UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

Distributed Systems and Middleware Technologies

PROJECT DOCUMENTATION

Design and development of "SinkAndWin"

A Distributed Application using Java EE and Erlang

Students

Luana Bussu

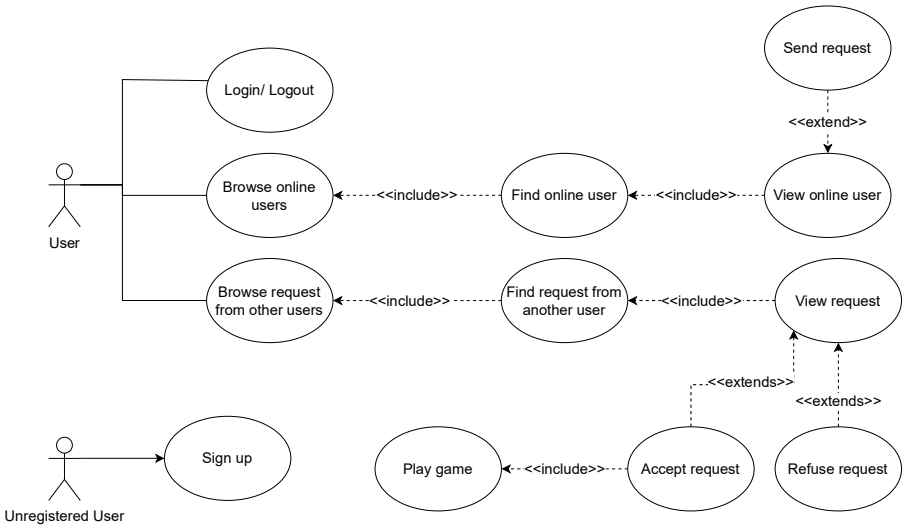
Roberta Matrella

Academic Year 2021/2022

1. APPLICATION REQUIREMENTS

1.1 FUNCTIONAL REQUIREMENTS

SinkAndWin is a web application that allows registered users to play the famous battleship game.



Once the user sign in he can:

- Browse the list of the online users
- Select a user to challenge him
- Receive a request from another user
- Accept a request from a user
- Play a battle after the game request has been accepted
- Log out

When a player accepts a game request, the two players are both redirect to the game page: as first step the system place randomly the ships into the battle grid and the game can start (the player who sent the request makes the first move). During his turn, the player tries to hit the opponent ships selecting a cell of the battleground, after which a message is displayed: "missed" if the cell was empty, "hit" if there was a ship and "hit and sunk" if the last cell of a ship has been hit. The game ends when one of the players sinks all the opponent ships; after that a message is displayed to both the players saying which player won. When the game is over, the user is sent back to the game requests page to choose another opponent and check his new score.

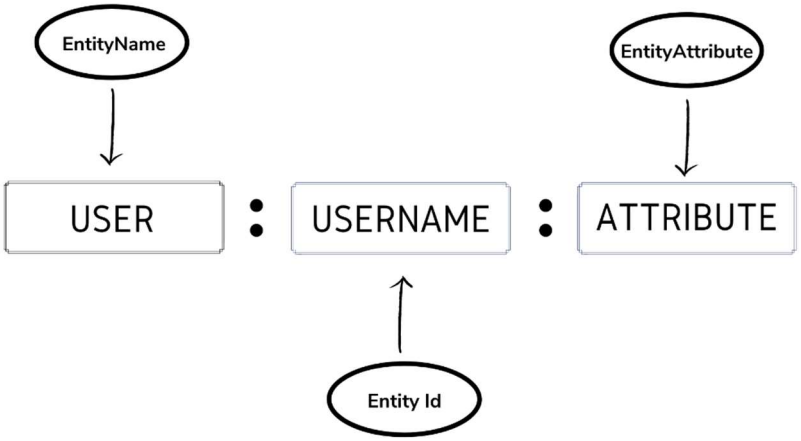
2. DATA MODEL

The application only handles user information as username, password and points and operations on data are easy and not computational heavy. For these reasons the application doesn't require a complex data structure, so we decided to adopt a key-value database that guarantees faster response and because in our application ease of storage and retrieval are more important than organizing data into more complex data structures, such as tables.

Key-value databases are the simplest of the NoSQL databases. The design of this type of data store is based on storing data with identifiers known as keys. We choosed LevelDB because combines the fast performance of a cache and the persistent storage of databases.

