UNIVERSITÀ DI PISA

Computer engineering

Distributed Systems and Middleware Technologies

Fitconnect

Tommaso Bertini

Giovanni Marrucci

Veronica Torraca

A.Y. 2023/2024

Contents

1 Project Specifications

1.1 Introduction

FitConnect is a website for managing a fitness facility. It allows users to register, enroll in different fitness courses, and book classes with trainers. It also includes features for user profile management, chat functionality for course participants, and a notification system to enhance the user experience.

1.2 Functional requirements

In this application there are three type of actors:

- Unregistered users
- Registered users
- Personal trainers

Unregistered users can:

- Create an account

Registered users can:

- Login/Logout
- Search and browse the training courses
- Select a training course
- Join a training course
- After selecting a training course a user can:
  - Book a slot for a course's class
  - Delete a booking previously made
- Unfollow a training course

Personal trainers can:

- Login/Logout
- Create a training course
- Generate classes of the course

1.3 Nonfunctional requirements

The web application guarantees the following properties:

- Concurrent service accesses management
- Strong consistency for users, courses and classes data stored in MongoDB and MnesiaDB

Chatroom and notifications functionalities are guaranteed thanks to the following properties:

- Concurrent service accesses management
- High service availability
- Fault tolerance to node faults
- Allow high horizontal scalability

2 System architecture

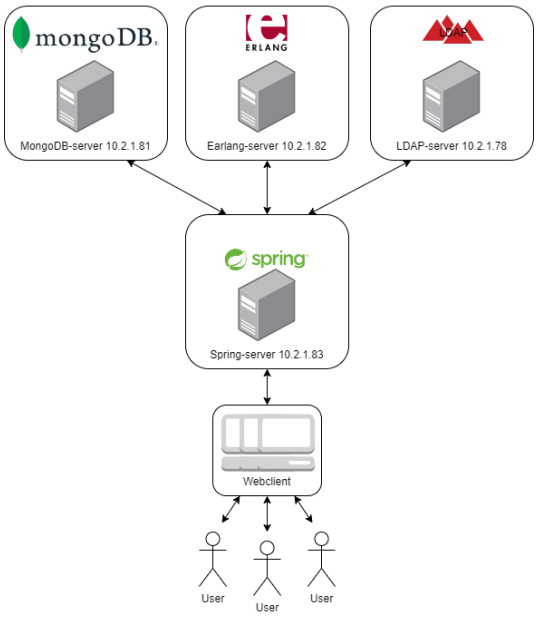

Figure 1: System architecture

2.1 Introduction

The FitConnect application has been developed in Java as a **Spring Boot** web application and as such is composed of several interconnected layers:

- *Presentation layer*, representing the front-end of the application, with controllers for receiving and mapping HTTP requests
- *Application layer*, for the business logic, managing the connection with several services, such as the authentication service, the entity definitions, like User, Course, etc. and the communication service with the Erlang server for notifications (it could have been used for messages exchange as well)
- *Infrastructure/Persistence layer*, includes the LDAP server and the MongoDB database, managed via **Spring Data**, so is responsible for all CRUD operations, and the Erlang server

More specifically, the different servers are located on different nodes for different purposes, such as:

- the **LDAP** server is located on the remote node `10.2.1.78`, used to manage the registration and authentication phase
- the **MongoDB** replica set instances were created on node `10.2.1.81`, used to store data persistently
- the **Erlang** server has been deployed on node `10.2.1.82`, used to handle concurrency and synchronization for notifications and messages

All of these modules are described in more detail below.

