Department of Computer Science

a.a. 2021-2022

Battleship Game

Exam project work for

Tecnologie e Applicazioni Web [CT-0142]

Mazzon Edoardo (870606)

Index

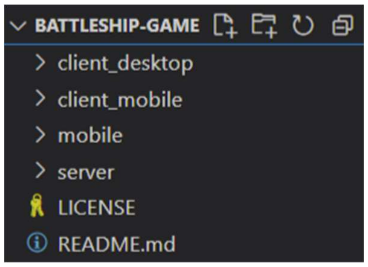
1. Introduction

This report will describe the system architecture and the software components used in the way in which they contribute to achieving the required functionalities.

1.1. Goals

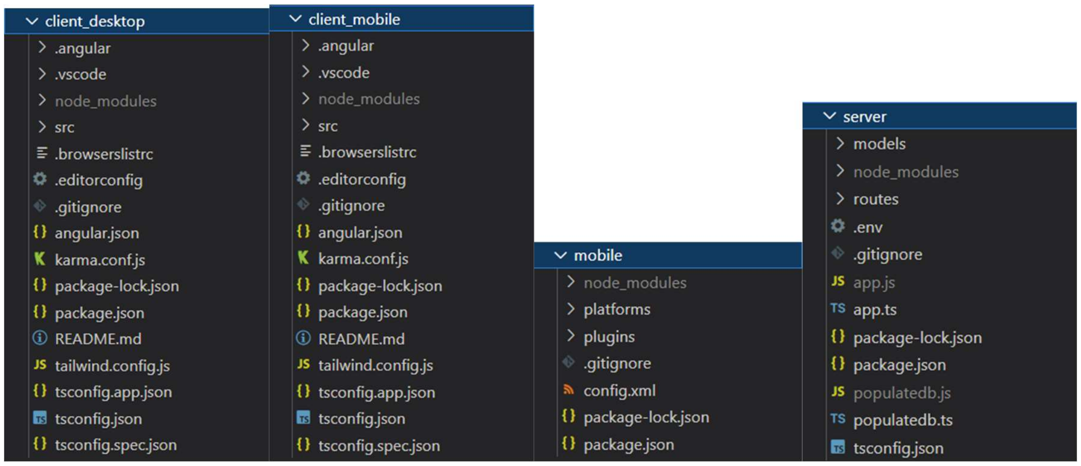
The goal is to develop a full-stack web application, comprising a REST-style API backend and a SPA Angular frontend to let the users play the game of "Battleship".
In addition to the given purpose, we also thought about making the design as user-friendly as possible.

1.2. Structure



```
BATTLESHIP-GAME
  > client_desktop
  > client_mobile
  > mobile
  > server
    LICENSE
    README.md
```

The project's structure is divided in four main directories: *client_desktop*, *client_mobile*, *mobile* and *server*.
The *client_desktop* folder contains the Angular frontend.
The client_mobile folder is a copy of the folder listed above but with few minor differences which will be explained later in this document.
In the *mobile* folder lies the app part to be run with Cordova.
The *server* folder is the part of the code relative to the MongoDB database, *expressJs* logic and *socket.io* (see the pictures below).



```
client_desktop
  > .angular
  > .vscode
  > node_modules
  > src
    .browserslistrc
    .editorconfig
    .gitignore
    angular.json
    karma.conf.js
    package-lock.json
    package.json
    README.md
    tailwind.config.js
    tsconfig.app.json
    tsconfig.json
    tsconfig.spec.json
```

```
client_mobile
  > .angular
  > .vscode
  > node_modules
  > src
    .browserslistrc
    .editorconfig
    .gitignore
    angular.json
    karma.conf.js
    package-lock.json
    package.json
    README.md
    tailwind.config.js
    tsconfig.app.json
    tsconfig.json
    tsconfig.spec.json
```

```
mobile
  > node_modules
  > platforms
  > plugins
    .gitignore
    config.xml
    package-lock.json
    package.json
```

```
server
  > models
  > node_modules
  > routes
    .env
    .gitignore
    app.js
    app.ts
    package-lock.json
    package.json
    populatedb.js
    populatedb.ts
    tsconfig.json
```

Later we will see what is in the individual folders of each main directory.

1.3. Tools and environment variables

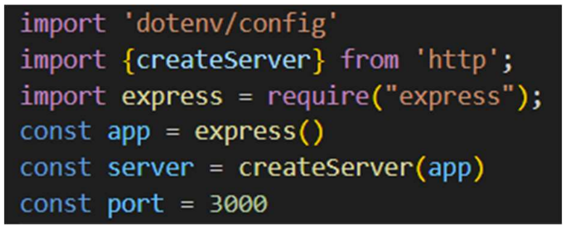
To develop the project we used Visual Studio Code for writing the entire code integrated with GitHub, for facilitating the team work. Also we used VSCode's LiveShare extension which allows editing code together in real time.
For database we have used the free service cloud provided by MongoDB Atlas, which a maximum of 500MB of free capacity. In order to connect to the database, we have created a string as an environment variable in the *.env* file in the *server* folder.

2. HTTP and Socket.io server

To create the server and handle the HTTP requests we used Express, a minimal Node.js web application framework that provides HTTP methods, and Socket.io, a Javascript library that enables real-time bidirectional event-based communication.

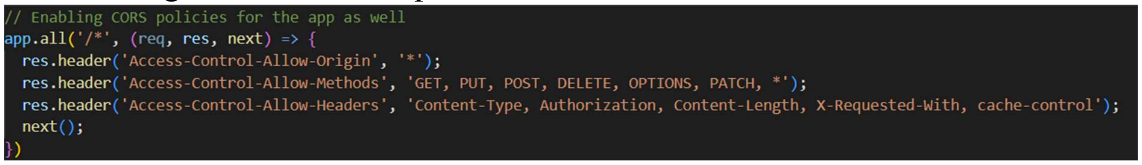
2.1 Client-server HTTP

To set up the server we used the HTTP function ‘createServer’ passing an instance of express as parameter and setting the port on default port for testing (port 3000).



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

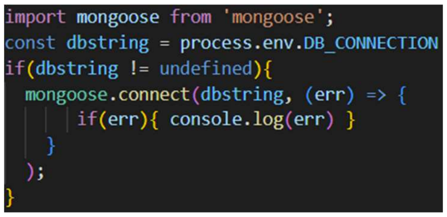
Thanks to CORS (Cross-Origin Resource Sharing) policies we specify the allowed origins for HTTP requests and the relative methods.



```
// 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();
})
```

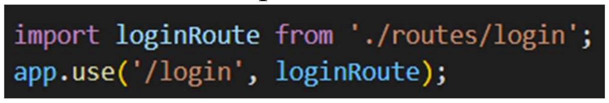
Every time the server receives a request from the client, the request gets parsed through the “body-parser” library.

Next, we connect to the MongoDB Atlas service through the connection string stored in the environment file.



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

The HTTP server needs to know where to redirect the requests coming from the client. In order to achieve that, we imported the routes from the various routes files and pass them to the ‘app’ object with “app.use(‘/routename’, importedRouteName” as in the example below.



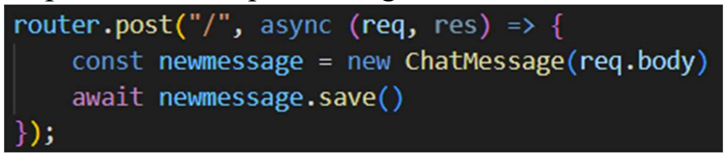
```
import loginRoute from './routes/login';
app.use('/login', loginRoute);
```

The app is ready to start, all we need is for the server to begin listening on the declared port: “server.listen(port)”

To send HTTP request to the server, the client must import the HTTPClient library, which contains, amongst others, functions like get, post, put, etc.
Let's take the chat between two players as an example. Whenever a user sends a message to another user, it must be inserted into the database, thus the client performs a call to the "post" function from HTTPClient:

```
this._httpClient.post(this.baseURL+'chatmessage', newmessage).subscribe()
```

As seen in the picture above, we must specify the route's name and then we pass an object as the request body. The server then redirects the call to the right route, which will process the request using the extracted data from the body.

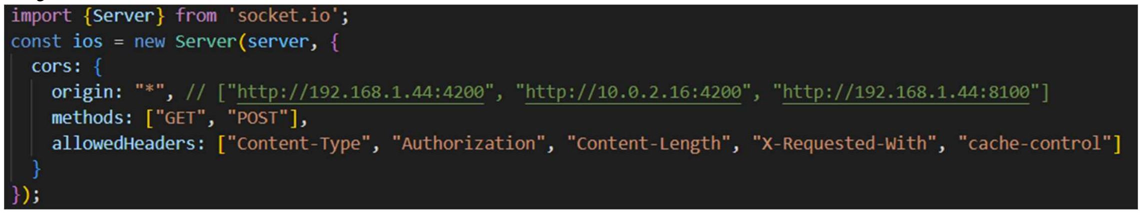


```
router.post("/", async (req, res) => {
    const newmessage = new ChatMessage(req.body)
    await newmessage.save()
});
```

The 'router' variable contains an instance of the Router object from the express library: const router = express.Router()

2.2 Client-server Socket.io

The server keeps a 'Server' instance running, which is an object from the Socket.io library that is responsible for listening and managing the emitted messages coming from the clients. CORS policies must be specified for this object as well.