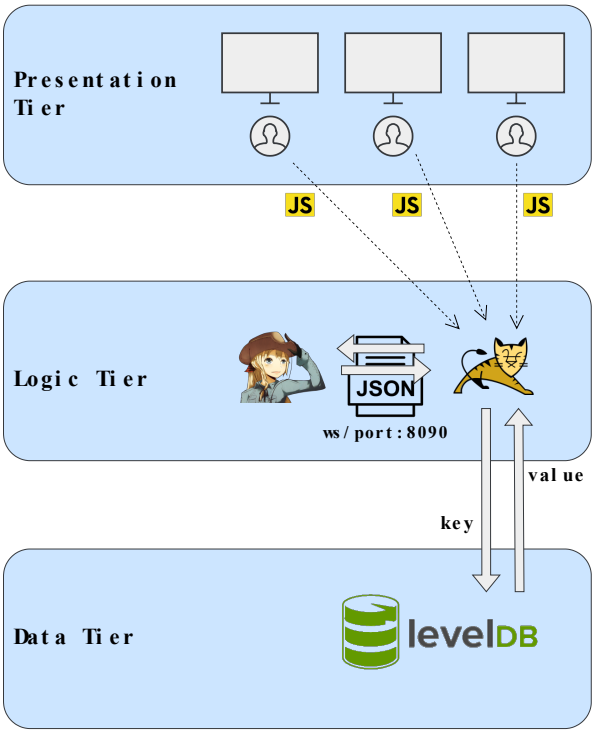
The key is made up of the word user, the value of the username and the name of the attribute, all separated by colons ':'.

An example of a record in the database is "user: lu : points = 100".



3. ARCHITECTURE

The application is designed as a client-server application: users through their web-browsers (the clients) can interact with the application that communicates with the web server.



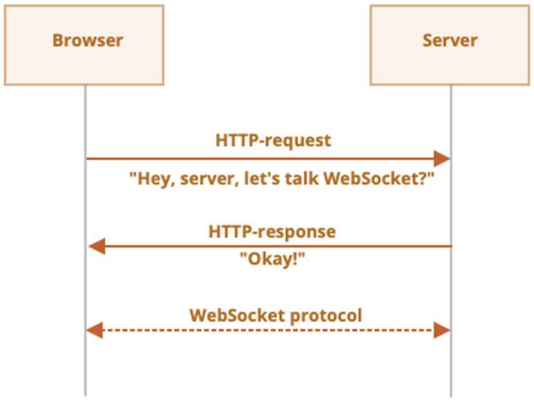
The application is structured into the typical web three-tier structure:

- The presentation level corresponds to the front-end level; the main task of this level is to display and collect information from the final user who, through this level, can interact with the application. It is built with HTML, cascading style sheets (CSS), and JavaScript, and it is deployed through a web client that is Tomcat.
- The logic tier has the task to process the information collected in the presentation tier (compared to other information in the data tier) using business logic. It consists of two parts, one written in Java and hosted on a Tomcat web server, and the other written in Erlang and hosted on a Cowboy server.
- The data tier is the point where the information processed by the application is stored and managed, it consists of a Key-Value Database and a Database Management System (LevelDB).

3.1 CLIENT SIDE

The web client is only in charge of the communication with the server and to provide an intuitive graphical interface where the user can visualize online users, make, and accept requests and so on.

For message exchanged we choose WebSocket; the reason why we make this decision is that WebSocket provide a way to exchange data between browsers and server via a persistent connection and they are suitable for services that require continuous data exchanges (such as in our case, being an online game). The data can be passed in both directions as "packets", without breaking the connection and other HTTP-requests.



Once created the socket, the client waits for an event and react based on the event type:

- open: established connection, an info message is printed on the console
- message: data received, based on the message type different actions are made.
- error: websocket error, the error message is printed on the console, the connection is closed, and the client is redirected on the main page (the error is shown to the user).
- close: connection close, an info message is printed on the console.

Otherwise, when is the client that wants to send a message, this can be done through the socket.send(data) method.

3.2 SERVER SIDE

The server handles the communication between clients and performs some operations:

- Keeps track of the online clients saving their usernames
- Notifies to all the online clients the status of the others
- Forwards the messages to the right client: if a client wants to contact another client, he needs to contact the server first and then it will forward the message to the right client using its username.

When a user logs in, an erlang process will be assigned to him until he logs out; in that case the process is closed. When two players start playing, indeed, their processes are destroyed, and two new processes are created.

In this project has been chosen Erlang as programming language of this part of the application because it is very suitable for achieving our goals. The application requires high performances, quickly handling each request. Erlang is generally used to build scalable soft real-time systems with requirements on high availability and supports concurrency, fault tolerance and distribution.

It has also been used Gen Server to keep track of the online clients in order to notify to the other clients when another one enters or exits the application.

