Dynamic Performance Framework

Iris

Status Report (Semester 2)

Dean Gaffney

20067423 Panel 3

Supervisor: Dr. Kieran Murphy

BSc (Hons) in Entertainment Systems

Contents

List of Tables

List of Figures

Abstract

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

Iris provides users with a dynamic performance framework which allows users to fully customise and centralise their application performance monitoring. This is achieved through a web interface where a user specifies a schema for a specific application they wish to monitor. Once a schema has been set up, a REST endpoint is generated for the application. This endpoint allows a user to send their monitoring data from their desired application to the framework in the form of JSON (matching the specified schema). Iris also contains features which allow a user to monitor and analyse incoming data, using an intelligent, fully customisable graph and dashboard builder. Iris then visualises any received data in real time to appropriate dashboards using websockets.

Introduction

2.1 Motivation for Iris

The following section is identical to the Motivation for Iris section in the Semester 1 report. It has been added here for convenience.

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 aggregations 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 allows 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

---

[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.

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 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.
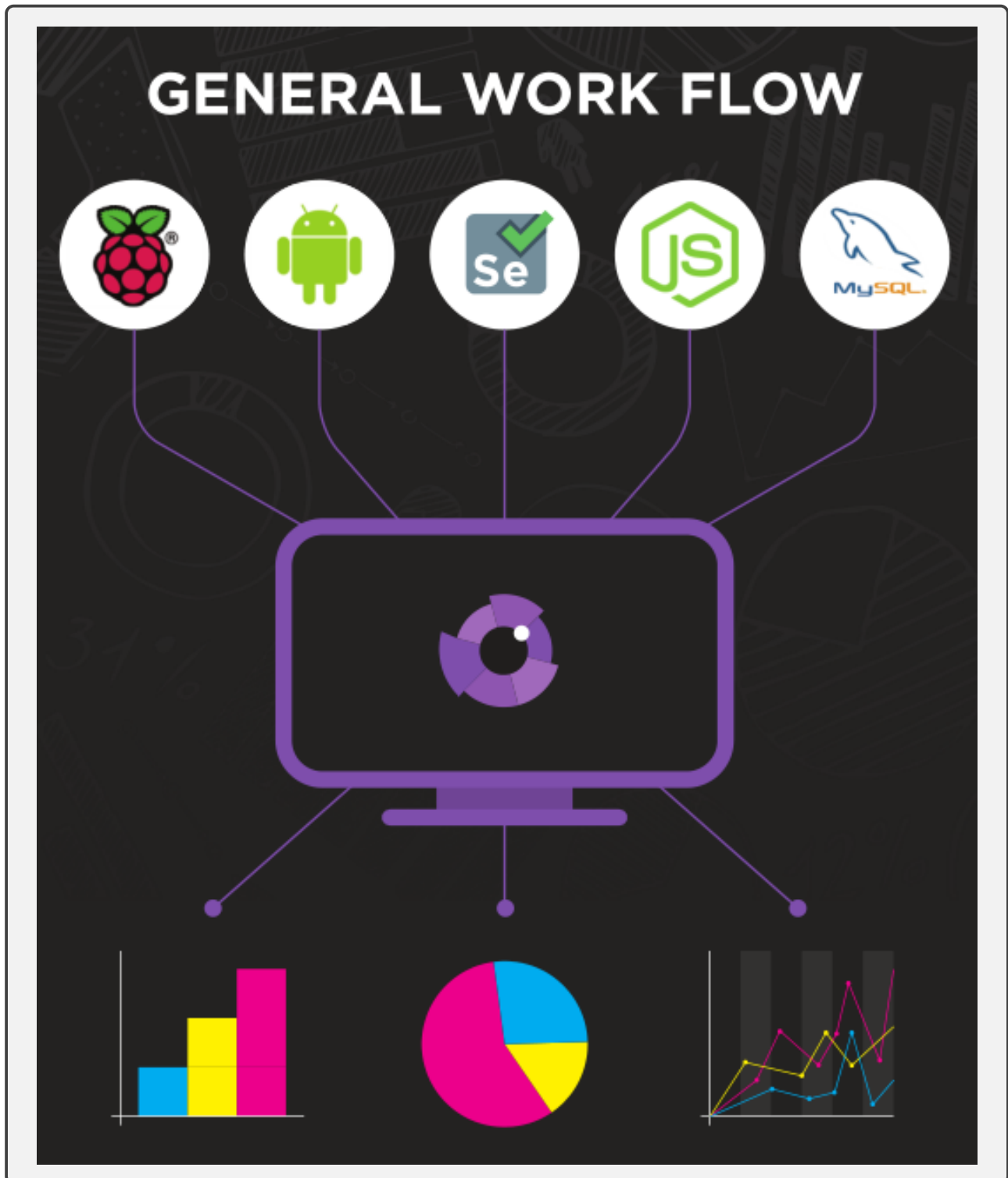
Figure 2.1 – High level view of the Iris system.
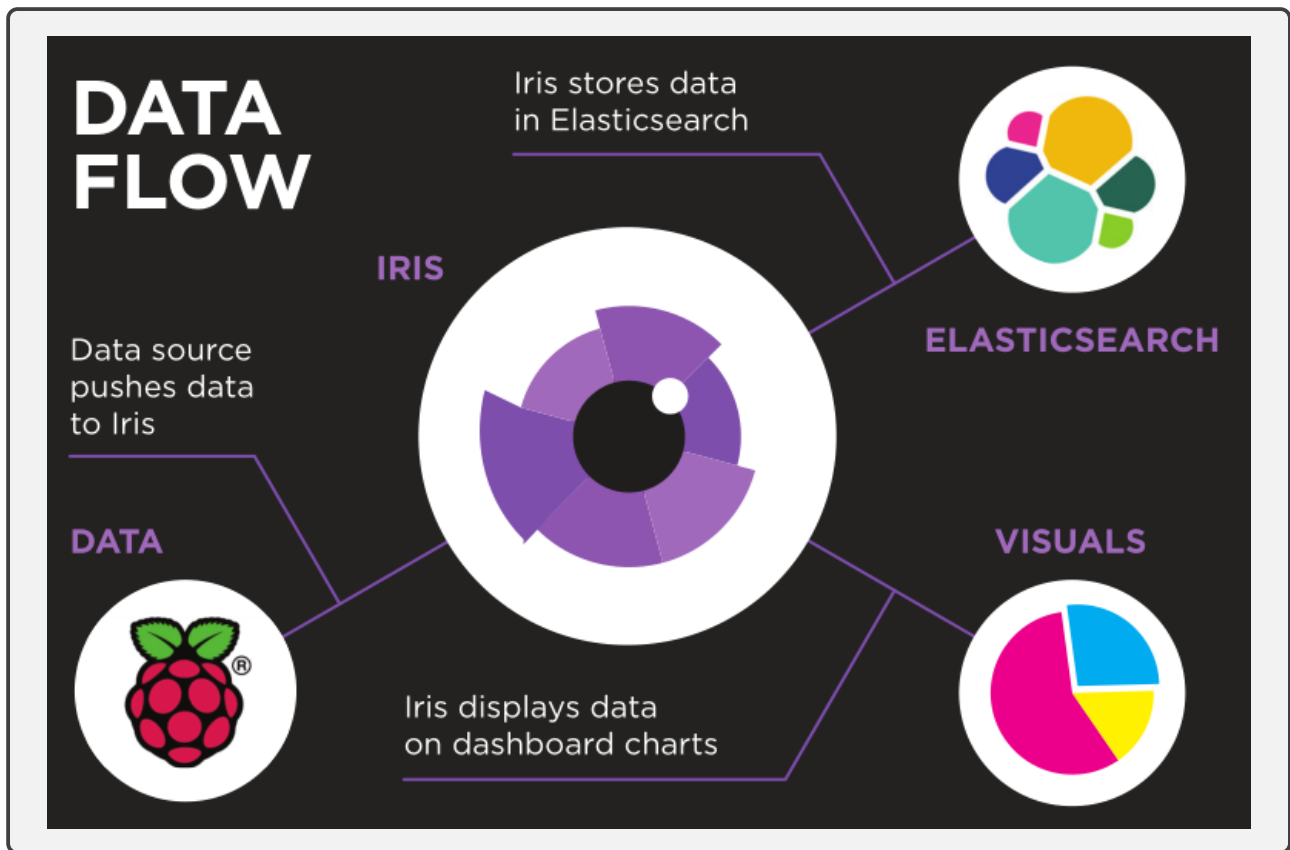
Figure 2.2 – High level view of the data flow in Iris.

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

2.2 Features List

The features of Iris follow closely to that of Kibana. Iris adds some extra features which Kibana does not support, these are as follows:

- Real-Time Monitoring

- Elasticsearch Aggregation Builder

- Automatic Elasticsearch Mapping

- Elasticsearch Aggregation Playground

- Centralised Data Transformation

- User Specific Dashboards

- Dashboard Builder

- Dashboard Revision History

- Unique Application Endpoint Generation

2.2.1   Types of Data

Iris currently supports several types of data, they are as follows:

- Numerical
- Categorical
- Raw

Support for the different types of data are demonstrated on dashboard charts in section 4.2.

Implementation

During Semester 1 a number of implementations were designed and tested. As a result the implementation described in the Semester 1 report required no further changes in this Semester. Hence, this section is fundamentally unchanged from that in the Semester 1 report and the summary of the implementation design is reproduced here with some minor modifications. The subsections dealing with components implemented in Semester 2 is fundamentally new. Table 3.1 lists the components and when they were completed.

Table 3.1 – Summary of component completion.

