
**Dipartimento di Informatica**

Anno accademico 2021-2022

# Battleship Game

Web and mobile app report based on the project developed for Tecnologie e Applicazioni Web [CT-0142]

**Cecconello Gabriele (870751)**

# Index

# 1. Introduction

This document describes the assigned project work in each of its aspects: realization, design choices, functionalities.

## 1.1 Project goals

The objective was creating a RESTful backend API and an Angular frontend Single Page Application to make users play Battleship.
Our goal for this project was not only to satisfy every mandatory requirement, but also to make the app esthetically pleasing and to add some extra functionalities.

This app is based both on the requirements and on our personal experience in other games.
An important aspect we focused on is performance optimization: we avoided flooding the server with HTTP requests or Socket.IO emits and preferred storing data more on the client side than the server side, in respect of the *stateless* paradigm.

## 1.2 Development tools and environment

From start to finish, this application has been developed using Microsoft Visual Studio as our IDE, integrated with GitHub for repository management. As for the database management, we used MongoDB's Atlas cloud platform, which provides us with a free 500MB database located in Frankfurt.

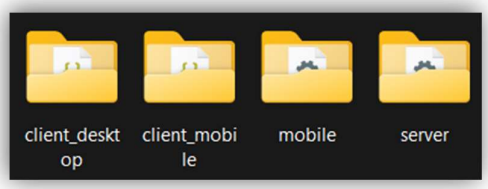
As required by the work assignment, the back-end webserver is running on a 16.16.0 Node.js deployment and uses Express.js for routing and MongoDB, a non-relational database, for storing data.
The front-end web app is based on the Angular framework and is running on a single page for the most part, and communicates with the back-end through HTTP requests or Socket.IO emits. We also used Tailwind to manage the app's styling.

Socket.IO is a library running in both front and back-end sides for low latency communication; we used Socket.IO to make clients communicate in real time: a client A emits a message to the server, which listens to that message and forwards it to client B, with B's identity being specified in the message itself.

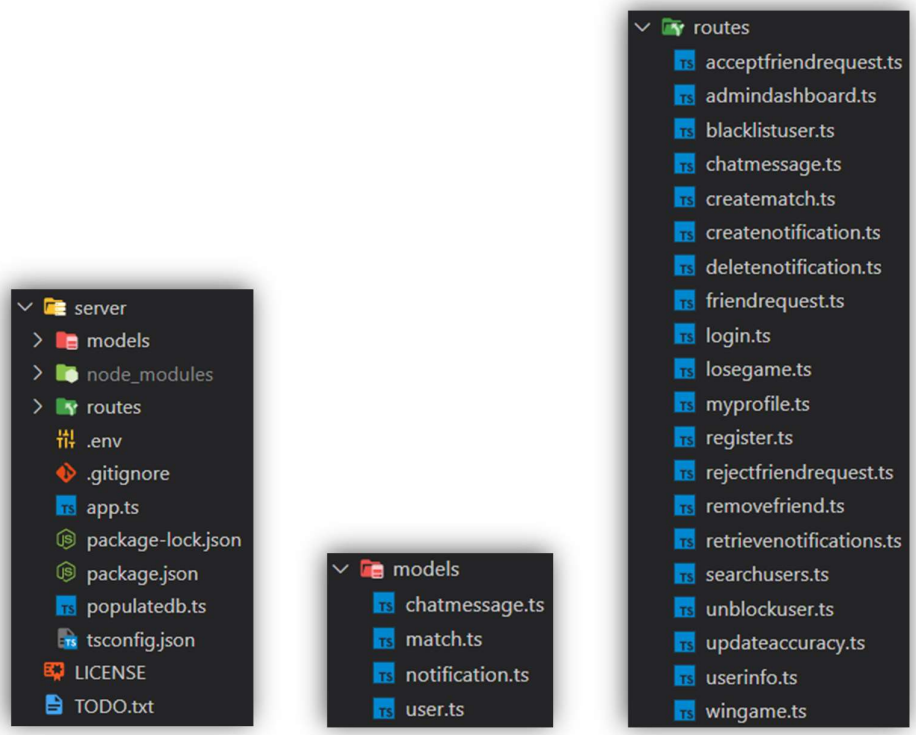
The desktop web app has been tested on Google Chrome, Microsoft Edge and Mozilla Firefox browsers in both Windows 10 and 11 environments.
As for the mobile app, we used Apache Cordova as required and we tested it on an emulator; the Android environment was set up with Android Studio (see **1.4** for the tutorial on starting the desktop and mobile app).

## 1.3 Project structure

As already mentioned, the project directory is divided into two main branches, the client side and the server side, with the client side being divided into three folders: "client_deskop", "client_mobile", "mobile":



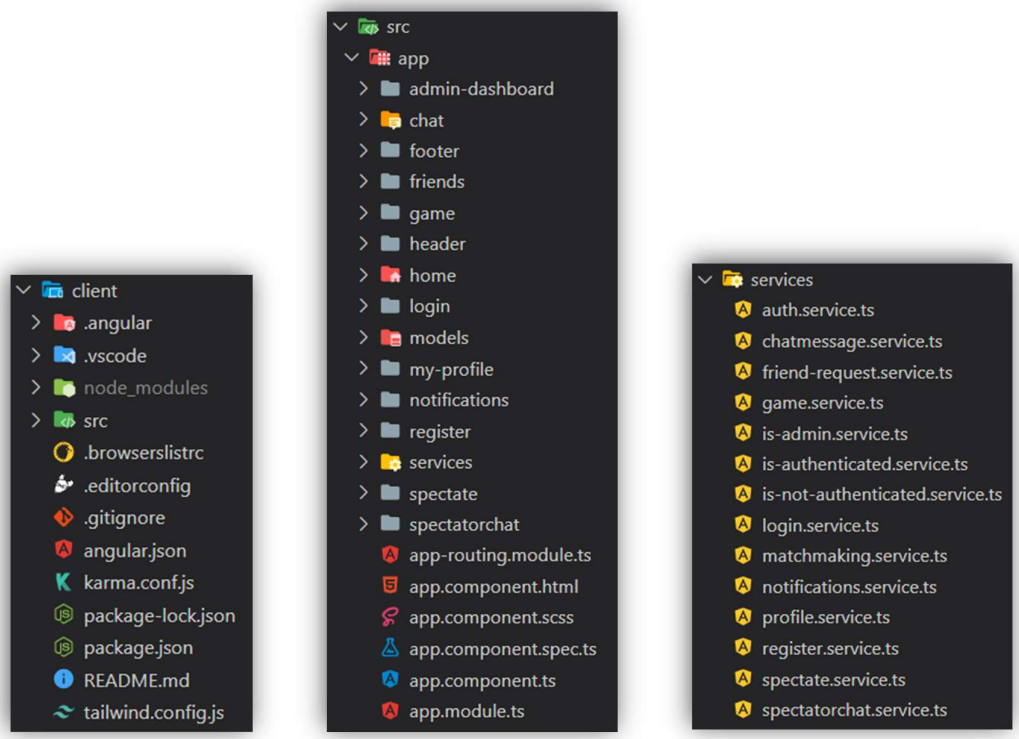

Below are the contents of the Server folder: the **‘‘app.ts’’** file is what will be compiled into JavaScript to be executed first and contains the MongoDB instance as well as the Socket.IO main server instance. It also hosts all the HTTP request policies, the body parser, the core Express instance and the matchmaking logic. The **‘‘models’’** folder contains all the MongoDB models, and the **‘‘routes’’** folder contains all the routes with their definition of GET and/or POST methods. The “populatedb.ts” file, if executed using **‘‘npm run populate’’** as explained in later chapters, resets the database’s data and populates it with dummy data.



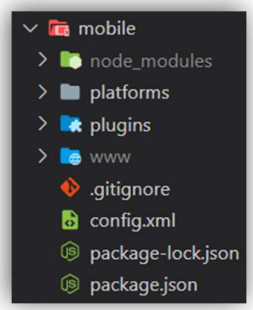

As for the client-side folders, the two “client” folders have the same structure, except the Angular App code in “client_mobile” is optimized for the mobile app. In fact, this folder has to be built into the “mobile” folder as will be explained later.

The “client_desktop” folder has a first layer (picture on the left below) containing all the libraries and packages, as well as the Tailwind style configuration file. In the second layer, which is the **‘‘src/app folder’’**, we can find all the components and services that build the front-end app. Every component, made of its TypeScript class and HTML part, is stored into a dedicated folder for code organization purposes. There is also a **‘‘services’’** folder, containing some TypeScript files that function as a bridge to the server; these services are injected into the constructor of the Angular components that make use of them and they are responsible for most HTTP requests and Socket emits. It’s rare for a component class to directly perform HTTP requests or Socket.IO emits (more details about this topic in chapter **2**).

This code separation between the Angular classes and the HTTP and Socket.IO requests into a “service” file helped us abstract and simplify the code in the Angular part.




Notice how even on the client's side there is a **''models''** folder; that folder is used to contain simple TypeScript classes that encapsulate data into JSON objects through their constructor. This way it's easier to send objects of certain types (Users, friend requests, etc.) to the server through HTTP requests.



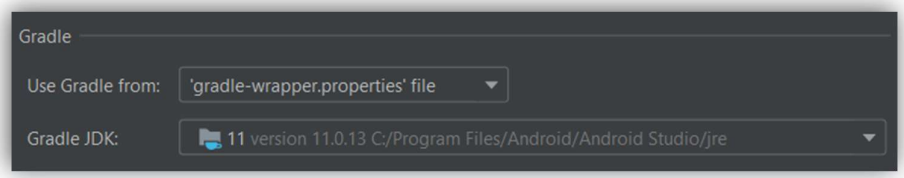

The "mobile" folder contains a "config.xml" file which dictates some general emulator rules, a "platforms" folder storing the mobile OS environments to run the app on (the default one being Android) and a "www" folder, generated whenever the "client_mobile" Angular folder is built into it.

