UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

DATA VISUALIZATION SYSTEM FOR AN IOT PLATFORM WITH JAVASCRIPT FRAMEWORKS

ÁLVARO GARNICA NAVARRO

JUNIO 2019

TRABAJO DE FIN DE GRADO

Título:	Sistema de visualización de datos para una plataforma IoT con frameworks de Javascript
Título (inglés):	Data visualization system for an IoT platform with Javascript Frameworks
Autor:	Álvaro Garnica Navarro
Tutor:	Miguel Ángel López Peña
Ponente:	Carlos Ángel Iglesias Fernández
Departamento:	Departamento de Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	—
Vocal:	—
Secretario:	—
Suplente:	—

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

TRABAJO FIN DE GRADO

DATA VISUALIZATION SYSTEM FOR AN IOT PLATFORM WITH JAVASCRIPT FRAMEWORKS

Álvaro Garnica Navarro

Junio 2019

Resumen

Este documento es el resultado de un proyecto cuyo objetivo ha sido el diseño y la implementación de un sistema de visualización integral de uso sencillo para una plataforma de *Internet de las Cosas* (IoT).

En primer lugar, se realizó un estudio exhaustivo para determinar las tecnologías que mejor encajarían con los requisitos establecidos. La decisión final fue utilizar MEAN Stack con librerías y frameworks de Javascript.

Para ofrecer a la plataforma IoT un sistema de visualización completa, la visualización es separada en dos: visualización de datos de aplicación y visualización de datos de sistema. La primera permite al usuario visualizar todos los datos de los dispositivos conectados en la plataforma IoT en tiempo real con paneles de seis gráficos. La visualización de datos de sistema ofrece una perspectiva del despliegue de la topología de la plataforma, compuesta por servidores de proceso y enlaces de comunicación entre ellos.

Para conseguir esto se desarrolla una aplicación web que está integrada en la plataforma IoT. El lado cliente es implementado con Angular y utiliza un servidor Node.js con una API de Express para comunicarse con la base de datos. La base de datos utilizada es MongoDB que almacena los paneles. La visualización es llevada a cabo por diferentes librerías de Javascript: Leaflet, CanvasJS, Vis.js y D3.js.

Además, se ha implementado un módulo de gestión de topología IoT que le da al usuario la posibilidad de diseñar una topología nueva para la plataforma IoT.

Nuestro producto final es una aplicación web que ofrece un entorno para crear, administrar y visualizar datos a través de paneles. Este sistema de visualización de datos puede ser utilizado para mejorar el rendimiento de la plataforma.

Palabras clave: Internet de las Cosas, Plataforma, Visualización de Datos, Panel, Javascript, Aplicación Web, Visualización de Sistema, Topología

Abstract

This document is the result of a project whose objective has been the design and implementation of a comprehensive visualization system simple to use for an *Internet of Things* (IoT) platform.

In the first instance, a comprehensive study was conducted to determine which technologies fit better with the established requirements. The final decision was to use the MEAN Stack with Javascript libraries and frameworks.

To offer the IoT platform a complete visualization system, the visualization is separated in two: application data visualization and system data visualization. The first one allows the user to visualize all the data from the connected devices on the IoT platform in real time with dashboards of six charts. The system data visualization gives an insight of the deployment of the IoT platform, composed of data processing servers and communication links between them.

To achieve this it is developed a web application integrated in the IoT platform. The client side is implemented with Angular and uses a Node.js server with an Express API to communicate with the database. The database used is MongoDB and stores the dashboards. The visualization is carried out with different Javascript libraries: Leaflet, CanvasJS, Vis.js and D3.js.

In addition, it has been implemented an IoT topology management module that enables the user to design a new topology for the IoT platform.

Our end product is a web application which provides an environment to create, manage and visualize data through dashboards. This data visualization system can be used for data analysis to improve the platform's performance.

Keywords: Internet of Things, Platform, Data Visualization, Dashboard, Javascript, Web Application, System Visualization, Topology

Agradecimientos

Quería agradecer a Natalia, lo más bonito de mi vida, por apoyarme siempre y estar a mi lado en los peores momentos.

También a mis padres Gonzalo y María Jesús, que son para mí un gran ejemplo de trabajo, esfuerzo y constancia, por haberme inculcado estos valores.

Muchas gracias también a Miguel Ángel por ayudarme en todo lo que he necesitado, incluso en los fines de semana y festivos.

Por último, agradecer a Héctor y mis compañeros Heinrich y Daniel por las dudas más técnicas.

Muchas gracias a todos.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Context

In the last years, the society has been changing at a really high speed. The number of devices interacting with us is rising, as well as the data they collect. This new era of the *Internet of Things* (IoT) gives humans the possibility of sharing data with things and connecting with them.

The *Internet of Things* collects data from a huge number of sensors, which is useless in itself. The real issue is how the data is used. That is the reason why *Data Visualization* plays an important role in IoT. The data has to be converted into a format where it can be analyzed.

The knowledge obtained through data processing allows us to establish a sustainable society at a social level, as an economic level and as an environmental level. This evolution from raw data or data without processing, to knowledge and wisdom is reflected in the knowledge pyramid, also known as DIKW Hierarchy (Data, Information, Knowledge and Wisdom)[1] which can be seen in Figure 1.1.

Figure 1.1: DIKW Hierarchy

This project has been conducted in a scholarship in the company SATEC. As regards the above, this company in its research line SATEC 4.0 is implementing IoT solutions that improve the data processing performance. This company is participating in the European H2020 project RECAP[1] (Reliable Capacity Provisioning). It aims to develop the next generation of cloud, edge and fog computing capacity provisioning via targeted research advances in cloud infrastructure optimization, simulation and automation.

As a member of RECAP Consortium, SATEC is developing an advanced IoT platform that will allow a Fog/Edge/Cloud convergence for IoT developments[2]. Fog/Edge computing is a new technological paradigm which purpose is to process data near the data source in order to reduce latencies and save bandwidth. This IoT platform, among other features, will facilitate the development of vertical applications by providing an advanced visualization module.

For this reason, *Data Visualization* has been decided as the central point of this project with the development of a visualization system. All the data gathered will be represented in different views in that system which will be integrated in the IoT platform developed by SATEC inside the RECAP project.

[1]https://recap-project.eu/

1.2 Project goals

In essence, the main objective of this project is to offer the IoT platform a complete visualization system, easy to use and with no need of programming skills from the user. The visualization will be separated into two different: application data visualization and system data visualization. The first one will enable the user to visualize all the data from the connected devices on the IoT platform in real time. This visualization will be customizable as the user will be able to design dashboards of six charts. The second visualization will offer a complete insight of the deployment of the IoT platform.

To achieve this objective, we will follow the succeeding steps:

- Conduct a study of different technologies in order to choose the most suitable for this project.
- Gathering of functional requirements.
- Implement the environments to design templates and dashboards.
- Store the templates and dashboards in a database.
- Create the component for the dashboards visualization.
- Create the component for the system visualization.
- Test the project with the test scenarios.

1.3 Structure of this document

In this section we provide a brief explanation of the chapters included in this document. The structure is as follows:

Chapter 1: On the one hand, it will be explained the context in which this project is developed. On the other hand, the main objectives to be achieved will be indicated.

Chapter 2: We will explain the technologies involved in the implementation of this project as well as the reasons that led us to use them.

Chapter 3: First, the system actors will be differentiated. Then, it will be specified the functional requirements of the system with user stories.

Chapter 4: It will be given an insight of the architecture of the project, dividing it into three modules with different components and submodules.

Chapter 5: Shows the results obtained with the test scenarios and a discussion about these results.

Chapter 6: Discusses the conclusions drawn from this project, the achieved goals, some problems faced and suggestions to improve the system.

CHAPTER 2

Enabling Technologies

This chapter describes and gives an insight into the web-based technologies used in the project. Taking into account the requirements that will be specified in Chapter 3, the best solution to develop the visualization module was to use the MEAN Stack. Other solutions that were considered were Jupyter Notebooks and Thingsboard. The first of them was discarded because it requires programming skills from the user. Conversely, Thingsboard, which is especially designed for IoT solutions, doesn't fit with the requirements because of the lack of control of the code.

All the technologies involved in the MEAN Stack will be described, as well as the Javascript libraries used for the visualization.

2.1 MEAN Stack

MEAN is an abbreviation for MongoDB, Express, Angular and Node.js, and it aims to use only Javascript - driven solutions to develop the whole application [3]. This is a great advantage as the developer only has to learn Javascript to implement the whole application. Furthermore, the reason that led to this powerful stack was the possibility of implementing every requirement with no restrictions.

2.1.1 Angular

Angular[1] is a client-side MVC (Model View Controller) framework for web applications. It is open source and maintained by Google, which supposes a major advantage as there is a supportive community to help you with any problem regarding Angular. It is built entirely in Typescript, which is an extension of Javascript intended to enable easier development of large-scale Javascript applications [4].

This framework is used to create Single Page Applications, also known as SPAs. A SPA is a web application with one HTML file. The page changes its content dynamically by manipulating the DOM elements with Javascript.

The principal features of Angular are:

- **Cross platform:** learning one way to build applications, you can reuse the code to build applications for any deployment target.
- **Speed and performance:** the code is highly optimized for today's Javascript virtual machines.
- **Productivity:** build quickly with declarative templates. You can also create your own components or use existing components.

An Angular app is defined by a set of NgModules, which provide a compilation context for components. Views in Angular are defined by components, and they are screen elements that change depending on the program logic and data. To perform an specific action, components use services that can be injected and reused by other components. To differentiate between components and services, that are both simply classes, there are used decorators which provide metadata that tells Angular how to use them.

Additionally, to enhance web application's appearance, there is a UI Component Library for Angular called Angular Material[2]. Its components help in building attractive and functional web applications. Also, it follows modern web design principles like device independence or browser portability. The components from this library used in this project are: *MatInput*, *MatToolbar*, *MatSelect*, *MatButton*, *MatList*, *MatSidenav*, *MatChips*, *MatRadio*, *MatTooltip* and *MatCard*. Apart from these components, it is used the *Drag and Drop* interacting from the Angular Material CDK. These are explained below in subsequent chapters.

[1] https://angular.io/
[2] https://material.angular.io/

2.1.2 Node.js

Node.js[3] is a Javascript runtime designed to build scalable network applications. It uses an asynchronous event-driven, non blocking input/output (I/O) model making it lightweight and efficient, ideally suited for DIRTy. This stands for *data-intensive real-time* applications [5]. Event-driven means that you register code to specific events and it will be executed when the events are emitted.

To achieve more complex functionality, Node.js uses a module system called NPM[4] (Node Package Manager). It allows to install third-party modules and it offers a CLI tool to manage your packages. These packages can be local or global:

- **Local:** the packages are located in a local folder called `node_modules` inside your application folder.
- **Global:** the packages are all in a single place in your system.

