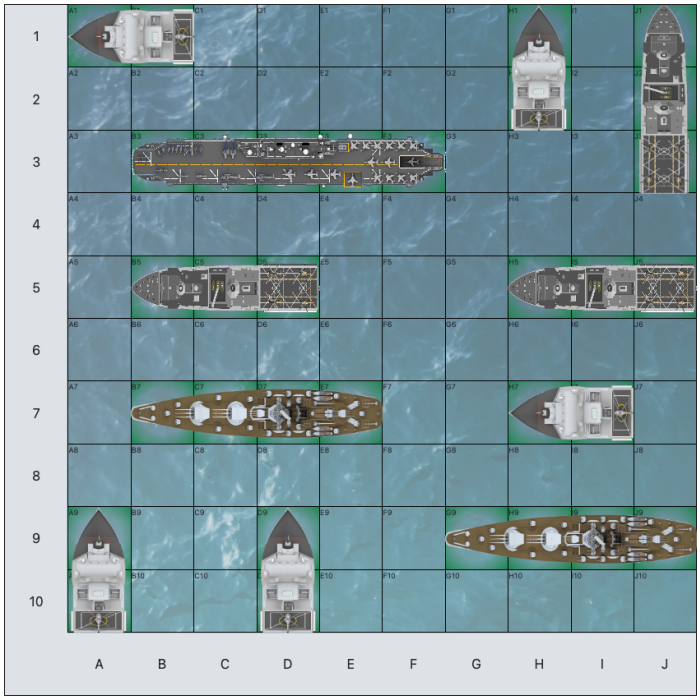
Battleship Web Application

(https://github.com/giova239/BattleShip)

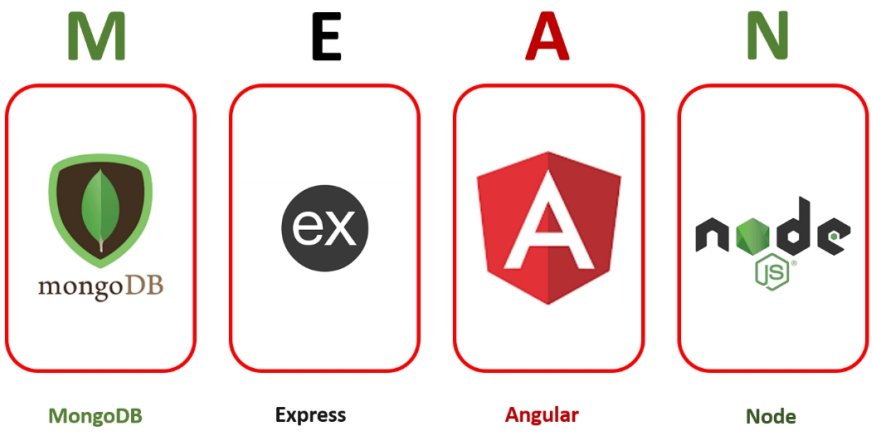


Developed by:

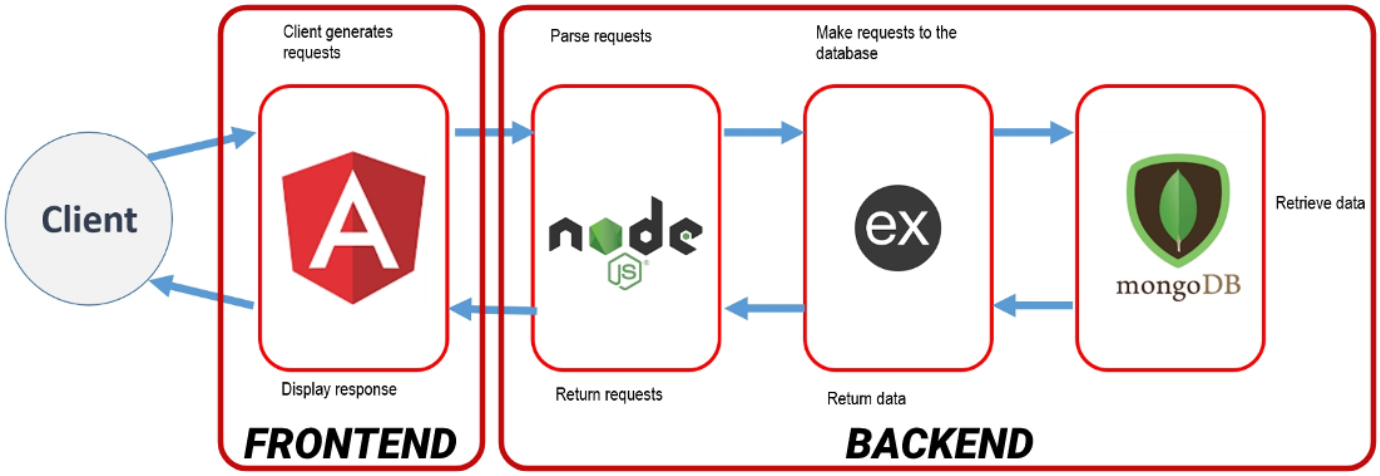
Giovanni Stevanato

(880077@stud.unive.it)

System Architecture



The system architecture used to implement all the application requirements is the **MEAN** architecture, a modern software stack for building web applications.
This acronym is made up of the first letters of **MongoDB**, **Express**, **Angular** and **Node**, all of these **JavaScript**-based technologies allows developers to create a **full-stack** application only using one programming language.

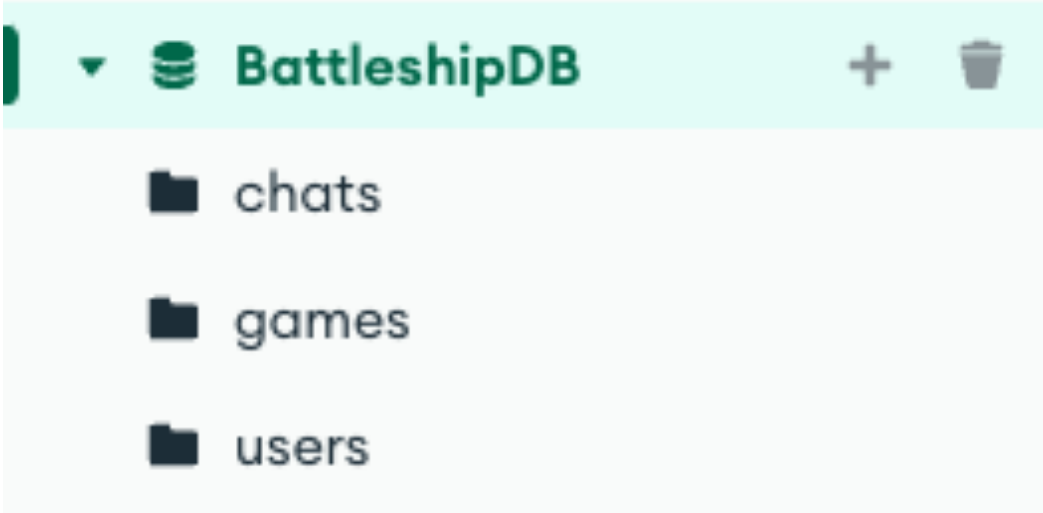


The **Front-end** is based on the Angular Framework, making use of **HTML** and **CSS** to implement the design of the application, and makes use of **Apache Cordova** to create a **mobile** application by scaffolding the front-end in a native android container.

Data Model

The system uses **MongoDB** as the chosen Database Management System (**DBMS**), to safely store all the persistent data needed by the system to fully function.
MongoDB is a **non-relational** DDMS that instead uses **JSON**-like **documents** to organize the data, making the JavaScript communication even better.

The Database is organized in **3 collections**:



- The **USERS** collection:
  - **Document** Model:

```
USER ID
_id:                ObjectId
USER INFO
username:           String          required = true  unique = true
mail:               String          required = true  unique = true
roles:              String[]        required = true
PASSWORD
salt:               String          required = false
digest:             String          required = false
FRIENDS
friends:            ObjectId[]      required = false default = []
pendingRequests:    ObjectId[]      required = false default = []
STATS
wins:               Number          required = false default = 0
losses:             Number          required = false default = 0
hits:               Number          required = false default = 0
misses:             Number          required = false default = 0
SPECIAL FIELD FOR NEW MODERATORS
temporaryPwd:       Boolean         required = false default = false
```

  - Document **example**:

```
_id: ObjectId('633af7e178b4d7050836c7be')
roles: Array
  0: "ADMIN"
  1: "MODERATOR"
friends: Array
  0: ObjectId('633af81678b4d7050836c7c0')
  1: ObjectId('633c57e399aa091997b964fc')
  2: ObjectId('6346b7a5f4b11c96396c999c')
  3: ObjectId('633da0ec3387c828d909682d')
pendingRequests: Array
  0: ObjectId('635001df8362b3186bcd8c32')
username: "admin"
mail: "admin@battleship.it"
salt: "d4012efc03ef6415024986b52105d063"
digest: "4e7ca5f748e9972bebad19fd7b12ec50fe48ce409834333328cff92be2dc2d694b0655…"
__v: 14
hits: 0
losses: 0
misses: 2
wins: 0
temporaryPwd: false
```

- The **CHATS** collection:
  - **Document** Model:

```
CHAT ID
_id:                ObjectId
USERS
user1:              ObjectId        required = true
user2:              ObjectId        required = true
MESSAGES
messages:           Message[]       required = false default = []
```

