



UNIVERSITÀ DI PISA

Computer Engineering

Distributed Systems and Middleware Technologies

CinemaBooking Documentation

Jacopo Carlon

Nicola Riccardi

Academic Year : 2024/2025

Contents

1	Project Specifications	2
1.1	Use Cases	2
1.2	Synchronization and Communication Issues	3
2	System Architecture	4
2.1	Deployment	5
2.2	Server Side	5
2.2.1	Main Server	5
2.2.2	Mnesia Database	6
2.2.2.1	Mnesia Data Persistence Options	6
2.2.2.2	Our Mnesia Tables	6
2.2.2.3	Mnesia Atomic Transactions vs TOCTOU	7
2.2.3	Supervisors	7
2.2.4	Main Supervisor	8
2.2.5	Show Monitor	8
2.2.6	Show Handler	8
2.3	Client Side	9
2.3.1	Web-server with Apache Tomcat	9
2.3.2	Login Filter	10
2.3.3	Communication Handler	11
2.3.4	Web Socket Endpoints	11
2.3.5	Java Listener	12
2.3.6	User Interface	12
3	Synchronization and Communication Approach	13
4	UserManual	14
4.1	Customer	14
4.1.1	Customer Registration and Login	14
4.1.2	Customer Page	15
4.1.3	Show Page seen by Customers	16
4.1.4	Cinema Page seen by Customer	17
4.1.5	Browse Shows Page	17
4.2	Cinema	18
4.2.1	Cinema Registration and Login	18
4.2.2	Cinema Page seen by Cinema	19
4.2.3	Show Page seen by its own Cinema	20
4.2.4	Create Show Page	20
5	Bibliography	22

1 Project Specifications

CinemaBooking is a distributed web-app.

Registered Cinemas can create Shows and keep track of bookings.

Registered customers can search available shows, book some seats, and check and modify their own list of bookings, within the Show's deadline.

1.1 Use Cases

An *Unregistered User* can :

- Register to the service as a Customer or as a Cinema.

An *Un-Logged Cinema* can :

- Log in as a Cinema.

An *Un-Logged Customer* can :

- Log in as a Customer.

A *Logged Cinema* can :

- Logout;
- View list of all current Shows;
- View own page, with list of own Shows;
- Create a new Show;
- for a non-own Show, view show details (showName, date, location, cinema, available seats)
- for a own Show, view list of user bookings stored in central server and of user bookings waiting for sync.

A *Logged Customer* can :

- Logout;
- View list of all current Shows;
- View own page, with list of own Bookings (with the last values that were committed to permanent storage in the main server);
- for any Show, view show details (showName, date, location, cinema, available seats), and create a new booking for that show;
- for a booked Show, on top of the above features, see also the booking by the user for that show that are waiting for sync.

The *System* must :

- Remember registered Cinemas and Customers;
- Remember created Shows details;
- Keep a periodically sync-ed memory of number of seats booked by customers for a show.
- Periodically synchronize main permanent storage with booking data (list of customer-seats pairs, and resulting number of available seats) for each active Show.
- Synchronize updated Show data for each user that is in a specific Show page.

The mock-up is contained within a large black rectangular border. It features several smaller boxes and buttons:

- Top Left:** An orange box with the text "This is a Show Name".
- Top Right:** A white box with the text "Cinema Paradiso (Roma, Via dei Galli 42)" and "Show date : 2024-11-30 18:45 A".
- Middle Left:** A white box with the text "Max Seats : 99" and "Available Seats : 55".
- Middle Right:** A white box with the text "Num Seats booked by you : 10".
- Bottom Right (Buttons):** Two rows of buttons. The first row has a blue button labeled "Want to buy seats ?" and a white box containing the number "1". The second row has a red button labeled "Want to free seats ?" and a white box containing the number "1".

Figure 1: Mock-up of a page a Customer sees regarding a Show that is still bookable

1.2 Synchronization and Communication Issues

On the application we will face the following synchronization and communication issues:

- Client nodes looking at the main page need to be synchronized with the same list of available Shows.
- Client nodes looking at a Show page need to be synchronized with the same updated booking data for that Show.
- In case a Customer makes (or cancels) a valid booking for a Show, the Show node will be in charge of communicating to other clients nodes the updates on available seats for that Show.
- When a Customer changes one of their bookings, the actual update of main storage is deferred, in order to potentially group many small changes into a single update and reduce the load on main storage. The server will be in charge of periodically backing up bookings and informing customers that the changes they made were permanently stored.
- In case a Cinema creates a new Show, the server will be in charge of communicating to other clients nodes the newly created Show.

These issues will be addressed in the following sections 2.2.2 and 3.

2 System Architecture

A graphical representation of the system architecture can be seen in 2.

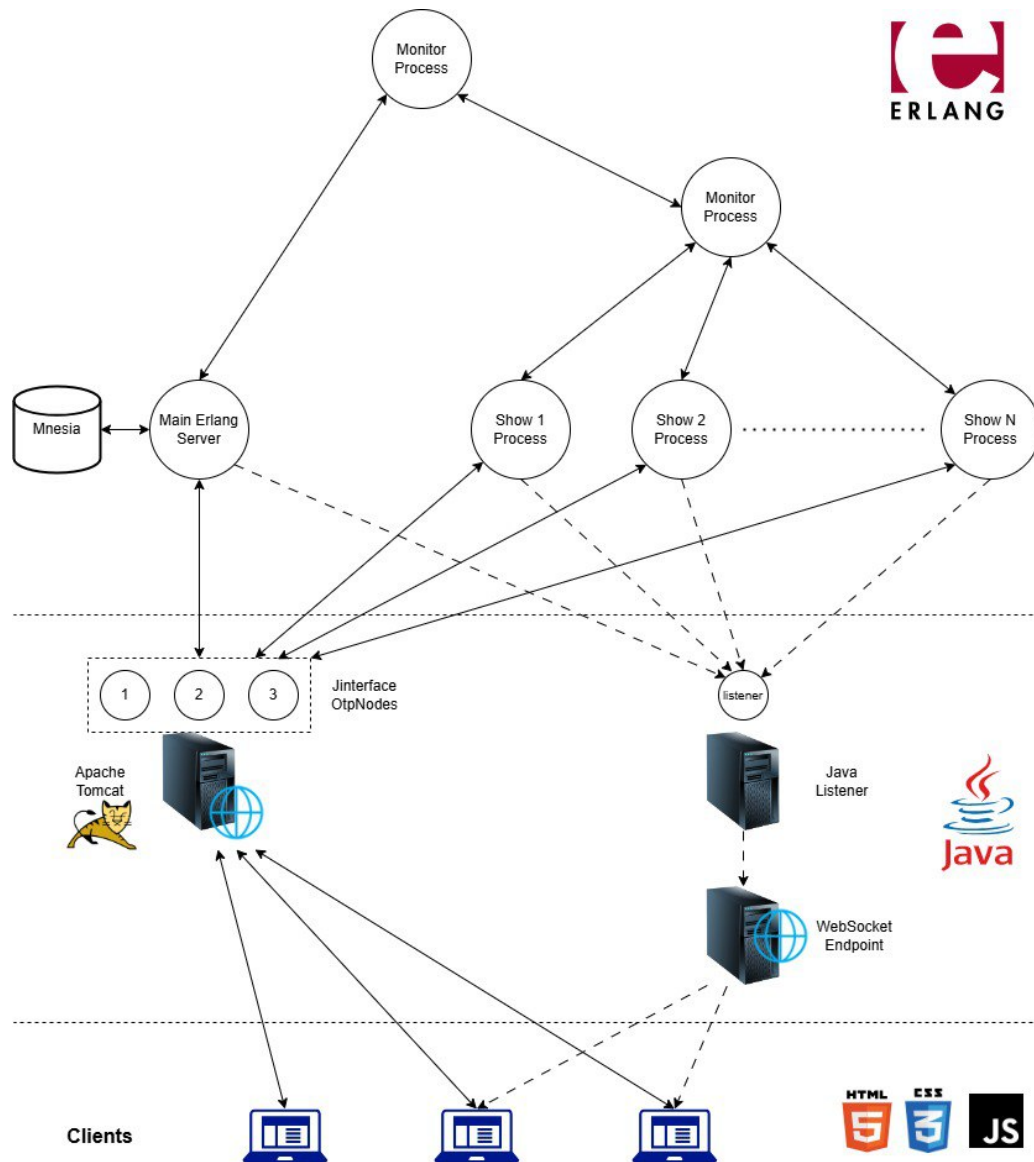


Figure 2: CinemaBooking System Architecture graphical representation

The overall system can be divided in two parts:

- The **server-side** part : developed in Erlang, it is in charge of handling the requests coming from the customers, and has to handle all bookable Shows and their bookings, and has to maintain a periodically updated copy of all sync-ed data, as well as keeping memory of credentials of Customers and Cinemas.
- The **client-side** part : developed in Java, using Servlets and JSP to handle HTTP requests and content generation, and Websocket for page updates; messages received by Websocket endpoints are handled in Javascript. This part is in charge of exchanging data with the server or with specific Show Nodes, creating the GUI and updating it in order to give the users a consistent view of the state of available Shows and of the overall system.

