Battleship Free-for-All

(v. 0.1.0)

Davide Di Marco
davide.dimarco5@studio.unibo.it

Ines Fraccalvieri
ines.fraccalvieri@studio.unibo.it

May 14, 2024

This project aims to design and implement a new version of the classic Battleship game by transforming it into a dynamic free for all experience for four players. In this game, individual players compete against each other in a challenge for naval dominance. The objective remains simple - strategically deploy their fleet to sink as many enemy ships as possible while safeguarding their own. After logging in, users can create lobbies (waiting rooms) or join existing ones. In the lobby, players must strategically position their ships on a grid within a time limit of 15 seconds. Once the game starts, players take turns calling out shots, and each turn has a 15-seconds time limit.

1 Goals

The project's main objective is to develop a unique version of the classic Battleship game, transforming it into a multiplayer, free-for-all experience. In this version, four players compete against each other on a shared virtual battlefield, each strategically placing their ships on the grid. After joining the platform, each player selects a nickname to use during their session. They can either create a new game lobby or join an existing one to compete against other players. The game is designed to provide real-time notifications to keep all participants informed about the ongoing activities, helping them devise effective attack strategies to win. The game includes specific rules summarized in the table 1:

Game Rules	**Description**
Grid size	15x15
Ship Positioning countdown	At the beginning, players have 15 seconds to choose random positions on the grid for their ships on the battlefield.
Turn countdown	Each turn, the player has 15 seconds to fire on the battlefield.
Number of ships	The total number of ships for each player is 5.
Single shot per turn	Each turn, player can make a single shot on the grid.
Shared cells	Each cell can contain more than one ship.
Private View	Each player can only view his own battlefield and the shots they have fired.

Table 1: Rules and details of the game.

1.1 Question and Answers

Here are some questions to help clarify the project requirements, particularly focusing on the user interaction with the system and describing some technical requirements that must be met.

- **Question:** *How do users connect to the system?*

  Answer: Users can connect to the system by accessing the game's platform through a web browser. Users will choose a nickname that will be used for the entire duration of their stay on the platform.

- **Question:** *What can users do when connected?*

  Answer: Once connected, users can create new game lobbies (waiting rooms) or join existing ones to start playing Battleship with up to four other players.

- **Question:** *How does the game start?*

Answer: The game starts once all players have connected to the lobby and a countdown has ended.

- **Question:** *When does the game finish?*

  Answer: The game finishes when only a single player remains in the game.

- **Question:** *What happens in case of a client's disconnection during the game?*

  Answer: If a client disconnects from the lobby, the system excludes them from the current game and lobby where they are connected. The game does not stop, and all remaining players receive a notification of the disconnection.

- **Question:** *How do players position their ships?*

  Answer: Players position their ships by choosing from a randomly generated grid provided by the game.

1.2 Usage Scenarios

This section explains the user scenarios identified in the project. It details the steps from the beginning, when the user connects to the platform, to the end of the game. Assuming the typical scenario of a user wanting to play Battleship, the following steps have been identified:

1. **Player Identification**
   A user accesses the game platform through a web browser. Upon entering the platform, they are prompted to provide a unique nickname to identify themselves during gameplay. Once a nickname is chosen, the user proceeds to the lobby creation interface. This process allows the user to distinguish themselves and facilitates interaction with the system.

2. **Lobby Access and Creation**
   After logging in, the user has the option to create a new lobby or join an existing one. Creating lobbies enables the user to organize Battleship matches with specific groups or join existing ones. The lobby management functionality enables users to organize and participate in Battleship matches.

3. **Ship Positioning Phase**
   Once the user has joined a lobby and the game is ready to start, the ship positioning phase begins. The user is given a set amount of time, typically fifteen seconds, to strategically place their ships on the game grid.

4. **Game Progression**
   The game progresses with turn-based logic. The first player is randomly selected when the game starts. Each player takes their turn, making a single shot on the grid. The user selects a coordinate on the grid where they suspect an opponent's ship may be located and attempts to hit it. After each shot, the game provides feedback on whether the shot was successful (hit) or missed (water).

1.3 Self-assessment Policy

For software testing, we have adopted an approach based on automated testing. We use JUnit in Java to develop and execute unit tests for all the components of the system. To enhance the reliability and maintainability of the server, a logging system has been implemented. This logger records all significant events and errors during the server's operation, crucial for debugging and monitoring the server's behaviour. This component ensures a clear and consistent log of interactions and game state, which is important for troubleshooting the server.

2 Background

This section explains the concepts and technologies involved in the development of our online Battleship game, which explores distributed systems and real-time communication in a web environment. Additionally, the project examines the **publish-subscribe pattern** and asynchronous communication. The theoretical foundations also include network latency handling and concurrency control, ensuring that the game remains consistent for all players, regardless of their network conditions.

In this project, we use several technologies:

- **Angular**: Chosen for its robust framework capabilities in building dynamic single-page applications (SPAs). Angular's data-binding and modular architecture facilitate the development of user interfaces [1].
- **Node.js**: Selected for its efficiency in handling asynchronous events, ideal for real-time applications like online games where performance and scalability are important [4].
- **WebSocket**: Implemented to enable real-time, bi-directional communication between the client and the server. This technology is crucial for ensuring that all actions in the game are transmitted instantly between players, a core requirement for the interactive nature of Battleship.
- **REST APIs**: Utilized for facilitating communication between the game's front end and back end. REST APIs provide a flexible, easily accessible method to handle various game actions and data exchanges.
- **JSON**: Standardizes all data exchanges between the client and the server in JSON format, ensuring that data is structured in a lightweight, easy-to-parse manner suitable for real-time processing. This standardization is crucial for maintaining the efficiency and clarity of data communication throughout the game.