2.2 Erlang

Using *Erlang*, we developed **chatroom** and **notification** functionalities. Each one has been assigned to a process and monitored by a supervisor to ensure high availability and fault tolerance. Below is shown how the processes communicate:

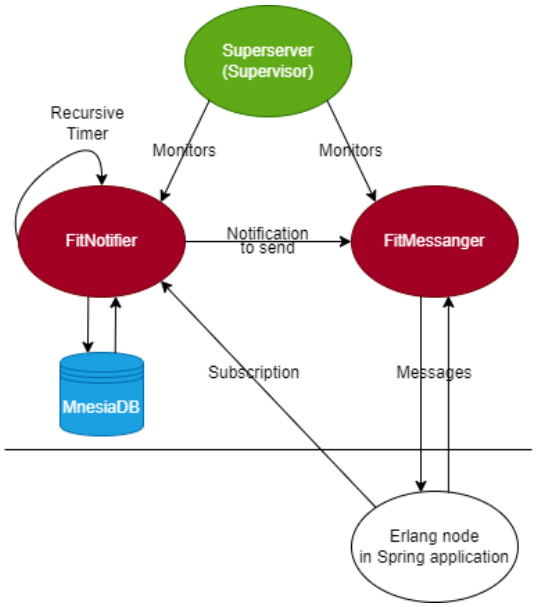

Figure 2: Erlang components schema

The processes are run on the **node** *server* using the **cookie** `dsmt`. The container used is the one at 10.2.1.82. Each element in the graph is implemented as a module. In total we have created 4 modules:

- `Superserver.erl`
- `FitNotifier.erl`
- `FitDb.erl` (Not present in Fig. 2 because used by the FitNotifier to interact with the MnesiaDB)
- `FitMessanger.erl`

2.2.1 Superserver

The *Superserver* is a simple *Erlang* module that implements the *supervisor* **behaviour**. In detail, the behaviour is as follows:

```
SupFlags = #{strategy => one_for_one,
        intensity => 3,
        period => 20}
```

We enforce fault tolerance and high availability by restarting only the process that has crashed and at the same time we prevent excessive restarts in a short period of time. As previously mentioned, we have two different possible processes, but they both retain the same properties. Here is an example:

```
FitMessanger = #{id => fitMessanger,
    start => {fitMessanger, start_link, []},
    restart => permanent,
    shutdown => infinity,
    type => worker,
    modules => [fitMessanger]}
```

In general, if the process crashes, it is restarted indefinitely, and it is never automatically shut down, and we have chosen worker processes to perform specific tasks.

2.2.2 FitNotifier

The *FitNotifier* is a simple *Erlang* module that implements the `gen_server` **behaviour**. It communicates with the underlying database through the functions exposed by the *FitDb* module. After the Notifier is started, it waits for calls from clients and recursively checks every 5 minutes if there are notifications to send. Notifications are added using "`handle_call(insert, Username, ScheduleId, Timestamp,_From, Timers)`", edited using "`handle_call(edit, _Username, ScheduleId, Timestamp, _From, Timers)`" and removed using "`handle_call(delete, Username, ScheduleId, _Timestamp, _From, Timers)`". To recursively check for expired notifications we use a timer that triggers "`notify()`". The timer is initially generated by "`init()`" and then after it expired is called by the "`notify()`" itself. Whenever a notification has expired a call is made to the *FitMessanger* as follows: "`gen_server:call(fitMessanger, Message)`" to indicate to which user send the notification to. We can't send notifications directly to users, since the Notifier doesn't have any node's pid identifier.

2.2.3 FitMessanger

The *FitMessanger* is a simple *Erlang* module that implements the `gen_server` **behaviour** and communicates with the Erlang nodes located in the Spring application. It maintains in its *state* all the *clients* connected to the platform and for each one remembers its *list of courses*. Moreover, it is in charge of forwarding both messages and notifications to users using its "`broadcast(Message, Pids)`".

2.3 Web Server

The web service is the front-end of the system, developed using **Spring Boot**, including **Tomcat** as the default WebServer to handle client-server interactions, **Thymeleaf** and **Bootstrap**. It exposes a browser-accessible GUI to allow users to interact with the application. It also uses several back-end services, in particular:

- An LDAP service, to manage access to the web application
- A MongoDB database service, to persistently store application data
- A Mnesia database service, deployed on an Erlang server, to store data needed for functionalities developed in Erlang

2.4 LDAP

The LDAP protocol was used to manage the user registration and authentication. The application interacts with the LDAP repositories via *Spring Data LDAP*. In particular, the users of type `trainer` are already registered, so in this case only the authentication phase is needed, instead the `clients` can register and then log into the application.

