Dynamic Performance Framework

Dean Gaffney

November 30, 2017

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Purpose of Iris

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 them 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.

1.2 Motivation for Iris

Onaware are IAM (Identity and Access Management) specialists.

Onaware is an international company that deal with IAM and have offices of 20 staff in Waterford.

More information on Onaware can be found here https://onaware.com/

The motivation for this project comes from database and system performance issues that Onaware 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 attempting 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 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.

Chapter 2

Specification

2.1 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 2.1: Example Schema Created for Iris

Iris will take the users data and create data mappings (Elastic.co, Mapping) inside Elasticsearch, as well as insert any incoming data into the correct Elasticsearch index (Elastic.co, Basic Concepts). 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 lets 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:

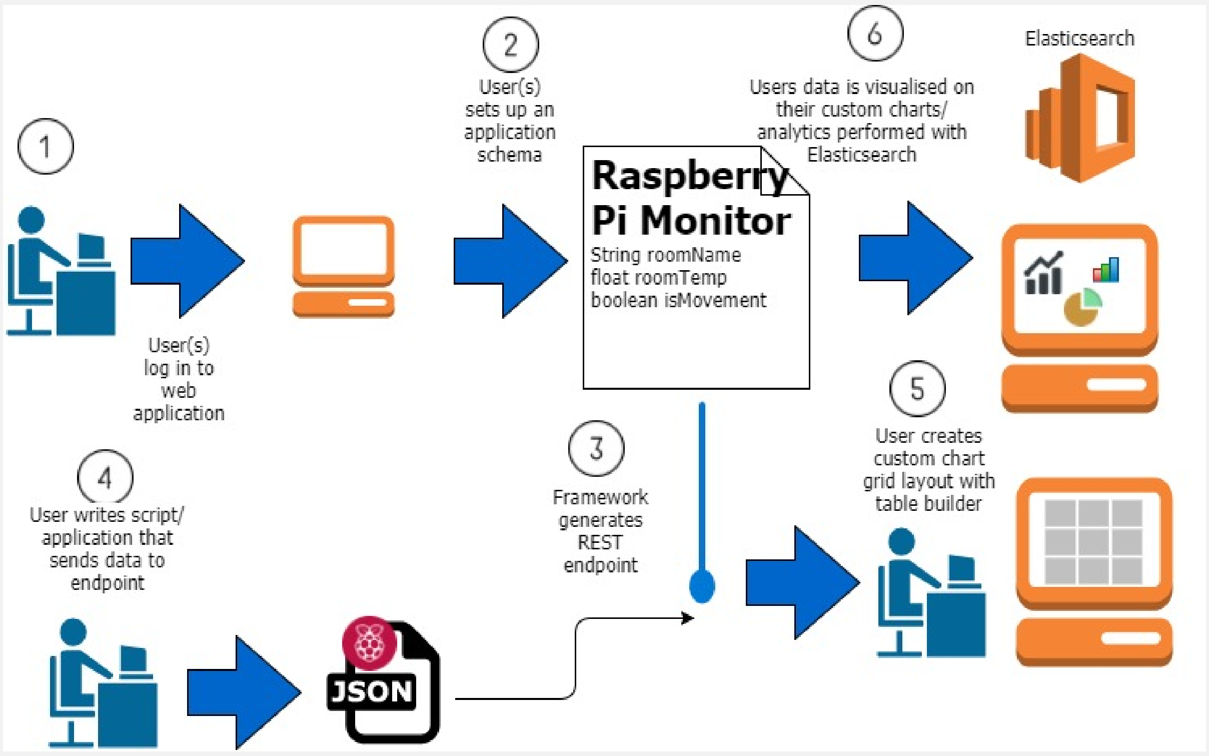
```
{
        "firstName": "Dean",
        "lastName":  "Gaffney",
            "fullName":  "Dean Gaffney"
}
```

Figure 2.2: 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, Aggregations, 2017) 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.

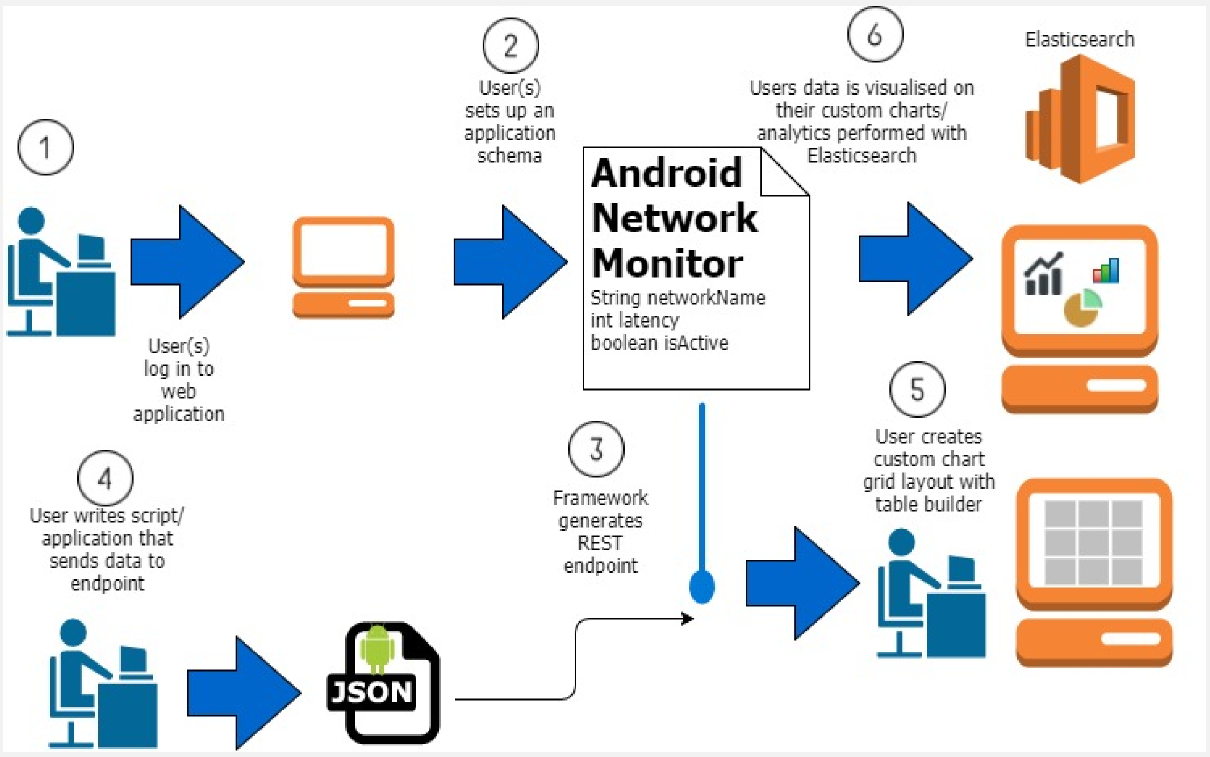
Iris will also come with some common scripts that will be downloadable for users to use straight away, some of these scripts will include JVM monitoring which can be placed into a java application, web page statistics which are retrieved from using Selenium web driver. Not all of the pre packaged monitoring scripts have been decided at this time, as the web application must be in place first.

