Web Technologies Project @ PoliMi, 2025

Creating a Playlist Manager with Thymeleaf & TS

Manuel Zani
zani.manuel328@gmail.com
https://github.com/manuel3053

Contents

1 Original submission (in Italian) 6
1.1 Versione HTML pura . . . 7
1.2 Versione RIA . . . 7

2 Project submission breakdown 10
2.1 Database logic . . . 11
2.2 Behaviour . . . 11
2.3 RIA version . . . 13
2.4 Added features . . . 13

3 SQL database schema 16
3.1 Overview . . . 17
3.2 The tables . . . 17

4 Codebase overview 20
4.1 Components . . . 23
4.2 Backend . . . 23
4.3 Frontend . . . 23
4.4 RIA subproject . . . 23

5 Sequence diagrams 24
5.1 Login sequence diagram . . . 25
5.2 Register sequence diagram . . . 27
5.3 HomePage sequence diagram . . . 29
5.4 Playlist sequence diagram . . . 31
5.5 Track sequence diagram . . . 33
5.6 UploadTrack sequence diagram . . . 34
5.7 CreatePlaylist sequence diagram . . . 36
5.8 Logout sequence diagram . . . 37
5.9 AddTracks sequence diagram . . . 37
5.10 TS GetTracksNotInPlaylist sequence diagram . . . 38
5.11 TS TrackReorder sequence diagram . . . 39
5.12 TS GetUserTracks sequence diagram . . . 40
5.13 TS Event: Login . . . 41
5.14 TS Event: Register . . . 42
5.15 TS Event: Logout . . . 43
5.16 TS Event: Access HomeView . . . 44

5.17 TS Event: Access PlaylistView 45
5.18 TS Event: Access TrackView 46
5.19 TS Event: Upload Track modal 47
5.20 TS Event: Reorder modal 48
5.21 TS Event: Create Playlist modal 49
5.22 TS Event: Add Track modal 50

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 subprojects: a (pure) **HTML version**, which is structured as a series of separate webpages; and a **RIA version** (TS)[1], which is structured as a multi-page webapp. The functionalities are quite the same. For more information about the requirements for each version see Section 2.

Tools To create the project, the following technologies have been used:

- **Java**, for the backend server with servlets leveraging the **Springboot framework**
- **Typescript** for for the RIA one
- **Thymeleaf**, a template engine, for the HTML version
- **MariaDB** instead of MySQL, since it's an open source fork of MySQL

Credits

ringrazia vittorio

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

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'abum 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'abum 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 RIA

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 personalizato 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 via **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 3.

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

Figure 1: ER diagram (HTML).

- If a playlist contains more than 5 tracks, there are available commands to see the others (in blocks of five)

Playlist tracks navigation If the PLAYLIST PAGE:

1. Shows the first group and there are subsequent ones, a NEXT button appears on the right side of the row
2. Shows the last group and there are precedent ones, a PREVIOUS button appears on the left side of the row that allows to see the five preceding tracks
3. Shows a block of tracks and there are both subsequent and preceding ones, then on left and the right side appear both previous and next buttons

Track creation The PLAYLIST PAGE includes a form that allows to add one or more tracks to the current playlist, if not already present. This form acts in the same way as the playlist creation form.

After adding a new track to the current playlist, the application refreshes the page to display the first block of the playlist (the first 5 tracks). Once a user selects the title of a track, the PLAYER PAGE shows all of the track data and the audio player.

Figure 2: IFML diagram (HTML).

2.3 RIA 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
- Every user interaction is managed without completely refreshing the page, but instead it asynchrounosly 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 desidered 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**.

2.4 Added features

- Logout function (Section 5.8)
- Subscribe function (Section 5.2)

Figure 3: ER diagram (RIA).

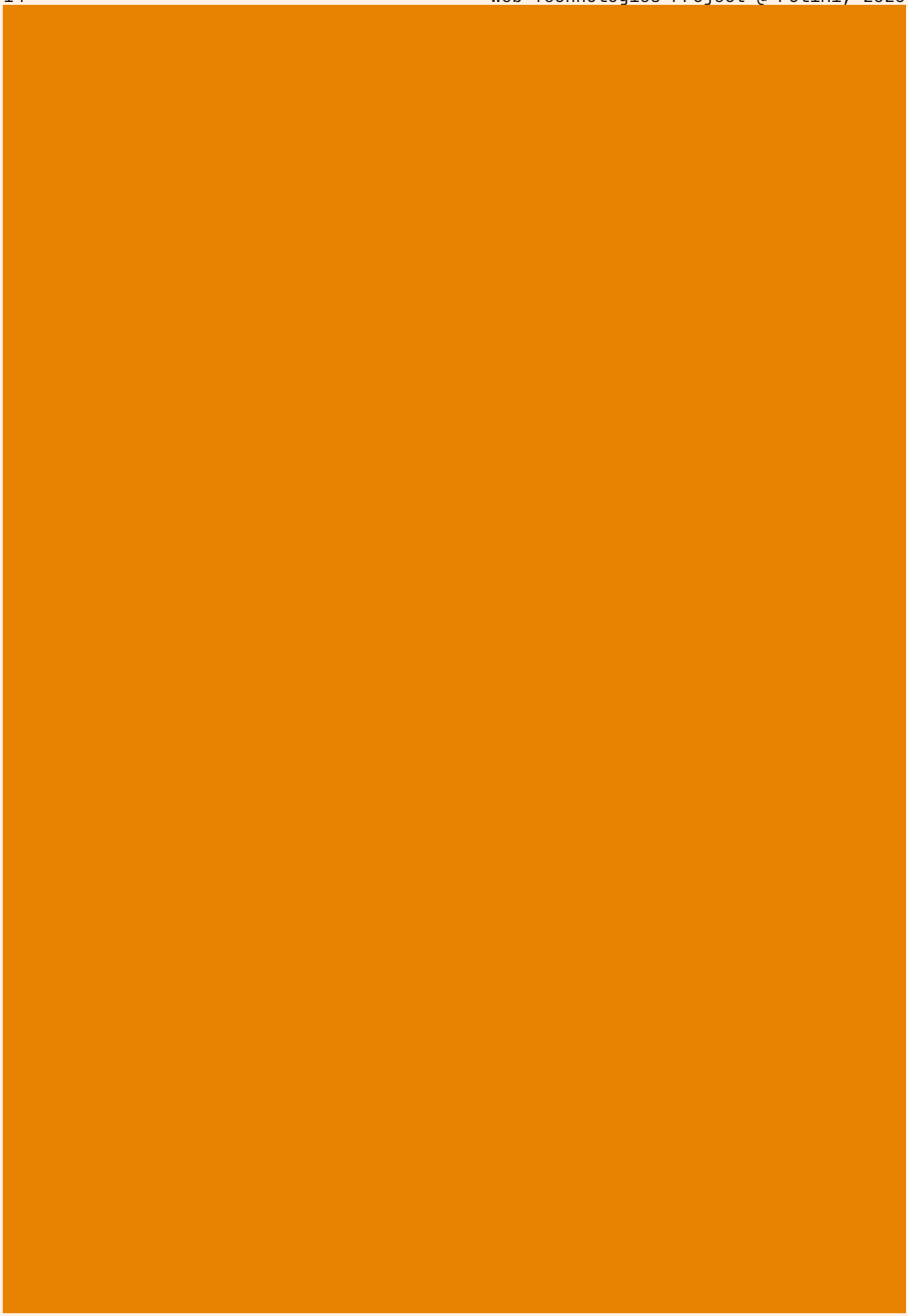


Figure 4: IFML diagram (RIA).

3

SQL database schema

3.1 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 by adding an optional position column in `track` and a custom order flag in `playlist`

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

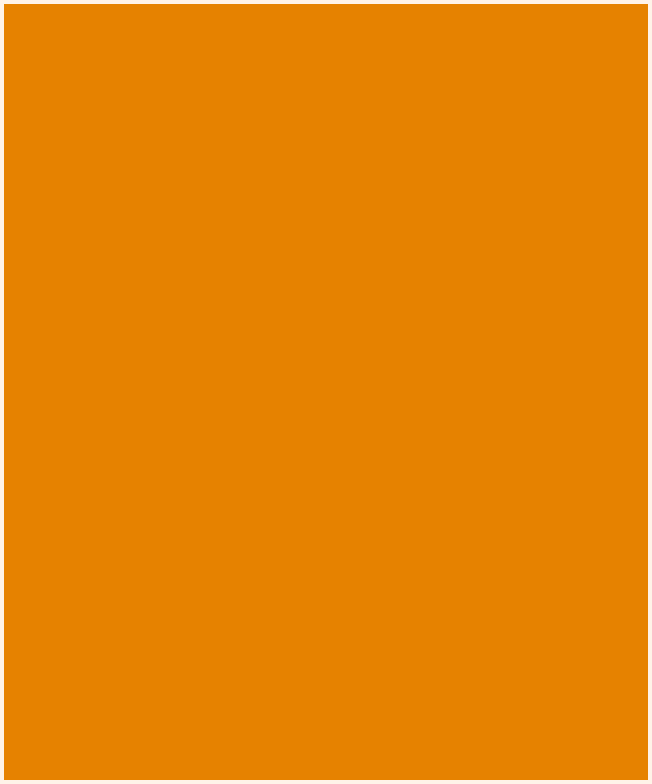


Figure 5: UML diagram.

3.2 The tables

- `user` table

```
CREATE TABLE user
(
    id       integer AUTO_INCREMENT,
    username VARCHAR(64) not null,
    password VARCHAR(64) not null,
    name     VARCHAR(64) not null,
    surname  VARCHAR(64) not null,
    PRIMARY KEY (id),
    UNIQUE KEY `username` (`username`),
    UNIQUE (username)
);
```

it is quite staightforward and standard. Apart from the `id` attribute, which is the primary key, the only other attribute that has a unique constraint is `username`.

- `track` table

```
CREATE TABLE track
(
    id           integer      AUTO_INCREMENT,
    loader_id    integer      not null ,
    file_path    VARCHAR(512) not null,
    image_path   VARCHAR(512) not null,
    title        VARCHAR(64)  not null,
    author       VARCHAR(64)  not null,
    album_title  VARCHAR(64)  not null,
    album_publication_year integer not null,
    genre        VARCHAR(64)  not null,
    position     integer,

    PRIMARY KEY (id),
    UNIQUE (loader_id, title, author,
album_title, album_publication_year),
    FOREIGN KEY (loader_id) REFERENCES user
(id) ON UPDATE CASCADE ON DELETE CASCADE
);
```

there unique constraint on `loader_id, title, author, album_title, album_publication_year` is to make sure that the user doesn't load duplicates (there are almost all the attributes inside it to address the unlikely situation where an almost identical track is loaded).

The `loader_id` foreign key references the `id` of the user and if it is removed, his tracks are also removed.

