Web Technologies Project @ PoliMi, 2025

Creating a Playlist Manager with Thymeleaf & TS

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

Contents

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 single-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

Thanks to Vittorio Robecchi for this beautiful template.

[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**.

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**:

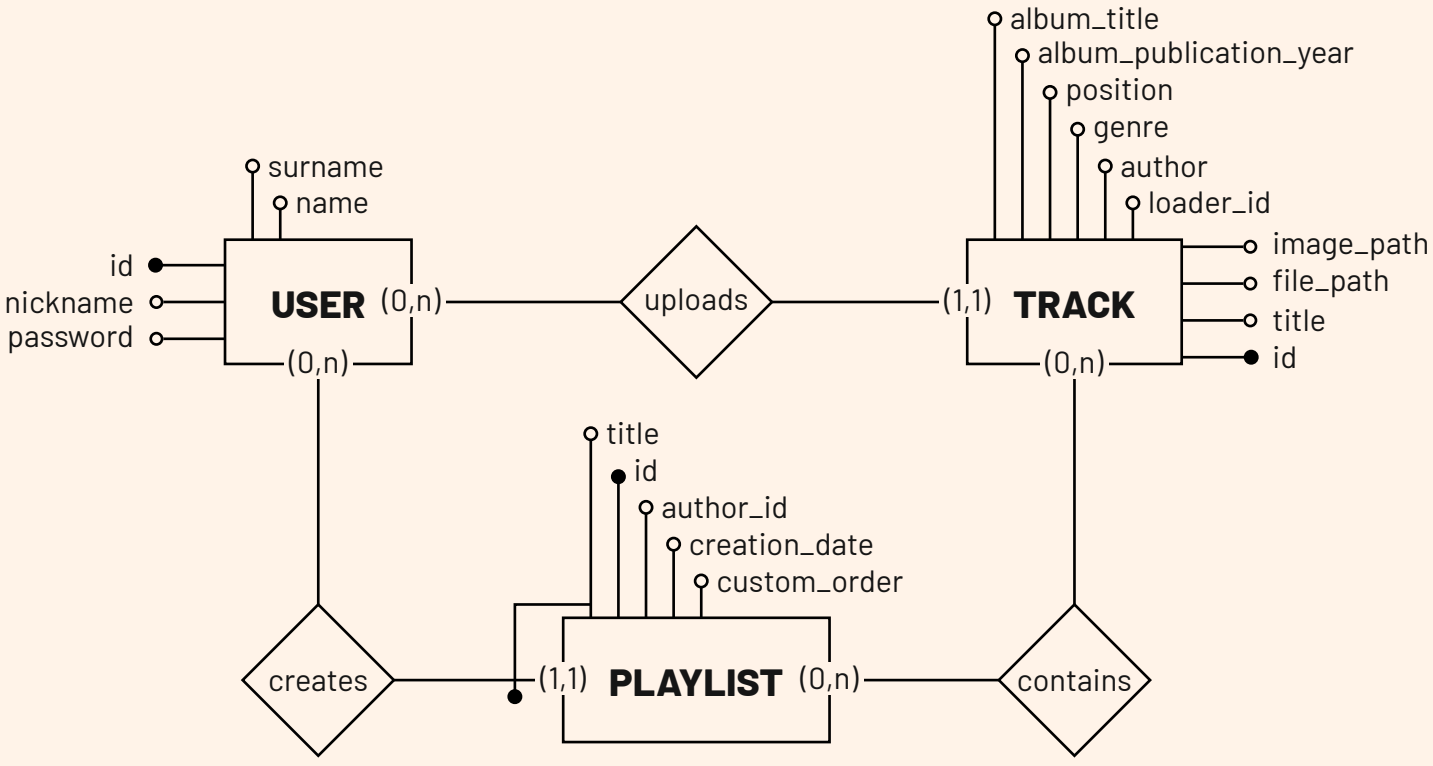


Figure 1: ER diagram (HTML and RIA).

- Every cell contains the track's title and album cover
- 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)

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.

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 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. From the HOME PAGE with an associated link to each playlist, the user access a modal window REORDER which shows the 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.

2.4 Added features

- Logout function:
  - HTML: Section 6.8
  - RIA: Section 7.11
- Subscribe function:
  - HTML: Section 6.9
  - RIA: Section 7.12

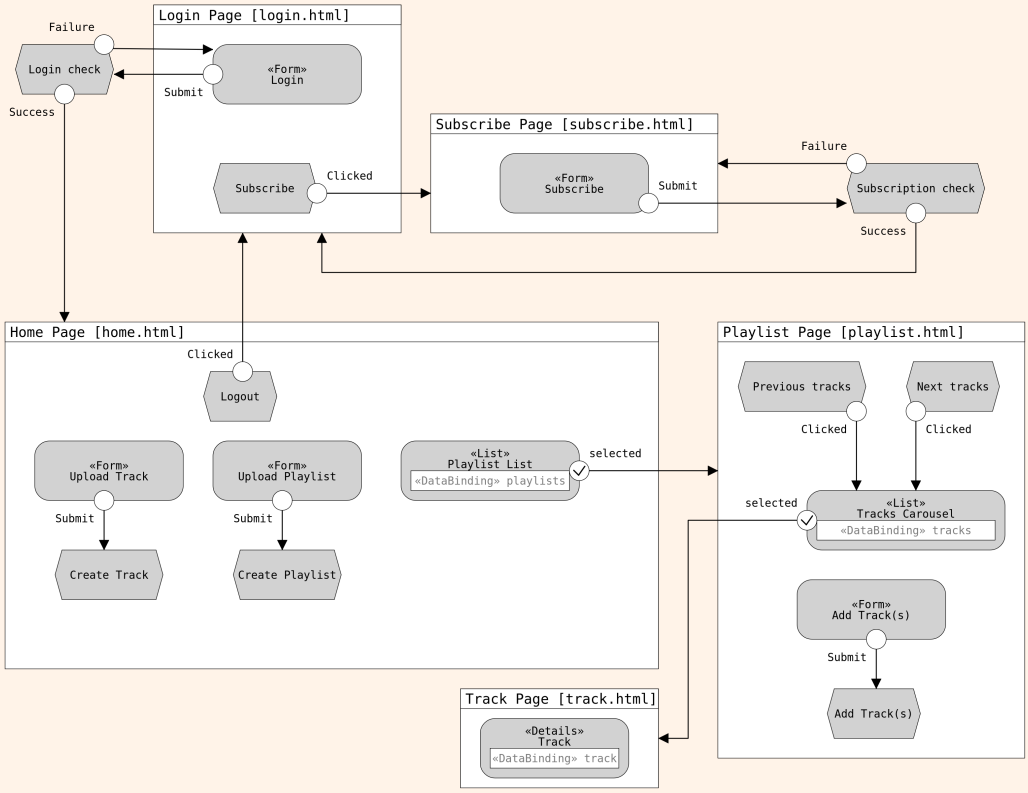


Figure 2: IFML diagram (HTML).

