UNIVERSIDAD
REY JUAN CARLOS

INGENIERÍA EN TECNOLOGÍAS DE LA TELECOMUNICACIÓN

Curso Académico 2017/2018

Trabajo Fin de Grado/Máster

ANGULAR - ELASTICSEARCH - DASHBOARD INTERFACE

Autor : Ismael Slimane Zubillaga

Tutor : Dr. Jesús M. González-Barahona

Trabajo Fin de Grado/Máster

Título del Trabajo con Letras Capitales para Sustantivos y Adjetivos

Autor : Nombre del Alumno/a

Tutor : Dr. Gregorio Robles Martínez

La defensa del presente Proyecto Fin de Carrera se realizó el día de de 20XX, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 20XX

Dedicado a

mi familia / mi abuelo / mi abuela

II

Agradecimientos

Resumen

Summary

Contents

List of Figures

Chapter 1

Introducción

1.1 Descripción del Problema

1.2 Objetivo Principal y Estructura

1.3 Disponibilidad del Software

Chapter 2

State of Art

2.1 ElasticSearch

Elasticsearch is an open-source, *broadly-distributable*, *readily-scalable*, *enterprise-grade* search engine based on Lucene[1]. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents.

2.1.1 History

Compass was the predecessor to Elasticsearch and it was created by **Shay Banon** in 2004. Through the implementation of the third version of Compass, came the necessity of a *"scalable search solution"*. This necessity would have meant a lot of work rewriting big pieces of code, so Shay decided to build *"a solution built from the ground up to be distributed"* and used a common interface, *JSON* over *HTTP*, available for other programming languages, and not only *Java*.

The first verision of Elasticsearch was released on February 2010.

2.1.2 Basic Concepts

All the following concepts definitions are extracted from the Elasticsearch documentation web page[2].

[1] `https://en.wikipedia.org/wiki/Apache_Lucene`

[2] `https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html`

- **Near Realtime (NRT)**: Elasticsearch is a near real time search platform. The practical meaning of this is that there is a little latency (about one second) from the moment you store a new document, to the moment it becomes available for searching.

- **Index**: An **index** is a set of documents that share some type of characteristics. For example, you can have an index for a shop product, another index for employee data and another for bill data. An index is identified by its name (in lowercase), and this name is used for several operations as: searching, deleting, updating, etc.

- **Shards & Replicas**: The index data can reach a large size exceeding the available physical memory. But this problem can be fixed by defining multiple **shards**. Each shard is a portion of data from the index data. The number of shards is defined when the index is created.

  But there is still another potential problem. It always exists the possibility to have a failure on the network system and to loose a shard/node, so it is advisable to have **replicas** of our data. A replica is a copy of the index shards.

- **Type**: We can see a **type** like an object class, with fields of different data-types.

  In Elasticsearch a type is defined by its *name* and its *mappings*. The *mappings* are a schema of the type, where it's defined the properties that our type has, and the data-type of each property, such as *integer, string, etc.*.

- **Document**: As we said about *types* being like classes, we could see a **document** like a record from a single class. Using the same example we used for *indexes*, you can have a document for a single product, another document for a single employee and yet another for a single bill.

- **Node**: We can see a **node** like a single server, as a single unit that along with other nodes, make up a cluster. This node take part on cluster's indexing and searching tasks.

- **Cluster**: As we said in the previous definition, a **cluster** is made up with multiple nodes(servers). The cluster stores all the data and allows to search and index all this data across all nodes. A cluster is a collection of one or more nodes (servers) that together

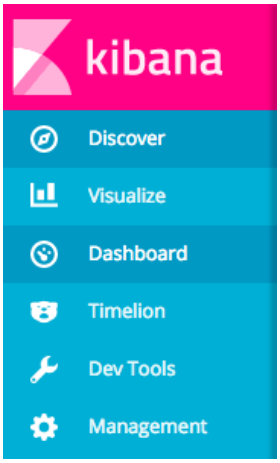

Figure 2.1: Kibana's nav-bar.

holds your entire data and provides federated indexing and search capabilities through all nodes.

2.2 Kibana

Kibana is an *Elasticsearch* open-source plug-in. It mainly works with Elasticsearch indexed data in order to represent it into different types of visualizations such as bar charts, scatter plot charts, pie charts, ...

Kibana, as an exploration tool, can be used to log and time series analytics, application monitoring, and operational intelligence use cases.

We can find another similar applications such as *Grafna*[3], *incubator-superset*[4] and *Tableau*[5].

2.2.1 Main Features

In the Figure 2.1, we can see the Kibana's *nav-bar*. Now we are going to speak about a couple of features.

- **Visualize**:

  Visualize allows you to represent data with a specific type of chart. The represented data will be chosen from the available index on Elasticsearch. In addition to the Elasticsearch

[3] `https://grafana.com/`
[4] `https://github.com/apache/incubator-superset`
[5] `https://www.tableau.com/`

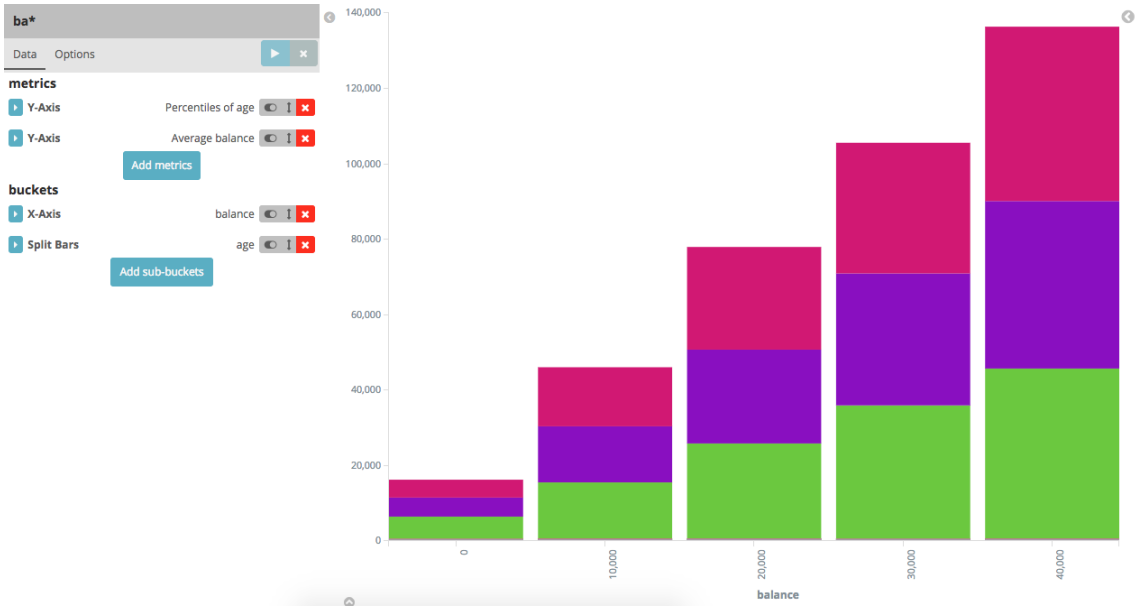

Figure 2.2: Visualization example.

index, we have to choose which type of *aggregations* are we going to use in order to extract our data. We can see an example on Figure 2.2.

- **Dashboard**:

  The Kibana's dashboard functionality allows us to display a collection of saved visualizations and organize them by dragging and dropping. We can see an example on Figure 2.3.

2.3 GitHub

"**GitHub** is a *Web-based* **Git version control repository hosting service**". Github is used mainly for programming code. In addition to Git functionalities as *deistributed version control* and *source code management*, Github provides more features such as collaboration features for *bug tracking* and for other porposes, task management, and *wikis* for documentation porposes.

For the current project, a repository was created on *Github* with the name of '*Angular-ElasticSearch-Dashboard_Interface*'[6].

[6] `https://github.com/islimane/Angular-ElasticSearch-Dashboard_Interface`

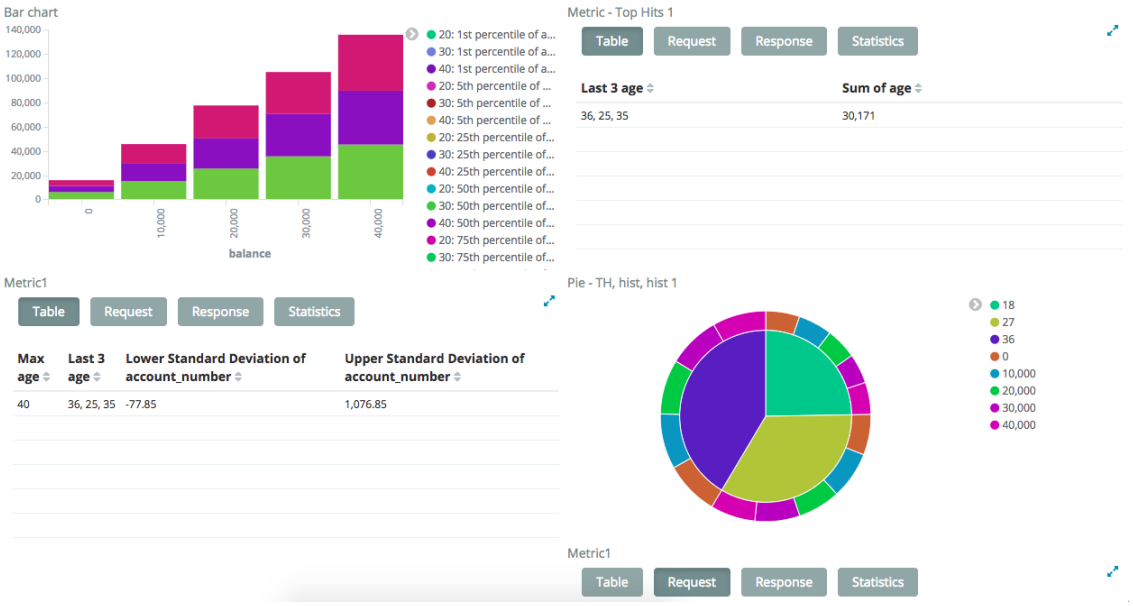

Figure 2.3: Dashboard example.

2.4 Webpack

"**Webpack** is an *open-source JavaScript* module bundler". The main reason for using Webpack is to have a modular structure on your web application project. This allows you to have a cleaner code and make it more reusable and scalable. Webpack can be used from the command line or be configured with a config file named *webpack.config.js*.

2.5 Angular

Angular is an open source *web application platform* based on *Typescript* and developed by Google and by a community of individuals and corporations. Angular is commonly referred to as Angular 2.0 or later versions.

2.5.1 Versions

Before Angular was created, there was a previous version called *AngularJS* or Angular1.X. Angular was created as a complete rewrite of AngularJS.