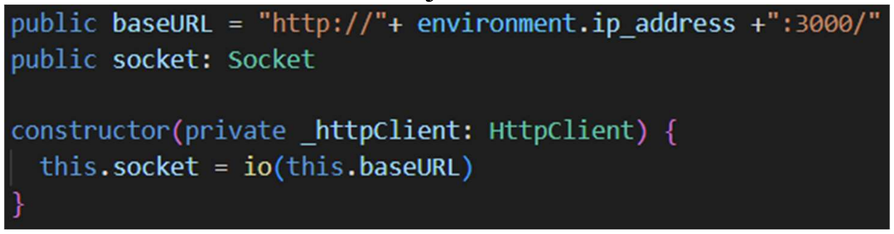


```
import {Server} from 'socket.io';
const ios = new Server(server, {
  cors: {
    origin: "*", // ["http://192.168.1.44:4200", "http://10.0.2.16:4200", "http://192.168.1.44:8100"]
    methods: ["GET", "POST"],
    allowedHeaders: ["Content-Type", "Authorization", "Content-Length", "X-Requested-With", "cache-control"]
  }
});
```

Meanwhile, the client must import the 'socket.io-client' library

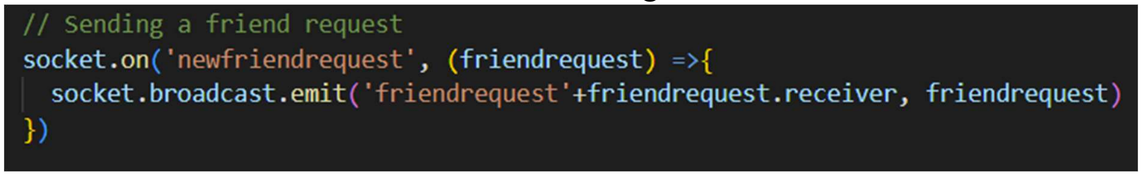
```
import {io, Socket} from 'socket.io-client';
```

and create an instance of the Socket object:



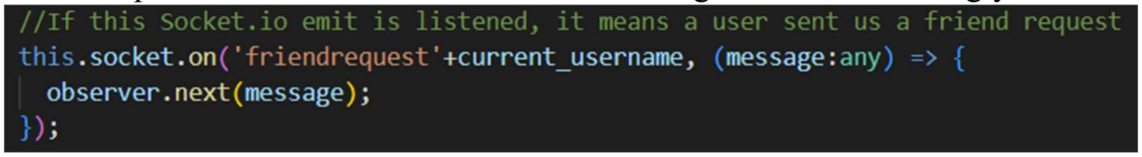
```
public baseURL = "http://"+ environment.ip_address +":3000/"
public socket: Socket

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

From then on, through the ‘emit’ function, the client cans end messages to the server, which will be able to detect them using the ‘on’ function:



```
// Sending a friend request
socket.on('newfriendrequest', (friendrequest) =>{
  socket.broadcast.emit('friendrequest'+friendrequest.receiver, friendrequest)
})
```

Each Socket.io message has a name and a content: in this case, the name is ‘newfriendrequest’ and the content is a JSON called ‘friendrequest’. In the chosen example, the ‘friendrequest’ JSON contains a ‘receiver’ field that describes the user who received the friend request invite; the server uses this field and uses the ‘emit’ function to forward the message to the designated receiver.
The ’on’ function can be used by the client-side socket.io instance; in fact, the friend request receiver uses it to detect the message and act accordingly:

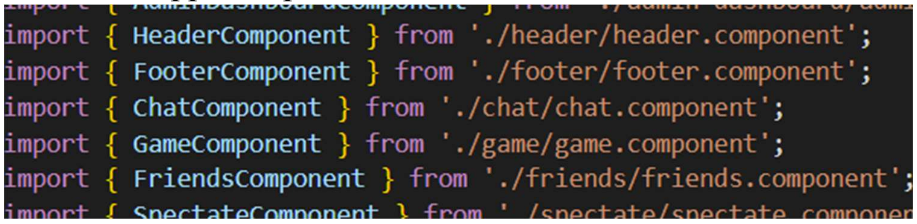


```
//If this Socket.io emit is listened, it means a user sent us a friend request
this.socket.on('friendrequest'+current_username, (message:any) => {
  observer.next(message);
});
```

3. System architecture

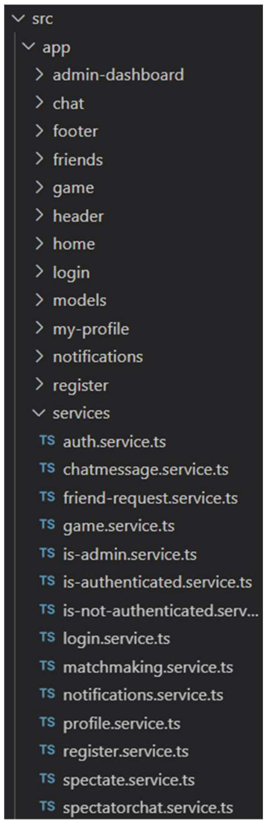
3.1 Angular components and services

The web app is made of Angular components, the main one being App Component which can contain and recall all the other ones.
Every component has an HTML file and a TypeScript file; the TypeScript file manages the component's logic. To dynamically load a component into the app, this component has to be declared in the "app.modules.ts" file (picture below) and must be then imported through the <app-componentname> tag in the HTML file of the App Component.



```
import { AdminDashboardComponent } from './admin-dashboard/admi
import { HeaderComponent } from './header/header.component';
import { FooterComponent } from './footer/footer.component';
import { ChatComponent } from './chat/chat.component';
import { GameComponent } from './game/game.component';
import { FriendsComponent } from './friends/friends.component';
import { SpectateComponent } from './spectate/spectate.componen
```

This way, the TS logic will be loaded too, and it can invoke functions from imported services that we created and organized into the dedicated "services" folder (picture on the right).
Such services act as a connection to the client and the server because they contain most of the logic responsible for the HTTP request and the Socket.IO messages emits.



```
src
  app
    admin-dashboard
    chat
    footer
    friends
    game
    header
    home
    login
    models
    my-profile
    notifications
    register
    services
      auth.service.ts
      chatmessage.service.ts
      friend-request.service.ts
      game.service.ts
      is-admin.service.ts
      is-authenticated.service.ts
      is-not-authenticated.serv...
      login.service.ts
      matchmaking.service.ts
      notifications.service.ts
      profile.service.ts
      register.service.ts
      spectate.service.ts
      spectatorchat.service.ts
```

Therefore, a component must make use of the right functions specified in the service file that they are importing in order to communicate with the server; in just a few cases, the HTTPClient library is utilized directly in a component without the support of a service file.

For further explanation about HTTP requests and the Socket.IO library, go back to chapter 2.

3.2 Endpoints list

The following is a list of all the route endpoints where I will explain the respective actions and methods for each one.

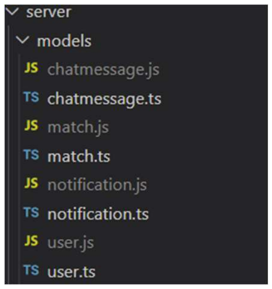
- “/acceptfriendrequest”: with POST method, login required. When a user accepts a friend request of other user, both friends list are updated in the database.
- “/admindashboard”: with POST method and administrator required. Contains user management operations which are activated based on the “request type”.
- “/blacklistusers”: with POST method and login required. Adds blocked users to the “blacklisted_users” list and updates it.
- “/chatmessage”: with POST method inserts a new message in the database and with PUT method returns the last 25 messages between the two users. Login required.
- “/creatematch”: with POST method and login required. When a match starts, a match is inserted in the database and both players’ “/games_played” counter increases. Also updates the “recently played” list.
- “/createnotification”: with POST method and login required. Inserts a new notification in the database.
- “/deletenotification”: with POST method and login required. Deletes the notification when a user marks it as read.
- “/friendrequest”: with POST method and login required. Sends a friend request to the other user if one of the two is not in the “blacklisted_users” list of the other user.
- “/login”: with GET method and no requirements, authenticates a user. With POST method and login required changes a user’s password.
- “/losegame”: with POST method and login required. When a user loses a game, we set his ‘current_winstreak’ field to 0.
- “/myprofile”: with POST method and login required. Refreshes the user’s localStorage with updated info from the database and sends a response containing the user data.
- “/register”: with POST method and no requirements. When a user registers, his information is saved in a new inserted database user.
- “/rejectfriendrequest”: with POST method and login required. When a user rejects a user's friend request, his username disappears from the ‘pending_friend_requests’ list of the other user.

- “/removefriend”: with POST method and login required. Deletes the desired user from the requester’s friend list.
- “/retrievenotifications”: with POST method and login required. When a user opens the notifications panel, returns all the user's unread notifications.
- “/searchusers”: with POST method and login required. Returns the 5 users with the most similar username as the one typed in.
- “/unblockusers”: with POST method and login required. Removes a user from the blacklist of the requester.
- “/updateaccuracy”: with POST method and login required. Called whenever a user takes a shot and gets the shot result back; the accuracy is updated as the hit shots count over the total shots count.
- “/userinfo”: with POST method and login required. Returns some user statistics, such as winrate, winstreak and accuracy.
- “/wingame”: with POST method and login required. When a user wins a game, increments his ‘gameswon’ counter and his current winstreak. If his current winstreak is greater than his max winstreak, the current winstreak becomes the max winstreak.

4. Data models

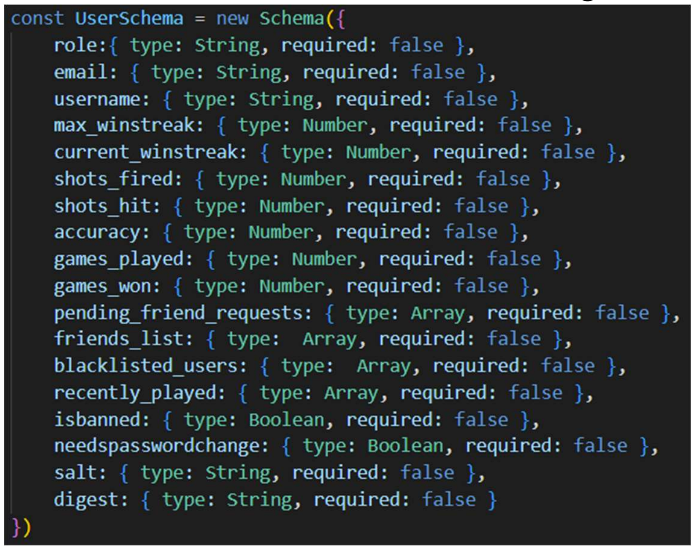
4.1 MongoDB models

To create MongoDB collections we used the 'mongoose' module, a MongoDB object modeling tool. We have created a folder named 'models' (see the picture on the right) which includes all of our MongoDB models.

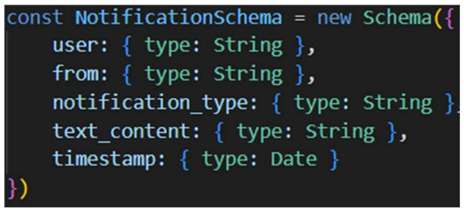


