# Tecnologie e Applicazioni Web - BattleShip Project

Zane Filippo – 880119

## Introduction

The BattleShip project is a web-based application developed using Node.js, Angular and Docker. The project utilizes a REST API to handle all communication between the frontend and the backend, allowing for seamless interation between the user and the application. The primary objective of the project is to create a digital version of the classic board game BattleShip, where users can play against each other in real-time over the internet.

The use of Docker allows for easy deployment and scalability of the application, while Angular provides a user-friendly interface and Node.js allows for efficient server-side processing.

In addition, the project ha salso been translated into a mobile application using Apache Cordova. Cordova is an open-source platform that allows for the development of mobile apps using web technologies such as HTML, CSS and JavaScript. The use of Cordova also enables the project to be easily ported to different mobile platforms, such as iOS and Android.


## Architecture

The application is built using the MEAN stack, which stands for MongoDB, Express.js, Angular and Node.js. MongoDB is used as the database to store user and game data, while Express.js is used as the web framework to handle routing and server-side logic. Angular is used as the frontend framework to build the user interface and handle client-side logic. Node.js servers as the runtime enviroment for the application, allowing it to run efficiently on the server.

Docker is used to containerize the application, which allows for easy development and scaling of the application. This means that the application can be deployed and run in various enviroments, without worring about compatibility issues.

**Data Model**

The data model for the BattleShip project is divided into three main entities: users, match and messages.

The user entity represents the players of the game and contains information such as the player's username, e-mail, and password. It also stores information such as the player's friends and authorizations.

```javascript
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 },
  looses: { 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: [] },
})
```

The match entity represents a game of Battleship and contains information such as the players involved in the match, the score and the winner.

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

The message entity represents the communication between the players. It contains information such as the sender, the recipient, and the message content.

```javascript
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 },
})
```

All three entities are stored in MongoDB and can be easily queried and manipulated using Mongoose, which is an Object-Document Mapping (ODM) library for MongoDB and Node.js. This allows for easy management of the data and integration with the rest of the application.

**REST APIs**

The Battleship project implements a set of RESTful APIs that handle communication between the front-end and back-end of the application. These APIs are built using Express.js and are responsible for handling the creation, retrieval, update, and deletion of resources such as users, matches, and messages.

**POST** /signup

```
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' })
  }
})
```

**POST** /signin

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

  User.findOne({ username: username, password: hashed(password) })
    .then(async (user: UserType) => {
      if (user) {
        const token = jwt.sign({ username: username, role: user.role }, secret)
        const hasMessages = await hasNewMessages(username)

        // TODO rename `newMessages` to `hasNewMessages`
        res.status(200).json({ token, user, newMessages: hasMessages })
      } else {
        res.status(400).json({ error: 'invalid credentials' })
      }
    })
    .catch((error: CallbackError) => res.status(500).json({ error }))
})
```

```
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
  }
})
```

**GET** /friend

```typescript
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 }))
})
```

**PUT** /friends/{friend}

```
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 }))
})
```

**DELETE** /friends/{friend}

```
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 })
})
```

**POST** /logout

```
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' })
})
```

**POST** /matchId

```
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' })
  }
})
```

**POST** /firstLogin

```
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 }))
})
```

**DELETE** /user/{username}

```typescript
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,
                    looses: 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,
                    looses: m.winner.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 }))
  }
)
```

**PUT** /moderator

```typescript
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 }))
    }
)
```

GET /matches

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

**GET** /moderators

```
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 }))
})
```

**POST** /history

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

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

**POST** /chat

```
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,
  })
})
```

**POST** /chats

```
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 }))
})
```

**POST** /readChat

```
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 }))
})
```

# Authentication

Authentication is handled using the following code which defines a middleware function callded mustAuth. The function starts by extracting the Authorization header from the HTTP request header, and if exists, it uses the split method to get the token.

The nit uses the jwt.verify() method to verify the token. If there is an error while verifying the token, the middleware sends a response with a status code of 403 (Forbidden) and stops the execution of the current process. If the token is valid, it adds the username property to the request body and calls the next() function.

If there's no Authorization header in the request headers, the middleware sends a response with a status code of 401 (Unauthorized) and a JSON error message "unauthorized".

