

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: []}
});
```

MESSAGE

```
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},
});
```

MATCH

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

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 (POST)

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 (POST)

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.put("/friend/:friend", mustAuth, async (req: Request, res: Response) => {
  const {username} = req.body;
  const {friend} = req.params;
  getUserId(username).then((usernameId)=>{
    getUserId(friend).then((friendId)=>{
      User.updateOne({_id: usernameId}, {$push: {friends: friendId}}).then(()=>{
        User.updateOne({_id: friendId}, {$push: {friends: usernameId}}).then(()=>{
          res.status(200).json({message: "ok"});
        }).catch((error: CallbackError) => res.status(500).json({error}));
      }).catch((error: CallbackError) => res.status(500).json({error}));
    }).catch((error: CallbackError) => res.status(500).json({error}));
  }).catch((error: CallbackError) => res.status(500).json({error}));
});
```

FRIEND/:friend (PUT)

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. Or an error if one of the operations has not been

completed.

```

app.get("/users", mustAuth, async (req: Request, res: Response) => {
  const username = req.body.username;
  switch (req.query.role) {
    case "admin user":
    case "user admin":
      if (await isModerator(username)) {
        User.find({username: {$ne: username}})
          .then((users: UserType[]) => res.status(200).json({users: viewUser(users)}))
          .catch((error: CallbackError) => res.status(500).json({error}));
      } else {
        res.status(401).json({error: "unauthorized"});
      }
      break;
    case "admin":
      User.find({isModerator: true, username: {$ne: username}})
        .then((users: UserType[]) => res.status(200).json({users: viewUser(users)}))
        .catch((error: CallbackError) => res.status(500).json({error}));
      break;
    case "user":
      User.find({isModerator: false, username: {$ne: username}})
        .then((users: UserType[]) => res.status(200).json({users: viewUser(users)}))
        .catch((error: CallbackError) => res.status(500).json({error}));
      break;
    default:
      res.status(400).json({error: `role ${req.body.role} not supported`});
      break;
  }
});

```

USERS(GET)

this endpoint receives no parameters, it is of type GET. Send as a response the list of all users of a given role.

```

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

  User.findOne({username})
    .then((user: UserType) => {
      User.find({_id: {$in: user.friends}})
        .then((users: UserType[]) => res.status(200).json({users: viewUser(users)}))
        .catch((error: CallbackError) => res.status(500).json({error}));
    })
    .catch((error: CallbackError) => res.status(500).json({error}));
});

```

FRIEND (GET)

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

```

app.delete("/friend/:friend", mustAuth, async (req: Request, res: Response) => {
  const {username} = req.body;
  const {friend} = req.params;
  User.findOne({username: username})
    .then(async (user: UserType) => {
      const idFriend = user._id;
      User.findOne({username: friend})
        .then(async (friend: UserType) => {
          if (friend) {
            User.updateOne({username: username}, {$pull: {friends: friend._id}}).then(()=>{
              User.updateOne({username: friend.username}, {$pull: {friends: idFriend}}).then(()=>{
                res.status(200).json({message: "ok"});
              }).catch((error: CallbackError) => res.status(500).json({error}));
            }).catch((error: CallbackError) => res.status(500).json({error}));
          } else {
            res.status(500).json({message: "not found"});
          }
        })
        .catch((error: CallbackError) => res.status(500).json({error}));
    })
    .catch((error: CallbackError) => res.status(500).json({error}));
  io.to(users[friend]).emit("friendRemoved", {friend: username});
});

```

FRIEND/:friend (DELETE)

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 (POST)
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 (POST)
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 (POST)
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.delete("/user/:username", mustAuth, mustBeAdmin, (req: Request, res: Response) => {
  const {username} = req.params;

  User.deleteOne({username: username}).then((data) => {
    Match.find({player1: username}).then((matches: MatchType[]) => {
      matches.forEach((m: MatchType) => {
        User.updateOne({username: m.player2}, {
          $inc: {
            matches: -1,
            wins: m.winner == username ? 0 : -1,
            loses: m.winner == username ? -1 : 0,
          }
        }).catch((error: CallbackError) => res.status(500).json({error}));
      });
    }).catch((error: CallbackError) => res.status(500).json({error}));