Both types of packages are required in the code the same way:

```
require('package-name');
```

To manage package dependencies NPM uses a configuration file named `package.json` in the root folder of the application. It is a JSON file with the following structure:

```
{
    "name": "MEAN",
    "version":"0.0.1",
    "dependencies": {
        "express":"^4.16.4"
    }
}
```

One very useful package used to implement the web server is Express.

2.1.3 Express

The Express framework provides a set of features of web and mobile applications. It acts as a light layer atop the Node.js web server, making it easier to develop Node.js web applications

[3] https://nodejs.org
[4] https://npmjs.com

[6]. This is achieved with middlewares and routing.

Middlewares or MW are functions to handle HTTP transactions. It is possible to call multiple MW instead of just one request handler, so that each one deals with a small chunk of the work. The routing manages the MW that are executed depending on what URL and HTTP method are sent by the client.

2.1.4 MongoDB

The last component of the MEAN Stack is MongoDB[5], responsible for data storage. MongoDB is a document-based NoSQL database solution. The main goals of MongoDB [3] are:

- Combine the robustness of a relational database with the fast throughput of distributed key-value data stores.
- To support web application development in the form of JSON outputs.

The first goal is accomplished thanks to ad hoc queries. With them the database will respond to dynamically structured queries without a predefined query. To do this, documents are indexed and they use a unique query language.

The second goal is achieved by storing data in BSON. This is a binary-encoded serialization similar to JSON, used to store documents and make remote procedure calls. BSON documents are a simple data structure representation of objects and arrays in a key-value format. These documents support all of the JSON specific data types along with other data types. In this format, the field used as primary key is `_id`. A BSON representation would look like the following:

```
{
    "_id": "52d12340d4t02b67b72ac566",
    "title":"Example of BSON format",
    "content": "Here goes some content"
}
```

The use of this format makes MongoDB an ideal option in the development of web applications, as it is really simple to work with data. Because of this, MongoDB is one of the fastest-growing databases in the world.

[5] https://mongodb.com

MongoDB is a schemaless database whose data needs to be modeled. For this purpose it is used Mongoose[6]. It is a Node.js module that adds MongoDB support to the Express application. It provides a schema-based solution to model application data and save it as MongoDB documents.

2.2 Bootstrap

Bootstrap[7] is an open source framework which was initially released by Twitter employees. In that company, engineers used almost any library that met with the front-end requirements. Because of that, they designed this framework to standardize the front-end toolsets [7]. It follows the approach "mobile-first design". This is progressive enhancement, it designs for mobiles and then upgrades to other devices.

In its official site, in the *Examples* section[8] are some templates of components which can be easily reused.

The major advantage of Bootstrap, which is implemented in this project, is the Grid System. This system utilizes twelve columns to organize the elements in the screen depending on the device as can be seen in Figure 2.1. This feature is a characteristic of the adaptative design.

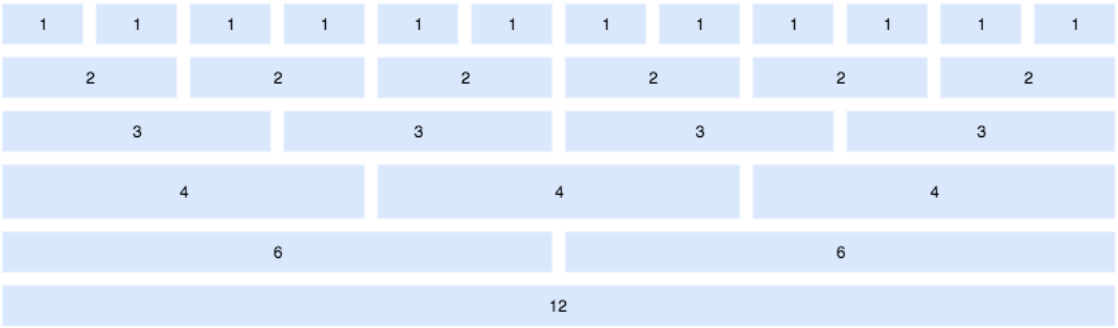

Figure 2.1: Bootstrap Grid System

To implement the Grid System, there are some predefined HTML classes:

- **.col-xl-:** for large desktops with a width of 1200px or higher.

[6]https://mongoosejs.com
[7]https://getbootstrap.com
[8]https://getbootstrap.com/docs/4.3/examples

- **.col-lg-:** for desktops with a width between 992px and 1200px.
- **.col-md-:** for tablets with a width between 768px and 992px.
- **.col-md-:** for landscape phones with a width between 576px and 768px.
- **.col-md-:** for portrait phones with a width lower than 576px.

Apart from this framework, we have used a variation: Angular Bootstrap. This library is installed via NPM and offers all the Bootstrap widgets and components adapted to Angular. The four components used in this project are *NgbAlerts*, *NgbPanels*, *NgbTabs* and *NgbModal*.

2.3 Visualization libraries

In this project, the visualization is distinguished between application data visualization and the system data visualization. In this section it will be described the different libraries used for each visualization.

2.3.1 CanvasJS

The application data visualization is carried out by CanvasJS[9], which is a responsive HTML5 and Javascript charting library. It offers more than 30 chart types that can be used in a similar way. In addition, CanvasJS has a great performance, being able to render 100,000 data-points in less than half a second.

This library can be integrated with the following technologies:

- **Front End:** React, Angular, Javascript and jQuery.
- **Back End:** PHP, ASP.NET MVC, Spring MVC and JSP.

When looking up for a chart technology for the application data visualization other libraries were considered, such as D3.js or Angular Chart. On one hand, D3.js is the most complete charting library including more than 100 types of charts, but has a steep leaning curve. On the other hand, Angular Chart, which is based on ChartJS, is relatively simple to use, but it offers very few types of charts.

[9] https://canvasjs.com

The reason that led CanvasJS to be chosen was the variety of chart types it offers besides its simplicity. CanvasJS has a simple and intuitive API in Javascript that allows to create charts in just a few lines of code. Chart type can be changed just by changing a word. The minimum inputs that have to be defined to create an Angular chart are the following:

- **Title:** a title for the chart, placed by default centered on top.
- **Data:** data to be represented and the chart type to be used. It has two properties:
  - **type:** chart type.
  - **dataPoints:** array with the data to represent.

There are other inputs that can be used to customize the different elements in a CanvasJS chart:

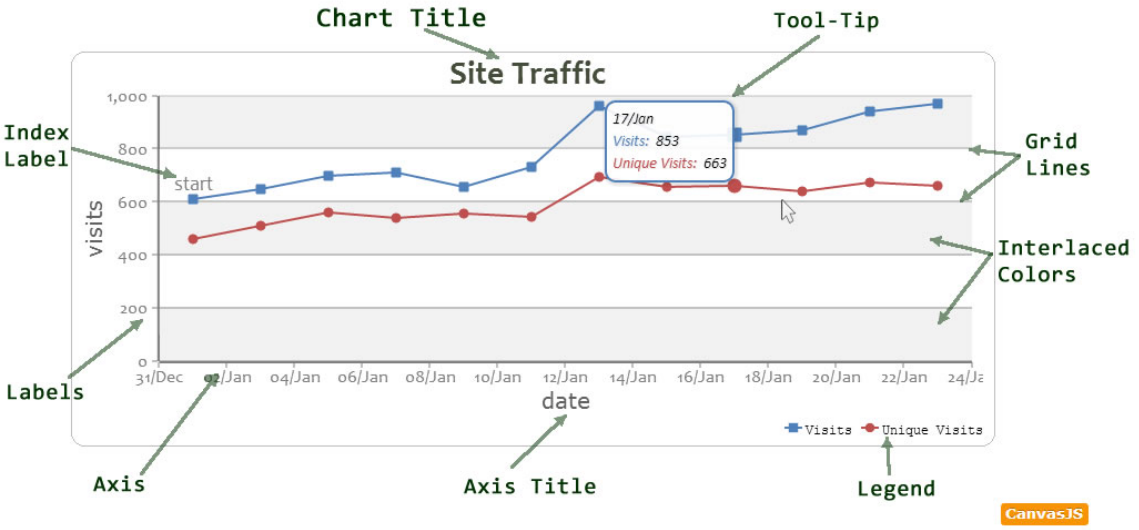

Figure 2.2: Chart elements in CanvasJS

2.3.2 Leaflet

Leaflet[10] is an open-source Javascript library for interactive maps. It is used to implement maps and heat maps in the dashboards of the application data visualization. Maps use markers to indicate locations and heat maps consist of a heat layer representing the concentration of the positions in a map.

For the system data visualization there are used two libraries: Vis.js and D3.js.

[10]https://leafletjs.com/

2.3.3 Vis.js

Vis.js[11] is a browser based visualization library used to implement the topology graphs of the system data visualization. For this implementation, it is used the *Network* component, whose layout consists of edges and nodes. Edges are the links between the nodes.

2.3.4 D3.js

D3.js[12] is a Javascript library that offers the possibility of adding any type of chart to a web application. This is achieved by combining powerful visualization components and a data-driven approach to DOM manipulation.

The D3.js component used in this project for the flows chart in the system data visualization is the Sankey Diagram. In a similar way to the topology graphs, this diagram is composed of nodes and links. Nodes are connected with links that have a value. That way the flows between nodes can be appreciated and the line becomes thicker as this value grows.

2.4 Javascript

By looking at the previous sections it can be inferred what is the programming language chosen: Javascript. It is a common factor among the technologies involved in this project. The fact that everything is developed with only one language is another reason that supports the choice of using the MEAN Stack.

According to the Octoverse report [8], which is based on Github activity, the top programming language by repositories created and by contributors is Javascript as seen in Figure 2.3.