2.2 Use Cases



In this example the user creates a schema in Iris for a raspberry pi. The user has a raspberry pi set up with a sensor to act as a home alarm. The user has written a script on the pi to send JSON data to the schema endpoint in Iris containing room name, room temperature and if there is any movement in the room. Data is being sent every five seconds to Iris. The user then logs into Iris and creates a dashboard for visualising the raspberry pi data.

Figure 2.3: Raspberry Pi Monitor for Iris



1
User(s) log in to web application
2
User(s) sets up an application schema
Android Network Monitor
String networkName
int latency
boolean isActive
6
Users data is visualised on their custom charts/ analytics performed with Elasticsearch
Elasticsearch
3
Framework generates REST endpoint
5
User creates custom chart grid layout with table builder
4
User writes script/ application that sends data to endpoint
JSON

In this example the user creates a schema in Iris for an android application. The android application is sending network data to the Iris generated endpoint, the user sends the network name, latency and if the network is currently active in the form of a JSON object, every five seconds. The user then logs into Iris and build a dashboard to visualise the android data coming into Iris.

Figure 2.4: Android Network Monitor for Iris

Chapter 3

Relevant Technologies

In this chapter the technologies that will be used and the rationale for their selection, in the development of the Iris system are outlined. One section in particular Generating Dynamic Elasticsearch Queries is long but is important because the choice of technology is not obvious and the advantages/disadvantages of the alternatives is of particular importance to Onaware.

3.1 Grails

Iris will be built using the MVC web framework Grails. Grails is suited to for Iris due its rapid scaffolding capabilities, meaning a web application with CRUD (Create, Read, Update, Delete) actions can be implemented extremely fast. Due to the scale of the project a framework which will provide a fast POC (Proof of Concept) is needed, due to Grails command line interface a web application skeleton can be auto generated in a couple of commands.

Also a significant factor in selecting Grails is that Onaware have several of their applications written using Grails and Iris will have a new lifecycle after this project cycle in which Onaware developers will be extending upon Iris.

3.2 Groovy

Groovy is used by default within the Grails framework, this is a major factor as to why the technology is being used in Iris. However due to Grails being built on the JVM pure Java code could be used instead of Groovy, the major advantages of using Groovy over Java is its expressive nature and syntactic sugar, which allow you to write less code and do more. Groovy integrates seamlessly with any Java library, which is a major factor for using Groovy in Grails. Due to the popularity of Java, if Grails doesnt have a native plugin for some needed functionality, a Java library more than likely exists to provide the required functionality, this library can then be used with Groovys expressive syntax.

```
//dynamic typing, will be cast as a string
def name = 'Dean'

//built in regex support
def matches = (name ==~ /\w+/)

def message = (matches) ? "Matches" :
                    "No match found in string $name"

println message

Output: Matches
```

Figure 3.1: Example of Groovy Syntax

3.3 Spock

Spock (http://spockframework.org/spock/docs/1.1/getting_started.html) is the default testing framework for Groovy and due to Grails using Groovy as its primary language. All unit and integration tests are written using the Spock framework. An advantage of the spock framework is that assertions are written in a behavior driven manner, which makes tests clear and concise unlike other JVM testing frameworks such as JUnit which relies on simple assertions.

```
void "test delete Schema"(){
        setup:
        setupData()

        when: "I delete a Schema"
        schema.delete(flush: true)

        then: "It is deleted from the database"
        assert Schema.count() == 0
}
```

Figure 3.2: Example of Spock Test in Iris

Peter Niederwieser has a concise introduction tutorial on Spock at the following link http://spockframework.org/spock/docs/1.1/introduction.html

3.4 Elasticsearch

Elasticsearch is a RESTful based search engine for indexing documents of data. Elasticsearch is suited for Iris as it accepts structured data in the form of index mappings, as well as unstructured data.

```
PUT twitter

{
   "mappings":{
      "tweet":{
         "properties":{
            "message":{
               "type":"text"
            }
         }
      }
   }
}
```

An example of an Elasticsearch mapping being set up for an index - https://www.elastic.co/guide/en/elasticsearch/reference/current/indices-put-mapping.html

Figure 3.3: Example of an Elasticsearch Mapping

There are alternatives to Elasticsearch, in particular Hadoop and Spark. However, the main reason for using Elasticsearch is that Onaware developers have experience using Elasticsearch on their other products, so to maintain the product after the project lifecycle Elasticsearch is the best choice for Onaware as they can remain consistent with their technologies.