```
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.status(401).json({ error: 'unauthorized' })
  }
}
```

# Angular Frontend
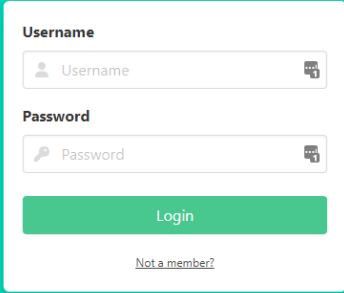
Angular is a front-end JavaScript framework used to build web applications. It provides a set of tools and features that make it easy to create rich, interactive, and dynamic user interfaces. Angular uses a component-based architecture, which allows for the modularization of code and the separation of concerns. Each component is a self-contained unit of functionality that can be easily reused throughout the application.

The project front-end is divided into 11 main components:

- battleship-game: manage information about a game
- chat: manage offline chats
- confirmation-dialog: it is used to show notifications
- friends: friends management
- game: an user can decide to watch or to play a game
- home: navbar and historical information
- login: login page
- moderator: manage all moderator's operations
- register: registration page
- watch-game: allow a player to watch a game
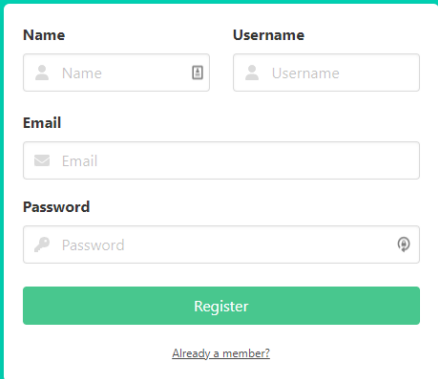
## Application's Workflow

Assuming that this is the first time we visit the application, we are presented with the login page. As we do not have any existing user credentials, we opt to create a new account by clicking on the "Not a member?" link.



After clicking on the "Not a member?" link, we are directed to the registration form where we can enter our personal information. Once we have filled in the required fields, we click the "Register" button. This action redirects us to the login page, where we can now log in to the application using the credentials we just registered.



Once successfully logged in, we are directed to the homepage of the application. Here we can see our game history, including any previous games we have played and their outcomes.
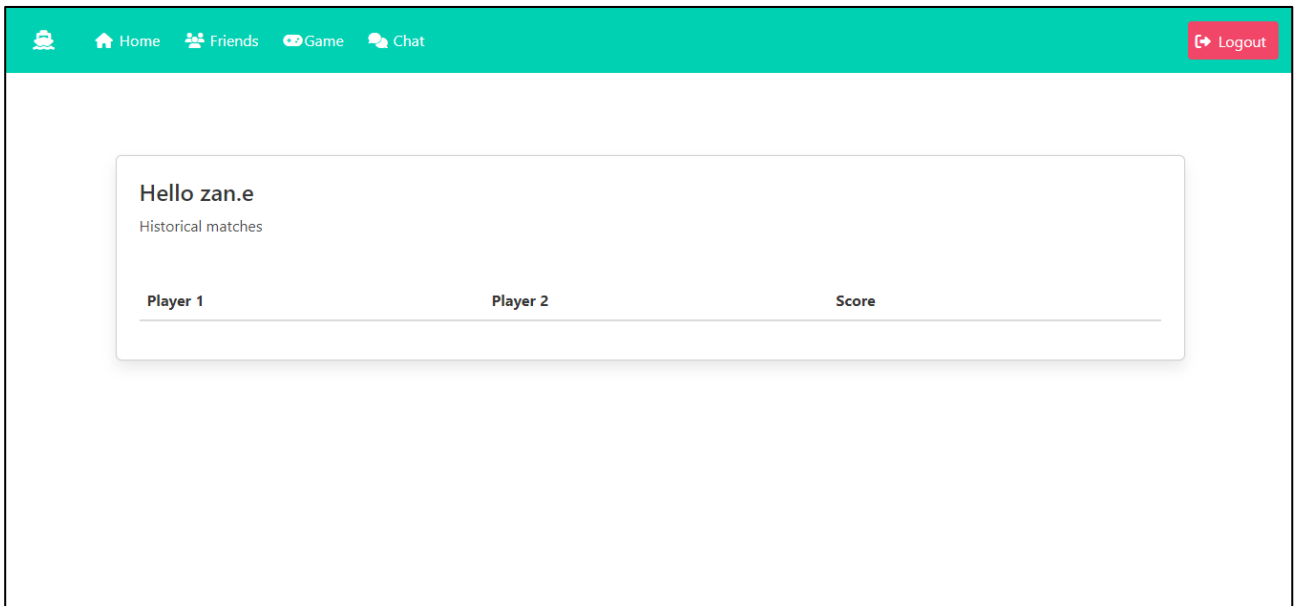
