UNIVERSITÀ DI PISA

COMPUTER ENGINEERING

Distributed Systems and Middleware Technologies

PROJECT DOCUMENTATION

Design and development of *"GameOn"*:

a Distributed Application using Java EE and Erlang

Students

Federica Baldi

Francesco Campilongo

Daniele Cioffo

Academic Year 2020/2021

1 APPLICATION REQUIREMENTS

1.1 FUNCTIONAL REQUIREMENTS AND USE CASES

GameOn is a web application that hosts a variety of multiplayer browser games. At the moment, featured games are *Connect Four* and *Tic-Tac-Toe*, but the list may be updated in the future.

In this application there is only one actor, the user. All features included in the application are intended to provide services for the user.

To access the application, users are required to register and login using a *username* and *password*. Once logged in, users can select the game they want to play from the list of featured ones.

After selecting a game, users enter the game's lobby, from which they can view a list of online users and a rank based on the number of wins.

Users can either choose their opponent from the list of online users or wait to be challenged. Users can also decide to change game, by returning to the list of games and selecting another one.

When users receive a game request, they can decide whether to accept or reject it. If a user accepts the request, the game begins on both sides.

During the game, users are given a limited amount of time to make their move, after which the turn will automatically pass to their opponent.

After a set number of lost rounds, users are considered offline and the game is won by their opponent. Moreover, if one of the two users disconnects, the game started is automatically won by the other user.

While playing, users can also chat with each other and send text messages commenting on their moves.

At the end of a match, users are automatically redirected to the lobby of the game, so they can start a new one.

1.2 NON-FUNCTIONAL REQUIREMENTS

In this subsection the non-functional requirements of the web application are considered, some of these had been defined by the professor.

1.2.1 Flexibility

As we want to leave the possibility to add new games over time, it is important to use a flexible model for our data.

1.2.2 Availability

For an application that hosts several users playing at the same time it is necessary to have a distributed structure that guarantees high availability.

1.2.3 Speed and Usability

The application must have very short response times and must be simple to use, for users to enjoy it.

1.2.4 Using Erlang

This requirement is part of those defined by the professor, it is necessary to develop at least one application module in Erlang.

1.2.5 Management of synchronizations, communications, coordination

This requirement is also part of those defined by the professor, the purpose of this project is to learn how to manage the synchronization, communication and coordination of processes in a distributed environment. A separate section will be dedicated for discussing these aspects.

2 STRUCTURE

GameOn is programmed using a *client-server* modeled structure: users are provided the use of the application through their web browsers (*clients*), which are responsible for communicating with the web server.

The application is organized into the classical three-tier architecture, as shown in Figure 1.

Presentation tier. The presentation tier is the front-end layer and consists of the user interface. It is built with HTML5, cascading style sheets (CSS), and JavaScript, and it is deployed through a web browser.

Logic tier. The logic tier contains all the business logic that supports the application's core functions. It consists of two parts, one written in Java and hosted on a Tomcat[1] web server, and the other written in Erlang and hosted on a Cowboy[2] server.

Data tier. The data tier consists of a Key-Value Database and a Database Management System. In particular, LevelDB is used. This gives us high data flexibility, thanks to the ability to easily add/remove new entity attributes, and excellent speed performance.