3 Requirements Analysis

By examining the requirements described earlier, we can identify several essential functions that the system needs to have to meet the specifications exposed in the previous

section. In particular we identify some non trivial aspects that are described below and to be clear we organize them into 3 main groups: Connectivity, Usability and Game Functionalities.

3.0.1 Connectivity Functionalities

The functional requirements that involve the user are represented in use case diagram in Figure 1.

Functional Requirements

- **User Access**: The system should allow users to access the game using a web browser and register by providing a unique nickname. This nickname will be used to identify the user during gameplay.

  Solution: Use of ReST API to manage the users and their unique implementing exceptions if the choosen nickname is already used by someone.

- **Creation and Access to Lobbies**: Users should be able to create new lobby or join an existing one.

  Solution: To manage more than one lobby, these must have a unique identifier LobbyID and maintain a list of players that are currently in the lobby.

- **Game Notifications**: The system should manage the game state effectively, ensuring that all players are synchronized and aware of the current state of the game at all times. This includes tracking ship positions, remaining ships for each player, and updating the game grid after each shot.

  Solution: Implement a centralized game state manager on the server-side to store game state information. Use WebSocket communication to update clients in real-time whenever there is a change in the game state.

Non-Functional Requirements

- **Real-time Communication**: The system should support real-time communication among players during gameplay. This can be achieved using WebSocket to allow users to exchange instant messages, such as turn alerts, shots notifications, and game state updates.

  Solution: Implement WebSocket communication between the game clients and the server to enable real-time messaging. Each client can subscribe to relevant topics implemented in the server. In this case we use the **publish-subscribe pattern** where each player can connect to a certain topic, to organize all the players connections and messages they need to receive.

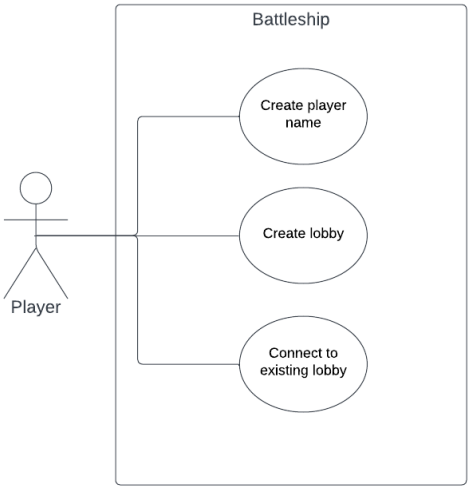

Figure 1: Connectivity use case diagram

- **Disconnection Handling**: The system should handle user disconnections robustly. If a player disconnects suddenly during a game, the system should exclude them from the current game and lobby, ensuring that the game can continue for other players without any failure.

  Solution: Implement a disconnection handling mechanism where the server websocket service monitors the status of each player. If a player disconnects unexpectedly, the server should remove it from the current game and lobby, allowing the game to continue without interruption.

- **Authentication**: The server must know who is connected and have to identify each player in order to allow the proper messaging between the nodes of the system.

  Solution: Each player (client) must be identified by a sessionID used for the communications management.

- **Service Availability - Fault Tolerance**: The system should be available also in case of fault of a service.

  Solution: The system must be redundant and in case of failure of a service another identical one must be reachable and usable. To be sure that the service is always available implementing a health check system is crucial for monitoring the status of services and components.

3.1 Usability functionalities

- **User Interface (UI)**: The system should provide an easy interface for players to interact with during gameplay. This includes features such as displaying the

game grid, ship positioning phase, and turn-based gameplay elements to better understand the logic of the game.

Solution: In this case we need to understand the UI should provide feedback to players on game state changes, like successful hits or misses and display relevant information such as remaining time for ship positioning, turn countdown amd notifications.

3.1.1 Game Functionalities

The functional requirements that involve the user are represented in use case diagram in Figure 2.

Functional Requirements

- **Ship Positioning Phase**: During this phase, players must strategically position their ships on the game grid within a set time limit.
- **Shot Handling:** During this phase, the user must be capable to shot one or more player, choosing a target cell.
- **Exit the game** The user must be capable to exit the game at the end of the game to return to the lobby page.

Non-Functional Requirements

- **Turn-based Game Progression**: The game progresses in a turn-based manner, with each player having 15 seconds to make the shot. After each turn, the system provides feedback on the success or failure of the shot.

  Solution: Implements timer-countdown to handle turns and create a notifications system that use websocket to communicate in real-time.

- **End of Game**: The game ends when only a player remains in game. The system should be able to determine the winner.
- **Error and Exception Handling - Reliability**: The system should handle errors and exceptions carefully during gameplay, ensuring that all unexpected behaviors are handled appropriately without compromising the integrity and the availability of the game caused by unhandled exceptions.
  Solution: Implement logging and error handling on the server side.

Anlizyng the requirements we choose to create an client-server architecture using the server as a service that expose a RestAPI who use HTTP communication protocol for the non critical operations such lobby creations and username creation. Instead, during game phase and for the critical operation that require real-time we choose to implement Websocket between the client and server using **publish-subscribe pattern** where each

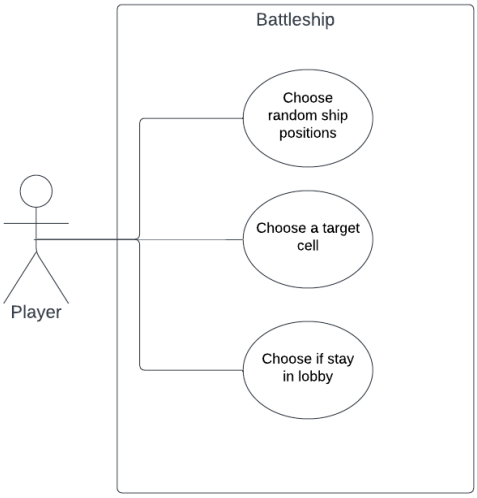

Figure 2: Game use case diagram

client can subscribe to relevant topic to manage and make sorted the multiple communications. For the client side we choose to use Angular[1] and Node.js [5] as environment to execute the web-application.