[11] http://visjs.org/
[12] https://d3js.org/

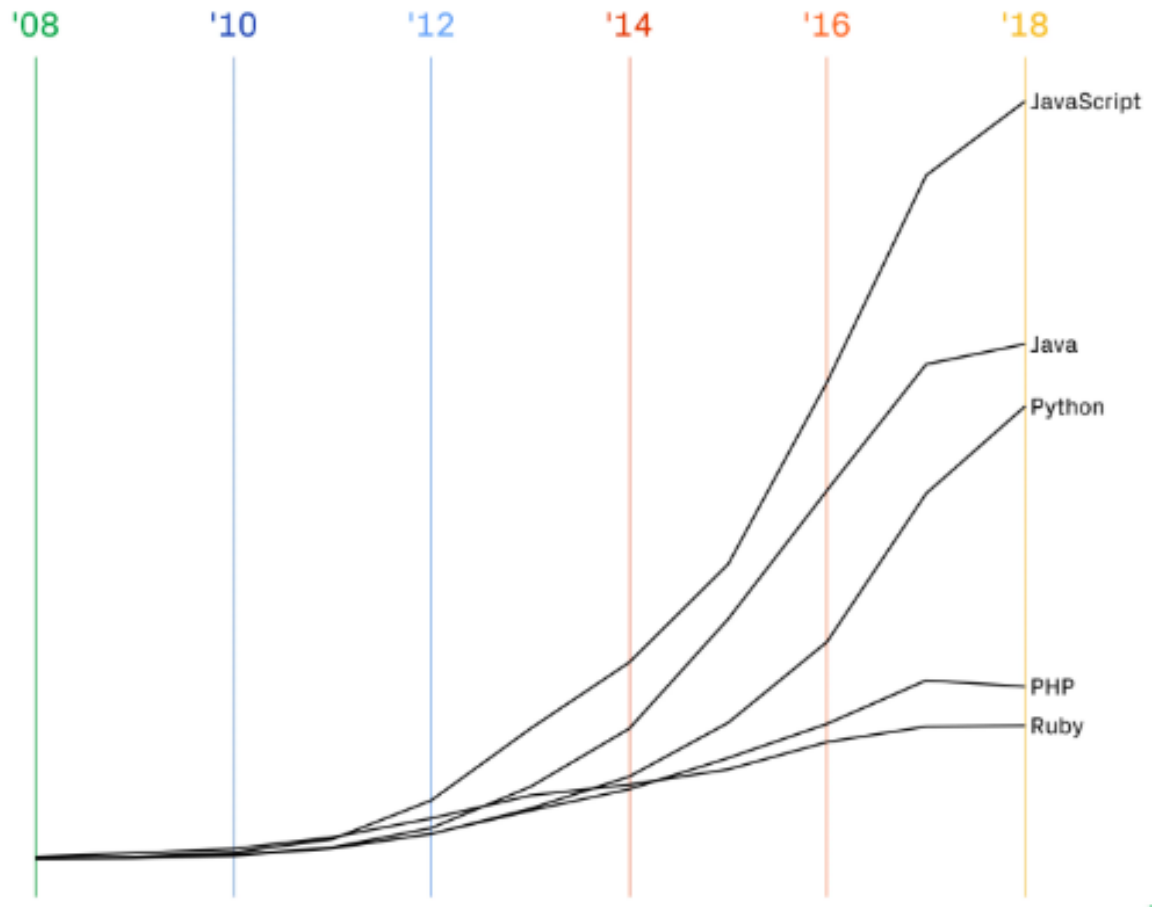

Figure 2.3: Top programming languages by repositories created[13]

[13]https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/

CHAPTER 3

Requirement Analysis

3.1 Introduction

The result of this chapter will be a complete specification of the functional requirements for the visualization system proposed. They will be matched in further sections with client side submodules. Furthermore, this specification helps to decide which aspects to implement first and take apart less important functionalities which can be implemented in future iterations.

First, it will be discussed the different actors involved in the system. Afterwards, the specification of the requirements will follow the approach of user stories. They describe functionality that will be valuable to a user of the system [9]. Test scenarios, which can be found in the Appendix, serve as a test environment for the user stories. A test scenario explains how to check the execution of these functional requirements.

3.2 System actors

According to ISO IEC standard 30.141:2018 concerning IoT Reference Architecture [10], all IoT related activities can be categorized into three user groups as listed:

1. IoT service provider
2. IoT service developer
3. IoT user

The users that use this module of the platform are from the first two groups.

On one hand is the IoT service provider. His/her role is to manage and to operate IoT services. We can identify some sub-roles that are the main users of this module: business manager, system operator, operation analyst and data scientist.

On the other hand, we have the IoT service developer. His/her roles include implementation, testing and integration of IoT services with the IoT platform. The only sub-role from this group that utilizes the module is the solution architect.

The activities from these sub-roles are indicated in Figure 3.1.

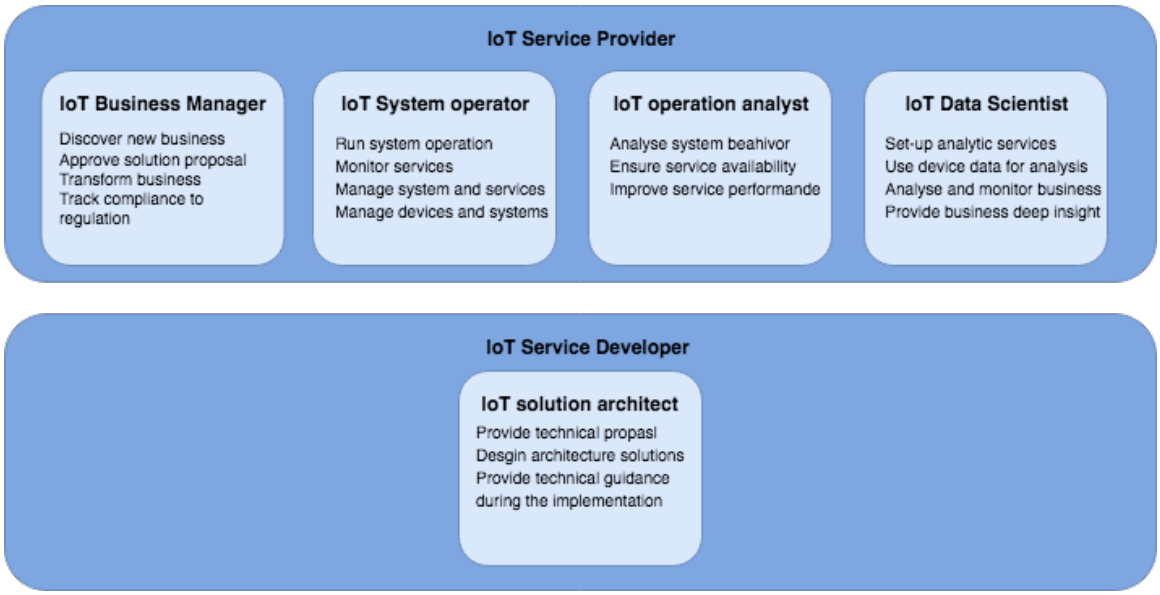

Figure 3.1: Activities from the different IoT sub-roles

3.3 User Stories

These sub-roles from the IoT users groups compose three different groups of user stories of this module:

1. Application data visualizer : US-C.1.1

2. System data visualizer : US-C.2.1

3. Visualization administration : US-C.3.x

3.3.1 US-C.1.1: Application data visualizer

US-C.1.1	
As a	IoT business manager / IoT data scientist
I want	To display all the information about the connected devices of the platform in a dashboard.
So that	I can use the device data for analysis and to get metrics to transform business.
Test scenario	TS-C.1
Comments/explanations	All the data is represented in real time in a single webpage with different types of charts and maps.

3.3.2 US-C.2.1: System data visualizer

US-C.2.1	
As a	IoT system operator / IoT operation analyst / IoT solution architect
I want	To see in a single view the deployment of the IoT platform.
So that	I can generate a report, analyze the system behavior and improve its performance by providing technical solutions.
Test scenario	TS-C.2
Comments/explanations	This view always has the same appearance. However, the data is dynamic and refreshes automatically. The deployment is represented by the topology of the platform and the flows between the nodes. If a node is clicked its attributes are shown.

3.3.3 US-C.3.x: Visualization administration

US-C.3.1	
As a	IoT data scientist
I want	To have all the data available.
So that	I can provide all the data to the visualizers in a single webpage and setup analytic services.
Test scenario	TS-C.3
Comments/explanations	The data is displayed in a list with labels indicating the type of data (float, integers, locations...) they contain.

US-C.3.2	
As a	IoT data scientist
I want	To design new dashboards.
So that	The visualizers can see more than one dashboard with different information in each one.
Test scenario	TS-C.3
Comments/explanations	These dashboards designed are the ones that will be viewed by US-C.1.1.

US-C.3.3	
As a	IoT service provider
I want	To design new templates by choosing the widgets and structure.
So that	Other visualization administrators can create a dashboard using the templates and showing the data in the most representative and optimal way.
Test scenario	TS-C.4
Comments/explanations	All the widgets available are shown on the left and can be dragged to the blank section where the template is being designed.

CHAPTER 4

Architecture

4.1 Introduction

In this chapter, we cover the design phase of this project, as well as implementation details involving its architecture. Firstly, we present an overview of the project, divided into several modules. This is intended to offer the reader a general view of this project architecture. After that, we present each module separately and in much more depth.

We can divide the project into three groups, each of them having a specific feature:

- **Database:** it stores which data will be represented and in which way, but it does not store the data in itself.

- **Client:** it represents the data with different visualization libraries and allows to choose which data to visualize.

- **Server:** as can be appreciated in the Figure 4.1, it is the link between the client side and the database. It retrieves data from the database and provides it to the client side with an Express API.

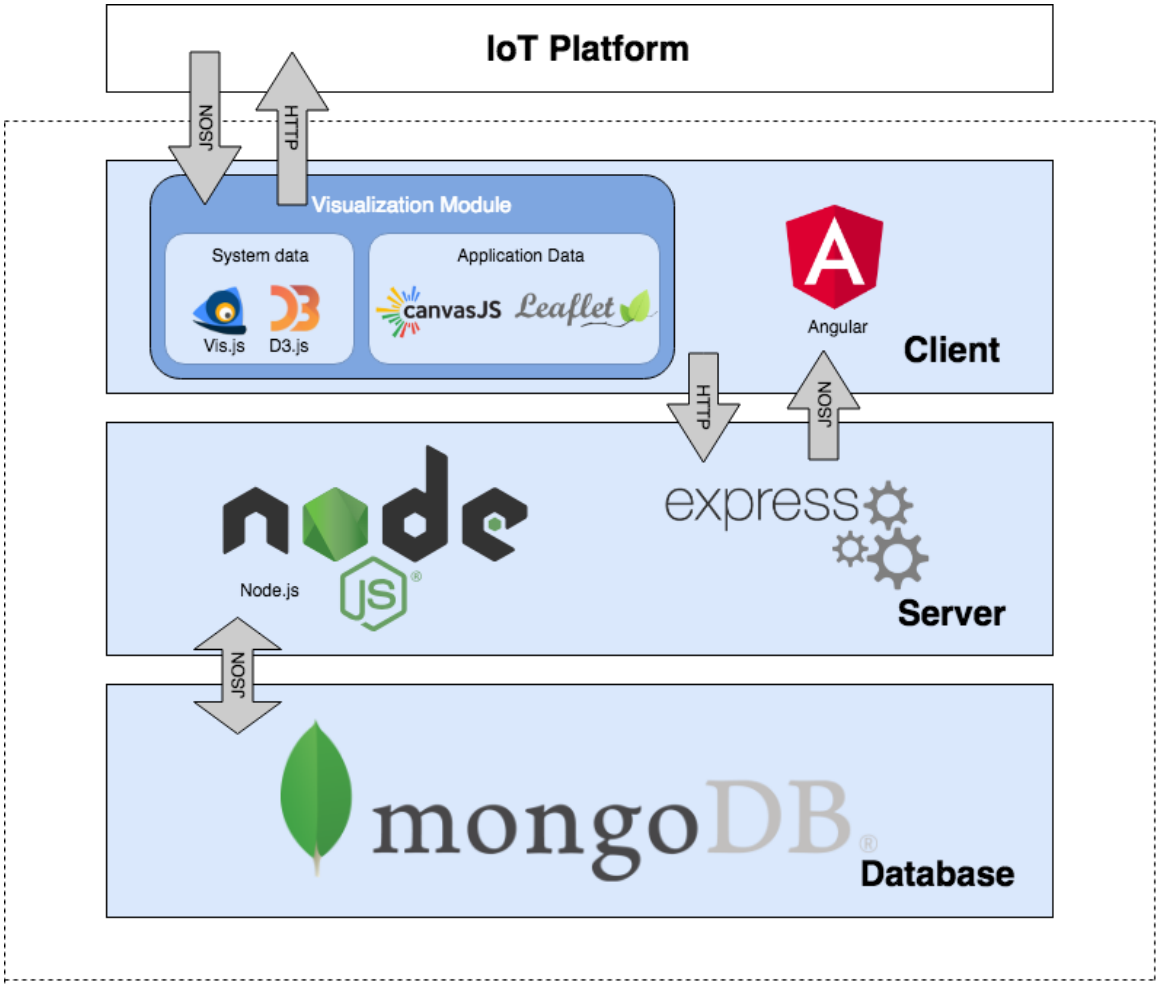

Figure 4.1: Global architecture of the system

The following sections describes deeply the three modules involved in the global system architecture.