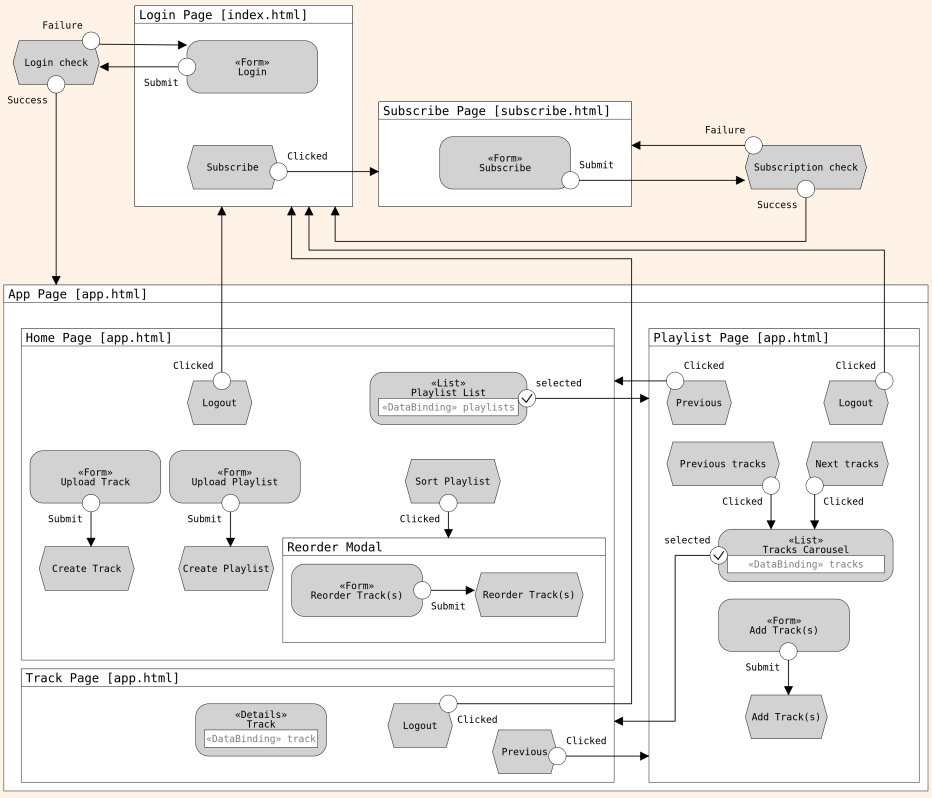


Figure 3: IFML diagram (RIA).

3

SQL database schema

3.1 Overview

The project requirements slightly change from `pure_html` and `js`, where the latter requires the playlist to support custom ordering the tracks – this is achieved by adding an optional position column in `playlist_tracks` and a custom order flag in `playlist`

In both scenarios, the schema is the same.

3.2 The tables

```
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`),
);
```

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

```
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,

    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
);
```

The 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.

```
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 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) via the foreign key.

```
CREATE TABLE playlist_tracks
(
    id          integer AUTO_INCREMENT,
    playlist_id integer not null,
    track_id    integer not null,
    position    integer,

    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 "Contains" 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

4.1 RIA subproject

The frontend was built by using the oop capabilities of Typescript.

4.2 Views

The views are represented by the following classes:

- LoginPage: the user can login in the app or go to the subscribe page
- SubscribePage: the user can subscribe
- HomePage: the user can load a track, create a playlist, open a playlist or open the modal to sort the playlist
- PlaylistPage: the user can add tracks to the playlist or open a track
- TrackPage: the user can see data about the track and listen to it
- Header: contains the logout and go to previous page buttons
- Modal: shows the tracks in the playlist and the user can sort them

The views are backed by two html files:

- index.html: is used by LoginPage and SubscribePage
- app.html: is used by all the others views

All the views, apart from LoginPage, implement a Component interface with the following methods:

- css: contains the styling specific to that page
- template: contains the static part of a page, in order to ease the creation of a page, instead of using only Typescript
- build: contains the code that builds the dynamic part of the page

This way the code is more organized.

4.3 Repositories

A very basic implementation of the repository pattern is implemented in the client, just to centralize the management of the api calls.

The repositories are:

- AuthRepository
- TrackRepository
- PlaylistRepository

As you can see they are a direct transposition of the controllers from the backend.

When a view has to make a call to the server, it instantiate the repository and uses its implementation, instead of writing each time the same code.

4.4 Actions

The possible actions are described in the following tables.

Client side		Server side	
Event	**Action**	**Event**	**Action**
Index ⇒ Login form ⇒ Submit	Data validation	POST (`username`, `password`)	Credentials check
HomePage ⇒ Load	Loads all User playlists and tracks	GET	Queries user playlists
HomePage ⇒ Click on a playlist	Loads PlaylistPage for that Playlist	GET	Queries the tracks associated to the given playlist
HomePage ⇒ Click on sort button	Load a modal to custom order the tracks in the Playlist	GET	Queries the tracks associated to the given playlist
HomePage ⇒ Close button	Saves the custom order to the database	POST (`trackIds`, `playlistId`)	Updates the `playlist_tracks` table with the new custom order
HomePage ⇒ add playlist form ⇒ Submit	Data validation	POST (`playlistTitle`, `selectedTracks`)	Inserts the new Playlist in the `playlist` table
HomePage ⇒ add track form ⇒ Submit	Data validation	POST (`title`, `artist`, `year`, `album`, `genre`, `cover`, `track`)	Inserts the new Track in the `tracks` table
PlaylistPage ⇒ add tracks to playlist form ⇒ Submit	Data validation	POST (`playlistId`, `selectedTracks`)	Inserts the selected tracks in the given playlist
PlaylistPage ⇒ Click on a track	Loads TrackPage for that Track	GET	Queries the requested track
Logout	Invalidates the current User session	GET	Session invalidation
Prev	Loads previous page	-	-

Table 2: Events & Actions.