- 2.0.0:

This was the first version of Angular. Its announcing was made at the *ng-Europe* conference on September 2014. This version was build up from the ground and these were the main characteristics:

  - Introduced the components hierarchy.
  - Modules for core functionality, improving the speed of the Angular core.
  - Possibility of using *TypeScript* language. The use of this language is recommended by the Angular team.
  - Improved *dependency* injection.
  - Dynamic loading.
  - Reactive programming support using RxJS.

And much more features. The final version was released on September 14, 2016.

- 4.0.0:

  This version was called *Angular 4* and was announced on 13 December 2016. Angular 3 was skipped due to some features already present on version 3.0.0 and to avoid confusion. This version introduced:

  - *Http* client.
  - A new *Rout Live Cycle*.
  - Conditionally disable animations.

- 5.0.0:

  This version was released on November 2017 and the main introduced features were:

  - Support for progressive Web apps.
  - A build optimizer and improvements related to *Material Design*.

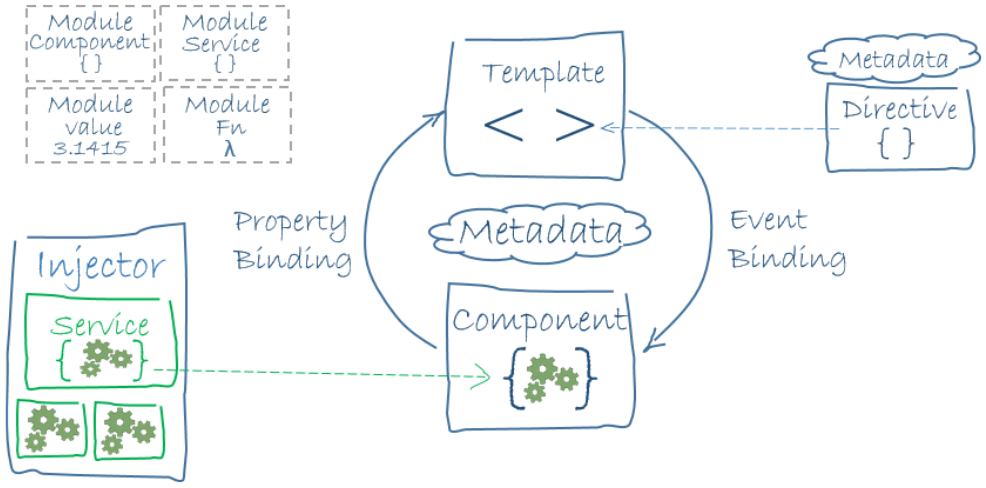

Figure 2.4: Architecture of an Angular application

2.6 TypeScript

"**TypeScript** is a free and open-source programming language developed and maintained by Microsoft. It is a strict syntactical *superset* of *JavaScript*, and adds optional static typing to the language".

2.6.1 History

Before it was made public, *TypeScript* gone through a two years of development process until it reached the 0.8 version. Then, was released on October 2012. Initially it had a lack of *IDE* support, until 2013 when some IDE's began to have support for this language, such as *Eclipse* with a dedicated pug-in, and some text editors such as *Sublime*, *Atom*, etc.

The reason TypeScript was created came from the necessity of a front-end programming language that fulfill the task of the development of complex and large-scale front-end applications, since JavaScript has shortcomings on that sense.

TypeScript is based on the *EMACScript* approach, the reason why is because it is a standard and because it has support for *class-based programming*.

2.6.2 Language Features

- Type annotations and compile-time type checking:

TypeScript provides an optional static typing, with annotations, that is checked at compile time. If this typing it is not used, then the Javascript's dynamic typing is used. Here is an example:

```
1 function planetMoons(planetName: string, moonsNum: number,
     spanish: boolean): any {
2   if(spanish){
3       console.log('El planeta ' + planetName + ' tiene ' +
           moonsNum + ' lunas');
4   }else{
5       console.log(planetName + '  planet has ' + moonsNum + '
           moons');
6   }
7
8   return null;
9 }
```

string, *boolean* and *number* are primitive types. For dynamic types it's used *any*.

- Type inference.
- Type erasure.
- Interfaces.
- Enumerated type.
- Mixin.
- Generic.
- Namespaces.
- Tuple.
- Await.
- Classes (backported from EMACScript 2015).
- Modules (backported from EMACScript 2015).

2.7 JavaScript

"**JavaScript**, often abbreviated as JS, is a high-level, dynamic, weakly typed, prototype-based, multi-paradigm, and interpreted programming language. Alongside HTML and CSS, JavaScript is one of the three core technologies of World Wide Web content production". Generally, the main purpose of Javascript is to make pages interactive and develop online programs such as online games. Javascript is supported by most of the web browsers, for that reason, the majority of the websites employ it. There are multiple engines for Javascript, but all are standardized with the *EMACScript* specification.

2.7.1 History

Javascript came with the necessity of dynamism on web browsers. The founder of Netscape Communications, Marc Andreessen, thought that HTML needed a "glue language" to assemble component like images or plug-ins into the web markup. In 1995, Netscape Communications began to use *Java* as a static programming language for Netscape Navigator, so the scripting language they had being searching for, had to be similar to Java and would complement it. Then, on May 1995, Brendan Eich wrote in 10 days this scripting language in order to compete against competing proposals. The first time this scripting language was called JavaScript was on December 1995, when the Netscape Navigator 2.0 beta 3 was released.

2.7.2 Syntax

These are a few examples to illustrate the JavaScript syntax:

- Variables:

```
1 var x; // defines the variable x and assigns to it the special
     value "undefined" (not to be confused with an undefined value
     )
2 var y = 2; // defines the variable y and assigns to it the value
      2
3 var z = "Hello, World!"; // defines the variable z and assigns
     to it a string entitled "Hello, World!"
```

- Printing:

```
1 console.log("Hello World!");
```

- Functions:

```
1 function add(a, b){
2     return a + b;
3 }
4 add(1, 2); // returns 3
```

2.8 Lodash

Lodash is a *JavaScript* library that provides extra functions for tasks that are often required on data structure managing, a lot of them, not implemented on the main JavaScript functionality.

For example:

- **_.difference(array, [values])**: Creates an array of unique array values not included in the other provided arrays using SameValueZero for equality comparisons.

```
1 _.difference([1, 2, 3], [4, 2]);
2 // returns [1, 3]
```

- **_.pluck(collection, path)**: Gets the property value of path from all elements in *collection*.

```
1 var users = [
2   { 'user': 'barney', 'age': 36 },
3   { 'user': 'fred',   'age': 40 }
4 ];
5
6 _.pluck(users, 'user');
7 // returns ['barney', 'fred']
```

As we can see, Lodash functions are called through the global variable "_". All this functions can be found on the Lodash documentation[7] page. This JavaScript library saves a lot of time when managing data structures.

[7] https://lodash.com/docs/3.10.1

There are other similar libraries such as *UnderscoreJS*[8].

2.9 jQuery

"*jQuery* is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML". jQuery it's the most used JavaScript library. jQuery's syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications.

2.9.1 History

jQuery was originally released in January 2006 at BarCamp NYC by John Resig and was influenced by Dean Edwards' earlier cssQuery library. It is currently maintained by a team of developers led by Timmy Willison.

2.9.2 Usage Examples

- HTML:

```
1 <p>Hello</p>
```

- JavaScript:

```
1 $("p").clone().add( "<span>Again</span>" ).appendTo(document.
    body);
```

- Result:

  See the result on Figure 2.5

2.10 HTML5

"**HTML5** is a markup language used for structuring and presenting content on the World Wide Web. It is the fifth and current major version of the HTML standard".

[8] http://underscorejs.org/

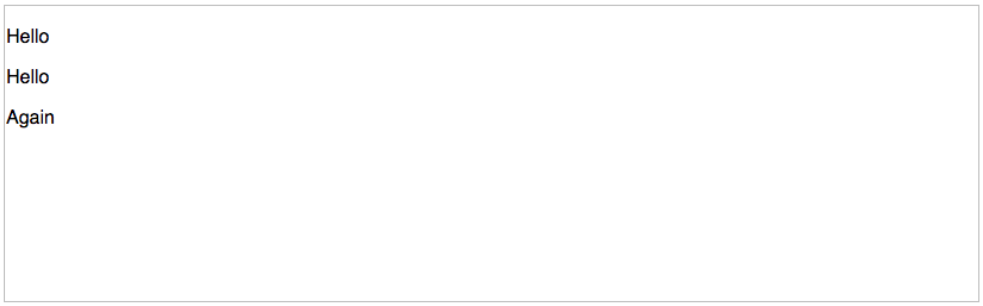

Figure 2.5: View result.

The *World Wide Web Consortium* released HTML5 on October 2014, their goal was "to improve the language with support for the latest multimedia".

2.10.1 Features

These are some of the feature introduced in HTML5:

- **Markup**:

  HTML5 introduces some elements that are commonly used on modern websites, they usually are replacements of old HTML elements, between these new elements we can find: <*nav*>, <*footer*>, <*audio*>, <*video*>, ...

- **New APIs**:

  HTML5 introduces *application programming interfaces* or *APIs* in addition to the markup elements that we have seen, such as *Canvas*, *Timed Media Playback*, *Drag and Drop*, *MIME type*, *Web Messaging*, *Web storage*, ... In this project we have mainly used *Canvas* through the *ChartJS* library.

2.11 ChartJS

ChartJS is a JavaScript library for producing dynamic, interactive data visualizations in web browsers, this is achieved by using the *HTML5 canvas* element. With ChartJS you create different charts such as pie charts, bar charts, line charts, etc. You can find visual examples of each type on the ChartJS samples page[9].

[9] `http://www.chartjs.org/samples/latest/`

2.11.1 Usage Examples

We have to instantiate the *Chart* class in order to create a chart, an then assign it to a *canvas* element on the DOM. The field *type* indicates which chart are we going to use, and the field *data* stores our chart info. Here is an example:

- HTML:

```
<canvas height='50' id='myChart' width='200'></canvas>
```

- JavaScript:

```
var ctx = document.getElementById("#myChart");
var myChart = new Chart(ctx, {
  type: 'bar',
  data: {
    labels: ["Red", "Blue", "Yellow"],
    datasets: [{
      label: '# of Votes',
      data: [12, 19, 3, 5, 2, 3],
      backgroundColor: [
        'rgba(255, 99, 132, 0.2)',
        'rgba(54, 162, 235, 0.2)',
        'rgba(255, 206, 86, 0.2)'
      ],
      borderColor: [
        'rgba(255,99,132,1)',
        'rgba(54, 162, 235, 1)',
        'rgba(255, 206, 86, 1)'
      ],
      borderWidth: 1
    }]
  }
});
```

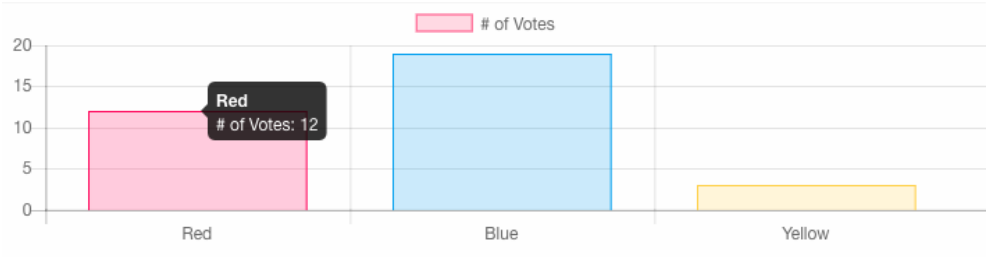

Figure 2.6: View result.