- `playlist` table

```
CREATE TABLE playlist
(
    id            integer AUTO_INCREMENT,
    title         VARCHAR(64) not null,
    creation_date DATETIME not null DEFAULT
NOW(),
    author_id     integer     not null,
    custom_order  BOOL        not null,

    PRIMARY KEY (id),
```

```
    UNIQUE (title, author_id),
    FOREIGN KEY (author_id) REFERENCES user
(id) ON UPDATE CASCADE ON DELETE CASCADE
);
```

The `creatione_date` attribute defaults to the today's date; and there is also the unique constraint on `title, author_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
(
    id          integer AUTO_INCREMENT,
    playlist_id integer not null,
    track_id    integer not null,

    PRIMARY KEY (id),
    FOREIGN KEY (playlist_id)
    REFERENCES playlist (id)
    ON UPDATE CASCADE ON DELETE CASCADE,
    FOREIGN KEY (track_id)
    REFERENCES track (id)
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

This table represents the "Contained in" relation in the ER diagram (Figure 1). If a track or a playlist is removed their relationship is also removed.

4

Codebase overview

	HTML	*TS*
Entity	• Playlist • PlaylistTracks • Track • User	
DAO	• PlaylistDAO ▸ findByAuthorIdOrderByCreationDateAsc ▸ save ▸ findByAuthorIdAndId TS ▸ setCustomOrder TS • TrackDAO ▸ save ▸ findTrackByIdAndLoaderId ▸ getAllByUserIdSorted ▸ getPlaylistTracksGroup ▸ getAllNotInPlaylist ▸ getAllInPlaylist TS ▸ getAllInPlaylistCustom TS ▸ updatePosition TS	• PlaylistTracksDAO ▸ save ▸ getAllByPlaylistId • UserDAO ▸ save ▸ findByUsername
Controller	• FileController ▸ serveSafeFile • HomeController ▸ showPage ▸ addTrack ▸ addPlaylist • LoginController ▸ showPage • PlaylistConstroller ▸ showPage ▸ addTrackToPlaylist • SubscribeController ▸ showPage ▸ subscribe • TrackController ▸ showPage	• FileController ▸ serveSafeFile • AuthController ▸ login ▸ subscribe • PlaylistConstroller ▸ getPlaylists ▸ getPlaylistSizeByid ▸ addPlaylist ▸ addTracksToPlaylist ▸ setCustomOrder • TrackConstroller ▸ getTrackById ▸ getTracks ▸ getAllNotInPlaylist ▸ getAllInPlaylist ▸ getGenres ▸ addTrack
Springboot Config	• UserService ▸ loadUserByUsername • UserWithId • SecurityConfig ▸ securityFilterChain	• SecurityConfig ▸ securityFilterChain ▸ securityContextRepository ▸ authenticationManager

Table 1: Components comparison

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 2: Events & Actions.

Client side		Server side	
Event	**Controller**	**Event**	**Controller**
Index ⇒ Login form ⇒ Submit	`makeCall()` function	POST (`username`, `password`)	Login (servlet)
HomeView ⇒ Load	`HomeView.show()` (its invocation is done by the `MainLoader` class)	GET	Homepage (servlet)
HomeView ⇒ Click on a playlist	`loadPlaylist()`	GET (`playlistId`)	Playlist (servlet)
HomeView ⇒ Click on reorder button	`loadReorderModal()`	GET (`playlistId`)	Playlist (servlet)
Reorder modal ⇒ Save order button	`saveOrder()`	POST (`trackIds`, `playlistId`)	TrackReorder (servlet)
Create playlist modal ⇒ Create playlist button	`makeCall()`	POST (`playlistTitle`, `selectedTracks`)	CreateNewPlaylist (servlet)
Upload track modal ⇒ Upload track button	`makeCall()`	POST (`title`, `artist`, `year`, `album`, `genre`, `image`, `musicTrack`)	UploadTrack (servlet)
Sidebar ⇒ Playlist button	`playlistView.show()`	GET (`last selected Playlist`)	Playlist (servlet)
Sidebar ⇒ Track button	`trackView.show()`	GET (`last selected Track`)	Track (servlet)
Sidebar ⇒ HomePage	`homeView.show()`	GET (`user playlists`)	Homepage (servlet)
Logout	`makeCall()` function	GET	Logout (servlet)

Table 3: Events & Controllers (or event handlers).

4.1 Components

Introduction Both versions of the project present a lot of similarities but fot the TS version have been reorganized and some features were added

4.2 Backend

FAI IN MODO CHE LA TABELLA SIA POSIZIONATA PER IL BACKEND

4.3 Frontend

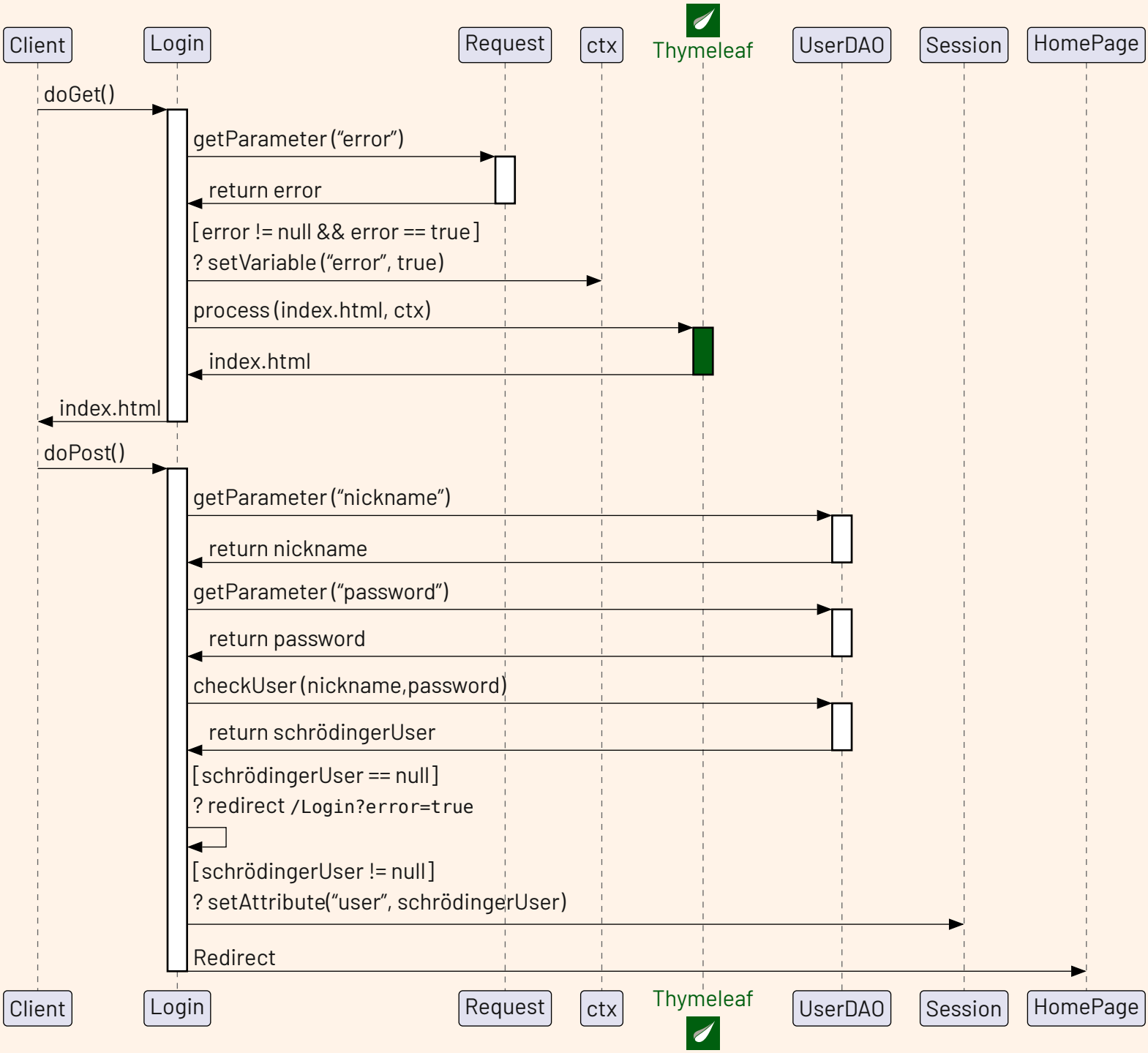
parla delle viste html e delle viste per typescript

4.4 RIA subproject

5

Sequence diagrams

5.1 Login sequence diagram



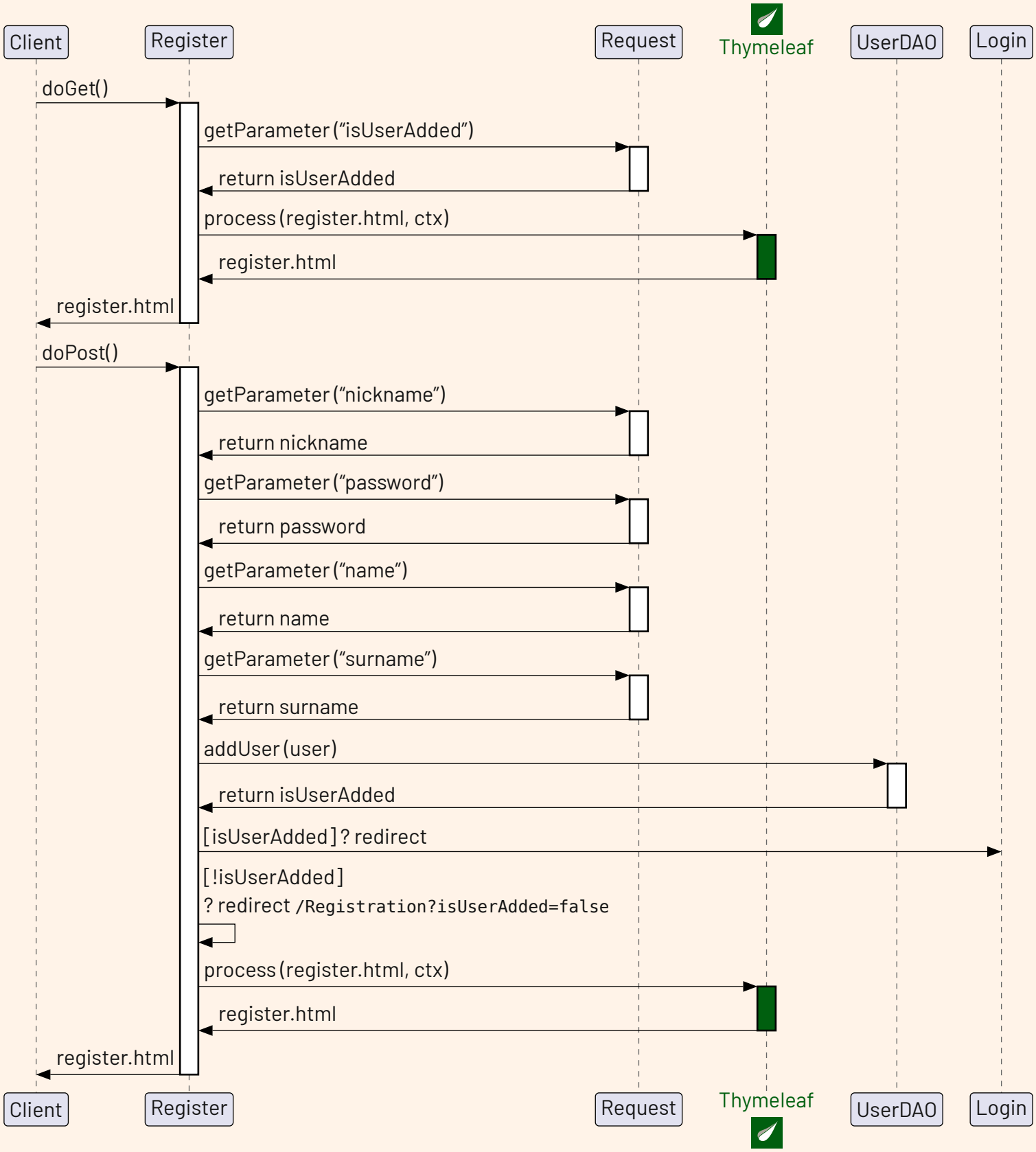
Comment

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 correct 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).