Client side		Server side	
Event	**Controller**	**Event**	**Controller**
Index ⇒ Login form ⇒ Submit	AuthRepository .login()	POST (`username`, `password`)	AuthController
HomePage ⇒ Load	PlaylistRepository .getPlaylists() TrackController .getTracks()	GET	PlaylistController and TrackController
HomePage ⇒ Click on a playlist	TrackRepository .getAllTracksInPlaylist()	GET	TrackController
HomePage ⇒ Click on sort button	TrackRepository .getAllTracksInPlaylist()	GET	TrackController
HomePage ⇒ Close button	PlaylistRepository .setCustomOrder()	POST (`trackIds`, `playlistId`)	PlaylistController
HomePage ⇒ add playlist form ⇒ Submit	PlaylistRepository .createPlaylist()	POST (`playlistTitle`, `selectedTracks`)	PlaylistController
HomePage ⇒ add track form ⇒ Submit	TrackRepository .createTrack()	POST (`title`, `artist`, `year`, `album`, `genre`, `cover`, `track`)	TrackController
PlaylistPage ⇒ add tracks to playlist form ⇒ Submit	PlaylistRepository .addTracksToPlaylist()	POST (`playlistId`, `selectedTracks`)	PlaylistController
PlaylistPage ⇒ Click on a track	TrackRepository .getTrackById()	GET	TrackController
Logout	AuthRepository .logout()	GET	Springboot internal LogoutController
Prev	App.pop()	-	-

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

5

SecurityConfig

Instead of manually creating filters to block mappings to an anauthenticated user, it's possible to use the system of filters provided by Springboot.

The version for the **HTML** subproject is the simplest:

- the requests that can always be accessed are /login and /subscribe (also the stylesheets are always available)
- all the other requests requires the user to be authenticated
- the login can be performed by making a POST request to /login and if successful, the user is redirected to /home
- the logout can be performed by making a POST request to /logout

The version for the **RIA** subproject is more complex because the login process is handled differently (because there isn't thymeleaf to help):

- the requests that can always be accessed are /login and /subscribe
- also the resources folder containing the scripts and the html files using those scripts, are always available
- all the other requests requires the user to be authenticated
- the logout can be performed by making a POST request to /logout and in case of success the user is redirect to /index.html

Since the login process is more complex, it also requires to use an AuthenticationManager and a SecurityContextRepository: by declaring them here, later they can be injected in the login function (see Section 7.10).

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {


    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.csrf(csrf -> csrf.csrfTokenRepository(new HttpSessionCsrfTokenRepository()))
                .authorizeHttpRequests(registry -> registry
                        .dispatcherTypeMatchers(
                          DispatcherType.FORWARD,
                          DispatcherType.ERROR
                        ).permitAll()
                        .requestMatchers("/login", "/subscribe", "/css/**").permitAll()
                        .anyRequest().authenticated()
                )
                .formLogin(httpForm -> httpForm
                        .loginPage("/login")
                        .defaultSuccessUrl("/home")
                        .permitAll()
                )
                .logout(logout -> logout.permitAll());

        return http.build();
    }

}
```

Listing 1: Springboot security configuration HTML

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http
                .authorizeHttpRequests(registry -> registry
                        .dispatcherTypeMatchers(
                          DispatcherType.FORWARD,
                          DispatcherType.ERROR
                        ).permitAll()
                        .requestMatchers("/login", "/subscribe", "/static/**", "/favicon.ico",
"/dist/**",
                                "/index.html", "/app.html")
                        .permitAll()
                        .anyRequest().authenticated())
                .csrf(csrf -> csrf.disable())
                .logout(l -> l
                        .logoutUrl("/logout")
                        .logoutSuccessUrl("/index.html")
                        .invalidateHttpSession(true)
                        .permitAll())
                .build();
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration config)
throws Exception {
        return config.getAuthenticationManager();
    }

    @Bean
    public SecurityContextRepository securityContextRepository() {
        return new DelegatingSecurityContextRepository(
                new RequestAttributeSecurityContextRepository(),
                new HttpSessionSecurityContextRepository());
    }

}
```

Listing 2: Springboot security configuration RIA

6

Sequence diagrams HTML

6.1 Disclaimer

Space on a page is not infinite and Springboot adds some intermediate passages, so, to keep things readable, these assumptions are applied to the following sequence diagrams.

6.2 Client

Represented by: 

6.3 DispatcherServlet

Represented by: 

This servlet is used by Springboot to route incoming requests to the correct servlets created by the mappings in Springboot controllers.

In the case of a traditional MVC application (like the HTML version of this project), this servlet is also used to manage the rendering of a page: in this case DispatcherServlet has to pass a ModelAndView object to a render function.

In order to build this ModelAndView object, a Model object and a View object are needed; it's possible to create manually a ModelAndView object but in this project a different solution is adopted:

1. Each mapping (associated for convention to a showPage() method) that has to load the page, obtains the Model from the framework (by using dependency injection)
2. The Model is then filled with data that needs to be used by the view (in this case a thymeleaf template)
3. The mapping returns a string with the name of a template (return "home" ⇒ loads home.html); this string is read by DispatcherServlet to create the view

At the end of the process DispatcherServlet can build the ModelAndView object and render it.

6.4 Redirects

After some operations (like adding a new track), the user wants to see changes: the mappings that are associated to those operations follow the same flow, like in the following example:

1. the user requests the home page with a GET call to /home
2. In the home page the user loads a new track with a POST call to /add_track
3. /add_track saves the track
4. /add_track redirects the user to /home
5. the home page is loaded again, this time with also the new track

6.5 Exceptions

All exceptions are managed, but since it wasn't a requirement of the project to manage them precisely (and because of time constraints), they are captured by simple catch statements that catch RuntimeExceptions