  - **Message** nested object structure:

```
MESSAGE INFO
isFromUser1:        Boolean         required = true
read:               Boolean         required = true
date:               Date            required = true immutable = true
text:               String          required = true
```

  - Document **example**:

```
_id: ObjectId('6353a4430d9588085d609ada')
user1: ObjectId('633af7e178b4d7050836c7be')
user2: ObjectId('633af81678b4d7050836c7c0')
messages: Array
  0: Object
      _id: ObjectId('6353a4460d9588085d609adb')
      isFromUser1: true
      read: true
      date: 2022-10-22T08:05:26.521+00:00
      text: "ciao
            "
  1: Object
      _id: ObjectId('6353a44c0d9588085d609adc')
      isFromUser1: true
      read: true
      date: 2022-10-22T08:05:32.559+00:00
      text: "come va?
            "
  2: Object
  3: Object
  4: Object
__v: 5
```

- The **GAMES** collection:
  - **Document** Model:

```
GAME ID
_id:                ObjectId
PLAYERS
user1:              ObjectId        required = true
user2:              ObjectId        required = true
BOARDS
board1:             Boolean[][]     required = false
board2:             Boolean[][]     required = false
MOVES
moves:              String[]        default = []
PLAYERS CONNECTION
isUser1Connected:   Boolean         required = false default = 0
isUser2Connected:   Boolean         required = false default = 0
TURN
isUser1Turn:        Boolean         required = true
```

  - Document **example**:

REST APIs

Endpoints	Method	Description
/	GET	Returns API version
---------- USER ROUTES ----------		
/users	GET	List all users and their info. Doesn't include salts and digests.

/users	POST	Creates a new user with MODERATOR ROLE and a temporaryPwd. Only works if logged in with a MODERATOR account.
		
		BODY:
		{
		username : String,
		password : String
		}

/users/:id	GET	Returns the user with the specified id and his info. Doesn't include salt and digest.
/users/:id	DELETE	Remove the user with the specified id. Only works if logged in with a MODERATOR account.
/login	GET	Log in an existing user using 'basic authentication technique'. Returns a JWT TOKEN containing session data that expires after 24 hours.

/register	POST	Creates a new user with NO ROLES and generates digest and salt.
		
		BODY:
		{
		mail : String,
		username : String,
		password : String
		}

/moderatorSetup	PUT	If logged in with a newly created moderator edits user credentials and set TemporaryPwd to false.
		
		BODY:
		{
		mail : String,
		username : String,
		password : String
		}

---------- FRIEND ROUTES ----------		
/friends	GET	Returns all the users that are friends with the logged user.
/friends/:userID	POST	The logged user adds himself on the user with the specified userID pending friend requests. If the other user already added the logged user adds both ids in the respective friendlists and remove the pending request. SOCKET: emit “newFriendRequest” to ROOM=userID and the logged user id so client can show the new request in real time.

/pendingRequests	GET	Returns all the users that have sent a friend request to the logged user.
/pendingRequests/:userID	DELETE	The logged user rejects a pending friend request
---------- CHAT ROUTES ----------		
/chat/:userID	GET	Return the chat between logged user and the user with the specified userID. SOCKET: emit "readMessage" to ROOM=userID and the amount of messages read so the client can update the unread messages in real time.

/chat/:userID	POST	Post a message to the chat with logged user and the user with the specified userID.
		
		BODY:
		{
		text : String
		}
		
		SOCKET:
		emit "newMessage" to ROOM=chatID and the newly sent message so the client can show it real time in the chat.
		emit "newUnreadMessage" to ROOM=userID and the logged user id so the other user can see he has new unread messages in real time.

/unreadMessages/:userID	GET	Returns the amount of unread messages with the user with the specified userID

/readMessages/:userID	PUT	Marks the unread messages with userID as read. SOCKET: emit “readMessage” to ROOM=userID and the amount of messages read so the client can update the unread messages in real time.
---------- GAME ROUTES ----------		
/matchmaking	POST	Adds the logged user to the matchmaking queue. Removes him from the queue if already queueing. When a pair of players with similar skill level is formed it creates a new game. SOCKET: emit “matchFound” to ROOM=userID and the game data when the match is found.
/challenge/:userID	POST	Creates a new game between the logged user and the user with the specified userID if they are friends. SOCKET: emit “challenged” to ROOM=userID and the game id so the challenged user can join the game.
/game/:gameID	GET	Returns the game with the specified gameID.

/game/:gameID	PUT	If the logged user is one of the players of the specified gameID it updates the parameters specified in the body.
		
		BODY:
		{
		[board1]:Boolean[][],
		[board2]:Boolean[][],
		[isUser1Connected]:Boolean,
		[isUser2Connected]:Boolean
		}
		
		SOCKET:
		emit "board1Update" or "board2Update" to ROOM=gameID and the new board array if a board is updated.
		
		emit "user1ConnectionUpdate" or "user2ConnectionUpdate" to ROOM=gameID and the new value if a player connects or disconnect.

/fire/:gameID/:move	POST	Play the specified move on the specified game if logged as a player for that game and if it is his turn, then changes turn. Also checks for win and update player stats SOCKET: emit "move" to ROOM=gameID and the move itself so client can show it on board real time. emit "win" ROOM=gameID and the winner username if the move was the winning one.

/gameChat/:gameID	POST	Send a message in the game chat of the specified gameID. SOCKET: emit “gameMessage” to ROOM=gameID and {id, username, text} so everyone connected to the game can receive the message and display it accordingly. N.B. The game chat has no data persistence and is only managed by sockets. Only the messages received while connected to the socket room will be displayed.
/surrender/:gameID	POST	If the logged user is a player of the specified gameID ends the game resulting in a win for the opponent. Also updates player stats. SOCKET: emit “win” ROOM=gameID and the winner username.

USER AUTHENTICATION

- User **signup**:

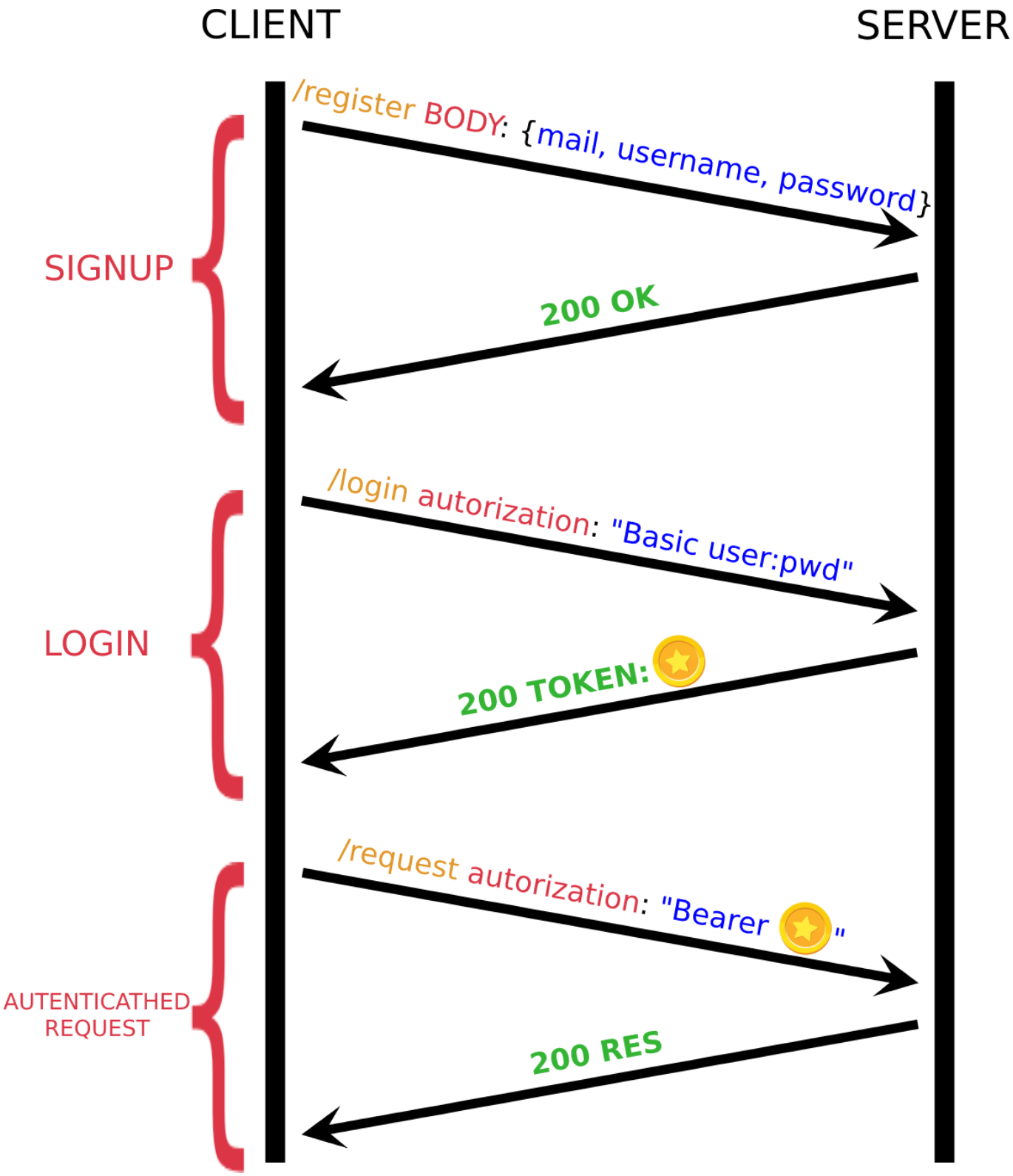
The user signup is performed by the client sending an **HTTP** `POST` **request** to the `/register` endpoint, specifying in the `BODY` section of the request `mail`, `username` and `password`. The **back-end** ensures that the **username** and the **mail** are **valid** and **not** already **taken** and in case of success generates the salt and hash of the password which is then **stored** inside the database with the **newly created user**.

