\

Authentication and Authorization (Node Component)

Developed by:

Diogo Leandro

Tiago Matias

João Barata

Guiding:

João Pereira, GFI

José Simão, ISEL

Project report developed in the context of the course Projeto e Seminário

Degree in Informatics and Computer Engineering

20th of April 2020

Instituto Superior de Engenharia de Lisboa

Authentication and Authorization (Node Component)

44827 Tiago David de Brito Matias



44868 Diogo Botelho Leandro



44857 João Rafael Marques Barata



Guiding:

João Pereira, GFI

José Simão, ISEL

Project report developed in the context of the course Projeto e Seminário

Degree in Informatics and Computer Engineering

20th of April 2020

# Abstract

Nowadays there is a huge need to ensure and confirm the identity of users who want to access a particular application, especially in the context of application development. As such, this will be the topic addressed in our project.

Two important concepts to understand the utility and operation of our project are authentication and user authorization.

Authentication is the act of confirming someone’s identity. It is necessary for us to understand which authorizations the user has authenticated.

Authorization represents a means to condition access to certain private resources. Therefore, the goal of this project is to simplify and automate this entire process of authentication and user authorization.

*There is little point in doing a project that merely regurgitates the work of others. Your own though, ideas and developments are important, and these are what people reading your report*

*are interested in. Through your project you will develop not only your own skills, but also the ideas and work of others*.

Dawson, C., 2009, Projects in Computing and Information Systems, Second Edition, Pearson

Education Limited, Essex, England

**Table of Contents**

[Summary 3](#_Toc39593693)

[Chapter 1 6](#_Toc39593694)

[Introduction 6](#_Toc39593695)

[1.1 Important concepts 6](#_Toc39593696)

[1.2 Project specification 6](#_Toc39593697)

[Chapter 2 8](#_Toc39593698)

[Framing 8](#_Toc39593699)

[2.1 Costumers - Hiring Dashboard 8](#_Toc39593700)

[2.2 Implementation Requirements 8](#_Toc39593701)

[2.3 Similar systems 11](#_Toc39593702)

[Chapter 3 12](#_Toc39593703)

[Architecture 12](#_Toc39593704)

[**3.1** **Web Application** 12](#_Toc39593705)

[**3.2** **Api** 13](#_Toc39593706)

[3.3 Database 18](#_Toc39593707)

[Chapter 4 21](#_Toc39593708)

[Current state of implementation 21](#_Toc39593709)

[Chapter 5 21](#_Toc39593710)

[Tests and Conclusions 21](#_Toc39593711)

[5.1 API tests 21](#_Toc39593712)

[5.2 Testing the web app 22](#_Toc39593713)

[5.3 Difficulties and problems encountered 22](#_Toc39593714)

[Chapter 6 24](#_Toc39593715)

[WeboGraphy 24](#_Toc39593716)

[Appendix A 25](#_Toc39593717)

[6.1 Detalhes dos Endpoints 25](#_Toc39593718)

[Appendix B 29](#_Toc39593719)

[Application Programming Interface (API) user manual 29](#_Toc39593720)

[Milestones 30](#_Toc39593721)

**Table of Figures**

[Figure 1- Block diagram representative of the project structure 7](#_Toc39593644)

[Figure 2- Scheme to use our application together with Hiring Dashboard 11](file:////Users/diogo/Desktop/isel/ps/Authentication-and-Authorization-Node-Component-/Reports&Documentation/middle_report.docx#_Toc39593645)

[Figure 3- Demonstration of our "Get all users page" on the left, and our login page on the right 13](#_Toc39593646)

[Figure 4- SAML procedure 17](#_Toc39593647)

[Figure 5- OAuth flow 18](#_Toc39593648)

[Figure 6- Entity-association model 19](#_Toc39593649)

[Figure 7- Milestones Table 31](#_Toc39593650)

[Figure 8- Project plan 31](#_Toc39593651)

# Chapter 1

## Introduction

The application provides the user with a linear way to authenticate users, manage blacklist and greylist settings, assign roles and permission sets.

The project will have 3 main components. The server, the client application and the *authization* module components. The module will be responsible to create the interaction between our database and the application using our module. The server will be responsible for meeting a number of HTTP requests and call the respective methods of our module, that has the necessary methods to create/update/delete users, lists, positions and permissions from the database, always ensuring that whoever makes the request is authenticated and authorized.

On the other hand, the client application consists of a UI that uses the ReactJS framework, that has a login page with access to a backoffice to facilitate orders creation. In the backoffice will be possible to:

* Managing positions – the position system follows a hierarchical RBAC model, in which, depending on your position, a user will have different permissions (e.g. permission to put someone in a blacklist);
* Manage lists – At the moment only two types of lists are projected, grey list and black list. When a user is placed on the grey list, he will lose access to part of their permissions, if it’s blacklisted the loss of access will be total;
* Permission management – If there is sufficient permission to do so, the user may create new permission sets and even remove existing permissions;
* User management – Each user will one be able to edit their information (username, password), except for extreme cases where an administrator has to intervene and he changes the credentials of a particular user. This feature will not be available to authenticated users through identity providers (idp’s).

