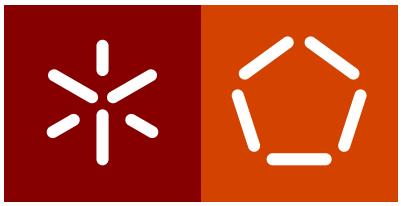


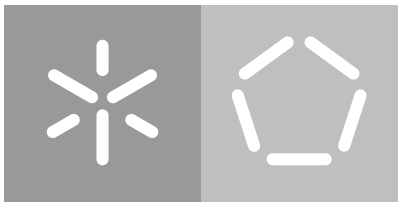
Universidade do Minho
Escola de Engenharia
Departamento de Informática

João Reis

GSD: A Web Application for Teacher Timetable Management

Dissertation Report

April 2020



Universidade do Minho
Escola de Engenharia
Departamento de Informática

João Reis

GSD: A Web Application for Teacher Timetable Management

Dissertation Report

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Pedro Rangel Henriques
Maria João Varanda

April 2020

AUTHOR COPYRIGHTS AND TERMS OF USAGE BY THIRD PARTIES

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

João Reis

ACKNOWLEDGEMENTS

It would not have been possible to write this master thesis without the help and support of the good people around me, to only some of whom it is possible to give a particular mention here.

To begin with, I'd like to thank my coordinators Prof. Pedro Rangel Henriques and Prof. Maria João Varanda for all the help, advice, knowledge and the availability they always offered me. I couldn't ask for a better guidance.

To my mother and my father, I'd like to thank them for all the unconditional help, sacrifice and trust invested in me through all these academic years; to my brother and my sisters for all the patience they surely had to have; to my cat, for keeping me company through all these endless working nights.

I also want to acknowledge all my friends for doing what friends do best. It wouldn't be fair to single somebody from this long list, but I'd like to mention those who kept company all the time through my classic foolishness.

Finally, to the beautiful city of Budapest and all the great friends I made there – thank you! You helped me live one of the best times of my life which I will never forget. A special mention to the Prof. Zoltan Porkolab for making this possible.

ABSTRACT

This document presents a Masters Thesis in Software Engineering, in the area of Academic Management, Support Tools and Web Software.

In a given higher education institution, the teaching service is assigned by it's department twice a year, following a model of their own self-re-creation.

Initially, it is intended to standardize the information in a transversal way to all the departments and create a model that will serve as the basis for the implementation of a web application.

This application will allow it's users to enter and verify information, generating database entries in a `DSL` of their choice (such as SQL), feeding a timetable construction platform.

RESUMO

Este documento tem como propósito apresentar uma tese do Mestrado em Engenharia Informática, na área de Gestão Académica, Ferramentas de Apoio e Software Web.

Numa determinada instituição de ensino superior, o serviço docente é atribuído por cada departamento aos docentes em funções, duas vezes por ano, seguindo um modelo da sua própria auto-recriação.

Pretende-se, numa primeira instância, normalizar a informação de forma transversal a todos os departamentos e criar um modelo que servirá como base para a implementação de uma plataforma web de especificação da distribuição de serviço docente.

A aplicação permitirá aos seus utilizadores inserir e verificar informação, gerando entradas para uma base de dados usando uma DSL (Domain-Specific Language, ou, em Português, Linguagem de Domínio Específico) da sua escolha (como SQL). As tabelas geradas servirão de base a plataformas de construção de horários.

CONTENTS

1

INTRODUCTION

The Instituto Politécnico de Bragança (IPB) is a public institution of higher education located in north Portugal, born in 1983. Having multiple departments, IPB offers its students several fields of study such as Agrarian, Education, Technology and Management, Communication, Administration, Tourism and Health.

At the beginning of every semester, the head of each department has to fill out a matrix assigning their teachers to their courses. This includes a lot of components which makes it very complex. Once it's finished it is sent to the responsible of timetable making, which, manually, inserts data into a software which works as a solver.

Another aspect of timetabling is concealing teacher's requirements to their chosen free time - such as their days off or schedule restrictions - that is also manually inserted into the timetable maker. Teachers send an e-mail with their requests and the responsible manually inserts it, possibly leading to failures due to incompatibilities.

From what is mentioned in the last two paragraphs, easily the problematic is understood. To begin with, the lack of modernization - taking into consideration what it's currently possible - leads to a substantial time waste. Putting it in perspective, every department sends their own version of their matrix - there's no data normalization, which causes misinterpretations and ambiguities. Sending e-mails through a trial-and-error process is also very inefficient. Gathering all these issues, the responsible of timetable making is engaged in that task in a scale of days, when it can easily be vastly reduced.

This masters work aims not only to develop a solution for this particular situation, but something broad and generic enough to be used in any institution, focusing on simplicity and time-reducing - by building a software by the name of `GSD` - initials for *Gestor de Serviço Docente*, or, in English, *Teaching Service Manager*.

1.1 OBJECTIVES

Overall, the main purpose of this master thesis is to streamline the process of data gathering from all the departments across a given academic institution. This whole process can be broken down in the following steps:

- Normalize and store all data that goes through timetable making;
- Create a platform for institutions collect data from their departments, given their restrictions;
- Give the platform the possibility to automatically remove, refuse or warn about incompatible data;
- Generate a DSL specification as output;
- Build a responsive and dynamic web application as front-end;

It is intended to have a functional and usable product for Instituto Politécnico de Bragança, but generic enough to be used in any academic institution. Considering that the requirements differ between the institutions there are, the solution for this problem undergoes by giving the system administrator the possibility to choose which fields are useful to them. This way, it can provide wide range of choices without depriving the use needed from each institution.

1.2 RESEARCH HYPOTESHIS

As exposed earlier in the document, the current methodology used to gather the assorted data from each department and teacher takes an significant amount of time that could be done in a most effective manner.

The current process goes through a system of trial-and-error, without any normalization of the data circulating. Teachers manifest their desired work-times by email, once again without any normalization, taking a lot of setbacks. Through this whole process, we're looking at an abundant amount of work-hours - therefore resulting in multiple working-days.

Having this in consideration, this master work hypothesis goes through the possibility of systematizing timetabling data gathering of multiple departments in order to make the whole process more effective. Aiming to reduce the amount of hours for every party involved, it will impact with a significant percentage of reduced work-hours.

1.3 DOCUMENT STRUCTURE

This document is split in 6 main chapters - the *introduction*, *state of the art*, *proposed approach*, *development*, *tests and results* and *conclusion*.

It begins by exposing the problematic in the current chapter - `INTRODUCTION` - where it is described the points at issue and the intended objectives, uncovering the research hypothesis. Afterwards, its disclosed current solutions and documents useful to solve the problematic in question, in the chapter `CURRENT TIMETABLING PROBLEMS AND SOLUTIONS`. Applying the concepts from the former chapter, in the `PROPOSED APPROACH` and using an appropriate template, it is documented the whole system architecture while going through the extended process of requirements study, constraints, system context and solution strategy. The chapter `GSD: DEVELOPMENT` goes into detail about the development circumstances, problems and decisions taken, following by `GSD: TESTS AND RESULTS` where tests and conclusions are taken about the work. Finally, in the `CONCLUSION` will be presented a summary of the work done, as well as the future work.

2

CURRENT TIMETABLING PROBLEMS AND SOLUTIONS

Timetabling is a crucial aspect of each semester planning. In the average academic institution, there are multiple departments containing multiple teachers and degrees, and each degree containing a set of subjects. Theoretically, the problem seems quite easy, and possibly solvable with simple forms. When it comes to practical terms it's not that trivial, mainly due to the system's shortcuts to aid everyone involved - for example, a *Mathematics* department organizes a *Mathematical Analysis* subject and for logistical reasons, a Bio-engineering and a Biology degree merge them in one lesson. But what if only the theoretical parts were merged together? What if the practical parts were merged with some other degree? A lot of *whats-ifs* will surge, because do they happen. Another good example is teacher distribution per shifts. As usual, a lesson can have multiple shifts to accommodate more students, but these shifts can be taught by different teachers. Keeping in mind that the former given examples only represent the superficial depth of the problem, the complexity rises from what it actually seems.

In this chapter it'll be approached current solutions and studies in the field of timetabling. Given that these solutions weren't used or the studies implemented, they'll be profiled and subsequently tackled their incompatibilities to the requirements to the subject of this master thesis.

Nonetheless, they're quite influential in this stage of development as they can offer answers to common problems and access ideas that weren't considered. For the reasons mentioned, each own will be approached in dedicated section.

2.1 REDOSPLAT

A very interesting and complete article about timetabling requirements is approached by Ribić et al. in *REDOSPLAT: A Readable Domain-Specific Language for Timetabling Requirements Definition*. This paper presents a domain-specific language called REDOSPLAT and its design methodology overview, requirements, syntax and semantic [16]. Although it won't be used in the final platform, it surely provides great detail in the required data for domain studying.

Aside from the establishing relations between entities and its attributes, it also contributed by its analysis for constrains which are branched in two - hard constraints and soft constraints. We'll be focusing on the soft constraints, since hard constraints have to be fulfilled in order to have a valid solution and these apply mostly to timetabling solving. However, some of them will also be taken in account since it can apply to both.

- **Resource utilization**
  When rooms are being requested they can made in distinguished ways. To begin with, a teacher can't - or shouldn't - book a room with a capacity inferior to the number of students attending that class.

  Rooms can also booked according by its type e.g., some type of practical classes require a lab to be taught. For this reason, room allocation can be monitored.

- **Workloads**
  When it comes to workloads, a few aspects of it can be introduced. Generally, in a regular work day there's a limit of hours in which a student can attend to. Teachers may also have a minimum or maximum workload defined, as a department having a established class regime.

Despite offering some key aspects of the platform and introducing interesting concepts, this solution takes into account a lot of solver algorithms and problem work, which, it's redundant and unnecessary.

2.2 School Timetabling in Theory and Practice

Written by van Heuven van Staereling, this master thesis was found while researching the timetabling problem. Although not having a software or requirements approach, it still does offer a very rich theory/practice about timetabling. Since it does take in account timetabling solving, we can only apprehend a handful of restrictions helpful for the project in mind:

- **Merging lessons in rooms**
  As previously mentioned, merging lessons is acceptable and efficient and thus desirable. These situations happen regularly, allowing to be efficient in terms of space and staff costs (considering the staff needs).

- **Teachers Unavailabilities**
  Another important aspect of the final product is the giving the possibility to teachers decide their work-times, such as days off or preferred working days, including preferred working hours (or off-hours, analogously). Their choices must be enforced into the institutions rules, e.g., it wouldn't make sense to choose to work from 1AM to 4AM when an institution schedule is between 8AM from 8PM.

2.3 CURRENT SOLUTIONS

Taking a closer look to the current solutions existent, most of them share a common problem - they're all designed having in mind the American Academic System which it's not fitting for the stakeholder responsible for the original software proposal.

- **UniTime.org**

  UniTime is a comprehensive educational scheduling system that supports developing course and exam timetables, managing changes to these timetables, sharing rooms with other events, and scheduling students to individual classes [20]. At a first glance seems like a powerful platform but exactly for that reason some problems arise - way to many redundant layers and features. The main structure of the software aims to cover all fields of an institution, such as teachers, students, exams, timetabling solver, credits system, room reservation, events and even administration. Their structure is as shown in figure 1.

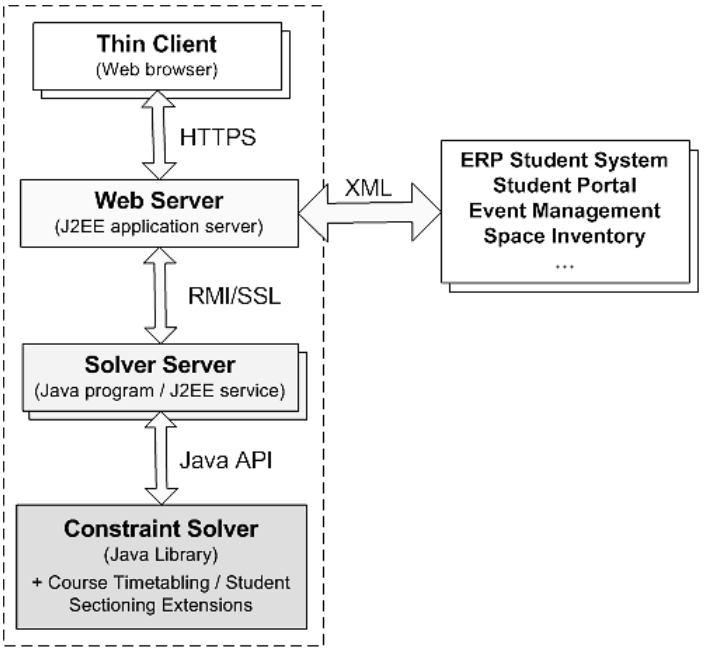


Figure 1: UniTime architecture

  This master thesis architecture would only need the first two laters - *Thin Client* and *Web Server* - the rest would redundant and resource wasting. Not only this, but it is optimized for the American system (making it incompatible with the Portuguese one) and it doesn't allow reading existent databases and export their own.

- **CourseLeaf Section Scheduler**

With Section Scheduler by CourseLeaf, staff can coordinate across multiple departments and offerings to build a schedule of classes that is consistent across campus [5]. This solution comes closer to what is intended, claiming also a 90 percent decrease in data entry by their registrar's office in an institution - one of the main objectives.

Many of its highlights are interesting and desirable such as visual representation and a interactive interface, giving the power to centralize the control of timetabling.

Then again, it's highly aimed for the American system and incompatible with the Portuguese one - which can easily be verified checking the institutions list, containing a high percentage of American or other country institutions using the same system. It also doesn't allow importing and/or exporting databases and revolves around timetabling solving or constructing which it isn't desirable.

- **Schedule My Teachers**

  Having the same flaws as the former ones - American system aiming, lack of data baste export and imports - Schedule My Teachers is also developed having in mind schools and not the high education problematic [18].

3

PROPOSED APPROACH

When it comes to software development, documentation constitutes one of the key aspects of it. Making a valuable documentation goes through a long process of numerous steps and components, such as requirements and the different types of modeling, which makes it a complex task. It's a common practice to follow already established and reputable templates such as arc42 - which is what's going to be done in order to achieve it.

Following this methodology will result in a final satisfactory product to each party involved, given it's quality architecture and flexibility to changes. In this chapter, through the next sections, will be presented the complete arc42 [2] template to this project.

3.1 INTRODUCTION AND GOALS

3.1.1 *Requirements Overview*

Through this document it has been discussed our objectives and main features without going in further detail or putting it in a formal manner. In this section we'll set it clear, branching them into two - functional and non-functional.

Requirements elicitation and subsequent writing is essential and should be done carefully to ensure every party involved understands it without ambiguities. For that same reason this document follows the procedure present in the book written by Fernandes and Machado. Requirements source arise mainly from meetings with the project's co-supervisor, given one's field specialization - although not exclusively, as the previous chapter exposes. As regards to it's writing, requirements should be written in a standardized way.

Not every requirement has the same importance when it comes to implement it. Some of them are more urgent than others and that's why the MoSCoW method [22] was considered as seen in the book written by Wiegers and Beatty. This method was used as a technique to reach a consensus between the stakeholders involved when it comes to establish the importance which they put into the delivery of each requirement.

The application will initially be developed in order to try to deliver all the requirements such as Must, Should, and Could, but Should requirements will be developed to the detriment of Could, if the delivery schedule is threatened.

To catalog the requirements, we'll them as following:

M Defines a requirement that must be satisfied in order for the final solution to be acceptable.

S Defines a requirement that must be satisfied in order for the final solution to be acceptable. This is a high risk requirement that should be included if possible within the delivery time.

C This is a desirable requirement that would be good to have if there is time and if the resources allow it. The solution must be accepted if the functionality is not included.

W This category represents requirements that stakeholders want to have, but agree that the requirement most likely won't be implemented in the current version of application.

Having all this in mind, as functional requirements, we have:

F.1 M The timetabling responsible should register the institution in the platform;

F.2 M The timetabling responsible should add institution staff to platform after registering it (**F.1**);

F.3 M The timetabling responsible can add institution teachers to the platform;

F.4 M The timetabling responsible should add departments to the platform after registering it;

F.5 M The timetabling responsible should assign a head of department for each department;

F.6 S The timetabling responsible can add institution restrictions;

F.7 M The timetabling responsible should add teachers to its department;

F.8 S The head of department should add teachers to its department;

F.9 M The timetabling responsible should add degrees;

F.10 M The timetabling responsible should add courses to a department;

F.11 M The timetabling responsible should add classes to a course;

F.12 M The head of department should assign shifts to each class;

F.13 M The head of department should assign shifts to one or more degrees;

F.14 M The head of department should assign teachers to each shift;

F.15 C The teacher can add desired days off;

F.16 C The teacher can add desired working days;

F.17 C The teacher can add desired working hours;

F.18 C The teacher can add desired non-working hours;

F.19 C The staff should add institution buildings to the platform;

F.20 C The staff should assign rooms to each building;

F.21 S The staff can add institution restrictions;

F.22 C The system can warn about restrictions not being followed;

F.23 C The system can enforce restrictions;

F.24 M The timetabling responsible should assign which components are required;

F.25 M The timetabling responsible can load a pre-existent database file;

F.26 M The system should interpret the database loaded by the timetabling responsible;

F.27 M The system should make a visual representation of a database loaded by the timetabling responsible;

F.28 M The timetabling responsible should assign each database entry;

F.29 M The timetabling responsible should assign which output is generated;

When it comes to non-functional requirements, they're split as following:

NF.1 M The system should be responsive;

NF.2 S The system interface should be simplistic;

NF.3 C The system's features should be done in few steps;

3.1.2 *Quality Goals*

In the next time are represented the main quality goals of the software.