```
server
  models
    chatmessage.js
    chatmessage.ts
    match.js
    match.ts
    notification.js
    notification.ts
    user.js
    user.ts
```

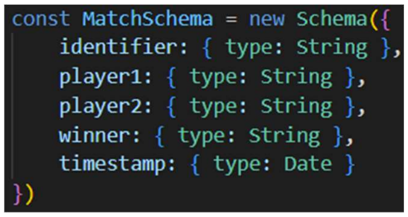
We thought of creating only four models in order to simplify the database structure as much as possible and provide with some more elasticity. The four collections are User, Match, Notification and ChatMessage:



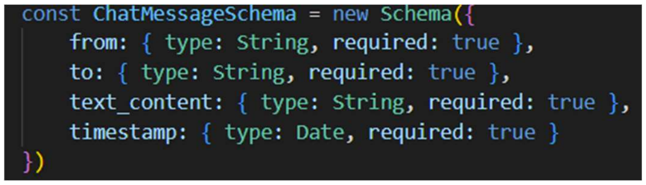
```
const UserSchema = new Schema({
    role:{ type: String, required: false },
    email: { type: String, required: false },
    username: { type: String, required: false },
    max_winstreak: { type: Number, required: false },
    current_winstreak: { type: Number, required: false },
    shots_fired: { type: Number, required: false },
    shots_hit: { type: Number, required: false },
    accuracy: { type: Number, required: false },
    games_played: { type: Number, required: false },
    games_won: { type: Number, required: false },
    pending_friend_requests: { type: Array, required: false },
    friends_list: { type:  Array, required: false },
    blacklisted_users: { type:  Array, required: false },
    recently_played: { type: Array, required: false },
    isbanned: { type: Boolean, required: false },
    needspasswordchange: { type: Boolean, required: false },
    salt: { type: String, required: false },
    digest: { type: String, required: false }
})
```



```
const NotificationSchema = new Schema({
    user: { type: String },
    from: { type: String },
    notification_type: { type: String },
    text_content: { type: String },
    timestamp: { type: Date }
})
```

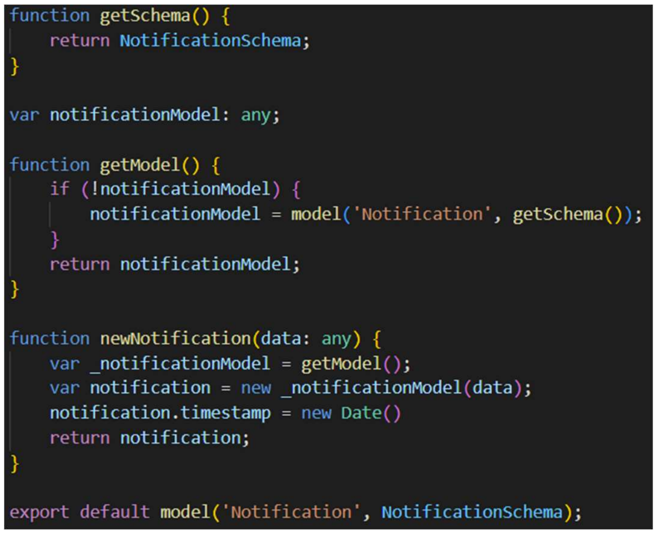


```
const MatchSchema = new Schema({
    identifier: { type: String },
    player1: { type: String },
    player2: { type: String },
    winner: { type: String },
    timestamp: { type: Date }
})
```



```
const ChatMessageSchema = new Schema({
    from: { type: String, required: true },
    to: { type: String, required: true },
    text_content: { type: String, required: true },
    timestamp: { type: Date, required: true }
})
```

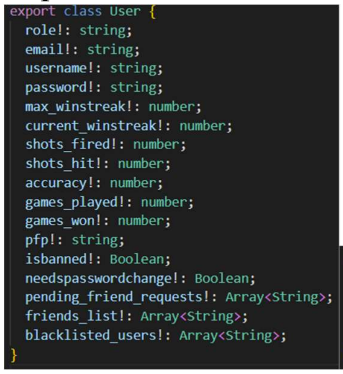
All of these Schemas have a ‘getSchema’, a ‘getModel’ and a ‘new<SchemaName>’ method which allows us to create instances of the model.



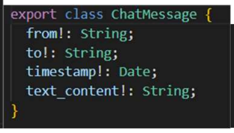
```
function getSchema() {
    return NotificationSchema;
}

var notificationModel: any;

function getModel() {
    if (!notificationModel) {
        notificationModel = model('Notification', getSchema());
    }
    return notificationModel;
}

function newNotification(data: any) {
    var _notificationModel = getModel();
    var notification = new _notificationModel(data);
    notification.timestamp = new Date()
    return notification;
}

export default model('Notification', NotificationSchema);
```

4.2 Angular models

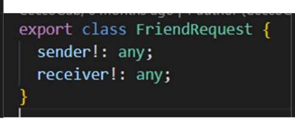
We also created some Angular models, which are totally optional, to help us incorporate information into JSONS that we pass to the server through http requests.



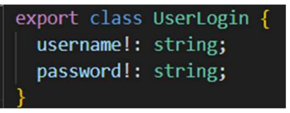
```
export class User {
  role!: string;
  email!: string;
  username!: string;
  password!: string;
  max_winstreak!: number;
  current_winstreak!: number;
  shots_fired!: number;
  shots_hit!: number;
  accuracy!: number;
  games_played!: number;
  games_won!: number;
  pfp!: string;
  isbanned!: Boolean;
  needspasswordchange!: Boolean;
  pending_friend_requests!: Array<String>;
  friends_list!: Array<String>;
  blacklisted_users!: Array<String>;
}
```



```
export class ChatMessage {
  from!: String;
  to!: String;
  timestamp!: Date;
  text_content!: String;
}
```