- User **login**:

The user login is performed by the client sending an **HTTP** `GET` **request** to the `/login` endpoint, specifying in the `authorization HEADER` section of the request the **authentication strategy:** `"Basic"` and the **Base64-encoding** of the string `username + ":" + password`. The **back-end** receives the **username** and the **password**, **decodes** them back to **UTF-16**, find the **user** with the **corresponding username**, retrieves the stored **salt** and calculate the **hash** which is then compared to the **digest** stored in the Database. If the **digests** match the server authenticates the user **by generating a JWT token** (valid for **24h**) containing all the **useful user data** which is then returned as a **response**. The **token** is signed with a **secret key** stored on the server.

- Performing **requests** as a **logged user**:

After a **successful login** the **client stores** the **token** in the **browser** and uses it to authenticate himself on the **succeeding requests** by including it in the `authorization HEADER`. The **back-end** receives the **token** and **validates** it using the **secret key** and then know what user is performing the requests. If the token is **missing**, **expired** or simply **not valid** the client will have to **login again** to provide a valid token before performing any action.


CLIENT
SERVER
SIGNUP
/register BODY: {mail, username, password}
200 OK
LOGIN
/login autorization: "Basic user:pwd"
200 TOKEN:
AUTENTICATHED REQUEST
/request autorization: "Bearer "
200 RES

• **SECURITY** features:

To **prevent data leaks** during authentication is **mandatory** to use the **HTTPS** protocol since we are using the **basic authentication strategy** which does not use any **encryption** to ensure privacy and integrity of the exchanged data.

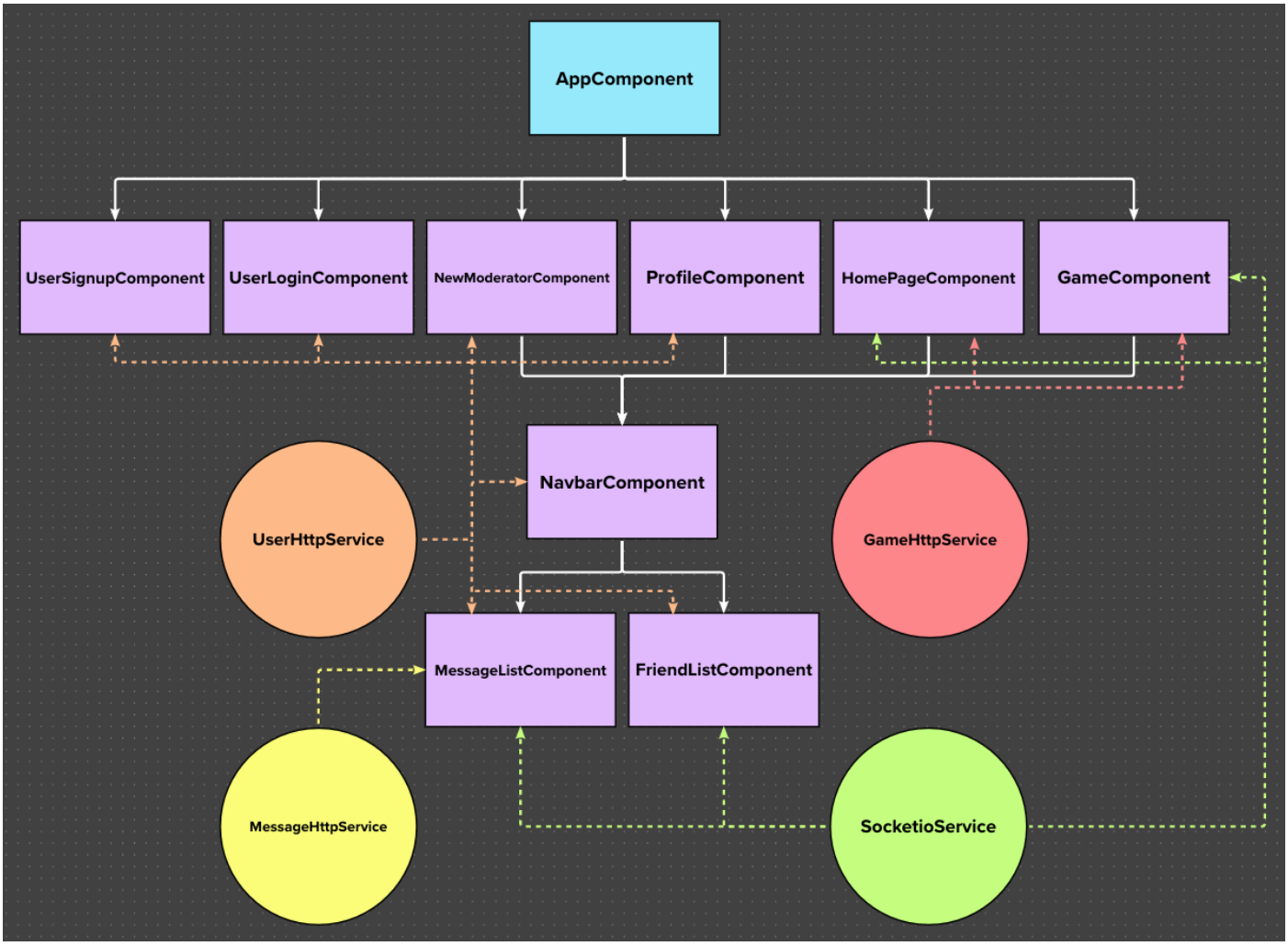
Passwords are an extremely **sensitive** data which should be stored in the safest way possible since most users use the same passwords for every account they create. to ensure the **safety** of the **stored passwords** the following **precautions** have been taken:

- **HASHING:** passwords are not stored inside the database in clear text, the backend stores the **sha512** hash of the password instead. This is a basic but very important security feature because, in the event of a **security breach**, since hashing is a **one-way function**, reconstructing the original passwords would be very complicated.

- **SALTING:** before going thought the hashing algorithm, a **random 16 bytes hex string** called salt, is attached to the password, the salt is then stored inside the database. This is to prevent **Dictionary Attacks** (or Rainbow table attacks) which uses **precomputed hashes** of **commonly used passwords** trying to reveal some of the stored hashes

FRONTEND

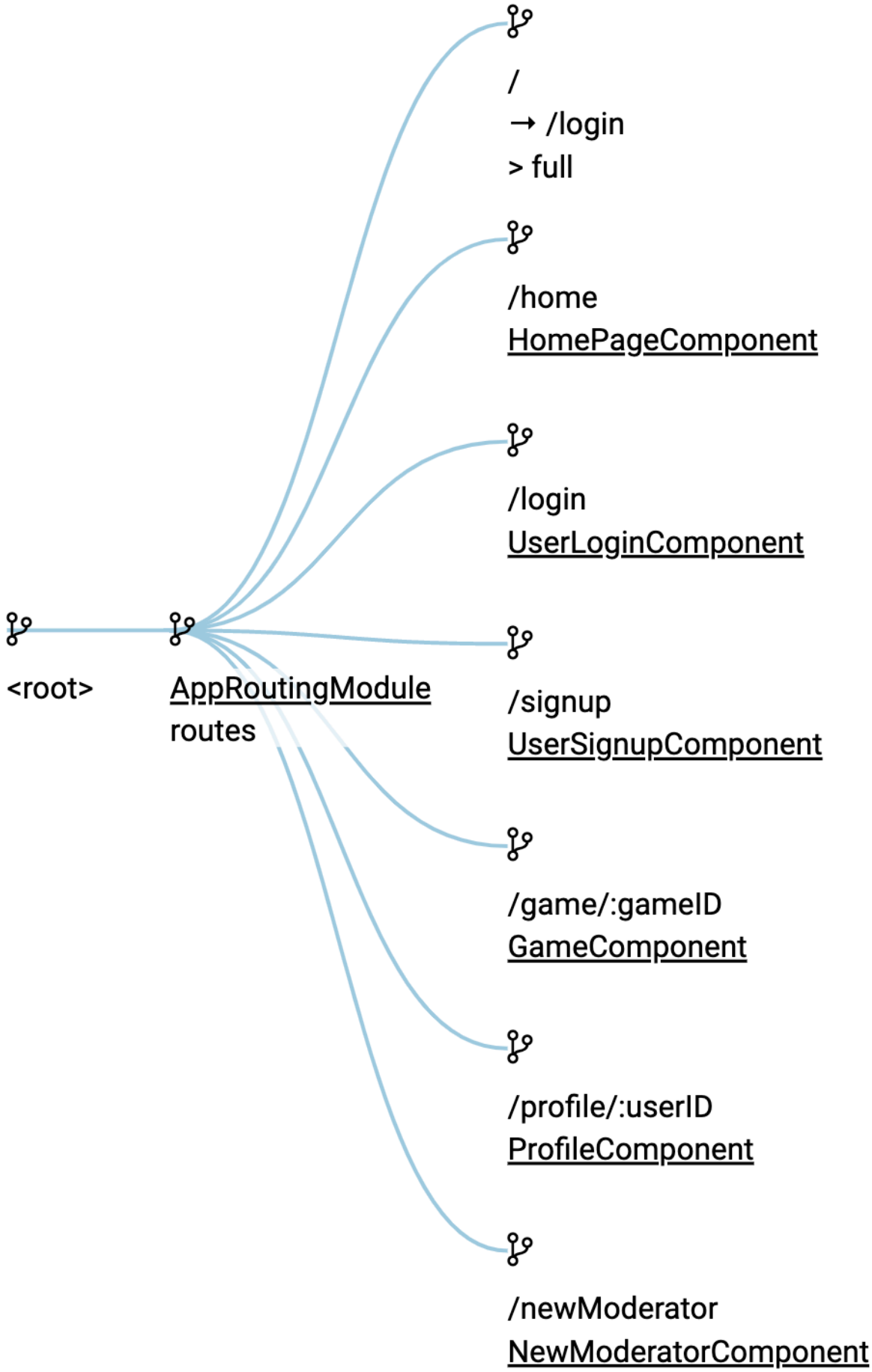
Angular **Components** and **Services overview**:



- **Components**:
  - **AppComponent**: is the main component where **SPA** is loaded.
  - **UserSignupComponent**: is a component including just the **form** for the user **sign-up**.
  - **UserLoginComponent**: is a component including just the **form** for the user **log-in**.
  - **NewModeratorComponent**: is a component including the **form** to create a **new moderator**, only works if the **logged user** is a **moderator** himself.
  - **ProfileComponent**: is a component that displays any user **profile**, allowing the logged user to **add** or **remove** as **friend**, to see his **stats** (if the logged user is a moderator or is friend with the user).
  - **HomePageComponent**: is a very **simple home page** just including the **navbar** and the **matchmaking button**.
  - **GameComponent**: is the component that handles the actual **game** and it is responsible to **display** the **boards** distinguishing between **player1 player2** or a **spectator**, also handles the **game chat**.
  - **NavbarComponent**: is the component that show the **navbar**, allows the user to **logout** or **navigate** to **other sections** and contains the **friend list** and the **chats**.
  - **MessageListComponent**: is the component that loads a specific **chat** between **two users** and allows them to exchange **messages** in real time.
  - **FriendlistComponent**: is the component that shows the **friend list** of the logged user and allows him to **challenge**, **chat** or **open** the **profile** of any of his friends. This component also list the incoming **friend requests** and allows the logged user to send a friend request himself.

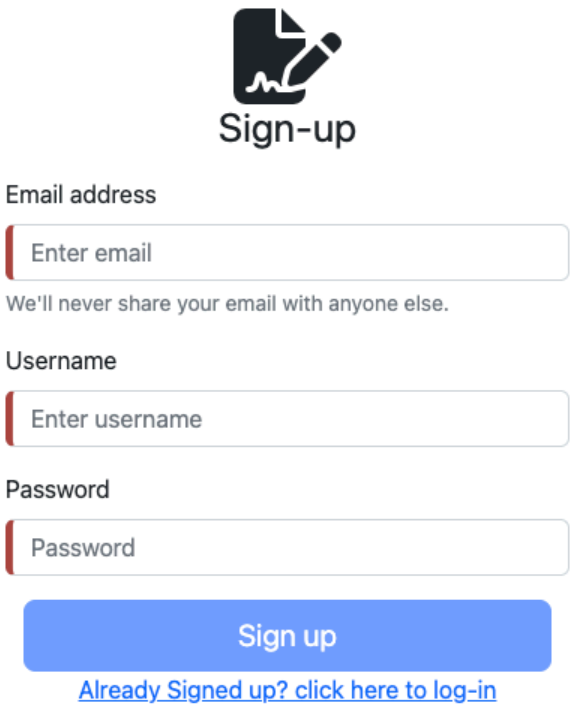
- **Services**:
  - **UserHttpService**: is the **service** that handles all the **HTTP requests** in the **user** section of the APIs.
  - **MessageHttpService**: is the **service** that handles all the **HTTP requests** in the **chat** section of the APIs.
  - **GameHttpService**: is the **service** that handles all the **HTTP requests** in the **game** section of the APIs.
  - **SocketioService**: is the **service** that **handles** all the **socket events** coming from the back-end**.**

- **Routes:**


<root>
AppRoutingModule
routes
/
→ /login
> full
/home
HomePageComponent
/login
UserLoginComponent
/signup
UserSignupComponent
/game/:gameID
GameComponent
/profile/:userID
ProfileComponent
/newModerator
NewModeratorComponent

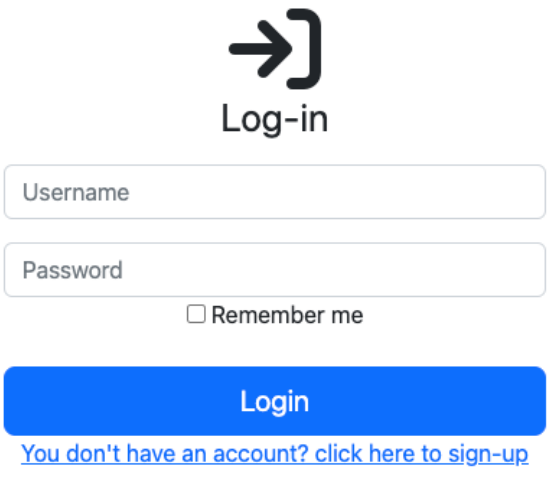
APPLICATION WORKFLOW

- **Signup**:



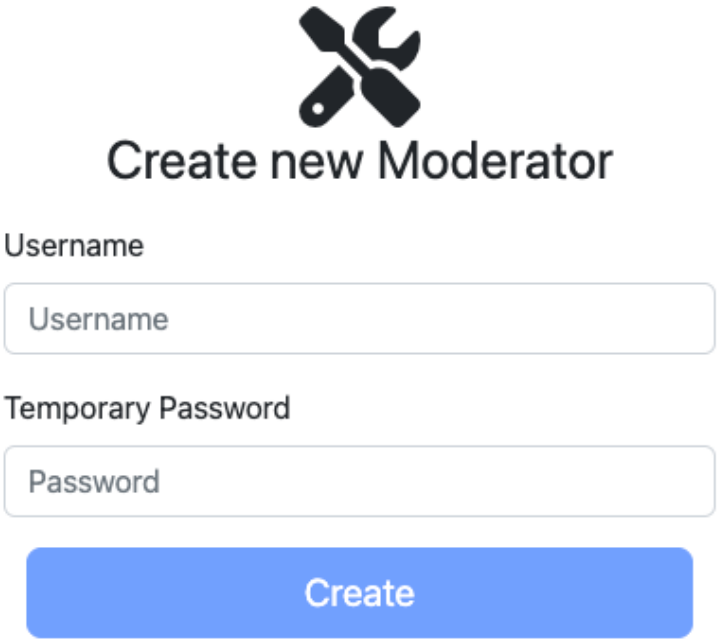
*The **signup** form.*

- **Login**:

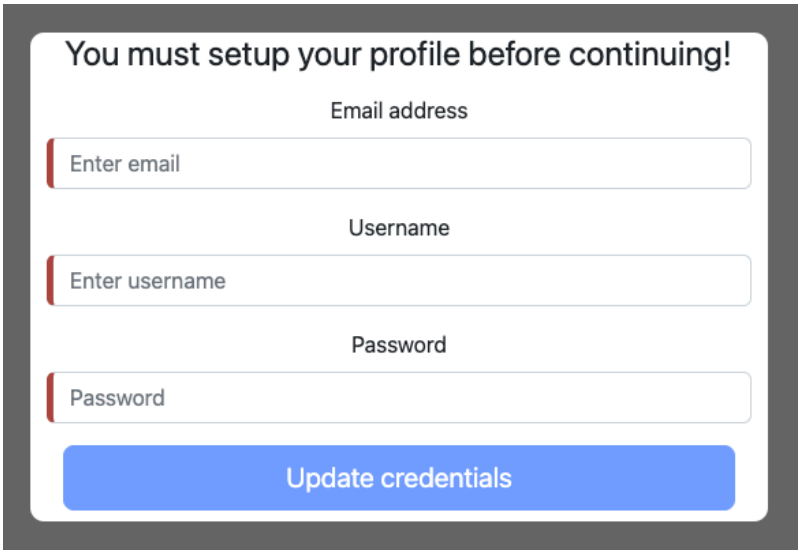


*The **login** form.*

- **New Moderator**:

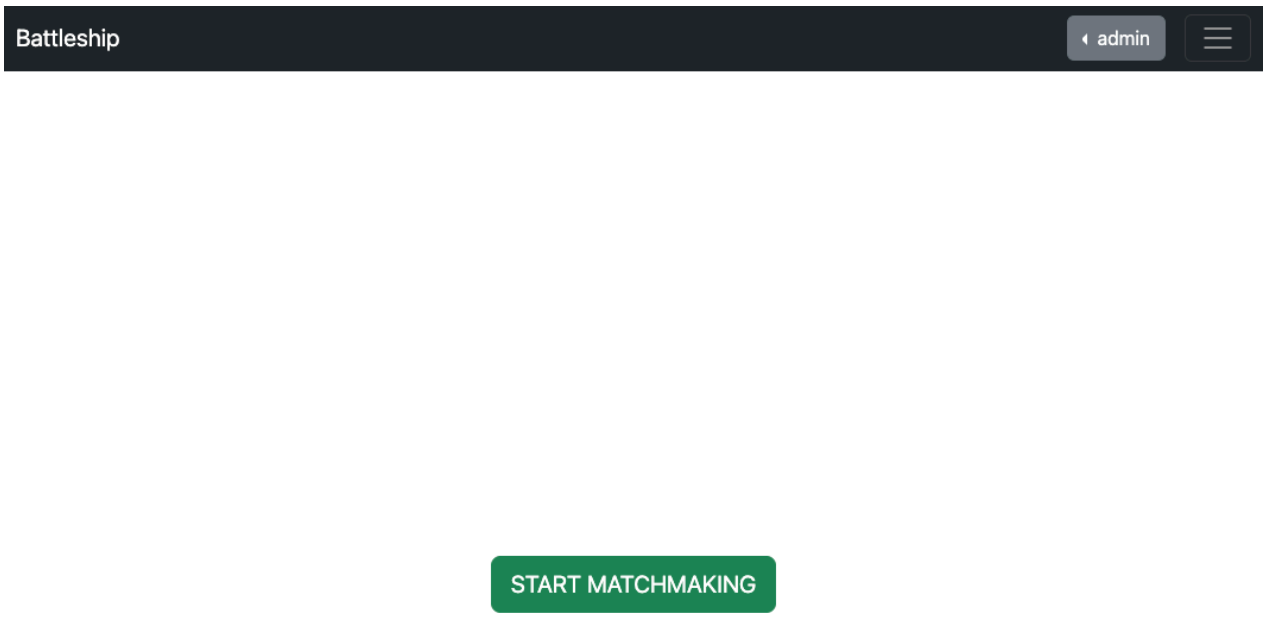


*A user logged in as a **moderator** can **create** a **new moderator** with a **temporary username** and **password**.*



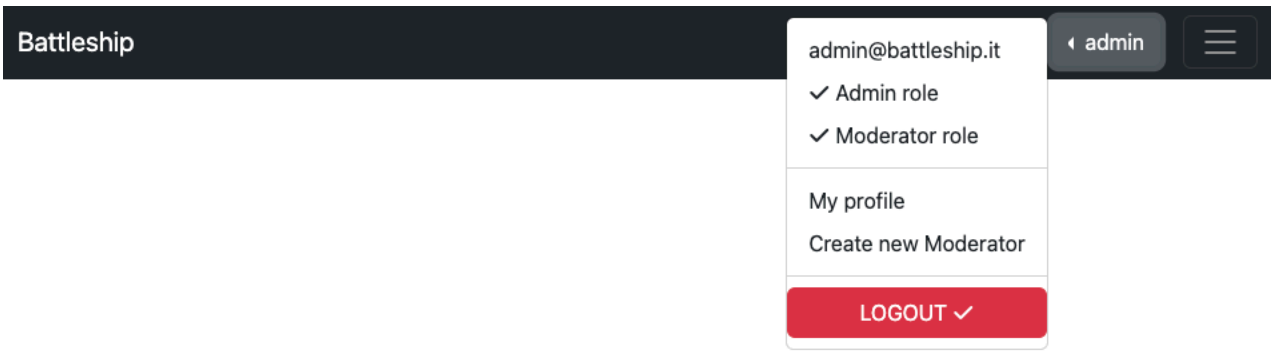
*On **first login** the **new moderator must insert** his **information**.*

- **Home**:



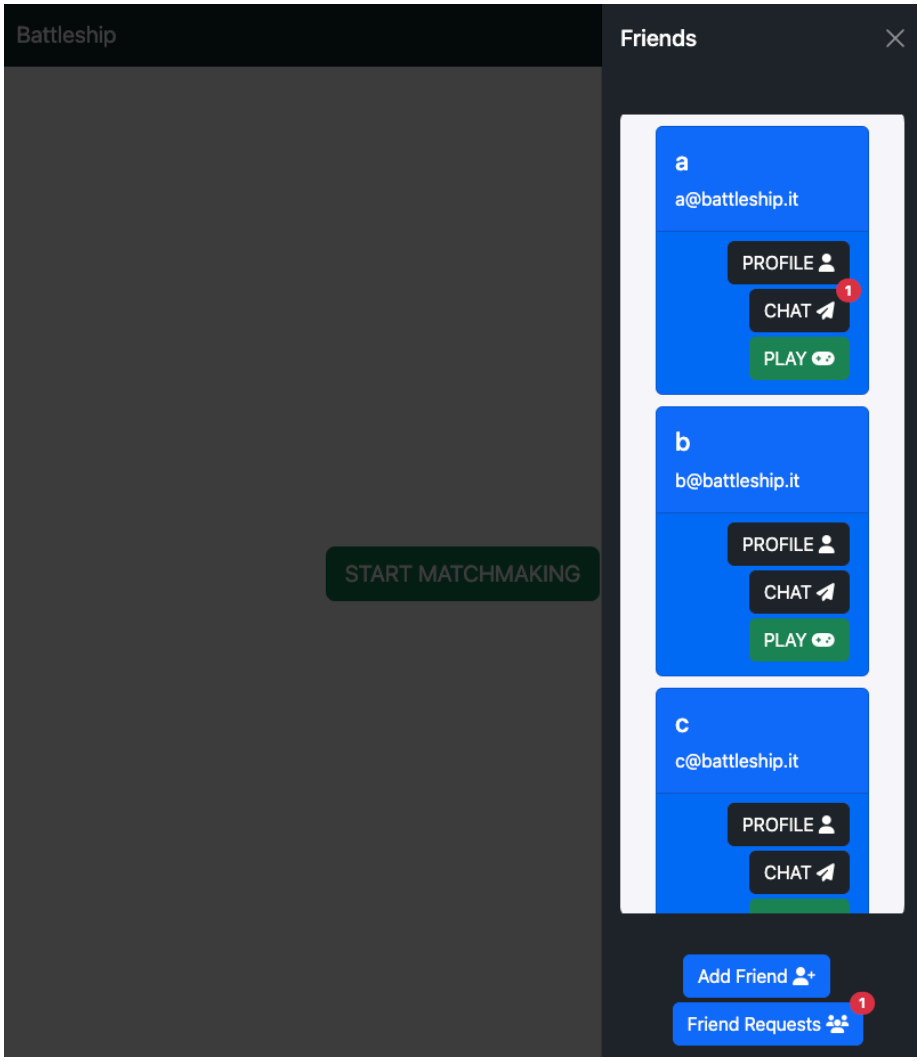
*The **home** page.*

- **Navbar**:

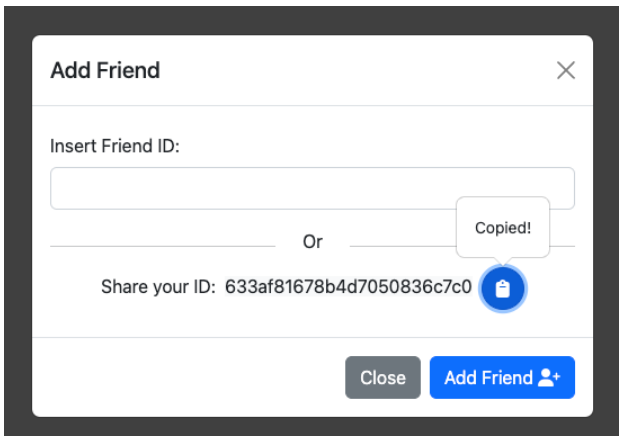


*Clicked on **user button***

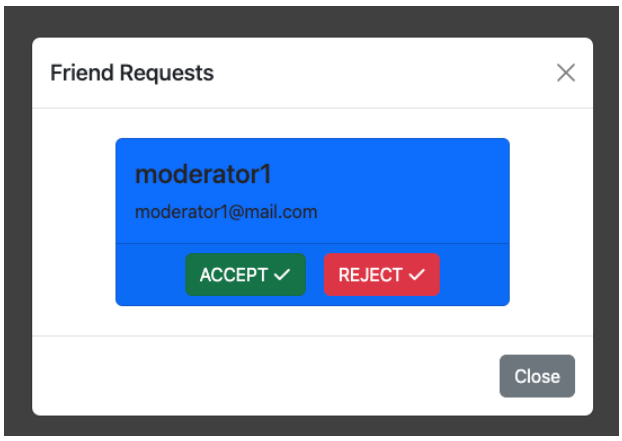
- **Friendlist:**



Clicked on the ☰ ***button*** *in the top right corner.*

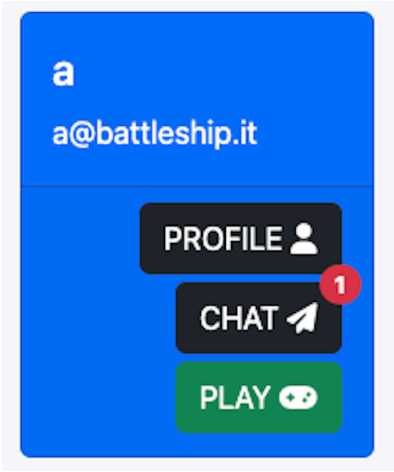


Clicked on ***Add Friend button****.*

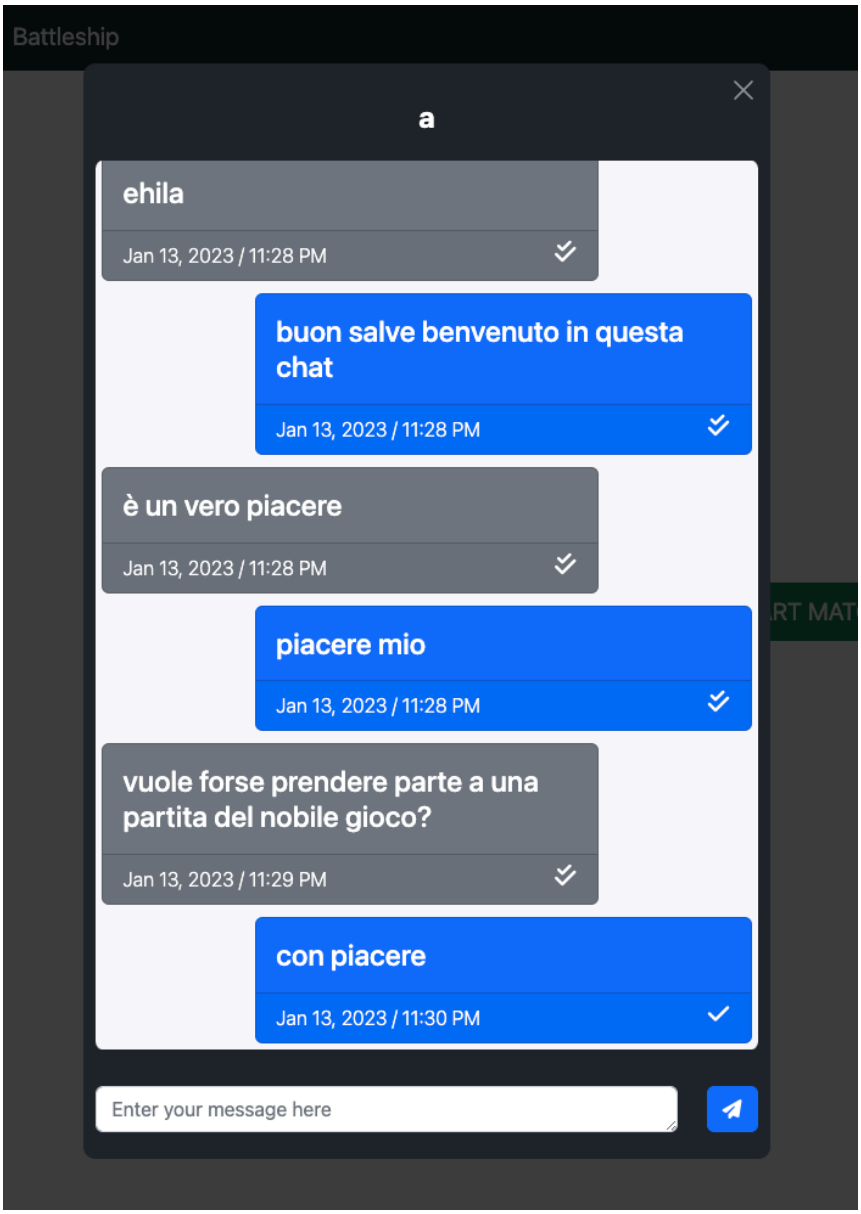


Clicked on ***Friend requests button****.*

- **Chat:**

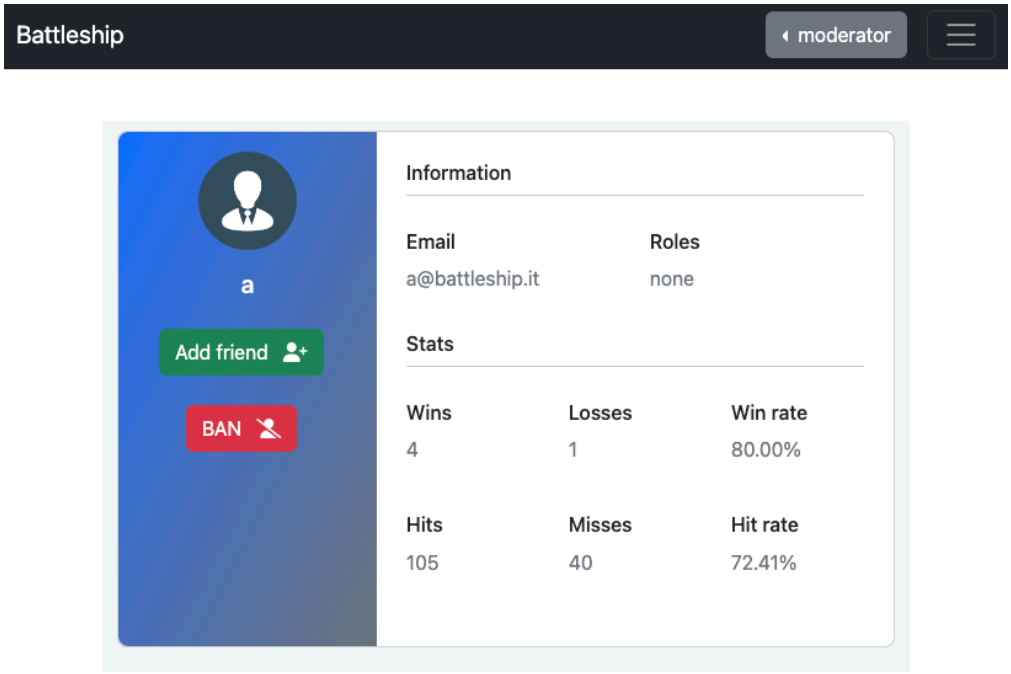


*Click on the **chat button** with user 'a'.*

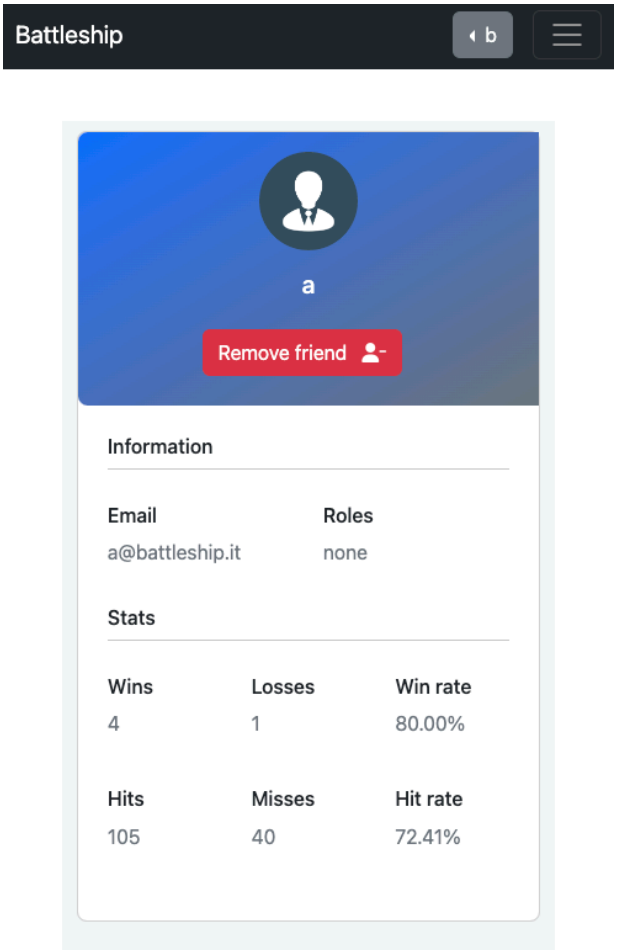


*The **chat** with user 'a' is opened.*

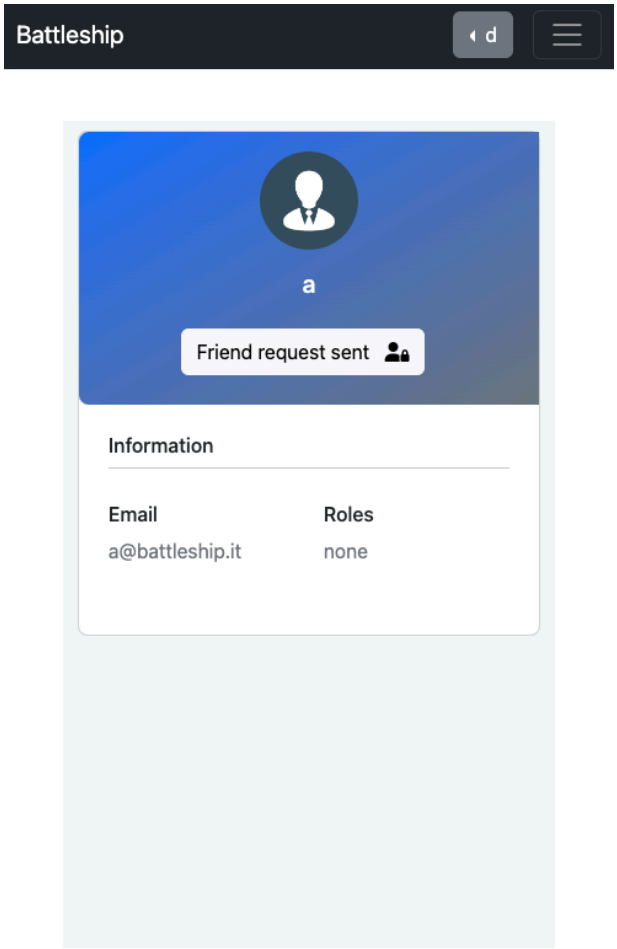
- **Profile:**