#	Quality Goal	Scenario
1	Simplicity	The product should be simple enough to any user understand how it works at first sight
2	Efficiency	The product should be fast and dynamic, without long loading times;
3	Attractiveness	The product should be attractive to the general masses;
4	Interoperability	The product should be able to work with other software; mainly with schedule solvers.

3.1.3 *Stakeholders*

Stakeholder	Context
João Reis	Being the author of the thesis, aims to successfully implement what is proposed and do it in high quality standards;
Pedro Rangel Henriques	The thesis supervisor intends to guide the thesis author to successfully accomplish his goals and have every party involved satisfied
Maria João Varanda	Representing Instituto Politécnico de Bragança interests and being this thesis co-supervisor, aspires to have a valuable and working software to support the making of timetables for the given institution
Teachers	Teachers who want a simple and interactive way to expose their preferences when it comes to working time
Timetabling Managers	Every institution or department has someone responsible to deal with timetabling. They'd be interested in saving time with paperwork and communicative bureaucracy.

3.2 CONSTRAINTS

The few constraints on this project are reflected in the final solution. This section shows them and if applicable, their motivation.

3.2.1 *Technical Constraints*

	Constraint	Context
1	OS independent development	The final product should be able to run on any operating system - Windows, Mac and Linux distributions
2	Web Application	The software should be running exclusively on any modern browser
3	Deployable to a Linux server	The application should be deployable through standard means on a Linux based server

3.2.2 *Organizational Constraints*

	Constraint	Context
1	Team	João Reis
2	Time Schedule	Started at the first semester of 2018/19 scholar calendar, it is planned to have a finished and tested version by 2 months before the thesis delivery deadline.
3	Version control/management	As the software is being worked on, it should be pusged to a git private repository with a complete commit history;
4	Testing	The final version should have been subjected to rigorous testing

3.2.3 *Conventions*

	Convention	Context
1	Documentation Language	The documentation should be written in english
2	Software Language	The software should be written in Portuguese

3.3 SYSTEM SCOPE AND CONTEXT

Before implementing software or even before modeling it, it's important to have a full understanding of the domain it's being worked on. A popular way to achieve it is by

building a domain model - making it possible to reach a consensus between all stakeholders involved since it's quite a simple approach which everyone can understand. For this reason, a domain model was built as it shows in the figure 2 on the next page.

Analyzing the model, it seems more complex than it actually is. The easiest way to understand it is to break it down into simple steps.

Starting with highest entity, we have `Institution`. It represents the academic institution that will be using the product.

- In a given `Institution`, there are one or more `Department`;
- An `Institution` can impose multiple `Restriction`, which should be abided across all departments and courses;
- In the institution there are multiple `Degree`. By `Degree`, it's intended to designate a degree program you can take when joining university, e.g., a law degree;
- It's also possible to map the every `Building` that the `Institution` has.

Down the hierarchy, there's the `Department`.

- The `Department` will also organize multiple `Course` which multiple `Degrees` can attend to. Putting in perspective, let's assume a Mathematical Analysis course - multiple degree programs can include that course;
- In a `Department` there are multiple teachers which are responsible for;
- Due to exclusive `Restriction` to each `Department`, each own can impose theirs.

It's relevant to explain the role of the described entities that are responsible for imposing restrictions. By its name, it's easily accessed its function, e.g., there can be restrictions for number of hours of classes a student can attend per day, the schedule of a class regime, between many others.

It's possible to map every institution building using the entities `Building` and `Classroom`. An `Institution` has multiple `Building`, each one having multiple `Classroom` too.

Going back to the `Degree`, it's intended to define multiple degrees that are offered by the institution, e.g., a law degree, as it was said said before.

- Every degree has to abide to rules imposed either by the `Institution` or `Department`;
- Obviously, along the course of a `Degree`, there are multiple `Subject`.

As we said before, each `Degree` has multiple `Subject`, and each Subject has multiple `Class`. The entity `Class` makes part of the core entities, due its relevance in the whole process.

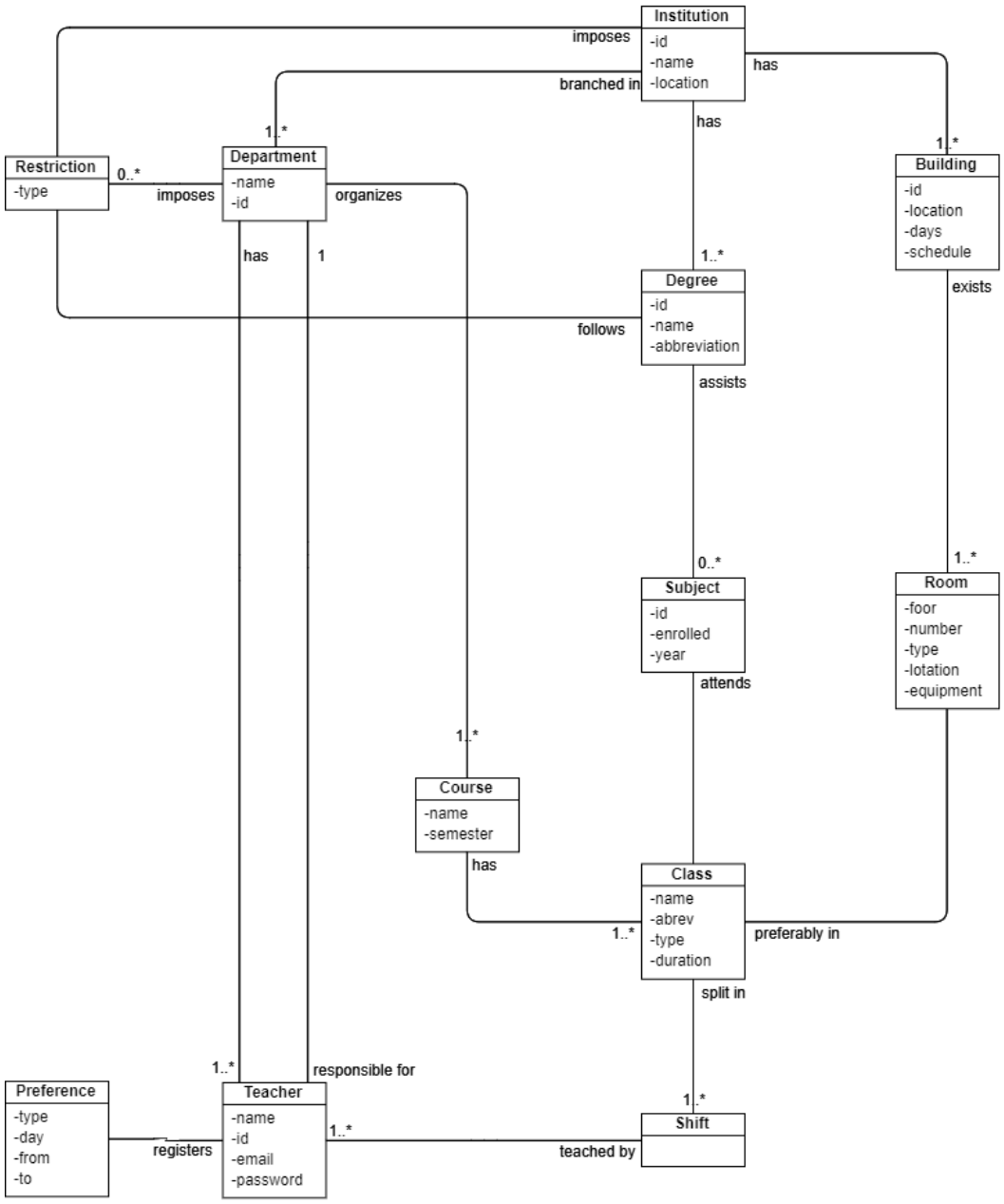


Figure 2: Domain model.

- Each `Class` can be taught in different `Shifts`. This can be due to multiple reasons, such as high number of attendees or just giving the various opportunities to everyone to attend.
- Each `Class` is related to a `Course`. It's important to make it clear that a `Course` can be attended by different degrees.
- It's also possible to make it possible to select a preferred `Classroom` given the necessities a `Class` can have.

A `Class Shift` can be taught by one or more `Teachers`.

- `Teachers` can register their preferences when it comes to its free time. This means, they can choose between by working or not working in a given interval.
- A teacher can also be responsible for a department and for defining the teaching service of that department.

In order to have a full understanding of the system and the features available for its users, the use case diagram as seen in the figure 3 was built - a simple representation of a user interaction with the system.

The system users' are split in two - the administrator and the head of department. To get in detail with the former, there are six sections for it: *Department*, *Degree*, *Course*, *Data*, *Association* and *Configuration*.

The *Department* section contains two use-cases:

- **Department management**: The user gets a list of departments currently in the system, having the possibility to consult each one's information or delete it, if requested.
- **Upload department database**: The user can a upload file containing a database of departments for the system to load and insert in batch;

The second section is *Degree*, containing two use-cases:

- **Degree Management**:The user gets a list of degree currently in the system, having the possibility to add a new one, edit or delete one degree previously inserted.
- **Upload degree database**: The user can upload a file containing a database of degrees for the system to load and insert in batch;

As for the third section, **Course** there's one use-case:

- **Course Management**: The user gets a list of courses currently currently in the system, having the possibility to add a new one, edit or delete one course previously inserted.

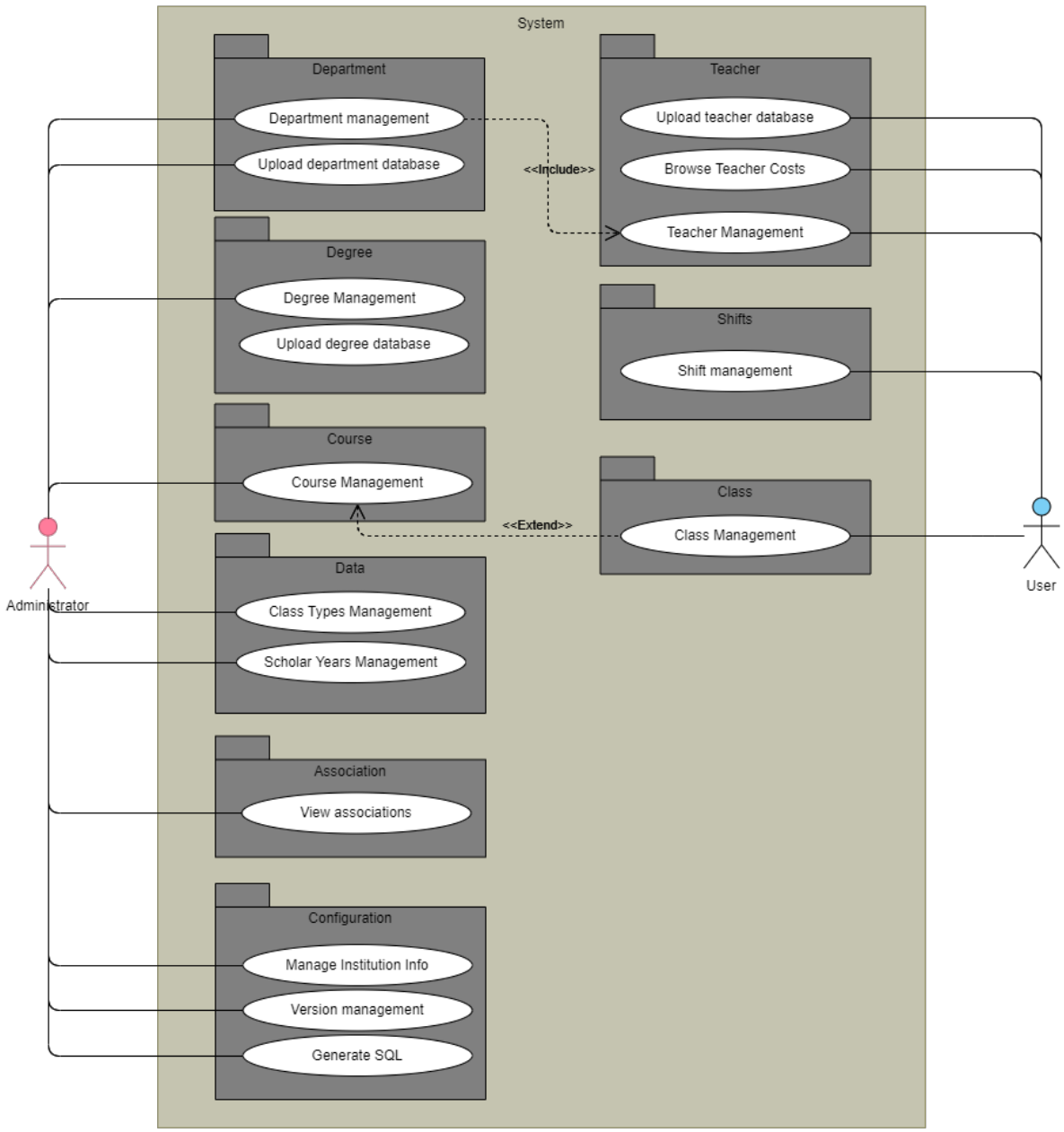


Figure 3: System Use-Case diagram

- **Upload courses database**: The user can upload a file containing a database of courses for the system to load and insert in batch;

As for **Data**, that makes the fourth use case there are three use cases:

- **Class Types Management**: The types of class that each course can have are predefined, and this is the section to manage it. The user gets a list of class types currently in the system, having the possibility to add a new one, edit or delete one course previously inserted.
- **Upload class type database**: The user can upload file containing a database of class types for the system to load and insert in batch;
- **Manage Scholar Years**: Being also predefined, scholar years are the different stages that a degree has. The user gets a list of scholar years currently in the system, having the possibility to add a new one, edit or delete one course previously inserted.

The third section, *Association*, has a use-case:

- The user can get a better of understanding of the associations done after inserting courses;

Lastly, the section *Configuration*, has three use-cases:

- **Manage Institution Info** The user can update manage institution, as the name that shows up along the system.
- **Version management** Each year there will be a new version of data that is the result of usage of the system. However, some of the data will remain the same along the versions, and versioning allows switch the data along each year. The user can add a new, change and delete a given version.
- **Generate SQL** The user can get a file that is the output of the system usage.

When it comes to use-cases for the department responsible, there are three sections of it - *Teacher*, *Classes* and *Shifts*.

The section *Teacher* has three use-cases:

- **Teacher Management**: The user gets a list of teachers currently currently in the system, having the possibility to add a new one, edit or delete one teacher previously inserted.
- **Upload teacher database**: The user can upload a file containing a database of teachers for the system to load and insert in batch;
- **Browse teacher costs**: For each teacher in the department, the user can view the number of weekly hours assigned, as well as the shifts;

The second section, *Shifts*, has an use-case:

- **Shift management**: The user can manage the shifts for a given course, creating new ones, editing or deleting.

The third and section, *Shifts*, has an use-case:

- **Class management**: The user can manage the classes for a given course, creating new ones, editing or deleting.

3.4 SOLUTION STRATEGY

Before drawing a strategy for the system, it's important to understand the architecture on high level and its flow. The diagram as shown in the figure 4 on the following page intends to represent the system's architecture on a high level view.

Breaking the architecture flow in parts, there are three of them that easily arise when analyzed the exposed diagram:

- The **data validation** part which as it suggests, validates the data infused into the system. There are two types of data that can be inserted. The data that usually exists in a given institution which concerns its departments, teachers and general institution information, thus making it possible to load this data through a file such as `CSV`. Alternatively, that same data can be inserted manually.

  On the other hand, the data that the system intends to gather and work on. These are information respective to *teacher service assignment*, *teacher preferences*, *class assignment*, *restrictions* and similar.

- The **output generator**. In this part, given the data filled in the previous section - and respectively normalized - the main user will chose the fields necessary to posteriorly generate it, in a database of his choice.

- Lastly, the **Timetable solver** part which, given the amount of solutions already existent, will not be considered into development. The idea is make the system interoperable with software that allows solving timetabling, i.e., giving the possibility to export multiple types of databases.