    Match.find({player2: username}).then((matches: MatchType[]) => {
      matches.forEach((m: MatchType) => {
        User.updateOne({username: m.player1}, {
          $inc: {
            matches: -1,
            wins: m.winner == username ? 0 : -1,
            loses: m.winner.username == username ? -1 : 0,
          }
        }).catch((error: CallbackError) => res.status(500).json({error}));
      });
    }).catch((error: CallbackError) => res.status(500).json({error}));

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

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

```

USER/:username (DELETE)

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.put("/moderator", mustAuth, mustBeAdmin, async (req: Request, res: Response) => {
  const {username, password} = req.body;

  await new User({
    username,
    password: hashed(password),
    isModerator: true,
    isFirstLogin: true
  }).save().then(()=>{
    res.status(200).json({message: "ok"});
  }).catch((error: CallbackError) => res.status(500).json({error}));
});

```

MODERATOR (PUT)

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("/moderators", mustAuth, (req: Request, res: Response) => {
  User.find({isModerator:true}).then((data)=>{
    res.status(200).json({data});
  }).catch((error: CallbackError) => res.status(500).json({error}));
});

```

MODERATORS (GET)

this endpoint return a list of all Moderators otherwise it sends an error message.

```

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

```

MATCHES (GET)

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

(POST)

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();

  new Message({
    from: username,
    to: friend,
    message: msg,
    timestamp: time
  }).save().then(()=>{
    res.status(200).json({message: "ok"});
  }).catch((error: CallbackError) => res.status(500).json({error}));

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

CHAT (POST)

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

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

CHATS(POST)

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 (POST)

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, the token and consequently allows access to the various resources.

```
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);
  }
};
```

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

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

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:

```
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"},
];
```

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.

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

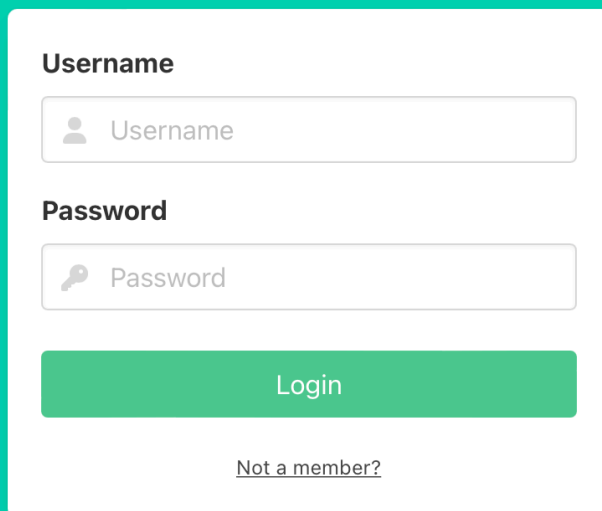
```
canActivate(
  route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot): boolean | UrlTree {
  console.log(AppComponent);
  if (AppComponent.logged && !AppComponent.isModerator) {