*User 'a' **profile** logged as a **moderator**.*

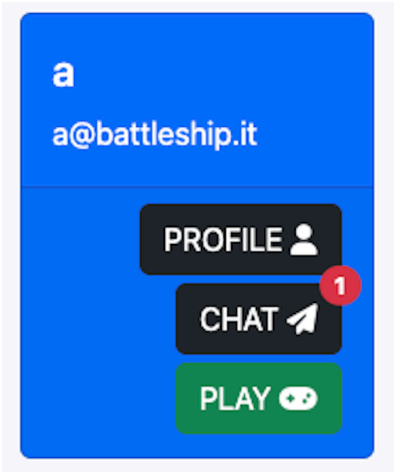


*User 'a' **profile** logged as a **friend**.*

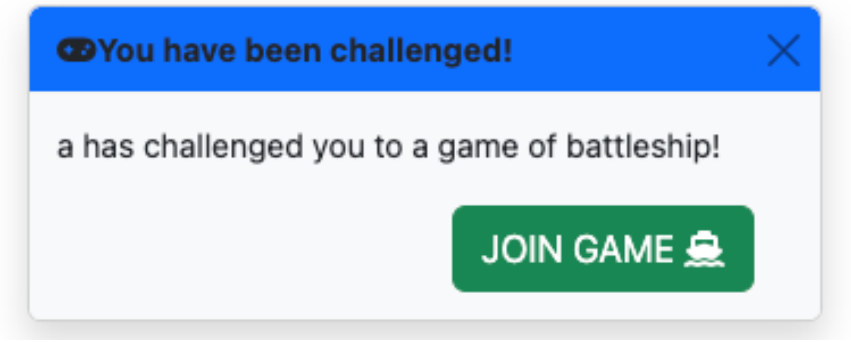


*User 'a' **profile** logged as someone who sent him a **friend request**.*

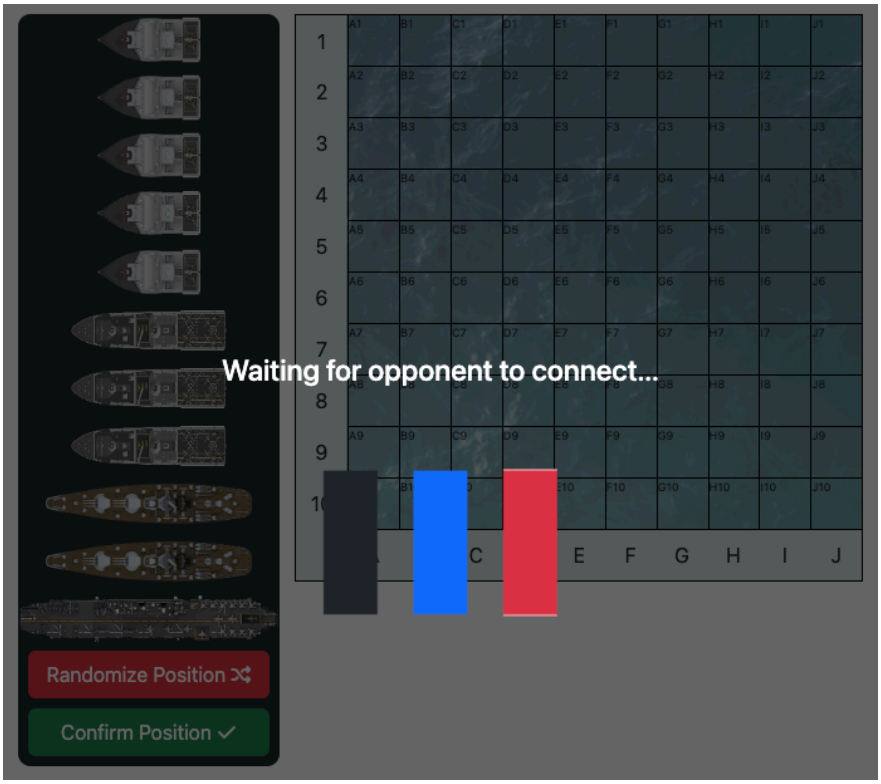
- **Challenge**:



*Click on the **play button** with user ‘a’.*

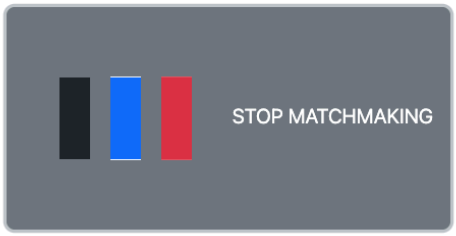


*User ‘a’ receives a **pop-up** Toast.*



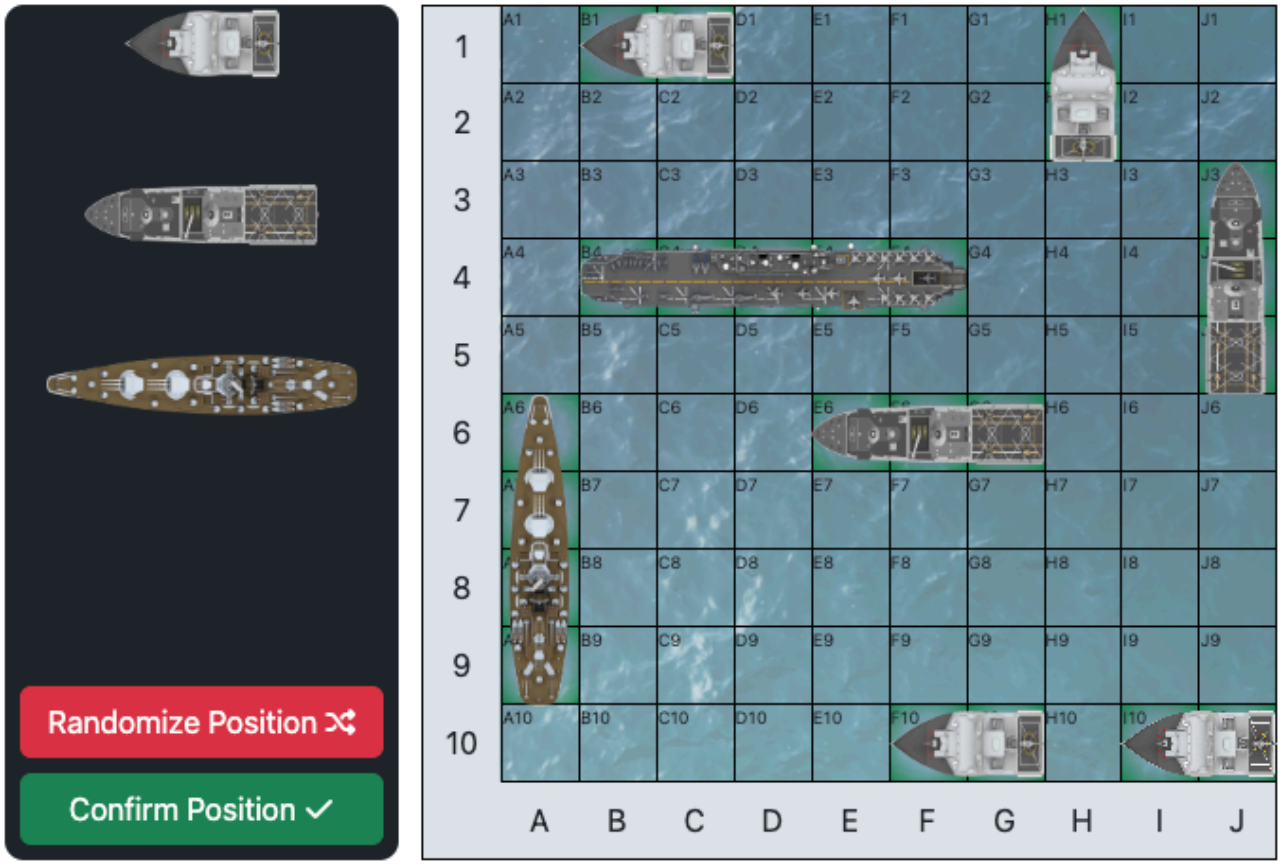
*User ‘b’ **waits** for user ‘a’ to join the game.*

• Matchmaking:



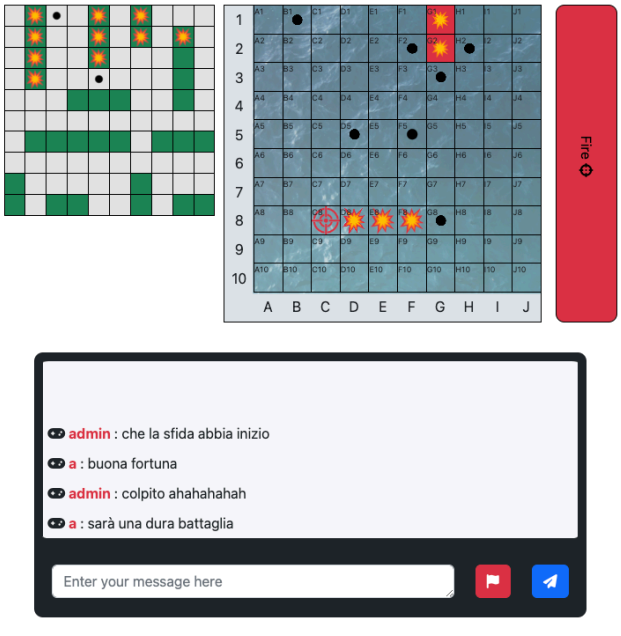
*Clicked on the **start matchmaking button**.*

