INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CÂMPUS CAMPO MOURÃO

Yabi - Yet Another Business Inteligence

Vitório Miguel Prieto Cilia

Dissertação apresentada à Escola Superior de Tecnologia e Gestão de Bragança para obtenção do Grau de Mestre em Sistemas de Informação. No âmbito da dupla diplomanção com a Universidade Tecnológica Federal do Paraná.

Trabalho orientado por:
Prof. Albano Alves
Prof. Lúcio Valentin

Bragança
2018-2019

Dedication

I want to dedicate this work to my parents, who always offered their support and guided me to what is good and right.

Acknowledgment

Firstly, I would like to thank my friends Daniel Costa Valério, Henrique Pinheiro and Sávio Camacam. Together we formed the HEDANVISA group, not only sharing technical knowledge but also forming long-lasting relations.

I would like to also thank both institutions, Federal University of Technology - Paraná (UTFPR) and Instituto Politécnico de Bragança (IPB) for the opportunity of realizing my masters in the scope of a double degree program.

Special thanks Professor Marcos Silvano for enabling Computer Science students of UTFPR, Campo Mourão to apply for this program and to Bruno Mendes for his clarifying explanations in regards to writing this document.

Resumo

No contexto do Instituto Politécnico de Bragança durante o período de matrículas, o departamento de serviços informáticos é frequentemente interrompido em busca de questionamentos sobre as informações contidas nas bases de dados da instituição.

Para amenizar isso, o Yabi foi desenvolvido. Esta é uma aplicação Web construida com uma interface de usuário feita no Framework Angular e uma aplicação remota que implementa as funcionalidades necessárias e é escrita em Java com o framework Spring. De maneira geral ela fornece um portal que possibilita os colaboradoes da instituição a ter acesso as informações contidas nas bases de dados sem que seja necessário o conhecimento técnico.

Por fim, considera-se que a aplicação final atende aos requisitos de maneira suficiente para ser considerada útil e ao mesmo tempo fornece uma plataforma para desenvolvimentos futuros.

Palavras-chave: plataforma web, business intelligence, angular, spring boot.

Abstract

In the context of Polytechnical Institute of Bragança during student registration time, the information technology department is found to be frequently interrupted in order to attend inquiries regarding the information stored in their databases.

To mitigate this, Yabi was developed. It is a Web application built with Angular Framework for the user interface and Java with Spring for the functionalities. In general it provides a portal in which the institution's employees can access the information found in its databases without the need to have technical knowledge.

The developed application is found to attend most of the elicited requirements to be considered useful and offers a foundation for future improvements.

Keywords: web platform, business intelligence, angular, spring boot.

Contents

1 Introduction **1**

1.1 Context . . . 2

1.2 Objective . . . 2

1.3 Textual Conventions . . . 3

1.4 Document Structure . . . 4

2 Concepts and Technologies **5**

2.1 Front-end . . . 5

2.1.1 HTML & CSS & JavaScript . . . 6

2.1.2 Typescript . . . 7

2.1.3 SASS . . . 7

2.1.4 Angular . . . 7

2.1.5 Angular Material . . . 9

2.1.6 Sb-Admin-Material . . . 9

2.2 Back-end . . . 13

2.2.1 Stateless Web Application . . . 14

2.2.2 HTTP . . . 14

2.2.3 Java . . . 15

2.2.4 Spring . . . 16

2.2.5 MariaDB . . . 21

2.2.6 Apache Directory Studio . . . 21

2.3 Development 22
2.3.1 Apache NetBeans 22
2.3.2 Maven 23
2.3.3 Lombok 23
2.3.4 Visual Studio Code 24
2.3.5 Docker 25
2.3.6 Angular CLI 25
2.3.7 Firefox 26
2.3.8 Webpack 27
2.3.9 Postman 27
2.4 Chapter Conclusion 28

3 Project **29**
3.1 Requirements 29
3.2 Conceptual Model 33
3.2.1 Project Details 33
3.3 Chapter Conclusion 34

4 Implementation and Results **35**
4.1 Front-end 35
4.1.1 Interfacing with Spring Repository 36
4.1.2 Component Structure 41
4.1.3 Components 42
4.1.4 Generic Form Control Builder 50
4.1.5 Temporal Caching Repository 51
4.1.6 Error Handler 51
4.1.7 authenticationInterceptor 51
4.1.8 apiEndpoint 53
4.1.9 Shared Module 53
4.1.10 Security Concerns 54

4.2 Back-end . . . 54
4.2.1 Entities . . . 55
4.2.2 Spring Configuration . . . 56
4.2.3 Custom Controllers & View Models . . . 61
4.2.4 Spring Repositories . . . 64
4.2.5 Multi-Database Support . . . 65
4.3 Development Environment . . . 65
4.3.1 Directory Service . . . 65
4.3.2 Local Database . . . 67
4.3.3 Database Initializer . . . 67
4.3.4 Postman Tests . . . 67
4.4 Chapter Conclusion . . . 68

5 Conclusion **69**

6 Future Work **71**
6.1 Code Re-structure . . . 71
6.1.1 Resource Filtering . . . 71
6.1.2 PermissionTree's cyclic reference . . . 72
6.2 Bulk information manager . . . 72
6.3 Testing and User Validation . . . 72
6.4 Parameterization . . . 73

A Proposta Original do Projeto **A1**
A.1 Proposta nº 2 . . . A1

List of Figures

2.1 Login Screen 10
2.2 Dashboard with collapsed side menu 10
2.3 Dashboard with visible side menu 11

3.1 User use case 32
3.2 Administrator use cases 32
3.3 Yabi Overview 33
3.4 Conceptual Model 34

4.1 Class diagram that represents the structure of a Spring Repository response 39
4.2 Listing of all registered `Query` for a given `User` and `Role` information . . . 44
4.3 Dialog for assigning a new `Permission` to a `User` 45
4.4 Dialog for running a `Query` 47
4.5 Dialog for running a `Query` after it was executed 47
4.6 Dialog for creating a new `Permission` 48
4.7 Dialog for creating a new `Directory` 49
4.8 Dialog for editing an existing `Directory` 50
4.9 Login screen with a authentication error caught by the `SnackBarErrorHandler` 52
4.10 Authentication Sequence Diagram 58
4.11 Directory structure and the properties of user `professor` 66

Chapter 1

Introduction

As companies and institutions develop, they tend to implement and depend on digital systems **ibm**. Generally speaking, these solutions involve the deployment of a persistence mechanism such as a relational database or a directory service, Relational Database Management System (RDBMS) being the most common **apachedp**. Once made available, they are frequently updated with new information that, if not properly processed and made understandable, does not generate any meaningful insights.

It is not a hard task to give access to the raw information contained within a RDBMS but because of how it is usually split in logical relations to avoid unnecessary data repetition **vaquinha**, technical knowledge is required to harness its potential into a tangible understanding that can help in the decision-making process.

A good system would enable anyone, expert in computational systems or not, to correlate and extract any information held in their institution. However such system would have to deal with many corner cases and thus an approximation that is able to handle the most frequent cases is useful even if it requires some amount of manual maintenance.

1.1 Context

With more than 8,500 students and professors, IPB is composed of 5 schools that span diverse fields of study including but not limited to Education, Administration, Communication, Health, Tourism, Agrarian and Engineering.

Towards the beginning and the end of the semester, during student registration time, professors often need some insights on their educational affairs. To do so, they reach out to Information Technology (IT) department inquiring about a specific need, the technician then stop his current task, write a Structured Query Language (SQL) script, run it on the institution's RDBMS and email back the results as a spreadsheet file.

Over time the technician has built a list of about 40 common requests and their respective scripts so that this process takes less of his time, enabling him to continue developing IPB's in-house software. However, interruptions still happen often enough that an automated system is still needed.

1.2 Objective

The objective of this project is to develop Yet Another Business Intelligence (Yabi) a web platform that exposes SQL scripts to IPB's employees in a convenient way that does not require expert knowledge of the underlying system architecture and eases the IT department workload during the institution's critical moments.

Such platform will be maintained by an administrator that is responsible for registering the desired SQL scripts and based on their role in the institution, professors and other employees are presented to a list of queries coupled with meaningful title and description in which they can run, see the resulting table and download a spreadsheet file.

Following is a translation of the original proposal found in Appendix A:

> Proposal nº 2
>
> To architect and construct from scratch a "business intelligence" system with emphasis on education management. Nowadays we know how valuable and

important information is for those who manage institutions and the impact that data analysis tools have in the decision-making process. At the time writing, IPB is already provided of a centralized database through which a large number of SQL of many diverse purposes queries are run. The intention is to provide, based on certain criteria, access to information without having to manually write queries that are often more than 30 lines long.

The proposed system would be fed with "clusters" of queries and provide a way to easily insert and validate individual or group of queries depending on the currently logged-in user's profile.

Keywords are reuse and automatic parameterization of queries that are supported by an automatic web search interface based on the current query.

All in all, it is about deploying a intelligent search system that is able to adapt to the necessities and profile of each user. The end result will always be tables of data that can be exported to many different formats

1.3 Textual Conventions

Throughout this document, some words and terms were made to look different from standard text to help convey the context and indicate the class in which the subject is part of.

`Typewriter` This is used to reference pieces of code such as data types, class names, method invocations that are written with it's class name and abstract parts of a system. E.g. `Authentication`, `SqlQueryController`, `DatabaseReader.runQuery`, `varchar`.

Italic Refers to function and method names. E.g. *authenticate*, *runQuery*.

Bold Indicates resource paths. E.g. **/user**, **/PermissionTrees**, **/runQuery/{id}**.

Small Capital Indicates HTTP verbs. E.g. DELETE, GET.

1.4 Document Structure

This document is divided in six chapters. Chapter 2 introduces the tools and concepts employed in this application, grouping them by whether they were used in the front-end, back-end or during development.

With an analysis of this project's proposal and the context it supposed to be used, Chapter 3 elicits requirements, use cases and entities that are found to compose the system. Chapter 4 then goes through the implementation of those entities and use cases into front-end and back-end applications alongside the tools used to mimic the production environment locally.

Chapter 5 concludes this document with an overview of what was done and comments its current state and lastly, Chapter 6 presents a few ways in which this project could be further developed.

Chapter 2

Concepts and Technologies

Throughout the development of this project quite a few tools and technologies were employed. Therefore this chapter is going to introduce each one of them in a way that relates to their usage in this project.

Section 2.1 is focused in defining the technologies that were used to write a Web application that is able to make use of remote Application Programming Interface (API)s and dynamically adjust its layout according to the information that is being shown. Section 2.2 introduces the technologies that are related to back-end services and remote APIs that expose the real functionalities of a system. Lastly Section 2.3 describes the tools used to write, debug and build this project.

2.1 Front-end

When referring to an Web application, the Front-end is usually what is perceived by the users of a system. It is a very important topic when it comes to user acceptance as it is concerned with presenting the information, indicating the current state of the system and what actions the user is able to take.

The technologies that follow are for the most part, specific to this kind of task and are concerned with bridging the gap between the functions the system provides and the user

by specifying how a system will be visually arranged and how the functionalities will be interacted with.

2.1.1 HTML & CSS & JavaScript

The Hypertext Markup Language (HTML), the "World Wide Web's core markup language " **html** is a declarative language through which the vast majority of online content is structured, shared and accessed. It is a specification of elements that can be used to structure the content of web pages, such as headings, images, links to other documents, buttons and forms **htmlcss**.

Cascading Style Sheet (CSS) is another declarative language that pairs with HTML. Its purpose is to describe how the elements present in a web page are presented. Some definitions handle colors, fonts, element arranging, visibility, interaction and many others aspects **htmlcss**.

JavaScript is a interpreted general-purpose scripting language that was initially designed to run on Web pages but now it found its way in many other areas**js**. Its specification does not define much of standard functions found in other languages such as input/output and file access, instead each host environment expose JavaScript-accessible objects that are tied to their specific functionalities. Since 1997 the specification is regulated by European Computer Manufacturers Association (ECMA) and thus the language is formally known as ECMAScript **ecma**.

Together, these three technologies compose what is known as HTML5. HTML being the document structure in the sense of logically grouping elements, CSS defining the appearance with colors, sizes and positions, JavaScript enabling interactions and dynamic content.

2.1.2 Typescript

“A super-set of JavaScript that compiles to plain JavaScript ” **tswebsite**, Typescript is a language maintained by Microsoft and developed by *Anders Hejlsberg* that started development in 2012 with the goal of improving the quality and manageability of JavaScript code bases with features such as static typing and object-orientated qualities **tsrevealed**.

Ultimately, Typescript must be compiled to JavaScript before being executed and for compatibility reasons, the default JavaScript target is version EcmaScript (ES) version 3 but newer back-ends are also available.

2.1.3 SASS

Syntactically Awesome Style Sheets (SASS) is a augmentation of CSS and its main premise is to make large style definitions more manageable by implementing features that are similar to an object-oriented language such as with loops, variables, functions and rule nesting **sass**.

SASS files are processed into plain CSS so that it can be interpreted by the browser. In this project, the template described in Section 2.1.6 was developed using this preprocessor.

2.1.4 Angular

Angular is a front-end web framework that started its life as a side project at Google that has proved itself as a valuable tool for modern application development. The core idea is that HTML faults when it comes to declare dynamic content **angularjs** and, to remediate this, a new middle-ware is introduced between the rendered page and the underling code so that all the elements and events in the Document Object Model (DOM) are captured and made available to Angular Components, which in turn is able to react to them. This binding goes both ways, not only the DOM can trigger Angular but also Angular can issue a page re-render to reflect its new state.

The first version of Angular is now called AngularJs and can be included in a HTML document just like any other JavaScript library. This version proved it’s value but was

considered confusing and some times slow. Since then it entered Long Term Support (LTS) stage and no features are added. Angular version 2 and newer are a Typescript re-write that includes some new features that aid in the architecture and development of scalable and reusable code, namely, the introduction of Components, Router, Ahead-of-Time compilation and Observables **angular**.

An overview of key Angular elements follows:

Module Internally referenced as a `NgModule`.

These are the basic elements through which an Angular application is structured**angularmodule**. They declare the elements that will be provided to its child Modules, Services and Components.

Component Binds a Template to behavior and data.

Components are the elements that directly interact with the information perceived by an user. They typically rely on Services to acquire information and on Modules to fulfill their dependencies.

Router A special kind of service that is responsible for managing the navigation through an application, mapping Universal Resource Locator (URL)s to Components.

Service Akin to a library.

A Service has methods that can be used by multiple Components and other Services to provide some functionality. They are commonly implemented to act as the means of interaction to a remote API.

Template An augmented HTML file that is bound to a Component.

Effectively, they are the medium through which information is displayed and interacted with. Among other things, Templates can have elements that are dependent of some expression, values that are provided by a Component and events that notify the underlying Component.

The relationship between such elements is that a Component may be dependent on Services. Components and Templates together form what is called a View. Views and

Routers are exposed to the application under a Module. All Modules are located under a single root Module. Other important features include the dependency injection mechanism, template directives and directional data binding. Through these features Angular aims to be a highly modular framework capable of fast development.