[1] http://tomcat.apache.org/
[2] https://ninenines.eu/

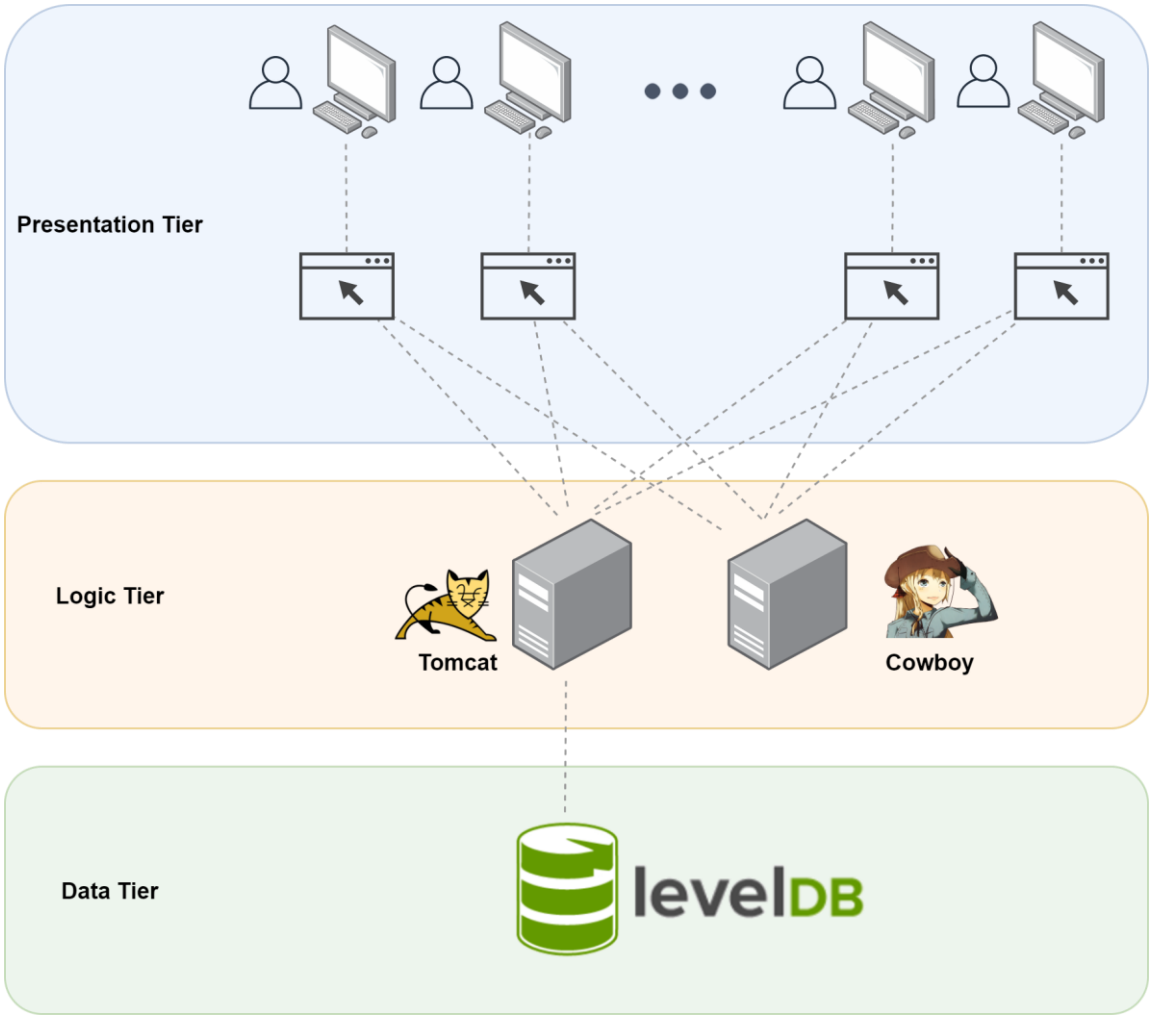

Figure 1. Application three-tier architecture

2.1 JAVA ENTERPRISE MODULE

As already mentioned, the web application is deployed on a Tomcat server. Figure 2 shows the structure of the directory of the web application.

The software design pattern used is the common MVC (*Model-View-Controller*). Indeed, each request from the client browser is passed at first to the Controller. In our case, the Controller is a Filter/Servlet pair.

The Filter is in charge of carrying out checks, for example on user authentication, and to pass the request to the Servlet only if the criteria are met.

The Servlet performs any logic necessary to obtain the correct content for display. For instance, it can communicate with the database and insert the information retrieved inside a User JavaBean (Model).

It then places the content in the request and decides which View it will pass the request to.

The View then renders the content passed by the Controller. In our application, Views are JSPs (JavaServer Pages).