Cowboy WebSocket has been used to communicate via HTTP requests. In fact, Cowboy is a fast HTTP server for Erlang/OTP that handles web socket requests. Cowboy provides routing capabilities and selectively dispatches requests to handlers in Erlang. Each client has a specific process thanks to with he will be able to communicate, exchanging messages with the process associated to another client. The exchange of messages is easier because the processes are registered with the user's username, which is unique. Inside the message there is also the information about sender e receiver.

4. HANDLING CONCURRENCY

4.1 SYNCHRONIZATION

This application makes use of the Servlet/Filters pairs. By default, servlets are not thread safe.

There is just one instance of a servlet, used in many different threads in charge of dealing with many different requests. This is dangerous because multiple threads could try to access simultaneously to the same resource. So, we have the issue regarding the concurrent access to LevelDB, the database used.

Using a synchronized method has been considered not a good idea, because it would have reduced considerably the performance of the application.

```
public class KeyValueDB {

    private static volatile KeyValueDB instance;
    private String path;
    public DB db;

    private KeyValueDB(){
        this("/db/database");

    }

    private KeyValueDB(String path) {
        this.path = path;
        openDB();
    }

    public static KeyValueDB getInstance() {

        if (instance == null) {
            synchronized (KeyValueDB.class) {
                if (instance == null)
                    instance = new KeyValueDB();
            }
        }
        return instance;
    }
```

In this case, we choose to make the KeyValueDb class a Singleton. So, when the getInstance() method is called for the first time, an instance of the class is created. This happens only the first time, the subsequent calls to the method will return the exact same instance.

In order to be sure that the construction of the instance has been completed before being available to other threads, the instance has been done volatile.

In this way, there is no need to take any other precaution: threads can call any method of the KeyValueDB class that queries the database and LevelDB implementation will take care of their synchronization.

Regarding the server written in Erlang, one of the points of strength of this language is the capacity to handle concurrency and distributed programming. The Erlang processes do not share data and there is only message communication, so there is not needed to handle synchronization.

4.2 COMMUNICATION

The communication between client and server is implemented using a Web Socket Server, which has been deployed using Cowboy.

The communication between them is based on asynchronous message exchanges. The client sends a request to the server and the server replies.

The messages exchanged between processes arrive asynchronously and at each message correspond some operations. The messages are in json format and a specific data structure has been defined for them. The different messages differ each other basing on the content of the *type* field. The *sender* and *receiver* fields specify, respectively, the sender and the receiver of the message.

```
class Message {
    constructor(type, data, sender, receiver) {
        this.type = type; // Type of the message, explain the content of the message (or type of error)
        this.data = data; // Data contained in the message, can be an object
        this.sender = sender; // username of sender
        this.receiver = receiver; // username of receiver
    }
}
```

Some types of messages are:

- *user_registration*: it causes the registration of a process within the erlang environment and adds the user in the online users list. In response, the user receives the list of the online users. In this way the user is now available to receive and send request messages. If the process is already registered means that the user is already online with another web client, so the access request is denied.
- *send_request*: sends to another user the game request.
- *accept_request*: used to accept a game request. When a user accepts it, the game starts.
- *cancel_request*: it is used to delete a game request, if it has not already accepted.
- *ongame_user*: notifies that the user has started a new game. The user is removed from the online users, and it is notified to all the online users that he is no more available to play. All the pendent game requests associated to this user are canceled.
- *surrender*: if one user decides to surrender, the opponent is informed.
- *game_move*: when a user makes a move, this message is sent to the opponent to inform him.

There is also the possibility that the WebSocket connection falls. For this reason, two callback functions have been defined to handle this problem, exploiting the terminate function provided by the framework.

When a user logs out or close the web browser, a '*delete_user*' message is sent to all the online users to notify it. In this way the other users cannot send requests to the unavailable user.

In case a user disconnects during the game, an '*opponent_disconnected*' message is delivered to the opponent, the game ends and he is redirected to the main page.

When a user sends a game request to another one and he accepts the request, a new game start. The Erlang system terminates the previous processes and starts new processes for these users, that will be no more available to receive requests. The server will handle the exchanging of messages between the clients during the game. In fact, the players send their moves during their turn and the server forwards them. The server notifies, also, if a player wins or surrenders and the game ends. At this point the points of the players are updated and they are redirected to the main page. In case a player is no longer available, due to any problem, the game is ended and the opponent wins.

4.3 COORDINATION

The coordination problems can occur in two different situations:

- A user challenges another one and the second one accepts the request.
- During the game

The coordination is implemented through message passing between the Erlang processes associated to each client; different messages, as described in the communication section, will lead to different operations.

For example, a user can accept a game request only after a game request has been received from another user. Indeed, during the game, after doing a move, the player must wait his next turn to play again.