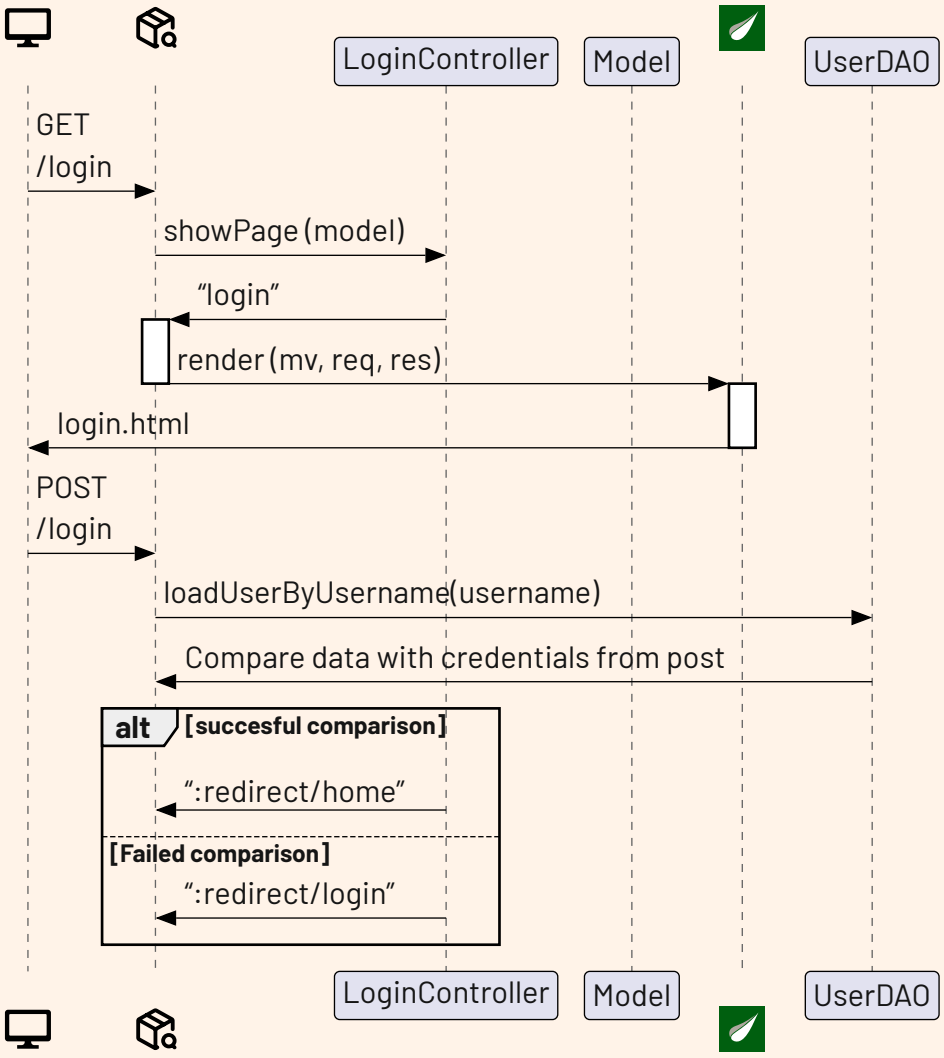
6.6 UserId

UserId appears a lot of times in the following diagrams: each time it is seen, the following snippet of code is executed:

```
public static int getUserId() {
    return ((UserWithId)
    SecurityContextHolder
    .getContext()
    .getAuthentication()
    .getPrincipal())
    .getId();
}
```

When a user is authenticated, Springboot stores his information in a SecurityContextHolder, and in this code we simply extract his id (see Section 6.7 for more).

6.7 Login sequence diagram



Comment

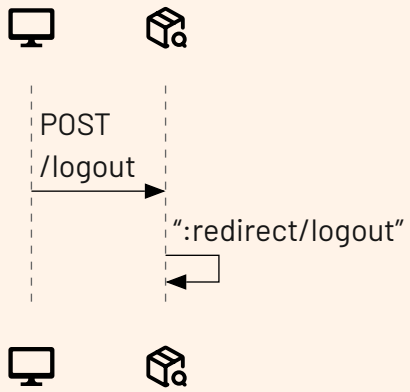
Once the server is up and running, the Client requests the login page. Then, thymeleaf processes the request and returns the correct view its data. Afterwards, the User inserts their credentials.

Those values are passed to DispatcherServlet which sends the message to a POST /login mapping offered by Springboot: here the message is analyzed by different filters (almost all of them are used internally by Springboot), till it reaches the AuthorizationFilter.

The AuthorizationFilter calls internally the UserDetailsService Bean which is implemented by the UserService class: this class the loadUserByUsername method, which queries the DB to receive a user associated to that username (that's why the UserDAO is involved).

After this operation, the AuthorizationFilter filter compares the data provided by the client and by UserDetailsService and if they are the same, the user is redirected to the home page. Otherwise the user is redirected to the login page.

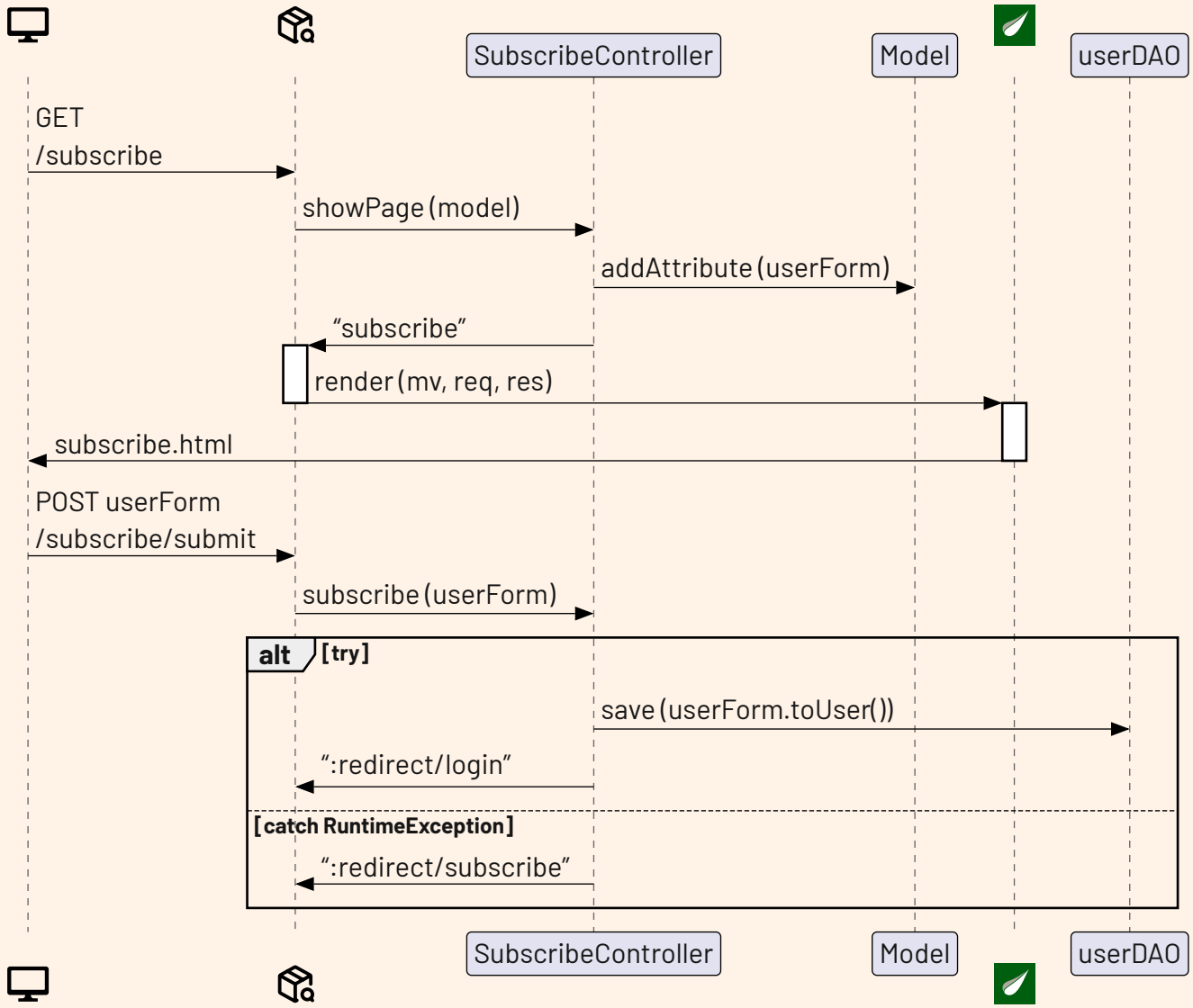
6.8 Logout sequence diagram



Comment

In order to perform a logout, Springboot requires the client to call the POST /logout mapping which internally invalidates session and all the other data associated to the user (of course the data inside the DB are preserved), and then the user is redirected to the login page.

6.9 Subscribe sequence diagram



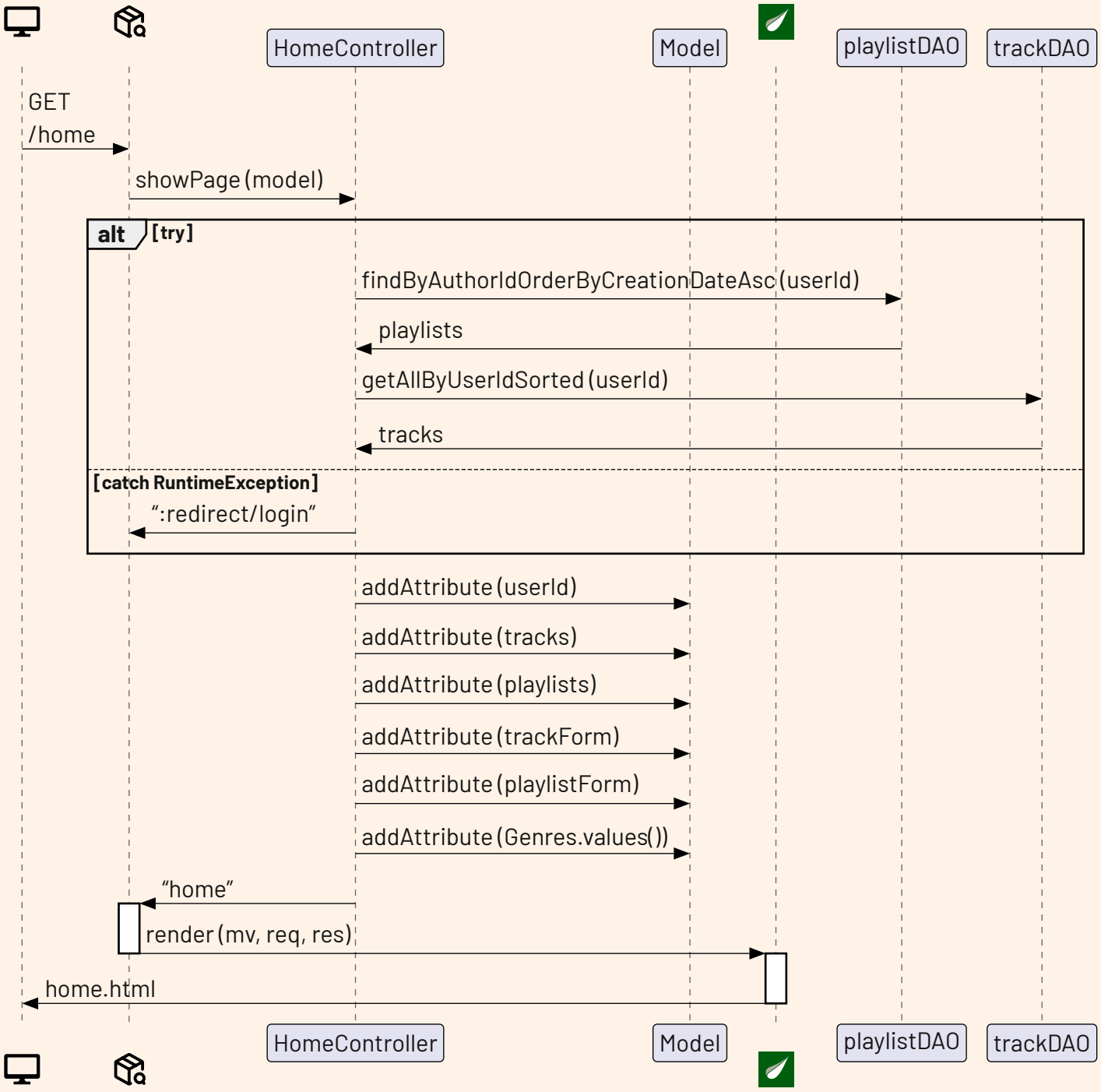
Comment

After requesting the page with GET /subscribe, the user can fill and submit the provided form to subscribe to the site.

The call is redirected to SubscribeController by DispatcherServlet: the controller tries to save the user by calling userDAO.save() and if any kind of RuntimeException exception occurs (so SQLExceptions are also considered), the user is redirected to the subscribe page. Otherwise the user is redirected to the login page.

In this situation an SQLException might also occur because in the DB there could be already a user with the same username.

