Surf Club Management Application

Authors: Bernardo Fragoso
Gonçalo Albuquerque
Miguel Sousa

Advisors: Filipe Freitas
Miguel Pires, ESC

Project report carried out under the Project and Seminar
Computer Science and Computer Engineering Bachelor's degree

July 2022

Instituto Superior de Engenharia de Lisboa

Surf Club Management Application

47203 Bernardo Filipe Martins Fragoso

a47203@alunos.isel.pt

47265 Gonçalo Jorge Carvalheira de Albuquerque

a47265@alunos.isel.pt

47270 Miguel Gustavo Beirão de Oliveira e Sousa

a47270@alunos.isel.pt

Advisors: Filipe Freitas

ffreitas@cc.isel.ipl.pt

Miguel Pires, ESC

miguel.toscano.pires@gmail.com

Project report carried out under the Project and Seminar
Computer Science and Computer Engineering Bachelor's degree

May 2022

Abstract

Managing an association and its members can be a difficult effort, especially without the right tools, one of these examples is Ericeira Surf Club [1], that uses a simple excel sheet to manage their members.

This approach can be a handy option with a simple application, however as the association grows, it becomes a less feasible option due to lack of scalability. Besides making the maintenance of members a long task and inefficient, it also makes the implementation of new functionalities difficult.

In addition to replacing the excel sheet which only managed its members and their personal information, it is now feasible to acquire member engagement, thanks to the future availability of a system that will allow members to view their profile and receive club related notifications, such as event or payment reminders.

A management application enables the creation of a shared infrastructure for achieving compliance with business policies, resulting in improved results in terms of both goal execution and budget management.

One of the services that our project offers in addition to the application is a digital membership card, which allows a club member to obtain discounts at partner stores. Member verification is done through a QRcode present on the card, upon scanning the code the store representative is presented with a quick overview of the member's state.

The main aim of this project is to solve this problem creating a more modern environment, bringing more exposure to the company, and allowing a simple and easier management of their members.

Keywords: Management Application, Membership Card, Email Notifications, Single Page Application, Relational Database, Web API.

Contents

List of Figures

Listings

Chapter 1

Introduction

The following chapter introduces the motivation behind the project and also the objectives and main functionalities that are to be developed.

1.1 Motivation

Administering an organization as well as its members without the proper tools can be challenging. Ericeira Surf Club is one such example, which manages its members using an excel spreadsheet.

This method is straightforward to begin with and may appear to be functional, but as the organization begin to grow, so will the excel sheet, making it more difficult to handle member management and also restricting the implementation of new functionalities.

The members may now access their profile and receive club-related notifications, such as events or payment reminders, in addition to the replacement of the excel sheet that solely controlled their members and their personal information.

As each member has their own quotas and sports to maintain, a user interface is a much easier method of keeping track of payments or the sports they practice. This also removes the need for contact between members and the organization to obtain more information regarding due quotas or other types of details. Furthermore, the application allows for a closer proximity between members and the organization, while also automating several tasks, not needing as much human intervention as the alternative.

1.2 Goals

The project's goal is the development of a system that intends to solve the problems listed in section 1.1 of this chapter by building a management application with features that allow the management of the organization's members, keeping track of the sports they practice, their quotas, and also the events they organize. Another key feature, is the creation of a digital membership card so that members can get discounts or offers in the different partner companies.

Our solution is to build a Single Page Application [3] with the front-end Javascript [4] library React [5], that connects with the developed Web API in order to provide these functionalities to the members.

1.3 Main functionalities

So that the project could reach the desired outcome, the definition of the requirements was needed, thus becoming the following:

- Allow the creation of members (users, and companies), quotas, events, sports, groups, and club candidates. It is also needed to allow the update and deletion of the resources that are to be created. Associate athlete users with their sports.
- Create a digital membership card that permits companies to identify club members and offer discounts or promotions.
- Notification by email of new events, candidates approval and quotas in debt, and contact administration through the application.
- Data analysis for statistic purposes.
- Import and export data through *CSV* files, also allowing to change credentials when members are inserted.

1.4 Report structure

This report is divided into several chapters, that outline the technologies chosen for the project as well as the way its architecture is assembled.

Chapter 2 lists the different functionalities within the project and also differentiates the types of members that exist.

Chapter 3 describes the project's architecture regarding the server and client, explaining in a broad manner the way each of them work. Lists the different functionalities within the project and also differentiates the types of members that exist, also explains the technologies chosen and why they were chosen, and provides an explanation regarding the database model used, including each entities relationships with one another.

Chapter 4 describes in further detail the server implementation along with an explanation about its structure and layers, the way errors are handled, as well as the authorization and authentication process, and the working methodology.

Chapter 5 describes the client implementation, providing an introduction to React and its components, as well as the way requests to the API are made and how the errors that originated from the requests are handled. Also explains the authentication and authorization process in the client and the implementation of internationalization.

Chapter 6 shows the functional aspects of the project, now in a graphical way.

Chapter 7 explains the process in which the membership card will be used.

Chapter 8 describes how pagination and filtering were implemented in the required views.

Chapter 9 reflects thoughts regarding the course of the project.

Chapter 2

Functionalities

The following section presents all functionalities that the application will be capable of executing. Visual examples can be found in chapter 6.

The system supports three types of roles: Administrator, Club member, and Candidate.

- **Administrator**: Able to access the system's maximum capability and functionalities without restrictions;
- **Club member**: There are two types of club member, a User and a Company. A User is able to access all common information such as Events and Sports, and also has the ability to view and edit their own personal information, as well as information related to sports, events or quotas. A Company can also access all common information and is able to view and edit their information as well as their attendance at Events.
- **Candidate**: Able to only access the homepage of the system and make his club application.

The present functionalities are:

- **Sign-up and Sign-in**: Non-members can apply to be a candidate or members of the club can be verified using the system's verification procedures.
- **Entity creation, update and deletion**:
  - **Candidates**: Candidates can be created and fully deleted. Its update consists in the members approval, passing from being a candidate to a proper member. All candidates can also be exported to a CSV file.
  - **Members**: Members can be created freely; its deletion, however, is only a soft delete due to the different dependencies and the need to keep a records of all members. The update allows to change every aspect relevant to the member, except its username and email. All members can also be exported to a CSV file.

- **Sports**: A Sport represents just that, and can be created with no restrictions; the deletion works like in members, having only a soft delete and the update is only useful in case it's needed to add a deleted sport, in which case only the attribute that represents deletion is altered.
- **Events**: Events can be created; its update consists in just changing the time interval in which the event occurs and the deletion is a full delete, removing the event as well as the attendance of that event from the database. An event within the organization is a broad subject, ranging from gatherings of practitioners of a specific sport or even a general assembly.
- **Groups**: Groups can be created by referring the categories desired in the members that will be added to this entity. There is no update for groups as that could generate some incoherence due to its linkage to events, as changing members of a group would change the attendance of the members added or removed. These groups allow the grouping of different members according to specific criteria chosen by the creator, whether it be the type of member or the type of a member in a sport.
- **Attendance**: The attendance is added automatically when creating an event, due to the characteristic of the event is that it's associated with as many groups as the creator wants, and those groups members will be added to the attendance list, afterwards being able to decide if they want to go or not.
- **User Sports**: The linkage between users and sports, offering its creation, ability to update and a soft delete, this deletion serves as a marker to indicate whether or not the user has retired from competitive sports, having always the possibility of coming back so it's necessary to keep record of previous activity within the sport.
- **Quotas**: Quotas can be created, being added to all members, updated singularly to update a quotas payment date, as well as a full delete of all quotas that share the same due date.

- **Upload Club's current data to the system**: Ability to upload CSV [6] files containing all the club's information to Surf Club Management Application, structuring the data according to the prerequisites.
- **Notification via email**: Besides communicating with administration through an email form in the application, members will get notification about events and quota payments. Candidates will also receive an email if they were accepted.
- **Club statistics**: Ability to view all information about the current state of the club.
- **Internationalization**: Ability to translate all pages between Portuguese and English.

- **Password reset and Credentials change**: Ability to reset the members password and change the username and password of members added through data import.

All these functionalities were mandatory according to the group, however there are a few additional features to be implemented in the future as described in the last chapter 9.1.

Chapter 3

Architecture & Data Model

The architecture of the developed system is described in this chapter, with its components illustrated in figure 3.1.

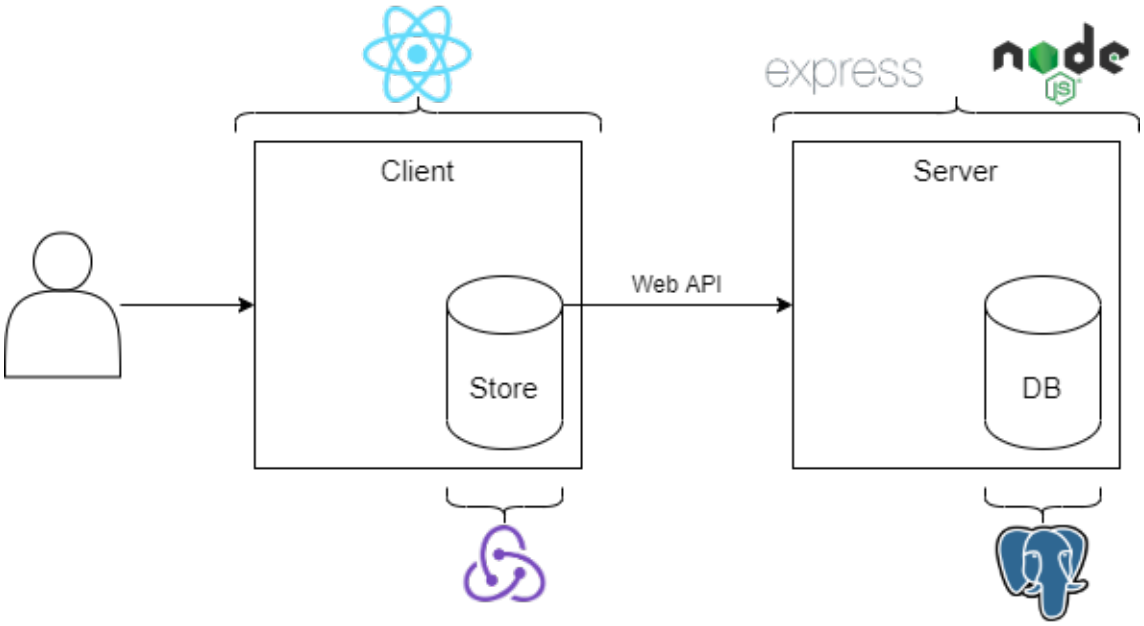

Figure 3.1: General Architecture

The system is divided into two parts. The server is the major one, and it's in charge of implementing the system's logic, storing data, and providing an HTTP API that offers a set of capabilities to potential customers. The other part of the system is a web application that serves as a client for the server application, allowing users to interact with it via a graphical interface.

This technique was adopted in order to accommodate the possibility of numerous types of user engagement, such as mobile or desktop applications. Clients do not need to maintain knowledge about the system or implement its logic in this way because these pieces are already available in the server.

A web application was chosen for the creation of a client application because it is compatible with a simple browser, which is what most club members use. It also allows to have just one implementation for it to be compatible with any operating system or device.

3.1 Technologies

This section describes the technologies used in this application in both backend side and frontend side.

The server application, or backend, is an application that exposes a web API, and it was developed using the Javascript programming language [4], the execution environment NodeJS [7] and uses the node-postgres [8] module for database access as well as the Express framework [9] for HTTP [10] requests. This API serves as the central point of contact for all clients, meaning that all clients (web, mobile, etc.) connect with the same server application.

Customers contact the different endpoints to access, change, or create the required resources, and here is also where all the system's logic is implemented. An endpoint [11] can be simplified as a pair consisting of a path and an HTTP method in which it the combination corresponds to a route in a server that will then carry out actions that correspond to its definition.

A PostgreSQL [12] relational database was utilized to store the data for this application. This Database Management System (DBMS) was chosen because of past expertise with it and the simplicity with which it can be hosted on the Heroku platform [13].

The web application consists of a frontend application that aims to design a User Interface (UI) to access the resources made available by the backend. The client side was built with ReactJs [5], a framework that enables building high-performance responsive apps as it's one of the most popular libraries for designing user interfaces. All views were designed using the Material-UI library to match design needs and industry standards.

This web application is an SPA (Single Page Application) [3], which means it just has one HTML document that renders different pages over the application's lifespan, depending on the path specified or the user interaction. An SPA was chosen over an Multi Page Application because an SPA is faster since most of its resources are only loaded once.

To ensure well managed state of the frontend application, the use of a Redux store [14] was required, since Redux provides a subscription mechanism which can be used by any other code. That said, it is most useful when combined with a declarative view implementation that can infer the UI updates from the state changes, such as React or one of the similar libraries available.

Heroku was chosen as the platform to launch the app since it has built-in support for PostgreSQL based projects.

3.2 Data Model

The following section presents the simple graphical representation of the database in its Entity Association [15] format without the attributes, describing the existing entities and the relations between them. The full Entity Association is in appendix A.

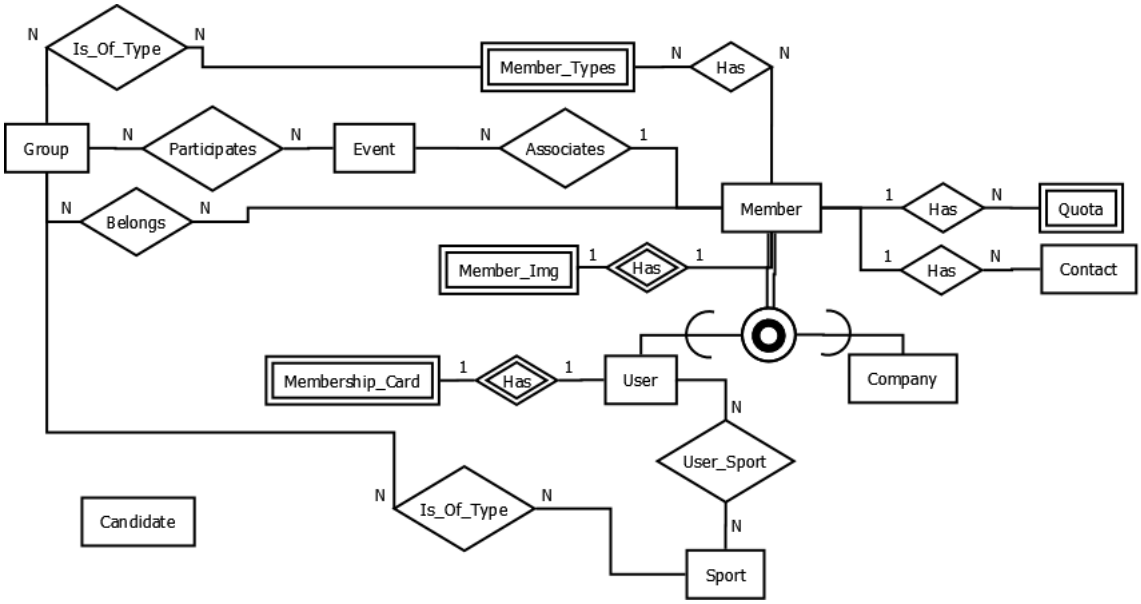

Figure 3.2: Simplified EA Model

As shown in the figure 3.2, **Member** is the core entity, representing a member of the surf club, and this member is one of two types, a **User** or a **Company**. These **Members** also have associated a **Member Type** to help define what kind of role the member has within the organization, characterized by the name of the type and its category that declares whether the type is used for users or for companies.

A **Member**, as it was referred before, can be either a **User** or a **Company**, being that a **Company** is more restricted in terms of its uses since it's only a way to allow a company representative to access its own profile and validate users by scanning their membership card.

A **User** is a broader type of Member representative of common members of the organization and preserves more relevant information in regards to their activities in the organization, it being related to which **Sports**, **Events** or **Groups** they are associated with.

The **Quota** entity allows to keep record of the organizations quotas and to identify which members have paid their dues regularly. The insertion of a **Quota** is done through specifying the date to which it is relevant, for example, that years quota, and it's added to all members

except those whose quota value to pay is 0. Its removal is also done through the date, removing that **Quota** from all members except if they were already paid by the member.

As there are **Events**, which can represent a plethora of different affairs within the organization, there are also records of **Attendance** so that participating Members can be identified. Each **Event** is associated to the **Groups** relevant, adding **Attendance** to this **Event** to all members from the groups chosen.

Groups, as said before, are an agglomeration of **Members** differentiated through several criteria, existing two types of groups, groups by member type and groups by member types within the sports chosen. A group by member type is a set of members that share the same type(s) chosen when creating the group. A group by member types within the sports chosen is a set of members that practice the sports chosen and filtered by the type within those sports, for example, coach or practitioner.

The association of **User** and **Sport**, named as a **User Sport**, holds information related to federations, which are not an entity being handled in this system.

Contact saves the relevant information regarding the ways to contact a **Member**, whether it be through email, phone number or address if necessary.

A **Candidate** is the state before it becomes a **Member**, saving all information needed for when the transitions happens.

Chapter 4

Server implementation

In this chapter it will be described the server's implementation, each layers' mission and how they accomplish it.

The server is the applications component that exposes a web API that allows to maintain and provide information required by the client making a request to the server. It manages all of this throug three distinct layers:

- Web API, which is divided in two layers – Routes and Controllers:
  - **Routes** – establishes endpoints and authorization middleware;
  - **Controllers** - responsible for extracting the data necessary to complete the action correspondent to the endpoint associated with the controller.
- **Services** - processes and validates the request, implementing the API logic;
- **Data** - Responsible for verifying the existence of the resource, inserting data or verifying conflicts when inserting, accomplished by calling methods from the DAL.
  - **DAL** - Considered to be part of the Data layer, it connects to the database directly and is centralized in a file, meaning all files in the Data layer access the same DAL. This is due to the existence of a mock in-memory database so there was a necessity for an easy way to interchange between the two when necessary.

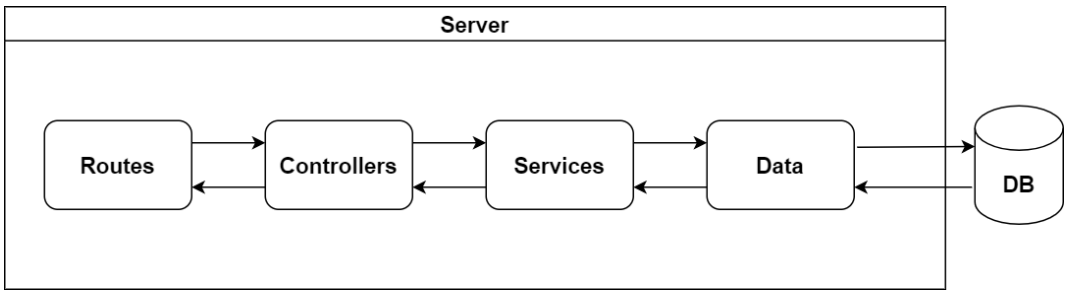

Figure 4.1: API structure

4.1 Dependency injection

Dependency injection is a well-known approach that makes creating independent and interchangeable modules easier.

This is a pattern where, instead of creating or requiring dependencies directly inside a module, we pass them as parameters.

The dependency injection starts in our entry point, *index.js*. Here it will be chosen which data storage the server uses, and it will also be sent as parameter an instance of express to the server, which in turn will fill it with routes. The server has the API routes definition, and will add them to the received express instance. Since this module receives by injection the storage module that the entry point parameterized previously to the server, each route module will receive it to instantiate an express router.

From that point on, every following module initializes the other passing the storage chosen, until the data access module is reached.

