

Tecnologie e Applicazioni Web Project

Suka Ardi 879439

ARCHITECTURE

the aim of the project was to create a web application that allows two users located on two different computers to play battleship and their communication both during gameplay and offline. The application is basically divided into two parts: the frontend where the graphical interface is present for the user, temporary data are stored in order to allow the management of the user, the chat and the game session, and the backend where persistent data such as user information, offline chats are processed via MongoDB DBMS and games between players are managed. The Android application is the copy of the desktop frontend created via Apache Cordova and therefore has the same functionality.

We also use Socket IO to send friend requests, game requests, chat during the game and the game itself, then send the moves and boards.

DATA MODEL

As for the data model, we have divided it into 3 collections: match, that is the history of the games, Users, with the main information but also their statistics, and message, that is the messages exchanged between the players.

USER

```
const UserSchema = new Schema({
  name: {type: String, default: ''},
  username: {type: String, default: ''},
  email: {type: String, default: ''},
  password: {type: String, default: ''},
  wins: {type: Number, default: 0},
  loses: {type: Number, default: 0},
  matches: {type: Number, default: 0},
  isModerator: {type: Boolean, default: false},
  isFirstLogin: {type: Boolean, default: false},
  friends: {type: [Schema.Types.ObjectId], default: []}
});
```

```
const MessageSchema = new Schema({
  from: {type: String, default: ''},
  to: {type: String, default: ''},
  message: {type: String, default: ''},
  timestamp: {type: String, default: ''},
  new: {type: Boolean, default: true},
});
```

MESSAGE

```
const MatchSchema = new Schema({
  player1: {type: String, default: ''},
  player2: {type: String, default: ''},
  score1: {type: String, default: ''},
  score2: {type: String, default: ''},
  winner: {type: String, default: ''},
});
```

MATCH

REST APIs

```
app.post("/signup", async (req: Request, res: Response) => {
  const {name, username, email, password} = req.body;

  const user = await User.exists({username});

  if (!user) {
    await new User({
      name,
      username,
      email,
      password: hashed(password),
    }).save();

    console.log("user has successfully signed up");
    res.status(200).json({message: "ok"});
  } else {
    console.log("error signing up user");
    res.status(400).json({message: "user exists"});
  }
});
```

SIGNUP

this endpoint receives as parameters: name, username, email and password. Returns "ok" as an answer if the user to be registered is not present in our database, otherwise an error message.

```
app.post("/signin", (req: Request, res: Response) => {
  const {username, password} = req.body;

  User.findOne({username: username, password: hashed(password)})
    .then(async (user: UserT) => {
      if (user) {
        const token = jwt.sign({username: username, role: user.role}, secret);
        const hasMessages = await hasNewMessages(username);
        res.status(200).json({token, user, newMessages: hasMessages});
      } else {
        res.status(400).json({"error": "invalid credentials"});
      }
    }).catch((error: CallbackError) => res.status(500).json({error}));
});
```

SIGNIN

this endpoint receives as parameters: username and password. It provides as a response the generated token, its information and if it has new messages from the

chats if the given credentials are correct otherwise an error message.

```
app.post("/addFriends", mustAuth, async (req: Request, res: Response) => {
  const {username, friend} = req.body;

  const usernameId = await getUserId(username);
  const friendId = await getUserId(friend);

  User.updateOne({_id: usernameId}, {$push: {friends: friendId}});
  User.updateOne({_id: friendId}, {$push: {friends: usernameId}});

  res.status(200).json({message: "ok"});
});
```

ADDFRIENDS

this endpoint receives as parameters: the username and the friend you want to add. Returns "ok" and updates the list of friends in our database.

```
app.get("/allUsers", mustAuth, (req: Request, res: Response) => {
  res.status(200).json({users});
});
```

ALLUSERS

this endpoint receives no parameters, it is of type GET. Send as a response

the list of users who are currently online on the server.

```
app.post("/getAllUsers", mustAuth, async (req: Request, res: Response) => {
  const {moderator} = req.body;

  if (await isModerator(moderator)) {
    User.find({isModerator: false})
      .then((users: UserT[]) => res.status(200).json({users}))
      .catch((error: CallbackError) => res.status(500).json({error}));
  } else {
    res.status(401).json({message: "unauthorized"});
  }
});
```