6.10 Show home page sequence diagram



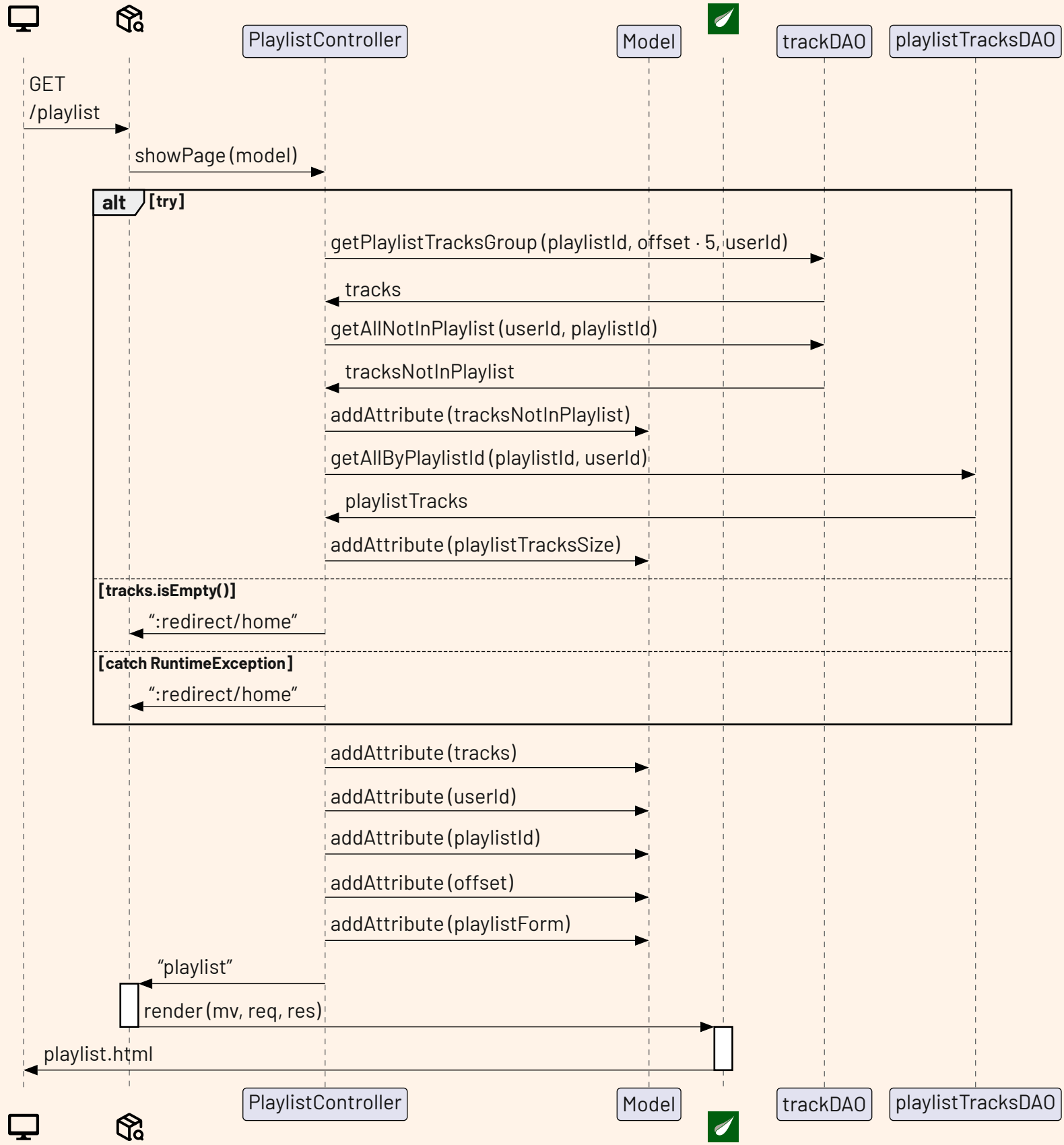
Comment

When the user requests the home page, the HomeController obtains the user id and with it, it requests: all playlists owned by the user, sorted as pointed in the submission, and all tracks owned by the user, sorted as pointed in the submission. The results of these requests are added inside the thymeleaf Model.

If a RuntimeException occurs the user is redirected to the login page.

Inside the Model are also added the trackForm and Genre.values() to manage the upload of a track and playlistForm to manage the upload of a playlist.