2.1 Deployment

The server-side written in Erlang, and the Java Listener are deployed in the container located at 10.2.1.41 Apache Tomcat web-server, which hosts the Web Application, is deployed 10.2.1.42.

The Web-App can be accessed through the link:

<https://10.2.1.42:8080/CinemaBooking> .

During testing, the period of backup on the main server db has been set to 1 minute.

2.2 Server Side

The server side is written in Erlang [2], and contains the following components:

- Main Server (`main_server.erl`);
- Mnesia database (`database.erl`);
- Show Handlers (`show_handler.erl`);
- Main Supervisor (`main_supervisor.erl`);
- A Monitor for Show Handlers (`show_monitor.erl`).

2.2.1 Main Server

The main server is in charge of managing access to the application through login and handling requests to read/write in the database (including show backups and most user requests).

The functionalities that the server exposes are:

- **Register a new Customer**
The (customer) user proposes a username and password, and will be registered only if the username does not already exist in the database;
- **Register a new Cinema**
The (cinema) user proposes cinemaName, password, and cinemaLocation. For the sake of simplicity, we assume that the validity of the data about a cinema is authenticated via other means. When registered, it will be assigned a cinemaID (which is unique and needed for login);
- **Login as Customer**
The (customer) user must provide username and password. If a match is found in the database, a positive reply is sent back;
- **Login as Cinema**
The (cinema) user must provide cinemaID and password. If a match is found in the database, a positive reply is sent back;
- **Create new Show**
A (cinema) user provides data about one of its own shows, that will be added to the database and assigned a unique showID and an handler node.
- **Retrieve Show Handler PID**
A user provides a showID, that is searched in the database. If a match is found and the related show is not expired, the Process ID of the handler node is returned.
- **Retrieve List of Bookable Shows**
Upon request by a user, the server will send a list of bookable shows (i.e. those shows whose deadline date is in the future).
- **Retrieve Cinema Details**
A user provides a cinemaID and the server returns its name, location and list of shows.
- **Retrieve Customer Details**
A (customer) user can request to access their own private page, containing their bookings.
- **Manage Show Backups**
Periodically (and at the end of their life) Show Handlers send a message with updated list of bookings for their show, and the server stores the updates in the database.

2.2.2 Mnesia Database

This module manages persistent data storage, using the Mnesia key-value DBMS [7] available from Erlang standard library.

2.2.2.1 Mnesia Data Persistence Options

The Mnesia DB has three options for *Data Persistence* , as per documentation :

- **ram_copies** : "A table can be replicated on a number of Erlang nodes. Property `ram_copies` specifies a list of Erlang nodes where RAM copies are kept. These copies can be dumped to disc at regular intervals. However, updates to these copies are not written to disc on a transaction basis."
- **disc_copies** : "This property specifies a list of Erlang nodes where the table is kept in RAM and on disc. All updates of the table are performed in the actual table and are also logged to disc. If a table is of type `disc_copies` at a certain node, the entire table is resident in RAM memory and on disc. Each transaction performed on the table is appended to a LOG file and written into the RAM table."
- **disc_only_copies** : "Some, or all, table replicas can be kept on disc only. These replicas are considerably slower than the RAM-based replicas."

We decided to use the option *disc_copies*, since it allows the database to be resilient against crashes, and lets us restart the process without issues.

The fact that the database is also kept on RAM obviously allows quicker operations (compared with the *disc_only* option).

2.2.2.2 Our Mnesia Tables

In our database we have three tables :

- **cinema** : this table has the attributes :
 - *cinema_id* : **unique** (numerical) CinemaID, assigned by the system upon registration ;
 - *password* : password chosen by the cinema upon registration;
 - *name* : name provided by the cinema upon registration;
 - *address* : address provided by the cinema upon registration;

The cinema table is in charge of remembering registered Cinemas.

- **customer** : this table has the attributes :
 - *username* : **unique** username chosen by the customer upon registration;
 - *password* : password chosen by the customer upon registration;
 - *bookings* : a set of ShowIDs , referencing to Shows to which the user has a booking that has been backed-up into the server.

The customer table is in charge of remembering registered Customers, and the Shows they have booked some seats of.

- **show** :
 - *show_id* : **unique** ShowID, assigned by the system upon creation of the show by a Cinema;
 - *show_name* : ShowName, provided by the Cinema at creation;
 - *show_date* : ShowDate, provided by the Cinema at creation;
 - *cinema_id* : CinemaID, is the unique identifier of the Cinema that created the Show. This field is used to create an index to quickly retrieve the list of Shows of a specific Cinema;
 - *cinema_name* : CinemaName, name of the Cinema that created the Show, used to avoid accessing cinema table, whenever possible;
 - *cinema_location* : CinemaLocation, address of the Cinema that created the Show, used to avoid accessing cinema table, whenever possible;
 - *max_seats* : MaxSeats, maximum number of seats bookable for this show, provided by the Cinema at creation;
 - *curr_avail_seats* : CurrentlyAvailableSeats, number of seats still bookable for this show, as per last back-up. This number is updated when the ShowNode executes a backup on the server;
 - *old_show* : IsEnded boolean value . If a show IsEnded, its deadline is passed and it is no longer bookable.
 - *pid* : **unique** PID of the Erlang Node that is the Handler for this Show (if it exists).

- *bookings* : map of <CustomerUsername, NumOfSeats> pairs, updated when the ShowNode executes a backup on the server.

The show table is responsible for all data regarding a Show, and for keeping the periodically updated map of customer-bookings_for_this_show .

We decided to have the Show table contain some redundant information about its cinema in order to avoid needing to query both Show and Cinema tables during the queries that need joined information (such as when loading the CustomerPage, or ShowPage).

2.2.2.3 Mnesia Atomic Transactions vs TOCTOU

We noticed a possible temporization bug called **TOCTOU** (*Time Of Control Time Of Use*). A basic example of such bug would be :

- Alice tries to CustomerRegister as "Pippo, AlicePWD"
- the system (triggered by Alice query) reads the Customer table searching for username "Pippo", finding nobody
- Bob tries to CustomerRegister as "Pippo, BobPWD"
- the system (triggered by Bob query) reads the Customer table searching for username "Pippo", finding nobody
- the system (following Alice request) writes in the table Customer by inserting the new user (Pippo, AlicePWD)
- the system (following Bob request) writes in the table Customer by inserting the new user (Pippo, BobPWD)

Since on table creation username was the first attribute (and thus Mnesia considers it a unique key [8]), the write operation by Bob will overwrite Alice's, thus the created user will have username Pippo, and password BobPWD.

This problem can be easily avoided by using *Mnesia's Atomic Transactions* : `transaction` [6].

An atomic transaction ensures that all operations inside it are performed in an atomic way (thus making it impossible for Bob's Read to happen before Alice's Write).

2.2.3 Supervisors

It is possible to see graphically the relationship between our Erlang Processes in figure 3. The monitoring for the system is performed by the **Main Supervisor** and the **Show Monitor**.