The following figure shows a schema of the dependency injection in our modules:

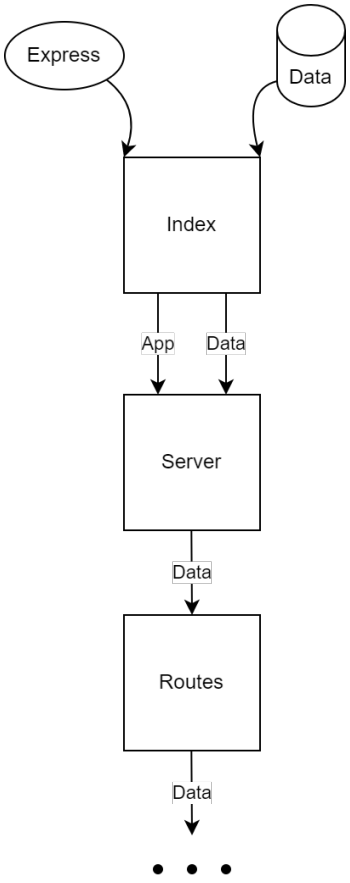

Figure 4.2: Dependency injection

4.2 Error handling

To maintain an API capable of handling all errors, a error handler, or middleware, was created as observed in the code listing 4.1. This error handler sets on the response the correspondent error it received that can derive from certain actions across the server, for example on the database access or missing parameters in requests.

Since the error has all information needed to pass to the response, that is all it needs to do, passing all properties of an error onto the response. An error in this context is composed of three elements, a status code, that will be set on the response, a message, which is a specific message in regards to the action that caused the error, and also the message code, that is an arbitrary unique identifier that the users of the API can make use of when implementing internationalization, for example. As there is a large number of possible errors, certain errors were grouped in the message code.

```
const errorHandler = (err, req, res, next) => {
  const status = err.status
  res.status(status || 500)
  res.json({message : err.message, message_code: err.message_code})
}
```

Listing 4.1: Error Middleware

To make the handling of errors easier, another middleware was used, called express-async-handler [16], that handles errors inside of async express routes and passes them onto the error middleware defined above. This is especially useful to maintain a clean code, meaning instead of a try/catch, the code can be written declaratively and simply throw an error when it's needed.

The middleware itself is rather simple to use, only needing to wrap the route controller on it and it essentially returns a callable function that will work just like the controller that was passed, having full access to the request and response due to the closure being created. A use of this middleware can be seen on the code listing 4.2.

```
const getSports = asyncHandler(async (req, res) => {
  const sports = await services.getSportsServices()
  res.json(sports)
})
```

Listing 4.2: Async Handler

4.3 Web API

The exposed web API receives HTTP requests from the application's client side, extracting all data necessary to accomplish the operation associated with the endpoint specified by the request and sending the data to the service layer.

When the service delivers the data that the client requested, the interface will map this information into an HTTP response. Similarly, all errors that occur are mapped into an HTTP error response. Each module in charge of this logic was referred to as a controller.

Since the server uses the Express framework, a router is used to define the API endpoints as well as associate a controller to each one. In this application, the server module creates the Express router, which is then provided to the module routes, where it registers all the paths.

The API documentation was made using Swagger and by following the OpenAPI specification. The documentation is hosted on a github page [17].

4.3.1 Authentication

Authentication is done by using the PassportJS [18] middleware, a NodeJs authentication middleware, which is based on a local strategy [19] with username and password and validates the authentication attempt by populating the session cookie [20], due to have chosen to use a session based authentication.

The strategy implemented consists in validating a username and password. The member is fetched according to the username, and if it does not exist an error is thrown. Otherwise, the clear password received is compared to the hashed one saved in the database and if it matches the authentication process is finished successfully, if it doesn't an error is thrown. This strategy can be observed in the following code listing 4.3.

```
passport.use(new LocalStrategy(
  async (username, password, done) => {
    try {
      const member = await data.getMemberByUsernameData(username)
      if (!member) {
        done(error(401, 'Incorrect username', 'MESSAGE_CODE_1'), false, null)
      } else {
        if (await crypto.comparepassword(password, member.pword_)) {
          done(null, member)
        } else {
          done(error(401, 'Incorrect password', 'MESSAGE_CODE_1'), false,
    null)
        }
      }
    } catch (err) { done(err, false) }
  }
))
```

Listing 4.3: Local Strategy

There are two methods that will be called and will serialize and deserialize the authenticated user:

- Serialize is the method that is called on a successful authentication and it decides what user information should get stored in the session.
- Deserialize is the method that is called on all subsequent requests and allows loading additional user information on every request. This user object is attached to the request in the user property making it accessible in our request handling.

Instances of these methods can be observed in the code listing 4.4

```
passport.serializeUser((user, done) => {
  done(null, user.username_)
})

passport.deserializeUser(async (username, done) => {
  try {
    const member = await data.getMemberByUsernameData(username)
    if(!member) {
      done(null, false)
    } else {
      if (member.category_ != 'company') {
        const user = await data.getUserByIdData(member.id_)
        member.is_admin_ = user.is_admin_
      }
      done(null, member)
    }
  } catch (err) {
    done(err, false)
  }
})
```

Listing 4.4: Serialize And Deserialize Methods

Regarding the cookie itself, when the application is deployed it will always have httpOnly and secure flags set to true to help minimize the risk of the client being able to access the cookie and making man-in-the-middle attacks harder to accomplish. This configuration can be observed in the code listing 4.5. A definition that must be set when deploying to Heroku specifically is enabling the trust proxy setting. This is needed because in Heroku, all requests come as plain HTTP but have the header X-Forwarded-Proto to define whether the original request was HTTP or HTTPS. Express will see non-ssl traffic and will refuse to set a secure cookie, and so we need to tell express to trust the information in the mentioned header.

```
let cookieSettings = {
  maxAge: 4 * 60 * 60 * 1000
}
if (process.env.NODE_ENV == 'production') {
  app.set('trust proxy', 1);
  cookieSettings = { ...cookieSettings, httpOnly: true, secure: true,
    sameSite: true }
}
const secret = process.env.SECRET
// ...
app.use(expressSession({ secret: secret || 'keyboard cat', resave: true,
    saveUninitialized: true, cookie: cookieSettings }))
```

Listing 4.5: Cookie Configuration

4.3.2 Private data management

Storing clear passwords in a database is a serious mistake that jeopardizes the data's security and privacy.

To fix this problem, the BCrypt library [21] is used, which does a hash of the entered password together with the salt [22]. The purpose of using a salt in a password hash is to ensure that identical passwords do not have the same hash, as the same hash function applied to the same password produces the same output, making the hash predictable and vulnerable.

The salt is just a string that is generated in an asynchronous manner and appended to the password in clear text so that the hash may be performed afterwards.

The following figure 4.3 is an schematic example from an article of StackAbuse [23]:

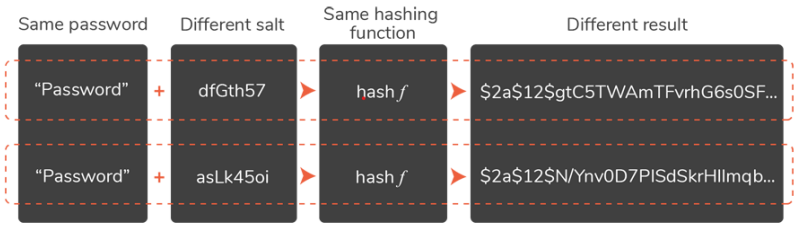

Figure 4.3: Hashing passwords plus salt

4.3.3 Authorization

Custom middlewares were created to handle authorization for each route of the API. These middlewares were necessary since administrators have access to all routes, members have access to common routes for all members as well as personal data routes, and anyone who is not a member has access to only the candidature and login route.

To avoid unwanted access to the protected routes, a middleware that validates whether the user is authenticated was created, as observed in the listing 4.6. This middleware extrapolates from the request the user property, that was filled by Passport when the user went through the authentication process, and if it does not exist an error is thrown, otherwise the middleware allows the request to proceed to its next destination.

In every case, if the user is not authenticated, that corresponds to the 401 status code.

```
const authMember = asyncHandler(async (req, res, next) => {
  if(!req.user) {
    throw error(401, 'Unauthorized', 'MESSAGE_CODE_5')
  }
  next()
})
```

Listing 4.6: Authentication Middleware

Besides the aforementioned middleware, there is still the need for more credentials verification, since there are routes that only admins can access, that only companies can access and that each member can only access when it's their respective information.

As observed in the code listing 4.7, the process to check if the authenticated member is an administrator consists in verifying, using the previously mentioned user property in the request, if the property that corresponds to whether the member is an administrator or not has the value true.

If there is an authenticated member but it is not an administrator, this situation corresponds to the 403 status code, in other words, the user is forbidden from accessing that resource. If there is no authenticated user, it corresponds to the unauthenticated status code, 401.

```
const authAdmin = asyncHandler(async (req, res, next) => {
  if(req.user && !req.user.is_admin_) {
    throw error(403, 'Access forbidden', 'MESSAGE_CODE_33')
  } else if (!req.user){
    throw error(401, 'Unauthorized', 'MESSAGE_CODE_5')
  }
  next()
})
```

Listing 4.7: Admin Middleware

As observed in the code listing 4.8, the process to check if the authenticated member is a company is similar to verifying whether the member is an administrator, with the difference being the member type is the property that is verified.

Like the previous middleware, if there is no authenticated member it corresponds to the 401 status code. If the authenticated member is not a company, then the correspondent status code is 403, which means the member is forbidden from accessing that resource.

```
const authCompany = asyncHandler(async (req, res, next) => {
  if(!req.user){
    throw error(401, 'Unauthorized', 'MESSAGE_CODE_5')
  } else  if(!req.user.member_type_ == 'corporate') {
    throw error(403, 'Access forbidden', 'MESSAGE_CODE_33')
  }
  next()
})
```

Listing 4.8: Company Middleware

4.4 Service

The business layer is in charge of receiving the parameters of the request and processing them, validating their existence ensuring an error is thrown in case a mandatory parameter is missing. After validating the request, the correspondent module that bridges the business and the data access layer is used.

An example of a method implementation in this layer is illustrated in the code listing 4.9 where all mandatory parameters are being validated regarding their existence, and also the correspondent errors are thrown if the parameter does not exist. After the validation, as referred previously, the requests processing is attributed to a different module to check integrity restrictions.

```
const getCompanyByIdServices = async(id) => {
  if(!id) throw error(400,'Parameter not found: id', 'MESSAGE_CODE_14')
  return await data.getCompanyById(id)
}
```

Listing 4.9: Service Layer Example - Parameter Processing

4.5 Data

This module also accomplishes the following of the business logic in regards to duplicate or inexistent content, always verifying if the resources being accessed exist, or in the case of insertion, if there are duplicate values that should be unique in the database, thus ensuring a safe environment to execute the queries on the data access layer.

An example of a method implementation in this layer is illustrated in the code listing 4.10 where all mandatory parameters are being validated regarding their existence, and also the correspondent errors if they do not exist. After the validation, as referred previously, the requests processing is attributed to a different module to check integrity restrictions.

```
const getCompanyById = async (id_) => {
  const company = await db.getCompanyByIdData(id_)
  if (!company) throw error(404, 'Company does not exist', 'MESSAGE_CODE_24')
  return company
}
```

Listing 4.10: Data Layer Example - Integrity Restriction Processing

4.6 DAL - Data Access Layer

The data access layer is a layer that separates the database from the service/business layer. It must be done in such a way that the database and service layer are completely independent of one another.

This layer keeps the pulling of data from a database separate from the service layer so that there is an abstraction regarding the database, meaning if there is a need to switch databases the only required thing to do is to switch out this module.

Before accessing the database, the connection must first be established, and this is done by defining the specific credentials for the database and after creating the connector a function is returned that receives another function which is considered as the transaction handler, which describes what will happen during the transaction as well as what the return value will be, as shown in the code listing 4.11 and 4.12.

```
const creds = {
  user: PG_USER,
  password: PG_PASSWORD,
  host: PG_HOST,
  port: PG_PORT,
  database: PG_DB
}
const connector = new pg.Pool(creds)

return async (transactionHandler) => {
  const client = await connector.connect()
  try {
    await client.query('Begin')
    const result = await transactionHandler(client)
    await client.query('Commit')
    return result
  } catch(e) { await client.query('Rollback'); throw e }
  finally { client.release() }
}
```

Listing 4.11: Database Connector

```
const getCompanyByIdData = async (id_) => {
  const handler = async (client) => {
    const company = await client.query(queries.QUERY_GET_COMPANY_BY_ID, [id_
    ])
    return company.rows[0]
  }
  return await pool(handler)
}
```

Listing 4.12: DAL Method Example

4.7 Email notifications

Sending emails to members is an important feature, since it keeps members up to date regarding different situations within the organization, specifically whenever an event is created, a member is imported, a candidate is approved or when notifying a member of their unpaid quotas.

This notification is done through server side, using NodeMailer [24], that facilitates the process, needing to create a transporter, in this case using SMTP [25], and afterwards using said transporter the email can be sent out.

The transporter can be defined as observed in the code listing 4.13, in which there are a few fields that are needed in order actually complete the process. The host defines the SMTP server name and service establishes that Outlook [26] is the provider. The credentials of the email used are also defined.

```
const email = process.env.EMAIL
const password = process.env.EMAIL_PASSWORD
let transporter = createTransport({
  host: 'smtp-mail.outlook.com',
  service: 'outlook',
  secureConnection: false,
  port: 587,
  tls: { rejectUnauthorized: false },
  auth: { user: email, pass: password }
})
```

Listing 4.13: NodeMailer transporter

After obtaining the transporter, emails can now be sent, following the example in the code listing 4.14, in which the sender is always the email defined in the transporter preparation, and the recipients can vary depending on the situation.

Sending an email also involves creating templates for the content, as such several templates were made to accommodate both text and HTML formats.

In case of exception, an error is thrown with the status code 554, to indicate the email did not go through.

```
try {
  let info = await transporter.sendMail({
    from: email,
    to: receivers,
    subject: subject,
    text: content.text,
    html: content.html,
  })
  return info
} catch (e) {
  throw error(554, 'Error while sending email', 'MESSAGE_CODE_42')
}
```

Listing 4.14: Send Email Using The Transporter

4.8 Data import

To import data from CSV files, the package Express-fileupload [27] was used, which is a middleware for uploading files. The CSV files are uploaded from the client as a Blob and afterwards, in the upload endpoint in the server, the file can be accessed directly through the request as observed in the code listing 4.15, and since the file is a blob, the data can be accessed directly without needing any further type of data stream.

Besides the file, the endpoint also expects a type of data being imported which could be one of member types, members and companies, quotas, sports, sport types and user sports.

It's also worth noting that there is a template Xlsx file [28] for each type with all the attributes the application needs to be able to import the data, to which it can then be exported to become a CSV file.

A particularity is that depending on the zone of the software being used to modify the Excel template, the CSV delimiter can be either a semi-colon or a comma, as such the delimiter must be verified when parsing the data.

```
const uploadFile = asyncHandler(async (req, res) => {
// ...
  const data = req.files.file.data
// ...
})
```

Listing 4.15: Access Uploaded File

After obtaining the data, it can now be parsed to be on a format easier to manipulate and form queries for insertion into the database. The parsing process is varied since as detailed before, there are different types of data that can be imported.

For example, uploading member types only requires a name for the type and its category, defining if it's a type for users or for companies. As such, before inserting the data into the database, first the query must be formulated and so both attributes must have a prime symbol wrapping the values as shown in the code listing 4.16, which for each tuple will add the prime symbol.

```
const uploadMemberTypes = async(data) => {
  let count = 0;
  for(let val in data){
    let value = data[val]
    value = value.split(delimiter)
    value[0] = ''${value[0]}''
    value[2] = ''${value[2]}''
    data[count++] = value
  }
  // access the database
  return await db.uploadMemberTypesData(data)
}
```

Listing 4.16: Parse Uploaded File Data

4.9 Tests

This chapter is meant to explain the methodology and the steps taken to ensure the quality of work in this application.

The project follows a test driven development environment [29] to validate all features present in the server.

To facilitate testing, the database was replicated into a functioning in-memory mock database, this is especially useful since it allows the developer to focus on code and not on the database state.

During the development of the project, several unit test suites were created to validate the code done, and to assist in finding bugs or errors that might have been missed through development.

The framework Jest [30] was used to make the unit tests.

Besides unit tests that use the in-memory database developed, those same tests were also used to test a database, since while it's useful to test the mock to find errors, it's also needed the verify if the database is well configured and corresponds to the expected features.

Having a tested mock and database, the next step was to create a different type of tests, integration tests. The integration tests verify the good behavior of the different routes and the server. The purpose of these tests is to expose defects in the interaction between different software modules when they are integrated.

Supertest [31] was the library used to make the integration tests.

Chapter 5

Client implementation

The following chapter explains the implementation and thought process behind the client side, that is responsible for the visual element of the project.

5.1 React

React makes the creation of dynamic views easy due to its ability to send properties, which are like function arguments, as attributes to components, that are small reusable pieces of code that render content into the UI, which means that any modification to a property triggers a re-render of the component that was modified. This effect is due to React using a virtual DOM [32], that compares the previous state of the components and updates the ones affected.

The virtual DOM is a programming concept in which a precise representation of a user interface is stored in memory and synchronized with the "real" DOM using a library like ReactDOM [33]. This is referred to as reconciliation. With this approach, you can use React's declarative API to tell it what state you want the UI to be in, and it will ensure that the DOM matches that state. This separates the attribute manipulation, event handling, and manual DOM updates that you'd have to do otherwise to construct your app.

5.1.1 Components

A React component goes through different moments in its life cycle as shown in figure 5.1.

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation. They serve the same purpose as JavaScript functions but return HTML and is maintained by state code that allow modifications.

The creation and render of the component enter the mounting process, in which the components constructor is called to define the needed properties, afterwards the view designed by the user is rendered onto the DOM.

Unmounting is the process in which the component is removed from the DOM and destroyed and performs any sort of cleanup in regards the residual code or state. In this project, the common use for the cleanup is in the different views, as to remove any data from redux used in this page, as it will no longer be used by other views and even if we returned to this view this data would be outdated so a new request is needed. An example of this, as observed in the code listing 5.1, is the unmount of the view that displays all events, which needs a lot of sanitizing due the several requests that were made either for fetching all of the events, deletion and creation of events and even a dependency for creating events, which are groups.

As displayed, this sanitizing is done through the return in the useEffect hook and this symbolizes that when the component (in this case the view) is removed from the UI, the code within will be triggered. Since the cleanup is related to redux, the action we want to do is remove the residual state and this is done by calling dispatch with the identifier of the action. The Redux technology is expanded upon in the section 5.4 and goes in in-depth detail on what actually happens when an action is dispatched and what an action really is. The useEffect function, considered as a React Hook, is also explained in section 5.3.

```
useEffect(() => {
    // ...
    return () => {
        dispatch({ type: EVENTS_FETCH_RESET })
        dispatch({ type: EVENT_DELETE_RESET })
        dispatch({ type: EVENT_CREATE_RESET })
        dispatch({ type: GROUPS_FETCH_RESET })
    }
}, [])
```

Listing 5.1: Cleanup And Sanitizing Example

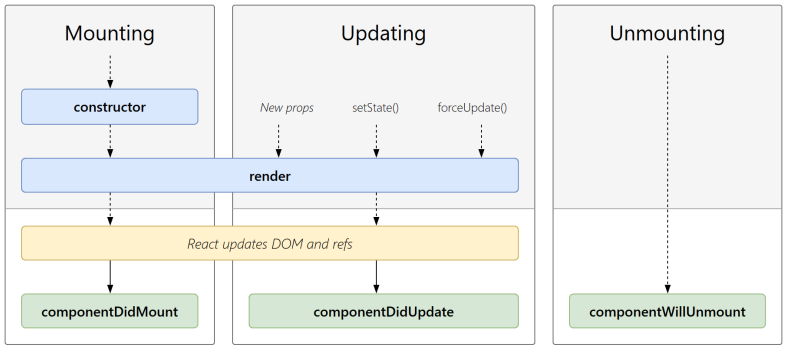

Figure 5.1: Component lifecycle [2]

There are two types of React components, them being class based components and function components. Class components allow the developer to define several methods as explained in the components lifecycle, such as mounting and unmounting the component.

