



UNIVERSITÀ DI PISA

MSc in Computer Engineering

Distributed Systems and Middleware Technologies

myTicket

Author

Lorenzo Bataloni

Github Repo: https://github.com/bata26/myTicket_DSMT

Accademic Year 2023/2024

1 Project Specification.....	3
1.1. Functional requirements.....	3
1.2. Non functional requirements.....	3
1.3. Synchronization and communication issues.....	3
2 System Architecture.....	4
2.1. Erlang.....	5
2.1.1. Master Node.....	5
2.1.2. Worker Node.....	6
2.1.2.1. Worker Node File Structure.....	6
2.1.2.2. WebSocket Communication.....	6
2.1.3. Mnesia Database.....	8
2.2. Java Spring Server.....	9
2.3. WebSocket Client.....	9
2.4. Rqlite Database.....	10
3 User Manual.....	11
3.1. Login Page.....	11
3.2. Register Page.....	11
3.3. Home Page.....	12
3.4. Tickets Page.....	12
3.5. Ticket Detail Page.....	13
3.6. Profile Page.....	13
3.7. Add Page.....	13
3.8. Old Auction Page.....	14
3.9. Auction Detail.....	14

1 Project Specification

myTicket is a distributed web-app which provides users possibility to sell tickets for events, concerts and party creating auction. Each user can upload ticket and create an auction for his own ticket. The other users can bid to buy the specific ticket.

1.1. Functional requirements

- A user can register to the platform.
- A registered user can join multiple auctions.
- A registered user can create or delete an auction w.r.t. his own tickets.
- A registered user cannot create a bid on his auction.
- A registered user can create a bid for an auction.
- A registered user can add a ticket to his own list of tickets.
- Every auction is accessible to all registered users.
- Every auction has a winner.
- A registered user can see every auction.
- A registered user needs to register to participate to an auction.
- During an auction, every user registered to it, can see the older bids.
- A user can login with username and password.

1.2. Non functional requirements

- Communication between client and server uses HTTP protocol.
- When a user joins an auction, communication between client and server uses WebSocket.
- System needs to be available even if one node fails.
- System needs to be consistent even if one node fails.
- There will be a load balancer to manage the request load.
- Each node is going to have the same data.

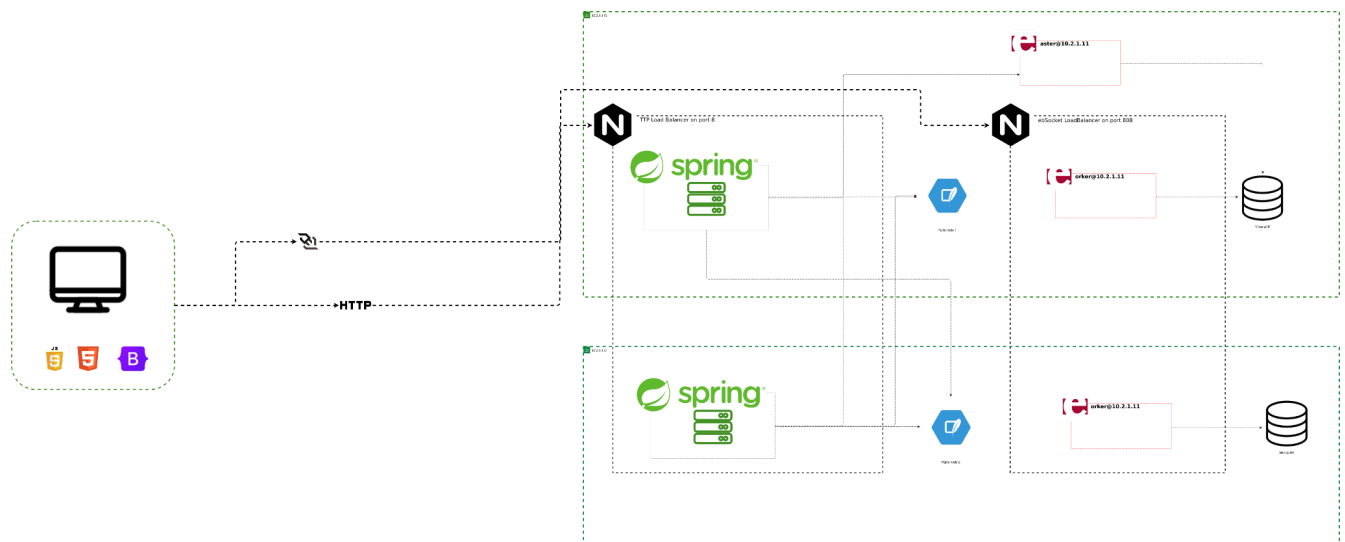
1.3. Synchronization and communication issues

The system is a fully distributed system with strong data replication to ensure the best availability and fault tolerance.