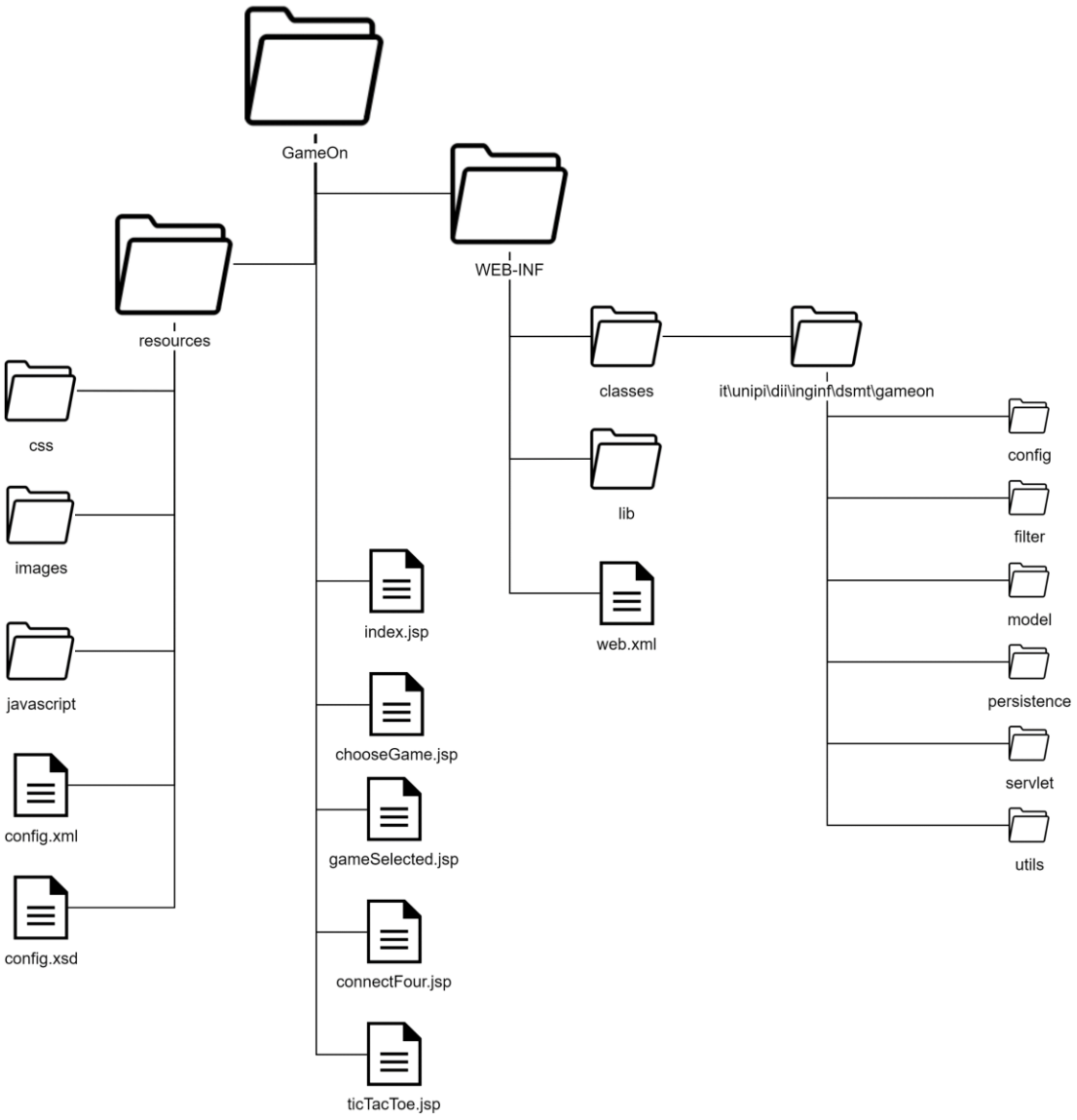

Figure 2. Structure of GameOn WebApp

Each JSP contains static HTML content along with *scriplets*, scripts written in Java. Moreover, a style sheet was created for each page to make the content more appealing.

Servlets handle the registration process, the login and the logout, the game selection, the saving of game results and the consequent ranking update.

Game dynamics and the interaction between two users are instead managed through JavaScript. On the one hand, by dynamically changing the HTML document in response to

certain actions and, on the other hand, by communicating via WebSocket with the Cowboy server (which we will discuss in detail in the next section).