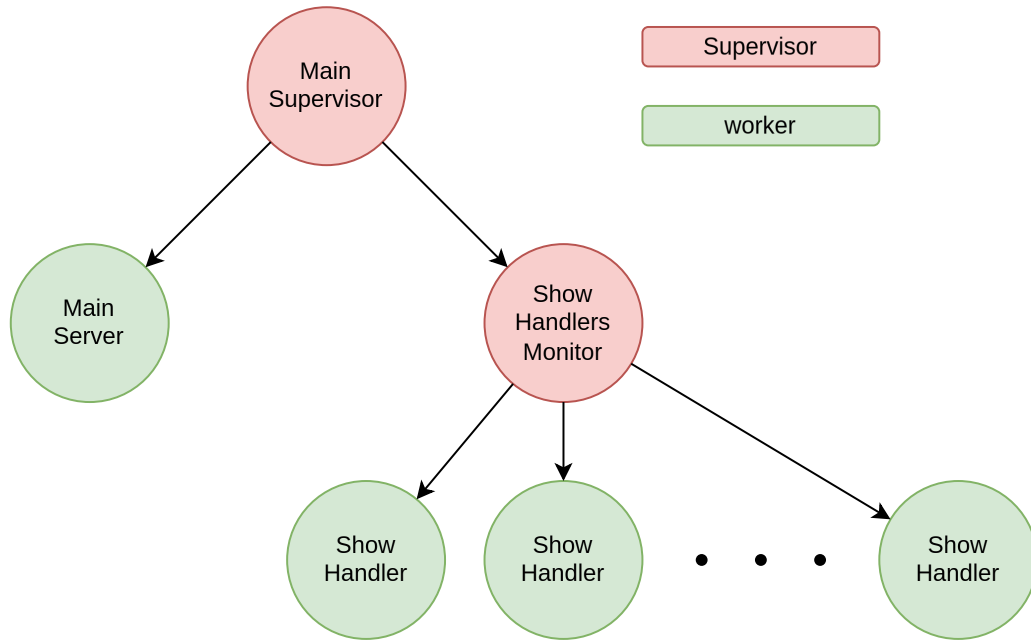


Figure 3: CinemaBooking System Architecture graphical representation

2.2.4 Main Supervisor

Main Supervisor was implemented by using Erlang *behaviour* "supervisor". Its children are statically defined and are the Main Server and the Show Monitor, both unique and permanent (always restarted when they crash).

Main Supervisor was initialized with the following flags:

- *strategy* => *one_for_one*
- *intensity* => *1*
- *period* => *5*

These flags mean that if one of the monitored process crashes, only that process will be restarted (*one_for_one*), and if a number of monitored processes greater than 1 crashes (*intensity*) within within 5 seconds (*period*), then the whole system is stopped.

2.2.5 Show Monitor

Show Monitor is a custom supervisor which is in charge of monitoring a dynamic number of Show Handlers and restarting them if they crashes. In particular, when a monitored Show Handler crashes, Show Monitor automatically restarts the handler and notifies the Main Server, sending it the new process ID associated to the show. The Main Server is in charge of providing the backup to the new handler node.

Therefore, when a Show Handler crashes, only the bookings that were made (or changed) after the last backup are lost, so users can be sure that once a booking is committed to main memory, it won't be lost.

If the Show Monitor crashes, it will be restarted by the Main Supervisor, but the list of Show Handlers that is was monitoring until then is lost. Handlers will still continue to run independently until the show deadline is reached, but if they crash they won't be restarted.

2.2.6 Show Handler

Each show that is still bookable (i.e. whose deadline date has passed yet) is managed by a **Show Handler** running on a private node. The Show Handler is in charge of:

- keeping two separated data structures for committed bookings and updates that will be sent in the next backup;
- providing updated data about available seats and bookings to users;

- handling booking requests by checking if they are valid and storing with the other updates waiting for backup;
- sending periodical backup messages to Main Server if there are pending updates;
- notifying the Java listener whenever its internal state changes (after new pending bookings and backups);
- sending the Main Server a last update once the deadline date is reached.

2.3 Client Side

The web application shown to the users is generated via Apache Tomcat Web-Server [1], while the page updates are managed asynchronously via web-sockets; the front-end part is done with Java, JavaScript, and with the CSS-JavaScript framework **Bootstrapp**.

2.3.1 Web-server with Apache Tomcat

In Apache Tomcat web-server, the HTTP requests arriving from users are first processed by **Java Servlets**.

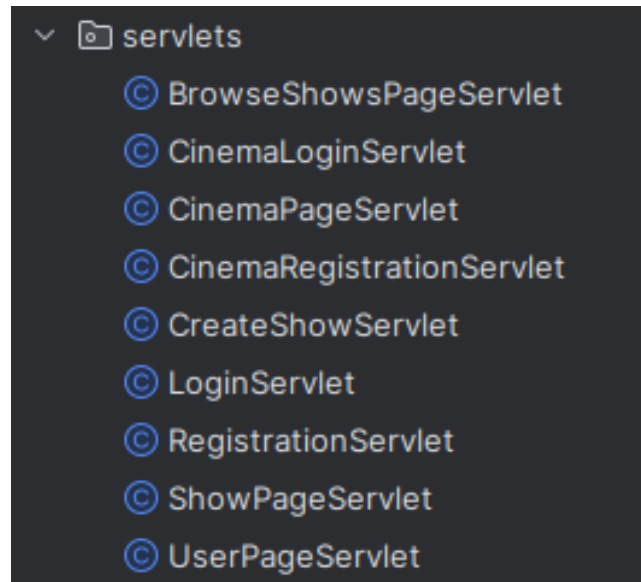


Figure 4: CinemaBookings Java Servlets

As shown in the above figure 4, there is one JavaServlet for each page of the web-application.

These Servlets mainly cover the following purposes :

- getting data from the Main_Server(MS), or from a specific Show_Handler(SH);
- execute operations like registrations on MS, or creations of new bookings on a SH.
- updating Session Attributes depending on the success of some operation, or on the state of the system.
- possibly redirecting the user to other pages, depending on the status of the system (e.g. SH is down, and thus Show_Page cannot be accessed)

The Webapp folder is organized as in image 5 :

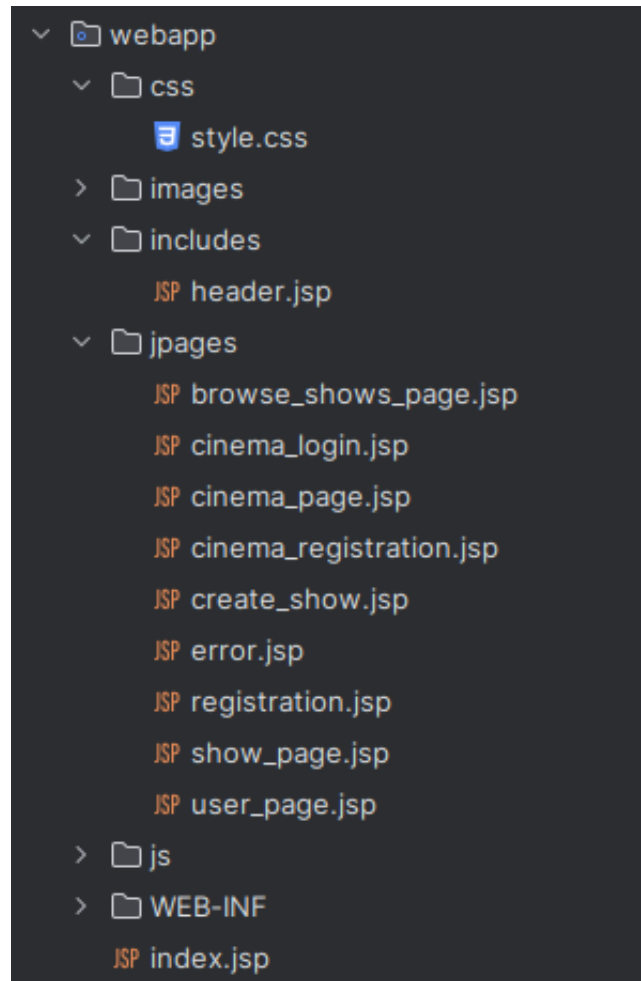


Figure 5: CinemaBookings Webapp Structure

The actual HTML code is generated via JSP "scriptlets" (which allow easier managing of data from the Java side), allowing for a clean separation between the "**View**" part and the rest of the application.

