

**University of Pisa**

**Distributed Systems and Middleware Technologies**



# **FooTicket**

-

**A distributed ticketing system for concurrent seats reservation of a football event.**

**Students**

**Davide Vigna**

**Alexandre Jean Antoine Camara Ferrer**

# Summary

<b>Abstract.....</b>	<b>3</b>
<b>Requirements.....</b>	<b>4</b>
Functional Requirements.....	4
Non-functional Requirements.....	4
Basic use cases .....	4
<b>Design.....</b>	<b>5</b>
Class diagram .....	5
System Architecture.....	6
<b>Implementation .....</b>	<b>7</b>
Technologies .....	7
Java Web Server.....	7
Erlang Implementation .....	9
Project structure .....	10
Protocols .....	13
<b>User Manual.....</b>	<b>15</b>
Authentication (Login-Sign in) .....	15
Creation of the map (only for Admin) .....	16
Buy a new ticket (only for Buyers).....	17
View purchased tickets (only for Buyers).....	18
Logout.....	18
<b>References .....</b>	<b>19</b>

# Abstract

The project consists in realizing a distributed ticketing system for football matches.

The service is designed for two types of users: administrators and buyers.

The administrators will handle the seats' map of the stadium assigning positions and prices.

Normal users (buyers) can see the corresponding seats 'map and they can select one or more places who want to buy. For both type of users, authentication is required. The seat's map is representable like a matrix (nxm), created at the beginning by administrator.

The map's structure is handled on external servers. Concurrency is managed and techniques for reducing server and network overhead are implemented. Communication errors and network problems have been considered to avoid the lock down of the entire system.

The goal is to create a high available distributed system with good performances.

The user can interact with the system sending HTTP requests (Get-Post) handled by a java web server. The server sends appropriate requests to external erlang nodes in order to update or retrieve information about the status of the seat's map. For this purpose, a specific communication protocol has been designed.

# Requirements

## Functional Requirements

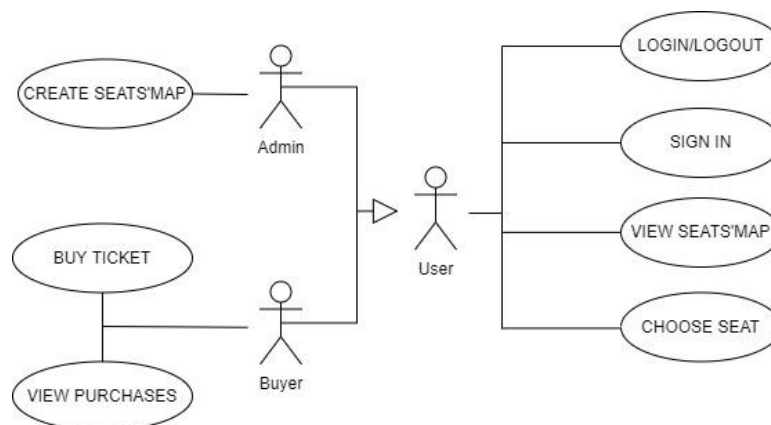
- A general user can log in to the system
- A general user can see the seats' map available in the stadium
- A general user can log out from the system
- An administrator can create a new map specifying the places (dimensions) and prices. For each seat in the map, he/she can also specify if it is reserved or not
- A buyer can select one or more seats and confirm his/her order
- A buyer can see a summarization of his/her purchased tickets

## Non-functional Requirements

- The system must be always **available** and be **fault tolerant** as possible. Communication errors should be properly handled and shown to the final user in a fine way
- The system should be **scalable**. It is possible to adding new nodes at runtime in order to enforce the replication, without the need to stop the entire system
- The system should be **responsive** as possible. A proper configuration of timeout interval and number of replicas influence the waiting time of the user
- The characteristic of **maintainability** is required, the code should be organized in modules with own responsibility, easy to be understood and structured in functions to guarantee more **reusability**.
- One or more persistent units are necessary to save the buyers operation during system life, **persistence**.

## Basic use cases

Here a basic representation of all the possible use cases that a logged user can do.



# Design

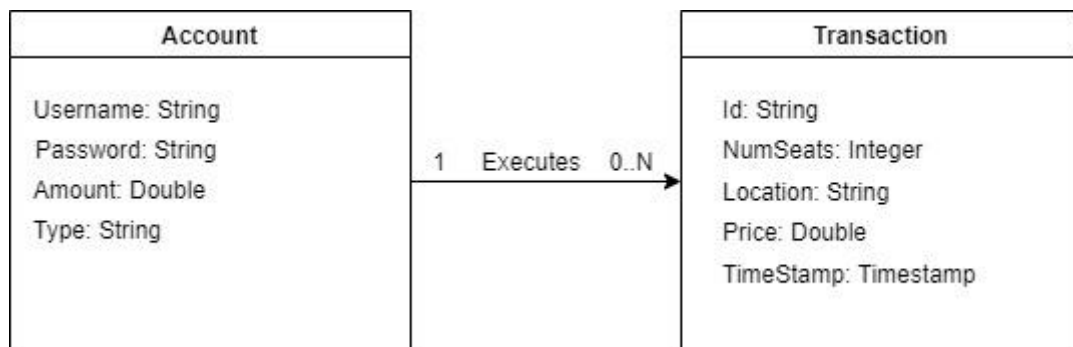
## Class diagram

The data entities identified to store and manage the data on persistent unit are mainly two: **account** and **transaction**. The account entity represents a generic user uniquely identified by the field username. Other fields are related to his/her password, an amount representing his/her actual available money and the attribute type that identifies if an account is referred to an administrator or a buyer. In case of administrator account the field 'amount' is meaningless.