Function components can also apply concepts within the lifecycle, but can be written with much less code.

For this project, the components are mainly function components. A simple example of a function component can be found in the code listing 5.2, which is simply rendering text, but could have within state that affects which content is rendered.

```
function Isel() {
    // State must be declared before the return of the content
    return <h2>Hi, I am from Isel!</h2>;
}
```

Listing 5.2: Component Example

Components can also have nested components, or children components. These are also components, but are now rendered from within a different component.

Each component can also receive several parameters that dictate state and content that will be rendered.

5.1.2 Hooks

React Hooks are a tool useful when there is a need to have component based state, in this project a use for it could be to hold the data fetched from the web API so that it could be delivered to a data grid so it could be displayed. For this there is the useState Hook, which receives an initial value and returns to the value and also a function to update the value, as shown in the code listing 5.3.

```
const [ searchState, setSearchState ] = useState({
    name_filter: '',
    event_initial_date_filter: '',
    event_end_date_filter: '',
    limit : 10
})
```

Listing 5.3: Hook UseState Example

Another useful Hook is useEffect, which is useful when we want to trigger a UI update since it receives an array of dependencies as the second argument and every time the value of a dependency within the array changes, the function sent in the first argument is triggered. A situation where this is useful in the project, is when first loading a view, data needs to be loaded, and so useEffect is used with an empty dependency array (meaning it will only be called once, when the component is in its mounting phase as mentioned in the sub-section 5.1.1). An example of its use is shown in the code listing 5.4.

```
useEffect(() => {
    dispatch(getEvents(searchState.name_filter, searchState.
    initial_date_filter, searchState.end_date_filter, 0, searchState.limit))
    return () => {
        // ...
    }
},[])
```

Listing 5.4: Hook UseEffect Example

5.2 Router

Since the client is a standalone application, disconnected from the web API, there's a necessity to create client side routing.

The definition of a route consists in associating a path with a component as well as defining a hierarchy and division.

This division is necessary to the different type of routes, specifically the dashboard views, login and candidature views, error views and home page views.

By defining groups of routes we can compose a specific layout that will be used for all children routes. This is especially useful to build the dashboard, and define that every children uses a sidebar for navigation and header. Besides defining the specific layout, it's also possible to do the authorization on the client side, which is done through the use of different components that check if there is an authenticated member or if the member has the necessary credentials.

The used methodology to define routes and groups of routes is shown in the code listing 5.5. Just creating an object like this is not enough, so a function that parses the object into a Router, useRoutes, which is part of the ReactDOM package, is used like it's shown in the code listing 5.6.

In this example, the main dashboard is described, in which there is base path and a set of nested components as the base element. This means that for that path the nested components will be rendered. There are also children routes which are descendents of the primary path, and describe the path segment that goes after the primary path and which element will be rendered at that path along with the parent component.

```
const mainRoutes = {
    path: '/',
    element: <Error><RequireAuth><RequireNotDeleted><AnimatedVideo><
    MainLayout/></AnimatedVideo></RequireNotDeleted></RequireAuth></Error>,
    children: [
        {
            path: '/dashboard/analytics',
            element: <RequireAdmin><DashboardAnalytics/></RequireAdmin>
        }
    ]
}
```

Listing 5.5: Client Routes Example

```
export default function Routes() {
    return useRoutes([homeRoutes, authRoutes, mainRoutes, errorRoutes]);
}
```

Listing 5.6: Client Routes Parser

5.3 API access

The web application serves as the web API client for our system, and as such there must be a connection between both components.

This bridge is maintained with the use of a redux store, that saves the global applicational state, in other words, the state of the API requests, in a way that allows each view to subscribe to the state and directly observe the state without the need of tools like useState to alter the step in which the request is at.

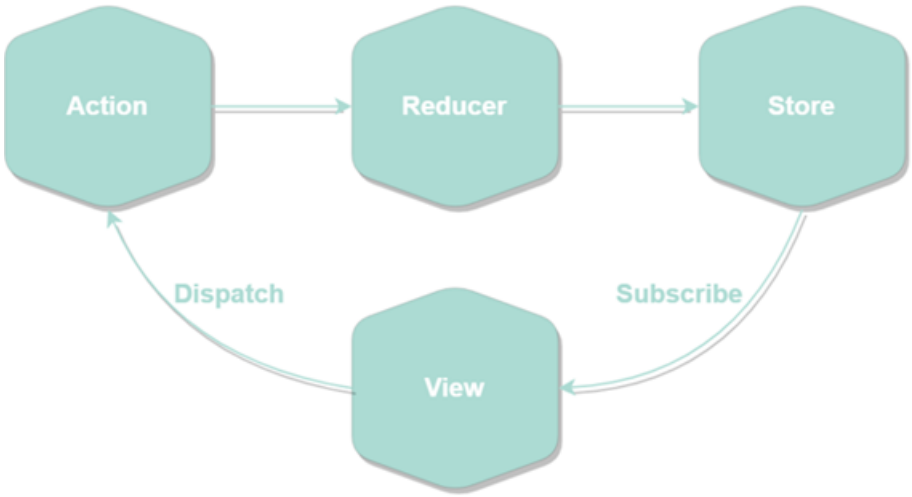

Figure 5.2: Redux scheme

As it was said before, the view subscribes to the chunk of state it will need, and this state may vary, however there are always 3 components: a loading, error, and value variables.

Afterwards, the view can dispatch an action, that in our case corresponds to an API request, also considered to be a process describer.

The action, as observed in the codes listing 5.7, also executes several dispatches to its correspondent reducer, also visible in the code listing 5.8, that based on the values received by the action, it updates the values of the state. These dispatches that the action does are referent to a request's lifecycle, in which it enters a loading state, and after receiving the response of the API it passes onto the state of success or failure depending on the response.

All these changes directly affect the centralized store, which in turn also affects the view that subscribed to the state.

```
export const getUserById = (id) => async (dispatch) => {
    try {
        dispatch({
            type: USER_FETCH_REQUEST,
        })
        const response = await fetch(`/api/users/${id}`, {
            method: 'GET',
            headers: { "Content-Type": "application/json" }
        })
```

```
        const users = await response.json()
        if (response.status !== 200) throw Error(users.message_code)
        dispatch({
          type: USER_FETCH_SUCCESS,
          payload: users,
        })
    } catch (error) {
    dispatch({
        type: USER_FETCH_FAIL,
        payload:
            error.response && error.response.data.message
            ? error.response.data.message
            : error.message,
    })
    }
}

// Calling an action in a view can be done as follows
dispatch(getUsers(...))
```

Listing 5.7: Action Example

```
export const userFetchReducer = (state = {userGet: {}}, action) => {
    switch (action.type) {
        case USER_FETCH_REQUEST:
            return { loading: true }
        case USER_FETCH_SUCCESS:
            return { loading: false, userGet: action.payload }
        case USER_FETCH_FAIL:
            return { loading: false, error: action.payload }
        default:
            return state
    }
}
```

Listing 5.8: Reducer Example

5.4 Client side error handling

As it was mentioned above, an API request can produce an error, and this error is saved on the Redux state.

The view, since it subscribed to the state was notified of this change in value and renders a new component that describes the error to the user.

5.5 Authentication and Authorization

The authentication in the client also resorts to the use of Redux, in which if in the login process the credentials match an existing account, then the necessary info about the member is saved in the local storage of the browser, along with the cookie expiration date, forcing a logout whenever the cookie is expired. This is due to the technique used in the login process, which is based in session authentication.

Now if there is not authenticated user there would still be a possibility to do the request, even if the request would fail. That wouldn't be a good user experience, and so authorization

was explored, preventing the client from doing requests to the web API completely if they aren't authenticated or don't have access to the request and this is done in two steps:

- The views themselves do not appear in the UI Sidebar if the authenticated member does not have the necessary credentials;
- If the member manages to access these restricted pages, for example by entering the URL directly, they are redirected to an unauthorized page.

5.6 Internationalization

So that the application has a broader prospect of future users, internationalization was implemented, having at the moment both english and portuguese.

In regards to implementation, the react-i18next [34] library was used. It's a popular internationalization library which uses components to render or re-render the translated content of our application once a user requests a change of language.

This library is using json [35] files to store all translations of each language and whenever a string is requested, the context of the application recognizes the language and fetches the string from its correct json file. The language is also stored in the browsers LocalStorage [36] so that the users choice can persist even after they close the window.

5.7 Code structure

The client project contains everything needed for the client side and is organized in the following way:

- **Assets**, which is divided into:
  - **Data** - Contains static files like the application logo and the background video of the home page;
  - **Scss** - Contains the css [37] of some pages and the definition of all colors used in the project.
- **Components** - Plethora of differenct components used in the project and can be considered to be a small library of components that are used throughout the application, like custom form components and profile tabs.
- **Hooks** - The different hooks [38] used on the application, allowing the use of state and other React features without writing a class. An example of use for a hook created would be to maintain state on the current language of the application, due to existing components that need certain translations and can't be accessed manually, only with proprietary translations of Material UI.

- **Layout** - Layouts used in various pages, like the sidebar and profile header on the dashboard or the header for the home page.
- **Locales** - Contains the different JSON files for both portuguese and english used on internationalization.
- **Menuitems** - Contains the definition of the different pages that will be shown in the Sidebar as well as restrictions regarding authorization.
- **Pages** - Despite its name, this consists in all views of the client application. There is only a single page, and different views are rendered in it.
- **Routes** - Specifies the different routes that exist in the client and creates a router with them.
- **Store**, which is divided into:
  - **Actions** - Actions are considered as an event that describes something that happened in the application, in this case it corresponds to a request to our API;
  - **Constants** - Helps joining the Actions and the Reducers by defining an identifier for each state of the action;
  - **Reducers** - Reducers are functions that affect the redux state based on the state or request it receives.
- **Themes** - Contains the customization and theme of the material UI components.

Chapter 6

User interface and Functionalities

The application's graphical user interface (UI) will be presented in this part. The responsive views below were created with the goal of delivering information with as few clicks as possible.

6.1 Homepage

The application's Homepage is represented by the figure 6.1. This is the first view that a user sees when they open the application.

At this time, the user is encouraged to enter the application if he or she is already a member of the club, or to register/candidate if they choose to proceed with their club membership application.

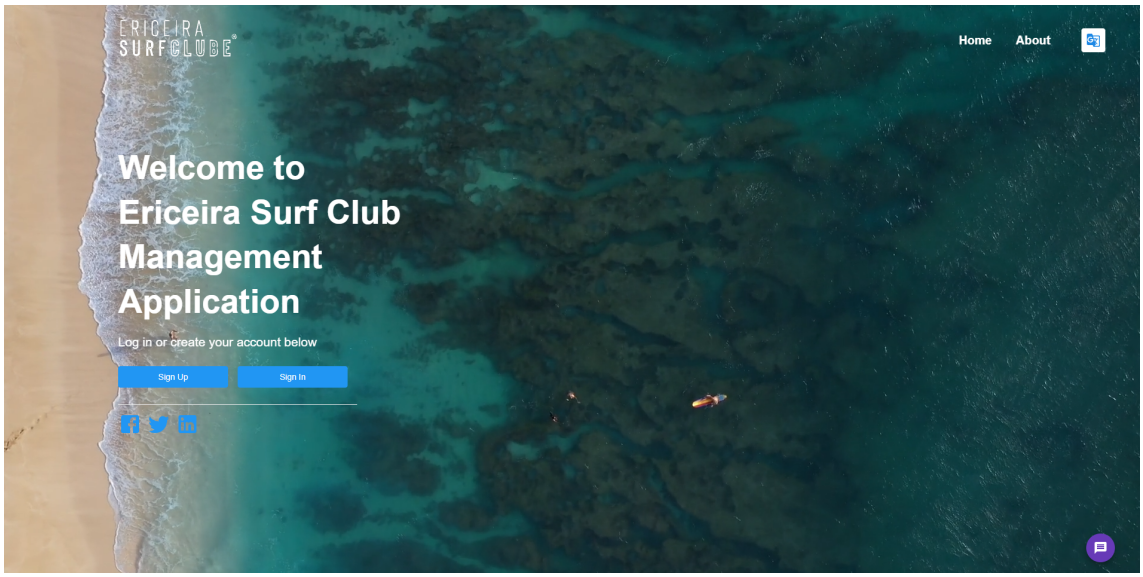

Figure 6.1: Homepage

It is important to note that the language displayed on this view, as well as all others in the application, may be changed.

It is also possible to learn more about the application in the About section, and users have the ability to communicate with the club's administration via email by filling out a form upon clicking the floating button in the bottom right corner.

This floating button is present in every other view of the application, so either a non-member or a member can contact the administration.

In the next figure 6.2, it's shown how is the form displayed:

Figure 6.2: Homepage

6.2 Sign In and Sign Up

It is possible to check the aspect of the view for a user to login in the following figure 6.3. It is also possible to be redirected to the application view if you do not yet belong to the club, or to the password recovery view.

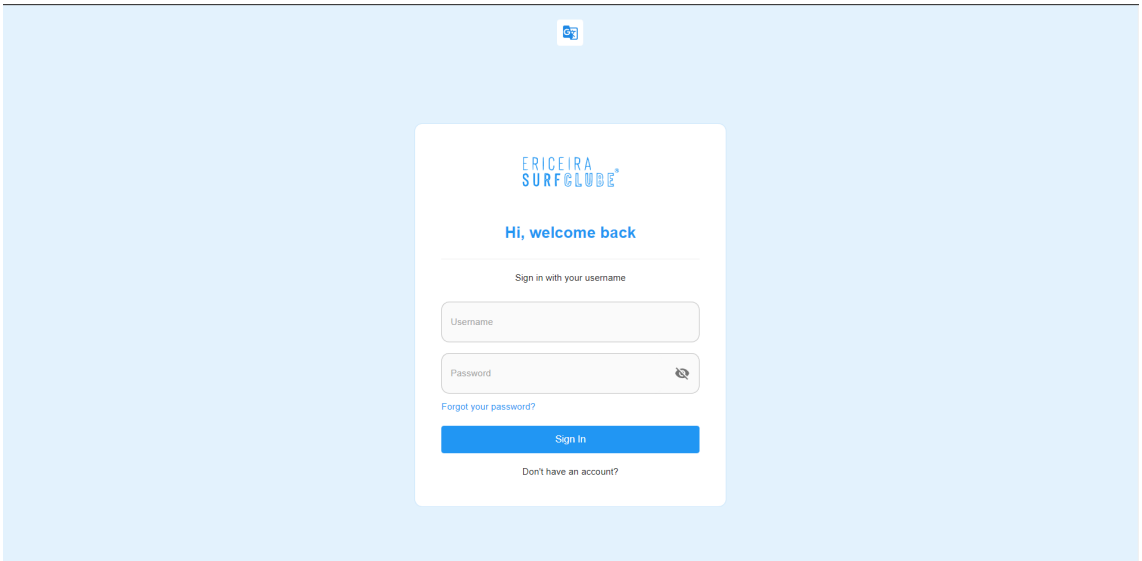

Figure 6.3: Sign In view

For the application procedure, a more comprehensive form must be filled out, including all pertinent information, so that an administrator may determine whether or not the candidate is eligible to join the club. This formula is identical to the sign in view, yet with more input fields, this can be seen in the appendix B.1.

After completing the procedure, the applicant is shown a message saying he must wait to be approved in order to login, this can be seen in the appendix B.2.

6.3 Overview

This view's sole purpose is to give the user the ability to reach his respective views without going to the NavBar.

Apart from what is available on the overview view, it is also possible to access the application's other views, such as those dedicated to members, sports, quotas, events, and finally, candidates.

The overview view is shown in the below figure, along with the Sidebar on the side, which has all of the application's options because the user who completed the login is the administrator.

However, if the person who logs in is not an administrator, their dashboard overview will have different contents, and the Sidebar will not provide access to views that include general information about the application or information about other users.

In the following figure 6.4, it's shown an example of an overview view.

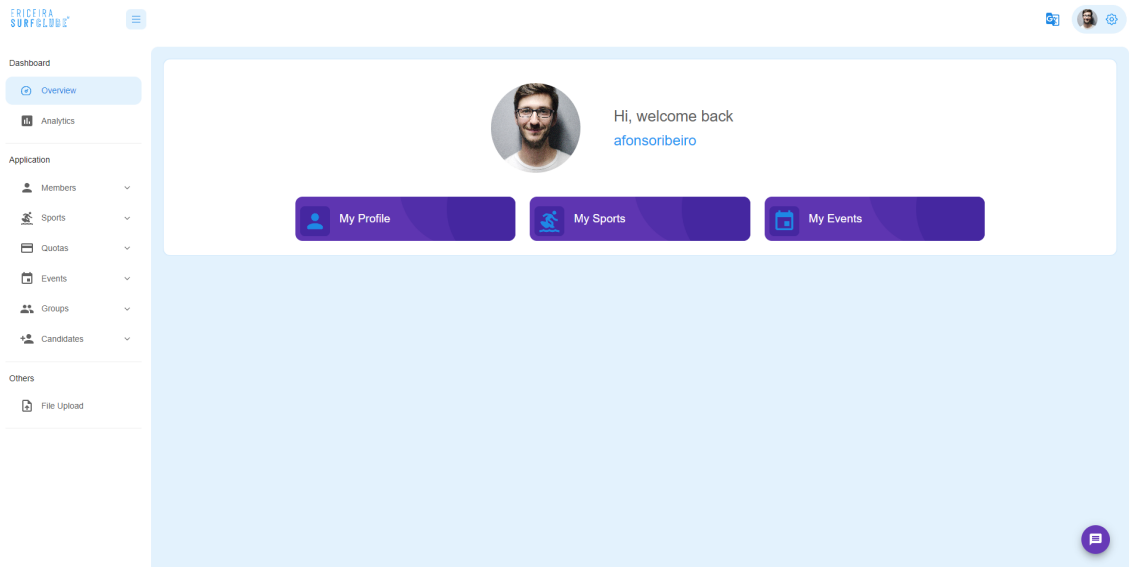

Figure 6.4: Overview from from an administrator view

In the following figure 6.5, it's shown an example of an overview view seen by a common user.

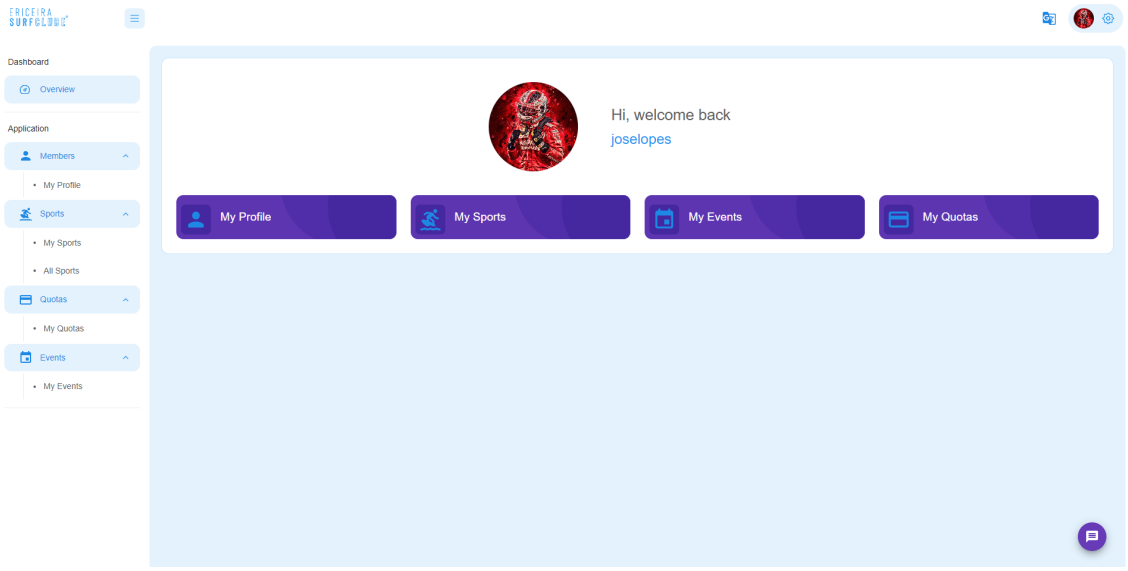

Figure 6.5: Overview from a non-administrator view

It's important noticing that the admin user does not have the *'My Quotas'* hyperlink because he is a founder and does not have to pay quotas.