```
export class FriendRequest {
  sender!: any;
  receiver!: any;
}
```



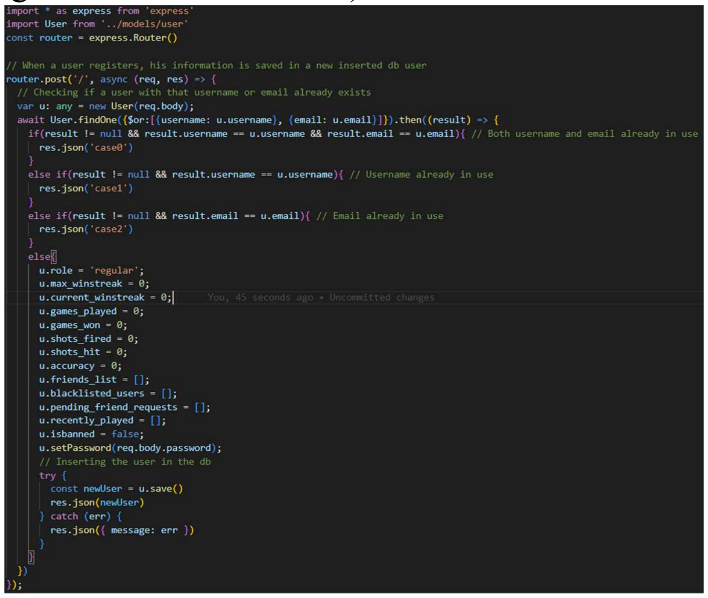
```
export class UserLogin {
  username!: string;
  password!: string;
}
```

For example, if we were to send a whole user object, we could avoid typing the whole JSON in curly brackets with all of its numerous fields: all we need to do is to create a User through the ‘new User’ constructor and passing the user data to it.

5. User authentication

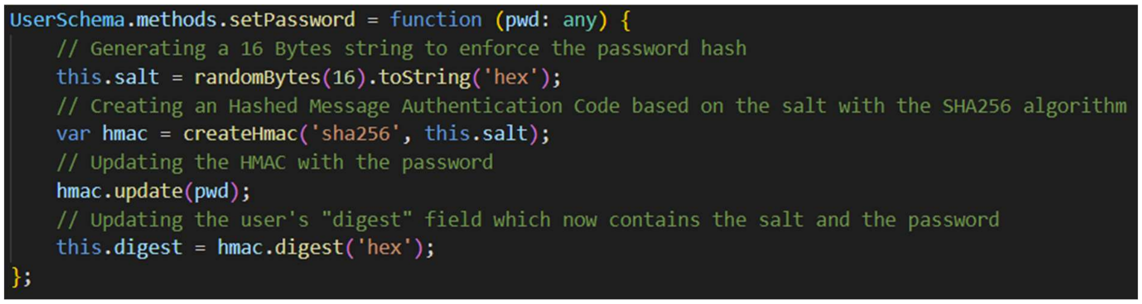
5.1. Registration and login

A new user can be registered by sending a POST request to the '/register' route; the request's body contains basic information such as the username, the e-mail address and the chosen password. First, the server checks if the user already exists by looking up the e-mail and the username. If the credentials are not already in use, a new user is insterted into the database and its fields (e.g. the number of won games or the friends list) are initialized.



```
import * as express from 'express'
import User from '../models/user'
const router = express.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: any = new User(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){ // Both username and email already in use
      res.json('case0')
    }
    else if(result != null && result.username == u.username){ // Username already in use
      res.json('case1')
    }
    else if(result != null && result.email == u.email){ // Email already in use
      res.json('case2')
    }
    else{
      u.role = 'regular';
      u.max_winstreak = 0;
      u.current_winstreak = 0;       You, 45 seconds ago • Uncommitted changes
      u.games_played = 0;
      u.games_won = 0;
      u.shots_fired = 0;
      u.shots_hit = 0;
      u.accuracy = 0;
      u.friends_list = [];
      u.blacklisted_users = [];
      u.pending_friend_requests = [];
      u.recently_played = [];
      u.isbanned = false;
      u.setPassword(req.body.password);
      // Inserting the user in the db
      try {
        const newUser = u.save()
        res.json(newUser)
      } catch (err) {
        res.json({ message: err })
      }
    }
  })
});
```

The value of the 'password' field is set through the 'setPassword' function defined in the User model:

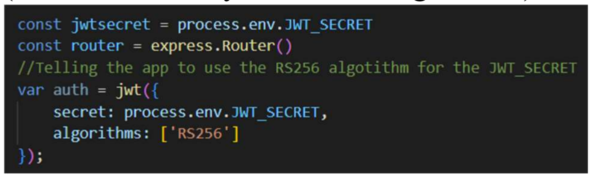


```
UserSchema.methods.setPassword = function (pwd: any) {
    // Generating a 16 Bytes string to enforce the password hash
    this.salt = randomBytes(16).toString('hex');
    // Creating an Hashed Message Authentication Code based on the salt with the SHA256 algorithm
    var hmac = createHmac('sha256', this.salt);
    // Updating the HMAC with the password
    hmac.update(pwd);
    // Updating the user's "digest" field which now contains the salt and the password
    this.digest = hmac.digest('hex');
};
```

This function uses the plain-text password from the HTTP request as a parameter, then sets the the 'salt' field of the newly created user to a 16 bytes string to enforce the password hash with the 'randomBytes' method provided by the 'crypto' module.

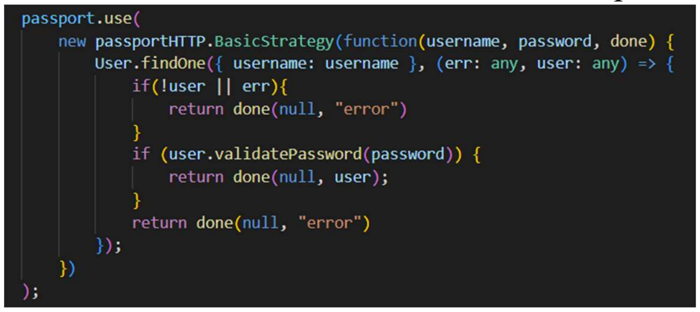
Then we assign the obtained string to a new variable named ‘hmac’ which creates a Hashed Message Authentication Code with the SHA256 algorithm using the ‘createHmac’ method and the inserted password as a parameter. Finally, we store the digested authentication code in the user’s ‘digest’ field.

As for the login implementation, we used libraries such as passport, an authentication middleware for Nodejs, jwt-decode, a small browser library that helps decoding JWTs token, and express-jwt, a middleware for validating JWTs (JSON Web Tokens) through the ‘jsonwebtoken’ module.
First of all, we set a variable that uses our secret string from the ‘.env’ file. Then, we tell the app what the jwt secret key is and a list of the allowed token-signing algorithms (in this case only the R256 algorithm).



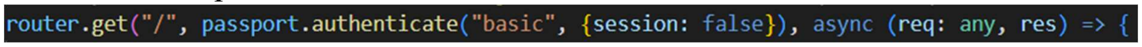
```
const jwtsecret = process.env.JWT_SECRET
const router = express.Router()
//Telling the app to use the RS256 algotithm for the JWT_SECRET
var auth = jwt({
    secret: process.env.JWT_SECRET,
    algorithms: ['RS256']
});
```

At this point we use the ‘basic’ scheme of passport which uses a username and password to authenticate a user. These credentials are transported in plain text.



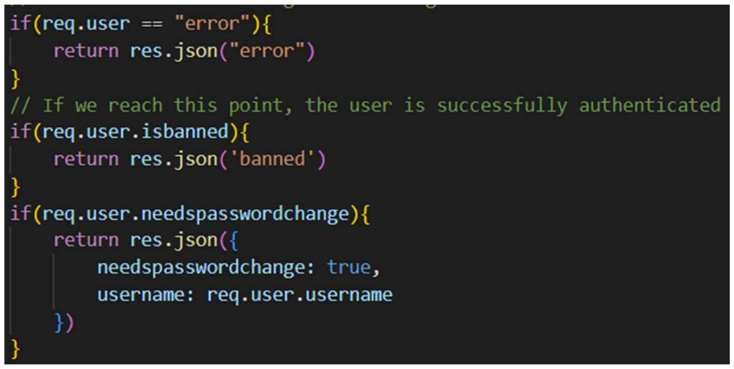
```
passport.use(
    new passportHTTP.BasicStrategy(function(username, password, done) {
        User.findOne({ username: username }, (err: any, user: any) => {
            if(!user || err){
                return done(null, "error")
            }
            if (user.validatePassword(password)) {
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

As seen above, if the credentials are wrong the “error” string is assigned to the ‘user’ object, whereas if the credentials are valid the ‘user’ variable will contain the whole User document. The ‘user’ object will be injected into the request as explained shortly ahead.
Now that we described to Passport the version of the ‘basic’ strategy we want to use, we can call the ‘authenticate’ function in the route’s GET request, injecting 'user’ in the request::



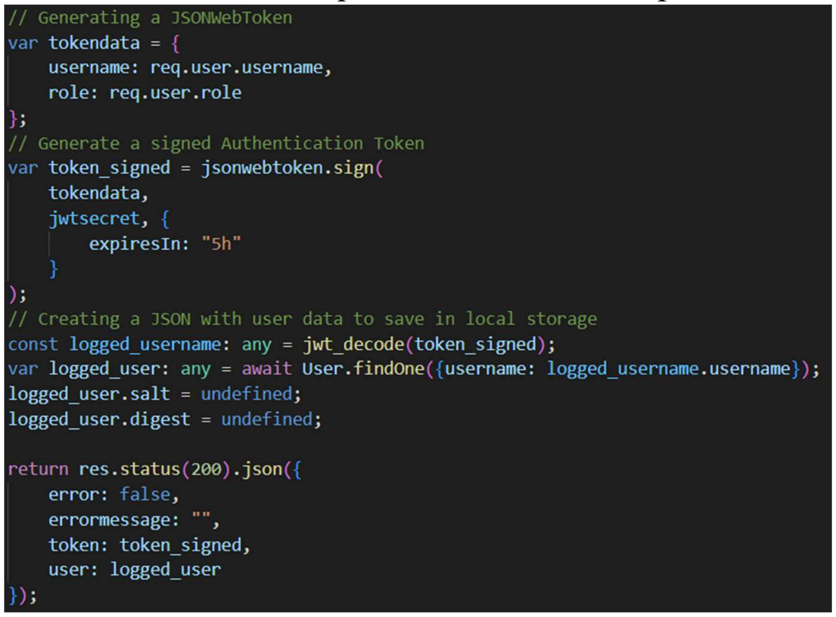
```
router.get("/", passport.authenticate("basic", {session: false}), async (req: any, res) => {
```