Another reason for Elasticsearch being used in Iris was due it being RESTful. All data coming to Iris will be made with REST calls already in the correct format for Elasticsearch, meaning a simple redirect to an Elasticsearch URL allows Iris to insert a users data to Elasticsearch. Querying Elasticsearch is also a major part of Iris and it can be done through REST calls using aggregations. Aggregations will allow Iris to come with out of the box analytics that can be easily mapped to a user's chart. As aggregations are the backbone of Iris an aggregation playground will be put in place where a user can build up complex aggregations using a user friendly interface and send them to Elasticsearch.

3.4.1 Generating Dynamic Elasticsearch Queries

3.4.1.1 Elasticsearch Java API Issue

Elasticsearch comes with a rich Java API for using Elasticsearch in your applications. The API covers all features of Elasticsearch which Iris requires such as creating indices, mapping indices and performing and parsing aggregations. The main issue with the Java API is that it doesnt expand well in Iris due to Iris dealing with dynamic data. The Java API is set up in a such a way that if a user wishes to create a complex aggregation with several sub-aggregations Iris would have to use reflection or have several complex factory design patterns in place. This approach is not ideal, and is best explained with an example using the Java API:

POST /exams/_search?size=0

```
{
   "aggs":{
      "avg_grade":{ "avg":{"field": "grade"}}
   }
}
```

An example of an average aggregation being performed on a field grade. As you can see an aggregation is just a JSON object which works upon a given field. However when you use the Java API building a similar aggregation without knowing beforehand what the data looks like would be quite difficult and may require reflection in the language you are using. Below is an example of a simple aggregation with a sub aggregation being created using the Java API. https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-metrics-avg-aggregation.html

```
AggregationBuilders
        .terms("genders")
        .fields("gender")
        .order(Terms.Order.aggregation("avg_height", false)
        .subAggregation(
           AggregationBuilders.avg("avg_height")
                                          .field("height")
        )
```

Looking at this example of the Java API, the first method call to terms, references an aggregation type and the next call adds the gender field to the aggregation. An Optional order is added to the aggregation, notice how a static call to Terms.Order.aggregation is used. Finally a sub aggregation is added, which refers to the average aggregation. From the example notice the large amount of calls to static methods, this would require Iris to have a factory in place for each aggregation type, which is possible. The problem occurs when users wish to add extra attributes such as ordering to an aggregation, as Iris would need a factory in place for every possible form of an aggregation. Once again this is possible, but not a very elegant way of building up a JSON object, as a user may not wish to use all of the available optional attributes.

Figure 3.4: Example Aggregation for Elasticsearch

The API is catered more towards having prebuilt aggregations in the backend where you can just fill in the blank parts of the aggregation with fields and is not the best for building aggregations from scratch due to the amount of code needed to fit with the existing API. It is for this reason Iris will implement its own approach for building aggregations.

3.4.1.2 Solution using Groovy/Grails

Groovy is an excellent candidate as Groovy objects are essentially just maps of keys and values which can be converted into JSON using standard Groovy functions. Groovy constructors will only use the fields given to it and add default values to the remaining fields which were not specified. With this knowledge Elasticsearch aggregation objects built in Groovy would automatically support optional attributes such as order, sort etc

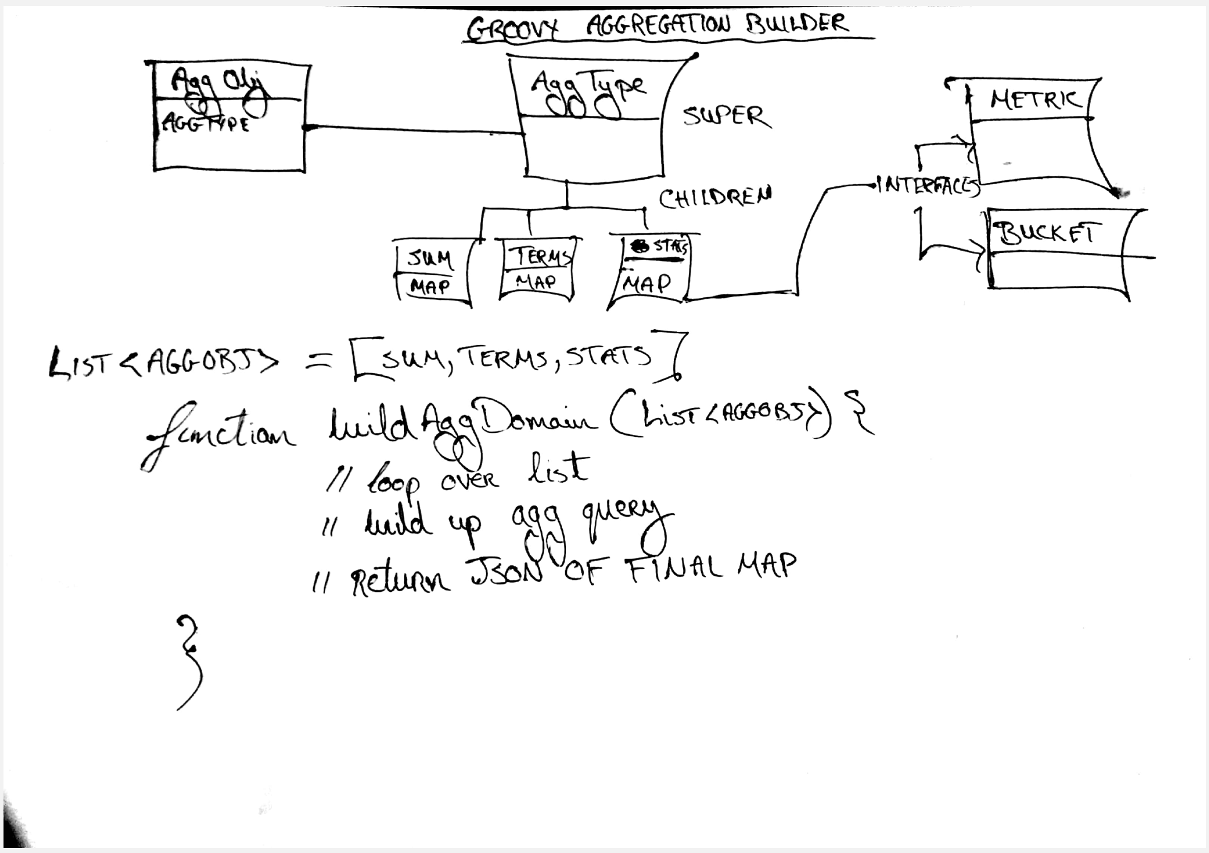
```
import Groovy.json.JsonBuilder

class Person{
    String name
    int age
}

// age will be given a default value
def person1 = new Person(name: 'dean')

 //notice the key/value style constructor
def person2 = new Person(name: 'shane', age: 20)

Map personMap = [name: 'kieran', age: 30]
//object constructed from a map
def person3 = new Person(personMap)

println new JsonBuilder(person1).toPrettyString()
println new JsonBuilder(person2).toPrettyString()
println new JsonBuilder(person3).toPrettyString()
```