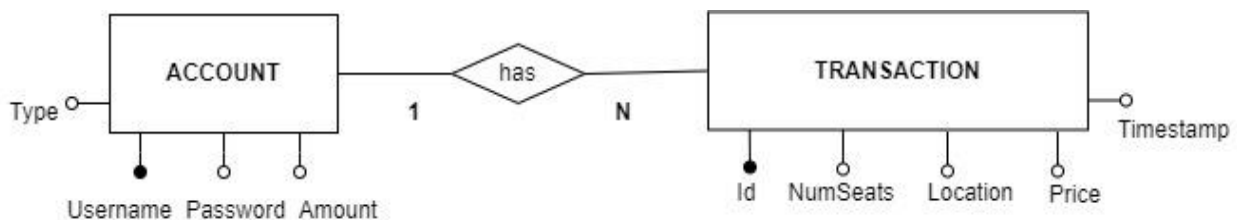
The transaction entity represents a generic user's shop operation who have bought some tickets in the system. It is identified by a uniquely id. The other fields are related to the number of seats bought, the location of the seats (specifying the exactly position in the map), the cost "price" and a timestamp indicating the exactly time the operation has been done.

An additional entity **temp\_transaction** specular to transaction is used to save the temporarily transaction (not yet confirmed by the buyer), useful for recovering operation in case of failures.

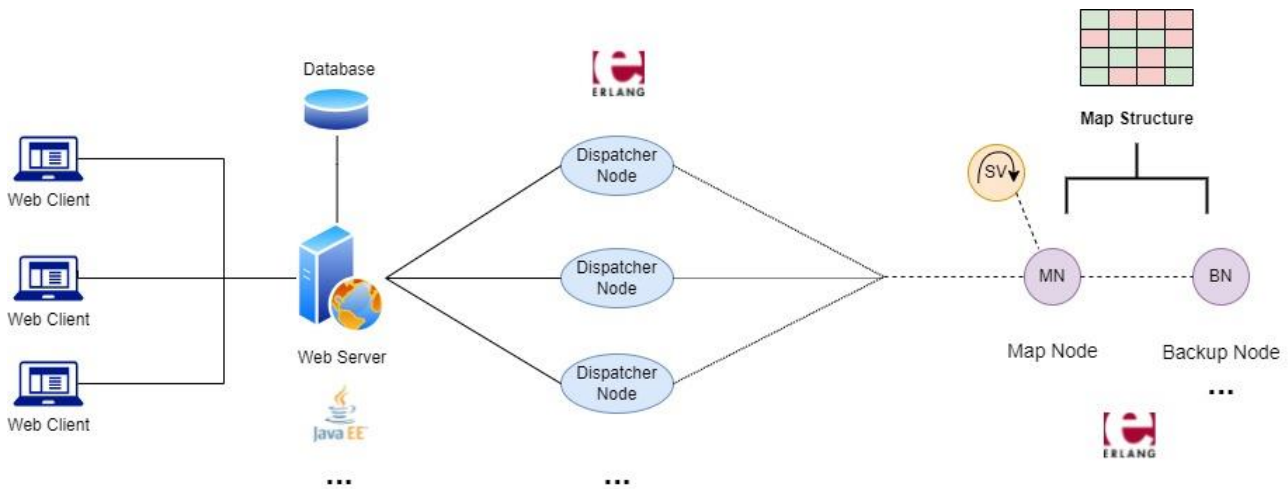
The relation between the account entities and transaction is **one to many** in the sense the one account can, optionally buy some tickets (execute more different transactions). Instead, a transaction is related just to a specific account.



This situation is very simple and minimal. Here there is a representation of ER Diagram:



## System Architecture



The first part on the left represents the structure developed in java making use **Java EE Web Server**. A user can interact with the service through HTML5 pages on the **web browser** which sends request to the server. For sake of simplicity, the image shows just only one web server. In a distributed situation the web servers could be more than one. The web server contacts also a **remote database** to store and retrieve information related to user authentication and purchased tickets by a specific logged user. The decision to use a relational database is due to the simplicity of data to be stored.

The next part is developed in Erlang and it consists of the following components:

- **Dispatcher Nodes**, they act as first layer to dispatch webservers' request to the following map node. At the first start of the system their number must be defined and each web server should know about their presence. It's possible to add new one manually during system life, but of course each webserver should be informed and it can be done gradually without the need of shut down the entire system.
- **Map Node**, it is in charge to maintain the seats map available to the dispatchers. The idea is delegating this node just to handle the map resource and avoiding webservers directly communicate with him.
- **Backup Node**, it is used to store the map status as map node, but it is not able to answer to dispatcher nodes because distributed consistency is not implemented here. It will return the map (if present) to map node when it will be restarted in order to reestablish a consistent situation.
- **Supervisor Node**, its task is to recover Map Node whenever it fails. The current implementation consists only in a single node. To have a more robust guaranties, a tree of supervisor nodes should be implemented.

A **caching system** (on web server side) is implemented to avoid map node's overhead and to reduce the response time between dispatcher and map node.

The original idea was to implement the mutual exclusion in a distributed way (like Ricart-Agrawala Algorithm) but because of time and complexity, the decision of a more centralized mutex solution prevailed.

# Implementation

## Technologies

### Java Web Server



The web server part is developed using a Spring Framework: **Spring Boot**. Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that can be "just run".

There is no need to deploy a WAR file on external server because it includes embedded servlet container like Tomcat (used in the project). There is a minimal configuration setup and no requirement for XML configuration. All the application parameters can be set on external application.properties file.

From the practical point of view, it is fast and easy to learn. It lets the programmer focus on what is the business logic of the application increasing the productivity. It is also high flexible and offers the possibility to integrate external libraries and extensions.