In the get function as a first step we check whether the user has provided the right credentials or not. Based on the authentication outcome, the user can be signed in or warned of the invalid credentials, his password being expired or his account being temporarily banned:



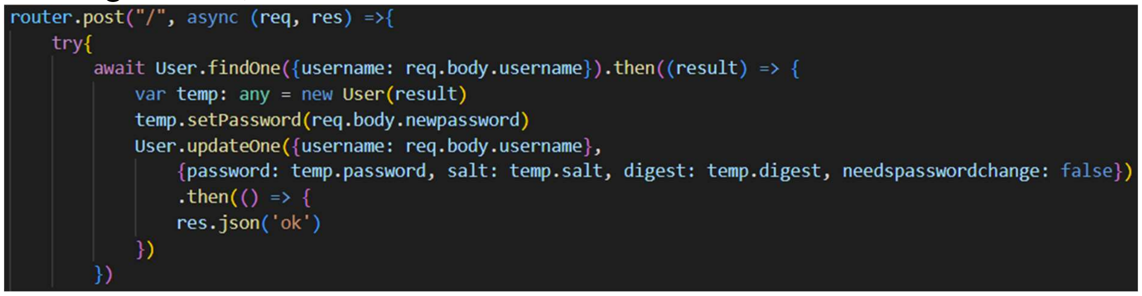
```
if(req.user == "error"){
    return res.json("error")
}
// If we reach this point, the user is successfully authenticated
if(req.user.isbanned){
    return res.json('banned')
}
if(req.user.needspasswordchange){
    return res.json({
        needspasswordchange: true,
        username: req.user.username
    })
}
```

To completely sign in the user, the server responds to the GET login request by sending the user's information and a signed token to be stored in the browser's local storage, containing the username and the role. Such token will be useful later once the user sends HTTP requests to authorization-protected routes:



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

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

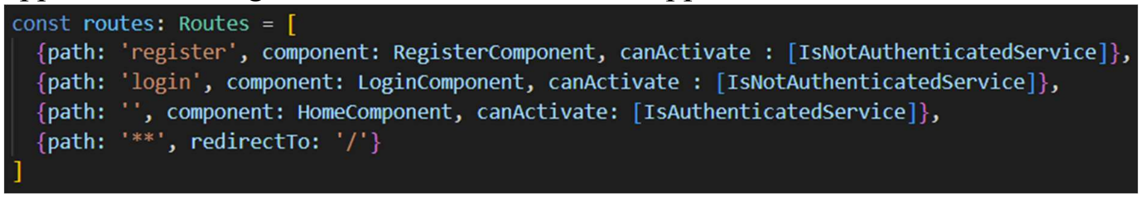
In case the password is expired, the client is notified and loads a password reset form, where he will be asked to input the current password and the new password which will be communicated to the server through a POST request to the '/login' route, as seen below.



```
router.post("/", async (req, res) =>{
    try{
        await User.findOne({username: req.body.username}).then((result) => {
            var temp: any = new User(result)
            temp.setPassword(req.body.newpassword)
            User.updateOne({username: req.body.username},
                {password: temp.password, salt: temp.salt, digest: temp.digest, needspasswordchange: false})
                .then(() => {
                res.json('ok')
            })
        })
```

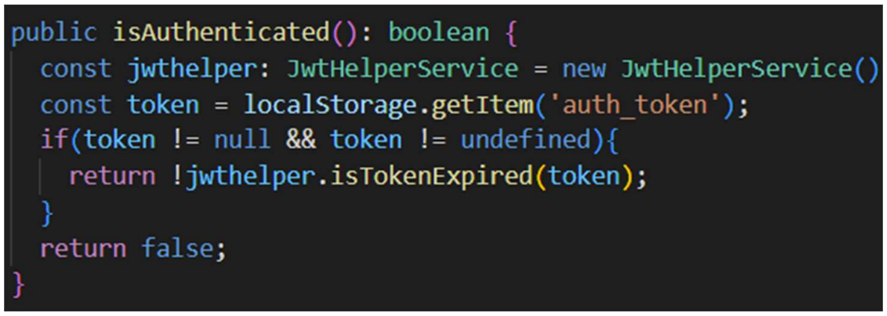
5.2 Protected routes

Since the user must be authenticated to use the app, we imposed that the Angular routes require an authentication token to be reached. Obviously the authentication pages such as the 'register' and the 'login' page require the user to not be authenticated.
This level of control is achievable thanks to the 'canActivate' keyword that we applied to the Angular routes declared in the 'app.module.ts' file:



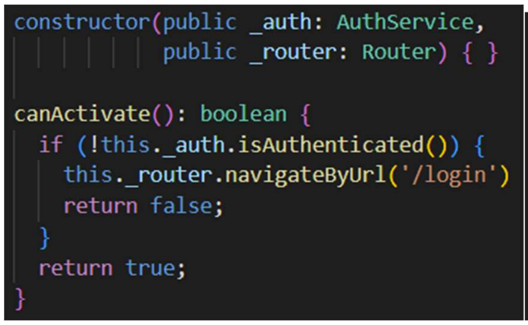
```
const routes: Routes = [
  {path: 'register', component: RegisterComponent, canActivate : [IsNotAuthenticatedService]},
  {path: 'login', component: LoginComponent, canActivate : [IsNotAuthenticatedService]},
  {path: '', component: HomeComponent, canActivate: [IsAuthenticatedService]},
  {path: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

To each 'canActivate' case we specify which method to use to check whether the user is authenticated or not; to do so, we created an 'isAutenticated' function that returns a Boolean value.

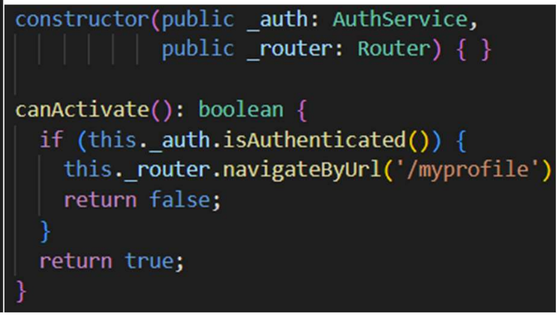


```
public isAuthenticated(): boolean {
  const jwthelper: JwtHelperService = new JwtHelperService()
  const token = localStorage.getItem('auth_token');
  if(token != null && token != undefined){
    return !jwthelper.isTokenExpired(token);
  }
  return false;
}
```

This function will be used by two particular services: isAuthenticatedService and isNotAuthenticatedService, both of which implements the CanActivate interface and consequently must implement its 'canActivate' function as well:



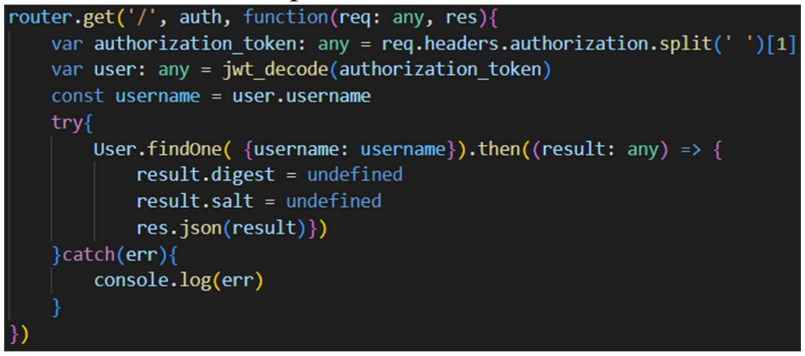
```
constructor(public _auth: AuthService,
            public _router: Router) { }

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



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

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

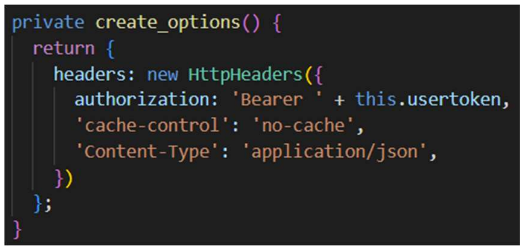
On the left we can see how the 'canActivate' function is implemented in the isAuthenticatedService: it returns true if the user is authenticated or, if not, redirects the user to the login page. On the right side, the isNotAuthenticatedService's 'canActivated' implementation works in the opposite way: it will return true if the user is not authenticated or it will redirect him to the homepage if he is already logged in. Notice how in the latter picture there is a redirect to the '/myprofile', which doesn't exist, but the user will still be redirected to the home page. This is possible since, as seen in the first picture of this section, we specified that in case of attempted navigation attempts to non-existing routes, the user will be sent to the home page, that is to say the '/ ' route.