GETALLUSERS

this endpoint takes a moderator's username as a parameter and checks that it is. If so, send the list of users otherwise send an error message.

```
app.get("/getModerators", mustAuth, (req: Request, res: Response) => {
    User.find({isModerator: true}, function (err, sub) {
        if (sub) {
            res.status(200).json({sub});
        } else {
            res.status(400).json({message: "We don't have any moderator!"});
        }
    });
});
```

GETMODERATORS

this endpoint receives no parameters, it is of type GET. Provide all moderators present in the database if there are any otherwise send an error message.

```
app.post("/friend", mustAuth, (req: Request, res: Response) => {
    const {username} = req.body;
    User.find({username: username}, function (err, sub) {
        User.find({_id: sub[0].friends}, function (err, sub) {
            if (sub[0]) {
                res.status(200).json(sub);
            } else {
                res.status(400).json({message: "Amici non trovati!"});
            }
        });
    });
});
```

FRIEND

this endpoint receives a player's username and returns its friends list if there are any otherwise it sends an error message.

```
app.post("/deleteFriend", mustAuth, (req: Request, res: Response) => {
    const {username, friend} = req.body;
    User.findOne({username: username}, function (err, sub) {
        const friends = sub.friends;
        const idFriend = sub._id;
        User.findOne({username: friend}, function (err, sub) {
            console.log(sub);
            const i = 0;
            if (sub) {
                User.updateOne({username: username}, {$pull: {friends: sub._id}}, function (err, res) {
                    console.log("Utente aggiornato");
                });
                User.updateOne({username: friend}, {$pull: {friends: idFriend}}, function (err, res) {
                    console.log("Utente aggiornato");
                });
                res.status(200).json({message: "ok"});
            } else {
                res.status(400).json({message: "error"});
            }
        });
    });
});
```

DELETEFRIEND

this endpoint receives the username of a player and the friend that you want to delete from your list and if it finds it in our database it deletes it and updates both lists otherwise it

sends an error message.

```
app.post("/logout", mustAuth, (req: Request, res: Response) => {
  const {username} = req.body;

  console.log(`deleting user session for ${username}`);

  delete users[username];

  io.emit("updatePlayers", users);
  res.status(200).json({message: "ok"});
});
```

LOGOUT
this endpoint receives the username of a player who has logged out and then updates the array of users who are online on the server.

```
app.post("/matchId", mustAuth, (req: Request, res: Response) => {
  const {id} = req.body;
  if(matches.filter(value => value.id == id)[0]){
    res.json({match: matches.filter(value => value.id == id)[0]});
  }else{
    res.status(400).json({message: "error"});
  }
});
```

MATCHID
this endpoint receives the id of a game and sends the information of the game if it is in progress otherwise it

sends an error message.

```
app.post("/firstLogin", mustAuth, (req: Request, res: Response) => {
  const {username, password, name, email} = req.body;
  User.updateOne({username: username}, {
    $set: {
      name: name,
      email: email,
      password: hashed(password),
      isFirstLogin: false
    }
  }).then(() => res.status(200).json({message: "ok"}))
  .catch((error: CallbackError) => res.status(500).json({error}));
});
```

FIRSTLOGIN

this endpoint receives the following parameters: username, password, name and email of the moderator who makes the first access in order to update his data. In case of an error in the update, it sends an error