Component	Prototyped	Implementation
Core Framework/Services	Semester 1	Semester 1
Schemas	Semester 1	Semester 2
Dashboard	Semester 1	Semester 2
Data Sources	Semester 2	Semester 2
Security	Semester 1	Semester 2

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 acts 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 consists of field names and corresponding data types specific to the application. Iris

uses the schema to know what data to expect from the user. Once a schema is in place, Iris generates a unique endpoint associated with the schema, this unique endpoint is made available to the user as a means of sending data to Iris. Data sent to the schema endpoint is in JSON format which conforms 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 then expects a json object to come back in the form:
{
        "writeSpeed": 3000,
        "tableName":  "students"

}
```

Figure 3.1 – Example schema created for Iris.

Iris takes the users data and creates data mappings (Elastic.co 2017c) inside Elasticsearch, as well as inserts 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 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:

```json
{
        "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:

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

Figure 3.2 – Iris transforming data.

To aid performance monitoring, Iris allows 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 is available in order to allow a user to test their data and get results back immediately, this will help a user to plan their dashboard charts before setting them up. The playground also allows a user to chain several aggregations together which allows 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.

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.

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.
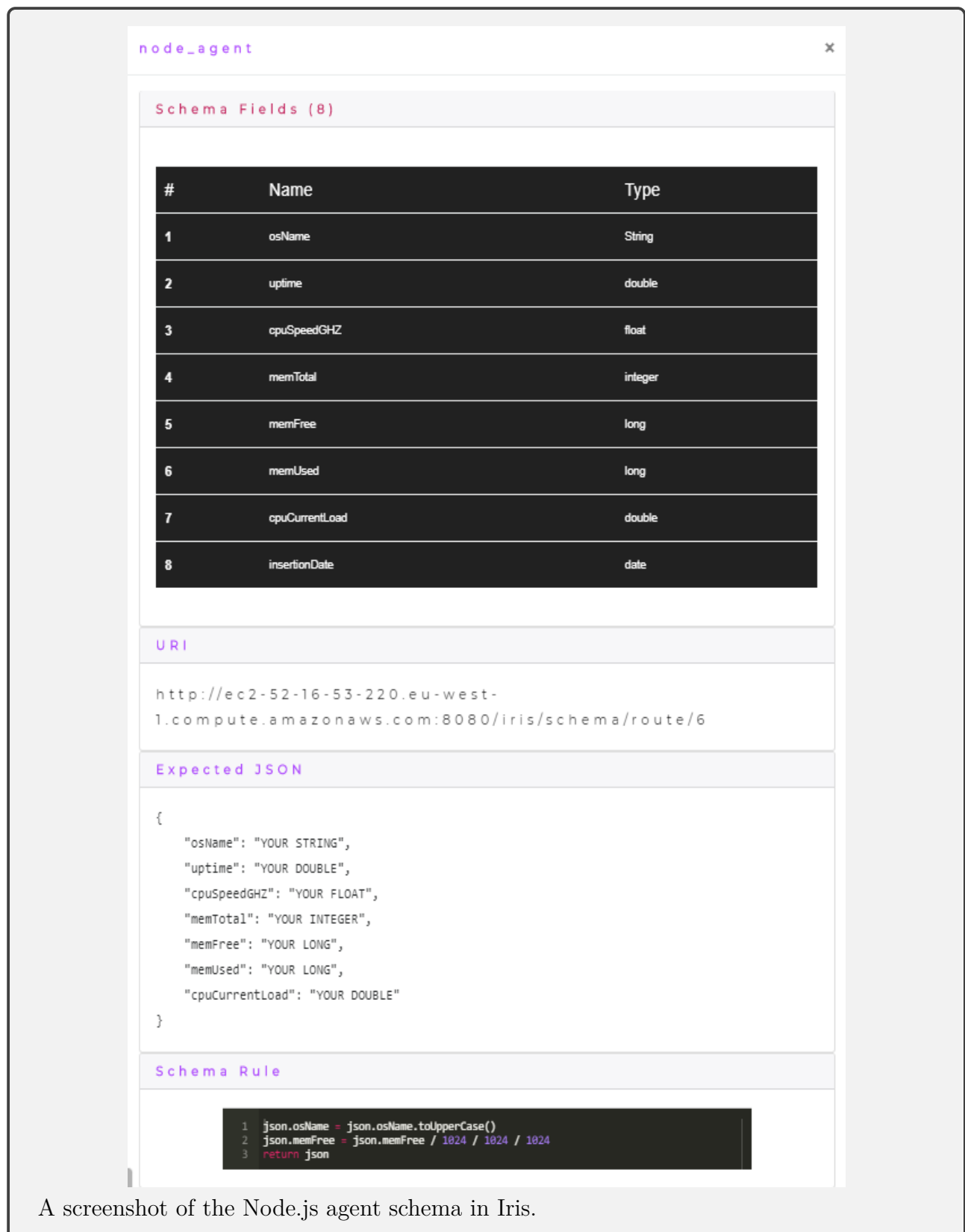
Figure 3.7 – Node.js agent schema in Iris.

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.
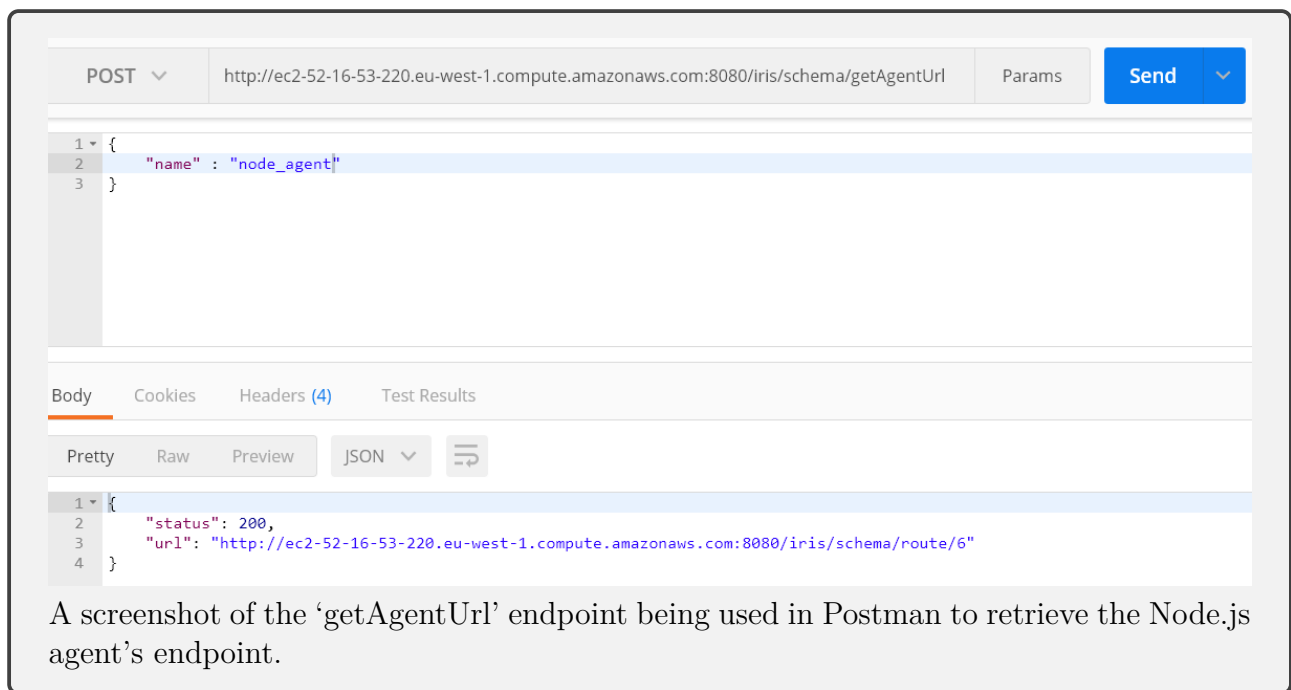
Figure 3.9 – SchemaController 'getAgentUrl' endpoint in Postman.

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

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)

```javascript
        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.

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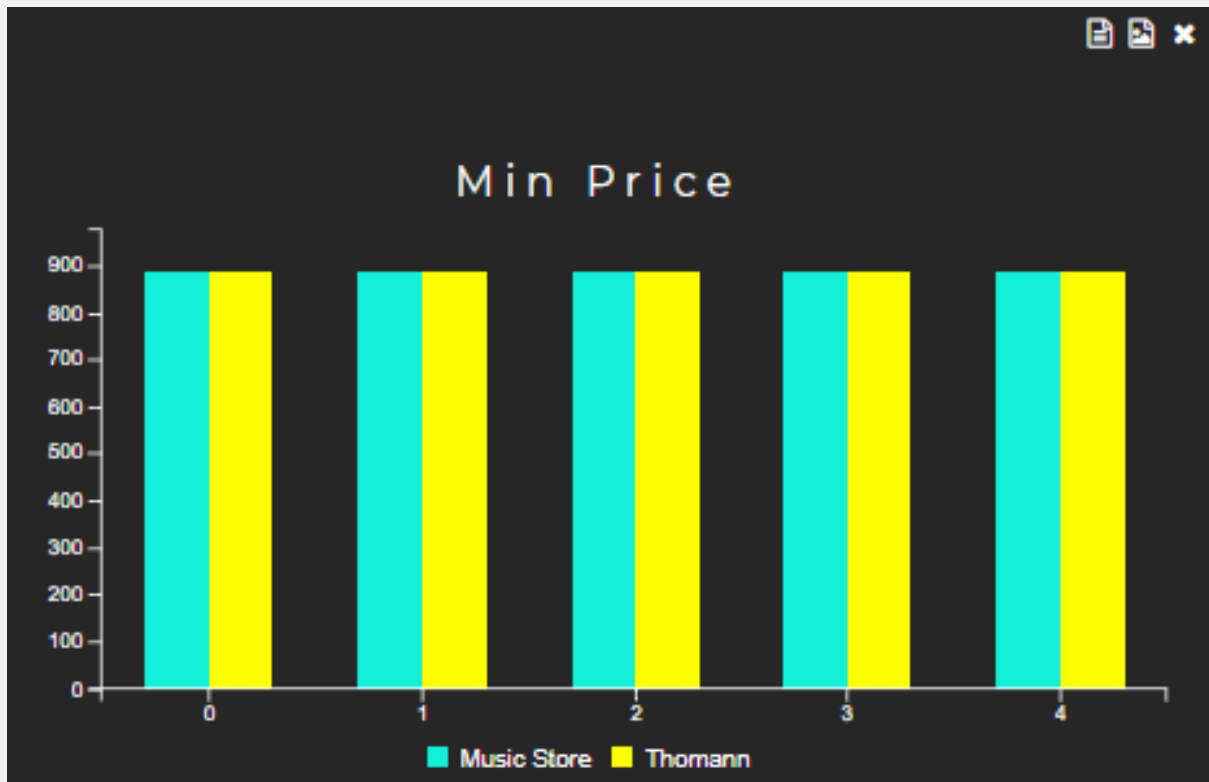
A screenshot of a downloadable chart in Iris, taken from the 'selenium agent' dashboard in Iris. The file icon in the top right is clicked to start the download.

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}]}]
```
JSON file data downloaded from a chart on the 'selenium agent' dashboard.(Shown in Figure 3.12)

Figure 3.13 – JSON file data downloaded from a chart.

