Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DO SOFTWARE

Euro-Inf Bachelor awarded by EQANIE

# Tool to automatically extract and import data from OpenStreetMap into relational databases

|  |  |
|---|---|
| **Estudante:** | David Gayoso Salvado |
| **Dirección:** | Alejandro Cortiñas Álvarez |
|  | Miguel Ángel Rodríguez Luaces |

A Coruña, junio de 2023.

*To my parents Juan and Conchi, I hope you are as proud of me as I am of you.*

**Acknowledgements**

Many of us who have done a end-of-degree project for the first time have been constantly pushing ourselves, not only with the deadlines for completing our studies and obtaining the much-prized degree, but also in more personal matters. This means that, as time goes by, everything revolves around this paper. In this situation I ask for forgiveness and at the same time I thank my friends for their patience, who understood me in the hardest moments of the process. I also want to thank my parents for their dedication. Without them and their unconditional support I would not have made it. Thank you for teaching me that hard work pays off and I am the one who must fight for what I want. I would also like to thank the CICAS chair for the funding received for the completion of this end-of-degree project. It is an incredible joy to see my work receives this kind of recognition. Finally, a special mention to some true commited people with the cause and who were my guide. Thanks to both my tutors for their constant follow up. I am grateful for having dedicated their time to direct my project and helping me through the whole process.

## Abstract

The objective of this end-of-degree project is to develop a tool which allows to extract and import automatically data from OpenStreetMap (OSM) to relational databases quickly and easily.

In order to achieve this goal, we defined a Domain Specific Language (DSL) which simplifies the requests to obtain the data and insert it on the corresponding database. Following up a library capable of interpreting and executing the language was designed, and finally a small web application to simplify the use of the previously mentioned library was implemented.

In the development, PostgreSQL is used for the storage of information, as well as Leaflet for data visualization on maps. Vue.js is used for the frontend as well as Spring for the backend.

The end-of-degree work is managed following an iterative and incremental methodology for software development.

## Resumo

El objetivo de este trabajo de fin de grado es desarrollar una herramienta la cuál permita extraer e importar automáticamente datos de OpenStreetMap (OSM) a bases de datos relacionados de manera rápida y sencilla.

Para lograr este objetivo, se definió un Domain Specific Language (DSL) que permitiera simplificar las peticiones para obtener los datos. A continuación, se desarrolló una librería capaz de interpretar y ejecutar el lenguaje diseñado y, finalmente, se creó una aplicación web que simplica el uso de la librería previamente mencionada.

En el desarrollo, se usa PostgreSQL como almacén de datos, así como Leaflet para visualizar los datos obtenidos en mapas. Se utiliza Vue.js para implementar el frontend y Spring para el backend.

El trabajo de fin de grado se gestiona siguiendo una metodología iterativa e incremental.

**Keywords:**

- OpenStreetMap
- Web application
- Java
- Vue.js
- PostgreSQL
- Spring Boot
- Leaflet
- Git
- Tasks
- Databases
- Users

**Palabras chave:**

- OpenStreetMap
- Aplicación web
- Java
- Vue.js
- PostgreSQL
- Spring Boot
- Leaflet
- Git
- Tareas
- Bases de datos
- Usuarios

# Contents

# List of Figures

# List of Tables

Chapter 1

# Introduction

This chapter discusses the main motivation for this project and the objectives it must meet in order to be successful.

## 1.1 Motivation

OpenStreetMap (OSM) [6] is a community platform for geographic data. It started as a free alternative to Google Maps, but over time it has become the leading repository of freely available geographic data. In OSM, very heterogeneous data can be found since the users themselves are in charge of digitizing the information, accessing it from any device to indicate the presence of any type of geographically localized element, usually real estate, but also road signs, traffic signals, zebra or barriers. OSM data set can be consulted in different ways, including dowloading data from a raw mapa section or using Overpass API [7] which allows to perform requests with advanced filters to recover only the neccessary data.

Currently, developers have to import the data manually, which indeed is not an easy process since it requires highly specific knowledge on how to work with raw data from OSM or the query syntax from Overpass API, which is extremely complex. If we had a Geographic Information System (GIS), we may want to use data from OSM, so then we would have to go through the tedious and manual process described above. If we had to do the same process for several GIS systems, it's a lot of repetitive work. The realization of this project arises from the need of obtaining data from OpenStreetMap in a simple and efficient way.

In this end-of-degree project the main goal is the development of a tool which simplifies the data integration from OpenStreetMap into relational databases. The application is called OSMParser and will allow users with different needs to import data, avoiding the use of an extremely complex API.

To achieve this goal, we will define a Domain Specific Language (DSL), design and implement a library capable of interpreting and executing this language, and finally develop a small web application that simplifies the use of this library.

## 1.2 Objectives

From the main objetive that has been described above, this project must meet the following specific requirements:

- Analysis of both the data and query language used by OpenStreetMap, and extraction of the tool requirements.
- Definition of a domain specific language that allows to perform both the data extraction and importation. This language must allow to execute the mapping between the data from OpenStreetMap and the data schema from the relational database.
- Design and implementation of a programming library responsible of analyzing instances of the defined language.
- Both the language and the programming library to be defined must be easy to use.
- Automate the process of inserting OSM data into relational databases.
- Design and implementation of simple and intuitive web application which acts as an interface for the library. The application must be capable of executing import tasks in a parallel way.
- Allow the visualization of previously imported data on a map viewer.

Chapter 2

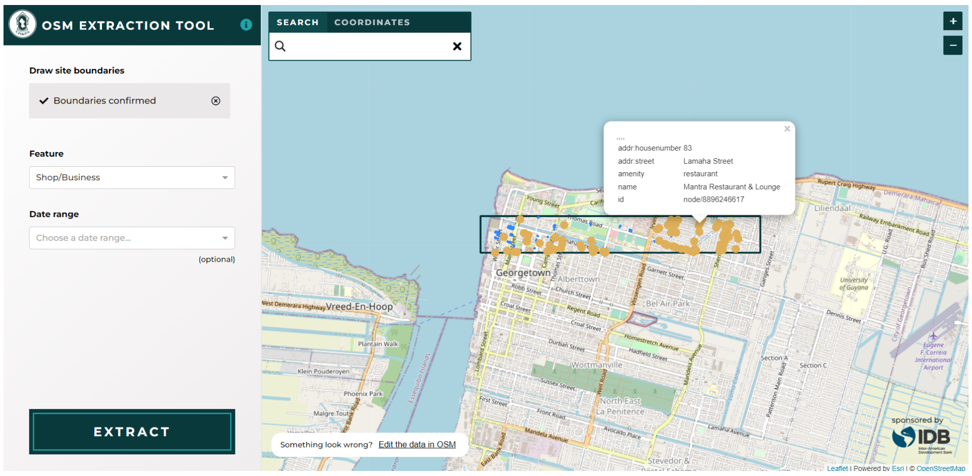
# Technology fundamentals

This chapter shows similar applications in which the author has based his work and the different technologies used in the project.

## 2.1 State of the art

Based on the objectives described in the previous section, a search for both tools and applications that can support our needs has been carried out. Among the different applications found, two of them stand out: OSM Extraction Tool and Overpass Turbo.

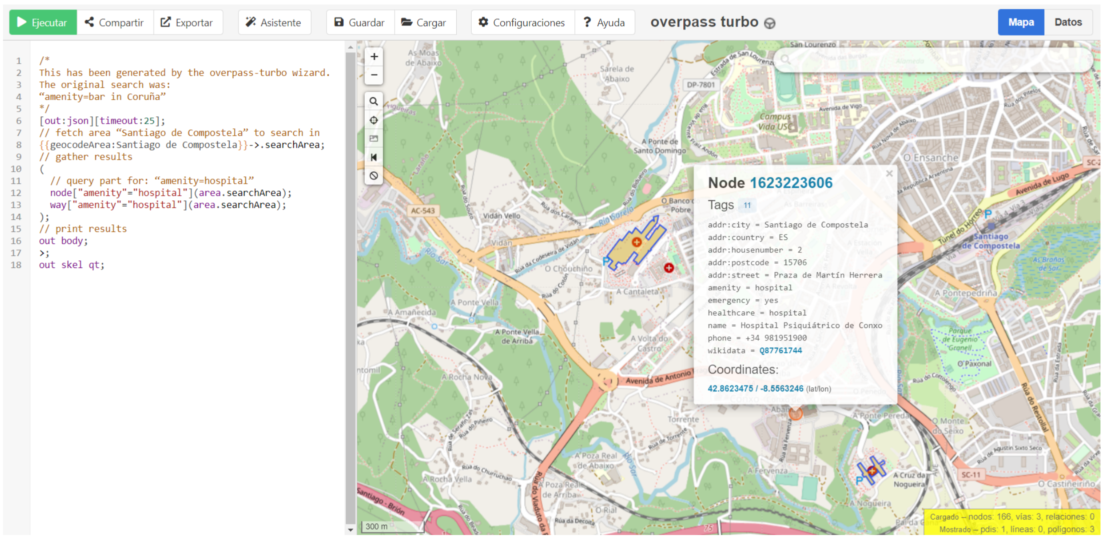
**OSM Extraction Tool** [8] is a web application that provides a user-friendly interface to facilitate access to OSM data, built on the basis of the Overpass API. The application allows to navigate to areas of interest on the map, access data generated by mass collaboration easily and download features of interest (selected using a menu of options) of OpenStreetMap. Its main disadvantage is that it does not allow to perform any type of query for the Overpass API.



| | |
|---|---|
| addr:housenumber | 83 |
| addr:street | Lamaha Street |
| amenity | restaurant |
| name | Mantra Restaurant & Lounge |
| id | node/8896246617 |

Figure 2.1: Feature search screen

**Overpass Turbo** [9] is a web based data mining tool for the Overpass API. It should be noted an interesting feature: it allows to export the data, the query and the map itself, all of them in different formats. Unlike the previous tool, it can execute any type of query from the Overpass API and display the data on a interactive map. Moreover, its filtering system is more advanced than the rest of alternatives. However, the API query syntax is extremely difficult and tedious and makes it hard to perform complex queries.



```
/*
This has been generated by the overpass-turbo wizard.
The original search was:
“amenity=bar in Coruña”
*/
[out:json][timeout:25];
// fetch area “Santiago de Compostela” to search in
{{geocodeArea:Santiago de Compostela}}->.searchArea;
// gather results
(
  // query part for: “amenity=hospital”
  node["amenity"="hospital"](area.searchArea);
  way["amenity"="hospital"](area.searchArea);
);
// print results
out body;
>;
out skel qt;
```

Node 1623223606

Tags 11

```
addr:city = Santiago de Compostela
addr:country = ES
addr:housenumber = 2
addr:postcode = 15706
addr:street = Praza de Martín Herrera
amenity = hospital
emergency = yes
healthcare = hospital
name = Hospital Psiquiátrico de Conxo
phone = +34 981951900
wikidata = Q87761744
```

Coordinates:

42.8623475 / -8.5563246 (lat/lon)

Figure 2.2: Feature search screen

On top of that, there is one more possible alternative apart from the ones mentioned before. As said before in the previous chapter, developers could import the data manually, that is: either downloading raw data and then processing, filtering and uploading it to the corresponding database, or creating a script to execute queries using the Overpass API and insert the results into the database. However, both cases are not easy to implement, since it is required deep knowledge on how to work with OSM raw data or how query syntax from Overpass works (which is very complex). In addition, in any case it would be necessary to perform the transformation between the data in the OSM model and the concrete domain from the database.

After analyzing all the alternatives explained in this section, it has been checked that the current alternatives do not meet all the project objectives. In order to **develop a tool which simplifies the data integration from OpenStreetMap into relational databases** the previous alternatives will be taken as reference in terms of data collection, data visualization,... and new features like the access to different databases and users management will be added.

## 2.2 Used technologies

- **Spring Boot** [10], is a framework that facilitates the configuration of a Spring project, making it automatic.
- **Spring Data JPA** [11], is a Spring framework which simplifies data persistence in a relational database.
- **PostgreSQL** [12], is an object-oriented relational database management system. It features multiversion concurrency control (MVCC) and the use of read locks.
- **PostGIS** [13], is a PostgreSQL extension which allows to save geometric data on any database.
- **ANTLR** [14], is powerful parser generator for reading, processing, executing, or translating structured text or binary files.
- **Vue.js** [15, 16], is a progressive framework for building user interfaces. Its core is quite small and scales through plugins. It includes in the same file HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript.
- **Node.js** [17], is a development platform for the creation of applications for the Web, network-oriented and focused on speed and scalability.
- **CSS (Cascading Stylesheets)** is the style language used to describe the design of a Web page (colors, elements layout and fonts). Even though it is independent from HTML (HyperText Markup Language), it is often used in conjunction with it.
- **Javascript** is the language used in Web pages in order to increase their functionality and the user interaction.
- **Leaflet** [18], is an Open Source JavaScript library for map creation and geographic data visualization.
- **Bootstrap** [19], is a toolkit for HTML, CSS and JavaScript development that provides Sass variables to customize the interface.

Chapter 3

# Methodology and planning

This chapter explains the development methodology followed in this project and the planning carried out.

## 3.1 Development methodology

The project realization process is divided into two really distinct stages.

During the first stage (during the 2021/2022 course), the author started the design of the DSL and many of its properties, as well as implementing support for Overpass API requests and the possibility of importing data in a specific type of relational database. However, no user stories and sprints were described and the project objectives were not clear and well defined. This is because no formal methodology was followed. Simply, the student worked 4 hours per week at the Database Lab (LBD) of the University of A Coruña and had several meetings with the project managers during this period to review the work done, so things were being implemented with sense.

It is important to mention that the student wrote a short article [20] describing what had been implemented so far, and it was presented at the JISBD conference in September. The presentation of this article means the completion of the first part of the process.

The second stage, which started at the end of 2022 once the project objetives were clear, was carried out following an iterative and incremental methodology driven by the system functionalities. Each iteration included the analysis, design, implementation and test phases and has led to an increase in functionality to a deliverable product.

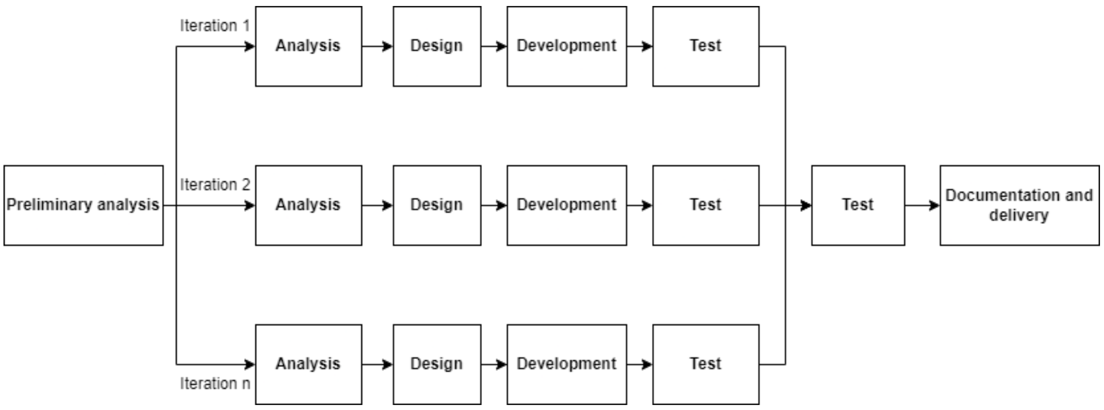


Figure 3.1: Incremental Methodology

The first part was the **preliminary analysis**, which consisted mainly in selecting the functionalities to be added. Then began the study of the neccessary technologies. Thus, both a search and an analysis of different tools was carried out, of which the ones on Section 2.2 were selected. Finally, after studying the application global objective it was neccessary to start thinking about the individual objectives of each iteration. A starting point was previously built for this purpose:

- **Mockups**: prototypes of the web application were created. They were used in conjunction with the results obtained by the preliminary analysis to plan the iterations.
- **Data model**: the different data that the system must manage and its relationships were defined with this model.

Regarding the structure of the different iterations, each one of them included the following steps:

- **Analysis**: the new iteration was analyzed taking into account the requirements.
- **Design**: the different modules were defined carefully since they can be reused in the following iterations.
- **Development**: what was defined in the previous phase was implemented, defining both the entities and methods to carry out the defined features.
- **Tests**: first unit tests and then global ones of the created functionalities were developed, which checked that the implemented met the expectations and had an accurate behaviour.

Once the iterations were completed, tests and documentation were carried out. The author carried out tests to check that the relationships between the different modules work as expected, that is, that this modules work together to offer the desired functionality. Regarding documentation, it consists of the writing of this report.

The **tools used** are the following ones:

- **Latex** [21]: is a text composition tool. It has been used in the writing of this report.
- **Postman** [22]: tools that allows to perform HTTP requests to any Application Programming Interface (API) to check that all our REpresentational State Transfer (REST) connection works perfectly. It was used in the test phase to try the REST service.
- **IntelliJ IDEA** [23]: it is nothing more than an Integrated Development Environment (IDE). It was used in the implementation of all the application.
- **Balsamiq Mockups** [24]: tool that allows to design interface prototypes. It was used to create a first idea of how the web application would look like.
- **Draw.io** [25]: web application which allows to create any type of diagram.
- **Gitlab** [26]: web-based version control and collaborative development service based on Git [27]. In addition to being a repository manager, it also offers wiki hosting and a bug tracking system.

## 3.2 Planning and monitoring

This section shows the initial project planning and estimation. Finally, the deviations produced and the cost will be taken into account. The first stage of the development of the project, as explained before, was not planned, but consisted of a research work and it is therefore not described in this section.

The work is divided into 5 development iterations with a duration from 2 to 3 weeks, depending on the workload and the author's availability. The **tasks** to be carried out on each iteration are the following:

- **First iteration**: this iteration covered everything related to the users managemement (Log in, Sign up,...) and part of the Database module (Add, list, delete and update operations).
- **Second iteration**: focused on tasks, that is, creation, cancellation, filtering and seeing execution report.
- **Third iteration**: this iteration was used to fix errors from previous iterations and to add a feature to filter databases.

- **Fourth iteration**: finished the Task module, that is, adding the final features of the module.
- **Fifth iteration**: the last one concentrated on finishing the Database part and adding the visualization of data on a map viewer.

In terms of **resources** for this project we can distinguish two different roles:

- **Managers**: they were in charge of both the planning and supervision of both the overall project and the iterations. In terms of the communication between the managers and the authors follow-up meetings were carried out every 2 weeks approximately, depending on the availability of the assistants, in which what to do in the next iteration was planned and the previous was reviewed.
- **Analyst/Programmer**: this is the role of the author, who was in charge of the analysis, design, implementation and tests of every iteration to be performed.

For the **planning** a Gantt Diagram was made which exposes as tasks the iterations mentioned on Section 3.2. The start and end dates of the project would correspond to November 15, 2022 and June 1, 2023. It is important to mention that the project was interrumpted two times, although these interrumptions were planned from the beginning so they did not affect to the project original planification:

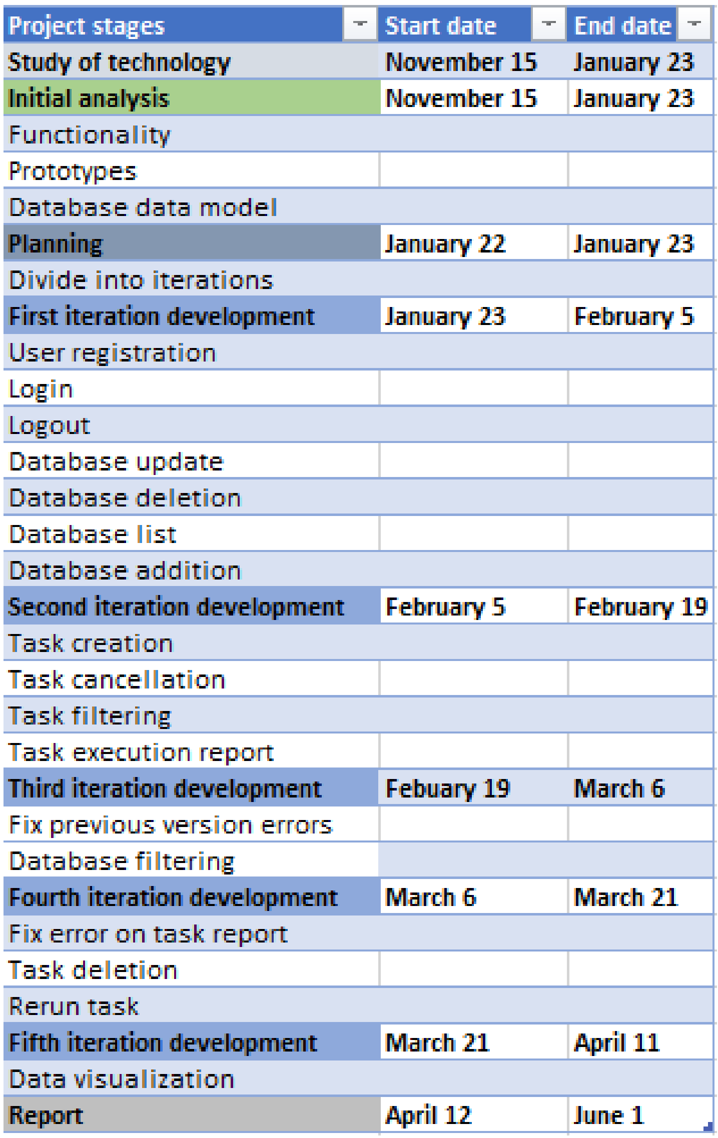
- **From January 9 to 19**: the cause was the preparation for the first term final exams.
- **From April 1 to 8**: because of the author's holy week vacations.

In terms of **costs**, the author worked **5 hours a day** approximately every day of the week, which means **35 week hours approximately**. Taking into account that the duration of the project amounted to approximately 200 total days, the total hours amounted to 1000 hours. On the other hand, the project managers dedicated approximately **40 hours** between meetings and the report revision. Resources' costs have been obtained the following way:

| Actor | Hours/person |
|---|---|
| David Gayoso Salvado | 1000 |
| Alejandro Cortiñas Álvarez | 40 |
| Miguel Ángel Rodríguez Luaces | 40 |

Table 3.1: Human resources costs

With regard to **monitoring**, during the project development there were deviations in the duration of some tasks. But these deviations in task durations, being only of a few days, ended up being compensated in later iterations, so the final effort for project completion did not vary. It was only included the Monitoring Gantt Diagram which we can observe at Figure 3.2 due to the low difference regarding the Initial Planning.



| Project stages | Start date | End date |
|---|---|---|
| Study of technology | November 15 | January 23 |
| Initial analysis | November 15 | January 23 |
| Functionality | | |
| Prototypes | | |
| Database data model | | |
| Planning | January 22 | January 23 |
| Divide into iterations | | |
| First iteration development | January 23 | February 5 |
| User registration | | |
| Login | | |
| Logout | | |
| Database update | | |
| Database deletion | | |
| Database list | | |
| Database addition | | |
| Second iteration development | February 5 | February 19 |
| Task creation | | |
| Task cancellation | | |
| Task filtering | | |
| Task execution report | | |
| Third iteration development | Febuary 19 | March 6 |
| Fix previous version errors | | |
| Database filtering | | |
| Fourth iteration development | March 6 | March 21 |
| Fix error on task report | | |
| Task deletion | | |
| Rerun task | | |
| Fifth iteration development | March 21 | April 11 |
| Data visualization | | |
| Report | April 12 | June 1 |

Figure 3.2: Monitoring

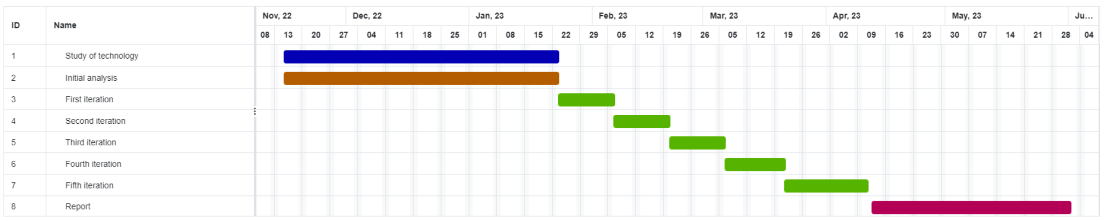


Figure 3.3: Monitoring Gantt Diagram

Chapter 4

# Analysis

In this chapter of the report we focus on the application analysis, describing the different requirements of both the web application and the parsing library, the system architecture and the data using a Class Diagram.

## 4.1 Requirements

### 4.1.1 Web application

One of the steps we have to perform before the development to describe the behaviour of the system is the Software Requirements Specification (SRS). The different final requirements are listed and detailed below:

#### User stories - Anonymous user

This actor refers to the user who can only login or register on the application.

- **US01 - Register**: the user will be able to register on the system after writing several valid fields through a registration form.
- **US02 - Login**: the user will be able to login on the application writing a valid username and password through a login form..

#### User stories - User

This actor refers to the normal user who is able to use any of the application available functionalities.

- **US03 - Logout**: an user will be able to logout from the application at any moment.