# 1.4 Configuring and starting the app

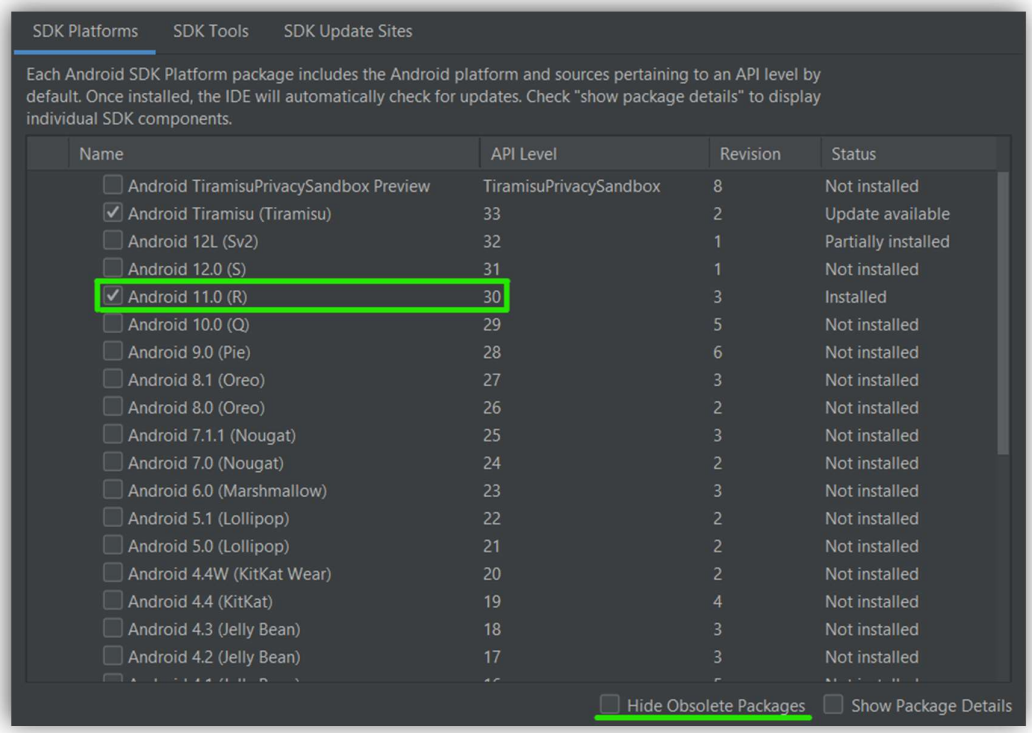
In this section I will guide the reader through the steps to start the server, the desktop application and the mobile application.

## 1.4.1 Installing the required software and packages

The first thing to install is Node.js version 16.16.0 from the official Node.js site.
Each folder contains a "package.json" and a "package-lock.json" file that determines the packages and libraries to be installed in order for the apps to run correctly. To install said packages, `cd` into each of the four top level folders ("server", "client_desktop", "client_mobile", "mobile") and run **''`npm i`''** in each one of them.
Android Studio and Gradle 7.4.2 need to be installed in order for the mobile app to work. Open Android Studio and go to: File\Settings\Build, Execution, Deployment\Build Tools\Gradle. From there, select the 11.0.13 JDK:




Then we need to select the correct SDK platform from Tools\Appearance and Behaviors\System Settings\Android SDK by choosing Android 11.0 (API Level 30). If that option is not showing, uncheck the "Hide Obsolete Packages" box.




## 1.4.2 Starting the app

To start the Node server, access the "server" folder via terminal and type **"`npm run populate`"** to fill the database with dummy data, then run **"`npm start`"** (or **"`npm run start`"**), which will start the actual Node.js app server on the localhost's port 3000.

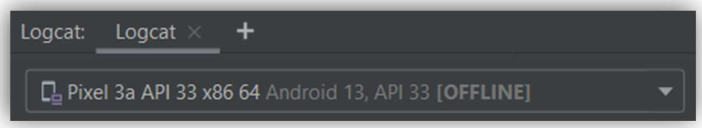
```
export const environment = {
  production: false,
  ip_address: '192.168.188.23'
};
```

To start the desktop app, there are two options: making the angular App visible or not visible in the local network.
To make it run only on the host device, go to "client_desktop/src/environments/environment.ts" and type "localhost" instead of the IP address (image on the left) in the **"`ip_address`"** field.
At that point, all that is left is to `cd` to the "client_desktop" folder and run **"`ng serve`"**.
To make the app visible in the local area network, substitute the device's actual IP address in that same field, then `cd` to the "client_desktop" folder and run **"`npm start`"** (or **"`npm run start`"**).
Now this angular app can be accessed from another LAN device by typing the host machine's IP address followed by the port number ":4200" (e.g., "http://192.168.1.10:4200").

In order to make the mobile app run in the emulator, aside from having Gradle 7.4.2 and Android Studio installed, there are a few more steps to take.
First, the ‘‘ip_address’’ field in “client_mobile/src/environments/environment.ts and environment.prod.ts” should be set to “10.0.2.2”, as it is the address that the emulator uses to reference the host machine.
Next, cd to the “client_mobile” folder and run ‘‘npm run build’’; this will create the “www” folder inside the top level “mobile” folder.
Enter the “www” folder and look for the “index.html” file; inside that file, make sure that the cordova.js script is imported correctly and that the base tag’s href is set to ‘‘./’’, like in the picture below:

```
<script type="text/javascript" src="cordova.js"></script>
<meta charset="utf-8">
<title>Battleship game</title>
<base href="./">
```

The config.xml file is already configured. After that, cd to the “mobile” folder and run ‘‘cordova build’’ (or ‘‘cordova build android’’; in our case ‘‘android’’ is omitted since it is the only platform we instantiated). After the app is built, run ‘‘cordova run’’ (or, again, ‘‘cordova run android’’); this will open the emulator after a few seconds and the mobile app will start automatically. Note that the ‘‘cordova run’’ command can be sent directly without running ‘‘cordova build’’ first, but it’s safer to run both commands in sequence.
To check the console or the logs for the Cordova app, open Android Studio and use the Logcat console after selecting the Android device from the drop down menu.




## 1.4.3 Using the app

Now that the server, the desktop app and the mobile app are running, they can communicate with each other. To open multiple user sessions on the same device, open an incognito page so that the cache isn’t shared with the normal browser pages. To open more than 2 sessions, another browser will be needed, since we noticed that incognito pages of the same browser share their cache data, meaning that each browser supports at most 2 separate sessions, unless it’s used with different profiles (like Chrome profiles or Edge profiles).

The application is obviously best optimized to be used in a full width window, but being responsive it can be used by splitting the screen into two pages to test the real time communication between the clients without having to continuously switch between pages.

The “npm run populate” command creates four users. Every ordinary user has “pass” as their password, while the administrator user has “Administrator” and “admin” as username and password (the username is case sensitive). Three ordinary users are created (Gabriele, Edoardo and Filippo are the usernames) and ready to be used and tested for playing, sending friend requests, sending match invites and spectating.

# 2. Client-Server interaction: HTTP and Socket.IO

In this section we will take a look at how the Server listens to and receives HTTP requests that point to its various routes, as well as how clients notify each other (and the server) about certain events through emitting and listening to messages thanks to the Socket.IO library.

## 2.1 HTTP server set-up

First, we need to create an Express routing instance and pass it to the ‘‘`createServer()`’’ function from the ‘‘`http`’’ library. Then we specify a port number and start listening through it.

```
import 'dotenv/config'
import {createServer} from 'http';
import express = require("express");
const app = express()
const server = createServer(app)
const port = 3000
```

We also need to enable CORS policies, with CORS standing for Cross Origin Resource Sharing, to allow certain header parameters and HTTP methods.

```
// Enabling CORS policies for the app as well
app.all('/*', (req, res, next) => {
  res.header('Access-Control-Allow-Origin', '*');
  res.header('Access-Control-Allow-Methods', 'GET, PUT, POST, DELETE, OPTIONS, PATCH, *');
  res.header('Access-Control-Allow-Headers', 'Content-Type, Authorization, Content-Length, X-Requested-With, cache-control');
  next();
})
```

A body parser is mandatory to be able to parse the content of the request body into a readable JSON:

```
import {json} from 'body-parser';
app.use(json());
```

Then we need to let the HTTP server know which routes to redirect the requests to, so we import the various routes from the route files (see **2.2**) and pass them to the Node.js app:

```
import loginRoute from './routes/login';
import winGameRoute from './routes/wingame';
import loseGameRoute from './routes/losegame';
import registerRoute from './routes/register';
```

→

```
app.use('/login', loginRoute);
app.use('/wingame', winGameRoute);
app.use('/losegame', loseGameRoute);
```

Finally, the Node.js app is completely set up and we can start listening on the previously specified port:

```
// Server starts listening on port 3000
server.listen(port)
```

# 2.2 Express Routes for HTTP Requests and MongoDB transactions

The ‘‘`app.ts`’’ file is not only responsible for starting the HTTP server, but also for connecting to the MongoDB database stored in the Atlas cloud. To do so, we require a database connection string from the ‘‘`.env`’’ file (in our case named ‘‘`DB_CONNECTION`’’) and pass it to the ‘‘`connect()`’’ function from the mongoose library:

```
import mongoose from 'mongoose';
const dbstring = process.env.DB_CONNECTION
if(dbstring != undefined){
  mongoose.connect(dbstring, (err) => {
      if(err){ console.log(err) }
    }
  );
}
```