Output:

```
{
    "age": 0,
    "name": "dean"
}
{
    "age": 20,
    "name": "shane"
}
{
    "age": 30,
    "name": "kieran"
}
```

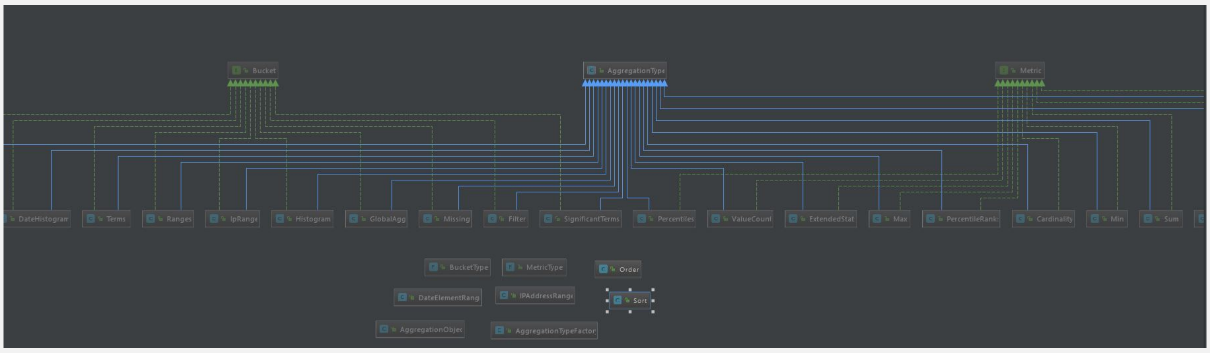
Figure 3.5: Example of Groovy Constructor

The Groovy implementation follows a standard OOP (Object Orientated Programming) approach. Each Aggregation type is given a class with optional fields, similar to the example above, a factory then accepts a map coming from the front end along with the name of the aggregation type and creates the correct object using the map.


GROOVY AGGREGATION BUILDER
Agg Obj
AGGTYPE
AggType
SUPER
CHILDREN
SUM
MAP
TERMS
MAP
STATS
MAP
INTERFACES
METRIC
BUCKET
LIST <AGGOBJS> = [SUM, TERMS, STATS]
function buildAggDomain (List <AGGOBJS>) {
// loop over list
// build up agg query
// return JSON OF FINAL MAP
}

This diagram shows how a Groovy implementation of Elasticsearch aggregations may look and how they may be built into a single object.

Figure 3.6: Iris Aggregation Implementation POC



The diagram above is a screenshot from IntelliJ. The diagram shows my Groovy implementation class structure.

Figure 3.7: Iris Aggregation Groovy Implementation

3.4.1.2.1 Advantages of using Groovy/Grails

- Groovy constructors are maps which can assign default values
- OOP approach
- Scalable, can be updated easily to match Elasticsearch aggregations
- Addition and removal of aggregation classes without system interference