In addition to the game history, the homepage also has a navigation bar located at the top of the screen. The navigation bar includes options such as "Friends", "Game" and "Chat". These options allow the user to navigate to different sections of the application, such as the friends list, game section, and chat section respectively. The user can use these options to interact with friends, play games, and communicate with other players in real-time.

By clicking on the "Friends" option in the navigation bar, the user is taken to the friends section of the application. In this section, the user can view information about their friends and other players who are online. The user can also send match and friend requests to other players they would like to connect with. This section allows the user to interact with other players, find new friends to play with and join matches with them.
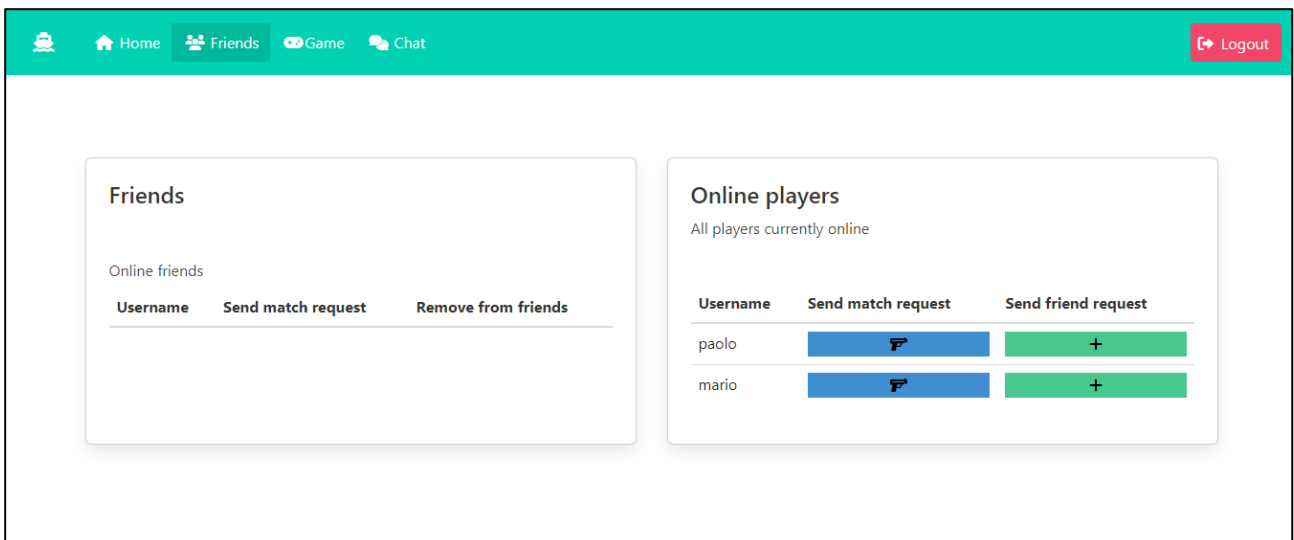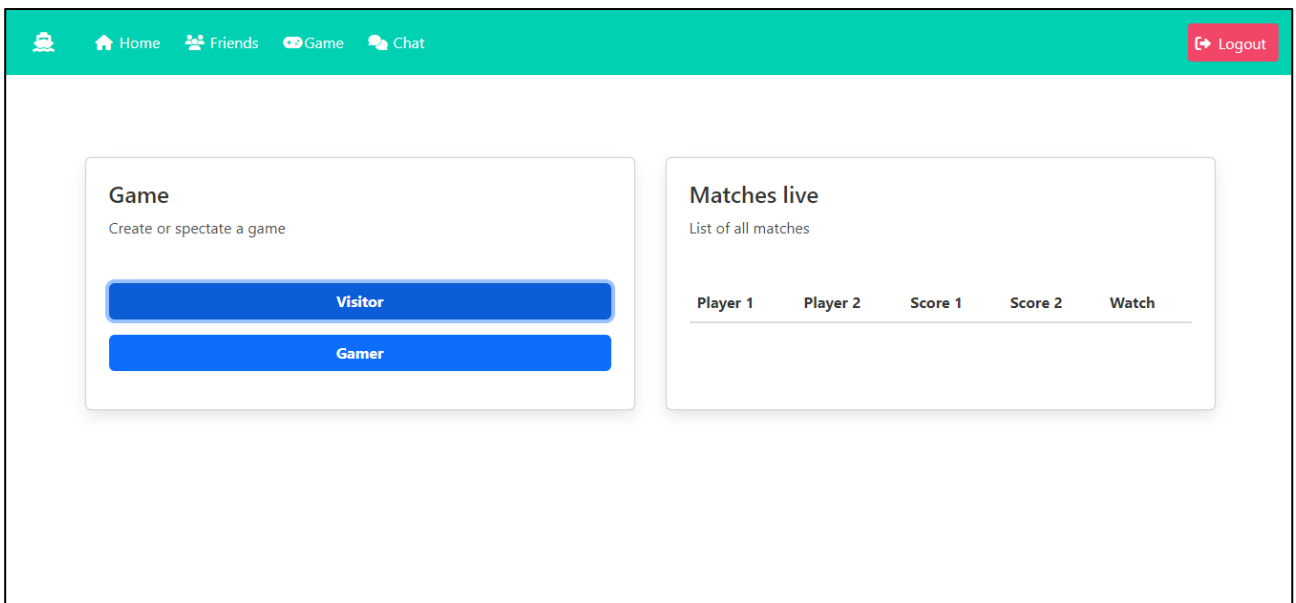


Any user can see which users are online and send them a friend or a match request
