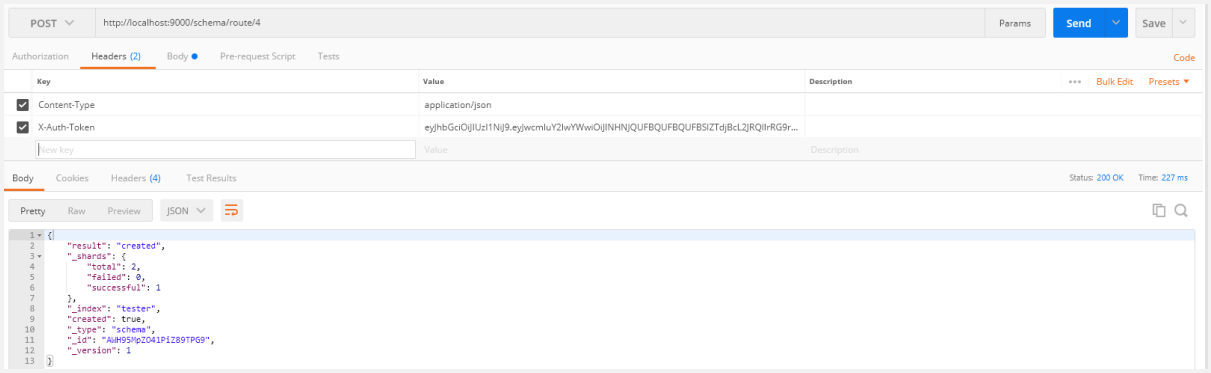
Throughout Semester 2 it was apparent that the data sources sending data to Iris needed to be authenticated as they may be posing as a fake data source. The Grails Spring Security Rest plugin was decided upon as being an appropriate way to authenticate data sources accessing Iris API endpoints.

The plugin allows a user to login to Iris using a POST request which contains their credentials in a JSON payload. If a user is successfully authenticated Iris responds with a JWT (JSON Web Token) for the user to send as part of future requests to the Iris API endpoints. If a request is received with a valid JWT token in the X-Auth-Token header the data source is granted access to the API endpoint.



A screenshot showing Iris API login being used to authenticate a valid Iris user, note the JWT token that was sent back.

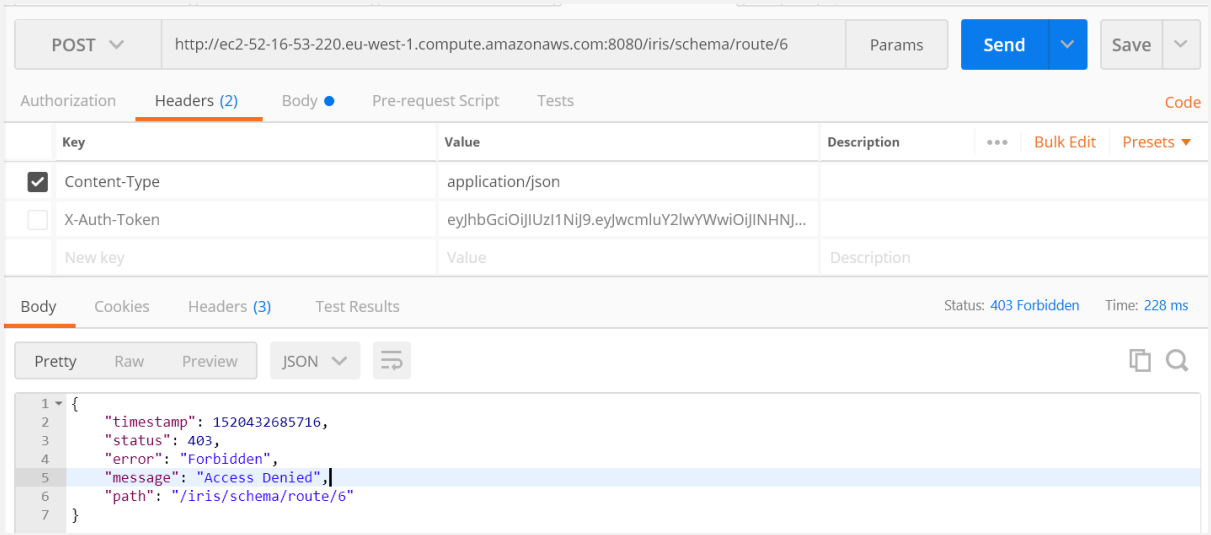
Figure 3.25 – Iris Login API.



A screenshot showing a data source being authenticated by its JWT in the 'X-Auth-Token' header of the request.

Figure 3.26 – Iris JWT Validation

If the user's credentials are invalid or the JWT sent by the data source is invalid the user will be denied access to the endpoint.