5.2 Register sequence diagram



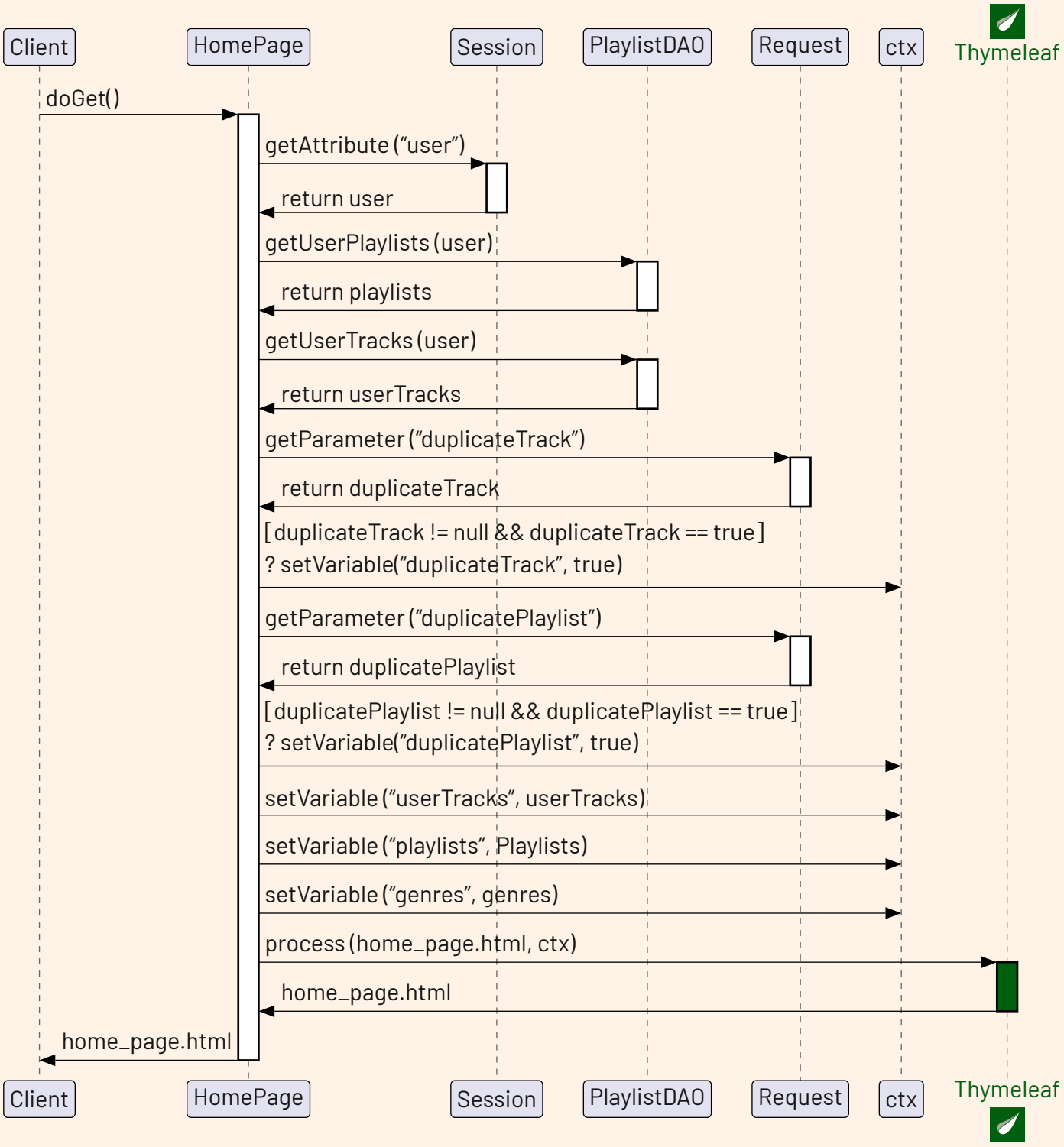
Comment

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` and there will be the redirection to the Login page.

Otherwise 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.

5.3 HomePage sequence diagram



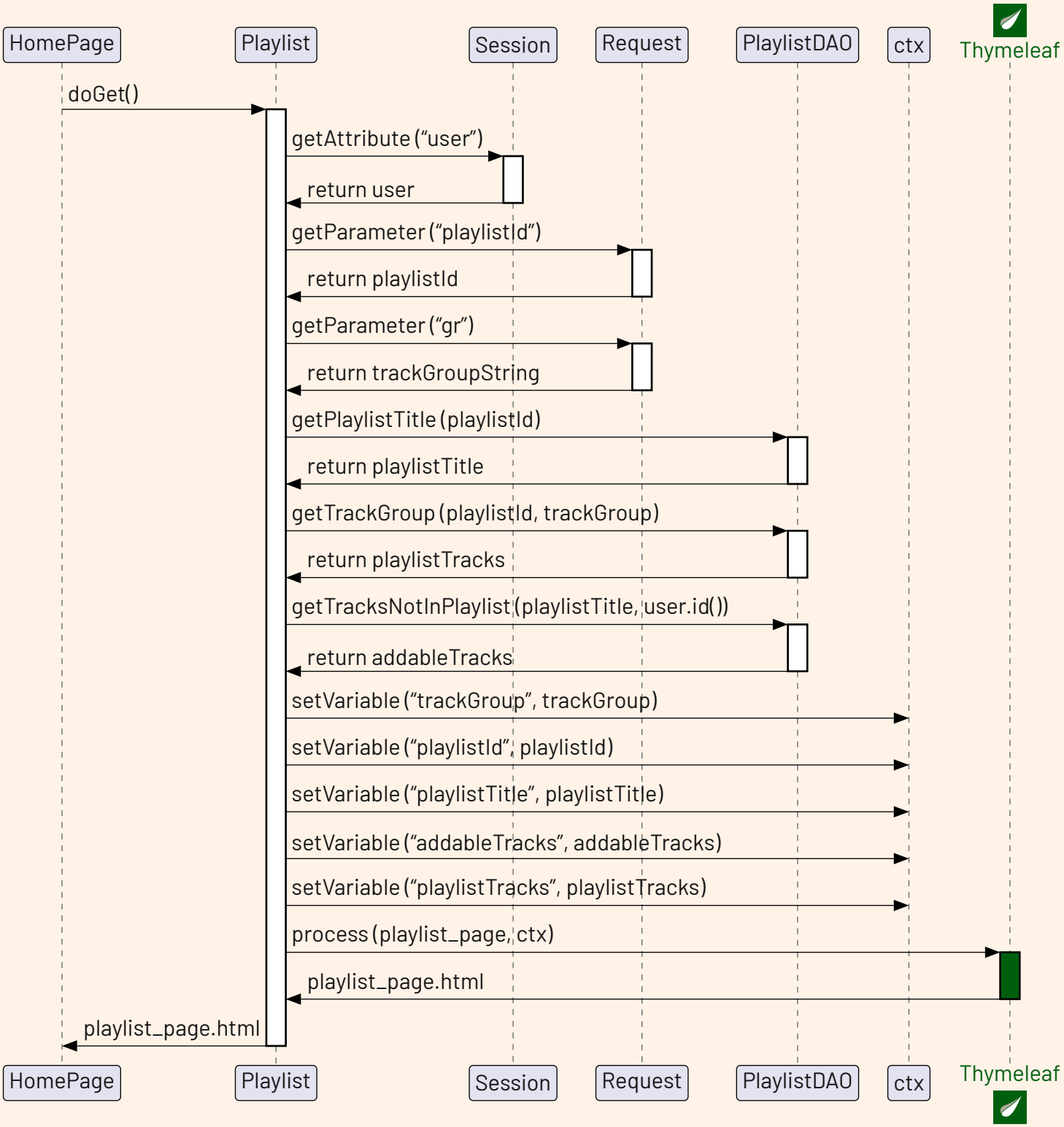
Comment

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()` method 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 is successfully added.

5.4 Playlist sequence diagram



Comment

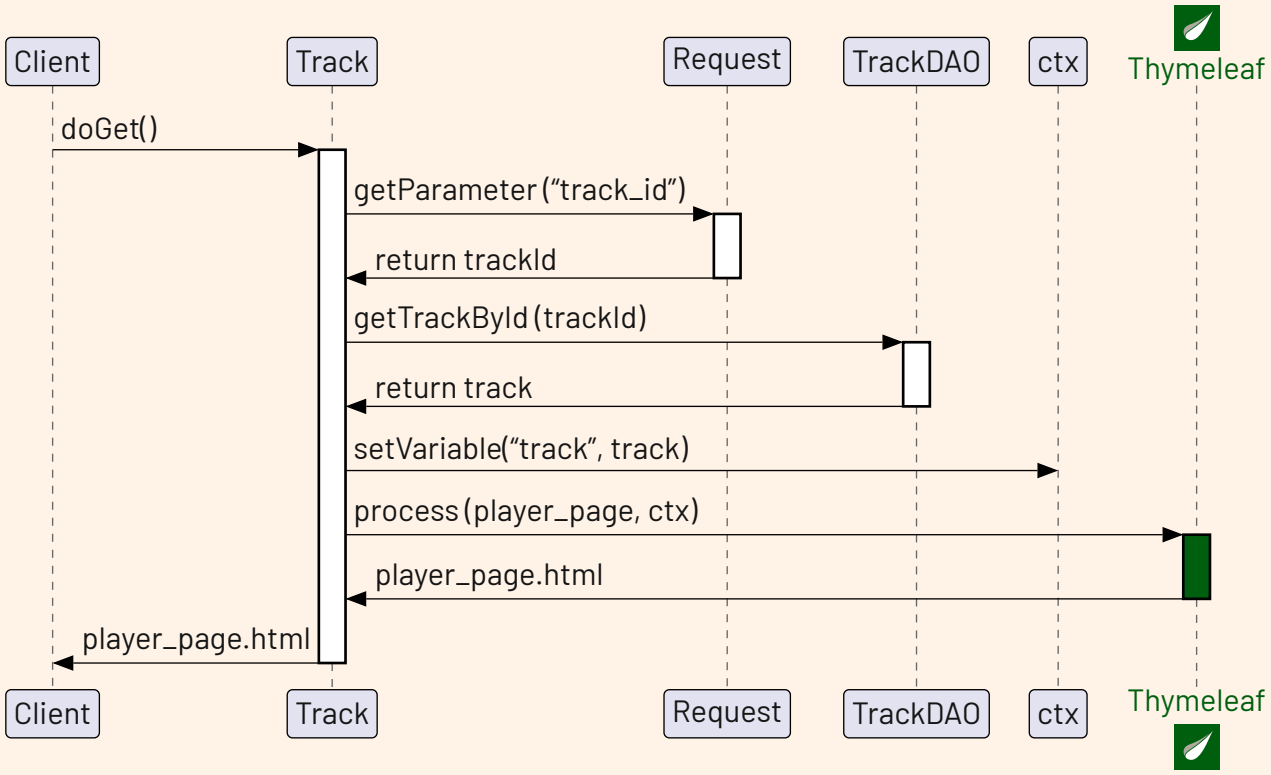
From the HomePage, the User is able to see all their playlists. By clicking on either one of them, the program redirects to the corresponding PlaylistPage, which lists all the tracks associated to that playlist.