2.5 MongoDB

The MongoDB server has been adopted to store and manage persistent data for the web server. Interactions with the database are handled by *Spring Data MongoDB*. MongoDB has been chosen to manage a large amount of data in a simple and reliable way.

2.5.1 Collections

The database, called *fitconnect*, contains four collections:

- **Users**: contains all user data, for both types of user - trainer and client - each identified by a unique username, and also a reference to the *Courses* collection, which shows all the courses a client has joined or a trainer is teaching
- **Courses**: contains all available courses with all the information needed to join the course or book a lesson
- **Reservations**: contains a document for each lesson (aka ClassTime) of each course held, with the actual time of the lesson. A new document is created each time a new class time is added to a course. These documents are deleted and re-created with the updated actual class time, if the `actualClassTime` field is earlier than the current time, so that only the lessons that have not yet been held in a day have the associated booking documents. In addition, each of these documents will have a list of clients who have booked that particular lesson. This association is made using the `@DocumentReference` with the `username` of the clients involved as the lookup key.
- **Messages**: contains all the messages exchanged in the group chat of a Course, identified by the Course ID, the sender and the time of sending

All *cross-references* between different collections are managed in Java code, using the `@DocumentReference` SpringBoot annotation, which allow us to create both One-to-Many and Many-to-Many relationships. Some of these references are also set with the `@ReadOnlyProperty` annotation, to indicate that the actual object is not stored in the database, but it can be accessed through the referenced field when we read it. In this way, we allow Spring Data to manage and update the relationships correctly.

2.5.2 Operations

In addition to the basic CRUD operations, the following operations can be performed on the data provided to and from the front-end:

- `browse` the list of all available courses, sorted by type
- `browse` the list of all the courses that a client has joined
- `browse` the list of all the reservations of a client
- `create/delete` a course (only the trainer)
- `add/edit/delete` a classTime (only the owner of the course)
- `join/leave` a course (only clients)
- `book/unbook` a lesson (only clients who have joined that specific course)

2.5.3 Concurrency

The system needs to manage concurrency for many reasons. In particular, a lesson can only be booked by a limited number of clients, so overbooking is not allowed, or if the trainer tries to edit or delete a lesson or course, there could be a concurrent access to the same documents. In all these cases, the data needs to be managed in a consistent way. To face and solve this problem, we used the **optimistic concurrency control** and we added a `version` field in the involved classes.
E.g. in *Reservations.java* class there is

```
@Version
private Long version;
```

From Spring Data Reference: *the `@Version` annotation provides syntax similar to that of JPA in the context of MongoDB and makes sure updates are only applied to documents with a matching version. Therefore, the actual value of the version property is added to the update query in such a way that the update does not have any effect if another operation altered the document in the meantime. In that case, an `OptimisticLockingFailureException` is thrown.*
This field is auto initialized with 0 and incremented at each document update.

3 Chat and Notification Systems

We also integrated a simple messaging system and a notification system using JavaScript for the front-end and Java for the middleware and also Erlang to trigger the sending of notifications (only as reminder for booked classes, not for chat messages).

3.1 STOMP protocol

We chose STOMP as the messaging protocol due to its simplicity and widespread adoption in this area. STOMP supports a channel-based messaging model where clients can publish (send messages) and subscribe (receive messages) to specific channels using WebSockets.

In our case, once users successfully log into the site, they are automatically connected to channels associated with the courses they are enrolled in, identified by the course ID.

All users are also subscribed to a ”private” channel to receive notifications about their own active class bookings.

3.2 Chat system

When the chat page loads, the user is presented with the most recent messages. To view older messages, they can simply press the ’Show More’ button, which trigger an AJAX call to retrieve additional messages from the backend.