4 Design

After analyzing the requirements, we proceed to illustrate the design adopted for the development of Battleship Free-For-All.

As it is designed as an online multiplayer game, it has been decided to maintain a high level regarding the protocols and communication technologies between the various components of the system, making the application more easily deployable in a real-world context.

4.1 Structure

The figure 3 shows the structure of the Model components through a UML class diagram, in which only the most significant fields and methods are shown for each entity.

Considering the **Player** entity, it encapsulates the concept of a player identified by a nickname and a sessionId, and its `PlayerRole` attribute identifies the role within the system. The PlayerRole attribute refers to an enumeration that identifies possible roles: `ATTACKER` for the user who has to choose the coordinates to attack, `DEFENDER` represents the role of a defender player, and `TURN_WAITER` is a special status used before the game starts. All the players have also a PlayerStatus: `NOT_IN_GAME` when users simply connected to the system but not playing and `IN_GAME` for a user who is taking

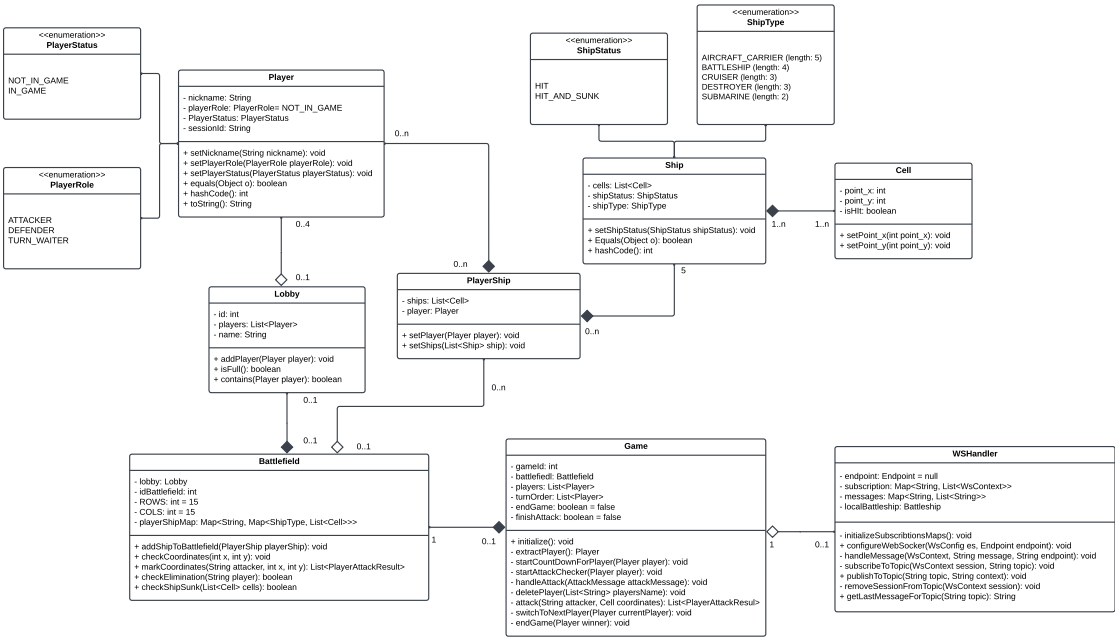

Figure 3: UML class diagram.

part to a battle. The User entity also possesses methods to properly set its role within the system.

Regarding the **Lobby** entity, as inferred from the requirements analysis, it is considered full when it is populated by exactly four users, the battle will begin only when the lobby is full. The lobby has an id that uniquely identifies it and a list of players connected to it. Furthermore, it has the method `addPlayer(Player palyer)` to add players, the method `isFull()` to check if it is full, and the method `contains(player)` to check if a player is taking part to the lobby.

The **Ship** entity is the model that describes each ship that can be placed on the battlefield. There are different types of ships, represetented in `ShipType` enumeration, and each of them has a predefined length. Moreover, each boat can have different states, `shipStatus`, such as `HIT` if at least one cell has been attacked, `HIT_AND_SUNK` if it has been entirely attacked.

Considering the **PlayerShip** entity, it encapsulates the concept of fleet; each player has five ships of different type for their battle.

Regarding the **Battlefield** entity, there is one for each lobby and indicates the size of the battlefield (`ROWS` and `COLS`). It is also identified by a unique id. Additionally, it has a reference to all players in the battle, along with the positioning of their ships. This class had a method to add the player and their ships `addShipToBattlefield(PlayerShip playerShip)`, `checkShipSunk(List<Cell> cells)` returns true if the ship was hit and sunk.

Finally, the most complex entity is presented, which is also central to the structure of the Model, the **Game** entity. It possesses all the elements described during the

requirements analysis phase. It contains within it: references to the players, the results, the current turn and the ships to be hit. Additionally, the Game entity possesses all the necessary methods for the progression of the game:

- `initialize()`: initialize the status of all the players and chooses the one who will start the turn.
- `extractPlayer()`: extract the first player that have to choose the coordinates to attack.
- `startCountdownForPlayer(Player player)`: each player has only 15 seconds to decide the cell to attack.
- `startAttackChecker(Player player)`: checks if an attack was published in the webSocket topic.
- `handleAttack(AttackMessage attackMessage)`: it manages the attack executed by a player.
- `deletePlayer(List<String> playersNames)`: the player who has lost all their ships is eliminated.
- `attack(String attacker, Cell coordinates)`: check and attack the cell.
- `switchToNextPlayer(Player currentPlayer)`: change the role of a player from `DEFENDER` to `ATTACKER` if it is his turn.
- `endGame(Player winner)`: change the status of all the players in `NOT_IN_GAME`.