Just like the Angular Routes are protected by the authentication, some server's Express routes have also adopted a similar solution. Let's take the '/myprofile' route's GET method as an example:



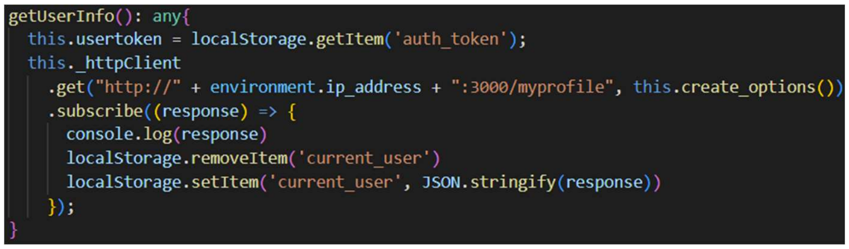
```
router.get('/', auth, function(req: any, res){
    var authorization_token: any = req.headers.authorization.split(' ')[1]
    var user: any = jwt_decode(authorization_token)
    const username = user.username
    try{
        User.findOne( {username: username}).then((result: any) => {
          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

To activate route with the 'auth' clause in the request the user must be authenticated and pass the authentication token in the header; the token was previously saved in the localStorage of the browser during the login.
This token is inserted in the header and is decoded by server that checks which user the token belongs to and only then can return the requested information to the user. This mechanic is safe because only an authenticated user can possess a valid token, meaning that the username contained in the token is the one of the authenticated user. In case that a user tries to access one of the administrative routes, the server check if the token's 'role' field contains the string "admin".
On the client-side, we have created a function in the "profile.service.ts" file that edits the header's authorization field so that it contains the signed token with the username and password.



```
private create_options() {
  return {
    headers: new HttpHeaders({
      authorization: 'Bearer ' + this.usertoken,
      'cache-control': 'no-cache',
      'Content-Type': 'application/json',
    })
  };
}
```

This function is utilized later in the "getUserInfo" function which actually performs the GET request and retrieves the user's data, saving it in the localStorage.