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

Then those values are passed to the `getPlaylistTracks()` *method, 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 so, similarly to HomePage with the upload, the program fetches all tracks that can be added, *thats is the ones that are not already in the playlist, and displays them to a User via a dropdown menu (again similar to genres in HomePage).*

5.5 Track sequence diagram

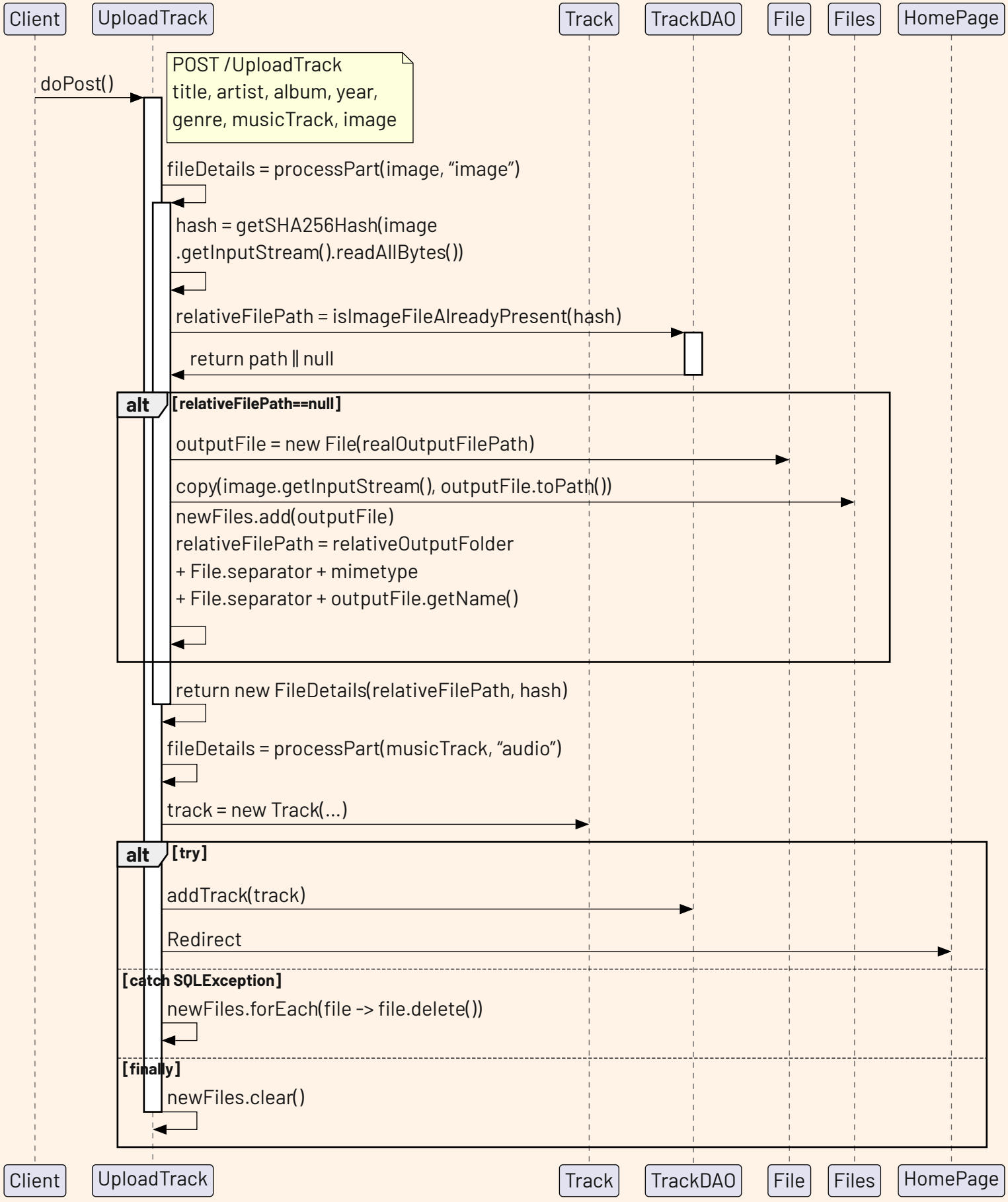


Comment

Once the program has lodead 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`.

5.6 UploadTrack sequence diagram



Comment

The User can upload tracks from the appropriate form in the HomePage (Section 5.3). When the POST request is received, the 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" part, and its expected MIME type. The latter does not need to be fully specified (i.e. the subtype can be omitted).

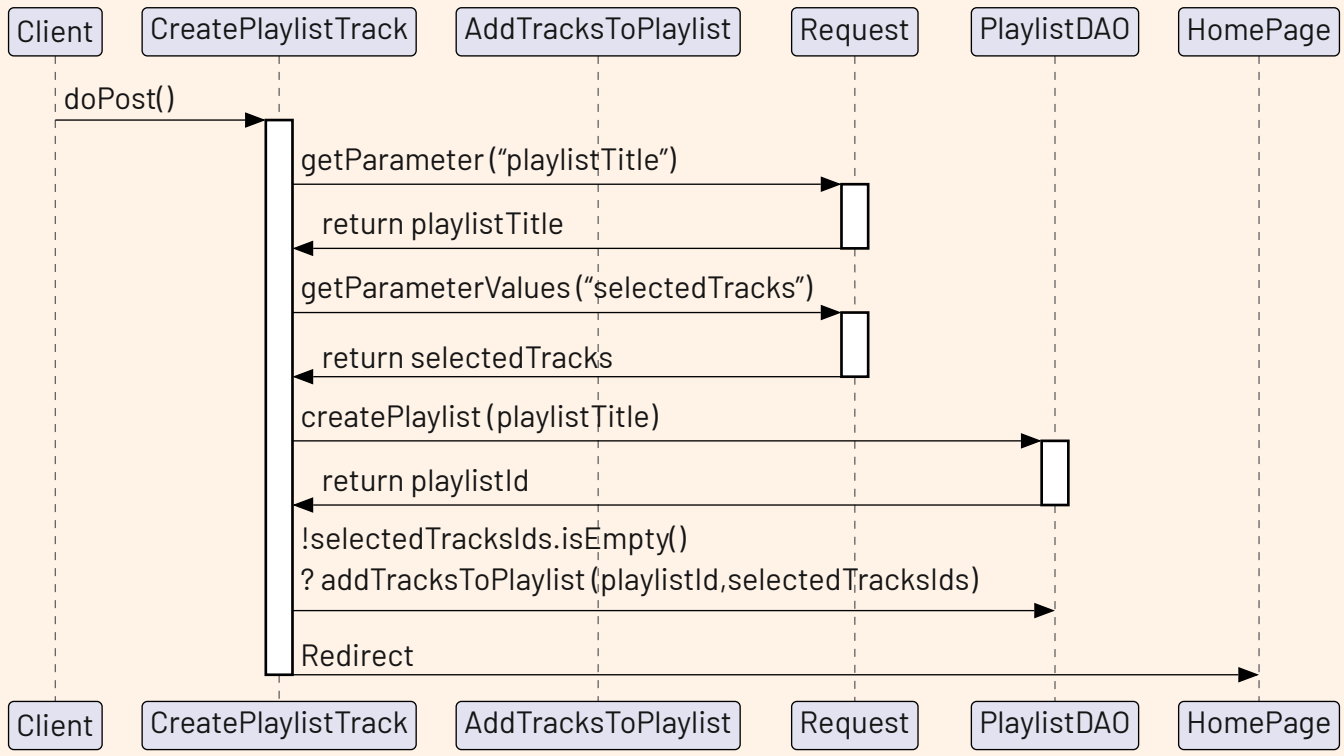
Before writing the file to disk, the method checks for duplicates of the file by calculating its SHA256 hash and querying the database with two methods: `isTrackFileAlreadyPresent()` and `isImageFileAlreadyPresent()` – present in TrackDAO.

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, thus 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.

5.7 CreatePlaylist sequence diagram

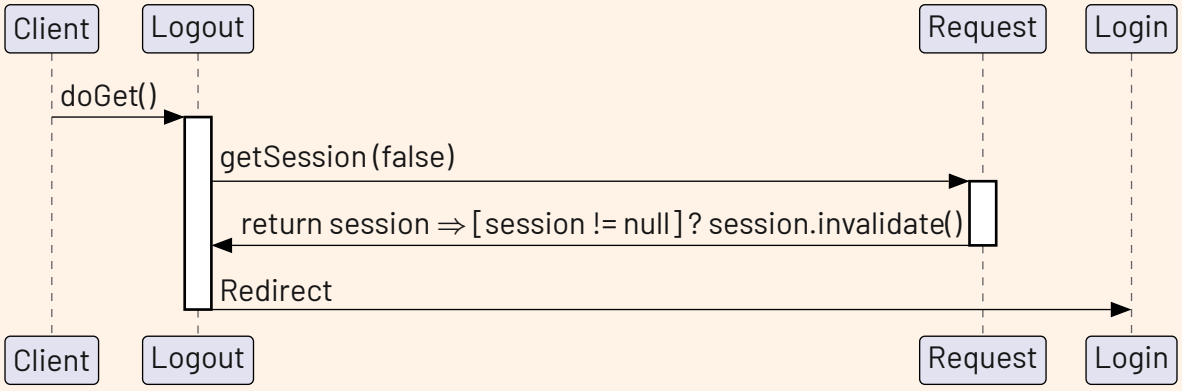


Comment

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 and parsing them with `Integer.parseInt()`.