4.2 Behaviour

In Figure 4, a state diagram is shown that exemplifies the behavior of the Lobby entity following interaction with the player. As described in the requirements, it is possible to create a new lobby or connect to an existing one. Ones the player connects to a lobby, he remains waiting for another players to connect to the lobby. Once the lobby is full and all the players are ready, the players have to wait 5 seconds and then they can choose how to position their ships on the grid, they have only 15 seconds. Ones it ends, the game will start in 5 seconds. We've decided to include small countdowns, like the three seconds after the lobby is full, to ensure real-time synchronization for all players and to prevent anyone from falling behind with the system state. All topic related to the specific countdown must have all players subsbribed before start. Once the game is finished, the main menu is shown again.

In Figure 5, the state diagram shows the behavior of the Game entity during the progression of a game. Since the game is structured in turn, the dynamics depend on the role assumed by the player. If the user is assigned the role of `ATTAKCER`, they are asked to choose a cell to attack and then placed in a waiting state where they view the result of his attack. If the user is identified as `DEFENDER`, they enter a state of waiting

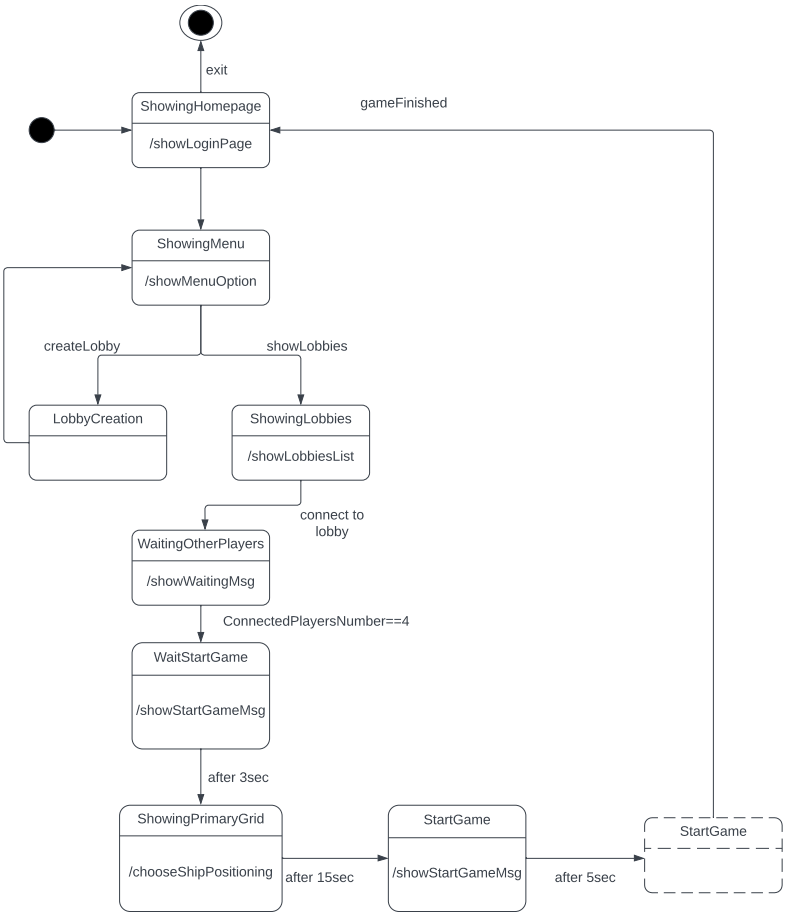

Figure 4: Lobby state diagram.

Figure 5: Game state diagram.

for the cell that the attacker want to attack. Once it is chosen, the `DEFENDER` is shown in his grid the cell attacked if it is included in his ships. After a turn, roles are reassigned, and the same dynamics proceed in subsequent rounds. When one of the four players remains with at least one ship not sunk, the game finish.

4.3 Interaction

Now, let's delve into the structure of interactions within the system. In Figure 6, a sequence diagram illustrates the creation of a lobby and the connection to it. Every resource in the system, including the server-side Lobby, is equipped with a Controller and a suite of APIs to manage client requests. When Client1 initiates a POST request to the server to establish the lobby, it is routed to the `LobbyController`, which then invokes the appropriate API within `LobbyApi`. This API subsequently interacts with the local instance of Battleship on the server. Responses to these requests follow a consistent chain of interactions among the system's various components. Similarly, the Player and Game entities feature their respective Controllers and APIs. Consequently, all client requests and responses are channeled to the relevant Controllers, maintaining the sequence of interactions outlined in the diagram.

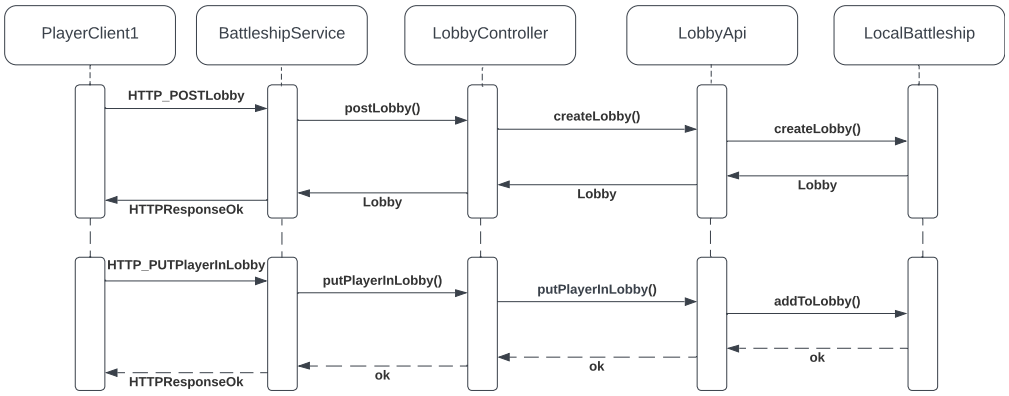

Figure 6: Lobby sequence diagram.