4.2 Server module

In this section, it is going to be described the server side architecture, detailing how has it been implemented and communicates with the other modules.

As mentioned in previous sections, the technologies used to implement the server side are Node.js and Express. The Node.js server is located at port 3000, and uses Express to define an API. The objective of this API is to communicate the client side with the database.

The available API calls that can be made from the client side to the endpoint server are presented in the following table.

Method	**Name**	**Description**
GET	/api/templates	Retrieves all the templates stored in the database.
POST	/api/templates	Inserts a new template into the database.
DELETE	/api/templates/:id	Removes the template with the selected id from the database.
GET	/api/dashboards	Retrieves all the dashboards stored in the database.
POST	/api/dashboards	Inserts a new dashboard into the database.
DELETE	/api/dashboard/:id	Removes the dashboard with the selected id from the database.

Nevertheless, there is one small quibble: CORS. *Cross-Origin Resource Sharing* is a security policy that manages cross-origin requests. If we do nothing, the client will not be able to obtain responses from the API as the server would only allow requests from the same origin. To prevent it, cross-origin requests have to be allowed, and to achieve this we need to set the correct headers in the response:

```
app.use((req, res, next) => {
  res.setHeader("Access-Control-Allow-Origin",
```

```
      "http://localhost:4200");
    res.setHeader(
      "Access-Control-Allow-Headers",
      "Origin, X-Requested-With, Content-Type, Accept"
    );
    res.setHeader(
      "Access-Control-Allow-Methods",
      "GET, POST, DELETE"
    );
    next();
  });
```

The first header enables exclusively the client to make successful requests to the server. This practice falls under a restrictive policy, where every request is denied except for the explicitely allowed. The second header enables CORS for all resources in the server by indicating which HTTP headers can be used in the request. Lastly, the third header limits the requests type to only GET, POST and DELETE, blocking any other request.

4.3 Database module

The objective of this section is to describe which data is stored in the system and how it is modeled. It is also important to make it clear that the data which is going to be visualized in the client is not stored in this database.

Firstly, we will talk about MongoDB which is, as mentioned before, the database used and how is it used in the project. Then, it will be explained the different models used for the data as well as how they are stored in MongoDB.

4.3.1 MongoDB

MongoDB databases can be set up in a local machine or in the cloud. In this case it is used a MongoDB cloud solution, MongoDB Atlas[1], which is a DBaaS (Database as a Service). This solution conducts the database administration, evading us from taking care of the database configuration or the infrastructure provisioning.

MongoDB Atlas uses clusters to host databases. A cluster is a group of linked servers

[1]https://www.mongodb.com/cloud/atlas

that work together. They improve the performance and availability over a single server, as they are designed with a minimum of three node replica distributed within a cloud region. MongoDB Atlas works with cloud providers, and in our case we chose AWS.

In this project there is one cluster where is located the database. Databases in MongoDB use collections to store data. The data of this module is stored in two collections: dashboards and templates. To visualize the content of your clusters, MongoDB offers two options: a web graphic user interface or command line interface. Due to its simplicity we utilized the CLI. The database and collections of this project are represented in the following figure.

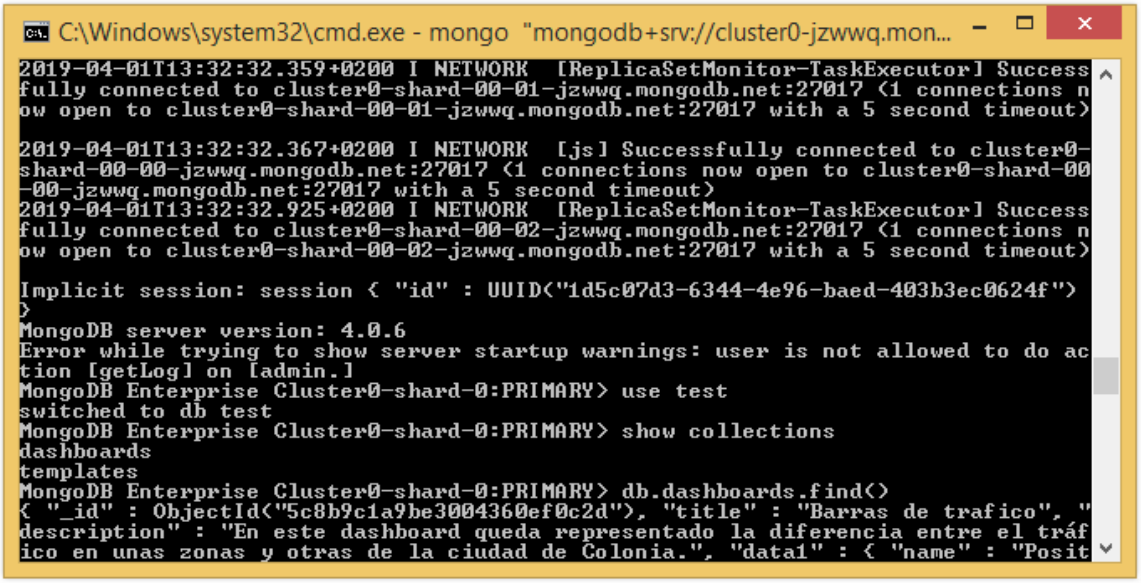

Figure 4.2: Databases and collections visualized in CLI

4.3.2 Data models

As mentioned above, all the data stored in the database consist of two collections, templates and dashboards.

- **Templates:** they settle the structure of the dashboards that will be designed, with different widgets available.
- **Dashboards:** they establish which data will be represented in each widget of a chosen template.

We will now proceed to explain in detail the models used to implement each collection:

4.3.2.1 Template model

The visualization in the client side is carried out with widgets from different libraries that will be further described in the client section. The templates define which of these widgets will be used in each chart of the dashboard to visualize data.

The Mongoose schema chosen to model the templates consists of strings for the widgets and for the title:

```
const templateSchema= mongoose.Schema({
    title: {type: String, required: true},
    widget1: {type: String, required: true},
    widget2: {type: String, required: true},
    widget3: {type: String, required: true},
    widget4: {type: String, required: true},
    widget5: {type: String, required: true},
    widget6: {type: String, required: true},
});
```

4.3.2.2 Dashboard model

Every dashboard is divided into six charts that can represent different data. They include a title and a description, so that the person who is designing the dashboard helps a future viewer to know what the dashboard is about. All this leads to a model including a title, a description and six data models.

The data from the IoT platform can be accessed via web service endpoints with HTTP. It arrives to the client in JSON format, and it may contain various fields with different data. A possible response from the server could be the following:

```
{
    DestLon: 6.98229407599,
    ip: "134.60.64.86",
    lon: 6.84042615507,
    fuelConsumption: 7.4,
    speed: 48.730759517,
    co2_emission: 310,
    zone: "West",
    port: "1883",
```

```
        DestLat: 50.8686742811,
        weather: "Clear",
        co_emission: 2,
        topic: "coche",
        id: "901",
        lat: 50.9484587409,
        timestamp: "9:37"
    }
```

As can be seen in this example, we need to specify in some way the fields to be represented. A chart need at least one field for the ordinate axis and another for the abscissa axis. With this JSON there would multiple options. To resolve this issue, the data model includes two fields to point which JSON fields will utilized by the chart. It will also have fields for the name, the widget to use and the URL. To sum up, the model for the data in each chart has the following fields, which are all strings:

- **Name:** the name is given to the data including the JSON.
- **Url:** url of the web service endpoint of the platform.
- **Widget:** the type of chart that was chosen with the template.
- **ValAttr:** the selected field of the JSON for the ordinate axis.
- **LabelAttr:** the selected field of the JSON for the abscissa axis.

Finally, taking into account that the data model for every chart is the same, the Mongoose schema model used for the dashboards is:

```
const dashboardSchema= mongoose.Schema({
    title: {type: String, required: true},
    description: {type: String, required: true},
    data1:{
        name: {type: String, required: true},
        url: {type: String, required: true},
        widget: {type: String, required: true},
        valAttr: {type: String, required: true},
        labelAttr: {type: String, required: true}
    },
```

```
        data2:{...},
        data3:{...},
        data4:{...},
        data5:{...},
        data6:{...},
    });
```

4.4 Client module

In this section it is given a deeper description of the implementation of the client module, as well as the three submodules it consists of. First we are going to give a general overview of the module. Afterwards, the three submodules will be explained in detail, connecting them with the requirements specified in Chapter 3.

4.4.1 Overview

As already stated, the main technology involved in the development of the client module is Angular. This Javascript framework utilizes components, which are the basic building blocks that make up all the views of the user interface. An easy way to create components which is used in this project is Angular CLI, a command line interface which enables to generate Angular components and Angular services with simple commands. To create a component with Angular CLI is as simple as writing this command:

```
$ ng g c <NewComponentName>
```

With this command, four files are generated:

- *NewComponentName.component.html:* file to write the HTML code of the component.
- *NewComponentName.component.css:* file to write the CSS rules to style the component.
- *NewComponentName.component.ts:* file with the logic of the component. Written in Typescript.
- *NewComponentName.component.spec.ts:* unit test file. It is not used in this project. Also written in Typescript.