6.4 Statistics

This view can only be seen by a club's administrator, and has the purpose of giving the member some statistics of the most important topics about the club, such as Quotas, Events, number of Users, Companies and Candidates, Sports and the growth of the club in terms of members.

In the first graphic, it is presented the total amount paid per month in the year selected in the top right corner. This graphic can be downloaded to a *CSV* file.

The second graphic is divided in slices, and each slice represents the amount of people that has the respective state for the select event in the top right corner. The only events able to be selected are the events upcoming in the next 7 days.

The 3 cards in the middle, give the administrator the information about how many users, companies and candidates, respectively, are currently in the club. Both users' and candidates' card can be restricted by their gender or nationality, in the above select input field. These three elements of statistics can be seen in the following figure 6.6:

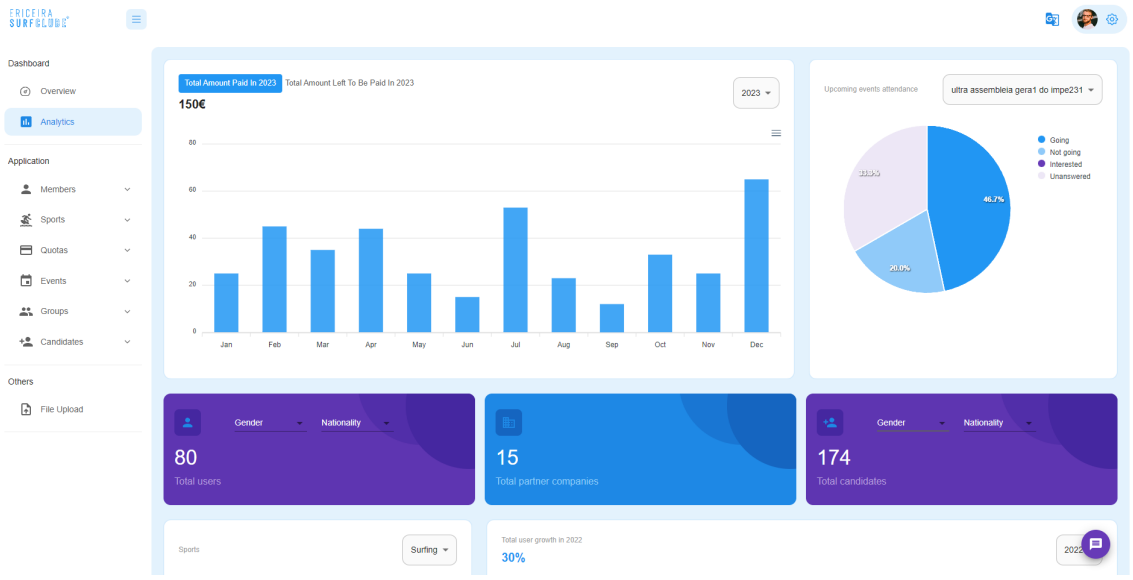

Figure 6.6: Statistics view

In the bottom left corner, there is a graphic that represents a donut divided in slices, and each slice represent the amount of people with that gender is currently practicing the selected sport in the top right corner.

Finally, the last graphic represents the growth of members per month for the selected year, in the top right corner. It's also given the percentage of growth in that year compared to the year before that. This graphic can be downloaded to a *CSV* file.

The last two graphics described above, can be seen in the following figure 6.7:

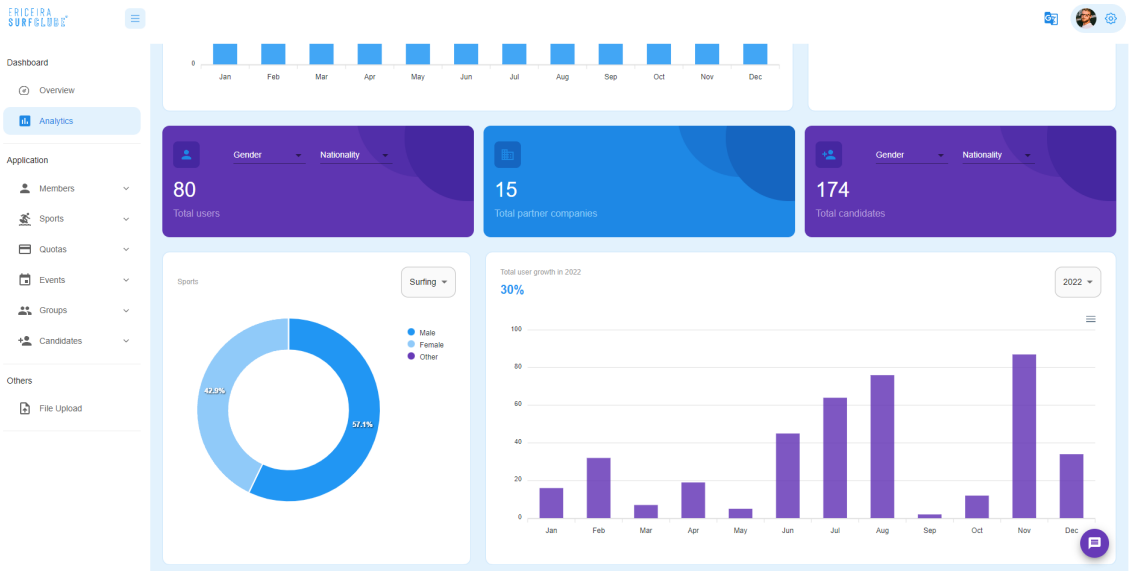

Figure 6.7: Statistics view

6.5 Members

This section is going to describe how the data about the club members is displayed on the user interface and how it is managed.

6.5.1 Profile

The visualization of the member's profile view is shown in this section.
This view has two types of views, which are:

- individual user profile visualization;
- visualization of a user's profile that represents a company.

However, only the administrator has access to all of the members' profiles, and the rest of the members can only see their own. Aside from that, the only people who may make changes to the data in a given profile are the person who owns it and the administrator.

The 'links' tab is only available for an administrative member, and it contains buttons that redirect the admin to every view that represent information for the specific member, for example, the button to member quotas, will redirect the admin to see the view of all quotas that the member is assigned to.

All non-administrative members have the Tab of *'admin privileges'* deactivated.

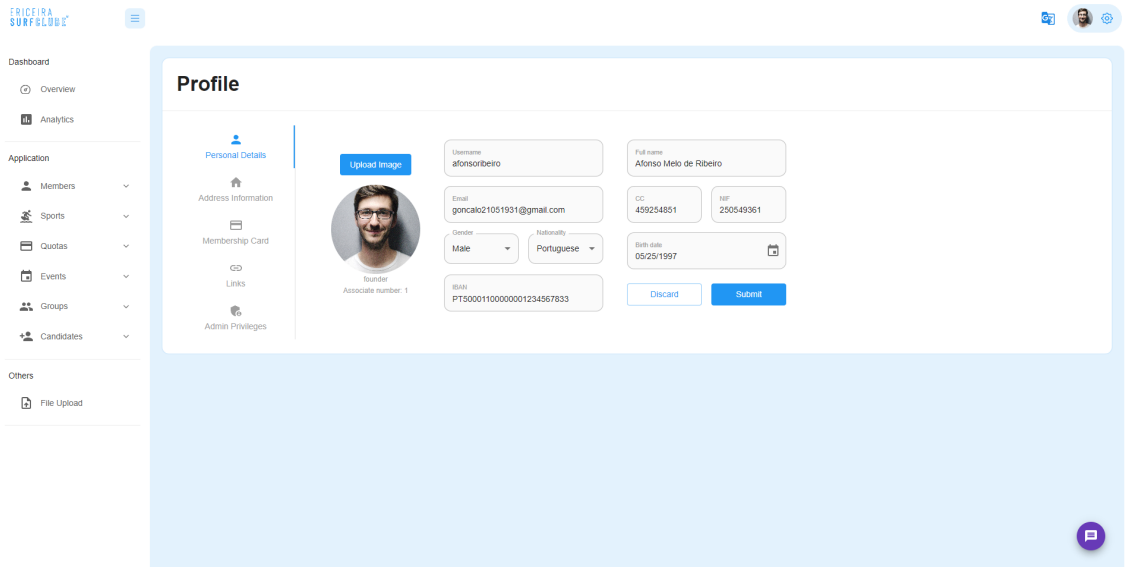

Figure 6.8: Individual user profile view

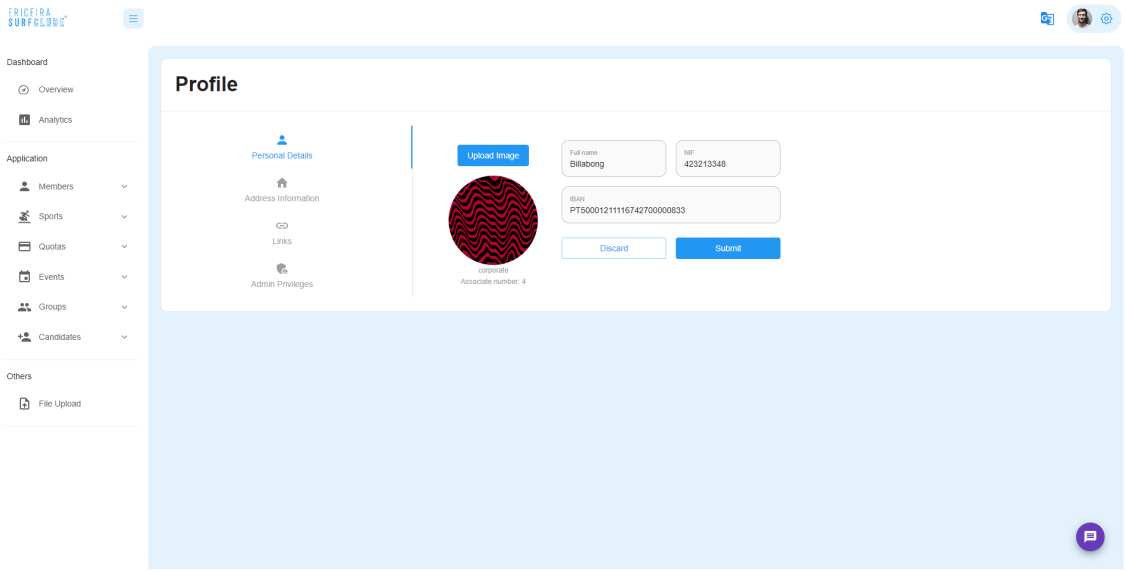

Figure 6.9: Corporate user profile view

As seen in the two figures above, there are differences between the two profile views when the member is an organization or a person, particularly the Tab 'membership card', because the corporate member does not possess a membership card.

6.5.2 All users and All companies

The visualization view for all individual members allows you to see a user's most important data in a table format with the rest of the club's users. This view can only be accessed by an administrator.

It is possible to be redirected from this table to a user's personal profile, as well as to delete it. However, this deletion is subject to the soft delete condition, which means that the data remains in the database but with just a flag indicating that the user has been removed.

Given that visualizing tables might be difficult due to the large number of data points, the ability for users to filter by username, full name, email and has debt flag has been included, it is also included the amount rows that the user wants to see in the table. This topic is covered in further depth in the pagination and filtering chapter 8.

The final feature of both of these views, is the export button in the bottom left corner, that export to a .csv file every single user currently in the club.

The following figure 6.10 is an example of what was described:

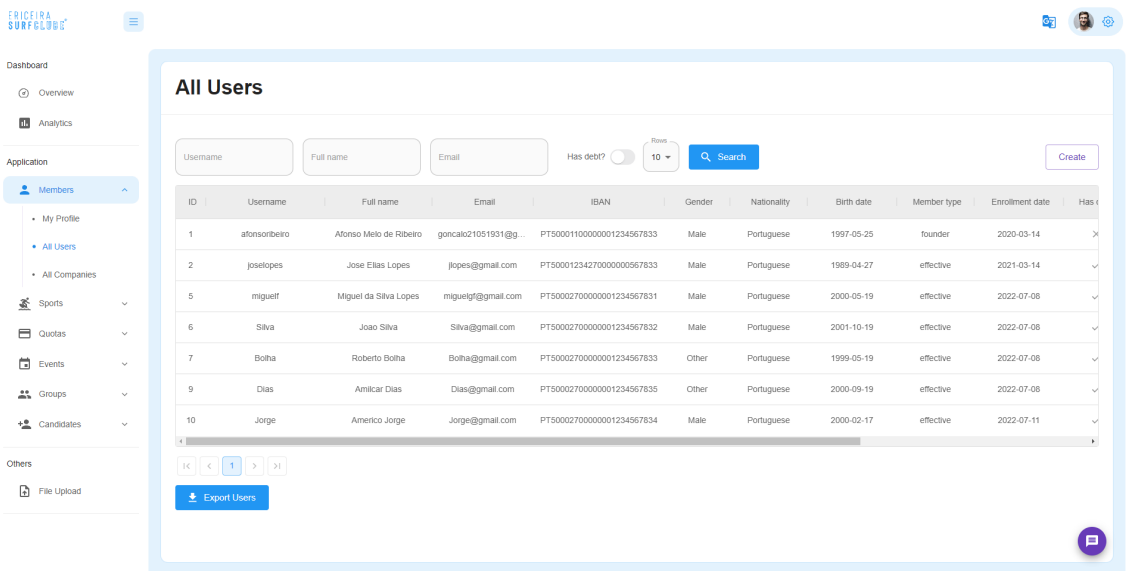

Figure 6.10: All users view

Aside from data visualization, this view allows you to create a user by displaying a Dialog [39], figure 6.11, which is similar to a candidate application form. This feature is crucial for an administrator's manual addition if the member does not need to go through the application process.

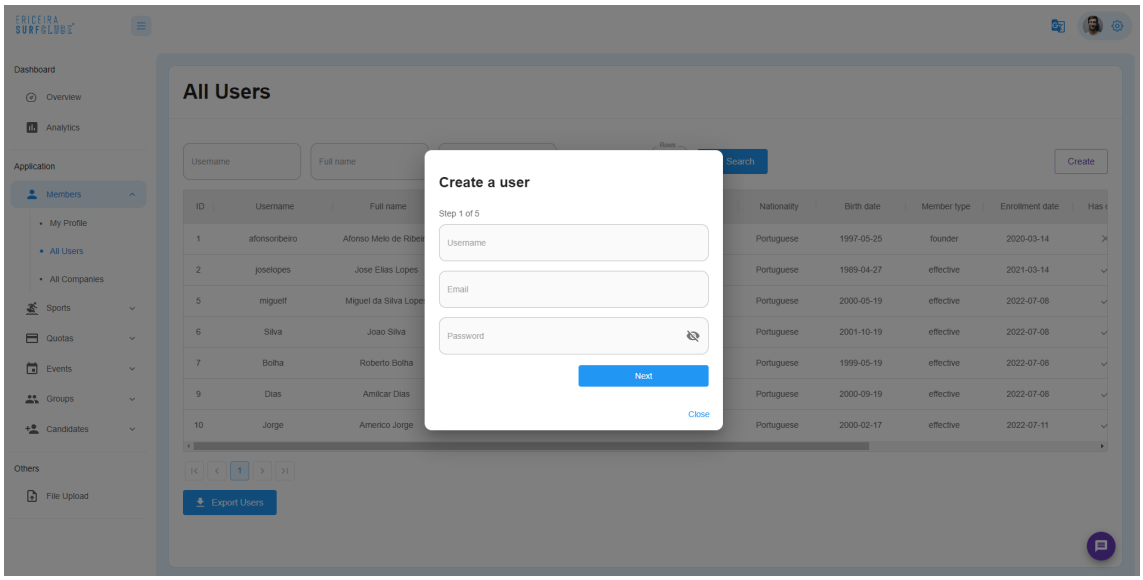

Figure 6.11: User creation modal

The view of all companies is identical to all users, with the exception of the table's content and the creation form that has different data to fill in.

6.6 Sports

This section is going to describe how the data about the sports of the club is displayed on the user interface and how it is managed.

6.6.1 My Sports

On this view, you may find information on various sports that the user is in, in a table format, along with the user's kind of position in relation to the sport, his federation number, the federation's number and name, the years he was federated, and if he still practices.

The user can edit his own data, like his current number o federated years, and can disassociate himself from that sport.

This information is shown in the appendix B.3.

6.6.2 All Sports

On the view dedicated to each sport, you may see all of the sports that the club offers for practice, which are shown using Cards [40].

These Cards provide information on the total number of participants in the sport, the ability to remove them (soft delete) and finally, the ability to be redirected to a specific view for the sport or if you are non-administrative member candidate to the sport, to make flexible the user and sport association, as shown in Figure 6.12.

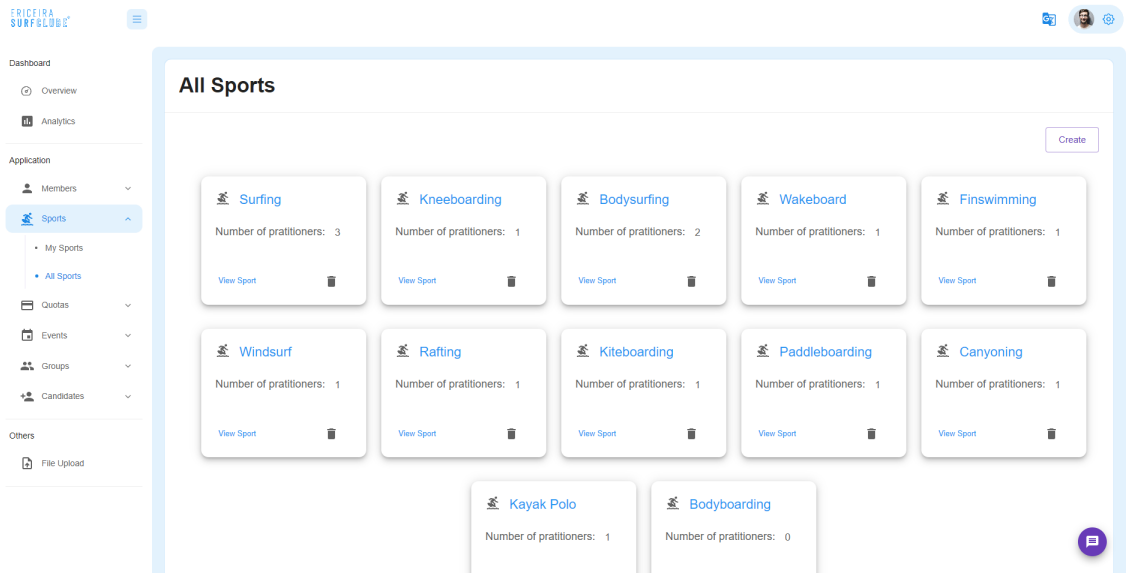

Figure 6.12: Sports view

The button to create a new sport it only appears to an admin member. Similar to every other creation button, it's popped up a dialog, as shown in the following figure 6.13:

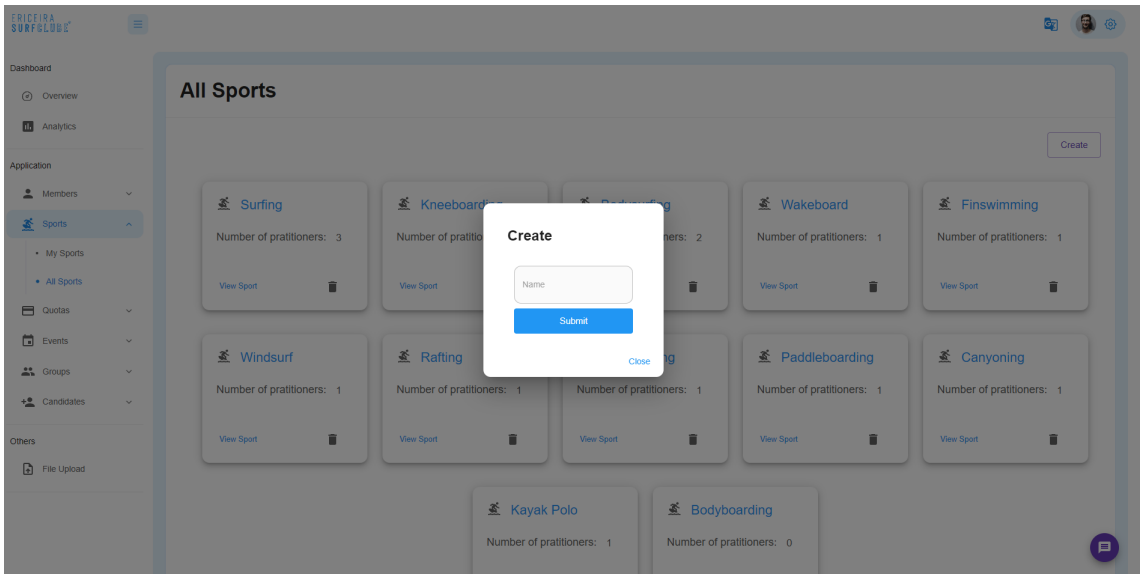

Figure 6.13: Sports creation modal

6.6.3 Sport

This view contains a table of the members associated to the specific sport. Given that visualizing tables might be difficult due to the large number of data points, the ability for users to filter by username has been included, it is also included the amount rows that the user wants to see in the table. The is candidate flag is used to switch to a table that has all candidatures to the sport.