2.1.5 Angular Material

Material Design is a set of guidelines and principles made by Google for designing User Interface (UI) that aims to bring natural and consistent interactions between users and computers. The guiding principle is based on paper and ink but it is not limited to what they can do in the physical world **materialdesign**.

Angular Material **angularmaterial** is the Google implementation of Angular Components such as buttons, text input and separators that follow the Material Design guidelines, providing a consistent look across devices and reducing the amount of effort required to design a consistent UI with common behaviors.

2.1.6 Sb-Admin-Material

To accelerate the development speed and have faster working prototypes, many web-based projects begin from a ready-made template. This saves time by keeping developers from re-writing common pieces of code commonly referred as "boilerplate".

SB Angular Material is an Angular re-write of a famous Bootstrap template called SB Admin **angulartemplate**. As the name implies this template tries to assess the need for an administrator panel, and in doing so it provides a set of ready-made components such as a login component as seen in Figure 2.1; the main screen with a top navigation and a collapsible side navigation components, seen in Figure 2.2 and 2.3. This template already encompass some amount of responsive design by toggling the side navigational panel to be collapsed depending on the user's screen width.

What follows is a brief explanation of this template's folder structure to highlight the main parts in which it can be extended to fit any particular project.

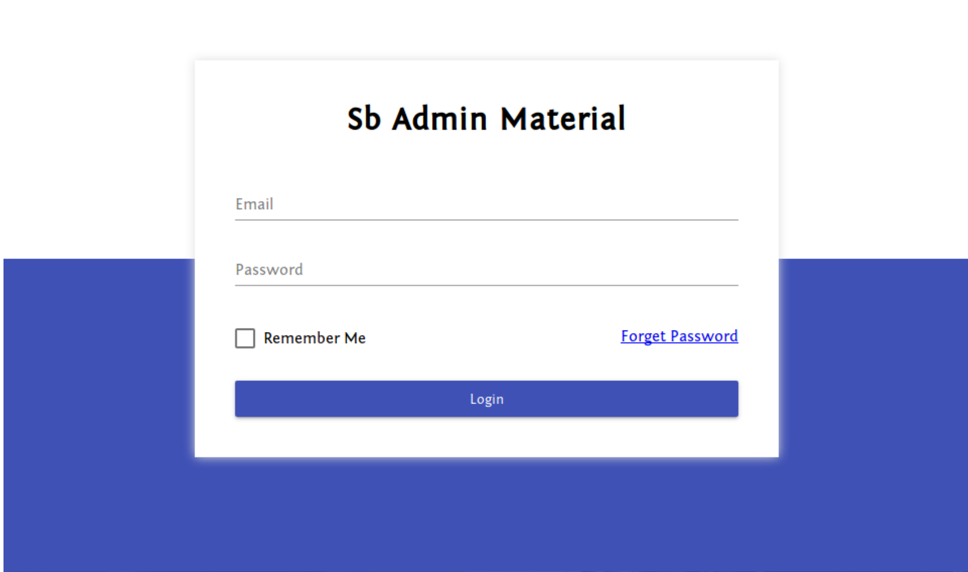

Figure 2.1: Login Screen

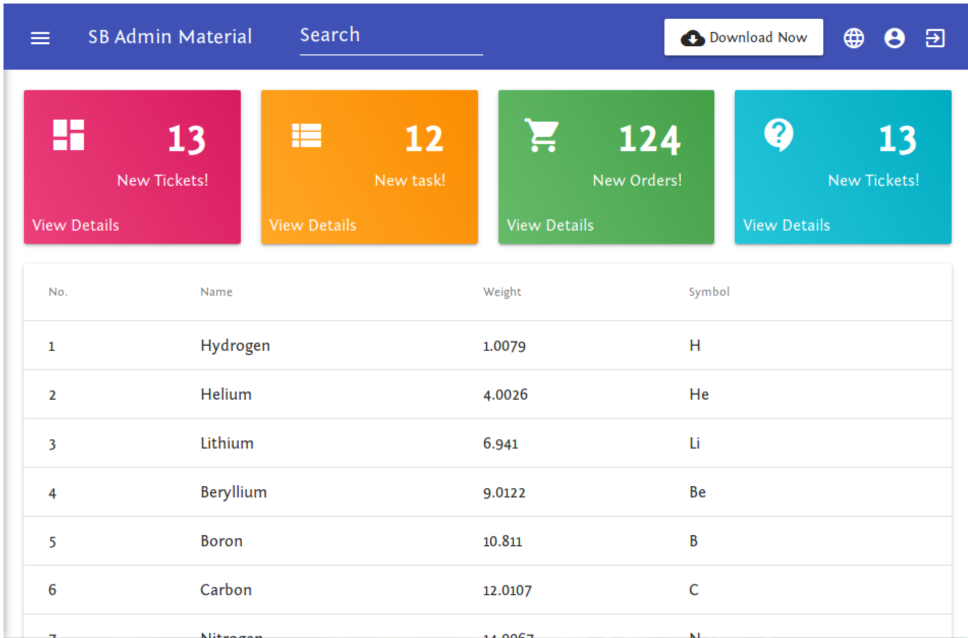

Figure 2.2: Dashboard with collapsed side menu

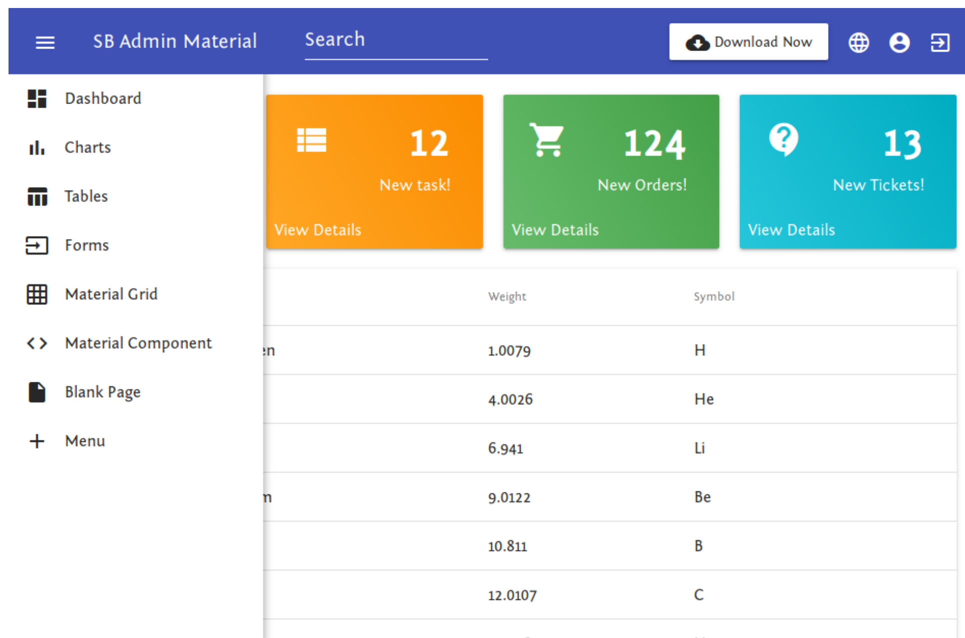

Figure 2.3: Dashboard with visible side menu

Project Structure

SB Admin Angular was written with the intention of being modified and extended by other developers. Because the team did not express any guidelines towards how it should be altered to accommodate a new project, it is important to give an overview of its file structure so that the changes made to it during the development of the present project are better understood.

- **root** This item is not a folder but the root of the project. In here there are configurations for code linter, JavaScript dependency descriptor and the license statement.
- **dist** Once the project is built for deployment, this directory will hold all the assets and optimized code ready for production, including the main `index.html` file that bootstraps the whole project.
- **e2e** This holds the source code for End-to-End test cases, hence the name.
- **src** This is the heart of this template, a directory that holds all the structure, content and behavior needed for each application.

- **app** The Angular entry-point. Contains the application-wide router module.
  - ➛ **layout** All the components used to compose the navigational elements and menus and their subsequent pages plus some example pages.
    - **black-page** A inaccessible component that does nothing, probably unfinished.
    - **blank-page** An example component that renders a white page.
    - **charts** A component that display chart capabilities of the integrated JavaScript module chartjs[1].
    - **components** Omnipresent page elements such as the Topbar and the collapsible Sidebar.
      - **topnav** The blue navigation top bar as seen on Figure 2.2.
      - **sidebar** The Menu on the left side of the screen as seen on Figure 2.3.
    - **dashboard** The page in which the user is redirected after logging in, shown in Figure 2.2.
    - **forms** Demonstration of the many different input methods such as Auto Complete text input, Date picker, Text Area and others.
    - **grid** A demo of the available page subdivisions.
    - **material-components** An example page displaying the main components of Angular Material such as buttons, Dialog and Notifications.
    - **nav** Unused component, deprecated by the top bar component.
    - **tables** A example component displaying Angular Material's table mechanisms.
  - ➛ **login** This is the Login component as see on Figure 2.1
  - ➛ **shared** Code that can be used in a application wide manner so that higher abstractions and code reuse can be achieved.
- **assets** Static content directory. Images, fonts, and i18n translations.

[1] Available at https://www.chartjs.org/

- **environments** Depending on how the project is run, either in development or in production mode, the respective configuration file that holds environment constants is used, allowing developers to use the same reference name throughout the code base no matter the environment.
- **styles** SASS files that define the look and feel.

After this overview, it is interesting to note the following:

- The Layout folder hosts, for the most part, Components and Modules that are listed in the sidebar.
- There are some unused components that were probably left over from design changes and were not deleted, which is the case of black-page and the nav component.
- There are many examples that proved as a handy reference during development, namely the forms and material-components.

2.2 Back-end

In a computer system, the back-end is concerned with providing functions that are required by an application, therefore they often interact with databases and perform data processing routines. The aggregate of these functions is also known as API. In the context of Web applications, the back-end is a separate program that runs on a remote machine and provide its functions through a set of URL addresses and Hypertext Transfer Protocol (HTTP) methods that together are referenced as a Web API.

This section describes the tools and concepts that were used to implement the functions that attend this application's requirements and the Web server that exposes them through an Web API.

2.2.1 Stateless Web Application

This is an architectural design that can applied when a system is composed from separate but communicating pieces. In this approach, messages exchanged between the logical entities contain all the necessary information for an action to be executed.

Often times web applications make use of session cookies or some similar mechanism to help the server-side to locate the current state for each specific user, in other words, the server has in its memory information that transcends requests. This is what is known as a state-full web application.

On a stateless model, each request contains all the information the server needs to generate a reply. When compared to a state-full application, this model requires more bandwidth as contextual information is added to every request, however it can mitigated with resource caching solutions.

2.2.2 HTTP

This is a stateless, general purpose, application-level protocol designed to be easily extendable. There are two roles that are involved during a HTTP communication, that of a client and a that of a server. The former initiates the connection and sends a request and the latter replies to the request and terminates the connection.

HTTP requests are composed of a textual structure that follows a Backus-Naur Form (BNF) with elements that indicate to the server what must be done. From its many elements, the main ones used in this document are the methods and Universal Resource Identifier (URI).

URI is a way to uniquely name a resource found in a registered name space. One of its applications is in the widely known URL, which is an URI that refers to objects accessible in the web.

HTTP methods, also known as verbs, are the indicatives of the action that is to be done on a resource. HTTP 1.1 Request For Comments (RFC) **http** proposes eight standard options, namely, OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT. Considered notorious for this document are GET, POST and DELETE.

The GET method purpose is to retrieve the information identified by the request's URI. POST is meant to send a new information to the server through the request itself. Lastly the DELETE request method expresses to the server that the resource is to be made inaccessible.

2.2.3 Java

Java is an Object-Oriented programming language firstly developed by James Gosling at Sun Microsystems. It is statically and explicitly typed and gets compiled to a machine-independent byte code that is then interpreted by the Java Virtual Machine (JVM) **java**.

Because of its high adoption, many concepts were developed to accommodate its deficiencies and improve the development cycle. In fact, because of the recurring solutions for recurring situations in software design, a group of skilled professionals got together to discuss these patterns and wrote a book entitled "Design Patterns: Elements of Reusable Object-Oriented Software" **patterns**, bringing common Object-Oriented solutions for recurring problems. Being an Object-Oriented language, Java programs often make use of such patterns, allegedly bringing a faster development model with code that is stable.

View Model

The ViewModel is a piece of a bigger pattern called Model View ViewlModel (MVVM) created by John Gossman. It addresses the scenario in which a model as described in the Model View Controller (MVC) pattern can't be completely mapped to a View **viewmodel**, that is, it has attributes that are part of a higher abstraction. This makes it sensible to specify another model that partially reflects the original model but is able to be completely bound to user interface elements.

During the implementation of this project, the ViewModel pattern was used outside of the MVVM pattern, however the goal of decoupling code remained the same. In some cases, ViewModel classes were implemented for classes that can not have all of its attributes serialized into a View or for those who must expose different attributes depending on the user who is requesting it.

2.2.4 Spring

Developed by Pivotal Software in 2002, the Spring Framework provides most of the "plumbing" necessary for fast development and deployment of enterprise Java applications **springdocs**. Some of the main features include: Dependency Injection which is an Inversion of Control (IoC) mechanism; A set of tools for information access such as Object-Relational Mapping (ORM) configurations called Spring Data and application-wide security mechanisms and configurations called Spring Security.

One of the strongest design philosophies of the Spring Framework is the following:

> "Provide choice at every level. Spring lets you defer design decisions as late as possible. For example, you can switch persistence providers through configuration without changing your code. The same is true for many other infrastructure concerns and integration with third-party APIs." **springdocs**

This leads to a feature-centered development model that is able to quickly deliver prototypes and changes on-demand.

The following elements in this section describe some of its functionalities that were considered important during development as they dramatically interfere with how the code is structured and features are implemented.

Dependency Injection

Conceptually, Dependency Injection is an implementation of the IoC principle, abdicating any given class from managing its own dependencies and leaving them to an overseer object that knows how to create and inject dependencies to each class**inversion**.

In practical terms, when writing a new class the programmer declare its dependencies through some mechanism in which the framework is able to reason about. Later in the run-time when a such dependent class is about to be instantiated, its dependencies are made available and injected into the new instance.

The key idea is that for the most part an application does not need to know which specific class is provided as long as it implements some given interface. It is the Framework's job to choose which class is injected. The programmer, however, is able to tailor the Framework's behavior to their liking.

When using Spring, one can express dependencies by declaring them in the class's constructor or by annotating an attribute using the Autowired annotation**springdi**.

Data

When developing an enterprise-level application, often times there is a need for some sort of persistence storage, in practice this usually translates to a RDBMS. To reduce the amount of code needed to manage such interactions, Spring Data module was developed. Its main interface is called `Repository` and it decouples entities that are being persisted from the underlying storage system **springdata**.

Bridging the gap between the `Repository` interface and its implementation, two other interfaces are provided such that when extended, provides the programmer with methods for interacting with a persistence system in a high level of abstraction. The base one is called `CrudRepository` and it provides the basic functionalities for persisting objects and a Web Representational State Transfer (REST) interface form them. Extending `CrudRepository` there is the `PagingAndSortingRepository` that adds pagination and sorting on top of it.

The `CrudRepository` on its own provide the following default methods:

save(Entity) Persists a entity in the data store and returns the saved instance with updated generated values.

findOne(ID) Retrieve one entity from the data store. If not found, returns null.

Listing 2.1: Repository for YabiUser

```
1 public interface YabiUserRepository extends PagingAndSortingRepository<
      YabiUser, Long> {
2     public YabiUser findByName(String name);
3 }
```

findAll() Retrieve all entities currently stored as a list.

count() Returns a number that represents the amount of entities currently stored.

delete(Entity) Deletes the given entity. Returns nothing.

exists(ID) Checks the data store for the existence of the given primary key value.

As shown in Listing 2.1, a entity-specific repository can be made by declaring a new interface that extends a `Repository` and specify the generic types for the entity and the primary key and lastly declare custom methods following a naming convention. In this case a `PagingAndSortingRepository` was defined for `YabiUser` as `YabiUserRepository` with a `Long` as its primary key and it exposes a new method that retrieves an instance of `YabiUser` given it's username.

These functionalities are made available through the following modules that come bundled with Spring Data:

- Spring Data Java Database Connectivity (JDBC).
- Spring Data Java Persistence API (JPA).
- Spring Data Lightweight Directory Access Protocol (LDAP).
- Spring Data REST

Hypermedia As The Engine Of Application State (HATEOAS)

It all begins with REST, that according to its creator:

> “REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication, while at the same time maximizing the independence and scalability of component implementations. This is achieved by placing constraints on connector semantics, where other styles have focused on component semantics. REST enables the caching and reuse of interactions, dynamic substitutability of components, and processing of actions by intermediaries, in order to meet the needs of an Internet-scale distributed hypermedia system.” **fielding**

In other words, this “new” architecture expresses requests and responses as the application state itself, transmitting it in a standard client and server approach. There are a few ways in which the state can be communicated and structured, one of which is the subject of this section.

HATEOAS is a response structure that enables a client to discover and navigate related information for that resource**fielding**. Mainly it is able to specify what are the related information, where it is located and how to interact with it. This is accomplished by including some meta-data in the response in which the client can parse, present to the user and issue proper requests.

In the context of Spring Framework, `CrudRepository` and `PagingAndSortingRepository` are interfaces that when implemented by Spring, generate API endpoints that follow both REST and HATEOAS conventions for Create Retrieve Update Delete (CRUD) operations. Figure 4.1 show a listing generated by `SqlQueryRepository`, a repository that extends the `PagingAndSortingRepository`. Note how it references the associated elements and available actions with a URL and a representation of the current pager state under the key `page`.

Security

This spring project aims to provide both authentication and authorization mechanisms throughout the application's components by exposing implementable interfaces that enable developers to override only the necessary parts for each specific need.

It is important to clarify the distinction between Authentication and Authorization because they have their respective software counterparts that play important roles in Spring Framework.

Authentication, by definition means "To prove real or genuine" **merriamwebster**. In Spring this translates to a custom extension of the `WebSecurityConfigurerAdapter` abstract class that defines how to verify that a given user exists and allows access the resources. There are a few ways to achieve this with two of the most common being: JDBC authentication, through which credentials are queried and matched from a RDBMS and LDAP authentication, that binds to some remote directory for the given user and password pair. Note that it does not define *what* may be accessed by such user, only if the user has access to the system as a whole.

Authorization, "the act of endorsing, or permitting by some recognized authority" **merriamwebster** Similarly to Authentication can also be specified via a custom extension of the interface `WebSecurityConfigurerAdapter`, co-existing with the previously mentioned Authentication mechanism. Authorization mechanisms are usually related to some attribute of the current authenticated user. In Spring, the `GrantedAuthority` interface is the central piece that unifies what the user has access to. Authorization points can be defined at the global level by the `WebSecurityConfigurerAdapter`, at the controller level or at the method level through proper annotations.

Boot

Although Spring Framework is a marvelous piece of software for its malleability and wide range of available features, for a while it was considered a *Configuration Hell* because of its eXtensive Markup Language (XML) configuration that would require a lot of expertise

into writing **xmlhell1 xmlhell2 xmlhell3 xmlhell4 xmlhell5**. In face of this, the Spring Team came up with Spring Boot, a dependency that can be inserted into a project and provides sane, pre-configured Spring packages to accelerate development and keep code organized.

2.2.5 MariaDB

Due to legal concerns Michael Monty Widenius founded Monty Program AB, whose main product, MariaDB, started as a fork of his previous work, the MySQL RDBMS **MAVRO:2014**. Such relational databases allow the user to define data structures and perform operations such as inserts, retrievals, updates and removals through a language known as SQL

MariaDB is an open source project licensed under the GNU General Public License (GPL) and its current stable version is 5.2. Its SQL dialect and configuration files are either identical or very similar to those of MySQL. One of the main goals of MariaDB is to keep enhancing its performance **BARTHOLOMEW:2012**

MariaDB server is said to properly execute in many operating systems, namely Microsoft Windows, Solaris, Linux, MacOS and Free BSD. There are many packages that handle connections to MariaDB, graphically like DBeaver or phpMyAdmin, textually like mycli and not further than that, programming language connectors such as Java's JDBC **MARIADB:2019**.

2.2.6 Apache Directory Studio

LDAP in its core is a protocol defined by Internet Engineering Task Force (IETF)'s RFC number 4511 **ldaprfc** that defines access to X.500 compliant directory services. A Directory is an agglomeration of cooperative systems that serve structured information about the real world **x500**. Different from a traditional RDBMS, directory services are expected to be automatically accessed by other interconnected systems, therefore they are better optimized for frequent queries and fewer updates.

Alex Karasulu, founder of the Apache Directory Project, was right when he stated that the need for interconnected systems grew alongside the expansion of the Internet but unlike his expectation, Directory services were replaced with RDBMS systems that don't exactly address the same goals and further complicate interconnected systems**apachedp**.

Given this situation, his project have the goal to modernize the tooling and functionalities of Directory systems and in doing so, two main sub-projects were created: Apache Directory Service **apachedservice**, a modern, LDAPv3 compatible, Java based implementation of a Directory Service that introduces triggers, stored procedures, view and queues and Apache Directory Studio **apacheds**, a complete LDAP tool developed as an Eclipse Rich Client Platform (RCP) extension that offers a more friendly user experience with visual elements for LDAP Data Interchange Format (LDIF) editor, tree explorer and permission management.

2.3 Development

This section cover the tools used during the development phase of this project. The tools in question do not only satisfy the coding needs but also mimics the production environment through which the application interacts with so that no sensitive information was touched by a potentially insecure, unfinished application.

In general, there was a need to comfortably edit Java and Angular projects, with code completion and refactoring support; a project manager that automatically downloads dependencies; a container tool to quickly deploy an environment with databases and directory services without the need of editing non-functional configurations and lastly, a browser to access the system as the end-user would.

2.3.1 Apache NetBeans

One of the Duke's Choice Award winner **dukechoice**, NetBeans is a general-purpose, cross-platform Integrated Development Environment (IDE) mainly focused for Java development with maximum productivity. In 2016 NetBeans was added to the Apache

Incubator so that it could be further developed by the community **incubation**. As of 2019 it became one of Apache's Top Level project **graduation** and is expected to attract an even bigger community.

Because Java was the main focus of NetBeans during its first few years, support for the language is very broad in features. Developers can easily operate code with context-sensitive refactoring; mark line, method, expression and class breakpoints; step through paused code; automatic JavaDoc generation; semantic code completion and more **nbassistance**.

Through its module system, support for other languages and resources were introduced, namely source code management with Git, Mercurial and Subversion, database management with support for viewing data and running SQL queries, unit testing, PHP, HTML, JavaScript and CSS **nettutorials**.

2.3.2 Maven

Maven is a Build System mainly used for Java applications, that is: Through the `pom.xml` file, developers declare their project's attributes and dependencies file and if needed, tweak the building process; from there on, Maven is capable of downloading dependencies from a remote repository, compiling them if necessary and generate an executable Java ARchive (JAR) file or loadable library **maven**.

The project goal is to unify the project structure so that there is less time spent by the developer to understand how a given application code is arranged and to centralize common project actions such as the previously mentioned dependency resolution, run unit and integration tests and generate packages that are able to be distributed **mavenintro**.

2.3.3 Lombok

This plugin offers an annotation-based code-scaffolding tool for Java definitions. Given the right annotations, common methods like getters, setters and no attribute constructor

are automatically generated in build or compile-time **lombok**. This tool consists of a two-part system that includes the integration with the compiler/build-system and another one that interacts with the developer's IDE so that the completion system is able to recognize the implicitly generated methods.

Some of the notable annotations include:

- `@Data`, useful for Plain Old Java Object (POJO) classes, this annotation generates getters, setters, a string converter and equality methods.
- `@NoArgsConstructor` and `@AllArgsConstructor`, as the name implies, one generates a constructor that takes no arguments and the other, a constructor that generates all arguments.

At first glance this might not be a necessary tool given that most IDEs often have support for a similar form of code refactoring but the key difference is that lombok does not clutter the classes with generated implementation therefore it reduces the project's Line of Code (LoC) count.

2.3.4 Visual Studio Code

One of Microsoft's take on open-source, this code editor gained traction among developers as one of the most used code editors **vscodesurvey**. Like any other modern code editor, it offers syntax highlight; auto-completion, through Microsoft's *IntelliSense* integration and lastly a plugin system that enables users to add custom behavior and further develop the editor's support for programming languages **vscode**.

One of the acclaimed features that arose with Visual Studio Code was the open source specification of Language Server Protocol (LSP) **lsplaunch**. This specification aims to define a communication protocol that is used between a code editor, referred as a LSP client and a editor-independent program referred as a LSP server, that takes care of features such as code completion, highlight, error detection, contextual variable renaming and jumping to definition **lspspec**. This decoupling reduces the amount of code needed

to develop a highly capable editor because language-dependent support is now transferred to a LSP server.

2.3.5 Docker

Akin to a Virtual Machine (VM), containers provides a way to have a different computing environment than the active running in the hardware. The key difference is that instead of emulating the whole computing stack, from processor to applicaion, a container system shares the core host resources with its guests and thus is generally less resource-hungry. One downside of a container is that the guest operating system must share the same kernel with the host.

In the other hand, Docker is more than just the sandboxing of processes. It handles image building through a `Dockerfile` specification, containers that can be shared among different machines, an online registry of extendable containers and command line interface that downloads, builds and manages containers **dockerfag**.

Core Docker definitions are brought up **dockeroverview**:

Dockerfile A file that declares the steps taken to build an Image.

Image A blueprint of a container generated once a `Dockerfile` is built.

Container If an image is a compiled binary, a container is the running process.

Volume Is a shared folder between the host Operating System (OS) and a running container.

2.3.6 Angular CLI

Angular applications have a basic directory for its components. Often times a directory contains most of the code for a specific piece of an application, such as a Component definition, a Service, a Template, a Style definition, a Class definition and lastly, these related pieces are then declared in a Module definition.

Managing this volume of files and relations can sometimes lead to confusion. Angular Command Line Interface (CLI) was developed to make this task more manageable. With it a developer can quickly initialize a new application skeleton, generate Components and Modules, build a deployment-ready applicaion and run a testing server that re-compile with code changes **angularcli**.

2.3.7 Firefox

From the downfall of Netscape browser and the release of its source code, the Mozilla project started with the mission to ensure the Internet to be a global public place, open and accessible to all **firemission**. Its main product is the open source web browser, Firefox.

Firefox comes bundled with plenty of tools that facilitate web development. What follows is a description of the tools most frequently used during the development of this application.

Source Mapping Is the ability to map some generated code back to its source. Some web frameworks like Angular 2.1.4 generate applications that are not developed in JavaScript itself but compiled to JavaScript in order to be executed **srcmapping**. In the beginning this led to great confusion because the browser's built-in debugger would display not the original source but the generated code.

Debugger Support for breakpoints, conditional breakpoints, expression stepping and variable lookup, which is highly useful when coupled with the previous element **dbgmodernweb**.

Network Monitor Often times it is needed to inspect the outgoing requests and their responses. The integrated monitor is able to expose all elements of the communication exchange and measure the different attributes of network operations **networkmon**.

Storage Inspector Provides access to the information storage that is managed by the browser for each page **storageinspector**, namely:

- Cache Storage
- Cookies
- Indexed Database (DB)
- Local Storage
- Session Storage

Console Enables the input of expressions in the page context and output information associated to the current page, including explicit calls for the `console.log` function.

Page Inspector Examine the page's HTML structure and CSS rules **inspector**. Developers can quickly experiment new possibilities by temporarily altering CSS rules and the page's structure.

2.3.8 Webpack

With the growing complexity of web applications, websites got slower and development, trickier. Webpack was developed to generate an optimized, ready to run, package that can be deployed in production **webpack**. Such packages are not meant for code only, they may contain images, CSS rules and anything else. It offers an API that can transform the contents of a package before it is bundled, for example, Typescript sources and its compiled counterpart or extracting inline CSS from a HTML document into a separate file.

In the context of Firefox's debugger and angular 2+ applications, when serving the application using Angular CLI, discussed in Section 2.3.6, it automatically bundles Typescript source code so that it can be instrumented and debugged inside the browser by accessing the Webpack element under the debugging tab.

2.3.9 Postman

As programs grew larger and were split into smaller pieces, it is now a common practice to have a API that concentrate on the business logic and a Graphical User Interface (GUI)s

that consume them. Such separation of concerns got even more pronounced when Web systems popularized, web browsers acting as GUI and interacting with remote HTTP servers as their source of information.

As with any software, APIs need to be tested and validated in order to provide a good quality product. In this context Postman was developed to be a Web API suite, offering a nice user interface through which developers can not only send, receive and analyze HTTP requests but also generate and manage documentation so that front-end and back-end teams have a single source of truth, manage test cases for remote HTTP APIs, mock APIs that are still under development **postman**.

2.4 Chapter Conclusion

Divided into three larger groups, front-end, back-end and development, this chapter described every tool and technology that was used to implement this application. Front-end being concerned with the development of a website in which users can access information, the back-end with an Web API that does the actual work and finally development, with tools that were used to implement this application.

The next chapter analyzes the proposal and context in order to provide in a high level the entities and requirements that are considered necessary for this system.

Chapter 3

Project

This chapter is focused on explaining how the application was defined in its logical terms and requirements, therefore it is not concerned with how the application will visually be presented nor how it should be implemented in code but instead gives an overview of its functionalities. In sum, there were found eight requirements that can be modeled with four relating entities.