The main problems to be addressed are:

- Multiple clients can join an auction, each user's bidding list must be up-to-date and consistent with reality.
- When a user joins an auction he must receive all the betting history up to that point and be able to participate in the auction as if he had participated from the beginning.
- When a user creates an auction, it must be available to all registered users.
- When an auction ends, all users must be notified at the same time to avoid inconsistencies.

2 System Architecture



The web application, as we can see from the figure, has an elaborate structure. We have a Java server that has both the role of web server and REST API server. Then there is a master node in erlang that handles worker nodes also in erlang.

Both the communication with the server in Java and the worker nodes is handled by a load balancer in nginx that takes care of managing the load on the machines where the services are deployed.

Finally, two different data management systems are used for data management.

In fact, the server in Java uses a distributed Rqlite database, this allows for a sqlite relational database on multiple nodes.

The services in erlang, on the other hand, use Mnesia, which is shared between the master node and worker node instances. As we can see, we therefore have a virtually fully distributed and replicated system that goes to make up for any failure of one of the components of the entire architecture.

The only service that could not be distributed is the master node in erlang because its supervisor role did not allow distribution with the current architecture.

This architectural choice has allowed for an equitable distribution of the load thanks to the use of load balancers. With only two nodes available, the decision to replicate the systems proved to be the optimal situation. This is because the responsibilities of the two nodes are comprehensive and encompass all functionalities - except for the master node - and therefore a division of responsibilities would have rendered the application unavailable as specified by the requirements.

Indeed, if the Rqlite database and the Java server had not been distributed, the failure of one of the nodes would have rendered the application unreachable. In fact, the approach of this architecture is precisely to be a stateless architecture - characteristic of REST architecture - and distributing the databases allows for consistency and availability under any circumstances.

However, it is important to note some critical aspects of the architecture and potential improvements. A problem could be related to Nginx running on the same node as the services; in an enterprise environment, Load Balancers are typically separated hardware

devices from the servers they handle the load for, allowing them to be independent from the machines on which the servers are actually running.

Another critical point to note is that there are no recovery mechanisms in the event of either programme panics or server reboots.

For server reboots, this could be solved by creating Unix services for the execution of all application components at server start-up.

For application panics, it would be useful to use services to check the actual execution, but this is not an effective solution.

A truly effective solution would be to use an orchestrator such as Kubernetes to manage all the components, however this solution is outside the scope of this project and will not be explored in depth.

2.1. Erlang

Within the project, the main functionalities were developed in Erlang. In particular, all functions related to creating an auction, joining an auction, betting, and winning an auction are handled completely by services in Erlang. We have two main actors, the master node and the worker node.

The master node is responsible for doing an initial setup of the cluster and the database and for going on to create worker nodes on both nodes. In addition, the master node handles communication via HTTP with the server in Java.

The worker nodes, on the other hand, handle the joining of a user to a specific auction, the betting of users, and the closing of the auction. The worker nodes communicate directly via WebSocket with the client.

The handling of all traffic via WebSocket is done through Nginx, which acts as a load balancer for the two nodes.

In order to implement the functionality related to communication via HTTP and WebSocket, the Cowboy library was leveraged, a very popular and lightweight library that allows not having to deal with low-level HTTP.

2.1.1. Master Node

As we have seen, the master node has the role of performing the setup of the cluster, the database and finally starting the execution of the worker nodes.

The master node also has the role of communicating with the Java server to find out whether an auction has been created and to inform the server of the termination of one. In addition, the master node exposes an endpoint to get information regarding the betting history of a specific auction.

Let us look in detail at endpoint management for the master node. The master node supports only HTTP traffic and communicates only with the server in Java.