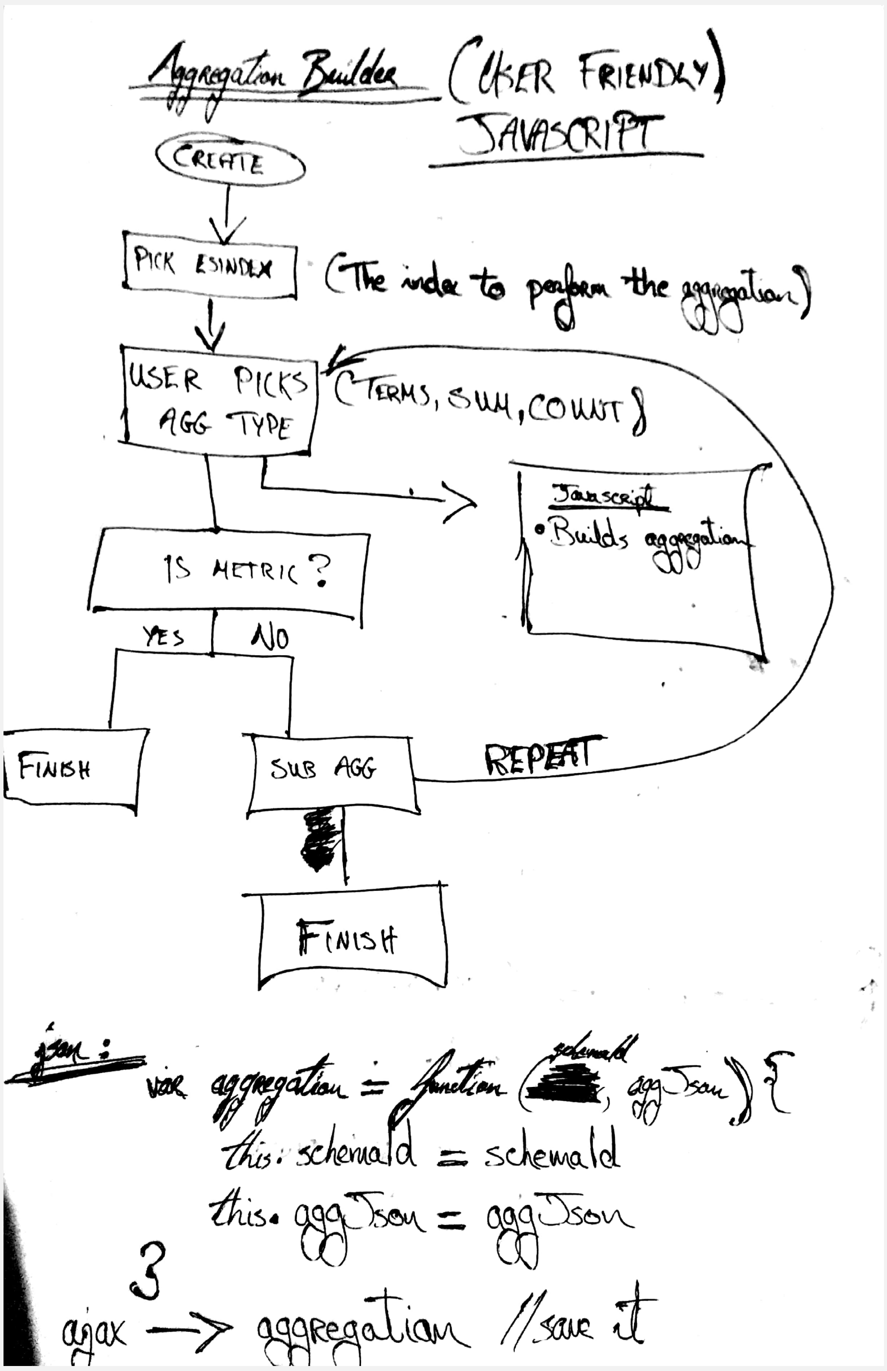
3.4.1.2.2 Disadvantages of using Groovy/Grails

- Link between frontend and backend requires complex command objects
- Command objects become very hard to maintain, as it must be index within HTML

An example of how command objects work can be found here http://guides.grails.org/command-objects-an
guide/index.html

3.4.1.3 Solution using Javascript

Currently a separate implementation is in development for Iris to deal with aggregation objects using Javascript. This implementation depends on building the aggregation object dynamically from user input on the front end. As the user builds the aggregation object on the front end, an aggregation JSON object is being built along with it, this JSON object will then be saved in Iris as a String object.



```
var aggregation = function (schemaId, aggJson) {
    this.schemaId = schemaId
    this.aggJson = aggJson

ajax -> aggregation //save it
```

This diagram shows the flow of how a user would create an aggregation using the frontend of Iris and shows how the Javascript builds the aggregation incrementally.

Figure 3.8: Javascript Implementation POC

3.4.1.3.1 Advantages of using Javascript

- Keep the aggregation object in its native language, JSON
- Remove 15 - 20 Groovy classes with a single Javascript file
- Store JSON as a string in database, making it immediately executable