Figure 4 shows the sequence diagram for the message exchange during an attack in the game, utilizing the **BattleshipWebSocket**. This main class is responsible for preparing the **WebSocketService** implemented in this project. Connection handling is managed by **WsHandler**, which configures the websocket. Before subscribing to any topics, each client must first establish a websocket connection to the */connection* path for authentication using a **sessionID** provided by the server. The **WsHandler** utilizes this **sessionID** to identify the subscribing client and manage their topic subscriptions.

At the start of each game, specific topics are created as detailed in the implementation section that follows. For this scenario, the relevant topics are */lobbyID/attack* for the attacker that send a message to this specific topic, and */lobbyId/notification* for all the other clients to manage real-time notifications.

Using the publish-subscribe pattern, all clients can subscribe to a specific topic to receive all messages published under it. In this scenario, when a **Client Attacker** publishes a message to */gameID/attack*, the **WsHandler** evaluates whether the message is valid and send it to the Game Implementation that generate an **AttackResultMessage** for the Attacker. Simultaneously, in the same way **WsHandler** also produces a **NotificationMessage** to be sent to all clients subscribed to the */lobbyID/notification* topic, as you can see in figure 7.

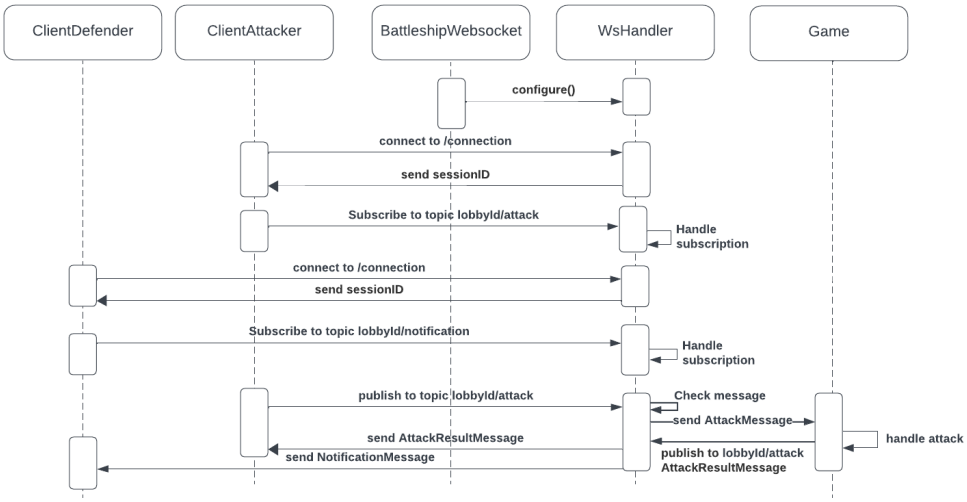

Figure 7: WebSocket attack sequence diagram.

In the sequence diagram in figure 8, the process of placing the ships within the grid is shown. Once the lobby is full, the client subscribes to the topic to start the countdown for the beginning of the game, *fullLobbyCountdown/time, which* lasts three seconds. Subsequently, each player has 15 seconds to choose how to position their ships. This countdown is also initiated through a subscription, *battlefieldPositioning/time*. At the end, an API call will be made to save all the selected cells, and another subscription will be made to trigger a 5-second countdown for the start of the battle, *gameStartsIn/time*. This time management has been designed to ensure that everyone is aligned and to better manage the real-time aspect of the game.

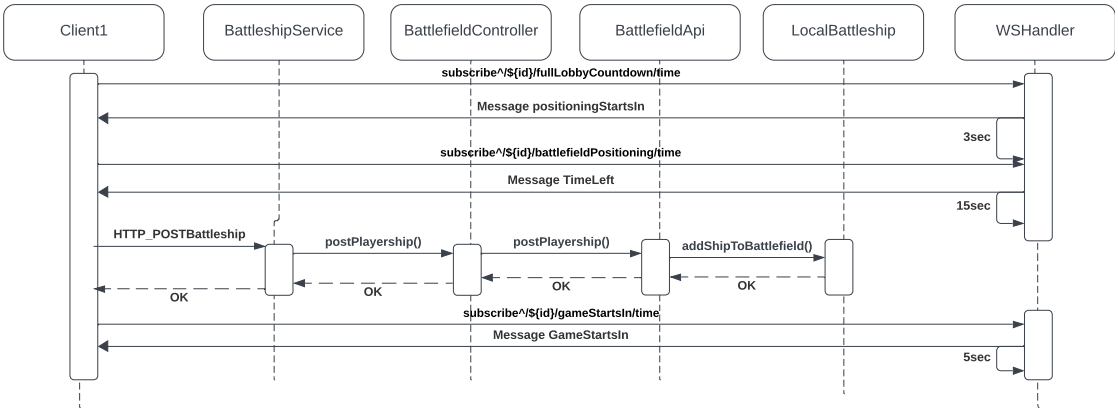

Figure 8: Positioning ships sequence diagram.

5 Implementation Details

This section will review the implementation strategies used and describe briefly the technologies used. The project was developed in Java for the Server Side and Typescript, HTML and CSS for the client side using Angular Framework. We can differentiate this projects in two main parts: Client and Server.

5.1 Client

The client has been implemented using Angular, an open-source framework for web application development.

This section contains all the methods necessary to make HTTP requests to the server, referencing the URLs needed to keep the game page updated, insert new data, and modify certain states:

- */players*: refers to all requests made for player management (get all players, connect a new one).
- */lobbies*: requests for managing the lobbies (get all lobbies, create a new one, connect a player etc.).
- */battlefield*: used to insert the player's fleet once the ships are positioned.
- */healthcheck*: to check every 10 seconds the availability of the server.