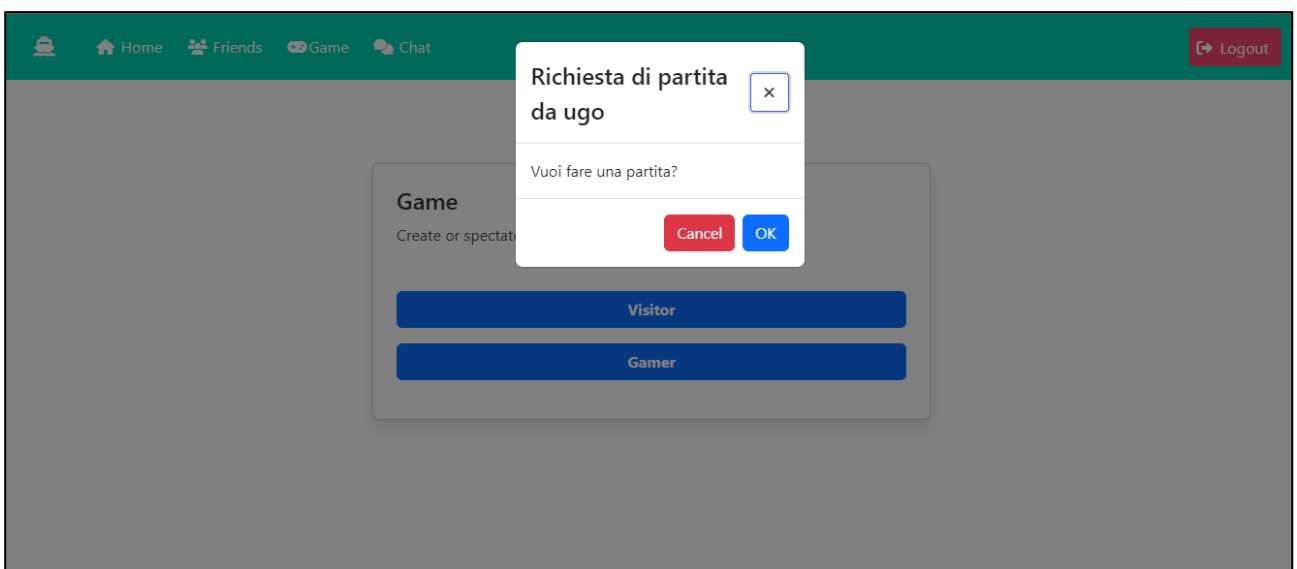
By clicking on the "Games" option in the navigation bar, the user is taken to the games section of the application. In this section, the user can start a new game or view a game that is already in progress.
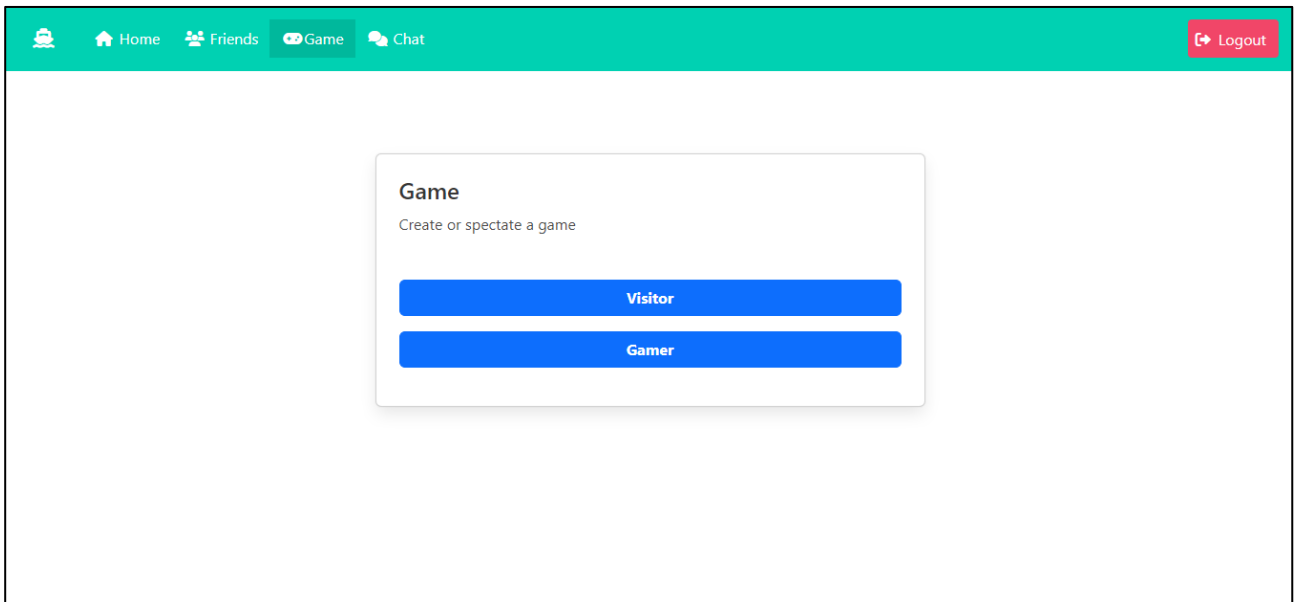
By clicking on the "Visitor" button we can see which matches are live at the moment.



By clicking on the" Gamer" button we are redirected to another page in which we will wait for another player to join the match.

Assuming we have found a player that is ready to play a game, we can send them a match request. Once the player receives the request and accepts it, the match is created and we will be redirected to the game page. In this page, users have the option to create their own custom game dashboard or generate a random one.



Once the match has been accepted and all the players are ready, the game can start. Players can now play the game and "sink some ships!". The players will take turns to guess the coordinates of the opponent's ships, and the first one to sink all the ships wins the game.