- Result:

  See the result on Figure 2.6

There are a lot of libraries that allows you to render charts or graphs on a web page[10] such as *C3.js*, *Plotly.js*, etc. In this project we used ChartJS because it adapted well to the project requirements.

2.12 SASS

"**Sass** is a scripting language that is interpreted or compiled into *Cascading Style Sheets* (CSS)". Sass uses two types of syntax. The first is original syntax, similar to *Haml*[11]. The second is *SCSS*, a syntaxt that uses blocks formatting like *CSS*

2.12.1 Features

- **Variables**:

  Sass allows defining variables using a dollar sign ($).

```
1 $primary-color: #3bbfce;
2
3 .content-navigation {
4   border-color: $primary-color;
5   color: darken($primary-color, 10%);
6 }
```

[10] https://blog.sicara.com/compare-best-javascript-chart-libraries-2017-89fbe8cb112d
[11] https://en.wikipedia.org/wiki/Haml

- **Block Nesting**:

  Sass allows nesting different style blocks, this a improves the code structure, makes it more readable and saves lines of code.

```
1 table.hl {
2   margin: 2em 0;
3   td.ln {
4     text-align: right;
5   }
6 }
```

- **Loops**:

  With Sass you can create loops of style blocks, this feature helps to save code when you have similar *id's* or *classes*. Her is an example:

```
1 $squareCount: 3
2 @for $i from 1 through $squareCount
3   #square-#{$i}
4    background-color: red
5    width: 50px * $i
6    height: 120px / $i
```

- There are more Sass features such as *arguments*, *selector inheritance*, ... All of these features can be found on the Sass documentation web page[12].

2.13 Flexbox

"The *CSS3 Flexible* Box, or **Flexbox**, is a layout mode intended to accommodate different screen sizes and different display devices". Flexbox is very useful when you have to organize your application view, so for example, sticking a *footer* to the bottom of a page, designing a *navigation bar*, creating a custom grid, ... is much easier using Flexbox.

[12] http://sass-lang.com/documentation/file.SASS_REFERENCE.html

Figure 2.7: no-wrap example.

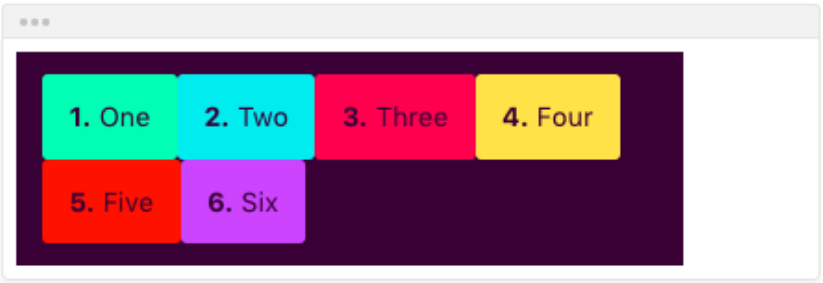

Figure 2.8: wrap example.

Related to the Flexbox concept, it allows to manage an element width and the height to improve its fitting in the available space on the display regardless of the device that is being used. This elements are shrunken to prevent overflow or expanded to fill remaining space.

2.13.1 Usage Examples

In order to use Flexbox styles, we have to apply the style "*display: flex*" in our container element. Let's see a few Flexbox usage examples.

- **flex-wrap**:

  Defines if flexbox items appear on a single line or on multiple lines within a flexbox container.

```
1 flex-wrap: nowrap;
2 flex-wrap: wrap;
```

- **justify-content**:

  Defines how flexbox items are aligned according to the main axis, within a flexbox container.

Figure 2.9: flex-end example.

Figure 2.10: center example.

```
1 justify-content: flex-end;
2 justify-content: center;
```

Chapter 3

Development

3.1 SCRUM Methodology

SCRUM is a work methodology designed mainly for *software development*. The main philosophy of this methodology consists in subdividing the work into groups of smaller tasks, each group is called "*Sprint*".

"Scrum is an iterative and incremental agile software development framework for managing product development".

A key principle of Scrum is the dual recognition:

- Customers decision may change during the sprint development process (requirements volatility).
- There will be unpredictable challenges—for which a predictive or planned approach is not suited.

In the *SCRUM* framework we can find three main roles:

- **Product owner**: Ensures that the team delivers value to the business and represents both the product's stakeholders and the voice of the customer. The product owner manage the work process defining the product, adding stories to the backlog and prioritizing work.
- **Development team**: The team is the one that completes the tasks of a Sprint, building the product increments.

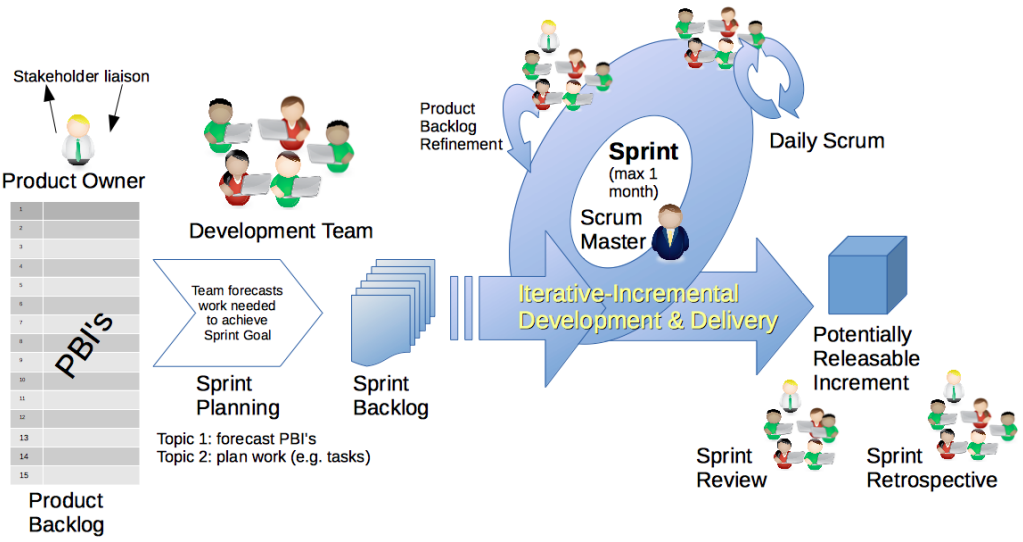

Figure 3.1: SCRUM Framework

- **SCRUM master**: Removes the impediments, allowing the team to deliver product goals. The SCRUM master filters all distracting influences from the development team.

On Figure 3.1 we can see a general SCRUM framework.

3.2 Sprint 1

3.2.1 Key Objectives

The main goals or tasks for this sprint are:

- Basic Elasticsearch interface which allows to visualize a simple metric. The data must come from Elastcsearch through an API request.
- Include more aggregations options and Index changing functionality.

3.2.2 Implementing a simple metric request

We are at the beginning of this project, the first thing we are going to do is to connect our Angular project to Elasticsearch. This can be achieved by installing the Elasticsearch library[1]

[1] `https://github.com/elastic/elasticsearch-js`

for Angular. This library will allow us to make requests to Elasticsearch through its javascript API.

We need an Elasticsearch client instance, this instance will give us the access to the Elasticsearch API. Elasticsearch, by default, it listens on the '9200' port, so we'll need to specify it too.

```
export class Elasticsearch {
  public clientElasticsearch: Client;
  constructor() {
    this.clientElasticsearch = new Client({
      host: 'localhost:9200',
      log: 'trace' // This is for debug purposes
    });
  }
...
}
```

Listing 3.1: Client Insance

This Elasticsearch class is created as an Angular **service**. In Angular, services are usually used for fetching data from the outside, this is what we are trying to achieve.

The next thing we are going to do is to request a simple "count" from Elasticsearch. In order to get this, we need to specify an Elasticsearch **index**.

```
...
public count(index): PromiseLike<number> {
  return this.clientElasticsearch.count({
      index: index
  }).then(
    response => response.count,
    this.handleError
  );
}
...
```

Now we just need to display the response.

```
// Html
<button (click)="displayData()">Display</button>
Count: <span>{{data}}</span>

// Typescript
...
data: number = 0;
...
displayData(): void {
  this.elasticsearch.count('bank')
  .then(count => this.data = count);
}
...
```

3.2.3 Adding aggregations and dynamic index

First, in order to fetch the available indexes we are going to use the "cat" request, supported by the javascript API.

```
...
public getIndices(): PromiseLike<string[]>{
     return this.clientElasticsearch.cat.indices({
      format: 'json'
     })
...
```

This method returns a JSON object, as we have specified in the "format" field. So we just need to parse this JSON response in order to build our indexes array. The next step is to retrieve all "number" fields for each index, since we are not going to implement aggregations that need another field type yet.

```
...
public getIndexNumFields(index): PromiseLike<string[]> {
    return this.map(index).then(function(response){
```

```
4 ...
```

The map method returns the index mappings, which we need to parse to build our "number fields" array.

The **PromiseLike** class is very useful because allows us to wait until the request is finished, in order to execute the appropriate callback function and manage the retrieved data asynchronously.

For now, we are going to include functionality for aggregations that works with fields of type "number" only. To achieve this we are going to use the "search" request provided by the Elasticsearch javascript API. This request returns a JSON object which we need to parse in a different way depending on the aggregation we are requesting, in order to get a displayable result.

```
1 ...
2 // avg, sum, min, max, median, std_deviation, unique_count
3 public numFieldCalculation(index: string, aggs: any): PromiseLike<
     any>{
4   return this.clientElasticsearch.search({
5       "index": index,
6       "body": {"size": 0, "aggs": aggs}
7   })
8 ...
```

The final stage view at the end of this sprint is shown on Figure 3.2.

For a better understanding a general architecture for this sprint is shown on Figure 3.3.

3.3 Sprint 2

3.3.1 Key Objectives

The main goals or tasks for this sprint are:

- Finish metrics functionality implementation.
- Implement Data Table visualization functionality.