Now that the server is connected to the database and knows where to redirect the HTTP requests, we’ll take a look at the routes.
Let’s take the ‘‘`register`’’ route as an example.

```
import * as express from 'express'
import User from '../models/user'
const router = express.Router()
```

```
export default router;
```

It’s essential that each route file imports ‘‘`express`’’ to create a Router, which will be configured with GET, POST, PUT, etc. methods and then exported, so that the main ‘‘`app.ts`’’ file can import said route as seen in **2.1**.
In this case, we also imported the User model since we are talking about the registration route which will have to insert a new User into the database. For more information about MongoDB models definition, consult chapter **3.4**.
After creating a Router, we can define its behavior in case of GET and POST methods by calling the `.get()` or `.post()` functions on the router:

```
// When a user registers, his information is saved in a new inserted db user
router.post('/', async (req, res) => {

  // Checking if a user with that username or email already exists
  var u = User.newUser(req.body);
  await User.findOne({$or:[{username: u.username}, {email: u.email}]}).then((result) => {
    if(result != null && result.username == u.username && result.email == u.email){ // Bot
```

From that function we will retrieve the user’s request and pass it to a lambda expression which will contain the route logic. Notice how the specified path passed to the ‘‘`post()`’’ function is ‘‘`/`’’ and not ‘‘`/register`’’, since the ‘‘`app.ts`’’ file already fills that up for us during the routes declaration shown in **2.1**:

```
import registerRoute from './routes/register';
```

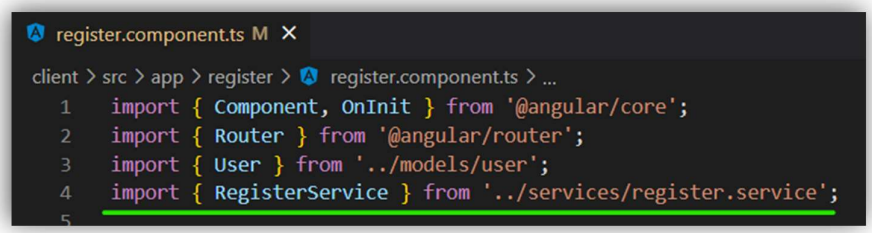
```
app.use('/register', registerRoute);
```

To send responses, we simply use the `.json()` function on the response parameter:

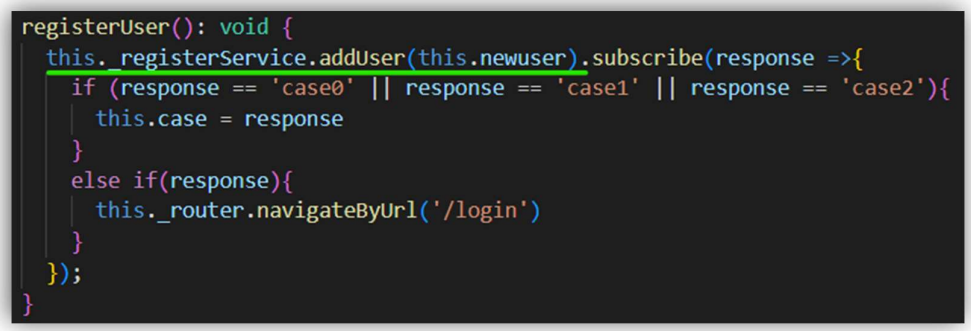
```
res.json(newUser)
```

# 2.3 Using HttpClient

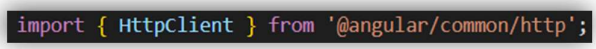
Now that the routes are set up to receive HTTP requests, elaborate on the request bodies and send a response, the only missing part is actually sending said HTTP requests from the client to the server.
To do so, the Angular components usually import a dedicated service class which will act as an interface between the components and the HTTP server or database.
Following the registration example, the "`register.component.ts`" file imports "`register.service.ts`":

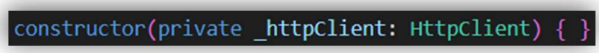



Then, when the user fills up the form fields with his/her e-mail address, username and password, the angular registration component invokes the "`.addUser()`" function from the registration service that has been imported:

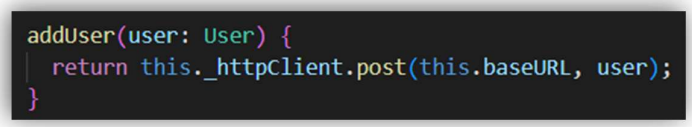



In the "`register.service.ts`" file, HttpClient is imported from Angular and is instantiated in the constructor:

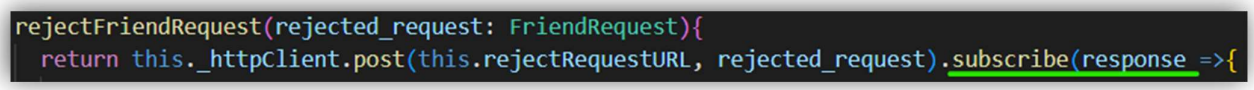



Then the "`.addUser()`" function uses HttpClient's "`.post()`" function and returns the response back to the component (picture below), which will retrieve it through the "`.subscribe()`" function.




The `.subscribe()` function can be invoked both in the component file or in the service file right after the HTTP request, like in this example from "`friend-request.service.ts`":



# 2.4 Socket.IO server set-up

Communication in real time between two clients or between a client and the server requires both sides to install their own Socket.io version. As already seen in section **2.1**, an Express server instance is created:

```
const app = express()
const server = createServer(app)
```

That instance will be used as a platform for a Socket.IO instance in the **''app.ts''** file to listen to incoming emitted messages:

```
import {Server} from 'socket.io';
const ios = new Server(server, {
  cors: {
    origin: "*",
    methods: ["GET", "POST"],
    allowedHeaders: ["Content-Type", "Authorization", "Content-Length", "X-Requested-With", "cache-control"]
  }
});
```

What's left is the longest part, which is defining what actions the server must perform whenever a message of a certain type has been listened by the Socket.io instance:

```
//Setting up Socket.io server side (ios stands for IO Server)
ios.on('connection', (socket) => {

  socket.on('startchat', (players) => {
    socket.emit('openchat', players)
  })

  socket.on('newmessage', (message) => {
    socket.broadcast.emit('youreceivedmessage'+message.to, message)
  })

  socket.on('confirmreception', (sender) => {
```

As can be seen in the picture above, a Socket.io message consists in a string that describes the message's topic and a message content. The message is detected through the `.on()` function and consists in a JSON containing various fields such as **''message_type''**, **''from''**, **''to''**, **''textcontent''**, etc. That message content acts similar to an HTTP request body.
The server then redirects such messages to the right clients, and those clients are specified in the message content JSON.

**Example: message between clients**

A client A wants to send a chat message to a client B, so A emits a "newmessage" string with attachments such as "textcontent", "from" and "to" fields. In this case, the "from" field is "A" and the "to" field is "B". The server detects that a "newmessage" is coming, but it still doesn't know who sent that message or who the recipient is. All the server knows is that it has to forward the message to the recipient indicated in the "to" field, so he broadcasts another message named "youreceivedmessageB".
Meanwhile, B's client will be listening to that specific "youreceivedmessageB" string, but further explanation for this is coming in the next section.

# 2.5 Using Socket.IO Client

For a client to be able to emit messages or listen to messages coming from the server, it needs to import the Socket.IO Client library (usually done in the service files). Then the service file needs to tell that Socket.IO instance to direct its messages to a specific URL, which is the localhost's address + port that the server is listening to:

```
import {io, Socket} from 'socket.io-client';
```

```
export class ChatmessageService {

  public baseURL = 'http://localhost:3000/'
  public socket: Socket

  constructor(private _httpClient: HttpClient) {
    this.socket = io(this.baseURL)
  }
```

From now on, that "io" object will listen to or send messages to that specified server URL. Each service using Socket.IO Client usually has a function that listens to incoming messages; that function returns an Observable object so that it can call the `.next()` function on the received message. That `.next()` function forwards said message to the Angular component who's using the service.

**Continuing example: message between clients**

The Angular Chat component immediately imports **"`chatmessage.service.ts`"** and invokes its `receiveMessages()` function (first picture below), then retrieves the messages through the `.subscribe()` function. That `.subscribe()` function works because `receiveMessages()` returns an Observable which keeps feeding the component with new messages (second picture below):

```
ngOnInit(): void {
  this._chatmessageservice.receiveMessages().subscribe((message) => {
    if(message.message_type == 'openchat'){ ···
    }
    else if(message.message_type == 'youreceivedmessage'){ ···
    }
    else if(message.message_type == 'yourmessagereceived'){ ···
    }
  })
}
```

```
receiveMessages(): Observable<any>{
  return new Observable((observer) => {

    this.socket.on('openchat', (message) => {
      observer.next(message)
    })

    this.socket.on('youreceivedmessage'+this.current_user.username, (message) => {
      message = { ···
      }
      observer.next(message);
    });

    this.socket.on('yourmessagereceived'+this.current_user.username, (message) => { ···
    })
```

Picking up from the previous example about A sending a message to B, we now know how B's client can listen to a new message: through the `receiveMessages()` function in the chatmessage service, it listens to a "youreceivedmessageB" string. As soon as that message is detected, the Observable object performs the `next()` function on that message, passing it to the component after adding a **''message_type''** field:

```
this.socket.on('youreceivedmessage'+this.current_user.username, (message) => {
  message = {
    from: message.from,
    to: message.to,
    text_content: message.text_content,
    message_type: 'youreceivedmessage'
  }
  observer.next(message);
});
```

Now the component receives a message and based on its **''message_type''**, which we just modified into **''youreceivedmessage''** since B actually received a message, performs different actions. In this case, it just adds the message to a local array of messages which will be displayed by the HTML part of the component:

```
else if(message.message_type == 'youreceivedmessage'){
  this.messages.push(message)
```

To send a message, the service just emits one with the appropriate fields:

```
sendMessage(chattype: String, newmessage: any){
  // Saving the message in the db only if it comes from a private chat and not a match chat, which won't be kept track of
  if(chattype == 'private'){
    this._httpClient.post(this.baseURL+'chatmessage', newmessage).subscribe()
  }
  this.socket.emit('newmessage', newmessage)
```

Then the server will detect it as explained in **2.4** and forward it to another client.