Every row has two types of actions, edit to updated the information of the user sport association, and delete (soft delete) in case of a user's retirement.

This information is shown in the appendix B.4.

Above the table, on the right side, there is a button that has the purpose to associate directly a user to this sport, without going through the candidature phase. The association is done via a modal, as every other creation operation.

The association modal can be seen in the following figure 6.14:

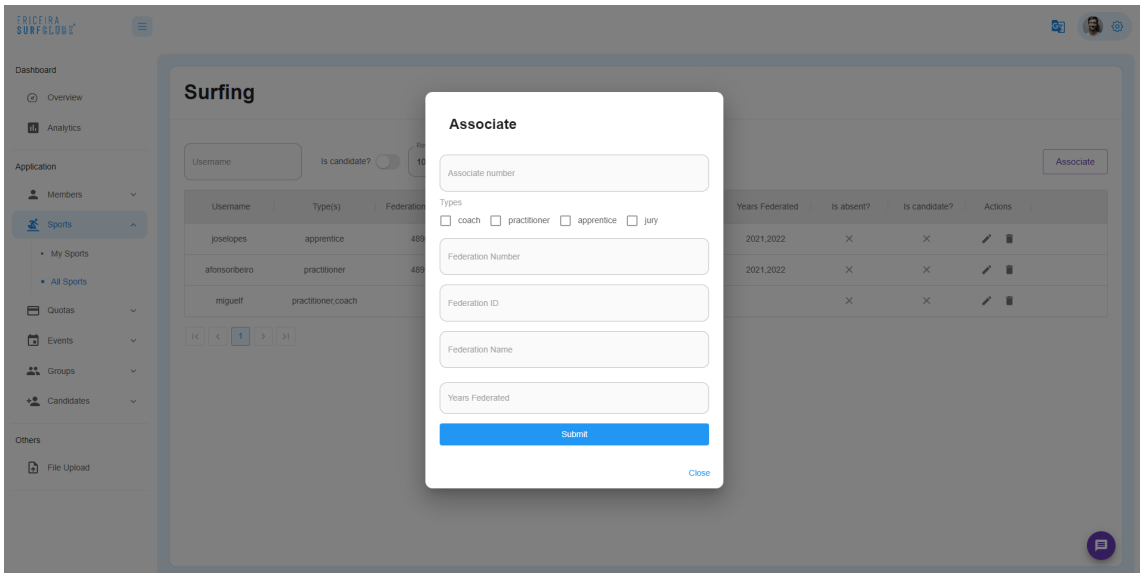

Figure 6.14: Associate user to a sport modal

6.7 Quotas

This section is going to describe how the data about the quotas of the members of the club are displayed on the user interface and how it is managed.

6.7.1 My Quotas

On this view the user will find information about his own quotas, namely the quota date and the payment date, which has a date if the quota was already paid or is empty if the quotas is still for paying. This information is shown in the appendix B.5.

6.7.2 All Quotas

This view is only accessible for admins, on this view the admins will have information about all the quotas of the club.

Each row represents a quota of an user and shows the username, email, phone number of the user as well as the date and payment date of the quota it also has a button so that the admin can signal that the quota was paid. This view has the ability to filter and view data.

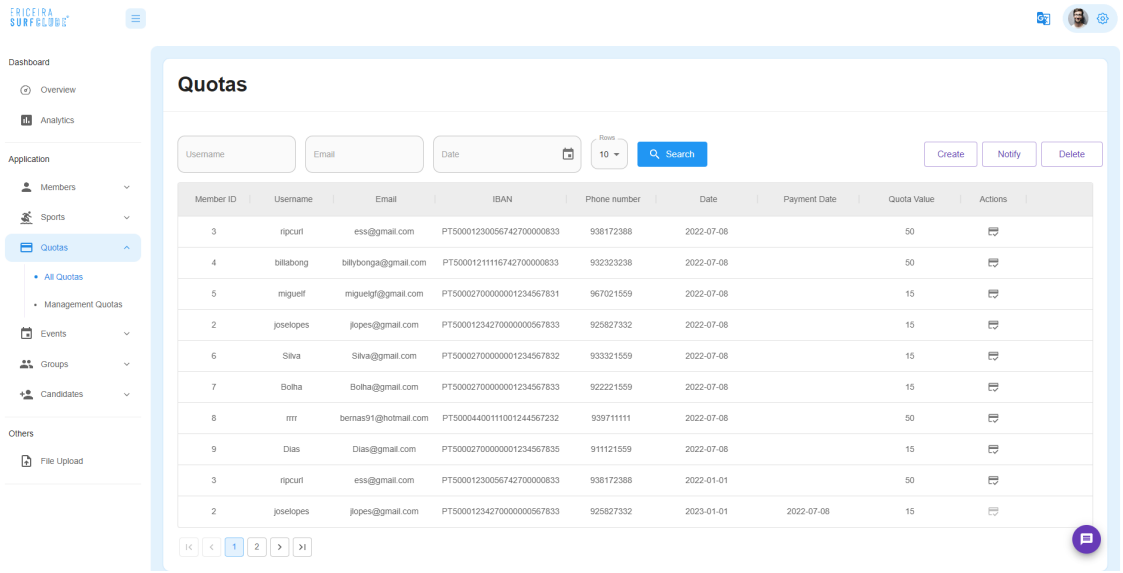

Figure 6.15: All Quotas view

Aside from data visualization, this view allows you to create a new quota by displaying the dialog in figure 6.16.

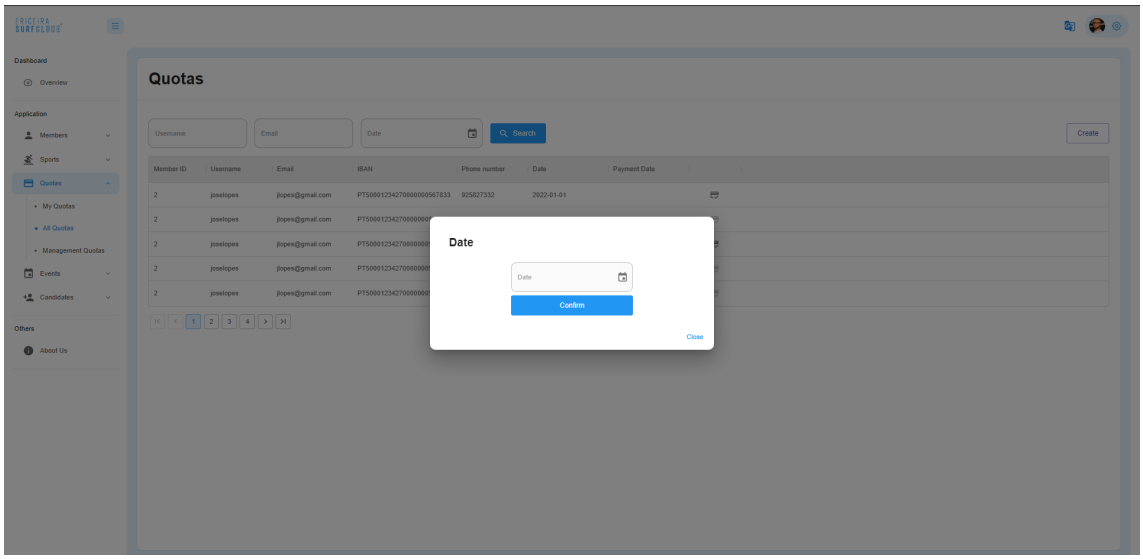

Figure 6.16: Quota Creation Modal

Besides that, the view has the ability to delete all quotas by the date, but the already paid quotas with that will not be deleted. The deletion is made by the dialog in figure 6.17.

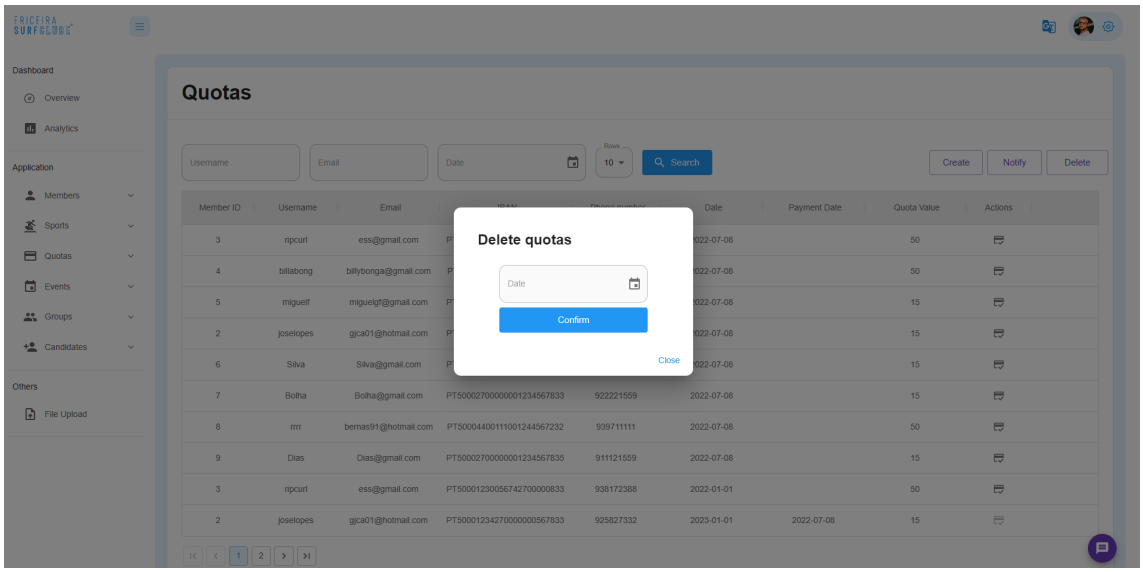

Figure 6.17: Quota Deletion Modal

It also has the ability to notify all users that haven't paid their quotas via an email. The notify is a two way steps, made by a dialog, to verify if the member is sure about notifying every member, as shown in the figure 6.18.

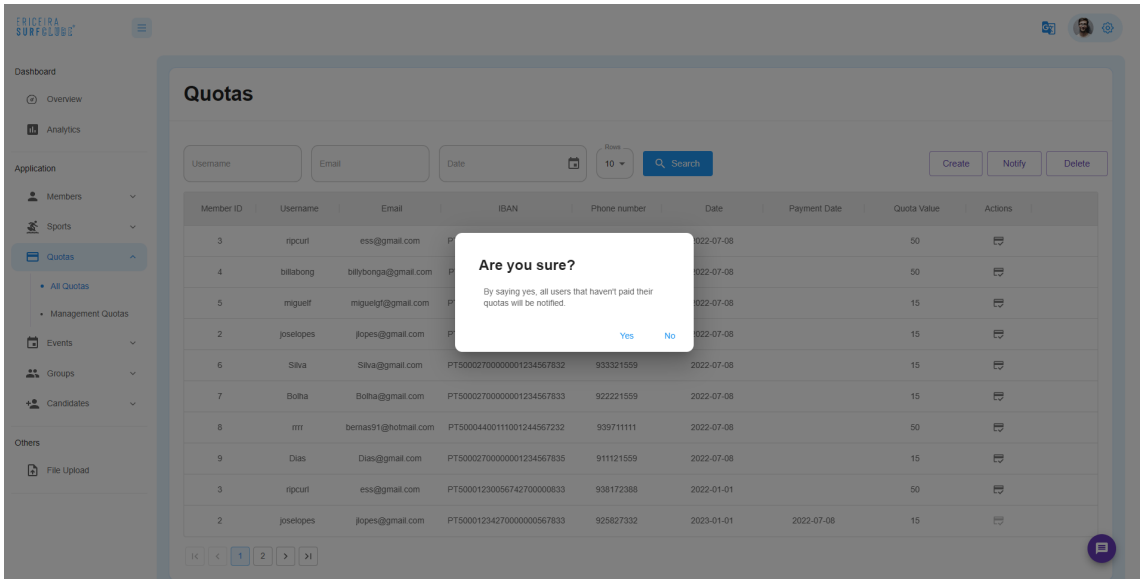

Figure 6.18: Quota Notify Modal

6.7.3 Management Quotas

This view will only be available for admins, and has the purpose of creating a new member type with a new quota value, and visualizing and edit the existing ones.

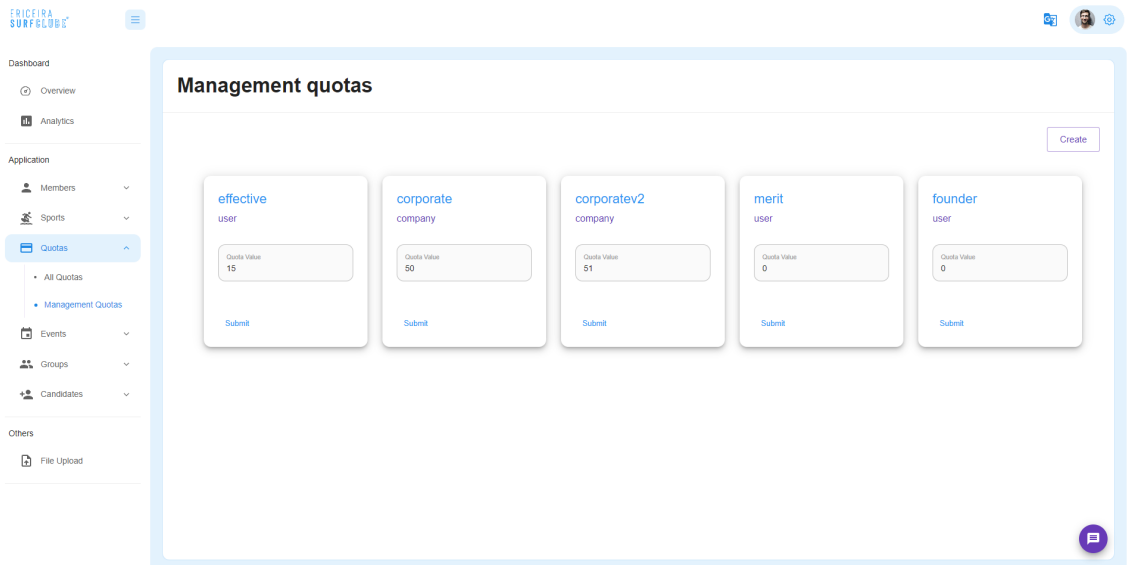

Figure 6.19: Quota Management view

The creation is via a modal, as shown in the figure 6.20.

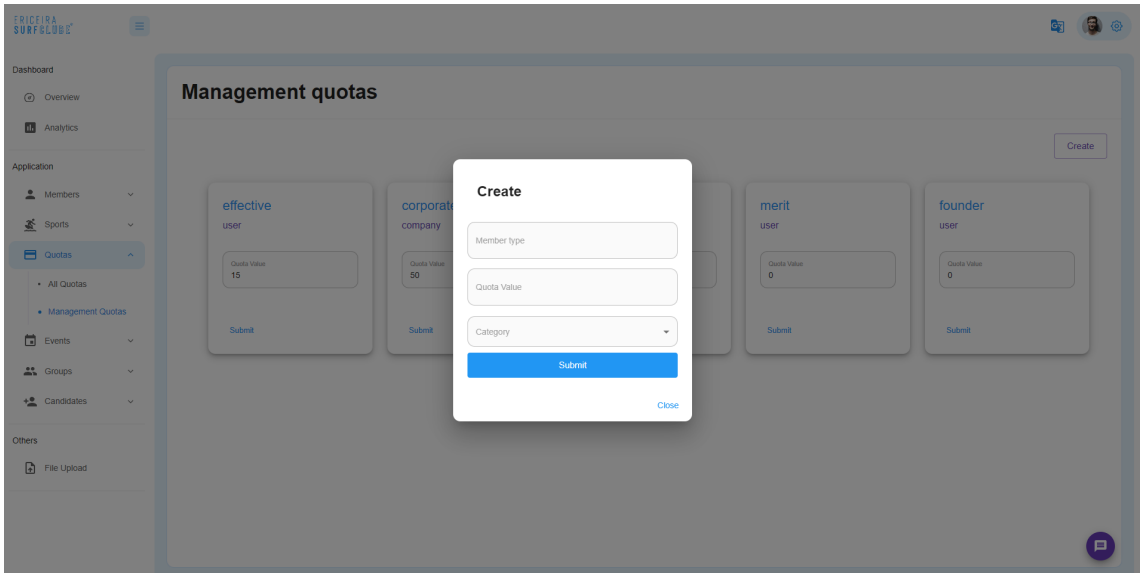

Figure 6.20: Quota management creation modal

6.8 Events

This section is going to describe how the data about the events of the club are displayed on the user interface and how it is managed.

6.8.1 My Events

On this view the user can see the different events that interacted with, showing the state, that can be going, not going, interested or none, the initial date and end date as well as the name of the event. The appendix figure B.6 represents what was previously described:

This state can be change by clicking on the action present in every row of the table, showing up a modal, as shown in figure 6.21 to change or set a new state to the selected event.

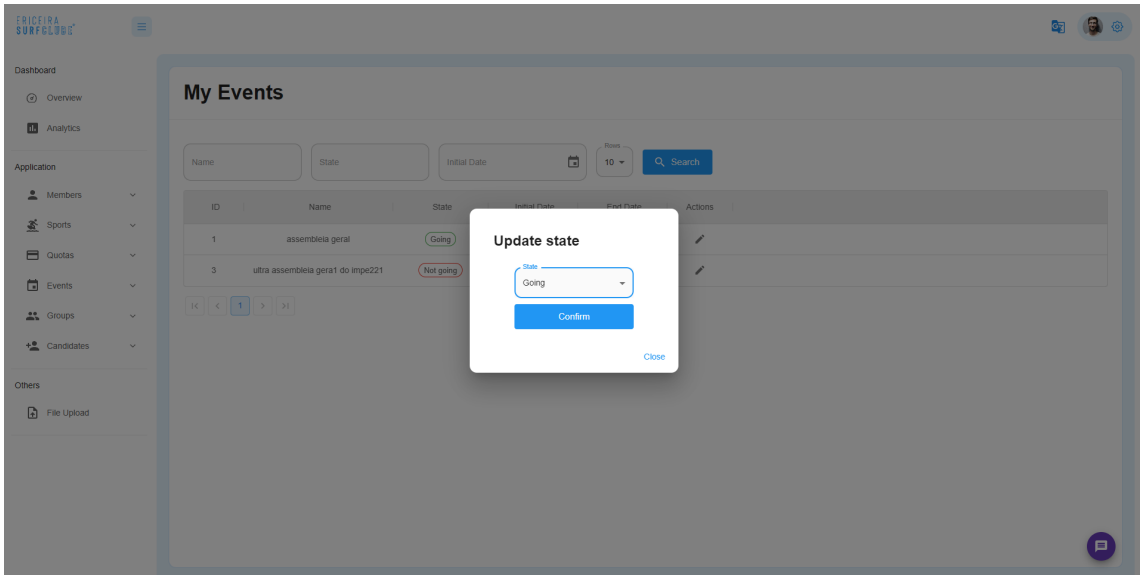

Figure 6.21: My Events view

6.8.2 All Events

This view is accessible for all the users and show all the events of the club, the ones that already happened, happening or even the ones that didn't start yet. Each event has a recycle bin icon so that a admin can delete the event, or has a calendar icon to go to the specific event view. This view can be seen in the appendix figure B.7.