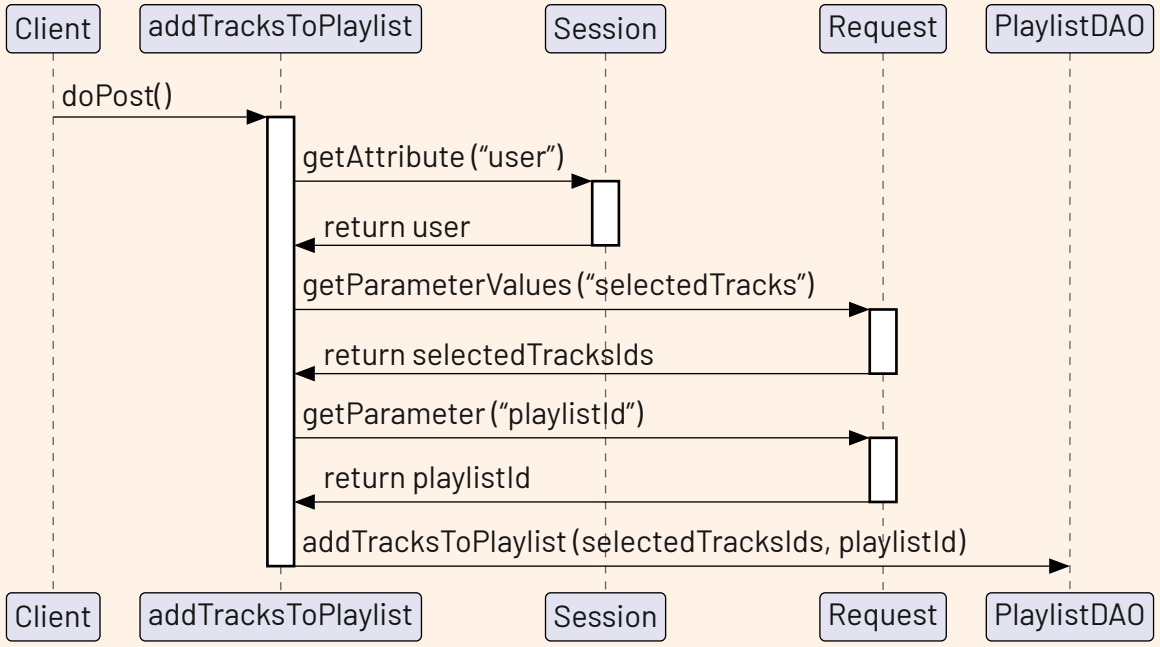
5.8 Logout sequence diagram



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.

5.9 AddTracks sequence diagram



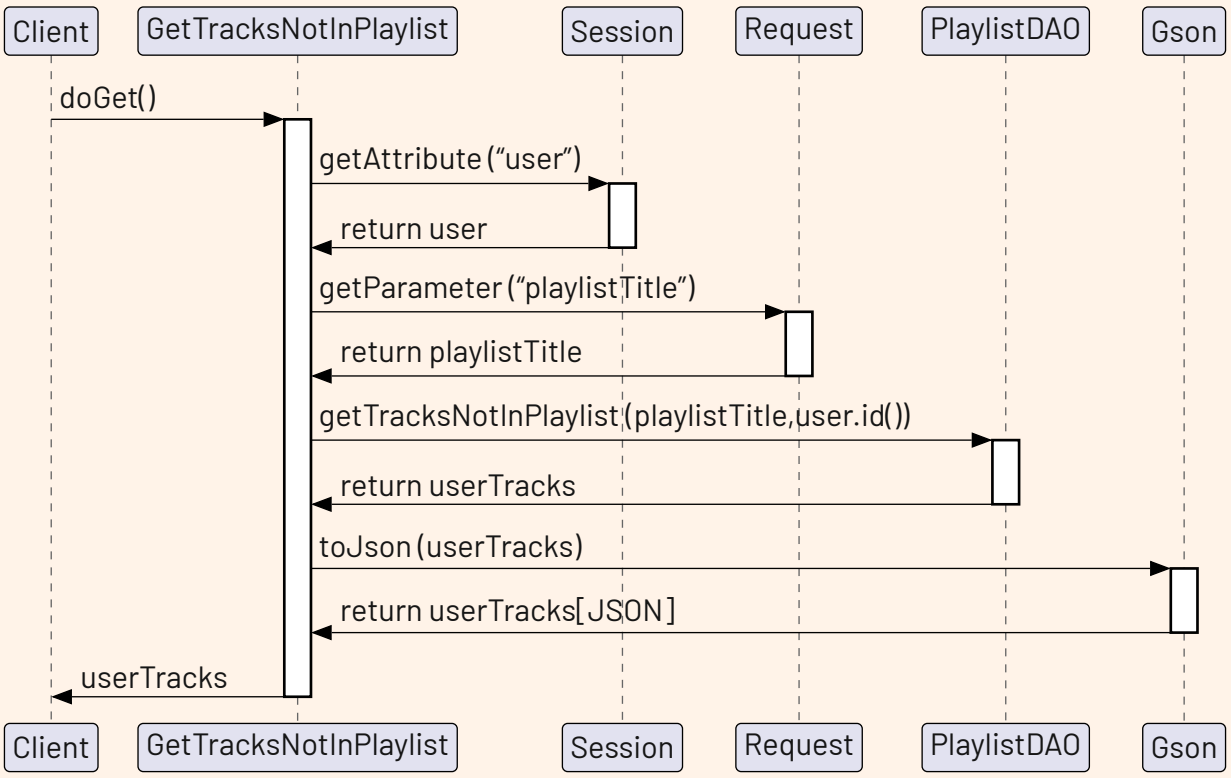
Comment

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 request.

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.

In the RIA subproject, the servlet responds with a `201` code instead of redirecting.

5.10 TS GetTracksNotInPlaylist sequence diagram



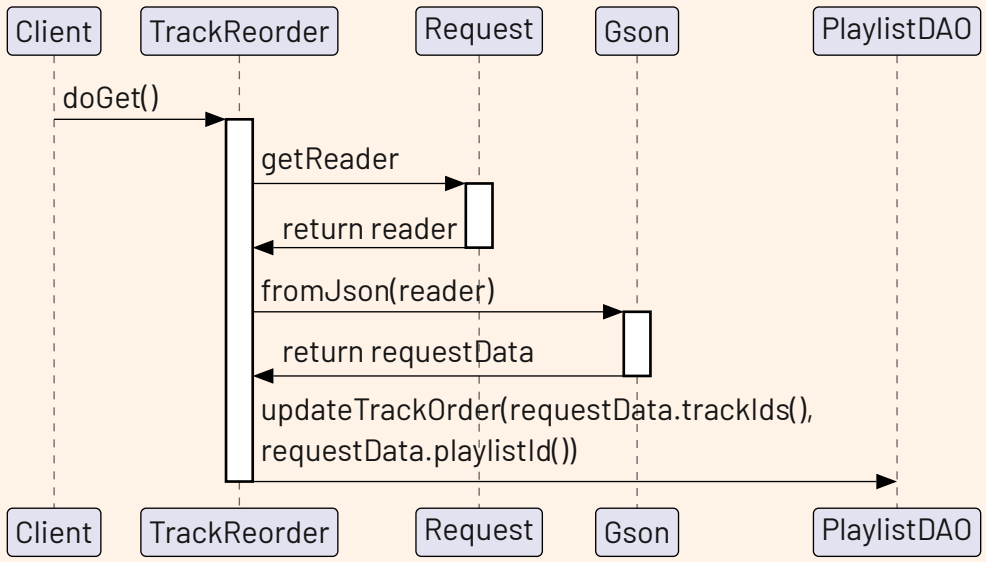
Comment

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.*

5.11 TS TrackReorder sequence diagram

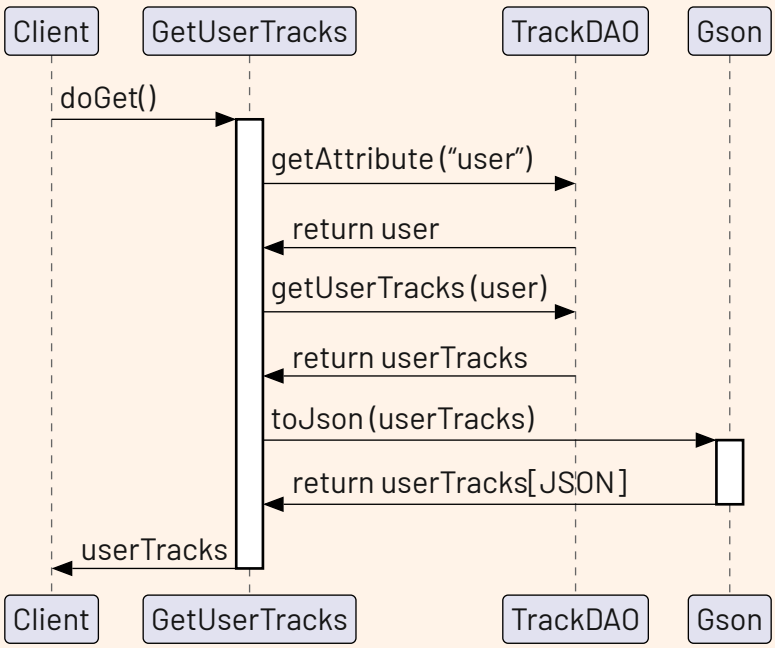


Comment

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.

5.12 GetUserTracks sequence diagram

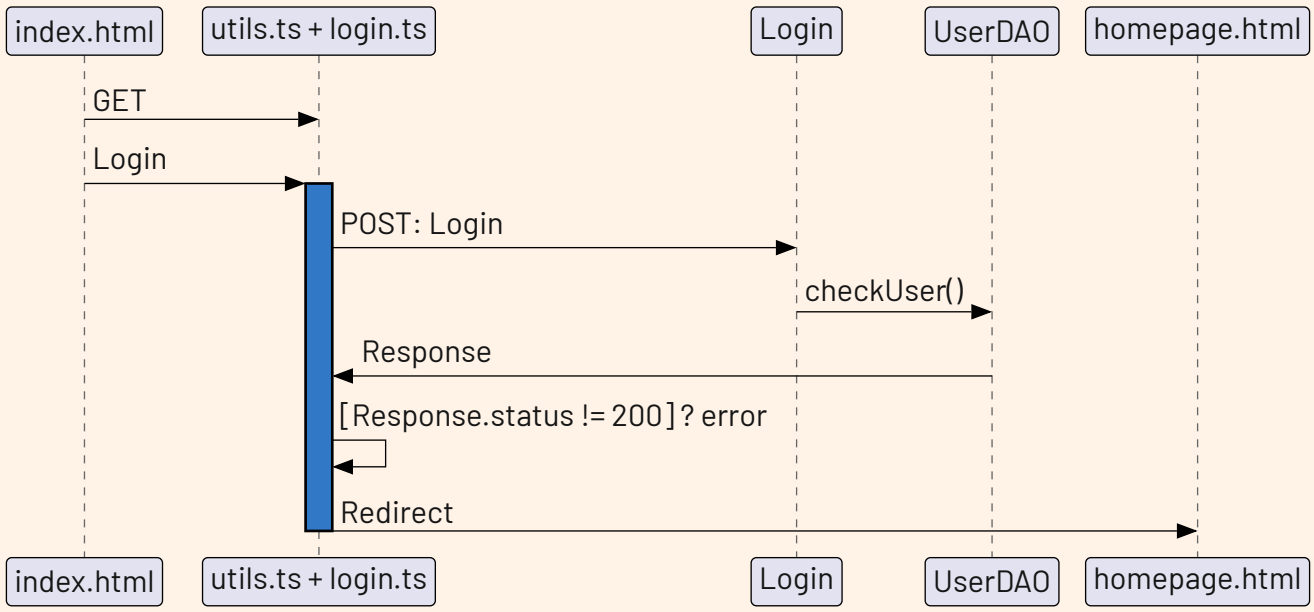


Comment

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.

5.13 TS Event: Login



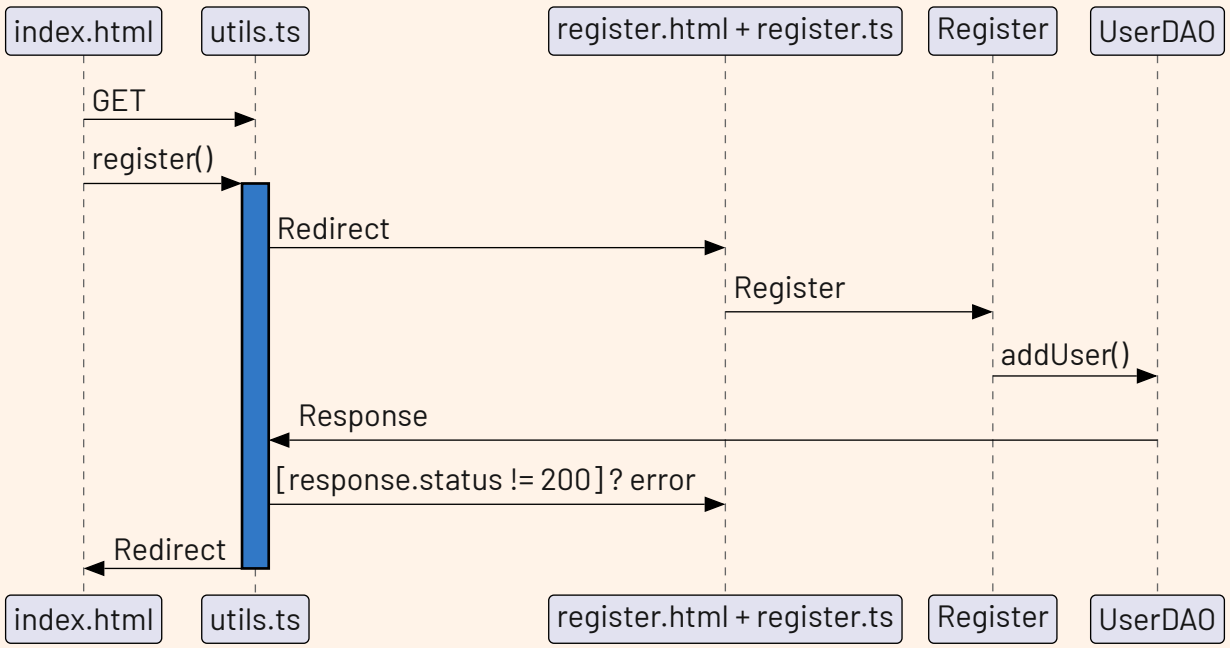
Comment

As the server is deployed the `index.html` requests the associated Javascript files (we use Typescript, but it transplies 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 5.1), 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.

5.14 TS Event: Register

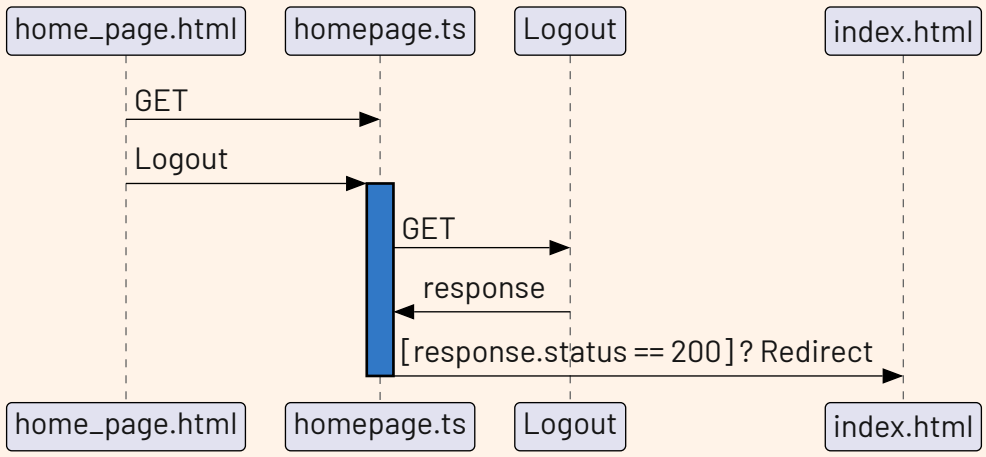


Comment

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 have been fetched, the User will be redirected to the `register.html` page.

From there, as seen in the Register sequence (Section 5.2), 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.

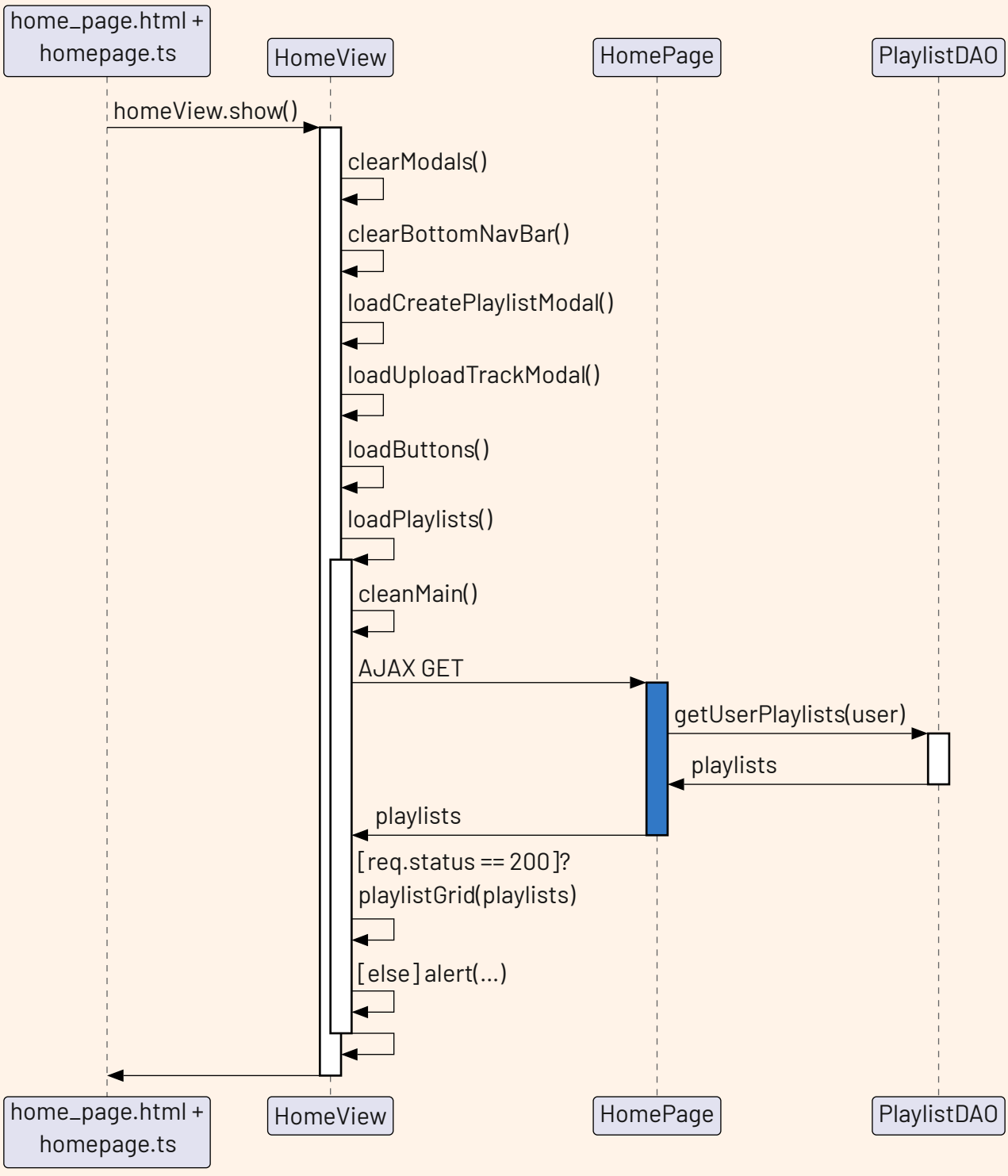
5.15 TS Event: Logout



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.

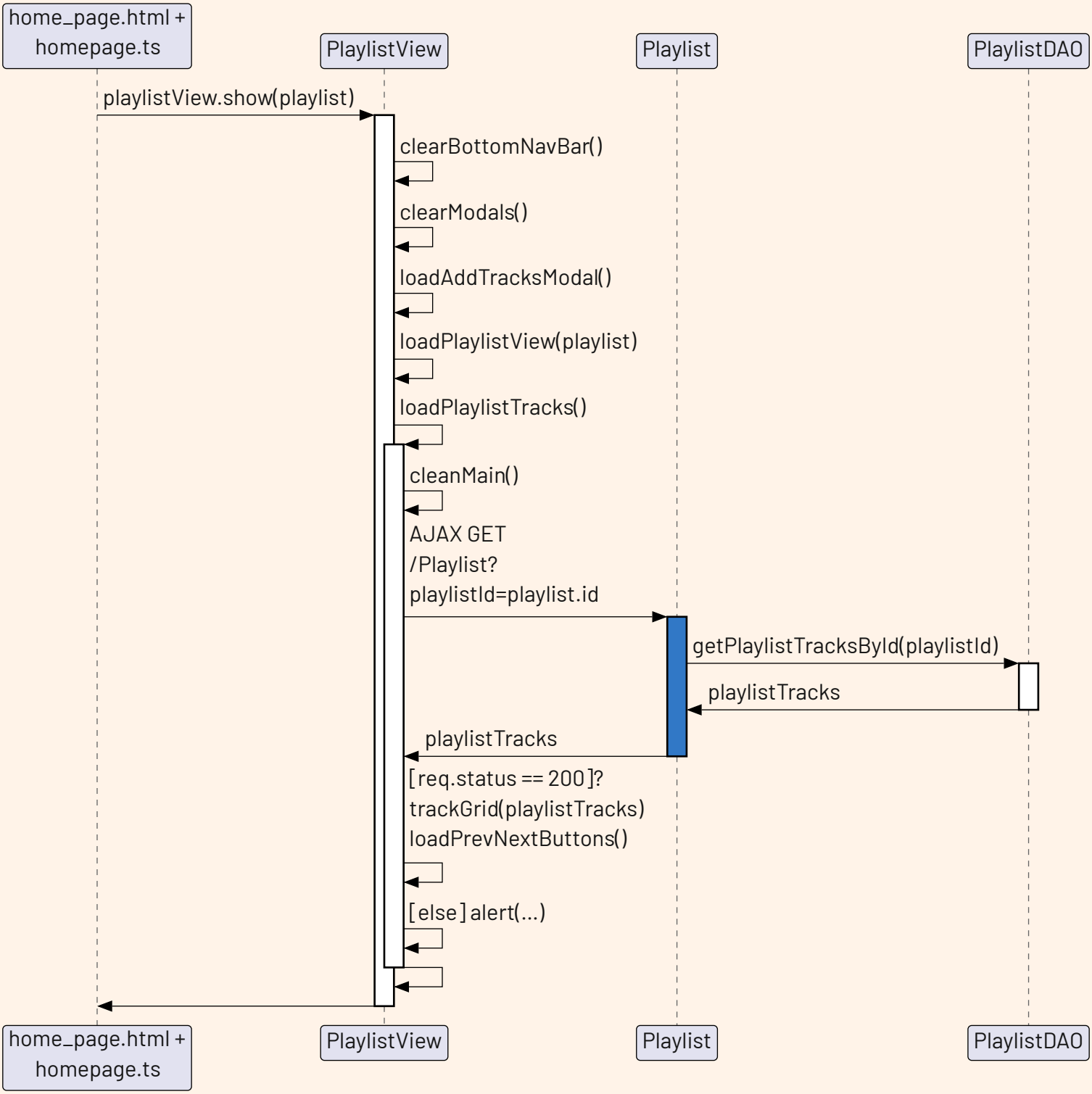
5.16 TS Event: Access HomeView



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.