# 3. Functionalities and route endpoints overview

The first half of this chapter goes over the various features available in the app and explains at a high level, meaning that showing code won't be necessary, all the functionalities that the users can use.
For more in-depth explanations about how certain mechanics work on a technical level along with some code snippets, consult chapter **4**.
The second half of this chapter provides some insights about the MongoDB database and the routes.

## 3.1 Moderators and user management

As required by the assignment, we separated the users into two categories: ordinary users and moderators. Moderators are not supposed to play the game using their own account: they will need to create another ordinary account for that.
A moderator can monitor general statistics as well as manage users by banning them, promoting them to moderators, etc. When a moderator logs in, he's given two choices: view the statistics or manage all the users.

In the first case, the moderator is presented a list of all the users, except for moderators since they can't play the game and therefore have no statistics to keep track of, which he/she can order by whatever value is most useful. By default, users are ordered by username, but they can also be sorted by: wins, games played, win rate, accuracy, games lost, current win streak, longest win streak. Clicking on the sorting filter symbol once makes the sorting ascendent, clicking it once more makes it descendent, and so on.
If a moderator enters the user management section, then a list of all users (sorted by username) will appear; next to each entry of the list are some function buttons: ban, unban, promote to moderator, wipe statistics, notify, force password change, as well as a separate button for notifying all users.
Each of these buttons activate a confirmation pop-up to avoid accidental bans or statistics wipes.

The "ban" button temporarily bans the user, which will be notified the next time it tries to sign in.
The "unban" button unbans a banned user; the "promote" button elevates the user to an administrator role and forces it to change its password at its next logon. Promoting a user to moderator means not letting him play anymore unless it gets demoted to regular user again.
The "wipe statistics" button is self-explaining, as it zeroes every counter and every score of the selected user.
The "notify" button sends a notification to the user, whereas the "notify all" button sends a notification to all regular users and is thought for warning users about server maintenance, a new incoming update, or other general announcements.
The "force password changes" sets the `''needspasswordchange''` field in the User model to `true`, so that the next time the selected user tries to log in, he is forced to input a new password.

## 3.2 Regular users

In this section I will explain every single functionality dedicated to the final users, that is to say the regular users, those who actually play the game.

### 3.2.1 Friends

The Friends section can be accessed by clicking on the "group" icon on the top right of the navigation bar; the Friends section is made by two sub-sections: the friends list and the users and requests management section, where a user can look up other users by username, see friend requests notifications and unblock blocked users.

In the friends list section, users can decide to send a message to a friend (this will open a chat box), invite a friend to play a game (**3.2.2**) or remove a friend from its friends list.

In the users and requests management section, users can use the search box to look up other users by typing their username or just part of it: the five most similar usernames will be shown. Once a user finds another user through that search box, it can send a friend request or block that user.

If user A sends a friend request to user B and B's friend section is already opened, A's request will show up immediately. If not, a red notification badge will appear on the "group" icon.
Now user B has three possibilities: reject A's request (A will not be notified), block user A (the request is automatically rejected, and A will not be notified) or accept A's request (A will see B's username appear in its friends list).
If B already blocked A, A will still be able to look up B's user and send a friend request, but that friend request gets automatically canceled; this way A can spam friend requests as much as it wants, but these will never reach B. The same happens if A and B are already friends or if A has already sent B a friend request.
A user can block another user anytime: it just needs to look up the other user's username, find the user and block him by clicking on the designated button. If user A blocks user B and A and B are friends, the users are unfriended.

### 3.2.2 Skill Based Matchmaking system and match invites

There are two ways to match up users for a match: either they send each other a match invite, or they both press the "ready up" button, which puts them in a waiting queue.

When user A sends a match invite to user B, B is immediately notified and by opening the notifications section he can view the invite and decide to accept or reject it. If B accepts A's request, A needs to send another acknowledgment message to B, indicating that A not in a game already or that A is not in the waiting queue.
If A is not available to play at that moment, B will see a message appear for a couple seconds saying "A is no longer available for a match". In case A is available when B accepts A's request, both users are loaded up into a game session.

When a user clicks on the "ready up" button, its skill level is calculated on the spot through the following formula:

$$(current\_winstreak \times 10) + win\_rate + accuracy \quad ^{*}$$

Once the skill level is calculated, a Socket.IO emit is sent to the server with the user's username, its skill score and the precise date and time it clicked on "ready up", so the server can store the users in the waiting queue. Every 5 seconds, all the users in ready-to-play state that have been waiting for at least 5 seconds are sorted by ascending skill level order and are paired up into a game.
For each pair of users, a Socket.IO emit notification is sent with all the information the two clients need to load up the game.
Users that have not been waiting for long enough, that is at least 5 seconds, must wait in queue for some longer.

*About the skill level calculation formula: a user's maximum skill level is potentially infinite, since it's 200 (100% win rate + 100% accuracy) plus the current win streak multiplied by 10. We decided to include the win streak as a parameter for the skill level formula to allow the value to fluctuate more.
With time, except for a few masterful players, the skill level based on just win rate and accuracy is bound to stall in an average between 80 and 120 (from 40% win rate + 40% accuracy to 60% win rate +60% accuracy); by adding the current win streak to the equation and multiplying it by 10, we avoid said stalling.

A more effective way of calculating the skill level would be that of considering two more parameters: the amount of time elapsed from the last match (if the user has not played in more than a week then it's expected to be rusty at the game) and the last value of the current win streak before it plummeted to zero after a loss (a player who won 20 games in a row and just lost a game should not be matched up with an average/much less skilled player with a low win streak value).
Then again, this is a Battleship game and skill matters almost as much as luck does.

### 3.2.3 Game mechanics

Once the two players are matched up, the navigation bar buttons, the profile section and all the opened chats disappear. The game is now in what we defined as the Positioning phase.

The interface shows a left pane with the ships that the player has to place; clicking on one of those ships selects it and hovering the pointer on the map on the right shows the cells where the ship would be places. The cells are colored green if the ship can be placed, red if they can't (we allowed touching corners). Clicking on the field when a ship is selected places that ship in the coordinates where the mouse is hovering. Clicking a placed ship removes it and puts it back in the list of ships to be placed.
To rotate a ship from vertical to horizontal, the user must use the scroll wheel (in the mobile version we included a rotation button).
The user can also randomly place all the ships by clicking on the "random placement" button as many times as needed; there is also a "reset placement" button that removes all the placed ships from the field.
The player must place all the ships, and only then he/she can confirm the placement; once the placement is confirmed, it cannot be modified. If both players confirm the placement, the Playing phase starts.

The enemy field appears on the right as well as the collapsible chat box (players can't communicate during the positioning phase). When it's the user's turn, the enemy field lights up and becomes clickable; a 15

seconds timer also shows up and becomes red when 5 seconds are left. If the timer goes to zero, the player loses the match.
When the user misses a shot, the enemy's cell shows a black dot; if the user hits, the cell shows an orange X and if the ship is sunk, all the cells show red Xs with black dots around the ship. If the enemy hits or sinks one of the user's ships, the cell shows a red X.
If the user hits a ship, the 15 seconds timer is reset as his/her turn restarts, but in case of a missed shot, the turn is over.

If a player leaves the match during the Positioning phase or the Playing phase (by pressing the "leave match" button, logging out, reloading the page or closing the browser), the other players wins.

When the match is over, both players can decide to ask for a rematch or leave the game session; in this case, leaving doesn't make anyone lose since the game is already over. After one of the two player leaves, the other player is notified by a message and the game interface is unloaded after 4 seconds.

All the players keep a list of other users they recently played with, which is displayed in the home page on the bottom-right corner; if a user from that list is not friends with the player, a friend request can be sent, whereas if it is already friends with the player, a match invite can be sent.

For in-depth details about how the game works, consult **4.1**.

### 3.2.4 In-game and private chat

There are two types of chat: in-game chat and private chat. The latter refers to a chat between two friends, whose messages are stored in the database and are reloaded each time one of the two user opens up the chat.
The in-game chat is used in the Game component and in the Spectators component; this chat doesn't load messages from or to the database, as games can be spectated by other users. Even if the two players are friends, the messages are not saved in the database and exist only within the match's scope. Once the match is over, the messages disappear.
This way we avoid spectators seeing private messages between the two players that may be loaded once the game starts.

If user A sends a message to user B and B has the chat opened in front of him, B sends a "message read" acknowledgement to A, so A knows not to notify B of an unread message. If B's chat with A is not open, then a red notification badge appears and by expanding the notifications section, B can view said message.

To avoid sending multiple notifications, if A already sent a message to B and B has already a notification from A regarding their chat, no new "unread message" notifications are sent from A to B.