message.

```

app.post("/deleteUser", mustAuth, (req: Request, res: Response) => {
  const {moderator, username} = req.body;
  User.findOne({username: moderator}, function (err, sub) {
    if (sub.isModerator) {
      User.deleteOne({username: username}).then(function () {
        Match.find({player1: username}, (err, sub: MatchT[]) => {
          sub.map(m => {
            if (m.winner == username) {
              User.updateOne({username: m.player2}, {$inc: {matches: -1, loses: -1}}, function (err, sub) {
                console.log("UTENTE AGGIORNATA!");
              });
            } else {
              User.updateOne({username: m.player2}, {$inc: {matches: -1, wins: -1}}, function (err, sub) {
                console.log("UTENTE AGGIORNATA!");
              });
            }
          });
        });

        Match.find({player2: username}, function (err, sub: MatchT[]) {
          sub.map(m => {
            if (m.winner == username) {
              User.updateOne({username: m.player1}, {$inc: {matches: -1, loses: -1}}, function (err, sub) {
                console.log("UTENTE AGGIORNATA!");
              });
            } else {
              User.updateOne({username: m.player1}, {$inc: {matches: -1, wins: -1}}, function (err, sub) {
                console.log("UTENTE AGGIORNATA!");
              });
            }
          });
        });

        Match.deleteMany({$or: [{player1: username}, {player2: username}]})
        // .then(() => res.status(200).json({message: "ok"}))
        .catch((error: CallbackError) => res.status(500).json({error}));

        User.find({isModerator: false})
          .then((users: UserT[]) => res.status(200).json({users}))
          .catch((error: CallbackError) => res.status(500).json({error}));

      }).catch((error: any) => {
        res.status(400).json({message: error});
      });
    } else {
      res.status(400).json({message: "Non sei il moderatore!"});
    }
  });
});

```

DELETEUSER

this endpoint receives the following parameters: the moderator and the username that you want to delete. First of all verify that it is a moderator and then that the user exists in the database. Then he eliminates him, also eliminating all the matches played and updating all the statistics of the other players who played against him. Send a confirmation message in case of success otherwise send an error message.

```
app.post("/addModeator", mustAuth, async (req: Request, res: Response) => {
  const {moderator, username, password} = req.body;

  if (await isModerator(moderator)) {
    await new User({
      username,
      password: hashed(password),
      isModerator: true,
      isFirstLogin: true
    }).save();

    res.status(200).json({message: "ok"});
  } else {
    res.status(401).json({message: "not authorized"});
  }
});
```

ADDMODERATOR
this endpoint receives the following parameters: the moderator and the new moderator you want to add. If the user is a moderator he is added otherwise it sends an error message.

```
app.get("/matches", mustAuth, (req: Request, res: Response) => {
  res.status(200).json({matches});
});
```

MATCHES
this endpoint receives no parameters, it is of type GET.

Send all games in progress to the user.

```
app.post("/history", mustAuth, (req: Request, res: Response) => {
  const {username} = req.body;

  Match.find({$or: [{player1: username}, {player2: username}]})
    .then((matches: MatchT[]) => res.status(200).json({matches}))
    .catch((error: CallbackError) => res.status(500).json({error}));
});
```

HISTORY
this endpoint receives a user's username and sends all the games it has made. Otherwise it

sends an error message.

```
app.post("/chat", mustAuth, async (req: Request, res: Response) => {
  const {username, friend, msg} = req.body;
  const time = new Date().toString();

  await new Message({
    from: username,
    to: friend,
    message: msg,
    timestamp: time
  }).save();

  io.to(users[friend])
    .emit("privateMessage", {
      username: username,
      msg: msg,
      timestamp: time
    });

  res.status(200).json({message: "ok"});
});
```

CHAT
this endpoint receives the username of a user, the friend and the message you want to send. Then create the new message by adding the timestamp and send a confirmation message.

```

app.post("/getChat", mustAuth, async (req: Request, res: Response) => {
  const {username, friend} = req.body;

  Message.find({
    $or: [{from: username, to: friend}, {
      from: friend,
      to: username
    }]
  }).then(async (messages: MessageT[]) => {
    await Message.updateMany({to: username}, {$set: {new: false}});
    res.status(200).json({messages});
  }).catch((error: CallbackError) => res.status(500).json({error}));
});

```

GETCHAT

this endpoint receives the username of a user and friend and returns all the messages that have been exchanged. Otherwise it sends an error message.

```

// TODO Remove this endpoint and keep message read logic only in /getchat
app.post("/readChat", mustAuth, async (req: Request, res: Response) => {
  const {username} = req.body;

  Message.updateMany({to: username}, {$set: {new: false}})
    .then(() => res.status(200).json({message: "ok"}))
    .catch((error: CallbackError) => res.status(500).json({error}));
});

```

READCHAT

this endpoint receives the username of a user and reports all messages as read as it has downloaded them and therefore viewed

them. Otherwise it sends an error message.