We consider it important to mention that with this project we decided to prioritize the **decoupling of our modules** so that the user can choose the objective he intends to give to our project. This way, the user can use our application programming interface (api) without having any implication or obligation with our client application, allowing them to create their own application to consume our api.

#### Project specification

The project consists of several blocks as shown in Figure 1, in a very succinct way, it contains a web application, a server and a database.

A picture containing object, clock

Description automatically generated

Figure 1- Block diagram representative of the project structure

###### Database

The Database block is used to save information from the various users and information regarding access control policy required for the authorization layer.

###### Server

The Server block consists of our settings management api, our User layer, authorization and data access layer. The api receives requests from a costumer and forwards these requests to the right tier and can go to the user management or authorization layer. Both of them can enter to the data access and use it to be able to fulfil the requests that were made to the api.

###### Module

The Module block, handles all communication with the database, ensuring that all request are authenticated as supposed. In order to achieve authentication in both our endpoints and the third-party’s endpoints, alternatively the module supplies a authentication middleware, that ensures that third party endpoints are only accessed by authenticated users which have permission to access that endpoint.

###### Web application

The Web Application is a back office that contains a user interface to manage all the settings available in our api.

Its main objective is to make simple and interactive all the management of user settings. It also has a great versatility because most of its functions are configurable, from the protocols to be used for authentication to the roles and available permissions to each user.

# Chapter 2

## Framing

Given the high interest in integration by other modules, not only by GFI, but also by third parties, the project was built with high focus on utilization and user experience.

#### Costumers - Hiring Dashboard

In order to provide an example of possible client of our application, we were given the chance to collaborate with Hiring Dashboard project of the group 20 of Project and Seminar. It was decided to progress with the implementation, having the collaboration proved symbiotic, both by our group, that has a “real” client with technical and specific requirements that might not have been implemented before, and by the other group that will not have to invest time and effort to develop an authentication module and that will have a module much more neat than would be expected by the requirements of their project.

#### Implementation Requirements

To use our application, it is necessary to have a database and the express module. The database and express must be passed as an argument when initializing our application. The entities will then be automatically created and some middlewares will be added to the express. The client must also install all the dependencies using the command npm install on the root folder of the project.

Considering that our project can be divided into two completely decoupled applications, it is important to mention all possible scenarios, these being:

* Use our module for a costumer web application;
* Use our back office for a client api;
* Use both components referred in this combined project.

To use the first scenario (Use our module for a client web application) just need to have our module on your project and require it with the express and the database arguments.

After that, all you need to do is call one of the methods of our module

The second scenario (Use our backoffice for a client api) involves some work from the client, because there are some requirements that will need to meet for everything to work. First you will need to insert the endpoints of your api in the links.js file, present in the root folder of our project. This will cause the requests to be made to your api instead of ours. Next it will be necessary that the api has at least the endpoints that ours has, otherwise the application will make requests that will not be satisfied. Finally, the api must return data in json with the same structure as the data in our data model, so that it can be represented in the application. Due to all these requirements, it is a scenario not very advisable because, although our two applications have been decoupled, they made to have compatibility and efficiency between them.

The third and final scenario (Use both components mentioned in this coupled project) is the most advisable and perhaps the easiest to put into practice because it is enough to run our server locally and, by default, when accessing the url http://localhost:8082/ will go to the homepage of our application. From there it will be guided by our user interface through buttons to other pages.

Figure 2 shows an example of the integration of our project with the Hiring Dashboard client, in which the user selects a loginbutton and will be redirected to our login client page, where he will authenticate, having then the option to return to the integral application or access the back office provided by our application.

In addition to the web application, it is also possible to use our API. Details of the different endpoints will be demonstrated later in Chapter 6.