This view also has the ability to create a new event, the same as every other view, by clicking in the button on the top right corner, and filling up the form that the modal contains.

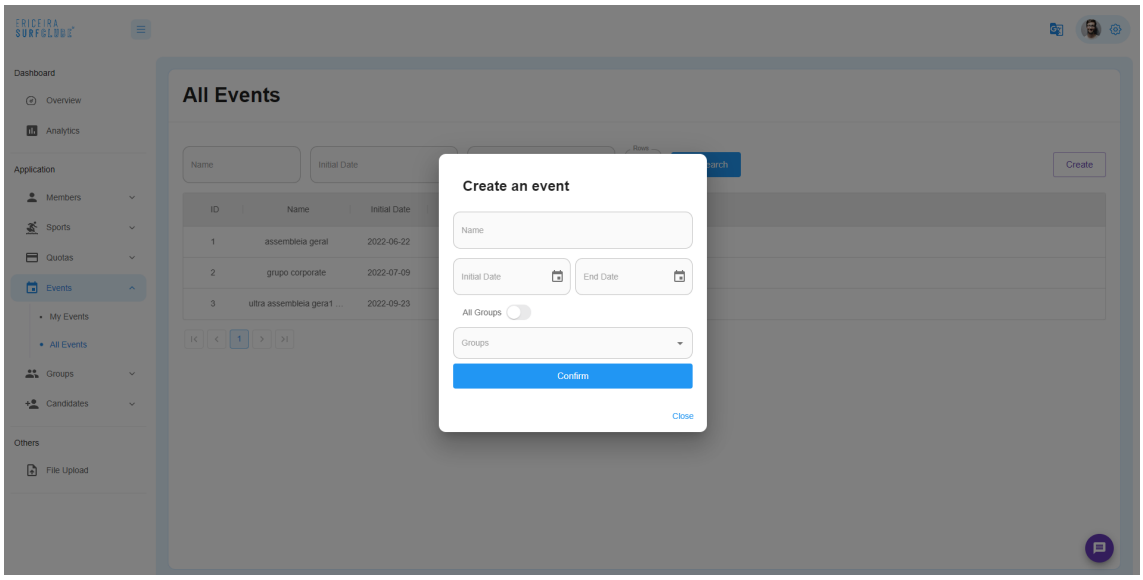

Figure 6.22: Event creation modal

Both my event and all event views have an icon to access an event view with a bit more information about the event.

6.8.3 Event view

This view shows all the information about a specific event, end and start date, a list with all the users that reacted with the event and a pie graphic that represents the attendance to the event, similar to the graphic shown in statistics view.

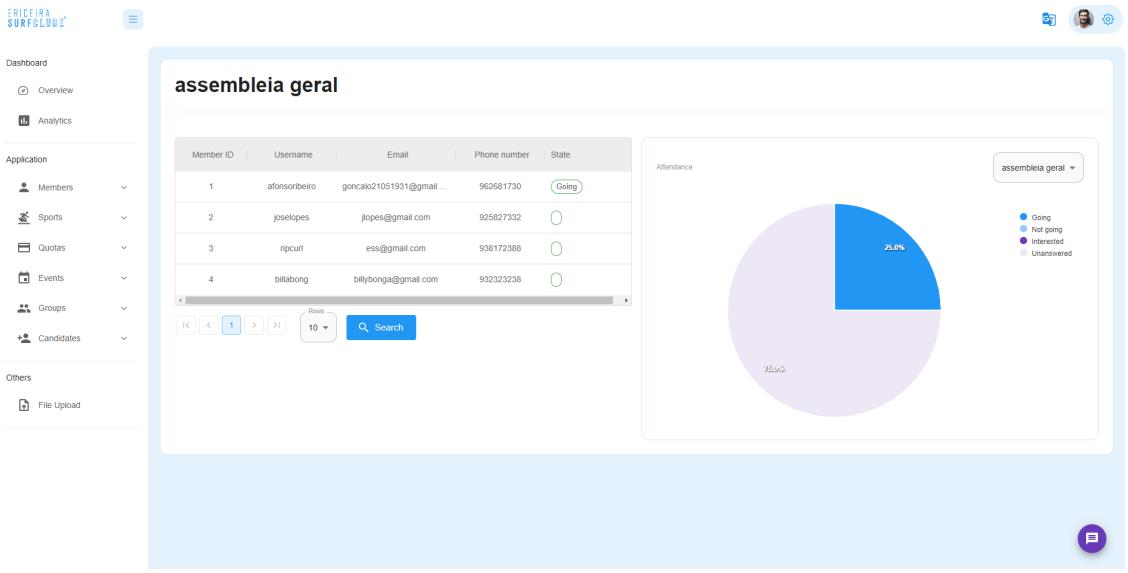

Figure 6.23: Event view

6.9 Groups

This section is going to describe how the data about the groups of the club are displayed on the user interface and how it is managed.

6.9.1 My groups

On this view, it's shown a table containing the groups that the user is linked with. Each row has a button that goes to the view of the individual group. This view can be seen in the appendix figure B.8.

6.9.2 All groups

This view sole purpose is to show all groups that the club currently has, having the ability to go to the specific group view or to delete the group. This view is only accessible by a member that is admin. This view can be seen in the appendix figure B.9.

The group creation is made by filling a form in a modal, as shown in the following figure 6.24]. The group is either targeted to certain types of members in the club or certain types of member within the sports.

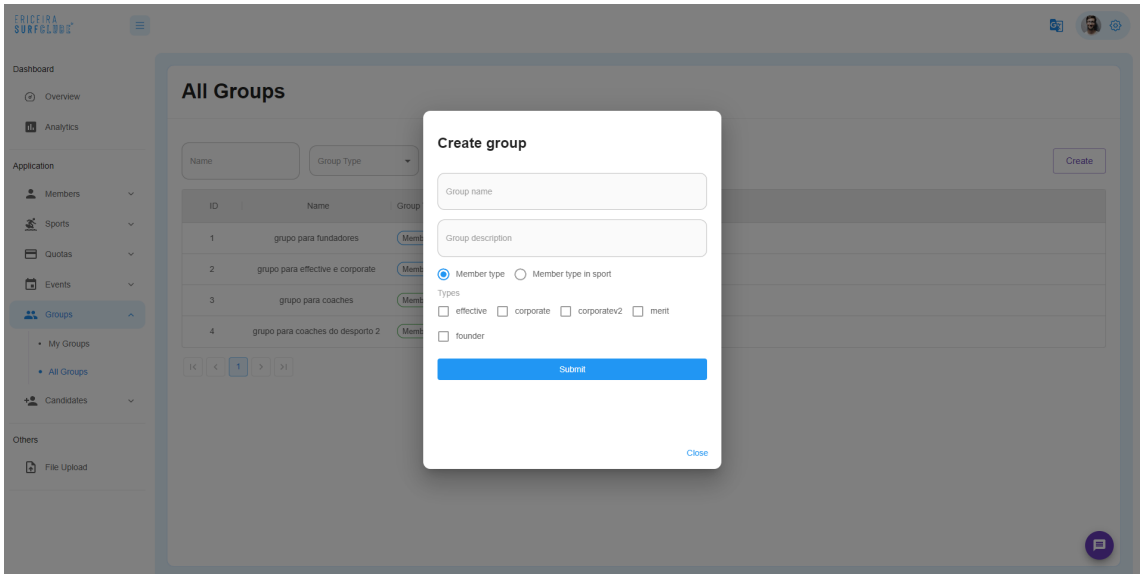

Figure 6.24: Group creation modal

6.9.3 Group view

The purpose of this view is to see which members are currently in the group. Each row has an action that goes to the profile of the member.

The view can be seen in the appendix figure B.10.

6.10 Candidates

The view that corresponds to all candidates is a table that contains all of the information about them, information that is also available in the form referred to in the application process at point 6.2. The previous-mentioned view can only be accessed by users who are administrators. This view also has the ability to export the list of all candidates to a .csv file.

This information may be verified using the appendix figure B.11:

6.11 Password Reset

Resetting the password is a quality of life feature to not lock members out of their account permanently with no solution other than with the intervention of the database manager, the process is rather comprehensive, as the user only needs to input their email, and will then receive an email with a link, valid for 24 hours, where after clicking the link the user will be prompted with the opportunity to insert the new password desired.

The link itself contains the token generated and the id of the member, this is so that it's possible to verify whether the token generated does match the id of the user who made the request.

The process previous-mentioned is shown in the appendix figures B.12 and B.13.

6.12 Upload Data

This feature allows application admins who have data prior to using the application, to import that data. For each type of import, an excel with dummy data is presented that shows how the *CSV* file to be uploaded should be formatted.

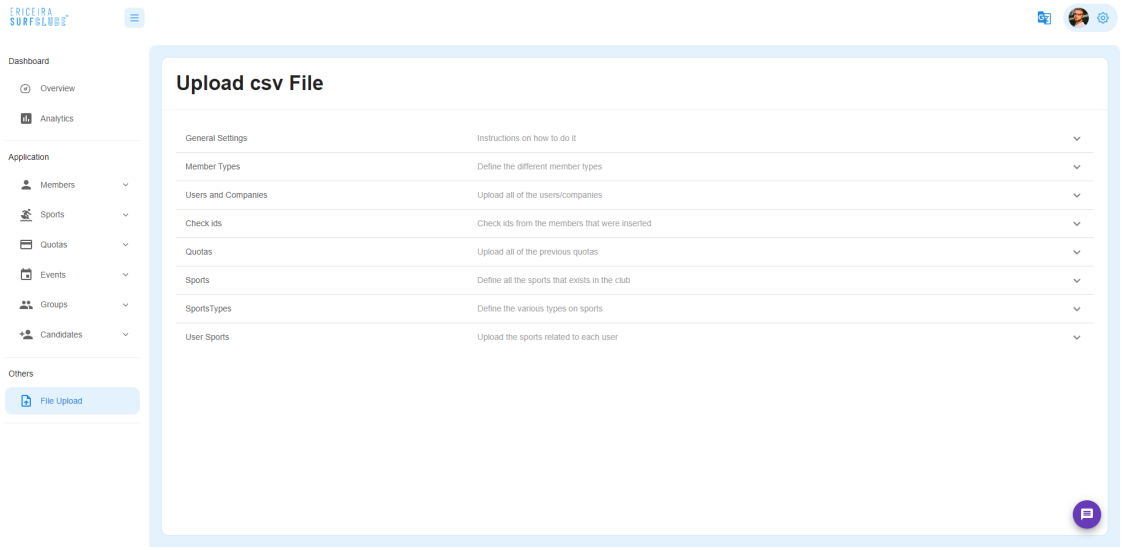

Figure 6.25: Importation Data view

In each section of the import there are 3 buttons, the first one to download the example, the second in which the file to be imported is chosen, on its right side it displays the name of the chosen file, finally there is a button to perform the import.

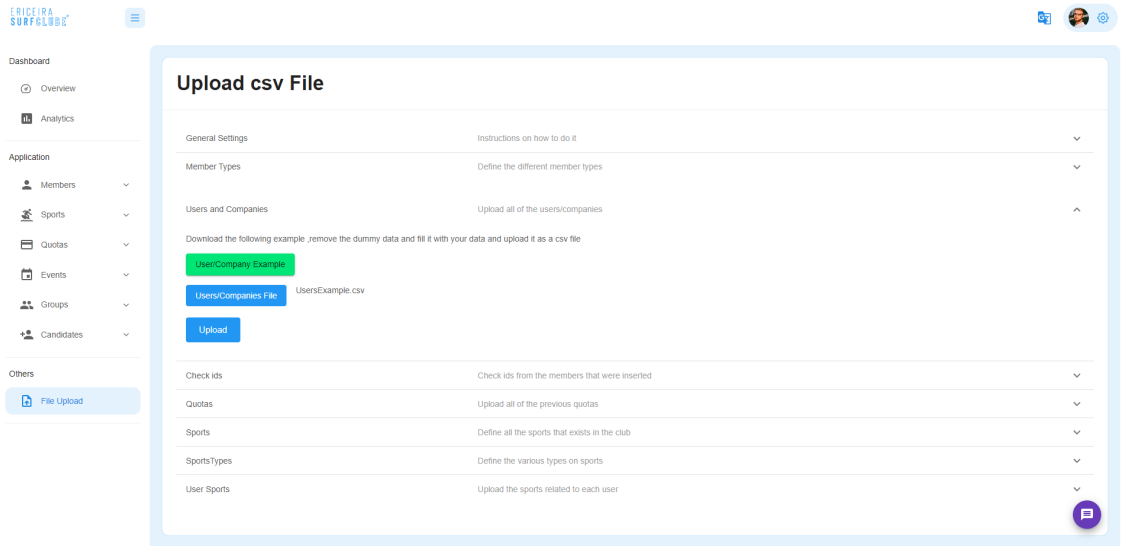

Figure 6.26: Importation Data view Section

6.13 Credentials Change

This feature is only available to members who were added through data import, since in that moment no username and password is defined. This works similarly to resetting the password as the user receives an email with a link that has a token, now with no expiration date, and an id and they are matched to verify if the token corresponds to the id and if the token exists in the database.

6.14 Responsive Design

The necessity for the application to be responsive was one of the major criteria mentioned at the outset of the project.

Responsive web design is a technique for making online views seem well on a wide range of devices and window or screen sizes. The Material UI library was used by the team to achieve this responsiveness, in the online application, several instances of this method are shown below.

As can be seen in the figure 6.27 shown below, the elements are arranged differently depending on the screen resolution, furthermore as can be seen by the Tabs [41] component of MUI, their orientation changes to suit better to the screen where it is inserted.

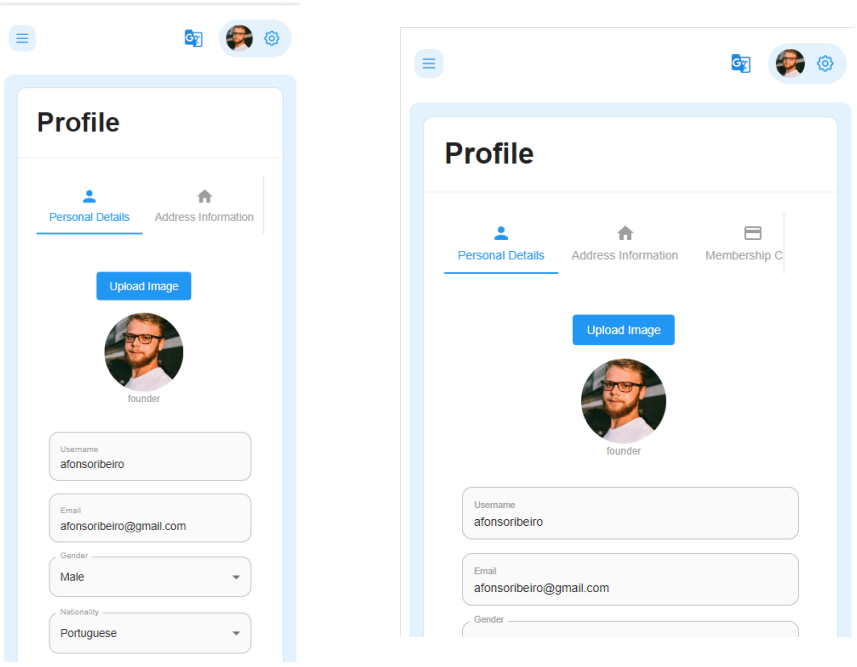

Figure 6.27: Profile in a phone and tablet resolution

Chapter 7

Membership Card

Ericeira surf club offers its members the possibility of discounts at partner stores, so it was important to have a way to identify its members, so the idea of a membership card came up. To be sustainable and environmentally friendly, it was decided that it would remain purely digital.

The card itself contains the members name, role, club enrollment date and a photo of the member, it will also contain a QRCode so that partner companies can validate the members information.

The QRCode is created by using a library named *qrcode* [42], containing the URI for the page of the user's validation. This QRCode is generated when a candidate is approved or when a user is directly created.

Figure 7.1: Membership card example

The user can access his membership card by going to his own profile, and access the Tab *Membership Card*.

The organization's partner companies will scan the QRCode presented on the membership card and will be redirected to a page where they will have to log in (if not yet authenticated) and where afterwards they will be able to validate if the member in question is entitled to discounts.

This process can be observed in figure 7.3 where the result of the validation is based on the members quotas and whether they are all paid for or not.

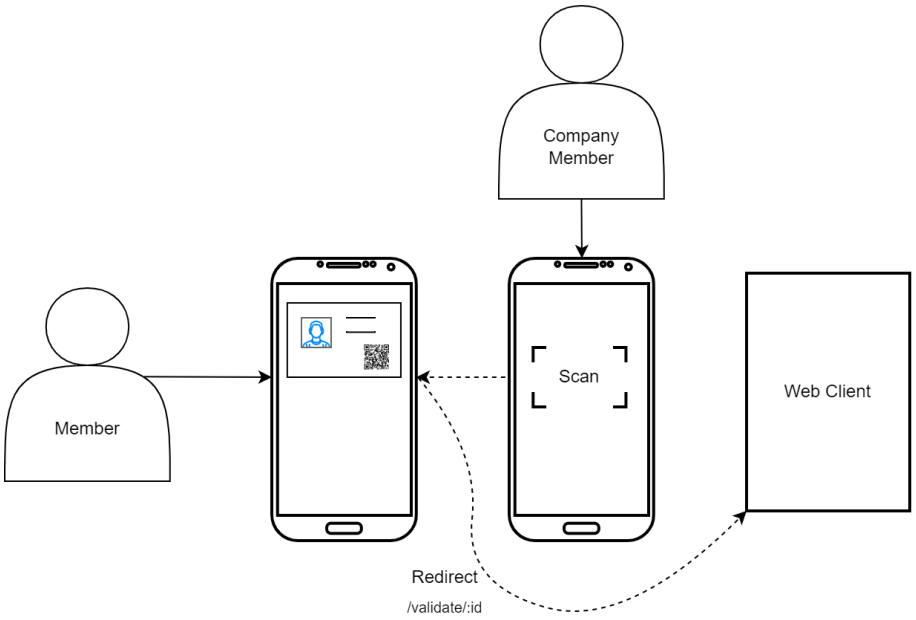

Figure 7.2: Member validation scheme

Figure 7.3: Validation Page

Chapter 8

Pagination and Filtering

8.1 Pagination

Pagination is a mechanism for dividing material across multiple pages and showing a new piece of content each time the page number is modified. Pagination not only makes the user interface more intuitive, but it also saves time over the alternative strategy of fetching all the resources in the database.

This technique is employed across all pages that allow the listing of different items, such as listing all users, companies, candidates, quotas, sports, and sports related with the user.

The sections that follow go through how the technique was implemented on both sides of the application in detail.

8.1.1 Client Side

The client side handles the state of the page number using the react hook *useState* to save the number of the page and the Material UI component Pagination to allow changing pages by altering the state.

This alteration provokes a trigger in the hook *useEffect*, that dispatches a new request to redux, making a new request to the web API sending the limit of rows that need to be fetched and the offset.