The user interface of this application, which is made up of components, is structured as follows:

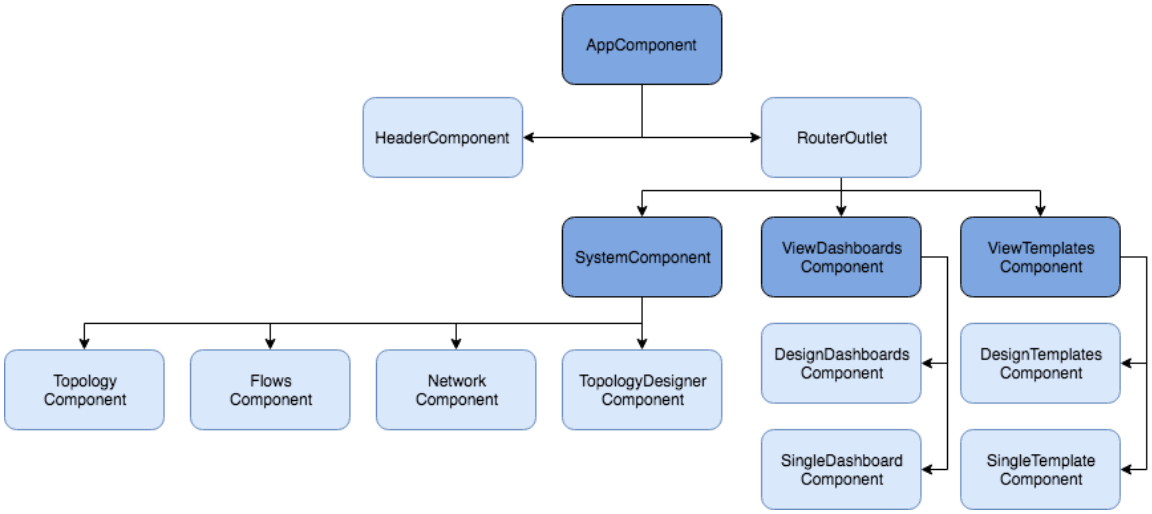

Figure 4.3: User Interface structure

The AppComponent is the parent component which contains all the components. The HeaderComponent, placed in the upper part of the application, includes the navigation menu. It allows to choose what component to render in the RouterOutlet. The RouterOutlet is a directive from the Angular Router library that acts as a placeholder that marks the spot in the view where the router should display the components.

In the file *app-routing.module.ts* are specified all the routes that the Angular Router should consider to render components in the RouterOutlet. The routes defined in the application are the following:

Route	**Component**
/	*SystemComponent*
/system	*SystemComponent*
/dashboards	*ViewDashboardsComponent*
/dashboards/design	*DesignDashboardsComponent*
/dashboards/view	*SingleDashboardComponent*
/templates	*ViewTemplatesComponent*
/templates/design	*DesignTemplatesComponent*
/templates/view	*SingleTemplateComponent*

These components belong to the three submodules of the client and are described in the succeeding paragraphs.

4.4.2 Visualization Administration

The objective of this submodule is to meet the needs specified in the user stories US-C.3.x. First of all, we are going to describe the functioning of the whole application (excluding the system data visualization) and then talk about the components that make it possible.

To be able to visualize a dashboard, you have to design it first, which is the main goal of this submodule. This dashboards are designed using templates. Therefore, first of all, templates have to be designed. To do this, it has to be chosen the widgets that will be used in the dashboards. After that, dashboards are designed by choosing a template that has been previously created, and selecting which data to represent in each widget. The result of this enables the visualization of real time dashboards, which is task of the application data visualization submodule.

The two Angular components that allow this process are *DesignTemplatesComponent* and *DesignDashboardsComponent*. It can be inferred from their names what are their role in the application.

The *DesignTemplatesComponent* offers the environment to design new templates. We created a mockup for this component before implementing it to have a general idea of how it should look like, which can be appreciated in Figure 4.4. In the end it would look roughly

different as there are buttons added to improve its usability.

Figure 4.4: Mockup of *DesignTemplatesComponent*

To implement this component it is used the Angular Material component *MatSidenav* which offers a side drawer on the left and place for the main content on the right side. In this place there are six blank boxes in which the widgets are dragged into. On the left are the widgets available, represented with a *MatList*. The widgets are from two libraries: maps from Leaflet and all the remaining from CanvasJS. Maps represent locations with latitudes and longitudes and the other widgets represent data with the tuple key/value. Lastly, there are two *MatButtons* at the bottom: one to finish with the template's design and another to show a preview of the template. This is made with a modal window using Bootstrap's modal component. In Figure 4.5 can be appreciated the final appearance of this component, which is very similar to the mockup.

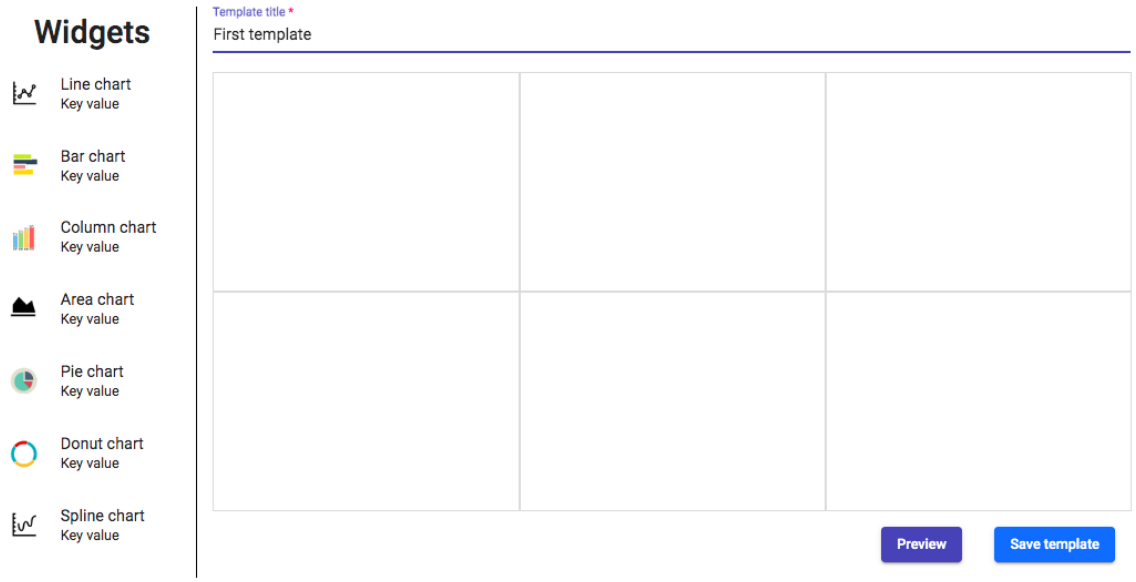

Figure 4.5: Final look of *DesignTemplatesComponent*

Furthermore, the *DesignDashboardsComponent* provides a context to design a dashboard using templates previously created. The layout of the elements in this component is the same as in *DesignTemplatesComponent*, but in this case the logic is more complex. In the upper right-hand corner, there is a *MatSelect* to choose a template for the dashboard. There are also two *MatInputs* to give a title and a description to the dashboard. On the left are all the data sources available, and on the right side, the blank boxes in which they are dragged into.

Reached this point, it is approximately the same as the previous component. What adds complexity to the implementation of this component is the selection of the fields of the data sources to represent in the dashboard. This is, the *ValAttr* and *LabelAttr* we discussed in the dashboard model. To achieve this, it is used the Angular Bootstrap's modal component *NgbModal*, different from the Bootstrap's modal component. The first one is used because it enables to pass data from the *NgbModal* component, as needed in this case. Inside the blank box, there is a *MatButton* that opens the *NgbModal* to choose the *ValAttr* and the *LabelAttr*. When the button is clicked, a HTTP request is made with the URL of that data source, to obtain the fields of its JSON. With that, in the *NgbModal* are represented the fields to be chosen with *MatRadioButtons*, which are only enabled if the data type matches with the needed one: the value of the field in *ValAttr* has to be a number and the value of the field in *LabelAttr* has to be a string. Having done this with the six boxes, the dashboard can be saved with the *MatButton* at the bottom. The final appearance of this component can be seen in the following figure:

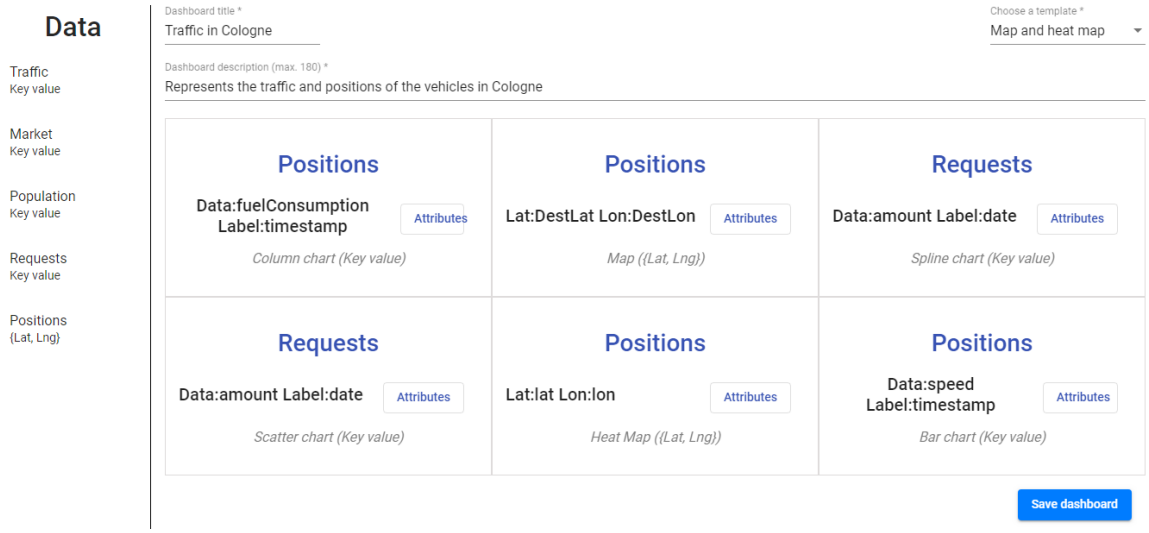

Figure 4.6: Final look of *DesignDashboardsComponent*

To improve the user experience, we added elements and features that make the application more user-friendly. These are the *NgbAlerts*, *MatTooltips* and the *Drag and Drop* interacting. *NgbAlerts* are launched when something is out of the ordinary, and they are from Angular Bootstrap. *MatTooltips* are added in some buttons and give a hint of the button's action. *Drag and Drop* is present in the two components explained above to make it easier to choose a widget or a data source, and is from the Angular Material CDK.

Aside from these components, there are two components regarding Visualization Administration which are *ViewDashboardsComponent* and *ViewTemplatesComponent*. These components show basic information about the dashboards and templates in the database. For its representation are used *MatCards*, which contain two actions: one to delete them from the database; and another to open its visualization.

4.4.3 Application Data Visualization

This part of the application is the most representative, as the primary objective is to visualize data. The Angular component in charge of this is the *SingleDashboardComponent*, and it aims to satisfy the requirement US-C.1.1. It is made up of six replicas of *ChartComponent*, one for each chart of the dashboard and they display the data coming from the server.

The *ChartComponent* is merely a Div element. It can render three types of content: maps, heat maps or Canvas JS charts. The latter type consists of a wide range of charts, but nevertheless they are all rendered the same way. The *SingleDashboardComponent* provides

these components with all information necessary to allow the *ChartComponent* display the data:

- **Title:** a title for the chart.
- **Widget:** map, heat map or Canvas JS charts.
- **Url:** endpoint to obtain the data by HTTP.
- **ValAttr and LabelAttr:** the JSON fields to be used in the chart.