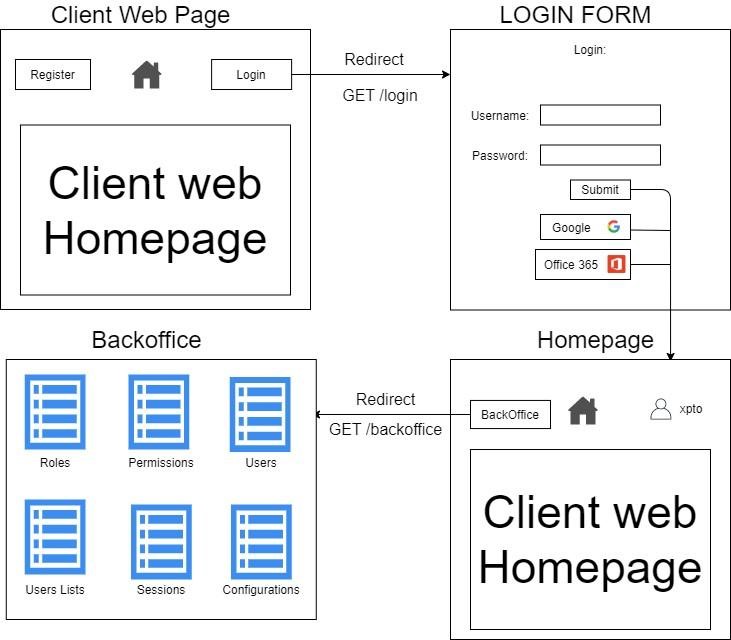


Figure 2- Scheme to use our application together with Hiring Dashboard

Having been correctly authenticated the user will then have access to the server endpoints. In back office, endpoints are accessible through a UI that allows you to place requests for all endpoints. Optionally, the user will be able to return to the page of the integral application and it will be the responsibility of that application on how it would access the different endpoints. Attached endpoints will be available to the full application.



#### Similar systems

A similar system to ours and currently present in the market is the keycloak in the way that is also open source and also provides an identity and access management.

To what extent are we introducing something new?

Beyond the basic differences, such as the different platform, since keycloak uses java and we use javascript and nodejs as interpreter, we also have some differences to highlight.

The presented points below are some of the features that our project has and the keycloak doesn’t:

* Hierarchy of roles;
* Black and grey lists;
* Notifications regarding user logins in lists;
* User’s historic;
* Providing customer identifiers and client secrets for the utilization of identity providers.

# Chapter 3

## Architecture

In order to synthesize the explanation of our architecture, we divided this point into 3 parts, these being:

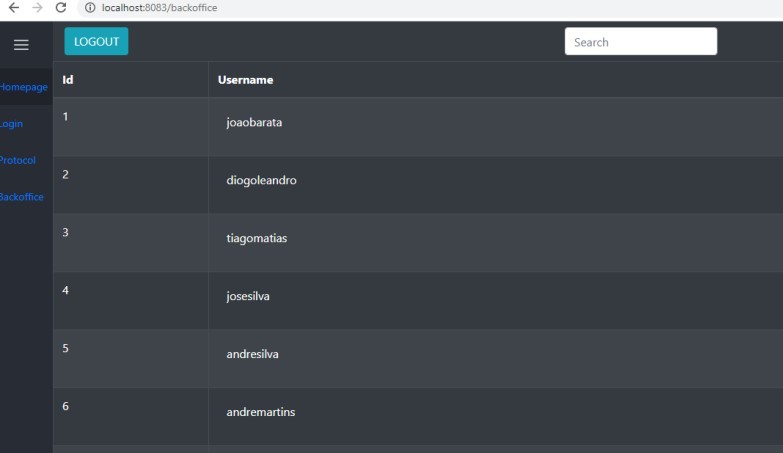
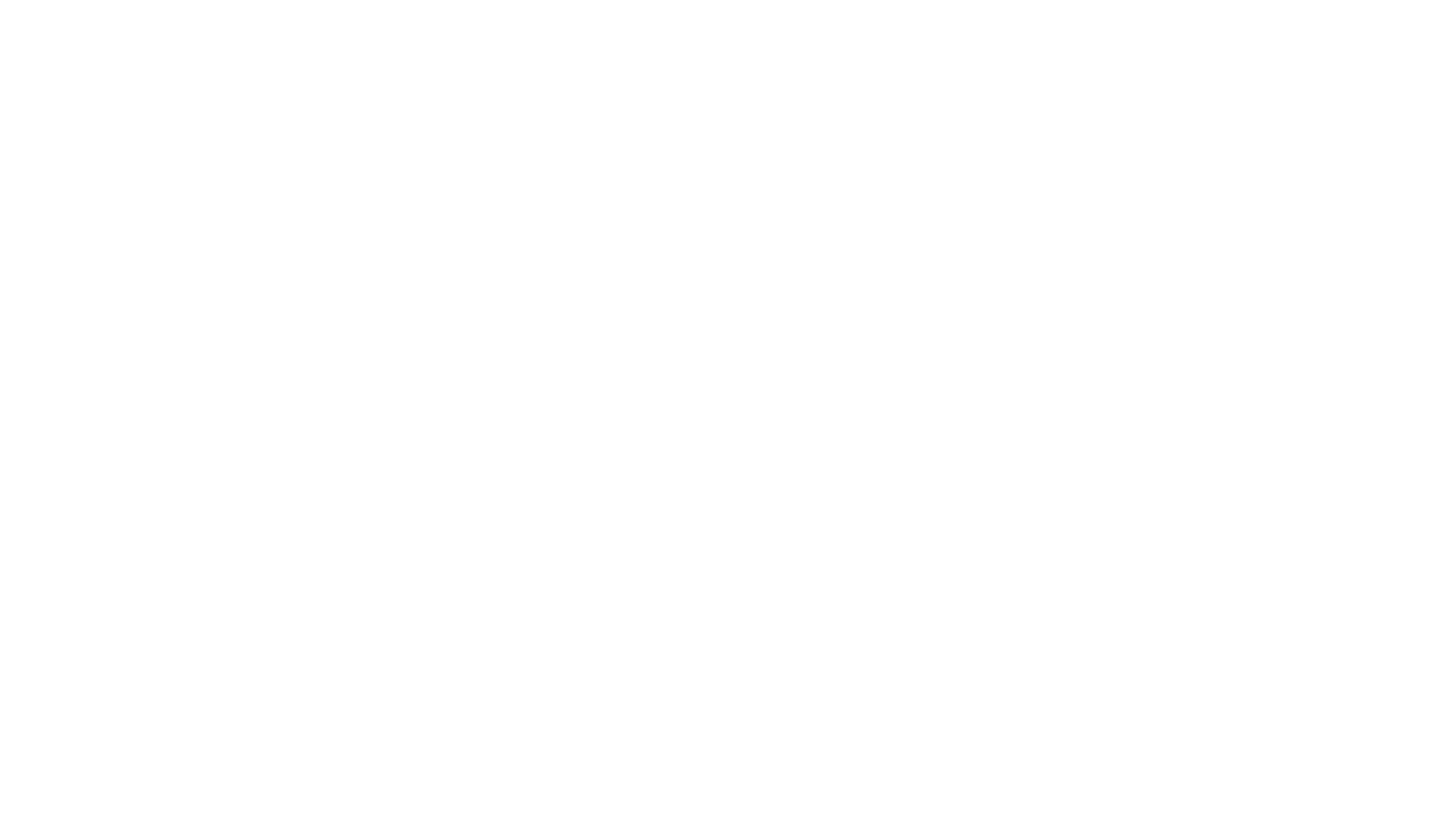
A screenshot of a cell phone

