UNIVERSITÀ DI PISA

Distributed Systems and Middleware Technologies Project

Federico Minniti

Tommaso Baldi

Matteo Del Seppia

Academic Year 2021-2022

Contents

1 Requirements **3**
1.1 Functional requirements 3
1.2 Non-functional requirements 3
1.2.1 Availability 3
1.2.2 Usability and low latency 3
1.2.3 Erlang 3
1.2.4 Coordination 3

2 Architecture **4**
2.1 Java Enterprise Module 5
2.2 Erlang module 6

3 Concurrency and Message Exchange **7**
3.1 Homepage 7
3.2 Game 7

Link to the web-app: http://172.18.0.46:8080/webapp-battleship/
Link to Github: https://github.com/federicominniti/Battleship

1 Requirements

1.1 Functional requirements

Battleship is a web application allowing registered users to play the famous battleships game against each other.
The only actor of the application is the registered user.
Anonymous users are required to sign up to the application and then login with their credentials to access the services offered by the web-app.
Once logged in, a user can start playing battleship either against a random opponent or wait for another online user to accept their battle request. When a user accepts a battle request, the game starts for the player who sent the request and the one who accepted it. A user can also decide to decline a battle request. Battle requests can be sent only to users present in the list of online users. The list of online users is shown to logged users in the application homepage, together with the logged user past scores and a ranking of the best 10 registered users given their wins/defeats ratio.
During the game, opponents are given a minute to play their move. If the time expires before the user has made any move, the turn goes to the opponent. The two opponents can chat with each other during the game.
If one of the two opponents disconnects or surrenders, their game is considered lost and their opponent wins.
After the end of a game, users are redirected to a page displaying the result and they can return to the homepage and start a new game.

1.2 Non-functional requirements

1.2.1 Availability

The application must be able to handle several players gaming and chatting and the same time without problems.

1.2.2 Usability and low latency

The application must be easy to use, with low response times.

1.2.3 Erlang

The application must exploit the functionalities offered by Erlang regarding concurrency and message-passing between processes.

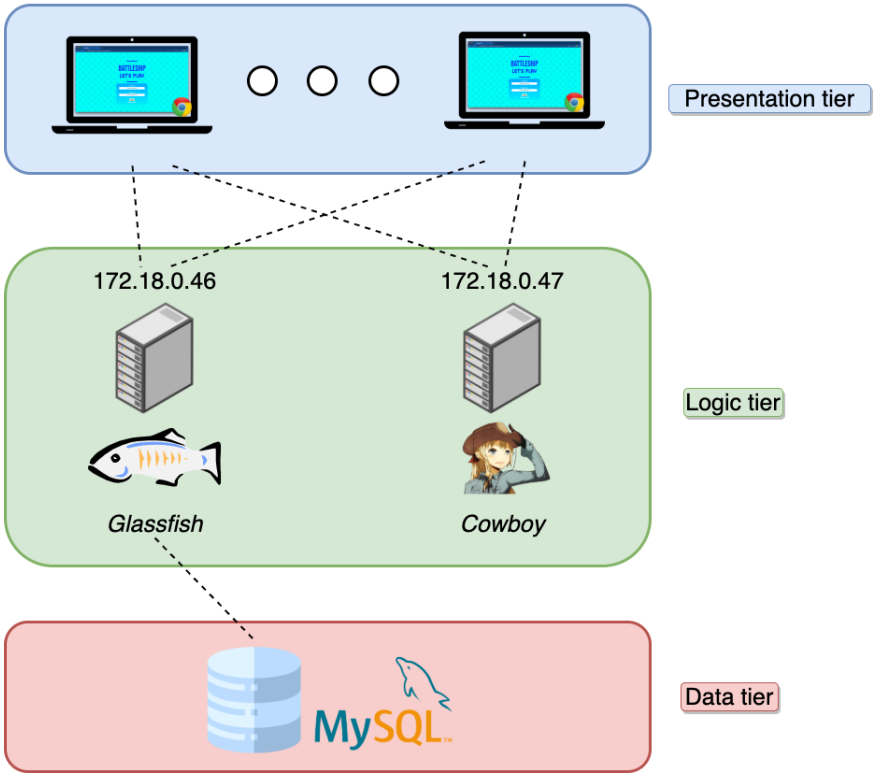
1.2.4 Coordination

The application must handle the coordination of processes in a distributed environment.