- **US04 - Create tasks**: the users will be able to create a new task. To do so, they must fill out a form with the corresponding parameters: the name of the task and the query itself. Users must specify the databases where they want to store the data, either using the corresponding option on the form itself or entering the database connection on the query.
- **US05 - Add database**: users will be able to add a new database to their account. They must enter several fields such as the name of the database, a valid port number, the host, the user, the password and the type of database. A notification showing the success or not of the operation will be shown.
- **US06 - My tasks**: a list of all tasks previously executed by the user will be displayed.
- **US07 - Filter tasks**: users will be able to filter their previously executed tasks by different parameters, such as the name of the task or the start and end date.
- **US08 - My databases**: a list of all databases previously added by the user will be displayed.
- **US09 - Delete database**: it will be possible to delete a database at any moment.
- **US10 - Update database**: users will be able to update the connection parameters of a database through an update form. A notification showing the success or not of the operation will be shown.
- **US11 - Filter databases**: users will be able to filter their previously added databases by different parameters, such as the name of the database or the host.
- **US12 - Cancel task**: users will have the option to cancel a running or waiting task if needed.
- **US13 - See task execution report**: users will be able to see the execution process of a task in real time. A notification will be displayed showing the success or failure of the task execution (only if the user is in the task report view).
- **US14 - Data visualization**: once the users have executed a task, they will be able to visualize the data imported on a interactive map viewer.
- **US15 - Rerun task**: it will be possible to rerun again a previously executed task. The users will be redirected to the form to create a new task, so they have the possibility to keep the task previous parameters or change them.
- **US16 - Delete task**: it will be possible to delete a database if needed.

### 4.1.2 Parser library

The parsing library must meet some requirements in order to be able satisfy the needs of the web application.

- **US17**: design of a declarative language composed of instructions that allow users, in a simple and convenient way, to describe which OSM data set they want to retrieve, and how the data should be transformed to fit the schema of the database where they want to import it.
- **US18**: support for specifying multiple database connections of different types on the same query.
- **US19**: support for carrying out several tasks in a parallel way.
- **US20**: allow to apply functions on the retrieved properties, e.g. type transformation or spatial operations to calculate new geometries, defining a series of operations specific to the DSL, but also allowing the execution of any function provided by the database.
- **US21**: possibility to add a default value for any attribute in case OSM does not retrieve any value for that specific feature.

## 4.2 System architecture

The designed system follows a layered architecture, composed of a Web Server, a Client Web and a self-developed library used by the Web Server. The client consumes data from the Web Server using the REST architecture, while the library is a dependency of the Web Server itself. This architecture is useful, since it keeps a total independence between the different layers.

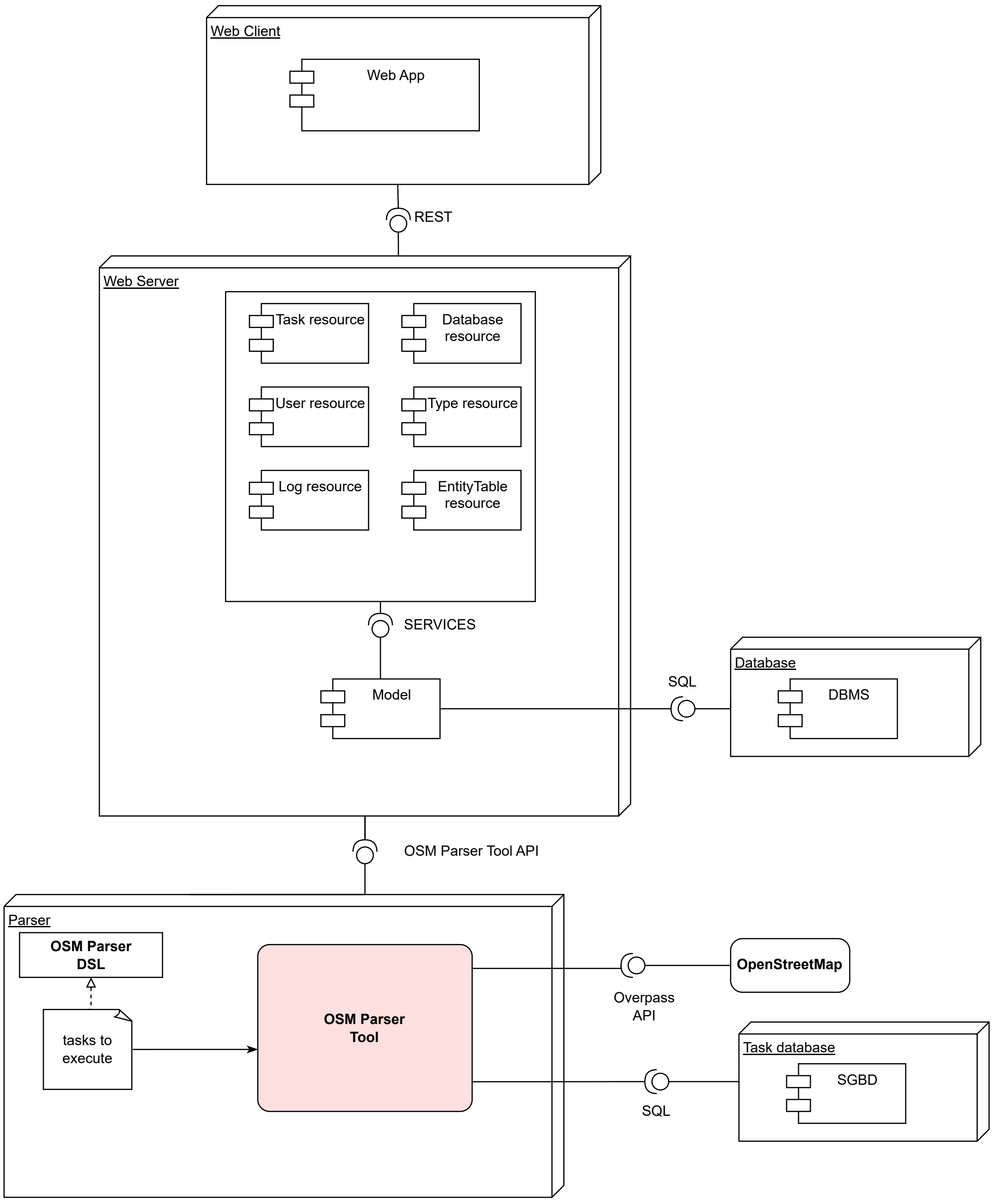


Figure 4.1: System architecture

## 4.3 User interface

The developed user interface is a web application which allows the user to use all the functionalities derived from the system requirements. Its main purpose is to ease the user interact with the application and to act as the entry point for the rest of the system. The interface is easy to handle and is very intuititve, but the target audience, as well as the system itself, is domain expert users.

Some of the most importants screens are described below:

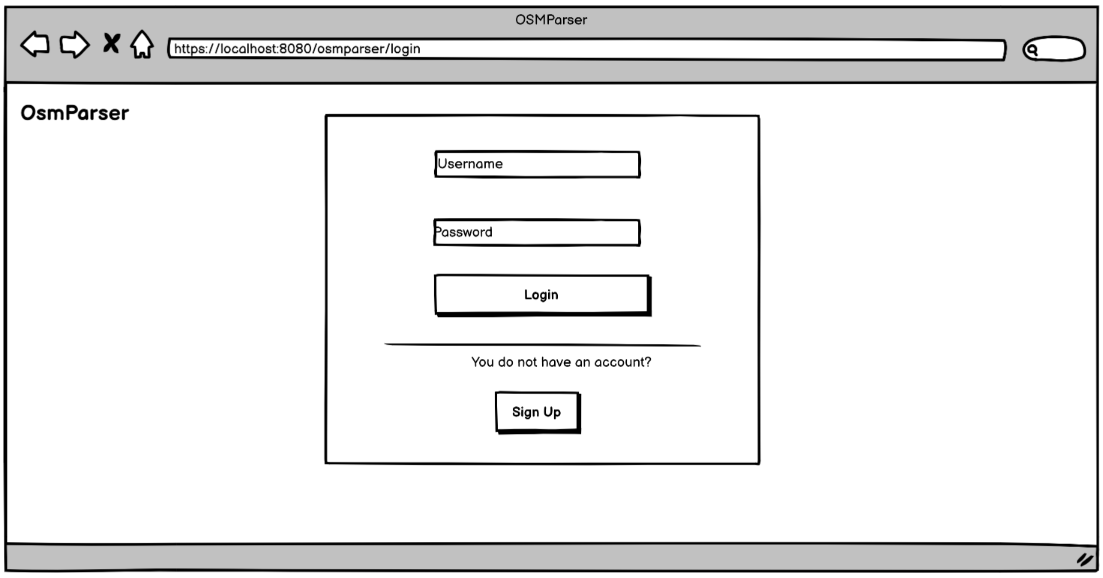
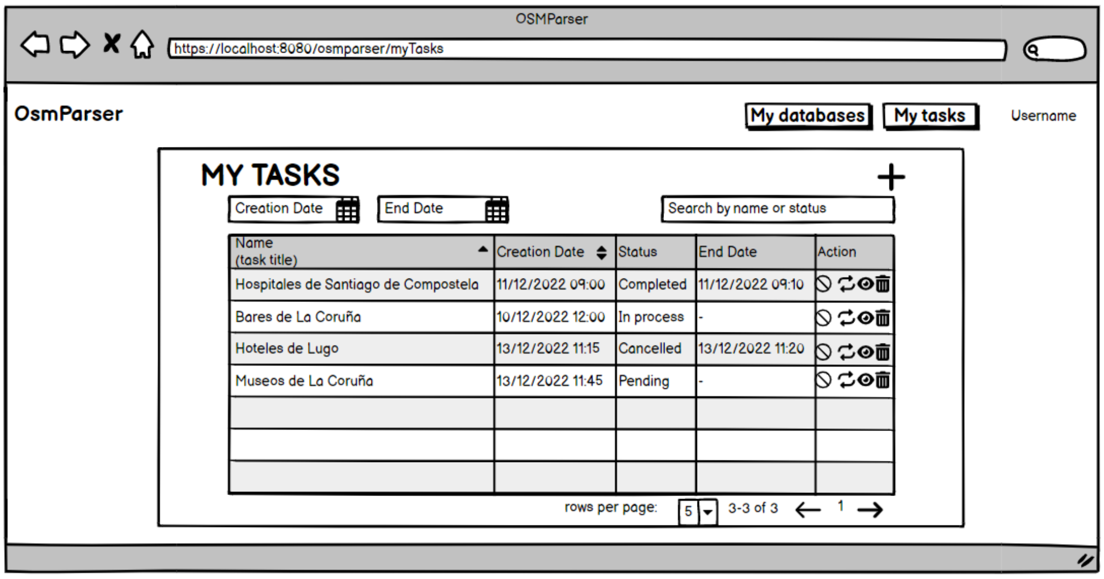

OSMParser
https://localhost:8080/osmparser/login
OsmParser
Username
Password
Login
You do not have an account?
Sign Up

Figure 4.2: Screen prototype - Login

On Figure 4.2 is shown how the users can access to the application after entering their credentials. After introducing the credentials and validating them, the user is redirected to the main page, as shown on Figure 4.3.

Figure 4.3 shows the view after log in on the application. It is possible to observe a horizontal menu on top which will allow to access to different sections. Furthermore, from this screen the user will be able to filter the different tasks and to perform several actions like deleting a task or see the task execution report.



| Name (task title) | Creation Date | Status | End Date | Action |
|---|---|---|---|---|
| Hospitales de Santiago de Compostela | 11/12/2022 09:00 | Completed | 11/12/2022 09:10 | ⊘ ⟳ 👁 🗑 |
| Bares de La Coruña | 10/12/2022 12:00 | In process | - | ⊘ ⟳ 👁 🗑 |
| Hoteles de Lugo | 13/12/2022 11:15 | Cancelled | 13/12/2022 11:20 | ⊘ ⟳ 👁 🗑 |
| Museos de La Coruña | 13/12/2022 11:45 | Pending | - | ⊘ ⟳ 👁 🗑 |
| | | | | |
| | | | | |
| | | | | |

Figure 4.3: Screen prototype - Home

After clicking on the '+' button, the users will be redirected to the section where they can create a new task, whose design is shown on Figure 4.4. After the user introduces the task parameters and executes the task, he/she will be redirected again to the home screen where he/she will be able to see the task he/she have just executed.

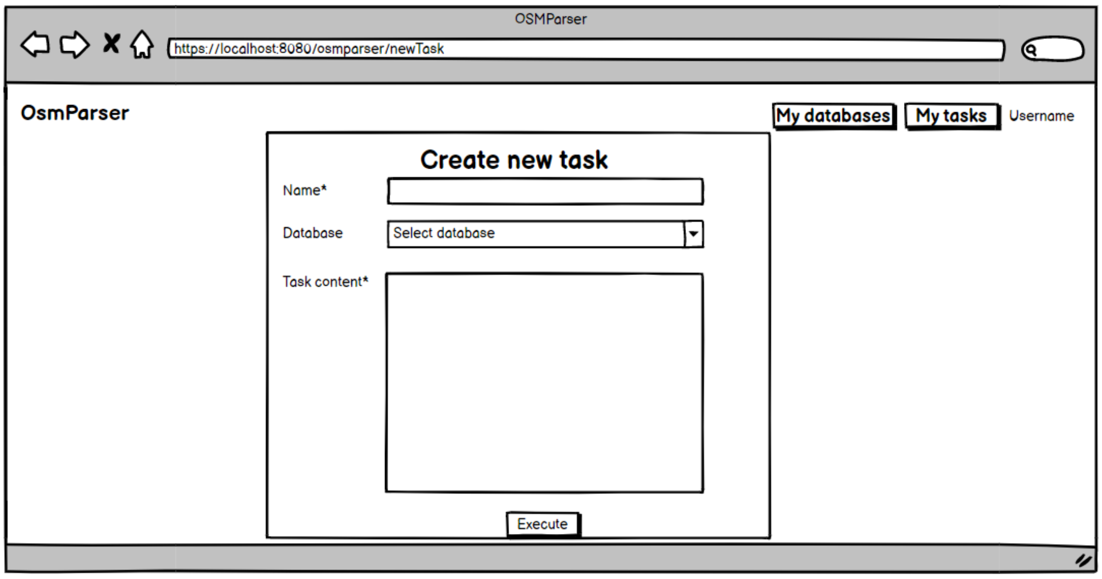


Figure 4.4: Screen prototype - Create task

Once the task is created and its execution finished correctly, users will be able to see the data imported on an interactive map viewer. Figure 4.5 shows the design for this view of the application.

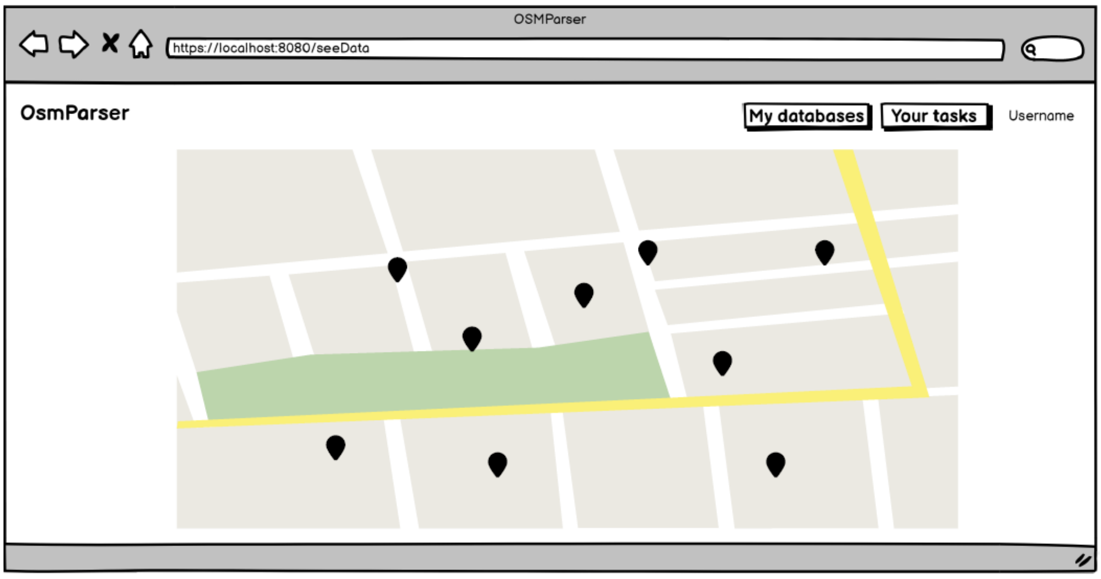


Figure 4.5: Screen prototype - Data visualization

Figure 4.6 shows the screen where the users can add a new database to be used in any of their tasks. After introducing the parameters, the system will check the connection to the database with the given parameters and will warn the user in case of any error.

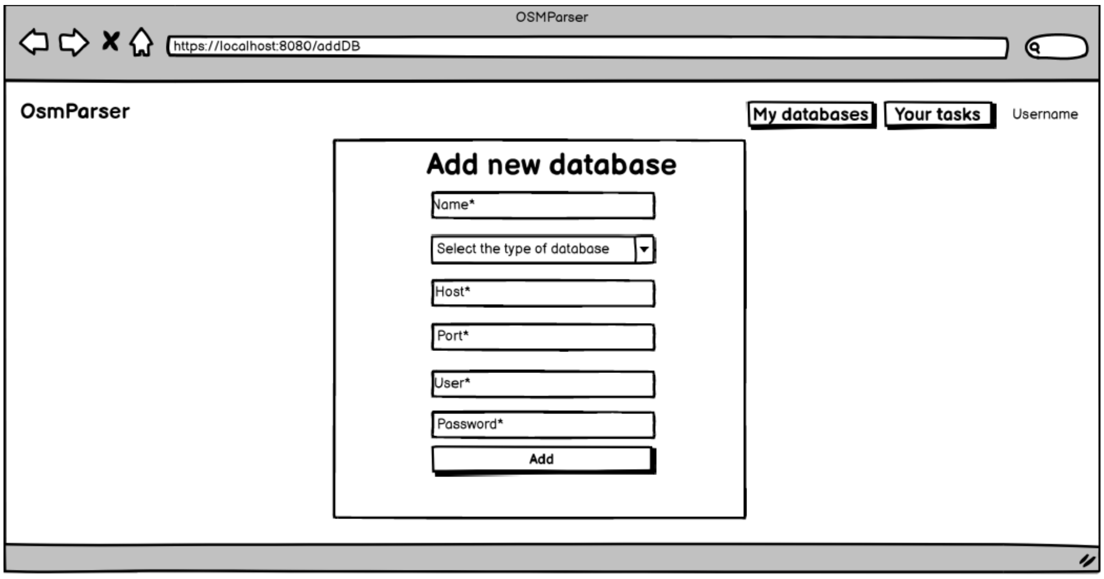
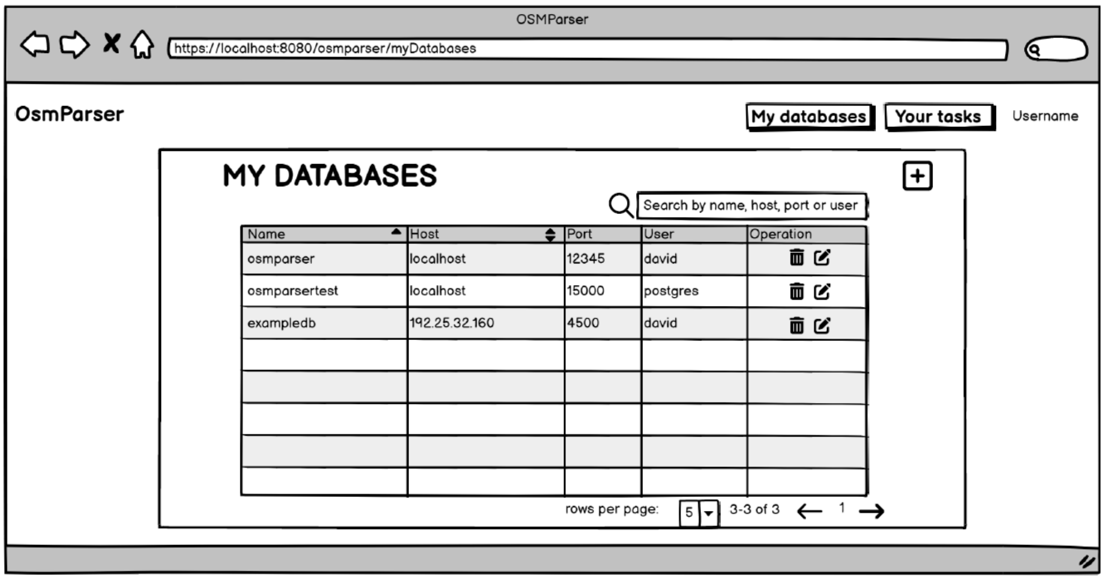


Figure 4.6: Screen prototype - Add new database

If the parameters are correct and the system can establish a connection with the database, the user will be redirect to the view shown on Figure 4.7. On this page, the user can see all his/her previously added databases, filter them and perform several actions, like deleting the database or updating it. Figure 4.8 shows the view which allows the user to update any database.



OSMParser

https://localhost:8080/osmparser/myDatabases

OsmParser

My databases | Your tasks | Username

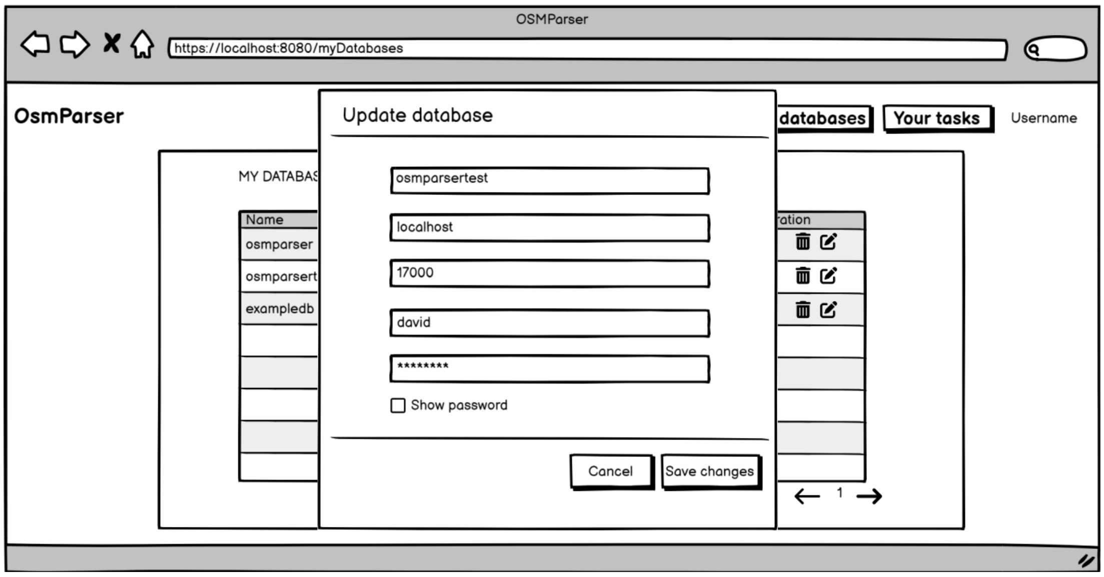
MY DATABASES

Search by name, host, port or user

| Name | Host | Port | User | Operation |
|---|---|---|---|---|
| osmparser | localhost | 12345 | david | |
| osmparsertest | localhost | 15000 | postgres | |
| exampledb | 192.25.32.160 | 4500 | david | |

rows per page: 5 3-3 of 3 1

Figure 4.7: Screen prototype - My databases



OSMParser

https://localhost:8080/myDatabases

OsmParser

Update database

osmparsertest

localhost

17000

david

********

Show password

Cancel | Save changes

Figure 4.8: Screen prototype - Update database

The prototypes created in the initial analysis phase, which could not be included here due to space constraints, can be visualized on Appendix B.

## 4.4 Conceptual data model

The Web server will use a relational database, which will store all the information related with users, tasks, databases, the different database types,... It will contain the information neccessary for the system to work correctly. The following image shows a general view of all the entities and their relationships. The different entities from the model together with their attributes will also be shown.