By consequence of the request, the page, that subscribed to the state that redux holds, is being modified as the state changes, and when that process is finished, the data has now been loaded into the state and is now rendered in the page.

8.1.2 Server Side

Each endpoint that uses pagination now receives two new required query parameters, limit and offset, that are used when creating the query string in the data access layer. The data access layer then fetches the rows based on the previously mentioned parameters.
A small detail that helps maintain a consistent state, is that each request returns the total number of rows, and since the number of pages is calculated on the client side based on this value, each request can alter the number of pages directly.

8.2 Filtering

Alongside pagination, filtration also helps reduce the load on the network by supplying the server with information that can outline and restrict which rows should be fetched, giving a more pertinent result based on what the application user needs to observe.

8.2.1 Client Side

On the client side, state must be maintained regarding the filtering criteria, since every new request to the API needs these parameters.

This technique consists in a form, that holds different types of information relevant for filtering, and upon submission a new request is dispatched, and much like pagination, the hook *useEffect* is triggered, although this time the page is reset to its original value, it being 1.

8.2.2 Server Side

Each endpoint that previously supported pagination can now also be supplied with query parameters relevant to each endpoint. These new parameters are used at the end of the pipeline, the data access layer, to build complex queries based on the existence of filtering parameters.

Since the filtering works alongside pagination, the previous aspect mentioned regarding pagination also apply to filtering, meaning the total number of rows that fit the criteria is also returned to facilitate the calculation of the number of pages needed.

8.3 Example

In the chapter 6, it's shown that there are many views with tables, and those always have input fields for filtering and for the amount of rows shown in the table, in other words, pagination. In figure 8.1 there is an example of usage.

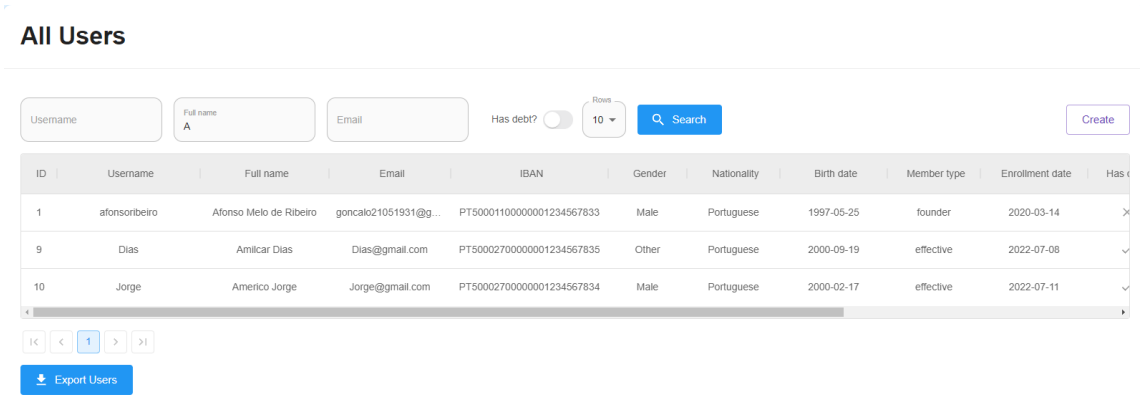

Figure 8.1: Filtering and Pagination example

These two attributes are query parameters in the request for the API. The path for requesting the list of users is through this path */api/users?offset=0&limit=10*. In the data layer these two parameters will be used to filter and limit the amount of rows that will be in the response, as shown in the code list below.

```
const getUsersData = async (username_filter,name_filter,email_filter,
    debt_filter,offset,limit) => {
  let query = queries.QUERY_GET_USERS, queryCount = '', count = 0
  if(username_filter || name_filter || email_filter || debt_filter){
    query = query + " where "
  }
  if(username_filter){
    count++
    query = query + ` position('${username_filter}' in username_) > 0`
  }
    ...
  queryCount = query
  query = query + ` order by u.member_id_ offset ${offset}`
  if (limit !== '-1') query = query + ` FETCH FIRST ${limit} ROWS only`
  const handler = async (client) => {
    const users = await client.query(query)
    const number_of_users = await client.query(queryCount)
    return { users: users.rows, number_of_users: number_of_users.rowCount }
  }
  return await pool(handler)
}
```

Listing 8.1: Query for users with filter and limit

For pagination purposes, the front-end has to know how many rows will be returned in order to determine how many pages for the table will be displayed, which is why the property number_of_users of the returned object is acquired with a query without the offset and limit.

Chapter 9

Conclusion

Through the development of the project, the knowledge acquired along the course was applied and also new technologies that are used in a real world environment. Since this project was devised to cooperate with Ericeira Surf Club the group also had the opportunity to experience how a customer relationship proceeds in terms of development.

The major challenges that was faced during the development of the project was the use of React, as it was a library no member of the group had any experience with and it exposes a lot of different tools, such as hooks like useState and useEffect. Since a choice of separating the render of the page from the request to the API was made, the learning of React-Redux was crucial, and although it is simple to learn and implement, its implementation is extensive which meant a more difficult time on debugging errors.

Despite the difficulties the group faced, the project is progressing well and offers most functionalities, albeit some are to be slightly changed to fit Ericeira Surf Club better, and as positive feedback has been received from the organization we can affirm that the project is progressing well and although a bit behind schedule, it is well within the groups reach to finish in time.

9.1 Future Work

In order to avoid the need for physical copies of important documents, such as those related to federate membership or member identification, it would be useful in the future to have a functionality that could store various documents regarding various members, such as their federation card or citizen card.

Currently, quotas are only being handled manually, being updated by the administrators as the members pay. To fix this, a payment system would allow the application to be more effective since there would be no more need for manual verification and update.

Bibliography

[1] Ericeira WSR+10. https://ericeirawsr10.com/ericeira-surf-clube/.

[2] React Lifecycle. https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/.

[3] Single Page Application. https://en.wikipedia.org/wiki/Single-page_application.

[4] Javascript. https://www.javascript.com/.

[5] React. https://reactjs.org/.

[6] CSV file. https://en.wikipedia.org/wiki/Comma-separated_values.

[7] NodeJs. https://nodejs.org/.

[8] node-postgres. https://www.npmjs.com/package/pg.

[9] Express framework. https://expressjs.com/.

[10] HTTP. https://developer.mozilla.org/en-US/docs/Web/HTTP.

[11] Endpoint, Express. https://expressjs.com/en/starter/basic-routing.html.

[12] PostgreSQL. https://www.postgresql.org/.

[13] Heroku. https://www.heroku.com/.

[14] Redux. https://redux.js.org/.

[15] Entity Association. https://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model.

[16] Async Handler. https://www.npmjs.com/package/express-async-handler/v/1.1.4.

[17] Web API Documentation. https://bernardofmf.github.io/surf-management-app/.

[18] passportJS. https://www.passportjs.org/.

[19] Local Strategy. https://www.passportjs.org/packages/passport-local/).

[20] Cookie. https://en.wikipedia.org/wiki/HTTP_cookie.

[21] BCrypt. https://www.npmjs.com/package/bcrypt.

[22] salt. https://en.wikipedia.org/wiki/Salt_(cryptography).

[23] Hashing, StackAbuse. https://stackabuse.com/hashing-passwords-in-python-with-bcrypt/.

[24] NodeMailer. https://nodemailer.com/about/.

[25] SMTP. https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol.

[26] Outlook. https://en.wikipedia.org/wiki/Outlook.com.

[27] Express-fileupload. https://www.npmjs.com/package/express-fileupload.

[28] Xlsx file. https://www.leadtools.com/help/sdk/v21/dh/to/file-formats-microsoft-excel-spreadsheet-xlsx-xls.html.

[29] Test Driven Development. https://en.wikipedia.org/wiki/Test-driven_development.

[30] Jest. https://jestjs.io/.

[31] Supertest. https://www.npmjs.com/package/supertest.

[32] Virtual DOM. https://reactjs.org/docs/faq-internals.html.

[33] React DOM. https://reactjs.org/docs/react-dom.html.

[34] react-i18next. https://react.i18next.com/.

[35] Json. https://www.json.org/json-en.html.

[36] Local Storage. https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage.

[37] Css. https://developer.mozilla.org/pt-BR/docs/Web/CSS.

[38] React hooks. https://reactjs.org/docs/hooks-intro.html.

[39] Dialog Component. https://mui.com/material-ui/react-dialog/.

[40] Card Component. https://mui.com/material-ui/react-card/.

[41] Tab. https://mui.com/material-ui/react-tabs/.

[42] QRCode Library. https://www.npmjs.com/package/qrcode.

[43] MUI library. https://mui.com/.

[44] DAL. https://en.wikipedia.org/wiki/Data_access_layer.