2.3.2 Login Filter

In order to add some layer of security to the application, we decide to use the Filter interface [3] :

The Filter folder is organized as follows :

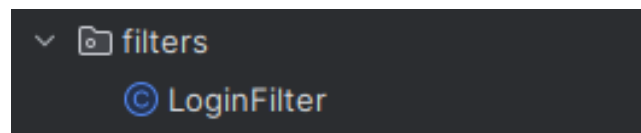


Figure 6: CinemaBookings Filters

- **LoginFilter** : is responsible to check access to all pages that don't cover login or registration functionalities , since all other pages in our website require the users to be logged : *CustomerPage*, *CinemaPage*, *ShowPage*, *BrowseShowsPage*, *CreateShowPage* .

This filter ensures that requests that arrive to our "protected" pages come in fact from already logged-in users, by checking that the Session Attributes "username" and "is_a_cinema" are properly set. Should something be wrong, the user is redirected to the LoginPage.

Some more "sensitive" pages in our WebApp are the CinemaPage , ShowPage and UserPage. However, considering how their behaviour strongly depends on who is requesting them, we decided to include them firstly in the LoginFilter as a "is-logged" check, while more detailed checks are made in the respective Servlet.java files.

2.3.3 Communication Handler

In order to update the "Model" part of the application, we need to interact with the Erlang back-end.

This interaction is done via the use of a CommunicationHandler :

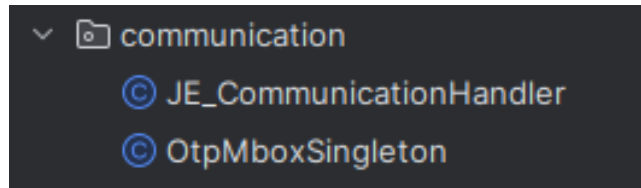


Figure 7: CinemaBookings CommunicationHandler

We use the **Jinterface Package** [5] to manage the communication with the Erlang modules. For each user, there is a different "mailbox" contained in a single "Erlang" node, obtained from the unique session token of the user.

This mailbox is used when request/reply communications happen with the Main Erlang Server or with a specific Show Handler. These communication always return, either with the requested data (e.g. list of bookings of a customer), or with a value representing the result of the requested operation (e.g. success/failure of creation of new show by cinema).

2.3.4 Web Socket Endpoints

In order to dynamically update a page without having to instead periodically re-load it, we used the Java Web Socket Endpoint [4] technology :

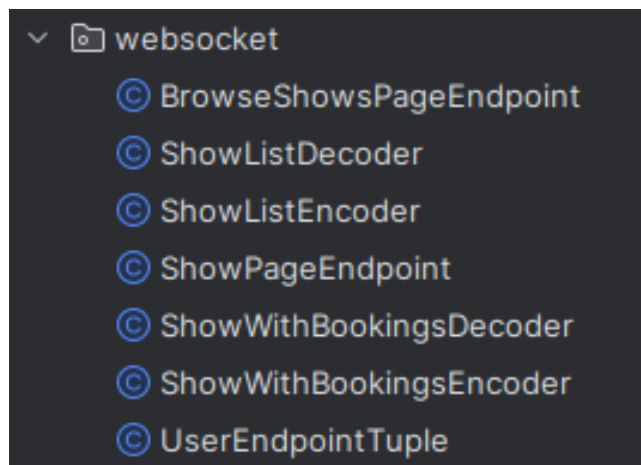


Figure 8: CinemaBookings WebSockets

As we can see in figure 8, we use two web-socket endpoints :

- **ShowPageEndpoint** : this endpoint is responsible for updating the show page. The page behaves differently depending on whether it has been requested by a cinema or by a customer.

For the Cinema, we load the saved show data and list of bookings, and also the waiting-for-backup show data and list of bookings. On the other hand, for the Customer we don't show the two list of bookings, but only the (possibly two) bookings made by customer themselves.

Thus, the work of the WebSocket Endpoint is getting the proper values(or lists) from the Show Handler and updating the visualized page.

- **ShowPageEndpoint** : this endpoint is responsible for updating the list of available shows that can be seen by users in the dedicated page.

2.3.5 Java Listener

In order to get updates from Erlang nodes, a separate Java program needs to be listening for those updates. This program, whose class structure is shown in figure 9, starts an Erlang node by using the **Jinterface Package** with the known name `listener@<host_address>` and known mailbox `mbox`. It then enters in a loop, in which it waits for a new message and creates a new task upon reception.

Each **ErlangMessageTask** represents one of those tasks, run by a different thread. Two type of task messages are recognized:

- `available_shows_list`, containing an updated list of Shows to be presented in the *BrowseShowsPage*;
- `update_show_state`, from a Show Handler node, containing the updated state of its managed show for its *ShowPage*.

After message parsing, a temporary WebSocket Client Endpoint is opened to send the updates to the right Server Endpoint, that will then forward them to its client.

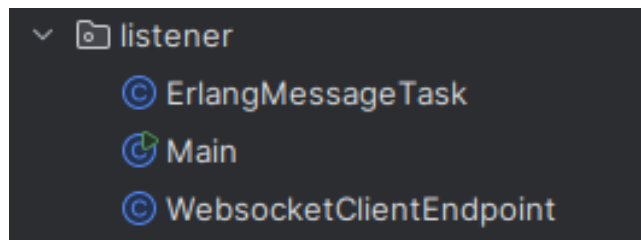


Figure 9: Java Listener

2.3.6 User Interface

The implementation of the UI interface used also the Bootstrap framework for JavaScript and CSS, as well as few other manually written files in the same languages.

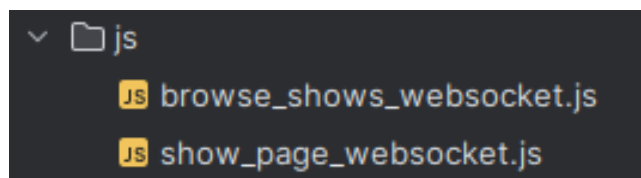


Figure 10: User Interface : JS scripts

The JavaScript scripts used are :

- **show_page_websocket.js** : This script is responsible for connecting the client to the ShowPage WebSocket Endpoint previously described at 2.3.4.

When an update is received, the show data will be updated, without reloading the page.

It also checks if a customer is inputting a valid number upon booking submission (checking against the displayed available seats).

- **browse_page_websocket.js** : This script is responsible for connecting the client to the BrowsePage WebSocket Endpoint previously described at 2.3.4.

Upon receiving a new list of Shows, this script will re-populate the page without reloading it.

3 Synchronization and Communication Approach

As introduced in section 1.2, the main issues regarding synchronization and communication are :

- Maintaining in the server a backup on-disk of the bookings by customers for shows;
- Maintaining a consistent list of pending bookings for each Bookable Show, so that all Customers and Cinemas have the same view of the show state they are looking at.
- Making sure that each user can see the same list of bookable shows.

As previously described in section 2.2.2, we use Mnesia Atomic Transactions to avoid TOCTOU, so that persistent storage has a single, logically correct, history of writes.

Regarding pending information, each show is managed by a single node handling requests one at a time, ensuring that its internal state is always consistent. To manage backups of pending information, each handler has two timers, both started at node creation:

- a *backup timer* with a short period, to send periodic backups to Main Server. This timer is restarted each time it expires;
- an *end_of_life timer* that expires when the show deadline is reached.

As far as updating viewed data for all users goes, this is achieved by using Erlang interaction with the Java WebSockets.