Some aspects of the web application are configurable. In fact, the application reads some parameters from the `config.xml` file, so it is sufficient to modify the file to obtain the desired behavior.

2.2 ERLANG MODULE

In this section we consider the use of Erlang, which was a requirement of the system given to us by the professor; in particular, we discuss the use of Cowboy.

Cowboy is a small, fast, and modern HTTP server for Erlang/OTP. Cowboy provides a full HTTP stack, and it is optimized for *low latency* and *low memory usage*, in part because it uses binary strings. This allows for low response times. Cowboy provides routing capabilities and selectively dispatching requests to handlers written in Erlang.

The Web is concurrent, and Erlang is a language designed for concurrency, so it is a perfect match. Moreover, Erlang allows you to use the same code for communicating with local processes or with processes in other parts of your cluster, which means you can scale very quickly if the need arises. This is very important for our availability requirement.

In this application, Cowboy 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.

Cowboy uses different processes for handling the connection and the requests, indeed there will be one process that handles the requests for each client.

There will be a specific process for each connected client, which will be able to communicate, through the exchange of messages, with the processes associated to the other clients. Therefore, we decided to register the processes with the user's username, being unique, and this has led us to simplify the exchange of messages, because in each message there are also information on the sender and on the receiver (their usernames).

In addition to these processes, there are others, one for each waiting list (keeps the usernames of online users waiting for that game) and one that has the task of switching the messages at the right list.

3 HANDLING CONCURRENCY

3.1 SYNCHRONIZATION

As we mentioned earlier, the design pattern we followed in the Java Enterprise module is the MVC, where the *Controller* part is done by *Filter/Servlet* pairs.

By default, *Servlets* are not thread-safe, so we have had be careful that our *Servlets* classes were indeed safe for operation in a multi-threaded environment.

The methods in a single *Servlet* instance are usually executed numerous times simultaneously. Each execution occurs in a different thread, though only one *Servlet* copy exists in the servlet engine. This efficient system resource usage is dangerous because multiple threads could try to access simultaneously to the same resource.

In particular, in our application *Servlets* access two different resources: the configuration file and the database file.

For efficiency reasons, we wanted the configuration file to be read only once when opening the application and not every time it was needed. Therefore, we decided to create the *ConfigurationParameters* class and make it a *Singleton*.

In this way, a single instance of the *ConfigurationParameters* class is shared by the various Servlets that will be able to retrieve the configuration parameters by invoking getter methods on it.

Following the *Singleton* design pattern, an instance of the class is created (and as a result the configuration file is read) only the first time it is requested; from that point on, all calls to the *getInstance()* function will return the exact same instance.

Obviously, being in a multithreaded environment, we had to take some measures.

We could have decided to make the *getInstance()* method synchronized; that way, we would have been sure that only one thread at a time could execute the function. However, this approach would have slowed performance down a lot, so we decided to follow the approach shown in Figure 3.

```
1.  public class ConfigurationParameters {
2.      public static volatile ConfigurationParameters instance;
3.      private String pathDatabase;
4.      private int howManySecondsForEachTurn;
5.      private int howManySkippedRoundsToStopTheGame;
6.      private int howManyUsersToSeeInTheRanking;
7.
8.      public static ConfigurationParameters getInstance() {
9.          if(instance == null) {
10.             synchronized (ConfigurationParameters.class) {
11.                 if(instance==null) {
12.                     instance = readConfigurationParameters();
13.                 }
14.             }
15.         }
16.         return instance;
17.     }
```

Figure 3. ConfigurationParameters class

Indeed, once an object is created, synchronization is no longer useful. So, we will only acquire lock on the *getInstance()* once, when the object is null. From there on, there are no more synchronization issues.

Regarding the database, as we have already mentioned, we decided to use a Key-Value Database and, in particular, LevelDB.

Reading the LevelDB documentation we saw that the database file can only be opened by one process at a time. Indeed, the LevelDB implementation acquires a lock from the operating system to prevent misuse.

That said, we decided to follow the same approach just described for the *ConfigurationParameters* class. In this case, the *KeyValueDBDriver* class is a *Singleton* and, when *getInstance()* method is called for the first time, at the same time as the single instance is created, the database is opened.