Description automatically generated

##### Web Application

Our web application is a SPA (Single Page Application), consists only of a page that changes its representation several times throughout the life of the application. We decided to choose a singlepage application instead of a multiple pageapplication because they tend to be faster due to most of its resources being loaded only once. We use the ReactJS library, because we think it is important to use this technology since it is currently one of the most used technologies for the elaboration of user interfaces in the market.

The package/web application module consists of a ***front-end*** application that aims to design a User Interface (UI) for access to login and back office features. To meet the design requirements and expectations of the industry both pages (login and back office) were built with the Bootstrap framework as you can see in Figures 4 and 5.



*4*



*-*



Figure 3- Demonstration of our "Get all users page" on the left, and our login page on the right

For ease of navigation and access as well as ease of development, it was decided to use the dependency also developed in react "@trendmicro/react-sidenav", which, together with the native component of react “React.Router”, made possible the creation of a useful and elegant sidebar.

##### Api

This block uses the Javascript language and nodejs as an interpreter.

As usual in api's nodejs, we used enough dependencies in order to save time and work on topics that are already covered, in a very satisfactory way by various tools in the market, as such, we will list the used tools and the reason for their use:

"body-parser": "^1.19.0",

"cookie-parser": "^1.4.5",

"cookie-session": "^1.4.0",

"cors": "^2.8.5",

"express": "^4.17.1",

"express-mysql-session": "^2.1.4",

"jsonwebtoken": "^8.5.1",

"lodash": "^4.17.15",

"mariadb": "^2.2.0", "node-fetch": "^2.6.0",

"passport": "^0.4.1",

"passport-azure-ad-oauth2": "0.0.4",

"passport-google-oauth20": "^2.0.0",

"passport-local": "^1.0.0",

"passport-office365-oauth2": "0.0.3",

"passport-openid": "^0.4.0",

"passport-saml": "^1.3.3"

We highlight the express for its simplicity and efficiency. It allows us to start the server and forward requests to the right router, making it easy for us to define the paths/endpoints.

We also highlight the passport that was chosen by us for its versatility and definition of strategies, since our application supports different authentication protocols. The size of the code base would be huge if written from scratch. The use of the passport allows us to create an authentication strategy for each protocol supported and in addition makes quite simple the tasks of login and logout.

To manage user settings and permissions we will use a Role-based access control (RBAC) by later setting the different roles and their permissions.

###### API structure

Our api contains an index.js file that is the entry point of our server. This file imports the module and all its dependencies and middlewares used and runs the server so that it is listened to by requests.

Each entity in our data model has a folder that has all the files referring to the entity. In most cases there are two, the entity's router file and the data access layer file of that entity. The router file receives the requests from the server and forwards them to the correct data access file, then receiving a response and sending it formatted to the client.

###### API functionalities

Our api is the central part of our project as it is the intermediary between our web application or any client application and our database.

This api is dedicated to managing user settings of a back office but, if our instructions are followed in the user manual (Appendix A) and our data model in 3.1.1, it can be used by any client application that requires user authorization and authentication management.

Our features are:

* Creation and editing user’s username and password;
* Insertion of new roles;
* Creation of new permissions;
* Binding permissions to roles;
* Association of users to roles;
* Blacklist management;
* Storage of user’s actions historic.