Taking into account the development architecture, and considering that system will be based on a web-service application, it'll rely on a usual front-end/back-end architecture, as it follows:

- **Front-end** is the component responsible for the interaction between the user and the system. Running a browser, this part will both represent and gather from its users,

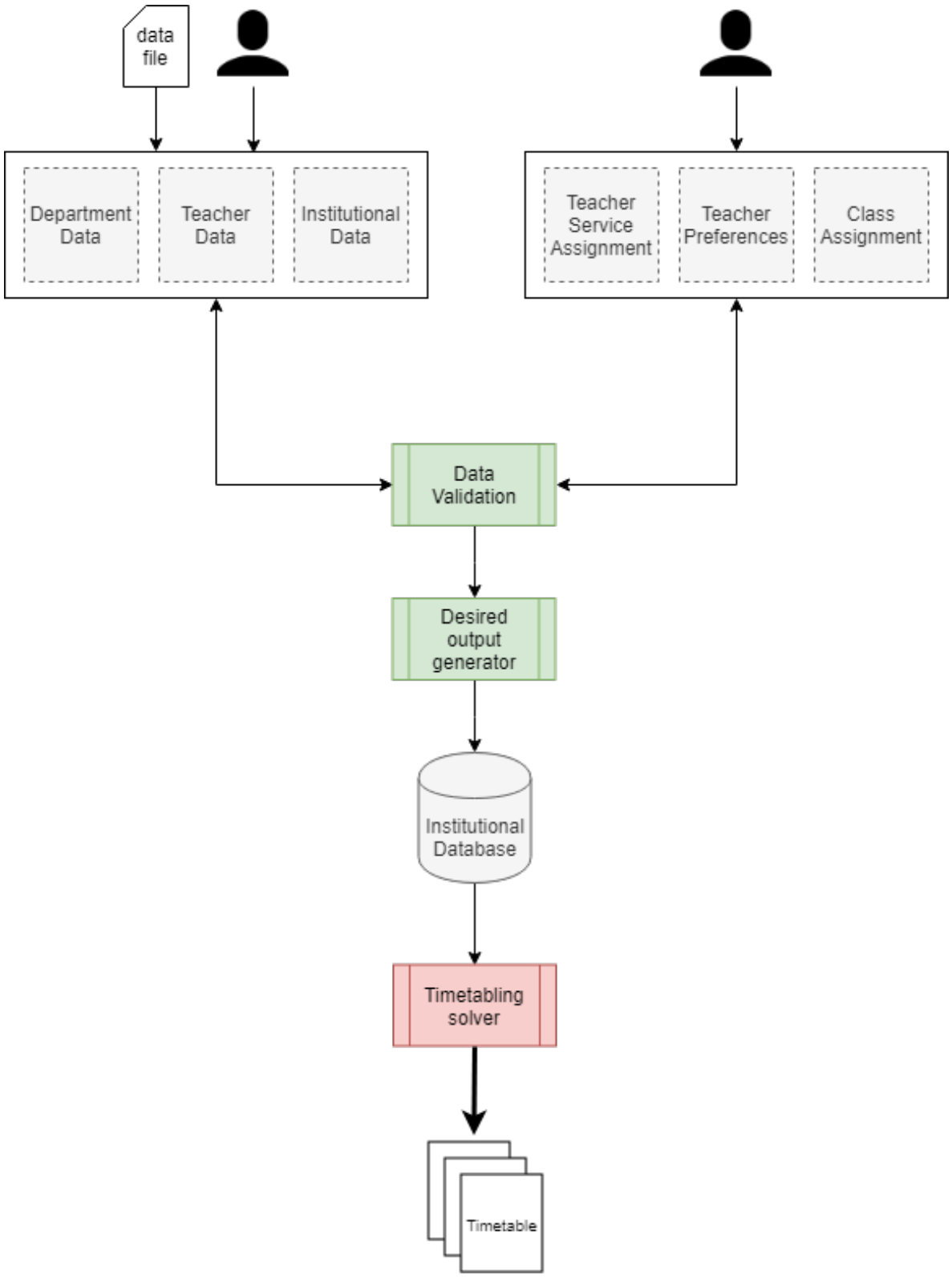


Figure 4: High level architecture view

either from a existing database file or from a simple text input. That same data is moved through a channel that connects it to the back-end.

In this section, the user will be able to do all of its requisites - but the processing will be done in the following back-end.

- The **back-end** will be operating on a server to provide availability as the users demand. On this part of the system, once the data channel reaches with multiple queries from its users, such as `Request Processing`, `Authentication`, `Data Validation` and `Output Generator`, it'll process them, access the database if needed and correspondingly send an answer.

3.4.1 *Database*

When it comes to build an application that handles data as its main purpose, it is essential to study which kind of database shows itself more competent for the job. There are two main types of databases, relational and non-relational, being the major difference between them the way they handle data.

Relational databases organize data into tables which can be related on data common to both, enabling the possibility to retrieve an entirely new table from data in one or more tables with a single query.

In the other hand, non-relational databases are databases that don't follow the relational model provided by traditional relational database management systems. This kind of databases - such as *MongoDB* - are very used nowadays when it comes to *webapps* due its advantages:

- No-SQL databases support rapidly changing designs when requirements change often;
- No-SQL databases allow to store large amounts of data with little to no structure.

Although very appealing due to its quick configuration and usage, it falls impotent to what an relational database has to offer. The following reasons lead to choose an SQL database:

- Data is highly related. As seen in the figure 2 on page 14, data is often correlated to each other, so a related database makes sense;
- Data structure is not changing. The design is strong enough to not change much in the event of requirement adjustments;
- Enforcing the ACID (Atomicity, Consistency, Isolation, Durability) principles, reducing anomalies and enforcing integrity.

After taking the decision to design the database SQL, the model was seen in figure 5 on the following page was built.

There X main tables: `sysversion`, `institution`, `department`, `teacher`, `course`, `class`, `degree`, `shift`, `year`, `types` and `poweruser`.

sysversion

As the name conveys, this table takes care of the system versioning. Its main purpose is to keep track of each year's data and giving the possibility to switch between them. This allows for the system to keep general data that (mostly) don't change along the years, such as departments, teachers, degrees and courses but start clean in other relations, such as shift making.

Column	Type	Context
`id`	`integer`	Primary key for each table entry;
`year`	`integer`	The year it is supposed to represent.

institution

In this table it's possible to find the basic information about the institution using the system, as well the version currently used.

Column	Type	Context
`id`	`integer`	Primary key for each table entry;
`name`	`text`	Institution's name;
`location`	`text`	Institution's location;
`sys_version_id`	`integer`	Foreign-Key to `sysversion(id)`, being possible to determine which version is currently running.

department

This table contains information for every department in the institution.

Column	Type	Context
`id`	`integer`	Primary key for each table entry;
`name`	`text`	Department's name;
`responsible`	`integer`	Foreign-key to `teacher(id)`, corresponding to the department's responsible;
`abbreviation`	`text`	x Department's abbreviation;
`_ipb_escola`	`integer`	IPB's particular required fields;
`_ipb_ccusto`	`integer`	IPB's particular required fields

Figure 5: Database diagram

teacher

This table contains information for every teacher in the system;

Column	Type	Context
`id`	`integer`	Primary key for each table entry;
`name`	`text`	Teacher's name
`local_identifier`	`text`	Short identifier for each teacher;
`email`	`text`	Teacher's e-mail;
`password`	`text`	Teacher's account password;
`password`	`text`	Foreign-key to `department(id)`, corresponding to the department it belongs to;
`_ipb_cod_escola`	`integer`	IPB's particular required fields;
`_ipb_emp_num`	`integer`	IPB's particular required fields;

course

This table contains information for every course in the system;

Column	Type	Context
`id`	`integer`	Primary key for each table entry;
`name`	`text`	Course's name; x
`semester`	`integer`	Semester when it occurs;
`department_id`	`integer`	Foreign-key to `department(id)`, corresponding to the department it belongs to;
`abbreviation`	`text`	Course's abbreviation;
`_ipb_cod_escola`	`integer`	IPB's particular required fields;
`_ipb_cod_curso`	`integer`	IPB's particular required fields;
`_ipb_n_opcao`	`integer`	IPB's particular required fields;
`_ipb_n_disciplina`	`integer`	IPB's particular required fields;
`_ipb_n_plano`	`integer`	IPB's particular required fields;

class

This tables contains the correlation between `courses` and `types` - basically, the type of classes each course has.

Column	Type	Context
`pair_id`	`integer`	Data entry identifier;
`course_id`	`integer`	Foreign-key to `course(id)`, corresponding to the course it belongs to; Composite with `type_id`, the primary key of with the table;
`type_id`	`integer`	Foreign-key to `type(id)`, corresponding to the type of class it is; Composite with `class_id`, the primary key of with the table;

degree

This table contains the information for every degree in the system;

Column	Type	Context
`id`	`integer`	Primary key for each table entry;
`name`	`text`	Degree's name;
`abbreviation`	`text`	Degree's abreviation;
`_ipb_cod_escola`	`integer`	IPB's particular required fields;
`_ipb_cod_curso`	`integer`	IPB's particular required fields;
`_ipb_n_plano`	`integer`	IPB's particular required fields;

shift

This table contains the information for every shift in the system;

Column	Type	Context
`id`	`integer`	Primary key for each table entry;
`sys_version_id`	`integer`	Foreign-key to `version(id)`, corresponding to the version it belongs to;
`counter`	`integer`	Number of classes attending this shift; When zero, a trigger will remove the respective row.

year

This table was built in order to normalize the data relative to scholar years; It'll be inserted by the administrator;

Column	Type	Context
`id`	`integer`	Primary key for each table entry;
`name`	`text`	Year's assigned name;
`abbreviation`	`text`	Year's assigned abbreviation;

types

This table was built in order to normalize the data relative to class types; It'll be inserted by the administrator;

Column	Type	Context
`id`	`integer`	Primary key for each table entry;
`name`	`text`	Type's assigned name;
`abrev`	`text`	Type's assigned abbreviation;
`color`	`text`	Type's assigned color;
`priority`	`integer`	Type's assigned priority;

poweruser

This table contains the information to the administrator's account.

Column	Type	Context
`id`	`text`	x Primary key for each table entry, and it's id;
`password`	`text`	Administrator's password;

The many-to-many relations between tables, make the rest of the tables - such as: `course_teacher`, `degree_course`, `assigned_degree_course`, `shift_class`, `shift_teacher` and `class_number`:

course_teacher

This table relates the tables `course` and `teacher`; For a given course, its possible to find the teachers that got assigned to it;

Column	Type	Context
`teacher_id`	`integer`	Foreign-key to `teacher(id)`, corresponding to an assigned teacher for a given course;
`course_id`	`integer`	Foreign-key to `course(id)`, corresponding to the course it belongs to;
`pair_id`	`integer`	Data entry identifier;

degree_course

This table relates the tables `degree` and `course`; Its main purpose is associate the for a given course the plan of courses it has;

Column	Type	Context
`course_id`	`integer`	Foreign-key to `course(id)`, corresponding to the course it belongs to;
`degree_id`	`integer`	Foreign-key to `degree(id)`, corresponding to the degree it belongs to;
`year_id`	`integer`	Foreign-key to `years(id)`, corresponding to the scholar year that the course that was assigned it belongs to;
`semester`	`integer`	Semester which when it occurs; x
`pair_id`	`integer`	Data entry identifier;

assigned_degree_course

This table relates the tables `degree` and `course`; Although the previous table does the same, this one has one specific entry that changes every year - the number of attendants.

Column	Type	Context
`degree_course_id`	`integer`	Foreign-key to `degree_course(id)`, corresponding to the course and degree it belongs to;
`attending`	`integer`	Number of students for that given degree attending the course;
`sys_version_id`	`integer`	Foreign-key to `version(id)`, corresponding to the version it belongs to;

shift_class

This table relates the tables `shift` and `degree`; For a given shift, its possible to find the degrees that got assigned to it;

Column	Type	Context
`shift_id`	`integer`	Foreign-key to `shift(id)`, corresponding to the shift it belongs to;
`class_id`	`integer`	Foreign-key to `class(id)`, corresponding to the class it belongs to;

shift_teacher

This table relates the tables `shift` and `teacher`; For a given shift, its possible to find the teachers that got assigned to it;

Column	Type	Context
`shift_id`	`integer`	Foreign-key to `shift(id)`, corresponding to the shift it belongs to;
`teacher_id`	`integer`	Foreign-key to `teacher(id)`, corresponding to the teacher it belongs to;
`sys_version_id`	`integer`	Foreign-key to `version(id)`, corresponding to the version it belongs to;

shift_teacher

This table is used to separate the various classes that exist per week for each class;

Column	Type	Context
`class_id`	`integer`	Foreign-key to `class(id)`, corresponding to the class it belongs to;
`number`	`integer`	Represents the number of class it is;
`duration`	`integer`	Duration of said class;
`id`	`integer`	Primary key for each table entry;

4

GSD: DEVELOPMENT

Along this chapter, the development circumstances, problems and decisions that together compose the final product are going to be addressed and studied into detail.

4.1 BUILDING BLOCK VIEW

Before making any choices into development, it should be reminded that the user-interface should be running on web, while very dynamic. The traditional web solutions couldn't apply since the problem is of a very specific domain. For this same reason, the solution goes through the use a modern framework to build the web application.

Since the used data isn't static but manipulated across usage, this system comes across as a full-stack development solution - this means that the development of both server-side back-end) and client-side (front-end) portions of the web app [8] is required.

Currently, there are a large number of front-end technologies that could be used to implement the client-side, but `react-js` was chosen:

- Being a JavaScript library, has available a substantial online documentation;
- Makes use of `NPM` (`Node Package Manager`) which contains numerous useful tools;
- It's very reusable - its component approach makes a simple piece-by-piece building method;
- It fits every usage requirement;
- The developer experience has past with it;

As usual in web-applications and as a adequate way to keep inoperability between servers, communication between the back and front-end servers will be performed via `REST` (Representational state transfer). Therefore, the back-end is going to work as a `REST API server`.

A `REST API` server should attend the following rules [15]:

- **Client–server**: The client and the server should be separate from each other and allowed to evolve individually and independently;
- **Stateless**: Requests can be made independently of one another, and each request contains all of the data necessary to complete itself successfully;
- **Cache**: The REST API should be designed to encourage the storage of cacheable data;
- **Uniform Interface**: In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components.
- **Layered System**: The REST API should adopt a layered system, in which each layer has a specific functionality and responsibility;

As for the back-end server, `Node,js` was chosen to implement it. `Node.js` is a `JavaScript` runtime environment which allows the infrastructure to build and run an application. It's a light, scalable, and cross-platform way to execute code. It uses an event-driven I/O model which makes it extremely efficient and makes scalable network application possible. It offers its users a set of advantages, such as [11]:

- Abstraction: `Node.Js`' single-threaded, event-driven architecture allows it to handle multiple simultaneous connections efficiently - while abstracting the developer of its implementation and make it simple to use;
- The ever-growing `NPM` (`Node Package Manager`) gives developers multiple tools and modules to use; An important usage of this package manager was the `Express.js` framework, which its usage is going to be explained later in the document;
- Being implemented in the same language as the front-end - `JavaScript` - it allows to keep a bigger inoperability, while increasing the efficiency of the development process;

As mentioned before, the `Express.js` framework is used along `Node.js`. `Express.js` is a minimal and flexible `Node.js` web application framework that provides a robust set of features to develop web and mobile applications. Its main features [19] are:

- Allowing to set up middlewares to respond to HTTP Requests - meaning that it's possible to implement a RESTful API using this framework;
- Defining a routing table which is used to perform different actions based on HTTP Method and URL.
- Allowing to dynamically render HTML Pages based on passing arguments to templates;

Having chosen the technologies in which the application servers will be running, it is time to establish which database is going to be used. There are multiple SQL databases available to use, but PostgresSQL was chosen. These factors were taken in account:

- Concurrency and Performance: Being a web-application, multiples accesses at the database will be done at the same time; Having a database that automatically takes control of the concurrency is a big advantage;
- Data Types: Having modern data-types, PostgresSQL has shown itself as a major candidate for the job;
- Popular choice to go along Node.js - being a popular choice means there's a fair amount of frameworks to make its connection easy and well documented;

4.1.1 *Back-end server: Node.js + Express*

Being the front-end application implemented adopting React.js, the system's back-end will use Node.js + Express as its back-end application. Together, these entities and its relationship will result in client-server architecture, as a requirement of a conventional RESTful API Server, where there back-end is need for

a. business logic that shouldn't be exposed as source code to the frontend application;
b. establishing connections to third-party datasources - such as the database.