Moreover, within the documentation we read that:

> "Within a single process, the same leveldb::DB object may be safely shared by multiple concurrent threads. I.e., different threads may write into or fetch iterators or call Get on the same database without any external synchronization (the leveldb implementation will automatically do the required synchronization)."

So, there was no need to take any other precaution: threads can call any method of the *KeyValueDBDriver* class that query the database and it is LevelDB implementation that will take care of their synchronization.

What has been described so far concerned the Java Enterprise module.

As for the Erlang module, we know that one of the strengths of this language is its ability to handle concurrency and distributed programming.

Indeed, in an Erlang program it is easy to create parallel processes and to allow them to communicate with each other. What is more, since these processes do not share data and their only means of communication are messages, there is no need to handle synchronization issues.

3.2 COMMUNICATION

The communication between client and server in this project is implemented using a Web Socket Server, which has been developed using Cowboy.

WebSocket is an extension to HTTP that emulates plain TCP connections between the client, typically a Web browser, and the server. It uses the HTTP Upgrade mechanism to establish the connection.

WebSocket connections are fully asynchronous, unlike HTTP/1.1 (synchronous) and HTTP/2 (asynchronous, but the server can only initiate streams in response to requests). With

WebSocket, the client and the server can both send frames at any time without any restriction. It is closer to TCP than any of the HTTP protocols.

The Erlang distributed structure of this project is characterized by multiple processes: one for each client (the browsers), and others for the system. The processes communicate with each other following an asynchronous message exchange.

The client sends a request to the server and the server replies back; this exchange of messages is carried out through a WebSocket.

In this project the communication between the client (browser) and the Erlang Web Socket server is the engine that allows the application to work.

From the login to the chat message into a game, almost everything is possible by means of an exchange of messages between a client and a server.

In the following figure there is an example of how the application works during a classical game match.

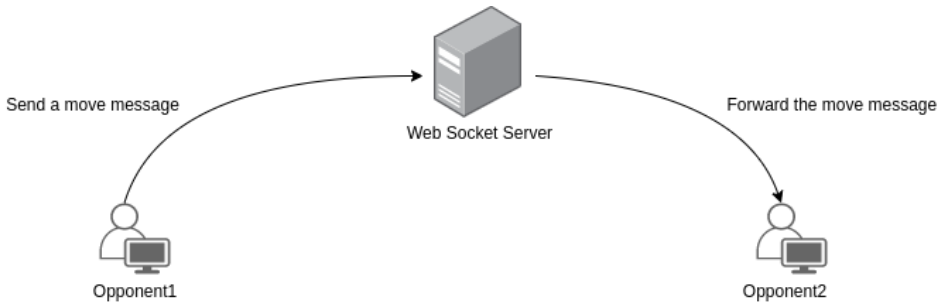

Figure 4. Example of communication between clients through the erlang web socket server

As described in Figure 4, the server forwards the move message (earlier received from the Opponent1) to the Opponent2. This message will be caught by the Opponent2 JavaScript code and the effects of the move will be displayed on the Opponent2 browser.

In this project, a Tomcat Web server was also implemented.

The communication between the browser and Tomcat server through Servlets and JSP files allows the user to navigate between the application pages, exploiting overridden methods of the specific Servlet.

In particular in this project, methods *doPost* (called by the server – via the service method – to allow a Servlet to handle a POST request) and *doGet* (called by the server – via the service method – to allow a Servlet to handle a GET request) will load the correct JSP, in order to correctly visualize the page needed for the specific operation requested.

3.3 COORDINATION

The coordination problem was encountered mainly in two situations:

- To send a game request and respond to it by accepting it.
- During the game.

This coordination takes place asynchronously, through the exchange of messages between the Erlang processes associated with the various users. Recall that these Erlang processes do not need coordination mechanisms.

More precisely, a Message data structure has been defined which contains, in addition to the message content, also the type of the message, the sender and the receiver.

The coordination is carried out by sending different types of Messages, for example, when I send my move during the game, I will not be able to send other moves until it is my turn again (this can happen either because I receive a move from the opponent or because the timer expires).

Based on the type of message received I will go to perform different operations, for example if I receive a chat message, I will enter it in the chat box.