### 3.2.5 Spectating a game

A user can find a list of ongoing matches in the home page; every 10 seconds, the server sends a new list to all the clients.
After 150 seconds, a match is removed from that list, since after that much time the game should be already in the end phase. A match is removed from the list of ongoing matches also when both players

leave after the game ended. If the two players agree to a rematch, the game stays in the list for another 150 seconds and the spectator doesn't have to re-join the session.

Spectators can read other spectators' messages as well as what the two players are texting each other, whereas the two players can't see what the spectators are texting in the chat.
Whenever a player positions a ship, removes a ship, resets the positioning or places the ships randomly, the spectators are notified with the new field status and load it into their interface. When the players start firing shots at each other, their ships are no longer visible to the spectators, who can still see when players miss, hit or sink.

As the game ends, a message is displayed in the spectator's interface showing the match result. When the two players leave the match, the spectators are not forced to leave even if the match is removed from the ongoing matches list (meaning that other spectators can't join in anymore), so they can keep chatting.
A spectator is free to leave at whatever stage of the game by pressing the "stop spectating" button: players will not be notified about a spectator joining or leaving the session.

## 3.3 Statistics

Each regular user can keep track of its statistics by looking at their profile section in the home page; there they will be able to see their accuracy percentage, their win rate, their total games played, their won and lost games, their current win streak and their highest win streak.

While the accuracy is updated whenever the player takes a shot at another player, all the other values are updated after the user ends a match, let it be by winning*, losing or leaving.

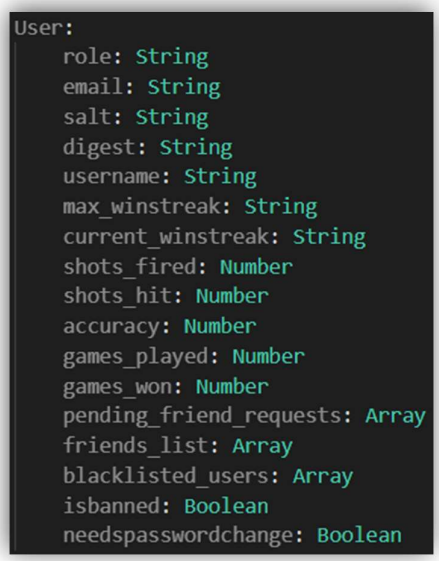
*Issue: we unfortunately encountered an issue we couldn't solve. The function `winGame()` is responsible for making a `POST` request to the `''/wingame''` route, which should update the user's win streak and `''games_won''` counter.
Such function is always invoked without parameters and uses data stored in the client's browser's `LocalStorage`. Such data (enemy's username, match start time, current user's username) does not change at any point in the execution of the code for the Game component.

What happens is, if the `winGame()` function is called after the enemy leaves the match or if the enemy times out on its turn, the `HTTP POST` request is successfully executed. But if the same exact function is called once one of the two player sinks the enemy's last remaining ship, the `winGame()` function is still invoked correctly but the `HTTP POST` request is not executed at all.
Every debugging method showed no information about the line of code where the `POST` request is made: it is almost like it gets skipped.
We tried printing something before and after the `POST` request to check whether the function was called in the first place and those console logs worked, but that specific line of code where the `POST` request is made doesn't seem to be executed. We even printed the data from the LocalStorage right before the POST request line and it seemed to be correct, but, again, just that POST request line was not executed.
The consoles didn't show any error.

We are sorry to confirm that, for the reasons explained above which we think are not in our scope of operation, the statistics are not correctly updated after the user wins a game by sinking the last ship.

## 3.4 MongoDB models

We created four models (or *schemas*) upon which MongoDB would create its documents:


For more information about the use of "digest" and "salt" fields for authenticatoin, please consult chapter **5**.

## 3.5 Routes and endpoints overview

The following table contains all the routes in alphabetical order. The "requester" is the user sending the HTTP requests.

| Endpoint | Method | Authentication required | Action |
|---|---|---|---|
| /**acceptfriendrequest** | POST | ordinary | When user A accepts user B's friend request, both users' friends list is updated as well as their pending friend requests. |
| /**admindashboard** | POST | moderator | Depending on the "request_type" field specified in the request body, this route contains all the logic behind the user management tools available for the moderators. |
| /**blacklistuser** | POST | ordinary | Updates the requester's "blacklisted_users" list by adding the blocked user's username in it. |
| /**chatmessage** | POST | ordinary | Inserts a new message in the database. |
| /**chatmessage** | PUT | ordinary | Retrieves the last N messages between the requester and the friend in the opened chat. |
| /**creatematch** | POST | ordinary | Inserts a new match in the database and increases the "games_played" field of the two players involved in the match. |
| /**createnotification** | POST | ordinary /mod. | Inserts a new notification in the database. |
| /**deletenotification** | POST | ordinary | Deletes a notification when the user reads it or simply interacts with it. |
| /**friendrequest** | POST | ordinary | Sends a friend request from the requester (user A) to user B. If user A is blacklisted by B, is |

| | | | |
|---|---|---|---|
| | | | already friends with B or has already sent a friend request to B, the request is not sent. |
| /**login** | GET | none | Authenticates the user. |
| /**login** | POST | ordinary /mod. | Changes a user's password. |
| /**losegame** | POST | ordinary | Takes the "current_winstreak" field of the loser back to 0.<br>This route is reachable both by the loser and the winner in case the loser disconnects to avoid sending the request. |
| /**myprofile** | POST | ordinary | Returns the requester's user information except for the digest, salt and password. |
| /**register** | POST | none | Inserts a new user into the database. |
| /**rejectfriendrequest** | POST | ordinary | Deletes the friend request from the requester's "pending_friend_requests" list. |
| /**removefriend** | POST | ordinary | Deletes the desired friend from the requester's friends list. |
| /**retrievenotifications** | POST | ordinary | Returns all the unread notifications to the requester. |
| /**searchusers** | POST | ordinary /mod. | Returns the five users with the most similar usernames to the string typed by the requester. |
| /**unblockuser** | POST | ordinary | Removes a user from the requester's blacklist. |
| /**userinfo** | POST | ordinary | Returns basic statistics about a specific user: accuracy, current winstreak, winrate. |
| /**updateaccuracy** | POST | ordinary | Increases the requester's total shots counter and shots hit counter. |
| /**wingame** | POST | ordinary | Sets the "winner" field of the desired match with the name of the requester and increases the requester's current win streak (and max win streak if it's the case). Also increases the requester's won games counter. |

# 4. Code snippets and explanation

In this section I will try to give a general overview of how the components intertwine and collaborate in building the app. Most of the code is thoroughly commented, but some links between components are not clear at first glance and some components use multiple services, making the code not intuitive.

## 4.1 Home Component

The Home component is the main component of the application since it contains almost all of the other components. Header, Footer, Login and Registration components are standalone but still contained in the same main App Component.

Based on the state of the app and what the user is doing, the Home component shows the Game, Chat, Profile, Spectator interface; these components are imported into Home's HTML code and are shown under certain conditions, meaning only if a certain combination of Boolean values is satisfied.

```
<div *ngIf=" !isadmin ">

  <div class="game-container w-50">
    <div *ngIf="isplaying">
      <app-game></app-game>
    </div>

    <div *ngIf="!isplaying">
      <div *ngIf="!isspectating">
        <div class="grid grid-cols-2 gap-x-10 max-w-[480px] lg:max-w-[1024px] mx-auto">
          <ul *ngFor="let match of ongoing_matches">···
          </ul>
          <app-my-profile></app-my-profile>
        </div>

        <div class="text-center my-8">···
        </div>
      </div>

      <div *ngIf="isspectating">
        <app-spectate></app-spectate>
        <div class="text-center pt-4">···
        </div>
      </div>

      <app-chat></app-chat>

    </div>
  </div>
</div>


<!-- ADMINISTRATOR CONTROL PANEL -->
<div *ngIf="isadmin">
  <app-admin-dashboard></app-admin-dashboard>
</div>
```

As seen in the picture above, based on the role of the user, the Home Component shows either the moderator dashboard (called "`admin-dashboard`") or the rest of the components. If the regular user is

playing, then he/she will see the game component, otherwise he/she will see the spectator component (if spectating), the profile component, the list of ongoing matches, and so on.

Using the Angular tags ‘‘`<app-game></app-game>`’’, the app imports a whole instance of Game component, as is for any other component declared using these tags.