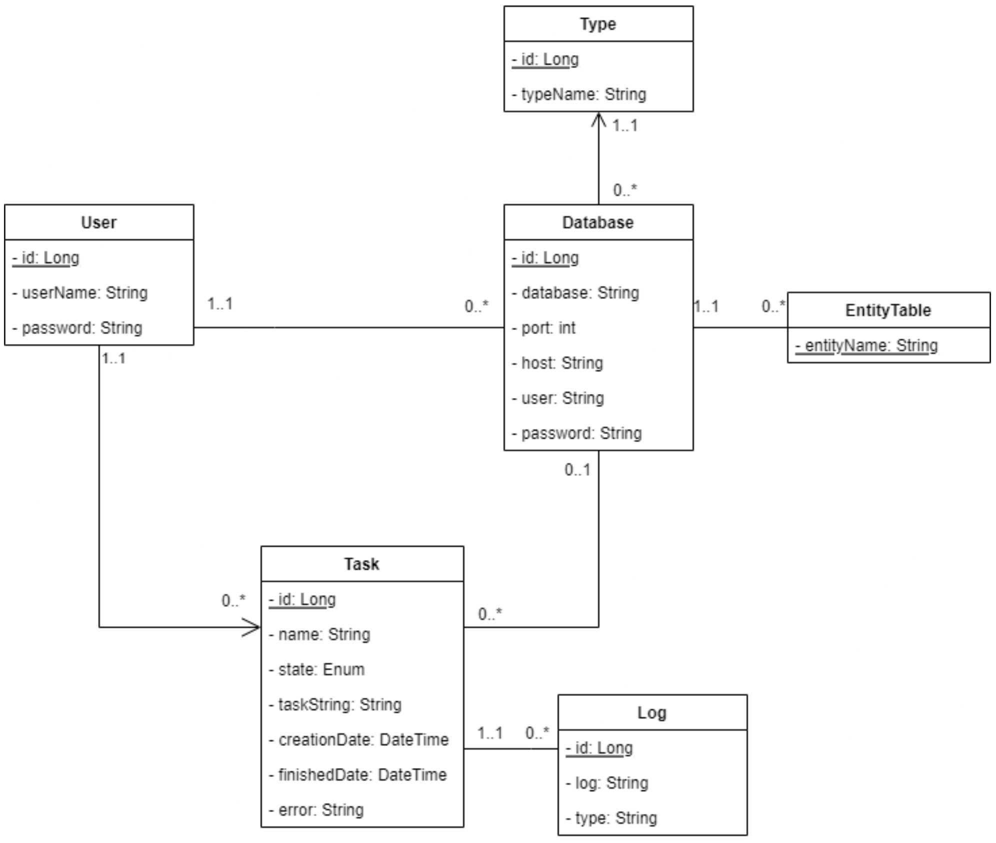


Figure 4.9: Application data model

| **User** | Represents all the users from the system | **Type** |
|---|---|---|
| **id** | Unique user identifier | Long |
| **userName** | Username which allows user to log in | String |
| **password** | Password which allows user to log in | String |

Table 4.1: User entity

| **Database** | Represents all the databases from the system | **Type** |
|---|---|---|
| **id** | Unique database identifier | Long |
| **database** | Field which contains database name | String |
| **port** | Field which contains database port | Integer |
| **host** | Field which contains database host | String |
| **user** | Field which contains database user | String |
| **password** | Field which contains database user | String |

Table 4.2: Database entity

| **Type** | Represents all the supported database types | **Type** |
|---|---|---|
| **id** | Unique type identifier | Long |
| **typeName** | Field which contains type name | String |

Table 4.3: Type entity

| **EntityTable** | Entities associated with a database | **Type** |
|---|---|---|
| **entityName** | Unique type identifier | Long |

Table 4.4: EntityTable entity

| **Task** | Represents all the tasks from the system | **Type** |
|---|---|---|
| **id** | Unique task identifier | Long |
| **name** | Field which contains task name | String |
| **state** | Status of the task | Enum |
| **taskString** | Query text of the task | String |
| **creationDate** | Date representing the beginning of the task | DateTime |
| **finishedDate** | Date representing the end of the task | DateTime |
| **error** | Error ocurred in the task if any | String |

Table 4.5: Task entity

| **Log** | Represents all the execution logs from a task | **Type** |
|---|---|---|
| **id** | Unique log identifier | Long |
| **log** | Text of the log | String |
| **type** | Type of the log | String |

Table 4.6: Log entity

It is important to mention that on Table 4.4 we have a weak database entity, so it is identified by the database itself and its name.

Chapter 5

# Design

This chapter focus on the design phase of the application. It starts with a description of the system architecture, then the logic data model and, finally, the application design, both back-end and front-end, is discussed.

## 5.1 Technological system architecture

When carrying out the implementation of this system, it was previously mentioned that it has been decided to implement a layered architecture.

PostgreSQL has been chosen to use for the database layer of the Web server component. Data Access Object (DAO) is used for the connection between the model and the server. These DAOs communicate with the service layer implemented with Spring and JPA (using Spring Data).

REST is used to connect the Web server and the Web client. This client is implemented with Vue.js and will make use of Leaflet, a Javascript library to create interactive maps and Bootstrap, a CSS framework.

ANother Tool for Language Recognition (ANTLR) is used to create the DSL used by the library component and Overpass API to import the data from OpenStreetMap.

This application is divided in three parts. The front-end part, the client, is the one the user interacts with, the back-end part, the service, the part which the user cannot see and is in charge of executing user requests and the parser, the library, the part which is in charge of performing the requests to the Overpass API and obtaining data from the tasks started by the users.

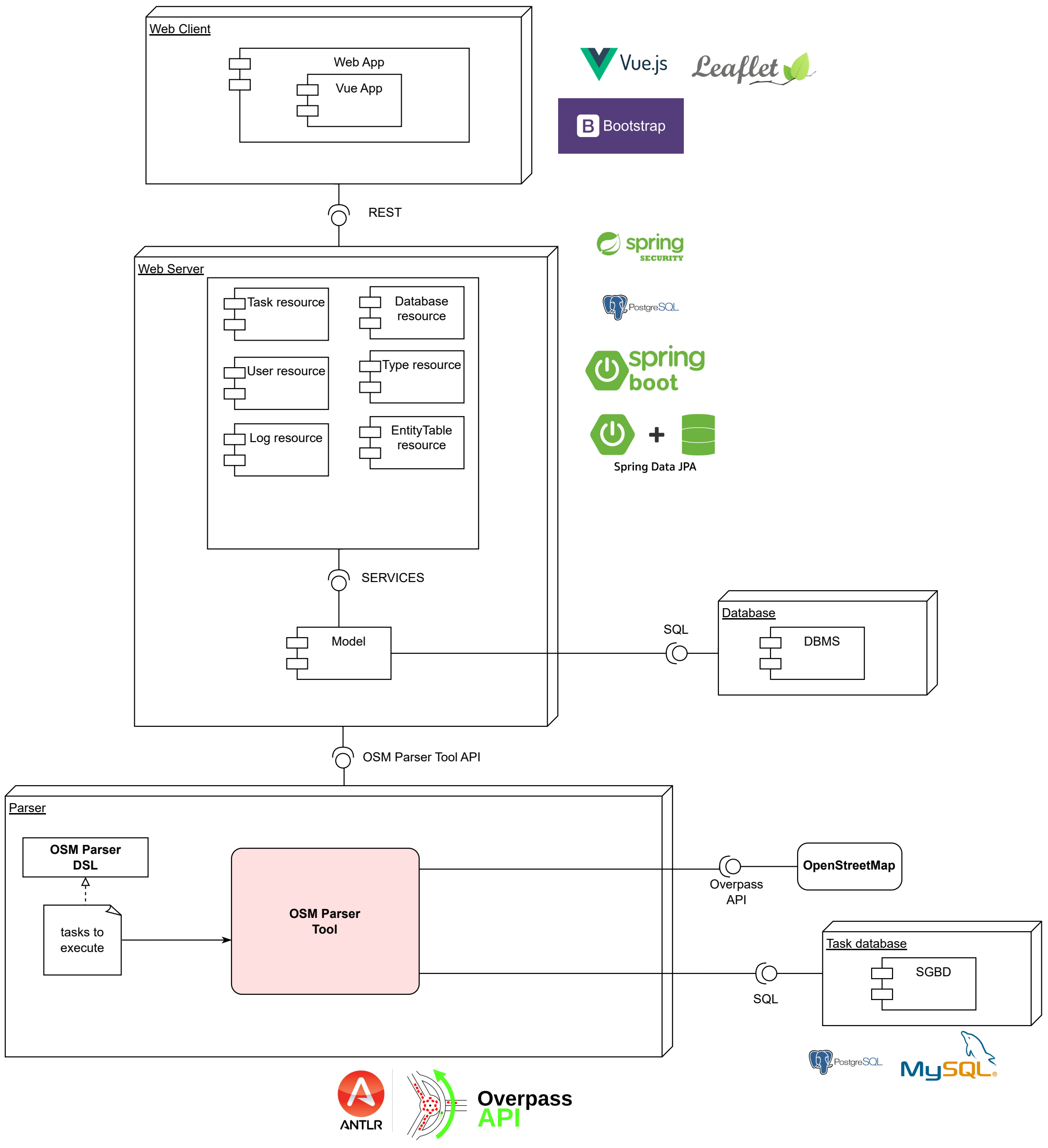


Figure 5.1: Technological system architecture

## 5.2 Application design

To explain the application design, it is divided into front-end, back-end and the Parser library component. The front-end is in charge of collecting user inputs and transforming and adjusting them to the back-end specifications. The back-end is in charge of receiving the data, processing them, returning an answer according to the business logic and persisting the necessary information. The front-end receives the response and shows it in a comprehensible manner for the final user. Finally, the library component is in charge of making the requests to the Overpass API to obtain the requested elements and to save the received data in the corresponding database.

### 5.2.1 Back-end

The back-end is a REST server. On the following figure is shown the server diagram, which contains the system model structure, which includes the data persistence layer, the access to them and the distinct fachades the services offer to the upper layers. It also contains the different REST controllers, which allow the communication between the Web client and the Web server.

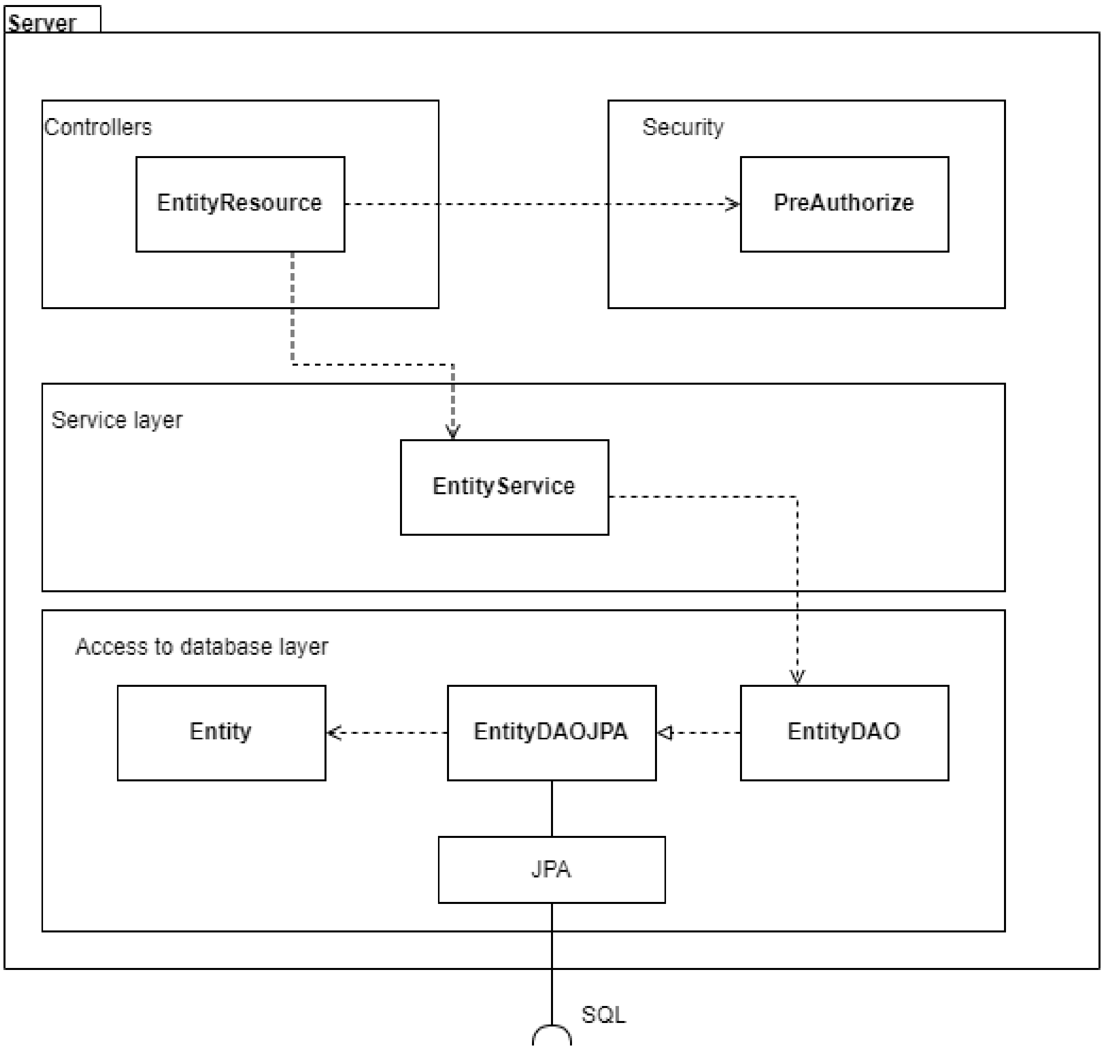


Figure 5.2: Server diagram

### Data access

DAO design pattern is used for the data access. This pattern allow the data access and, in turn, allows to hide the way they are accessed. For DAOs organization, they are grouped in the same package.

- *es.udc.fig.tfg.backend.model.entities*: package where both entities and their corresponding DAOs are allocated.
- *es.udc.fic.tfg.backend.model.entities.nameDAO.java*: DAO interface. The class *Entity-DAOJPA* from the Figure 5.2 is automatically implemented by Spring Data. However, it allows to add custom methods if needed.

The following image show one of the the most important DAO class diagram. Not all of them are shown because they are the same.

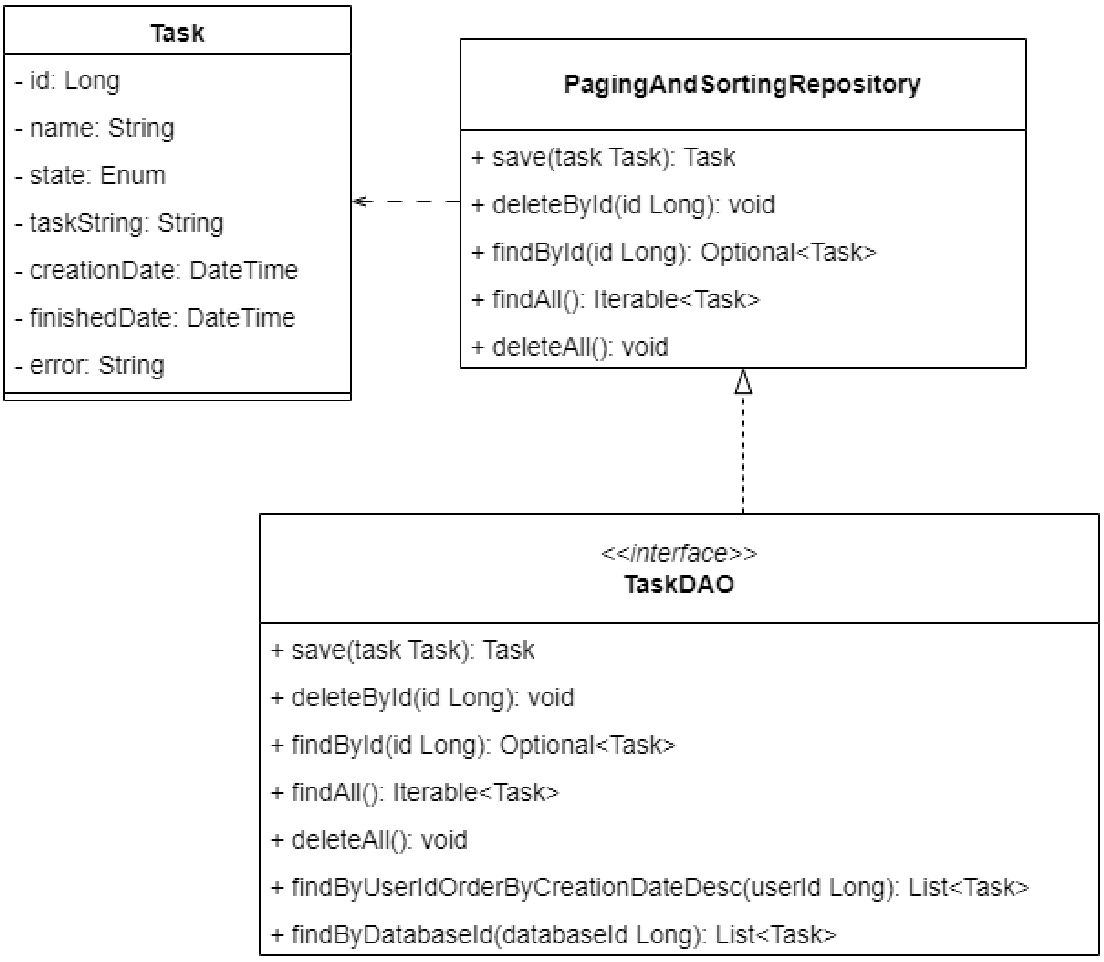

Task
- id: Long
- name: String
- state: Enum
- taskString: String
- creationDate: DateTime
- finishedDate: DateTime
- error: String
PagingAndSortingRepository
+ save(task Task): Task
+ deleteById(id Long): void
+ findById(id Long): Optional<Task>
+ findAll(): Iterable<Task>
+ deleteAll(): void
<<interface>>
TaskDAO
+ save(task Task): Task
+ deleteById(id Long): void
+ findById(id Long): Optional<Task>
+ findAll(): Iterable<Task>
+ deleteAll(): void
+ findByUserIdOrderByCreationDateDesc(userId Long): List<Task>
+ findByDatabaseId(databaseId Long): List<Task>

Figure 5.3: Task DAO

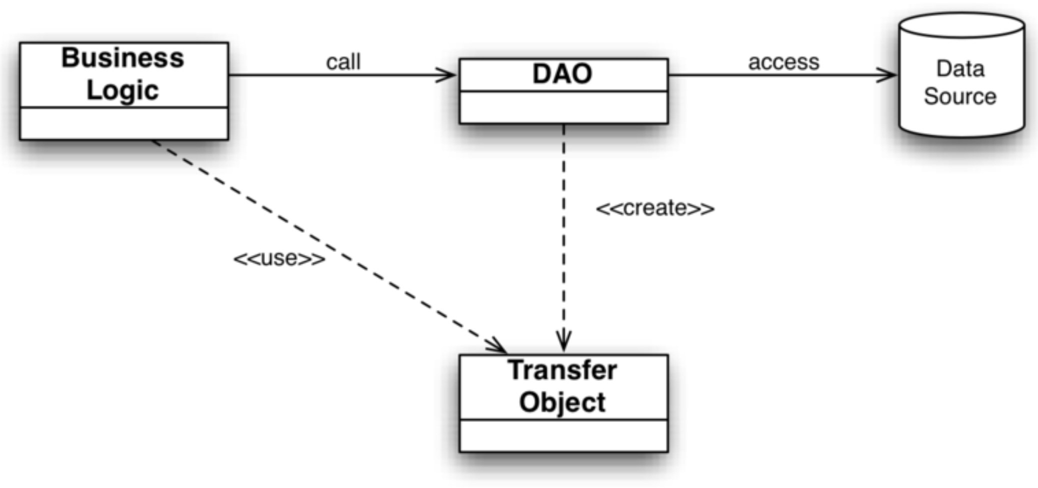


Figure 5.4: DAO pattern diagram [1]

### DTO

The Data Transfer Object (DTO) is a type of object which encapsulates the domain classes. In this project, the controllers makes use of some custom converters which perform the conversion from JavaScript Object Notation (JSON) to DTO and from DTO to JSON.

DTO attributes does not completely match with the ones of the persistent classes. A DTO will only contain the necessary attributes to be used in the client for certain views or actions, thus avoiding sending unnecessary attributes.

For the object conversion, Spring Boot uses Jackson transparently for the developer. Jackson is in charge of mapping JSON objects from the client into the DTO it expects to receive.

The first reason to use this option is to hide information to the client, like in the case of AuthenticatedUserDTO, since this way the password is kept in the database, but it is not passed to the client side as it is a critical value. Another advantage is that DTOs allow to reduce the number of remote calls bringing all the neccessary data in a unique object in a single call.

### Services

The service layer consists of the logic that performs the main functions of the application, the high-level operations. For this, the operations call the DAOs which will be in charge of data access and then the service use this data to offer much more complex and advanced functionalities.

In the project, if we look to the *es.udc.fic.tfg.backend.model.services* folder we will find the Service facade *es.udc.fic.tfg.backend.model.services.TaskService* and its implementation *es.udc.*

*fic.tfg.backend.model.services.TaskServiceImpl*. In our case, various services with their respective operations that can be accessed from the client through requests to the controllers can be found. The different services that are accessible are:

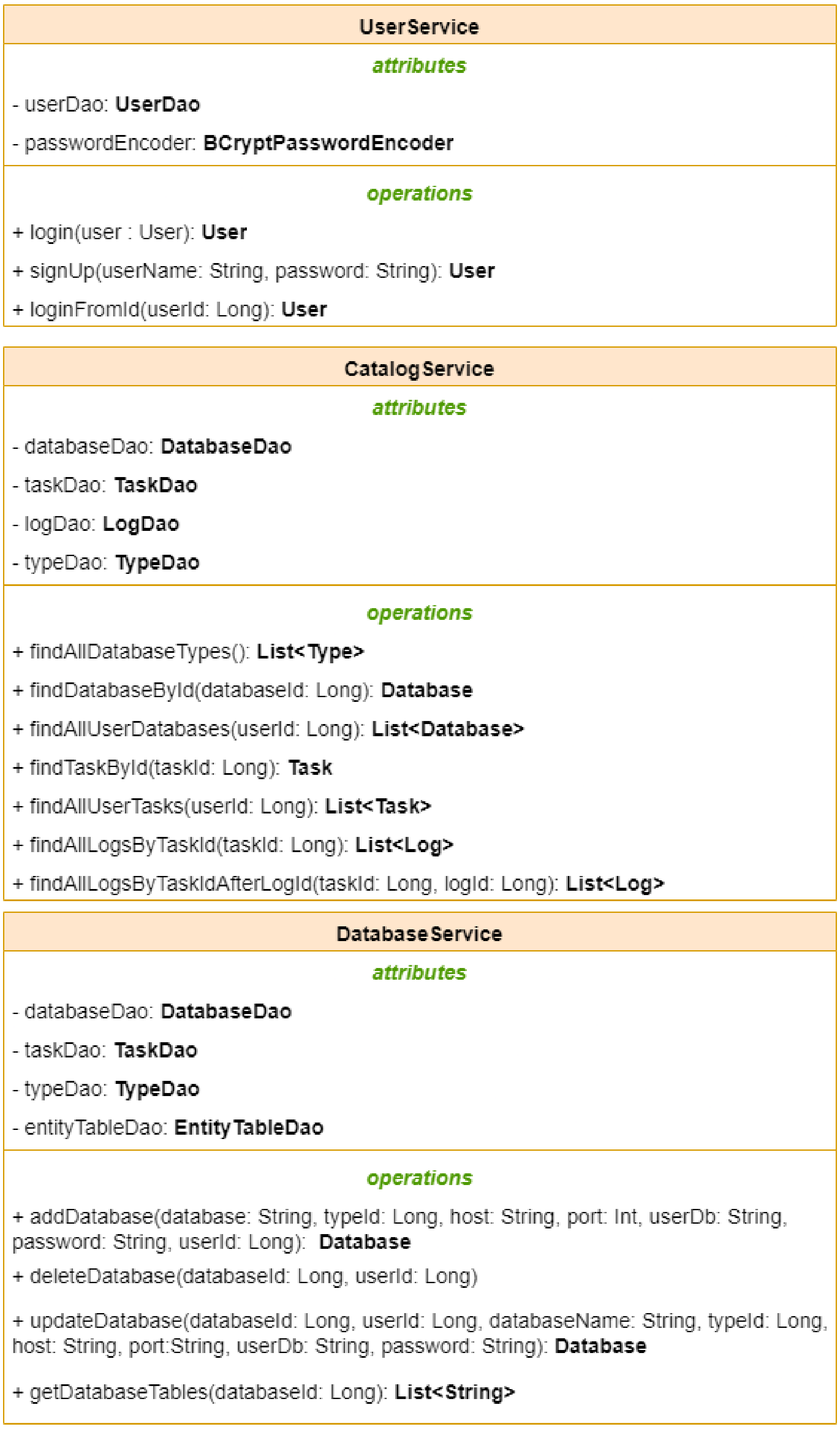

UserService
attributes
- userDao: UserDao
- passwordEncoder: BCryptPasswordEncoder
operations
+ login(user : User): User
+ signUp(userName: String, password: String): User
+ loginFromId(userId: Long): User
CatalogService
attributes
- databaseDao: DatabaseDao
- taskDao: TaskDao
- logDao: LogDao
- typeDao: TypeDao
operations
+ findAllDatabaseTypes(): List<Type>
+ findDatabaseById(databaseId: Long): Database
+ findAllUserDatabases(userId: Long): List<Database>
+ findTaskById(taskId: Long): Task
+ findAllUserTasks(userId: Long): List<Task>
+ findAllLogsByTaskId(taskId: Long): List<Log>
+ findAllLogsByTaskIdAfterLogId(taskId: Long, logId: Long): List<Log>
DatabaseService
attributes
- databaseDao: DatabaseDao
- taskDao: TaskDao
- typeDao: TypeDao
- entityTableDao: EntityTableDao
operations
+ addDatabase(database: String, typeId: Long, host: String, port: Int, userDb: String, password: String, userId: Long): Database
+ deleteDatabase(databaseId: Long, userId: Long)
+ updateDatabase(databaseId: Long, userId: Long, databaseName: String, typeId: Long, host: String, port:String, userDb: String, password: String): Database
+ getDatabaseTables(databaseId: Long): List<String>