```
getUserInfo(): any{
  this.usertoken = localStorage.getItem('auth_token');
  this._httpClient
    .get("http://" + environment.ip_address + ":3000/myprofile", this.create_options())
    .subscribe((response) => {
      console.log(response)
      localStorage.removeItem('current_user')
      localStorage.setItem('current_user', JSON.stringify(response))
    });
}
```

6. Mobile

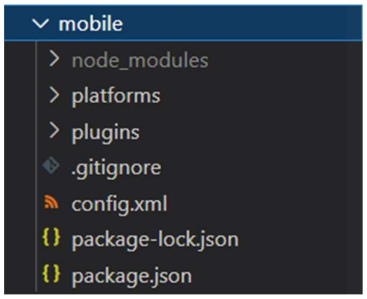
To generate the mobile app we used Apache Cordova, a cross-platform tool for developing a Hybrid App. Cordova is responsible for converting the web app into a native app by properly building and packaging the web resources so that they can run on the targeted device platform.
First install the Cordova module globally by typing on your terminal "npm i cordova -g".

6.1 Angular implementation for mobile

To implement Angular for mobile applications, we have to clonate the 'client_desktop' folder into the root folder of the project, which we called 'client_mobile'.
There are little difference between these two versions, for example in the client_mobile project there aren't some codes uses for desktop version. There are also some additional code such as the functionality of rotate a ship in the placement phase.

6.2 Implementing the mobile app

After installing Cordova globally, we can go to the project's root folder and create a new Cordova project using the "cordova create mobile io.cordova.hellocordova mobile" command, which creates the required directory structure for your Cordova app.
Once executed we will have a structure similar to the image on the right, where in the platforms folder we will add the platforms that we want our app to run on (in our case only 'android') and in the plugins folder we will install all the plugins useful for the functioning of the app.
The config.xml is a global configuration file that specifies the core Cordova API features, plugins, and platform-specific settings, and package.json is the standard package.json file with dependencies and versions of Cordova Plugins.



We chose the Android platform among the three available ones (Android, iOS and Browser). To develop Android there are some prerequisites for each platform:

- Android and Android Studio
- JDK 8
- Gradle
- Android SDK
- Environment Variables — JAVA_HOME and ANDROID_SDK_ROOT

Here is the link at documentation for the installation's guide:
https://cordova.apache.org/docs/en/10.x/guide/platforms/android/index.html

After installing everything, we move to the 'mobile' folder by typing "cd .\mobile\" on the terminal and actually add the Android platform to the project through the 'cordova platform add android' command.

Here are some plugins that we added to the 'mobile' project with the "cordova plugin add <pluginname>" :

- *cordova-plugin-device*: defines a global device object, which describes the device's hardware and software. To add it run "cordova plugin add cordova-plugin-device"
- *cordova-plugin-dialogs*: provides access to some native dialog UI elements via a global navigator.notification object. To add it run "cordova plugin add cordova-plugin-dialogs"
- *cordova-plugin-file*: implements a File API allowing read/write access to files residing on the device. To add it run "cordova plugin add cordova-plugin-file"
- *cordova-plugin-splashscreen*: displays and hides a splash screen while your web application is launching. To add it run "cordova plugin add cordova-plugin-splashscreen"

For all this plugins (except splashscreen), although the object is in the global scope, it is not available until after the deviceready event.

6.3 Running the emulator

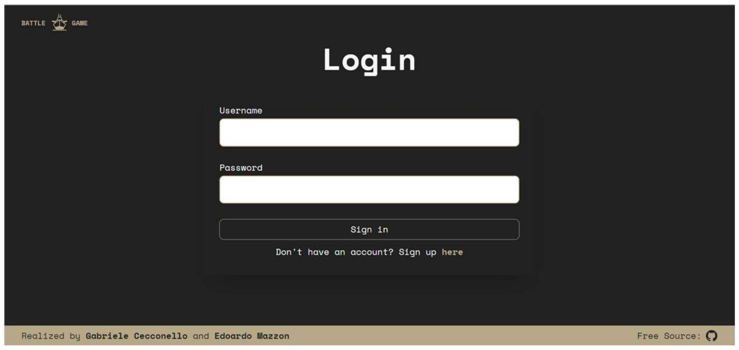
If you changed anything in the client_desktop folder, repeat the changes on client_mobile too.
In the client_mobile folder edit package.json on the "start" script by pasting your own IP address; the same goes for src/environment/environment.ts with the

IP_ADDRESS variable (to obtain your IP address, open a new terminal session and run the ipconfig command).
Open a new terminal session and type “cd .\client_mobile\” then “npm run build”.
Now, in the mobile folder we have all the files ready to run the mobile application. Navigate to the "mobile" folder (cd.. if you are in the previous directory and, in every case, “cd .\mobile\”).
Last step, type cordova run android which will start the emulator automatically.

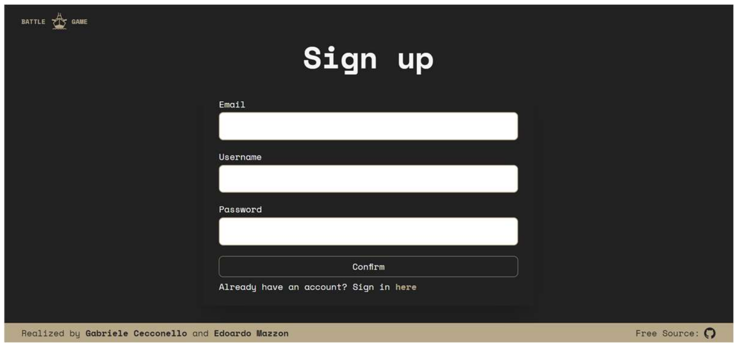
7. Application overview and usage examples

I will start with making an overview of the components for both existing roles in the application, i.e regular users and administrators, and then I will list the specific parts for both in the two sub-chapters. The two things both have in common are login and registration.

Login is made of a form that requires a username and a password to sign in and enter the game. There is also a link to the registration panel.


BATTLE GAME
Login
Username
Password
Sign in
Don't have an account? Sign up here
Realized by Gabriele Cecconello and Edoardo Mazzon
Free Source:

The registration panel contains one more form than the login page, that is the email field. There is also a link to the login panel.


BATTLE GAME
Sign up
Email
Username
Password
Confirm
Already have an account? Sign in here
Realized by Gabriele Cecconello and Edoardo Mazzon
Free Source:

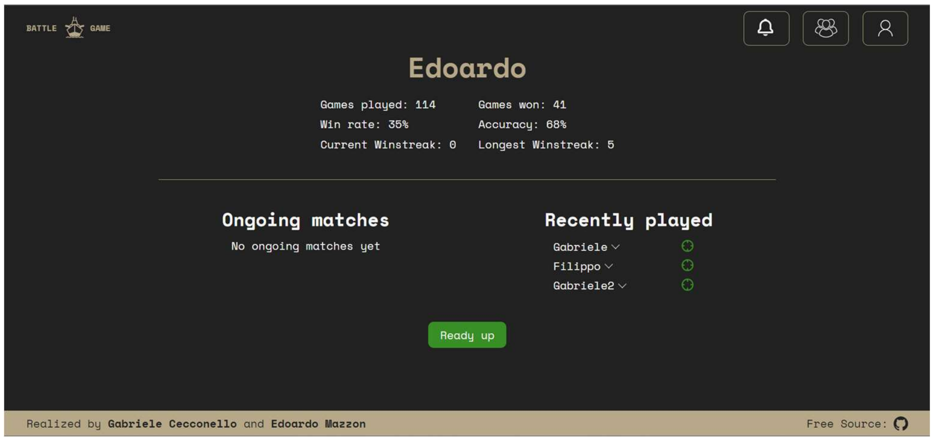
7.1 Regular users

After logging in successfully, the home page shows a panel with a resumé of the user's statistics, a list of the ongoing games of other users and a list of recently played users. If a recently played user is already a friend, the user can invite him to play a game, otherwise a friend request button appears.

There are three buttons in the header:

- The left button is used for seeing all the user's notifications

- The central button allows to see the friends list of the logged user and to search new friends as well as to manage the friend requests
- The right button gives the user the option to logout



Edoardo

Games played: 114 Games won: 41
Win rate: 35% Accuracy: 68%
Current Winstreak: 0 Longest Winstreak: 5

Ongoing matches
No ongoing matches yet

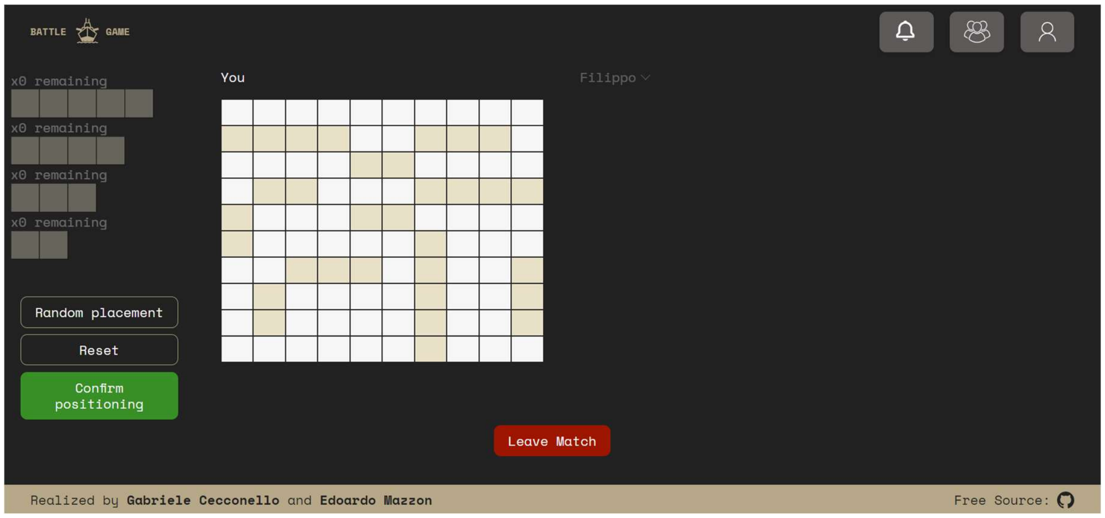
Recently played
Gabriele
Filippo
Gabriele2

Ready up

Realized by **Gabriele Cecconello** and **Edoardo Mazzon** Free Source:

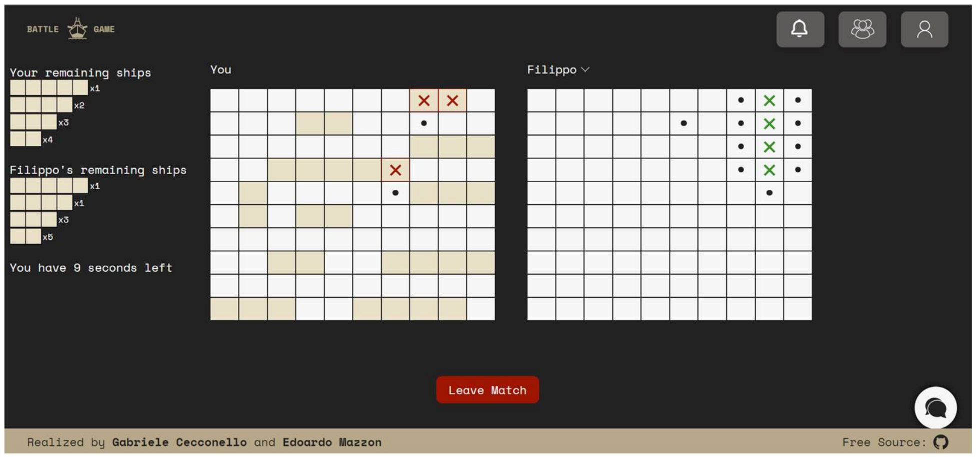
When a user clicks on the "ready up" button, he is put into the waiting queue to find a game. The skill based matchmaking follows this formula: $accuracy + winrate + (current\ winstreak * 10)$, and each user is matched up against another user who has a similar score.