Section 3.1 describes the process used to elicit the requirements, the requirements themselves, the use cases that fit them and the entities that model them. Section 3.2 presents an Entity-Relational model that exposes the relations and constraints between the entities and in Section 3.2.1 there are some considerations about the developed system design.

3.1 Requirements

The requirement analysis done for this application relied mostly on what could be abstracted from the written proposal and from a few meetings with a subset of the stakeholders. In this case all evaluated stakeholders are members of IPB. They are professors, administrative staffs and members of the IT department. The first two being the target

audience for this application and the last one being responsible to maintain it by assessing any problems that may arise during application usage and augmenting it as new requirements appear.

After reading the written proposal alone, the following functional requirements were elicited:

1. The system should be able to run queries in the database currently employed at the institution.
2. Users can only run queries in which they have Permission to.
3. Query commands may be longer than 30 lines long.
4. Running a Query yields a table that can be downloaded.
5. No SQL knowledge is needed to execute a Query.
6. The system enables the insertion of new Queries.
7. Queries should be automatically parameterized.
8. Queries should be validated before they are added to the system.

With these requirements in place, a small set of use cases and four entities were found to attend the functional requirements to an acceptable extent. It is important to note that because these entities function more as information containers and do not exchange messages directly, they are better expressed through an Conceptual Model diagram as seen in Figure 3.4. Following this paragraph is a description of each elicited entity alongside their purpose and relationship to each other.

Database Where a Query is run.

It is responsible to hold the information that enables the access to a given database, effectively meeting requirement 1. In the most basic form, a connection requires the network address and authentication credentials.

Query A script that is run in a Database and gather information into a single table.

A SQL script must be issued during a session with a Database. To fulfill requirement 5, some meta information such as a title and a description so that the target audience is able to find the Query that fulfill their needs.

Permission The binding between Users and Queries.

In order to fulfill requirement 2, this entity is responsible to handle the relation between a User and all the Queries that he may access.

User Represents the person currently logged-in.

It has two purposes, first is to differentiate users according to their roles, either "Administrator" or "User" so that certain actions are disabled, for example the Administrative task stated in requirement 6. The second is to be used when filtering Queries so that requirement 2 is met.

The use case diagram on Figure 3.1 identifies one actor that represents the stakeholders that are meant to interact with the system for its functionalities, in other words, professors and administrative staffs. Under this "User" actor, there are two actions that can be taken. To "Run" a Query and to "Export a Spreadsheet". The former refers to viewing the results of running a query in the front-end itself, discarding the retrieved data when closing the session. The latter aims to provide a way to export the results of running a query into a spreadsheet file that can be downloaded.

The other use case diagram shown in Figure 3.2 references the actions that can be taken by the IT department represented here through the Administrator actor. Being in charge of maintaining the system, most of their use cases revolve around CRUD operations. The only exception is the "Associate User and Permission" use case that is to grant a user the privilege to run all queries under a permission.

According the intents of the stakeholders, this system should follow an architecture similar to that employed in their other projects, culminating to what is shown in Figure 3.3, with an web front-end that is accessed by both actors to realize their use cases, a

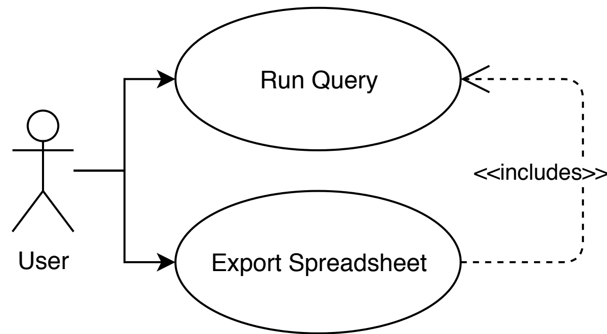

Figure 3.1: User use case

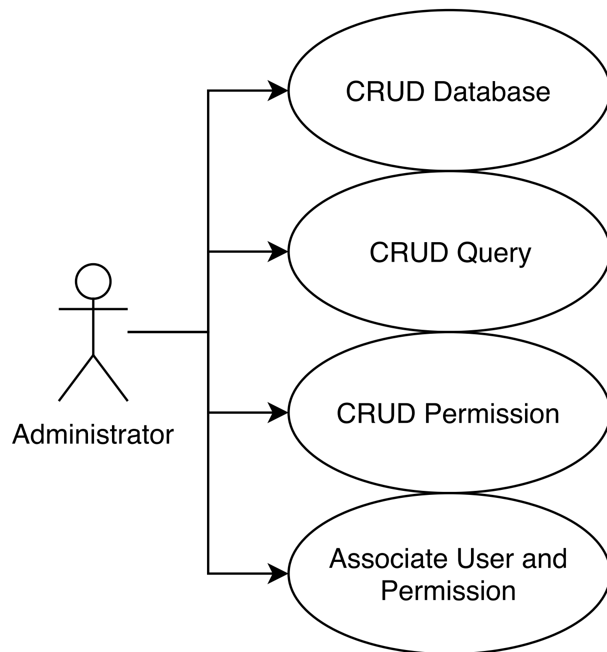

Figure 3.2: Administrator use cases

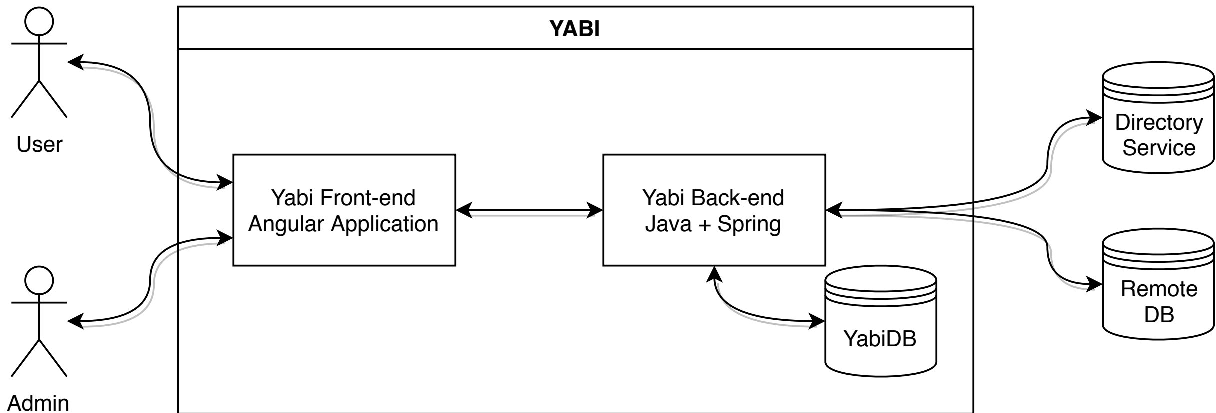

Figure 3.3: Yabi Overview

back-end service that provides business-specific functionalities and a database system to persist the system's own artifacts.

3.2 Conceptual Model

When implementing the entities that were previously evaluated, it was found that some of the proposed names were reserved to RDBMS and synonyms had to be used in the back-end, therefore Query became `SqlQuery`, User became `YabiUser` and Database became `Directory`. Permission is a separate case as it is not reserved by the RDBMS and was instead named `PermissionTree` to express its tree-like behavior.

More concretely, the entities relate to themselves as shown in Figure 3.4. In sum, a Query is associated to a single Database and a single Permission, User on the other hand may have many Permissions and Permission has a relation to its parent.

3.2.1 Project Details

Due to how the entities and their relations were designed, it is important to note some peculiarities and restrictions that came with it.

To begin with, Queries are associated to a single Permission and this led the listing of an User's Queries to be a unique list. An Administrator can manage all parts that

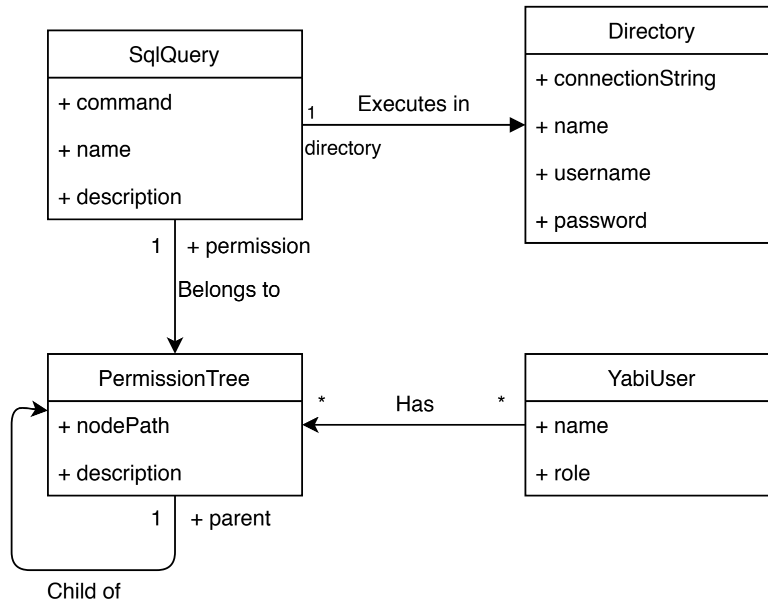

Figure 3.4: Conceptual Model

are within this system's domain and thus, he cannot change the username as it is bound to the Directory Service. Every permission has a reference to its parent and the root Permission has a reference to itself. There are only two roles that any given user may be assigned to, either Administrator or User.

3.3 Chapter Conclusion

This chapter expressed the scope of the proposed application, how it was logically modeled and the requirements it was expected to meet.

In order to reduce the amount of interruptions received by the IT department during student registration periods, this application seeks to give access to the information stored in IPB's databases in an organized manner. In sum, two actors were defined, the User whose main use cases consists of executing a Query and possibly downloading the results as a spreadsheet file and the Administrator, that associates one User with one or more Permissions.

The next chapter will be presenting how the entities and functionalities evaluated here were implemented into a functional application.

Chapter 4

Implementation and Results

In this chapter it is explained the implementation details of Yabi as a whole, encompassing the front-end Angular application in Section 4.1, the Java back-end Web API in Section 4.2 and lastly in Section 4.3, how some of the development tools were used to mimic the production environment and assess the back-end.

The resulting application is composed of two minor applications that are executed in different contexts, one meant to run in the browser the other in a remote server. The first being a friendly Web interface for the functions implemented in the last.

4.1 Front-end

This section describes the implementation of a website that exposes the functionalities of the developed system. It is meant to be accessed by all stakeholders and thus, it adapts to the current logged-in user based on their role. It was written using Angular Framework and closely models the back-end Web API, mapping its entities and responses into re-usable classes.

Beginning with Section 4.1.1 that describes the one of the most interesting part of this chapter that is the interaction with Spring Framework's Repositories in a generalized manner that is used in all entities. Section 4.1.2 explains in a generalized manner how the evaluated entities were translated into Angular Components that attend this system's

requirements, leaving each particular implementation to Section 4.1.3. The remaining sections discuss the implementation of more specific details that begins with Section 4.1.4 describing a custom helper function for generating forms, Section 4.1.5 showing how temporal caching was implemented, Section 4.1.6 explaining how errors were shown to the users, Section 4.1.7 describing how the authentication with the back-end was implemented, Section 4.1.8 explaining how the API endpoints were centralized, Section 4.1.9 showing how application-wise constants were implemented and Section 4.1.10 analyzing the application in regards to possible security flaws.

4.1.1 Interfacing with Spring Repository

Because the back-end is mostly implemented using Spring Repositories, which is an abstraction over the persistence of information, many API requests are followed by replies that have a general but not consistent JavaScript Object Notation (JSON) structure, making it time-consuming to use Typescript's typing capabilities for each and every entity and their Spring Repository structure.

Therefore the motivation behind the development of this architecture is to enable developers to make use of Typescript's typing system when dealing with a Spring Repository default responses and in doing so their IDE can show auto-completion suggestions, type-check the code and show possible errors before running the application.

Listing 4.1 show such a response from `SqlQueryRepository` that list all elements registered in its database.

Under the key `_embedded`, line 2, there is a single element whose key is the plural version of the entity's name and the value is a list of entities. The `_links` key exposes the actions that can be taken from this resource and `page` provides information about the current state of the pager.

For most applications, the most interesting part are the elements that make up the `_embedded.sqlQueries` array as it composed of serialized definitions of the back-end models. These definitions also follow the HATEOAS pattern of having their relations

listed as URLs. For example, a `SqlQuery` is related to one `Directory`, therefore it can be accessed through the **/sqlQueries/99/directory** URL, made explicit on line 15 and 16.

The common elements between all the implemented repositories are the pager section seen on line 37, the relative links to the listing itself seen on line 25 and the previously stated `_embedded` key that references the element array. Every one of the listed elements contain the self URL seen between lines 8 and 11 that references itself.

Although it is very declarative to human beings, this JSON format is not very machine-friendly due to the ever changing key that references the array of elements. As seen on line 3, `SqlQueryRepository` use the key `sqlQueries` and other repositories follow suit with `DirectoryRepository` using the `directories` key and `PermissionTreeRepository` using the `permissionTrees` key. This approach is not easily modeled in Typescript because the key being accessed is not known before running the code.

With the previous observations it was possible to abstract every repository response into three classes, a `Repository`, an `Accessor` and an `Entity`, enabling the implementation of a generalized repository that is able to interact with Spring's `PagingAndSortingRepository` in a convenient and typed manner. Logically a `Repository` provides access to an array of `Entity` through the key whose name is define through an `Accessor`. Figure 4.1 provides an overview of how these entities relate.

Throughout this document the term "hateoas class" refers to a Typescript class that extends an `Entity` and a "normal class" or "non-hateoas class" to be the class that reflects the entities evaluated in the Project section but do not extend `Entity`. This happens because when the system is accessed by an user, the requests are served by an hand-written endpoint that does not comply with the Repository structure and when it is accessed by an administrator Spring Repositories were used.