6.11 Show playlist page sequence diagram



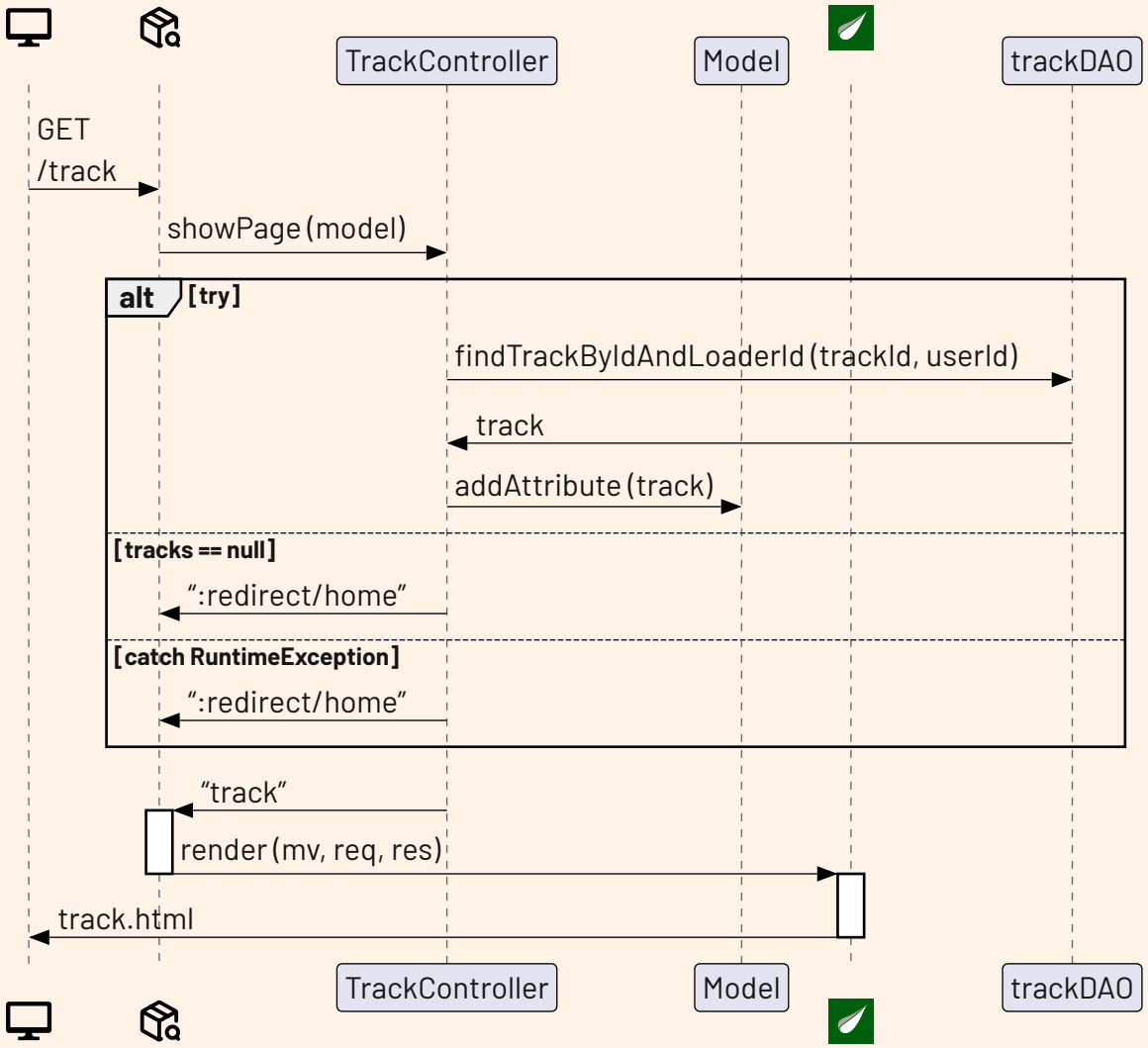
Comment

When the user requests the playlist page, PlaylistController obtains the user id and with it, it requests: the first group of tracks in the playlist; all tracks not in the playlist; all tracks in the playlist. The results of these requests are added inside the thymeleaf Model (for the last one only the size is considered).

If a RuntimeException occurs or the playlist is empty, the user is redirected to the login page.

Inside the Model is also added playlistForm to manage the insertion of more tracks in the playlist.

6.12 Show track page sequence diagram

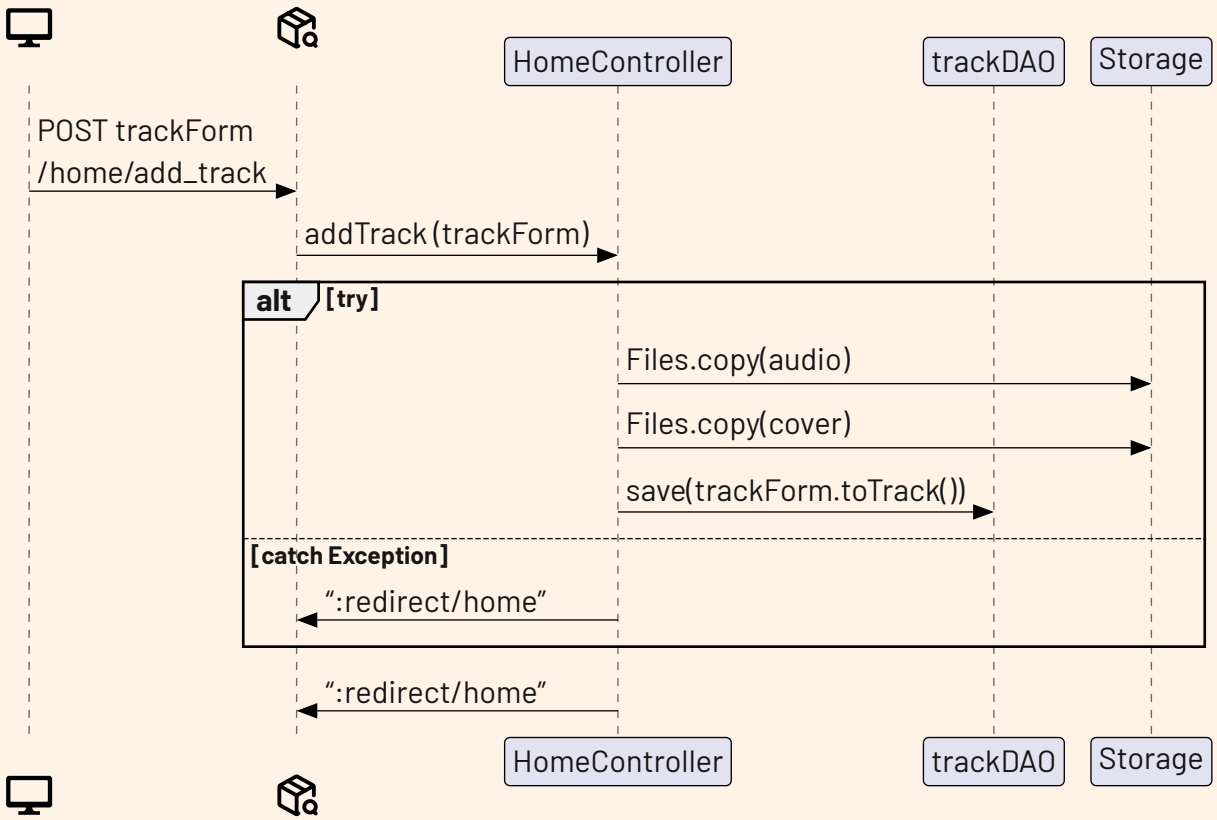


Comment

When the user requests the track page, the TrackController obtains the user id and it requests the track that the user wants to display. The result is added to the model to display the data about the track.

If a RuntimeException occurs or the track is null, the user is redirected to the home page.