The Home component utilizes the matchmaking service:

```
import { MatchmakingService } from '../services/matchmaking.service';
```

And right when the component loads, it immediately invokes the `listenToMatchmaking()` function:

```
listenToMatchmaking(){
  this._matchMakingService.listenToMatchmaking(this.current_user).subscribe((observable) => {
    // When the user is matched up in a game, the game component loads up
    if(observable.message_type == 'yougotmatched'){ ···
    }
    // Unloads the game once the match is over
    else if(observable.message_type == 'matchended'){ ···
    }
    // Fetches the update list of ongoing matches
    else if(observable.message_type == 'newongoingmatches'){ ···
    }
    // If a user accepted our invite to play
    else if(observable.message_type == 'matchinviteaccepted'){ ···
    }
  })
}
```

Home Component’s ‘‘`listenToMatchMaking()`’’ function calls the Matchmaking Service’s ‘‘`listenToMatchMaking()`’’ function, and its only use is that of forwarding to the Home Component all the messages coming from the Server’s Socket.IO instance:

```
listenToMatchmaking(current_user: any): Observable <any>{
  return new Observable((observer) => {
    this.socket.on('matchstarted'+current_user.username, (matchinfo) => {
      observer.next(matchinfo)
    })

    this.socket.on('matchended'+current_user.username, (message) => {
      observer.next(message)
    })

    this.socket.on('newongoingmatches', (message) => {
      observer.next(message)
    })

    this.socket.on('matchinviteaccepted'+current_user.username, (message) => {
      observer.next(message)
    })
  })
}
```

As can be seen in the pictures above, the Home Component is responsible for listening to matchmaking events, notifications about match invites being accepted or matches being ended and loading new ongoing matches in the homonymous list. The Matchmaking Service has many other functions:

```
readyUp(current_user: any) {
  this.socket.emit('readytoplay', current_user)
}

cancelMatchMaking(current_user: any) {
  this.socket.emit('cancelmatchmaking', current_user)
}

createMatch(match_info: any){
  this._httpClient.post(this.baseURL+'creatematch', match_info).subscribe()
  this.socket.emit('matchcreated', match_info)
}

availableForMatch(current_user: String, accepting_user: String, starttime: Date){
  this.socket.emit('availableformatch', {
    user: accepting_user,
    from: current_user,
    starttime: starttime})
}

notAvailableForMatch(current_user: String, accepting_user: String){
  this.socket.emit('notavailableformatch', { user: accepting_user, from: current_user})
}

closeMenus(current_user: String){
  this.socket.emit('closeheadermenus', { user: current_user, notification_type: 'closeheadermenus' })
}

openMenus(current_user: String){
  this.socket.emit('openheadermenus', { user: current_user, notification_type: 'openheadermenus' })
}
```

The `readyUp()` function notifies the server's Socket.IO instance that the user is ready to play a game and wants to be inserted into the waiting queue, whereas the `cancelMatchMaking()` function does the contrary. `availableForRematch()` and `notAvailableForRematch()` notify the server's Socket.IO that the user is available (or not available) after another user accepted its match invite. The server will then notify the other user. Finally, `closeMenus()` and `openMenus()` tell the header component to compress all the drop-down menus and make them disappear or to make them reappear. `closeMenus()` is usually called when a game starts, and `openMenus()` when a game ends.

# 4.2 Game Component

Developed in 1200 TS and HTML lines of code, the Game Component is probably the most articulated one, since it has to provide ship placement and rotation mechanics, shooting and sinking ships, shot results, the turn and timeout system, the rematch functions and the notifications to the spectators, as well as the control over a player disconnecting by leaving the match, logging out, closing the browser or reloading the page.

The Game Component is separated in three main sections: Positioning section, Playing section and After-Match section. Right as the Component loads up, the app starts listening to various events:

```
// If the user quits the game component, he loses the game
@HostListener('window: beforeunload', ['$event'])
unloadHandler(event: Event) {
  if(!this.youwon){this.leaveMatch('enemyleftwhileplaying')}
}

ngOnInit(): void {
  // Immediately start listening to various game-related events such as
  // "enemyleft", "yougotshot", "enemyconfirmedpositioning", etc.
  this.startGame()
}

// If the user quits the game component, he loses the game
ngOnDestroy(){
  if(!this.youwon){this.leaveMatch('enemyleftwhileplaying')}
}
```

The ‘‘`startGame()`’’ function will be explained later.

### Positioning Phase

Most functions in the positioning phase are self-explanatory thanks to the comments, but it’s worth explaining the variables in the various cells of the player and enemy’s field.
Each cell of the ‘‘`myfield`’’ matrix contains the following values by default:

```
value: 0,
orientation: '',
hit: false,
preview_success: 'none'
```

‘‘`value`’’ goes from 0 to 5 and indicates the length of the ship placed on that cell. 0 means no ship is placed there, $2 \leq N \leq 5$ means that the cell contains a portion of a ship of length $N$.
‘‘`orientation`’’ can be h or v, meaning horizontal or vertical. ‘‘`hit`’’ indicates if the ship in that position has been hit or not and ‘‘`preview_success`’’ has three String values: ‘‘`none`’’, ‘‘`true`’’, ‘‘`false`’’. ‘‘`None`’’ makes that cell colored white, ‘‘`true`’’ makes it green (meaning that the ship is placeable in that cell) and ‘‘`false`’’ makes it red, indicating that the ship cannot be placed there.

Each cell of the ‘‘`myships`’’ array contains the following values by default:

```
shiplength: Number /* 2, 3, 4, 5 */,
orientation: '',
sunk: false
```

‘‘`shiplength`’’ and “orientation” indicates the length and orientation of the ship, while “sunk” indicates if the ship has been sunk or not. Once all of the ships in ‘‘`myships`’’ are sunk, the user loses the game.

Moving on with ‘‘`placedShips`’’ and ‘‘`enemyShips`’’:

```
this.placedShips = new Array()
this.placedShips.push({ shiplength: 5, isselected: false, remaining: 1, orientation: ''})
this.placedShips.push({ shiplength: 4, isselected: false, remaining: 2, orientation: ''})
this.placedShips.push({ shiplength: 3, isselected: false, remaining: 3, orientation: ''})
this.placedShips.push({ shiplength: 2, isselected: false, remaining: 5, orientation: ''})
this.enemyShips = new Array()
this.enemyShips.push({ shiplength: 5, remaining: 1})
this.enemyShips.push({ shiplength: 4, remaining: 2})
this.enemyShips.push({ shiplength: 3, remaining: 3})
this.enemyShips.push({ shiplength: 2, remaining: 5})
```

The ‘‘placesShips’’ array is used when positioning the ships, while the ‘‘enemyShips’’ array is used to show how many enemy ships are remaining during the playing phase.

Finally, the ‘‘enemyField’’ matrix is initialized as follows:

```
initEnemyField(){
  this.enemyfield = new Array()
  for(let i = 0; i < 10; i++){
    this.enemyfield[i] = new Array()
    for(let j = 0; j < 10; j++){
      this.enemyfield[i][j] = '?'
    }
  }
}
```

A cell of this matrix can contain different values: ‘‘?’’ meaning we don’t know what is in that position, ‘‘water’’ meaning the user fired a shot but missed, ‘‘hit’’ if that cell is part of a ship the user fired at, ‘‘sunk’’ if that cell is part of a sunken ship.

The user’s client never stores the enemy field’s exact ship coordinates; to decide whether the ship is hit or not, the user must wait for the other player to send a ‘‘shot result’’ as explained soon.
Once the user locks in the ship positioning by sending a Socket.IO emit to the server, the ships are no longer placeable or removable and the startChat() from the ChatMessage Service is invoked, specifying that it is a “match” chat and not a “private” chat (consult **3.2.4** for more details about this).

**Playing phase**
When both players confirm their ship placements, the game can begin. The first turn is randomly selected each time, and the player that has to wait for its turn starts immediately a 15 seconds timer through this function:

```
waitForEnemyActivity(){
  this.detectedenemyactivity = false
  this.timeout = setTimeout(() => {
    if(!this.detectedenemyactivity){
      this.enemytimedout = true
      this._gameService.notifyEnemyTimeout(this.enemy)
      this._gameService.loseGameDB(this.enemy)
      this.winGame()
    }
  }, 15000)
```

Thanks to this timer, the user is able to wait for 15 seconds and if the enemy doesn’t show any type of activity, the player wins by default.
This function is used not only to simply set a turn timer, but it works even in the case of an AFK or disconnected player: if the enemy disconnects and for some reason that event is not caught immediately, the player wins after 15 seconds of enemy inactivity.
This function is called every time a user expects a response from the enemy; for example, as soon as the user fires at the enemy, this timer starts again and stops only when the enemy sends the shot result.
Whenever the enemy shows some kind of activity, the ‘‘this.detectedenemyactivity’’ variable is set to `true`.

As with the functions in the Positioning section, the ones in the Playing section (`fire()`, `sinkShip()`, `autoFillWater()`, `isSunk()`, `youLost()`, `youWon()`) are intuitive and explained by the comments.

The core function of this component is the startGame() function, which is invoked right as the component gets loaded. Said function is responsible for listening to any kind of game-related events, such as an enemy shooting at us or sending the result of a shot we fired at them, the enemy leaving or wanting a rematch, and so on.

The Game Component relies on the Game Service file, whose duty is that of communicating all the actions to the server which will then notify the enemy, and vice versa.

Here is the startGame() function with its sections compressed:

```
startGame(){
  // Start listening to all game-related events
  this._gameService.startGame(this.current_user.username, this.enemy).subscribe((message: any) => {
    // If the enemy confirms his ship positioning, we can start playing
    if(message.message_type == 'enemyconfirmed'){···
    }
    // If the enemy fires a shot in our field, we prepare the result that
    // is to be given back to him with the shot results
    else if(message.message_type == 'yougotshot'){···
    }
    // If we shot and receive the result from the enemy we update our
    // enemyfield with the appropriate symbols
    else if(message.message_type == 'shotresult'){···
    }
    // If the nemy leaves while we're in the playing phase
    else if(message.message_type == 'enemyleftwhileplaying'){···
    }
    // If the nemy leaves during the positioning phase
    else if(message.message_type == 'enemyleftwhilepositioning'){···
    }
    // If the enemy leaves after he won and we were waiting to see
    // if he wanted a rematch
    else if(message.message_type == 'enemyleftaftermatchended'){···
    }
    // If the enemy wants a rematch after the game is finished
    else if(message.message_type == 'requestrematch'){···
    }
    // If the enemy accepted our rematch request after the game is finished
    else if(message.message_type == 'acceptrematch'){···
    }
    // If we didn't make a move in time
    else if(message.message_type == 'youtimedout'){···
    }
    // When a new spectator joins the match, the players must send him their
    // field (if in the positioning phase) or the enemy's field (if in the playing phase)
    else if(message.message_type == 'imspectatingyou'){···
    }
  })
}
```

Almost every “if” or “else if” branch has its own sub-conditional branches. For example, in case of a ‘‘`shotresult`’’ message, there can be many cases which lead to different actions: the player missed, the player hit a ship, the player hit a ship and sank it, the player hit a ship, sank it and wins the game.
Each detail of these branches is explained by the comments on the source code which I won’t report here to keep this document short.