Once the two users are matched, the game starts with the positioning phase, where the player can position his ships arbitrarily or randomly. To rotate the ships, the user must use the scroll wheel over the map, whereas if the user is playing on a mobile device he must use a rotation button that appears above the map. Only after every ship is placed on the field, the player can definitively confirm the positioning, and when both players confirm their ship positionings, the playing phase begins.

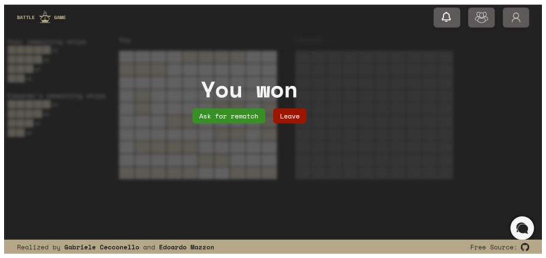


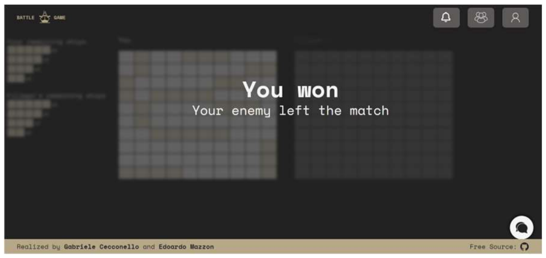
Realized by **Gabriele Cecconello** and **Edoardo Mazzon** Free Source:

The first turn is chosen randomly and each player has 15 seconds to make his move; if the time runs out, the other player wins. If a player hits or sinks an enemy ship, the timer is reset since it is still his turn, but if he misses a shot, the turn goes to the other player. When a player sinks each of the enemy ships, he wins the game and the win rate statistics are updated in the database.

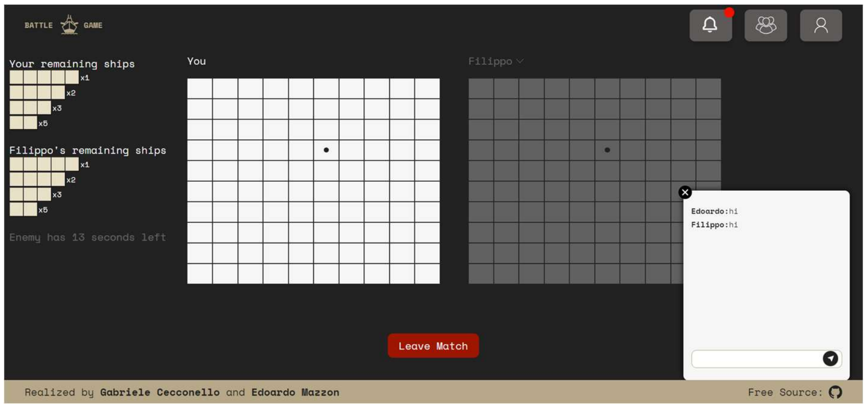


The two players can leave the match whenever they want to, but that means losing the game, unless the game is already over. In fact, when a game ends, two buttons appear: “leave game”, which makes the player leave the match and go back to the home page without paying any consequences, and “ask for rematch” which sends a rematch request to the enemy. The enemy can then accept or ignore the request by leaving the match. When the match is left, both players go back to the home page.



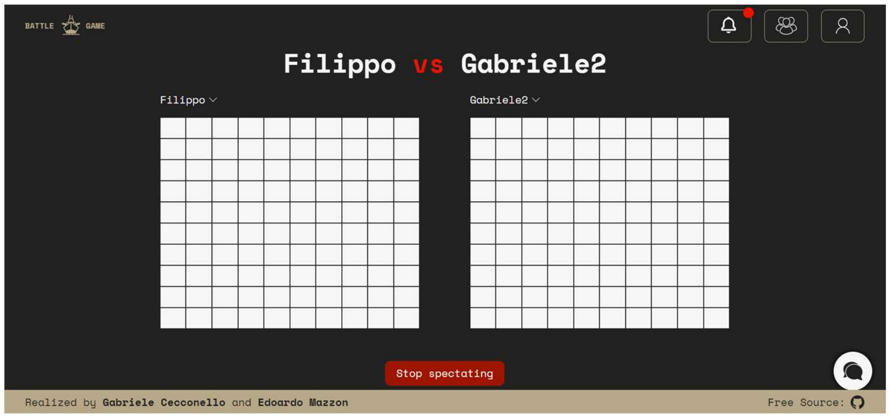


During the match, but not during the positioning phase, the players can text each other through a expandible chat.
There are two types of chats: private chat between two friends and public chat between two players in a match. While the last 25 messages are loaded from the database when a user opens up a chat with a friend, when a game chat is opened, these messages are not loaded, even if the two players are friends, since the spectators could read the private messages.

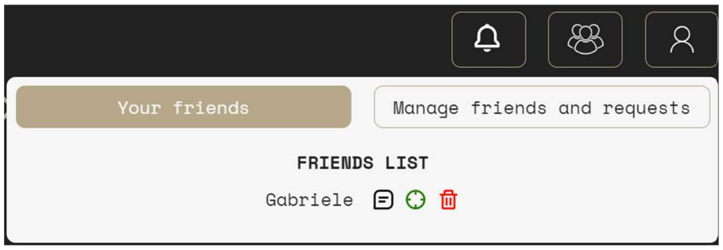

BATTLE GAME
Your remaining ships
x1
x2
x3
x5
Filippo's remaining ships
x1
x2
x3
x5
Enemy has 13 seconds left
You
Filippo
Leave Match
Realized by Gabriele Cecconello and Edoardo Mazzon
Free Source:

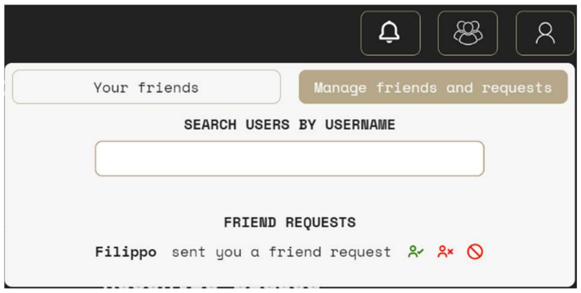
Edoardo:hi
Filippo:hi

A user can in fact spectate other games which are listed in the "ongoing matches" list in the home page. A spectator can use the game chat, but his messages can only be seen by other spectators; moreover, the spectators can chat even during the positioning phase. A spectator can leave the match whenever he wants, and the players are never notified if a spectator enters or leaves the game.



Two users can become friends after one of the two sends a friend request to the other. If a user wants to spam friend requests to another user, only the first one is actually sent while the other ones are ignored.
The second button on the right side of the header opens what we called the "friends section", which is separated into two subsections:

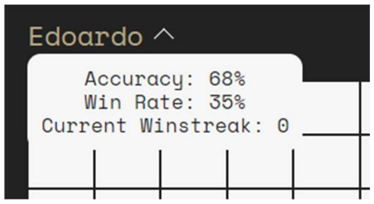




Filippo sent you a friend request

The “your friends” section lists all of the user’s friends and allows him to open a private chat, ask a friend to play a match or remove a friend from the list.
The “manage friends and requests” section shows an input bar where the user can search up other users by their username; then he can send a friend request to the found users or even blacklist them. In this section there is also a friend requests list showing all the pending friend requests, which can be accepted or ignored.
When user 1 sends a game invite to user 2, user 2 receives a notification and a red badge will appear on the notifications icon on the top right corner. If user 2 accepts the invite, but in the meantime user 1 already entered the waiting queue or started another game, a message saying “user 1 is no longer available” will appear to user 2. Otherwise, the game component is loaded and the two start playing.

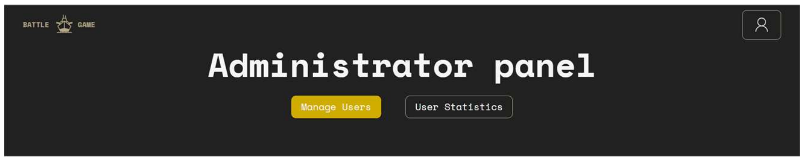
By clicking on a user’s username from the recently played list or on the enemy’s username during a match, that user’s information is shown in a little dialog box:


Edoardo

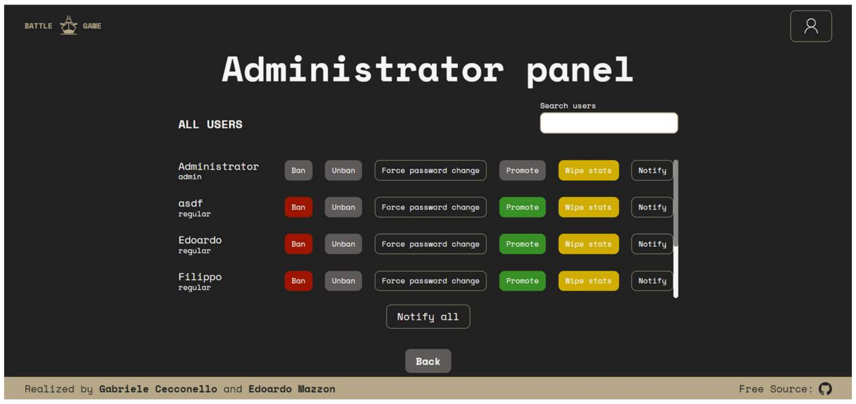
Accuracy: 68%
Win Rate: 35%
Current Winstreak: 0

7.2 Administrator users

In the administrator panel we have an interface that shows two buttons, one for managing the users and one to see all users’ statistics.

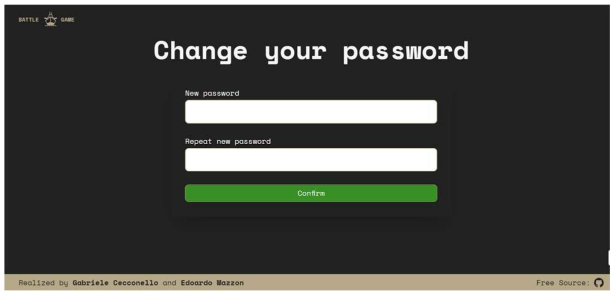


If we choose ‘Manage users’ button, we will be presented a list of all users in which it is possible to do some administrative operations such as banning and unbanning a user, forcing a password change, promoting a user to administrator, wiping a user’s stats and notifying a user.



Administrator admin	Ban	Unban	Force password change	Promote	Wipe stats	Notify
asdf regular	Ban	Unban	Force password change	Promote	Wipe stats	Notify
Edoardo regular	Ban	Unban	Force password change	Promote	Wipe stats	Notify
Filippo regular	Ban	Unban	Force password change	Promote	Wipe stats	Notify

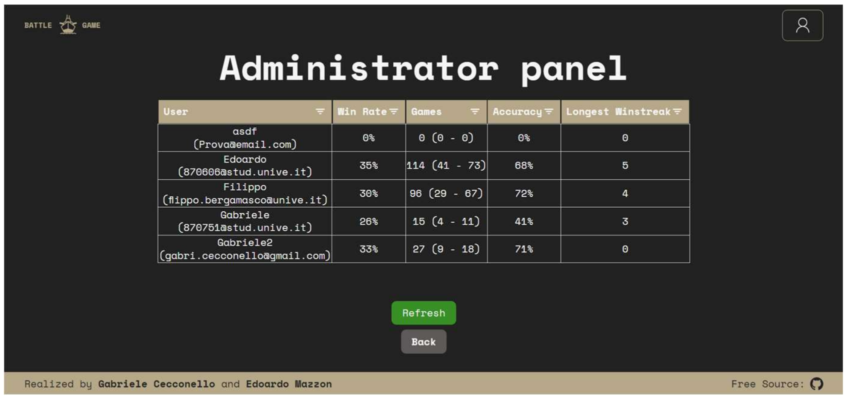
If a user is banned, he will be notified the next time he signs in; only an administrator can unban a banned user. If a moderator makes a user's password expire or promotes him to moderator, the next time that user signs in he will be shown a form to input the new password:



Moreover, there is also an input bar, to simplify searching a user, a "notify all" users button and "back" button to return to the initial administrator dashboard.

If we click the 'User statistics' button, the page will show a table, or a list in the mobile devices, that contains some game statistics of all the users, like the win rate, games played and relative results, the accuracy and the longest win streak.

At the bottom of page, there is a "refresh" button for reloading the statistics and a "back" button to return to the initial administrator dashboard.



Administrator panel

User	Win Rate	Games	Accuracy	Longest Winstreak
asdf (Prova@email.com)	0%	0 (0 - 0)	0%	0
Edoardo (870606@stud.unive.it)	35%	114 (41 - 73)	68%	5
Filippo (flippo.bergamasco@unive.it)	30%	96 (29 - 67)	72%	4
Gabriele (870751@stud.unive.it)	26%	15 (4 - 11)	41%	3
Gabriele2 (gabri.cecconello@gmail.com)	33%	27 (9 - 18)	71%	0

Realized by **Gabriele Cecconello** and **Edoardo Mazzon**

Free Source: