

Web Technologies Project @ PoliMi, 2025

Creating a Playlist Manager with Thymeleaf & TS

VLAD RAILEANU

`raileanu.vlad29@gmail.com`

<https://github.com/rokuban>

VITTORIO ROBECCI

`vittorio.robecchi@gmail.com`

<https://github.com/VictuarVi>

Contents

1 Original submission (in Italian) 4

1.1 Versione HTML pura 5

1.2 Versione con JavaScript 5

2 Project submission breakdown 8

2.1 Database logic 9

2.2 Behaviour 9

2.3 JavaScript version 11

3 Specifications completion 14

3.1 In all subprojects 15

3.2 HTML-specific features 15

3.3 RIA-specific features 15

4 SQL database schema 16

4.1 Overview 17

4.2 The tables 17

5 Codebase overview 20

5.1 Components 21

5.2 DAOs methods 21

5.3 RIA subproject 21

6 Sequence diagrams 26

6.1 Login sequence diagram 27

6.2 Register sequence diagram 29

6.3 HomePage sequence diagram 31

6.4 Playlist sequence diagram 33


6.5 Track sequence diagram 35


6.6 UploadTrack sequence diagram . . . 36


6.7 CreatePlaylist sequence diagram . . 38


6.8 Logout sequence diagram 39


6.9 AddTracks sequence diagram 39


6.10  GetTracksNotInPlaylist sequence diagram 40


6.11  TrackReorder sequence diagram . 41


6.12  GetUserTracks sequence diagram 42


6.13  Event: Login 43

6.14  Event: Register 44

6.15  Event: Logout 45

6.16  Event: Access HomeView 46

6.17  Event: Access PlaylistView 47

6.18  Event: Access TrackView 48

7 Filter mappings 50

7.1 UserChecker filter 51

7.2 InvalidUserChecker filter 51

7.3 PlaylistChecker filter 52

7.4 SelectedTracksChecker filter 53

7.5 TrackChecker filter 54


8 Cascading Style Sheets (CSS) styling 56

8.1 Introduction 57

8.2 Buttons 57

8.3 Containers 57

8.4 Modal 59

8.5  Sidebar 60

A Cut content 62

A.1 Features 63

A.2 Optimizations 63

A.3 Code cleanup 64

Bibliography 66

Abstract

Overview This project hosts the source code – which can be found [on Github](#) – for a web server that handles a playlist management system. A user is able to register, login and then upload tracks. The tracks are strictly associated to one user, similar to how a cloud service works. The user will be able to create playlists, sourcing from their tracks, and listen to them.

It should be noted there are two versions: a **HTML version**, which is structured as a series of separate webpages; and a **RIA version** ([ts](#))¹, which is structured as a single-page webapp. The functionalities are quite the same, the code changes mostly at a frontend level. For more information about the requirements for each version see [Section 2](#).

Both of the them feature the same CSS code (see [Section 8](#)).

Tools To create the project, our professor decided to adopt the following technologies: **Java**, for the backend server with servlets leveraging Jakarta's API capabilities; **Apache Tomcat**, to run the server; for the HTML version, **Thymeleaf**, a template engine; whereas for the RIA one **Javascript**.

Many liberties were taken in regards to the DMBS and RIA:

- We decided to use **MariaDB** instead of MySQL, since the former is a open source fork of MySQL, one of the most widely used DBMS
- Instead of Javascript, we opted for **Typescript**

Last but absolutely not least, this very document you are reading now has been typeset with none-other than **Typst** [\[1\]](#), the much needed successor to $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$. Also, to create sequence diagrams we made use of the **chronos** package [\[2\]](#).

Configuration & Run In order to run this project, the following packages and their respective versions are to be installed:

- Java JDK 24 [\[3\]](#)
- Apache Tomcat 10 [\[5\]](#)
- MariaDB [\[7\]](#)
- Apache Maven [\[4\]](#)
- Thymeleaf [\[6\]](#)

Then Maven will fetch all the corrected dependencies (such as the JDBC driver). We opted to use IntelliJ Idea Ultimate Edition [\[8\]](#) though there are no restrictions on the IDE – feel free to use whatever editor you want, even Eclipse, *if you must*². Once you made sure all the dependencies are correctly installed, run the desired Tomcat configuration and let it deploy the server, which will be found at³:

[http://localhost:8080/\[version\]_war_exploded](http://localhost:8080/[version]_war_exploded)

The credentials are stored in plain text in the database (see [Section 6.2](#), [Section A.2](#)), while the tracks and images are stored in target/webapp (see [Section 6.6](#)).

The repository is bundled with some mock data, which can be found at the root of the project in the mockdata folder. They are copyright free songs [\[9\]](#) because we didn't want to get sued 🙄.

¹For historic reasons, in the project is is referred as just js.

²I wrote that only out of kindness, since I wouldn't recommend it even to my worst enemy. — victuarvi.

³[version] is either pure_html or js depending on what you run.

1

**Original submission
(in Italian)**

1.1 Versione HTML pura

Un'applicazione web consente la gestione di una playlist di brani musicali. Playlist e brani sono personali di ogni utente e non condivisi. Ogni utente ha username, password, nome e cognome. Ogni brano musicale è memorizzato nella base di dati mediante un titolo, l'immagine e il titolo dell'album da cui il brano è tratto, il nome dell'interprete (singolo o gruppo) dell'album, l'anno di pubblicazione dell'album, il genere musicale (si supponga che i generi siano prefissati) e il file musicale. Non è richiesto di memorizzare l'ordine con cui i brani compaiono nell'album a cui appartengono. Si ipotizzi che un brano possa appartenere a un solo album (no compilation). L'utente, previo login, può creare brani mediante il caricamento dei dati relativi e raggrupparli in playlist. Una playlist è un insieme di brani scelti tra quelli caricati dallo stesso utente. Lo stesso brano può essere inserito in più playlist. Una playlist ha un titolo e una data di creazione ed è associata al suo creatore. A seguito del login, l'utente accede all'HOME PAGE che presenta l'elenco delle proprie playlist, ordinate per data di creazione decrescente, un form per caricare un brano con tutti i dati relativi e un form per creare una nuova playlist. Il form per la creazione di una nuova playlist mostra l'elenco dei brani dell'utente ordinati per ordine alfabetico crescente dell'autore o gruppo e per data crescente di pubblicazione dell'album a cui il brano appartiene. Tramite il form è possibile selezionare uno o più brani da includere. Quando l'utente clicca su una playlist nell'HOME PAGE, appare la pagina PLAYLIST PAGE che contiene inizialmente una tabella di una riga e cinque colonne. Ogni cella contiene il titolo di un brano e l'immagine dell'album da cui proviene. I brani sono ordinati da sinistra a destra per ordine alfabetico crescente dell'autore o gruppo e per data crescente di pubblicazione dell'album a cui il brano appartiene. Se la playlist contiene più di cinque brani, sono disponibili comandi per vedere il precedente e successivo gruppo di brani. Se la pagina PLAYLIST mostra il primo gruppo e ne esistono altri successivi nell'ordinamento, compare

a destra della riga il bottone SUCCESSIVI, che permette di vedere il gruppo successivo. Se la pagina PLAYLIST mostra l'ultimo gruppo e ne esistono altri precedenti nell'ordinamento, compare a sinistra della riga il bottone PRECEDENTI, che permette di vedere i cinque brani precedenti. Se la pagina PLAYLIST mostra un blocco e esistono sia precedenti sia successivi, compare a destra della riga il bottone SUCCESSIVI e a sinistra il bottone PRECEDENTI. La pagina PLAYLIST contiene anche un form che consente di selezionare e aggiungere uno o più brani alla playlist corrente, se non già presente nella playlist. Tale form presenta i brani da scegliere nello stesso modo del form usato per creare una playlist. A seguito dell'aggiunta di un brano alla playlist corrente, l'applicazione visualizza nuovamente la pagina a partire dal primo blocco della playlist. Quando l'utente seleziona il titolo di un brano, la pagina PLAYER mostra tutti i dati del brano scelto e il player audio per la riproduzione del brano.

1.2 Versione con JavaScript

Si realizzi un'applicazione client server web che modifica le specifiche precedenti come segue:

- Dopo il login dell'utente, l'intera applicazione è realizzata con un'unica pagina.
- Ogni interazione dell'utente è gestita senza ricaricare completamente la pagina, ma produce l'invocazione asincrona del server e l'eventuale modifica del contenuto da aggiornare a seguito dell'evento.
- L'evento di visualizzazione del blocco precedente/successivo è gestito a lato client senza generare una richiesta al server.
- L'applicazione deve consentire all'utente di riordinare le playlist con un criterio personalizzato diverso da quello di default. Dalla HOME con un link associato a ogni playlist si accede a una finestra modale RIORDINO, che mostra la lista completa dei brani della playlist ordinati secondo il criterio corrente (personalizzato o di default). L'utente può trascinare il titolo di un brano nell'elenco

e collocarlo in una posizione diversa per realizzare l'ordinamento che desidera, senza invocare il server. Quando l'utente ha raggiunto l'ordinamento desiderato, usa un bottone "salva ordinamento", per memorizzare la sequenza sul server. Ai successivi accessi, l'ordinamento personalizzato è usato al posto di quello di default. Un brano aggiunto a una playlist con ordinamento personalizzato è inserito nell'ultima posizione.

2

**Project submission
breakdown**

2.1 Database logic

LEGEND	Entity	Attribute
	Attribute specification	Relationship

Each **user** has a **username**, **password**, **name** and **surname**. Each musical **track** is stored in the database by **title**, **image**, **album title**, **album artist name** (single or group), **album release year**, **musical genre** and **file**. Furthermore:

- Suppose the *genres are predetermined* // the user cannot create new genres
- It is not requested to store the track order within albums
- Suppose each track can belong to a unique album (no compilations)

After the login, the user is able to **create tracks** by loading their data and then group them in playlists. A **playlist is a set of chosen tracks** from the uploaded ones of the user. A playlist has a **title**, a **creation date** and is **associated to its creator**.

For the UML diagram, see [Section 4](#).

2.2 Behaviour

LEGEND	User action	Server action
	HTML page	Page element

After the login, the user **accesses** the **HOME PAGE** which **displays** the **list of their playlists**, ordered by descending creation date; a **form to load a track with relative data** and a **form to create a new playlist**. The playlist form:

- **Shows** the **list of user tracks** ordered by artist name in ascending alphabetic order and by ascending album release date
- The form allows to **select** one or more tracks

When a user **clicks** on a playlist in the **HOME PAGE**, the application **loads** the **PLAYLIST PAGE**; initially, it contains a **table with a row and five columns**.

- Every cell contains the track's title and album name
- The tracks are ordered from left to right by artist name in ascending alphabetic order and by ascending album release date
- If a playlist contains more than 5 tracks, there are available commands to see the others (in blocks of five)

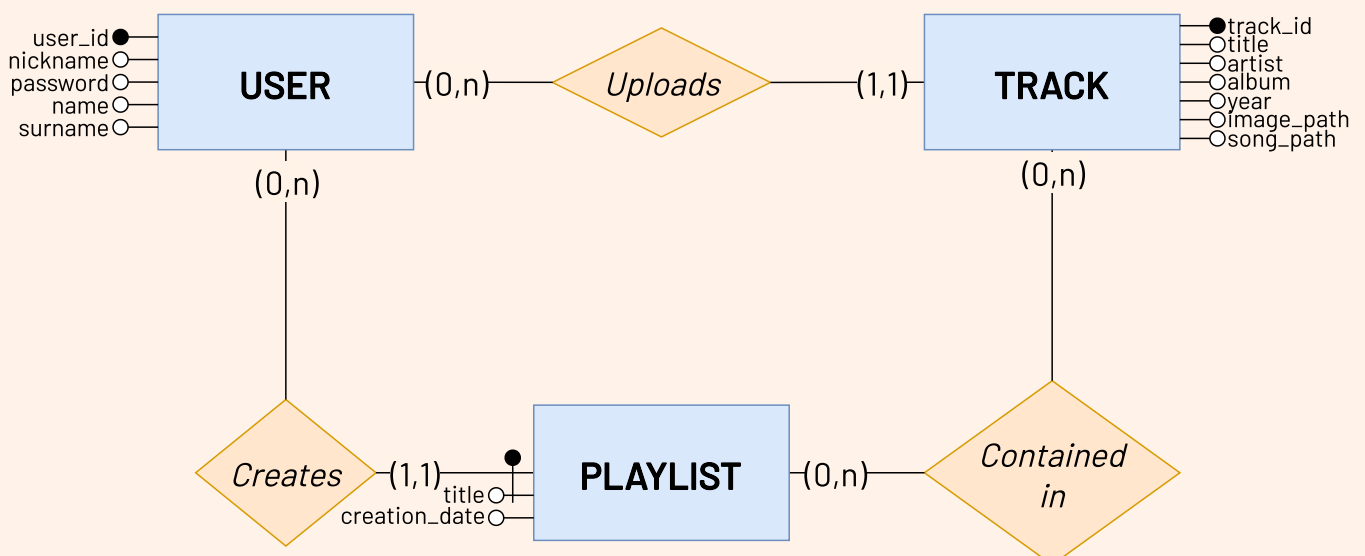


Figure 1: ER diagram (HTML).

2.3 JavaScript version

Create a client-server web application that modifies the previous specification as follows:

- After the **LOGIN**, the entire application is built as a single webapp // RIA
- Every user interaction is managed without completely refreshing the page, but instead it asynchronously invokes the server and the content displayed is potentially updated
- The visualization event of the previous/next blocks is managed client-side without making a request to the server

Track reordering The application must allow the user to reorder the tracks in a playlist with a personalized order, different from the default one. From the **HOME PAGE** with an associated link to each playlist, the user is able to **access** a modal window **REORDER** which shows the full list of tracks ordered with the current criteria (custom or default).

The user can **drag** the title of a track and **drop** it in a different position to achieve the desired order, without invoking the server. Once finished, the user can click on a **button to save the order** and **store** the sequence on the server. In subsequent accesses, the personalized track order is **loaded** instead of the default one. A newly added track in a custom-ordered playlist is **inserted always at the end**.

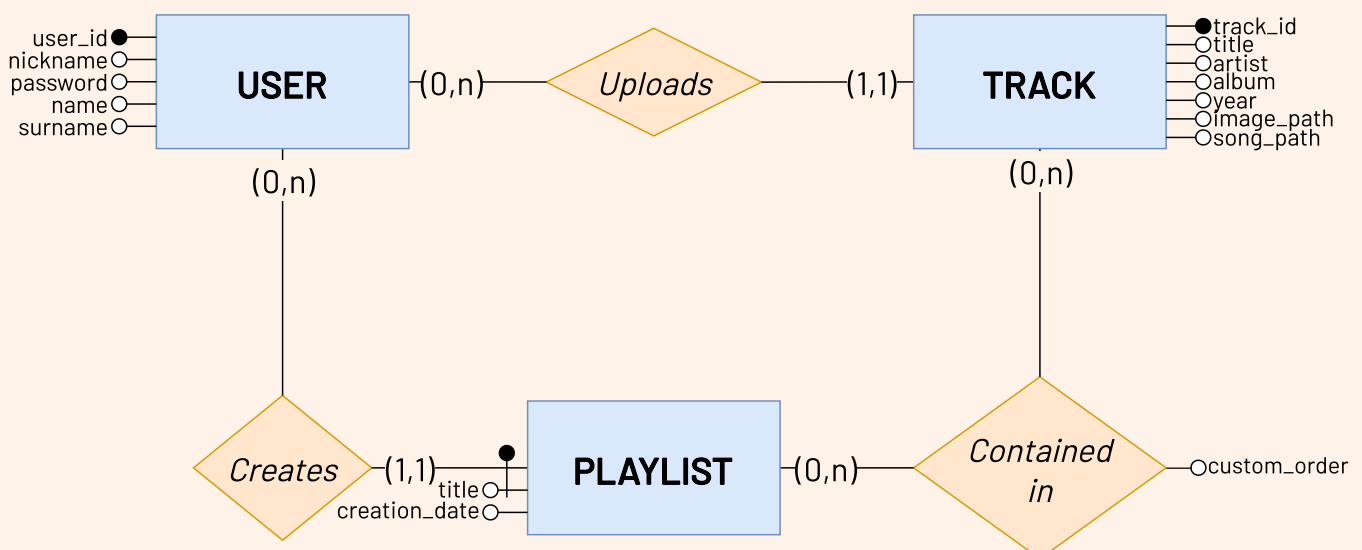


Figure 3: ER diagram (RIA).

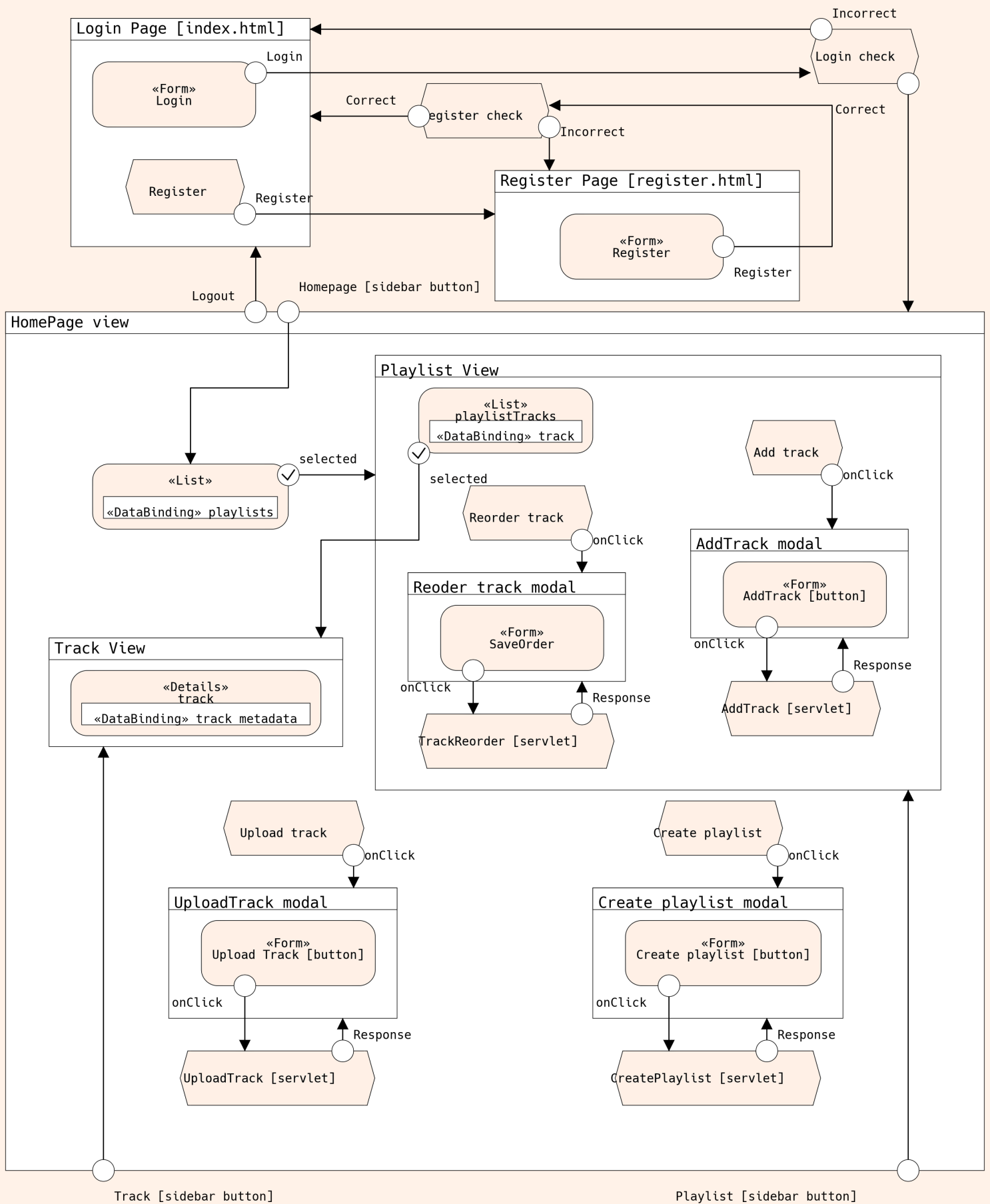


Figure 4: IFML diagram (RIA).

3

**Specifications
completion**

Introduction There were many features planned, but due to time constraints they weren't implemented. They are all listed in [Section A](#).

3.1 In all subprojects

In addition to the requirements, we implemented a series of new features:

- Since the application does need to implement a login function, we thought to complement this with a logout button ([Section 6.8](#))
 - The motivation is the same with the registration button ([Section 6.2](#))
- All the CSS styling ([Section 8](#)), from colours to buttons to modals
- The top navigation bar, where the title of the page and various buttons are located (upload or add track, create playlist and logout)


3.2 HTML-specific features

The following features are HTML specific, which means they do not appear in the RIA subproject:

- The bottom navigation bar, where the Home button is located

3.3 RIA-specific features

This project started from the HTML, where we developed most of the servlets, the ideas and the HTML webpages. Then, we moved on the next task: the RIA, that is the *Rich Internet Application*.

For this very reason, most of the features were ported to the latter subproject: there is no need to go through the components and logic all over again. Still, the RIA subproject – the “JavaScript version” – requires more features and some changes to how the overall server works. The  symbol means that the currently described features is present *only* in the RIA subproject, NOT in the HTML one.

Furthermore, by using Javascript, some features can be upgraded – this mainly applies to the modal [Section 8.4](#). Briefly, the main changes are as follows:

- Thymeleaf has been completely removed, since everything it did can be done via Javascript
- All the project now runs in a single webapp after the user has logged in⁴
- To account for the new User Experience, we developed the sidebar to host the various buttons ([Section 8.5](#))

Typescript Instead of Javascript we opted for **Typescript**. This was done mainly for two reasons: the retrocompability with Javascript (since Typescript transpiles in JS) and the static typing system, which can be quite bothersome in some cases, but saves a lot of time overall.

⁴This effectively means that before the homepage, the playlist and the player were *all* separate webpages; now, they are all-in-one – hence why *Rich Internet Application*.

4

SQL database schema

4.1 Overview

The project requirements slightly change from pure_html and js, where the latter requires the tracks to support an individual custom order within the playlist to which they are associated – this is achieved via a simple addition in the SQL tables schema.

In both scenarios, the schema is composed by four tables: user, track, playlist and playlist_tracks.

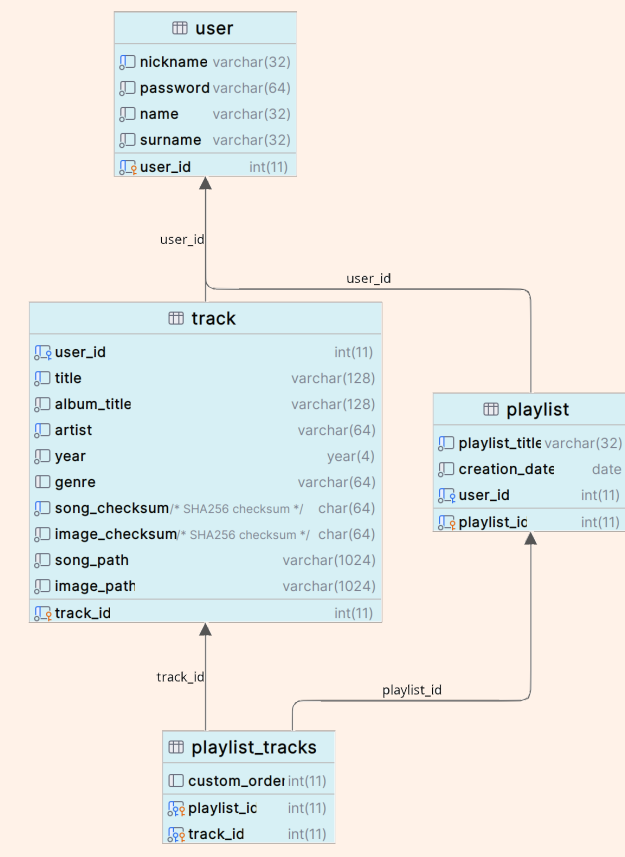


Figure 5: UML diagram.

4.2 The tables

- user table

```

CREATE TABLE user
(
    user_id integer not null
auto_increment,
    nickname varchar(32) not null unique,
    password varchar(64) not null,

```

```

    name varchar(32) not null,
    surname varchar(32) not null,

    primary key (user_id)
);

```

it is quite straightforward and standard. Apart from the user_id attribute, which is the primary key, the only other attribute that has a unique constraint is nickname. It couldn't be a multiple primary key because in that case there could have been multiple users with the same nickname, which isn't our goal.

- track table

```

CREATE TABLE track
(
    track_id integer not
null auto_increment,
    user_id integer not null,
    title varchar(128) not null,
    album_title varchar(128) not null,
    artist varchar(64) not null,
    year year not null,
    genre varchar(64),
    song_checksum char(64) not null
default '0...0',
    image_checksum char(64) not null
default '0...0',
    song_path varchar(1024) not null,
    image_path varchar(1024) not null,

    primary key (track_id),
    foreign key (user_id) REFERENCES user
(user_id)
ON DELETE CASCADE ON UPDATE CASCADE,
    unique (user_id, song_checksum),
    unique (user_id, title, artist),
    check (genre in ('Classical', 'Rock',
'Edm', 'Pop', 'Hip-hop', 'R&B', 'Country',
'Jazz', 'Blues', 'Metal', 'Folk', 'Soul',
'Funk', 'Electronic', 'Indie', 'Reggae',
'Disco'))
);

```

this needs to be addressed since we implemented a special feature, which is the checksum for the song and the album image. As their name implies, they are the SHA256 checksums of the song and image: their purpose is to let the server store only one copy of the same file, which couldn't have been properly achieved by checking only the filename.

Next, the other attributes are pretty standard. As per the user table, there are some unique constraint placed on `user_id`, `song_checksum` to account for what is written above; while `user_id`, `title`, `artist` does the same job though internally in the database⁵. Finally, a track is strictly bound to a user: that's the the foreign key is for.

- `playlist` table

```
CREATE TABLE playlist
(
    playlist_id    integer    not null
auto_increment,
    playlist_title varchar(32) not null,
    creation_date  date       not null
default CURRENT_DATE,
    user_id        integer    not null,

    primary key (playlist_id),
    unique (playlist_title, user_id),
    foreign key (user_id) REFERENCES user
(user_id)
    ON DELETE CASCADE ON UPDATE CASCADE
);
```

once again this table is rather. The `creation_date` attribute default to the current date, which is today; and again there is the unique constraint on `playlist_title`, `user_id` because a playlist is bound to a single User (who can't have duplicate playlists – that is with the same title) via the foreign key.

- `playlist_tracks` table

```
CREATE TABLE playlist_tracks
(
    playlist_id integer not null,
    track_id    integer not null,

    primary key (playlist_id, track_id),
    foreign key (playlist_id) REFERENCES
playlist (playlist_id)
    ON DELETE CASCADE ON UPDATE CASCADE,
    foreign key (track_id) REFERENCES track
(track_id)
    ON DELETE CASCADE ON UPDATE CASCADE
);
```

this table represents the “Contained in” relation in the ER diagram (Figure 1). Its primary key is multiple (the only one in the project) and has to link a

track to a playlist – unlike the other tables, which *explicitly needed* a primary key *and* a unique constraints, in this case a composite key it's correct because a track can appear in multiple playlists.

-  `playlist_tracks` table in RIA project

```
CREATE TABLE playlist_tracks
(
    playlist_track_id integer
auto_increment,
    playlist_id        integer not null,
    track_id           integer not null,
    custom_order       integer,

    primary key (playlist_track_id),
    unique (playlist_id, track_id),
    foreign key (playlist_id) references
playlist (playlist_id)
    ON DELETE CASCADE ON UPDATE CASCADE,
    foreign key (track_id) references track
(track_id)
    ON DELETE CASCADE ON UPDATE CASCADE
);
```

similarly to the previous code, this one too is the translation of the “Contained in” relation in the RIA ER diagram (Figure 3), with the added `custom_order` attribute. The previous primary key has been converted to unique – similarly to other parts in the projects – and the rest is unvaried.

⁵A User can't have duplicate track.

5

Codebase overview

5.1 Components




The projects is composed of the following components:

1. DAOs
 - PlaylistDAO
 - TrackDAO
 - UserDAO
 - DAO interface

The DAO interface has only the default method `close()`, which is used in nearly all DAOs – this way we are able to follow the DRY principle (*Don't Repeat Yourself*).

2. Entities
 - Playlist
 - Track
 - User

Unlike most WT projects, these are record classes [10]: basically they are built-in old-school beans. We opted their use to drastically reduce boilerplate and simplify the codebase.


3. Servlets
 - Login
 - HomePage
 - Playlist
 - Register
 - Track
 - Logout
 - AddTracks
 - UploadTrack
 -  GetTracksNotInPlaylist
 -  TrackReorder
 -  GetUserTracks

4. Filters
 - UserChecker
 - InvalidUserChecker
 - TrackChecker
 - SelectedTracksChecker
 - PlaylistChecker
5. Utils (short-term for Utilities)
 - ConnectionHandler
 - TemplateEngineHandler

As per the DAO interface, the same idea has been applied to the `ConnectionHandler` and `TemplateEngineHandler` classes.

5.2 DAOs methods

PlaylistDAO methods:

- `getPlaylistTitle`
- `deletePlaylist`
- `getTrackGroup`
- `addTracksToPlaylist`
- `removeTracksFromPlaylist`
- `checkPlaylistOwner`
- `getUserPlaylists`
- `getPlaylistTracksByTitle`
- `createPlaylist`
- `getPlaylistTracksById6`
- `getTracksNotInPlaylist`
-  `updateTrackOrder`

TrackDAO methods:

- `addTrack`
- `isImageFileAlreadyPresent`
- `checkTrackOwner`
- `isTrackFileAlreadyPresent`
- `getTrackById`
- `getUserTracks`

UserDAO methods:

- `checkUser`
- `addUser`

All the methods are intuitively named and don't need further explanations. Either way, they are explained throughout the next section in their respective sequence.

5.3 RIA subproject

HomeView, class that manages the homepage:

- `show()` – Show the homepage
- `loadPlaylist()` – Loads all the User Playlists
- `loadButtons()` – Load buttons in the top nav bar and button functionality in the sidebar

⁶Modified in the RIA version to account for custom track order.

- `playlistGrid()` – Load the Playlists
- `createPlaylistButton()` – Creates and returns a button based on the playlist parameter
- `loadCreatePlaylistModal()` – Loads the modal for creating playlists to the modal container
- `loadUploadTrackModal()` – Loads the modal for uploading tracks to the modal container
- `loadUserTracksOl()` – Get user Tracks and creates draggable list items
- `dragStart()` – As soon as the User drags an Element
- `dragOver()` – The User is dragging the Element around
- `dragLeave()` – The User has started dragging the Element
- `drop()` – The User has dropped the Track in the desired location
- `loadReorderModal()` – Generates the modal to reorder the Tracks
- `closeReorderModal()` – Removes the reorder tracks modal
- `saveOrder()` – Save new Tracks custom order

MainLoader, centralized management of the HomePage:

- `start()` – Initializes the HomeView: adds listeners on buttons, refreshes the page
- `refreshPage()` – Refresh the HomeView: clear all modals and reload them
- `loadYears()` – Load year from 1900 to the current one for upload track modal
- `loadGenres()` – Load the musical genres for upload track modal

PlaylistView class:

- `trackGrid()` – Load all the Tracks associated to a Playlist
- `loadPlaylistTracks()` – Load all the Tracks associated to a Playlist
- `loadPlaylistView()` – Load everything needed for viewing and interacting with the Playlist and its contents
- `loadAddTracksModal()` – Load the modal for adding tracks to a playlist to the modal container

- `loadPrevNextButton()` – Load the buttons for changing the viewed track group in the playlist view

TrackView class:

- `show()` – Show the Track to play
- `trackPlayer()` – Load the Track player DOM elements. Unlike the other loaders, it's only a center panel
- `loadSingleTrack()` – Load a single Track from a Playlist

Utils(not a class but a file):

- `makeCall()` – Make an asynchronous call to the server by specifying method, URL, form to send, the function to execute and whether to reset the given form
- `loadUserTracks()` – Get user tracks and add them to the track selector parameter
- `createModal()` – Create basic modal element; used as a building block for creating modals
- `clearModals()` – Delete everything from modals div
- `cleanMain()` – Delete everything from main div
- `clearBottomNavbar()` – Delete the bottom navbar if present
- `showModal()` – Make the modal visible

And finally the interfaces, which are the TypeScript translation of the Record classes.

CLIENT SIDE		SERVER SIDE	
EVENT	ACTION	EVENT	ACTION
Index ⇒ Login form ⇒ Submit	Data validation	POST (username, password)	Credentials check
HomeView ⇒ Load	Loads all User playlists	GET (user playlists)	Queries user playlists
HomeView ⇒ Click on a playlist	Loads all tracks associated to that Playlist	GET (playlistId)	Queries the tracks associated to the given playlistId
HomeView ⇒ Click on reorder button	Load a modal to custom order the track in the Playlist	GET (playlistId)	Queries the tracks associated to the given playlistId
Reorder modal ⇒ Save order button	Saves the custom order to the database	POST (trackIds, playlistId)	Updates the playlist_tracks table with the new custom order
Create playlist modal ⇒ Create playlist button	Loads the modal to create a new playlist; returns the newly created playlist if successful	POST (playlistTitle, selectedTracks)	Inserts the new Playlist in the playlist table
Upload track modal ⇒ Upload track button	Loads the modal to upload a new track; returns the newly uploaded track if successful	POST (title, artist, year, album, genre, image, musicTrack)	Inserts the new Track in the tracks table
Sidebar ⇒ Playlist button	Views the last selected Playlist, if one had been selected	GET (last selected Playlist)	Queries the tracks associated to the given playlistId
Sidebar ⇒ Track button	Views the last selected Track, if one had been selected	GET (last selected Track)	Queries the data associated with the given trackId
Sidebar ⇒ HomePage	Returns to the HomeView	GET (user playlists)	Queries user playlists
Logout	Invalidates the current User session	GET	Session invalidation

Table 1: Events & Actions.

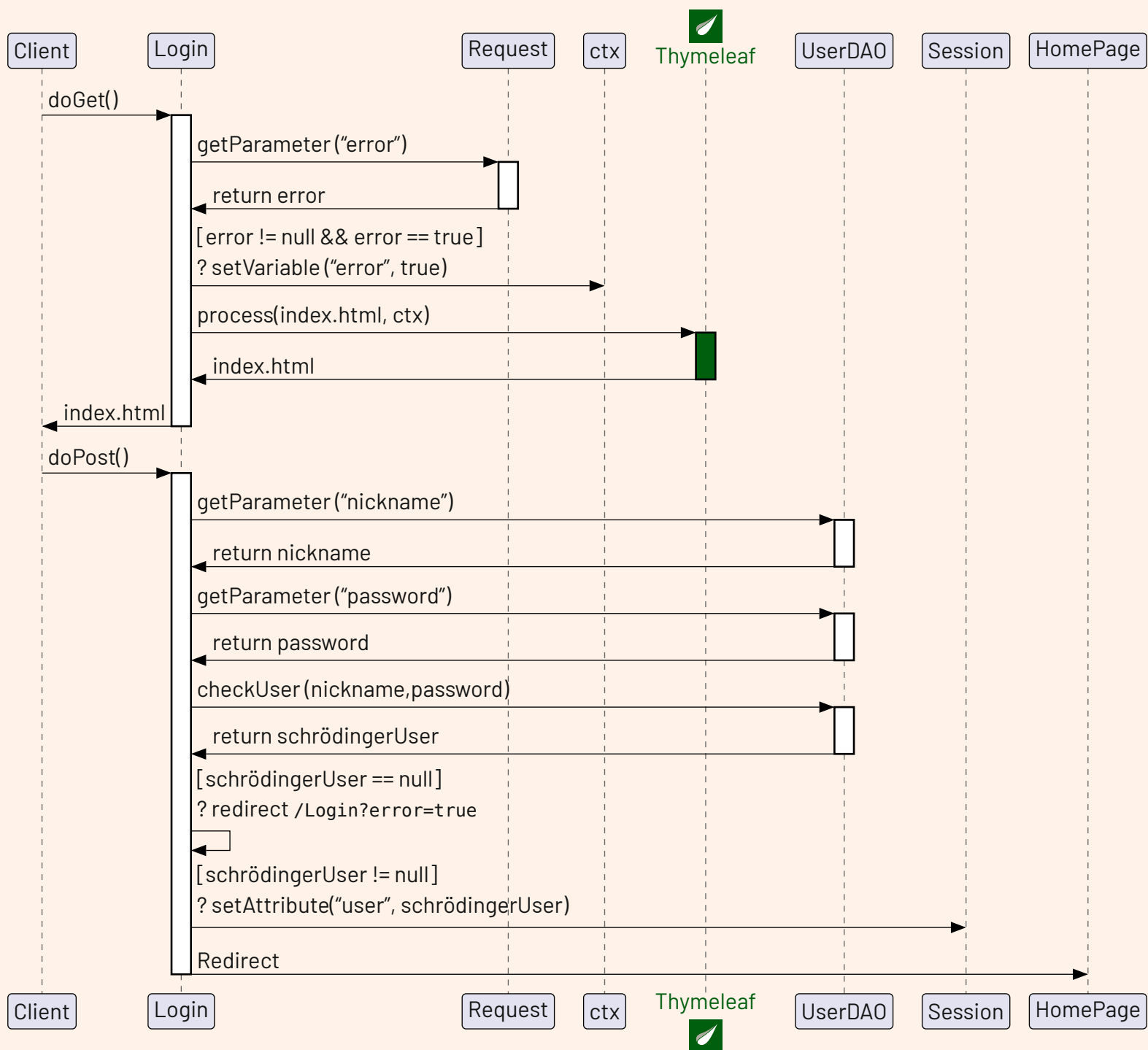
<i>CLIENT SIDE</i>		<i>SERVER SIDE</i>	
<i>EVENT</i>	<i>CONTROLLER</i>	<i>EVENT</i>	<i>CONTROLLER</i>
Index ⇒ Login form ⇒ Submit	makeCall() function	POST (username, password)	Login (servlet)
HomeView ⇒ Load	HomeView.show() function (its invocation is done by the MainLoader)	GET	Homepage (servlet)
HomeView ⇒ Click on a playlist	Loads all tracks associated to that Playlist	GET (playlistId)	Playlist (servlet)
HomeView ⇒ Click on reorder button	Load a modal to custom order the track in the Playlist	GET (playlistId)	Playlist (servlet)
Reorder modal ⇒ Save order button	Saves the custom order to the database	POST (trackIds, playlistId)	TrackReorder (servlet)
Create playlist modal ⇒ Create playlist button	Loads the modal to create a new playlist; returns the newly created playlist if successful	POST (playlistTitle, selectedTracks)	CreateNewPlaylist (servlet)
Upload track modal ⇒ Upload track button	Loads the modal to upload a new track; returns the newly uploaded track if successful	POST (title, artist, year, album, genre, image, musicTrack)	UploadTrack (servlet)
Sidebar ⇒ Playlist button	Views the last selected Playlist, if one had been selected	GET (last selected Playlist)	Playlist (servlet)
Sidebar ⇒ Track button	Views the last selected Track, if one had been selected	GET (last selected Track)	Track (servlet)
Sidebar ⇒ HomePage	Returns to the HomeView	GET (user playlists)	Homepage (servlet)
Logout	makeCall() function	GET	Logout (servlet)

Table 2: Events & Controllers (or event handlers).


6

Sequence diagrams


6.1 Login sequence diagram



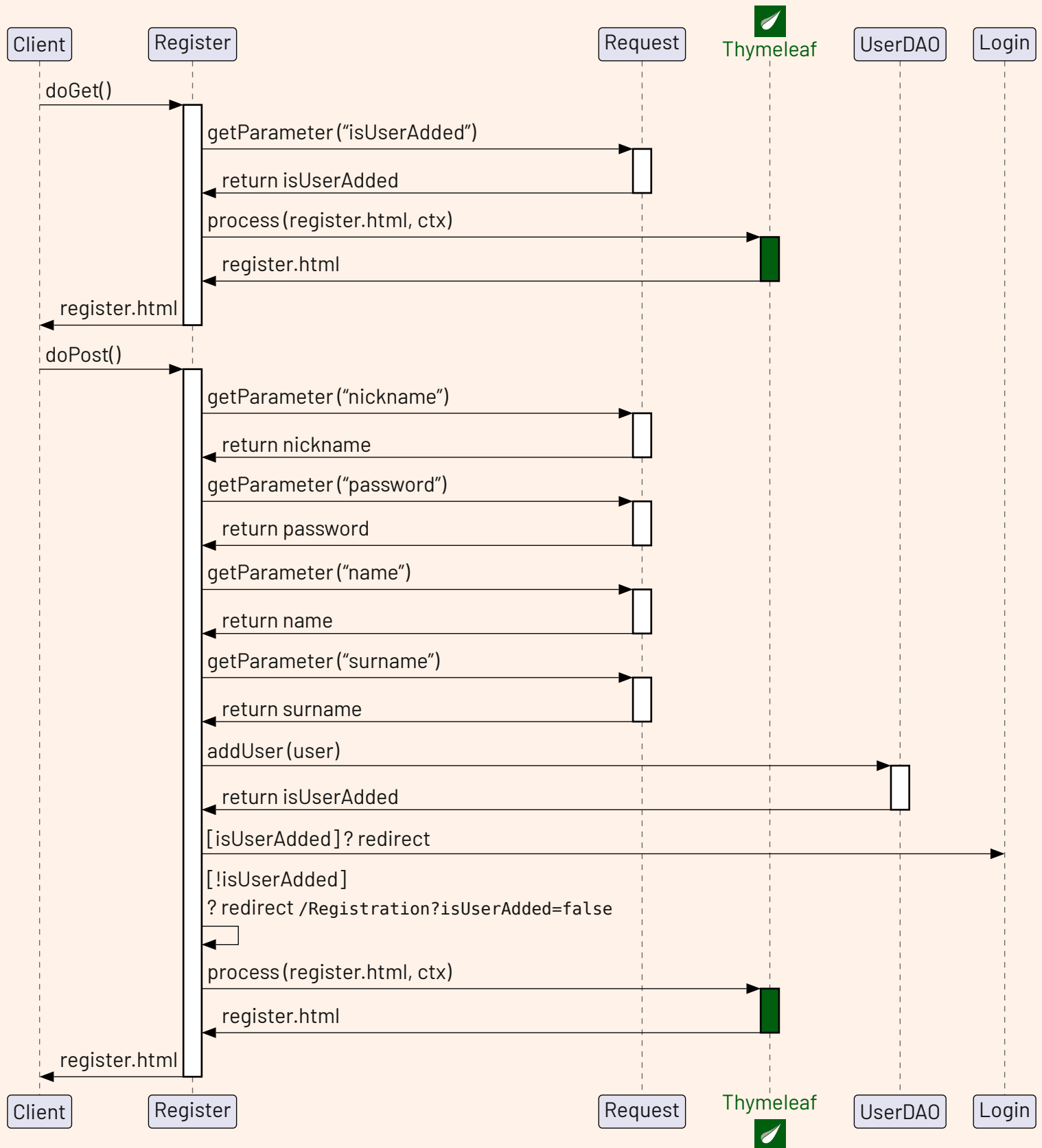
Comment

Once the server is up and running, the Client requests the Login page. Then, **thymeleaf**  processes the request and returns the correct context, to index the chosen locale. Afterwards, the User inserts their credentials.


Those values are passed to the `checkUser()` function that returns `schrödingerUser` – as the name implies, the variable might return a `User`; otherwise `null`. If `null`, then the credentials inserted do not match any record in the database; else the User is redirected to their `HomePage` and the `user` variable is set for the current session.

If there has been some error in the process – the credentials are incorrect, database can't be accessed... – then the servlet will redirect to itself by setting the variable `error` to `true`, which then will be evaluated by **thymeleaf**  and if `true`, it will print an error; otherwise it won't (this is the case for the first time the User inserts the credentials).

6.2 Register sequence diagram




Comment

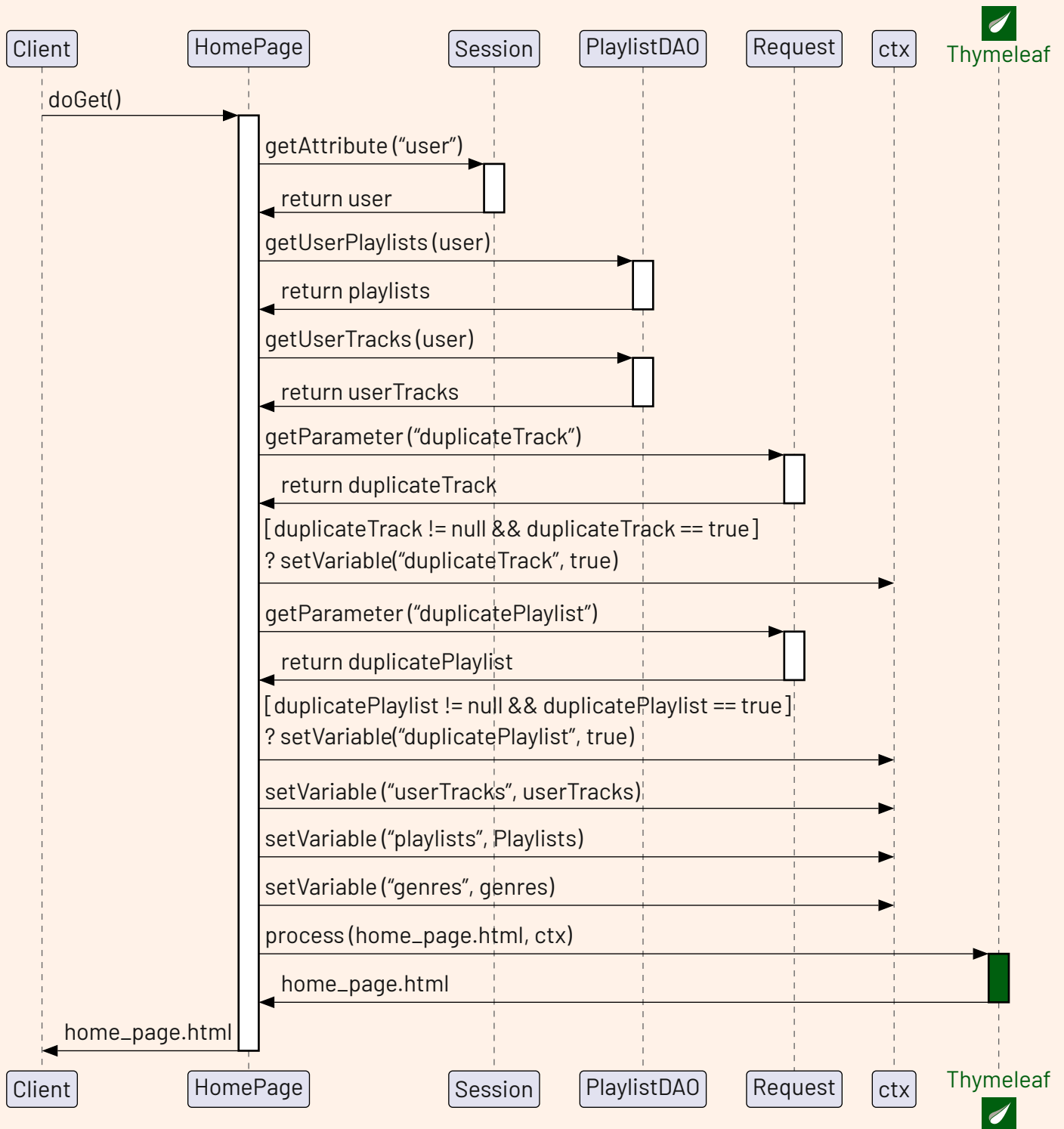
If the User is not yet registered, they might want to create an account. If that's the case, as per the Login sequence diagram, once all the parameters are gathered and verified (omitted for simplicity) initially **thymeleaf**  processes the correct context, then the User inserts the credentials.

Depending on the nickname inserted, the operation might fail: there can't be two Users with the same nickname. If that does not happen, then


isUserAdded is *true*, then there will be the redirection to the Login page.

Else the program appends *isUserAdded* with *false* value and redirects to the Registration servlet: **thymeleaf**  checks for that context variable and if it evaluates to false, it prints an error.

6.3 HomePage sequence diagram



Comment

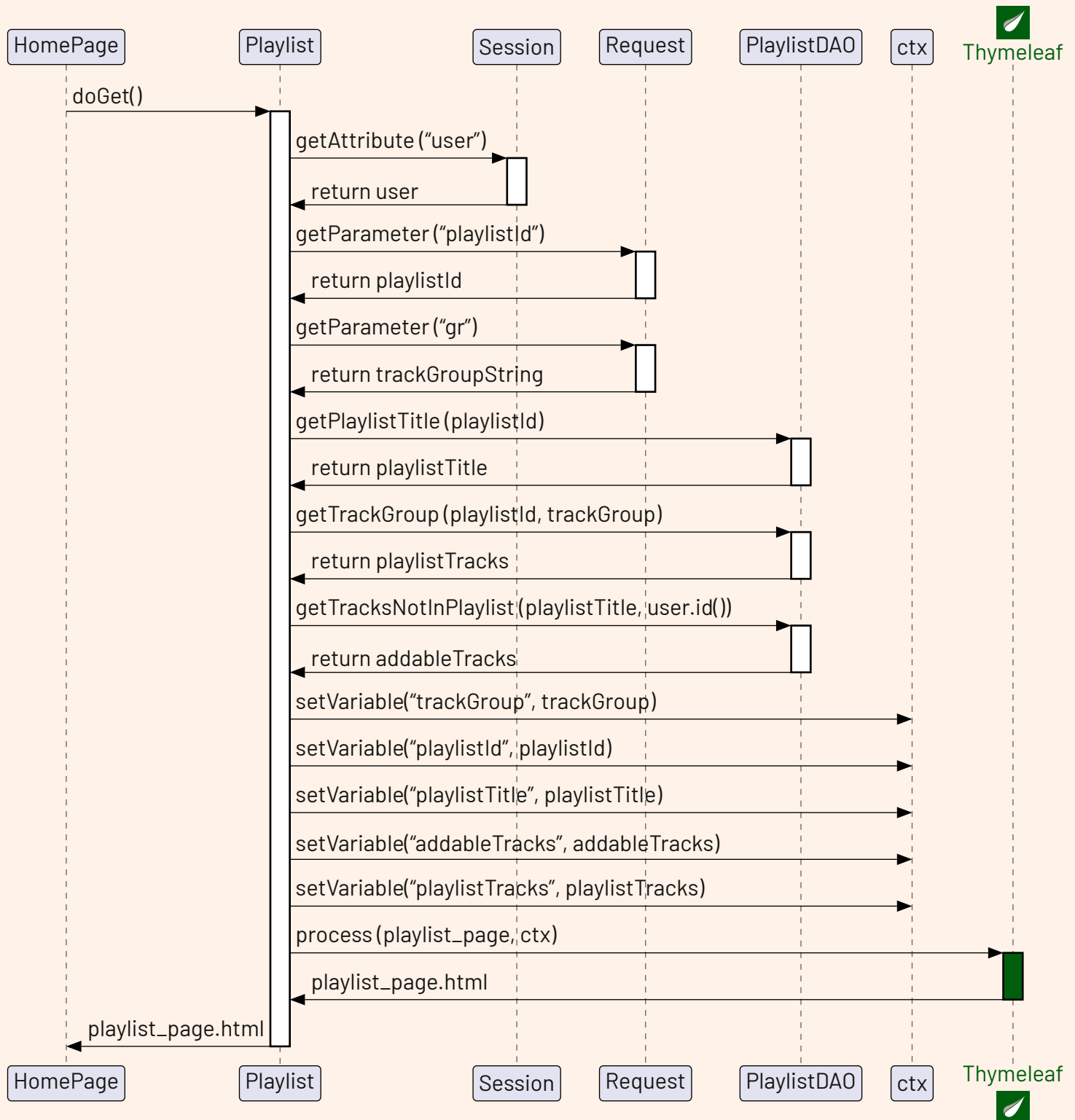
Once the Login is complete, the User is redirected to their HomePage, which hosts all their Playlists. In order to do so, the program needs to User attribute – which is retrieved via the session; then, it is passed to the `getUserPlaylists` function and finally **thymeleaf**  displays all values.

From this page, the User can upload new tracks. for this reason the HomePage servlet fetches all the user tracks (which are not to be displayed). Then, as the User presses the upload button, the modal

shows up allowing to fill the information for a new track (title, album, path, playlist...); the genres are predetermined: they are statically loaded from the `genres.json` file.

Once the information are completed, the servlet checks if a playlist or track is duplicate – hence the need to fetch all the tracks – and if so it redirectes to itself with a `duplicate`- error, the same principle applied to the precedent servlets. Otherwise, the track would have been successfully added.


6.4 Playlist sequence diagram



Comment

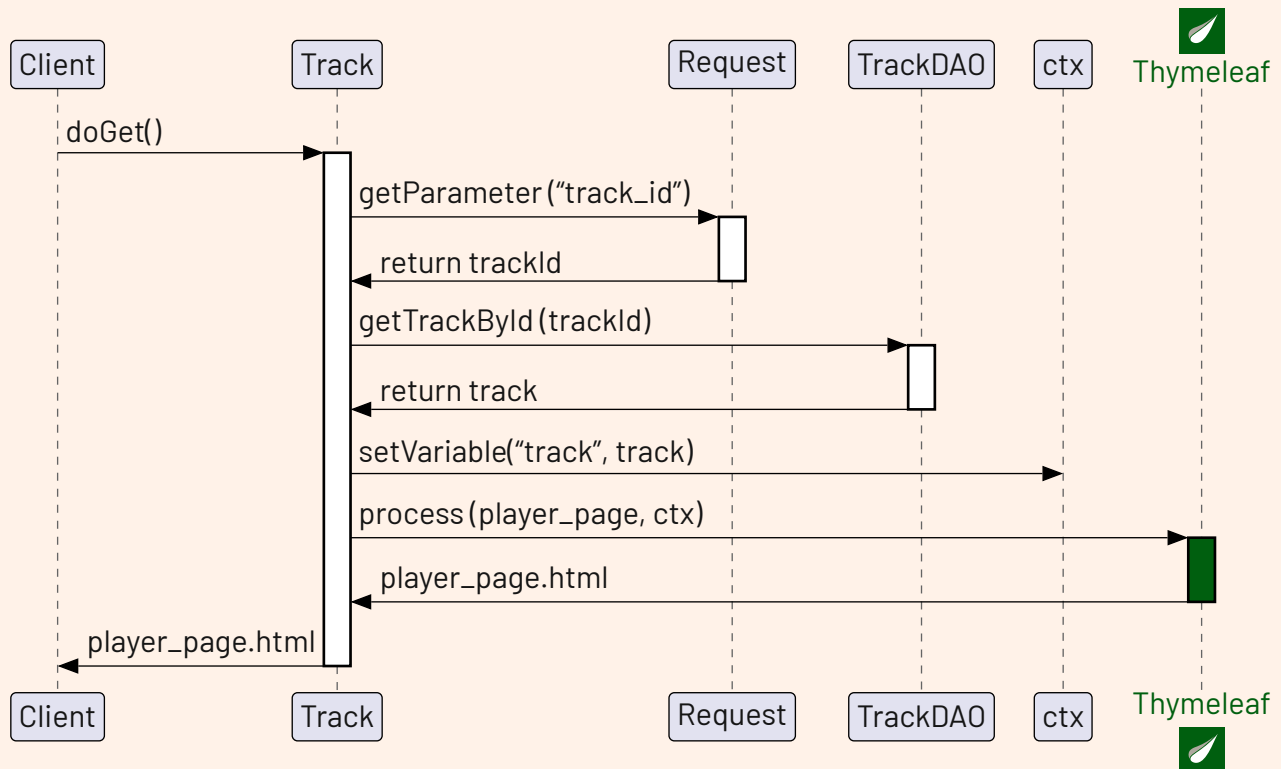
From the HomePage, the User is able to see all their playlists. By clicking on either one of them, the program redirects to the corresponding PlaylistPage, which lists all the tracks associated to that playlist.

In order to do so, the program needs the User attribute – which is retrieved via the session – and the title of the playlists, which is given as a parameter by pressing the corresponding button in HomePage.

Then those value are passed to `getPlaylistTracks()`, that returns all the tracks. Finally, **thymeleaf**  processes the context and display all the tracks.


From this page the User is also able to add chosen tracks to a playlist. In order to do, similar to HomePage with the upload, the program fetches all tracks that can be added, thats is the ones that are not already in a playlist, and displays them to a User via a dropdown menu (again similar to genres in HomePage).

6.5 Track sequence diagram

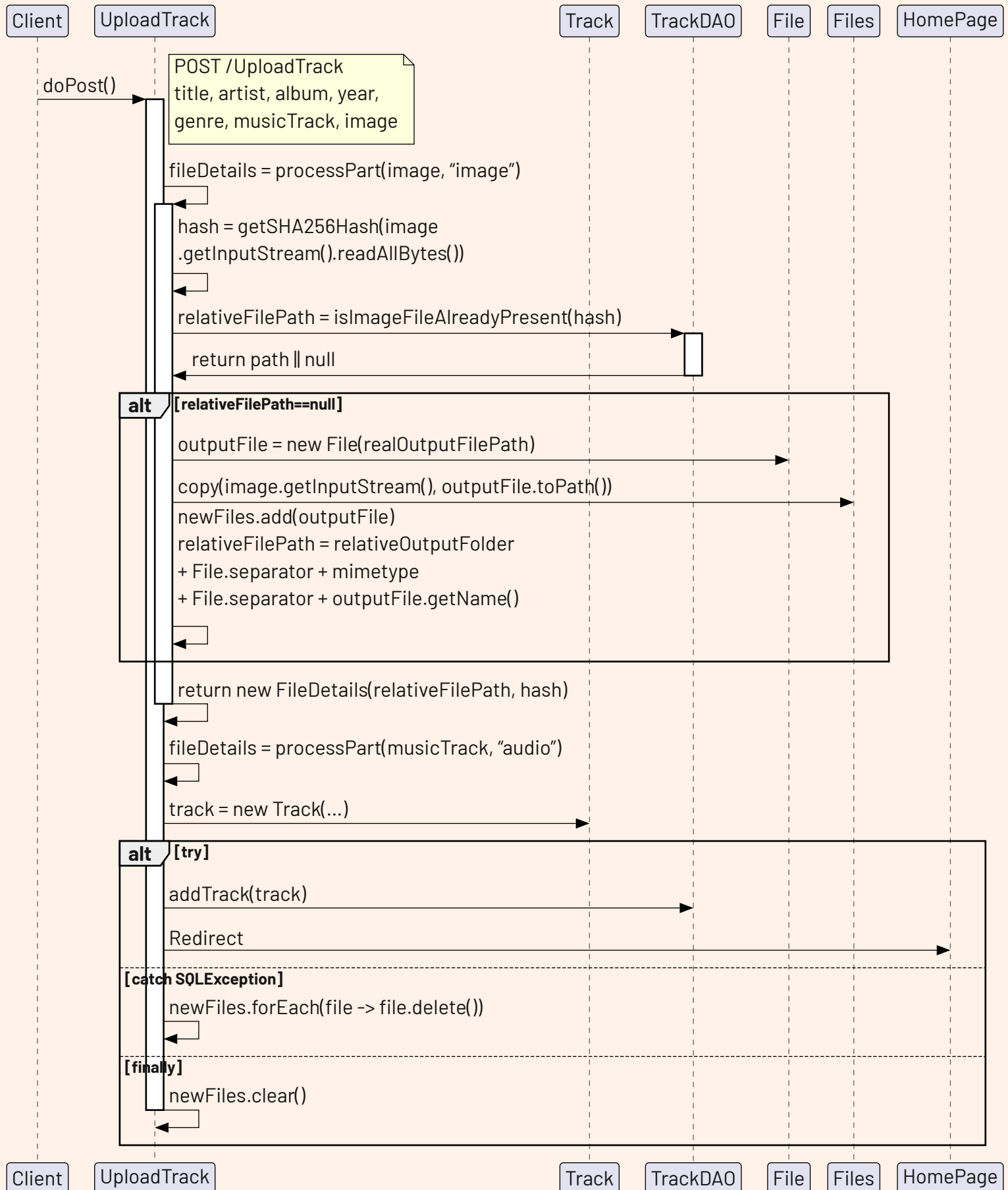


Comment

Once the program has loaded all the tracks associated to a playlist, it allows to play them one by one in the dedicated player page. In a similar fashion to the `getPlaylistTracks()` method, in order to retrieve all the information regarding a single track the program is given the `track_id` parameter by pressing the corresponding button.

Finally, `getTrackById()` returns the track metadata – that is title, artist, album, path and album image – **thymeleaf**  then processes the context and displays all the information. If an exception is caught during this operation, the server will respond with `ERROR 500` (see [Section 7.5](#)).

6.6 UploadTrack sequence diagram



Comment

The User can upload tracks from the appropriate form in the homepage (Section 6.3). When the POST request is received, the request parameters are checked for null values and emptiness (omitted in the diagram for the sake of simplicity), and the uploaded files are written to disk by the *processPart* method, which has two parameters: a *Part* object, which “represents a part or form item that was received within a multipart/form-data POST request” [11], and its expected MIME type. The latter does not need to be fully specified (i.e. the subtype can be omitted).

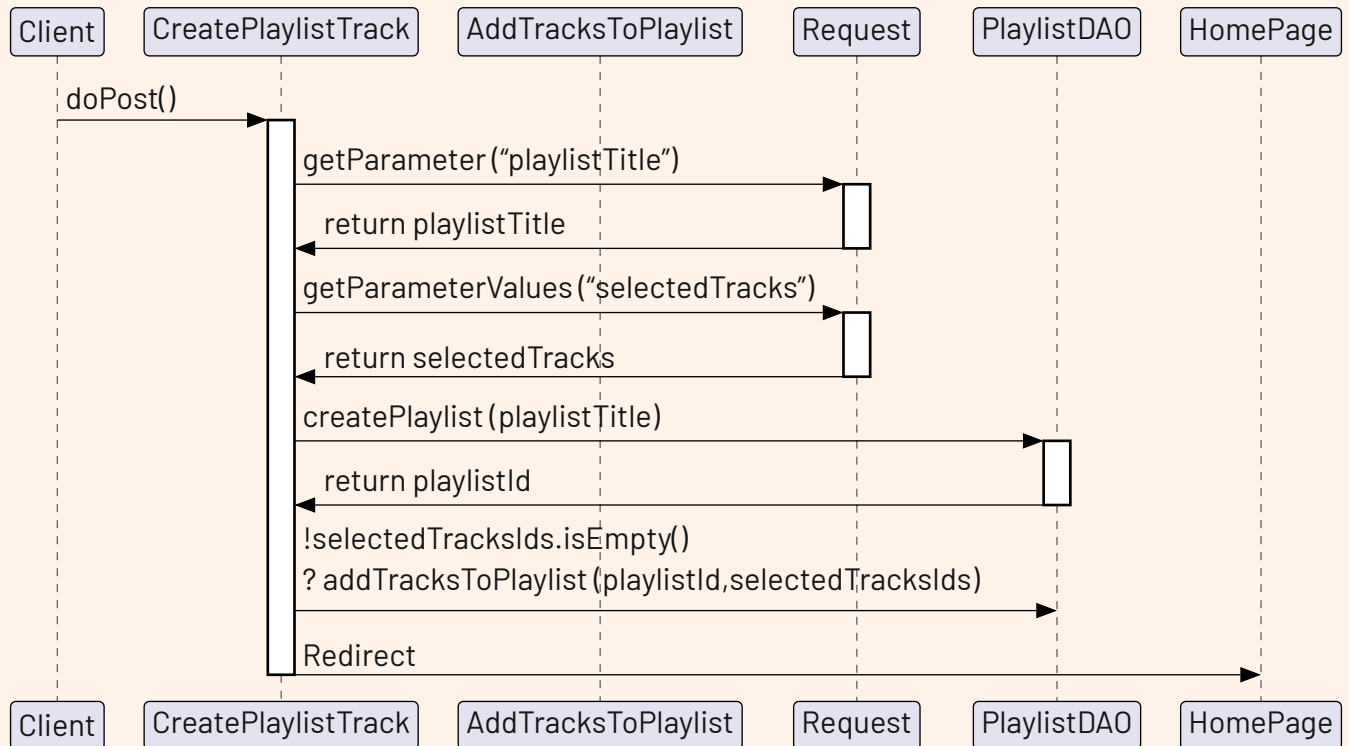
Before writing the file to disk, the method checks for duplicates of the file by calculating its SHA256 hash and querying the database with the two methods: *isTrackFileAlreadyPresent* and *isImageFileAlreadyPresent*; present in *TrackDAO*.

Those two return the relative file path corresponding to the file hash if a matching one is found, otherwise null. In the former case, *processPart* returns the found path and the new track is uploaded using the already present file, this avoiding creating duplicates; in the latter case *processPart* proceeds by writing the file to disk and returning the new file’s path.

To write the file to the correct path in the webapp folder (*realOutputFolder*), the method *context.getRealPath(relativeOutputFolder)* is called, where *relativeOutputFolder* is obtained from the *web.xml* file and is, in our case, “*uploads*”; *realOutputFolder* is obtained by appending, with the needed separators, the MIME type to the result of *getRealPath*; to get *realOutputFilePath*, a random UUID and the file extension are appended to *realOutputFolder*. Having obtained the desired path, the file can be created and then written with the *Files.copy* method. The file can be found in *target/artifactId-version/uploads/* in the project folder.

In conclusion, *processPart* adds the new file to the *newFiles* list in *UploadTrack* and returns the path relative to the webapp folder because that’s where the application will be looking for when it has to retrieve files. Once this is completed, the new *Track* object is created and passed to the *addTrack* method of *TrackDAO*; if an *SQLException* is thrown, all the files in *newFiles* list are deleted and then, in the finally block, the list is cleared.

6.7 CreatePlaylist sequence diagram



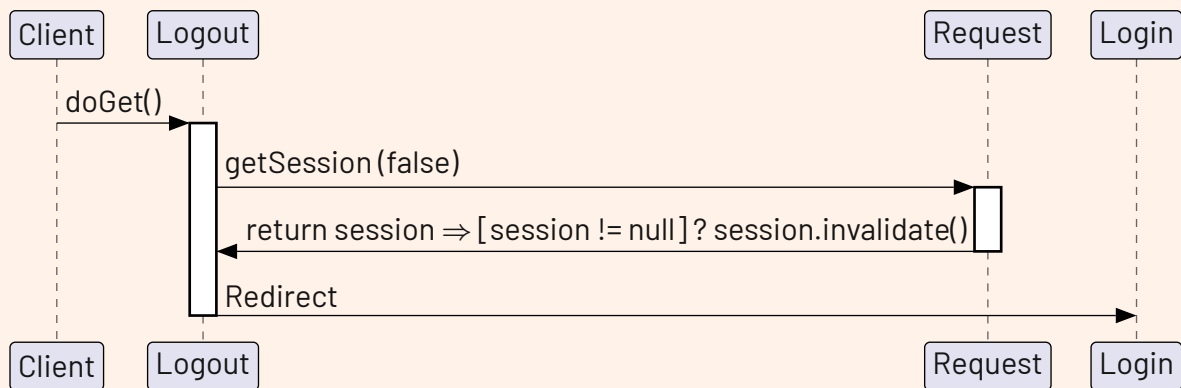
Comment

The User can create playlists with the appropriate form in the homepage. There, a title needs to be inserted and, optionally, one or more tracks can be chosen from the ones uploaded by the User. When the servlet gets the POST request, it interacts with the `PlaylistDAO` to create the playlist with the

`createPlaylist()` method and to add the selected tracks with the `addTracksToPlaylist()` method.

Note that `selectedTracksIds` is a list of integers obtained by converting the strings inside the array returned by the `getParameterValues("selectedTracks")` method with `Integer.parseInt()`.

6.8 Logout sequence diagram

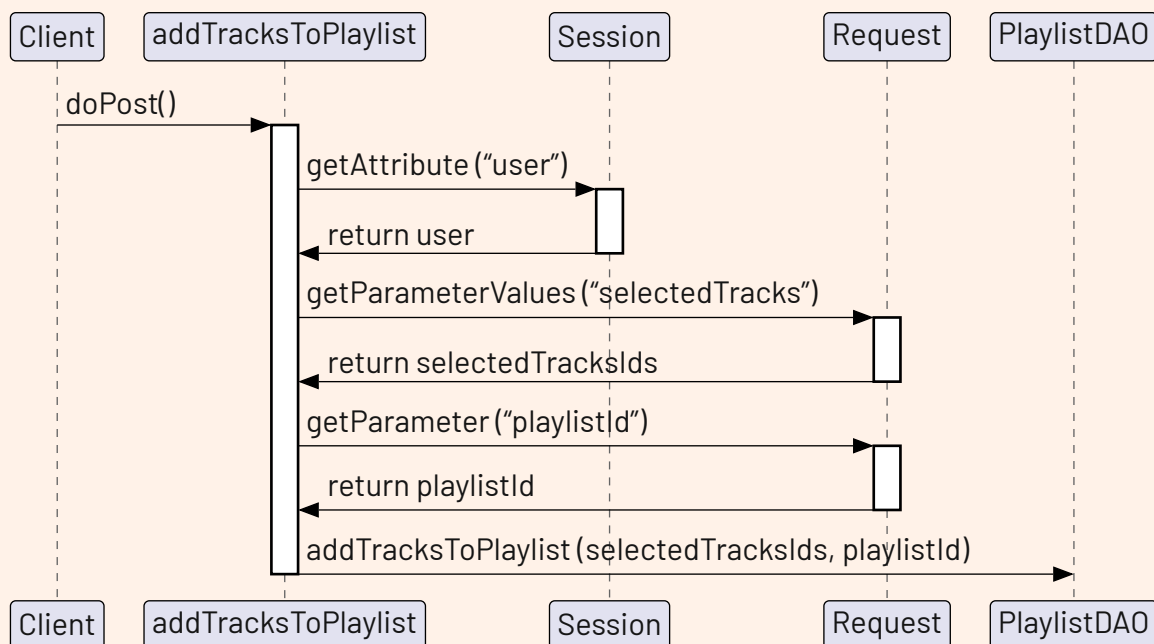


Comment

From every web page except Login and Register, the User is able to logout, at any moment. It's a simple GET request to the Logout servlet, which

checks if the user session attribute exists; if it does, then it invalidates the session and redirects the User to the Login page.

6.9 AddTracks sequence diagram



Comment

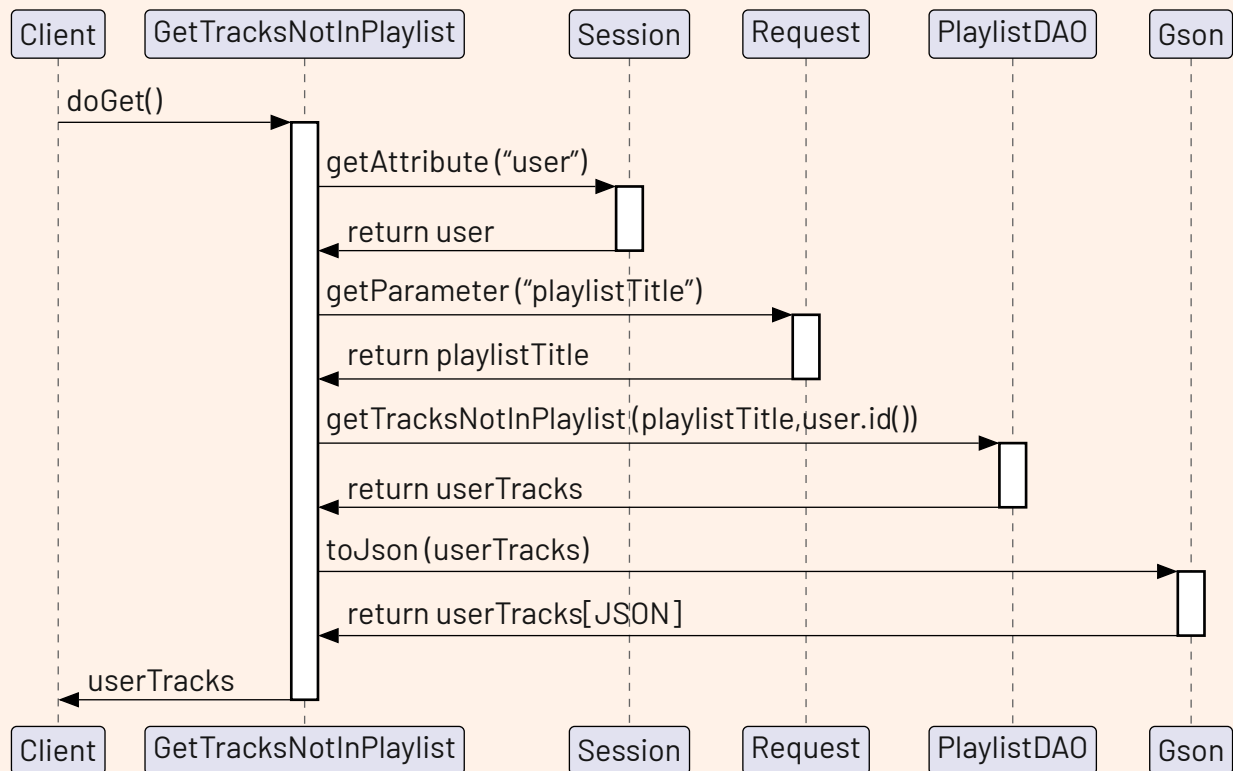
From the modal, once the User has completed the selection of the Tracks to add in the current Playlist, the form calls the AddTracks servlet via a POST method.

Afterwards, by making sure there are no `nulls` in the `selectedTracksIds`, the `addTracksToPlaylist`

method is called: it performs an insertion in the `playlist_tracks` table. Finally, the User is redirected to the newly created Playlist.

TS In the RIA subproject, the servlet response with a 201 code instead of redirecting.

6.10 TS GetTracksNotInPlaylist sequence diagram



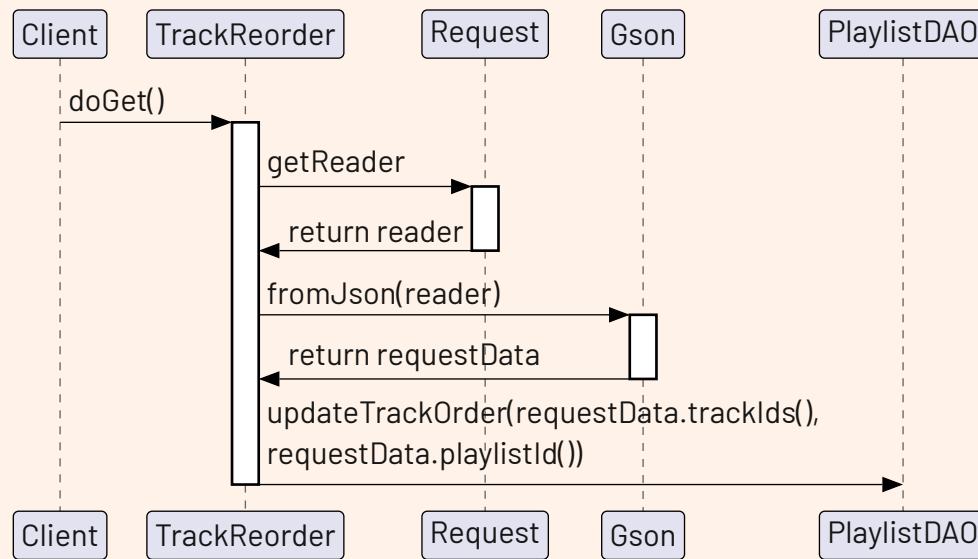
Comment

As the name suggests, this servlet obtains the tracks are not in the given Playlist, in order to display them when the User wants to add a new track to a Playlist – this happens when the User clicks on the corresponding button.

Then, the User attribute is retrieved from the session while the playlist title from the request.

In conclusion, the tracks that are not in the playlist are retrieved by the `getTracksNotInPlaylist` method: it returns a list which is converted to a JSON object via Gson for JavaScript.

6.11 TS TrackReorder sequence diagram

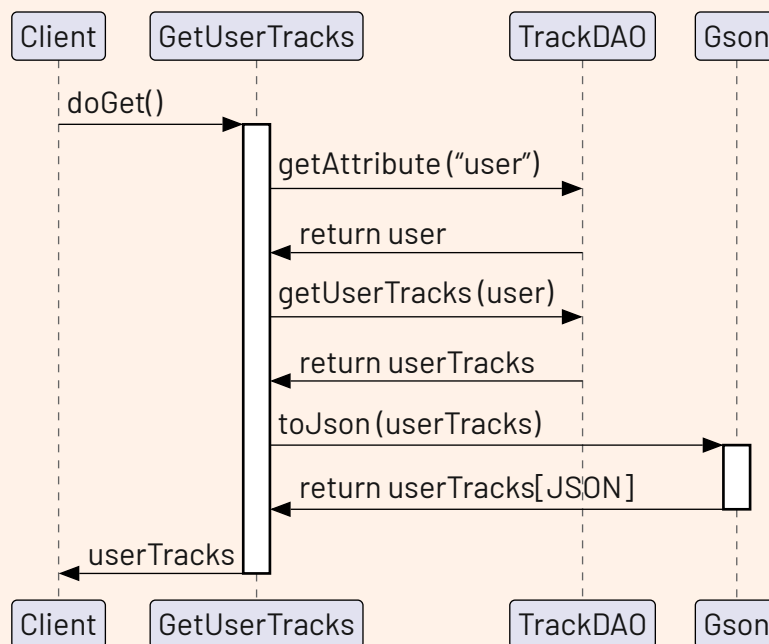


Comment

This servlet obtains the needed parameters by leveraging a JSON and a Record class. Javascript parses all the information and then sends them as a JSON to Java, which maps it all to the `RequestData` record class.

Afterwards, the `tracksIds` and `playlistId` attributes are passed to the `updateTrackOrder` method that loads multiple insertions in the database: instead of iterating and performing a query at each cycle, it prepares a transaction to be committed one single time.

6.12 GetUserTracks sequence diagram

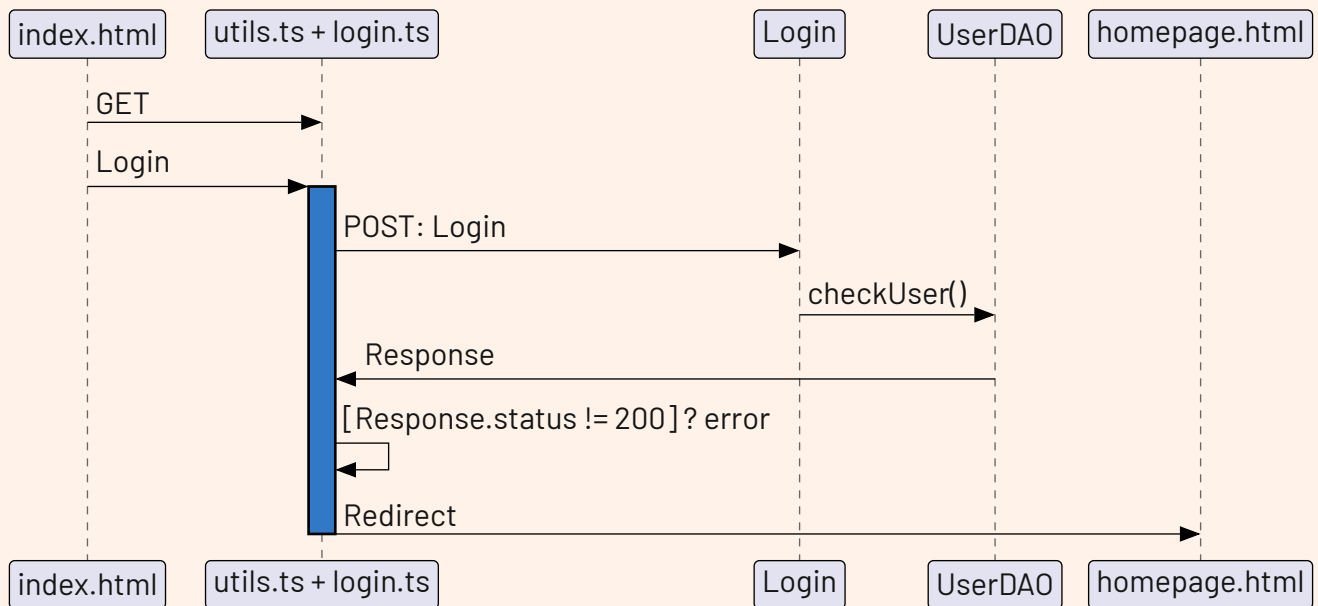


Comment

As the name implies, this servlet retrieves all the Tracks associated to a User. This is fetched as usually from the session.

Similar to the previous sequences, once it retrieves the track from the database, the list is transformed into a JSON by Gson and finally sent to the browser.

6.13 TS Event: Login



Comment

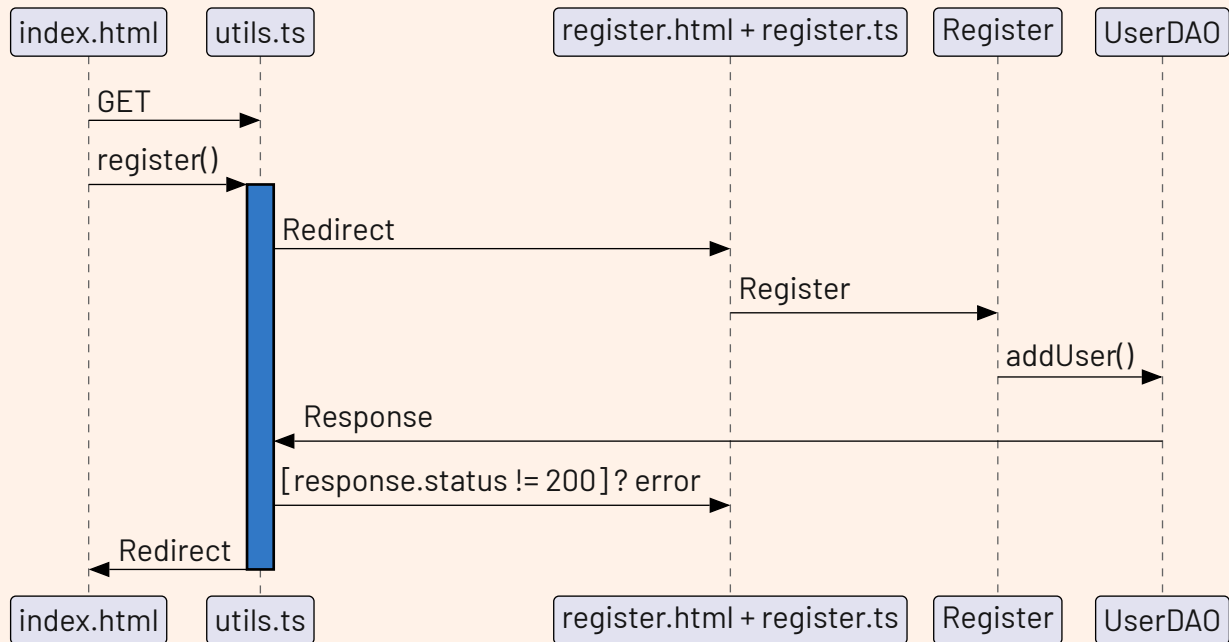
As the server is deployed the *index.html* request the associated Javascript files (we use Typescript, but it transpiles to Javascript and that's what is imported in the HTML files). As they have been loaded thanks to the IIFE, the User is able to Login.

Once the button has been clicked, Javascript performs a POST request – always via the *makeCall()*

function – to the Login servlet, which, as seen in the Login sequence diagram ([Section 6.1](#)), checks if the User exists: if that's the case it returns a 200 OK and the User is redirected to the Homepage.

If not, then a error div will appear above the Login button.

6.14 TS Event: Register

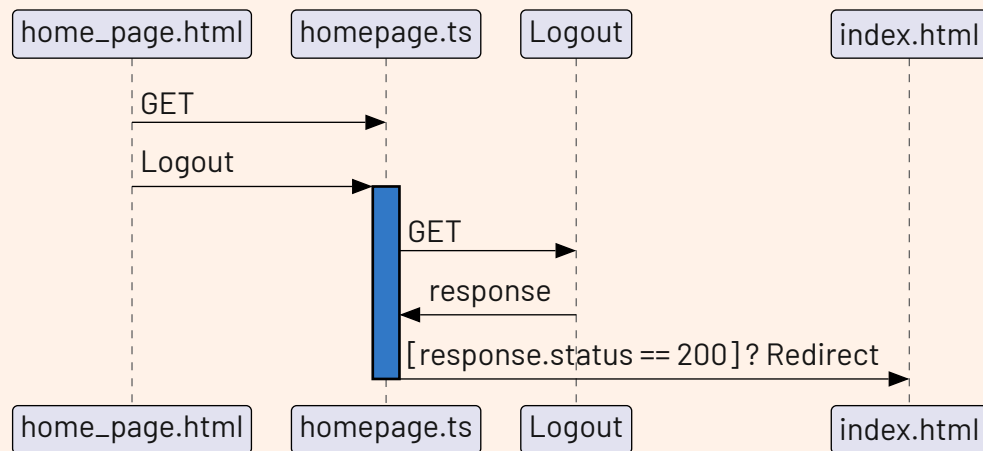


Comment

Instead of logging in, the User may want to register: probably because there they have no account. If that's the case, after the Javascript files will have been fetched, the User will be redirect to the `register.html` page.

From here, as seen in the Register sequence (Section 6.2), the servlet adds the User: if that's successful, then there will the redirect to the `index.html`; if not, an error message will appear above the Register button.

6.15 TS Event: Logout

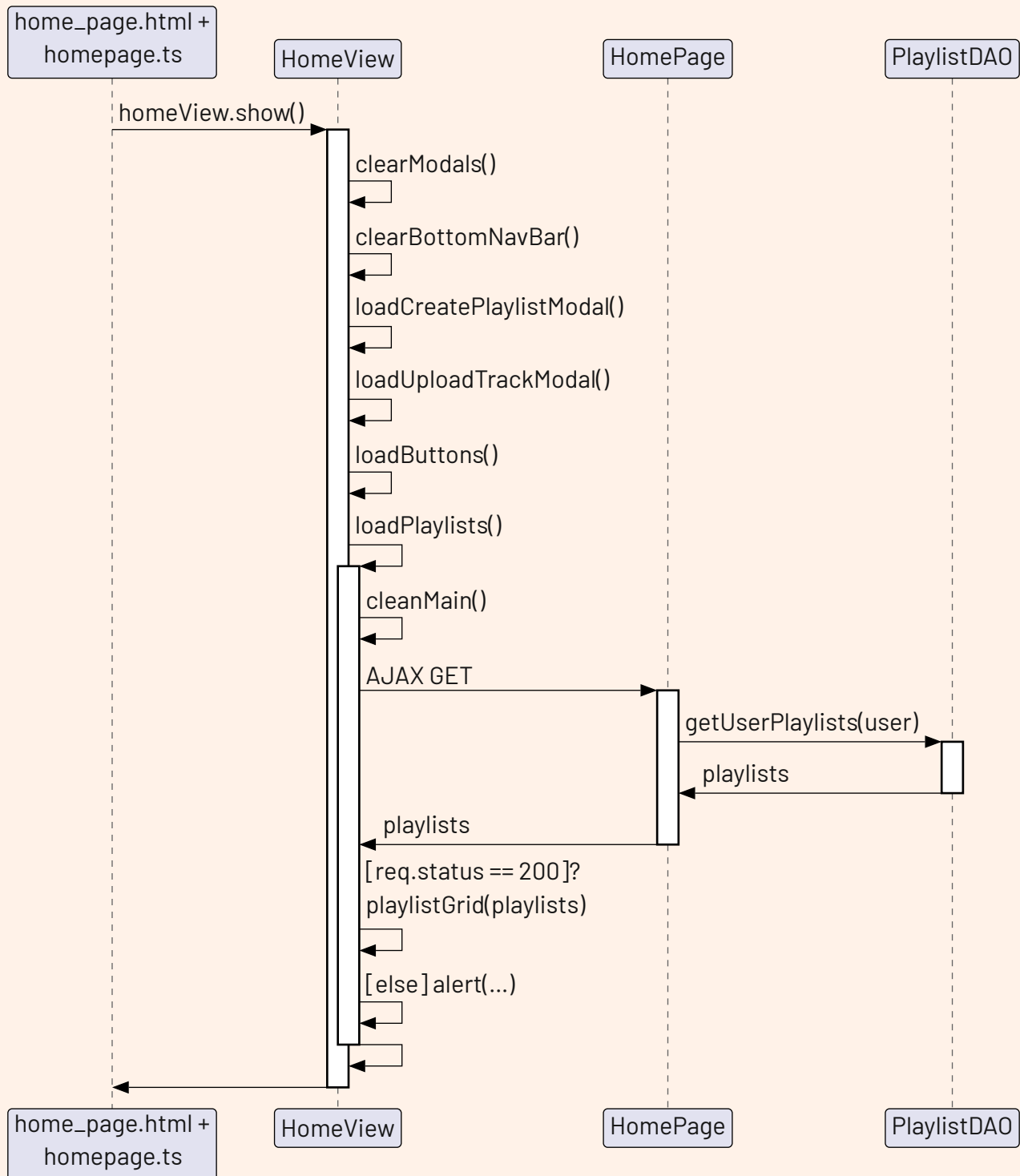


Comment

The User is able to logout every moment after the Login. As the Logout button is pressed, Javascript performs a GET request to the Logout servlet: it

responds with 200 OK if the session has been invalidated; else nothing will happen.

6.16 TS Event: Access HomeView

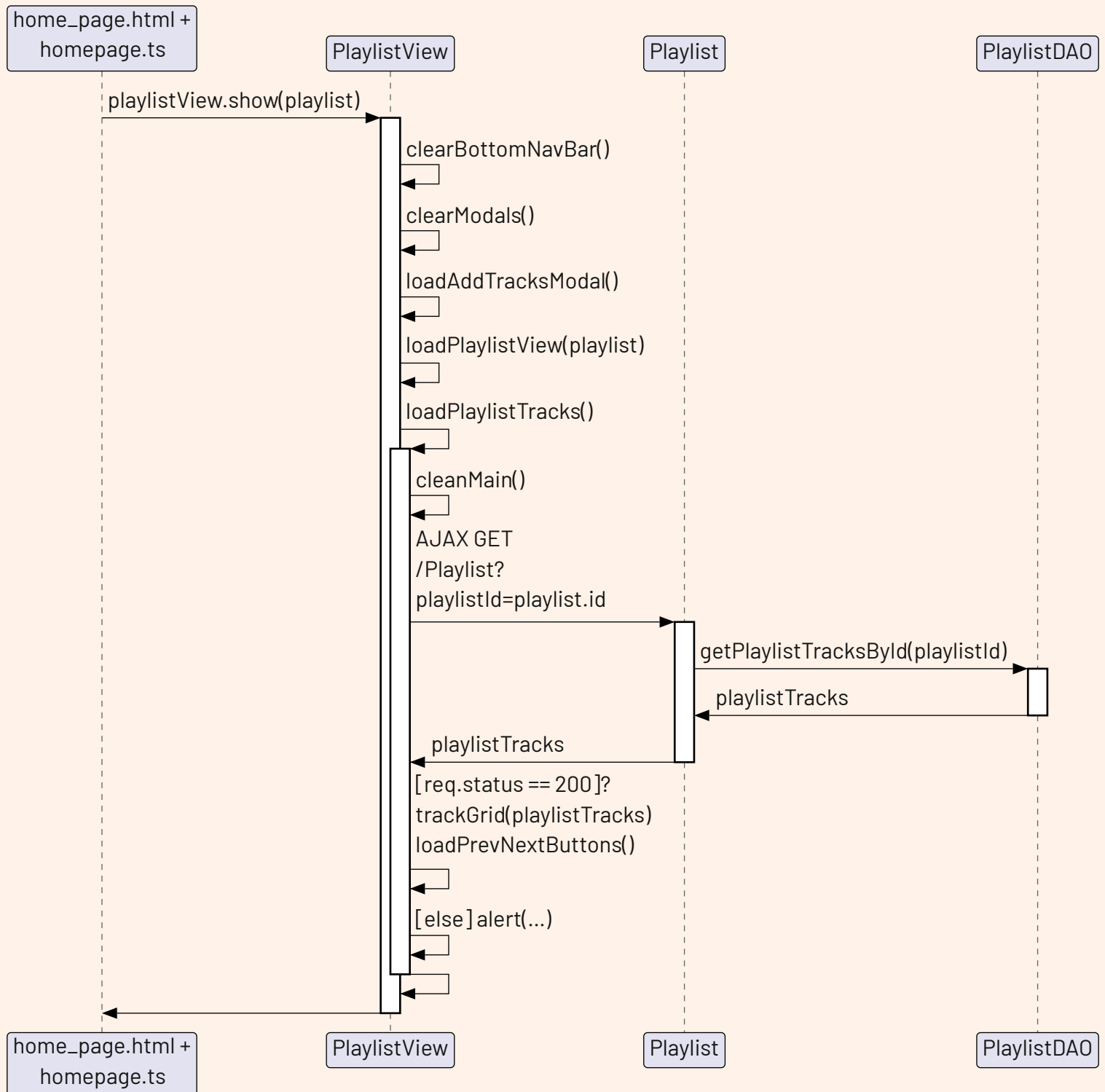


Comment

The user can access the home view when the `home_page.html` first loads or after pressing the `Homepage` button in the sidebar. The view is loaded by calling the `show` method of the `HomeView`

object, which clears the possibly remaining elements left by other views and loads the modals, buttons, playlists and event listeners associated to them.

6.17 TS Event: Access PlaylistView

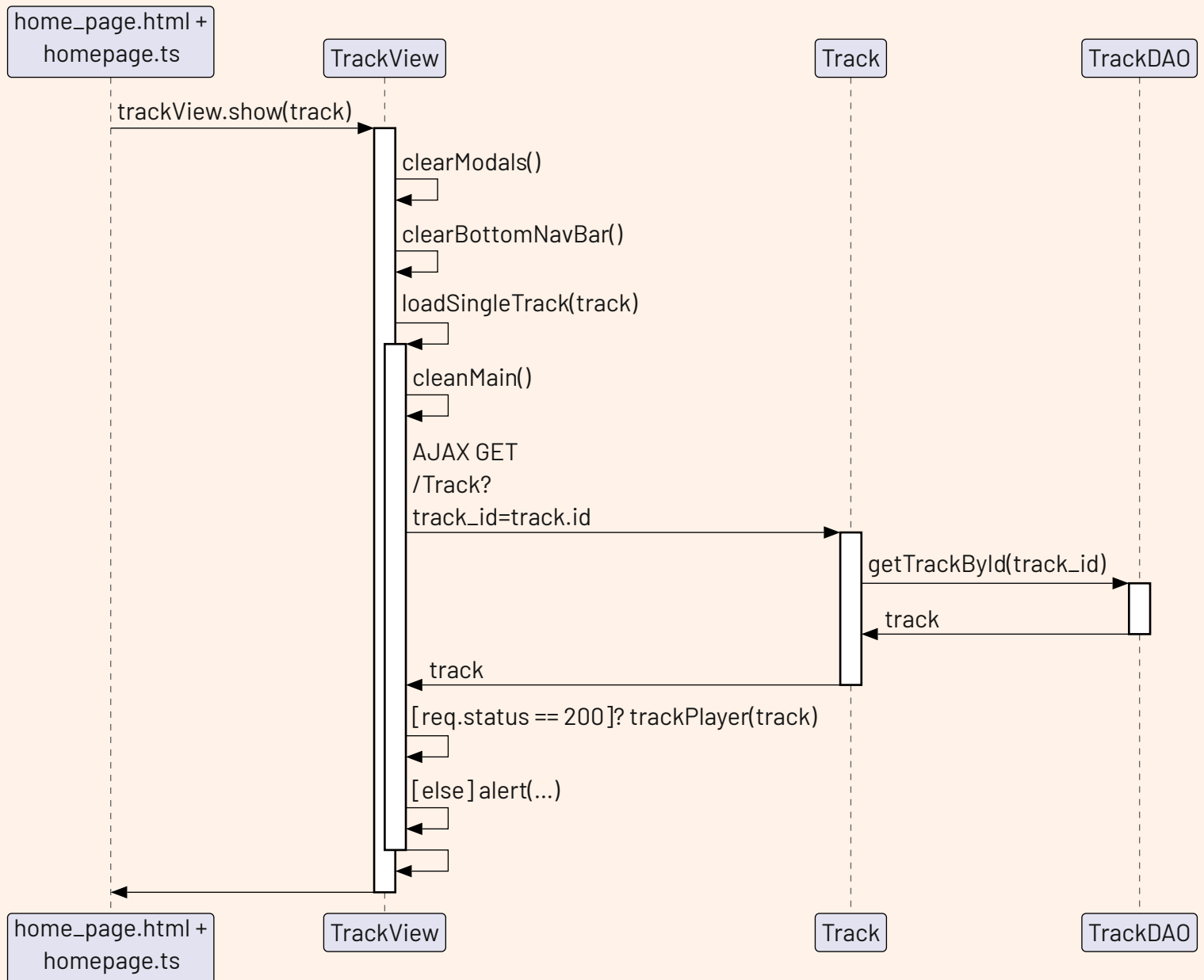


Comment

The user can access the playlist view by selecting a playlist in the home view or by pressing the Playlist button in the sidebar, which will open the last visted playlist. The view is loaded by calling

the `show` method of the `PlaylistView` object, which clears the elements from other views and loads the modal, buttons, tracks and event listeners associated to them.

6.18 TS Event: Access TrackView



Comment

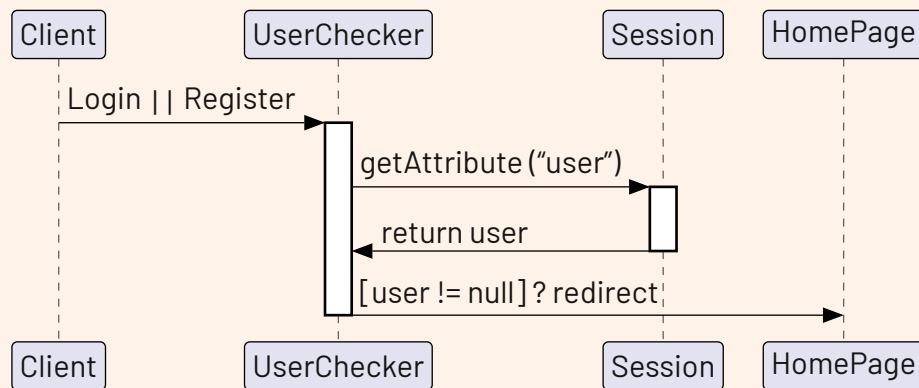
The user can access the track view by selecting a track in the playlist view or by pressing the Track button in the sidebar, which will open the last

visited track. The view is loaded by calling the *show* method of the *TrackView* object, which clears the elements from other views and loads the player and the song details.

7

Filter mappings

7.1 UserChecker filter

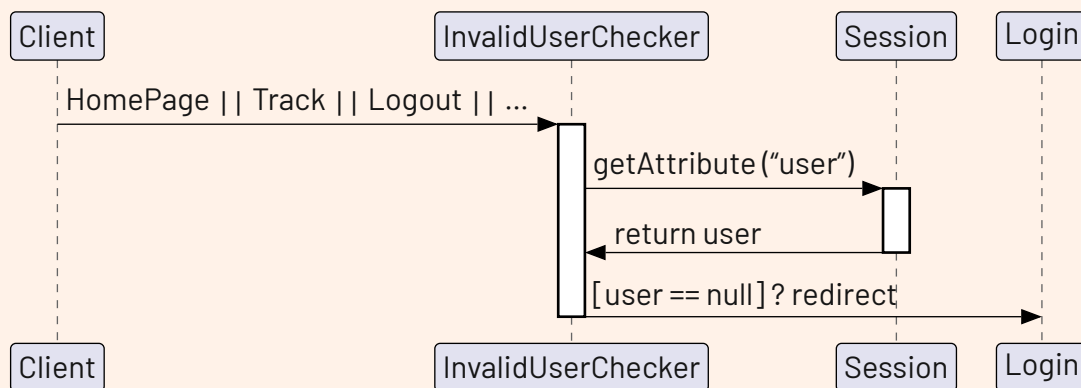


Comment

The *UserChecker* filter checks, once the client accesses the Login or Register webpage, if the User is logged.

If that's the case, then the program redirects to the HomePage. If not, then the *InvalidUserChecker* filter comes in.

7.2 InvalidUserChecker filter

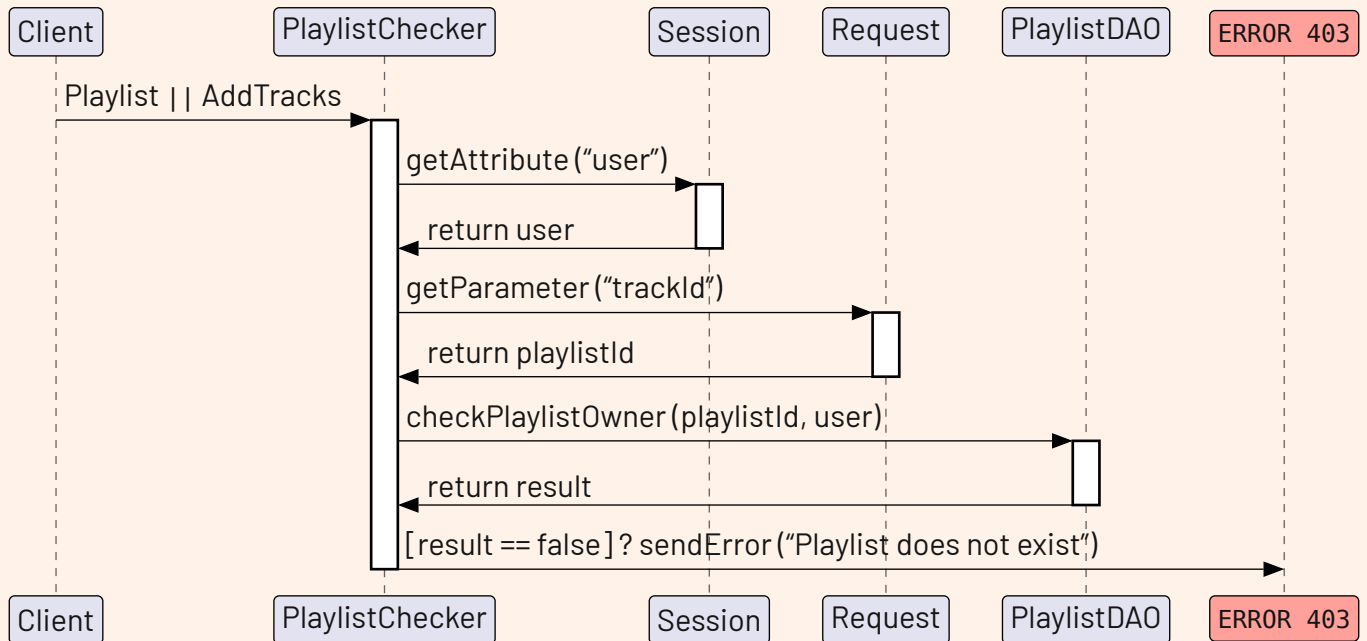


Comment

The *InvalidUserChecker* filter does the exact opposite of *UserChecker*. If the client accesses pages all the other pages – HomePage, PlaylistPage,

Track, Logout... – and is not logged in, then the program redirects to the Login page.

7.3 PlaylistChecker filter



Comment

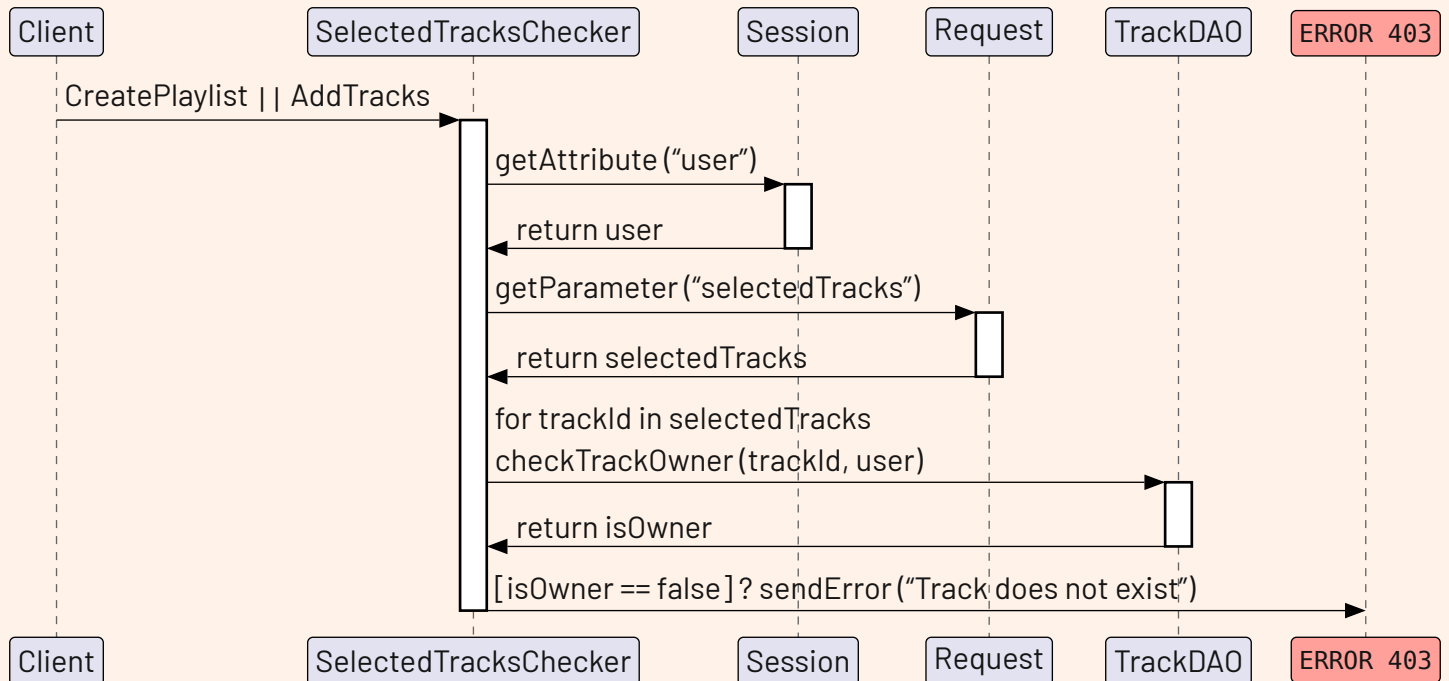
The *PlaylistChecker* filter is invoked in two scenarios: after the User has clicked on a playlist on *HomePage* ([Section 6.4](#)) and when uploading a track ([Section 6.6](#)).

It is in charge of checking if the requested playlist actually belongs to the User requesting or trying to upload it. This is done via obtaining the User

attribute from the session – which is impossible without extending the *HttpServlet* or *HttpFilter* classes – and getting the needed parameters from the request.

Finally, a query is performed against the database. If the result is false, then the server will respond with *ERROR 403: forbidden*.

7.4 SelectedTracksChecker filter



Comment

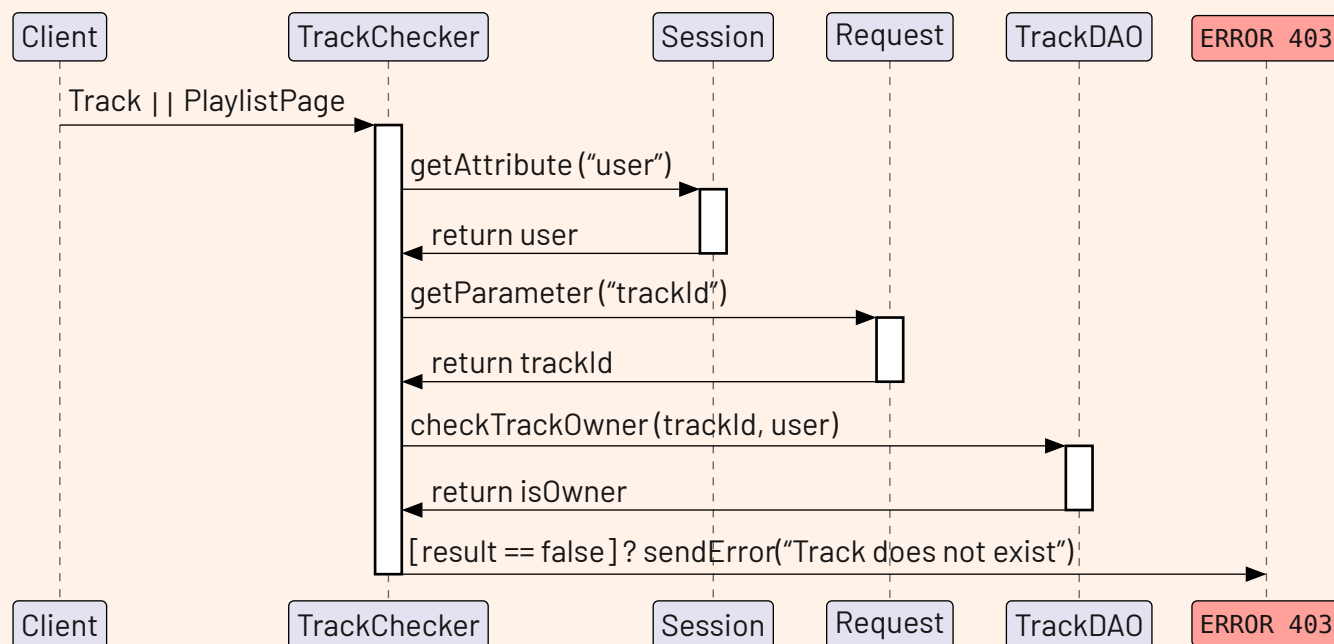
Even the *SelectedTracksChecker* filter is invoked in two scenarios: during the creation of a playlist ([Section 6.7](#)) and during the *UploadTrack* sequence ([Section 6.6](#)).

SelectedTracksChecker applies a very similar pipeline *PlaylistChecker*: instead of checking the

playlist, it does the same job but for one of more tracks when the User requests to add them to a playlist.

Again similarly to *PlaylistChecker*, it also obtains the User attribute from the session and the needed parameters; if the User does not have access rights to the requested track(s), the response is *ERROR 403*.

7.5 TrackChecker filter



Comment

Finally the *TrackChecker* filter does the same exact job as *SelectedTracksChcker*, but for a single track

once a User presses the corresponding button in the *playList_page* (see [Section 6.5](#)).

8

Cascading Style Sheets (CSS) styling

8.1 Introduction

The project is based on a single CSS file, `components.css`, and all the others rely upon it to retrieve the styles. Furthermore, all the colours are sourced from the `colors.css` file, which is based on *tinted-theming* [12], a collection of commonly used themes in the developing world. We have chosen to use the *Classic Light* theme⁷.

If you want to change the overall theme of the website, just switch to a new colorscheme by looking at the [tinted-theming gallery](#). In `colors.css` there are a few commented styles to choose from.

```
body {
  background-color: var(--default-
background);
  padding: 1rem 2rem 2rem 2rem;
  line-height: 1.6;
  word-spacing: 1px;
  font-family: "JetBrains Mono",
monospace;
  height: 100vh;
  text-overflow: ellipsis;
}
```

As stated earlier, the `background-color` is sourced from the `colors.css`. Then the padding is always `2rem`, except above, where it's `1rem`. The text is able to wrap thanks to `ellipsis` option on `text-overflow`.

After the body, we styled all the elements in a consistent manner.

8.2 Buttons

```
.button {
  color: var(--selection-background);
  background-color: var(--default-
foreground);
  border: 2px solid var(--dark-
```

⁷This very documentation also is sourced from the exact same colorscheme.

```
foreground);
  height: 3rem;
  border-radius: 6px;
  font-weight: bold;
  vertical-align: middle;
  margin: 0.5rem 0 0.5rem 0;
  padding: 1em;
  font-family: "JetBrains Mono",
monospace;
}
```

Every button is derived from the one above. The text is aligned in the center both horizontally and vertically; its weight set to bold. Then there are some margin and padding to help the user see better⁸.

A notable exception to the buttons colorscheme is the `logout` button:

```
.logout {
  background-color: var(--variables);
  font-weight: bolder;
  color: var(--lighter-background);
}

.logout:hover {
  background-color: var(--data-types);
}
```

Both the `background-color`, `font-weight` and `color` are different, to further imply that the `logout` button is different from the others (`upload track`, `create playlist...`).

8.3 Containers

The first container the user sees is the Login one, which shares its design with Register and the track player:

```
.center-panel {
  width: 300px;
  background-color: var(--lighter-
background);
}
```

⁸There will be later an exception.

```
border: 1px solid var(--dark-foreground);
padding: 3rem;
text-align: center;
}
```

An important aspect of login and register is their horizontal bar:

```
hr {
  display: block;
  height: 1px;
  border: 0;
  border-top: 1px solid var(--light-background);
  margin: 1em 0;
  padding: 0;
}
```

which is not used in the track player.

A basic function of a Playlist Manager is being able to display all the playlists and tracks of a given user. To achieve that, we opted for a classic layout composed of a top and bottom navigation bars and a main, central section.

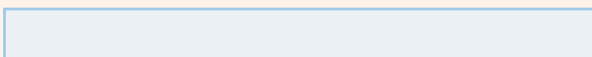
```
.nav-bar {
  width: 100%;
  margin: 0;
  display: flex;
  flex-wrap: wrap;
  align-content: space-around;
  justify-content: center;
  align-items: center;
  gap: 1rem;
}
```

The navigation bar is the same both above and below. It's a flex container because it's important to have a flexible container for the main-title (e.g. "All Playlists") and the buttons (with a variable number between screens).

The layout is computed as follows:

title	spacer	button	button	logout
-------	--------	--------	--------	--------

so we created the *spacer* element:



```
.spacer {
  flex-grow: 1;
}
```

which takes all the space available.

Next, the tracks and playlists containers.

```
.items-container {
  width: 100%;
  display: grid;
  grid-template-columns: 1fr 1fr 1fr 1fr 1fr;
  align-content: baseline;
  justify-content: center;
  gap: 1rem;
  padding: 1rem 0 1rem 0;
}

.single-item {
  display: flex;
  flex-wrap: nowrap;
  background-color: var(--light-background);
  border: 2px solid var(--data-types);
  border-radius: 5px;
  color: var(--lighter-background);
  padding: 1rem;
  height: 150px;
  font-family: "JetBrains Mono", monospace;
  font-weight: 700;
  text-align: left;
  align-content: end;
  align-items: end;
  justify-content: space-between;
}

.single-item:hover {
  background-color: var(--variables);
  cursor: pointer;
}
```

According to project the specifications ([Section 2](#)), there must be at most 5 tracks per page: we opted for a CSS grid. This works well along with the body previously set because the grid can expand and shrink its items accordingly.

As per the navigation bar, the single items are themselves flexible boxes. The difference lies in the fact they are not allowed to wrap – one might ask: why not, since the tracks must list

*both track title and album title? because we handle that line break manually with the
 tag.*

Last but not least, the errors.

```
.error{
  color: var(--variables);
  padding-top: 0.5rem;
  width: 100%;
  display: flex;
  flex-wrap: wrap;
  align-content: space-around;
  justify-content: center;
  align-items: center;
}
```

When the User tries to do something forbidden – adding duplicate tracks, creating a duplicate playlist... – an error will appear. It's exclusively used in the modal and due to how it's spaced it requires the flex display.

The simpler implementation is sql-error:

```
.sql-error {
  color: var(--variables)
}
```

which is used during registration.

8.4 Modal

RIA version The modals in the RIA project are dynamically generated via Javascript only when needed and removed when the screen is changed. This section has been written with the HTML version in mind.

Finally, undoubtedly the most difficult CSS component in this project to comprehend is the modal, which is a dialog window created entirely with CSS.

A complex element, it can be broken in multiple parts:

- The window

```
.modal-window {
  position: fixed;
  background-color: rgba(255, 255, 255,
0.25);
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  z-index: 999;
  visibility: hidden;
  pointer-events: none;
  transition: all 0.5s;
}
```

it's hidden by default, but once it's invoked it must be above everything – this is handled by the z-index property. Its position must be fixed, since it's not a movable window; also it can't be targeted by cursor: pointer-events are none. Another key aspect is the background color: in order to make it stand from its background, a slight blurred white is needed:

- The target, when the user presses a button that launches the modal (e.g. Upload Track)

```
.modal-window:target {
  visibility: visible;
  opacity: 1;
  pointer-events: auto;
}

.modal-window > div {
  width: 400px;
  position: absolute;
```

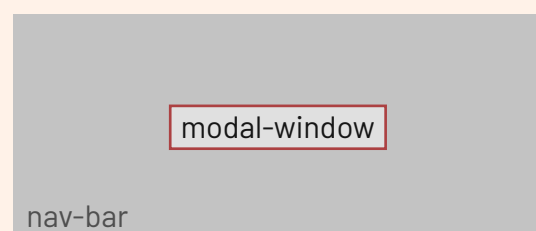


Figure 29: Modal representation.

```

top: 50%;
left: 50%;
transform: translate(-50%, -50%);
padding: 1em;
background: var(--lighter-background);
border: 2px solid var(--variables);
}

```

once the modal has been invoked, its visibility must be switched to visible and opacity to 1. The child element div of the window must at the center of screen, both horizontally and vertically: this is managed with the top, left and translate properties.

- The close button

```

.modal-close {
  color: var(--lighter-background);
  background-color: var(--variables);
  border-radius: 5px;
  position: absolute;
  top: 2%;
  right: 2%;
  cursor: pointer;
  padding: 0.2rem;
  font-size: 0.8rem;
  font-weight: bold;
  text-align: center;
  text-decoration: none;
}

.modal-close:hover {
  color: black;
}

```

as stated previously, the modal-close button is an exception to the button rule. It's considerably smaller than the others, the cursor is immediately pointer. Its position is computed on the modal-window, from above right.

- The dropdown menus

```

select:invalid {
  color: #505050;
}

```

this pseudoclass causes the color in the placeholder in dropdown menus (Year, Genres) to be gray, as a regular placeholder should be⁹.

⁹Otherwise it would have been black as the text, which is not aesthetically pleasant.

8.5 TS Sidebar

This components exists on in the RIA project, where we implemented a lateral bar that contains three menus: Homepage, Playlist and Track. It effectively acts as an upgraded version of the bottom navigation bar of the HTML project.

```

.side-bar {
  height: 100%;
  width: 8rem;
  position: fixed;
  z-index: 1;
  left: 0;
  top: 0;
  background-color: var(--light-
background);
  overflow: hidden; /* disable
scrolling */
  display: flex;
  flex-direction: column;
  justify-content: start; /* buttons
vertical alignment */
}

```

It always sits in the left side, with a 100% height. Its position can't be changed and neither its content. The background acts as contrast with the playlist tracks buttons.

Then the styling the menus.

```

.side-bar .entry-button {
  margin: 1rem;
  padding: 1rem;
  display: flex;
  flex-direction: column;
  align-items: center;
}

.entry-button:hover {
  cursor: pointer;
}

.entry-button .icon {
  width: 100%;
  background-color: transparent;
  border-color: blue;
  border-width: 1cm;
}

.entry-button .title {

```

```
font-family: "JetBrains Mono",  
monospace;  
font-size: 100%;  
}
```

The menus are of *entry-button* class, which can exist only in a *side-bar* container (as seen in the first line of the previous source code extract). They are horizontally centered.

The class *icon*, *.title* can exist only inside the *entry-button* – thus they can exist only inside the *side-bar* class.

And lastly, legacy code.

```
.side-bar .close-button {  
  color: var(--lighter-background);  
  background-color: var(--variables);  
  border-radius: 5px;  
  font-family: "JetBrains Mono",  
  monospace;  
  position: absolute;  
  top: 0.5rem;  
  right: 2rem;  
}  
  
.side-bar .close-button:hover {  
  color: var(--dark-foreground);  
  cursor: pointer;  
}
```

Originally the side bar was supposed to open and close, like a lateral menu as can be seen in many mobile apps – that's why we needed a close button. For the current implementation, it's not used.

A

Cut content

During the development, we had many ideas and thought about ways to implement them – however, due to time and work restrictions, some features didn’t make it to the final release. They can be categorized in features, optimizations and code cleanup.

A.1 Features

In regards to the features, we wanted to implement **Next/Previous** buttons for the Playlists too, to make the application behave in a more coherent way: according to the submission, only the tracks in the playlist must implement it. Following this cohesion, the same can be said about the **Delete** functionalities: initially, along with the creation of a track/playlist we wanted to add a delete option – if you add something, you might want to remove it at a later date.

To comply with best practises, since the application does offer a logout button – which is an added feature, not required – it should also provide a user profile window; or at least welcome the user upon login.

JS For the JavaScript project – which is not correct to call like that, since we used TypeScript – there were plans to implement a localization function, similar to how the HTML project works. It would have been a parser for the .properties files already created: they would have been recycled. The most ambitious idea was, however, to deploy the JavaScript project to Github pages. This is not possible with thymeleaf since it needs a server running at all times, but with JavaScript running in the client... it was perfect. To access the database, we planned to use SQL.js [13] and SQLite [14].

CSS frameworks We all know CSS is awesome and very powerful, however, as is the case with many technologies, its usage in a raw form is often neglected: as no one dares to write in plain $\text{T}_{\text{E}}\text{X}$ because $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ exists, software like Hibernate (more on this later) abstract the SQL from the developer, the

same applies to CSS. In the wild there are many frameworks – Tailwind-CSS, Sass just to name a few. We wanted to have our fair share and use Bulma [15]; in the end, we wrote everything ourselves.

A.2 Optimizations

The OG database The first database implementation was created with a different logic than the one we ended up with. We thought that the tracks were a common pool, similarly to all the tracks of a streaming service, and then each user could select some among them. In this way, if a user wanted to upload a track but it had already been uploaded by someone else, the server would have just linked that track to the current user – the reason behind was to optimize track storage and forbid duplicates.

To support such a logic, there used to exist a track and a user_tracks tables. This allowed us to perform some further optimization – we had thought about creating a trigger in the database: it would have deleted a track from the corresponding table if that track wasn’t associated with at least a user (in the ER diagram it was a weak entity, that is it existed only as long as there was a link).

The issue was quite simple... the submission *didn’t specify* this; instead, every user has their tracks. They can be the same exact files of another user – pretty much like how a cloud service behaves. And that’s how the project works. Still, we couldn’t just let the user upload track at will without some checks. And that trigger evolved in the path and image checksums attributes.

Missing hashing One could argue: “You went above and beyond to ensure the user doesn’t upload the same exact file multiple times, yet you don’t even hash the passwords”. And you would be right. We wanted to do that by leveraging the power of Password4J [16], but once again the specification didn’t ask for it and so... we had other features to work on.

Connection pooling Another important optimization technique is [connection pooling](#): to put it simply, instead of opening every time a new connection to the database – which is the most computationally expensive operation database-wise – there is a pool of reusable connections, that are always open. This way, the database is accessed once and then the queries are performed by those same connections. Our library of choice was HikariCP [17].

ORM The proper (or *elegant*) way to interact with the database isn't by directly writing raw SQL code but by using APIs written for this very reason. There are many examples in web technologies – such as jQuery – though for the Java programming language, pioneer of the Object-Oriented Programming paradigm, there is a more potent concept: [Object Relational Mapping \(ORM\)](#). As the name suggests, a relational object is mapped to a Java object. By using Hibernate [18] a table could be mapped 1:1 to a class and its attributes: every query – select, insert, delete... – can be performed through it with commits, transactions and so on.

Springtrap Probably the saddest turn back was not being able to use the Spring Boot framework [19], which is commonly used. It's a framework to create production-level applications: as such, it's definitely a must, whatever the future may hold. Also, during research of how thymeleaf operates, it was basically *always* paired with Spring Boot.

Caching If a playlist doesn't change and the user requests it again, there should be no reason to make another GET request to the server. Instead, it could be cached. We didn't search for specific software to do so: it remained a idea.

- Thymeleaf can process the parameters value directly, effectively bypassing the context setting: this could potentially reduce the boilerplate and make for a more elegant code
- Since the Record classes are immutable, in order to get them to work without settings all nulls there have been some workarounds what could be rewritten and polished
- The homepage.ts file is not at all easily navigable – we often found ourselves scrolling up and down because it's 1000+ pages long – though it's thoroughly commented: we wanted to dismember it in separate files (homepage.ts, playlist.ts, playerpage.ts) to fix this, but due to the use of global variables and time we weren't able to do it
- The createModal function was being created in parallel with the track-reorder modal in different branches, thus the latter couldn't have been generated with the former and now there is some code duplication; this could have happened either way because this particular modal is quite different from the others

A.3 Code cleanup

The last scrapped ideas were all about some refactoring here and there:

Bibliography

- [1] L. Mädje, M. Haug, and The Typst Project Developers, *Typst*. [Online]. Available: <https://github.com/typst/typst>
- [2] Louis Heredero, *chronos (typst package)*. [Online]. Available: <https://typst.app/universe/package/chronos>
- [3] "Java Development Kit." [Online]. Available: <https://openjdk.java.net/>
- [4] "Apache Maven." [Online]. Available: <https://maven.apache.org/>
- [5] "Apache Tomcat." [Online]. Available: <https://tomcat.apache.org/>
- [6] "Thymeleaf." [Online]. Available: <https://www.thymeleaf.org/>
- [7] "MariaDB." [Online]. Available: <https://mariadb.org/>
- [8] Vittorio Robecchi, *Web Technologies IntelliJ Guide @ PoliMi*. [Online]. Available: <https://github.com/VictuarVi/wt-intellij-guide>
- [9] NoCopyrightSounds. [Online]. Available: <https://ncs.io/>
- [10] "Record Classes." [Online]. Available: <https://docs.oracle.com/en/java/javase/17/language/records.html>
- [11] "Part interface." [Online]. Available: <https://jakarta.ee/specifications/servlet/6.1/apidocs/jakarta.servlet/jakarta/servlet/http/part>
- [12] *Tinted Theming*. [Online]. Available: <https://github.com/tinted-theming/home>
- [13] *sql.js*. [Online]. Available: <https://github.com/sql-js/sql.js/>
- [14] "SQLite." [Online]. Available: <https://sqlite.org/>
- [15] "Bulma." [Online]. Available: <https://bulma.io/>
- [16] *Password4j*. [Online]. Available: <https://github.com/Password4j/password4j>
- [17] *HikariCP*. [Online]. Available: <https://github.com/brettwooldridge/HikariCP>
- [18] "Hibernate." [Online]. Available: <https://hibernate.org/>
- [19] "Spring Boot." [Online]. Available: <https://spring.io/>