The back-end directory has the following structure:

```
--controllers
--file-generate-logs
--migrations
--models
--routes
--views
.env
app.js
auth.js
config.js
institutiondb
knexfile.js
middleware.js
package.json
```

`passport.js`

`app.js` is the main entry point of application, serving HTTP requests that reach the servers - these requests can be `GET`, `POST`, `PUT`, `PATCH` and `DELETE`. In order to keep an organized and understandable code, multiple endpoints were created for main parts of the database:

`/auth` Endpoint for authentication requests, being the only public one;

`/teacher` Endpoint for teacher-related requests;

`/degree` Endpoint for degree-related requests;

`/course` Endpoint for course-related requests;

`/department` Endpoint for department-related requests;

`/assigned` Endpoint for assigned-related requests;

`/class` Endpoint for class-related requests;

`/shift` Endpoint for shift-related requests;

`/types` Endpoint for type-related requests;

`/generator` Endpoint for output generator;

`/years` Endpoint for years-related requests;

`/institution` Endpoint for institution-related requests;

Each one of these endpoints will have their own controller, inside of the `controllers` directory. For example, for every endpoint starting in `/auth`:

```
const auth = require('./controllers/auth')
const app = express()
app.use('/auth', auth)
```

it's possible to find every correlated sub-route in the `controllers/auth.js` file:

```
router.post('/login', async (req, res, next) => {
/* database middleware */
})

router.post('/ping', async (req, res, next) => {
/* database middleware */
})
```

Database ORM

One key aspect of using a back-end server is using it as platform to a database. A solution often opted for is the use of an `ORM - Object-Relational-Mapper` - which implements a technique that lets querying and manipulating data from a database using an object-oriented paradigm, or, in this case `JavaScript`.

There are some `ORM`s available for `PostgresSQL`, such as `sequelize`, `TypeORM` and `objection` [13]. Taking in consideration and analyzing the previous options, `objection` [17] was chosen for taking full power of SQL and the underlying database engine while making it simple javascript queries, with an astounding documentation hat leaves no room for doubts.

Having the database built, it's required to establish relations for the `ORM` to work. For that same reason, the directory `/models` was made to contain a file for each main table in database, and each file containing a `Model` for that same table. A `Model` subclass represents a database table and instances of that class represent table rows. `Model` are created by inheriting from the `Model` class - provided by the `objection` package. An `objection` class can define relationships to other models using the static `relationMappings` property.

As example, in the file `/models/teacher.js` there's the models for `teacher` table, as follows:

```
class Teacher extends Model {
    static get tableName () {
       return 'teacher'
   }
/* */
}
```

This way the ORM assigns the class `Teacher` for the table `teacher`. Having in mind that the table `teacher` has relations with other tables, such as `department` (meaning that a given teacher belongs to given department), it's appropriate to establish their relation:

```
static get relationMappings() {
    const {Department} = require('./department');

    return {
        teacher_department: {
             relation: Model.BelongsToOneRelation,
             modelClass: Department,
             join: {
                 from: 'teacher.department_id',
```

```
                    to: 'department.id'
                },
            }
        }
    }
```

Examining the snippet above, the following can be interpreted:

- The `Department` model is being imported;
- A relationship identifiable by `teacher_department` is being created;
- The relation type is `BelongsToOneRelation`, meaning that a `teacher` belongs to a `department`
- The columns from each table which when joined make the relation;

The relation mentioned above was an one-to-many relation. When it comes to a many-to-many, that's when the pairing tables comes to use - as an example, for the relation teacher-shift:

```
    shifts: {
        relation: Model.ManyToManyRelation,
        modelClass: Shift,
        join: {
                from: 'teacher.id',
                through: {
                    from: 'shift_teacher.teacher_id',
                    to: 'shift_teacher.shift_id',
                    extra: ['sys_version_id']
                },
                to: 'shift.id'
        }
},
```

Giving that multiple teachers can be assigned to a single shift, and a teacher can teach multiple shifts, this makes a many-to-many relation. In the snippet above, it's possible to understand the use of the middle table *shift_teacher* to establish the relation mapping.

Authentication: JWT + Passport

An important aspect of a client-server connection is maintaining the users authenticated once a successful login is made. Every time a given user has an open session and makes a request for protected data, it's a necessary to validate

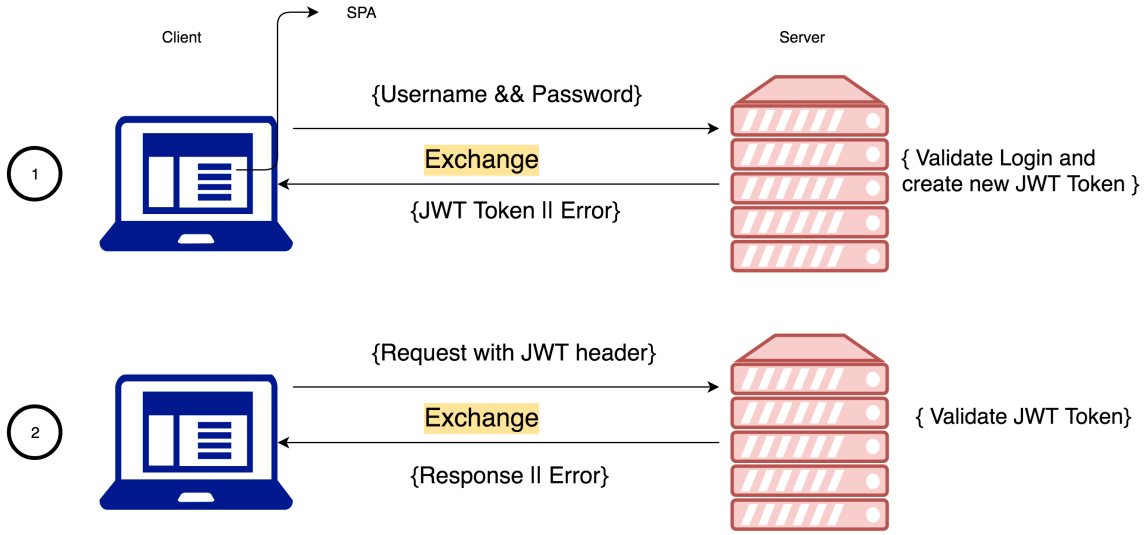


Figure 6: JWT: How it works

a) if the user is logged in;

b) if the user's session is valid.

One possible solution would be storing the user's authentication credentials in the browser, but it represents a big safety threat. For that same reason, another alternative had to be considered and thus the application of `JWT - JSON` web tokens. `JSON` web tokens are text strings that can be used by client and server to authenticate and share information easily.

The diagram 6 [14] provides a better understanding:

a. A client sends username/password combination to the back-end server;

b. The back-end server validates the credentials;

c. If authentication is successful, the back-end server generates a `JWT` token; otherwise establishes an error response;

d. On a successful authentication, the client receives `JWT` token as response;

e. Client stores that token in local storage or session storage.

f. Henceforward, the client attaches the `JWT` token in every request header: `Authorization: Bearer <jwt_token>`

g. Upon receiving the `JWT`, the back-end validates it and replies with a successful or error response.

Every time a user gets authenticated, the back-end server generates a token based on the username and a random word selected for every token, valid for 24 hours:

```
 let token = jwt.sign({username: username},
    config.secret,
    { expiresIn: '24h' // expires in 24 hours
});
```

Later, when an authenticated user sends a request with a token in its header for a protected route, that's when the module `passport` comes into action. Besides `/auth`, every route is protected, having to be logged in to access it:

```
app.use('/teacher',
    passport.authenticate('jwt', {session: false}), teacher)
```

If a request token is expired, null or invalid, the `passport` module automatically sends a HTTP code 403 - Forbidden - refusing to serve every request to any protected data.

Logging

Even performing a full plan and test of the system, it can still fail. For this very reason, it was implemented a logging mechanism adopting `log4js` [12]. The module `log4js` allows the developer to log multiple levels of severity or priority - `DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`, `TRACE` - while choosing its output. While developing, the main output used was via console (for a quicker debug); however in production, the module saves the output in the file `./error.log`. In case of a unexpected behaviour, the developer can easily check the problem by checking that same file.

4.1.2 *Front-end application: NEXT.js*

`React.js` is an open-source, component-based `JavasSript` library which is used to build user interfaces specifically for single-page applications. Is was built by Facebook developers in 2011, keeping until today a large community support along with a lot of documentation, hence it making a logical choice for front-end implementation.

Having such a massive community, it's innate that some frameworks flourished developed on top of it, in order to make things even more desirable. `Next.js` is one of these frameworks. The implemented front-end takes advantage of the following features:

- Server-Side Rendering
  Renders `React` components on the server-side, before sending the HTML to the client.

- Automatic Routing - Dynamic Pages
  Any URL is mapped to the filesystem, to files in the pages folder, dismissing configuration;

- Automatic Code Splitting
  Pages are rendered exclusively with the libraries and JavaScript they require. Instead of generating one single `JavaScript` file containing all the app code, the app is broken up automatically in several different resources.

- Prefetching
  Its `Link` component, used to link together different pages, supports a prefetch prop which automatically prefetches page resources in the background;

The front-end server directory has the following structure:

```
--auth/
--components/
--forms/
--layout/
--pages/
--static/
--storage/
--stylesheets/
    --modules/
    --partials/
    --vendor/
next.config.js
package.json
```

Page layouts

As stated formerly, the folder `/pages` contains every page available to render by the front-end server, e.g. a file `pages/page.js` will route the page to `example.com/page`. This means that every page in the folder `/pages` should contain the code for each available system use-case. The following pages were created:

`_app.js`
`_document.js`
`associate.js`
`config.js`

`costs.js`
`courses.js`
`data.js`
`degrees.js`
`department.js`
`index.js`
`shifts.js`
`teachers.js`

The file `_app.js` is a special configuration file provided by `Next.js` and will be later addressed with detail in the 4.1.2 section. The `_document.js` is helpful when it comes to provide pages' name and favicons.

LOGIN PAGE The `index.js` file routes to `/`, where the login page can be found. Being a simple login page with nothing but a header and two inputs for the user's credentials, there isn't the need to build a special layout for it.

DASHBOARD After successfully authenticating in the system, the users have access to the remaining pages that they are designated to. This group of pages will constitute the *Dashboard* which has a general layout for every page to use. In order to keep a standard design and give use to the re-usability that `Next.js` offers, the layout `layouts/dashboard.jsx` was built.

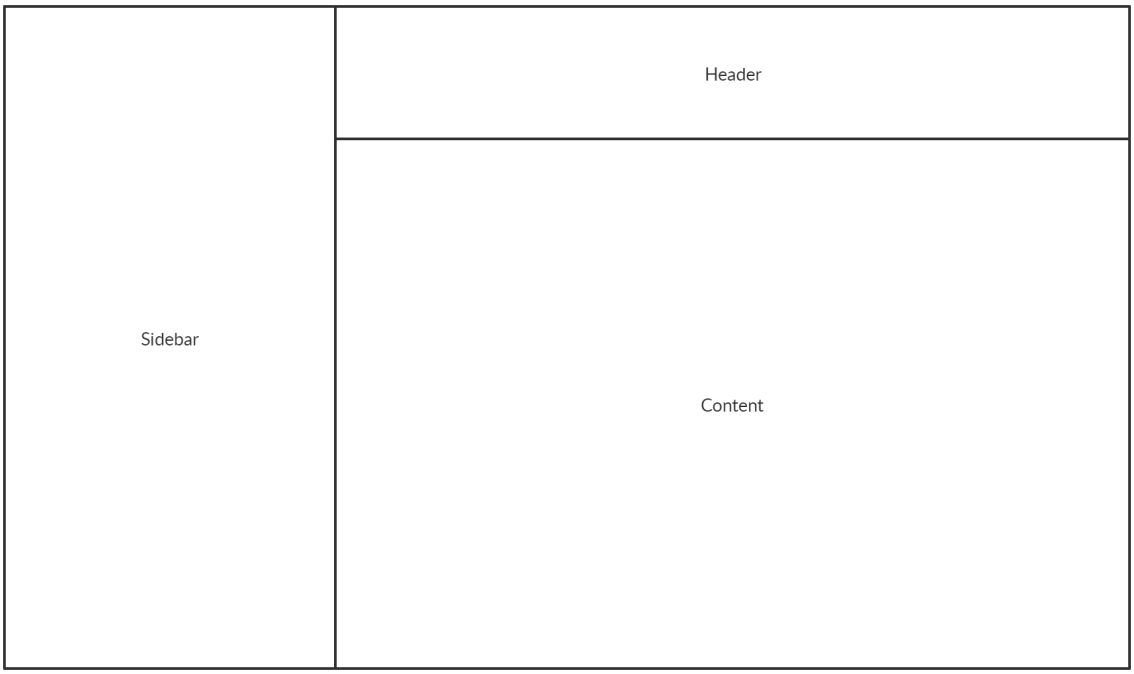


Figure 7: Dashboard layout

As it possible to check in the figure 7, the dashboard is split in three main parts - `Sidebar`, `Header` and `Content`. `Content` is the only volatile part of the three, changing between pages. The remaining two follow always the same structure and can be implemented once to be used every page, taking use of the component approach `React.js` has to offer.

In the directory `/components` was built two new components and imported to the dashboard layout:

- Header as `/components/header.jsx`
- Sidebar as `/components/sidebar.jsx`

```
import Sidebar from '../components/sidebar';
import Header from '../components/header';

export default function withDashboard(Component) {

    return class Dashboard extends React.Component {
        render() {
            return(
                <div className="dashboard-container">
                    <Sidebar
```

```
                    /* props */
                />

            <div className="content">
                <Header
                    /* props */
                />

                <div className="body">
                    <Component
                        /* props *//
                    >
                </div>
            </div>
        </div>
        )
    }
  }
}
```

Upon inspection, there are three main components along the `HTML` code in the `render()` function - `<Sidebar/>`, `</Header>` and `<Component>` - the former two were imported after being implemented as separate components, and later one comes as a parameter for the function which implements the `Dashboard`. When given page uses the `Dashboard` layout, it calls the `withDashboard(component)` function e.g. the `/pages/teachers.jsx` file:

```
import withDashboard from '../layout/dashboard';

class Teachers extends React.Component {
    /* implementation */
} export default withDashboard(Teachers)
```

This way each file has only to render the code for the content part of the `Dashboard`.

Authentication

Part of authentication process was already discussed section 4.1.1, in respect to the back-end part of the procedure. In a quick rundown, the back-end after a successfully log-in sends the

user information as well the JWT token. The figure 16 resumes as an activity diagram how the procedure goes.

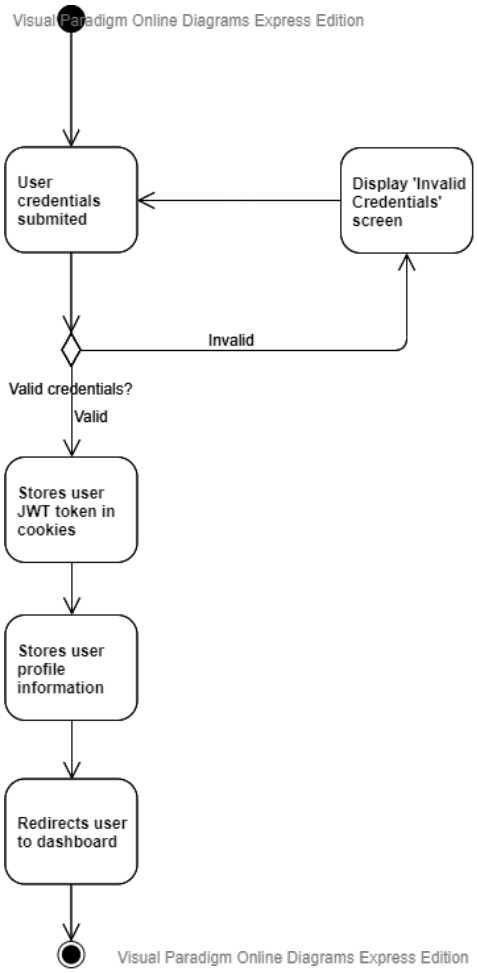


Figure 8: Login procedure

Once the login is successfully made, both the user's profile data and JWT token are saved in the browser's cookies. Initially, the aim was to save user profile data in `Local Storage` (browser storage in disk); however it revealed to be a problem from browser to browser. As the data to save is very small in size, it is kept in cookies as it works perfectly.

As mentioned before, `/pages/_app.js` takes a major role for keeping an user authenticated. `Next.js` uses the App component to initialize pages. Overriding it lets take control of the page initialization.

The procedure that occurs during a page initialization is easily described using a diagram, as seen in figure 9. Every time a user tries to access a page that isn't the login page, the system will check the token and the route that is being accessed (and if the user has access to it).