2 Architecture

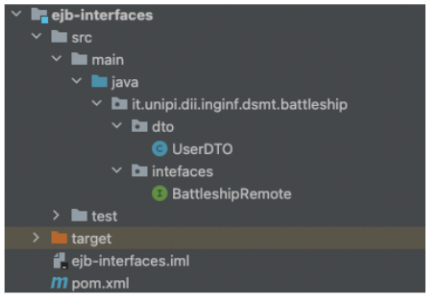
Battleship has a client-server architecture. The application is structured in three tiers:

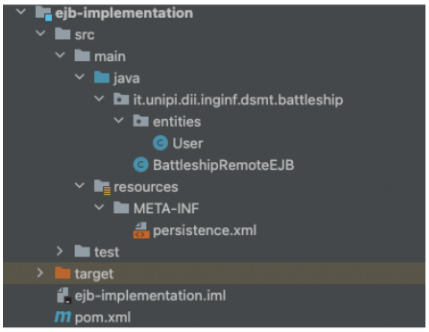
- **Presentation**: this is the front-end layer (GUI) offered to users. It is built as a standard web-application in HTML, CSS and Javascript.
- **Logic**: this is the layer supporting the application core function. It consists of two modules: one written in Java, hosted on a Glassfish web-server, and one written in Erlang, running a Cowboy web-socket server.
- **Data**: This layer exploits EJB/JPA to communicate with the MySQL server specified in the Glassfish container. EJB has been used to offer services for persistence of data to the servlets, which are not aware of how the services the database are implemented, but only know which methods are callable. In this way we can also export our application on a different platform, for example a mobile application, and use the same EJB interface for the persistence of data. JPA has been implemented through the Hibernate framework.

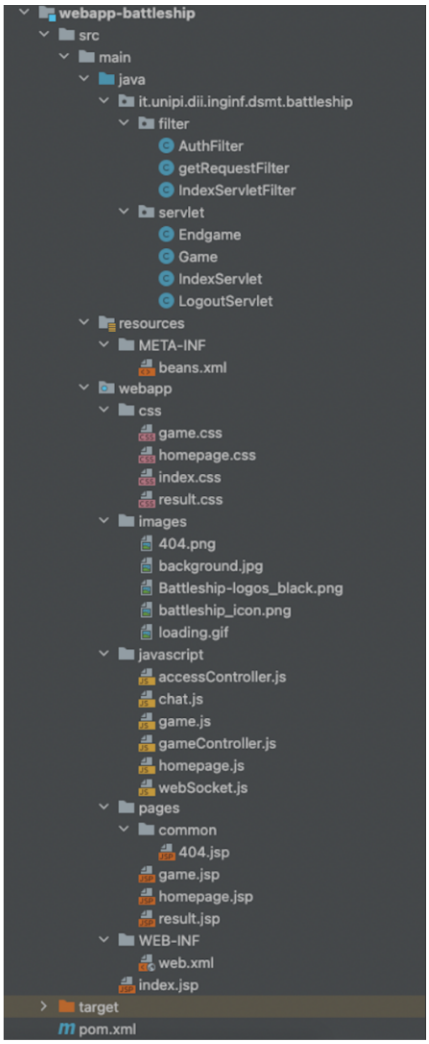


2.1 Java Enterprise Module

The software model of the Logic tier is the common MVC design pattern.
Requests from the clients' browsers are filtered and handled by web Filters and Servlets.
Filters have been used to make checks on the type of request the user are making to the server: for example, the AuthFilter checks that only logged users will be able to access the pages offered to only registered users.
Servlets have been used to handle the logic of the requests made by clients: for example, the IndexServlet accepts POST requests by users trying to login with the application and has the responsibility of using the EJB interface to validate login credentials and provide the user a different response based on the success or failure of the validation.