- Balances load on the server, as clients now build aggregations

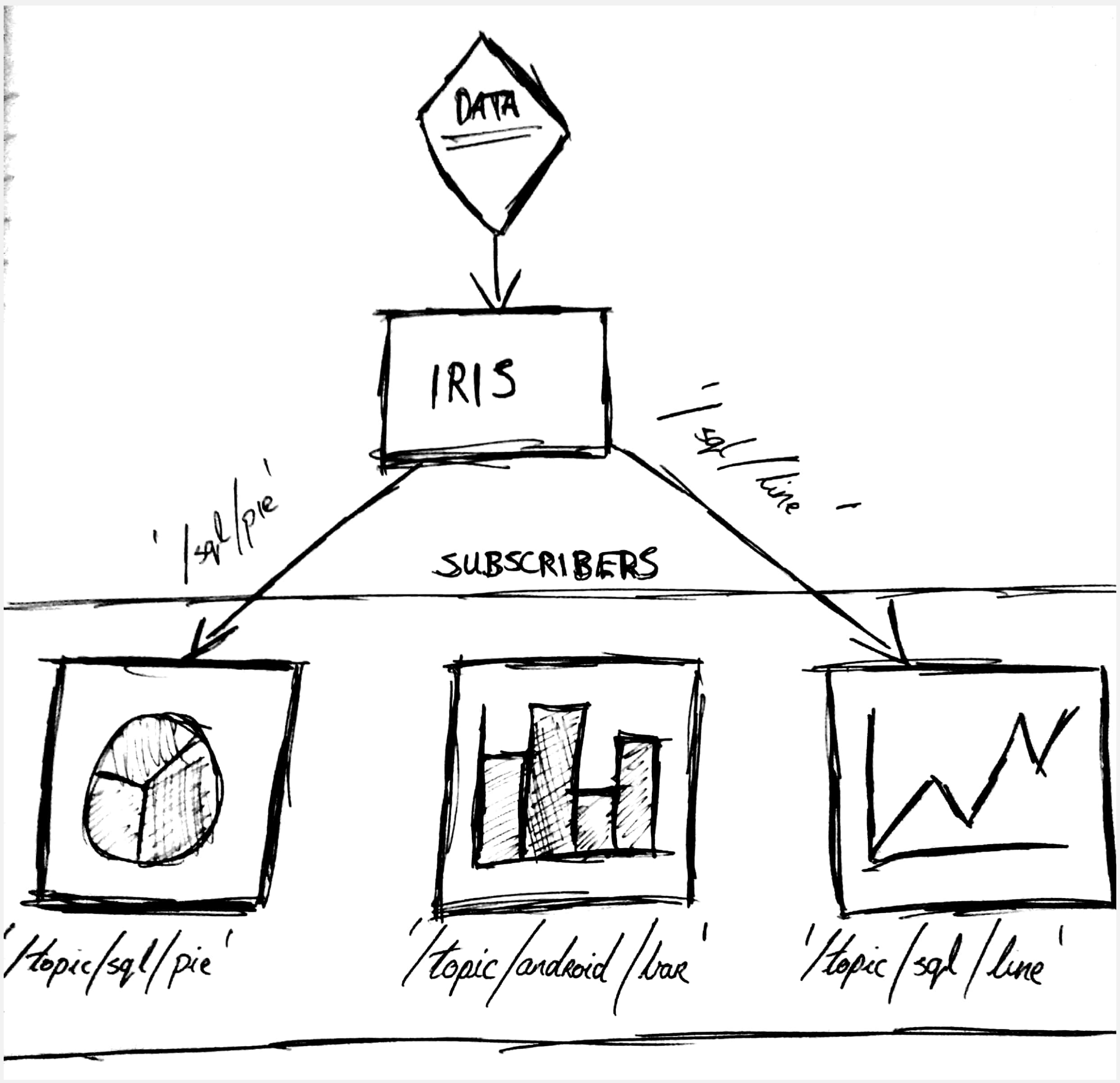
A practical tutorial on Elasticsearch aggregations and sub-aggregations can be found here https://qbox.io/blog/elasticsearch-aggregations-for-analytics

3.5 JQuery

JQuery (http://api.jquery.com) will be used to post form data, make ajax calls as well as traversing and manipulating the DOM (Document Object Model). JQuery is a defacto industry standard library for accomplishing these tasks.

3.6 Grails Spring Websockets Plugin

This plugin will be used mainly for updating charts. The sockets will allow for charts to be updated as soon as new data comes into Iris. The Spring plugin allows a Grails Service or Controller to deal with passing data to specific endpoints. These endpoints are referred to as STOMP (Simple [or Streaming] Text Orientated Messaging Protocol) endpoints (Docs.spring.io, 2017). It is then possible for Iris to target charts specific to a schema, which is much more efficient than all charts receiving all data. Controlling STOMP endpoints in Iris can be taken a step further by creating dynamic STOMP endpoints which would be specific to both a schema a chart type, meaning a pie chart could subscribe to an endpoint only related to pie charts, so the data being sent to the pie chart subscriber will already be formatted for pie charts. This once again prevents charts receiving data which is not relevant to them. The endpoints are secured with the Spring Security plugin.



The diagram above shows data coming into Iris. The data coming into Iris is for a schema called sql. Iris sends out data to "/sql/pie" and "/sql/line", this means any pie charts or line charts associated with the "sql" schema will receive the data and be updated. This is shown by the pie chart and line chart subscribing to the correct topics "/topic/sql/pie" and "/topic/sql/line". The bar chart is subscribed to an endpoint called "/topic/android/bar" which will not receive any updates as no data has been sent to Iris related to the "android" schema.

Figure 3.9: Iris Socket POC

For documentation on the Grails Spring Websockets plugin see https://plugins.grails.org/plugin/zyro/grails-spring-websocket

3.7 Selenide (Functional Testing)

Selenide (http://selenide.org/documentation.html) will be used for functional tests in Iris. Selenide is a wrapper library for Selenium which removes the boilerplate code for Selenium such as Implicit and Explicit waits. Selenide allows your tests to be more clear and concise due to its expressive function naming convention. Selenide also uses Javascript style selectors to make the developer feel as though they are targeting DOM elements using Javascript rather than Java.

3.8 Dashboards

Iris will allow users to create dashboards which consist of charts, the dashboard tiles should be moveable and resizable and most importantly serializable.

3.8.1 Dashboard Library Requirements

- Moveable tiles
- Resizeable tiles
- Adding tiles
- Removing tiles
- Have serializable functionality
- Be under an MIT License so Onaware may use it
- Active Developer Support (Optional, but preferable)

3.8.2 Review of Dashboard Libraries

Four libraries were researched in the design process of Iris dashboard system. Refer to Appendix C for a detailed comparison of the dashboard libraries. Below is a table summarising the comparisons.

Library	Moveable Tiles	Resizable Tiles	Add Tiles Dynamically	Remove Tiles	Serializable	MIT Licensed	Active Support
Gridster	✓	✓	✓	✓	✓	✓	✗
Packery	✓	✓	✓	✓	✗	✗	✗
GridList	✓	✓	✗	✗	✓	✓	✗
Gridstack	✓	✓	✓	✓	✓	✓	✓

Table 3.1: Dashboard Review

3.9 Chart Libraries

3.9.1 Chart Library Requirements

- D3v4 wrapper library or better alternative to D3
- Support basic chart types (Bar, Bubble, Pie, Line etc)
- Out of the box legends, zooming and responsiveness
- Allow for chart updates without the need for redrawing
- JSON data support
- Active Developer Support (Optional, but preferable)
- Advanced Chart Types (Radar, Heatmap etc)

3.9.2 Review of Chart Libraries

Four libraries were researched in the design process of Iris charting system. Refer to Appendix D for a detailed comparison of the chart libraries as well as a information on why D3 is not being used for Iris. Below is a table summarising the comparisons.