Figure 5.5: User, Catalog and Database Service

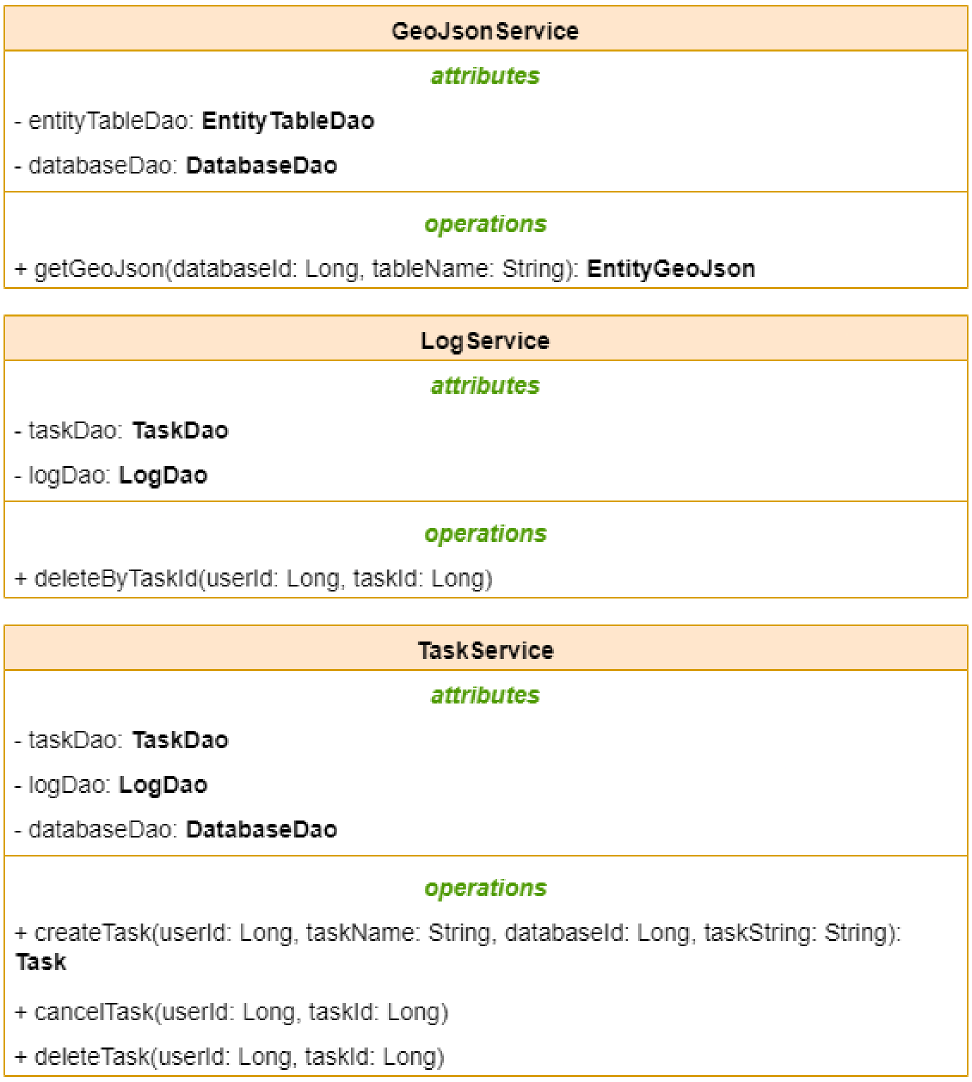

GeoJsonService
attributes
- entityTableDao: EntityTableDao
- databaseDao: DatabaseDao
operations
+ getGeoJson(databaseId: Long, tableName: String): EntityGeoJson
LogService
attributes
- taskDao: TaskDao
- logDao: LogDao
operations
+ deleteByTaskId(userId: Long, taskId: Long)
TaskService
attributes
- taskDao: TaskDao
- logDao: LogDao
- databaseDao: DatabaseDao
operations
+ createTask(userId: Long, taskName: String, databaseId: Long, taskString: String): Task
+ cancelTask(userId: Long, taskId: Long)
+ deleteTask(userId: Long, taskId: Long)

Figure 5.6: GeoJson, Log and Task Service

Now, the different services will be explained in detail.

- **UserService**: this facade contains all the operations related to the users management.
  - **login**: it allows an user to access to the application after specifying its username and password.
  - **signUp**: it registers an user in the system.
  - **loginFromId**: it allows users to log in into the application by using their associated user id.
- **CatalogService**: this facade contains the most important search operations of the application.

  - **findAllDatabaseTypes**: it returns all the database types supported by the system.
  - **findDatabaseById**: it receives a database id as input parameter and it returns the database associated with the passed id.
  - **findAllUserDatabases**: it receives a user id as input parameter and returns a list with all the databases added by the user.
  - **findTaskById**: it receives a task id as input parameter and returns the corresponding task.
  - **findAllUserTasks**: it receives a user id as input parameter and returns a list of the task executed by the user.
  - **findAllLogsByTaskId**: it receives a task id as input parameter and returns a list with all the saved logs associated with the task.
  - **findAllLogsByTaskIdAfterLogId**: it receives both a task id and log id as input parameters and returns a list of the logs associated with the task with an id higher than the one passed as parameter.

- **DatabaseService**: this facade contains the operations related to the databases.
  - **addDatabase**: it receives the connection parameters for the database as input. If the parameters are correct, it saves the database and returns it.
  - **deleteDatabase**: it receives both a database id and a user id as input parameters and deletes the corresponding database. If the database does not exist or it does not belong to the user passed as parameter, it returns an error.
  - **updateDatabase**: it receives a database id and the connection parameters as input and returns the database with the updated properties. If the system cannot connect to the database with the new parameters, the database is not updated.
  - **getDatabaseTables**: it receives a database id as input and returns a list containing the name of all the tables of the database that were used on a task executed by the user.

- **GeoJsonService**: this facade is in charge of obtaining the GeoJSONs [28] to be able to see the data imported on the map.
  - **getGeoJson**: it receives the database id and the name of the table from which the user wants to extract the GeoJSON as input and it returns an object containing the name of the table and the GeoJSON associated to it.

- **LogService**: this facade contains an operation to handle the logs associated to a task.

  - **deleteByTaskId**: it receives both a user id and a task id as input parameters and deletes all the logs associated with a task.

- **TaskService**: this facade contains all the operations related to the tasks management.
  - **createTask**: it receives the different parameters to create a task and it returns the task created. Moreover, it sends the request to the library to start the task.
  - **cancelTask**: it receives both a user id and a task id as input parameters and tries to cancel a running or waiting task.
  - **deleteTask**: it receives both a user id and a task id as input and deletes a task.

### Controllers

Controllers are placed on *es.udc. fic.tfg.backend.rest.controllers.* On this section, both the REST operations and the Uniform Resource Identifier (URI) to access the client will be shown.

| URI | OPERATION | ACTION |
|---|---|---|
| **/users/login** | POST | login |
| **/users/signUp** | POST | signUp |
| **/users/loginFromServiceToken** | POST | loginFromId |

Table 5.1: User Controller

| URI | OPERATION | ACTION |
|---|---|---|
| **/catalog/databaseTypes** | GET | findAllDatabaseTypes |
| **/catalog/user/databases** | GET | findAllUserDatabases |
| **/catalog/user/tasks** | GET | findAllUser |
| **/catalog/tasks/{taskId}** | GET | findTaskById |
| **/catalog/tasks/{taskId}/logs** | GET | findAllLogsByTaskId |
| **/catalog/tasks/{taskId}/logs/logId** | GET | findAllLogsByTaskIdAfterLogId |
| **/catalog/databases/{databaseId}** | GET | findDatabaseById |

Table 5.2: Catalog Controller

| URI | OPERATION | ACTION |
|---|---|---|
| **/databases/add** | POST | addDatabase |
| **/databases/{databaseId}/delete** | DELETE | deleteDatabase |
| **/databases/{databaseId}/update** | UPDATE | updateDatabase |
| **/databases/{databaseId}/tables** | GET | getDatabaseTables |

Table 5.3: Database Controller

| URI | OPERATION | ACTION |
|---|---|---|
| **/geoJson/database/{databaseId}/{tableName}** | GET | getGeoJson |

Table 5.4: User Controller

| URI | OPERATION | ACTION |
|---|---|---|
| **/logs/database/{taskId}/delete** | DELETE | deleteByTaskId |

Table 5.5: Log Controller

| URI | OPERATION | ACTION |
|---|---|---|
| **/tasks/add** | POST | createTask |
| **/tasks/{taskId}/cancel** | PUT | cancelTask |
| **/tasks/{taskId}/delete** | DELETE | deleteTask |

Table 5.6: Task Controller

**Facade pattern**

The Facade pattern has the feature of hiding the complexity providing a high level interface, which is in charge of performing the communication with all the neccessary subsystems. The facade is a great strategy when we require to interact with several subsystems to execute a specific operation, as it is neccessary to have technical and functional knowledge to find out which operations of each subsystem we have to execute and in which order, what may result really complex when the systems start to grow too much. That is, in summary, the facade pattern provides an unified communication interface between client and subsystems.

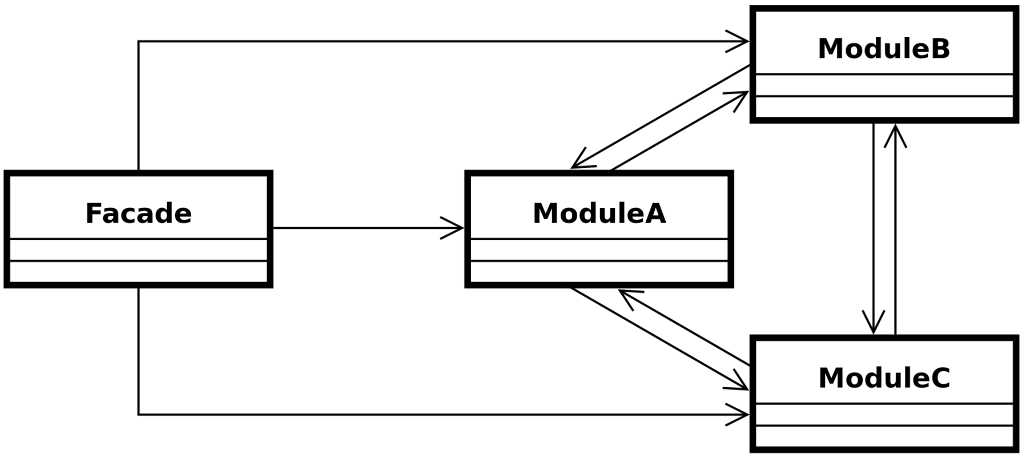


Figure 5.7: Facade pattern diagram [2]

### Other patterns

The back-end also makes use of the following design patterns:

- **Singleton**: it ensures that a class has only one instance, and it provides a global access point to it. With Spring, the singleton pattern is the default scope for a *Bean*. Spring creates a unique shared instance of the class assigned to that *Bean*, so that everytime that *Bean* is requested, the same object is injected. That is, our classes with the @Service annotation are Singleton (services).
- **Dependency Injection**: this pattern implements the Inversion of Control principle (IoC), where the control is inverted establishing dependencies between the objects. This is achieved by injecting objects in other objects using an assembler, instead of the objects itselves. Dependency injection solves the problem of making a class independent from how the object it requires are created.

### Error handling and security measures

Error handling is managed in such a way that it allows to specify both the response status and body for each error.

In terms of security, the web server implements several security measures to protect the interface from unauthorized access or malicious attacks.

- **Authorization and access control**: to control authorization to access the system endpoints, a JSON Web Token (JWT) signed with a symmetric encryption algorithm (HS256) is used. This token must be added in any request to the server that needs privileges. Access control is carried out using Spring Security, that allows to perform access control in a declarative way using, in this case, code. Furthermore, it is important to mention that the access control policy being followed is prohibitive, that is, everything which is not explicitly allowed is forbidden.
- **Encryption and authentication**: important information is encrypted to ensure confidentiality and integrity. Users' password is encrypted using a PasswordEncoder [29] from Spring Security, specifically one that uses a Bcrypt encryption algorithm. This encoder performs a one-way transformation of the password to let the password be stored securely. In order to check that the users are who their claim to be, the password they specify when log in is encrypted and compared with the one stored. Passwords specified in the connection parameters of the databases added by the users are encrypted using a substituion algorithm.

It is also important to mention that the interface uses the Bean Validation API to apply validation rules to the data received.

### 5.2.2 Front-end

For the front-end section, Vue is used. Vue helps to link our presentation layer to our business layer easily and efficiently. It is a progressive framework, that is, it is really easy to add to already existent projects; in contrast, frameworks like Angular are more oriented to start projects from scratch.

To better understand the application, we can see a package diagram (Figure 5.8) which reflects the organization of the different components of the client. In this diagram, we can observe the different packages of the application. In particular, the *components* package contains the different views that are accessible from the application.

A Vue component, as previously explained, contains the HTML, the needed logic to operate inside the component (Javascript) and CSS to format the HTML code.

### Components

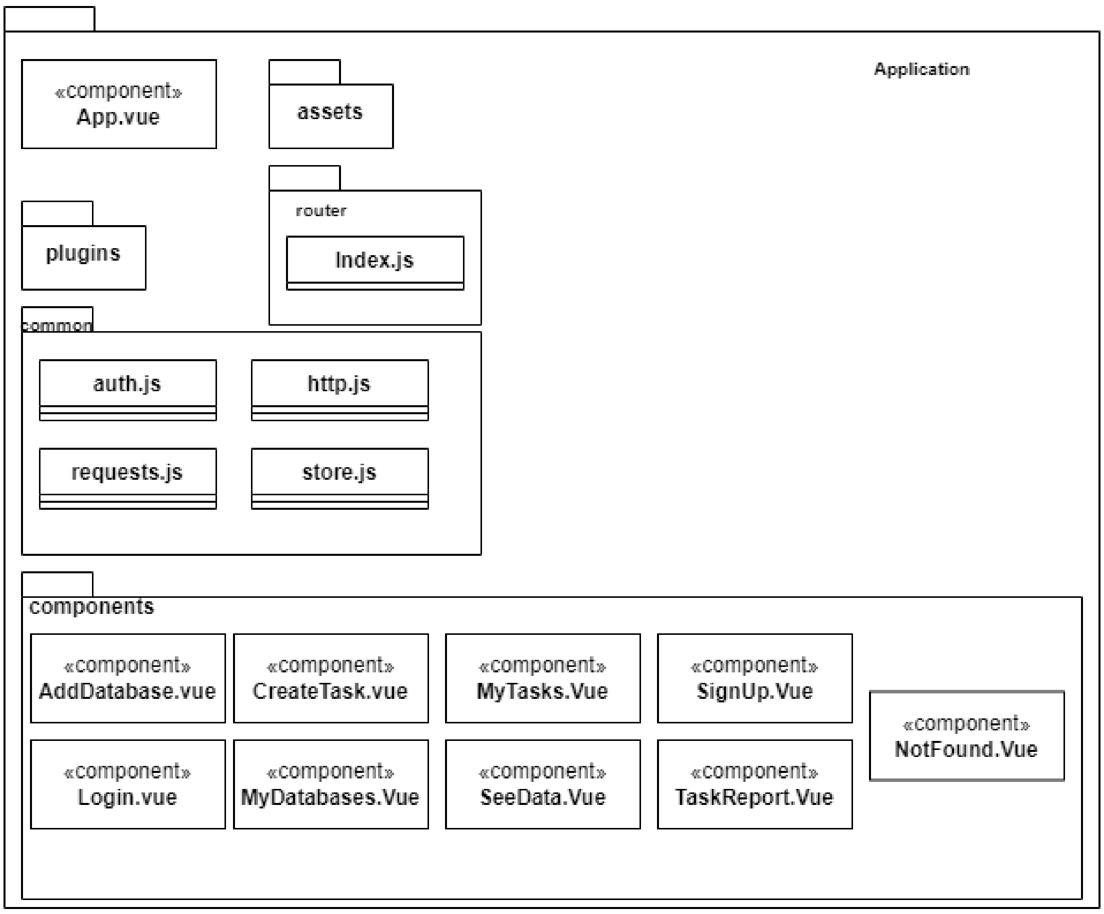


Figure 5.8: Client package diagram

### Routes

In the *index.js* file located at the *router* package there is the definition of all the routes of the Web application and the relation with the component that is loaded in that endpoint.

| NAME | ROUTE | COMPONENT | AUTHORITY |
|---|---|---|---|
| Home | / | MyTasks | logged |
| Login | /login | Login | public |
| Sign Up | /signUp | SignUp | public |
| | * | NotFound | public |
| AddDatabase | /addDatabase | AddDatabase | logged |
| MyDatabases | /myDatabases | MyDatabases | logged |
| CreateTask | /createTask | CreateTask | logged |
| RerunTask | /rerunTask/{taskId} | CreateTask | logged |
| TaskReport | /taskReport/{taskId} | TaskReport | logged |
| SeeData | /data | SeeData | logged |

Table 5.7: Web Application routes

### MVVM

Vue.js is based on the **MVVM (Model - View - ViewModel)** pattern. In this way, we separate the user interface from the business logic linking the public properties and methods with the view.

- **View**: it is the structure or template of the user interface (HTML and CSS).
- **View-Model**: it is the Javascript associated with the view. It is a view abstraction, in which it is possible to access to both public properties and methods. It represents the state of the domain data at a specific point in time.
- **Model**: the REST client.

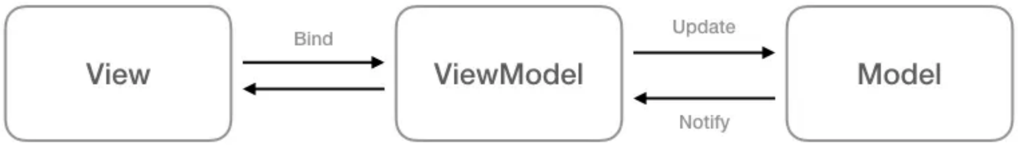


Figure 5.9: Model-View-ViewModel pattern [3]

### Callback and Promise

Vue.js makes use of the **Callback and Promise** patterns. By incorporating these patterns, the client is able to continue working until the information is available. That is, a function is attended that cannot immediately return its result and returns a promise that it will have the result in the future, which will be handled by a callback.

The client perform the request and that request does not immediately return the response, so the client keeps working. It promises the client that he/she will have the answer and by the time the result arrives, the callback is executed.

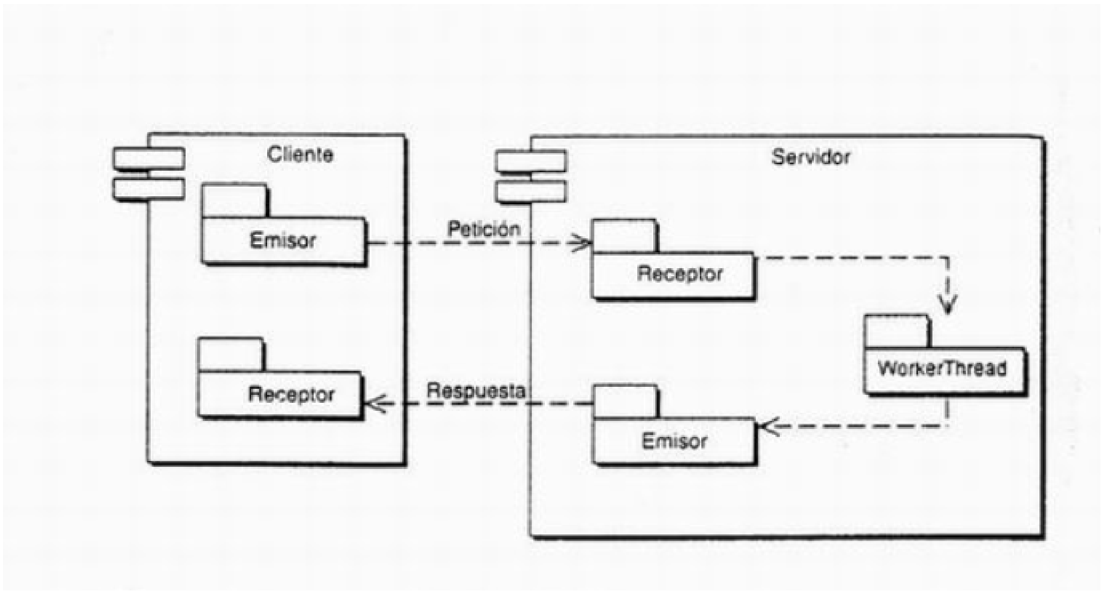


Figure 5.10: Callback pattern [4]

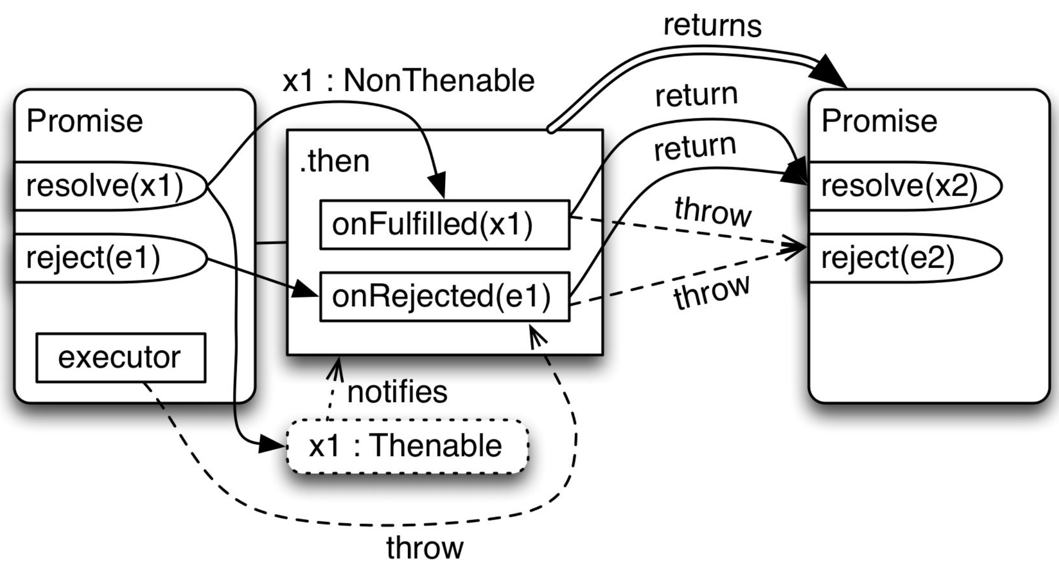


Figure 5.11: Promise pattern [5]

### 5.2.3 Parser

The parser component acts as a dependency library for the back-end. Furthermore, it can also be used by terminal executing a command where you can specify a file containing the requests you want to execute. As explained before, the component makes use of ANTLR to design the DSL and Overpass API to import the requested data from OpenStreetMap.

To better understand the application, we can see a package diagram (Figure 5.12) which reflects the organization of the different components of the parser.

### Components

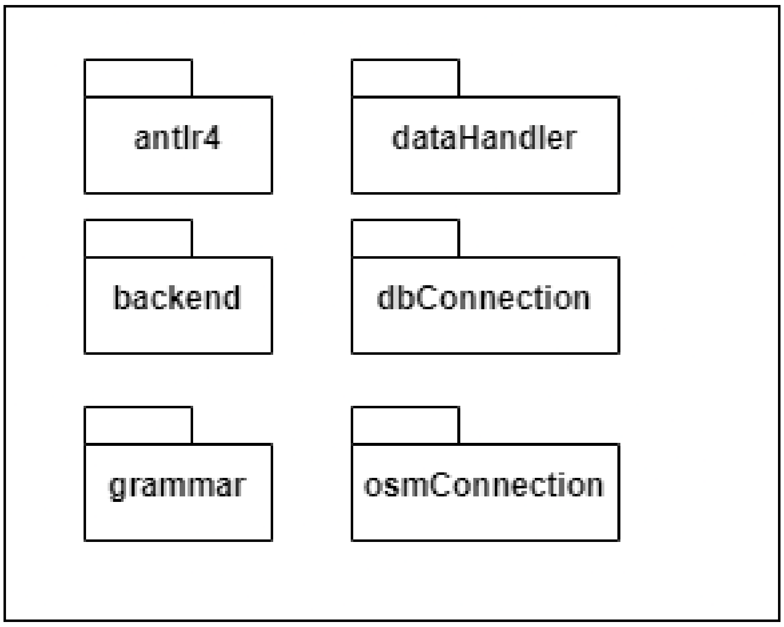


Figure 5.12: Parser package diagram

A description of each one of the packages from the above image will be given below.