    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");
  }
}
```

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 the placeholder text 'Username'. Below this is a 'Password' label above a text input field with a key icon and the placeholder text 'Password'. A green 'Login' button is centered below the fields. At the bottom, there is a link that says 'Not a member?'.

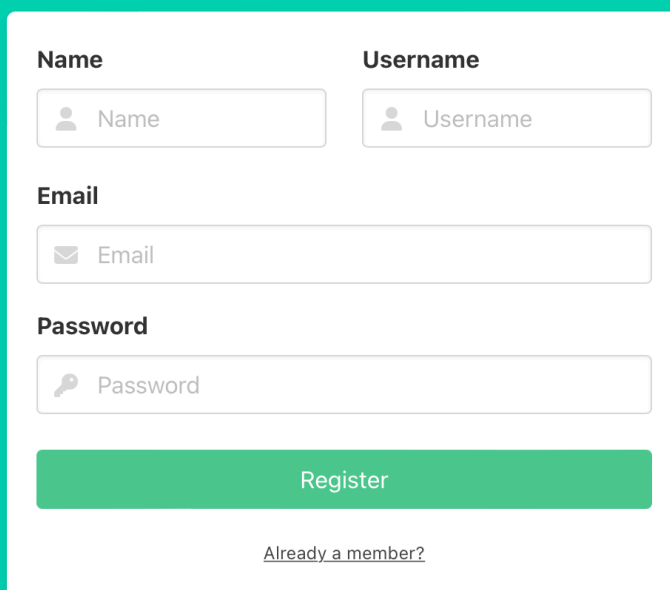
Username

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 features 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 and the placeholder text 'Email'. This is followed by a 'Password' label above a text input field with a key icon and the placeholder text 'Password'. A green 'Register' button is centered below the fields. At the bottom, there is a link that says 'Already a member?'.