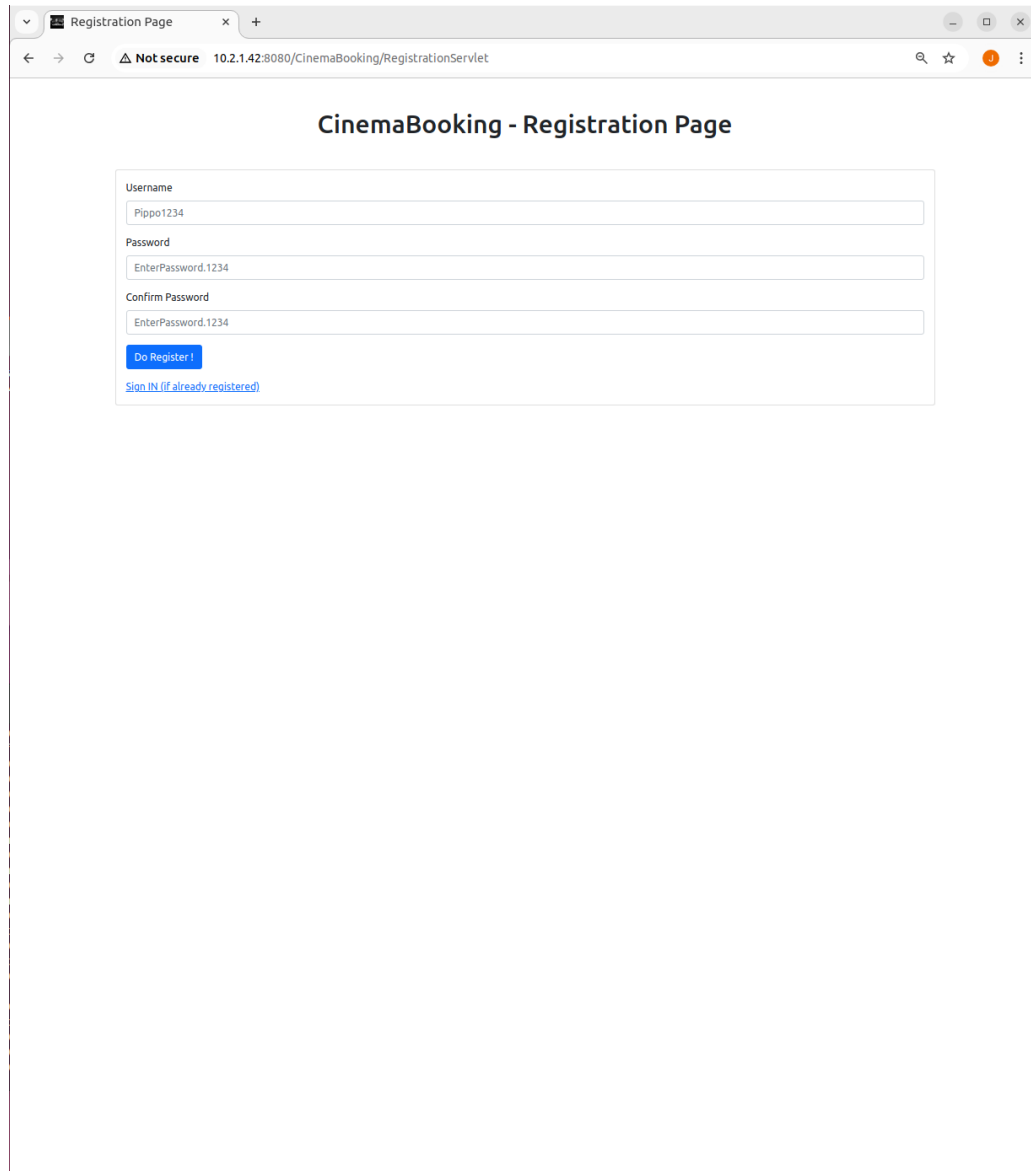
The WebSockets receive information about the up to date data that is on the Show Handler nodes and in the Main Server, and dynamically update the page that is being viewed, without needing to reload the page.

4 UserManual

4.1 Customer

4.1.1 Customer Registration and Login

An Unregistered Customer can only Register itself in the CustomerRegistrationPage , by declaring its UserName and UserPassword :



The screenshot shows a web browser window with the title "Registration Page". The address bar shows the URL "10.2.1.42:8080/CinemaBooking/RegistrationServlet". The page content is titled "CinemaBooking - Registration Page". It features a registration form with the following fields and elements:

- Username:** A text input field containing the value "Pippo1234".
- Password:** A text input field containing the value "EnterPassword.1234".
- Confirm Password:** A text input field containing the value "EnterPassword.1234".
- Do Register!** A blue button.
- Sign IN (if already registered)** A blue link.

Figure 11: User Registration Page

If the Registration is not successful (could be many reasons, most commonly because the username is already taken), an error message appears (this error is generic so as to avoid *customer-mapping* or other security attacks) If the Registration is successful, the User will be redirected to the Login Page, where the same inputs will be required :

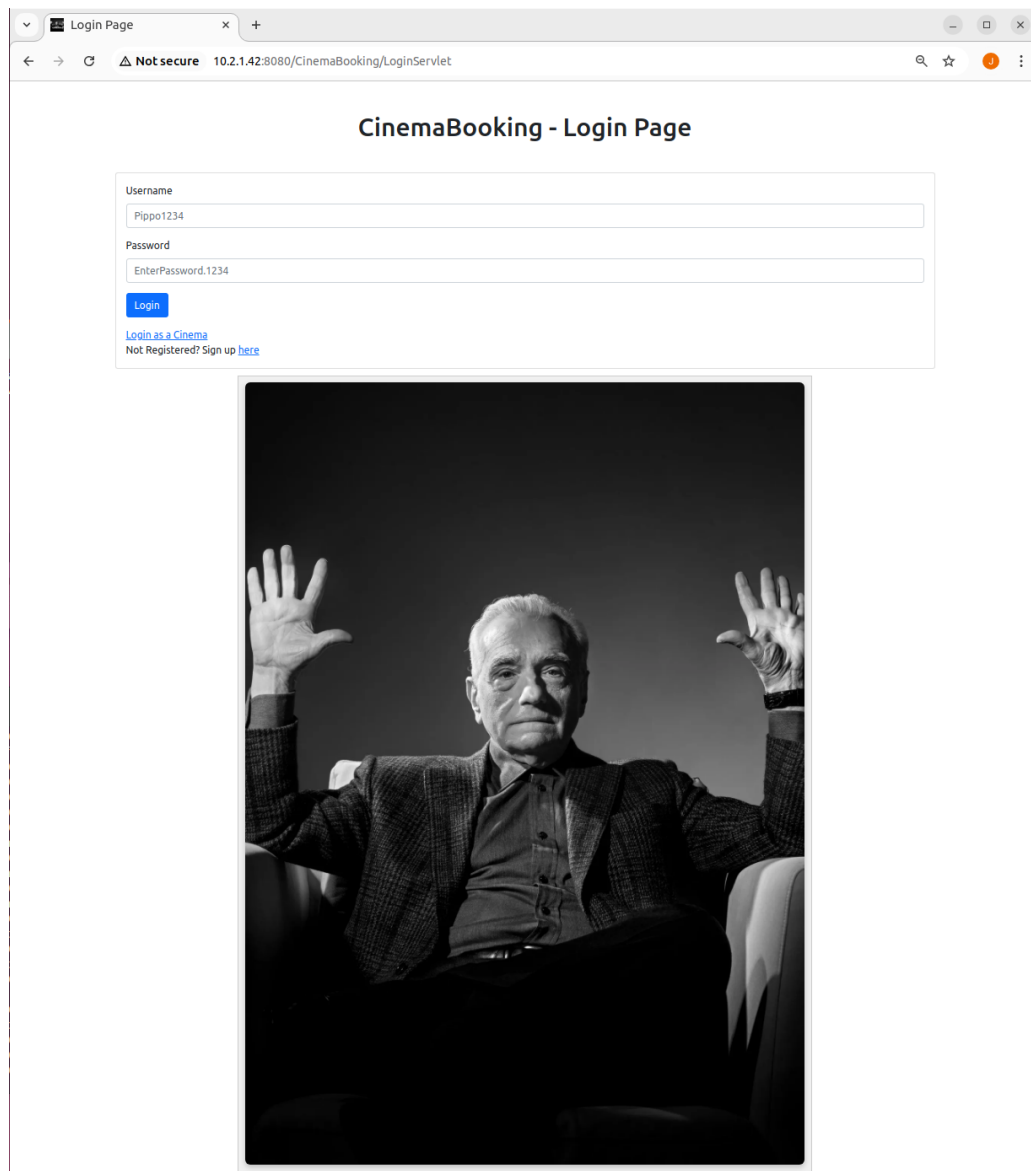


Figure 12: Customer Login Page

If the Login is not successful (e.g. could be due to non existant username, or non matching userName-Password, ...), then a generic error message is shown (again, for security reasons). If the Login is successful, then the customer is sent to their own personal page.