The *ValAttr* and *LabelAttr* are used in two different ways depending on the chart type. If its a map or a heat map, the *ValAttr* field is the JSON field where the chart will collect the latitudes, and the *LabelAttr* for the longitudes of the positions in the map. In both cases their values are numbers. In any other type of chart, as described previously in the dashboard model, the *ValAttr* is for the ordinate axis and the *LabelAttr* for the abscissa axis, being a number and a string the values in that JSON fields respectively.

With all this information, the *ChartComponent* makes a HTTP request to obtain the JSON with the data. Then, it is filtered with the *ValAttr* and *LabelAttr* to generate tuples {latitude, longitude} for maps or {key, value} for the other types of charts. Afterwards, the result of this is the data in the correct format to be rendered. Furthermore, each of these components have a timer to update its content every five seconds with the following code:

```
interval(5000).subscribe( x=> {
    this.load();
});
```

These two client submodules, Visualization Administration and Application Data Visualization are very closely bound up. Now it is going to be explained the other type of visualization available in this project.

4.4.4 System Data Visualization

This project aims to offer a complete visualization of the IoT platform. Just as the data collected from the connected devices can be visualized, the deployment of the platform has to be visualized somehow, as specified in the user story US-C.2.1. For this reason, this submodule has been added to the visualization system.

This submodule corresponds to the Angular component *SystemComponent*. On its own it does not offer any visualization, but it contains four components which do so. *SystemComponent* is the parent component and consists of a set of *NgTabs* that show the four components with the system data visualization: *TopologyComponent*, *FlowsComponent*, *NetworkComponent* and *TopologyDesignerComponent*.

The last mentioned does not offer visualization as it goes beyond: it enables the user to manage the topology of the IoT platform. All these components are going to be explained in detail in the succeeding paragraphs.

4.4.4.1 TopologyComponent

This component is comprised of two *NgbPanels*, one in the left half and another in the right half. In the left panel is displayed the topology of the IoT platform with a node tree. To implement the node tree is used the *Network* component of the library Vis.js. It can be appreciated the three types of nodes existing in the platform: cloud, mid and low nodes. To identify which type is each node there is a legend in the upper right-hand corner of the panel.

Furthermore, if a node is clicked, in the right panel appears a table. This table, which is implemented with Bootstrap, shows all the metrics of the node clicked: CPU usage, Network In/Out, Disk In/Out, used RAM and free RAM. In the table's last row there is a toggle button that displays another table with all the microservices executed in that node and their metrics. The data represented in both tables is obtained with a HTTP request to an endpoint of the platform which returns the metrics of the node clicked.

4.4.4.2 FlowsComponent

In the left panel, a Sankey Diagram represents the flows through the nodes of the IoT platform topology. Each flow is depicted as a trace whose thickness indicates the amount of requests is being sent in that link. This chart is implemented with the library D3.js. To facilitate the user's visualization, flows have the same color as the origin node, as seen in Figure 4.7.

In the right panel, it is displayed the topology with transit traffic. This topology is different from the one in the *TopologyComponent*. The topology in that component represents all the nodes and connections available, and the topology in this panel consists of the nodes and links from the other topology which are processing traffic.

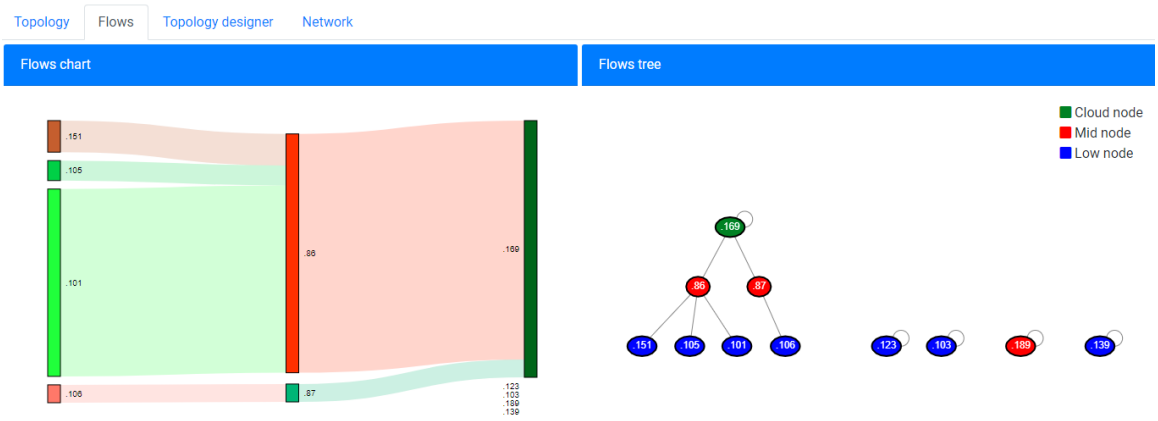

Figure 4.7: Final look of *FlowsComponent*

4.4.4.3 NetworkComponent

This component is very similar to the *FlowsComponent* but they differ in the data which define the value of the flows. In the *FlowsComponent* the value of the flows is the amount of requests between two nodes. In this case, this value is the network out metric of each node.

The right panel consists of two column charts in which can be appreciated the network in and out metrics of each node of the IoT platform. Both the column charts and the flows chart update every three seconds the same way as it is done in Section 4.4.3.

4.4.4.4 TopologyDesignerComponent

While developing the *TopologyComponent*, we realized the topology could be easily changed. Just as the topology is obtained with a HTTP request, the topology can be updated if a JSON with the same format is sent in a HTTP POST request. Moreover, this component adds value to the system as it enables to manage the topology of the platform besides its visualization.

Firstly, it is going to be explained how is the topology design process. Afterwards, we are going deeper in the implementation of this component and the elements it contains.

When this component is initialized, in the left panel are shown all the nodes available in the platform without linking them. There are two turns, turn to choose an origin and turn to choose a destination node. In the right panel is indicated which turn it is. If it is the origin's turn and you click in Node A, it changes to destination's turn. The next click

will create a link between Node A and the node clicked, and it will be appreciated in the component as both nodes will be connected. To design the topology, the links have to be added one by one. If you make a mistake and you want to undo it, there is a button that will delete the last link created. There is also a button to reset the topology and start from scratch. To finish with the design you have to click in *Finish* and that will send the HTTP POST request to update the topology.

To implement this component it is also used the component from Vis.js. It needs two arrays to render the node tree: *Nodes* and *Edges*. If the *Edges* array is empty, it will display isolated nodes. An *Edge* consists of a *nodeOrigin* and a *nodeDestination*, so if two nodes are consecutively clicked during an origin's turn and a destination's turn, a new *Edge* or link will be added to the topology. To undo a link it just has to be deleted from the *Edges* array, and to reset it has to be emptied. In the right panel, there is a table in which in each step done, it is indicated what parent nodes there are with their children nodes. This table, along with the buttons, are developed using Bootstrap.

CHAPTER 5

Case study

Traffic is a major problem in many big cities where it supposes an unnecessary wastage of time. For this reason, RECAP has decided traffic to be a case study of the IoT platform. It is monitored the traffic in the German City of Cologne, which is divided into several zones to appreciate the traffic's distribution in the city. Moreover, there are two types of traffic represented: one called *Traffic* which represents the vehicles sending data to the platform (positions, speed, CO_2 emission, fuel consumption...), and another called *Requests*. This one corresponds to the vehicles requesting for a route to reach its destination.

The dataset used in the IoT platform to simulate the traffic is TAPASCologne[1]. It is a open source simulation scenario describing the traffic within the city of Cologne for a whole day. This dataset is processed in the platform and results in the two types of traffic described in the previous paragraph.

In this chapter we are going to apply the test scenarios with the traffic data to check if the project meets the needs of the requirements. These scenarios can be found in Appendix C. In order to properly interpret the results from the visualization, we are going to apply the scenarios in this order:

1. TS-C.4: IoT dashboard template design

[1] https://sumo.dlr.de/wiki/Data/Scenarios/TAPASCologne

2. TS-C.3: IoT dashboard design with predefined templates
3. TS-C.1: IoT application data visualization
4. TS-C.2: IoT system data visualization

5.1 TS-C.4: IoT dashboard template design

This test scenario aims to check the functioning of the *DesignTemplatesComponent*. In the first step we browse to the URL where the client is located. At the moment this web application is not deployed, so the URL for this scenario is *http://localhost:4200/templates*. This URL leads us to the a page with all the templates already designed. By clicking in *New template* is rendered the *DesignTemplatesComponent*. As can be appreciated in Figure 5.1, this component has all the widgets on the left indicating the type of data required and they are draggable. Also, there is a blank section on the right where the widgets can be dragged into.

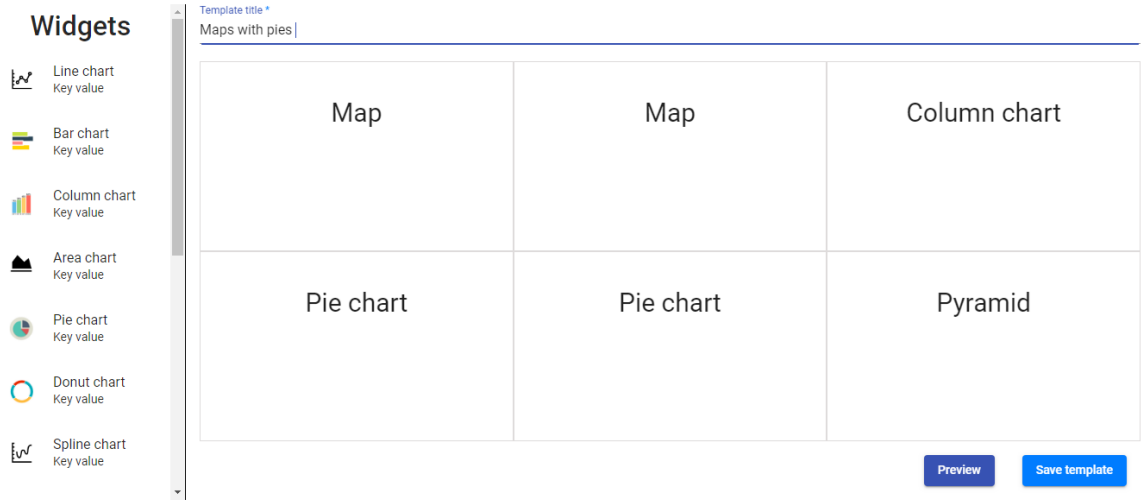

Figure 5.1: IoT dashboard template design

For this use case, we are creating a template with a map and a heat map for the traffic and the requests in Cologne, which are accompanied by two pies charts for the zone distribution and two more charts for the network monitoring. Once every widget is dragged into the blank section, a preview of the template can be seen by clicking in *Preview*. Finally, to store the template in the database it has to be clicked the *Save template* button and we are redirected to the page in the first step.