The following lines will be dedicated to all aspects about Spring studied and customized in this project:

- **Embedded Tomcat:** It's the default servlet container configured by spring boot. Custom configurations are introduced in properties file, like the maximum threads handled (parallelism) and the session timeout;
- **Spring MVC:** The web application is based on the paradigm Model-View-Controller to separate the business logic from the presentation layer. All the controllers' classes are annotated by @Controller annotation. Inside the class methods (Get / Post) and paths are specified to tell Spring which request it should handle. In this project the 3 logical components are separated in different packages.
- **REST Service:** In this project it is implemented also a REST web service used to expose to external the seat map. This is designed to have an easy way to get the map from erlang nodes in a very simple format: JSON. It could be useful for further implementations. The methods that use the service are basically 2 (reading of the map, selection/ deselection of a seat). The annotation to use on a rest controller is @RestController.
- **Dependency Injection:** This allows for coupling of components and moves the responsibility of managing components onto the container. In the Spring framework, the interface *ApplicationContext* represents the IoC container. The Spring container is responsible for instantiating, configuring and assembling

objects known as *beans*, as well as managing their life cycles. In this project we used this principle to inject into the controller all the bean classes like services or more generically components. This means that at the initialization of the web context these objects are instantiated and made available to all who request. In the controller class it's necessary to specify the injected services with the annotation of `@Autowired`.

- **Services:** Spring `@Service` annotation is used with classes that provide some business functionalities. Spring context will autodetect these classes when annotation-based configuration and classpath scanning is used. We decided to separate the definition of a service and its implementation, so we used this annotation on the service interfaces. It's possible to create more than one implementation of a services class, so to tell to the context what implementation use at start-up, it's necessary to mark all the auto wired services with the annotation `@Qualifier` and the name of specific implementation to use. An example on this project can be on *AccountService* and *AccountLocalService*. They both implement the same interface but the second service is used just to test application without a remote access to the database.
- **Spring Security:** Not all the resources and the services are available for all. Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications. In this project, we customized the spring authentication in order to define 2 types of users with two different roles ( `ROLE_ADMIN` and `ROLE_BUYER`). We configured the framework to reject unregistered users redirecting to the login web page. Finally, we set the filter to be applied on all the requests coming to the web server in order to protect the resources from users who doesn't have the privilege and the role to use them.
- **Spring Data JPA:** This module deals with enhanced support for JPA based data access layers. Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that's actually needed. We implemented some repository interfaces with custom finder methods and Spring "automatically" provides the implementation at start-up of the system. Thanks to this functionality we did not have to write any SQL query, we just wrote the generic sign method with proper format and we had to extend the repository interface with *CrudRepository* that provides the basic (create, read , update and delete ) operations on a persistent unit.
- **Spring Session:** The web server session is very important because it allows to identify all the operation and data about a user. Its creation happens once the user is logged in and it terminates when he/she logs out or if the session timeout expired (this means that the user has not sent any requests for a certain time). Spring Session makes it trivial to support sessions without being tied to an application container specific solution. We introduced a custom *HttpSessionListenerConfig* to intercept the event when session expired to avoid to block the seats map reserved from a user who forgot to buy definitively.
- **Error Handling:** We decided to create a specific controller to handle some of the possible errors that can happen during the web server life in order show different views for different errors. The types of error handled are (403,404,500). For the 500 error (Internal Server Error), we dedicated a specific web page for the detailed log of the error with all its characteristic to make the life easier for the person who will debug.
- **Thymeleaf:** It's a modern server-side java template engine able to apply a set of transformations to template files in order to display data and/or text produced by your applications. Its architecture allows a fast processing of templates, relying on intelligent caching of parsed files in order to use the least



possible number of I/O operations during execution. The reason why we choose Thymeleaf respect to JSP template engine is because Thymeleaf is closer to the HTML format and you have more control to the template code. The development speed is a little bit bigger with Thymeleaf. All the web pages are located in resource folder. Some contents are repeated in all the pages so to avoid to repeat the same code in all the points, we have created *fragment template* that are included in all the points necessary. This is the case of user detail information, always present on the top of the page.

- **JUnit and Integration tests:** In this project we added some dependencies for making tests. In particular thanks to some annotations, we tested all the controllers' methods. During the integration tests the web server is not really started so it's necessary to "simulate" the creation of the various bean in the context by using the annotation `@Mock` and if we want to simulate the behaviour of the web application for a given user, it's necessary to simulate the authentication too.

## Erlang Implementation



As we can see in the schema above, we can distinguish 4 erlang modules: dispatcher, replica, and replica\_bu (for back up), that have gen\_server behaviour, and sup\_replica, that has supervisor behaviour.

- **Gen\_server:** it is a specific finite state machine working like a server. It can handle different type of event:
  - o **Synchronous** request with `handle_call`
  - o **Asynchronous** request with `handle_cast`
  - o **Other messages** (not defined in OTP specification) with `handle_info`
- **Supervisor:** he supervises other processes called child processes that can either be another supervisor or a worker process.

We defined a fixed number of nodes that run the dispatcher module(`dispatcher_n@localhost`), one with the supervisor that also runs the replica module (`replica@localhost`) and one last node for the replica\_bu.

We used different erlang types for the requests:

- Integers for the number of rows, number of seats by rows, the price and for the "hash" (at the end we used a counter to notify the java server that the map has been modified or not).
- Atoms for the command (for example create, show, ...).
- Strings for the process Id we need.
- A map that represents the seats, with the number of the seat ("Row\_Seat") as the key and a string ("not\_used or used") as the value.

## Project structure

The project is divided in several packages, each with its own responsibility.

- **Configuration** : there are all the java custom configuration classes
  - *CustomUrlAuthenticationSuccessHandler* for managing the successful authentication and for determining the proper web page depending on the role of the authenticated user
  - *GlobalConfiguration*, it contains all the global constants of applications
  - *HttpSessionListenerConfig* for managing session (creation and destroy). In the destroy method it calls the service to remove reserved places on erlang map
  - *SecurityConfiguration*, it contains the filter configuration for servlet requests and for user authentication.
- **Controller**: there are all the java controller classes
  - *AdminController*, it handles all the possible requests that comes from an Administrator
  - *BuyerController*, it handles all the possible requests that comes from a Buyer
  - *CustomErrorController*, it handles all the possible errors happening during the server life
  - *HomeController*, it handles the request of authentication and sign in. It also handles the redirect request in case an authenticated user tries to the login page
  - *MapController* this is the REST controller and it handles the asynchronous requests coming from the client about the reading and the modification of the map
- **DTO** (data transfer object) : all the classes used for exchanging information
  - *CreateMapDTO*, the class used to represent the information of the map created to send from the client to web server
  - *MapDTO*, the class used to answer the map present on server side
  - *SeatInfo*, the class used to represent the selection/deselection of seat
- **Erlang**:

### Dispatchers:

From the java server, we use the Jinterface module to create an erlang node that we will call javaNode. The messages are sent as OtpErlangTuple.