Name

Username

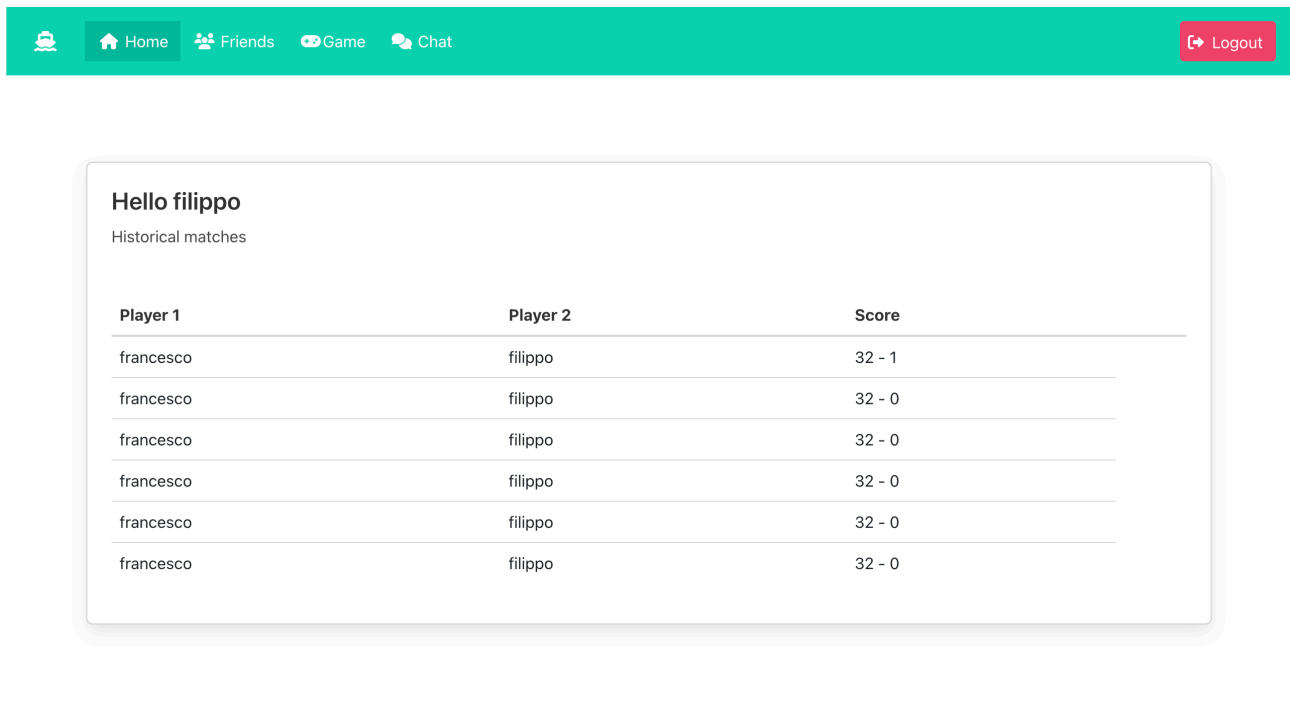
Email

Password

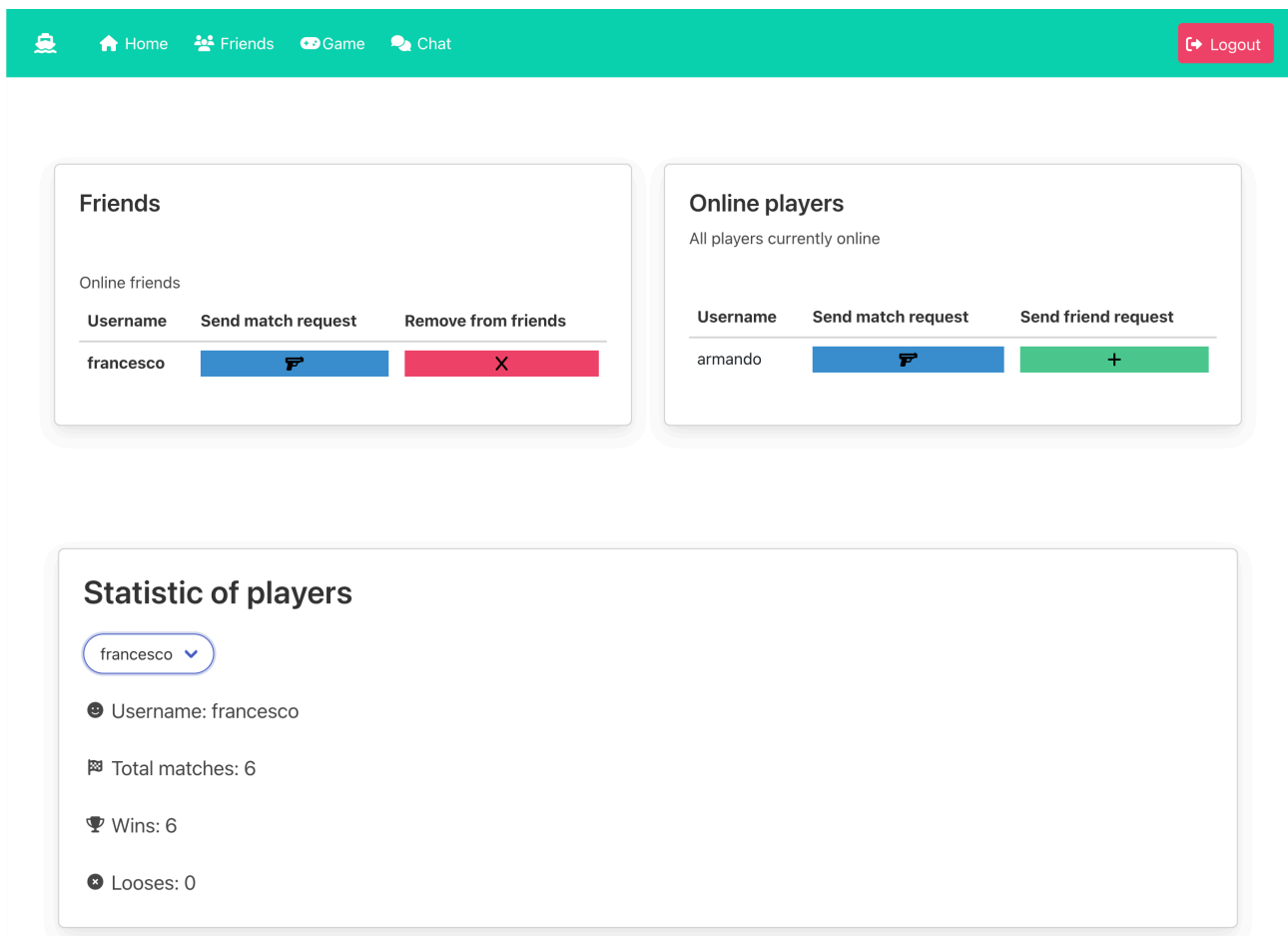
Register

[Already a member?](#)

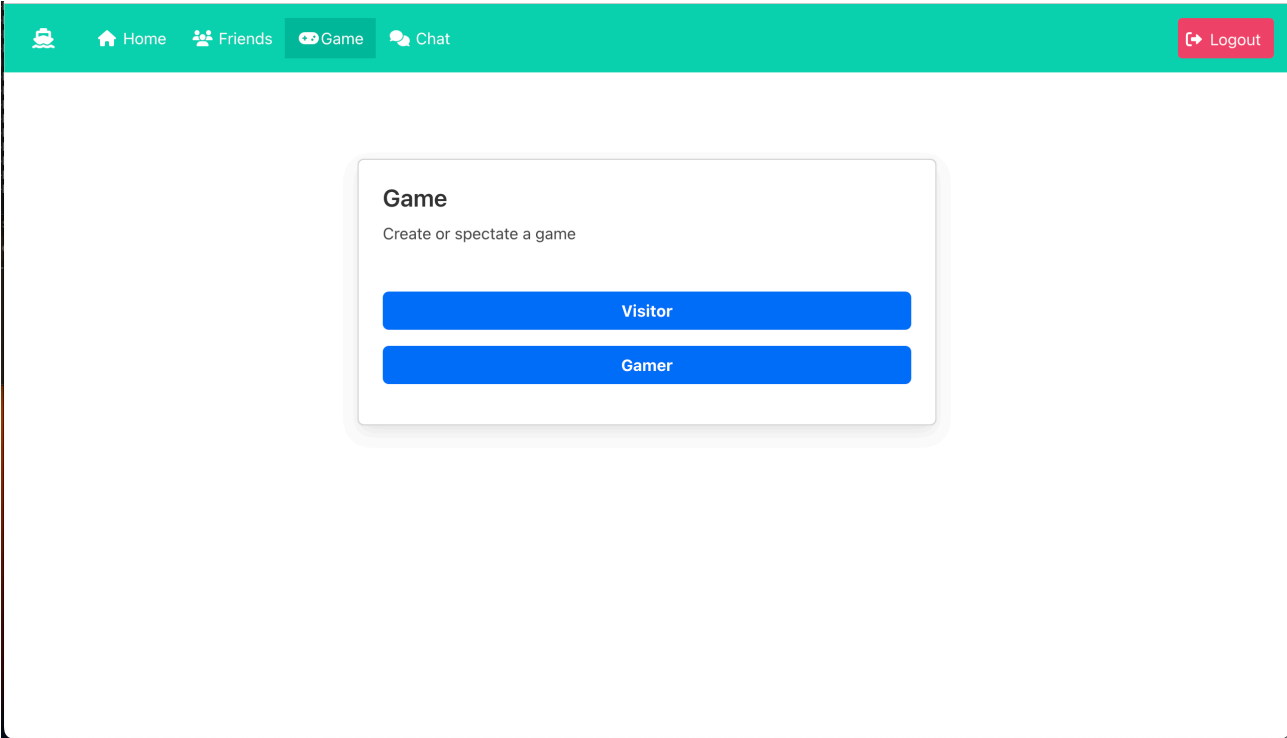
The home page will display the history of his games and a navbar where he can select the operations he wants to carry out. Or log out via the button at the top right.



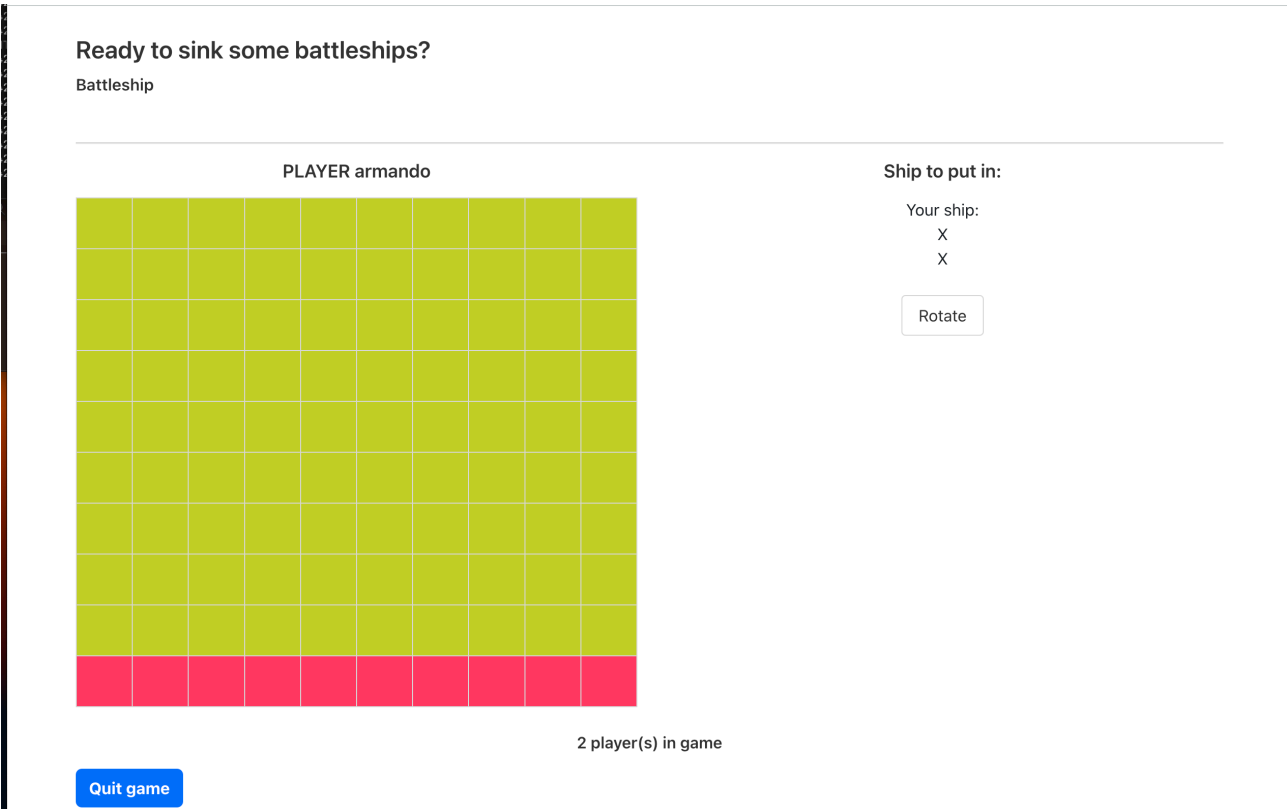
In the friends section, you can send friend and game requests and see who is online. He can also see the stats of his friends.



In game section you can decide whether to play or watch a game.



In game mode, you can decide whether to build the board himself or have the computer do it and then send it to the server. After that the game begins.



During the game you can chat with the other player.

Ready to sink some battleships?

Battleship

PLAYER armando You **SCORE: 0**

Send

In the chat section you can chat with your friends or moderators whether they are online or not.

The screenshot shows a web application interface with a teal header bar. The header contains navigation icons for Home, Friends, Game, and Chat, along with a Logout button. Below the header, there are two chat sections. The left section is titled 'Chat with your Friends' and the right section is titled 'Chat with moderators'. Both sections have a dropdown menu at the top, a large text input area in the middle, and a 'Text input' field with a 'Send' button at the bottom.

On the other hand, as far as the admin is concerned, he can carry out the operations I have previously described.

The screenshot shows an admin interface with a teal header bar containing Home and Logout buttons. Below the header, there is a login form with fields for Username and Password, and an 'Add moderator' button. Below the login form, there are three management sections. The first section is titled 'Remove players' and contains a list of players with 'filippo' and 'francesco' highlighted in red. The second section is titled 'Statistic of players' and contains a dropdown menu. The third section is titled 'Chat with players' and contains a dropdown menu and a 'Type here' field with a 'Send' button.

So He can add new moderators with the form on the top. He can also remove players from the system, he can see all the statistics of all players and chat with them.