Appendix A

Entity Association model

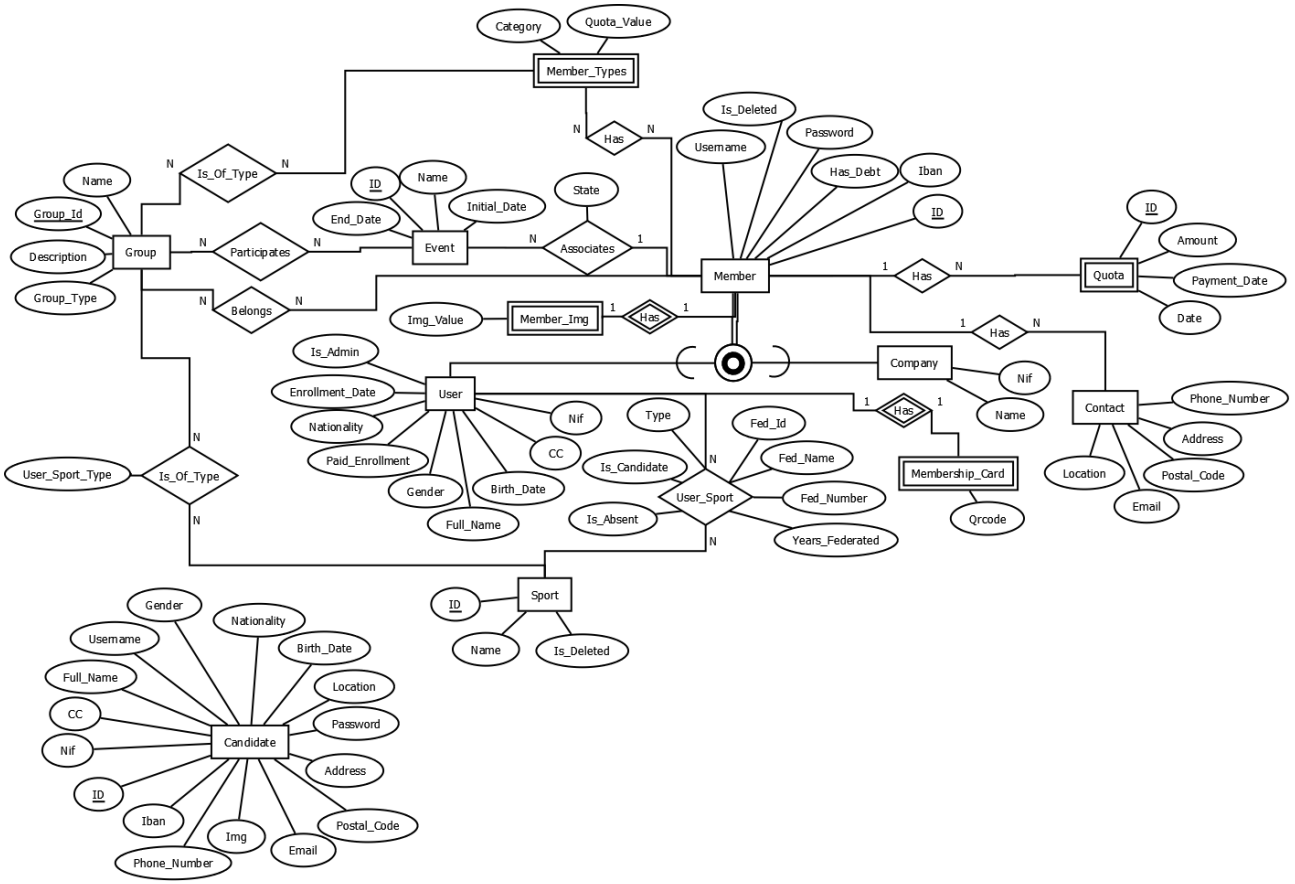

Figure A.1: Full Data Model

Appendix B

Views

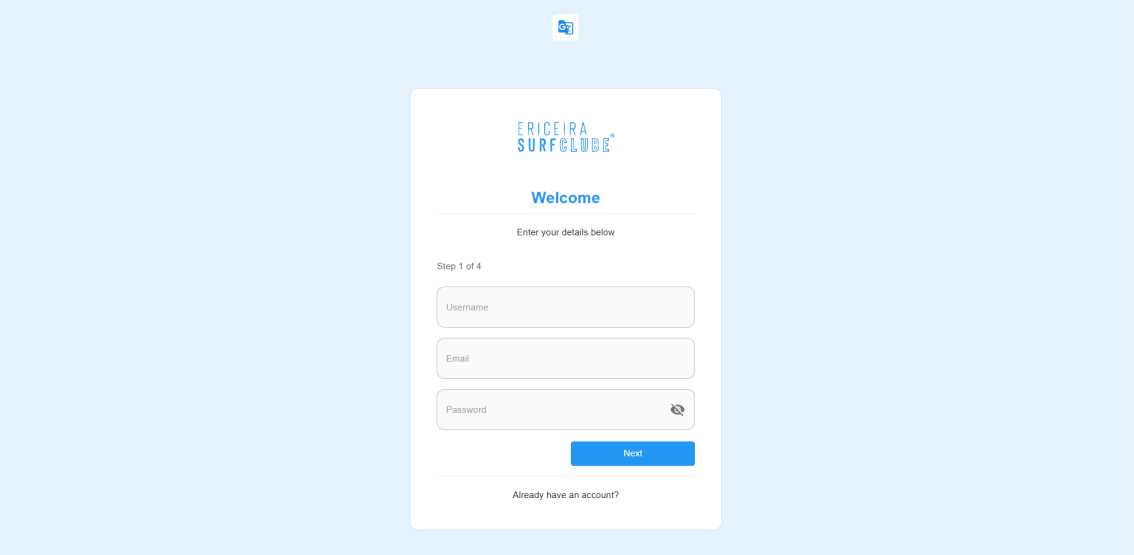

Figure B.1: Sign up view

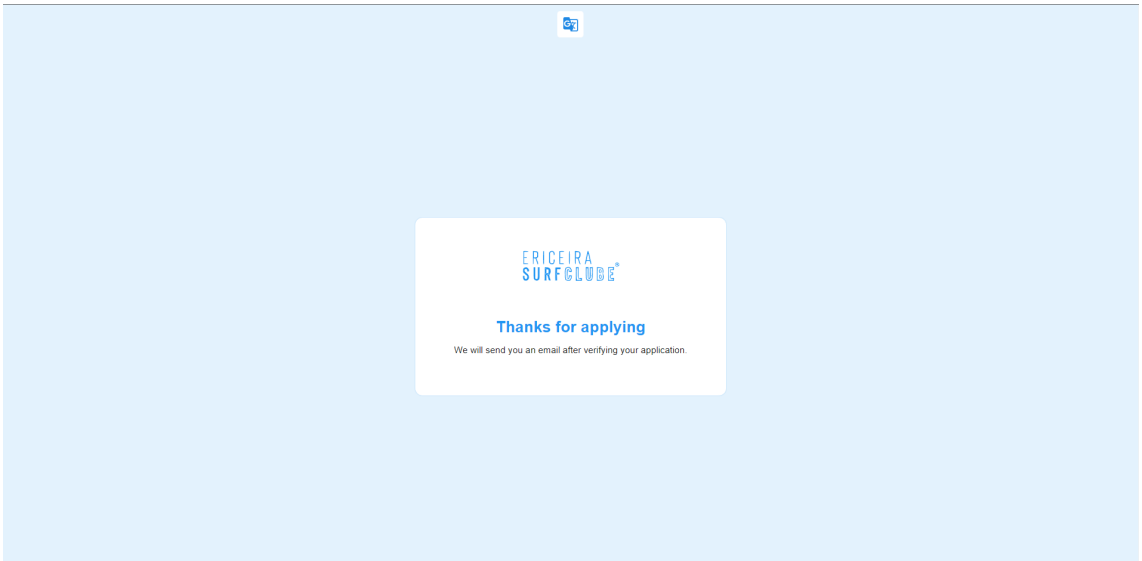

Figure B.2: Application message view

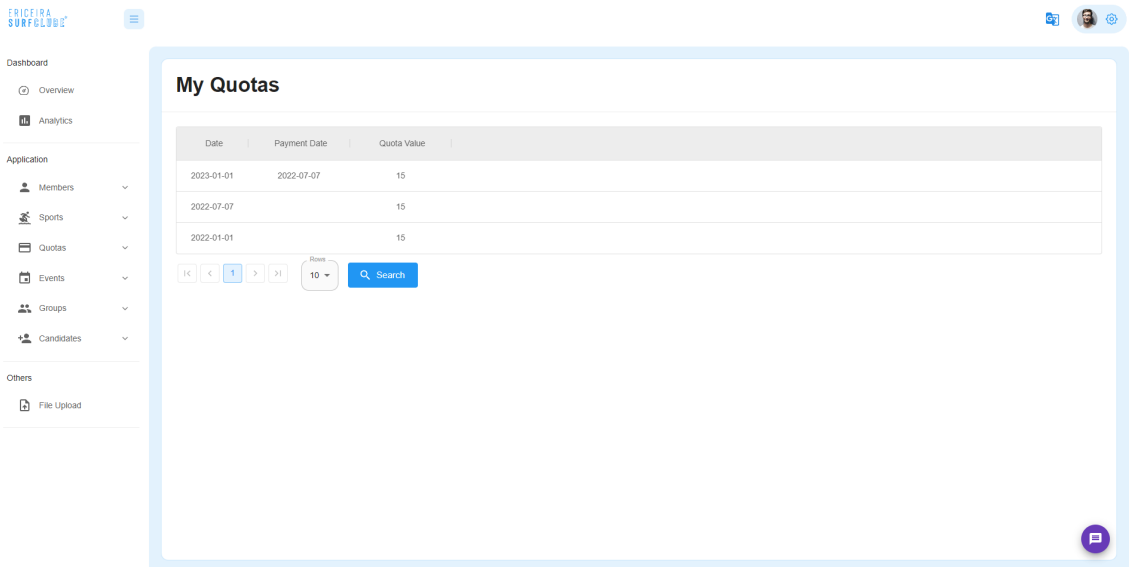

Figure B.5: My Quotas view

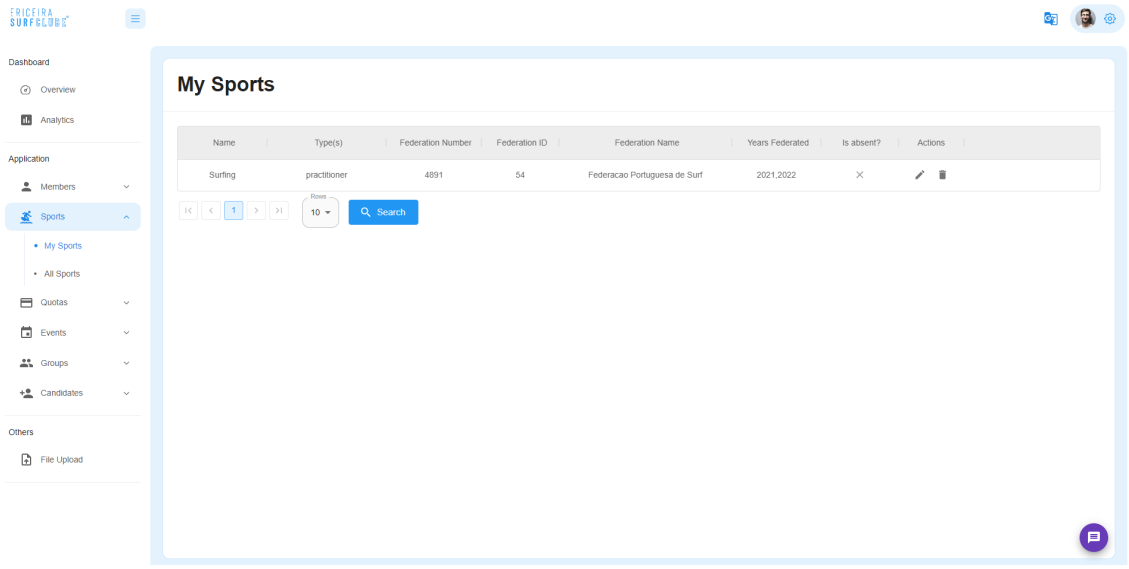

Figure B.3: Member's sports view

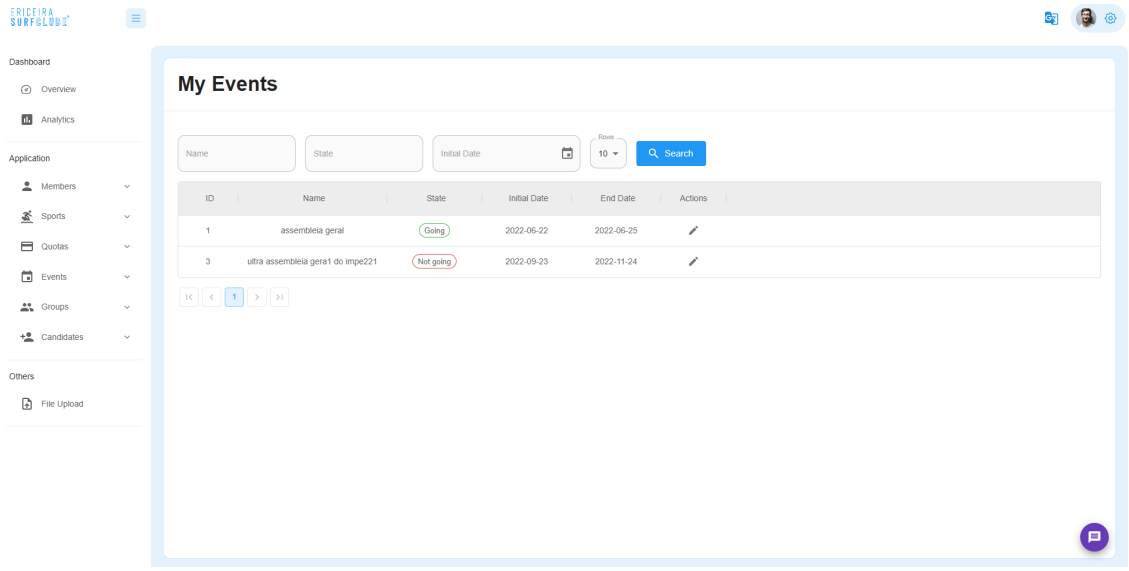

Figure B.6: My Events view

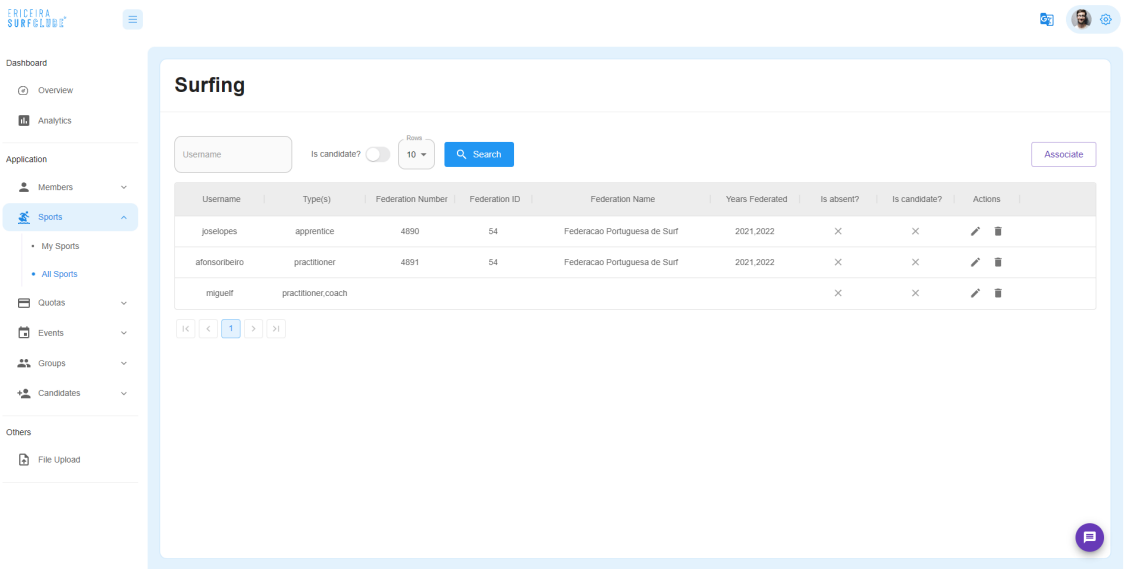

Figure B.4: Member's sports view

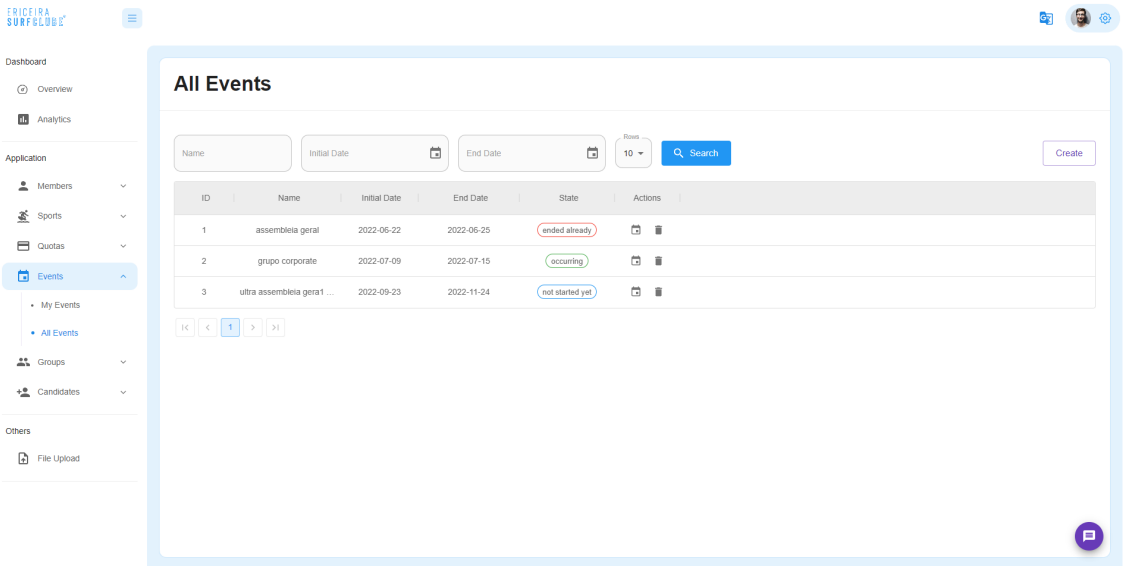

Figure B.7: All events view

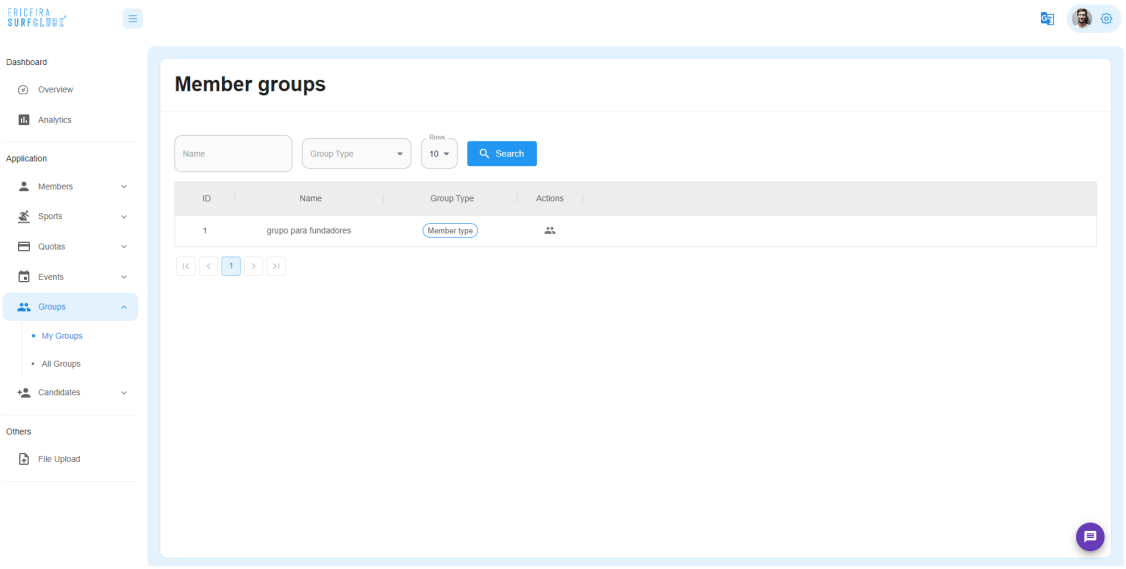

Figure B.8: Group view

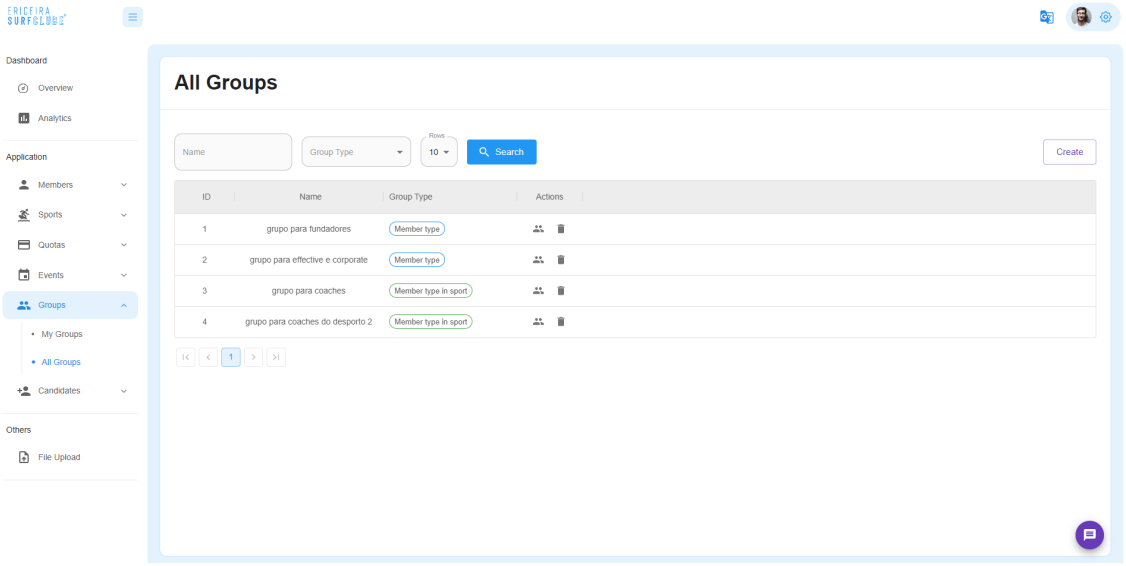

Figure B.9: All groups view

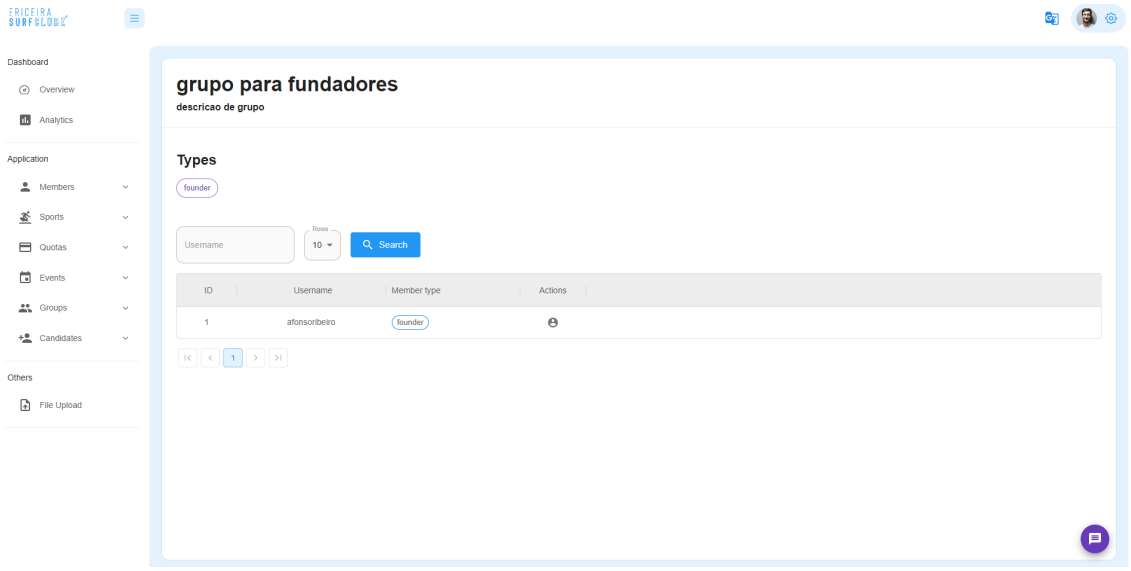

Figure B.10: Group view

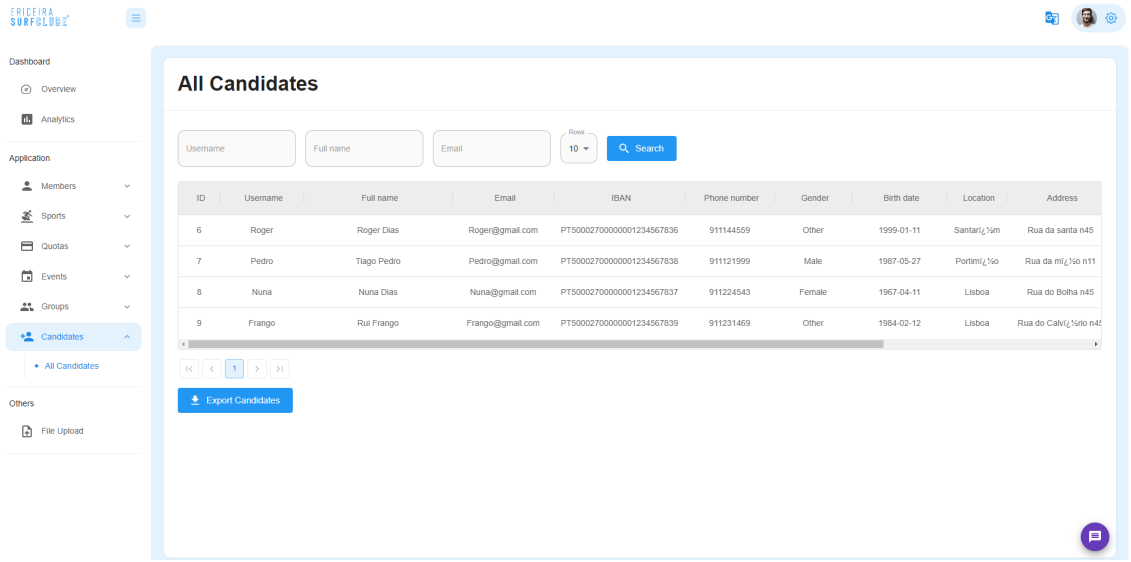

Figure B.11: Candidates view

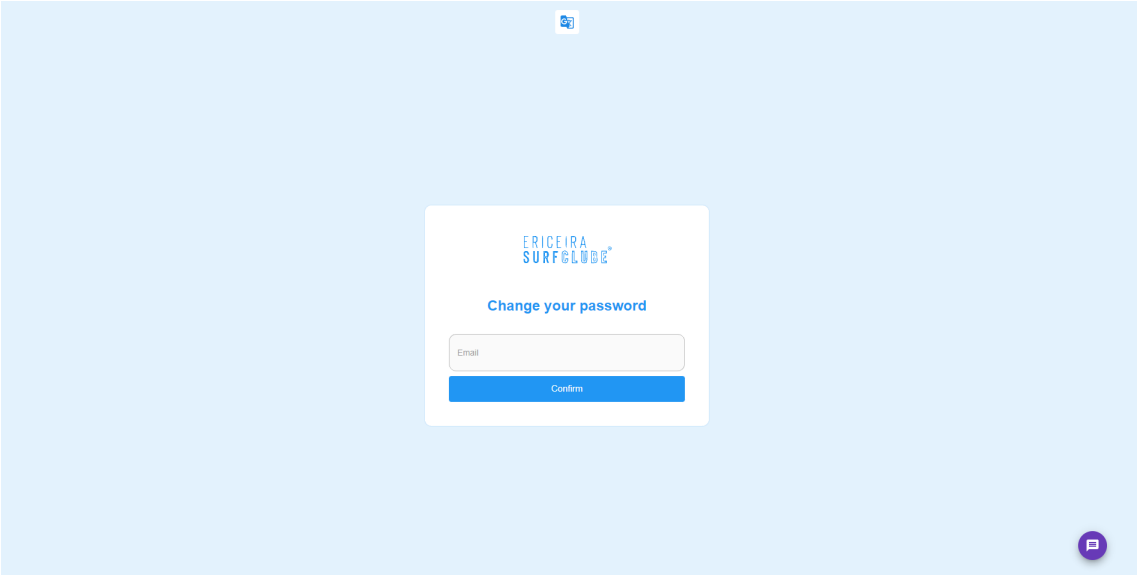

Figure B.12: Insert email to change password view

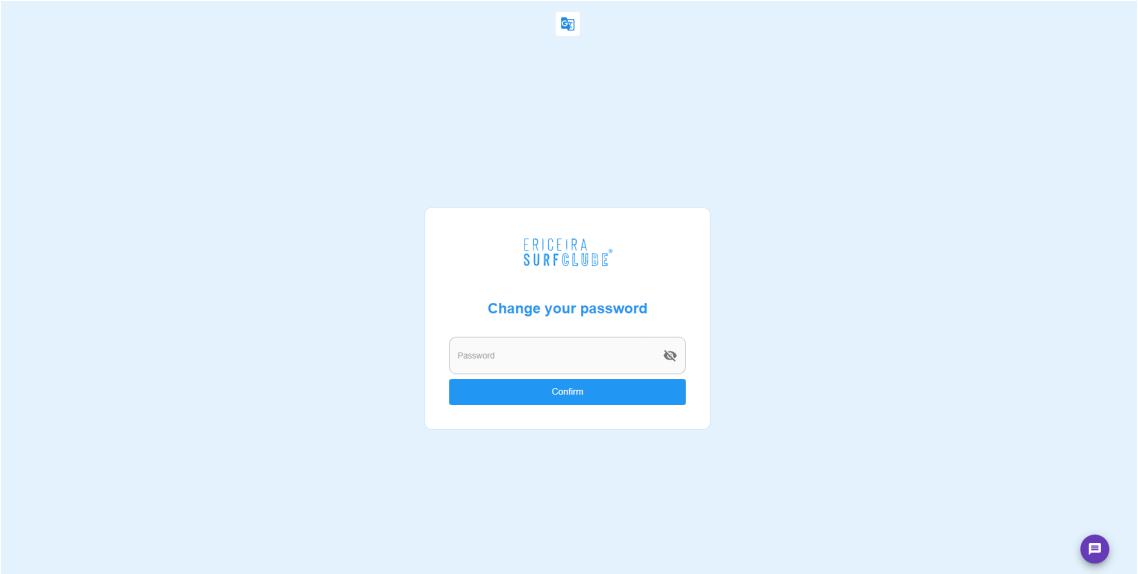

Figure B.13: Insert new password to change password view