###### Supported authentication protocols and identity providers

At this time our application supports saml and oauth protocols using openid as identity layer.

We also support authentication using identity providers office 365 and google.

For a better understanding of our application we will make a brief summary of each protocol.

###### Saml

In the Saml protocol there are 3 main entities: :

* The identity provider - Entity responsible for creating and saving entities on a given user;
* The user - user who wants to log in through certain idp (identity provider) compatible with saml protocol;
* The services provider - Application in which the user wants to authenticate.

The saml protocol uses xml Documents with digital signatures to send information from a particular identity provider user to the service provider.

The figure below shows the procedure of a client by authenticating using the Saml protocol.

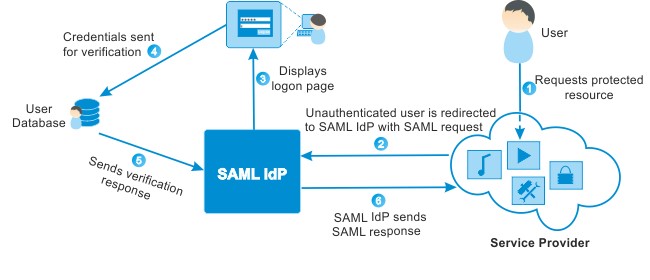


Figure 4- SAML procedure

###### OAuth 2.0 e OpenID Connect

OAuth 2.0 is an authorization protocol that provides specific authorization flows for all types of applications.

OpenID Connect 1.0 is a simple layer of identity that is used in together with the OAuth protocol. It allows clients to verify the identity of users based on authentication made by an authorization server, and it also allows to obtain basic information about the user's profile.

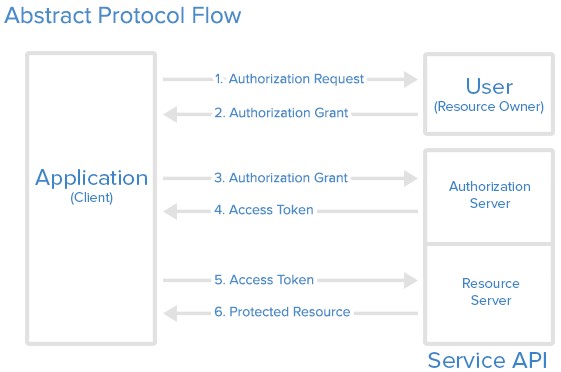


Figure 5- OAuth flow

In addition to these types of authentication the user can choose to register in our applicationby providing the application with a unique username and password.

Our authentication strategies, except the local strategy (username/password authentication via session cookies), when receiving a request search in the database for that user you want to authenticate. In case you find any registration the login is done, otherwise a new user is registered.

It should be noted that a user registered through of one of the identity providers will only be able to login using the same strategy.

#### Database

Our database is relational, uses the sql language and as a database management system we chose to use mariadb. This choice was based on the fact that our main client (GFI) uses mariadb in its client application, so we would make a lot easier to join the projects creating greater compatibility with the used tools.

In a first approach the group decided to handle communication with the data base through a simple mariadb dependency which provided a basic way of establishing connection with the data base and make direct queries to it. Later in development the group members together with the advisors reached a consensus that it would be a viable alternative to use an ORM(Object-relational mapping), for this purpose the dependency *sequelize* was chosen. With sequelize there were no need to manually make all the querying, as it automatically generated all desired queries and functionalities based on the provided model. Sequelize made it easier to implement compatibility with other database management system such as compatibility with postgres which was implemented since it was useful to our other main customer, the Hiring Dashboard team.

The database structure was designed to follow the RBAC model (Role-based access control) more specifically the RBAC1 model that corresponds to the RBAC model with role hierarchy. We intend to use a tool for encryption of passwords, that will be defined and implemented until the next delivery.

Our data model is as follows:

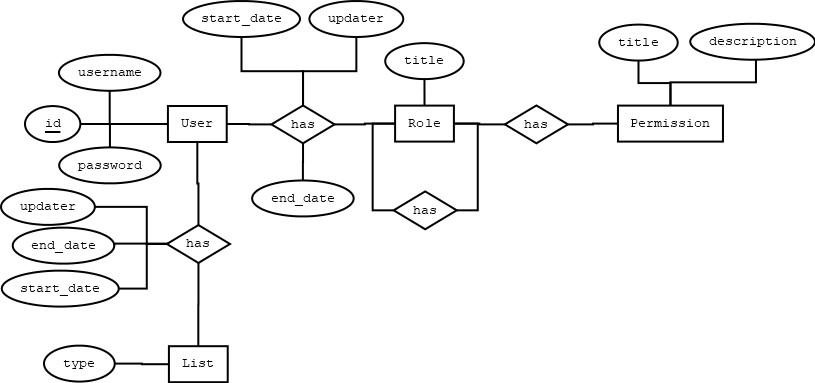


Figure 6- Entity-association model

Entities:

User- Entity responsible for saving user’s information.