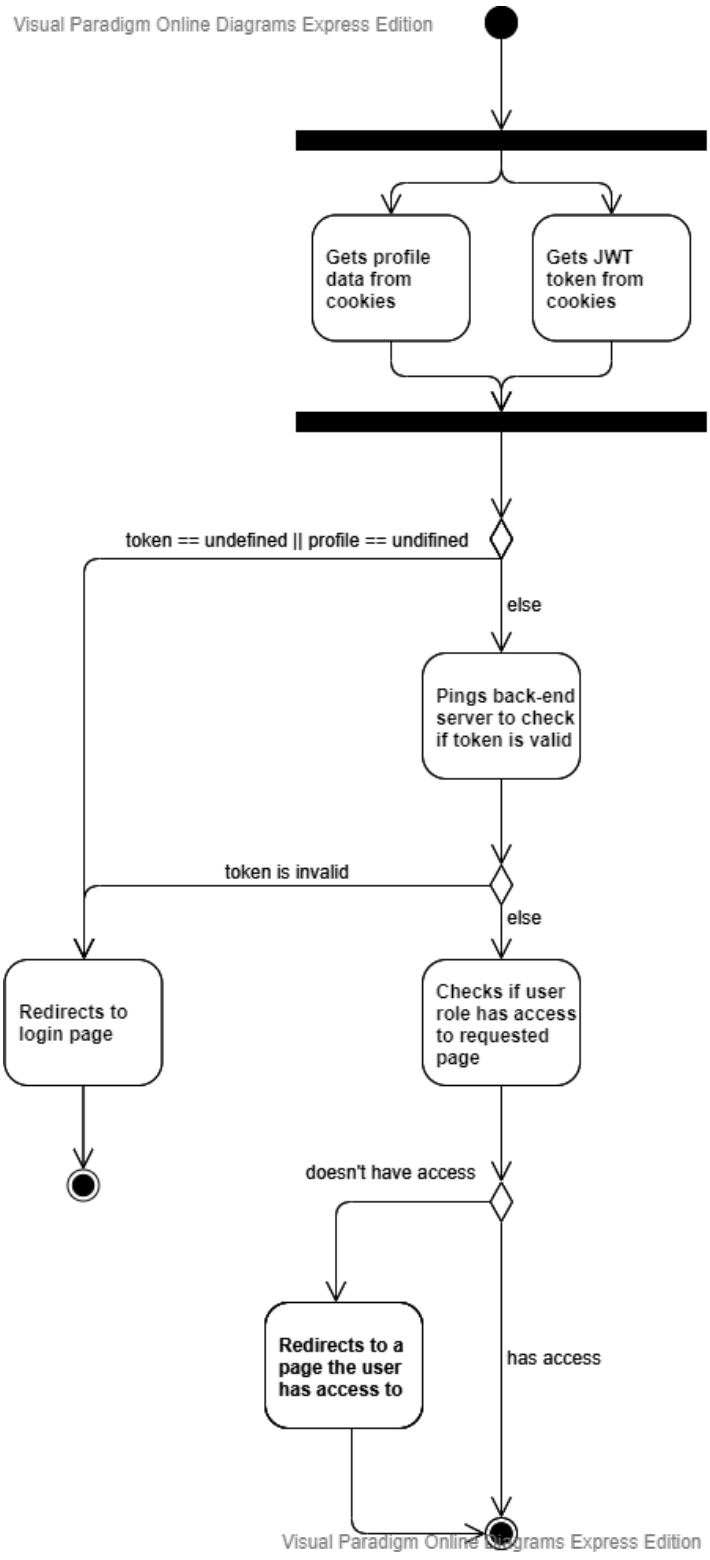


Figure 9: Authentication procedure

Page Styling - SASS

While the `React.js` generates the `HTML` to render a page, styling is still needed to make them look structured. Usually, `CSS` is the language chosen to style the `HTML` pages, and that's no exception in this case, although with a help of a `CSS` pre-processor, `SASS`. It enables the use variables, mathematical operations, mixins, loops, functions, imports, and other interesting functionalities that make writing `CSS` much more powerful [3] - and easier.

The styling folder `/stylesheets` has three subfolders - `modules`, `partials` and `vendor`.

- Besides the subfolders, `/stylesheets` contains the `main.sass`, which is the primary `SASS` file which imports the remaining files;
- The `/modules` sub-folder contains the files for common modules, such as the header in the dashboard;
- The `/partials` sub-folder contains the files for each page;
- Finally, the `/vendor` sub-folder contains third-party files.

4.1.3 *Output generator*

The output generator is an important feature of the system, given that it produces the final result that's going to be imported by other software. In a point of view of development, it was one of the most difficult features to implement due to its extensive planning.

Initially, it was established which database the system is going to adapt to; for this same reason the developer requested a sample of the database for Instituto Politécnico de Bragança, the target user of this system. This part required a manual analysis work, and building an excel spreadsheet revealed itself an excellent employer for the job. The analysis work was split in 4 parts:

a. Inspect the database file and understand which tables needed to be supplied with data; Upon confirmation by coordinator the developer proceeded to next step;

b. For each table needed do provide data to, the developer linked each column in that given table to the systems database containing the same data;

c. For each table needed do provide data to, it was established its relation to other tables (*foreign-keys*);

d. Having the information from the step before, settled the order by which each table will be fed data.

The item **b.** of the previous list exposed a problem in the system's database - it was missing some IPB's specific fields. However, this problem was not of bigger extent because its fix was actually simple - add columns to existent tables, no need to create new tables or relations. This new columns were added with an underscore prefix, `_`, meaning they're specific fields. E.g., the Teacher table added the new columns `_ipb_cod_escola` and `_ipb_emp_num`.

JSON to SQL: A failed approach

The first approach taken to implement this feature was to build tables with JSON and later convert it to SQL. Given the project is implemented mainly using JavaScript, it was intended to follow the same strategy for generating the output.

The ORM once queried, presents its results in JSON. The plan was to build a representation of a SQL database using JSON - a JavaScript Object Connotation. By the definition of both, it seems already a detrimental solution to the problem - but it was possible, and so, the plan proceeded.

The related tables would be nested, and its properties stored in variables. There are tools that would transform JSON into SQL - such as JSON-SQL [1] , which was used. As the implementation continued, the code got chaotic, hard to read and debug - it was hard to keep track of code and the data. At the end, it seemed pointless and this approach was abandoned.

ETL - Extract, Transform, Load

After discarding the last plan, a new research was made in interest to find a solid solution to this problem. The procedure that correctly translates what's intended is ETL - which is short for extract, transform, load - three database functions that are combined into one tool to pull data out of one database and place it into another database. After inspecting numerous documentation for it online, an article [10] described a simple yet detailed method to do it, as well as a real-word example.

The first step is to analyse both data models, and work out a mapping for the data between - which already has been done, as stated before, with the help of the spreadsheets. The next step was to design the architecture for the custom ETL solution. As it is possible to check in the figure 10, there are two new schemas - migrate and gal.

Figure 10: ETL Architecture

The migrate schema has the structure of the destination database, containing also the identification from the originating table i.e. the systems corresponding table. In the other hand, the gal schema has only the structure of the destination database, ready to import.

The migrate schema is used in order to assert the many relations in the structure of the destination database. To do this, a SQL file script was made and loaded into the database, which migrates the data from one schema to another.

In a simple example:

```
CREATE TABLE migrate.departamento (
    id SERIAL PRIMARY KEY,
    old_id int NOT NULL,
    nome varchar(100) NOT NULL DEFAULT '',
    abrev varchar(10) NOT NULL DEFAULT '',
    ipb_cod_escola int DEFAULT NULL,
    ipb_emp_ccusto int DEFAULT NULL
);

CREATE TABLE migrate.docente (
    id SERIAL PRIMARY KEY,
    old_id int NOT NULL,
    id_depart int DEFAULT NULL
        REFERENCES migrate.departamento (id) ON DELETE CASCADE,
    nome text NOT NULL DEFAULT '',
    abrev text NOT NULL DEFAULT '',
    eti float NOT NULL DEFAULT '1',
    mail text DEFAULT NULL,
    credito float NOT NULL DEFAULT '0',
    ipb_cod_escola int DEFAULT NULL,
    ipb_emp_num int DEFAULT NULL
);
```

The migrate.docente table needs to reference migrate.departamento(id). This is easily done accessing the old_id from migrate.departamento and retrieving the actual new id.

A new SQL script was imported to the database, this one being responsible for migrating from the contents from *migrate* to *gal*. This step was simpler as it will replicate the same values of migrate, excluding only the old_id for each table.

Having now the data transformed into the structure of the destination database, the next step is to gather said data and create a script to load it. As opposed to the system's database - which is written using `PostgresSQL` - the destination database is running on `MySQL`, adding a new level of complexity.

The `PostgresSQL` database offers the command `pg_dump` that, for given a database, creates an output containing an `SQL` script to load the data. After performing that command and save the output to a file, the final step was to convert it to `MySQL` accepted script. The tool `PG2MySQL Converter` [4] solves that problem, as it uses a command-line solution to solve the problem.

It's now a matter of joining all the procedures above mentioned in a single, executable and repeatable script that would work every time. Thus, a shell script was assembled:

```
clear;
printf "${RED}-> creating migration schema...${NC}\n"
psql -U admin -d institutiondb < /home/admin/db/migrate.sql
printf "${RED}-> migrating data..${NC}\n"
psql -U admin -d institutiondb -c "select proc1();";

printf "${RED}-> creating gal schema...${NC}\n"
psql -U admin -d institutiondb < /home/admin/db/gal.sql
printf "${RED}-> migrating data..${NC}\n"
psql -U admin -d institutiondb -c "select proc2();";

pg_dump --no-acl --no-owner --format p --data-only institutiondb
    -n gal -f /home/admin/db/pgfile.sql;
php /home/admin/db/converter/pg2mysql_cli.php
    /home/admin/db/pgfile.sql /home/admin/db/mysqlfile.sql
```

Breaking the script in steps:

a. Imports the `migrate` schema to the database; if it already exists, deletes the existent and its data;

b. Migrates the data from the `public` schema (the system's database) to the `migrate` schema

c. Imports the `gal` schema to the database; if it already exists, deletes the existent and its data;

d. Migrates the data from the `migrate` schema to the `gal` schema

e. Dumps data from the `gal` schema to a file;

f. Using `PG2MySQL Converter`, convert the SQL above generated into a `MySQL` appropriate script.

When the user requests the `MySQL` file, the back-end server executes the script above, waits for it to finish, loads the `mysqlfile.sql` and sends it to the user.

4.2 RUNTIME VIEW

In this section will be covered the user's interaction with the system.

4.2.1 *General Data*

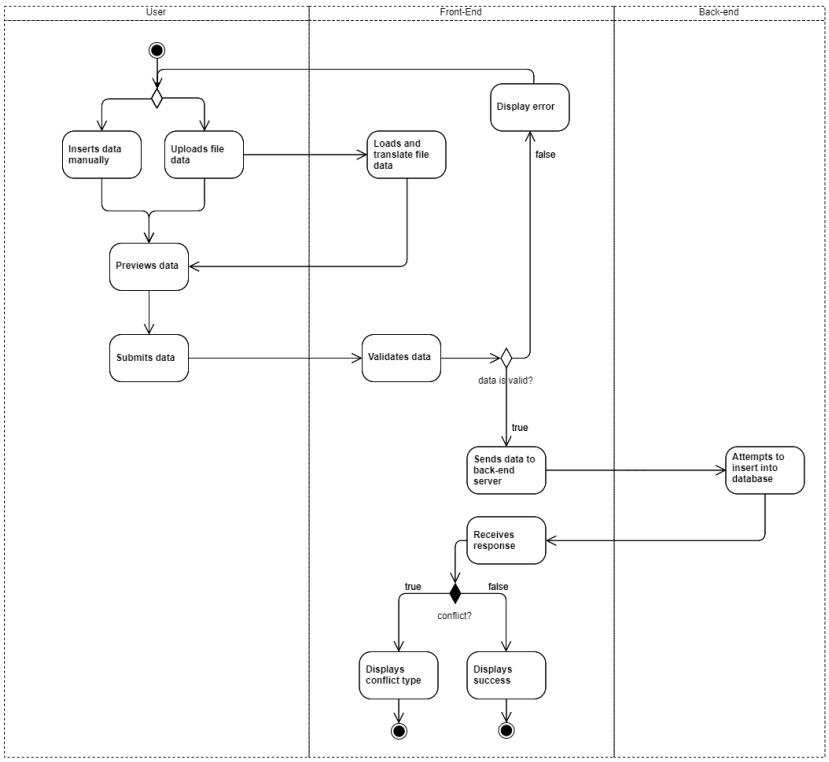


Figure 11: Activity Diagram for data insertion

There are two possible ways to insert data into the system - either manually or through file uploading. The figure 11 shows an activity diagram that helps understand the interaction when loading data into the system.

a. The user inserts data - manually or uploading a file;

b. The user submits the data above inserted;

c. The front-end validates the submitted data;

   a) If the data is invalid or missing one or more fields, the user is warned and the data not sent to the back-end server;

   b) Otherwise, the data is sent to the back-end server;

      i. The back-end-server attempts to insert it into the database - it could fail due to conflicts;

         - Some examples of conflicts could be e.g. some repeated values;

   c) The front-end receives response from the back-end server;

      i. If the response is positive, indicates success;

      ii. Otherwise, indicates which fields cause conflict;

4.2.2 *Association*

There's always a course for each different degree, i.e. the same Analysis will result in two different courses for two different courses, each own having their own code and a code for the degree associated. Initially, the system would load both independently and later the user would link them; later in development, the system links them automatically. The activity diagram shown in 12 clarifies the interaction between the user and system in the section in charge of managing the association between degrees and courses.

In a brief rundown, the link is made when a user inserts a new course:

- The system validates the data, as seen previously in 11;
- The system checks if the degree code associated with the course exists in the database:
  - If it doesn't exists, the data isn't inserted into the database;
    - The system displays error, detailing the degree missing;
  - If it does, the system inserts the data into the database and displays success;

4.2.3 *Shifts*

The shift management section is an important feature of the system. For that reason, it was designed an activity diagram - shown in 13 - to better illustrate its interaction with the user.

a. The user gets a selection of courses from its department and picks one;

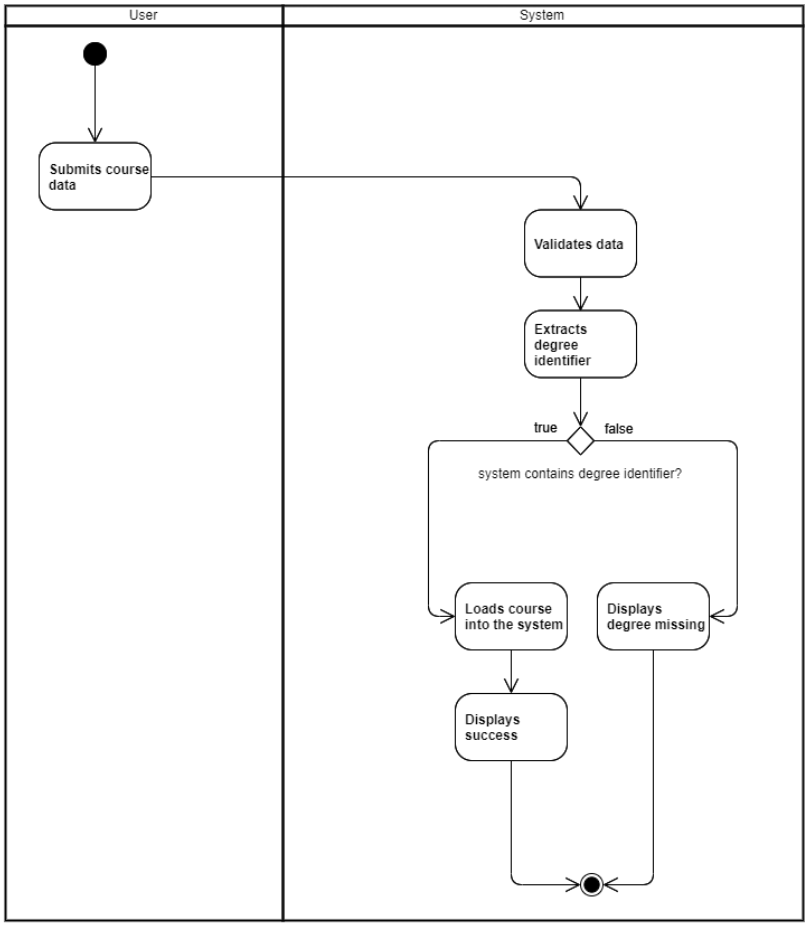


Figure 12: Activity Diagram for assigning a degree to a course

b. The system displays the number of participants that were assigned to the selected course, as well the existent shifts;

c. The user selects the class;

   a) The user can select a shift if more than one exist;

      i. The user can delete the selected shift;

      ii. The user can assign one or more teachers the selected shift;

      iii. The user can join this course with another one;

      - The course to join has to be from the same department;
      - The course has to have the same number of classes;
      - The course has to have the associated teachers;

   b) The user can add a new shift;