The role of the dispatchers will be to transmit the request to the replica, and then send the response to the java server.

The java node sends to a random dispatcher a special command that depends on what we need (it arrives in the handle\_info function):

- Creation of the map: {javaNodePID, create, NRows, NCols, Price, Map}.  
The map is created directly in the java server for simplicity reasons. We also want to save the number of rows and columns because as the data structure we used is a map, there is no information about it so it can be hard to display after (when we send the map, it got mix).
- Select or unselect a seat: {javaNodePID, select/unselect, Seat}.
- Show the map: {javaNodePID, show, Hash} The dispatcher will also receive answers from the replica to notify the java server if the request has been handled or not and replies also the elements requested.

## Replica:

The main objective of the replica is to save to map and make the modifications requested to it. For saving the different element (Map, Hash, Nrows, Ncols, Price), we use an ets (Erlang Term Storage) table that is passed throw the State to get saved.

For each dispatcher request, the replica checks if it is possible and then replies to the dispatcher:

- Creation of the map: First the replica checks if a map is already saved in the ets table with a member request, that returns a boolean. If the map exists, we return to the dispatcher a tuple with an atom "alreadyExist"; if not, it adds all the elements that are sent by the java server to the ets table and replies to the dispatcher a validation atom (created) and the Hash (that has value 0 at the beginning).
- Select or unselect a seat: if we want to select a seat, the replica first checks that the seat is available (its value is "not\_used"). If the seat is available, we update the map with the value "used" for the seat we want to buy and we add 1 to the hash to notify the java server that the map has been updated. For unselecting a seat, the replica works the same way, but checking if the seat has value "used".  
If the seat is not available, the replica returns "notSelected" or "notUnselected" to notify the java server that the modification failed.
- Show the map: The first we want to do is to compare the hash the java server sends to the one saved in the ets table. If the hashes are the same, that means that the java server got the same map that the replica, so we don't need to reply the entire map. In case of the hashes are different, we return the map and the new hash.

**Supervisor:** the goal of the supervisor is to maintain the replica up. If the replica fails for whatever reason, it will be restarted by the supervisor. The supervisor has many options:

- Supervisor flags: they represent the way the supervisor will work:
  - o Strategy: it is the restart strategy of the child; we used the one\_for\_one option: if one child process terminates and is to be restarted, only this process is affected.
  - o Intensity and Period: they are used to prevent the child to go into an infinite loop of termination and restarts. If we set MaxRestart as the intensity and MaxTime as the period, if more than MaxRestart occurs in less than MaxTime.
- Child specification:
  - o Id: this is the key that the supervisor use to identify the child.
  - o Start: it is the function that is started by the supervisor as a tuple ({Module, Function, Arguments}).
  - o Restart: defines when a terminated child process should be restarted. We used "permanent", so the child is always restarted.
  - o Type: specifies if the child process is a supervisor or a worker (in our case, it is a worker).
  - o Module: it is the module used to determine which processes are using a certain module.

## Replica Back Up:

The supervisor is a great tool that allows us to have the replica node always up, but each time it goes down, we lose the data (Map, ...) because it is saved in the process. The replica back up (BU) is here to save the data in case of the replica dies. Each time the replica starts, it asks the BU if he got a map saved. So, the first time we start both, there is no data in the replica neither in the BU, so nothing happens. Each time the replica updates the map, it sends the new information to the BU, so they got the same information. If the replica

dies and get restarted, it asks again to the BU if he got data; this time, the BU replies to the replica all the data so the entire system can continue working fine.

- **Erlang Interfaces:** there are all the files needed to implement the communication with erlang nodes
  - *DispatcherConversionUtilities*, it contains a series of static method for the conversion (JSON-OtpErlangTuple)
  - *DispatcherInterface*, it is the core object of the project. It is initialized once as a java bean. It contains all the information to contacts dispatchers and also contains a cached version of the seat's map received. A technique of repeating send operation is implemented inside.
  - *MapState*, it a java class to represent the information exchanged throw java and erlang
- **Model:** there are the three models used for persisting information on database
  - *Account*
  - *TempTransaction*
  - *Transaction*
- **Repository:** there are all the interfaces which define the methods to use to have access to database
  - *IAccountRepository*
  - *ITempTransactionRepository*
  - *ITransactionRepository*
- **Service:** there are all the service interfaces that define the method to be implemented, one for model
  - *IAccountService*
  - *ITempTransactionService*
  - *ITransactionService*
- **ServiceImpl:** there are all the classes that implements the previous interfaces
  - *AccountDetailsService*, implement the logic to recognize what type of user is authenticated
  - *AccountService*, handles the account information with database
  - *AccountServiceLOCAL*, handles the account information without database, just for tests
  - *TempTransactionService*, used to save and recover transaction in case of errors on server
  - *TransactionService*, used to save the purchased tickets
- **FootTicketsApplication** it is the entry class to start the web application
- **Resources**, here are present all the static file in the project
  - *db*, contains the static script SQL to be executed each time at the start of the system
  - *logs*, used to print the logs instead of the console
  - *static*
    - *css*, all the styles file
    - *script*, all the javascript and jquery file to handle web user interaction
  - *templates*, all the HTML file to show user content information
    - *errors*, the pages related to the errors output
    - *fragments*, the part of HTML code that is repeated in several pages
- **Application.properties** the very important file used to configure parameter of web application
- **Test:** contains the test of all the controller class of the application

## Protocols

In this section we show the protocols used to send and receive request from Java node to Erlang Nodes. They are basically 3:

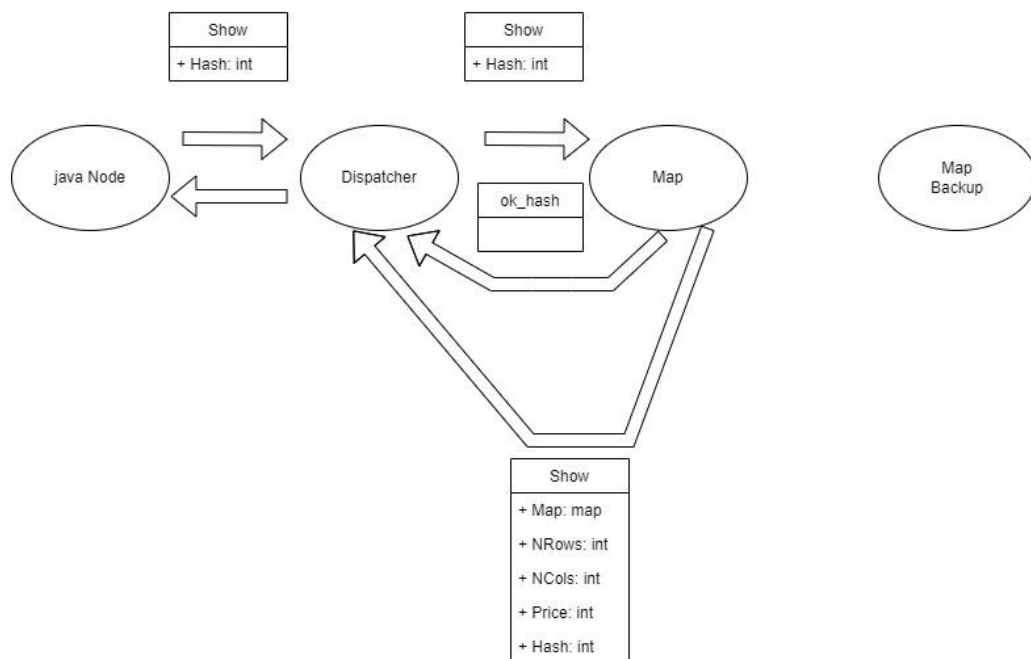
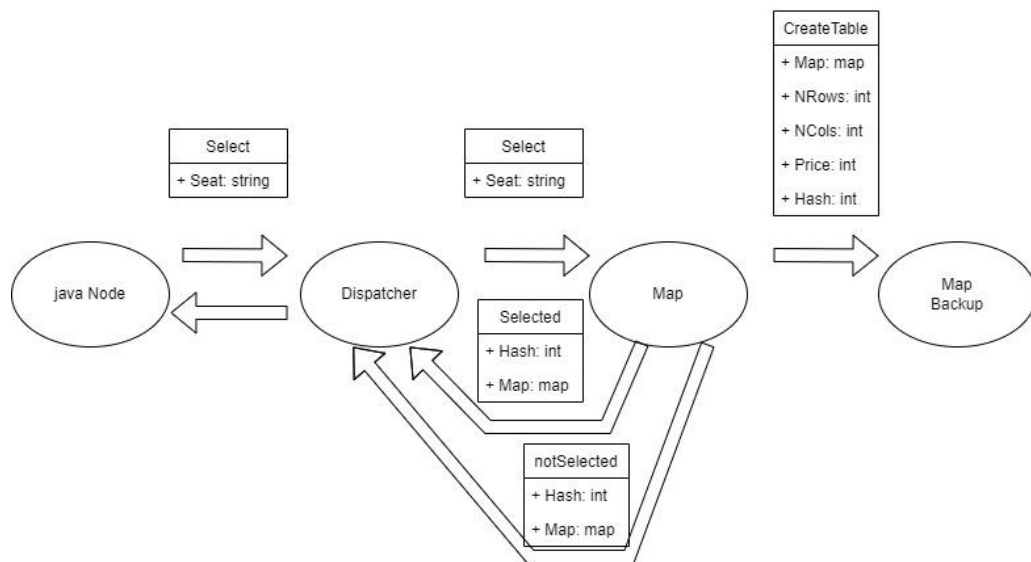
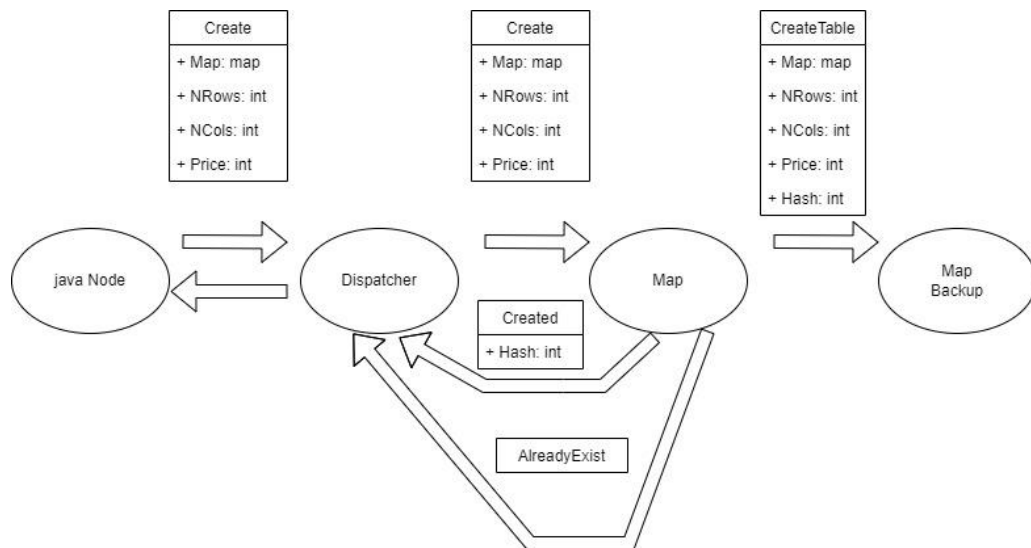
- creation of the map
- selection/deselection of a place
- show of the map

The first starts from an administrator request and it consists of a synchronous call. The other two are made by a buyer user and they are both asynchronous requests, coming from the web browser (ajax-call). In this way the user has not to reload the page to see changings on the screen and he / she is not forced to necessary wait for the answer and he/she can do somewhat else.