Entity Class

This class is typed to accommodate the structure found in the array of elements contained within a repository response, leaving each specification to extend it and add their own

Listing 4.1: SqlQueryRepository HATEOAS response

```
{
  "_embedded": {
    "sqlQueries": [
      {
        "command": "select␣*␣from␣grau;",
        "name": "Degrees␣of␣REBIDES",
        "description": "List␣scholl␣degrees␣declared␣in␣REBIDES",
        "_links": {
          "self": {
            "href": "http://localhost:8080/sqlQueries/99"
          },
          "sqlQuery": {
            "href": "http://localhost:8080/sqlQueries/99"
          },
          "directory": {
            "href": "http://localhost:8080/sqlQueries/99/directory"
          },
          "permission": {
            "href": "http://localhost:8080/sqlQueries/99/permission"
          }
        }
      }
    ]
  },
  "_links": {
    "self": {
      "href": "http://localhost:8080/sqlQueries{?page,size,sort}",
      "templated": true
    },
    "profile": {
      "href": "http://localhost:8080/profile/sqlQueries"
    },
    "search": {
      "href": "http://localhost:8080/sqlQueries/search"
    }
  },
  "page": {
    "size": 20,
    "totalElements": 4,
    "totalPages": 1,
    "number": 0
  }
}
```

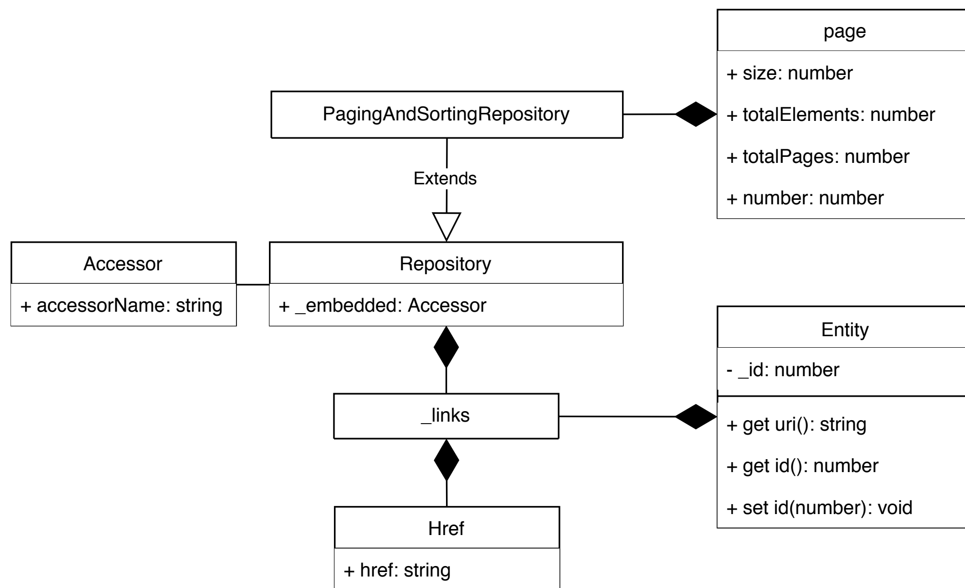

Figure 4.1: Class diagram that represents the structure of a Spring Repository response

fields. It also provides default implementations for `id` getter and setter and `uri` getter, that are useful when converting from hateoas classes to non-hateoas classes.

Accessor Class

To deal with the fact that the array of persisted elements is found under a key whose name depends on the entity in which the repository provides, the `Accessor` interface was written.

In its definition, there is only one field, `accessorName`, that contains a string which can be used to reference the array of persisted elements. Even though this approach functionally works, it compiles and runs, it does not provide the IDE with enough information to aid the programmer into navigating the response structure because the value of accessing an object with a key that is unknown in compile-time is also unknown.

To circumvent this limitation all classes that implement the `Accessor` interface should also contain an option key whose name is the same one defined in the value of `accessorName` and its type is an array of entities provided by the repository. With this, the IDE is able

to auto-complete and type-check the whole response, from repository to individual entity fields.

Repository Class

This repository attempts to map Spring Repository response as a whole. In other words, they are the top-most level of the response, mapping the first level of key-values to Typescript types. There are two classes that make up this abstraction, the `Repository` class that map the `_embedded` and `_links` keys and the `PagingAndSortingRepository` extension that adds a `page` key to the mapping.

Repository Service Class

When the previous classes were written to map the API responses, their similarities became more evident and the motivation to implement a generic service arose. With this came the `PagingAndSortingRepositoryService`, which is a service that implements CRUD interactions with a Spring Repository in a generically-typed manner. These CRUD interactions are translated to four functions. Following the `CRUD` acronym, they are the following: *create*, *index*, *patch* and *delete*.

Implementation-wise this class is composed of the three generic elements previously evaluated, `Entity`, `Accessor` and `PagingAndSortingRepository` its constructor needs a function that return new instances of `Entity`, an instance of `Accessor`, the `HttpClient` service and a string with the address of the remote Spring Repository.

While developing this service, an unexpected behavior was met. When specifying the type of a `HTTP` response, it did not create a new instance of the received data, which made the methods associated to the `Entity` class and its extensions to not function. This was mitigated by adding the required parameter in the constructor called `entityConstructor` whose value is a function that takes no arguments and returns an instance of the repository's entity, then every response follows the pattern of creating a new of such instance and copying all properties from the textual response to it using the `Object.assign` function.

Listing 4.2: Implementation of the Query model

```
1 export class DirectoryService extends PagingAndSortingRepositoryService<
2   HateoasDirectory,
3   DirectoryAccessor,
4   DirectoryRepository
5 > {
6   constructor(private _http$: HttpClient) {
7     super(() => new HateoasDirectory(), new DirectoryAccessor(), _http$,
          ApiEndpoint.DIRECTORIES);
8   }
9 }
```

With all this in place, implementing a service that has no behaviors other than CRUD operations is no more than specifying the `PagingAndSortingRepositoryService` class. Listing 4.2 show the implementation of `DirectoryService`. Note that all it does is extending `PagingAndSortingRepositoryService`, providing its `Directory`-specific classes instead of the generic triad and overloading the constructor on lines 21 to 23 so that it can further specify the parent's behavior.

4.1.2 Component Structure

When developing the Angular front-end over the `Sb-Admin-Material` template, it was noted that the example pages that could be accessed by links listed in the left sidebar, seen on Figure 2.3, were found inside the `/app/layout` folder. Therefore it made sense to follow this approach and implement Yabi's custom pages in the same place.

In general, each of Yabi's entity have got a folder that contains:

- A "model" file with:
  - A class that represents the entity, to be used when retrieving entities depending on the current user.
  - A class that extends `PagingAndSortingRepository`, to map Spring Repository responses.

    - A class that extends `Entity`, representing the elements contained within the repository response.
    - A class that extends `Accessor` that indicates key used to access the list of `Entity` contained within the repository response.

- A Service file that extends `PagingAndSortingRepositoryService`, specifying it for the given entity.
- The Template file that renders a listing with the available entities.
- A Component file that interacts with the Template and creates dialogs.
- The Style file with rules to correctly render the "add" button.
- A folder with a dialog Component for creating more entries.
- A folder with a dialog Component for editing an entry.
- The Module file declaring its dialog Components to also be loaded with `entryComponents`.

There are some variations to this rule. `User` Component has only one dialog that is used to show more information about the current user. `Query` Component was one of the first components to be developed and it has three dialogs, one for showing more information and the results of running it, a "form" dialog that maps a `Query` to input elements that is used for editing and creating.

4.1.3 Components

Every entity defined in the project section have got a corresponding Angular Component that follows the structure previously defined with a certain degree of freedom. In this section the particularities of each component in regards to each element will be shown, starting with their Service followed by Template, Component and ending in their Module, if applicable.

Login

`LoginService` is different than the other services because it does not extend the class `PagingAndSortingRepositoryService` and instead is concerned with providing login functionality and system-wide predicates about the current user or session. The two main predicates are *isAdmin* and *isAuthenticated*. The first is used on templates to hide elements that should not be seen by non-administrative users and the second is run before every request sent to API so that the application can redirect to the login page if by some reason the user access a page without first logging-in. It also provides *login* and *logout* function, the former saves the username and password combination in the local storage to be used by `authenticationInterceptor` and requests the API route **/user** for information about the current user; the latter deletes the current user's instance, clears the local storage information and redirects to the login page at **/login**.

This component came as part of the `Sb-Admin-Material` and can be seen on Figure 4.9. Mainly, it provides a standard text input for user identification, protected text input for passwords and a Login button that is associated with the component's *onLogin* method. The other HTML elements are not bound to any function.

Its Component is very short. It provides a *onLogin* method that in turn call the `LoginService.login` with the content written in the text inputs. If the login was successful it redirects to **/query**, otherwise it throws an error saying "Authentication Unsuccessful".

User

`UserService` extends `PagingAndSortingRepositoryService`, specifying it for the `User` model and provides functions for managing the association between user and permission, listing an user's permissions with the *permissions* method, creating the relation with *assignPermission* and removing it with *unAssignPermission*. It is worth noting that the process of creating this association involves a POST request to the user's permissions

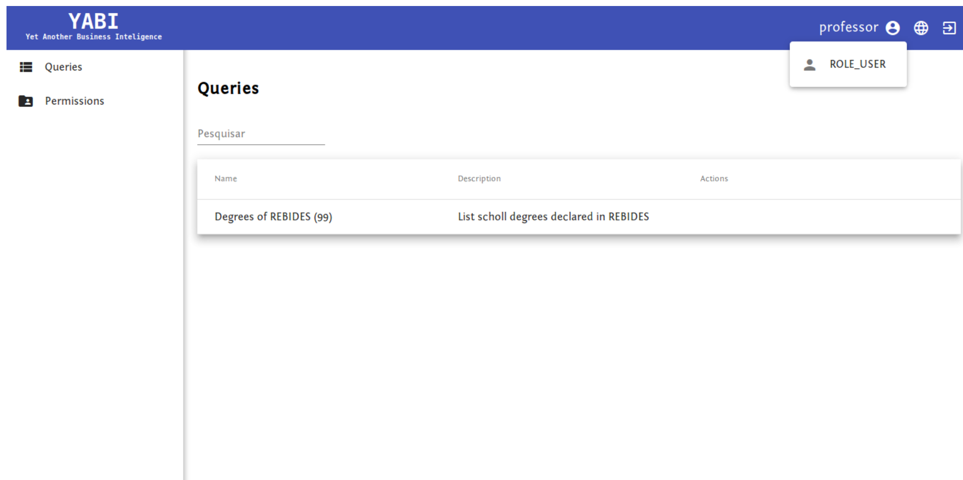

Figure 4.2: Listing of all registered `Query` for a given `User` and `Role` information

address in which the body is composed of line-break separated URL links and the header indicates a content type of `text/uri-list`.

When non-administrative users logs in the application, they are redirected to the **/queries** page so they can quickly execute one of their use cases. Figure 4.2 show the page with queries available to the current user. In the top right corner the username is shown to be `professor` and when clicking the first circle to its right a small pop-up appears with their role. One last thing to note is that non-administrative users such as the one from the figure, do not have access to all screens and buttons. In comparison to Figure 4.3, administrators have buttons on the side menu that redirect to `Directory` and `User` components, and also a pink round button with an plus sign on the bottom right that enable the insertion of new entries.

These administrative elements are triggered visible though the *isAdmin* method from `LoginService` and they serve a purely cosmetic function as the back-end denies the execution of administrator functions for standard users.

The use case of assigning permissions to users is done by administrators at the **/user** page. The listing itself is very similar to the one seen on Figure 4.2 albeit with two columns, one for the username and the other for role.

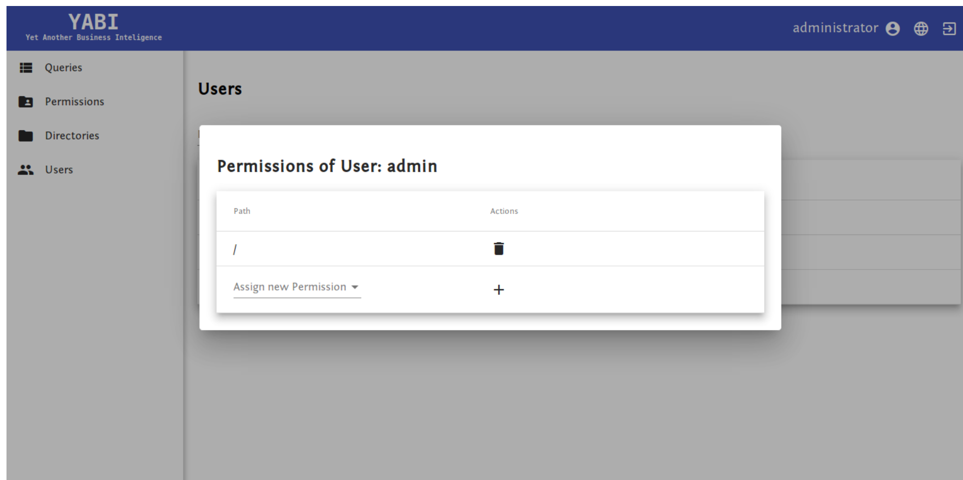

Figure 4.3: Dialog for assigning a new `Permission` to a `User`

As seen on Figure 4.3, when a line of the user listing table is clicked, a `UserShowComponent` dialog appears with a table that lists their associated permissions and possible actions. From there, the administrator is able to remove a user's permission by clicking on the trash can symbol or grant new permissions by choosing one in the drop-down list that says "Assign new Permission" and clicking on the adjacent plus symbol.

The `HateoasUser` class extends the generic `Entity` with attributes that compose the back-end implementation of the user. Interestingly enough, because back-end class is written to fit within the Spring Framework authentication mechanism, it has some unused back-end specific attributes added to what was initially defined in the project, these include the attributes `enabled`, `authorities`, `password` (left empty), `accountNonExpired`, `accountNonLocked` and `credentialsNonExpired`.

Query

`QueryService` is quite different when compared with other services because it implements the standard features found in `PagingAndSortingRepositoryService`. This is because it was one of the first services to be developed and the need for a common interface was not yet evaluated. To keep compatibility, with the current modules, the function names and signatures were kept intact but their implementation makes use of `HateoasQueryService`,

which does extend the common service. Regarding the other services, this one provides a different behavior for *index* that requests the queries in which the current user have access to. Lastly there is the *run* method which is specific to this service, requesting the API into running a query and yielding its results.

Shown on Figure 4.2, the **/query** page presents a table that list queries in regards to theirName, Description and Actions. The last one being dependent on the user role. If accessed by an administrator there are two icons, a pen that when clicked raises a editing dialog and a trash can that deletes the query; if accessed by an user, as seen on the figure, this column is not rendered. Create and edit dialogs are similar to those made for the Directory entity but the form fields match the attributes of query model.

QueryComponent, like the others, interacts with the listing template by handling the actions that can be done on this page. Clicking on the list element is bound to *on-QueryShow* which raises the QueryShowComponent dialog; the add button and the edit action raises QueryFormComponent dialog with different data, creating a new query passing no information to the dialog and editing passes the clicked query element. QueryFormComponent is smart enough to take different actions depending on its input so that a new query gets persisted and an already existing query it patched.

When a table element is clicked, the dialog seen in Figure 4.4 is shown. It presents to the user the query's Name, Description and Permission alongside two button on the lower right, "Run" and "Download". The former invoke an API method on **/run-Query/{id}** to execute it, retrieve the results and display them in a table similar to what is shown in Figure 4.5; the latter does the same but in the end prompts the user to save a Comma Separated Values (csv) file with the results.

Because queries must be filtered in regards the current user's permissions, QueryService requests the API on a custom controller located at **/queries** that does not comply to the HATEOAS pattern, inducing the creation of two models, one that extends the common Entity class to map eventual HATEOAS responses and the other that is on par with the response from **/queries**. It is interesting to note that UI elements are build around the latter model because it enables one single screen to serve both user roles, the conversion

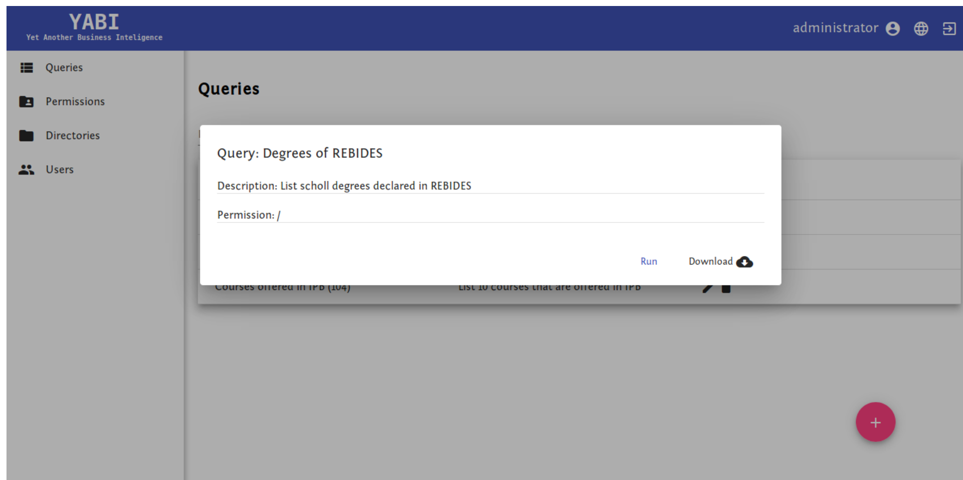

Figure 4.4: Dialog for running a `Query`

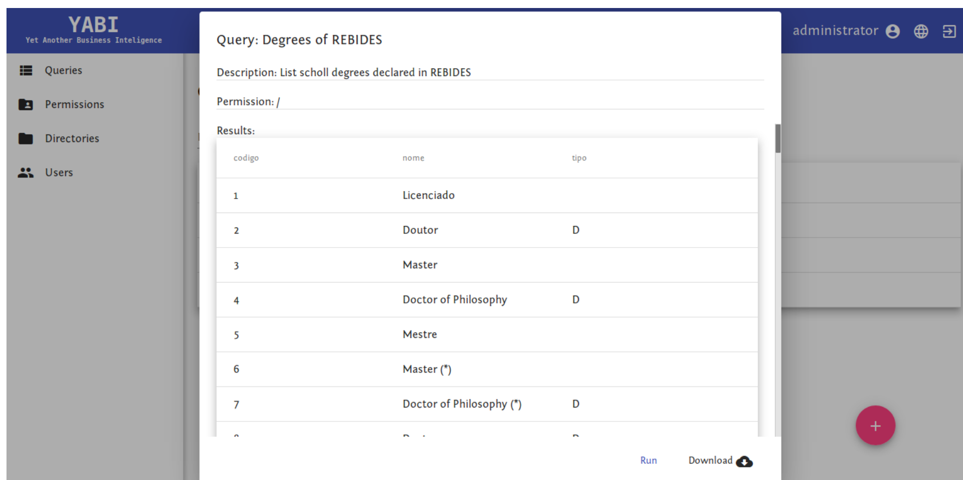

Figure 4.5: Dialog for running a `Query` after it was executed

from `HateoasQuery` to the non-hateoas version is straight forward given that `Query` is simpler with no nested attributes.

Permission

`PermissionService` have only two particular methods, one for requesting the current user's permissions called *userIndex* and a custom implementation of *delete* which uses Yabi's custom implementation in **/delete**.

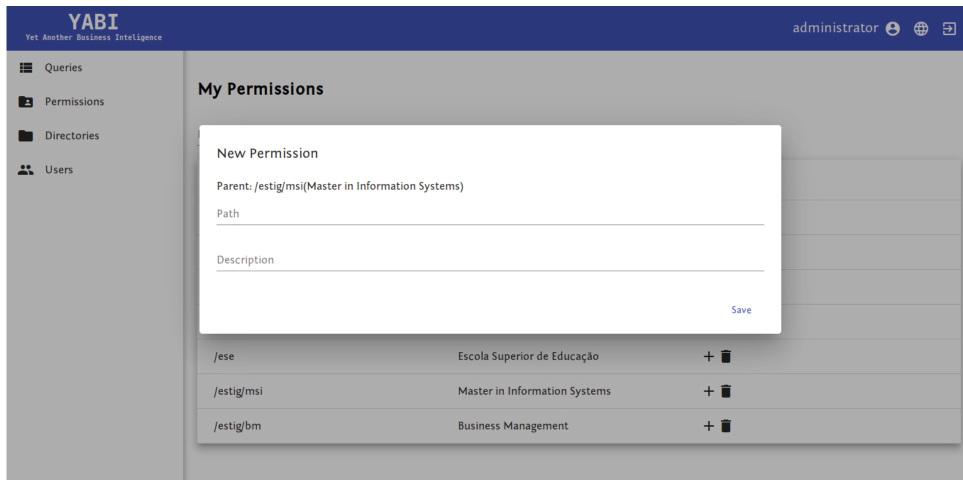

Figure 4.6: Dialog for creating a new `Permission`

The screen that lists the available permissions is also similar to that seen on Figure 4.2, although its table has different columns, `Path`, `Description` and `Actions`, with this last one visible only to administrators. The action column has two icons, a plus sign that indicates the creation of a new child permission and a trash can for deletion. Clicking the row raises a dialog for editing its description. The key difference between this entity and the others when it comes to creating a new entry is that because permissions follow a hierarchy, the first step towards creating a new entity is to choose its parent permission and clicking on the plus sign on its row. Figure 4.6 presents the dialog raised when doing so. Note that it shows the parent permission's canonical path alongside its description.

The main component is bound to listing the available permissions. Similar to `QueryComponent`, this page serves both user roles the same way but contrary to it, `HateoasPermission` is used as the base model. This is because `Permission` class implements the *toHateoas* method and even thought they share much of the same fields, the converted `HateoasPermission` is generated with an undefined URI attribute. There are two sub-components accessible form the main screen, `PermissionFormNewComponent` and `PermissionFormEditComponent`. As the name suggests, they interact with the user and API when creating and editing permissions. One interesting thing to note is that when persisting a new permission, the *onSubmit* method appends the separator character to the path.

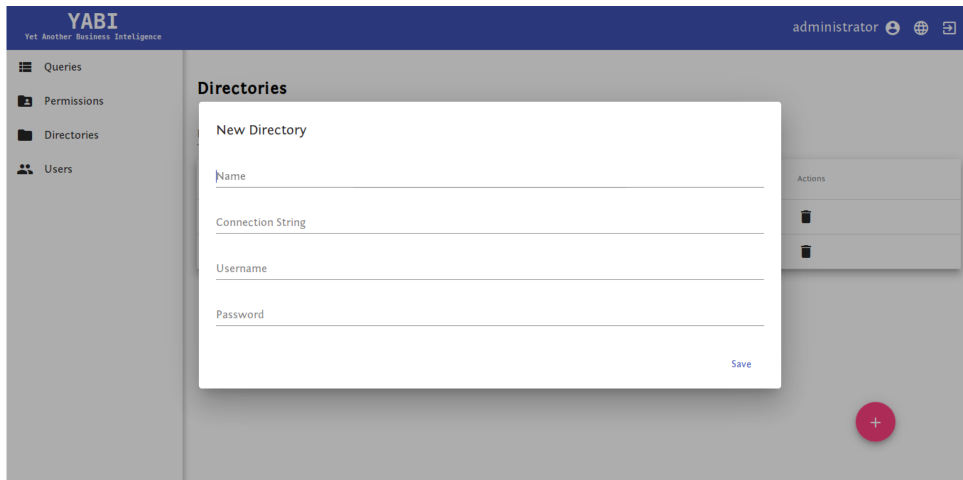

Figure 4.7: Dialog for creating a new `Directory`

Directory

Lastly there is the `DirectoryService` whose implementation is nothing more than the proper extension of `PagingAndSortingRepositoryService`. In fact, it is so small that it was used as the minimal usage example in Section 4.1.1.

The directory listing page is only accessible to administrative users, which made its implementation very compliant to HATEOAS pattern used by Spring Repository. It uses the same listing presentation shown in Figure 4.2 with columns for `Name`, `Connection String`, `Username`, `Password` and `Actions`, showing the icon of a trash can that signifies deletion. Clicking the row itself raises the `DirecotryFormEditComponent` in a dialog, allowing the administrator to edit the clicked Directory. Figure 4.7 shows the dialog used to create a new Directory, composed of one textual input per model attribute and a "Save" button on the lower right corner. The previously `DirecotryFormEditComponent` shown on Figure 4.8 share the same structure but with inputs filled with values from the Directory being edited.

The component implementation is very straightforward with the listing page fetching elements using the `DirectoryService` and binding the buttons to trigger their respective behavior. The plus button that indicates the creation of a new directory is bound to method *onDirectoryNew*, clicking the row to *onDirectoryShow* and the trash can to

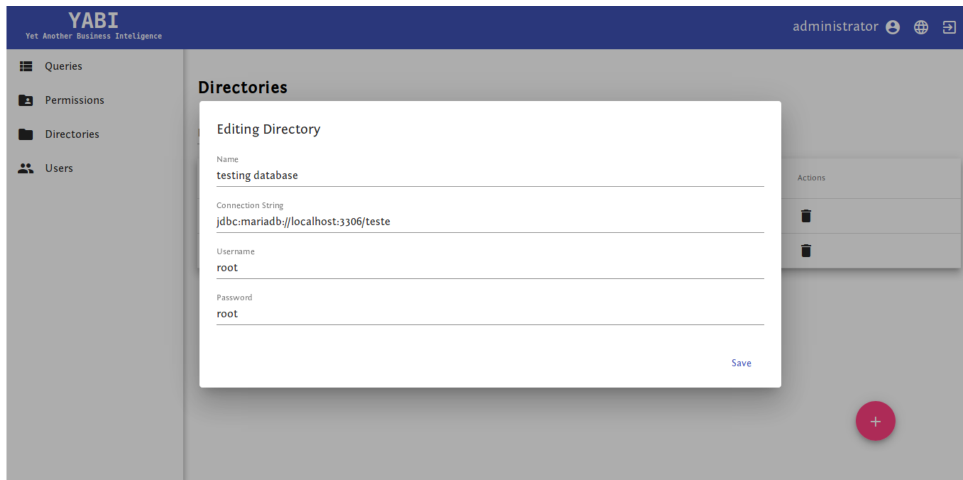

Figure 4.8: Dialog for editing an existing `Directory`

onDirectoryDelete. The first two delegate the interaction to other components that are displayed in a dialog.

Being accessible only to administrative users means that the whole API interaction can be done through Spring Repositories. This means that the only model being exchanged is that of `HateoasDirectory` that extends `Entity` and nothing more.

4.1.4 Generic Form Control Builder

Every form used for editing and creating new entries follow the pattern of providing a text input for each attribute of the model. To reduce repeated code for generating a `FormGroup` with the given model's attributes the function *genericFormControl* was implemented.

What it does is take an instance of any object and an optional list of attributes to ignore and assign to an object every attribute name evaluated from calling `Object.keys` on the instance with a new `FormControl` with that attribute's value, skipping the attributes found the ignore-list. In the end, it returns a new `FormGroup` with such object.

This way, when creating a form for editing a model, this function will return all the control with values already filled, ready to be bound to a form in the template definition. Also, when creating a new entry, the resulting `FormControl` edited by the user will have

matching fields with the model, allowing for fewer conversion between what is inputted and what is sent to the API.

4.1.5 Temporal Caching Repository

Because `PermissionService` is used by a few different classes and because permissions are not frequently inserted in the system, it was found that not many API requests are necessary during a session. Therefore a temporal caching system was developed on top of the existing `PagingAndSortingRepositoryService` so that in a configured time span, the service will not actually request the API but instead provide elements that are kept in memory. Once the time expires, the local cache is considered invalidated and a new API is issued.

Implementation-wise it intercepts the actual *index* method saving a local copy of the response in memory. When a request to *index* happens within the time frame specified in `SharedModule.serviceCacheExpirationTime`, it returns an observable of its local cache. The cache itself is implemented as an array of elements in which the service is providing.

4.1.6 Error Handler

By default, every Angular app provides a `ErrorHandler` that simply writes to the browser's console. However users must be notified when an error happens so that they can either take action or contact the IT department if it persists. To do so, `ErrorHandler` was extended into `SnackBarErrorHandler`, which as the name suggests creates a snack bar element with the error message, and was provided in the root module so that any uncaught error in the application is handle by it, effectively replacing the standard `ErrorHandler`. Figure 4.9 show the visual element that is created when an error occurs.

4.1.7 authenticationInterceptor

Because the back-end handle requests in a stateless fashion, authentication information must be sent alongside every API request.

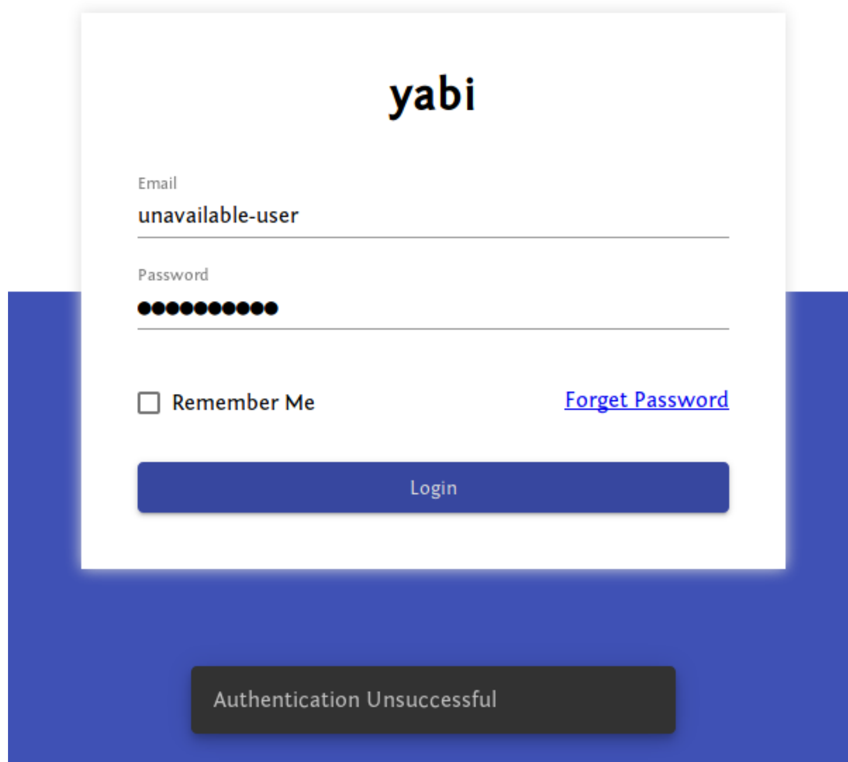

Figure 4.9: Login screen with a authentication error caught by the `SnackBarErrorHandler`

This was achieved by creating the `authenticationInterceptor` class that implements the `HttpInterceptor` interface and overrides the method *intercept* by injecting the `Authorization` header to requests that addressed to the API.