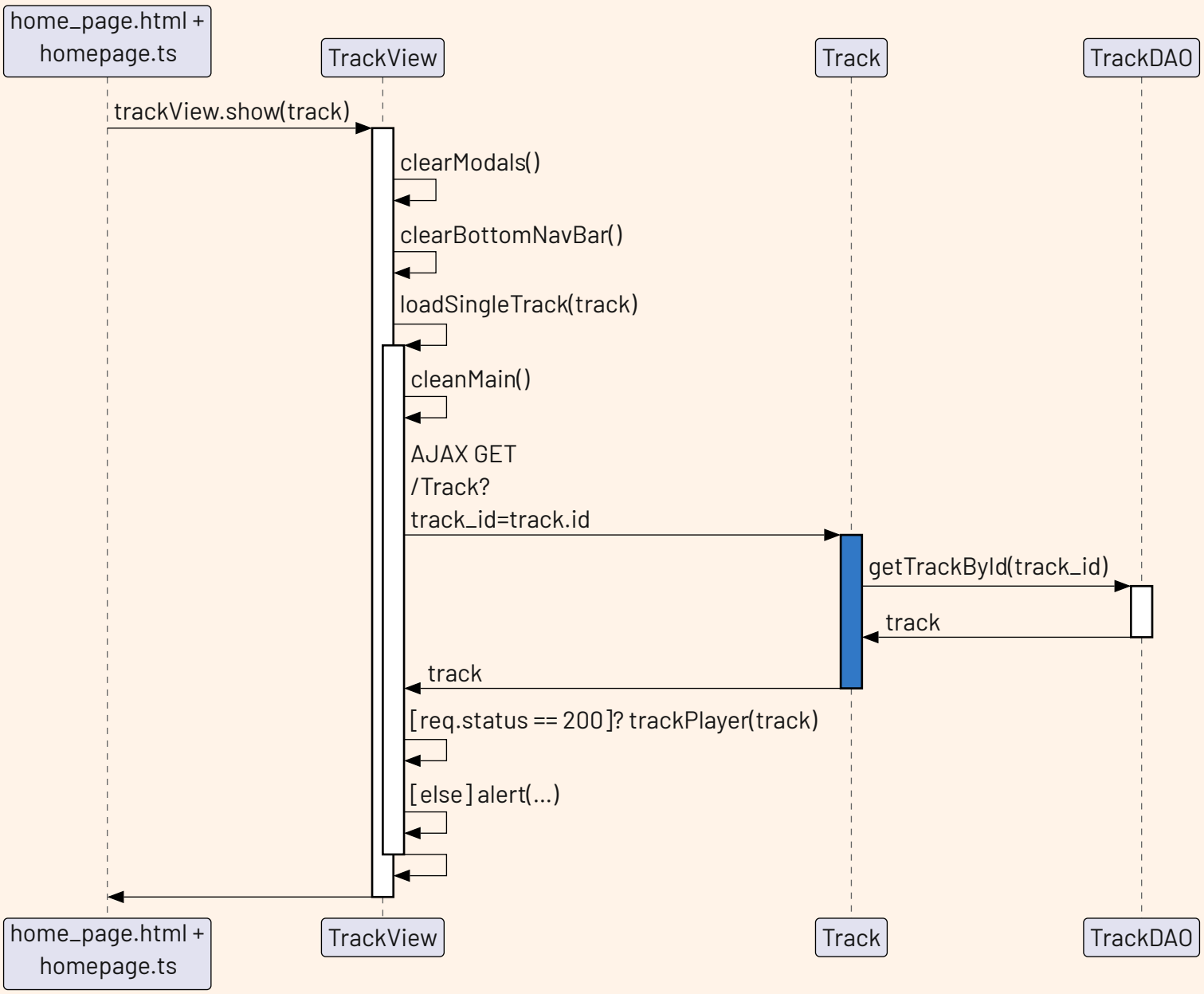
5.17 TS Event: Access PlaylistView



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 visited 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.

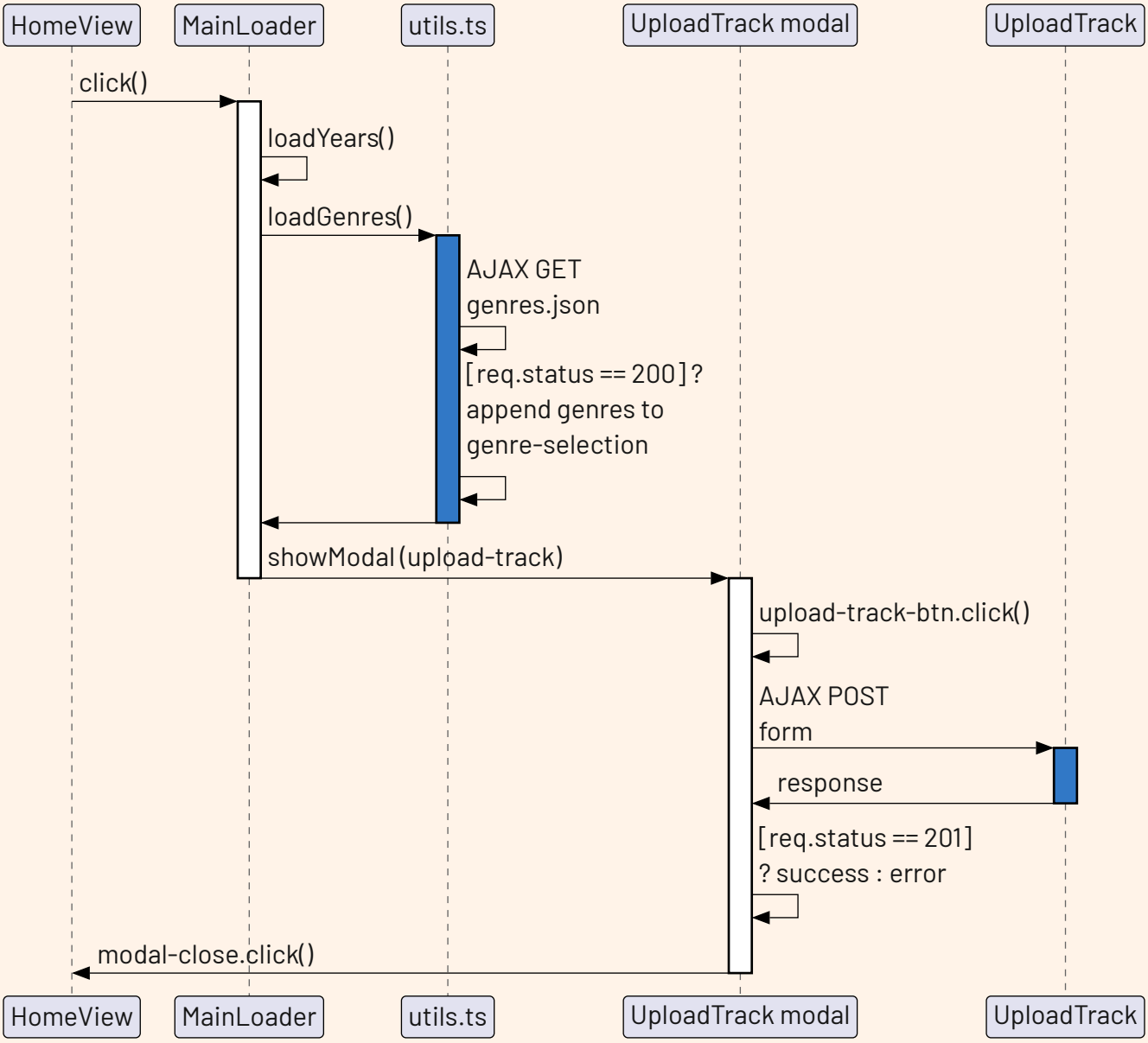
5.18 TS Event: Access TrackView



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 metadata.