- **Antlr4**: this package contains the file with the definition of the DSL. Section 5.3 explains how the DSL is designed.
- **Backend**: this package contains the classes neccessary in the communication with the back-end component. It also holds the class that the back-end component must use in order to perform the requests to the Overpass API and the class which manages the tasks parallel execution.
- **Grammar**: this package contains the classes neccessary to manage the conversion between the Parser DSL and the Overpass API query language in order to be able to perform the requests.
- **DataHandler**: this package contains the classes that perform the requests to Overpass API and they also process data they receive from OpenStreetMap.
- **DbConnection**: this package holds the class which perform the insertions on the corresponding databases after all data have been processed.
- **OsmConnection**: it holds the class which acts as intermediary between OpenStreetMap and the databases. That is, it calls the corresponding class from the DataHandler pack-

age to perform the request and once it receives the data, it sends the data to the class in charge of performing the insertions.

## 5.3 Domain Specific Language design

The developed Domain Specific Language (DSL) is a declarative language composed of instructions that allow users, in a simple and convenient way, to describe which set of OSM data they want to retrieve, and how it should be transformed to fit the schema of the database where they want to import it.

The DSL is designed making use of ANTLR. ANTLR is a parser generator for reading, processing, executing or translating structured text or binary files. It helps to build languages, tools, and frameworks.

Figure 5.13 shows part of the DSL structure and how it is defined. The rest of the DSL, which could not be included here due to space constraints, can be visualized on Appendix C.

```
grammar OSMGrammar;

parse
    : (connectStatement | statement)+
    ;

statement:
    SELECT_SYMBOl elements (
        selectStatement
    )
;

selectStatement:
    OBRA_SYMBOL attributeDefinition (COMMA_SYMBOL?
    attributeDefinition)* CBRA_SYMBOL
    fromStatement entityStatement bboxStatement
;

fromStatement:
FROM_SYMBOL OPAR_SYMBOL valueExpression CPAR_SYMBOL
;

entityStatement:
TO_SYMBOL entity
;

bboxStatement:
WHERE_SYMBOL bboxDefinition
;

connectStatement:
    CONNECT_SYMBOL TO_SYMBOL dbaseElement OF_SYMBOL typeElement
    FROM_SYMBOL portElement OF_SYMBOL hostElement WITH_SYMBOL
    userElement
    AND passwordElement

```

Figure 5.13: Domain Specific Language

Figure 5.14 shows an example of a query and the two instructions of which the DSL is composed. Using the CONNECT clause, the user must first specify the connection details for the target database. We may specify which features we want to retrieve from the OSM and how we want to import them into the database using the second command that defines the language, SELECT. This instruction contains numerous components, which are described below.

On the one hand, it specifies the filters that the characteristics to be retrieved must adhere to. This means that the language is intended to obtain data of a specific type and placed in a specified area of the map, rather than all data from everywhere in the globe from OSM. The filters relating to the requirements for the features are listed in the **FROM** clause. For instance, we can use the filter **FROM (amenity=hospital OR amenity=clinic)** if we want to retrieve *hospitals* and *clinics*. The **WHERE** clause similarly indicates the geographic filter. In the illustration, we'd be talking about a section of Santiago de Compostela. On the other hand, it is required to specify which database table should hold the data. The **TO** clause and the table name are both used for this purpose. Since we want to store both the hospitals and the clinics in that table, we indicate TO hospital in our example.

Finally, the **SELECT** clause allows us to indicate whether we want to retrieve points, lines or related elements (*node*, *way*, *relation*), and which specific set of properties of the retrieved features are mapped to columns of the specified database table. The simplest way to map property of an OSM feature to an attribute of the table is to use the operator =>, e.g. name => nombre. However, since the way data is stored in OSM is very flexible and does not necessarily follow a schema (although it does follow community conventions), the language also allows you to associate a set of properties to an attribute of the table, using the first of these properties that has some non-null value.

In this example of Figure 5.14, for each recovered item, the name, the city and the street where it is located are stored. When it comes to the name, we attempt to extract the Spanish name first using **name:es**, and if that is unsuccessful, we use the generic name. We can see that in the instance of the city, the literal value *Santiago de Compostela* would be utilized straight if there is no value for the **addr:city** attribute.

The retrieved attributes can also be subjected to functions, such as type transformations or spatial operations to compute new geometries, defining a set of operations unique to the DSL while also enabling the execution of any function made available by the database. Additionally, the geom attribute represents the geometry value of the OSM feature. The location of the hospitals is stored in the example using **ST_CENTROID(geom)** function, which determines the geometrical center from the geometry acquired from OSM.

```
CONNECT TO DBASE=osmparser OF TYPEDB=PostgreSQL FROM
PORTDB=5432 OF HOSTDB=localhost WITH USERDB=david AND
PASSWORDDB=password

SELECT node, way {
    name:es, name => name,
    addr:city, "Santiago de Compostela" => city,
    addr:street => street,
    ST_CENTROID(geom) => location
}
FROM (amenity=hospital OR amenity=clinic) TO Hospital
WHERE BBOX=(42.84866, -8.59242, 42.88672, -8.50325)
```

Figure 5.14: Query example

Chapter 6

# Implementation and tests

## 6.1 Implementation

This chapter describes those complex algorithms that are available throughout the application. In addition, the different tests that have been carried out will be shown.

### 6.1.1 OverpassAPI performance limitation

Overpass API has a big limitation in terms of performance. The API assigns a limited and low number of 'slots' per IP, that is, it automatically limits the number of requests that can be made in a period of time. As you make requests, you fill in those slots. If you end up filling all of them, the next requests will fail until some slot is free again. Therefore, it is necessary to implement an algorithm which takes care of retrying the requests. In this case, the retries for each query is limited to twenty which means that if an error occurs more than twenty times, the task will be stopped.

Figure 6.1 shows the algorithm which is in charge of retrying the requests. Note that before sending any request for the first time to the API, the system waits two seconds. This is done to reduce the risk of filling the slots. If an error still occurs, then the system starts to wait ten seconds. As said before, if the request fails twenty times then the task is stopped and an error is shown.

```java
public List<Map<String, Double>>
doWayNodesOSMConnection(ServerCallback serverCallback, Long
taskId, int times) throws InterruptedException {
    if(times == 0) TimeUnit.SECONDS.sleep(2);
    else if(times > 0) // come from error
        TimeUnit.SECONDS.sleep(10);
    WayNodesMapDataHandler handler = new
        WayNodesMapDataHandler();
    if(times == MAX_TRIES){
        throw new RuntimeException();
    }
    try{
        mapApi.queryElements(queriesList.get(0), handler);
    }catch (OsmConnectionException e){
        doWayNodesOSMConnection(serverCallback, taskId,
        ++times);
    }catch (OsmApiException e){
        doWayNodesOSMConnection(serverCallback, taskId,
        ++times);
    } catch (OsmApiReadResponseException e){
        doWayNodesOSMConnection(serverCallback, taskId,
        ++times);
    }

    return handler.getCoordinates();
}
```

Figure 6.1: Retry algorithm

### 6.1.2 Communication between the library and the server

To carry on the communication between the library and the server when needed (for example, when the status of a task needs to be updated), a common interface was developed. The Figure 6.2 shows the mentioned interface. The interface is implemented by the server and then the library uses it to 'send messages' to the server. In the example of Figure 6.4, the library is sending a message to the server in order to update the state of the task. The server provides implementation for every method of the interface. In Figure 6.5 we can see the implementation of the method which allows to change the status of a task to running.

This solution makes use of the idea of a 'callback'. In Java, a callback refers to a technique in which an object can pass a reference from a function or method to another object, thus allowing the latter to invoke the callback function when needed. In the case of this implementation, the Parser component exposes the interface with the methods shown on Figure 6.2, which are implemented by the Web server. Since the Web server implements the interface, it passes an instance of the interface to the library when it makes use of it, as we can see in Figure 6.3. This allows the library to invoke the methods defined in the interface, using the implementation provided by the Web server.

The implementation flow is the following:

- The Web server calls the method of the library shown by Figure 6.3 and passes an instance of the callback as parameter.
- At some point, the library calls a method from the interface like the one in Figure 6.4.
- Finally, the callback is performed and the code in Figure 6.5, implemented by the Web server, is executed.

```
public interface ServerCallback {

    void parseStart(Long taskId);

    void parseFinishWithError(Long taskId, String error);

    void parseFinishedOk(Long taskId);

    String getTaskState(Long taskId);

    void sendLogs(Long taskId, String log, String type);

    void checkAndAddDatabase(Long taskId, String database, int
    port, String host, String userDb, String password, String type);

    void addNewEntityTable(Long taskId, String database, int port,
    String host, String userDb, String password, String type,
                                String entityName);

}

```

Figure 6.2: Callback interface

```
    parser.parseQuery(task.getId(), task.getTaskString(),
        serverCallback);

```

Figure 6.3: Callback passing by parameter

```
serverCallback.parseStart(parserTask.getTaskId());

```

Figure 6.4: Communication between library and server

```
@Override
public void parseStart(Long taskId) {

    Optional<Task> task = taskDao.findById(taskId);

    Task finalTask = task.get();

    finalTask.setState(Task.State.RUNNING.toString());
    taskDao.save(finalTask);
}

```

Figure 6.5: Callback method implementation

### 6.1.3 Strategy pattern

The system is designed to be able to save data in any relational database. The application uses a strategy pattern to automatically select the corresponding database driver depending on the requests. This pattern also allows to control the insertion of data in each one the different databases types and the query to get the GeoJSON from any table. Furthermore, the use of this pattern makes easier to both maintain the code and add new drivers in the future if necessary.

In order to carry on this solution, a common interface was developed. The Figure 6.6 shows how it looks.

```java
public interface DbConnectionGenerator {

    boolean testConnection(String user, String password);

    String getCompleteConnectionString();

    String createQueryStringForGeoJson(String entityName,
        String user, String password);

    StringBuilder addValuesQueryString(StringBuilder
        queryString, DbElements values, int count,
           List<DbElements> attributesListMap);

    default boolean checkPointPartOfWay(Point point,
        List<DbElements> attributesListMap){
        boolean isPart = false;

        for (DbElements values : attributesListMap){
            for (String key : values.getMap().keySet()){
                if(values.getMap().get(key).getValue()
                    instanceof LineString){ //way
                        LineString lineString = (LineString)
                            values.getMap().get(key).getValue();
                        for (Coordinate coordinate :
                            lineString.getCoordinates()){
                                if (point.getX() == coordinate.getX() && point.getY() == coordinate.getY())
                                    isPart = true;
                            }
                    }
            }
        }

        return isPart;
    }
}
```

Figure 6.6: Strategy Pattern interface

### 6.1.4 Parallel execution

To be able to run several tasks in parallel the system makes use of an ExecutorService [30], a JDK API that simplifies running tasks in asynchronous mode. Generally speaking, ExecutorService automatically provides a pool of threads and an API for assigning tasks to it, facilitating thread management. Currently, the system only allows 10 parallel running tasks at most, but it could be increased easily.

The executor works as follows: when it receives a new task, checks if there is some thread available to attend the requests. If there is one available thread, it assigns the task to the free thread and the task starts its execution, but if not, it puts the task in a waiting queue. When any thread is available, the first task in the waiting queue is automatically assigned to the thread which was released.

The executor is implemented making use of the Singleton pattern so that all the users use the same instance. The Figure 6.7 shows how the singleton is implemented. The Executors [31] class provides several implementations for the ExecutorService. Some of the options allow to customize several parameters, but in this case it was decided to use the simplest one: a fixed size thread pool.

Figure 6.8 shows how the tasks are submitted to the Executor.

```
public class ExecutorSingleton {

    private final static int NUM_THREADS = 10;

    private static ExecutorService instance = null;

    private ExecutorSingleton(){}

    public static ExecutorService getInstance(){
        if(instance == null){
            instance =
                Executors.newFixedThreadPool(NUM_THREADS);
        }
        return instance;
    }

}

```

Figure 6.7: Executor Service

```
    //executor manages requests' queue and assign tasks depending
    on thread availability on the pool
    Future<?> future = ExecutorSingleton.getInstance().submit(() ->
    {
        try {
            parse(parserTask, serverCallback);
        } catch (InvalidQueryException | IOException | SQLException
            | InvalidDatabaseException |
            InvalidDbParamsException | ClassNotFoundException |
            InterruptedException |
            SyntaxErrorException | InstanceNotFoundException e) {
            throw new RuntimeException(e);
        } finally {
            cleanUpTaskList(parserTask);
            cleanUpThreadList(parserTask.getTaskId());
        }
    });
```

Figure 6.8: Executor use

## 6.2 Tests

### 6.2.1 Unit tests

Unit tests [32] consists of checking the correct performance of a specific element of the software. For these tests we will use the Spring Boot support framework and JUnit [33].

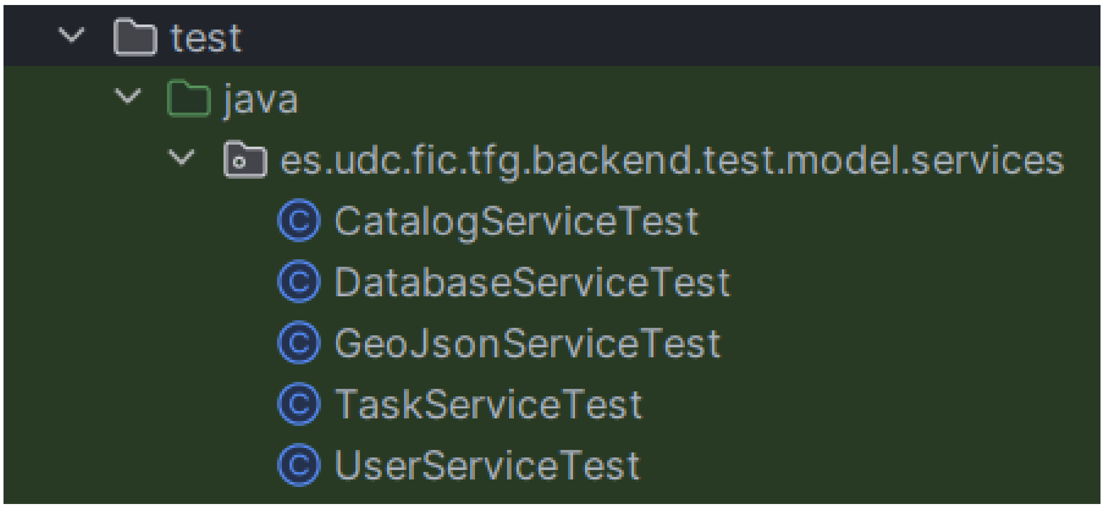
In order to be able to carry out these tests the following annotation were needed:

- *@RunWith(SpringRunner.class)*: used as a bridge between the Spring Boot test functions and JUnit.
- *@SpringBootTest*: specifies Spring Boot that it must search the main configuration class, annotated with *@SpringBootApplication*, and use it to initialize a Spring application context.

We have performed tests intended for checking the right behaviour of the functionalities exposed in all services, from basic operations like create, delete and update to more advanced functionalities, as we can see on Figure 6.10, and the test cases in which the output is not a certain value, but that an exception occurs like in Figure 6.11.

In each Test class, all possible use cases for each service have been tested, both basic CRUD operations and complex queries that each service may have.

To give an example of a test, in the database service we will test that the service updates the information of the database when desired. Therefore we insert a database in the system and then we call the method of the service that updates the data of the database. Once the service is called, it checks that the database information has changed. Finally, the database is removed from the system.



```
test
  java
    es.udc.fic.tfg.backend.test.model.services
      CatalogServiceTest
      DatabaseServiceTest
      GeoJsonServiceTest
      TaskServiceTest
      UserServiceTest
```

Figure 6.9: Unit tests package structure

```java
@Test
public void testUpdateDatabase() throws DuplicateInstanceException,
InstanceNotFoundException, PermissionException,
 NoDatabaseConnectionException {
    User user = createUser(USERNAME);
    List<Type> types = catalogService.findAllDatabaseTypes();
    Database database = databaseService.addDatabase(DATABASE,
    types.get(1).getId(), HOST, PORT, USER, PASSWORD,
        user.getId());
    database = databaseService.updateDatabase(database.getId(),
        user.getId(), DATABASE, types.get(1).getId(),
        HOST, PORT, "postgres", "postgres");
    assertEquals(database.getDatabase(), DATABASE);
    assertEquals(database.getType(), types.get(1));
    assertEquals(database.getHost(), HOST);
    assertEquals(database.getPort(), PORT);
    assertEquals(database.getUserDb(), "postgres");
    assertEquals(PasswordCipher.decode(database.getPassword()),
        "postgres");
}
```

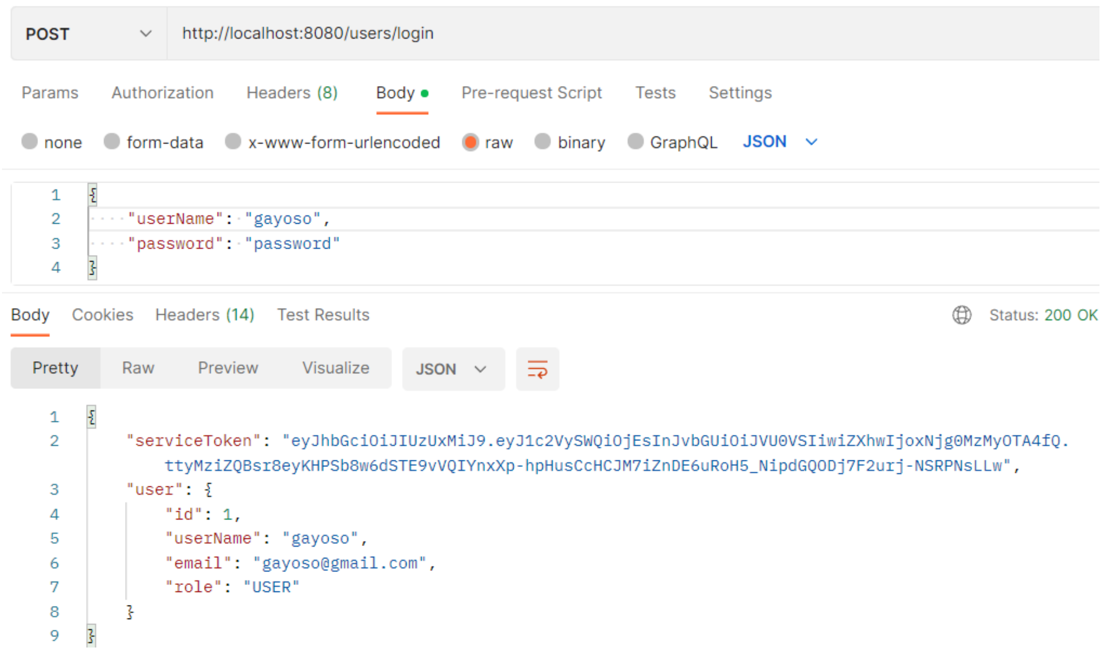
Figure 6.10: Unit test - Structure

```
    @Test
    public void testUpdateNonExistentDatabase() throws
    DuplicateInstanceException {

        User user = createUser(USERNAME);

        List<Type> types = catalogService.findAllDatabaseTypes();

        assertThrows(InstanceNotFoundException.class, () ->
            databaseService.updateDatabase(NON_EXISTENT_ID,
                user.getId(), DATABASE, types.get(1).getId(), HOST,
                    PORT, "postgres", "pass"));
    }

    @Test
    public void testUpdateWithNonExistentUser() throws
    DuplicateInstanceException, InstanceNotFoundException,
    NoDatabaseConnectionException {

        User user = createUser(USERNAME);

        List<Type> types = catalogService.findAllDatabaseTypes();

        Database database = databaseService.addDatabase(DATABASE,
            types.get(1).getId(), HOST, PORT, USER, PASSWORD,
                user.getId());

        assertThrows(InstanceNotFoundException.class, () ->
            databaseService.updateDatabase(database.getId(),
                NON_EXISTENT_ID, DATABASE, types.get(1).getId(),
                HOST, PORT, "postgres", "pass"));
    }

```

Figure 6.11: Exception test

### 6.2.2 REST tests

In order to check that all REST requests work as expected PostMan, a tool which allows us to perform HTTP requests to any API to check the right performance of our developments, has been used. First, a POST request will be made to authenticate us and receive back the *token* needed to be able to make any request.



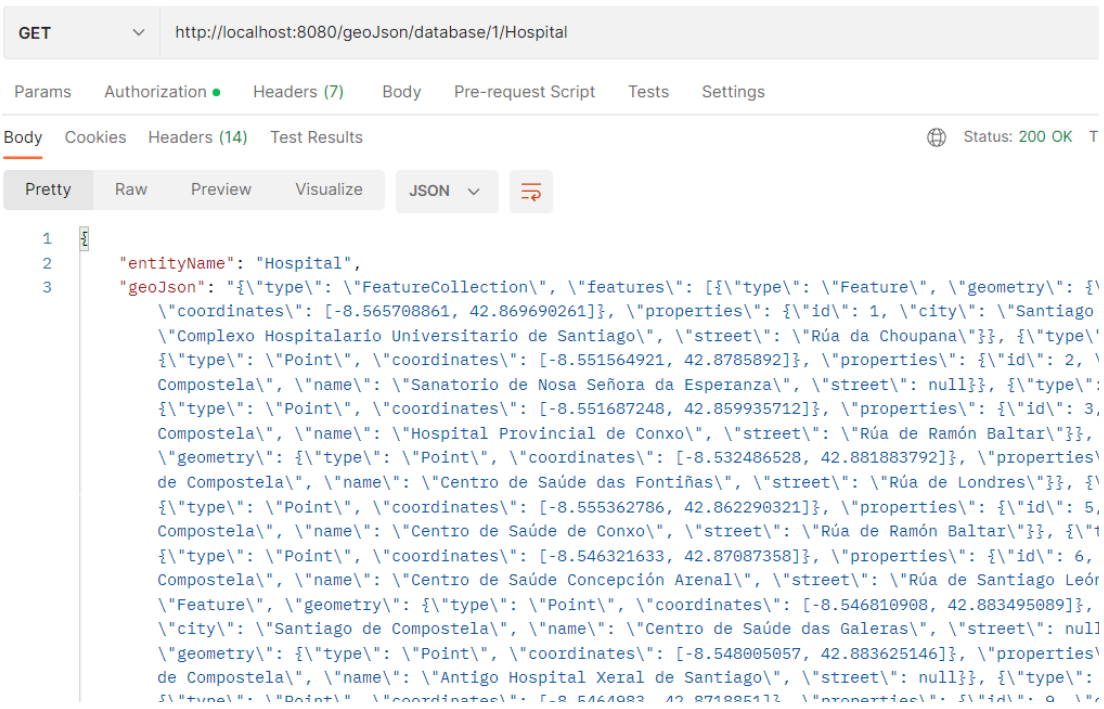
POST http://localhost:8080/users/login

```
{
    "userName": "gayoso",
    "password": "password"
}
```

Status: 200 OK

```
{
    "serviceToken": "eyJhbGciOiJIUzUxMiJ9.eyJ1c2VySWQiOjEsInJvbGUiOiJVU0VSIiwiZXhwIjoxNjg0MzMyOTA4fQ.
        ttyMziZQBsr8eyKHPSb8w6dSTE9vVQIYnxXp-hpHusCcHCJM7iZnDE6uRoH5_NipdGQODj7F2urj-NSRPNsLLw",
    "user": {
        "id": 1,
        "userName": "gayoso",
        "email": "gayoso@gmail.com",
        "role": "USER"
    }
}
```

Figure 6.12: POST authentication request



GET http://localhost:8080/geoJson/database/1/Hospital

Status: 200 OK

```
{
    "entityName": "Hospital",
    "geoJson": "{\"type\": \"FeatureCollection\", \"features\": [{\"type\": \"Feature\", \"geometry\": {\
        \"coordinates\": [-8.565708861, 42.869690261]}, \"properties\": {\"id\": 1, \"city\": \"Santiago
        \"Complexo Hospitalario Universitario de Santiago\", \"street\": \"Rúa da Choupana\"}}, {\"type\'
        {\"type\": \"Point\", \"coordinates\": [-8.551564921, 42.8785892]}, \"properties\": {\"id\": 2, \
        Compostela\", \"name\": \"Sanatorio de Nosa Señora da Esperanza\", \"street\": null}}, {\"type\":
        {\"type\": \"Point\", \"coordinates\": [-8.551687248, 42.859935712]}, \"properties\": {\"id\": 3,
        Compostela\", \"name\": \"Hospital Provincial de Conxo\", \"street\": \"Rúa de Ramón Baltar\"}},
        \"geometry\": {\"type\": \"Point\", \"coordinates\": [-8.532486528, 42.881883792]}, \"properties\
        de Compostela\", \"name\": \"Centro de Saúde das Fontiñas\", \"street\": \"Rúa de Londres\"}}, {\
        {\"type\": \"Point\", \"coordinates\": [-8.555362786, 42.862290321]}, \"properties\": {\"id\": 5,
        Compostela\", \"name\": \"Centro de Saúde de Conxo\", \"street\": \"Rúa de Ramón Baltar\"}}, {\"t
        {\"type\": \"Point\", \"coordinates\": [-8.546321633, 42.87087358]}, \"properties\": {\"id\": 6,
        Compostela\", \"name\": \"Centro de Saúde Concepción Arenal\", \"street\": \"Rúa de Santiago Leór
        \"Feature\", \"geometry\": {\"type\": \"Point\", \"coordinates\": [-8.546810908, 42.883495089]},
        \"city\": \"Santiago de Compostela\", \"name\": \"Centro de Saúde das Galeras\", \"street\": null
        \"geometry\": {\"type\": \"Point\", \"coordinates\": [-8.548005057, 42.883625146]}, \"properties\
        de Compostela\", \"name\": \"Antigo Hospital Xeral de Santiago\", \"street\": null}}, {\"type\":
        {\"type\": \"Point\", \"coordinates\": [-8.5464983, 42.871885]}, \"properties\": {\"id\": 9, \"c
```

Figure 6.13: GET request

### 6.2.3 Integration and acceptance tests

These tests have been carried out through the execution of the different use cases. All system operations were reviewed to check if they worked properly: creating users, tasks and databases, testing parallel execution of multiple tasks, visualizing huge amounts of data on the map viewer, in summary, check that the application as a whole worked as expected.