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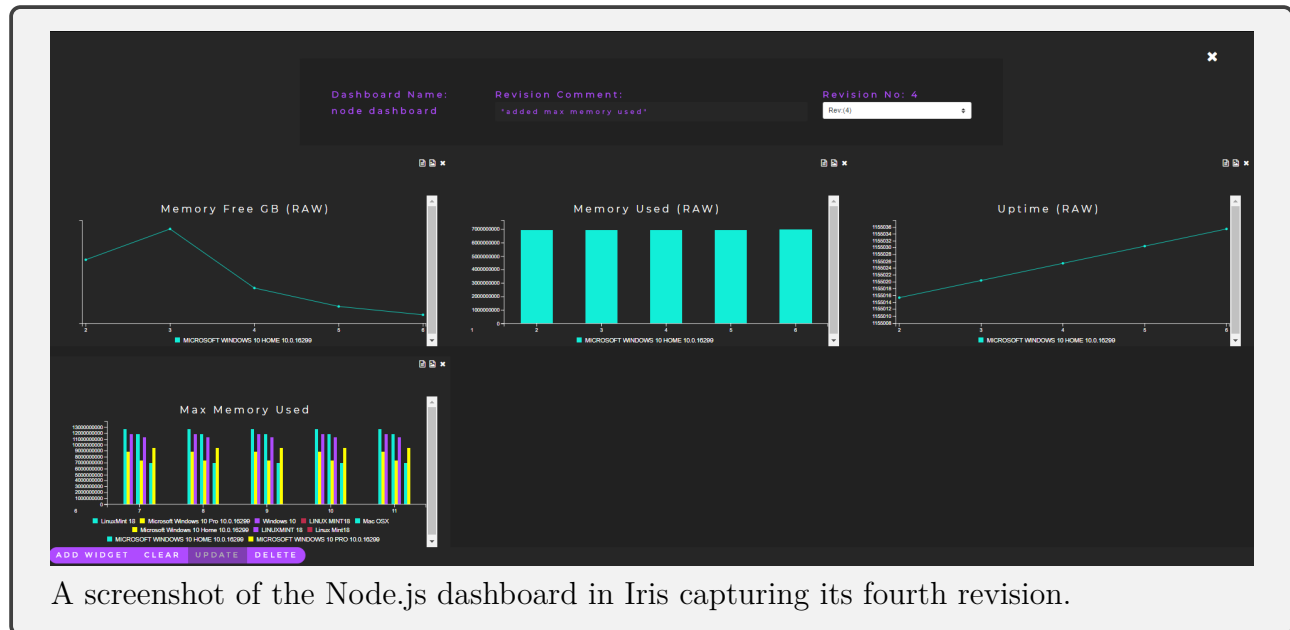
A screenshot of the Node.js dashboard in Iris capturing its fourth revision.

Figure 3.14 – Node.js dashboard revision No. 4.
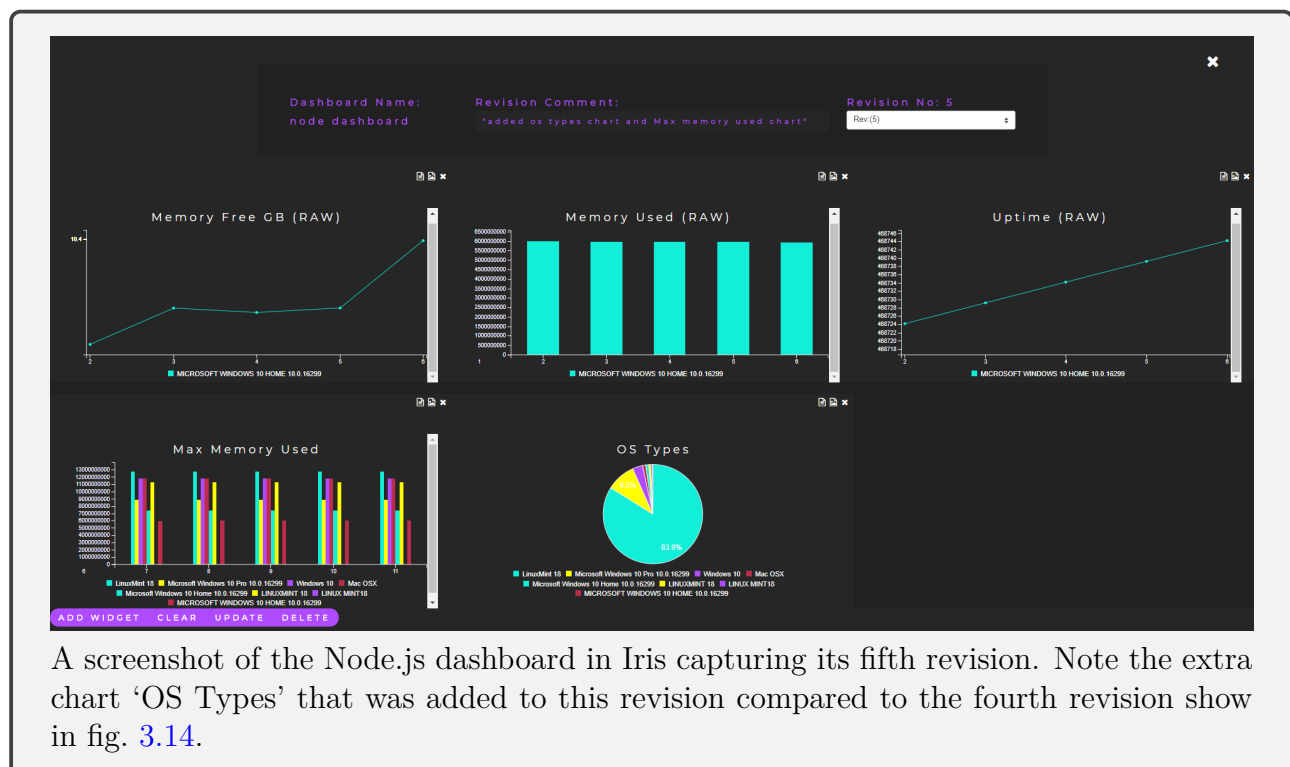
A screenshot of the Node.js dashboard in Iris capturing its fifth revision. Note the extra chart 'OS Types' that was added to this revision compared to the fourth revision show in fig. 3.14.

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.

```groovy
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.

```groovy
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.
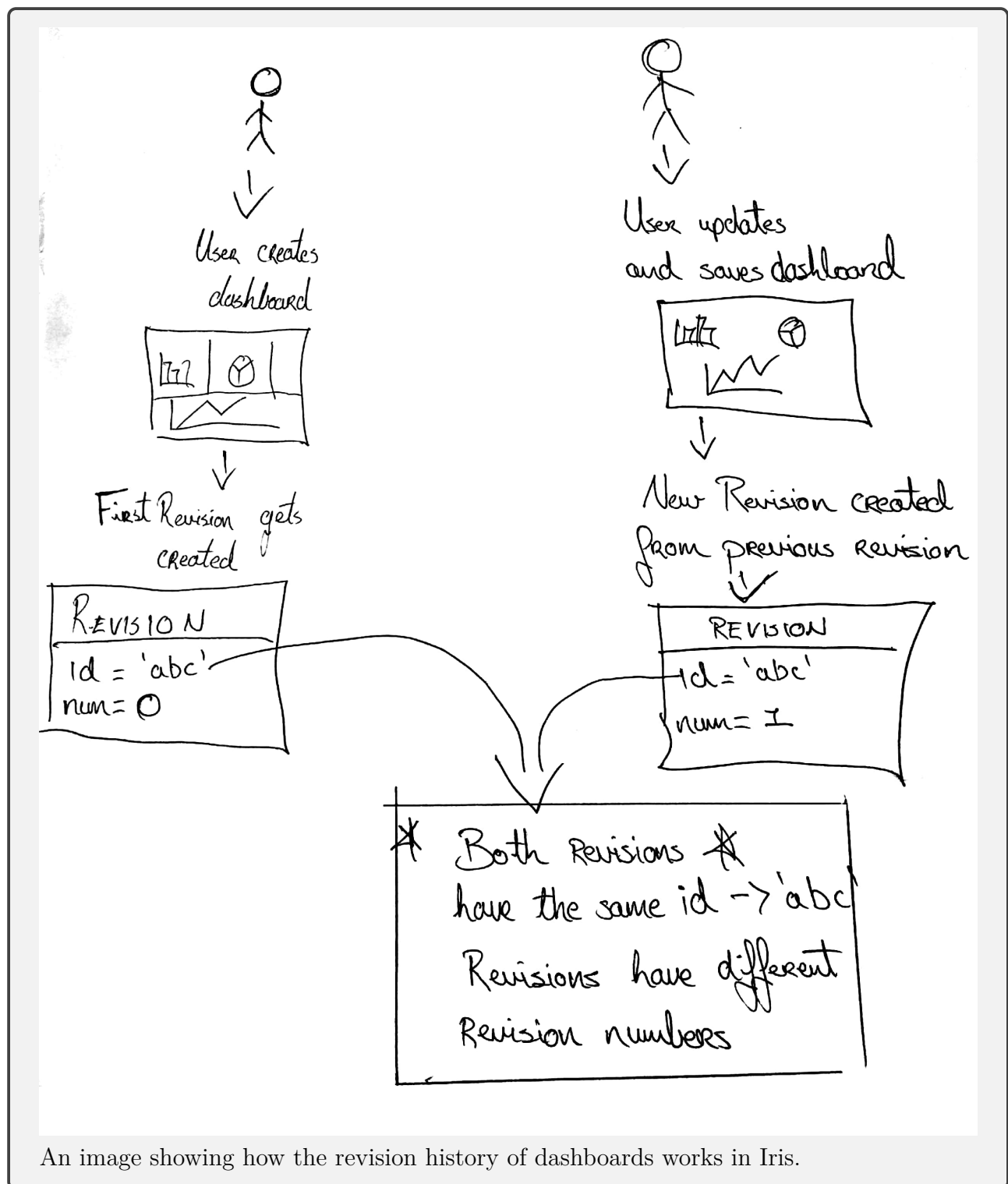
Figure 3.18 – Iris dashboard revision diagram.