Specifically, the endpoints it exposes are:

- **POST /auction:** this endpoint is contacted by the server when a user creates a new auction for a ticket.
- **GET /auction/history:** this endpoint is contacted to get the list of all the bids made for a specific auction.

2.1.2. Worker Node

The worker node is the module that implements the main functionality of the application, handling all the main operations such as joining an auction, betting and closing the auction itself.

Worker nodes expose the /ws endpoint on port 8081 to receive messages via WebSocket. Once a message arrives, via the opcode field of the received message, the handler will implement a different handling.

2.1.2.1. Worker Node File Structure

- **worker_node_app:** It's the first module that is executed, implements the application behavior and executes the supervisor.
- **worker_node_sup:** The supervisor callback module. It is responsible for spawning a process that will manage the lifecycle of the process running the Cowboy server. It implements the behavior of the supervisor.
- **cowboy_listener:** This is a gen_server module that starts an instance of Cowboy listening on port 8081 to handle all incoming messages via WebSocket.
- **ws_handler:** This is the module that implements all the message handling logic via WebSocket, specifically it receives the message, parses it to get the information, and finally assigns the actual handling to the bid_server module, which once it has finished handling informs the ws_handler module to send a response message.
- **bid_server:** This is the module that implements the actual logic of the application; it could be seen as the controller of the MVC pattern. Indeed, it implements all the logic for controlling, saving and managing the auctions, in case some operations also affect the database, it will contact the mnesia_manager module to perform them.
- **mnesia_manager:** This is the module that handles all operations involving the Mnesia database, is contacted exclusively by the bid_server, and executes the queries requested by returning the result.

2.1.2.2. WebSocket Communication

Let us now analyze the messages that are exchanged between the worker node and the various Javascript clients.

The operations that the client can request from the worker node are JOIN and BID. The JOIN allows a user to join the list of auction participants and see all previous bids at the time of the join. The BID, on the other hand, allows a user who has already joined the auction to place a bid.

All messages are exchanged in JSON format, since Erlang does not natively support this type of data, the jiffy library was used for decoding and encoding incoming and outgoing JSON.

Regarding the JOIN operation, the payload sent by the client has the following structure:

```
{
  "opcode": "JOIN",
  "auctionID": parseInt(localStorage.getItem("auctionID")),
  "userID": parseInt(localStorage.getItem("userID")),
  "username": localStorage.getItem("username")
}
```

In detail we have:

- **opcode**: Allows the worker node to understand what action is requested by the client.
- **auctionID**: Uniquely indicates the auction the user wants to join to.
- **userID**: Uniquely indicates the user who wants to join.
- **username**: Indicates the username of the user.

Regarding the BID operation, the payload sent by the client has the following structure:

```
{
  "opcode": "BID",
  "auctionID": parseInt(localStorage.getItem("auctionID")),
  "userID": parseInt(localStorage.getItem("userID")),
  "username": localStorage.getItem("username"),
  "amount": amount,
  "ts": formattedDate,
}
```

The structure is very similar to the JOIN one, so we are going to analyze only the additional fields:

- **amount**: indicates the amount of the user's bid.
- **ts**: indicates the timestamp of the bid.

Let us now see what are the responses sent by the worker nodes to the Javascript client:

- **JOIN**: the join returns a list of bids in the form:

```
[
  {
    "auction_id": 1,
    "user_id": 1,
    "username": "user1",
    "amount": 10,
    "ts": "2022-09-30T12:00:00Z"
  },
  ...
]
```

- **BID:** The BID operation returns “OK” if the bid is added in the Mnesia database.
- **RECV BID:** When a user adds a bid, all other users connected to the auction receive a packet containing information about the bid they have just placed. The structure of the packet is the same as one of the list items received after the JOIN, plus a field which specifies the opcode.
- **END AUCTION:** When an auction ends, each client connected to the auction receives a package with the structure:

```
{
  "opcode": "END BID",
  "winnerID": 1
}
```

2.1.3. Mnesia Database

Mnesia is a distributed DBMS that has been used to maintain information related to auctions and betting. This information is distributed for all nodes.

Regarding the distribution of information, persistence is managed so that only the master node keeps the database information on disk, while all other nodes keep everything in RAM. This distinction is made through the *ram_copies* and *disc_copies* directives during schema creation by the master node.