Regarding game management, the *WebSocketService* has been created using the *publish/subscribe pattern*. This allows the client to subscribe to the topic of its battle, receiving all messages published by the server to keep it updated on the game's progress, turn changes, attack results, whether it has been hit, and more.

In the client part, controls have been implemented to verify that the main server is still active, both for API requests (port 10000) and for WebSocket (port 7070). If not,

the same request will be attempted on the secondary service (API port 8181, WebSocket 9090). However, a small implementation issue has been encountered due to CORS Origin errors. Despite several attempts, unfortunately, this control only partially works for API requests; POST and PUT requests always return a CORS error. However, WebSocket functionality works correctly.

5.2 Server

The server has been developed using the Javalin framework [3] and it expose two services: BattleshipService and WebSocketService.

BattleshipService contains different resources, each equipped with a Controller and a set of APIs that, as mentioned, allow the Client to interact with the Server. Specifically, for each resource, the corresponding Controller is registered at the designated address, allowing it to receive Client requests and redirect them to the correct component. The APIs, on the other hand, interact with a local instance of the system.

All routes registered in the Web Server have been documented using Swagger [6] and can be viewed by following the path *localhost:10000/swagger* after the server is started.

BattleshipWebSocket is responsible for handling all WebSocket communications between the client and the server. This service integrates with the Battleship game logic through its interaction with the local instance of the Battleship model. The server is set up to handle multiple WebSocket endpoints, each of which is dedicated to a specific part of the game process and contains different topics where each client can subscribe or publish messages:

- **/connection**: This endpoint is designed for initial WebSocket connections, where clients can establish and maintain a stable connection to the server. It might also handle authentication sending back the sessionID when a new client connect to the /connection endpoint.
- **/lobby**: The lobby endpoint manages all the topics related to lobby for example the ones related to specific countdown in lobby before the start of the game or the ships positioning countdown. The topics are: */{lobbyId}/fullLobbyCountdown/time*, */{lobbyId}/battlefieldPositioning/time*, */{lobbyId}/gameStartsIn/time*
- **/game**: Dedicated to the gameplay itself, this endpoint handles all game-related topics such as countdown timing and attack message. The topics created for this phase are: */{gameId}/notification*, */{gameId}/turn/time*, */{gameId}/attack*,

Therefore, each topic will be exclusive to a game, so that messages can be sent and received only by players in the current game or lobby, avoiding broadcast messages to all the client connected to the server.

For the *presentation* of both services we use JSON format and to manage the serialization and deserialization of JSON objects in our project, we have chosen to use the

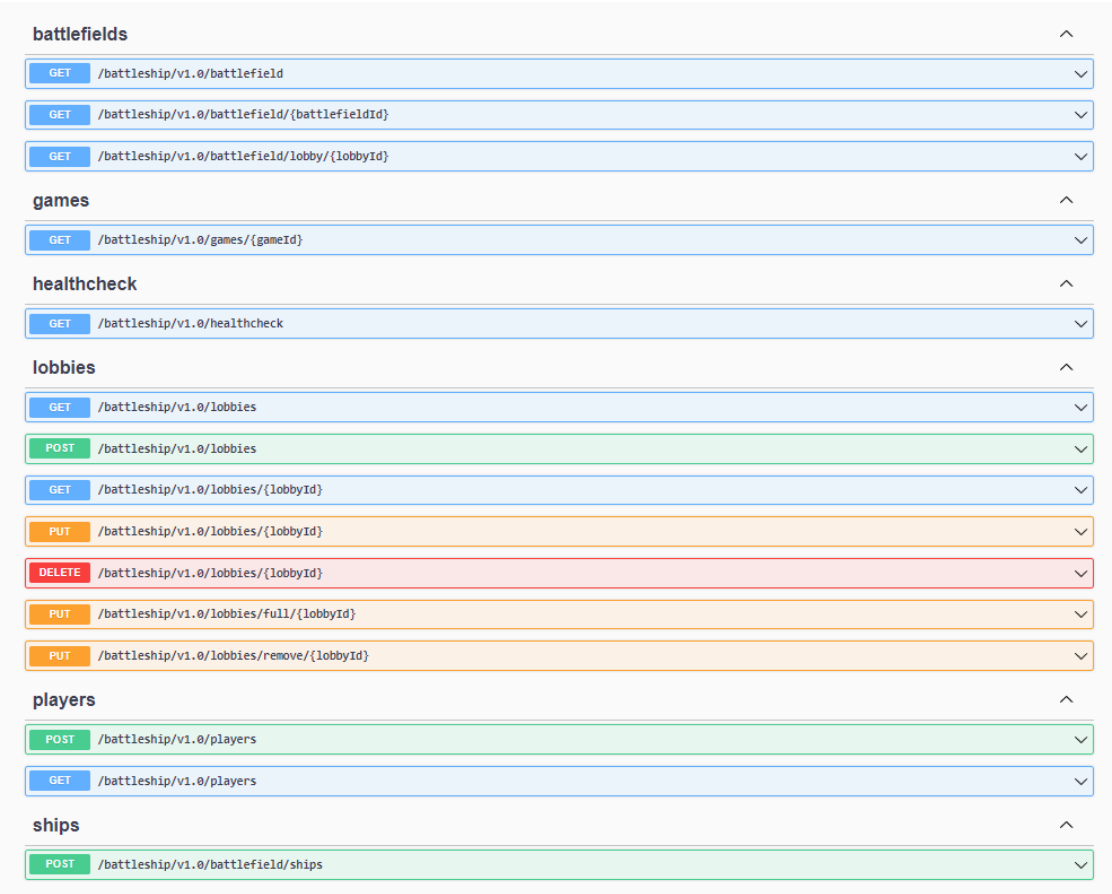

Figure 9: Apis.