4.1.2 Customer Page

If the customer requiring a UserPage is the one owning it, then they can see their personal data (not the pwd), and the list of active bookings they own, with the booked-seats-numbers that have been backed-up in the main server (for each booked show) :

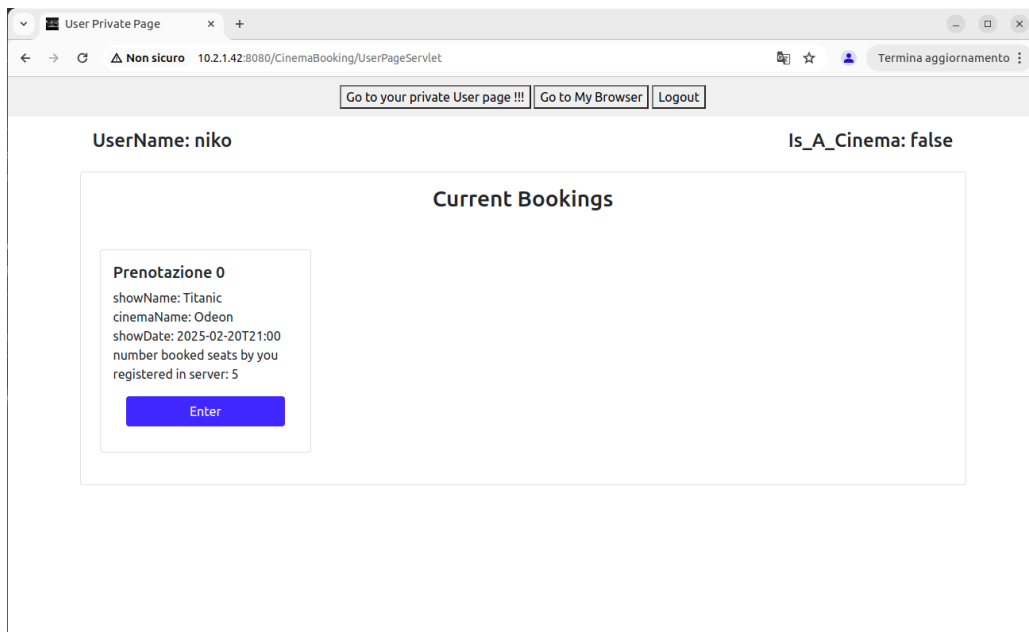


Figure 13: Customer Personal Page

By pressing on a show, they will be redirected to the specific ShowPage.

Otherwise, by pressing the header they can visit a page where they can Browse for other shows.

4.1.3 Show Page seen by Customers

When a customer reaches a ShowPage, it can see the ShowName, ShowDate, the Name and Address of the Cinema it is held at, the *Maximum Number* of seats for that Show, the *Backed-Up Number* of available seats, and the *Updated Number* of available seats :

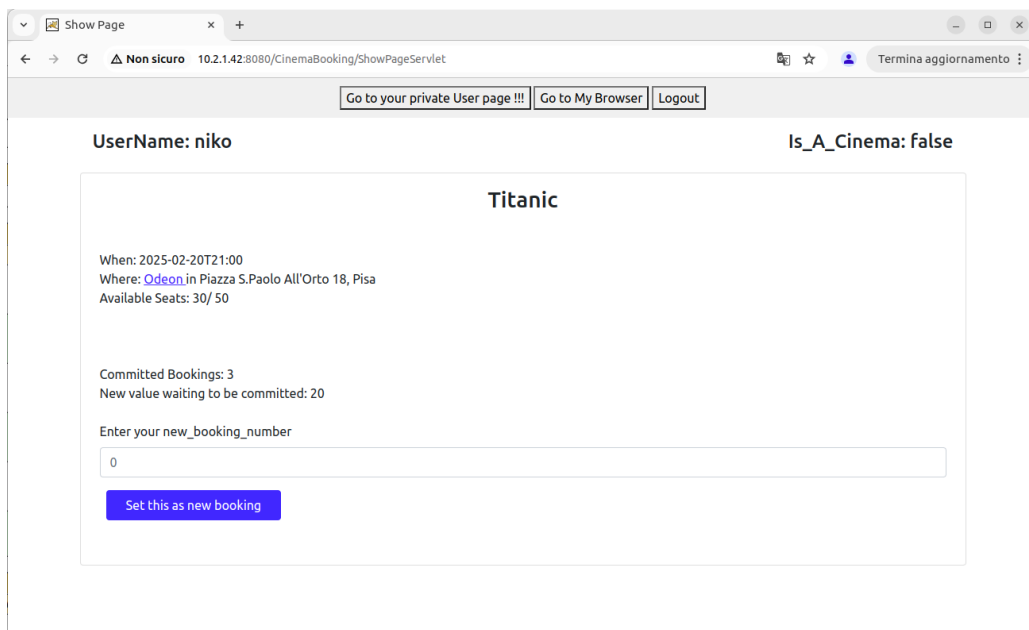


Figure 14: Show Page (seen by a customer that has already some bookings for this show)

If the Customer has a booking with the Show, then they will also see both the Backed-Up Number of seats, and the "Waiting-For-Update" Number of seats booked. Customers should remember that **we backup bookings every 15 minutes**, so there might be mis-matches between the two numbers !!!

In any case, Customers will also have the option to **create** a new Booking, which also is the way to **delete** a booking, or to **replace** a previous one.

From the ShowPage, a Customer can also navigato to the relative CinemaPage.

4.1.4 Cinema Page seen by Customer

The CinemaPage shows the list of Shows of the Cinema :

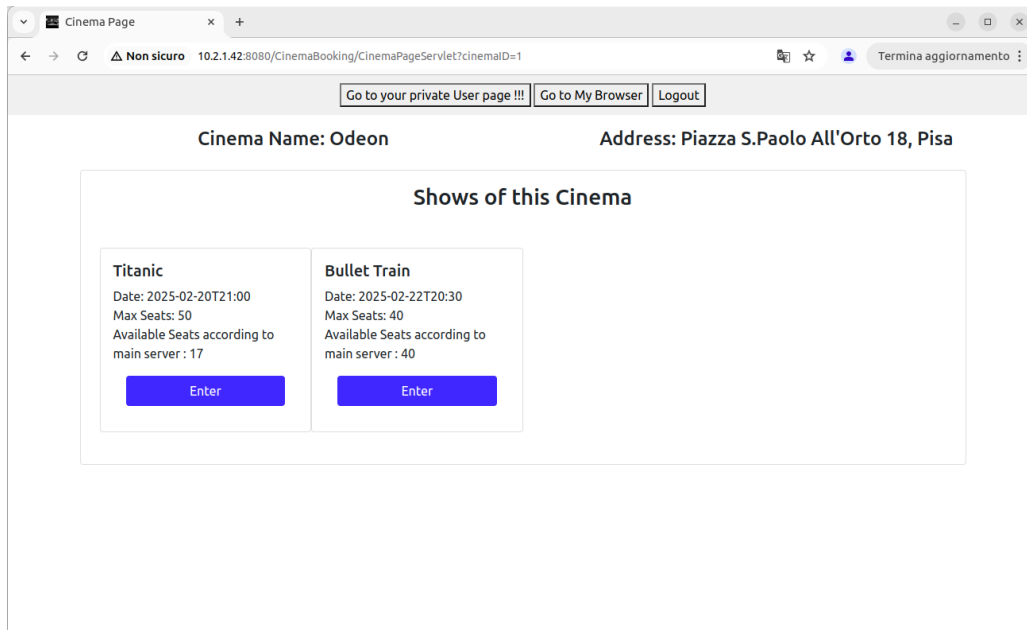


Figure 15: CinemaPage seen by Customers

By clicking on a show in the list, one is redirected to that specific ShowPage.