A screenshot showing the access denied message recieved upon authentication failure with Iris.

Figure 3.27 – Iris API Access Denied

Using the plugin, Iris configured its endpoint security based on the endpoints available to the user. All endpoints which are part of the frontend of Iris are completely locked unless a user is fully authenticated, meaning a user must sign in to use any of the user interface of Iris.

The two REST endpoints Iris exposes to data sources are '/schema/route/' and '/schema/getAgentUrl'. These endpoints are used by data sources to get unique schema endpoints and sent data to Iris. Both of these endpoints are secured fully and need full authentication by logging in using the previously mentioned '/api/login' as well as a valid JWT. In the backend of Iris these methods are also marked as needing a user with the 'ROLE_USER' role to be given access.

```
/**
 * Takes in data for transformation and routing to elasticsearch
 * data is also sent to any charts needing to be updated
 */
@Secured('ROLE_USER')
def route(long id){
    Map resp = ["status": 200, "message": "data inserted"]
    IrisSchema schema = IrisSchema.get(id)
    if(schema == null){
        resp.status = 500
        resp.message = "Schema with id $id does not exist"
    }else{
        resp = routeService.route(schema, request.JSON).json as Map        //route and transform data
        //get all dashboards that are currently marked as rendering
        dashboardService.updateDashboardCharts(id, request.JSON)
    }
    render resp as JSON
}
```

A screenshot showing the '/schema/route/' endpoint being secured with the 'Secured' annotation.

Figure 3.28 – Iris Secured Route.

The only endpoints in Iris that require no authentication are the '/login/auth' endpoint which returns the login page for a user who is using the web application to login and the '/api/login' for a data source who is logging in through a POST request. All other endpoints are secured.

```
// Added by the Spring Security Core plugin:
grails.plugin.springsecurity.userLookup.userDomainClassName = 'com.wit.iris.users.User'
grails.plugin.springsecurity.userLookup.authorityJoinClassName = 'com.wit.iris.users.UserRole'
grails.plugin.springsecurity.authority.className = 'com.wit.iris.users.Role'
grails.plugin.springsecurity.securityConfigType = 'InterceptUrlMap'
grails.plugin.springsecurity.interceptUrlMap = [
        [pattern: '/',              access: ['permitAll']],
        [pattern: '/error',         access: ['permitAll']],
        [pattern: '/index',         access: ['permitAll']],
        [pattern: '/index.gsp',     access: ['permitAll']],
        [pattern: '/shutdown',      access: ['permitAll']],
        [pattern: '/assets/**',     access: ['permitAll']],
        [pattern: '/**/js/**',      access: ['permitAll']],
        [pattern: '/**/css/**',     access: ['permitAll']],
        [pattern: '/**/images/**',  access: ['permitAll']],
        [pattern: '/**/favicon.ico', access: ['permitAll']],
        [pattern: '/login/**',            access: ['permitAll']],
        [pattern: '/api/login',         access: ['permitAll']],
        [pattern: '/api/logout',       access: ['isFullyAuthenticated()']],
        [pattern: '/rest/**',    access: ['isFullyAuthenticated()']],
        [pattern: '/schema/route/**',    access: ['isFullyAuthenticated()']],
        [pattern: '/schema/getAgentUrl',    access: ['isFullyAuthenticated()']],
        [pattern: '/**',            access: ['isFullyAuthenticated()']]

]



grails.plugin.springsecurity.filterChain.chainMap = [

        [pattern: '/api/**', filters:'JOINED_FILTERS,-anonymousAuthenticationFilter,-exceptionTranslationFilter,-authenticationProcessingFilter,-secu

        [pattern: '/schema/route/**', filters:'JOINED_FILTERS,-anonymousAuthenticationFilter,-exceptionTranslationFilter,-authenticationProcessingFil
        [pattern: '/schema/getAgentUrl', filters:'JOINED_FILTERS,-anonymousAuthenticationFilter,-exceptionTranslationFilter,-authenticationProcessing

        [pattern: '/**', filters:'JOINED_FILTERS,-restTokenValidationFilter,-restExceptionTranslationFilter']

]
```

A screenshot showing the spring security configuration settings for all of the Iris endpoints.

Figure 3.29 – Iris Spring Security Configuration.

For more information on JWT, Spring Security Core and Spring Security Rest, please refer to these urls:

- JWT - https://jwt.io/
- Spring Security Core - https://github.com/grails-plugins/grails-spring-security-core
- Spring Security Rest - https://github.com/alvarosanchez/grails-spring-security-rest

3.1.5.3 Data Source Adaptation

Data sources communicating with Iris now need to get authenticated by sending valid user credentials to the '/api/login' endpoint. The data source must then use the JWT token with every request as part of the X-Auth-Token header which was given to them from Iris upon a successfully logging in.

The data sources used to test Iris have hardcoded JWTs in their code to make testing easier, however all real use cases must use the '/login/api' to retrieve their JWTs.

3.2 Data Sources

A data source can be any process that generates data and pushes it to Iris. In section 4.2 a 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.

CHAPTER 4

Deployment, Testing and Evaluation

4.1 Deployment

4.1.1 AWS

4.1.1.1 EC2 Instance

4.1.1.2 Elasticsearch Instance

4.1.2 Jetty

4.1.3 MySQL Instance

4.2 Testing and Evaluation — Data Sources

4.2.1 Introduction

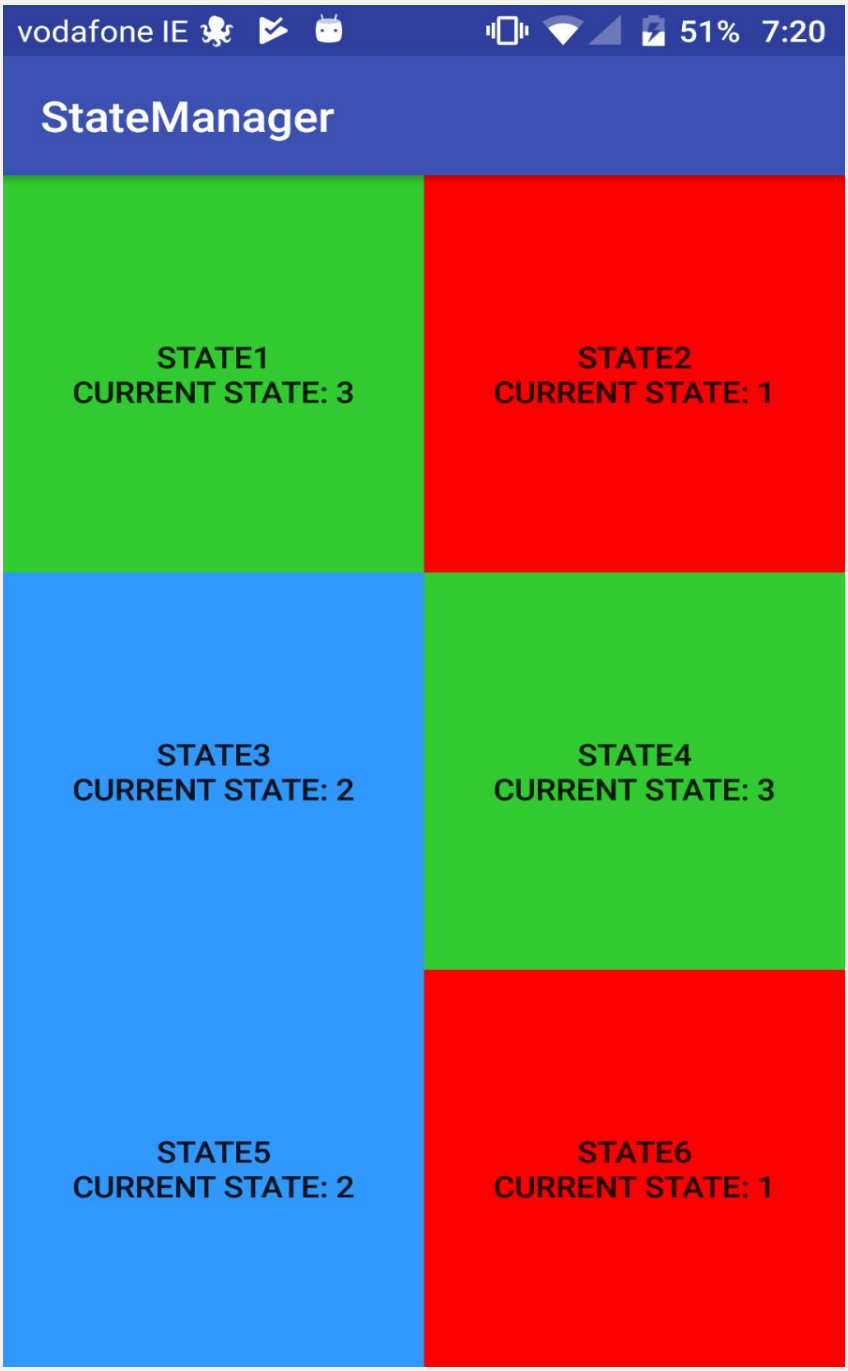
A number of data source have been implemented to demonstrate the flexibility of Iris and to test the aggregation. Each of the following sections describe a data source and its unique features.

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.



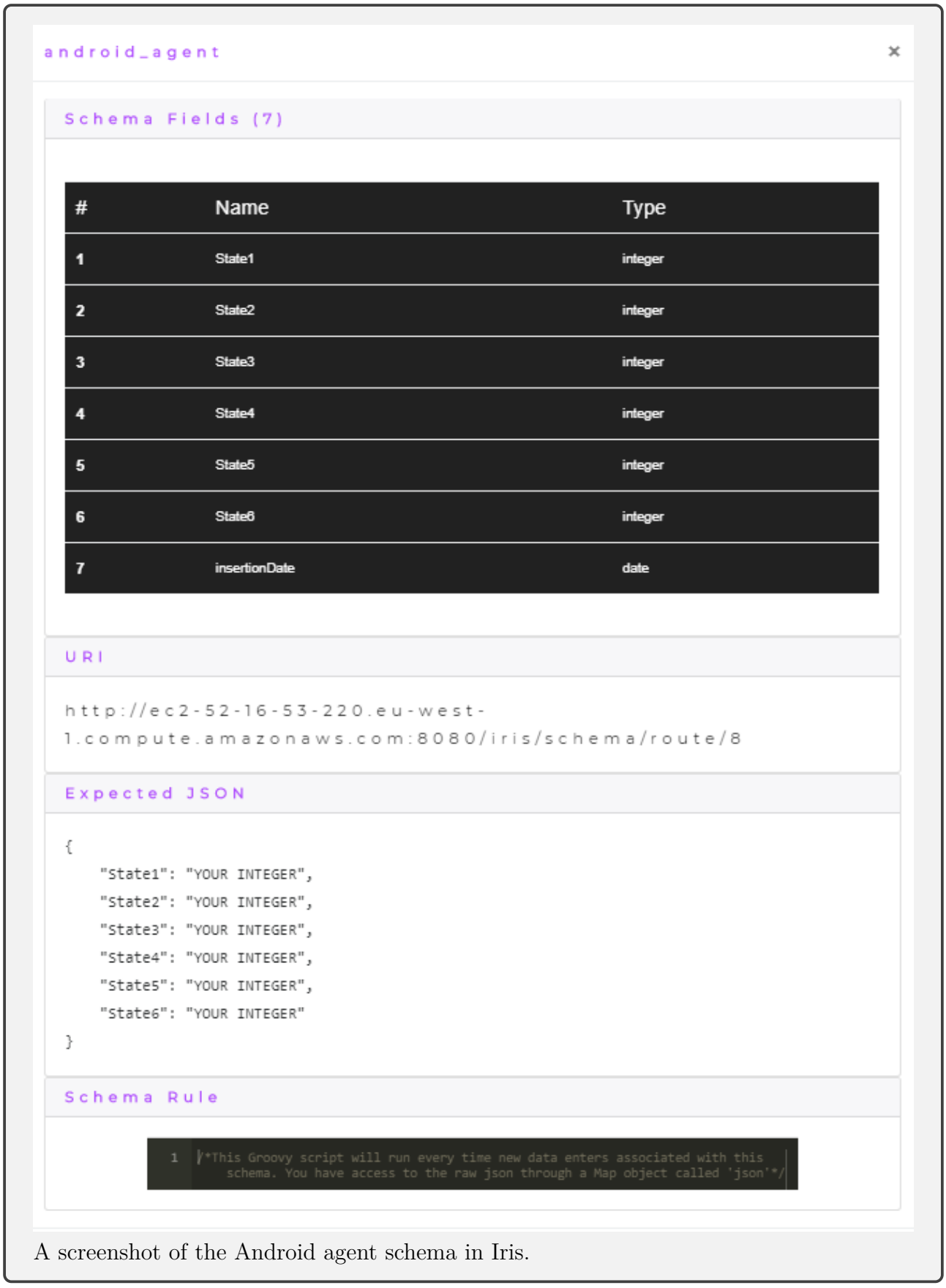
A screenshot of the android agent and its state based tiles. Each of the tiles on the screen holds a state, the states are as follows:

- 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.

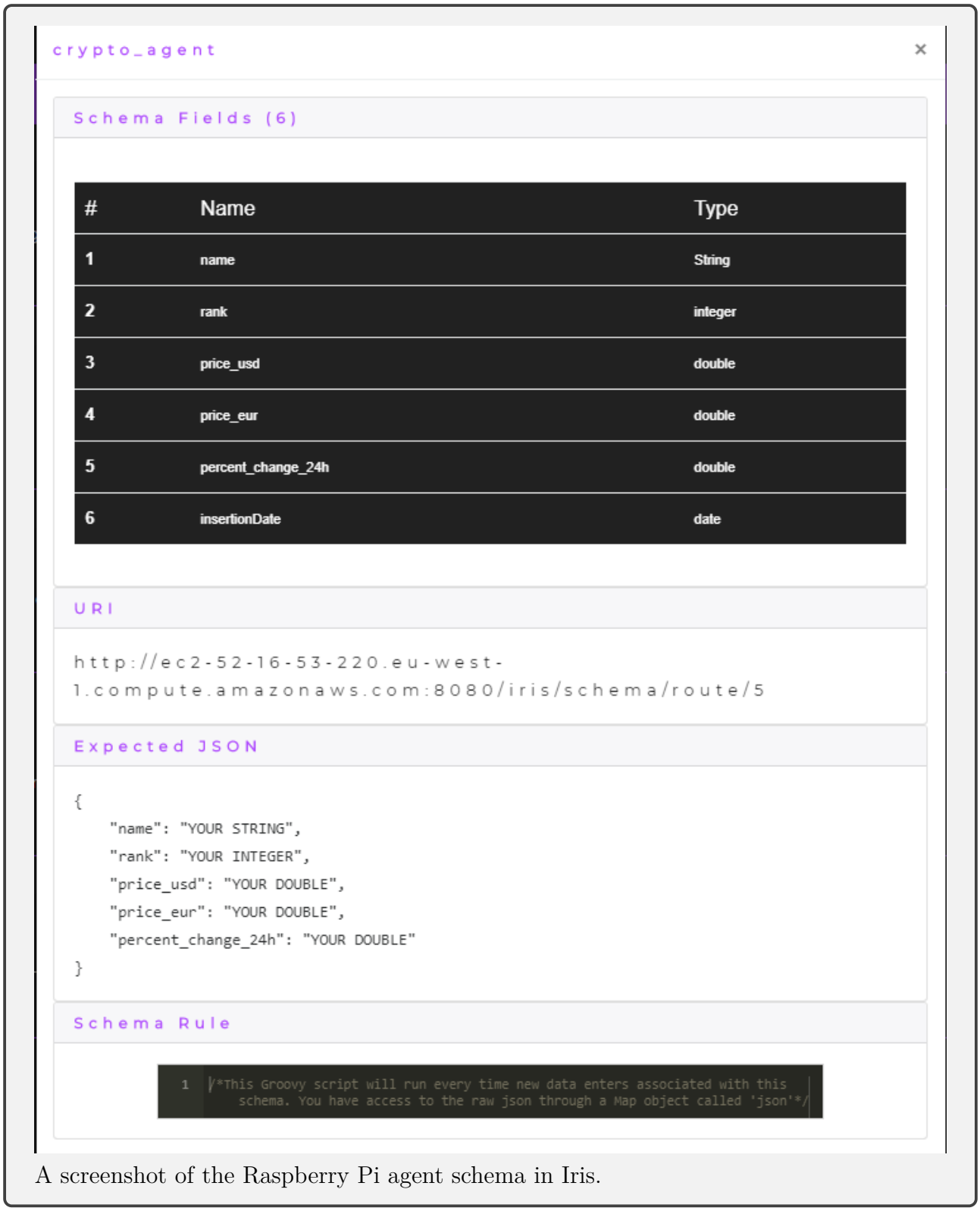
4.2.3.1 Description

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/

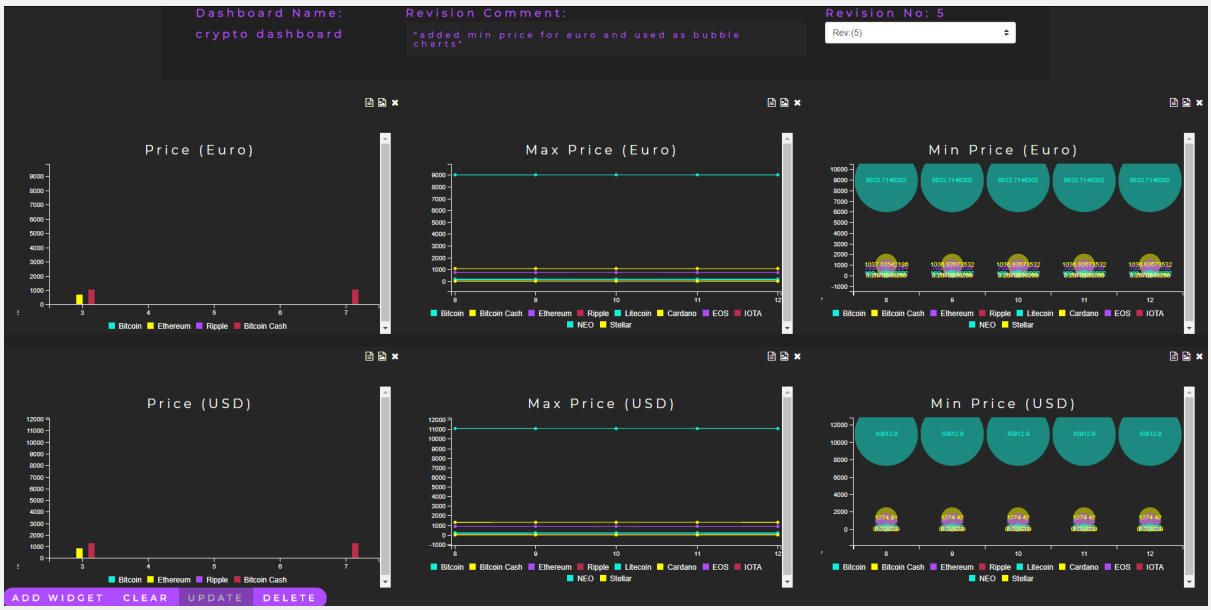
4.2.3.2 Linkage with Iris

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

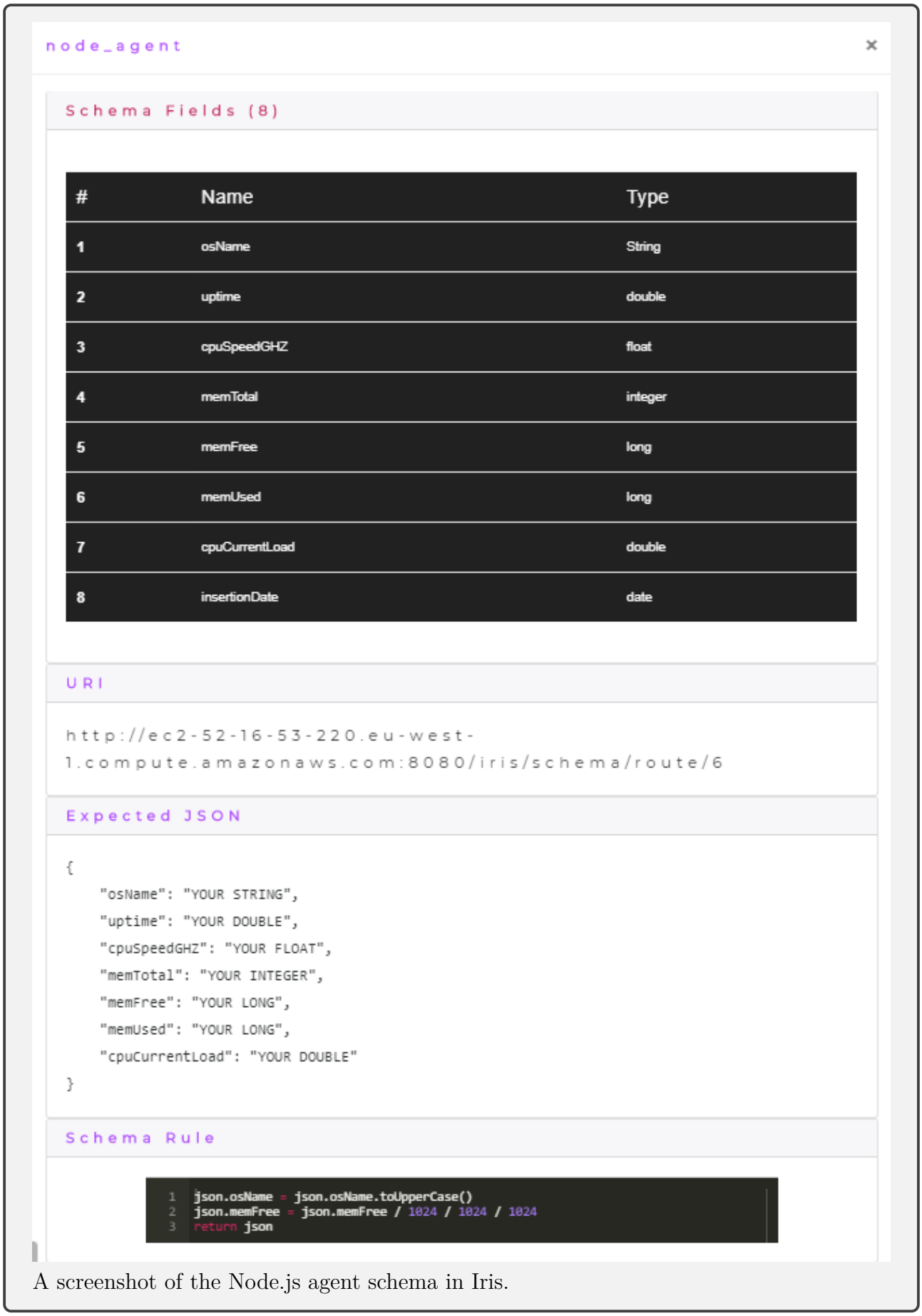
```
1  json.osName = json.osName.toUpperCase()
2  json.memFree = json.memFree / 1024 / 1024 / 1024
3  return json
```

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.

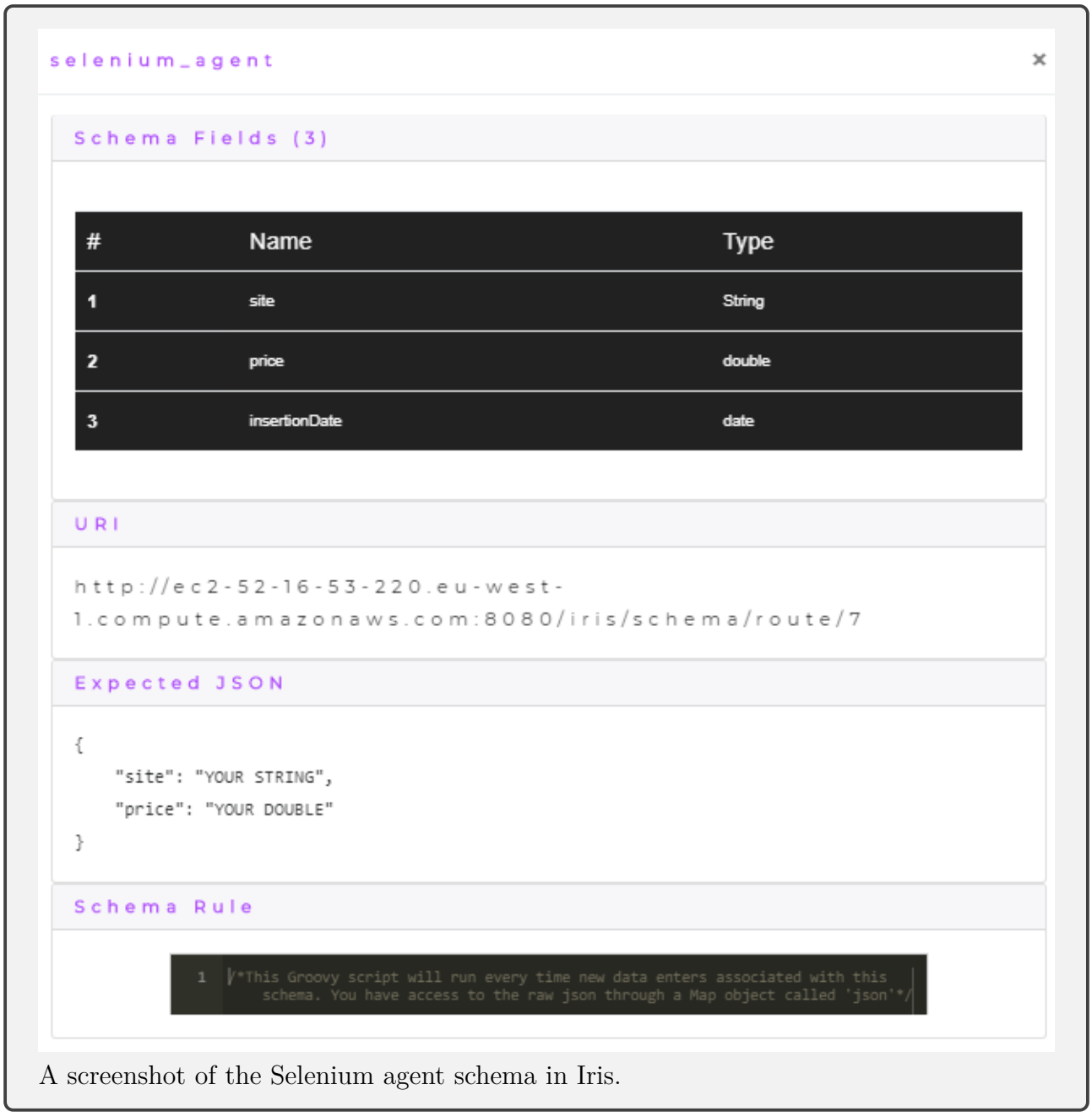
4.2.5.1 Description

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/

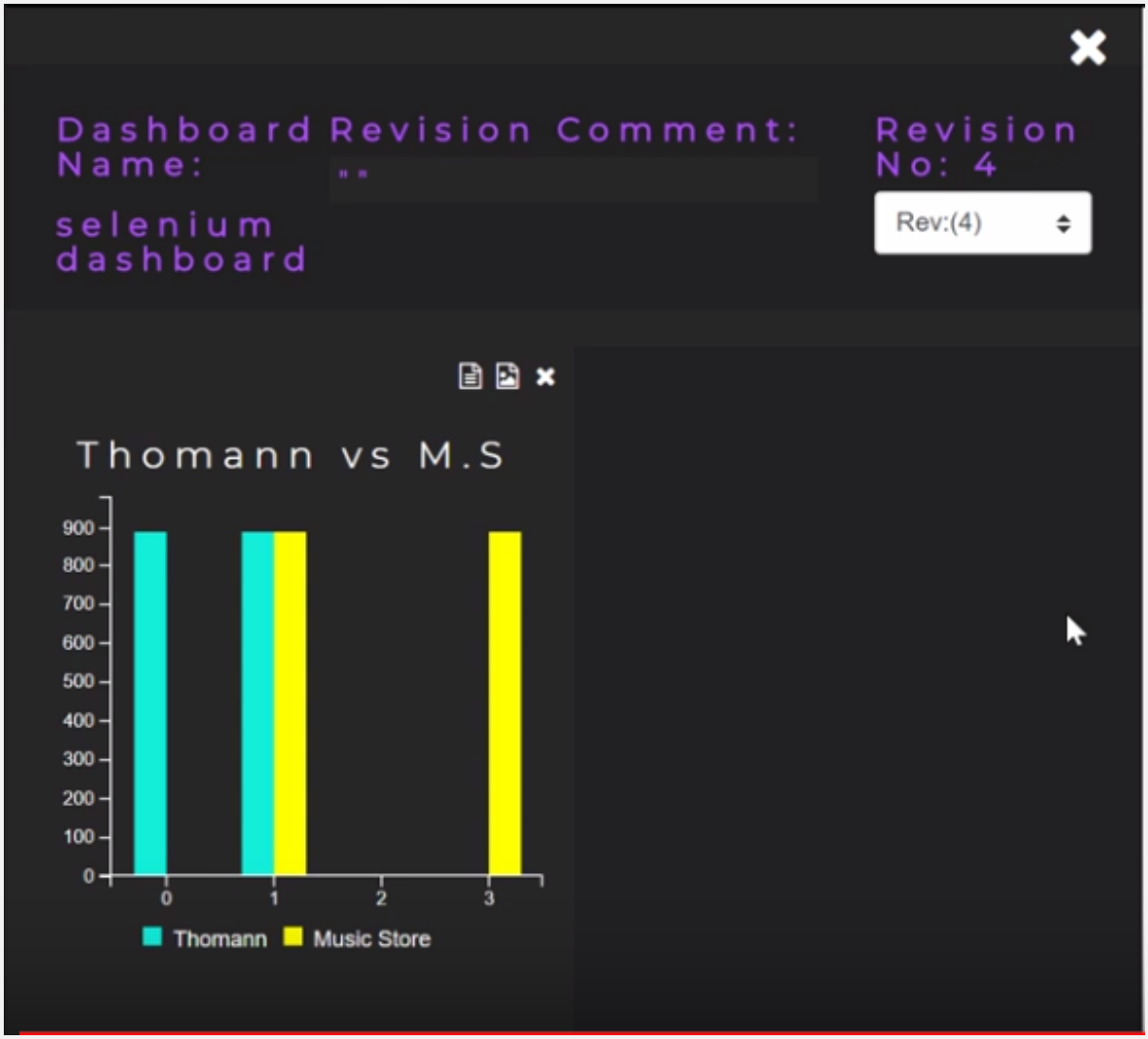
4.2.5.2 Linkage with Iris

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.



A screenshot of the Selenium agent schema in Iris.

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.

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) (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

.1 Overview of Code Repository

All of the code developed in ths project is availalbe on bitbucket. Following common convewntions a seperate repository is used for the Irrs and for each of the separate agents. These are

itemize

iris repository

agents repository

Due to the amount of code developed it is inpracticable to include in this report. So this overview will be limited to high level summary. Figure_COUNT give a breakdown of code by lauguage, using the colc utility. Third paryty code was exclused from this count. Since the various components of the systems wer edeveloped using different languages the breakdown by language slosely follows the breadown by component.

THis breakdown closely follows the seperate components of the sysysm were developed see table 2

description

item

dashboards ¡-¿ javascript + grales

iris ¡-¿ grovey

grovey Services 333 Domains 3333 Controlers 333 test 4444 3333