As can be appreciated, this component works in line with the steps in the TS-C.4

5.2 TS-C.3: IoT dashboard design with predefined templates

For this scenario, as a precondition, there must be a template previously designed. This was made in the preceding section, and we called that template *Maps with pies*. Just as above, we browse to the URL *http://localhost:4200/dashboards* and appears a list with all the dashboards designed. Clicking in *New dashboard* is rendered the *DesignDashboardsComponent*.

In this case study, we want to visualize in a map the positions of the vehicles in Cologne and in a pie chart the distribution of these vehicles in the different zones. We also want the same for the requests. In addition, we will visualize the network bandwidth consumption in each node. To design it, we choose the template designed in the above section. Then, we drag every data to each widget in which we want it to be represented. This corresponds to steps three and four from the TS-C.3. We also give the dashboard a title and a description. Prior to saving the dashboard, the JSON fields have to be selected, which is possible with a modal window that asks us to choose them. Figure 5.2 shows the window of the traffic JSON in which we are going to choose *lat* and *lon* to represent the positions.

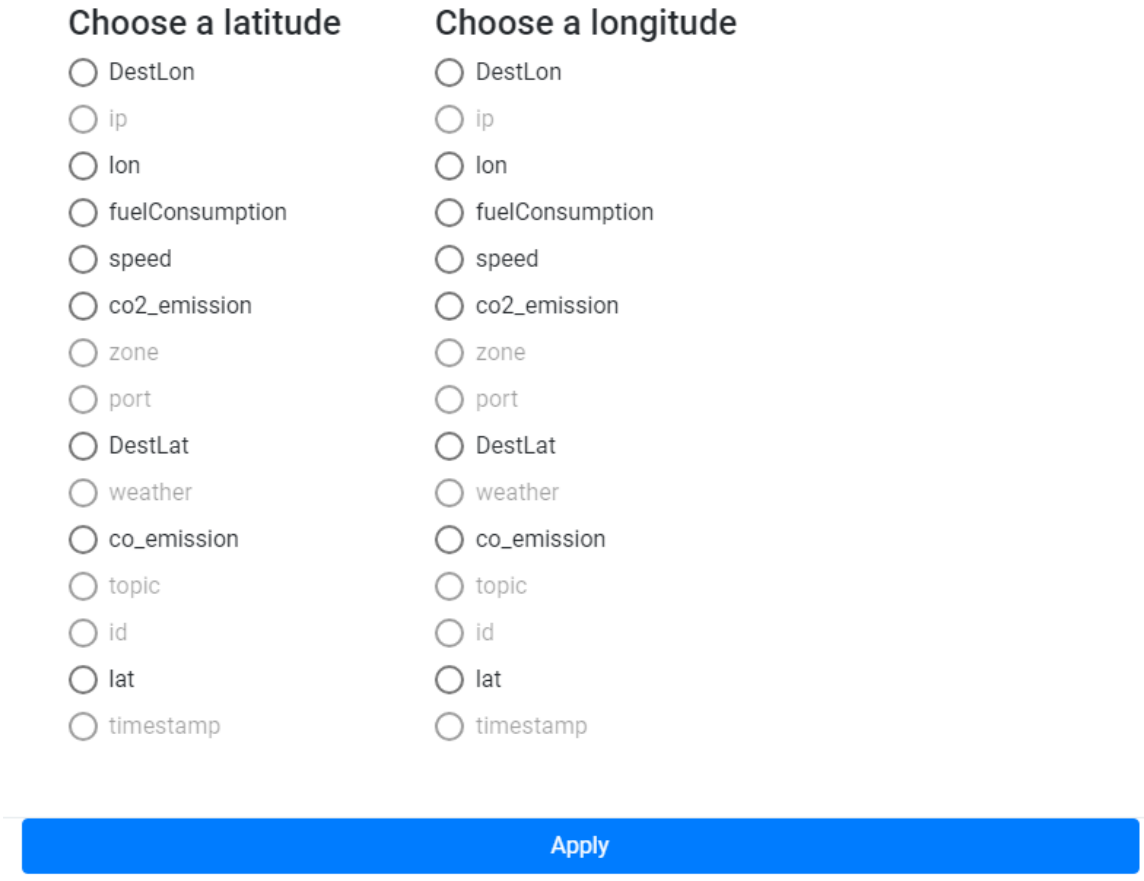

Figure 5.2: JSON fields selection in *DesignDashboardsComponent*

As can be seen, the fields whose values are inconsistent with the data required are disabled. Doing this with the six charts, we are ready to save the dashboard with the *Save dashboard* button and the test scenario would be successfully applied.

5.3 TS-C.1: IoT application data visualization

The objective of this test scenario is to check that data is properly visualized and the *SingleDashboardComponent* works correctly.

First, we browse to the URL *http://localhost:4200/dashboards* and all the dashboards already designed are listed. In this case we want to visualize the dashboard designed in the preceding section, which we called *Traffic in Cologne*. To open it, we click the button *View* and the dashboard is displayed. Figure 5.3 shows a snapshot of the visualization.

Figure 5.3: *Traffic in Cologne* visualization

In this image the real time feature cannot be appreciated, but the data in these charts and maps are updating every five seconds. The vehicles move around the map and the metrics of the nodes change its values. Moreover, if the cursor passes over an element of the charts, its exact value is indicated.

With all this, we can confirm that this scenario is working properly too.

5.4 TS-C.2: IoT system data visualization

Lastly, this scenario checks if the web application is meeting the requirements established in US-C.2.1, which aims to represent the deployment of the IoT platform.

Applying the scenario, we browse to the URL *http://localhost:4200/system*. With this, it is rendered the *SystemComponent*, with four tabs. By default, it is opened the first tab, which is the *TopologyComponent*. As explained in Section 4.4.4.1, if a node is clicked it is displayed its metrics in a table:

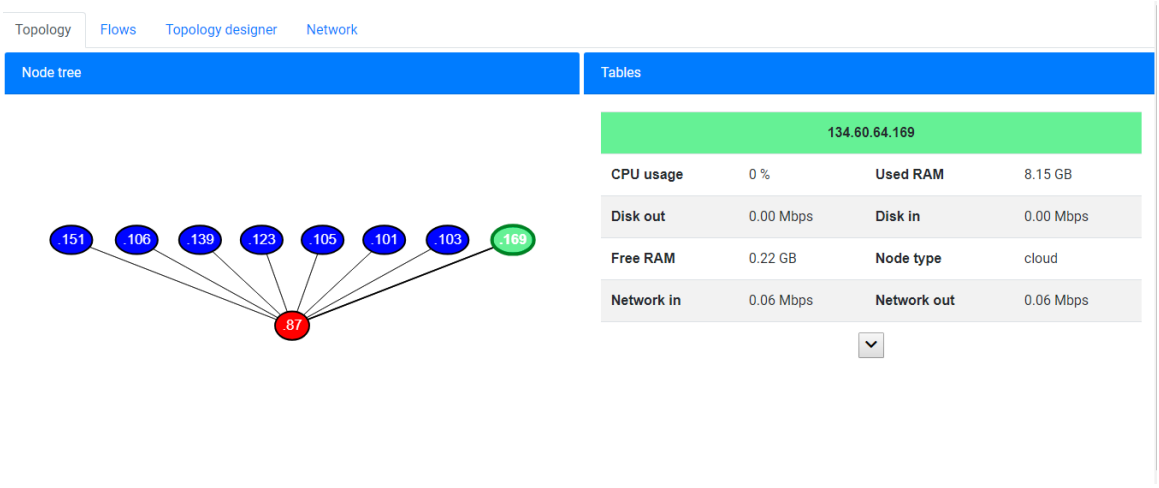

Figure 5.4: Topology of the IoT platform

Furthermore, there is a toggle button under the table. If clicked, it shows the microservices that are executed in that node and its metrics, as can be seen in Figure 5.5.

Service	Status	JVM Memory	CPU's	CPU usage	Uptime
REDIS	UP	143.80 MB	1	1 %	260488 s
ZONEDESTINATION	UP	164.16 MB	1	1 %	260488 s
OPTIMIZER	UP	150.17 MB	1	1 %	260491 s
LOCALMETRICS	UP	147.11 MB	1	1 %	260487 s
METRICSSERVICE	UP	141.56 MB	1	1 %	259449 s
REQUESTPROXY	UP	174.42 MB	1	1 %	260484 s
ROUTESCALCULATOR	UP	165.72 MB	1	1 %	260487 s

Figure 5.5: Microservices table of node *134.60.64.169*

In this user story is also required that the flows through its nodes are represented too. This is achieved in the next tab of the *SystemComponent*, which is *FlowsComponent*. In Figure 4.7 can be appreciated a snapshot of this component.

With these two tabs of the *SystemComponent*, it is confirmed that the requirements in US-C.2.1 are achieved. The other two tabs, which are explained in Section 4.4.4.3 and Section 4.4.4.4, go beyond the requirements of the system.

5.5 Results

By applying the test scenarios in this case study, we have drawn really interesting conclusions that may be very helpful.

In the dashboard of Figure 5.3, if we look at the first map and the *Traffic per zone* chart, we easily detect that more than a half of the vehicles are located at the center of the city, and nearly a quarter at the east. Something similar occurs with the second map and the *Requests per zone*, which represent the vehicles that are requesting routes to reach their destination. In this case it is more exaggerated the clustering of the vehicles in the center of the city.

Alternatively, the other charts show the network bandwidth consumption for each node. By looking at these charts, it is appreciated that the node *134.60.64.103* is using almost all the network resource. With this, a system manager from the IoT platform could manage the flows in the topology to prevent it from a collapse.

In conclusion, we have verified that the system works as intended. It is expected to be used to present the state of a platform, so that it can be detected configuration problems such as excessive consumption of a node server. Likewise, it can be used to monitor the devices connected in an IoT platform.

CHAPTER 6

Conclusions and future work

In this chapter we will describe the conclusions that have been drawn after the implementation of this system. Furthermore, it discusses some problems faced during the project and the solution to them. Finally, it mentions some possible improvements to the visualization modules.

6.1 Conclusions

In this project, after undertaking an in-depth study of different visualization technologies, we have created a visualization system with the MEAN Stack as main tool and using different Javascript frameworks and libraries. The main objective of this project is to create a comprehensive visualization system which is simple to use. To achieve this, the system offers two different visualizations: application data visualization and system data visualization. The first one is carried out with dashboards containing six charts. They are customizable as templates can be designed to choose the types of charts in a dashboard. Also, the data to visualize can be selected in the design of dashboards. In contrast, the system data visualization is immutable and represents always the same data. It displays the deployment of the platform, showing node metrics and the flows in the topology. Additionally, it has

been added a topology management module that enables to design a topology for the IoT platform.

After testing both visualizations with the test scenarios, we are very pleased with the results as they meet the needs of the established requirements. Moreover, the primary objective of this project has been achieved, which is to offer a complete visualization of the IoT platform.