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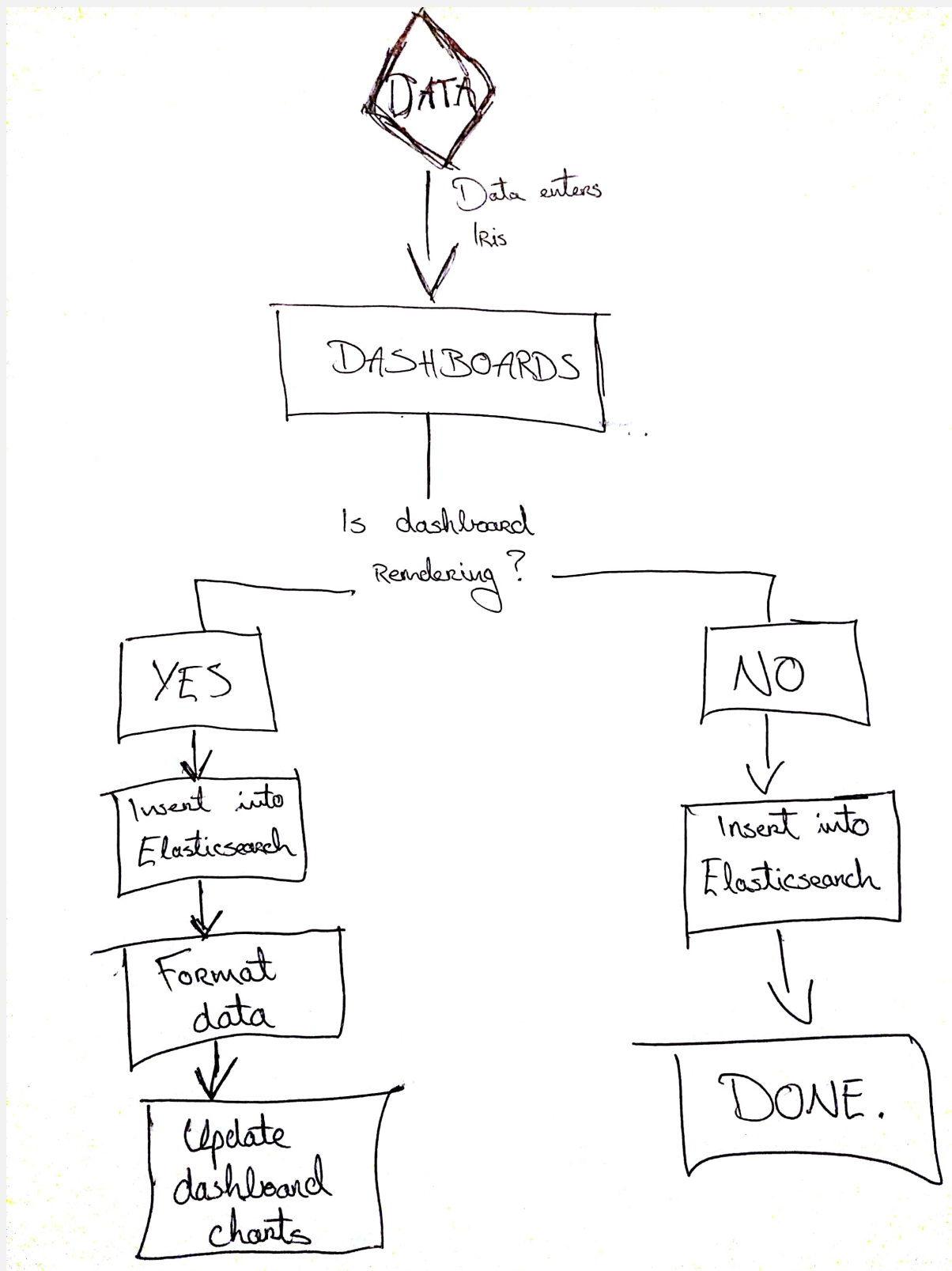
A flow chart showing how Iris handles dashboards that are rendering versus dashboards that are not rendering.

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.
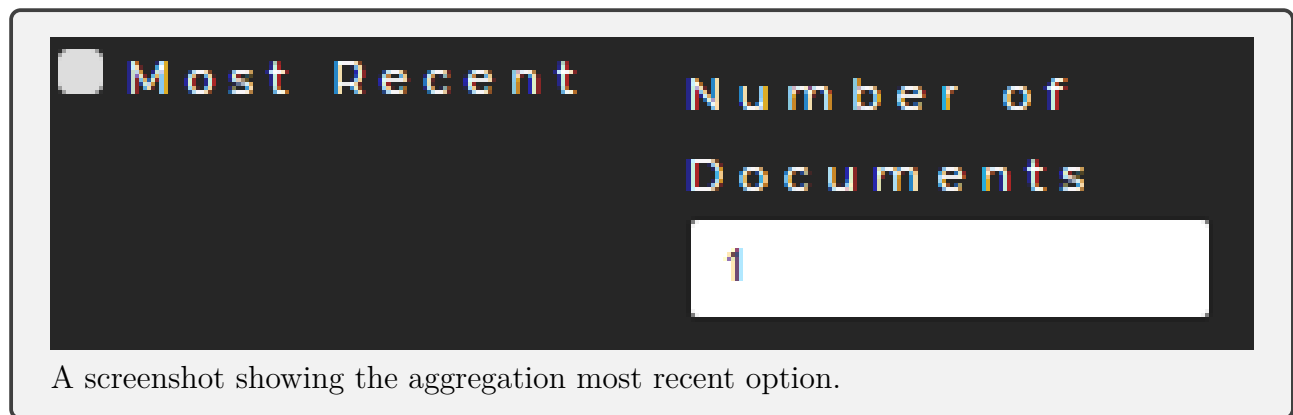
Figure 3.21 – Iris aggregation most recent option.

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.

```
Current Aggregation:
    {
        "aggs": {
            "aggs_1": {
                "terms": {
                    "field": "name"
                }
            }
        },
        "size": 0
    }
```

```
Current Aggregation:
    {
        "aggs": {
            "aggs_1": {
                "terms": {
                    "field": "name"
                },
                "aggs": {
                    "aggs_2": {
                        "max": {
                            "field": "price_eur"
                        }
                    }
                }
            }
        },
        "size": 0
    }
```

(b) Current aggregation after nesting an aggregation inside the previous aggregation.
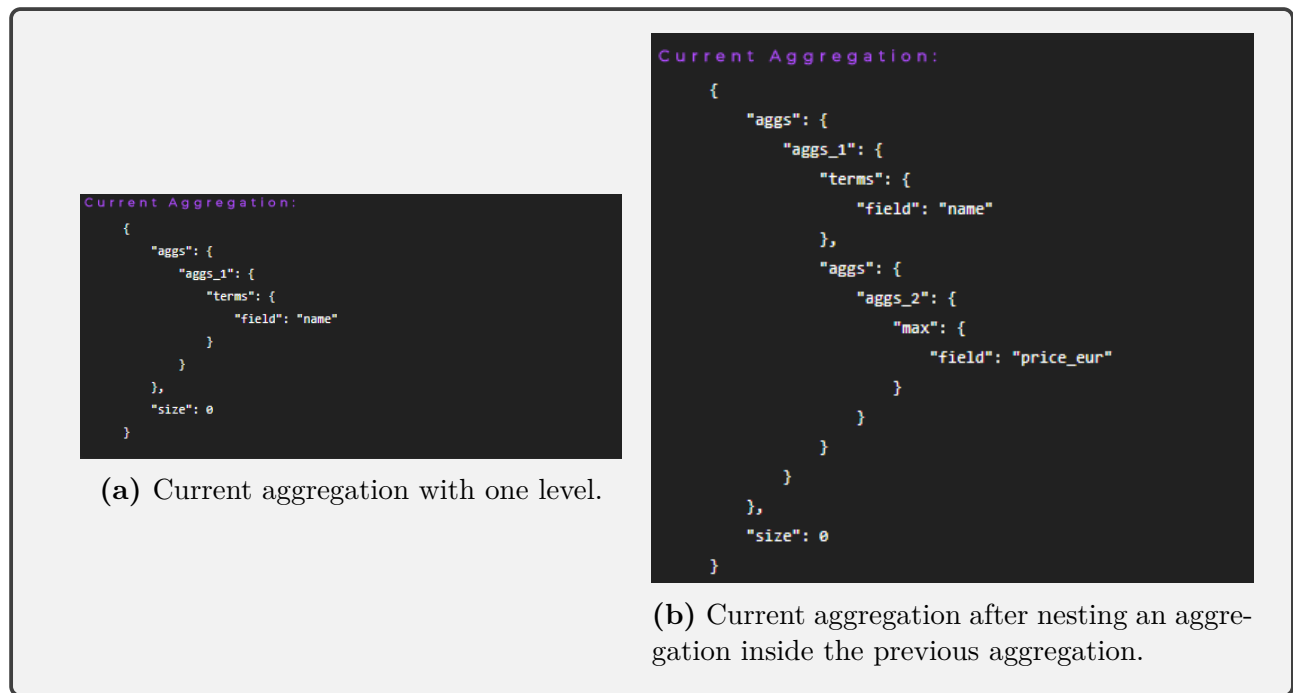
Figure 3.22 – Iris aggregation builder preview.