Jackson library [2]. Jackson offers control over the serialization and deserialization process through the use of Java annotations. For example, annotations like **@JsonCreator** and **@JsonProperty** allow us to manage constructors and properties very specifically directly inside the Java class of the object, facilitating mapping.

6 Self-assessment / Validation

Regarding testing, in addition to manual tests, automated tests have been developed using the JUnit framework. Within the project, there is a module called "test" containing four classes developed for testing purposes.

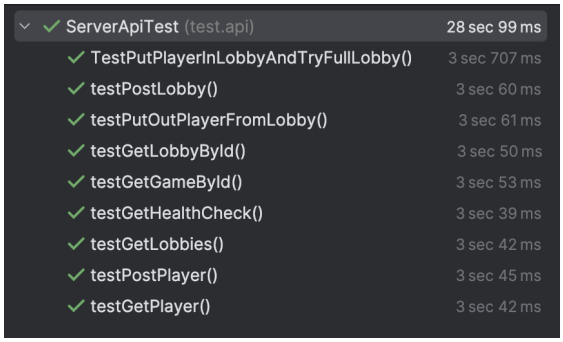

Figure 10: Test API.

The class in figure 10, all API calls made to the server for the Lobby, Player, Game entities, and the server's Healthcheck are tested. Each test verifies all the different responses that we can receive from the server. Basic tests are present to verify the correct connection and disconnection of users to the system, the proper creation and deletion of lobbies, and the control of exception handling in case of errors.

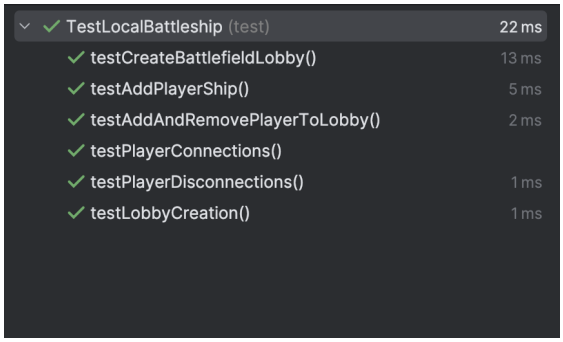

Figure 11: Test Local Battleship.

Furthermore, advanced tests are present to simulate a use case of the system by different players. These tests cover: connection and disconnection to a lobby, creation

of the PlayerShip to insert in the battlefield and the start of a game with the random generation of the turns.

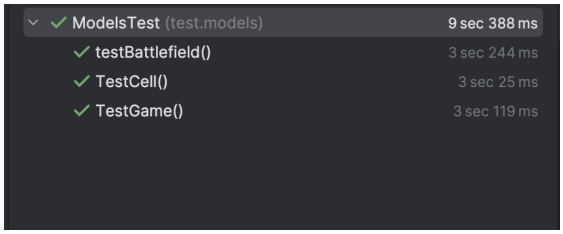

Figure 12: Model test.

In addition, figure 12, the main models have been tested to verify that the game logic functions correctly. The implementation resembles that of Battlefield, with its attack and defense logic, along with the game allowing players to insert their battlefield and connect via WebSocket.

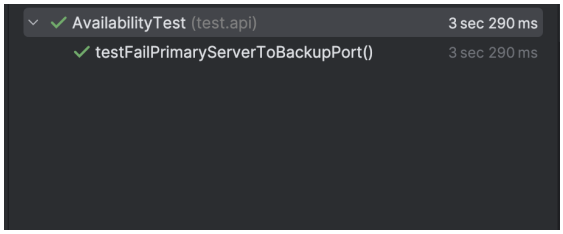

Figure 13: Availability test.

Fault tolerance is the ability of a system to continue operating without interruption in the event of a failure or fault. To ensure fault tolerance, a second support server has been created, which comes into operation when the first one loses connection. The second server is updated and kept aligned with the game to guarantee continuity. A test has been included to verify that this process works correctly, figure 13.

7 Deployment Instructions

To start the application it's needed to start both the server and the client.

Server Prerequisites and execution

- Machine with JVM Machine

To start the server, open terminal, position inside the Project folder **Battleship** and execute the following command **./gradlew –console plain server:run**

Client Prerequisites and execution

- Node.js installed on the machine (npm command)

To start the client, open terminal, position inside the client folder **Battleship/client/web pages**. Execute **npm install**, wait for the dependecies to be installed and then execute the **npm start**

This command ensure that the client application starts on Node.js local server and it can be reacheable at **localhost:4200** using a web browser like Chrome.

8 Usage Examples

When starting a client, the system prompts for entering a nickname, and if the nickname does not generate conflicts, the menu is displayed.

Figure 14: Choose Nickname page.

Once the nickname is chosen, we can access the page where all existing lobbies are displayed and optionally create a new one. Polling was chosen to request the status of the lobbies from the server because it offers a straightforward and efficient method of keeping clients informed about the ongoing status of the lobbies, eliminating the need for dependency on notification events or websockets.

Once the lobby is full, and all the players are sincronized, each player can choose how to position their ship on the grid. They have only 15 seconds and then the battle will start, figure 16.

The last main page, is the one of the battle. The grid on the left shows the status of the player ships, and the one on the right, the grid to attack the other players. Each cell will change color in order to the result of the attack, `WATER` in blue, `HIT` and `HIT_AND_SUNK` in orange.

At the top, the name of the player who has the turn to attack is displayed, while they see the remaining seconds to select the cell. If the player attacks before the end of the

Figure 15: Lobbies page.