Each time the *java node* has to send a request to a dispatcher it uses the following strategy:

- 1) Look if there are some dispatchers reachable (look if a list of *dispatcherNodesAvailables* is empty)
  - a. If it is empty: try to ping all the dispatcher, in order to determine if they are available now
  - b. If someone reply positive it adds the dispatcher to the list
  - c. If no one is reachable, it avoids to send the message (output of timeout). End.
- 2) It selects randomly a reachable dispatcher from the list and tries to send a message
  - a. If it doesn't receive any response message (before timeout), he tries another time until a max number of trials expired
    - i. If the max number of trials expired, it removes the dispatcher from the list and repeat 2
  - b. If it receives a response message (before timeout) this means that's all ok and the nodes have communicated each other. End.
- 3) If at this point any dispatcher is reachable, there could be a network problem so an output message of timeout is returned.

The next image shows a detailed view of what data structures are exchanged during the various requests. The image contains the three protocols described before, in the same order. The select protocol is very similar to deselect so we avoid to represent it.

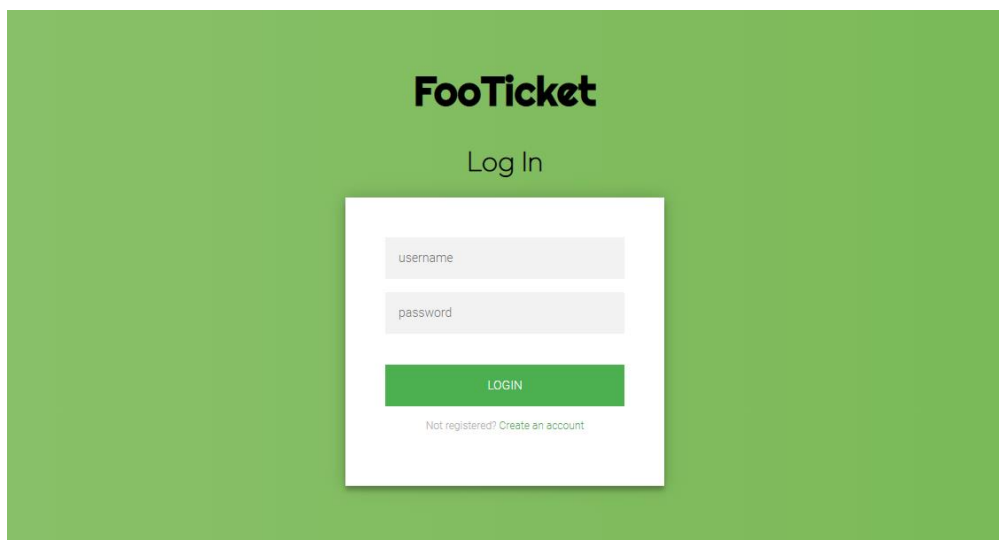


# User Manual

The following lines are dedicated to explain to the final user how the entire system works. For simplicity all the paths indicated are referred to a localhost. Once the application is deployed on external web server, it's necessary to substitute localhost with the address and the port indicated on application.properties file.

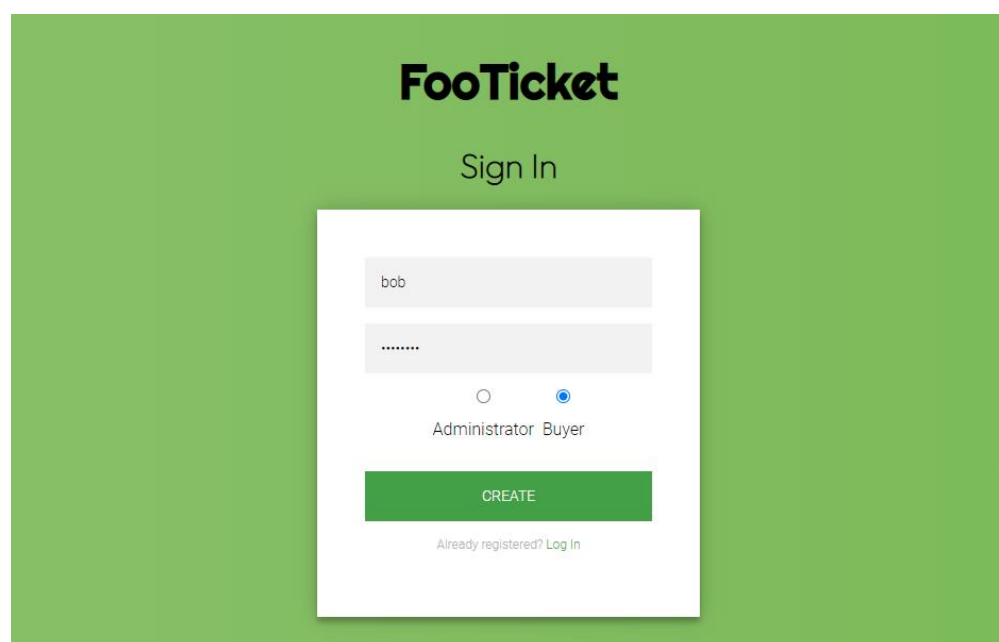
## Authentication (Login-Sign in)

A user that is not authenticated should go on the following path: <http://localhost:8080/login>. The following page will be shown.



The screenshot shows the 'FooTicket' login interface. It features a green background with the 'FooTicket' logo at the top. Below the logo is the text 'Log In'. A white form box contains two input fields labeled 'username' and 'password', followed by a green 'LOGIN' button. At the bottom of the form, there is a link that says 'Not registered? Create an account'.