List- Users associated with this entity will have restricted access to certain resources, this restriction being total (BLACKLIST) or partial (GREYLIST). Finally we have a Boolean variable that allows you to distinguish the active constraints from the disabled constraints.

User\_History- This entity will be responsible for saving all changes to a user's information and their logins and logouts. For each entry in the table will be saved the date on which it occurred and a brief description of the action occurred and possibly some additional explanation.

Roles- The purpose of this entity is to save information about a particular role.

Permissions- This entity is responsible for saving information about a particular permission. For our implementation we considered that a permission would be based on a particular http request. For this to happen, the Permissions entity will be saved the method and the respective path for a given resource.

Roles\_Permissions= Entity that associates the Role entity with the permissions entity.

Users\_Roles= Entity that associates the Role entity with the users entity.

IDP= Entity responsible for saving information about an identity provider and associating it with a user.

# Chapter 4

## Current state of implementation

At the present moment we have a fully implemented module and have a well-defined data model. As such, and by the deadline of delivery of the final project, our efforts will be focused on security issues and on continuing and finalizing the realization of the web application.

# Chapter 5

## Tests and Conclusions

###### **API tests**

To test our api we used the mocha testing framework and the supertests framework to simulate api requests. .

These tests consist of requests with test data directed to our test database so as not to harm any real data.

Tests wait for a response from the api and use the nodejs assert library to verify that the responses match the expected.

* 1. Testing the web app

Tests yet to be implemented.

* 1. Code Analysis

To verify that the code produced respect all usual code good practises and that it follow all rules that are industry standard we decided to use code analysis tools such as SonarQube and SonarLint. SonarQube is one of the most well known tools for code verifications allowing developers to check their code quality according to SonarQube’s parameters like bugs, code smells, vulnerabilities, duplications and tests coverage. SonarLint is a IDE plugin based on SonarQube which provides to the user the ability to apply SonarQube’s rules in real-time as he is writing code and allowing him to customize the rules that are applied in real-time checks, to make sure our code complies with market requirements we selected all rules available.

###### **Difficulties and problems encountered**

**Conflicts in the implementation of the kerberos strategy in the passport module:**

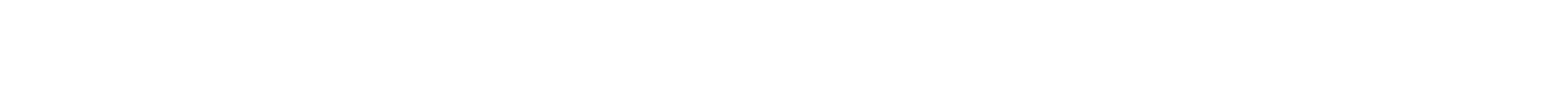
In searching for a module that implemented the kerberos protocol and that had compatibility with nodejs and passport we found several problems.

Most of the libraries that we found need extra dependencies and don’t document this necessity which led us to waste a lot of time with this protocol.

At the moment we do not support this protocol and do not intend to due to the difficulty it presents in testing and putting into practice and the poor use in general compared to other protocols.

The libraries we tried were as follows:

* kerberos - https://www.npmjs.com/package/kerberos
* simple-kerberos- https://www.npmjs.com/package/simple-kerberos



* passport-kerberos - https://www.npmjs.com/package/passport-kerberos
* node-krb5 - https://www.npmjs.com/package/node-krb5
* passport-negotiate - https://www.npmjs.com/package/passport-negotiate

**Conflicts in implementing integration tests trough the mocha library:**

In carrying out api integration tests, we found several problems due to tests of mocha library not having a specific order of execution, and since our code is asynchronous it becomes quite difficult to test our api.

For the next phase we will try to use the tape library to run the same tests.

**Problems with the express-session:**

The express-session library began to provide a warning about the MemoryStore not being intended for use in production environments. This warning was solved with the addition of the mssql-session-store library that automatically stores all sessions in our database.

**Problems with different source authentication:**

In the past we have tried to have the web application running on port **3000** and the api

Running on port **8082** and by trying to execute the login request for our api we noticed that after the user authenticates, all subsequent requests lose the session posteriorly created. After adding the header credentials: “include” in the response and indicate that the session accepted requests from different sources, the attempts were in vain and we chose to follow a different approach. At this time, we serve the web application listed in the folder webapp/build as static path to our server. This way we lose some decoupling but not all because both applications are able to run alone.

**Problems when implementing different dbms compatibility:**

# Chapter 6

## Bibliography

Projects in computing systems

Writing for computer science

**Webography**

https://oauth.net/2/ https://openid.net/connect/ React Documentation -

https://reactjs.org/tutorial/tutorial.html

How To Write College Essays - http://ieeeauthorcenter.ieee.org/wp-content/uploads/Howto-Write-for-Technical-Periodicals-and-Conferences-1.pdf https://www.keycloak.org/documentation https://www.npmjs.com/package/mssql-session-store

Appendix A

#### Detalhes dos Endpoints

GET http://localhost:8082/api/users

Requires: Authentication.