**After-Match section**

After the match ends, the players are free to leave without any consequences, or they can decide on a rematch. The first player to ask for a rematch makes an “accept rematch” option appear on the other player’s screen. When the other player agrees to the rematch, the game is taken to its initial state and the function ‘‘`prepareForRematch()`’’ is called:

```
prepareForRematch(){
  clearTimeout(this.timeout)

  this.myturn = false
  this.gamestarted = false
  this.isplaying = false
  this.hasconfirmedpositioning = false
  this.youwon = false
  this.youlost = false
  this.enemyleft = false
  this.enemywantsrematch = false
  this.enemyfield = new Array()
  this.stopTimer()
  this.initMyShips()
  this.resetPlacement()
}
```

As seen in this picture, every Boolean game-control variable is set to false in order to take the game to its original state, meaning that now the players are in the positioning phase again.

Meanwhile, the ‘‘acceptRematch’’ function makes the accepting user create another match in the database and tells the other player to update the ‘‘starttime’’ filed in its LocalStorage, since this match is considered as a different match from the previous one.
Updating the ‘‘starttime’’ value of the match in the LocalStorage is very important as it is one of the identifying values of a match in the database as well as one of the parameters passed to the POST request when the player wins a game and needs to update the match’s ‘‘winner’’ field in the database.

If a player decides to leave during the positioning phase or the playing phase, he/she loses the game and notifies the other player of his victory. Leaving after the match ended simply unloads the game component.

## 4.3 Chat Component

The chat component serves as a perfect example of an interaction between two clients through the Socket.IO library. This Component is almost always loaded into the application, but it’s only shown when required; to make the chat actually appear, the component must receive an ‘‘openchat’’ emit containing who the other user in the chat is and the chat type (in-game chat or private chat):

```
ngOnInit(): void {
  this._chatmessageservice.receiveMessages().subscribe((message) => {
    // When another component tells us to open a chat between us and another user
    // (let it be a friend or an opponent during a random match)
    if(message.message_type == 'openchat'){
      this.player1 = message.current_user
      this.player2 = message.other_user
      this.chattype = message.chattype
      if(message.chat_type == 'private'){
        this.getMessages()
      }
      this.chatopened = true
    }
```

We can see that the ‘‘`getMessages()`’’ function, which queries the last messages between the two users, is invoked only if the chat is private.
To send a message, we use the ‘‘`sendMessage()`’’ function in the ‘‘`chat-message.service.ts`’’ file. If the chat type is private, the message is inserted into the database with an HTTP request to the ‘‘`/chatmessage`’’ route. It the players are chatting in-game, a ‘‘`newplayersmessage`’’ is emitted so that the spectators can see it.

```
sendMessage(chattype: String, newmessage: any){
  if(chattype == 'private'){
    this._httpClient.post(this.baseURL+'chatmessage', newmessage).subscribe()
  }
  this.socket.emit('newmessage', newmessage)

  if(chattype == 'match'){
    this.socket.emit('newplayermessage', newmessage)
  }
}
```

Then the service emits a “newmessage” which the server catches and redirects to the other player by adding the recipient’s username in the broadcasted emit’s title string:

```
socket.on('newmessage', (message) => {
  socket.broadcast.emit('youreceivedmessage'+message.to, message)
})
```

This way the emitted message is personalized and only the correct user can receive it, since it listens to a ‘‘`youreceivedmessage`’’ + *‘‘myusername’’* string:

```
this.socket.on('youreceivedmessage'+this.current_user.username, (message) => {
  message = {
    from: message.from,
    to: message.to,
    text_content: message.text_content,
    message_type: 'youreceivedmessage'
  }
  observer.next(message);
});
```

If the recipient’s chat is opened, the latter will send a confirmation about receiving and reading the message by emitting a ‘‘`yourmessagereceived`’’, which sets the ‘‘`otheruseronline`’’ variable to `true`; if this doesn’t happen in the first 1.5 seconds after the message is sent, the sender creates a notification with an “unread message” subject and sends it to the recipient:

```
setTimeout(() => {
  if(!this.otheruseronline){
    this._chatmessageservice.notifyUnreadMessage(this.player1, this.player2)
  }
  else{ this.otheruseronline = true}
}, 1500)
```

The ‘‘`notifyUnreadMessage()`’’ function makes an HTTP request to the ‘‘`/createnotification`’’ route and emits a ‘‘`newnotification`’’ message, which the server’s Socket.IO instance forwards to the other client, making a red notification badge appear.

# 4.4 Notifications

The notifications in this app are very versatile as there is only one Socket.IO emit keyword used for them, which is ‘‘newnotification’’. What really matters is the ‘‘notification_type’’ field; this structure allows every component to implement the ‘‘listenToNotifications()’’ function from the ‘‘notifications.service.ts’’ file in their own way:

```
// Whatever the notification is, we forward it to the Notification or Header Component
listenToNotifications(current_user: String): Observable <any>{
  return new Observable<any>((observer) => {
    this.socket.on('newnotification'+current_user, (notification) => {
      observer.next(notification)
    })
  })
}
```

In fact, both the Notification Component and the Header Component implement this function for their own purposes: the first one actually detects and shows notifications (such as match invites, unread messages, communications from the moderators) and the second one acts as a notification counter to make red badges appear in the header’s HTML component.

To send a notification, all a client service has to do is: 1) POST to the ‘‘/createnotification’’ server route so the notification gets inserted into the database and 2) emit a ‘‘newnotification’’ message with Socket.IO-client. Then the server will redirect the notification to the receiver specified in the “user” field as seen in the picture below:

```
// When a user sends a notification to another user
socket.on('newnotification', (notification) => {
  socket.broadcast.emit('newnotification'+notification.user, notification)
})
```

For a header to detect an incoming notification, it needs to listen for notifications directed to the current user, but since the header component is loaded even in the Login or Register page when the user is not yet authenticated, we need to wait for the user to log in first:

```
ngOnInit() {
  // Checking every second if the user is logged before starting to listen to notifications sent to him
  this.interval = setInterval(() => {
    this.current_user = JSON.parse(JSON.parse(JSON.stringify(localStorage.getItem('current_user'))))
    if(this.current_user != null && this.current_user != undefined){
      this.countUnreadNotifications()
      clearInterval(this.interval)
    }
  }, 1000)
}
```

Every second, the header component checks if the user is authenticated; as soon as the ‘‘current_user’’ object appears in the browser’s LocalStorage, it means that the user logged on and the component starts listening for incoming notifications.

Based on the notifications' type, the red badge appears either on the friends' section button (in case of friend requests) or on the notifications' section button (in case of match invites, unread messages from a friend or communications from a moderator):

```
countUnreadNotifications(){
  this._notificationsService.listenToNotifications(this.current_user.username).subscribe((notification) => {
    if(!this.notificationsTab && ( notification.notification_type == 'matchinvite'
                                || notification.notification_type == 'newmessage'
                                || notification.notification_type == 'modmessage')){
      this.unreadnotifications += 1
    }
    else if(!this.friendsTab && notification.notification_type == 'friendrequest'){
      this.unreadfriendrequests += 1
    }
    else if(notification.notification_type == 'closeheadermenus'){
      this.notificationsTab = false
      this.friendsTab = false
      this.profileTab = false
      this.canopenmenus = false
    }
    else if(notification.notification_type == 'openheadermenus'){
      this.canopenmenus = true
    }
  })
}
```

On the other hand, the Notification Component makes the notifications actually appear in the list and makes them interactive, meaning that the user can choose to ignore a match invite or accept it, to read a message from a friend or from a moderator.
Once the notification is interacted with, it gets marked as read, therefore it is deleted both from the client's list and from the database, since it has no reason to exist any longer.

First, the component retrieves all the notifications from the database and then starts listening for new ones:

```
ngOnInit(): void {
  this.retrieveNotifications()
  this.listenToNotifications()
}

// Waiting for new notifications
listenToNotifications(){
  this._notificationsService.listenToNotifications(this.current_user.username).subscribe((notification) => {
    // If we accepted a friend's invite to pla but that friend is no longer available, we add a notification
    // (to be deleted automatically after 3 seconds) to the list saying that he's no longer available
    if(notification.notification_type == 'friendnotavailable'){···
    }
    // When we receive a match invite
    else if(notification.notification_type == 'matchinvite'){···
    }
    // When we receive an "unread message" notification
    else if(notification.notification_type == 'newmessage'){···
    }
    // When we receive a communication from the moderators
    else if(notification.notification_type == 'modmessage'){···
    }
  })
}
```

To avoid spam, we made it so that a notification from A to B is created only if A has not sent B a notification with the same notification type yet. For example, if A keeps sending match invites, only the first one creates a notification while the others are ignored.
In case the system fails to detect this sort of spam, some notifications may result doubled.
To make up to this case, when A interacts with a notification sent from B, therefore marking it as read, every other notification from B to A with the same notification type is deleted both from the database and the client's notifications list.