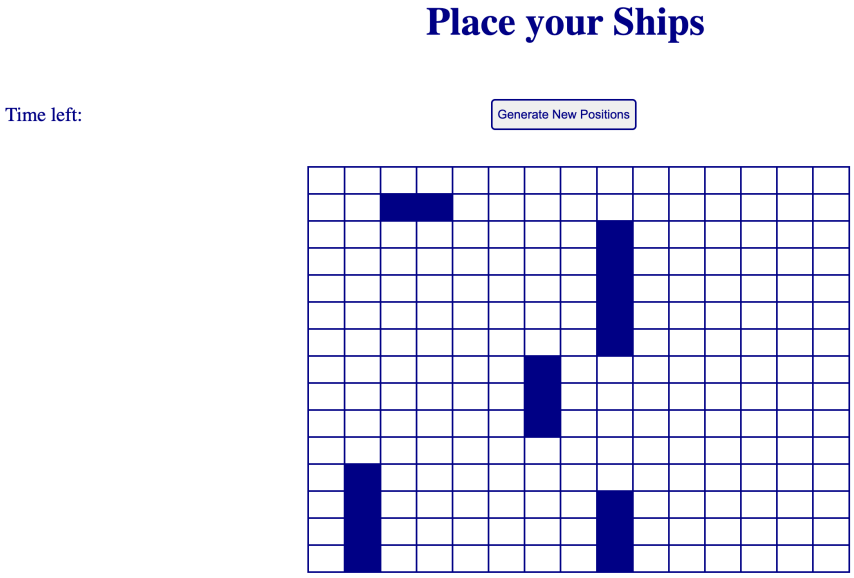

Figure 16: Positioning ships grid.

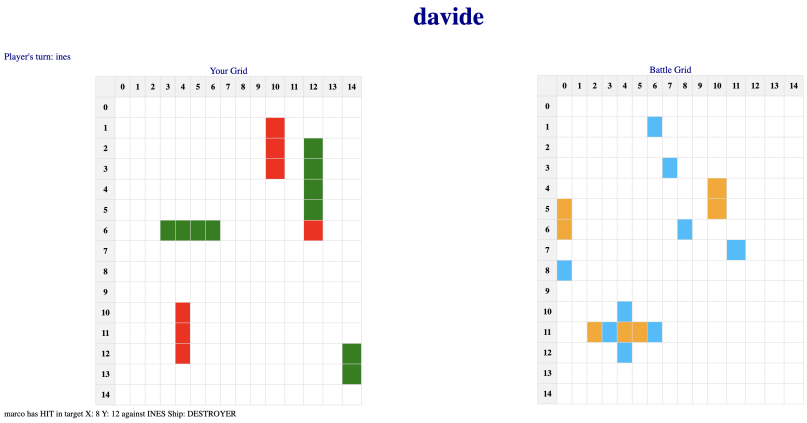

Figure 17: Battle page.

turn, the attack is sent, and at the bottom, a notification is displayed to show all players the outcome, then it moves to the next player.

Once the player loses all their ships, they can decide whether to stay and watch how the battle progresses or return to the main menu to create or join another lobby.

9 Conclusions

In this project, we successfully transformed the classic game of Battleship into a dynamic, multiplayer free-for-all experience, designed to support four players in a shared virtual battlefield. Our primary goal was to enhance the traditional two-player game format by increasing its complexity and strategic depth through a modern online platform. We developed the game using Angular for the frontend and Node.js for the backend, with WebSocket technology facilitating real-time communication between players. This setup adhered to the publish-subscribe pattern to manage real-time updates, ensuring all players received consistent and timely information about the game state. The system we implemented allows players to register with unique nicknames, join or create game lobbies, and engage in strategic gameplay involving ship positioning on a 15x15 grid and turn-based combat with real-time decision-making enforced by a timing mechanism. We also ensured the backend architecture could handle simultaneous connections and disconnections. Through this project, we have successfully achieved our objectives and gaining invaluable insights into the complexities of real-time communications and distributed system design.

9.1 Future Works

This project gives a lot of ideas in future works, some interesting could be the following:

- Allow players to select specific opponents to target within the game. This feature

would ensure that a shot does not hit ships in the same cell by accident but targets only those belonging to the indicated player. This would add a strategic layer to the game.

- Containerization with Docker: Implementing Docker for containerization could streamline deployment and scalability of the game. By containerizing the game server and other components, we can ensure a consistent environment across different development and production platforms, simplify updates, and enhance the scalability to handle an increased number of concurrent players without compromising performance.

- Dedicated Game Statistics Database: To further enhance the game, implementing a dedicated database for storing game statistics could be very beneficial. This database would track various metrics such as number of games played, win/loss ratios, player rankings, and more detailed statistics like accuracy or favorite targets. These statistics not only foster a competitive environment but also provide valuable data that can be used to refine game balance and enhance user experience.

9.2 What did we learned

During the development of battleship game, we learned to use several technologies and programming concepts. Angular allowed us to create a dynamic user interface, facilitating the management of data binding and components to synchronize the game state between the client and server. We utilized Node.js for the backend, leveraging its capability to handle non-blocking I/O operations, essential in an online gaming context. Real-time communication was implemented using WebSockets, ensuring a constant flow of data between players, crucial for an interactive game like battleship. The publish-subscribe pattern was another important aspect of our project, allowing us to efficiently update all connected clients without overloading the server. REST APIs played a vital role in the interaction between the frontend and backend, managing the operations necessary for game control. Additionally, we improve our understanding and use of JSON, essential for exchanging data between different parts of our system in a readable and easily manageable format. In this project, we also realized the critical importance of timing in distributed systems, where delays can significantly impact user experience.

References

[1] Angular website. `https://angular.io/`.

[2] Jackson github repository. `https://github.com/FasterXML/jackson`.

[3] Javalin website. `https://javalin.io/`.

[4] Node.js reference. `https://a-team.global/blog/main-benefits-and-limitations-of-node-js/`.

[5] Node.js website. https://nodejs.org/en.

[6] Swagger. https://swagger.io/.