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.

```
Result
{
    "_shards": {
        "total": 5,
        "failed": 0,
        "successful": 5
    },
    "hits": {
        "hits": [],
        "total": 109,
        "max_score": 0
    },
    "took": 15,
    "timed_out": false,
    "aggregations": {
        "aggs_1": {
            "doc_count_error_upper_bound": 0,
            "sum_other_doc_count": 0,
            "buckets": [
                {
                    "doc_count": 27,
                    "key": "information_schema"
                },
                {
                    "doc_count": 27,
                    "key": "iris"
                },
                {
                    "doc_count": 27,
                    "key": "mysql"
                },
                {
                    "doc_count": 27,
                    "key": "performance_schema"
                },
                {
                    "doc_count": 1,
                    "key": "sys"
                }
            ]
        }
    }
}
```

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 discusses 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 example.

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 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 and need 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 file.

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 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.

Deployment, Testing and Evaluation

In this chapter the procedure to deploy, test and evaluate Iris is discussed. In the Semester 1 report the deployment was not taken into consideration meaning the deployment section is fundamentally new. The testing and evaluation of data sources is new in regards to the data sources being created and tested against Iris, however they were briefly discussed in the Semester 1 report in regards to what their overall interaction with Iris would be. The details of what data the data sources would send and what platform they would use to send the data were not discussed.

4.1 Deployment

In this section the procedure of deploying Iris will be discussed in terms of how the application is is built and served as well as how it interacts with Elasticsearch and MySQL instances while in production.

4.1.1 AWS (Amazon Web Services)

Amazon Web Services is used to host Iris as well as host the Elasticsearch and MySQL processes which Iris interacts with, these will be discussed in the following subsections.

4.1.1.1 EC2 Instance

Iris is hosted on an AWS EC2 instance. The instance is part of the free tier options meaning the instance is limited in regards to its hardware specifications which can be found in table 4.1. The EC2 instance is configured to allow incoming HTTP requests on port 8080 so that users can send requests to Iris as well as receive responses from Iris. The EC2 instance also runs the MySQL process where Iris stores its user related data which will be discussed in section 4.1.1.4.

Table 4.1 – AWS EC2 instance specifications.

Model	vCPU	Memory	Storage
t2.micro	1	1GB	8GB

For more information on the EC2 instance types refer to this url https://aws.amazon.com/ec2/instance-types/.

4.1.1.2   Jetty

Jetty is an open source HTTP server that can be used to act as web application container (Eclipse.org 2018). Jetty was not the first choice for being the container for Iris. Tomcat was the original choice as Onaware have their java web applications in Tomcat instances, this meant Iris was also going to be a candidate for being put in a Tomcat container. However due to the limitations of the EC2 instance mentioned in table 4.1, Tomcat proved to be unstable and was running out of memory for the JVM (Java Virtual Machine) once it was launched.

Due to Tomcats' instability, a different web application container was required. Several other containers were tested including GlassFish and JBoss, but these containers also failed to deploy Iris due to limited memory. Jetty was then tested and deployed Iris successfully. In order to test the deployment, the Jetty container was launched and shutdown several times over several days to ensure the container was stable. Once Jetty proved to be satisfactory Iris was deployed.

For more information on Jetty refer to this url https://www.eclipse.org/jetty/documentation/9.4.x/introduction.html.

4.1.1.3   Elasticsearch Instance

The Elasticsearch instance which Iris interacts with is hosted on AWS as part of the Elasticsearch Services offered by Amazon. The instance is part of the free tier selection but proved to have sufficient hardware specifications for Iris which are detailed in table 4.2. The instance was tested heavily by sending requests from Iris at a rapid and constant rate over a period of time. The instance did not appear to have any issues dealing with the requests and was considered suitable for Iris.

Table 4.2 – AWS Elasticsearch instance specifications.

E.S Version	Documents	Storage
5.5	46,590	7.22GB

4.1.1.4   MySQL Instance

The MySQL instance that Iris uses to store user specific data such as schemas, dashboards etc. is run on the EC2 mentioned in section 4.1.1.1. This allowed Iris to configure its database connector to use localhost, which made development and production environments easier to test as the database configuration was the same.

4.2   Testing and Evaluation — Data Sources

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

4.2.1   Android App

4.2.1.1   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.
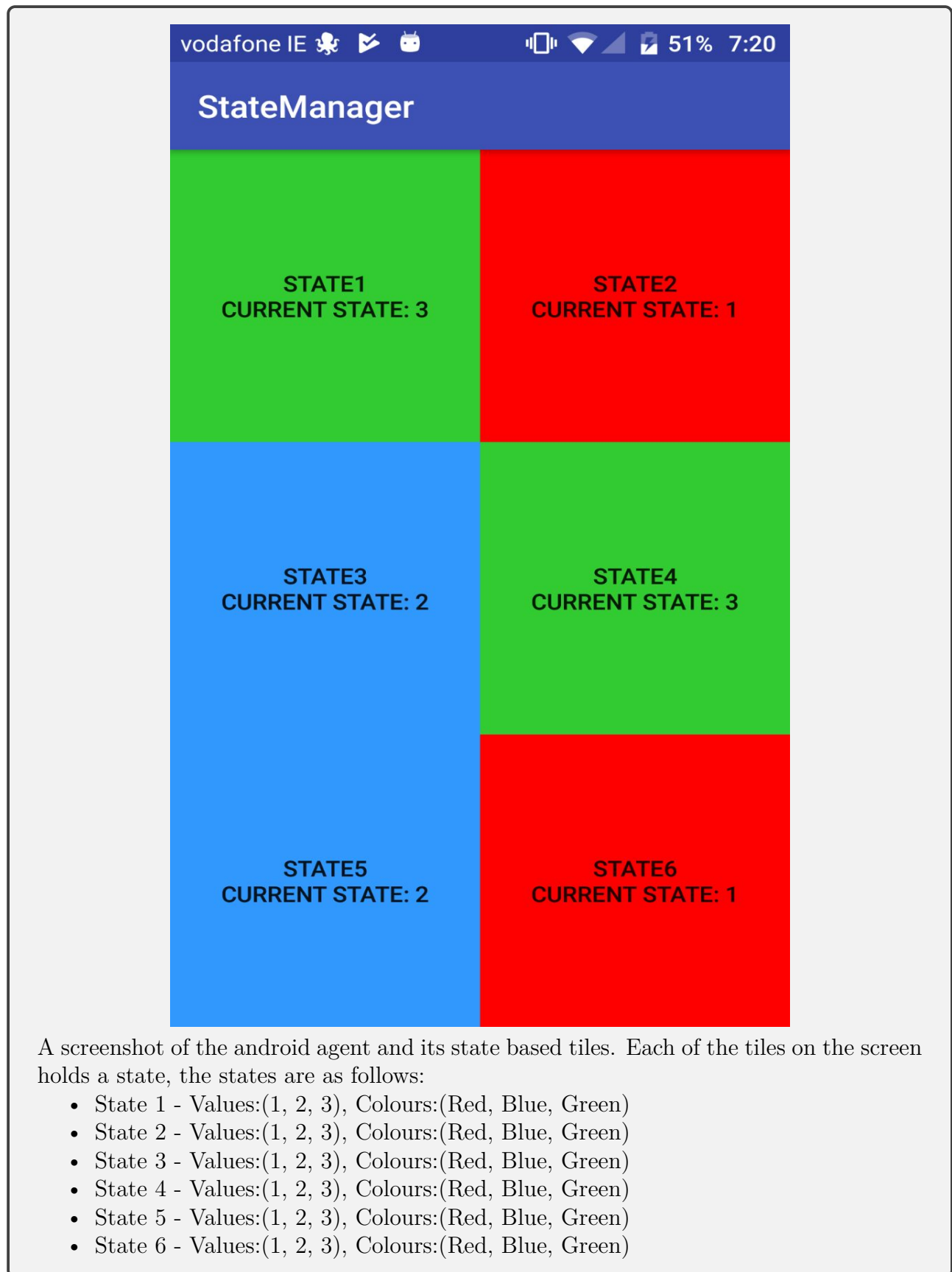
A screenshot of the android agent and its state based tiles. Each of the tiles on the screen holds a state, the states are as follows:

- 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.

4.2.1.2   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.

4.2.2   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.

4.2.2.1   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/

4.2.2.2   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.

crypto_agent   ✕

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.