4.1.5 Browse Shows Page

Customers have the option to browse existing Shows, even filtering them by name keywords :

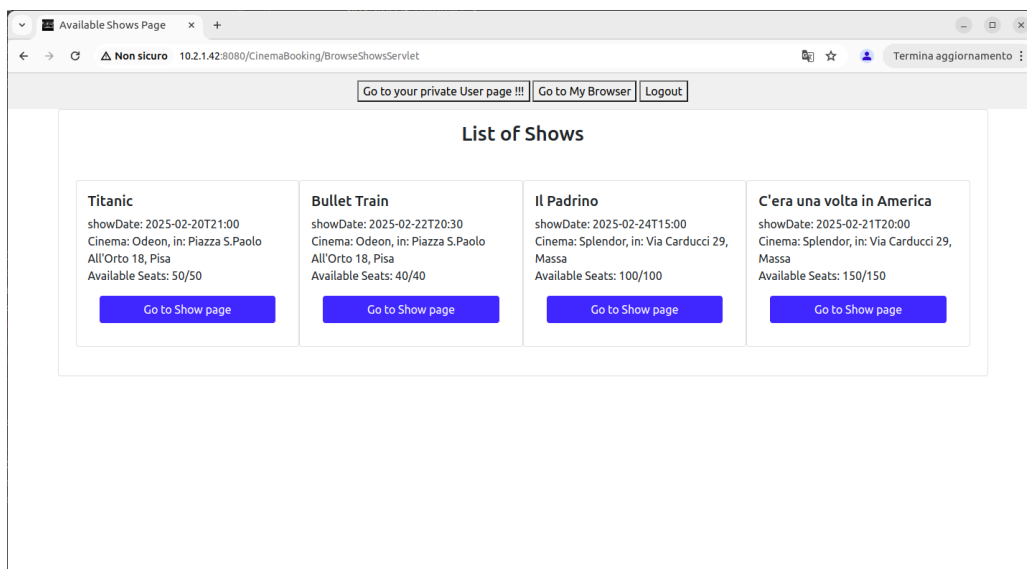


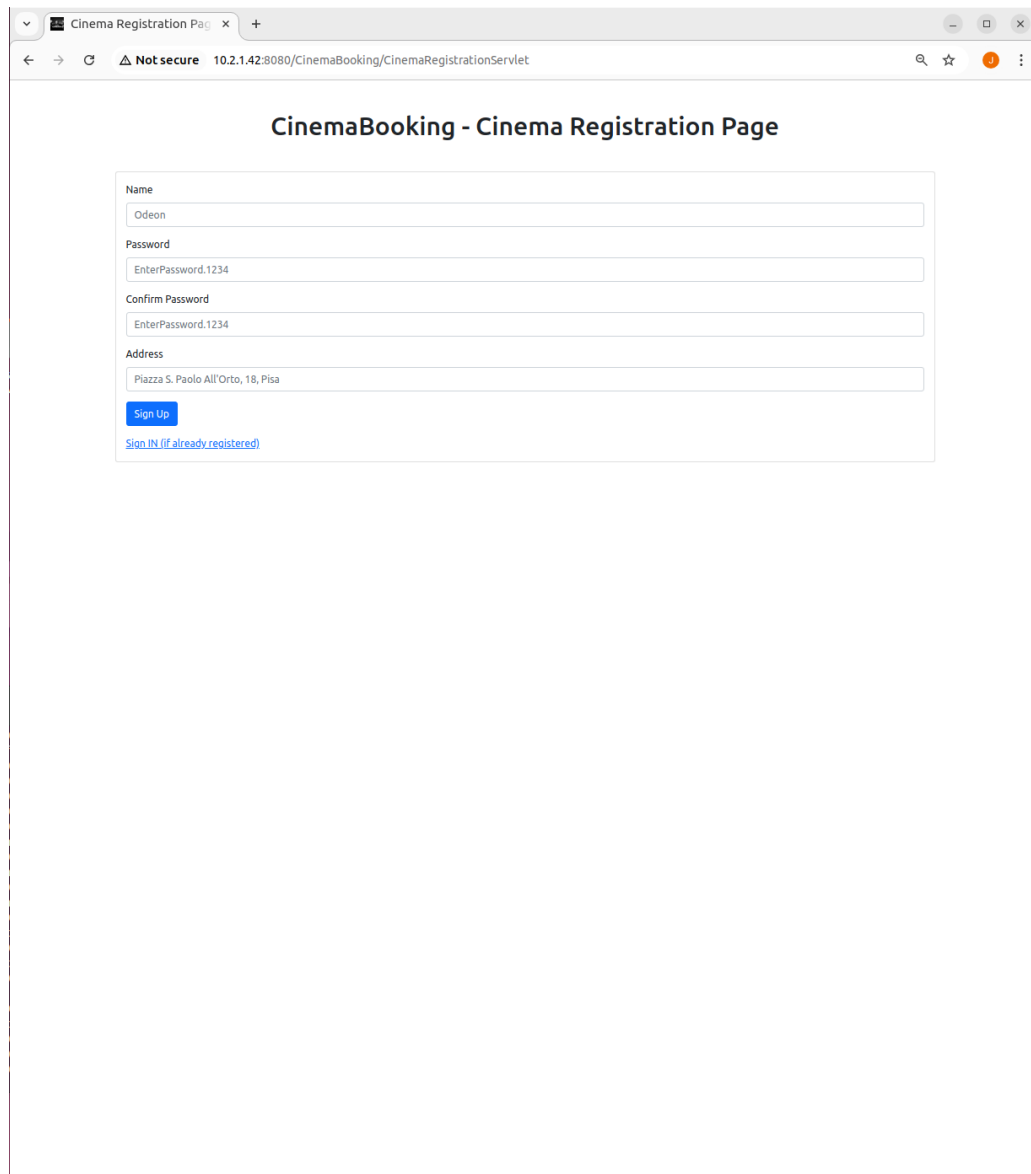
Figure 16: Show Page (seen by a customer that has already some bookings for this show)

By pressing on the Show, they will be redirected to that Show Page.

4.2 Cinema

4.2.1 Cinema Registration and Login

An Unregistered Cinema can only Register itself in the CinemaRegistrationPage , by declaring its CinemaName, CinemaPassword, and CinemaAddress :



The screenshot shows a web browser window with the address bar displaying "10.2.1.42:8080/CinemaBooking/CinemaRegistrationServlet". The page title is "CinemaBooking - Cinema Registration Page". The form contains the following fields and values:

- Name: Odeon
- Password: EnterPassword.1234
- Confirm Password: EnterPassword.1234
- Address: Piazza S. Paolo All'Orto, 18, Pisa

Below the form, there is a blue "Sign Up" button and a link "Sign IN (if already registered)".

Figure 17: Cinema Registration Page

After a Cinema Registration, the Cinema will be redirected directly to its own (newly created) CinemaPage (where it will be able to learn the cinemaID that has been assigned to it).

After a log-out, the Cinema will be able to log-in again by using the CinemaID and CinemaPassword in the CinemaLoginPage :