There are some occasions that require different or no treatment. If the request is not destined to Yabi's API, the request is processed as usual and if the user is not authenticated the application redirects to **/login**, prompting the user to login again.

4.1.8 apiEndpoint

Though as a Service, this class is declared in the app's root module and therefore can be injected into any component. It's job is to abstract all the endpoints available in the interface either through constant strings that point to repositories or functions that assemble an address for an entity given it's id.

Some example endpoints include the `PERMISSIONS` attribute that refers to the API's **/permissions** that return all the permissions associated to the current user; `ADMIN_PERMISSIONS` referring to **/permissionTrees**, which is the interface managed by a Spring Repository and lastly there is the *USER_PERMISSION* function that takes in an user id and a permission id and returns the address that represents the association between them so it can be, for example, deleted.

4.1.9 Shared Module

The `SharedModule` is a class that imports and exports all Angular Material directives. This was done to speed up development time, removing the necessity of importing each specific module when it is needed. The performance impact of including all of Angular Material was not found to be noticeable during development when compared to building it with Ahead of Time Compilation (AoT).

Another use of this module is providing application-wide constants. There are three static attributes, `debounceTime` which is the time in milliseconds between keystrokes to consider that the user has finished typing their search term; `serviceCacheExpirationTime`

which is the lifespan of the information kept in a `CachedPagingAndSortingRepositoryService` and lastly, `apiBase`, which is the base API address.

4.1.10 Security Concerns

When a user logs-in the application, their username and password is stored as a base64 string in the local storage. Should the application be compromised with attacks such as Cross Site Scripting (XSS), the injected code is able to access the user's credentials.

Another concern is in regards to the stateless nature of the back-end, requiring the credentials to be sent with every request. If the connection is not encrypted, credentials will be sent as plain base64 encoded string, which is easy to spot and decode. In the front-end this can be mitigated with HTTP Secure (HTTPS) encryption and in the general view a token could be exchanged to avoid having the credentials being frequently sent to the API.

4.2 Back-end

This section describes the server-side implementation of Yabi. Developed in Java using the Spring Framework, this program provides the functionalities that are harnessed in the front-end application. In other words, it implements the entities and functionalities elicited during the requirement analysis, providing them through a Web API.

Section 4.2.1 is a discourse more specifically on how the entities found during the design of this application were translated into Java code and the modifications that had to be done in order for this to happen. Being a fairly complete framework, Spring has most development use cases covered, however they must be configured in order to attend the specific needs of this application and thus Section 4.2.2 explain the main configurations that were necessary for the desired behaviors. Section 4.2.3 explains the need and implementation of custom endpoints and ViewModels that were used. In Section 4.2.4, the interaction with the local database and the HATEOAS API are explained. The solution for Requirement 1, to access the institution database, is discussed in Section 4.2.5.

4.2.1 Entities

Following the Entity-Relational diagram in Figure 3.4, classes were created and properly annotated with so that JPA is able to properly generate a relational model. Hence the `Entity` annotation is present in all classes.

Listing 4.3 presents the implementation of the Query model defined in Section 3.1. It servers as an overview into other model implementations as it shares much of the common features but also adds some of its own.

Lines 1, 2 and 5 are provided by the Lombok package as discussed in Section 2.3.3, instructing the creation of constructor and other common methods during compilation. All models make use of `@Data` and `@NoArgsConstructor` annotations and all but `PermissionTree` use `@AllArgsConstructor`.

In regards to ORM, `@Entity` annotation in line 3 is the entry-point through which JPA evaluates what classes are meant to be taken into account when building the relational model. In general, attributes don't need to be declared as they are correctly inferred but in some cases it is desired to configure the generated database, `@Column` annotation on line 11 changes the default behavior so that the length of the corresponding `varchar` field in the table is able to hold larger amounts of characters; The other models, `YabiUser` `PermissionTree` and `Directory` make extensive use of `@Column` to specify columns that shouldn't have repeated values.

Relation between entities are made though `@OneToOne`, `@ManyToOne` and `@ManyToMany` annotations. The first two represent single value association between entities, however, they represent different semantics and where the foreign key will be created. `@ManyToOne` indicates that the foreign key will stay in the table in which this model is mapped to, `@OneToOne` makes no distinction, leaving for the ORM back-end implementation to decide. Line 17 is declaring that more than one instance of `SqlQuery` may reference a single `Directory` and that `SqlQuery` will hold the foreign key to `Directory`. `@ManyToMany` represents a collection of associations, here used to associate `YabiUser` to `PermissionTree` so that many users can have many, overlapping, permissions. One possible parameter to

Listing 4.3: Implementation of SqlQuery class

```
1  @NoArgsConstructor
2  @AllArgsConstructor
3  @Entity
4  @Table(uniqueConstraints = @UniqueConstraint(columnNames = {"permission"
       , "name"}))
5  public @Data class SqlQuery {
6
7      @Id
8      @GeneratedValue
9      private Long id;
10
11     @Column(length = 2048)
12     private String command;
13
14     private String name;
15     private String description;
16
17     @ManyToOne
18     @JoinColumn(name = "directory_id", nullable = false)
19     private Directory directory;
20
21     @OneToOne
22     @JoinColumn(name = "permission", nullable = false)
23     private PermissionTree permission;
24
25 }
```

`@ManyToMany` is the `FetchType`, instructing the ORM engine to retrieve the associated entity when it is accessed or when its parent is retrieved.

`@JoinColumn` annotation is a general purpose configuration for relational fields, in lines 18 and 22 it's used to configure the name in which the column will be called and whether it can have no specified value.

The remaining entities follow a similar pattern in it's implementation.

4.2.2 Spring Configuration

In order for Spring Framework to stay out of the way as much as it can and allow developers to focus on the implementation of business functionalities, it makes many assumptions

about how its components interact, however, at some point the application being developed grow new requirements that conflict with Spring defaults. When this eventually happens, which was the case with Yabi, developers can override some of Spring's default behavior by re-implementing specific interfaces.

This section presents Spring configurations that took place so that Yabi is able to work as expected.

Security

Yabi's security model uses a directory server to authenticate and a relational database to load user roles and execute authorization checks. Because this is a stateless application, every request follows the steps shown in Figure 4.10 before being executed by the controllers.

Authentication is done through an anonymous bind to a LDAP server, implemented with Spring's LDAP authentication that configures the server address and the base Distinguished Name (dn).

In regards to authorization, there are two possible roles in which a user must be assigned to, either `ADMIN` or `USER`. With this, non-administrative resources are simply not dependent on the user's role therefore accessible to all users, meanwhile, administrative resources are explicitly marked to be accessible by users whose role is `ADMIN`.

Figure 4.10, presents in a general view the steps taken to authenticate and load user details. It is infeasible to model the whole of Spring Web and Spring Security as it is quite an extensive feature and it is out of the scope of this report, therefore it was abstracted into fewer elements, namely `WebSecurity` entity representing the objects that get built by the configuration shown in Listing 4.4, the *authenticate* message is abstracting Spring's authentication provider voting system, `LDAP AuthenticationManager` entity representing LDAP's `AuthenticationProvider` and *anonymous bind*, as the name implies is the authentication that takes place in the directory server.

The main point of Figure 4.10 is to show that once the bind takes place in the `LDAP AuthenticationManager`, the library requests an object that implements the interface

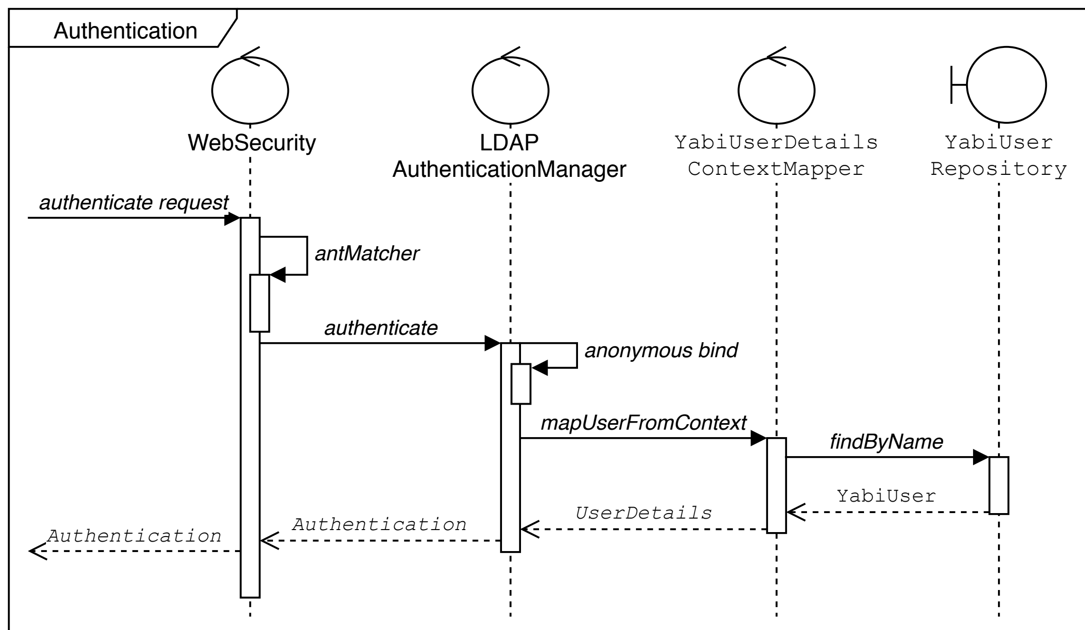

Figure 4.10: Authentication Sequence Diagram

`UserDetailsContextMapper` to populate it's newly created `Authentication` object with business-specific information in the form of `UserDetails`. In this case `YabiUserDetailsManager` is chosen by Spring's dependency injection mechanism and it provides an instance of `YabiUser`, that is retrieved from the database.

Admin Resources

Not all endpoints are freely accessed to all users because they involve possibly destructive interactions with the information contained in the system. In this implementation, non-administrative users get their information through custom `RestController` that expose fewer functionalities and administrative users may directly request the repositories.

Suffice to say that certain endpoints require the user to be authenticated and to have an administrator role. Listing 4.4 is the configuration that enforces this statement.

The call to `antMatchers` in lines 9, 11 and 13 works by specifying a list of HTTP paths and applying some definitions or restrictions whenever an incoming request is found to match a string pattern. In this specific case, all requests to repositories are eligible to continue being processed only if the `hasRole` rule evaluates to true.

Listing 4.4: HttpSecurity configuration

```
1     protected void configure(HttpSecurity http) throws Exception {
2         http
3                 .cors()
4                 .and()
5                 .csrf().disable()
6                 .httpBasic()
7                 .and()
8                 .authorizeRequests()
9                 .antMatchers("/user").permitAll()
10                // Spring Repositories
11                .antMatchers("/directories/**", "/yabiUsers/**", "/
                      permissionTrees/**", "/sqlQueries/**").hasRole("ADMIN
                      ")
12                // Custom Controllers
13                .antMatchers(HttpMethod.DELETE, "/permission/**").
                      hasRole("ADMIN")
14                .anyRequest().authenticated()
15                .and()
16                .logout().permitAll();
17    }
```

Line 13 is protecting the **/permission** endpoint from being requested with a HTTP DELETE verb. Lines 8 and 14 declare that all HTTP requests are to be authenticated.

Cross-Origin Resource Sharing (CORS) Mapping

Because Yabi is a web API and a front-end application, it is necessary that both parties can interact but due to security reasons, most browsers implementations block Asynchronous JavaScript and XML (AJAX) calls if the remote server does not explicitly state the allowed domains in it's response header.

In Spring, allowing domains to access it's resources is done by implementing the `WebMvcConfigurer`, overriding the method *addCorsMapping(CorsRegistry)* and calling *allowedOrigins* on it's parameter. The method *allowedOrigins* takes a list of strings that contain valid URL addresses.

Yabi configures this using the value of property `yabi.web.allowedOrigins`, allowing for a centralized configuration.

Listing 4.5: LDAP Authentication Configuration

```
1       @Autowired
2       public void configureGlobal(AuthenticationManagerBuilder auth,
            YabiUserDetailsContextMapper ap) throws Exception {
3           auth
4                   .ldapAuthentication()
5                   .userDetailsContextMapper(ap)
6                   .userDnPatterns(env.getProperty("yabi.ldap.
                        userDnPatterns"))
7                   .groupSearchBase(env.getProperty("yabi.ldap.
                        groupSearchBase"))
8                   .contextSource()
9                   .url(env.getProperty("yabi.ldap.url"));
10      }
```

LDAP

Following the authentication specification in Section 4.10, Listing 4.5 presents the configuration that implements the desired behavior.

In this configuration, `AuthenticationManagerBuilder` is a class used by Spring in it's security pipeline. It comes with built-in support for LDAP, JDBC and in-memory authentication mechanisms; line 4 is declaring LDAP authentication to be used.

Line 5 is considered important because it is mapping a custom details context mapper to the authentication pipeline. What this does is to provide a hook in the authentication pipeline to allow explicit customization of the user object after it is authenticated. The given mapper, `YabiUserDetailsContextMapper` retrieves the authenticated user's instance of `YabiUser`.

Lines 6 to 9 configure the connection to the remote directory service, including it's address and what to bind with.

User Details Context Mapper

Often times a directory service is used as an authentication mechanism. Applications then issue an anonymous bind request to the server passing their user's credentials and after, load its business-specific information according to the username. To do so, Spring Security

utilizes `UserDetailsContextMapper` interface to retrieve an `UserDetails` instance that gets injected into the commonly accessible `Authentication` interface.

For this application, a new implementation of the `UserDetailsContextMapper` interface is provided, `YabiUserDetailsContextMapper` returns an instance of `YabiUser`, which implements said `UserDetails` interface adding Yabi-specific attributes, enabling other parts of the system to query the current user's related information such as their role, `PermissionTree` and name.

4.2.3 Custom Controllers & View Models

`@RestController` is a Spring Web annotation that enables a given class or method to handle HTTP requests. In essence it is a combination of two other annotations, the `@Controller`, which is what trigger the framework into considering the class as a possible resolver of HTTP requests and `@ReponseBody`, that wraps the method call into a response body. In general cases the returned Object is mapped to a JSON string.

However the `PermissionTree` class contains a reference to it's parent, and the root references itself, there was a need to circumvent an infinite loop during it's JSON serialization. To do so, rather simple and serializable classes whose role were to convey information to the front-end were written, namely, `SqlQueryViewModel`, `PermissionTreeViewModel` and `YabiUserViewModel`.