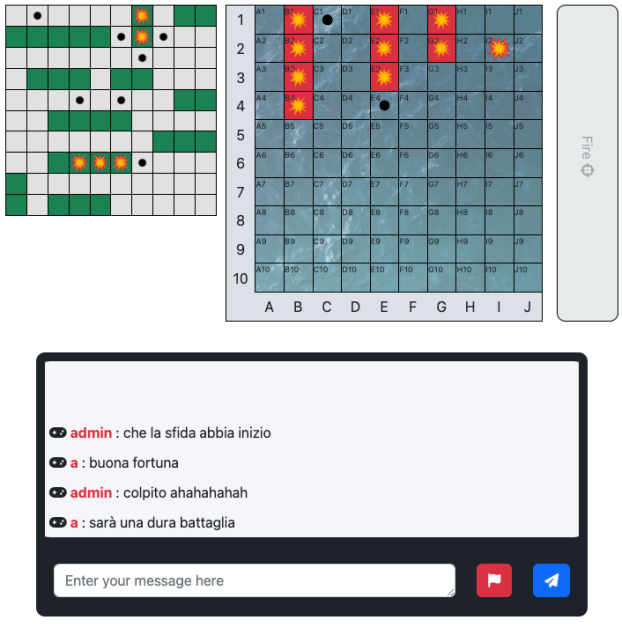
• Positioning phase:



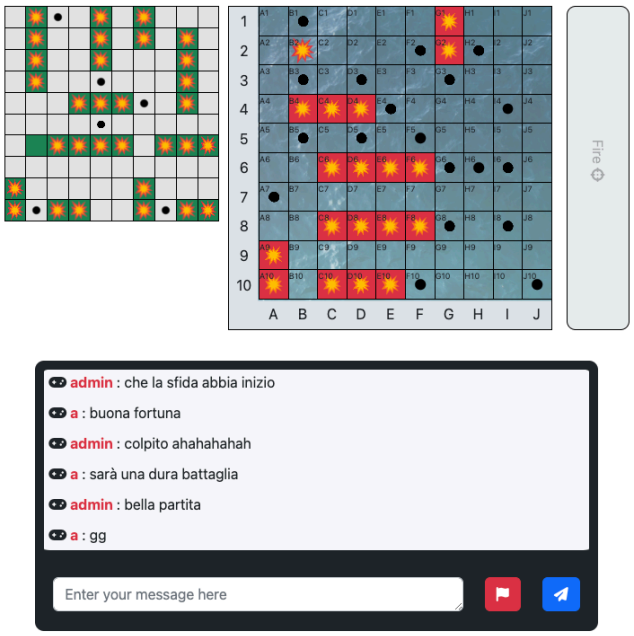
***Dragging** the **ships** to position them, **double click** to **rotate**.*
***Click** on the **Randomize button** to **automatically place ships**.*
***Click** on **confirm Position** when **ready**.*

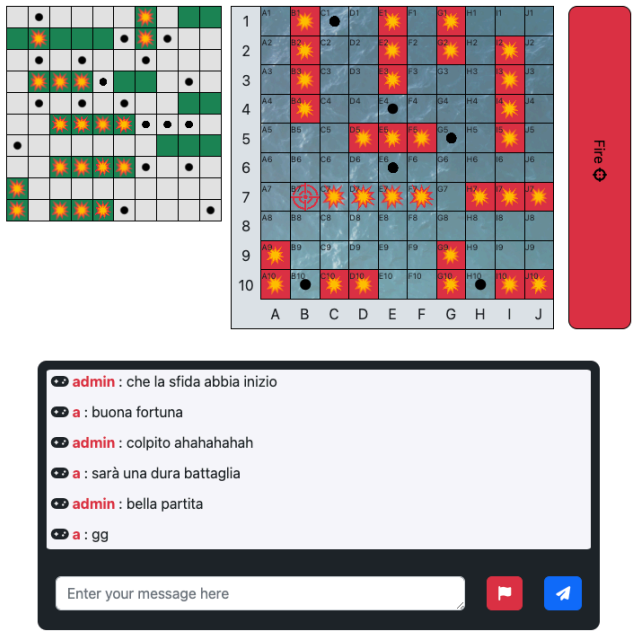
- **Game:**



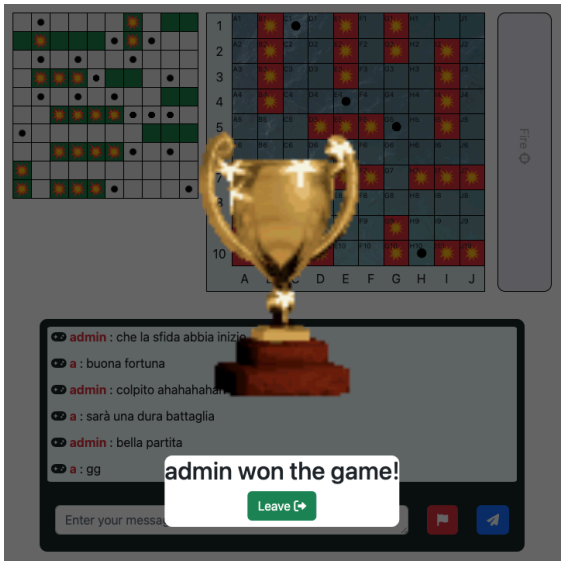


*Game is **just started***



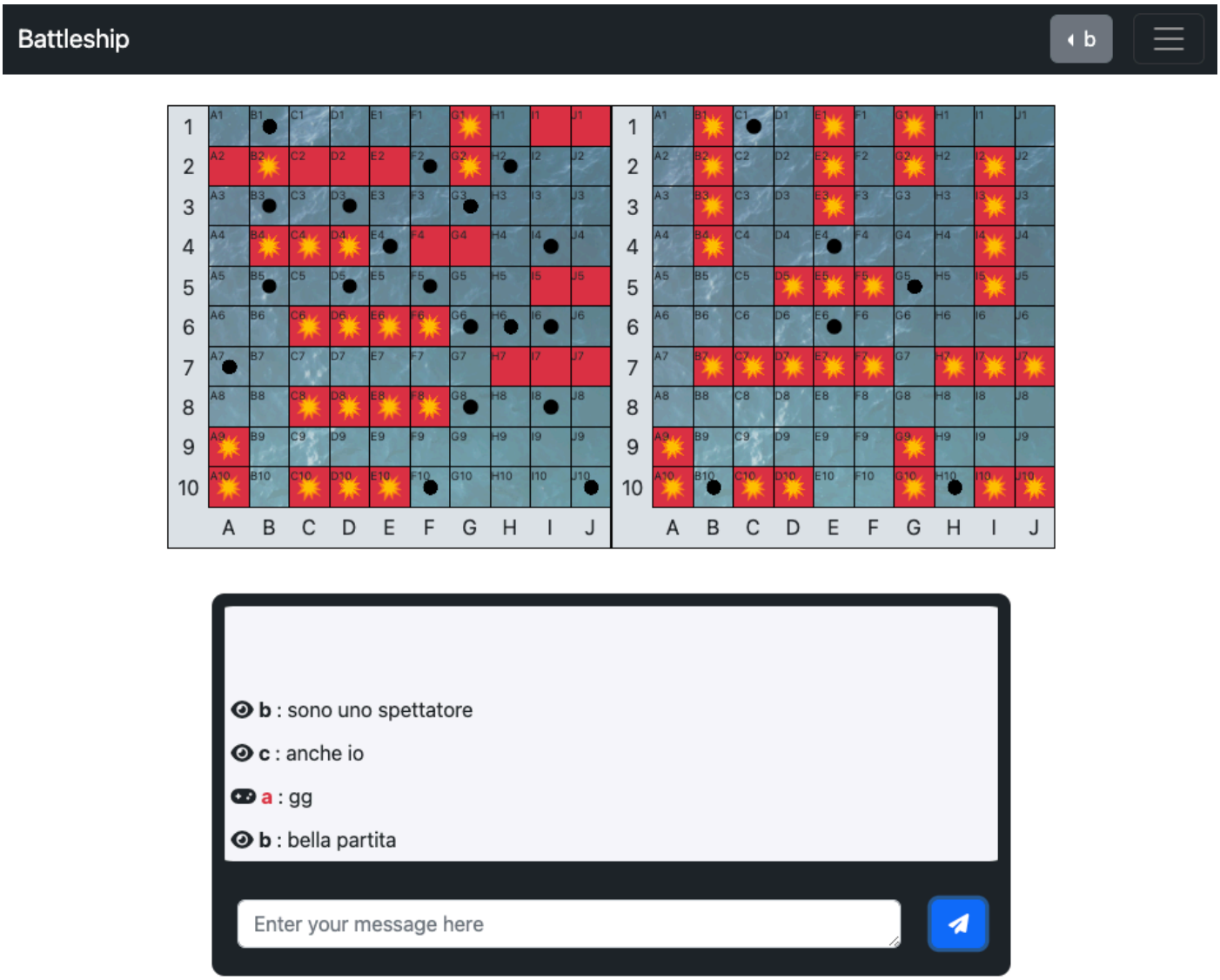


*Game is **almost over**.*



*Game **over**.*

- **Spectator:**



*User ‘b’ is **spectating** the game **chatting** with other spectators (Spectators messages are not visible by players).*