### 6.2.4 Library tests

In the case of the parser component, different tests were developed, e.g checking that the requests were done correctly, testing that the conversion between the DSL and Overpass API syntax was correct and verifying that the library obtained and inserted the corresponding data correctly.

Chapter 7

# Developed solution

This chapter shows the main features of the developed system. A guided tour of the main functionalities of the application will be made, accompanied by screenshots of the final application.

## 7.1 Log in and sign up

Here we can see both the log in and sign up page of the application. As we can see, both pages allow the user to navigate to each other at any time.

It is important to mention that in both cases several checks are made. For example, users cannot register in the application with an already in use username or email.

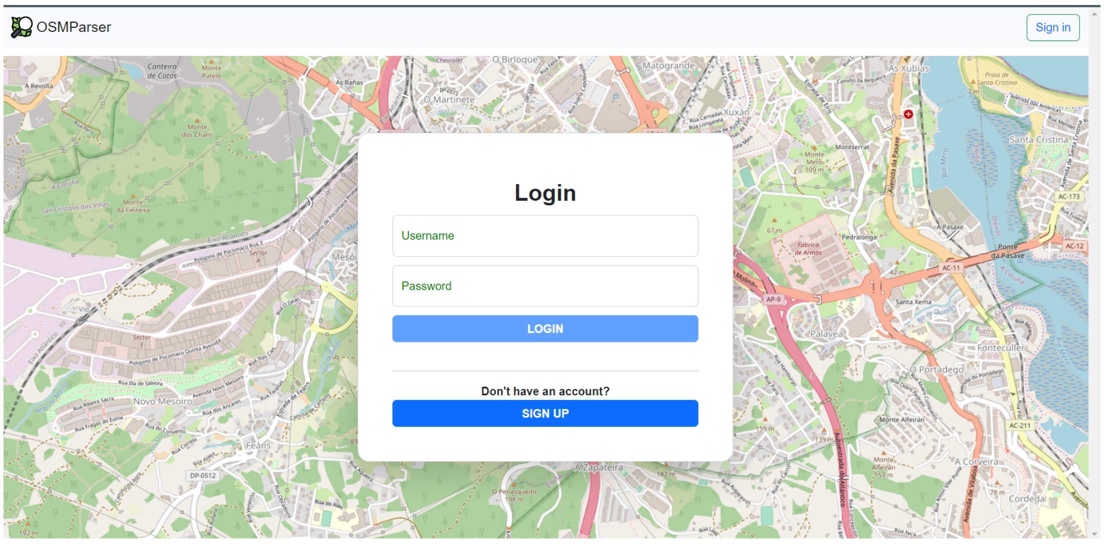


Figure 7.1: Login page

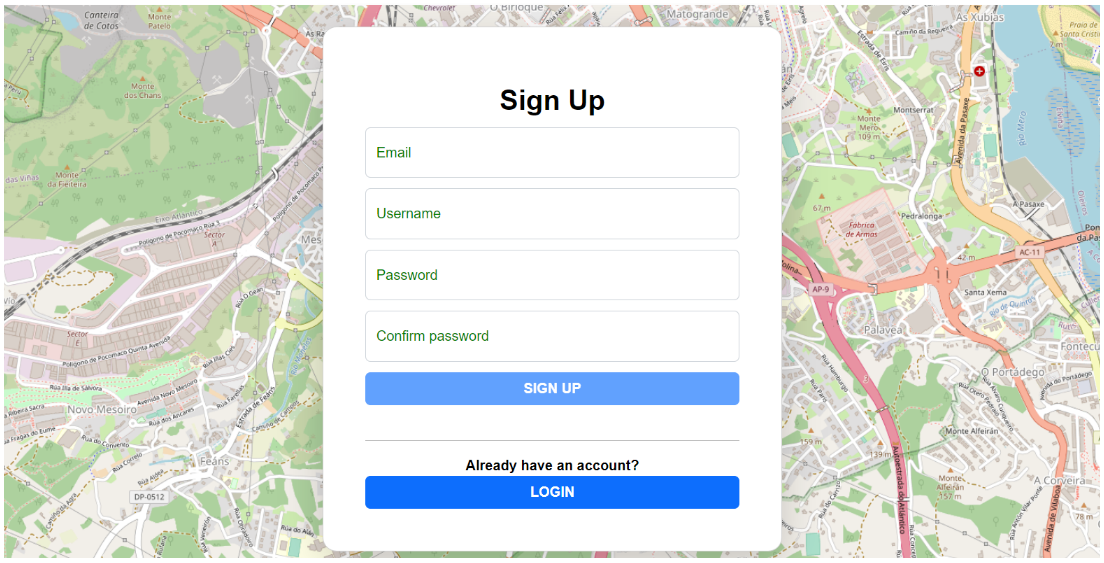
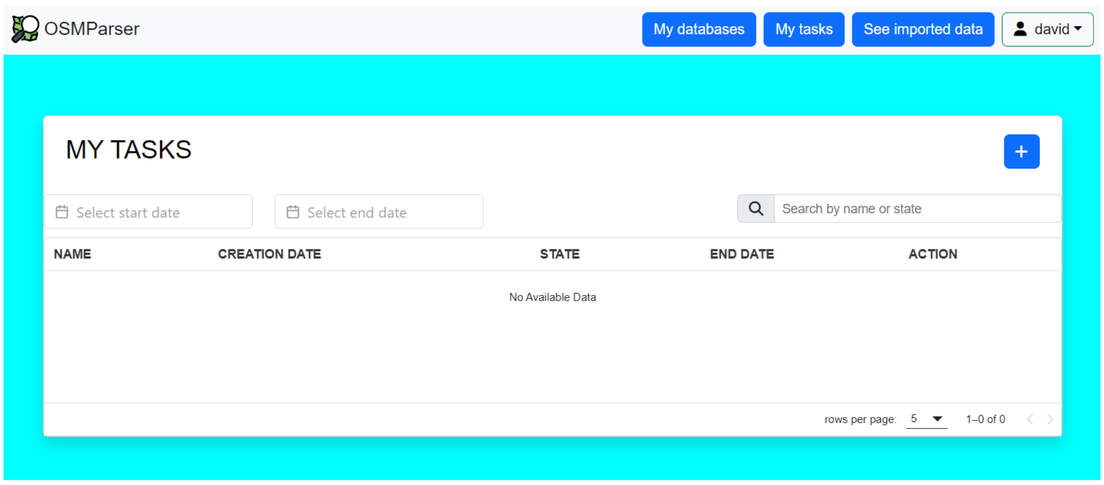


Figure 7.2: Sign up page

## 7.2 Tasks historic

This is the main page of the web application. It is interesting to mention that once the users log in into the web, they can access this page at any time using the top button in the navigation bar. Users are able to see all their previously executed tasks or their currently running ones. Moreover, they can filter them to find any task faster. Using the '+' button they will be redirected to another page where they can start a new task.

Looking at Figure 7.4, we can appreciate that users can take a look to important data of their tasks. Furthermore, they are able to perform several actions on their tasks. For example, users can cancel a running or waiting tasks, rerun them again if needed, delete them or see the execution report in real time.

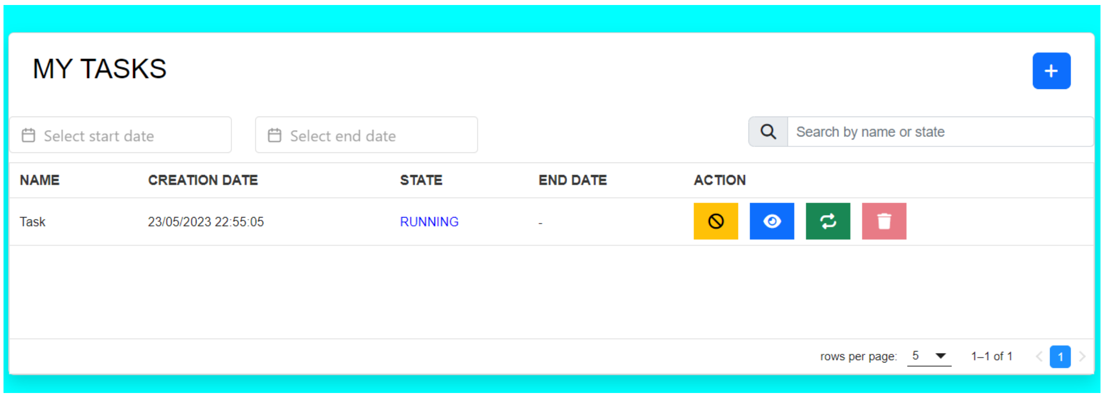


| NAME | CREATION DATE | STATE | END DATE | ACTION |
|---|---|---|---|---|

No Available Data

rows per page: 5 1–0 of 0

Figure 7.3: My tasks



| NAME | CREATION DATE | STATE | END DATE | ACTION |
|---|---|---|---|---|
| Task | 23/05/2023 22:55:05 | RUNNING | - | |

rows per page: 5 1–1 of 1 1

Figure 7.4: Task running

## 7.3 Create tasks

This functionality allows to create new tasks with a given name and using a specific database. It is possible to select the database from the dropdown (which contains the user previously added databases) or directly include the connection parameters in the query itself. In any case, several databases could be included in the same query.

In case the users want to rerun a specific task, they will be redirect to this page but with the corresponding form elements covered with the task specific data.

From this view, as shown by Figure 7.6, users are able to rerun previously executed tasks.

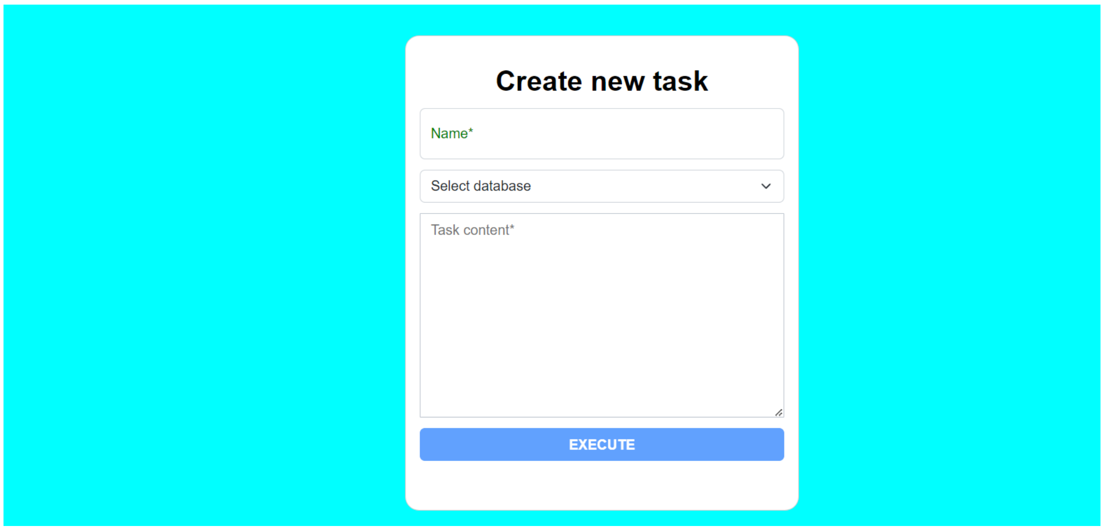
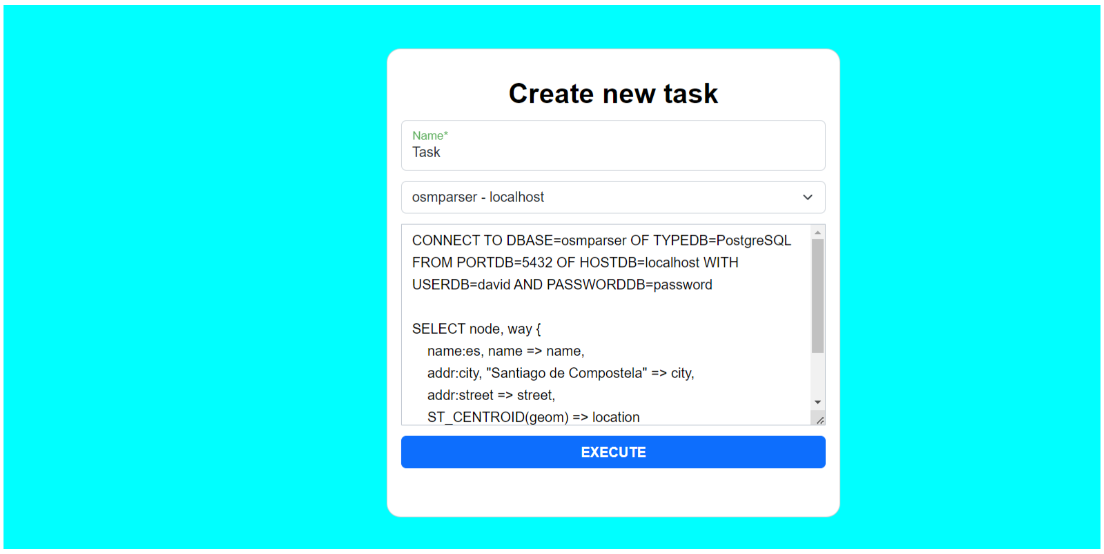


Figure 7.5: Create task



Name*
Task

osmparser - localhost

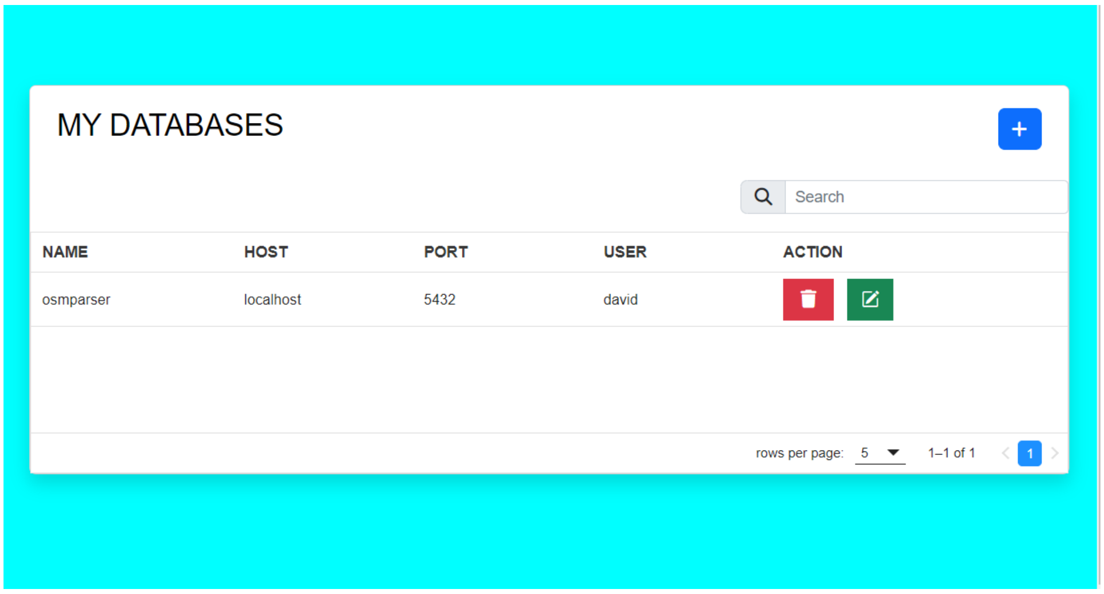
```
CONNECT TO DBASE=osmparser OF TYPEDB=PostgreSQL
FROM PORTDB=5432 OF HOSTDB=localhost WITH
USERDB=david AND PASSWORDDB=password

SELECT node, way {
    name:es, name => name,
    addr:city, "Santiago de Compostela" => city,
    addr:street => street,
    ST_CENTROID(geom) => location
```

Figure 7.6: Rerun task

## 7.4 My databases

The page shown by Figure 7.7 describes the functionality which allows users to observe their databases added on the application. Clicking the '+' button redirect users to the page where they are able to enter the connection parameters to add a new database to the application. Like in the case of the Figure 7.3 section, users can navigate to this page whenever they want using the top button of the navigation bar.

It is possible to both delete and update a database at any time using the buttons located inside the table at the 'action' column.



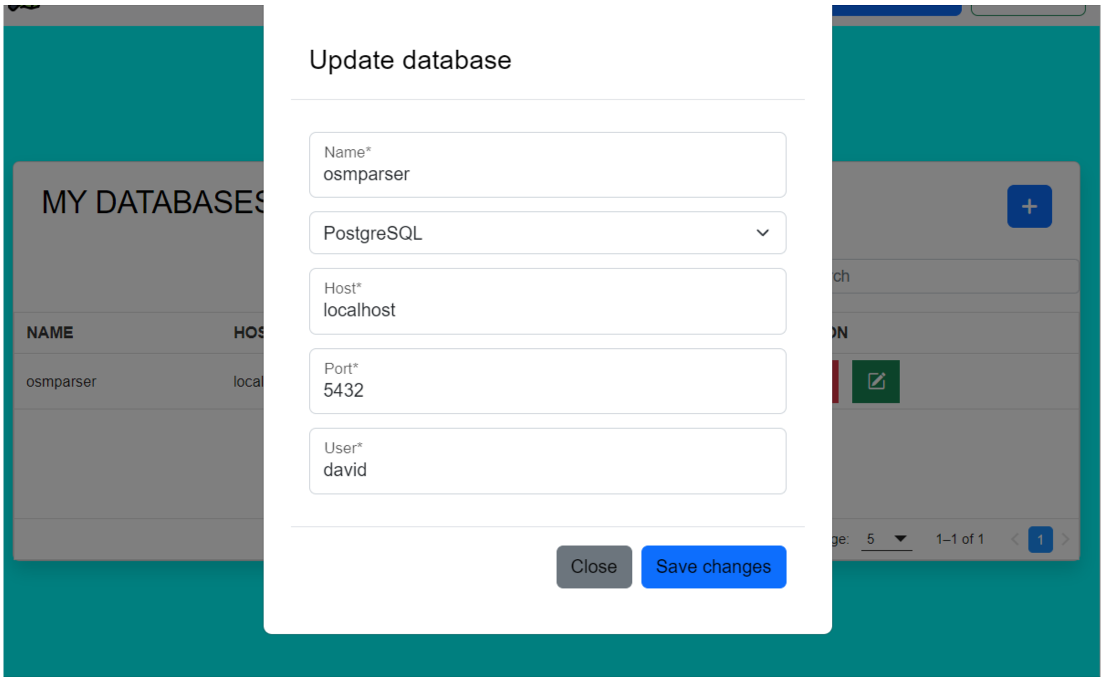
| NAME | HOST | PORT | USER | ACTION |
|---|---|---|---|---|
| osmparser | localhost | 5432 | david | |

Figure 7.7: User databases

## 7.5 Update database

This functionality is only accessible from the Figure 7.7 view. It allows to update the connection parameters of any database.

It is essential to mention that once the user press the button to update the database, the system checks if it has connection to the database using the corresponding parameters. If the application cannot connect to the database, it alerts the user with an error notification. This check prevents the user from having problem when using that database on a task.


Update database

Name*
osmparser

PostgreSQL

Host*
localhost

Port*
5432

User*
david

Figure 7.8: Update database connection parameters

## 7.6 Add database

This page shows a form where the user can enter the corresponding connection parameters to add a new database to the application.

It is important to note that like in the case of updating, the system checks the connection with the database using the entered parameters. If the check is successfull, the user is redirected to Figure 7.7 and a notification appears showing that the database was successfully added.

Moreover, some inputs checks are performed. For example, the application checks that the entered port is a number that corresponds to one of the computer's port.

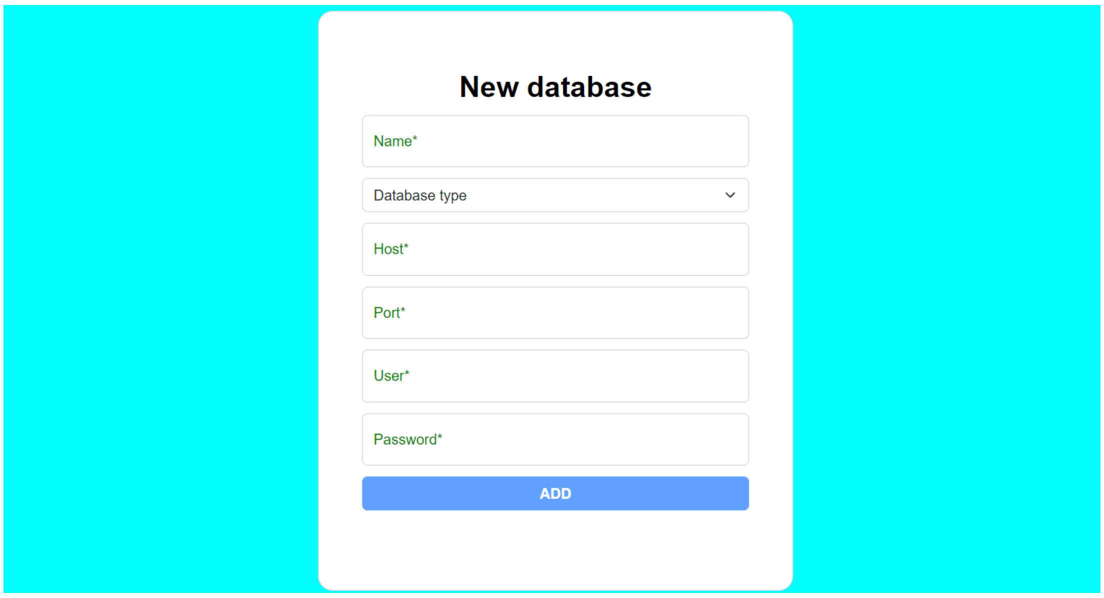
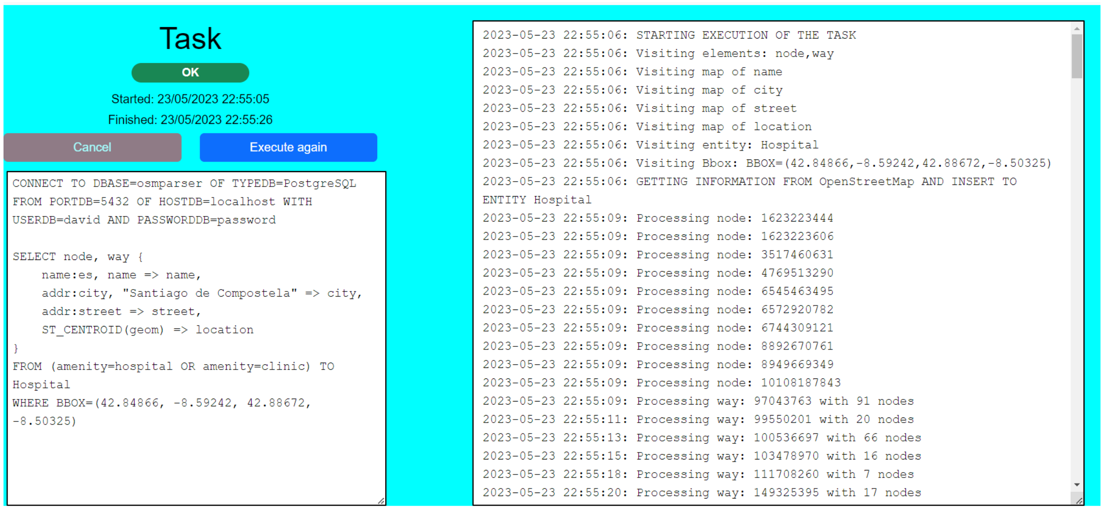


Figure 7.9: Add database

## 7.7 Task execution report

This page shows several essential data about a specific task, such as the start and end times, the written query and real times logs of what the task is performing. Moreover, if the task failures for some reason, an error notificacion will show up and the message box will turn red.

As we can see in Figure 7.10, it is possible to cancel or rerun the task also from this view.