Returns a list with all the users presente on the database. response: [

{

id

:

1

,

username

:

"exampleUsername1"

{



**POST http://localhost:8082/api/users** Creates a new user and adds it to the database.

body: {

username : "newName",



password : "newPass"

}

GET http://localhost:8082/api/users/:id

Returns a user object with its information (username and password).

If no user with this id is available on the database it returns with a 404 error (No user found).

**DELETE http://localhost:8082/api/users/:id**

Deletes user with specified id if one exists on the database.

PUT http://localhost:8082/api/users/:id/username

Updates user’s username

body: {

|  |  |  |
| --- | --- | --- |
|  | username | : "UpdatedName" |
| } | | |
|  | | |

PUT http://localhost:8082/api/users/:id/password Updates user’s password body: {

|  |  |  |
| --- | --- | --- |
|  | password | : "UpdatedPass" |

}

GET http://localhost:8082/api/lists



Returns all lists on the database, if no list is available it returns a 404 (List not found error).

**POST http://localhost:8082/api/lists** Adds a new list to the database.

body: {

user

LIST : "NewList", start\_date : "2020-04-

:

6

,



09

02:55:05",

end\_date : "2020-06-

09

02:55:05",

updater : 10,

}

**DELETE http://localhost:8082/api/lists/:id** Deletes specified list.

**GET http://localhost:8082/api/lists/active** Returns active lists on the database.

**GET http://localhost:8082/api/lists/active/user/:id**

Returns the active list from specified user, if he is not associated to a list it returns a 404 (List not found error).

**GET http://localhost:8082/api/roles**

Returns all roles on the database

POST http://localhost:8082/api/roles

Adds a new Role to the database.

body: {

role : { id :

role : "admin", parent\_role : 2

5

,

}}

DELETE http://localhost:8082/api/roles/:id

Deletes role specified by id

**GET http://localhost:8082/api/roles/:id** Returns the role with specific id.

**GET http://localhost:8082/api/users-roles** Returns all roles associated to users.

POST http://localhost:8082/api/users-roles

Adds a new user\_role.

{ user : 1,

:

2



role

}

**GET http://localhost:8082/api/users-roles/active** Returns all active user roles.

**GET http://localhost:8082/api/users-roles/active/user/:id** Returns the specified user’s active roles.

GET http://localhost:8082/api/permissions

Returns all Permissions available on the database

**POST http://localhost:8082/api/permissions** Adds a new Permission to the database

body: {

: "GET",

method

:

path "/homepage"



}

DELETE http://localhost:8082/api/permissions

Deletes Permission.

body: {

: "GET",

Method

:

path "/homepage"



}

POST http://localhost:8082/api/roles-permissions

Adds a new role\_permission association.

body: {

: 2, role permission :



1

}

DELETE http://localhost:8082/api/roles-permissions

Deletes a roles permission association

body: {

: 2, role permission :



1

}

**GET http://localhost:8082/api/authentications/login/google**

Login using google identity provider

**GET http://localhost:8082/api/authentications/login/saml**

Login using SAML protocol

**GET http://localhost:8082/api/authentications/login/azureAD**

Login using office 365 identity provider

POST http://localhost:8082/api/authentications/login

Login using username and password on the body **POST http://localhost:8082/api/authentications/logout** Logout from our application

Appendix B

### Application Programming Interface (API) user manual

To run our application must be followed the following steps

1. Clone heroku repo using

heroku git:clone -a APP-NAME

APP-NAME is authentication-authorization

1. Correr npm install na root folder do projeto
2. Run data model in your database, data model available in /database folder 4. Manually configure the json file present in the

pasta/server/config/production.json

This file is intended to enter the information necessary for the database and for authentication with identity providers (client id's and client secrets).

We also support configuration through endpoints available in our documentation. 5. Npm run start to run the application

Milestones:

|  |  |
| --- | --- |
| Milestones | Percentage of completion |
| Module app and flow skeleton construction | 100% |
| Planning and development of the data access system (MariaDB relational database) Data Access Layer Development | 100% |
| Performing the authentication module (user login and register) using passport | 100% |
| Manage and control resource access permissions by certain users | 100% |
| Management of different sessions of the same user in different clients. | 0% |
| Gestão de listas negras e cinzentas | 80% (needs testing) |
| Construction, structuring and testing of the API dedicated to configuration management | 80% (needs testing) |
| Authentication enabling compatibility with protocols (openId, SAML) and idps (google,  office365) | 80% (needs testing) |
| Framework study for web application (React) | 100% |
| Web application implementation (backoffice) | 50% |
| Test web application | 0% |
| Notifications to administrators when a banned user's logins occur | 0% |
| Development of Data Protection Measures  (Let's Encript) | 0% |
| Registration of user actions | 80% (needs testing) |
| Save required information from deleted users  (GDPR) | 0% |
| Test and protect app against Xss and other possible attacks | 0% |