5.19 TS Event: Upload Track modal



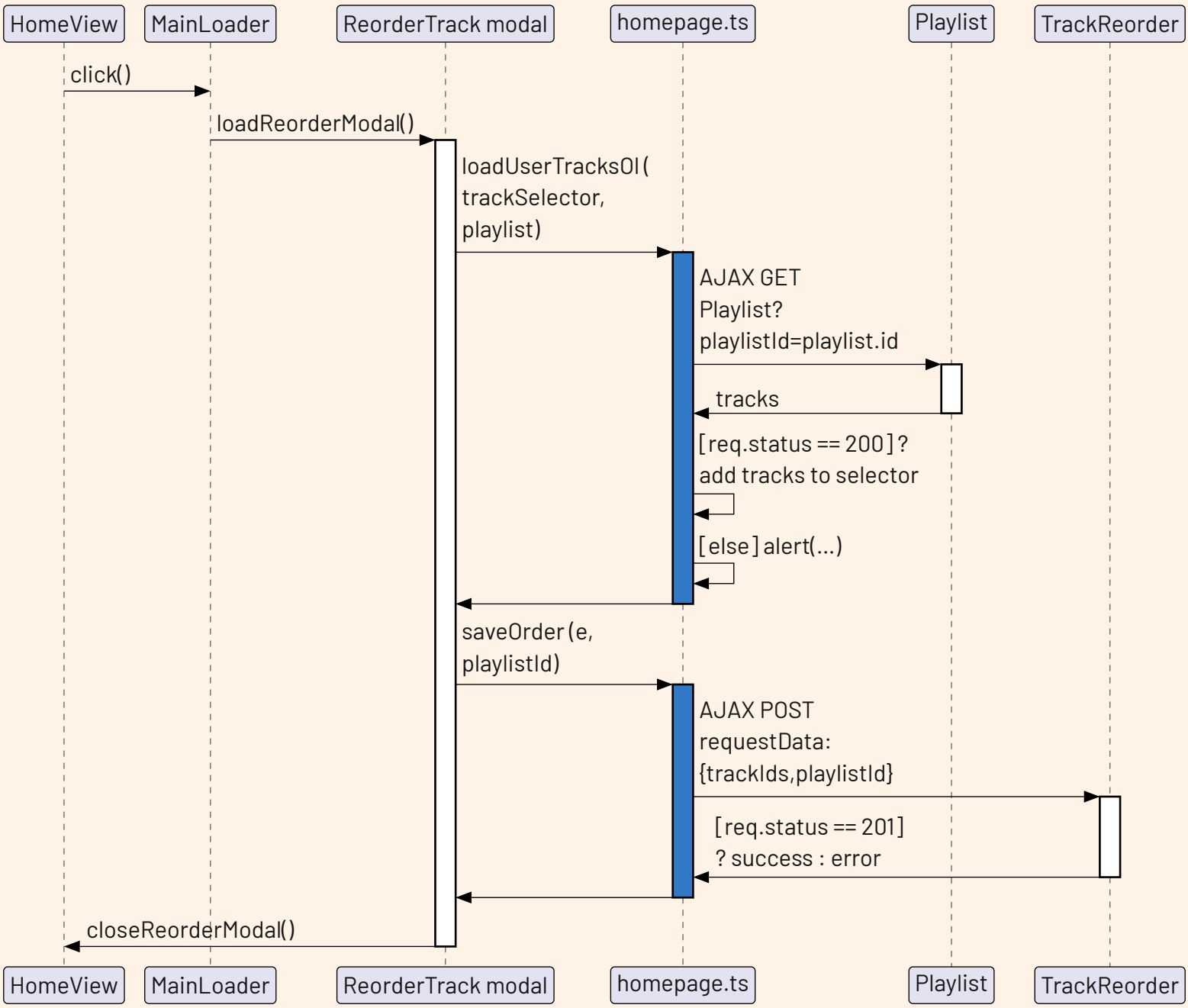
Comment

As the User logs in, in order to being able to listen the tracks, they must be uploaded. From the top nav bar, in the HomeView, there is the corresponding button.

After the click event, the MainLoader calls the `loadYears()`, `loadGenres()` and finally the `showModal()` functions. From there, the User is able to create new tracks by inserting the necessary metadata: title, artist, album, year, genre, image and file path (these are the same as seen in the Upload track sequence, see Section 5.6 – that's why they are omitted in the diagram).

If the operation is successful, a div with "Success" is shown; otherwise error. Finally the User can close and modal and return to the HomeView.

5.20 TS Event: Reorder modal



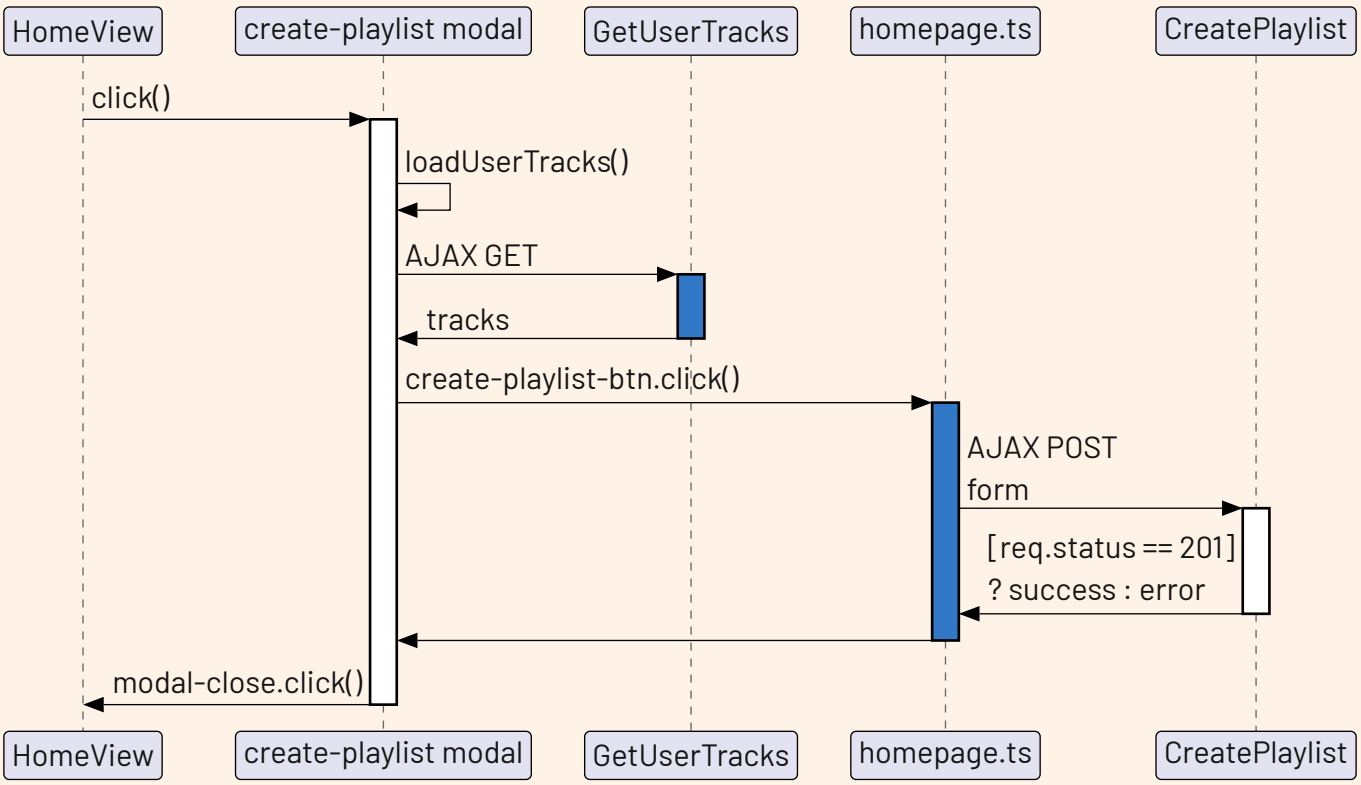
Comment

This modal is quite different from the add tracks modal because, in that case, the User must see the tracks that are not *in that playlist, however if they are to be reordered, the User has to see them all. And also being able to drag and drop them: that's why the* `loadUserTracksOl()` *function is called – `ol` stands for* Ordered List. *The logic is very similar to the regular* `loadUserTracks()`.

After being satisfied with the new order, the User clicks on the save order button that POSTs a JSON-formatted object to Java – the only function in the project to do so.

If the operation is successful, a div with "Success" is shown; otherwise error. Finally the User can close and modal and return to the HomeView.

5.21 TS Event: Create Playlist modal

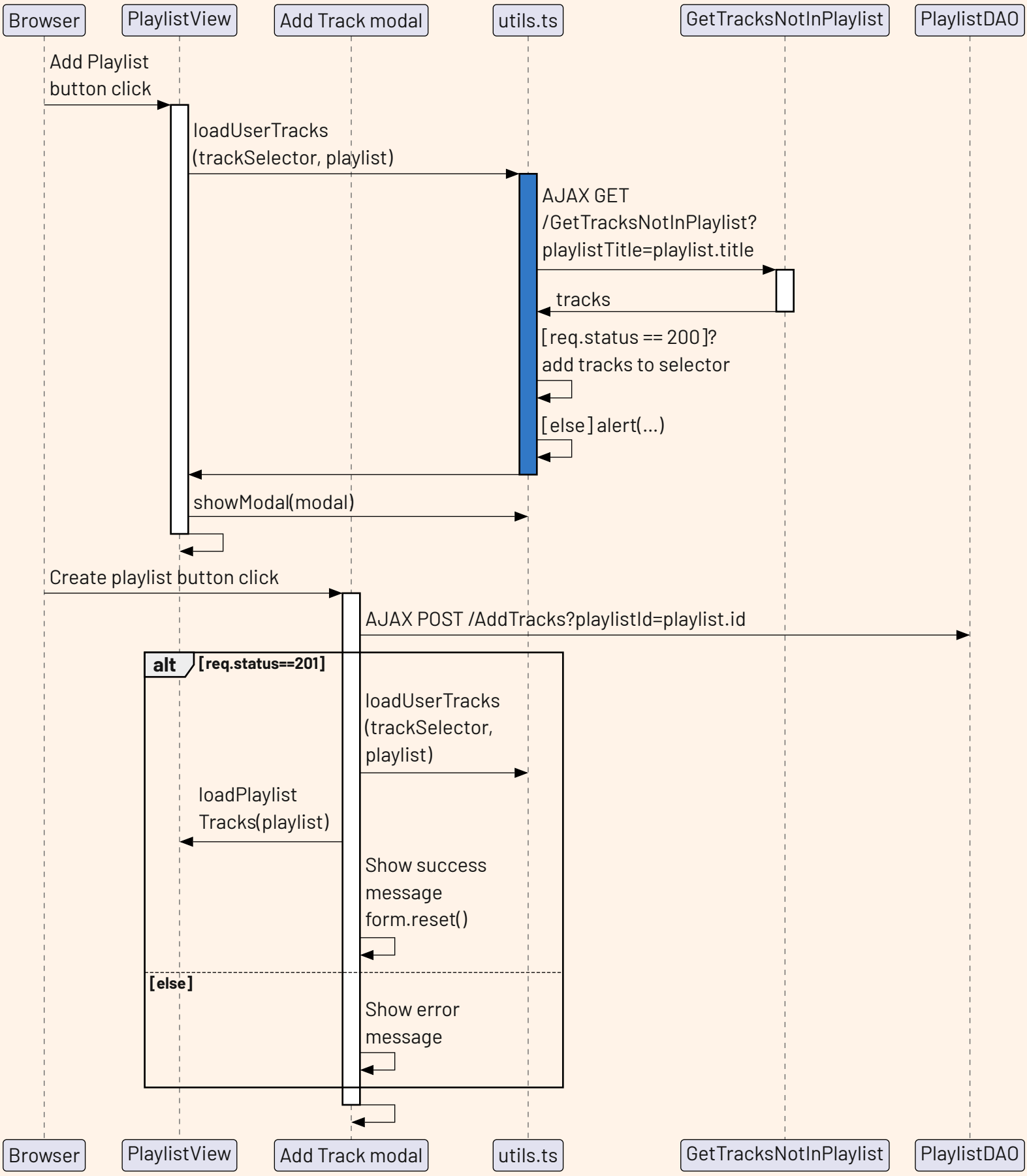


Comment

The User is able to create playlists by clicking on the corresponding button in the top navigation bar. Then, the User must insert the title – optionally some tracks (as seen in Section 5.7) – and, once satisfied, click the button to save the updates.

If the operation is successful, a div with "Playlist created successfully" is shown; otherwise error. Finally the User can close the modal and return to the HomeView.

5.22 TS Event: Add Track modal



Comment

The User can access the add-tracks modal by clicking the Add Tracks button in the playlist view. The click event listener on the button gets the user tracks not already added to the playlist from the server, adds them to the track selector and then makes the modal visible. When the Add Tracks button inside the modal is clicked, an AJAX POST request containing the selected tracks and the `playlist_id` *is sent; depending on the response status an error or success message is shown in the modal.*