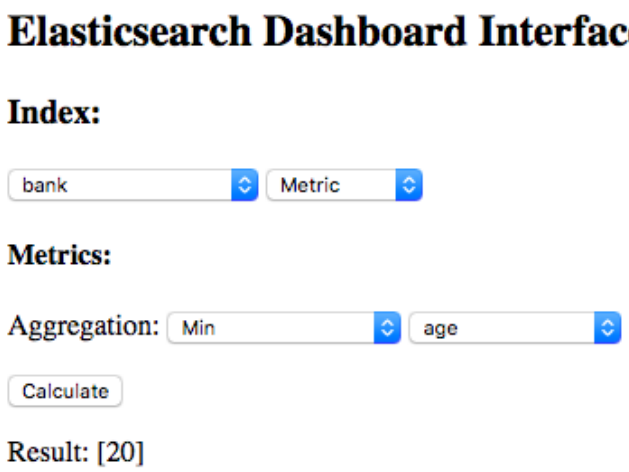

Figure 3.2: Final stage view - Sprint 1

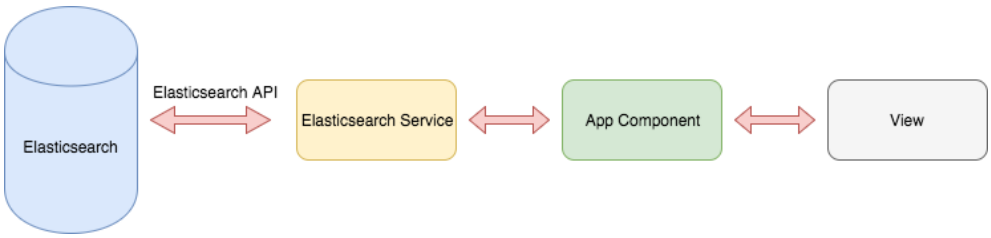

Figure 3.3: Sprint 1 architecture.

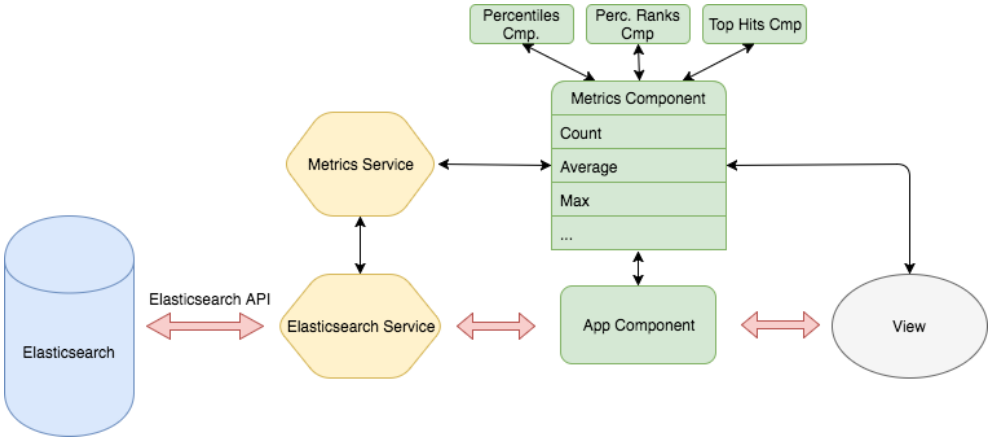

Figure 3.4: Sprint 2 architecture - Step 1.

- Load multiple metrics or buckets and changing between visualizations, both dynamically.
- Allow to load and save visualizations.

3.3.2 Finishing metrics

Following the same logic as we did with the previous aggregations, we implement 'Percentiles', 'Percentile Ranks' and 'Top Hit' aggregations. The complexity of this aggregations it is a little bit higher because this aggregations need more data in order to perform the request.

For this reason, we are going to have individual components for each one of this aggregations and a general *metrics component* for the other aggregations since they share the main calculation logic.

The architecture on Figure 3.4 shows how remains the general structure.

3.3.3 Implementing "Data Table" feature

In relation with the "Data Table" visualization implementation we followed the same structure as with the "Metric" visualization. The main difference here is the way we structure the data in order to display it in a table.

Another important difference with the "Metric" visualization is the introduction of Elasticsearch **buckets**. Because of the way "Data Table" visualizations works, Elasticsearch returns the requested data in groups of results, depending on the "bucket aggregation" that has been

used. For this project we are going to use two different bucket aggregations between all the possible aggregations:

- Histogram
- Range

Here the complexity of the requests grows significantly. Because of this complexity, we are going to use a javascript library called "**bodybuilder**"[2]. As we could understand by its name, this library helps us to build the request bodies. For example if we want to make a simple count over an index, and retrieve the data as "histogram buckets" with an interval:

```
// Builder process
bodybuilder().aggregation('histogram', null, 'bucket_1', {
    field: 'age',
  interval: 5
}).build()

// Result
{
  "aggs": {
    "bucket_1": {
      "histogram": {
        "field": "age",
        "interval": 5
      }
    }
  }
}
```

[2] https://github.com/danpaz/bodybuilder

3.3.4 Load multiple metrics or buckets and changing between visualizations, both dynamically

During the implementation of this project we found several issues using the Angular live-cycle hook in order to load, create, delete or modify components dynamically, specially when the component had a complex structure. For this reason we decided to have a dedicated component that allows us to complete all this tasks. We named this component "DynamicComponent".

This dynamic component works as a container for a single component or for multiple components.

Let's see the component's API:

```
1  // This method allows us to insert one or more components, it
       requires:
2  // - uniqueId: to store the component in a map for future
       modifications or access.
3  // - inputs: an object with the component parameters values.
4  // - events: an array with the event names we want to listen from
       this component.
5  // - componentType: the component type we want to instantiate.
6  addComponent(uniqueId: string, inputs: any, events: Array<string>,
       componentType: any): any{};
7
8  // this method is already executed on the addComponent(), but we
       may need it after the component initialization to modify some
       component parameter
9  setInputs(uniqueId:string, inputs: any): void{};
10
11 // This method destroys a loaded component by its unique ID
12 destroyCmp(uniqueId: string): void{};
```

Due to the Angular live-cycle hook defects we have commented before, we need a more efficient way to set specific component parameters. Usually when a parent component needs to pass parameters to a child component, we get it by adding attributes on the child component selector, but due to the complexity of the bucket and metric components, the Angular live-

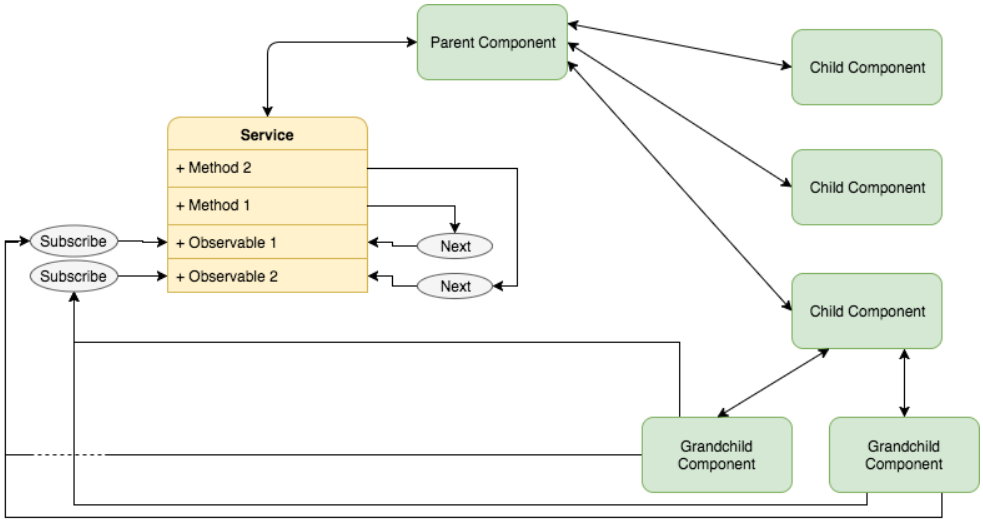

Figure 3.5: Components communication via Service.

cycle hook was acting erratically and braking the parent-child parameters synchronization. We needed to communicate with grandchildren components too and this was bringing on more synchronizing problems.

For this reason, we used a different technique for communicating a parent with its children or grandchildren components or even another component without direct relation. This approach for communicating components is made via **service**, we can see a diagram on Figure 3.5.

As we can see on Figure 3.5, a "Parent Component" communicates with its grandchildren employing **observable** variables declared on a **service**, each time a method executes "**next**" on a observable, a callback function is executed on each grandchild that was subscribed to this observable accessing the data the sent data.

To give an example, we can see this technique in the project applied for the components:

- **VisualizationsComponent**: acting as a "Parent Component".
- **VisualizationsService**: acting as the communicating service.
- **BucketsComponent**: acting as a "Grandchild Component".

Another advantage of this technique is found when we need a bidirectional communication between components, in this case we wouldn't need *input* variables or component events.

3.3.5 Loading and saving visualizations

At this point we are able to create two types of visualizations, "Metric" and "Data Table", but we are still not able to store them in Elasticsearch neither loading saved visualizations.

To achieve this features we have to create a new Elasticsearch index which will store all visualizations data. The name of this index we'll be "sakura".

Now we need a *type* which will define what data represents our visualizations.

Our index mapping looks like this:

```
visualization: {
    properties: {
        kibanaSavedObjectMeta: {
            properties: {
                searchSourceJSON: {
                    type: "text",
                    fields: {
                        keyword: {
                            type: "keyword",
                            ignore_above: 256
                        }
                    }
                }
            }
        },
        title: { type: "text" },
        visState: { type: "text" }
    }
}
```

Each field stores data necessary in order to load a visualization:

- **kibanaSavedObjectMeta**: data related with the visualization index.
- **title**: visualization's title.
- **visState**: data related with the aggregations.

Once we've created the visualization object, the saving process consist of 2 steps:

- Check if there is already a visualization on Elasticsearch with the same id (title).
- If there is already a visualization, update it, otherwise create a new one.