The structure of the records defined within the database is the following:

```
-record(auction, {auction_id, owner_id, timer}).
-record(bid, {auction_id, user_id, username, amount, ts}).
-record(users, {auction_id, user_pid, user_id}).
```

The following information is maintained in the **auction** record:

- auctionID
- ownerID
- timer

The **timer** field deserves special attention. Indeed, this contains a reference to a timer that is updated for each bid and that allows the timeout to be managed following the winning bid in the auction.

The **bid** record, on the other hand, contains the fields:

- auctionID
- userID
- username
- amount
- ts

These fields will not be discussed in detail as they are fairly self-explanatory.

Finally in the **users** record we maintain:

- auctionID
- userID
- userPID

In this record we maintain the information of the users who join for a specific auction.

The operations that have been implemented on the records are the main **CRUD** operations:

- when creating an auction, a new entry will be added to the **auction** record.
- at the time of joining an auction the user will be added in the **users** record and a readout of all past **bids** will be performed.
- at the time of the bid the bid will be added into the **bid** record. In addition, each time a bid is placed on a specific auction, the reference to the timer in the specific **auction** record entry will be obtained and will be updated so that it will be updated with a more recent timer and thus manage the timeout for auction termination.
- On the other hand, when the bid history for a specific auction is requested, all the entries in the **bid** record for that specific auction will be read.

2.2. Java Spring Server

As already anticipated, a server was developed for both the management of static files, particularly the HTML pages that structure the application GUI, and the creation of APIs reachable from both the application frontend and the master node.

To create this server, the Spring Boot library was used, a library that implements a web server and has transparent management for what concerns static files. This allowed me to focus mainly on the API logic and actual functionality without going to spend too much time regarding GUI management.

The server was implemented using the **MVC** (Model View Controller) pattern, one of the most popular patterns for this type of application.

We have three main entities which are **User**, **Auction** and **Ticket**. Views, models and controllers have been implemented for each of these identities.

A class was then implemented via the Singleton Pattern to manage and maintain the connection to the database.

A static class was then added to handle outgoing requests for the master node when auctions are created or the history of one is requested.

Finally, special classes called DTO were defined to manage communication with the frontend.

2.3. WebSocket Client

As already mentioned, communication between the client and the worker nodes takes place via WebSocket. In particular, this takes place on the frontend page dedicated to the details of an auction where the user has the option of joining and placing bids for a specific auction.

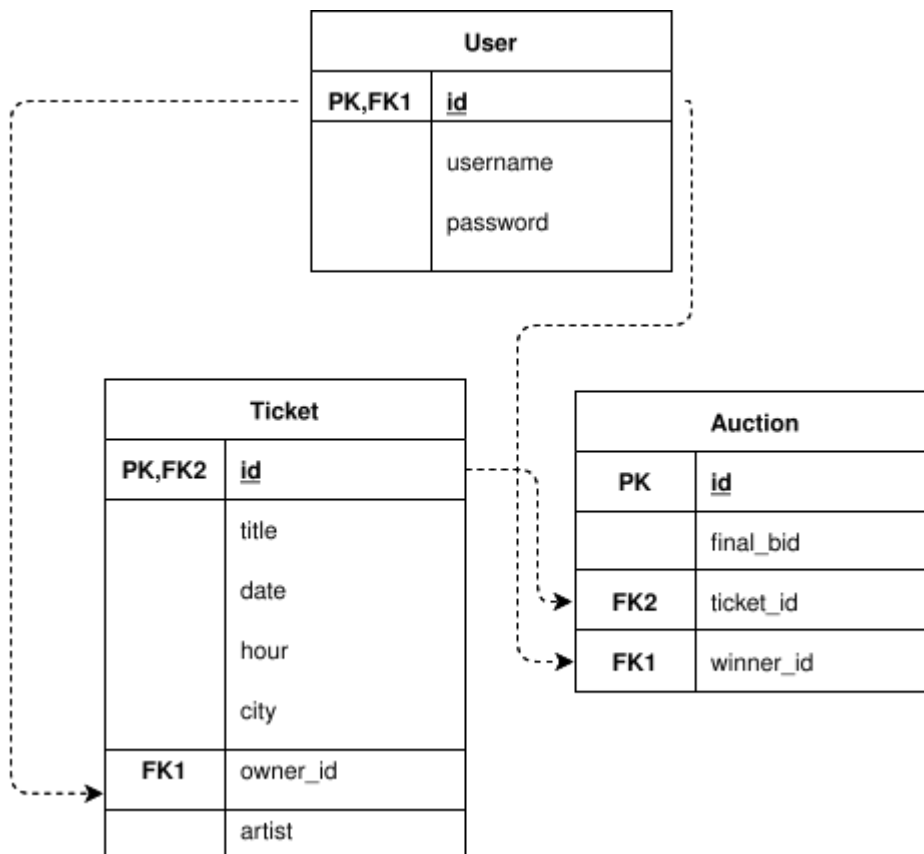
The connection management uses the WebSocket object offered by Javascript and implements the functionality of sending and receiving messages.