Figure 7- Milestones Table

**Project Plan**

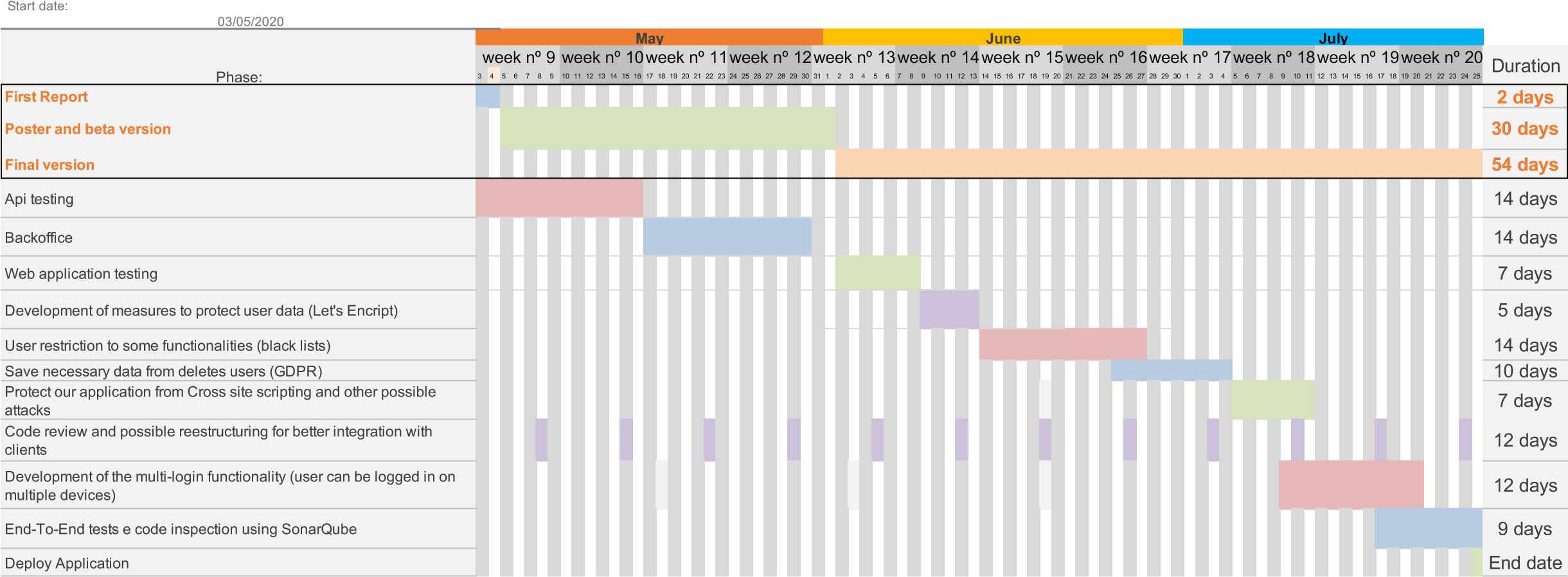


Figure 8- Project plan

|  |  |
| --- | --- |
| Semana | Work Done |
| 8-14 March | Project Proposal |
| 15- 21 March | Database structure initialization, basic working passport app implementation, react front-end initialized, login endpoints and login interface created. |
| 22-28 March | Deployment of application to Heroku so all group members could access same database, DAL implementation connection with database implemented, bug fixes, google IDP integration initialization. |
| 29 March – 4 April | Error handling, EA model improved, react Router implemented, login system improved. |
| 5-11 April | API continued (endpoints added), DAL continued, Redirect functionality added to UI. |
| 12-18 April | UI functionalities implemented (user and list management), Tests implemented, SAML protocol initialization. |
| 19-25 April | User history system implemented, Office 365 IDP integration initialized, different passport strategies implemented for each IDP or auth method, new endpoints added (configs endpoints) and user session system initialized. |
| 26 April – 2 May | React deep study and embracement of its lifecycle, SAML and Office 365 protocols continued, endpoints added and better restructuring for 3rd party integration. |
| 3-9 May | Better error handling, bug with CORS requests fixed, code clean-up and module created for decoupled integration. |
| 10-16 May | Error handling restructured, resolved authentication problems, react context API implemented and dynamic logout system implemented, async model reviewed, [more simplified setup mechanic](https://github.com/dleandro/Authentication-and-Authorization-Node-Component-/commit/3134d085938db8e8b03d6959d925bf60a4ef82b9) of module, SonarLint and SonarQube code revision, Postgres compatibility implemented and user session improved. |
| 17-23 May | Migration of communication with database from direct queries to Sequelize ORM and tests to this new system implemented, RBAC dependency implemented. |
| 24-30 May | Final Report draft |
| 31 May – 6 June | TBD |
| 7-13 June | TBD |
| 14-20 June | TBD |
| 21-27 June | TBD |
| 28 June – 4 July | TBD |
| 5-11 July | TBD |
| 12-18 July | TBD |
| 19-25 July | Project Deployment |

Figure 9- Weekly working history