In Elasticsearch we use the "search" action in order to look for a document:

```
this.cli.search({
    "index": '.sakura',
    "type": type,
    "body": body
})
```

In this case the type is "visualization" and the body is build in the following way:

```
bodybuilder().filter('term', '_id', title).build();
// Returns:
{
  "query": {
    "bool": {
      "filter": {
        "term": { "_id": title }
      }
    }
  }
}
```

To create a document we use the "create" action:

```
this.cli.create({
  index: index,
  type: type,
  id: id,
  body: body
})
```

And we use "update" to modify an existing document:

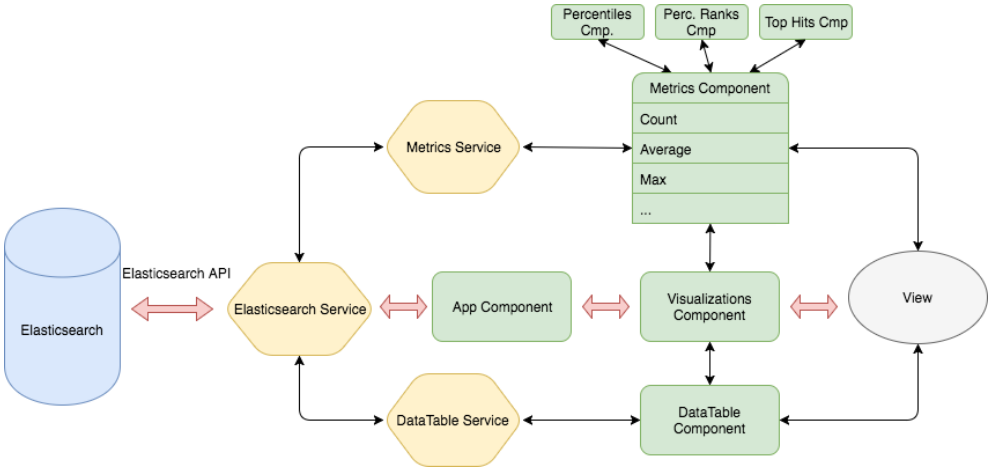

Figure 3.6: Sprint 2 final structure.

```
this.cli.update({
  index: '.sakura',
  type: type,
  id: id,
  body: { doc: doc }
})
```

The general structure at the end of this sprint is shown on Figure 3.6.

3.4 Sprint 3

3.4.1 Key Objectives

The main goals or tasks for this sprint are:

- Implement "Pie Chart" visualization.
- Implement "Bar Chart" visualization.
- Design the application user interface.
- Implement deleting feature for visualizations.

3.4.2 Implementing the "Pie Chart"

For this visualization we'll need a library that provides us chart features and allows us to display a "Pie Chart". In this case we decide to use *ChartJS*.

ChartJS provides a dedicated module for Angular but it has some defects and limitations and still needs improvements, for this reason we decide to import the ChartJS library directly to the HTML and arrange each chart to the HTML through *jQuery*.

For this visualizations, the process of request body building, and the process of response parsing are much higher compared with the visualizations already implemented.

The request body has the following format:

```
-> aggs
    ++ bucket_1
        ** aggs
            >> metric_aggregation
            >> bucket_2
                ## aggs
                    ~~ metric_aggregation
                    ~~ bucket_3
                        -- ... // keeps going on with until we
                           reach the desired number of levels
                ## bucket_aggregation_2
        ** bucket_aggregation_1
```

As we can see, each bucket object contains the selected metric, and this metric is the same for every bucket.

The Elasticsearch response object doesn't fit at all with the way we want to display the received data, so we need to parse it into a more "friendly" object in order to make it easy creating the required data for the render process with ChartJS. The parsed object will have the following format:

```
// This map contains the bucket(level) id as key and the array of
   results as value.
-> Map<String, Array<any>>
    ++ bucket_1
```

```
        ** [{result_obj_1}, {result_obj_2}, {result_obj_3}, ...]
    ++ bucket_2
        ** [{result_obj_1}, {result_obj_2}, {result_obj_3}, ...]
    ...
```

Listing 3.2: Custom results object

To display a "Pie Chart" we have a canvas element which we have to reference from our component.

```
<canvas baseChart id="myChart"></canvas>
```

The chart object is formed by the following fields:

- **type**: In this case this is 'pie'.
- **data**: This will contain our **datasets** with the pie data defining each one of the different levels that forms our "Pie Chart".
- **options**: This defines our chart configuration an characteristics such us responsiveness, tooltip customization, etc.

Each **dataset** defines a different level on our "Pie Chart" and is formed by the following fields:

- **data**: An array of values where each value appertain to a slice of the current level. Examples of this array are shown on code snippet 3.6.
- **backgroundColor**: An array of hexadecimal colors for each slice of the current level. This array is created randomly, the way we created this array is keeping and array of colors that are already displayed and calculating a given distance between hexadecimal colors in a decimal form, thus we ensure that all colors will differ one from each other.
- **labels**: An array of labels with value information for each slice of the current level. Here we had to use a library named "text-table". The reason we need this library is because ChartJS doesn't take *HTML* code in order to display a label but it takes a simple *String*, in this case we need to display a table within the label, since each row will represent an upper level, so this library parses an array of arrays and convert it into a table in a *String* format with "line breaks".

3.4.3 Implementing the "Bar Chart"

For this visualization we are also using ChartJS.

The request and response objects have the same complexity as the request and response objects for the "Pie Chart" visualization.

For the "Bar Chart" visualization, the request body is not as complex as the "Pie Chart" visualization request but still has some complexity. Let's see its format:

```
-> aggs
    ++ bucket_1
        ** aggs
            ~~ metric_aggregation_1
            ~~ metric_aggregation_2
            ~~ ...
        ** bucket_aggregation_1
```

Listing 3.3: Bar Chart request body format

As we can see there is just a single bucket and over it we are going to calculate one or more metric aggregations.

In this case we have to parse the response object into an object with a more "friendly" data structure like we did before with the "Pie Chart" visualization response. This new object has the following format:

```
// This map contains the metric result label as key and the array
   of results as value.
-> Map<String, Array<any>>
    ++ metric_result_label_1
        ** [{result_obj_1}, {result_obj_2}, {result_obj_3}, ...]
    ++ metric_result_label_2
        ** [{result_obj_1}, {result_obj_2}, {result_obj_3}, ...]
    ...
```

Listing 3.4: Custom results object

Each result object for a metric aggregation appertains to a bucket value, so that if we have 5 results for the requested bucket, we'll have 5 bars containing a single result from each requested

metric.

To display a "Bar Chart" we are going to use the same method as we did before with the "Pie Chart" visualization.

The *Chart* object contains the same fields we mentioned earlier, but now we set the *type* to 'bar'.

The *dataset* object is now much simpler, now it only contains information for a single result, so this is how remains:

- **data**: An array of values for each slice of the current level. Examples of this array are shown on code snippet 3.6.
- **backgroundColor**: A single hexadecimal color. This will be calculated randomly using the same logic as we did with the "Pie Chart".
- **labels**: A single *string* value which identifies the current values collection.

3.4.4 Designing a user interface

We have reached a point on the development process where it would be great to have a more intuitive, colorful and dynamic user interface.

For now we only have one view, "Visualizations", so we are going to design a user interface thinking on elements that we already have and their functionality.

On Figure 3.7 we can see a mock-up schema we want to get implemented.

To get all elements in their right position, such as the nav-bar or the metrics and buckets side-bar we are going to use **Flexbox**.

The tough task comes when we have to implement the metrics and buckets window. As we can see, it has a *scroll-bar* and the window is not positioned on the top of the page, it position depends always on the elements above it, and this elements can expand when we click on them, so the metrics and buckets window needs a dynamic height.

We achieve this dynamic height by employing two things: Angular dynamic attributes and *jQuery*.

As we can see bellow, the attribute takes a dynamic value through the *getHeight()* function, this function receives the element *id* as a parameter in order to reference it later with *jQuery*:

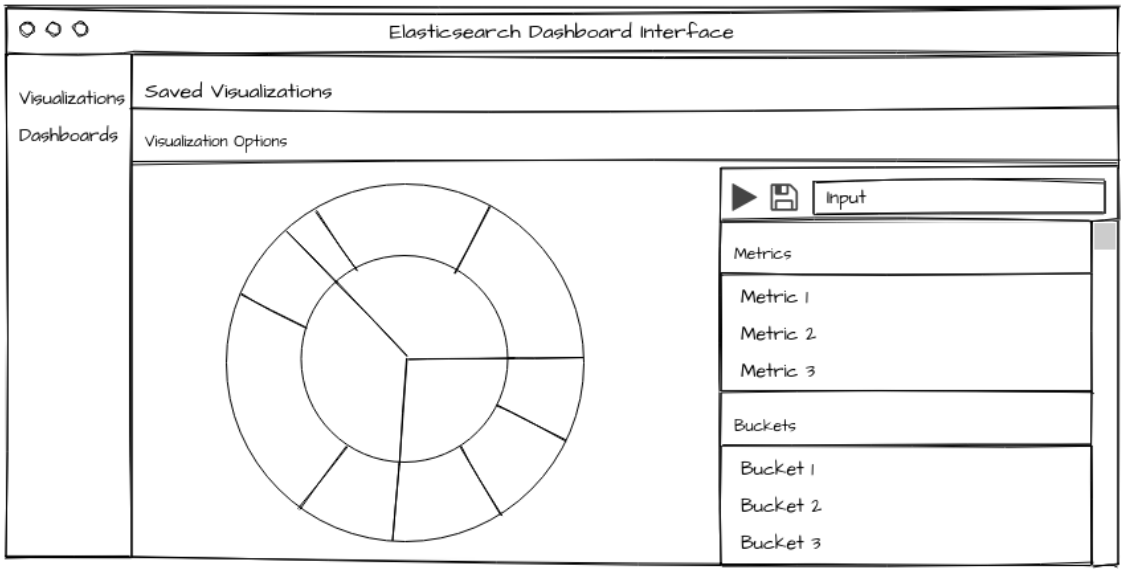

Figure 3.7: Use Interface mock-up

```
1 <div
2     id="metricsAndBuckets"
3     [style.height.px]="_getHeight('metricsAndBuckets')">
4         <metrics ...></metrics>
5         <buckets ...></buckets>
6 </div>
```

Listing 3.5: Dynamic height with Angular

Bellow we can see how the function calculates the element's height give the current window height and the current top position of our element:

```
1 private _getHeight(elemId: string): Number {
2   let configHeight = ($(window).height() - $('#' + elemId).
      position().top);
3   return configHeight;
4 }
```

Listing 3.6: Dynamic height with Angular

Setting this height to our element and giving it a "*overflow-x: scroll;*" style, we make this window "*scrollable*".