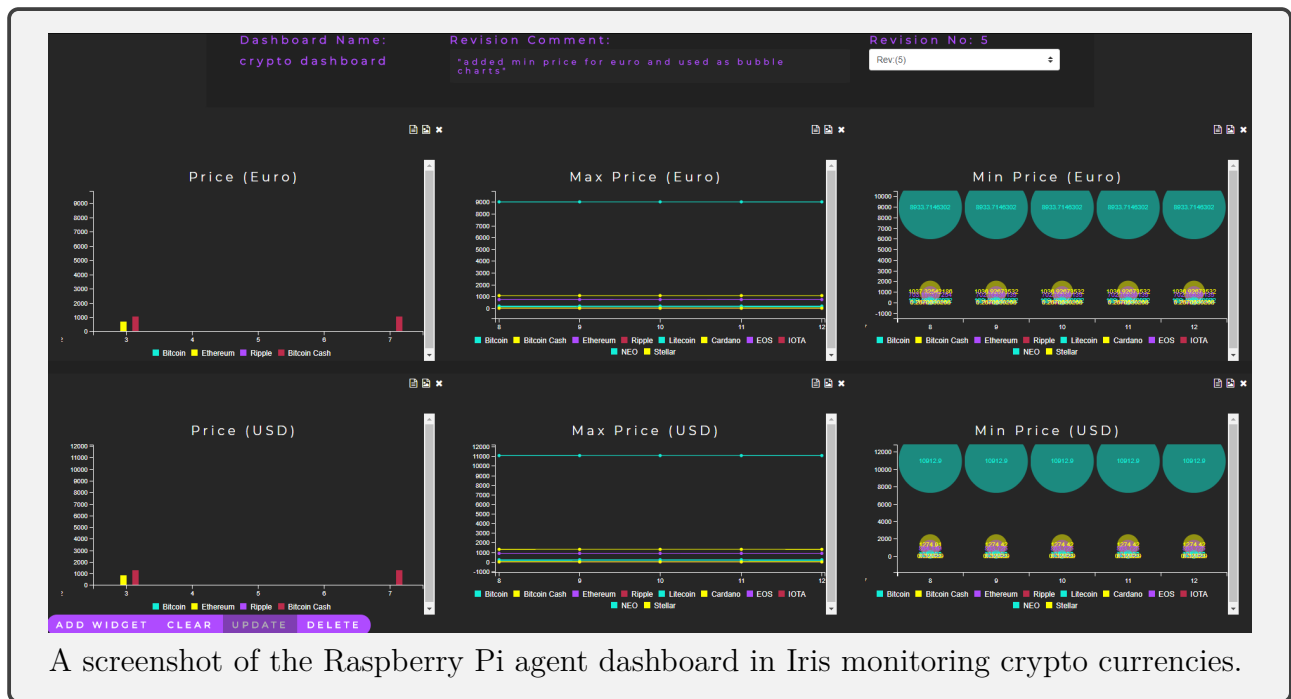**Figure 4.4** – Raspberry Pi agent schema in Iris.

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

Figure 4.5 – Raspberry Pi agent dashboard in Iris.

4.2.3 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'.