The following sections will describe the achieved goals, the problems faced and improvements for the system.

6.2 Achieved goals

This section explains the achieved goals that this project has accomplished:

- **Supply the IoT platform with a complete visualization system:**
  As commented above, this is the primary objective of the project. It is achieved by providing two different visualizations: one for the application data and another for the system data.

- **Visualize customized dashboards:**
  This goal was achieved by adding templates. They enable to choose which types of charts or *Widgets* to use. These are from two libraries: maps and heat maps from Leaflet and twelve types of chart from CanvasJS.

- **Store the templates and dashboards:**
  Both are stored in MongoDB using a cluster in MongoDB Atlas. We had to implement an API to access the database from the client side.

- **Show the deployment of the IoT platform:**
  The deployment is represented in different tabs that show the data in different ways. To implement the charts in this module we have used the libraries D3.js and Vis.js.

- **Addition of a topology managament module:**
  This was not expected to be done as it was improvised along the way. It has a user-friendly interface that allows to design a new topology for the IoT platform in an easy way.

- **Develop a user-friendly web application:**
  This is a personal goal we proposed, as we think it plays a vital role. If a web

application is unwieldy, users won't be utilizing it anymore. To achieve it we have added elements that makes the interaction with the user simpler, such as *Drag and Drop*, *NgbAlerts* or *MatToolTips*.

- **Module integration in the IoT platform:**
  This system is the IoT Visualization Entity of the IoT platform [2] and has been used as a demonstration of the platform in different IoT European Congresses[1]. Moreover, this system can be easily integrated in other platforms, in which case it would not take long.

6.3 Problems faced

During the development of the system, we came up against some problems that were not expected and took more time to be solved. In this section these problems are described and the solutions we reached to them:

- **HTTP requests blocked by CORS:**
  The first time trying to retrieve the templates from the dashboard we came across this error. To solve it we had to indicate in the server who can access to it and with what HTTP methods.

- **Use Leaflet in Angular:**
  Leaflet is a Javascript library, so it cannot be used directly in Angular, which is written in Typescript. To use Leaflet in Angular we had to install via NPM the following packages: *Leaflet*, *Leaflet-Heatmap*, *Ngx-Leaflet* and *Heatmap.js*.

- **JSON data to visualize:**
  At first, we did not consider how to indicate the chart which data from the JSON to represent. Later, we thought of a modal window to show the user which fields are in that JSON and choose which ones to use in the visualization. We used the *NgbModal* component to implement this.

6.4 Future work

In this section we talk about some improvements that could be added to the system and couldn't be implemented because of lack of time:

[1] http://wfiot2019.iot.ieee.org/

- **Add multi-series charts as a *Widget*:**
  Now the charts represent data from one field of the JSON. With CanvasJS is possible to represent multiple datasources in the same chart and even from different types.

- **Allow user to choose the number of charts:**
  Dashboards are comprised of six charts, and that is unchangeable. If the user could choose how many charts to use, it would offer the user a higher level of customization. This change would take considerable time as it would affect almost all the project because the models have to be changed in the database and server.

- **Improve the integrability in other projects:**
  Currently, this system can be easily integrated in other projects or platforms by modifying a few components. However, this could be much easier if there were some interfaces that enable the user to choose the web service URL to load data. For this, the database would store also these URLs and the data available in the *DesignDashboardsComponent* would be loaded dynamically from the database.

APPENDIX A

Impact of this project

This project aims to offer an IoT platform a complete data visualization system, which can be very useful to a lot of companies. In this appendix we will explain the impacts related to the project from a social, economic, environmental and ethical perspective.

A.1 Social impact

The use of a system such as this, can be very useful for a lot of people. *Data visualization* helps to obtain conclusions that can produce changes in a society. In the case study of Chapter 5, we could see that the traffic had more than a half or its vehicles in the center of the city. Knowing this, the city council could take action to fix this and avoid traffic jams, benefiting the citizens.

This is just one application of many that could impact the society.

A.2 Economic impact

In this section we will discuss the possible economic impacts this project could provide to a company or organization using it.

From the point of view of a company, this system could be highly beneficial. Returning to the use case of the traffic in a city, this system would be very interesting for a company with a fleet of vehicles. To monitor where are all the vehicles and its distribution can help the company to cover all the city. In a taxis company for example, this would avoid big losses of services.

Another application could be to monitor the performance of a system or platform. If it is detected that a server or equipment is collapsed with a lot of requests, it can be balanced this traffic or requests to another server that is not busy and prevent a denial of service.

These applications provide solutions to companies and can result in great profits.

A.3 Environmental impact

This section covers the environmental impacts of the visualization system.

Returning to the case study in Chapter 5, this system could have big environmental impacts. It offers the possibility of monitoring the CO and CO_2 emissions and the positions in which they are emitted. This makes it possible to have control over the emissions and take measures to reduce them in the locations most affected. For this reason, this system slots perfectly into environmental care projects.

Conversely, for the design and implementation of this system is not required any special hardware equipment beyond the computer. It could be considered the sensors of the vehicles of the case study, but as this system is independent and can be used in any environment, we will not contemplate them.

Nevertheless, developing software requires a certain power, which leads to a huge charge on the electricity networks and contributing to the gasses emissions. Additionally, this needs of a cooling system, which means a greater electricity expense. It is noteworthy that the project has been carried out in the company SATEC which uses the environmental management system ISO 14001:2015.

A.4 Ethical impact

The main ethical problem of this project comes from the idea of treating data about people. The data visualized in this project is the responsibility of the company using this system, so the company should be assured that its data is permitted to be treated. We recommend to be in compliance with the European General Data Protection Regulation (GDPR).

APPENDIX B

Economic budget

This appendix details an adequate budget for the realization of the project. The main parts of the budget are explained in the succeeding sections.

B.1 Physical resources

The main physical cost of the project is the computer needed to develop the system. This computer requires some features to run the system:

- **CPU:** Intel core i7.
- **RAM:** 8 GB.
- **Disk:** 256 GB.

The price of a computer with these characteristics is around 1.100€.

B.2 Human resources

This project has been carried in about 9 months working 20 hours per week. Considering a developer with no previous experience, it is estimated a salary of 6€/hour. With these, the cost of the developer would be:

9 months x 4 weeks/month x 20 hours/week x 6 €/hour = ***4.320€***

B.3 Licenses

In this project is used the free version of CanvasJS which is free for non-commercial uses. If this system is used to obtain economic benefits, you have to acquire a license. CanvasJS offers a Single Developer License which would be more than enough and its price is **399€**.

All the other software used in the project is totally Open-source thus it costs 0€.

B.4 Total budget

The next table shows all the costs involved in the development of the system:

Economic budget	
Entry	**Cost**
Computer	1.100€
Developer (Partial time)	4.320€
Software licenses (for commercial use)	399€
Total	**5.819€**

Lastly, we must consider the Spanish tax VAT. It supposes a 21% increase in the total cost. Applying it the final budget is: *5.819€ + 21% =* ***7.040,99€***

APPENDIX C

Test scenarios

There are four test scenarios for the user stories identified above.

C.1 TS-C.1

- **Identifier:** TS-C.1
- **Tests stories:** US-C.1.1
- **Test set-up:** IoT application data visualization
- **Preconditions:**
  - Pre#1: IoT distributed data repository.
- **Actions:**
  - A#1: Go to the corresponding URL in a web browser.
  - A#2: Choose the dashboard you want to visualize.
- **Results:**
  - R#1: A list of dashboards appears.

  - R#2: The dashboard is displayed.

C.2 TS-C.2

- **Identifier:** TS-C.2
- **Tests stories:** US-C.2.1
- **Test set-up:** IoT system data visualization
- **Preconditions:**
  - Pre#1: Topology management module.
- **Actions:**
  - A#1: Go to the corresponding URL in a web browser.
- **Results:**
  - R#1.1: The topology graph is displayed and there you can select the nodes you want to get information about.
  - R#1.1: The flows are represented in different colors to identify their origin and destination.

C.3 TS-C.3

- **Identifier:** TS-C.3
- **Tests stories:** US-C.3.1, US-C.3.2
- **Test set-up:** IoT dashboard design with predefined templates
- **Preconditions:**
  - Pre#1: IoT distributed data repository.
  - Pre#2: Predefined templates.
- **Actions:**
  - A#1: Go to the corresponding URL in a web browser.
  - A#2: Click in “Create new dashboard”.

  - A#3: Choose a template by selecting it in a list.
  - A#4: Match the data and widgets by choosing the data for each widget.
  - A#5: Click the button "Finish" to save the dashboard.

- **Results:**
  - R#1: Three options are available, "Create new template", "Dashboards" and "Create new dashboard".
  - R#2.1: All the data available in the platform is eligible with its keyword.
  - R#2.2: A list of predefined templates is displayed.
  - R#3: The template shows its widgets, indicating the type of data required.
  - R#4: The data name appears in the widget chosen.
  - R#5: The user is redirected to the main page.

C.4 TS-C.4

- **Identifier:** TS-C.4
- **Tests stories:** US-C.3.3
- **Test set-up:** IoT dashboard template design
- **Preconditions:**
  - Pre#1: IoT distributed data repository.
- **Actions:**
  - A#1: Go to the corresponding URL in a web browser.
  - A#2: Click in "Create new template".
  - A#3: Drag the widgets to the blank section.
  - A#4: Click the button "Finish" to save the template.
- **Results:**
  - R#1: Three options are available, "Create new template", "Dashboards" and "Create new dashboard".
  - R#2: All the widgets available to use in your template appear on the left with the type of data they require. On the right side, a blank section to insert the widgets

- R#3: A preview shows how does the template looks.
- R#4: The user is redirected to the main page.

Bibliography

[1] Jennifer Rowley. *The Wisdom hierarchy: representations of the DIKW hierarchy. Journal of Information Science*, 33(2):163–180, 2007.

[2] Miguel Ángel López Peña and Isabel Muñoz Fernández. *SAT-IoT: An Architectural Model for a High-Performance Fog/Edge/Cloud IoT Platform. Proceedings of the 5th. IEEE World Forum on IoT*, 2019.

[3] Amos Q Haviv. *MEAN Web Development*. Packt Publishing Ltd, 2014.

[4] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding Typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[5] Mike Cantelon, Marc Harter, TJ Holowaychuk, and Nathan Rajlich. *Node. js in Action*. Manning Greenwich, 2014.

[6] Evan Hahn. *Express in Action: Writing, building, and testing Node. js applications*. Manning Publications,, 2016.

[7] Jake Spurlock. *Bootstrap: Responsive Web Development*. " O'Reilly Media, Inc.", 2013.

[8] Thomas Elliott. The State of the Octoverse: top programming languages of 2018. URL. `https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/`. November 2018.

[9] Mike Cohn. *User stories applied: For Agile software development*. Addison-Wesley Professional, 2004.

[10] ISO/IEC. *Internet of Things - Reference Architecture*. Technical Report International Standard 30.141, International Organization for Standardization, International Electrotechnical Commission, 2018.