Started: 23/05/2023 22:55:05
Finished: 23/05/2023 22:55:26

```
CONNECT TO DBASE=osmparser OF TYPEDB=PostgreSQL
FROM PORTDB=5432 OF HOSTDB=localhost WITH
USERDB=david AND PASSWORDDB=password

SELECT node, way {
    name:es, name => name,
    addr:city, "Santiago de Compostela" => city,
    addr:street => street,
    ST_CENTROID(geom) => location
}
FROM (amenity=hospital OR amenity=clinic) TO
Hospital
WHERE BBOX=(42.84866, -8.59242, 42.88672,
-8.50325)
```

```
2023-05-23 22:55:06: STARTING EXECUTION OF THE TASK
2023-05-23 22:55:06: Visiting elements: node,way
2023-05-23 22:55:06: Visiting map of name
2023-05-23 22:55:06: Visiting map of city
2023-05-23 22:55:06: Visiting map of street
2023-05-23 22:55:06: Visiting map of location
2023-05-23 22:55:06: Visiting entity: Hospital
2023-05-23 22:55:06: Visiting Bbox: BBOX=(42.84866,-8.59242,42.88672,-8.50325)
2023-05-23 22:55:06: GETTING INFORMATION FROM OpenStreetMap AND INSERT TO
ENTITY Hospital
2023-05-23 22:55:09: Processing node: 1623223444
2023-05-23 22:55:09: Processing node: 1623223606
2023-05-23 22:55:09: Processing node: 3517460631
2023-05-23 22:55:09: Processing node: 4769513290
2023-05-23 22:55:09: Processing node: 6545463495
2023-05-23 22:55:09: Processing node: 6572920782
2023-05-23 22:55:09: Processing node: 6744309121
2023-05-23 22:55:09: Processing node: 8892670761
2023-05-23 22:55:09: Processing node: 8949669349
2023-05-23 22:55:09: Processing node: 10108187843
2023-05-23 22:55:09: Processing way: 97043763 with 91 nodes
2023-05-23 22:55:11: Processing way: 99550201 with 20 nodes
2023-05-23 22:55:13: Processing way: 100536697 with 66 nodes
2023-05-23 22:55:15: Processing way: 103478970 with 16 nodes
2023-05-23 22:55:18: Processing way: 111708260 with 7 nodes
2023-05-23 22:55:20: Processing way: 149325395 with 17 nodes
```

Figure 7.10: Task execution report

## 7.8 Data visualization

This page is accessible everywhere using the top button from the navigation bar. It shows a map where the user can select any database and it automatically retrieves on the right corner legend all the tables used by the user on that database.

Once the user clicks on one checkbox, the system makes an asynchronous call to get the geographic elements associated to the clicked table. For example, in the case showed in Figure 7.11 when the user clicked on the checkbox an asynchronous call to get all the hospitals was made. With this method, the system avoid to download huge amounts of data all at once.

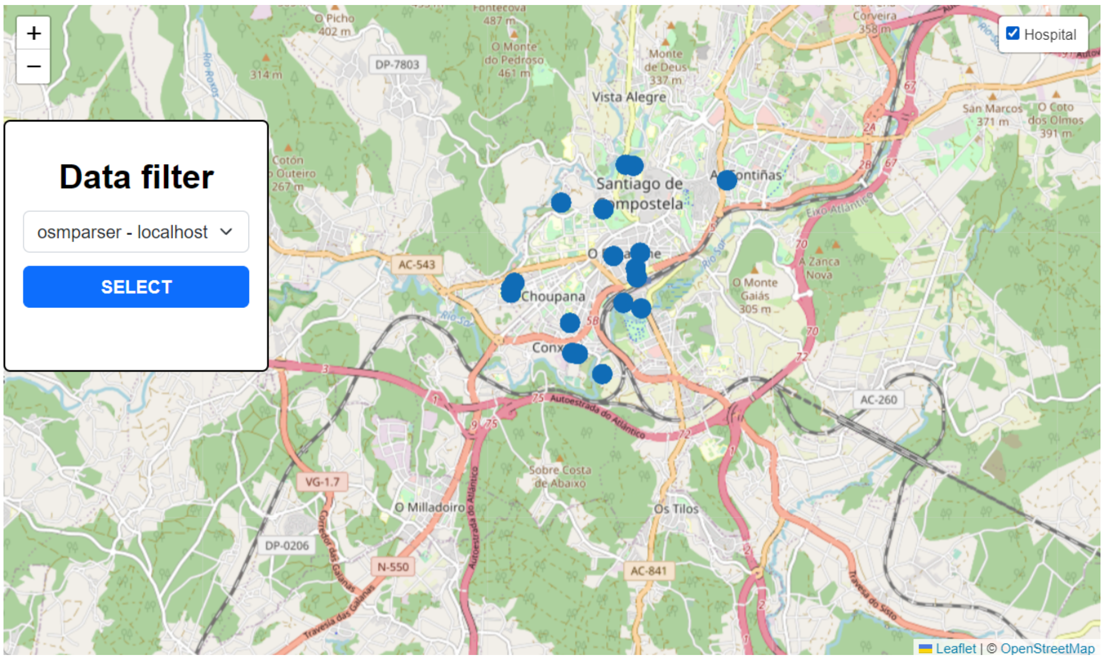


Figure 7.11: Database imported data

Chapter 8

# Conclusions and future work

## 8.1 Conclusions

At the end of the project, it can be affirmed that all the objectives that had been proposed at the beginning of the project have been satisfactorily achieved, creating an application with the following features:

- Allows successful user management.
- Allows to display a list user executed tasks and to perform different type of actions on them.
- Allows to save different types of databases and to display a list of previously saved database.
- Allows to create and execute parallel tasks to import data from OpenStreetMap.
- Allows to visualize the imported data on a map viewer.
- The entire data extraction and import process is automated

With this project, experience has been gained in the use of Spring, JPA and Jackson inside Java. HTML and CSS knowledge has been improved and Javascript knowledge about events has increased. Moreover, new skills have been acquired:

- A new framework has been learned (Vue.js).
- Grammar definition with ANTLR.
- Asynchronous programming and use of threads.
- Knowledge in spatial information systems.

- Use of Leaflet to implement web map viewers.
- Design and integration of a Java library in another project.

## 8.2 Future work

This project complies with everything proposed at the beginning, but it has many more functionalities that could be added to improve the application. Some of these possible improvements could be:

### 8.2.1 Periodic tasks

The system allows to execute parallel task, but it would be interesting to be able to schedule tasks to be executed at a specific time automatically or to be executed periodically during a period of time.

### 8.2.2 Reverse transformation

Exploring the reverse transformation (that is, take the geographic data we have in a database of our own and publish it automatically in OSM) could also be interesting.

# Appendices

Appendix A

# Installation steps

## A.1 Required software

In order to be able to deploy the application, previously you will need:

- Apache Maven 3.8.6.
- Node.js 18.12.1.
- PostgreSQL database management system.
- Java virtual machine.

## A.2 Deploy steps

The following are the different required steps to deploy the application:

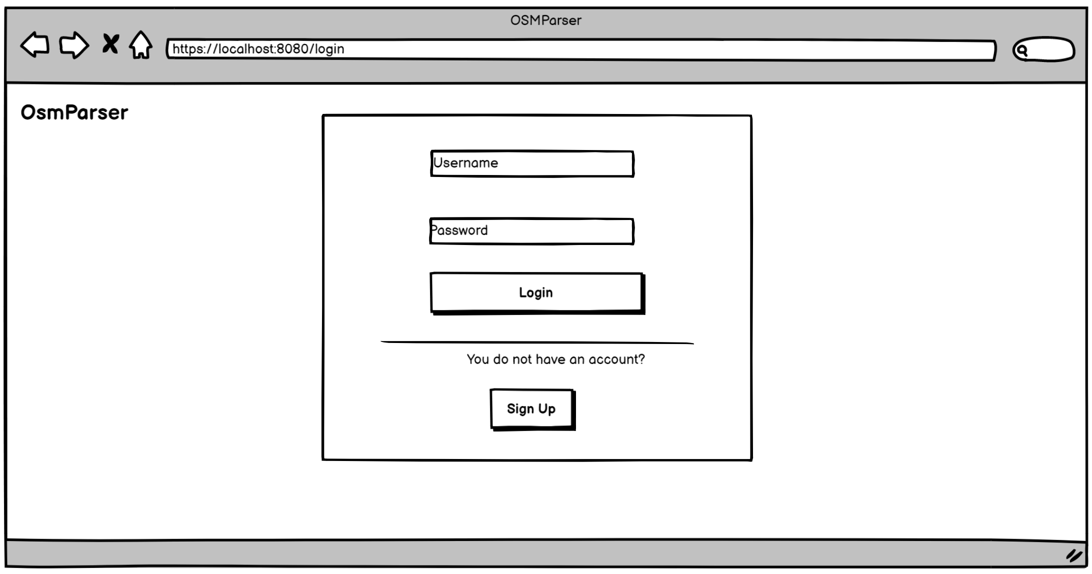
- **Databases**:
  - Create the databases where you want to store the data.
- **Library**
  - Inside osmparser project, execute: mvn compile install
  - **Important note**: if you want to make use of the library directly, you can use it by executing the following command after the previous one: mvn exec:java -Dexec.args='pathtofile' where 'pathtofile' is the path of a file with contains different tasks to execute.
- **Server**:
  - Inside server project, execute: mvn sql:execute install spring-boot:run

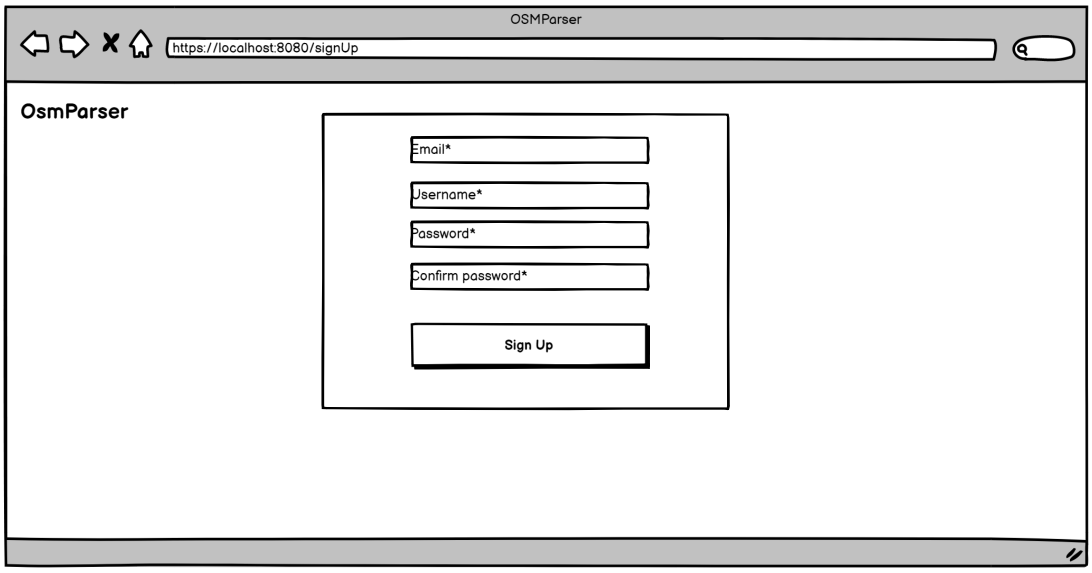
- **Client**:
  - Inside client project, execute:
    - npm install
    - npm run dev

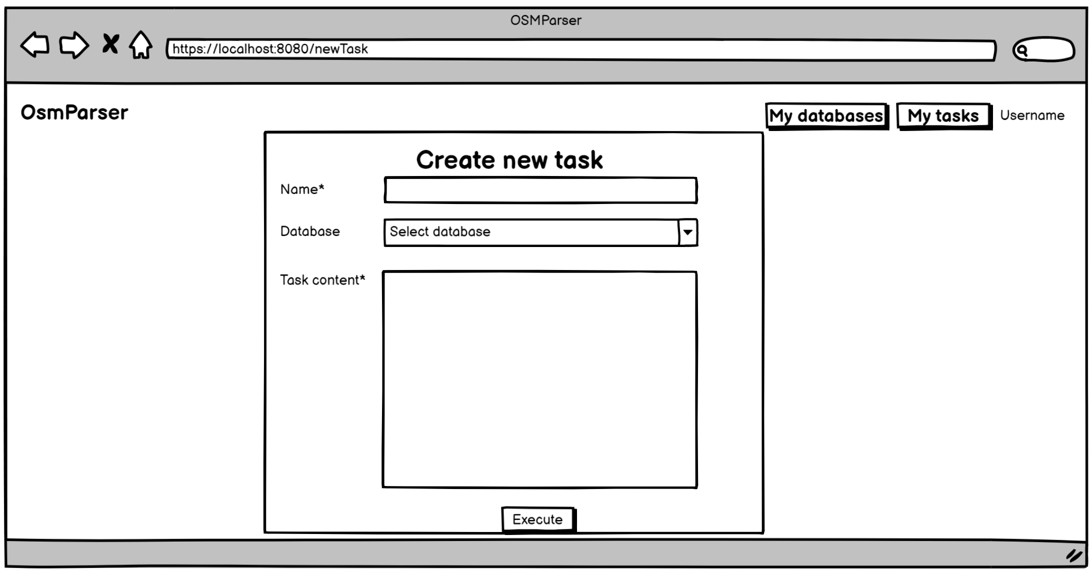
Once the client is running, we launch http://localhost:3000/ in a browser which will open the main screen of the application.

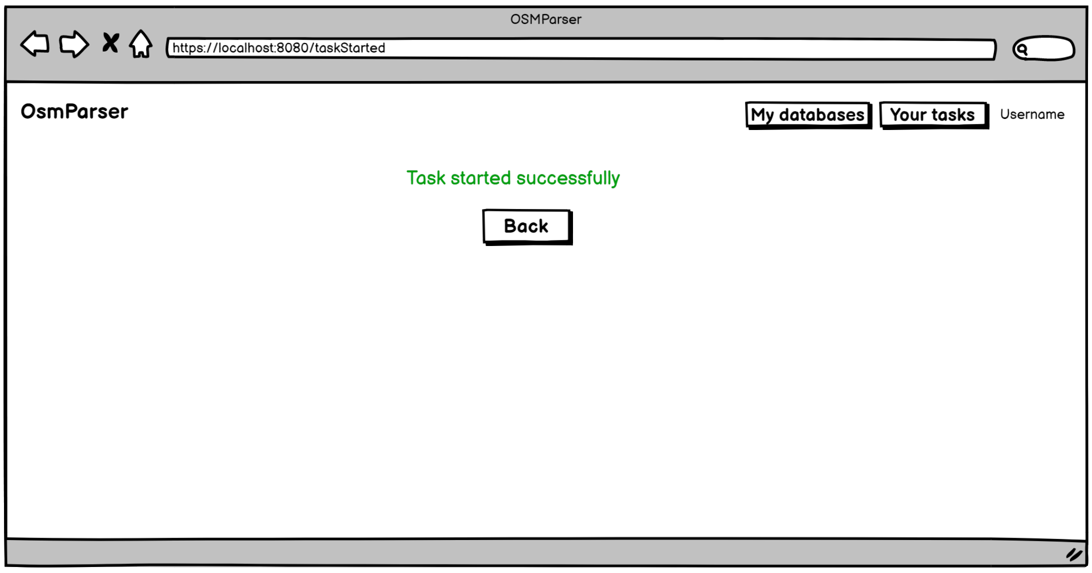
# Appendix B
# Mockups

Next, all the mockups made in the Initial Phase, explained in Section 3.1, will be incorporated. It is worth mentioning that these screens are a previous idea of what we wanted to do in order to plan the project work and have a base to start from so there will be changes with respect to the screens of the final application.

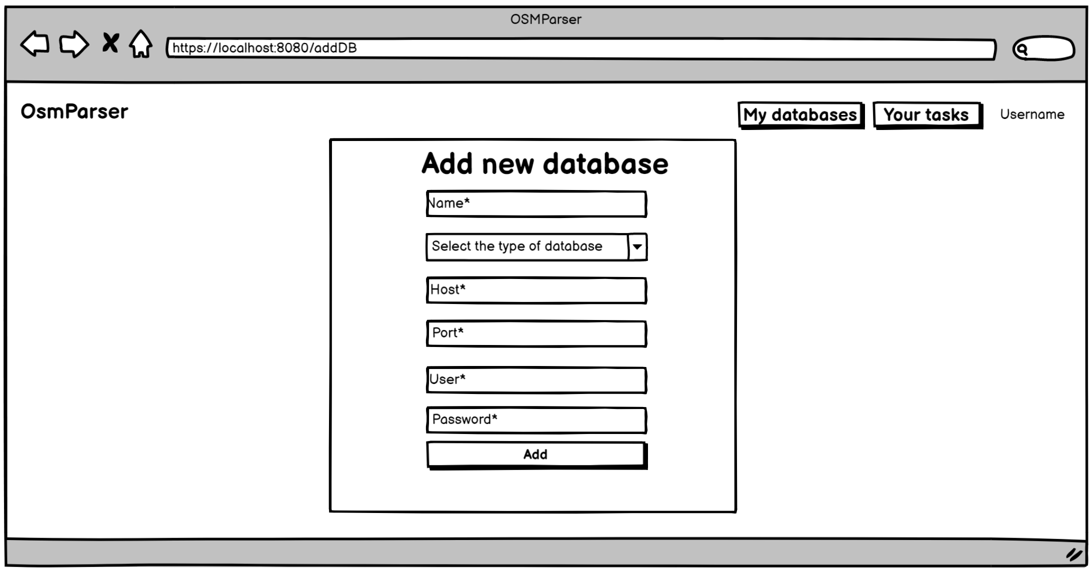

OSMParser
https://localhost:8080/login
OsmParser
Username
Password
Login
You do not have an account?
Sign Up

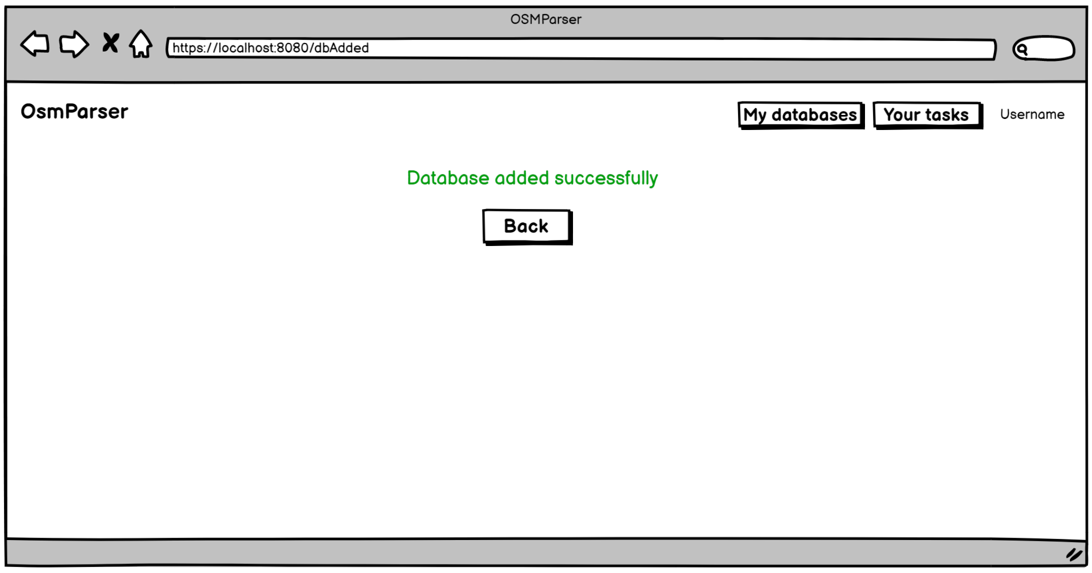

OSMParser
https://localhost:8080/signUp
OsmParser
Email*
Username*
Password*
Confirm password*
Sign Up


OSMParser
https://localhost:8080/newTask
OsmParser
My databases
My tasks
Username
Create new task
Name*
Database
Select database
Task content*
Execute

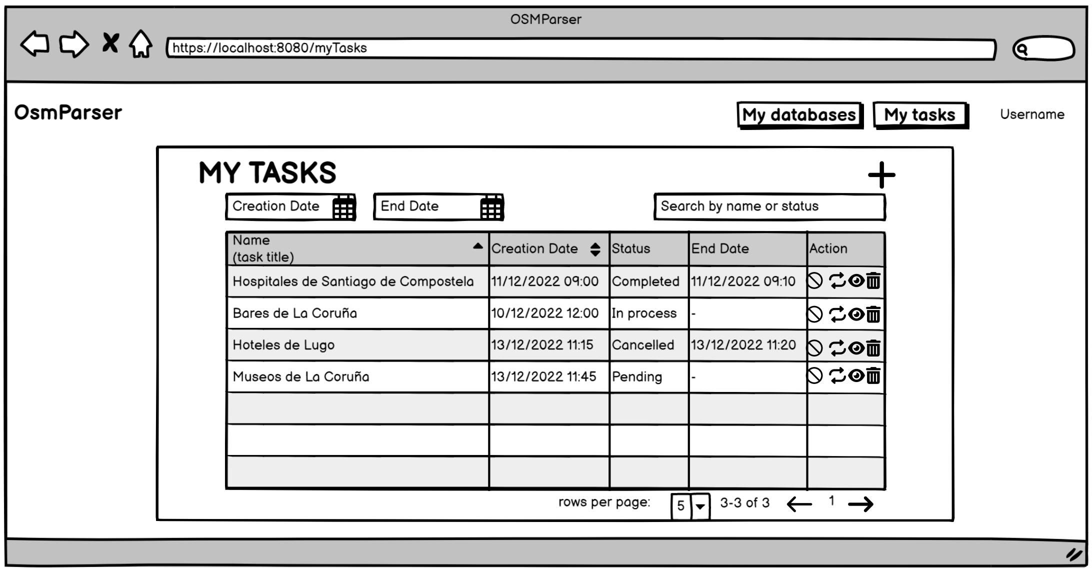


Task started successfully


OSMParser
https://localhost:8080/addDB
OsmParser
My databases
Your tasks
Username
Add new database
Name*
Select the type of database
Host*
Port*
User*
Password*
Add

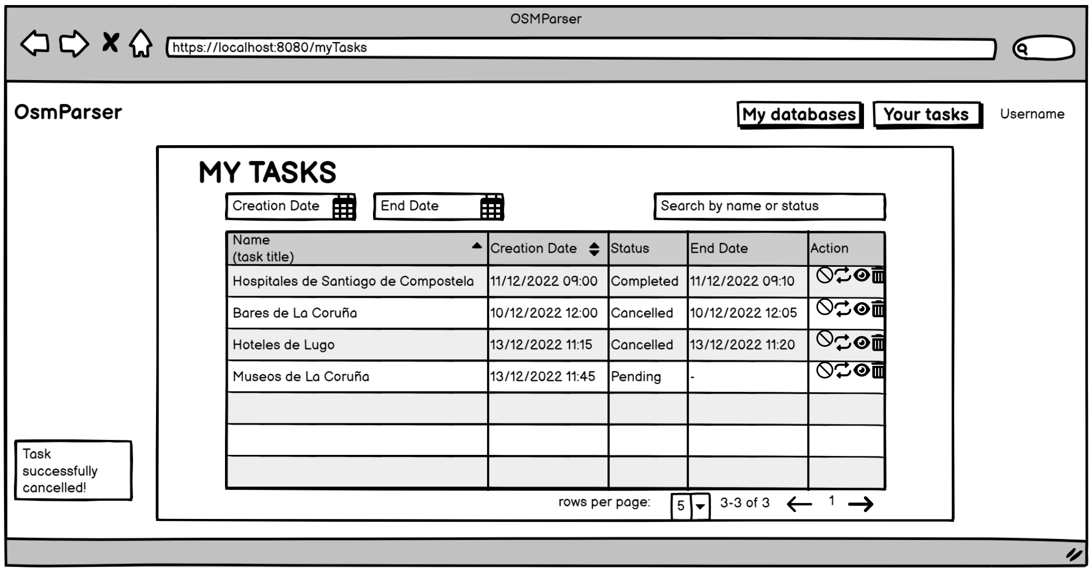


Database added successfully



MY TASKS

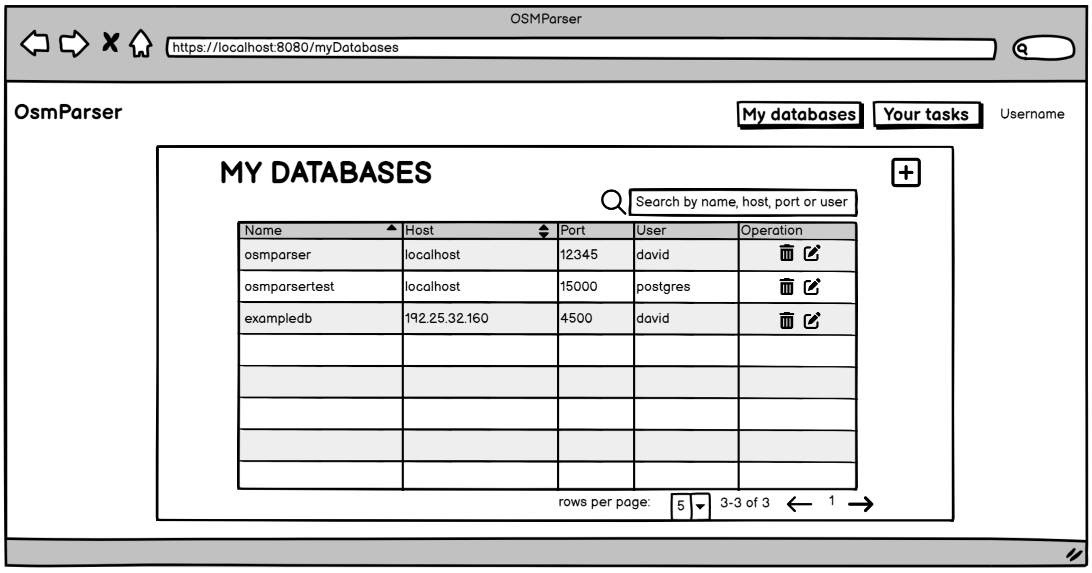
| Name (task title) | Creation Date | Status | End Date | Action |
|---|---|---|---|---|
| Hospitales de Santiago de Compostela | 11/12/2022 09:00 | Completed | 11/12/2022 09:10 | |
| Bares de La Coruña | 10/12/2022 12:00 | In process | - | |
| Hoteles de Lugo | 13/12/2022 11:15 | Cancelled | 13/12/2022 11:20 | |
| Museos de La Coruña | 13/12/2022 11:45 | Pending | - | |
| | | | | |
| | | | | |
| | | | | |



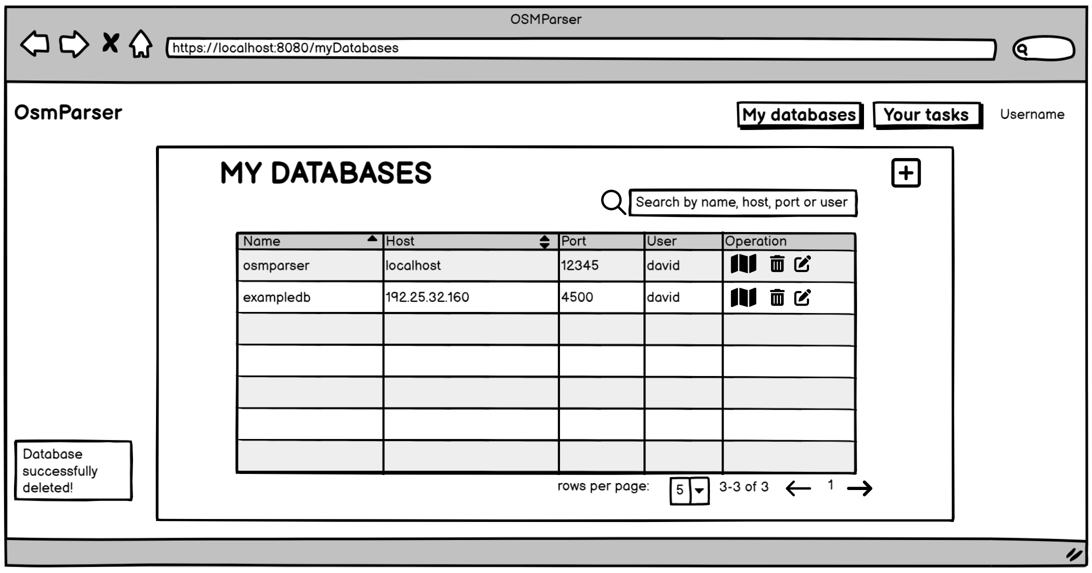
MY TASKS

| Name (task title) | Creation Date | Status | End Date | Action |
|---|---|---|---|---|
| Hospitales de Santiago de Compostela | 11/12/2022 09:00 | Completed | 11/12/2022 09:10 | |
| Bares de La Coruña | 10/12/2022 12:00 | Cancelled | 10/12/2022 12:05 | |
| Hoteles de Lugo | 13/12/2022 11:15 | Cancelled | 13/12/2022 11:20 | |
| Museos de La Coruña | 13/12/2022 11:45 | Pending | - | |
| | | | | |
| | | | | |
| | | | | |

Task successfully cancelled!