AUTHENTICATION

Authentication is handled as follows: at login time, if the credentials are correct, a JWT token is created using the "JSONWebToken" library and the sign function. This is then sent to the user and saved in the frontend in the sessionStorage and then maintained throughout the session. This token then allows you to access all the endpoints that have in the "mustAuth" declaration. In fact, this is a function that checks with the function "verify", every time these endpoints are accessed,

```
const mustAuth = (req: Request, res: Response, next: NextFunction) => {
  const header = req.headers.authorization;

  if (header) {
    const [, token] = header.split(" ");
    jwt.verify(token, secret, (err, user) => {
      if (err) {
        res.sendStatus(403);
        return;
      }

      req.body.username = user.username;
      next();
    });
  } else {
    res.sendStatus(401);
  }
};
```

the token and consequently allows access to the various resources.

```
private static getOptions(token: string) {
  return {
    headers: new HttpHeaders({
      authorization: "Bearer " + token,
      "cache-control": "no-cache",
      "Content-Type": "application/json"
    })
  };
}
```

In the frontend in the HTTP request to one of these endpoints contains the following Header:

ANGULAR FRONTEND

In the frontend we use `sessionStorage` to keep the user data of the game session even when the user refreshes the page. Such as the logged variable which indicates whether a user is logged in.

The application is divided into 11 components:

BattleshipGame: which manages all the information of the game and allows it to run smoothly. Inside we have the **board** service for the more optimized management of boards.

Chat: which manages offline chats with other friends and game moderators.

confirmation-dialog: which is used to show friend or game requests.

Friends: which allows the management of friends by sending the friend request, sending the game request and deleting a friend from your list. It also allows you to view all users who are online.

game: where a user can decide whether to play or watch a game.

Home: where the historical information is managed and the navbar of the other options is present.

login: which manages the sending of login information.

moderator: which manages all the operations that a user can carry out, therefore deleting a user, chatting with other players, looking at everyone's statistics and adding other moderators.

Register: which manages the sending of register information.

watch-game: which allows a player to watch a game in progress.

As for routing, initially you are redirected to the login page if you are not logged in and then if logged = false. Otherwise you are redirected to the home.

```
const routes: Routes = [
  {path: "login", component: LoginComponent},
  {path: "register", component: RegisterComponent},
  {path: "playGame", component: BattleshipGameComponent, canActivate: [ExpenseGuard]},
  {path: "moderator", component: ModeratorComponent, canActivate: [ExpenseGuard]},
  {path: "watchGame", component: WatchGameComponent, canActivate: [ExpenseGuard]},
  {
    path: "home", component: HomeComponent, canActivate: [ExpenseGuard], children: [
      {path: "main", component: MainComponent, canActivate: [ExpenseGuard]},
      {path: "friends", component: FriendsComponent, canActivate: [ExpenseGuard]},
      {path: "game", component: GameComponent, canActivate: [ExpenseGuard]},
      {path: "chat", component: ChatComponent, canActivate: [ExpenseGuard]},
    ]
  },
  {path: "", redirectTo: "/login", pathMatch: "full"},
];
```

```

canActivate(
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot): boolean | UrlTree {
  console.log(AppComponent);
  if (AppComponent.logged && !AppComponent.isModerator) {

    if (state.url == "/login" || state.url == "/moderator" || state.url == "/home") {
      return this.router.parseUrl("/home/main");
    } else {
      return true;
    }

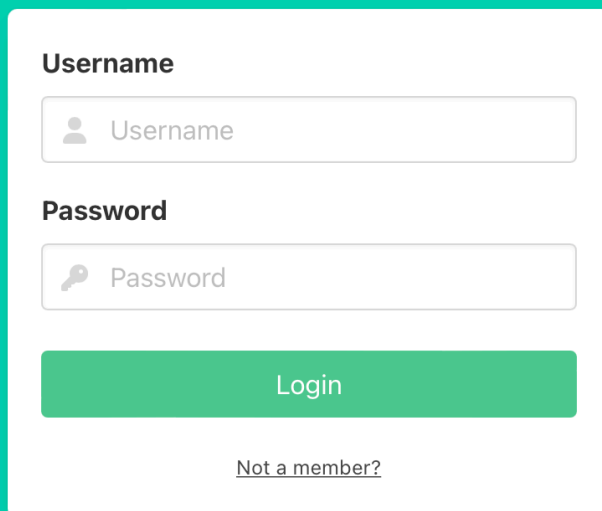
  } else if (AppComponent.logged && AppComponent.isModerator) {
    if (state.url != "/moderator") {
      return this.router.parseUrl("/moderator");
    } else {
      return true;
    }
  } else {
    return this.router.parseUrl("/login");
  }
}