To accommodate the special handling of `PermissionTree` model and provide custom functionalities, controllers were implemented for SqlQuery, PermissionTree and YabiUser entities. An additional controller, `DatabaseReaderController` was implemented to attend one of the functional requirements. When it comes to providing entities in function of the current user's permissions, the general implementation was done by adding a custom method to repositories that enables them to retrieve elements based on their entity's permission `nodePath`. Listing 4.6 show the implementation of endpoint **/queries** that retrieves `SqlQuery` elements that are associated to the current user. Note on Line 8

Listing 4.6: **/queries** endpoint implementation

```
1      @CrossOrigin
2      @GetMapping("/queries")
3      public List<SqlQueryViewModel> getQueries(Authentication auth){
4          YabiUser user = (YabiUser) auth.getPrincipal();
5          List<SqlQueryViewModel> queries = new ArrayList<>();
6
7          for ( PermissionTree permission : user.getPermissions() ){
8              for ( SqlQuery q : queryRepo.
                   findByPermissionNodePathStartingWith(permission.
                   getNodePath()) ){
9                   queries.add( new SqlQueryViewModel(q) );
10             }
11         }
12         return queries;
13     }
```

the retrieval of all elements whose permission `nodePath` starts with the user's permission `nodePath` and on Line 9 the instantiation of a non-recursive, serializable `ViewModel`.

DatabaseReaderController

The retrieval of information contained in a remote database is exposed through the **/runQuery/{id}** address. The interpolated `id` argument is a number that refers to a persisted `SqlQuery` class. When this controller is properly requested, it first validates by checking if the requesting user may run it by finding at least one permission associated to the current user that is either higher or equal to that in which the `SqlQuery` is associated to and calls `DatabaseReader.runQuery` with it.

The method *runQuery* in turn retrieves the associated `Directory` from the repository and with its credentials, it makes a JDBC connection and executes the query command. Lastly the resulting `ResultSet` is completely read and interpreted into a matrix of strings which is then returned to the controller and finally is used to answer the `API` request.

PermissionTreeController

Because of it's tree-like nature, once a `PremissionTree` is deleted, all of it's child nodes need also to be removed. However, because it references its parent but not its children,

leaving the RDBMS to execute a cascading delete would delete every parent permission untill the root node. Therefore `PermissionTreeController` implements a custom delete that cascades through its children. It is bound to a DELETE **/permission/{id}**, `id` being the identifier of the permission to be deleted.

It is implemented by first retrieving the permission whose id was specified in the HTTP request, retrieve all of its children with the custom repository method *findAllByNodePathStartingWith* and sequentially deleting all permissions found and then deleting the permission itself.

One restriction is that the root node can never be deleted, otherwise there wouldn't be any parent to insert new permissions. Therefore, before executing the before-mentioned steps, the permission to be deleted is matched with the root node, if it does, the operation is aborted with an error.

YabiUserController

To provide the front-end with information about the current user, `YabiUserController` replies to GET requests on the path **/user** with information provided inside the current `Authentication` that was generated during the user's authentication procedure and thus avoid reaching out to the database a second time.

SqlQueryController

`SqlQuery` is one of those entities that are related to the current logged-in user. In other words, administrators may see all registered `SqlQuery` and users see only those which they can run, therefore this customized behavior was implemented through a `SqlQueryController` that abstracts the Spring Repository interface.

It has only one method that replies to GET requests to **/queries** endpoint with a list of `SqlQueryViewModel`. Implementation-wise it returns a list containing every `SqlQuery` found in the database by calling `SqlQueryRepository.findAllByNodePathStartingWith` for every permission associated to the current `YabiUser`.

4.2.4 Spring Repositories

`PagingAndSortingRepository` were created for all entities evaluated during the project evaluation phase. The `Directory` entity whose implementation did not require any changes in regards to what Spring already provides won't be discussed. The remaining entities had their repositories augmented with functionalities presented below.

YabiUserRepository

After binding to the directory service, `UserDetailsContextMapper.mapUserFromContext` is used to fill an instance of `Authentication` class with business data. In Yabi's custom implementation, this data comes from a relational database that reflect `YabiUser` objects.

Because the method *mapUserFromContext* uses the username as a key to retrieve information, `YabiUserRepository` had to be augmented with a new method to do just that. Therefore the following declaration was added:

```
YabiUser findByName(String username);
```

PermissionTreeRepository

When the system needs to validate an action or filter information depending on a permission, it retrieves all of its child permissions. Because this action is frequently used, this repository had also to be augmented.

In this case, the method signature used was as follows:

```
List<PermissionTree> findAllByNodePathStartingWith(String nodePath);
```

SqlQueryRepository

When an user request a list of queries that he can execute, the system must retrieve from the relational database all queries in which the permission is a child of the user's permission. Again, this repository had to be augmented.

This was accomplished by using the following method interface:

```
List<SqlQuery> findByPermissionNodePathStartingWith( String nodePath );
```

4.2.5 Multi-Database Support

Paraphrasing Requirement 1, the application must be able to retrieve information from the institution's database; however, due to its many in-house applications, more that one RDBMS might be employed. Therefore Yabi cannot be constrained to a single database vendor for its information retrieval.

The core part of this feature is JDBC 4.0's `DriverManager` class and it's *getConnection* method that upon being called, attempts to make a connection using drivers that were loaded on initialization-time, therefore, as long as the driver is loaded and the connection string is properly formed, *getConnection* will select the correct database driver.

Because of this, implementing this feature was as simple as declaring dependencies for database drivers in the `pom.xml` configuration file. Notably, Oracle[1] requires the creation of an Oracle account and the configuration of a custom maven repository in order to download their drivers.

4.3 Development Environment

This section is focused on the work that was done indirectly to Yabi, but was still crucial to its development. Section 4.3.1 expresses how IPB's directory service was imitated with a local service. Section 4.3.2 briefly describes how local databases were deployed for use during development, Section 4.3.3 presents some configurations that can be done on Yabi's property file that aids during the production deployment and lastly, Section 4.3.4 presents some API tests that assess some of Yabi's restrictions.

4.3.1 Directory Service

When developing an application, it is good practice to avoid reaching out and interacting to remote services and instead provide a local instance that is able to mimic the real-world environment.

[1] https://www.oracle.com/technetwork/database/database-technologies/express-edition/downloads/index.html

Listing 4.7: Local server LDAP configuration

```
1  yabi.ldap.url=ldap://localhost:10389/dc=ipb,dc=pt
2  yabi.ldap.groupSearchBase=ou=groups
3  yabi.ldap.userDnPatterns=uid={0},ou=users
```

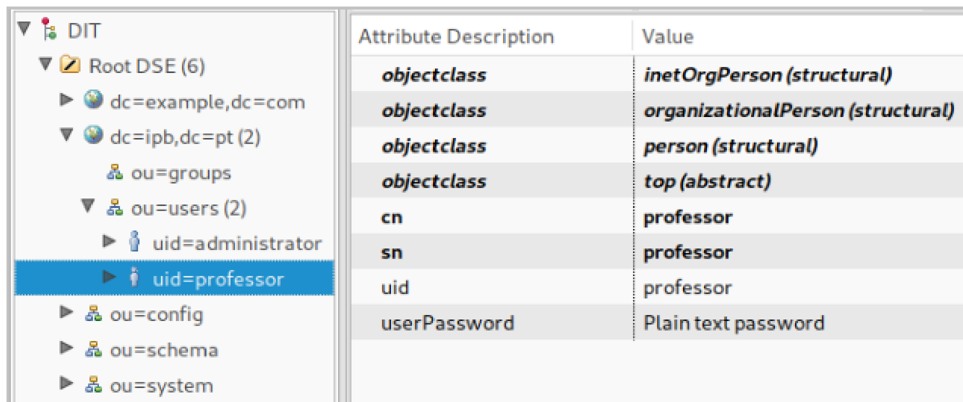

Figure 4.11: Directory structure and the properties of user `professor`

In this case because a directory service is used as a part of it's authentication mechanism, a local LDAP server was created in Apache Directory Studio that mimics IPB's directory service close enough so that the configuration provided by their IT team is able to be used locally with minimal changes, in fact, the only difference is the service's address.

Listing 4.7 exposes the configurations Yabi uses to access the server. Note that the directory address is declared in line 1 and line 3 declare the entry whose elements are anonymously bound.

Figure 4.11 show how the directory is structured in the server-side. Users are of class `inetOrgPerson`, they are held under the `users` organizational unit which in turn is under the `ipb`, `pt` domain component. At the moment this figure was taken, passwords were stored in plain text for local testing purposes.

4.3.2 Local Database

MariaDB was deployed locally with two main purposes: To serve a testing database in which Yabi could connect and run queries similarly to how it would do in production and at the same time, provide a database in which Yabi could store its own information.

For this, the official MariaDB Docker image was used[2] in conjunction with an empty database for Yabi and a testing database that was sourced from Portugal's Biographic Registry of Higher Education Instructors (REBIDES).

4.3.3 Database Initializer

When starting Yabi for the first time, it needs to generate its database, create the root permission and an administrator account. To do so, the class `DbInitializer` was written. It implements the `CommandLineRunner` interface so that Spring executes it just after initializing.

There are two properties that interact with `DbInitializer`, `yabi.db.init` and the `yabi.db.init.admin.username`. The former regulates when yabi should create a new database, the root permission and the administrator account, the latter declares the administrator's username to be used. If the initialization is desired, `yabi.db.init` should be set to `create`, otherwise, it can be set to anything else. It is necessary that the administrator username is able to be bound in the directory service otherwise the authentication will fail.

4.3.4 Postman Tests

When developing the API, some test cases were created in Postman to assess the prevention of data duplication. All four entities were tested. In essence, every test consists of two requests, one that creates a new entity and expects a HTTP status 201 response and the other that tries to re-create the same entity and expects a HTTP status 409.

[2]https://hub.docker.com/_/mariadb

It is important to note that these tests are validating the following restrictions imposed in each entity:

- There must be only one `PermissionTree` per `nodePath`.
- One `Directory` per `connectionString`, so that each database is referenced once.
- One `Directory` per `name`, so that each name maps to only one database.
- One `SqlQuery` under a `PermissionTree` with a given `name`, avoiding ambiguous `SqlQuery` entries.
- No more than one `YabiUser` with a given `name`.

4.4 Chapter Conclusion

This chapter discussed the implementation of all the parts that compose an application entitled Yabi. It has a Web front-end written in Angular that is meant to be accessed by both users and administrators and a Web API written in Java with Spring Framework.

The current implementation enables administrators to create database entries for Databases, Permissions and Queries and associate them. Users are validated using the institution's Directory Service and are brought into the system by being directly inserted in the database.

The following chapter will conclude what was done in this work and how it relates to the context in which it is meant to be introduced.

Chapter 5

Conclusion

With the growing adoption of digital processes by companies and institutions, the access to information becomes less available to the general public and more focused to experts in the field. These experts then end up mediating the interaction between those who are part of the decision-making process and the information storage.

Being in this situation, IPB's IT department is often interrupted form their daily tasks to handle the most diverse inquiry and questions about the institution's data base.

The present system is designed to aid everyone that takes part in this process to access information in an efficient and organized manner without the need for much technical knowledge, which in turn lowers the amount of daily interruptions in the IT department. It currently achieves this by providing a web interface in which the institution members can login, choose one of the many inquiries registered by the IT department and download its results.

In the end, some functionalities were let out of this implementation, with the most missed one is the ability for the users to tweak their inquiries to their specific needs. In spite of this the final system accomplishes the main task of providing users with the most used inquiries given that the IT department registers them. Not only providing the inquiries, it also supplies the necessary tools to manage users, permissions and remote databases, all through a web interface.

Chapter 6

Future Work

The development of this project yielded an application that suffices the core needs of what was proposed. Although considered good, it does not fully satisfy all the elicited requirements, leaving Requirements 7 and 8 outside of this implementation.

Therefore, this Chapter will be listing not only the missing requirements but also what to improve in a later version.

6.1 Code Re-structure

During the project development, some architectural mistakes were made. Even though they are not fatal errors, the technical debt was certainly increased. The two main points are in regards to Resource filtering, that is, what is an user authorized to access and the design of the `PermissionTree` data structure.

6.1.1 Resource Filtering

Currently, resource filtering is implemented by having a small set of controllers that are eligible to handle non-administrative users. However this led to some confusion because its response structure does not follow the HATEOAS convention, making the front-end

app implement two similar classes for each entity. One for HATEOAS responses and the other for the special case.

Some re-structuring could be done at the repository level by configuring the authorization mechanism to filter HTTP methods based on the current user's role and altering the front-end to always use Spring Repositories and always receiving HATEOAS responses.

Ultimately, all resources should follow the HATEOAS pattern so that the complexity of the front-end application is low.

6.1.2 PermissionTree's cyclic reference

In the back-end, the `PermissionTree` class is implemented with a reference to its parent. In the end this attribute did not contribute to anything useful to the architecture and instead induced the creation of custom controllers and ViewModels that brought inconsistencies in the angular application.

6.2 Bulk information manager

One of the requirements found in the proposal is the possibility to insert validated queries individually or in a group. The current system only supports inserting single, not validated queries.

6.3 Testing and User Validation

Unit testing was done during the early stages of development but unfortunately they required too many resources to execute. The multi-database support required all database instances to be running during the test execution. As the project development sped up, tests were ditched. Therefore the implementation of a proper test suit that validates this project's requirements is considered needed.

The introduction of the system in the proposed context was neither executed nor evaluated, leading to no conclusion whether it actually decreases the amount of times the

IT department is interrupted. Taking actual measurements before and after deployment would answer this question.

The developed UI's usability was not evaluated. If the interface is found to be confusing for many users, the IT department would be frequently interrupted for usability reasons, therefore tests should be done to assess its intuitiveness and ease-of-use.

6.4 Parameterization

The proposal states that this tool should enable SQL queries to be parameterized, which would allow the system to meet the specific needs of each user. Unfortunately this field was not developed at all and queries can only be run as they were inserted.

Appendix A

Proposta Original do Projeto

A.1 Proposta nº 2

Pretende-se desenhar e construir de raíz um sistema de "business inteligence" aplicado à gestão letiva. Sabemos bem o valor e a importância que a informação tem hoje em dia para quem gere instituições e as mais valias que as ferramentas de análise de dados trazem para a tomada de medidas e decisões. O IPB tem já uma base de dados centralizada à qual é aplicado diariamente um grande número de queries em SQL para os mais diversos fins. Pretende-se abrir o acesso a esta informação de forma criteriosa mas sem implicar a escrita manual de queries muitas delas com mais de 30 linhas de código.

O sistema seria pré-alimentado com "clusters" de queries mas teria uma característica evolutiva que daria a possibilidade de introduzir de forma fácil, suportada e validada novas queries ou novos grupos de queries, dependendo do perfil do utilizador.

As palavras chave serão a reutilização e a parametrização automática desses grupos de queries suportadas pela geração automática de interfaces web de pesquisa de informação, tendo por base o tipo de query a implementar.

Trata-se da disponibilização de um sistema de consulta inteligente no sentido em que se adapta às necessidades e ao perfil de cada utilizador. O resultado final serão sempre tabelas exportáveis para os mais variados formatos.