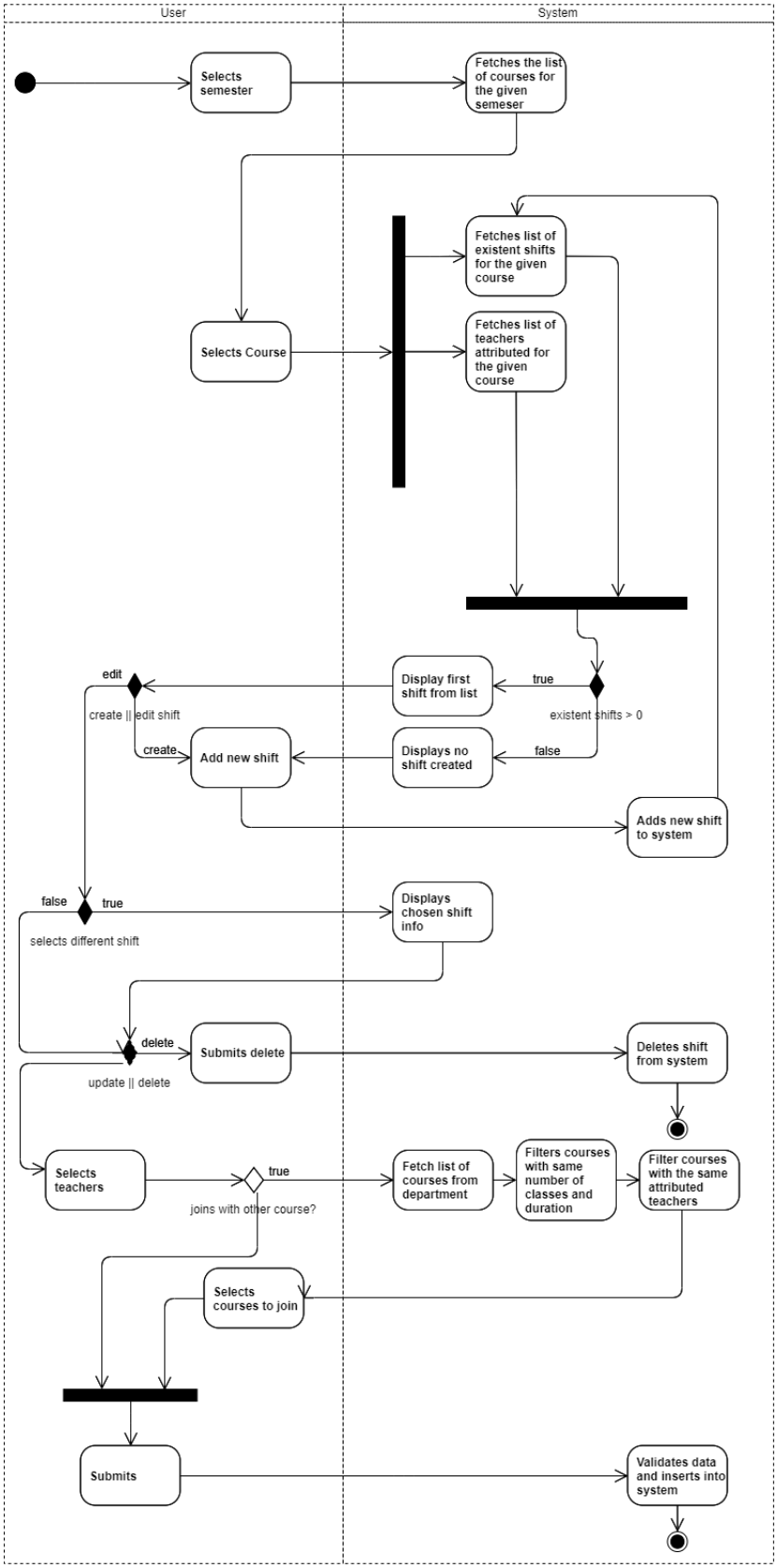


Figure 13: Activity Diagram for shift management

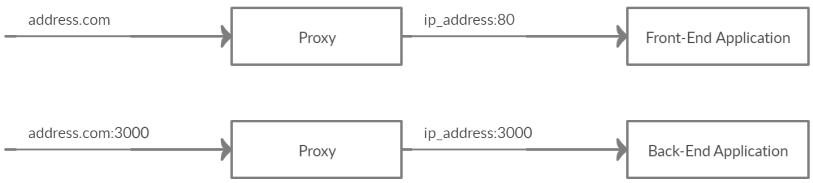


Figure 14: How the proxy works

4.3 DEPLOYMENT VIEW

Taking into account all the technologies used in the system and their interaction, the diagram shown in 15 represents the full picture of its architecture. The diagram split in two - the public network and the private network. The public network is the part accessible by the use regular user, i.e. the landing page after accessing the public address on his or her browser. The private network is all the logic part hidden from the user. Both the front-end and back-end will be hosted on an `apache` web server, which redirects requests to the instance which the servers are running on, as the diagram 14 illustrates.

Breaking the diagram in detail, the following happens when an user accesses a page:

a. The server's proxy, listening for requests in the port 80 - default port for browser addresses - redirects them to the instance where the Web App Server is deployed;

b. The Web App Server generates the web-page to render and sends it back to the browser;

c. The browser renders it.

If the rendered page needs to fetch, insert, update or delete data:

a. The browser sends a REST call to the proxy server through the port 3000;

b. The server's proxy, listening to requests on the port 3000, redirects it to the instance where the API Server is deployed;

c. The API Server accesses the database, performing the task that will accomplishes the request;

d. The API Server returns to the browser the status of the operation.

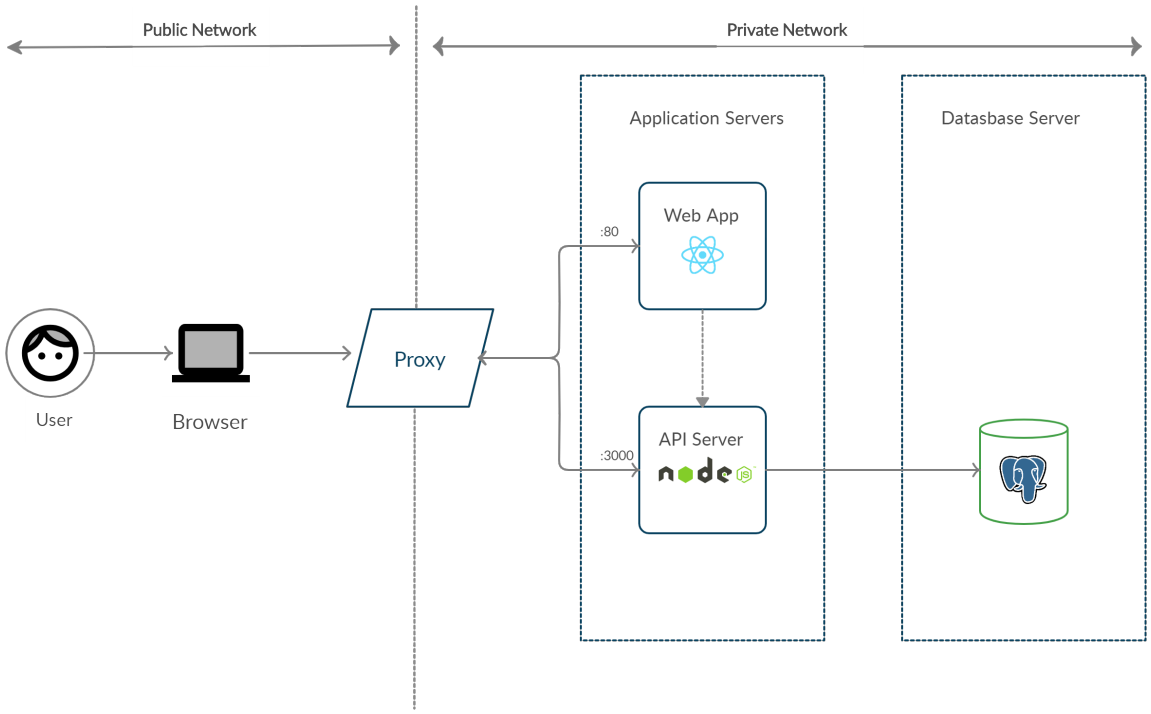


Figure 15: Fullstack architecture

4.4 CONCEPTS

Persistency

The database contains all the data needed for the application to work; However, some session information - such as profile information and the `JWT` token - are stored locally in the browser's cookies.

Session Handling

The session is entirely handled by the use of cookies and the Json Web Tokens (`JWT`).

Build, Test, Deploy

While developing the system, the system was tested locally with the support of `React.js` auto-reloading feature which allows quick testing while working on its source.

When deployed, both servers used `forever` [7] - a simple CLI tool for ensuring that a given script runs continuously.

Safety

When comes to prevent anomalies, there was an effort to prevent bad usage of the system through the front-end - if the user interface doesn't allow the user to misuse the system, the user won't. Then there's a second layer of protection which is the database validation, i.e. when inserting data into the database, it checks if it is incompatible - refusing if so.

Security

The main security feature in the use of web-server adopting `HTTPS`. HTTPS (Hypertext Transfer Protocol Secure) is an internet communication protocol that protects the integrity and confidentiality of data between the user's computer and the site, providing the three key layers of protection - encryption, data integrity and authentication. [9]

Exception/Error Handling

The front-end is ready to handle any error that comes from the front-end. Most of the errors come from data insertion in the database e.g. in repeated values. If the token is invalid, the front-end cancels the current operation and sends the user to the login page.

-

5

GSD: TESTS AND RESULTS

Through this chapter, the final system is on display - which be tested and the results analyzed.

5.1 EXPERIMENT SETUP

After offline development, both servers were deployed online in the structure shown as the section 4.3. Being both servers already set up and deployed, it's just a matter of accessing it via browser.

This experiment's main purpose is to study how agile and intuitive the system is, while performing the all the requirements.

5.2 RESULTS

Every page accessible by the system is going to addressed into detail in following subsections, being split for each user.

5.2.1 *Login*

The figure 16 displays the login page. It's a minimalist page that is shown to the users if they aren't logged in or have a valid session. If an invalid login attempt is made, a red box under the log-in button warns the user.

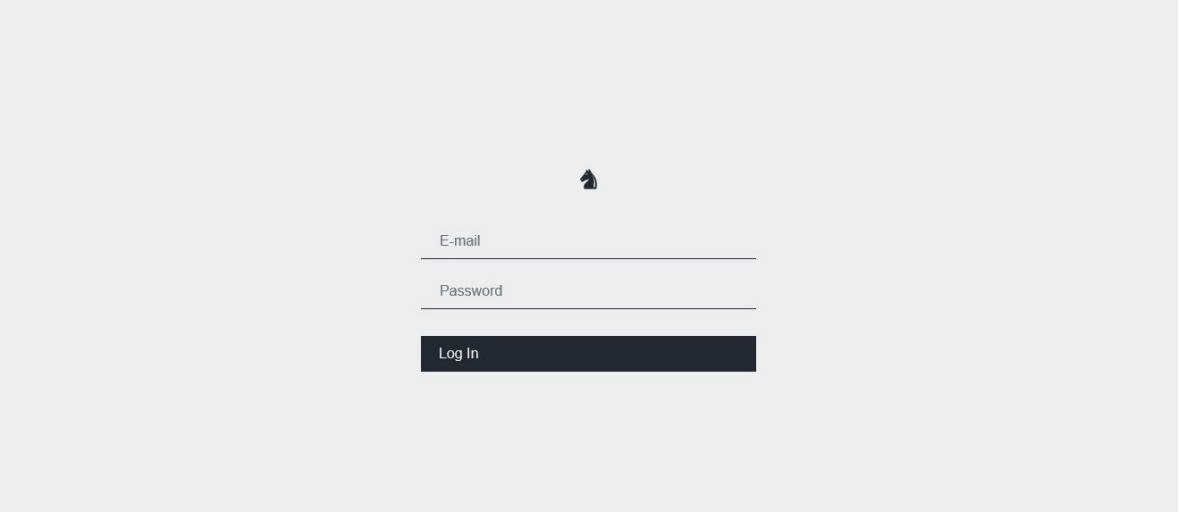


Figure 16: Login Page

5.2.2 *Administrator*

If the administrator makes an successful login attempt, the access to the dashboard section is granted. The first item in the menu is *Class types and scholar years*, leading to the data management page, as displayed in the figure 17.

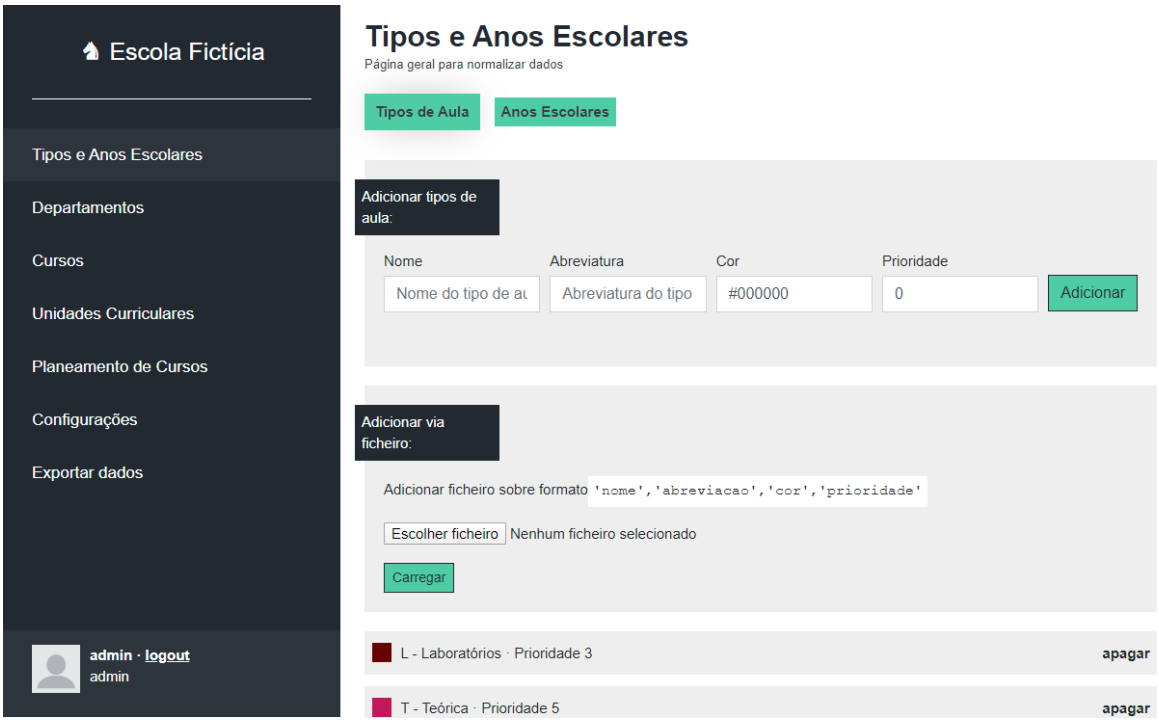


Figure 17: Class types and scholar years section page

There are two sub-menus in this page - one for class types and another for the academic years, respectively. This allows for a superior data normalization. Both sub-menus support a manual or mass insertion, the later through a file upload. The chosen format data for data uploading was `.csv`, for every section as well. Deleting one of these will lead to a `CASCADE` delete - this means that the relation course-type is deleted for every type deleted. Later, the administrator shall update the courses to make sure they have the correct types.

Next in the menu, there is the Departments management. In this page, as displayed in the figure 18, the administrator should upload the departments along the department's responsible teacher.

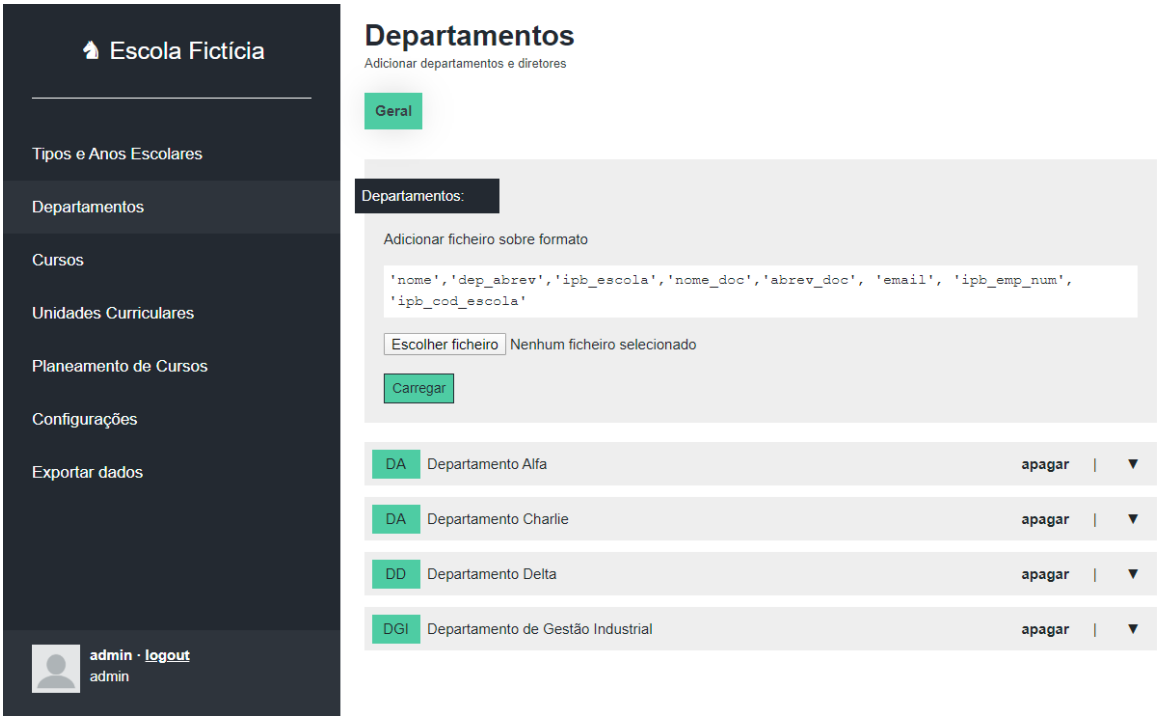


Figure 18: Login Page

When a department list is successfully inserted, the system generates a random password for each responsible teacher added, and returns it to the screen for later use. In this same page there are the features to delete or see extra information for each department. Deleting a department will lead to:

- Delete every teacher attributed to that department;
- Delete every course attributed to that department, and consequentially:
  - deleting every degree association made to it;
  - deleting every shift made to it.