# 5. Authentication & Security

## 5.1 Angular Route Guarding and forcing the authentication

As specified in the project requirements, a user needs to be authenticated in order to use the application. In fact, the table in section **3.5** shows how only the Log-in and Register routes don't require any authentication. On the Angular front-end, we avoid an unidentified user to access the app's functionalities by imposing a Route Guard with the **''canActivate''** criterion:

```
const routes: Routes = [
  {path: 'register', component: RegisterComponent, canActivate : [IsNotAuthenticatedService]},
  {path: 'login', component: LoginComponent, canActivate : [IsNotAuthenticatedService]},
  {path: '', component: HomeComponent, canActivate: [IsAuthenticatedService]},
  {path: '**', redirectTo: '/'}
]
```

As shown, the 'register' and 'login' Angular routes are reachable by those who are not authenticated, whereas any other route is reachable by logged users. Here is how the **''IsAuthenticatedService''** works:

```
import { Injectable } from '@angular/core';
import { Router, CanActivate } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable({
  providedIn: 'root'
})

export class IsAuthenticatedService implements CanActivate {

  constructor(public _auth: AuthService,
              public _router: Router) { }

  canActivate(): boolean {
    if (!this._auth.isAuthenticated()) {
      this._router.navigateByUrl('/login')
      return false;
    }
    return true;
  }
}
```

This service imports another service called **''AuthService''**, whose only function is that of determining whether a user is authenticated or not; said function is called **''isAuthenticated()''** (picture below). The **''IsAuthenticatedService''** also implements the **''CanActivate''** interface, so it needs to specify what the **''canActivate()''** function does. In this case, the **''canActivate()''** function returns true if the user is logged in, whereas the **''IsNotAuthenticatedService''** 's version of the **'canActivate()'** function returns true if the user is not logged in.

```
public isAuthenticated(): boolean {
  const jwthelper: JwtHelperService = new JwtHelperService()
  const token = localStorage.getItem('auth_token');
  if(token != null && token != undefined){
    return !jwthelper.isTokenExpired(token);
  }
  return false;
}
```

The ''`isAuthenticated()`'' function from ''`AuthService`'' uses the Java Web Token Helper library to determine if an authentication token is expired or not.

All that's left to do is telling the ''`app.module.ts`'' file to use the `IsAuthenticatedService` (or `IsNotAuthenticatedService`) as a ''`CanActivate`'' criterion for a specific route as seen in the first picture of this **5.1** section, which will then invoke its ''`canActivate()`'' function and return a Boolean value, resulting in stopping or letting the user access the desired Angular route.

# 5.2 Saving and verifying the password

The user-password validation logic is stored in the ''`/login`'' route, but first we need to look at how the client sends the credentials through a GET request to the server. The Login Component covers basic functions such as capturing the user's input credentials, showing error messages and loading the password-change forms if required.

The authentication request is performed by the ''`login.service.ts`'' file, which contains the ''`login()`'' function. Such function creates request headers specifying the authorization level for the authentication; in our case, we use a Basic Authentication strategy:

```
login(userLogin: UserLogin) : Observable < any >  {
  const options = {
    headers: new HttpHeaders({
        Authorization: 'Basic ' + btoa(userLogin.username + ':' + userLogin.password),
        'cache-control': 'no-cache',
        'Content-Type': 'application/x-www-form-urlencoded'
    })
  };
```

Mind that ''`UserLogin`'' is just a typescript class containing a username and a password.

```
return this._httpClient.get(this.baseURL, options).pipe(tap( (data) => {
    this.response = data;
    if(this.response != 'error'){
      if(this.response != 'banned'){
        if(this.response.needspasswordchange == true){
          localStorage.clear()
          localStorage.setItem('username', JSON.stringify(this.response.username))
        }
        else{
          localStorage.clear()
          // Saving the authentication token and user info in the browser's LocalStoragae
          this.token = this.response.token;
          localStorage.setItem('auth_token', this.token);
          this.user = this.response.user;
          localStorage.setItem('current_user', JSON.stringify(this.user));}}}}));
```

The GET request showed in the image above is directed to the ‘‘`/login`’’ route, which will send two types of response: an error message (basically an ‘‘`error`’’ string indicating that the credentials are wrong) or the user’s information, with some optional conditions such as ‘‘`banned`’’ or ‘‘`needspasswordchange`’’. In the first case, the Login Component shows an error message, and in the second case the Login Component tells the user that he has been banned or that he needs to change the password, showing two new forms for inputting the new password.

The server’s login route uses Passport middleware to manage the authentication:

```
passport.use(
    new passportHTTP.BasicStrategy(function(username, password, done) {
        User.findOne({ username: username }, (err, user) => {
            if(!user || err){
                return done(null, "error")
            }
            if (user.validatePassword(password)) {
                return done(null, user);
            }
            return done(null, "error")
        });
    })
);
```

Usually, an error is returned when the password is not validated or if the user does not exist, but we returned “null” so that the ‘‘`router.get()`’’ function can still receive a positive response and returning an error message to the client. This is because errors with status codes such as 403 or 500 are not detected by the HttpClient’s ‘‘`get()`’’ function in the ‘‘`login.service.ts`’’ file.
The ‘‘`validatePassword()`’’ function is defined inside the User model file and is based on the SHA256 hashing algorithm, which confront the digest stored in the database with the one calculated with the same HMAC it had when it was stored in the database for the first time.
If the two digests are equal, then the password is validated (picture on the left). To store the encrypted passwords, we defined the ‘‘`setPassword()`’’ function (used it in the ‘‘`/register`’’ route) in the User model (picture on the right).

```
UserSchema.methods.validatePassword = function (pwd) {
    var hmac = crypto.createHmac("sha256", this.salt);
    hmac.update(pwd);
    var digest = hmac.digest('hex');
    return (this.digest === digest);
};
```

```
var crypto = require("crypto");

UserSchema.methods.setPassword = function (pwd) {
    this.salt = crypto.randomBytes(16).toString('hex');
    var hmac = crypto.createHmac('sha256', this.salt);
    hmac.update(pwd);
    this.digest = hmac.digest('hex');
};
```

As seen above, we added a Salt string to the hash of the password in order to make it more secure; the standard use of a Salt string is that of generating a new one every time a password needs to be stored, and that is exactly what we did by creating a new random string which is 16 characters long.
The next step is to create an HMAC (Hashed Message Authentication Code) based on both the salt and the password. Finally, we store a digest of that in the user's ''`digest`'' field.

Back to the ''`/login`'' route: said route detects GET requests from the Client and uses Passport's ''`authenticate()`'' function to verify the request. The ''`user`'' field in the request ''`req`'' is injected by Passport, and as explained before it can be an ''`error`'' message or the user itself:

```
router.get("/", passport.authenticate("basic", {session: false}), async (req, res) => {
    // If Passport's authentication function returns an "error",
    // it means the user has given us wrong credentials
    if(req.user == "error"){
        return res.json("error")
    }
    // If we reach this point, the user is successfully authenticated
    if(req.user.isbanned){
        return res.json('banned')
    }
    if(req.user.needspasswordchange){
        return res.json({
            needspasswordchange: true,
            username: req.user.username
        })
    }
```

If the user is banned or needs to change the password, the Client is notified, but if it is correctly authenticated then a JSON Web Token is generated containing the user's info, as well as a signed Authentication Token which will be used by the Angular's Route Authentication Guard (**5.1**):

```
else{
    // Generating a JSONWebToken
    var tokendata = {
        username: req.user.username,
        role: req.user.role
    };
    // Generate a signed Authentication Token
    var token_signed = jsonwebtoken.sign(
        tokendata,
        process.env.JWT_SECRET, {
            expiresIn: "5h"
        }
    );
    // Creating a JSON with user data to save in local storage
    const logged_username = jwt_decode(token_signed);
    var logged_user = await User.findOne({username: logged_username.username});
    logged_user.password = undefined;
    logged_user.salt = undefined;
    logged_user.digest = undefined;

    return res.status(200).json({
        error: false,
        errormessage: "",
        token: token_signed,
        user: logged_user
    });
}
```

If a password change is needed, the Clients asks the user to provide a new password; only when the two inputted passwords correspond, a new login request is performed to obtain the Authentication Token.

# 5.3 Preventing unauthorized HTTP requests

In order to prevent random users to send dangerous HTTP requests (such as a POST request to `''/admindashboard''` to ban a user), we implemented the "auth" clause to the routes. This way, only authenticated users can activate certain routes by inserting their authorization token in the request headers (containing username and role), which the server will then decode using the shared `JWT_SECRET` key stored in the `''.env''` file. If the authorization token is invalid, a 401-error code is sent back to the client.

```
const { expressjwt: jwt } = require("express-jwt");
const jwtdecode = require('jwt-decode');

var auth = jwt({
    secret: process.env.JWT_SECRET,
    algorithms: ['sha1', 'RS256', 'HS256']
});
```

As shown above, the `express-jwt` library and `''jwtdecode()''` functions are required to decode the authorization token. That `''auth''` object will be passed to the `''post()''` function:

```
router.post("/", auth, async (req, res) => {
    //Checking wether the user that performs the POST request is
    // authenticated and has an administrator role
    var authorization_token = req.headers.authorization.split(' ')[1];
    var token = jwtdecode(authorization_token)
    if(token.role == "admin"){
        //...
```

In this example, the operations are performed only if the user that made the HTTP request has a moderator role. The same principle is followed by the ‘‘`/myprofile`’’ route, otherwise a user could obtain information about another user just by sending a `POST` request with the desired username. This time, the user role is not essential to the authorization process, but we forced the ‘‘`/myprofile`’’ route to return the information about the user whose username is encoded in the authorization token:

```
router.get('/', auth, function(req, res){
    var authorization_token = req.headers.authorization.split(' ')[1];
    var username = jwtdecode(authorization_token).username
    try{
        User.findOne( {username: username}).then((result) => {
            result.password = undefined
            result.digest = undefined
            result.salt = undefined
            res.json(result)})
    }catch(err){
        console.log(err)
    }
})
```