When messages are received, they are parsed in order to extract information about the type of message and thus the operations to be performed in order to have an optimal management of these.

2.4. Rqlite Database

In addition to **Mnesia**, the Java server uses a **Rqlite** database to maintain information on users, tickets and auctions.

Rqlite is a wrapper for sqlite that implements the functionality to have a distributed database cluster. This allowed me to distribute the database in both nodes. In addition, a very interesting feature is that it is completely transparent to the programmer, as we do not need to manage the cluster at the code level as any faults or crashes are handled directly by the DBMS. Furthermore, it was very useful to deal with Sqlite anyway, as it meant that we did not have to rewrite queries or implement new management mechanisms. Another interesting feature is that of being able to specify the desired consistency level during query execution. As far as the entities in the database are concerned, the structure can be seen in the following image:



Within the schema, the relationship between ticket and auction is a 1-to-1 relationship. Although relational database theory envisages the merging of this type of entity, a non-relational approach has been used since these two entities are logically separated.

3 User Manual

3.1. Login Page

On this page, the user can log in by entering a username and password. If the user is not already registered, he can do so via the register page.

If you aren't registered yet, [register first!](#)

Username

We'll never share your username with anyone else.

Password

Submit

3.2. Register Page

Here a user can register on the platform and log in.

Username

We'll never share your username with anyone else.

Password

Register

3.3. Home Page

On the home page, the user can see the currently existing auctions and filter them by entering a string that will be used for the search.

Title

test owner

L

Pisa , 2024-02-10 at 23

Auction Details

test owner 2

L

L , 0001-01-01 at 23

Auction Details

3.4. Tickets Page

On the ticket page, the user can view the tickets on the platform and filter them by title. For each ticket, the user can then view the details by clicking on the relevant button.

Title

test

L

Pisa , 2024-02-15 at 23

Details

test owner

L

Pisa , 2024-02-10 at 23

Details

test owner 2

L

L , 0001-01-01 at 23

Details

3.5. Ticket Detail Page

This page shows the details of the selected ticket, and also informs the user whether or not there is an auction for the selected ticket.

test
Artist: L
City: Pisa
Date: 2024-02-15
Hour: 23

Founded auction for this ticket

See Auction

3.6. Profile Page

On this page users can view their tickets and possibly create an auction if one does not already exist, and users can also view the tickets they have won at auctions.

Ticket Owned

Ticket Wonned


test
L
Pisa , 2024-02-15 at 23

test owner 2
L
L , 2024-02-08 at 23

3.7. Add Page

On this page, the user can fill in a form and add a ticket to the platform.

Insert new ticket

Title	<input type="text"/>
City	<input type="text"/>
Artist	<input type="text"/>
Hour	<input type="text"/>
Date	<input type="text" value="gg/mm/aaaa"/> 
<input type="button" value="Submit"/>	

3.8. Old Auction Page

All auctions that have ended can be viewed on this page.

test

L
Pisa , 2024-02-15 at 23
[Details](#)

test owner 2

L
L , 2024-02-08 at 23
[Details](#)

3.9. Auction Detail

This page represents the core of the application, as soon as the user enters the page, the user can join an auction and place a bid, on the right hand side all the bids of the auction being viewed will be displayed.

test auction

Artist: Lorenzo
City: Pisa
Date: 2024-02-26
Hour: 23

Amount

5

Bid

18:49:18

bata

5