Going further the list there is the Degrees page, as seen in figure 16. In this page it's possible to insert data both manually or through file uploading. For every courses added, there is the possibility to edit it - rendering a form such as the manually inserting one, but having already having the fields filled. When a degree is deleted the following occurs:

- Every association for that degree is deleted;
- If a shift contains that same degree, it'll be deleted; However if, that shift is shared with another degree, it'll be kept.

To test the system's error handling, a file containing a value already in database was loaded. As expected, the system didn't allow the insert, pointing out the fields in conflict. The figure 20 illustrates this instance.

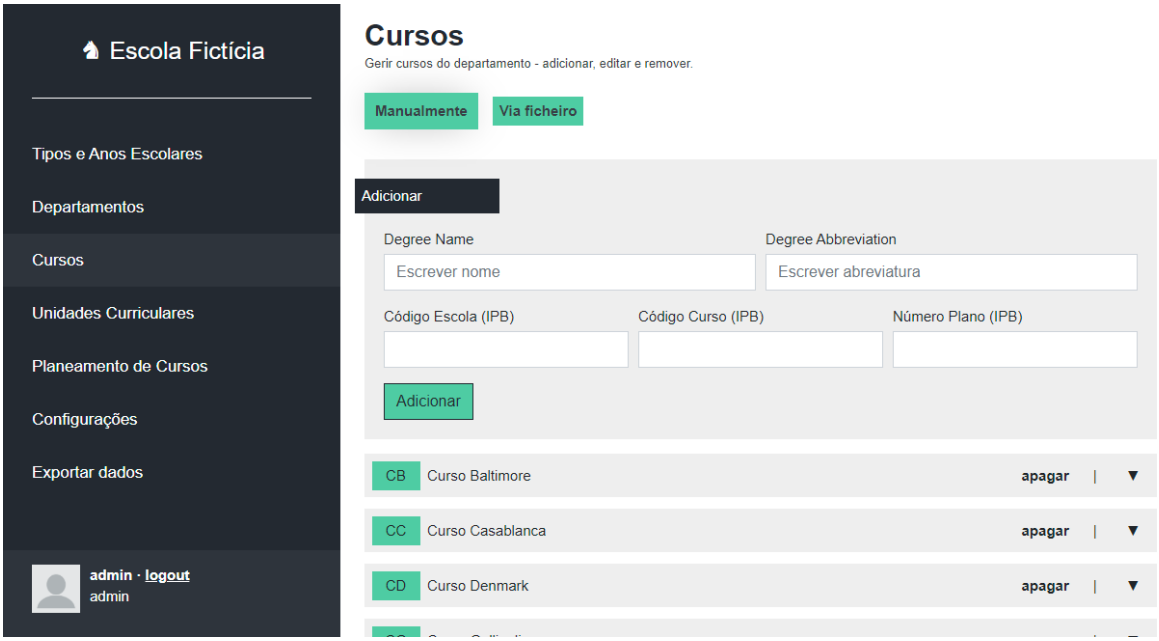


Figure 19: Degree management page

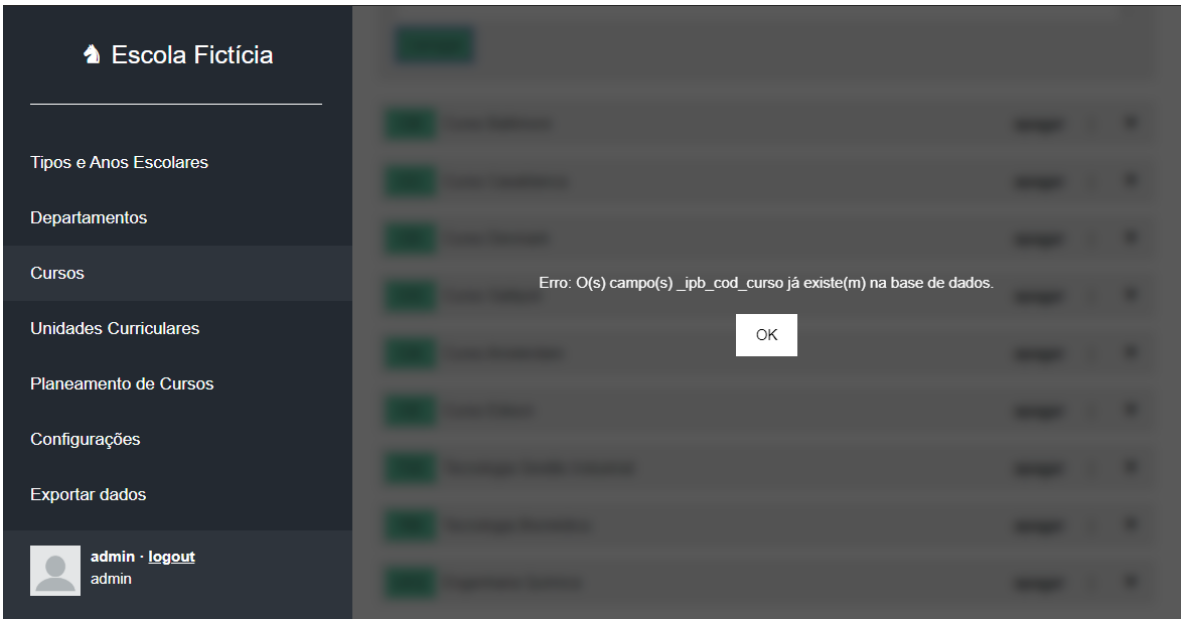


Figure 20: Degree management page - conflict

Next in the menu is the course management page, as the figure 21 shows.

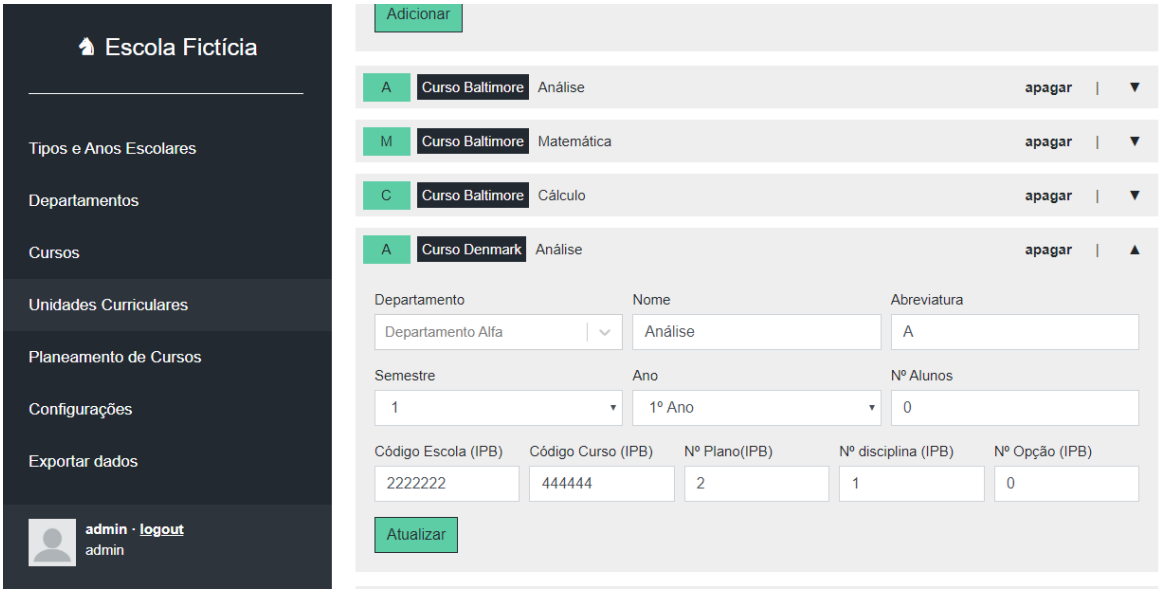


Figure 21: Course management page

In this page it is possible to add, edit and remove courses. The figure illustrates the editing of an existing course. Similarly, to add courses manually the input box to render is the same as editing - however, the fields are empty. It is also possible to insert batch data - through file uploading.

When a new course is submitted, the system analyzes if the degree code which is associated to already exists in the system - if it exists, it is inserted into the database; if not, the user is warned and not added. Removing a course will lead to:

- Delete every class created by the given course; Therefore, resulting in:
  - Deleting every shift created for each class;
- Delete every degree association for the given course;

Figure 22: Course associations page

The figure 22 illustrates the page which displays the associations between degrees and courses. For a given degree, year and semester, the pages displays the courses attributed for that selection.

Next on the available pages for the administrator, is the configuration page - as the figure 23 suggests.

There two sub-menus in this page, for each section:

- Versioning
  The administrator can consult, change or create a new version; If a version is deleted:
  - Every association and shift for that given version is deleted;
- Information
  The administrator can change the name that shows up in the sidebar; Currently it is *Testing*

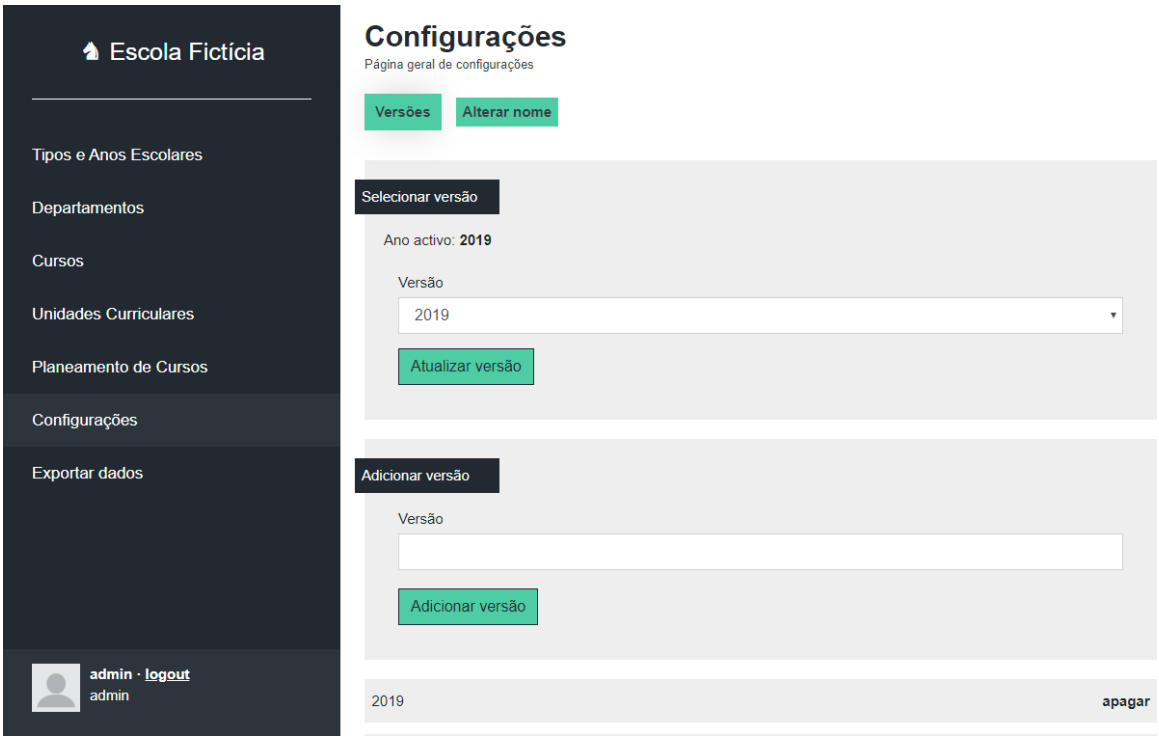


Figure 23: Configuration page

Lastly, there's the data export page for the administrator, as seen in the 24. Here, the administrator requests the system to generate a SQL file given every data worked in the system;

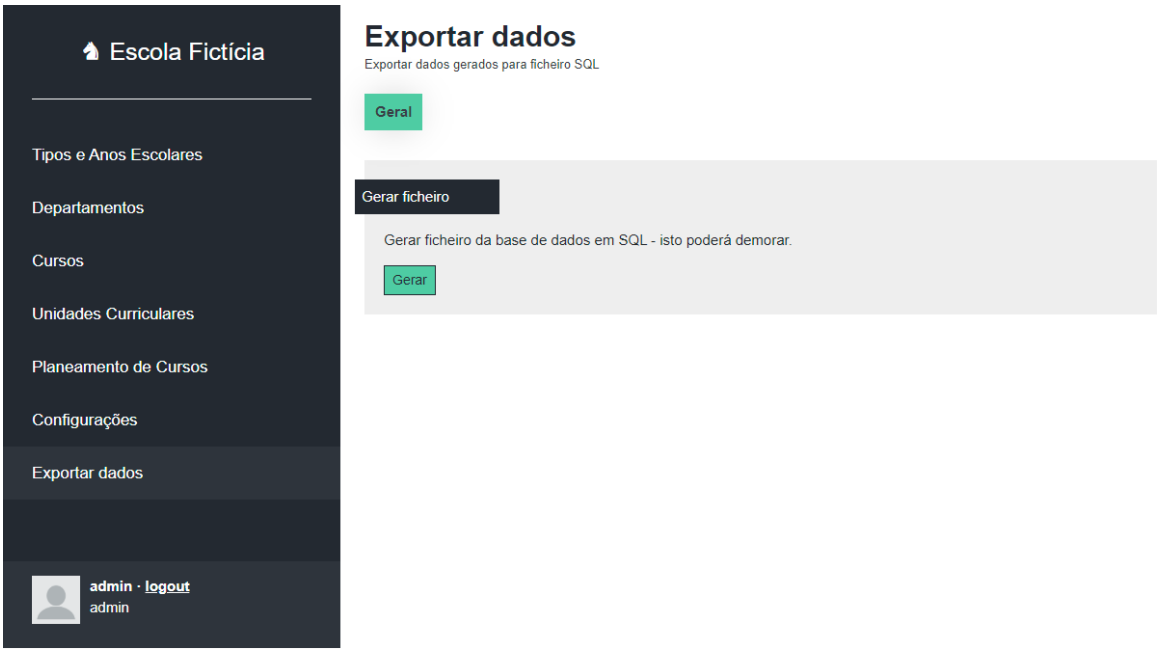


Figure 24: Data export page

5.2.3 *Head of Department*

The first accessible menu for the head of department is teacher management - as seen in 25. In this section, the user can insert teacher data either manually or via file upload, to batch upload. When a teacher is deleted from the system, the system will delete classes associated to them to avoid problems - and therefore shifts for that same class will also be deleted.

Figure 25: Teacher management page

The second item on the menu for head of department is class management, as 26 suggests. Four each course selected, the head-of-department can:

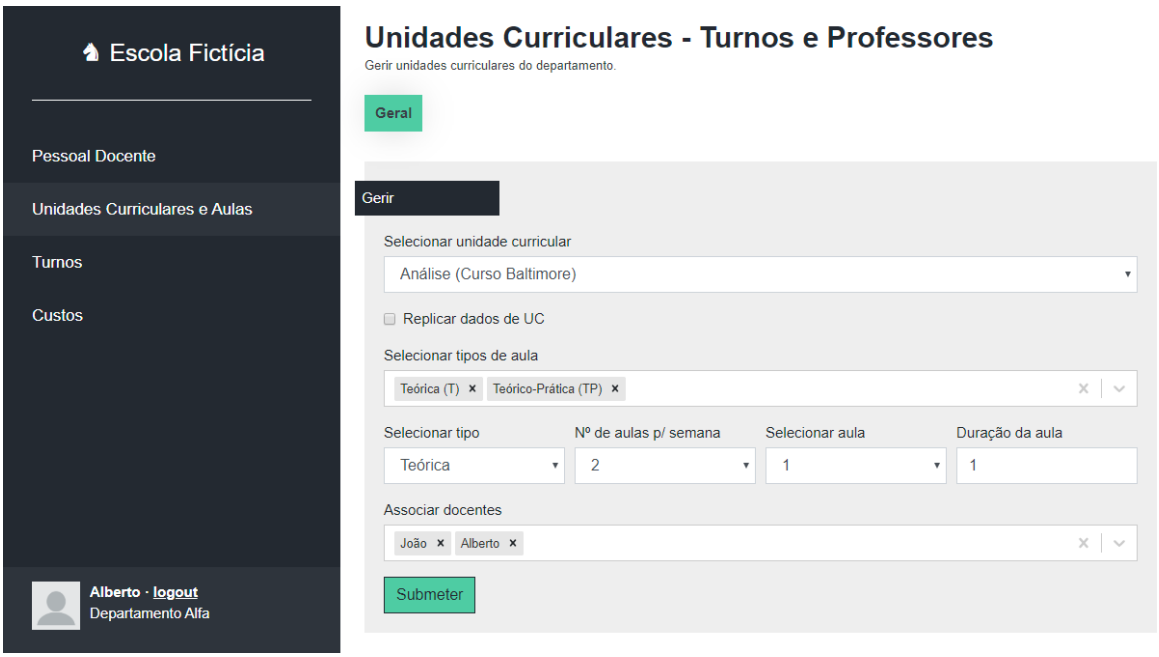


Figure 26: Class management page

- Set the types of classes existent for that course; For each type of class added:
  - The number of classes and the duration of each;

- Associate teachers for that course;
- Replicate the data inserted in the previous points via another course - as seen in the 28

If a class has already shifts attributed to it, the system wont let the user edit or delete it - to avoid data mismanagement.