Library	D3v4 - Wrapper	Basic Charts	Advanced Charts	Included Functionality	Updateable Charts	JSON Support	Active Support
C3	✗	✓	✓	✓	✓	✓	✗
Billboardjs	✓	✓	✓	✓	✓	✓	✓
Chartjs	✓	✓	✗	✓	✓	✓	✓

Table 3.2: Chart Libary Review

3.10 Similar Products

3.10.1 Kibana

Kibana (https://www.elastic.co/guide/en/kibana/5.6/getting-started.html) is a web application which allows you to visualise your ELK (Elasticsearch, Logstash, Kibana) stack. Kibana supports aggregations to be performed on Elasticsearch data and the visualisation of aggregations using charts and metrics. Kibana also supports persistent dashboards, however they are not personal dashboards meaning all users with access to the Kibana instance can view and modify the same dashboards, which is an unusual design choice (Logz.io, 2017). Kibana does not have any concept of a user, meaning there are no personal aspects to the application. Kibana also doesnt have any user friendly ways to insert data into Elasticsearch meaning you have to understand Elasticsearch concepts in order to use Kibana. Kibana is also aimed towards managing and visualising log data rather than generic data.

3.10.2 Grafana

Grafana (http://docs.grafana.org/features/datasources/elasticsearch/) is an open source dashboard tool for visualising time series data. Grafana supports several backend data sources, including Elasticsearch. Each data source has a corresponding query editor for users to query their data. Grafana is extremely powerful and has a very active community where users can collaborate and share dashboards. Grafana has the same issues as Kibana and doesnt allow a user to enter data into the data source without having background knowledge of the data source.

3.10.3 New Relic

New Relic (https://newrelic.com/application-monitoring/features) is a monitoring application which comes close to what Iris wishes to achieve. New Relic is mainly used for anticipating bottlenecks and issues with applications. It is extremely powerful and supports multiple languages (Java, Nodejs and Python) and frameworks however it wont support data for unsupported languages and frameworks and is not dynamic this is a fundamental feature of Iris.

3.10.4 AWS (Amazon Web Services)

AWS will be used for hosting an Elasticsearch instance which Iris can use throughout its development cycle. The advantage of using AWS is that Iris can query the Elasticsearch instance from any location making the development flexible.

Chapter 4

Iris Current Progress

4.1 Design Iterations

Iris has had four design iterations up to this point. This section will talk about Iris current design. See Appendix A, for details on the other design iterations of Iris.

Iris development process is being carried out using the agile methodology of BDD (Behavior Driven Development). Iris is developed in iterations of two week sprints. Every two weeks Iris is branched over to a new branch and Trello is updated with a new label which corresponds to the branch number. Iris is currently on version 0.0.3. For more information on Trellos roll in the development of Iris please refer to Appendix B.

4.1.1 Iris (Current)

Iris currently has all of the models from the following diagram in place.