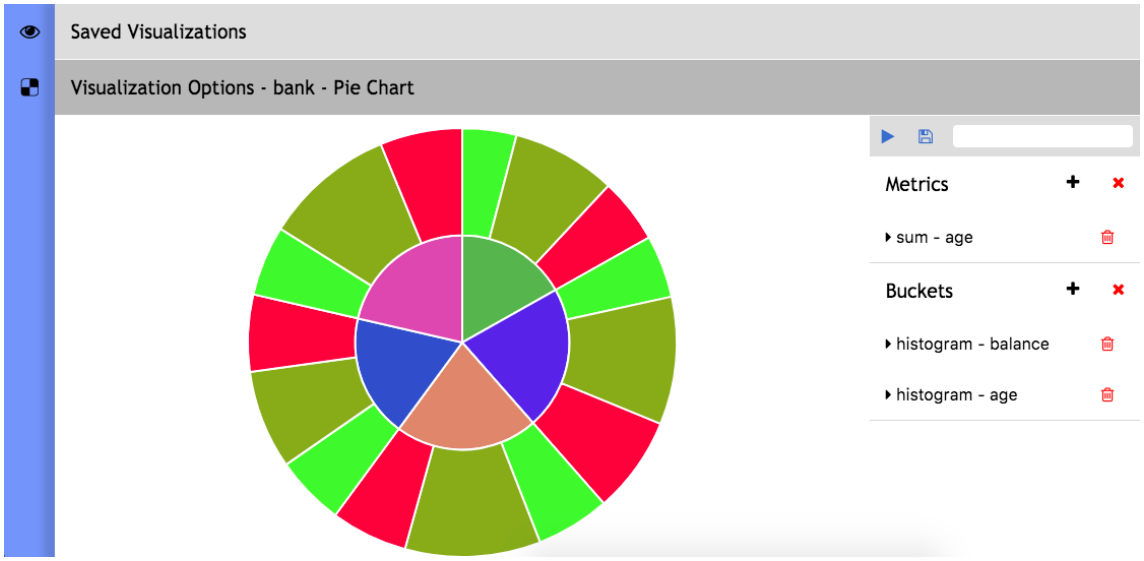

Figure 3.8: Final User Interface

To achieve a expandable panels for "Saved Visualizations" and "Visualization Options" we used a dynamic style and a *click* event, this process is similar to the dynamic height process we have shown before.

All icons are extracted from the *font-awesome* library. On Figure 3.8 we can see a final view of our user interface, with the same visualization example as in the mock-up schema.

3.4.5 Deleting visualizations

To achieve the visualization deleting process, we are going to use another Elasticsearch action named "delete".

```
1 public deleteDoc(type: string, id: string): PromiseLike<any>{
2   return this.cli.delete({
3     index: '.sakura',
4     type: type,
5     id: id,
6     refresh: 'wait_for'
7   }).then(...
8 }
```

Listing 3.7: Calling the Elasticsearch "delete" action

As we can see, we have to pass the document "type" and the document "id". The "*refresh: 'wait_for'*" option allows as to wait until the action has finish in order to refresh the saved visualizations panel and make the deleted visualization disappear.

3.5 Sprint 4

3.5.1 Key Objectives

- Implement dashboard user interface capable of showing multiple saved visualizations.
- Implement saving, modifying and deleting features for dashboards .

3.5.2 Implementing the visualizations dashboard

Once we've finished all the "Visualizations" section, we are going to reuse our visualization components in order to make a "*customizable*" dashboard interface.

To implement our dashboard grid where our visualizations will be shown we are using an angular module named "*angular2gridster*"[3]. This module will allow us to:

- Drag and drop visualizations containers.
- Store the position, size and another useful data for each visualization container.

On Figure 3.9 we can see how this module fits in our dashboard.

The dashboard object has the following structure:

```
var obj = {
    title: String,
    w: Number,
    h: Number,
    dragAndDrop: Boolean,
    resizable: Boolean,
    visualization: any
}
```

Listing 3.8: Dashboard object structure

[3] https://github.com/swiety85/angular2gridster

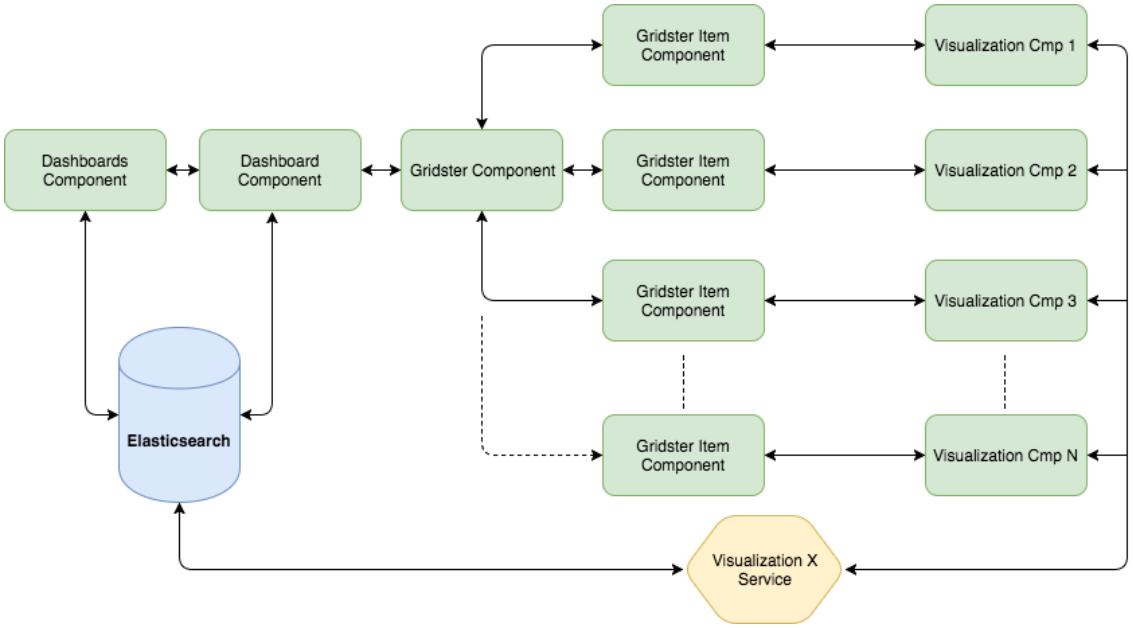

Figure 3.9: Dashboard section diagram

The visualization field stores the same visualization object that is stored on Elasticsearch.

3.5.3 Saving, editing and deleting dashboards

To implement all this features we have followed the same process as with the "*visualization*" type. We have reused the same methods from the "Elasticsearch Service".

In this case we have to create again another type like we did with visualizations. This time we'll name our type "dashboard" and it will be under our "sakura" index too. Our type mapping looks like this:

```
dashboard: {
    properties: {
        title: { type: "text" },
        widgetsJSON: { type: "text" }
    }
}
```

Listing 3.9: Elasticsearch "dashboard" type mapping

Each field has the following description:

- **title**: Dashboard title.

- **widgetJSON**: This is the dashboard object we have described.

Chapter 4

Results

4.1 Introduction

On the previous chapter we saw in detail how the web application was developed step by step. On this chapter we are going to explain how this application works, how to use it and its functionality.

4.2 User Guide

On this section we are going to explain one by one how the web application's different features works. First we are going to begin with the "Visualizations" feature, we are going to explain each type of visualization and how to build them, then we are going to explain the "Dashboard" section and how to build dashboards from different visualizations.

4.2.1 Visualizations

The first view we are getting when we open the application is the "Visualizations" section. We can see this view on Figure 4.1.

From this view, if we open the "Saved Visualizations" by clicking on its bar (Figure 4.2), we can either load a saved visualization or delete it by clicking o the trash icon.

Another option is to create a new visualization by opening the "Visualization Options" bar. To create a new visualization we must select an index about which we are going to calculate the available visualization types. This is shown on Figure 4.3.

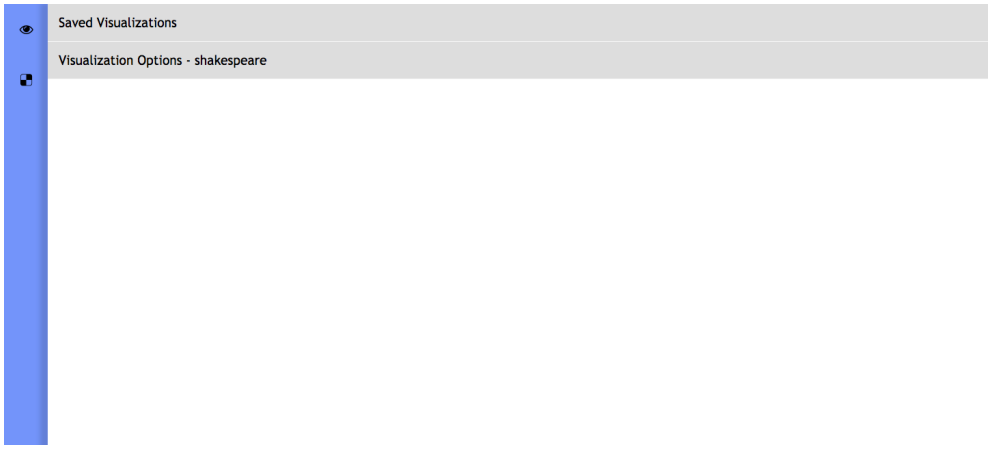

Figure 4.1: Visualizations section

Figure 4.2: Saved Visualizations panel

Let's see how to create each of the visualization types.

4.2.1.1 Metric Visualization

A "Metric" visualization performs an operation over the index data an shows the result as a String, a Number or a collection of them. We can see an initial view of this "Metric Visualization" on Figure 4.4.

To add a metric operation to our visualization, we have to click on the "+" icon, this will add a "count" calculation over the selected index by default, and if we click the "play" icon will

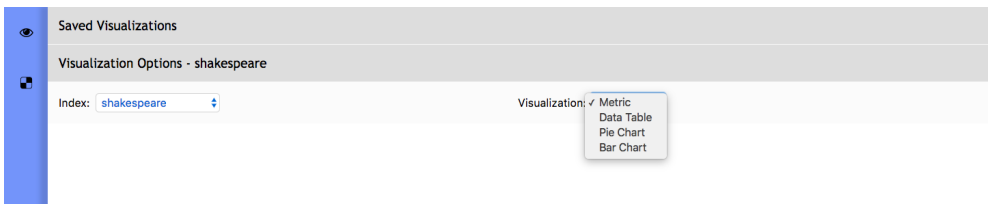

Figure 4.3: Visualization Options panel

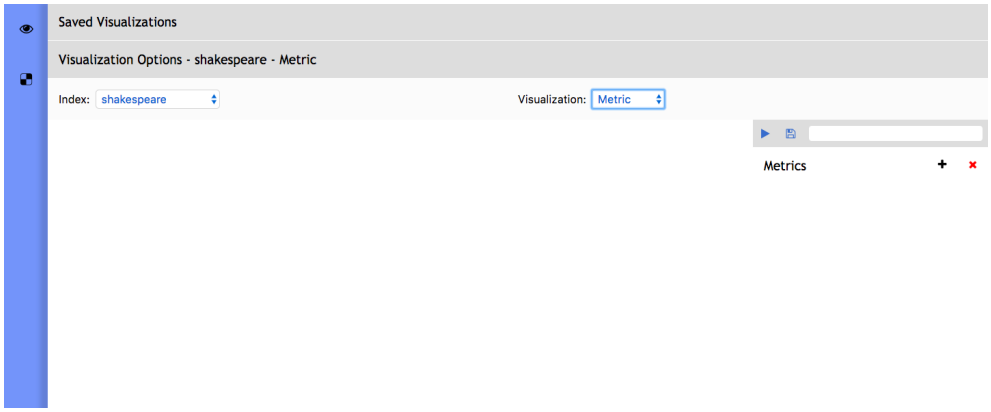

Figure 4.4: Initial view of a Metric Visualization

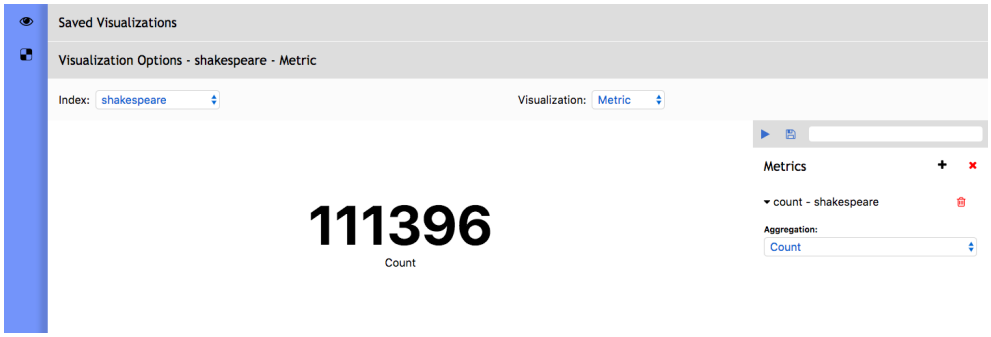

Figure 4.5: Metric "count" calculation

see the result on calculation result on the screen (Figure 4.5).