At the end of the game, the user will be redirected to the home page. Here, the user will see that the game has been added to the game history section. The user can see the match details, the results and the players that participated in the game.
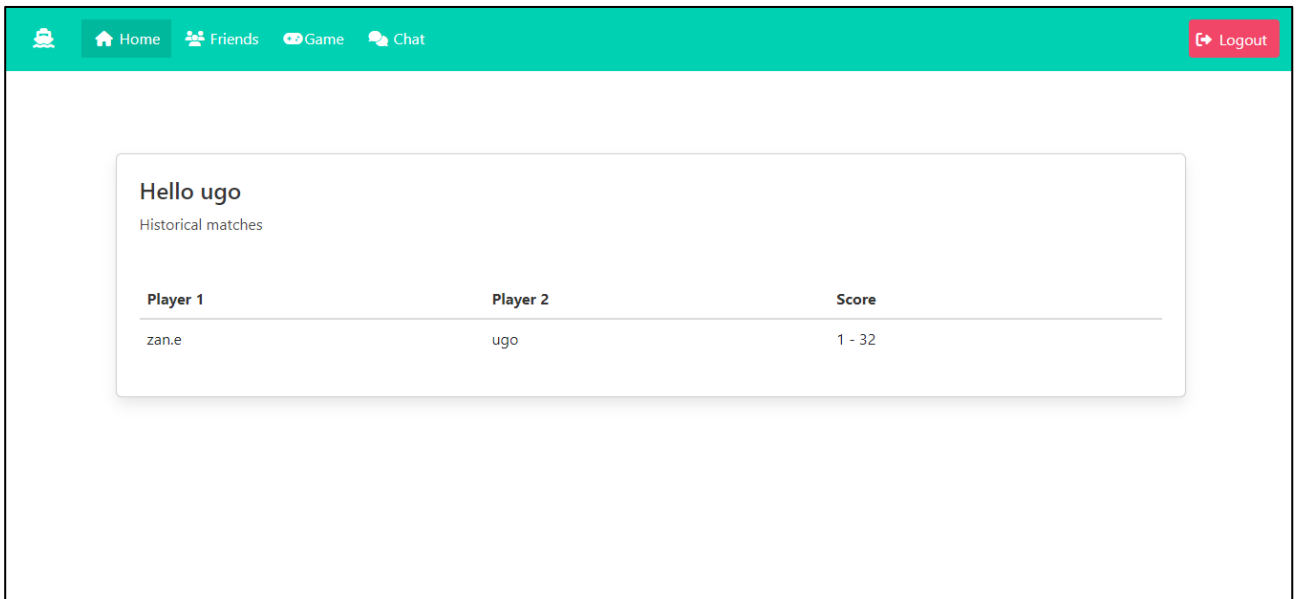
By clicking on the "Chat" option in the navigation bar, the user is taken to the chat section of the application. In this section, the user can send messages to friends and to moderators. The chat allows the user to communicate with friends, other players, and the game moderators. The user can also join a chat room or create a new one, where they can discuss with other players, ask for help or give feedback about the game. This section allows the user to communicate with the community, get support, and have a real-time conversation with friends and other players.