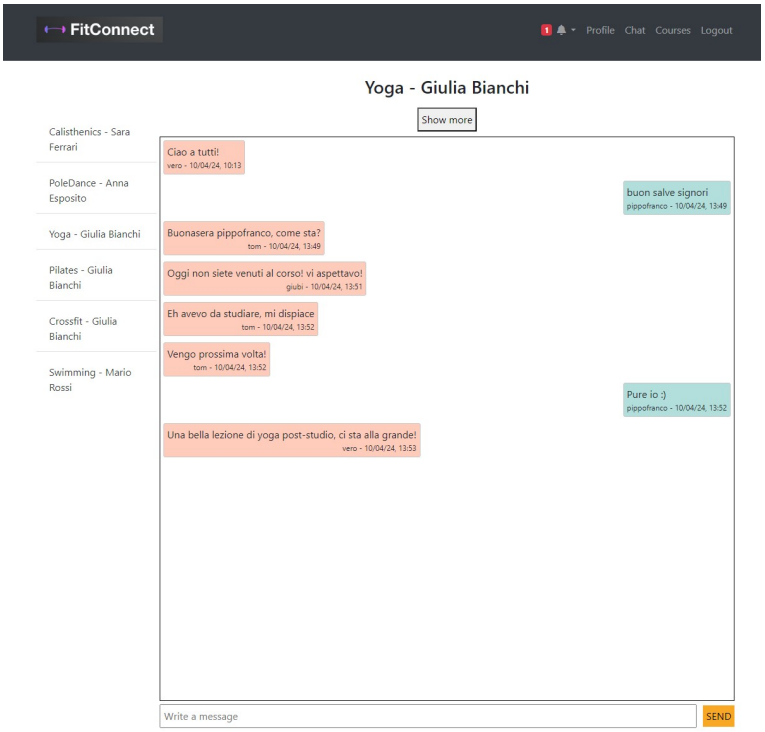

Figure 3: Chat example

3.2.1 Sending messages

When a user sends a message, it is broadcast to all connected users in the channel. Using the Java controller, the message is received and store correctly in the MongoDB database.
Users who are currently offline will not receive the message immediately, but will see it when they access the chat, as messages are retrieved from the MongoDB database via a query when the chat is opened.

3.3 Notification system

As already mentioned, the notification system is implemented using Erlang and the STOMP protocol over Web Socket. The sending of notifications is triggered by the Erlang module `FitNotifier`, which periodically checks for notifications that will expire in half an hour, or notifications that have already expired but have not been sent because the target client is not connected.
Additionally, whenever a trainer changes or deletes a class schedule, a notification is sent to all clients booked into that class.
Notifications are also sent to all members of a course, both trainer and clients, when a new client joins or leaves the course.

On the frontend side, notifications are stored in the browser's local storage to persist them during web page navigation until the user deletes them.

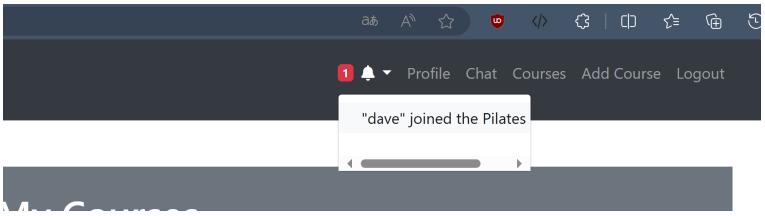

Figure 4: Notification example

4 Appendix: Further Functionality No Longer Developed

In the early stages of the project, we implemented functions in the *Erlang Server* to facilitate communication among users or to broadcast messages to entire courses. At that time, the entire *Spring application* had not yet been deployed, so testing was conducted via the command line. However, we opted not to further develop this functionality because messages would have had to pass from Spring to the Erlang Server and then back to Spring. This intermediary step seemed unnecessary since we could have used websockets for communication, bypassing the Erlang Server entirely.

Nonetheless, within the "**iterate_over_connected_clients(Mode, Data, Clients)**" function, there remains a section dedicated to constructing messages:

```
case Mode of
    .
    .
    .
    sendToCourse ->
            {Course, Username, Text} = Data,
            Msg = {message, Course, Username, Text};
    .
    .
    .
    sendToCourse ->
            {Acc, OldUser, NewUser}; % Do nothing, just return the list
    .
    .
    .
    broadcast(Msg, Receivers),
    if
        Mode == sendToCourse ->
            ok; % No need to update the server state
```

Although the functionality is no longer developed, it's still valuable to understand the structure of this code segment.