The operation description for each "aggregation" option is:

- **Count**: Counts the exact number of documents for a given index (Figure 4.5).
- **Average**: Calculates the average for a given field of a documents collection for a given index. This field's type must be "number" (Figure 4.6).
- **Sum**: Calculates the summation over a documents collection for a given field and index. This field's type must be "number" (Figure 4.7).
- **Min**: Calculates the minimum value for a given field of a documents collection for a given index. This field's type must be "number" (Figure 4.8).
- **Max**: Calculates the maximum value for a given field of a documents collection for a given index. This field's type must be "number" (Figure 4.9).

- **Median**: Calculates the median value for a given field of a documents collection for a given index. This field's type must be "number" (Figure 4.10).

- **Standard Deviation**: Calculates the standard deviation values for a given field of a documents collection for a given index. This field's type must be "number" (Figure 4.11).

- **Unique Count**: Calculates number of unique values for a given field of a documents collection for a given index. This field's type must be "number" (Figure 4.12).

- **Percentiles**: Calculates a multi-values percentiles result for a given field of a documents collection for a given index and for a given list of percentiles. This field's type must be "number" (Figure 4.13).

- **Percentile Ranks**: Calculates a multi-values percentiles result for a given field of a documents collection for a given index and for a given list of percentiles. This field's type must be "number". Percentile ranks show the percentage of documents which selected field's value is below certain value (Figure 4.14).

- **Top Hits**: This aggregation is very different from the others. Top Hits keeps track of the top matching documents depending on the selected options and fields. We have to select multiple options in order to perform the aggregation calculation (Figure 4.15). These fields are:

  - **Field**: This is the same as with the other aggregations, it indicates the document field over which we are going to perform the request.

  - **Aggregate With**: This indicates the way we are going to process the received hits, such as "summation", "maximum", "minimum", "average" or "concatenation". Except for the last type, all must be with a "number" type field.

  - **Size**: This indicates the maximum number of documents or hits we are going to receive on the response.

  - **Sort On**: This, again, is a document field. This field affects to the documents or hits we are going to receive in the way that these documents are going to be sorted by this field. If this field is a "number", the documents will be sorted "numerically", and if the field is a "string", the documents will be sorted alphabetically.

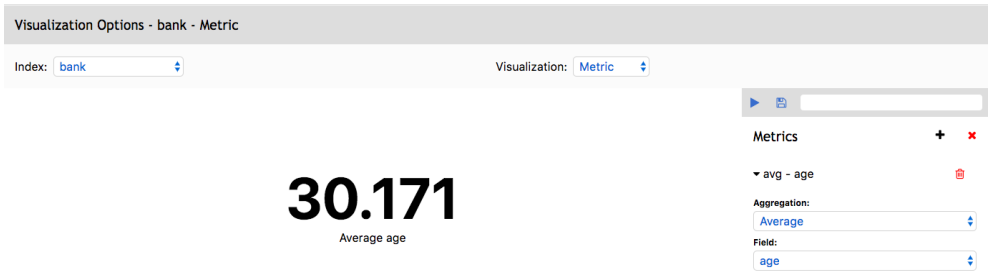

Figure 4.6: Metric "average" calculation

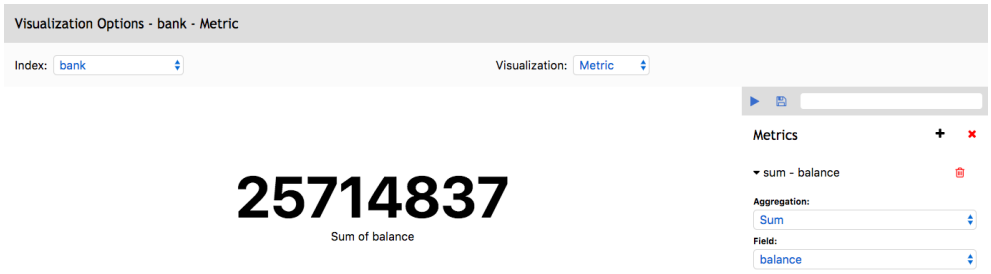

Figure 4.7: Metric "sum" calculation

– **Order**: This indicates how the documents will be sorted, i.e. ascending or descending.

4.2.1.2 Data Table Visualization

A "Data Table" visualization performs an operation over the index data an shows the result as a table. We can see an initial view of this "Data Table Visualization" on Figure 4.16.

As we can see, we have a new element on the configuration panel, "Buckets". This bucket aggregations defines a rule to create buckets of documents when making requests to Elastic-