2.2 Erlang module

Cowboy is a fast (has low latency and low memory usage, in part thanks to the fact that it uses binary strings) HTTP server for Erlang/OTP. Cowboy provides a full HTTP stack. Cowboy provides routing capabilities and selectively dispatching requests to handlers written in Erlang. The concurrency of the Web is handled by Erlang.
Cowboy in Battleship is used to handle web socket requests, that are extensions to HTTP that emulates plain TCP connections between the client, the Web browser, and the server.
The Erlang module uses Cowboy to build a web-socket server that will be used by clients to exchange messages and by the server itself to maintain information about the state of the list of online users.
In fact, at the start of the Cowboy web-socket server also two other processes will be started: one in charge of handling a list of online users and the other in charge of searching a random opponent in the list of online users.
A specific process will be associated for each client connected to the web-socket server (from Cowboy), and it will be able to communicate, through the exchange of messages, with the processes associated to the other clients. Clients' processes are registered with the respective usernames, thanks to the fact that they must be unique.

The structure of the messages sent over the web-sockets is:

- **type**: the type of the message
- **data**: the payload of the message, if present
- **sender**: the username of the sender
- **receiver**: the username of the receiver, if present

3 Concurrency and Message Exchange

Our project uses web-sockets to exchange messages between clients and between clients and the Erlang daemons. Frames can be sent at any time, in a fully asynchronous way, without any restriction.
In general, all messages are sent to the Cowboy web-socket server, which uses the callbacks defined by us to route messages to the correct processes, based on the type of the message.

3.1 Homepage

When a user enters the homepage, it has to signal to the Erlang web server that they are to be put in the list of online users. A message of type *user_online* will be sent to the web-socket server and the Erlang handler will put them in the list. Afterwards, the handler will send the updated list of online users to all the users in the list.
Messages of type *battleship_request* and *battleship_accepted*, respectively sent to the Erlang web-server when the user sends a game request or accepts a game request, are handled by simply redirecting the message to the receiver specified in the message.
A message of type *decline_battle_request* is sent to the Erlang web-server by the user declining a battle request. The Erlang web-server will redirect the message to the user who had sent the battle request, to communicate that a new request may now be sent to the user who just declined.
To implement the search of a random opponent, a particular message of type *random_opponent* is sent to the Erlang web-server. If there is already a client process waiting for a random opponent, the server sends to both the processes a message of *battleship_accepted*, consequently starting a game for both the players against each other.