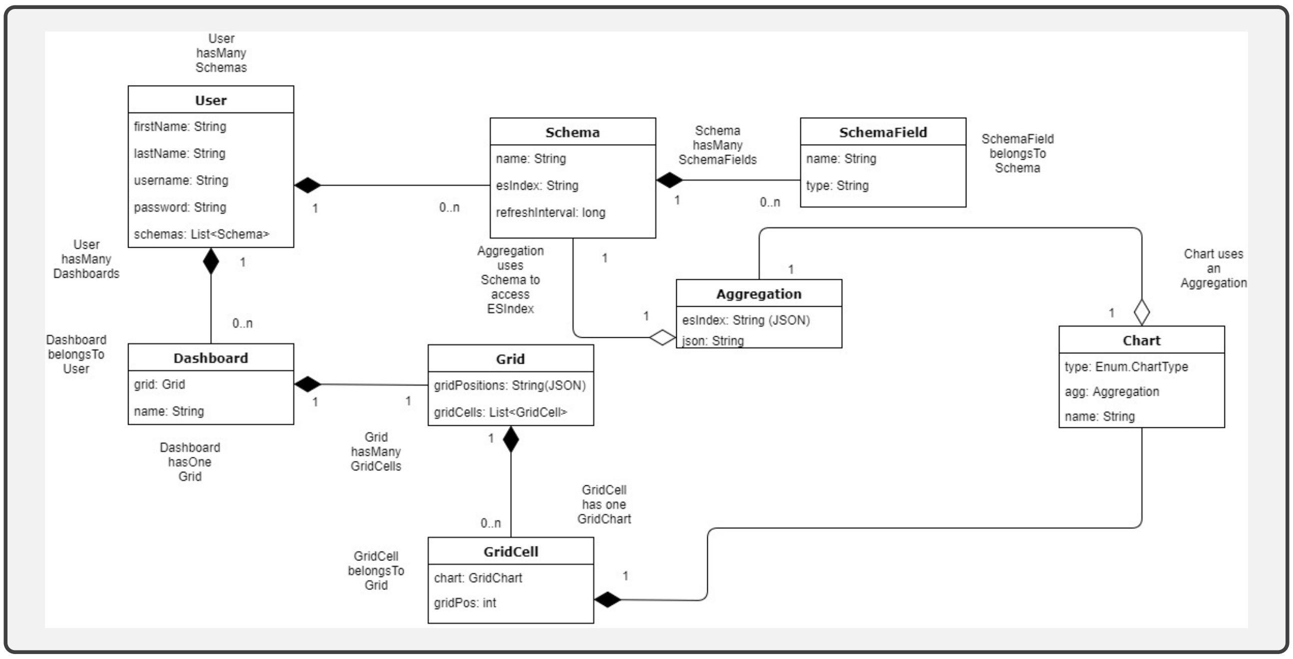


Figure 4.1: Iris UML Diagram

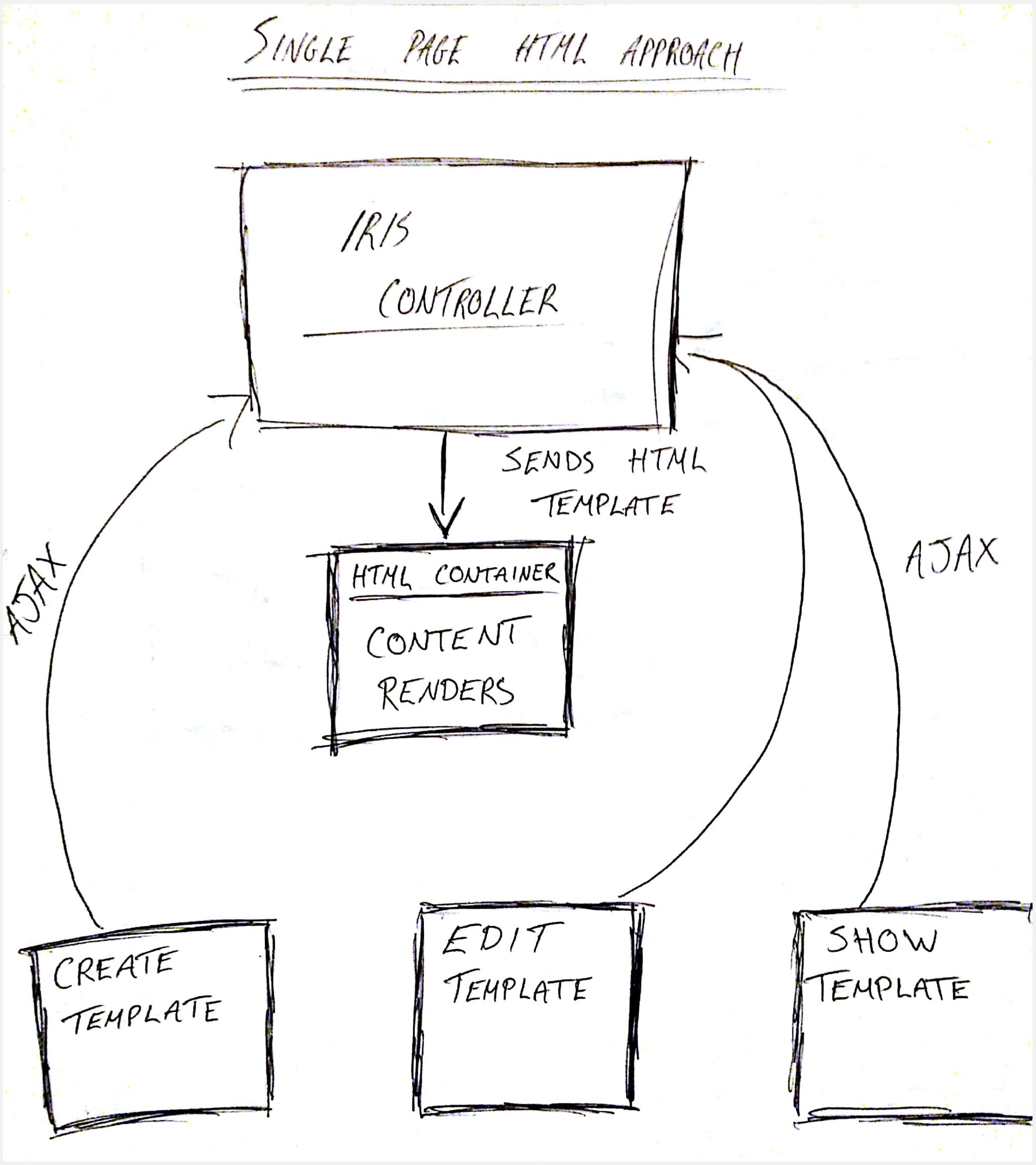
All models in the previous diagram have been implemented and fully tested in Grails. Iris currently allows a user to create a schema with schema fields, which in turn creates Elasticsearch mappings and indices specific to the created schema. With a mapping setup Iris allows a user to target an endpoint to send their data, the logic for putting this data into Elasticsearch has been written and tested but it is not linked up yet.

Iris has an Elasticsearch instance hosted on AWS (Amazon Web Services) which will be used for the remainder of the development process.

Iris also has the two implementations of the aggregation building system in development. The Groovy version is completed but not linked to the backend, as progress on the Javascript version has begun which seems to be a more promising. It is likely that the Javascript version will completely replace the Groovy version. There is currently a single page application within Iris which allows a user to build up an aggregation on the front end which was completed using Javascript. This is still quiet rough on the front end, but proves that the Javascript logic works.

Iris also has a single hard coded chart which sits inside a gridstack grid (dashboard). The chart is being updated with randomly generated data being every five seconds to update the chart. This is to make sure data can be flushed smoothly to Billboardjs and ensure that the chart will not overflow inside of the dashboard grid.

As of now the next steps in the current sprint is to populate Elasticsearch with some real data and to develop the socket logic to allow charts to be updated when data is being routed to Iris. Serialization of dashboards is another time consuming task which must be implemented, but still needs to be planned out some more in order to ensure it is done properly. The front of end Iris is still rough, and the following diagram shows how the front end plans to interact with the backend.



Iris will act as several single page web applications. As show in the diagram, each page will contain a main area for html content, which will be replaced by templates using ajax calls to the backend.

Figure 4.2: Several Single Page Applications Design