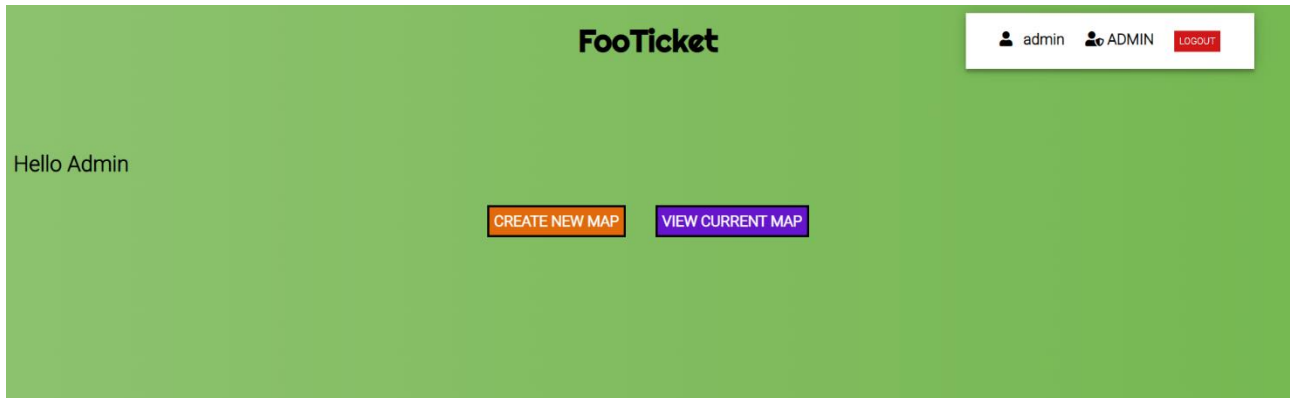
The user has to insert its own credentials (username and password) then press Login button. If the user is not registered to the system, a registration is required. Clicking on Create an account a new popup will be shown. After inserting the credentials and clicking on Create button, the registration will be executed.



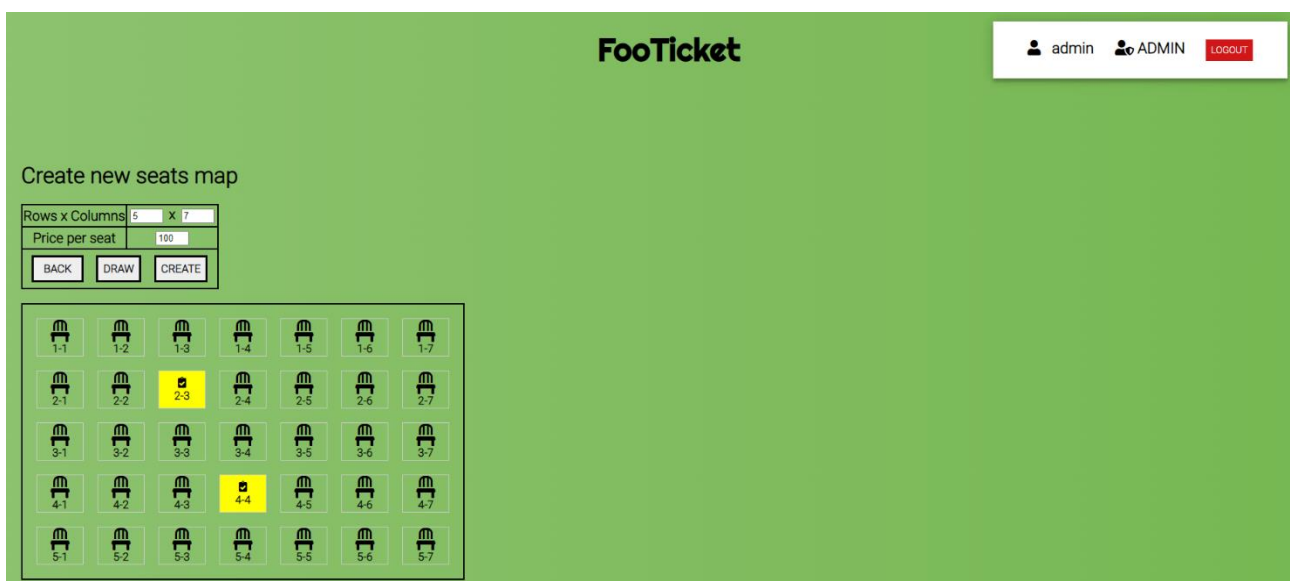
The screenshot shows the 'FooTicket' sign-in interface. It features a green background with the 'FooTicket' logo at the top. Below the logo is the text 'Sign In'. A white form box contains two input fields, the first with the text 'bob' and the second with masked characters '\*\*\*\*\*'. Below these fields are two radio buttons; the first is unselected and the second is selected. Under the radio buttons are the labels 'Administrator' and 'Buyer'. A green 'CREATE' button is positioned below the radio buttons. At the bottom of the form, there is a link that says 'Already registered? Log In'.

## Creation of the map (only for Admin)

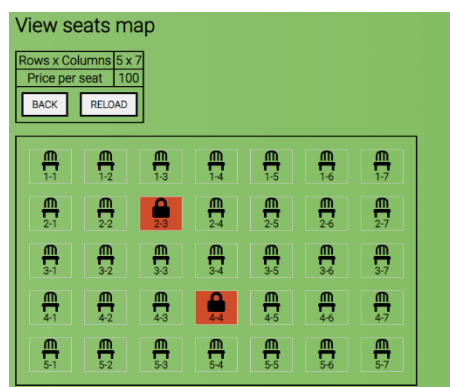
Once authenticated the administrator who wants to create a map should click on the button Create New Map.



In the next page he/she can specify the number of seats per row and columns and the price of each. He/she can also select some places to be reserved at first, so any further buyer could not select them. These places are underlined in yellow.