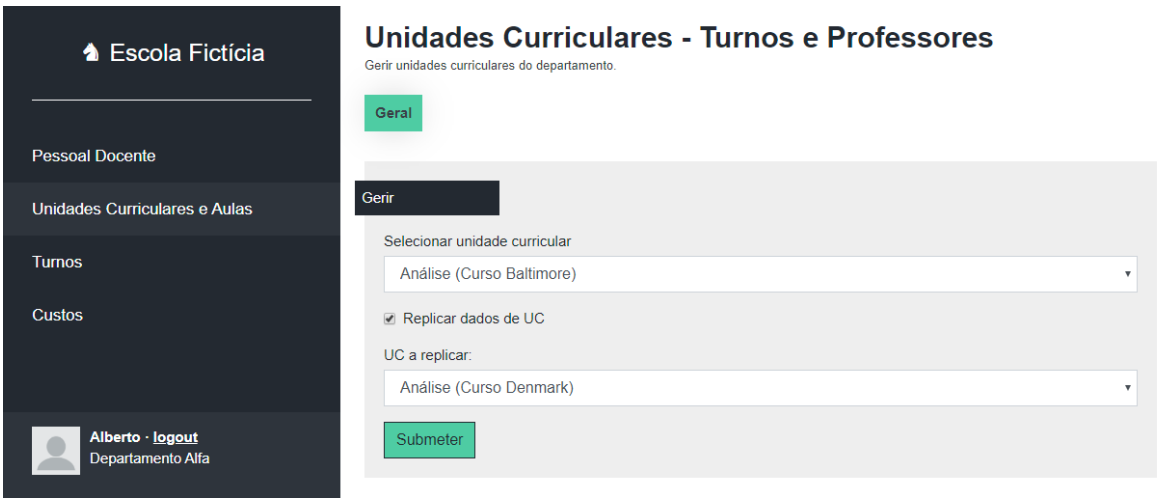


Figure 27: Class management page - data replication

Next on the list, there is the shift management page. In this page, for a class of a given course, the head of department can:

- Add a new shift;
- For each existent shift, it's possible to:
  - Select the teachers responsible for it;
  - Join it with another course:
    - For this, both courses should have the same attributed teachers and number (as well duration) of classes; Otherwise the system won't let the user join them.
  - Delete the selected course from this shift - if however the course is the only constitute of the shift, the system will delete the whole shift;
  - Delete the whole shift;

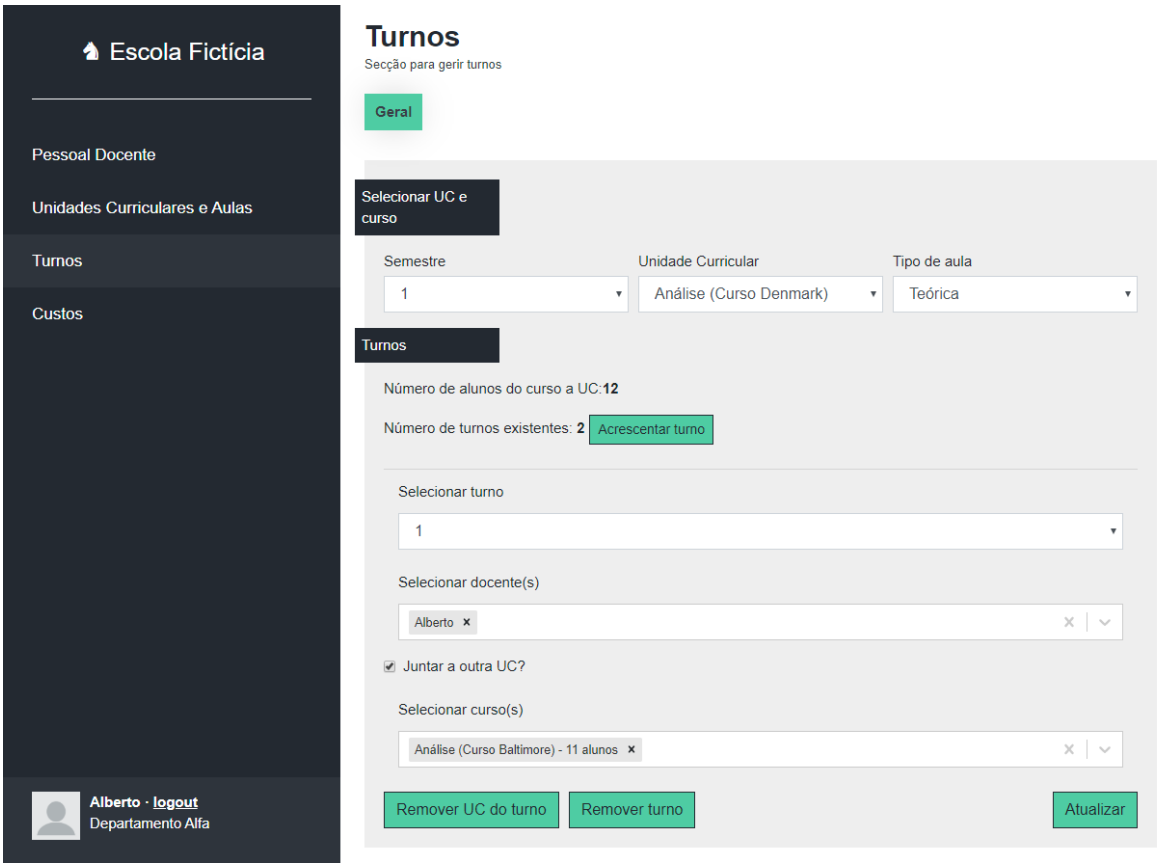


Figure 28: Data export page

The last option of the menu selection available for the head of department is Costs, as seen in 29. In this section, it's listed the number of weekly hours attributed for each teacher in the department. From that number, it is also possible to consult how it is distributed. For a quicker navigation, there are three ways to filter the list:

- by teachers with no workload i.e. zero weekly hours;
- by name or id;
- by number of hours.

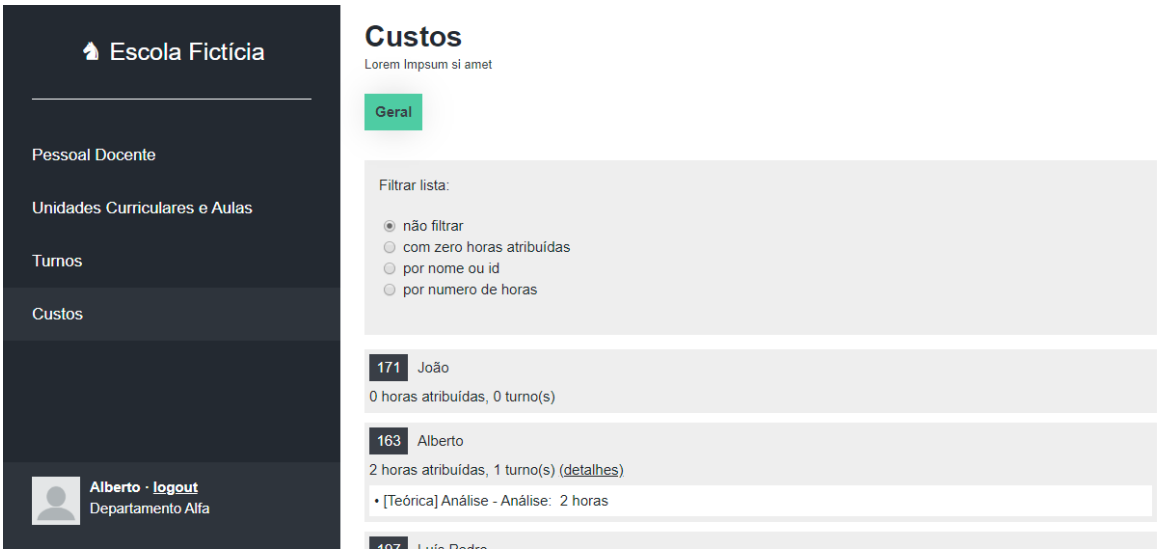


Figure 29: Costs Page

5.3 DISCUSSION

The results above allows a bigger perception of the system developed and to extrapolate some important aspects of its functionalities and features.

Reinstating the quality goals set before its implementation, it may be regarded as certain that the system fullfills all of them: the **simplicity** is evident, as every feature in the system offers a self-explanatory use; the features are well distributed and easily accessible without any bureaucracy, revealing its **efficiency**; the sleek and modern design displays a high degree of how **attractive** it is; and the all the testing made previously how much **testable** it got.

In the scale of `MoSCoW` previously used to set the priority of the requisites, it is possible to say with certainty that all the **must** requirements - i.e. level 1 priority - were implemented with success, as well a fair part of the **should**. Unfortunately a feature that wasn't implemented on time - the possibility for the teachers add their preferences - but further addressing on this subject will be done in the future work.

The main objective of this dissertation was to normalize the input of data into a system, which is accomplished. The main shared data is restricted from the beginning i.e. the types and academic years. With the use of types, every type has to be in conformity with those offered by the institution. This system assures that every data is inserted in the same way, as opposed as is currently done - a table done by each department with no normalization from department to department - which allows to save time not only establishing the teacher service, but also when translating these tables to SQL manually.

A big feature that aims in reducing the manual work for the users was implementing versioning. With versioning, it's possible to allocate a database version for each year. Every year, the main attributes of the system stay the same - departments, courses, degrees, teachers, classes and more importantly, the course plan each degree has - changing only the number of attendants and consequentially, the shifts for each class. Even if some changes need to be made to the more static components, they're easily manually tweaked with the features available.

With the feature **costs**, there is a bigger understanding by the head of department of should the distribution be done since it is detailed how much work there is attributed for each teacher.

6

CONCLUSION

The fundamental objective of this master project is to achieve a working quality software for Instituto Politécnico de Bragança (that could also work for other institutions). Aiming to reduce the constant wasted time on timetabling every semester, the thought web-application will be capable to gather data from every department when it comes to the documentation relating teachers, classes, degree programs and many domain-associated terminology - all this with proper data validation and restriction enforcing. It'll should also able to provide the possibility to teachers express their preferences with work-time and verify them along restrictions and previous stated data.

Throughout this document, and along a more detailed objective such as above, the problematic and offered solutions of timetabling were considered and tackled, moving then for the approach where the requirements were made explicit and prioritized, taking also in account the constraints, the thorough study of a domain and the solution strategy where the architecture and basic flow were exposed. The second stage of the dissertation takes account of development and testing of said system.

6.1 CONCLUSIONS

In consideration of the objectives initially established, it's a fair assessment to consider they were achieved in the majority. The imperative to achieve the minimum viable product was accomplished - data normalization through a web-app gathering data and generating an output to conclude. There were two main components - distribute teachers and courses after conceiving shifts for the different existent courses and teacher preference gathering. Unfortunately, the later wasn't developed in time.

The major lackluster in system developed was the failure to make it a broad enough to be used in any institution. This is a result from every institution having its own organizational structure and not being standardized enough - it's possible to recognize this situation e.g. in the database fields with the underscore prefix. One possible approach to solve this problem would be taking an extensive set of samples from different institutions, analyze the similar

fields and associations and build a model from that - the more specific components would be, for example, stored in `JSON` file (given an database that allows this datatype). Bearing in mind this is a solution solicited for a particular institution and the database was already established, this approach wasn't taken in consideration.

6.2 PROSPECT FOR FUTURE WORK

From what was previously state as conclusion, a discernible prospect for future work is the development of the feature that allows the teachers to state their schedule preferences. By virtue of the system's database and overall structure, this feature could be clearly be implemented. Every teacher inserted into the system get a randomly generated password, it'd be a matter of attributing each teacher the correct permission. Thereafter, create a SQL for storing the preferences.

Along missing preferences, the following parts would be taken in consideration for optimization:

- **Responsiveness**: Although the web-app works well in every device, some components could escalate better. For example, hiding the sidebar as the width decreases;
- **Safety**: The use of `HTTPS` by the web-app already protects the users information; however, an extra layer of protection wold be ideal - such as the use of password salting;
- **Documentation**: Making the documentation for the code implementation improved;
- **Restrictions**: Despite some restrictions being already placed - i.e. the academic years and class types - the system could benefit from more restrictions e.g. teaching hours;
- **Classrooms:** Although the classrooms preferences were considered in bargaining, it wasn't implemented - the implementation would go through append new tables to the database and associate classes to classrooms;

BIBLIOGRAPHY

[1] 2do2go. Json-sql: Node.js library for mapping mongo-style query objects to sql queries. https://github.com/2do2go/json-sql. [Online; accessed 05-01-2019].

[2] arc42. arc42 overview. https://arc42.org/overview/. [Online; accessed 26-10-2018].

[3] Carlos Mauri. 7 benefits of using sass over conventional css. https://www.mugo.ca/Blog/7-benefits-of-using-SASS-over-conventional-CSS. [Online; accessed 05-01-2019].

[4] Chris Lundquist. Pg2mysql converter. https://github.com/ChrisLundquist/pg2mysql. [Online; accessed 05-01-2019].

[5] CourseLeaf. Section scheduler. https://www.courseleaf.com/section-scheduler/. [Online; accessed 12-12-2018].

[6] João Miguel Fernandes and Ricardo Jorge Machado. *Requirements in Engineering Projects*. Springer, 2016. ISBN 978-3-319-18597-2.

[7] forever. forever. https://github.com/foreversd/forever. [Online; accessed 05-01-2019].

[8] GeeksForGeeks. What is full stack development? https://www.geeksforgeeks.org/what-is-full-stack-development/. [Online; accessed 05-01-2019].

[9] Google. Secure your site with https. https://support.google.com/webmasters/answer/6073543?hl=en. [Online; accessed 05-01-2019].

[10] Jeffrey Kemp. A simple data etl method – nothin' but sql. https://jeffkemponoracle.com/2011/02/simple-data-etl-method/. [Online; accessed 05-01-2019].

[11] Karolina Gawron. 6 main reasons why node.js has become a standard for enterprise-level organizations. https://www.monterail.com/blog/nodejs-development-enterprises. [Online; accessed 05-01-2019].

[12] log4js-node. log4js-node. https://log4js-node.github.io/log4js-node/. [Online; accessed 05-01-2019].

[13] miku86. Nodejs postgresql: Orm overview. https://dev.to/miku86/nodejs-postgresql-orm-overview-3mcn. [Online; accessed 05-01-2019].

[14] Naren Yellavula. A guide for adding jwt token-based authentication to your single page node.js applications. https://medium.com/dev-bits/a-guide-for-adding-jwt-token-based-authentication-to-your-single-page-nodejs-applicat [Online; accessed 05-01-2019].

[15] RESTfulAPI.com. What is rest. https://restfulapi.net. [Online; accessed 05-01-2019].

[16] Samir Ribić, Razija Turčinhodžić, Amela Muratović-Ribić, and Tomaž Kosarc. Redosplat: A readable domain-specific language for timetabling requirements definition, 2018.

[17] Sami Koskimäki. Objection.js: An sql-friendly orm for node.js. https://vincit.github.io/objection.js/. [Online; accessed 05-01-2019].

[18] Schedule My Teachers. Schedule my teachers. http://schedulemyteachers.com. [Online; accessed 12-12-2018].

[19] Tutorials Point. Node.js - express framework. https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm. [Online; accessed 05-01-2019].

[20] UniTime. University timetabling. https://www.unitime.org. [Online; accessed 12-12-2018].

[21] Irving van Heuven van Staereling. School timetabling in theory and practice. Master's thesis, Vrije Universiteit Amsterdam, 2012.

[22] Karl Wiegers and Joy Beatty. *Software Requirements*. Microsoft Press, 2013. ISBN 978-0-7356-7966-5.