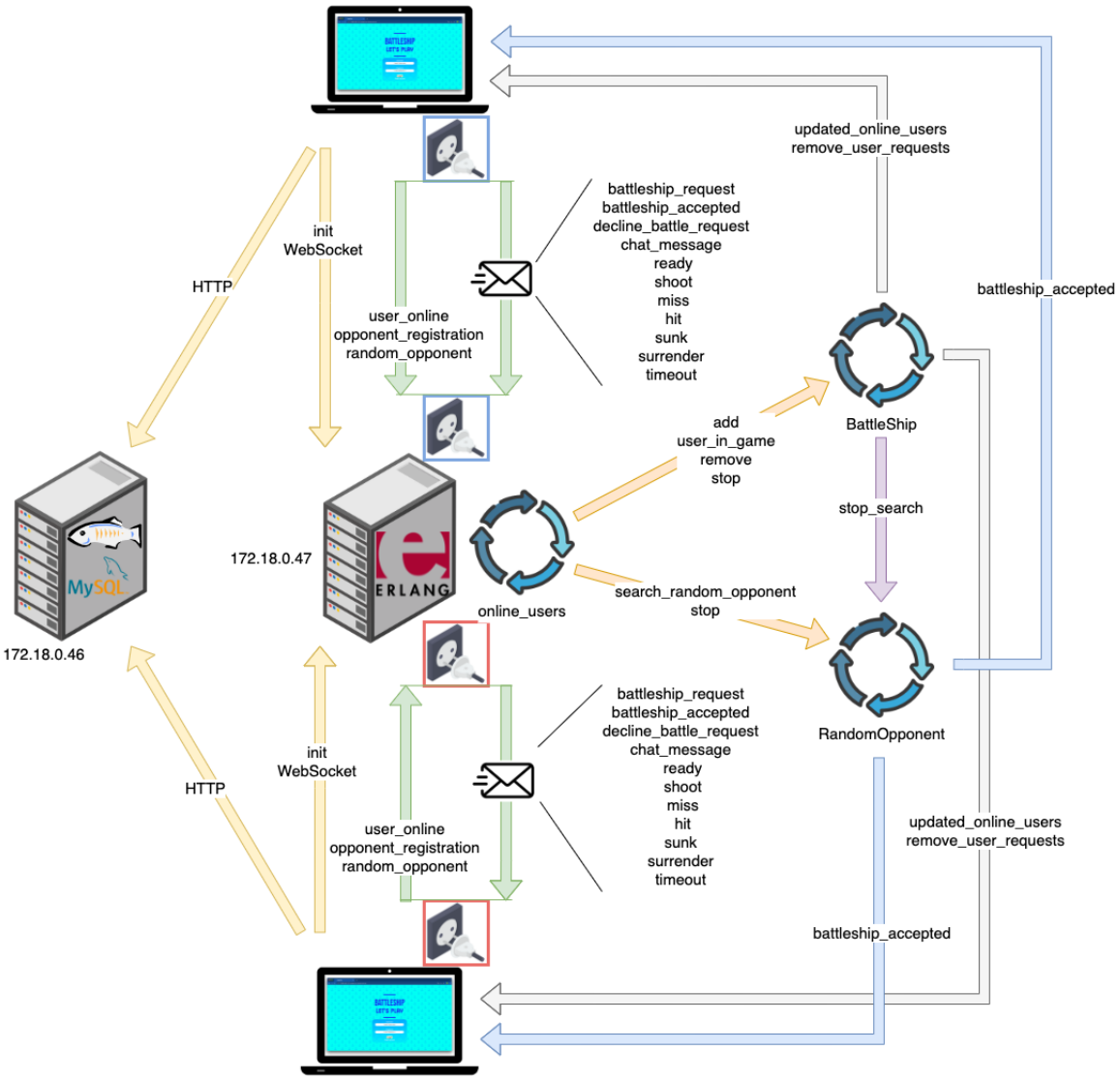
3.2 Game

During the game, several type of messages are exchanged between the clients over the web-socket server to coordinate the game interactions. The first message sent by the user is the *opponent_registration*, which is used to communicate to the handler of online users that the player is currently in game and should not be shown in the list of online users.
To decide which player has the first turn, a message of type *ready* containing a number between 1 and 10000 is exchanged between the two clients. Each client checks if its own number is bigger than the number sent by the opponent. The client who generated the highest number will take the first turn.
In general, messages of type *chat_message* are redirected by the Erlang web-server to the specified receiver, implementing a simple live chat between the two players.
When a user shoots on the enemy field, a message of type *shoot* containing the coordinates of the targeted cell on the grid is sent to the opponent, which can

respond with three different type of messages:

- *hit*: contains also the coordinates of the hit cell
- *miss*: contains also the coordinates of the cell containing water
- *sunk*: contains also the coordinates of the cell hit and the length of the ship that has been sunk

If a user doesn't act during his turn, a message of type *timeout* will be sent to the opponent to pass the turn.
At any time during the game, a player may surrend. In case of surrend, a message of type *surrender* is sent to the opponent and the game ends instantly. The player are then redirected to the result page and then to the homepage, where they are reinserted in the list of online users.


updated_online_users
remove_user_requests
battleship_request
battleship_accepted
decline_battle_request
chat_message
ready
shoot
miss
hit
sunk
surrender
timeout
init
WebSocket
HTTP
user_online
opponent_registration
random_opponent
battleship_accepted
BattleShip
add
user_in_game
remove
stop
stop_search
172.18.0.47
online_users
search_random_opponent
stop
172.18.0.46
RandomOpponent
battleship_request
battleship_accepted
decline_battle_request
chat_message
ready
shoot
miss
hit
sunk
surrender
timeout
init
WebSocket
HTTP
user_online
opponent_registration
random_opponent
updated_online_users
remove_user_requests
battleship_accepted