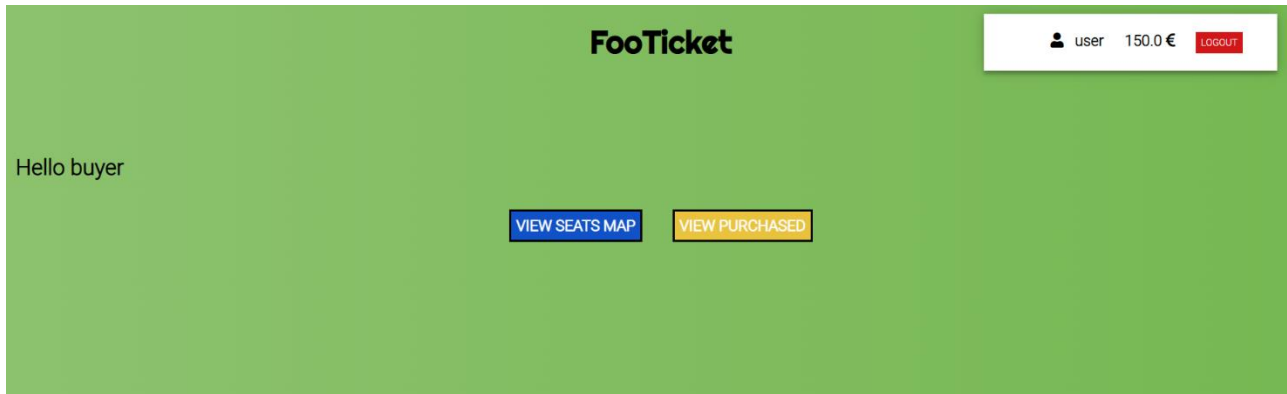
The map will be created pressing on the button create. A confirmation message is shown in the next page. To have a confirm of the correct operation, the administrator can click on the View Current Map button and have a view about it (violet button).



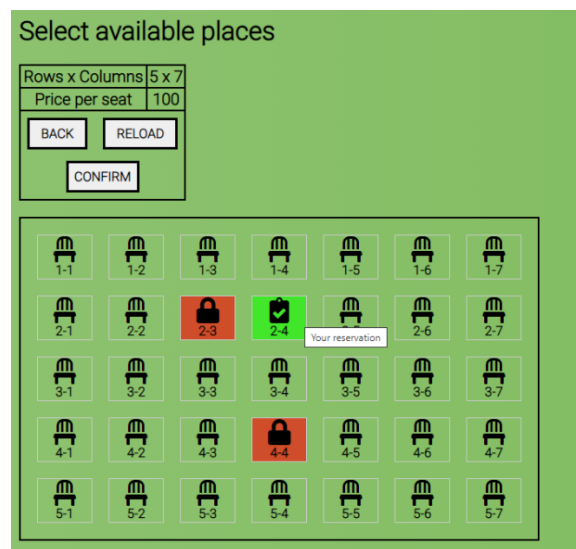


## Buy a new ticket (only for Buyers)

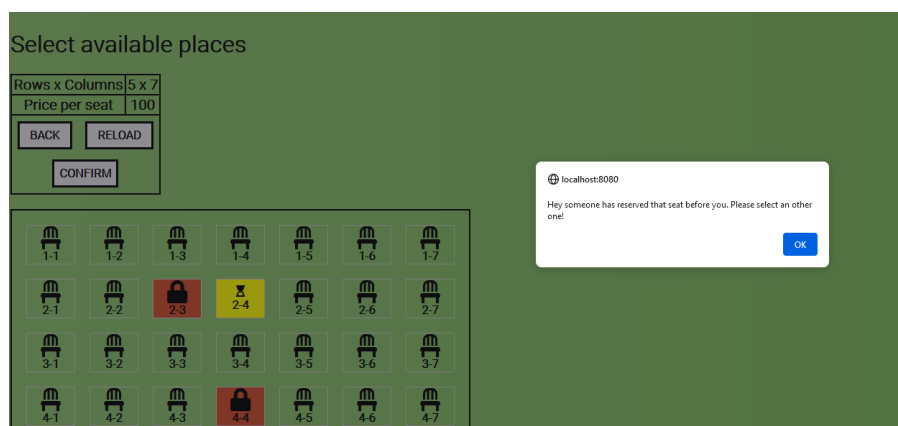
Once authenticated the buyer who wants to shop a new ticket has to click on the View Seats Map button on the main page.



A buyer has to click on an unselected seat (not coloured) to select some seats on the map. The red seats represent the places already selected by someone else. The system will send a request about the reservation and if all is ok, the selected place will be coloured in green.



Instead, if someone else does the same operation at the same time, a buyer can receive a popup message and the place will be re-coloured in red. The operation is like it is never done.



If the user decides to change its own reverberation (deselecting) the place, he/she can do clicking on the green seat. The place will be re-coloured in transparent to indicate that now it is free. Clicking on Reload button the page will be reload and any modification to the map by other user are displayed updated. This operation is also done when the user select/deselect a place.

## View purchased tickets (only for Buyers)

A buyer who wants to see his/her previous purchased ticket can click on the button View Purchased. The next page will show a table with the last shops done.

FooTicket

user 150.0 €
LOGOUT

Purchased Tickets

Id	Num Seats	Location	Price	Timestamp
8	1	0,2	50.0	2022-01-31T23:32:28
7	1	0,2	50.0	2022-01-31T23:31:08
6	1	0,1	50.0	2022-01-30T23:41:27
5	2	0,3 1,3	100.0	2022-01-28T18:13:38
4	2	1,3 1,4	100.0	2022-01-28T18:02:58
3	3	1,3 2,2 3,2	150.0	2022-01-28T17:58:10
1	3	A1-A2-A3	150.0	2022-01-28T16:37:32
2	2	B1-B2	100.0	2022-01-28T16:37:32

BACK

## Logout

The logout operation can be done in any page of the system clicking on the button Logout on the top of the page. If you are a Buyer user, the system will also try to deselect all your previous selected places (not confirmed) in order free the seats (resource) to other buyers. The system will redirect on the login page.

user 150.0 €
LOGOUT

## References

The source code is available on the GitHub repository at the link:

<https://github.com/AlterVigna/FootTickets>