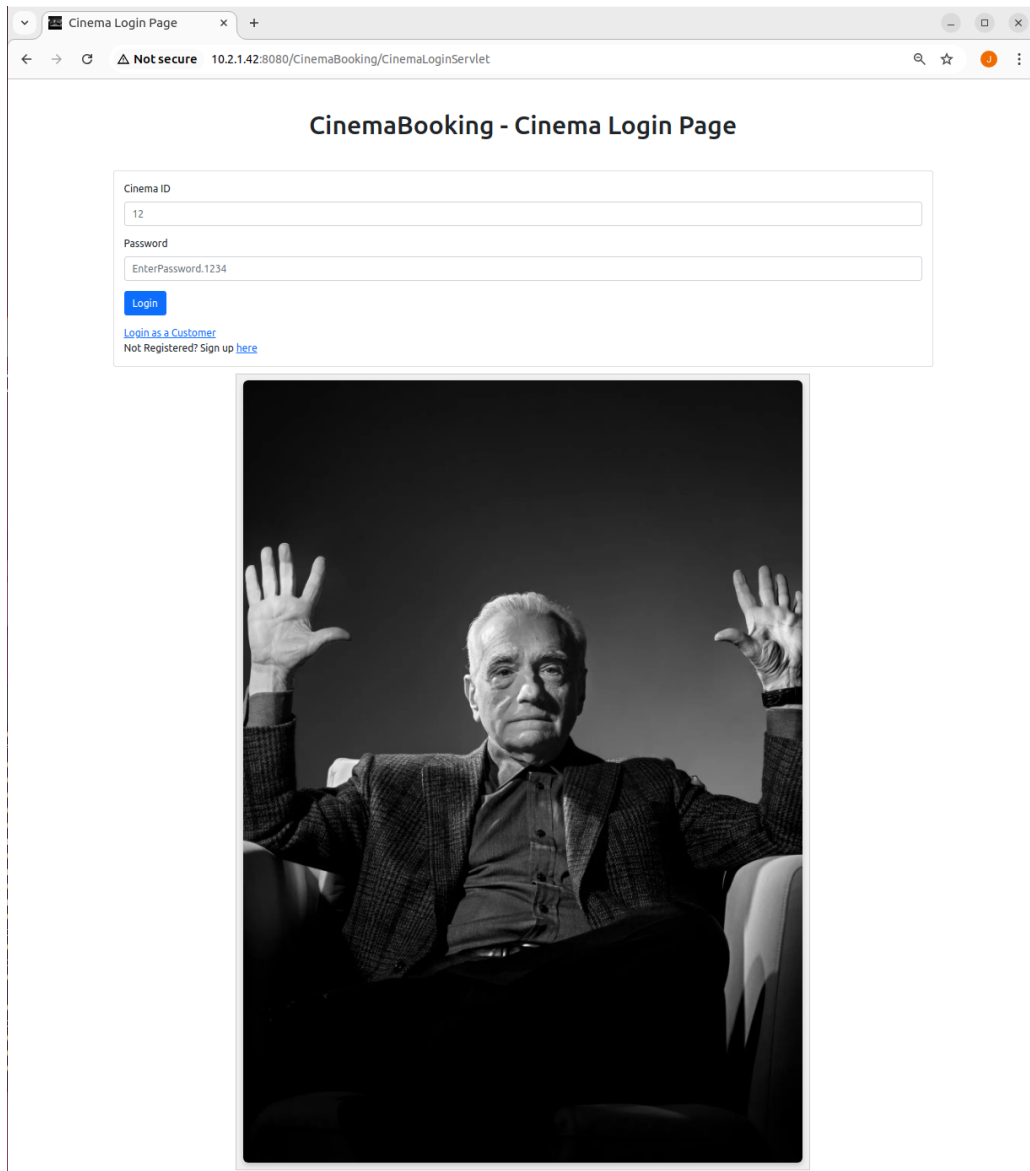


Figure 18: Cinema Login Page

Upon login, the Cinema is sent to its own CinemaPage.

4.2.2 Cinema Page seen by Cinema

If a Cinema sees their **own Cinema Page**, then a new option appears : "CreateShow", which sends to the CreateShowPage (unlike what happens when other visit this page as per previous 4.1.4 :

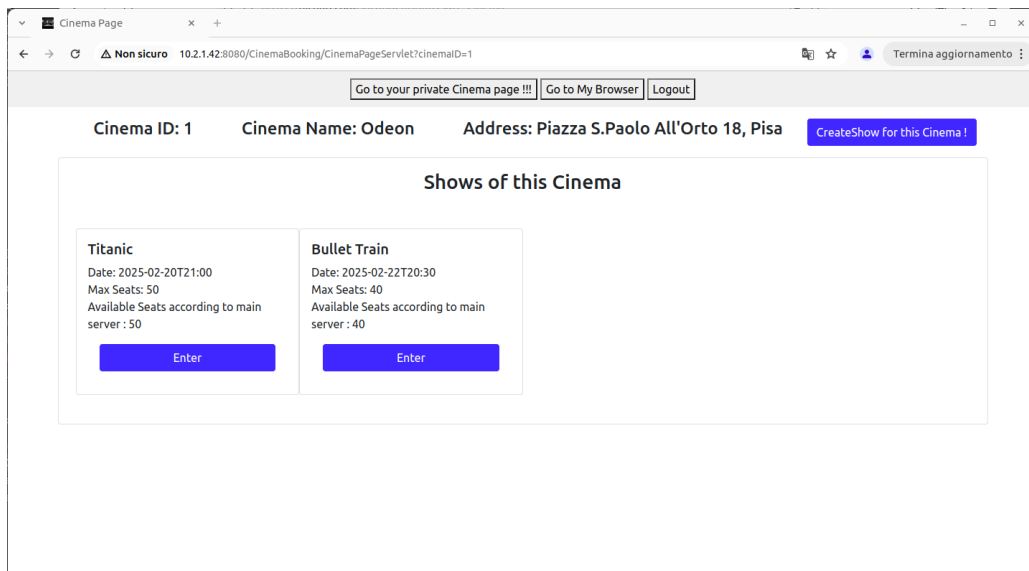


Figure 19: CinemaPage seen by Cinema itself

4.2.3 Show Page seen by its own Cinema

When a Cinema reaches a ShowPage of one of their **own Shows**, in addition to what anybody else can see (previously described at 4.1.3, the Cinema also sees a table showing all the bookings of this Show.

The table has a column for the UserName, a column for the Server-BackedUp booking seats number, and a column for the ShowHandler (Waiting for Backup) booking seats number :

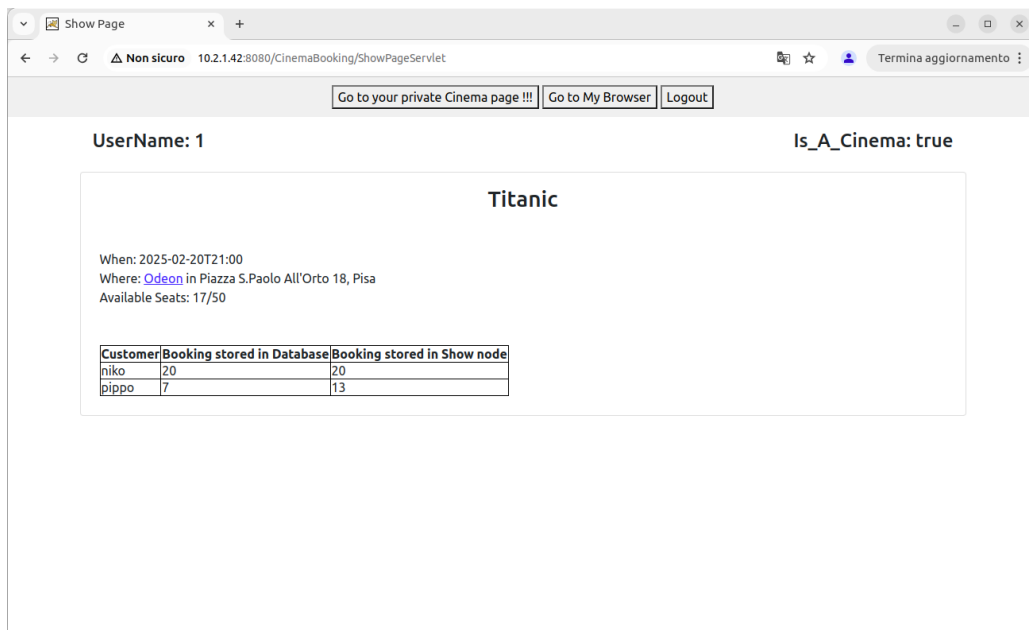


Figure 20: Show Page seen by its own Cinema

4.2.4 Create Show Page

In this Page, a Cinema can create a new Show, by setting ShowName, ShowDate, and MaxNumberOfSeats :

Create Show

Non sicuro 10.2.1.42:8080/CinemaBooking/CreateShowServlet

Go to your private Cinema page !!! Go to My Browser Logout

Back

Create a new Show !!!

Show Name

Enter showName

Show Date and Time: 11/11/2025, 11:11

Maximum Seats

Create Show !!!

Figure 21: CreateShowPage seen by a Cinema

5 Bibliography

References

- [1] *Apache Tomcat Web Server*. URL: <https://tomcat.apache.org/>.
- [2] *Erlang*. URL: <https://www.erlang.org/>.
- [3] *Java Filter Interface*. URL: <https://docs.oracle.com/javaee/7/api/javax/servlet/Filter.html>.
- [4] *Java Web Socket Endpoint*. URL: <https://docs.oracle.com/javaee/7/api/javax/websocket/Endpoint.html>.
- [5] *Jinterface*. URL: https://www.erlang.org/doc/apps/jinterface/jinterface_users_guide.html.
- [6] *Mnesia Transaction*. URL: https://www.erlang.org/doc/apps/mnesia/mnesia_chap4.html.
- [7] *MnesiaDB*. URL: <https://www.erlang.org/doc/apps/mnesia/mnesia.html>.
- [8] *MnesiaDB : Create Table*. URL: https://www.erlang.org/doc/apps/mnesia/mnesia.html#create_table/2.