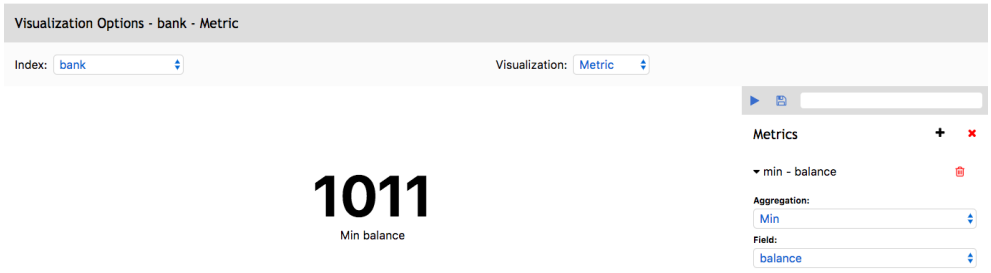

Figure 4.8: Metric "min" calculation

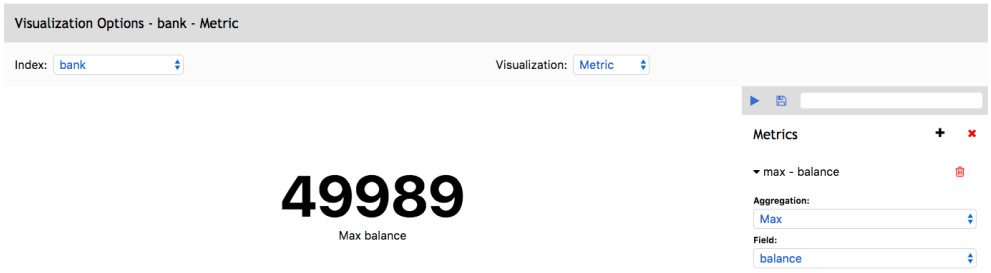

Figure 4.9: Metric "max" calculation

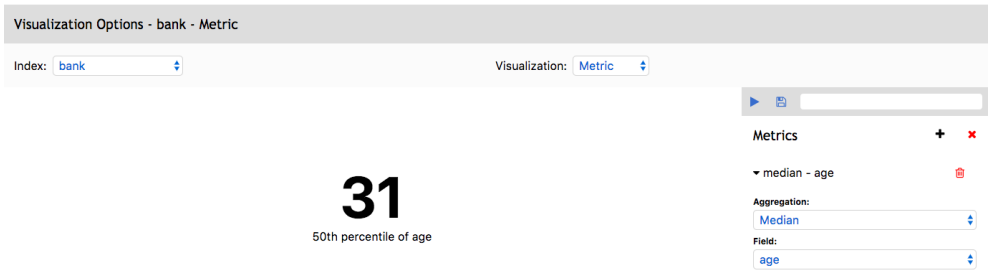

Figure 4.10: Metric "median" calculation

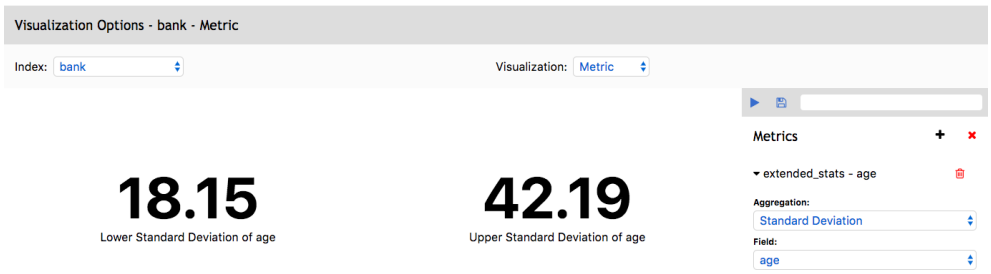

Figure 4.11: Metric "standard deviation" calculation

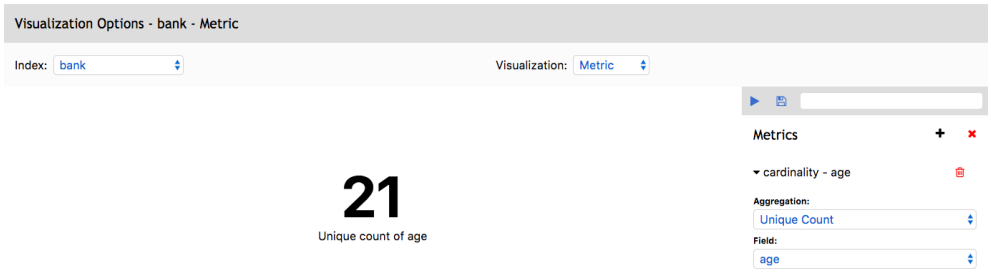

Figure 4.12: Metric "unique count" calculation

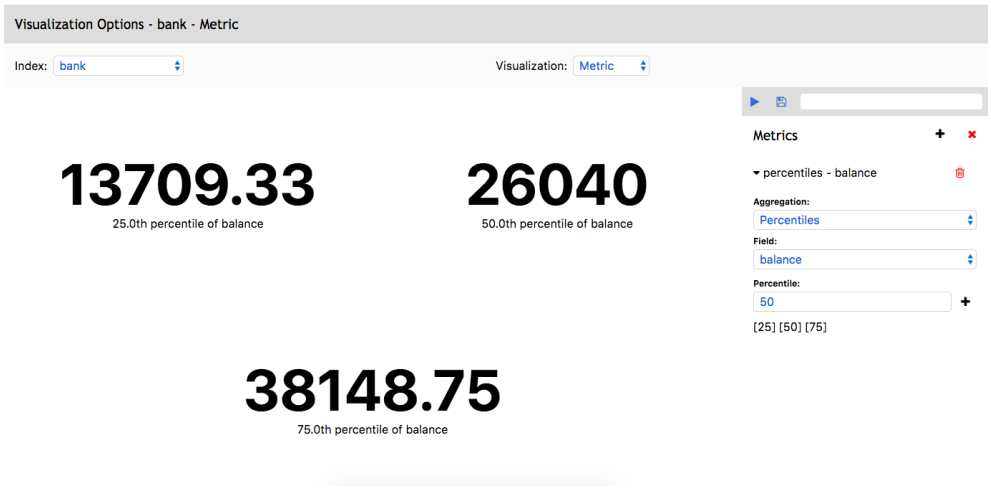

Figure 4.13: Metric "percentiles" calculation

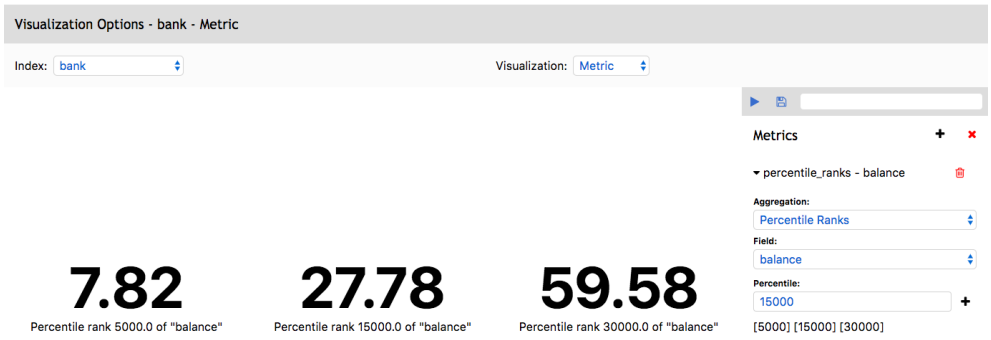

Figure 4.14: Metric "percentile ranks" calculation

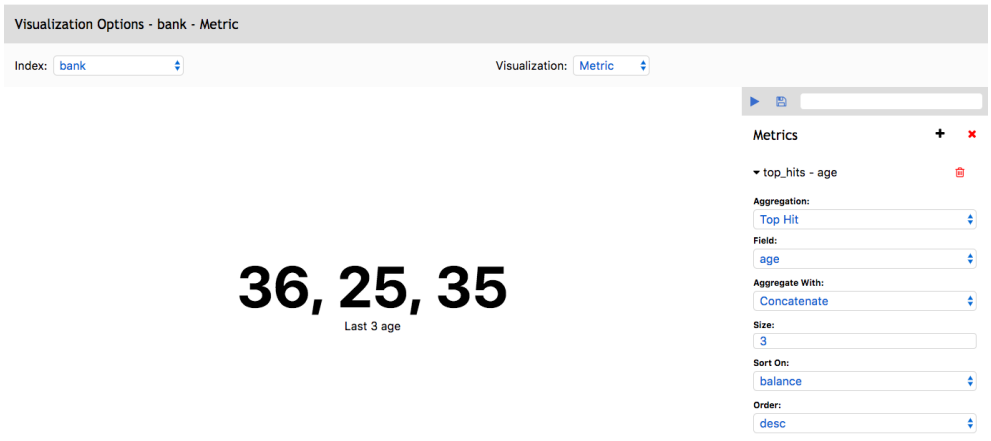

Figure 4.15: Metric "top hits" calculation

Figure 4.16: Data Table visualization initial view

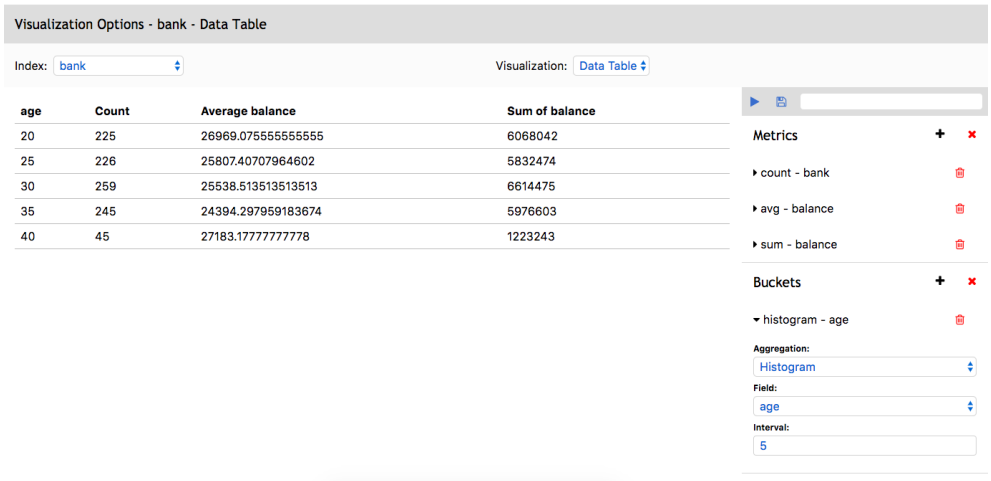

Figure 4.17: Data Table histogram example.

search. Now, the metrics will be performed over this buckets of documents.

We have two types of bucket aggregations:

- **Histogram**: This aggregation builds interval buckets of documents for a given numeric field and numeric interval value (Figure 4.17).
- **Range**: This aggregation builds range buckets of documents for a given numeric field and a collection of numeric ranges (Figure 4.18).

The table will have as many columns as metric results plus the number of buckets. And it will have as many rows as received hits over all buckets.

The table allows any number of metrics and any number of buckets. If we have more than one bucket, the bucket results will be created recursively (Figure 4.19).

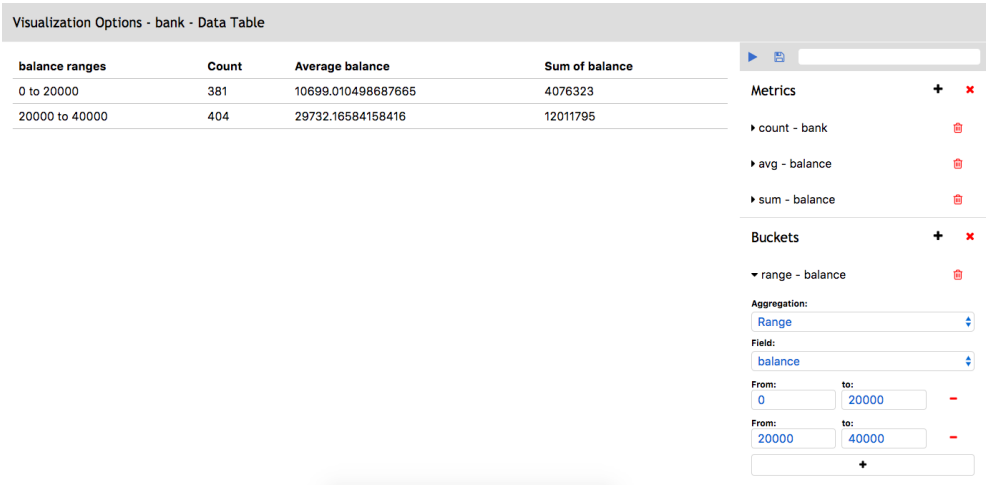

Figure 4.18: Data Table range example.

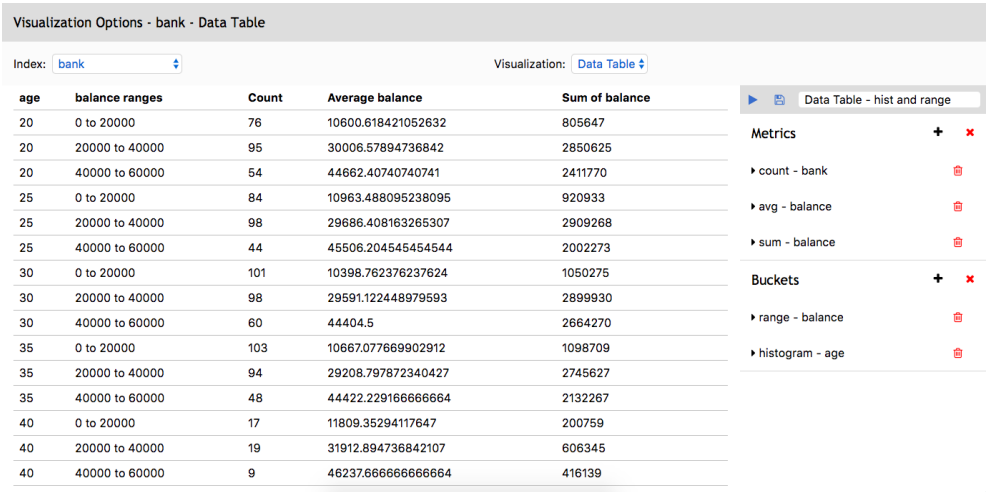

Figure 4.19: Data Table multiple buckets example.

Chapter 5

Conclusiones

5.1 Consecución de objetivos

Esta sección es la sección espejo de las dos primeras del capítulo de objetivos, donde se planteaba el objetivo general y se elaboraban los específicos.

Es aquí donde hay que debatir qué se ha conseguido y qué no. Cuando algo no se ha conseguido, se ha de justificar, en términos de qué problemas se han encontrado y qué medidas se han tomado para mitigar esos problemas.

5.2 Aplicación de lo aprendido

Aquí viene lo que has aprendido durante el Grado/Máster y que has aplicado en el TFG/TFM. Una buena idea es poner las asignaturas más relacionadas y comentar en un párrafo los conocimientos y habilidades puestos en práctica.

1. a
2. b

5.3 Lecciones aprendidas

Aquí viene lo que has aprendido en el Trabajo Fin de Grado/Máster.

1. a

2. b

5.4 Trabajos futuros

Ningún software se termina, así que aquí vienen ideas y funcionalidades que estaría bien tener implementadas en el futuro.

Es un apartado que sirve para dar ideas de cara a futuros TFGs/TFMs.

Apéndice A

Manual de usuario

Bibliography