```

All the other components can be accessed only if logged in according to the logic of the guard we have created.

That is, if a user is a moderator he can only access the moderator component while if he is a player he can access all pages except that of the moderator.

APPLICATION WORKFLOW



A login form with a teal background. It contains a 'Username' label above a text input field with a user icon and placeholder text 'Username'. Below it is a 'Password' label above a text input field with a key icon and placeholder text 'Password'. A green 'Login' button is centered below the fields. At the bottom, there is a link that says 'Not a member?'.

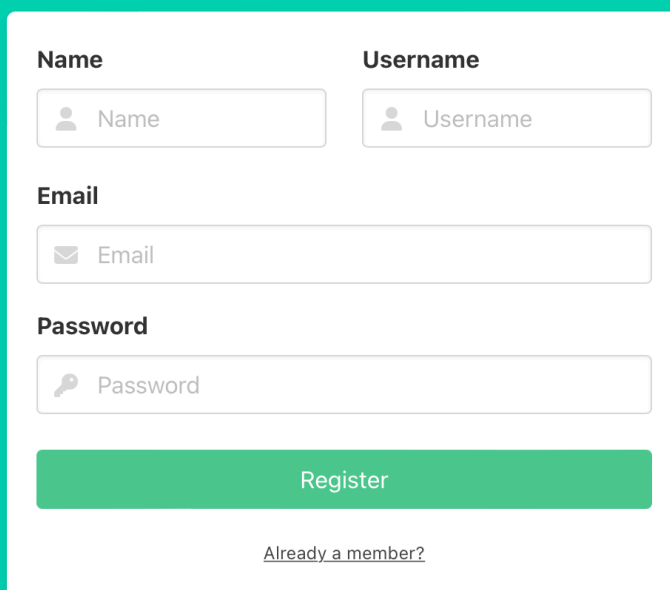
Username

Password

Login

[Not a member?](#)

Initially the user can login, or if he is not present in our database he can click on "Not a member?" and register through our forms by entering your data.



A registration form with a teal background. It has two columns: 'Name' and 'Username', each with a text input field and a user icon. Below these is an 'Email' label above a text input field with an envelope icon. Then a 'Password' label above a text input field with a key icon. A green 'Register' button is centered below the fields. At the bottom, there is a link that says 'Already a member?'.

Name

Username




Email

Password

Register

[Already a member?](#)

The home page will display the history of his games and a navbar where he can select the operations he wants to carry out. Or log out via the button at the top right.

 Friends  Game  Chat




Logout

Hello filippo

Historical matches

francesco : 32 filippo : 0

In the friends section, you can send friend and game requests and see who is online. He can also see the stats of his friends.

 Friends  Game  Chat

Logout

Friends

Online friends



Username	Send match request	Remove from friends
----------	--------------------	---------------------

Offline friends

Username	Send match request	Remove from friends
----------	--------------------	---------------------

Online players

All players currently online

Username	Send match request	Send friend request
filippo		

Statistic of players


Battleship

2 player(s) in game

Battleship


Send

In the chat section you can chat with your friends or moderators whether they are online or not. On the other hand, as far as the operator is concerned, he can carry out the operations I have previously described.


 Home

Logout

Username

 Username

Password

 Password

Add moderator

Remove players

Remove players from the app

filippo

x

francesco

x

Statistic of players

Displays player statistics

Chat with players

Chat with other players

Type here

Send