4.2.3.1 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.
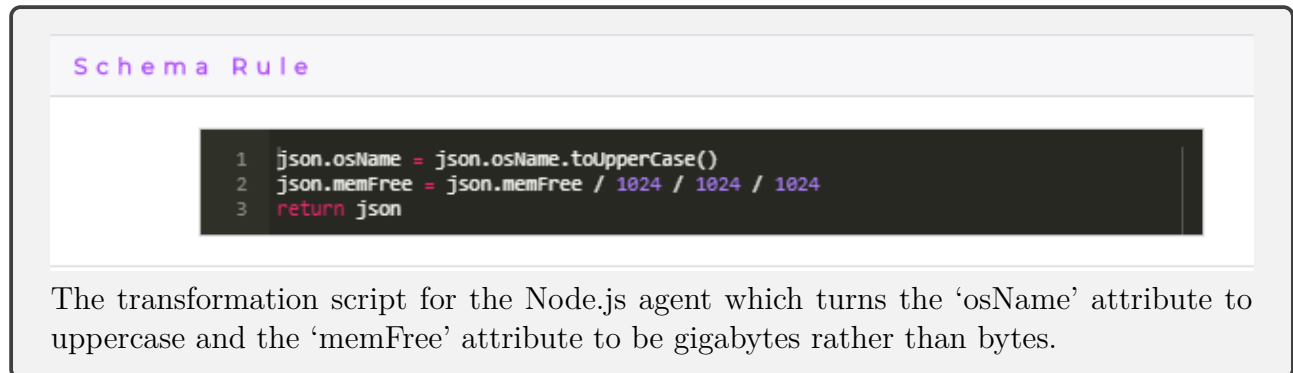


```
Schema Rule

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.

4.2.3.2   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.

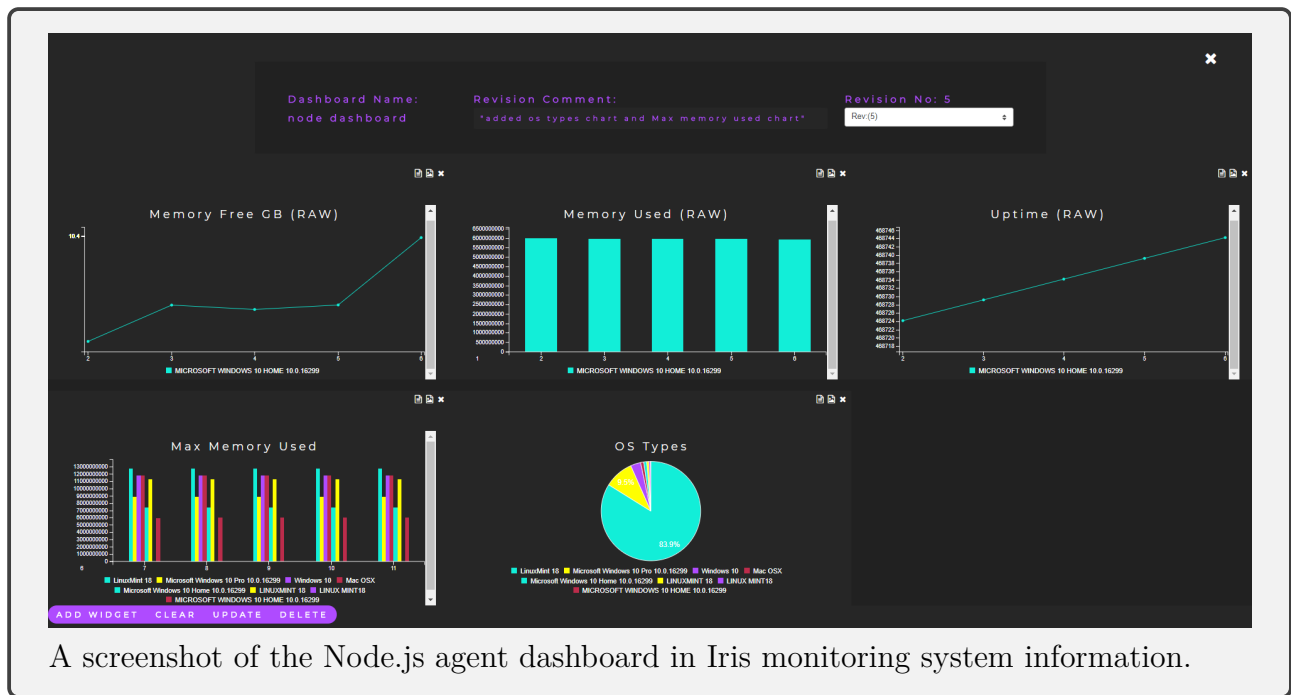**Figure 4.7** – Node.js agent schema in Iris.

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

Figure 4.8 – Node.js agent dashboard in Iris.

4.2.4   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.

4.2.4.1   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/

4.2.4.2   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.

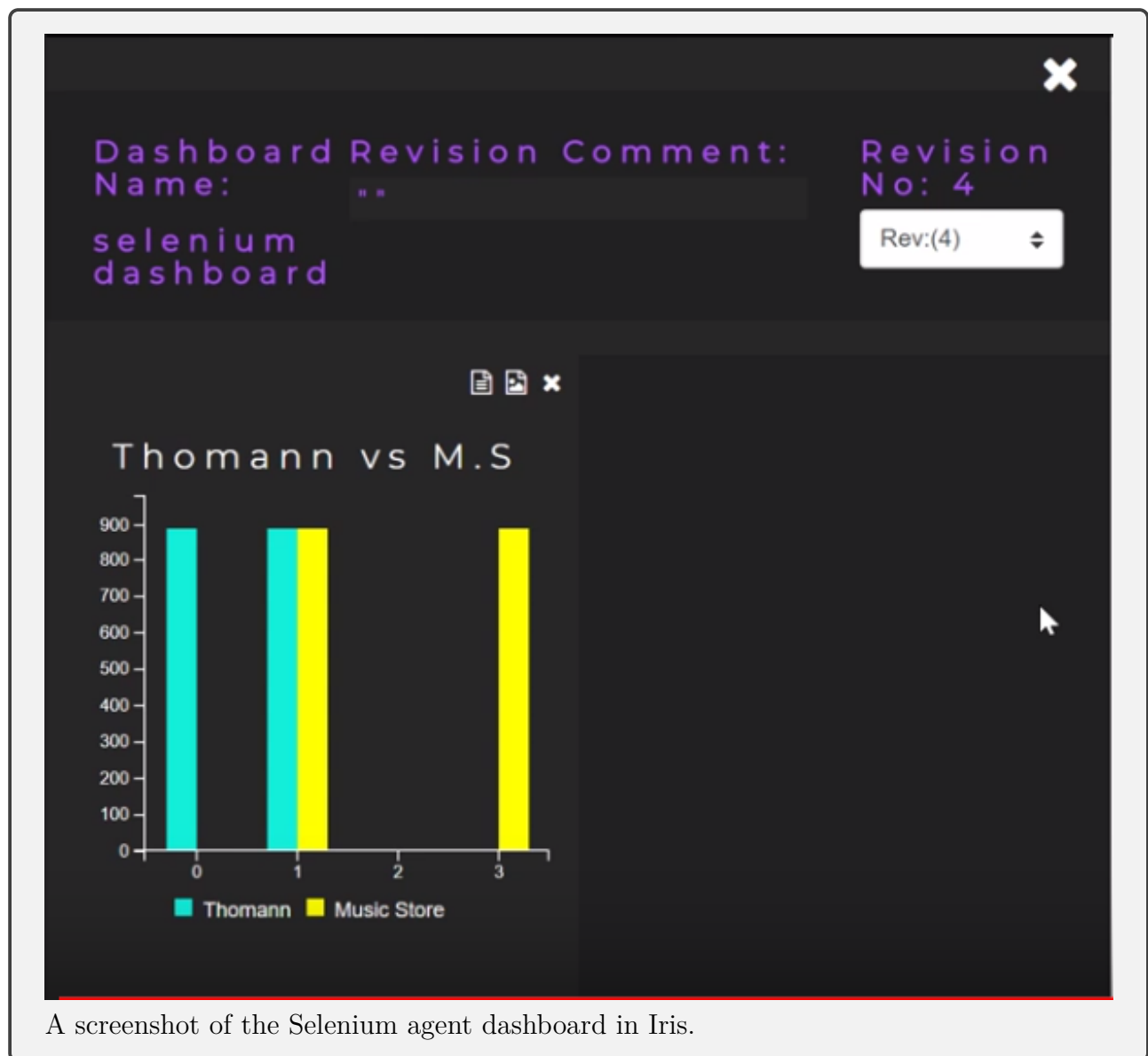A screenshot of the Selenium agent dashboard in Iris.

Figure 4.10 – Selenium agent dashboard in Iris.

4.2.5   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.

4.2.5.1   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

4.2.5.2   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.
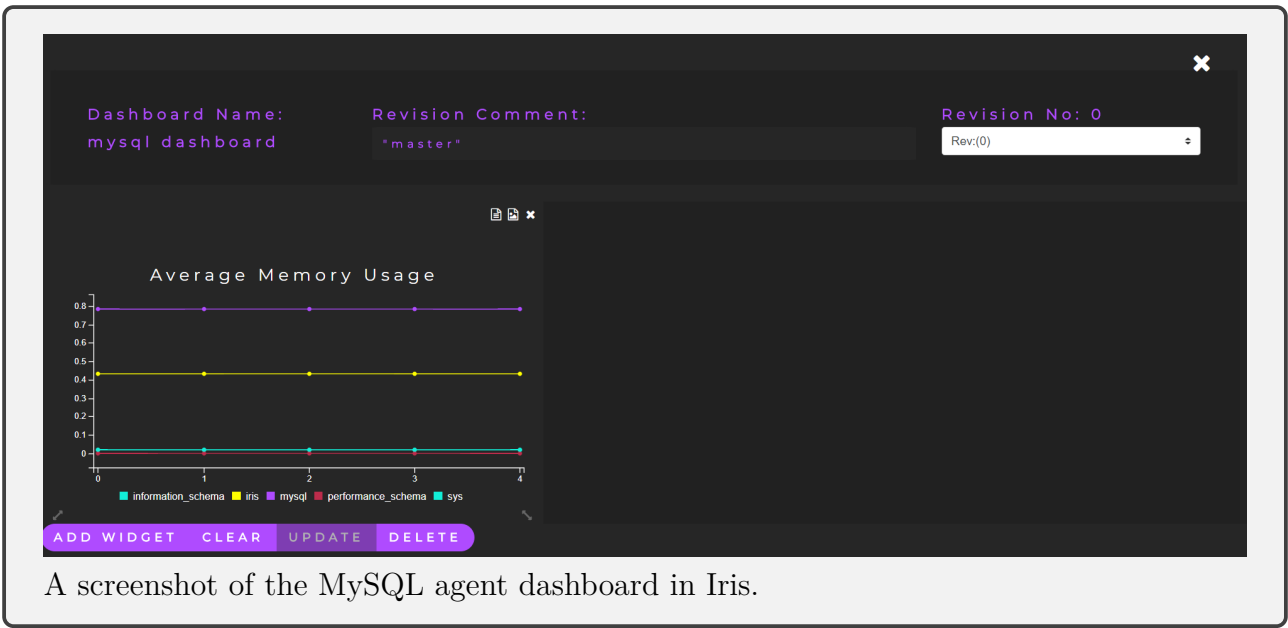
Figure 4.13 – MySQL agent dashboard in Iris.

Conclusions

5.1 Current Application State

Currently Iris is in a stable state and has achieved what it was designed to do dynamic data monitoring. In the Semester 1 report the design and features of the Iris core framework are discussed in regards to schemas, dashboards and Iris' use of Elasticsearch. All of these features were implemented successfully and collaborate to deliver a fully dynamic data monitoring implementation.

Iris demonstrates its capabilities in live production on AWS where it monitors data from several data sources which are active on different platforms and push of different forms of data to Iris. Iris handles setting up an application schema for the data sources and handles the incoming data being pushed from these data sources. Iris allows users to track their data through dashboards and Elasticsearch aggregations in realtime which helps verify Iris' current state in this lifecycle.

5.2 Future Development

The goal of this lifecycle was to implement the core framework of Iris which Onaware could work with and expand upon. Throughout the development of Iris a potential features list has been growing within Onaware. Below is a summary of some of the features which are being considered for Iris:

- Storing unmodified data. Currently when a user writes a transformation script to transform the data, the transformed data is being stored in Elasticsearch. It would be better to store the raw unmodified data as well as the modified data.

- Replace Elasticsearch Aggregation terminology with more user friendly terminology, making it easier for anyone to build Elasticsearch aggregations.

- Add a State Disc Chart list. This would be a list of state disc charts inside one widget.

- Add a dashboard widget which simply displays text, this can be useful for dumping programming exceptions or to send log file contents into a widget on a dashboard.

- Chart merging. Allow a user to merge two or more charts to aid their data visualisation.

- Create a dedicated area for downloading and uploading common applications for pushing data to Iris, this would be similar to a marketplace.

The features listed above will be discussed in more detail with Onaware once Iris starts its new lifecycle within Onaware.

5.3   Alternative Approaches

The main alternative approach which was discussed for building Iris was the idea of using Node.js rather than Grails 3.

Iris was built using Grails[1], a technology which appears to losing popularity as a means for building web applications. This became evident relatively quickly in the research stage of Iris as there was difficulty finding appropriate plugins and libraries for Grails 3 (the newest release of Grails) that are essential for Iris. The reason Grails 3 is being discussed and not its prior releases is because it shows how Grails is beginning to decline in popularity. For example there is 1,255 plugins for Grails versions 1&2, but currently there is only 239 plugins for Grails 3, which can be seen here http://plugins.grails.org/. Due to the lack of community support for Grails 3, using a different technology such as Node.js would have benefited Iris. Node.js which uses the 'npm' package manager contains over 650,000 packages, this would have decreased the development time of Iris dramatically as a lot of Iris' code had to be written from scratch when using Grails.

Another reason for considering Node.js for Iris, was due to Iris needing to monitor data in realtime. Although Iris currently doesn't struggle with this operation, Node.js' I/O architecture is designed to be non-blocking, which allows requests to be dealt with asynchronously, meaning Iris can handle incoming data requests asynchronously.

Another reason for considering Node.js for Iris is the fact that the majority of requests sent from the client to Iris are in the form of JSON. Once the JSON request is recieved by Iris it needs to be parsed into Groovy code. Groovy has made excellent libraries for parsing JSON but it would be more ideal if possible to work with the same languages both frontend and backend if possible. Node.js makes this possible by having the server code in javascript.

Lastly, Iris could benefit from using a more efficient way of dealing with DOM manipulation, jQuery is an older technology and is now starting to be replaced by frontend frameworks (Allen 2018). Using a frontend framework such as React.js would allow Iris to render dashboard charts more efficiently due to its use of a virtual DOM, as well as bind the DOM data to javascript variables (React.js 2018). React.js also uses components which add a great increase in reusability and testability for the frontend. All React.js code is written in javascript meaning the server and client can communicate in the same language if Node.js was being used for the server code. Using a frontend framework allows the frontend and backend of a project to remain loosely coupled making them both easier to work on.

---

[1]Onaware's expertise is in Java and Grails, this is the main reason why Grails was picked for Iris.

Bibliography

Allen, Ian (2018). *The Brutal Lifecycle of JavaScript Frameworks*. URL: https://stackoverflow.blog/2018/01/11/brutal-lifecycle-javascript-frameworks/ (visited on 04/04/2018) (cit. on p. 52).

Eclipse.org (2018). *Jetty*. URL: https://www.eclipse.org/jetty/ (visited on 04/04/2018) (cit. on p. 35).

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. 8).

– (2017b). *Basic Concepts*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html (visited on 11/04/2017) (cit. on p. 7).

– (2017c). *Mapping*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html (visited on 11/04/2017) (cit. on p. 7).

React.js (2018). *Virtual DOM and Internals*. URL: https://reactjs.org/docs/faq-internals.html (visited on 04/04/2018) (cit. on p. 52).

Appendices

Overview of Code Repository

All of the code developed in this project is available on bitbucket. Following common conventions a separate repository is used for Iris and for each of the separate agents. They can all be found at the following urls:

- Iris (Core) - https://bitbucket.org/DeanGaffney/iris.

- Iris-Selenium - https://bitbucket.org/DeanGaffney/iris-selenium.

- Iris-Raspberry Pi - https://bitbucket.org/DeanGaffney/iris-crypto-rates.

- Iris-MySQL - https://bitbucket.org/DeanGaffney/iris-mysql.

- Iris-Android - https://bitbucket.org/DeanGaffney/iris-android.

- Iris-Node.js - https://bitbucket.org/DeanGaffney/iris-node.

Due to the amount of code developed it is impractical to include the code within this report. This overview will be limited to a high level summary. Figure A.1 gives a breakdown of code by language using the cloc (count lines of code) utility. Third party code was excluded from this count. Since the various components of the systems were developed using different languages the breakdown by language closely follows the breakdown by component.

```
C:\Users\Gaff_000\Documents\college\year_4_modules\fyp\iris (0.0.6)
λ cloc --fullpath --not-match-d="ace|billboard|bootstrap|gridstack|jquery|lodash|vue|font-awesome|build"
 --not-match-f="gradlew|grailsw" . --exclude-ext=css,md,bat,less,json,xml,log
     403 text files.
     402 unique files.
     429 files ignored.

github.com/AlDanial/cloc v 1.74  T=2.34 s (50.4 files/s, 3474.3 lines/s)
-------------------------------------------------------------------------
Language                files          blank        comment          code
-------------------------------------------------------------------------
Groovy                     75            968            475          3272
Grails                     32            252             74          1357
JavaScript                  9            124            222           742
Sass                        1             75             56           386
YAML                        1              5              3           129
-------------------------------------------------------------------------
SUM:                      118           1424            830          5886
-------------------------------------------------------------------------
```

A screenshot of the cloc utility run on Iris, the command filters all third party code and only counts the lines of code in Iris.

Figure A.1 – Cloc results.

The breakdown shown in table A.1 closely follows the separate components of the system.

Table A.1 – Summary of component lines of code.

Component/Language	Files	Line Count
Schemas		
Groovy	10	461
Grails	5	169
Dashboards		
Groovy	3	292
Grails	9	427
Javascript	1	191
Charts		
Groovy	2	128
Javascript	4	308
Elasticsearch		
Grails	3	169
Groovy	6	245
Javascript	2	110
Integration Tests		
Groovy	14	761
Unit Tests		
Groovy	13	668
Utilities		
Groovy	18	384
Other Services		
Groovy	9	1573

Methodology

Iris issues and sprints are being tracked using Trello. These are organised using the *kanban* system and custom labels inside of Trello. Trello allows filtering of cards by their labels, meaning issues related to a particular sprint can be the only cards visible. This makes tracking sprints and issues for the sprint easy to manage.
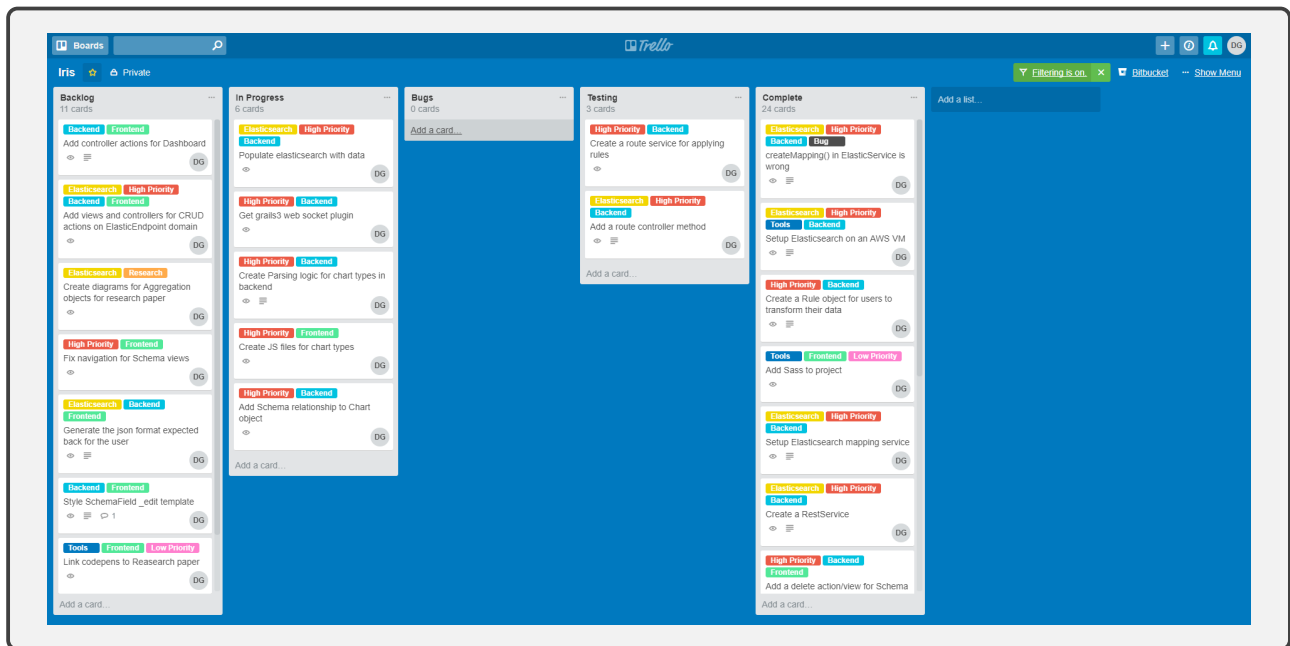

Figure B.1 – Screenshot of the Iris Trello board.

B.1 Trello

Trello was used to keep track of issues and sprints in Iris. The sprint numbers correspond to the version of Iris at the time of the sprint. Below is a list of the issues for each sprint of Iris. This

list is generated using a python script written by the author which interacts with the Trello API to retrieve the issues from the Trello database based on their sprint numbers.

Sprint 0.0.1

- Scope project with Onaware

- Draw ER Diagram for User owning Dashboards

- Research d3 wrapper libraries

- Draw diagrams for Frontend

- Setup grails 3.3.1 + groovy 2.4.11 + java 8u144 on devices

- Research Dashboard libraries for serialization

- Generate Domain classes based off UML diagram

- Look into Elasticsearch Java api and see if it can be used for building complex aggregations dynamically

- Use Grails to generate all controllers and views for Domains

- Add Spring Security Plugin for User Domain

- Build Elasticsearch Objects

- Update to latest version of jQuery

- Update to Bootstrap 4

- Add controller actions for Schema Domain

Sprint 0.0.2

- Style SchemaField edit template

- Add views and controllers for CRUD actions on ElasticEndpoint domain

- Generate the json format expected back for the user

- Fix navigation for Schema views

- Add levels to javascript aggregation logic

- Setup Selenide on all machines

- Create a route service for applying rules

- Add a route controller method

- Populate elasticsearch with data

- Get grails3 web socket plugin

- Link codepens to Research paper

- createMapping() in ElasticService is wrong

- Create diagrams for Aggregation objects for research paper

- Add Schema relationship to Chart object

- Setup Elasticsearch on an AWS VM

- Create a Rule object for users to transform their data

- Add Sass to project

- Setup Elasticsearch mapping service

- Create a RestService

- Add a delete action/view for Schema domain

- Deleting Elasticsearch index is not working from front end

- Create ElasticEndpoint domain

- Create a Domain for storing Elasticsearch endpoint addresses

- Research updating Elasticsearch mappings

- Move over to branch 0.0.2

- Add Logging to app

- Add controller actions for Dashboard

Sprint 0.0.3

- Display aggregation execution result in playground

- Schema level counter in JS should match a DOM element to keep it more dynamic

- Iris demo presentation

- Write gradle task to kill geckodriver.exe

- Try adding underscore.js to project to stop javascript error

- Make aggregations executable in Aggregation Playground

- Create param link for Schema for routing data

- Create script to generate data for Iris demo

- Add Navbar navigation

- Create Selenide Test for demo

- Make a test index in Elasticsearch to test out sockets

Sprint 0.0.4

- Unsubscribe the chart from onChartLoad

- Add download chart image button to each chart widget

- Check if billboard is storing all data points on charts

- Execute Chart aggregation on dashboard load

- Add download chart data button to each chart widget
- Add modal to prompt user for a revision comment when they click update on dashboard
- Add a dashboard to the bootstrap file
- Add CRUD functionality to widgets on Dashboard
- Add update functionality for Dashboards
- Create XL Modal for dashboards
- Set and resize chart to be the same as the widget area
- Create JS files for chart types
- Dashboard isRendering flag not working in some cases
- Add date by default to all json objects going into Elasticsearch
- Create Parsing logic for chart types in backend
- Load a saved dashboard client side
- Limit socket messages to dashboards which are currently rendering
- Save dashboard server side
- Use browser cache to store a widget's aggregation attribute
- Use data- HTML attribute for storing widget information
- billboard.js charts are not displaying correctly
- Need to add chart Id to subscription charts

Sprint 0.0.5

- Add StateChartList to Dashboards
- Remove all the Grails/Groovy aggregation builder code to reduce code and war size
- Add titles to charts
- Add StateChartDisk to Dashboards
- Add an admin user in production environment for deployment
- Add revision history for Dashboards
- Dashboard Revision objects are getting new UID rather than copying older ones
- Add pattern colours to charts
- Turn off stompjs debug logging for client speed improvement
- Change the Aggregation section on Dashboards, to contain a UI rather than raw JSON
- Edit or Remove extra aggregation attributes from aggregation builder views

Sprint 0.0.6

- Accept raw data from data source without running aggregations
- Add X-Auth-Token header and JWT to each agent
- Set up VS Code Latex work Shop and config settings in detailed description
- Add JWT token to prod and dev bootstrap envs
- Add code editor for creating rules on the schema creation page
- Add most recent document option to aggregations
- cloc command
- Put iris-crypto-rates on raspberry pi and schedule it as a cron job
- Add dynamic REST call to all agents to grab url from iris for agent
- Deploy iris v0.0.6