OSMParser
https://localhost:8080/myDatabases
OsmParser
My databases
Your tasks
Username
MY DATABASES
Search by name, host, port or user

| Name | Host | Port | User | Operation |
|---|---|---|---|---|
| osmparser | localhost | 12345 | david | |
| osmparsertest | localhost | 15000 | postgres | |
| exampledb | 192.25.32.160 | 4500 | david | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |



OSMParser

https://localhost:8080/myDatabases

OsmParser

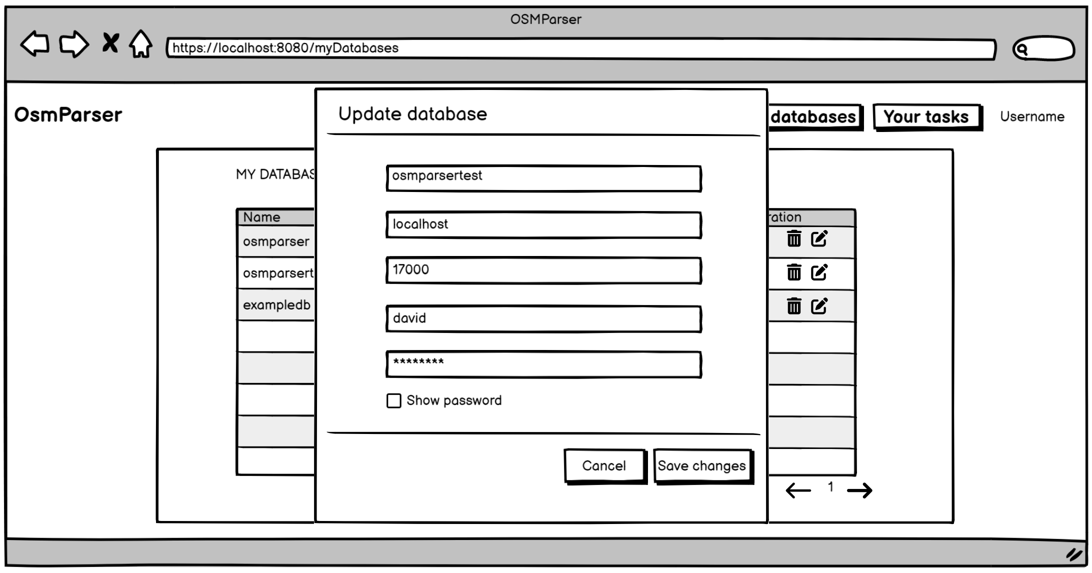
MY DATABASES

Search by name, host, port or user

| Name | Host | Port | User | Operation |
|---|---|---|---|---|
| osmparser | localhost | 12345 | david | |
| exampledb | 192.25.32.160 | 4500 | david | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

rows per page: 5 3-3 of 3 1

Database successfully deleted!



OSMParser

https://localhost:8080/myDatabases

OsmParser

databases

Your tasks

Username

MY DATABAS

| Name | ration |
|---|---|
| osmparser | |
| osmparsert | |
| exampledb | |

1

Update database

osmparsertest

localhost

17000

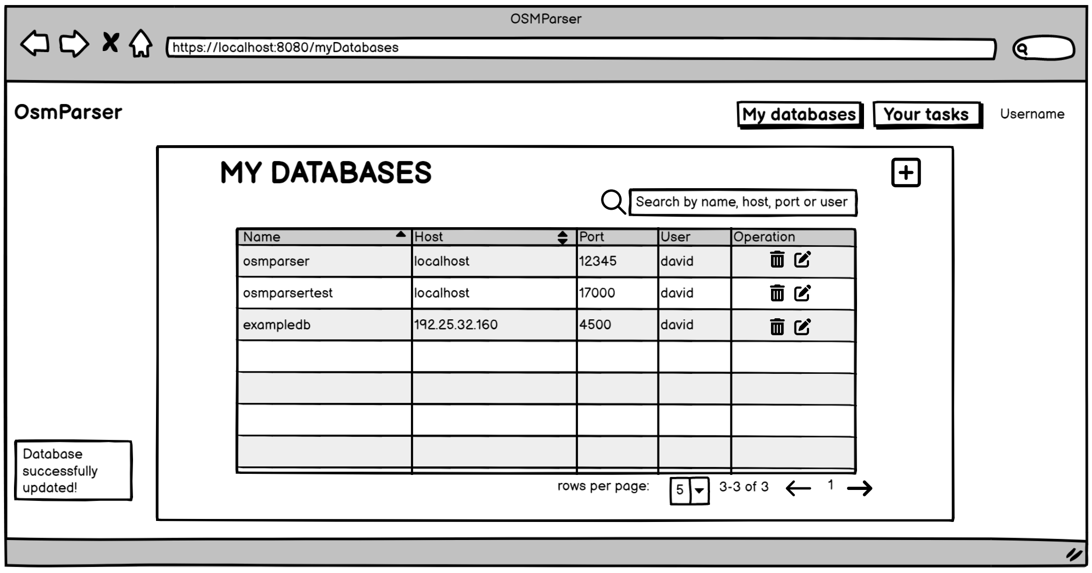
david

********

Show password

Cancel

Save changes



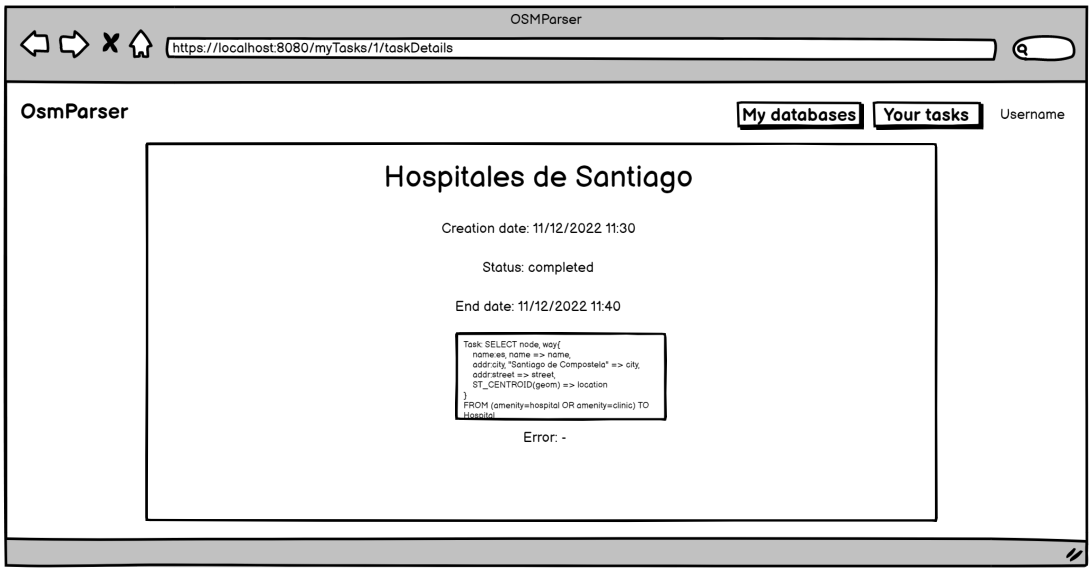
MY DATABASES

Search by name, host, port or user

| Name | Host | Port | User | Operation |
|---|---|---|---|---|
| osmparser | localhost | 12345 | david | |
| osmparsertest | localhost | 17000 | david | |
| exampledb | 192.25.32.160 | 4500 | david | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

rows per page: 5 3-3 of 3 1

Database successfully updated!



Hospitales de Santiago

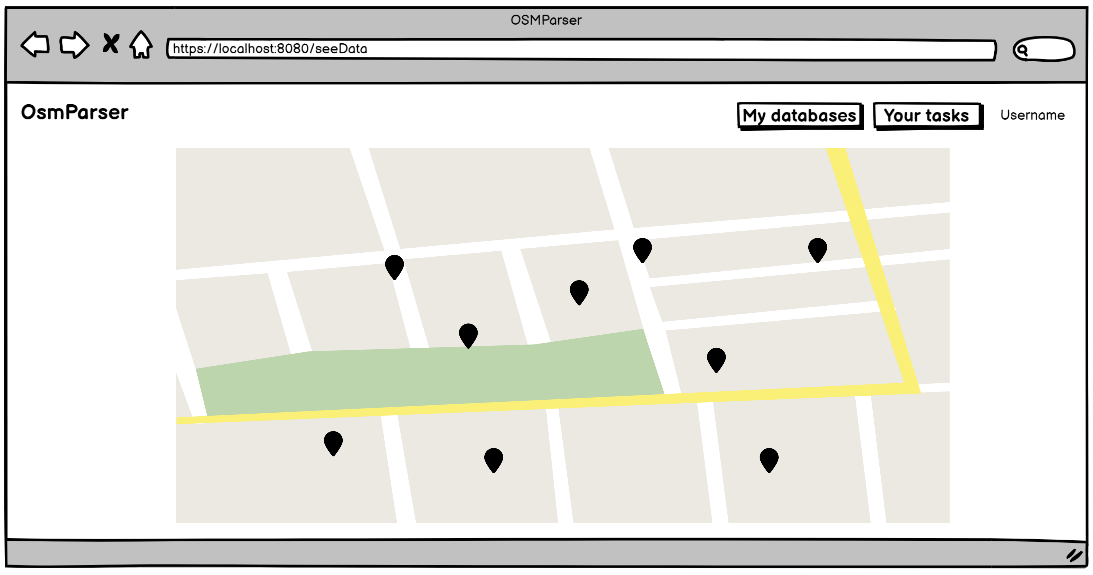
Creation date: 11/12/2022 11:30

Status: completed

End date: 11/12/2022 11:40

```
Task: SELECT node, way{
   name:es, name => name,
   addr:city, "Santiago de Compostela" => city,
   addr:street => street,
   ST_CENTROID(geom) => location
}
FROM (amenity=hospital OR amenity=clinic) TO
Hospital
```

Error: -


OSMParser
https://localhost:8080/seeData
OsmParser
My databases
Your tasks
Username

Appendix C

# DSL structure

```
grammar OSMGrammar;

parse
    : (connectStatement | statement)+
    ;

statement:
    SELECT_SYMBOl elements (
        selectStatement
    )
;

selectStatement:
    OBRA_SYMBOL attributeDefinition (COMMA_SYMBOL?
    attributeDefinition)* CBRA_SYMBOL
    fromStatement entityStatement bboxStatement
;

fromStatement:
FROM_SYMBOL OPAR_SYMBOL valueExpression CPAR_SYMBOL
;

entityStatement:
TO_SYMBOL entity
;

bboxStatement:
WHERE_SYMBOL bboxDefinition
;

connectStatement:
    CONNECT_SYMBOL TO_SYMBOL dbaseElement OF_SYMBOL typeElement
    FROM_SYMBOL portElement OF_SYMBOL hostElement WITH_SYMBOL
    userElement
    AND passwordElement
;

passwordIdentifier
: identifier
| PASSWORD_SYMBOL
;

dbaseElement:
DBASE_SYMBOL EQUAL_SYMBOL identifier
;

typeElement:
TYPE_SYMBOL EQUAL_SYMBOL identifier
;

portElement:
PORT_SYMBOL EQUAL_SYMBOL INT_NUMBER
;

hostElement:
HOST_SYMBOL EQUAL_SYMBOL identifier
| HOST_SYMBOL EQUAL_SYMBOL IP_SYMBOL
;

userElement:
USER_SYMBOL EQUAL_SYMBOL identifier
;

passwordElement:
PASSWORD_SYMBOL EQUAL_SYMBOL passwordIdentifier
;

elements: elementsValues (COMMA_SYMBOL? elementsValues)*;

elementsValues
: NODE
| WAY
| RELATION
;

entity: IDENTIFIER;

```

```
bboxDefinition:
    BBOX_SYMBOL EQUAL_SYMBOL OPAR_SYMBOL FLOAT_NUMBER
    (COMMA_SYMBOL? FLOAT_NUMBER)* CPAR_SYMBOL
;

valueExpression
: OPAR_SYMBOL valueDefinition (separator valueDefinition)*
    CPAR_SYMBOL (separator valueExpression)*
| valueDefinition (separator valueDefinition)*
;

valueDefinition
: identifier EQUAL_SYMBOL identifier
| identifier
| identifier IS_NOT_NULL_SYMBOL
;

separator
: OR
| AND
;

attribute: identifier (COMMA_SYMBOL? identifier)*;

attributeDefinition:
    attribute ARROW_SYMBOL IDENTIFIER
;

identifier: IDENTIFIER | function | STRING_DPOINTS | IDENTIFIER
    (MINUS_SYMBOL | UNDER_MINUS_SYMBOL) IDENTIFIER;

function:
    functionName OPAR_SYMBOL arguments? CPAR_SYMBOL // número
    ilimitado de parámetros
;

functionName: IDENTIFIER;

arguments
: expression (COMMA_SYMBOL? expression)*
;

expression
: function
| bool
| identifier
```

```
;

//----------------------------LEXER
    RULES---------------------------------------

fragment A : [aA];
fragment B : [bB];
fragment C : [cC];
fragment D : [dD];
fragment E : [eE];
fragment F : [fF];
fragment G : [gG];
fragment H : [hH];
fragment I : [iI];
fragment J : [jJ];
fragment K : [kK];
fragment L : [lL];
fragment M : [mM];
fragment N : [nN];
fragment O : [oO];
fragment P : [pP];
fragment Q : [qQ];
fragment R : [rR];
fragment S : [sS];
fragment T : [tT];
fragment U : [uU];
fragment V : [vV];
fragment W : [wW];
fragment X : [xX];
fragment Y : [yY];
fragment Z : [zZ];

fragment DIGIT    : [0-9];
fragment DIGITS   : DIGIT+;
fragment HEXDIGIT : [0-9a-fA-F];

fragment LETTER_WHEN_UNQUOTED_NO_DIGIT: [a-zA-Z_$\u0080-\uffff];
fragment LETTER_WHEN_UNQUOTED: DIGIT |
    LETTER_WHEN_UNQUOTED_NO_DIGIT;
// Any letter but without e/E and digits (which are used to match a
    decimal number).
fragment LETTER_WITHOUT_FLOAT_PART: [a-df-zA-DF-Z_$\u0080-\uffff];

fragment UNDERLINE_SYMBOL : '_';
fragment QUOTE_SYMBOL     : '"';

```

```
FROM_SYMBOL      : F R O M;
WHERE_SYMBOL     : W H E R E;
ENTITY_SYMBOL    : E N T I T Y;
SELECT_SYMBOl    : S E L E C T;
BBOX_SYMBOL      : B B O X;
TO_SYMBOL        : T O;
CONNECT_SYMBOL   : C O N N E C T;
OF_SYMBOL        : O F;
DBASE_SYMBOL     : D B A S E;
PORT_SYMBOL      : P O R T D B;
USER_SYMBOL      : U S E R D B;
HOST_SYMBOL      : H O S T D B;
PASSWORD_SYMBOL : P A S S W O R D D B;
WITH_SYMBOL      : W I T H;
TYPE_SYMBOL      : T Y P E D B;

TYPE
    : B O O L E A N
    | L O C A L D A T E
    | S T R I N G
    | I N T E G E R
    | L O N G
    | D O U B L E
    | L I N E S T R I N G
    | M U L T I L I N E S T R I N G
    | P O L Y G O N
    | M U L T I P O L Y G O N
    | P O I N T
    | M U L T I P O I N T
;

bool
: TRUE
| FALSE
;

OBRA_SYMBOL          : '{';
CBRA_SYMBOL          : '}';
OPAR_SYMBOL          : '(';
CPAR_SYMBOL          : ')';
COMMA_SYMBOL         : ',';
PCOMMA_SYMBOL        : ';';
DOT_SYMBOL           : '.';
ARROW_SYMBOL         : '=>';
HTAG_SYMBOL          : '#';
EQUAL_SYMBOL         : '=';
```

```
MINUS_SYMBOL       : '-';
UNDER_MINUS_SYMBOL : '_';
DDOTS_SYMBOL       : ':';
IS_NOT_NULL_SYMBOL : 'is not null';
IP_SYMBOL
: DIGITS DOT_SYMBOL DIGITS DOT_SYMBOL DIGITS DOT_SYMBOL DIGITS
;

AND : 'AND';
OR  : 'OR';
NOT : 'NOT';

TRUE  : 'true';
FALSE : 'false';

NODE     : 'node';
WAY      : 'way';
RELATION : 'relation';

INT_NUMBER : MINUS_SYMBOL? DIGITS;
FLOAT_NUMBER : MINUS_SYMBOL? (DIGITS? DOT_SYMBOL)? DIGITS;

WHITESPACE: [ \t\f\r\n] -> channel(HIDDEN); // ignore whitespace
COMMENT: '//' ~[\r\n]* -> skip;
SQL_COMMENT: '--' ~[\r\n]* -> skip;

IDENTIFIER:
    STRING
    | DIGITS+ [eE] (LETTER_WHEN_UNQUOTED_NO_DIGIT
    LETTER_WHEN_UNQUOTED*)? // Have to exclude float pattern, as
    this rule matches more.
    | DIGITS+ LETTER_WITHOUT_FLOAT_PART LETTER_WHEN_UNQUOTED*
    | LETTER_WHEN_UNQUOTED_NO_DIGIT LETTER_WHEN_UNQUOTED* //
    INT_NUMBER matches first if there are only digits.
;

STRING
: QUOTE_SYMBOL ( '\\' [\\"] | ~[\\"\r\n] )* QUOTE_SYMBOL
;

STRING_DPOINTS
: IDENTIFIER DDOTS_SYMBOL IDENTIFIER
;
```

# List of Acronyms

**ANTLR** ANother Tool for Language Recognition. 27, 43, 45, 71

**API** Application Programming Interface. 10

**CSS** Cascading Style Sheets. 6

**DAO** Data Access Object. 27

**DSL** Domain Specific Language. 1, 8, 43–47

**DTO** Data Transfer Object. 31

**GIS** Geographic Information System. 1

**HTML** HyperText Markup Language. 6

**IDE** Integrated Development Environment. 10

**JSON** JavaScript Object Notation. 31

**JWT** JSON Web Token. 39

**OSM** OpenStreetMap. 1, 17, 45–47

**REST** REpresentational State Transfer. 10, 27

**URI** Uniform Resource Identifier. 36

# Bibliography

[1] "DAO pattern documentation web page." [Online]. Available: https://gl.wikipedia.org/wiki/Data_access_object

[2] "Facade pattern web page." [Online]. Available: https://es.wikipedia.org/wiki/Facade_(patrón_de_diseño)

[3] "MVVM documentation web page." [Online]. Available: https://medium.com/flawless-app-stories/how-to-use-a-model-view-viewmodel-architecture-for-ios-46963c67be1b

[4] "Callback patern web page." [Online]. Available: https://www.monografias.com/trabajos37/call-back/call-back2

[5] "Promise pattern web page." [Online]. Available: https://es.stackoverflow.com/questions/64265/que-es-una-promesa-en-javascript#answer-64403

[6] "OpenStreetMap web page." [Online]. Available: https://www.openstreetmap.org/

[7] "Overpass API web page." [Online]. Available: https://wiki.openstreetmap.org/wiki/Overpass_API

[8] I.-A. D. Bank, "OSM Extraction Tool web page." [Online]. Available: https://code.iadb.org/en/tools/osm-extraction-tool

[9] M. Raifer, "Overpass turbo web page." [Online]. Available: https://wiki.openstreetmap.org/wiki/Overpass_API

[10] P. Software, "Spring web page," 2002. [Online]. Available: https://spring.io/projects/spring-boot

[11] "Spring Data JPA web page." [Online]. Available: https://spring.io/projects/spring-data-jpa

[12] M. Stonebraker, “PostgreSQL web page,” 1996. [Online]. Available: https://www.postgresql.org/

[13] “PostGIS web page.” [Online]. Available: https://postgis.net/

[14] T. Parr, “ANTLR web page.” [Online]. Available: https://www.antlr.org/

[15] E. You, “Vue.js web page,” 2013. [Online]. Available: https://vuejs.org/

[16] “VueRouter documentation,” 2013. [Online]. Available: https://router.vuejs.org/

[17] R. Dahl, “Node.js web page,” 2009. [Online]. Available: https://nodejs.org/es/docs/guides/getting-started-guide

[18] V. Agafonkin, “Leaflet web page,” 2019. [Online]. Available: https://leafletjs.com/

[19] “Bootstrap web page.” [Online]. Available: https://getbootstrap.com/

[20] “JISBD web page.” [Online]. Available: https://biblioteca.sistedes.es/articulo/simplificando-la-importacion-de-datos-de-openstreetmap-a-bases-de-datos-relacionales-mediante-un-lenguaje-especifico-de-dominio/

[21] “Latex web page.” [Online]. Available: https://www.latex-project.org/

[22] “Postman web page.” [Online]. Available: https://www.postman.com/

[23] “IntelliJ IDEA web page.” [Online]. Available: https://www.jetbrains.com/idea/

[24] “Balsamiq web page.” [Online]. Available: https://balsamiq.com/wireframes/

[25] “Draw.io web page.” [Online]. Available: https://www.draw.io

[26] “Gitlab web page.” [Online]. Available: https://about.gitlab.com/

[27] “Git documentation.” [Online]. Available: https://git-scm.com/

[28] “GeoJSON documentation web page.” [Online]. Available: https://geojson.org/

[29] “Password Encoder web page.” [Online]. Available: https://docs.spring.io/spring-security/reference/features/authentication/password-storage.html#authentication-password-storage

[30] “ExecutorService documentation web page.” [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ExecutorService.html

[31] “Executors documentation web page.” [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Executors.html

[32] “Spring Boot testing web page.” [Online]. Available: https://www.baeldung.com/spring-boot-testing

[33] “JUnit official web page.” [Online]. Available: https://junit.org/junit5/