6.13 Add track sequence diagram

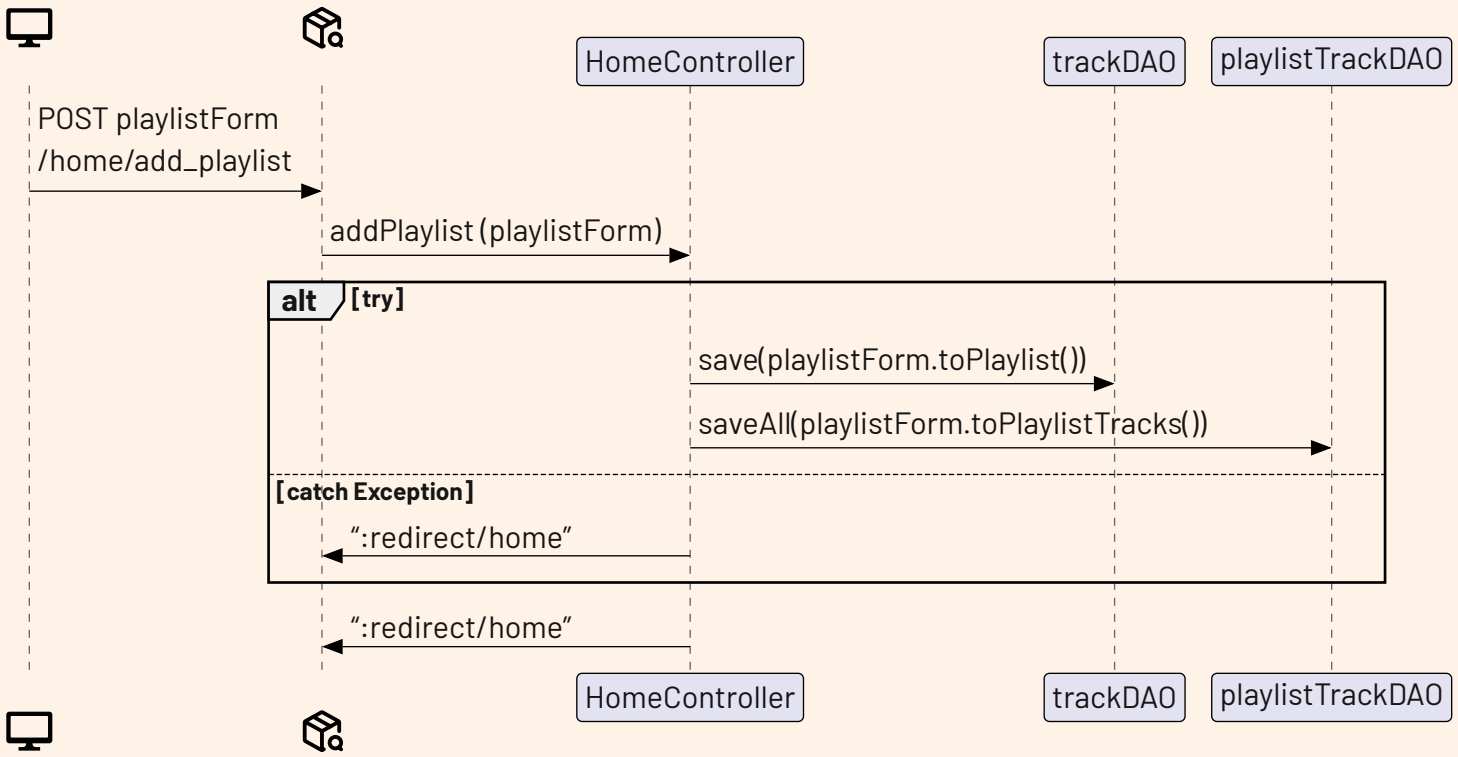


Comment

When the user loads a track, the request is handled by a method in HomeController where the mime type of the loaded files is checked and if they are right, the track is also added to the database.

If a RuntimeException occurs or if everything went fine, the user is redirected to the home page anyway.

6.14 Add playlist sequence diagram

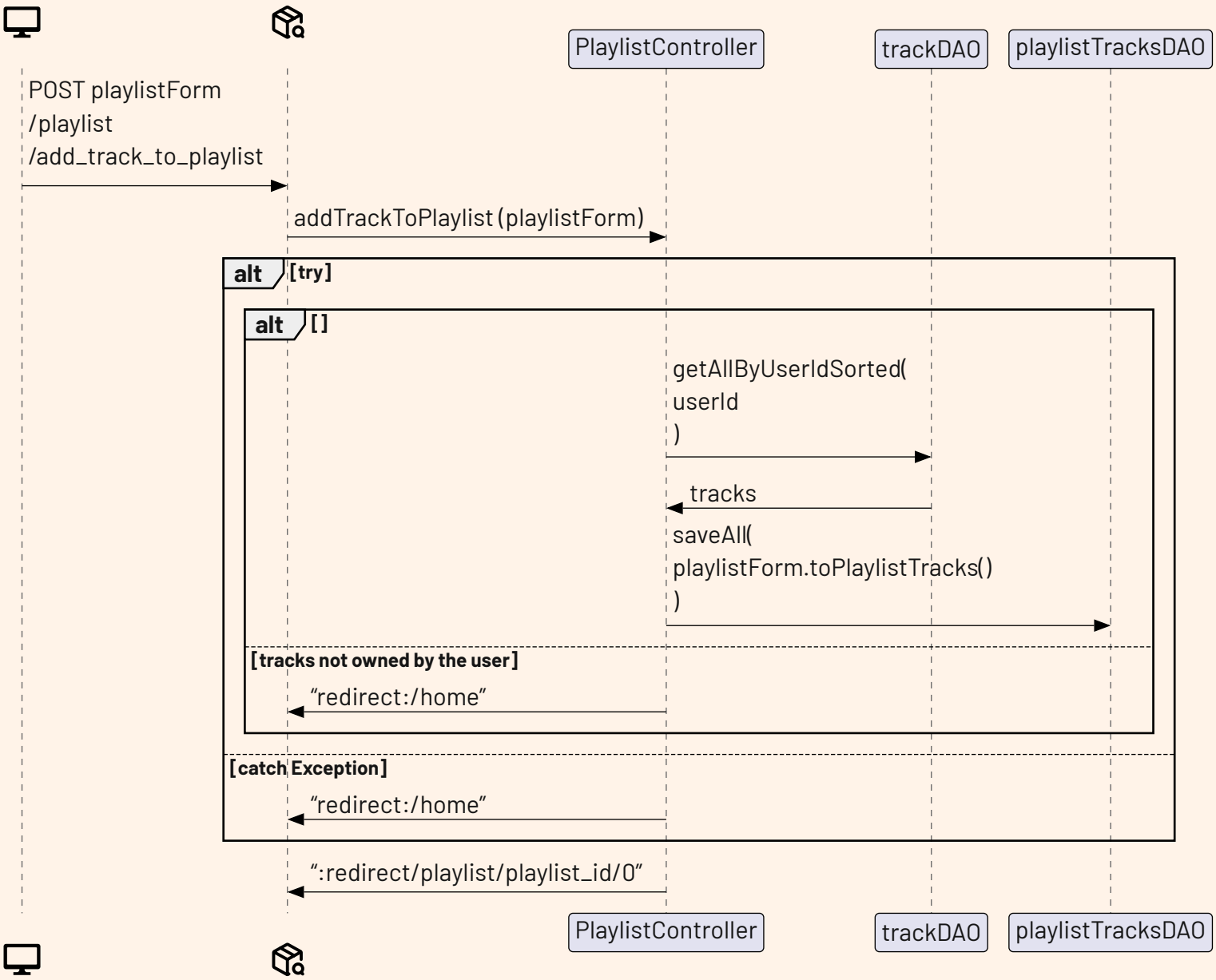


Comment

When the user loads a playlist, the request is handled by a method in HomeController which uses trackDAO and playlistTracksDAO to store the data from playlistForm.

If a RuntimeException occurs or if everything went fine, the user is redirected to the home page.

6.15 Add track to playlist sequence diagram



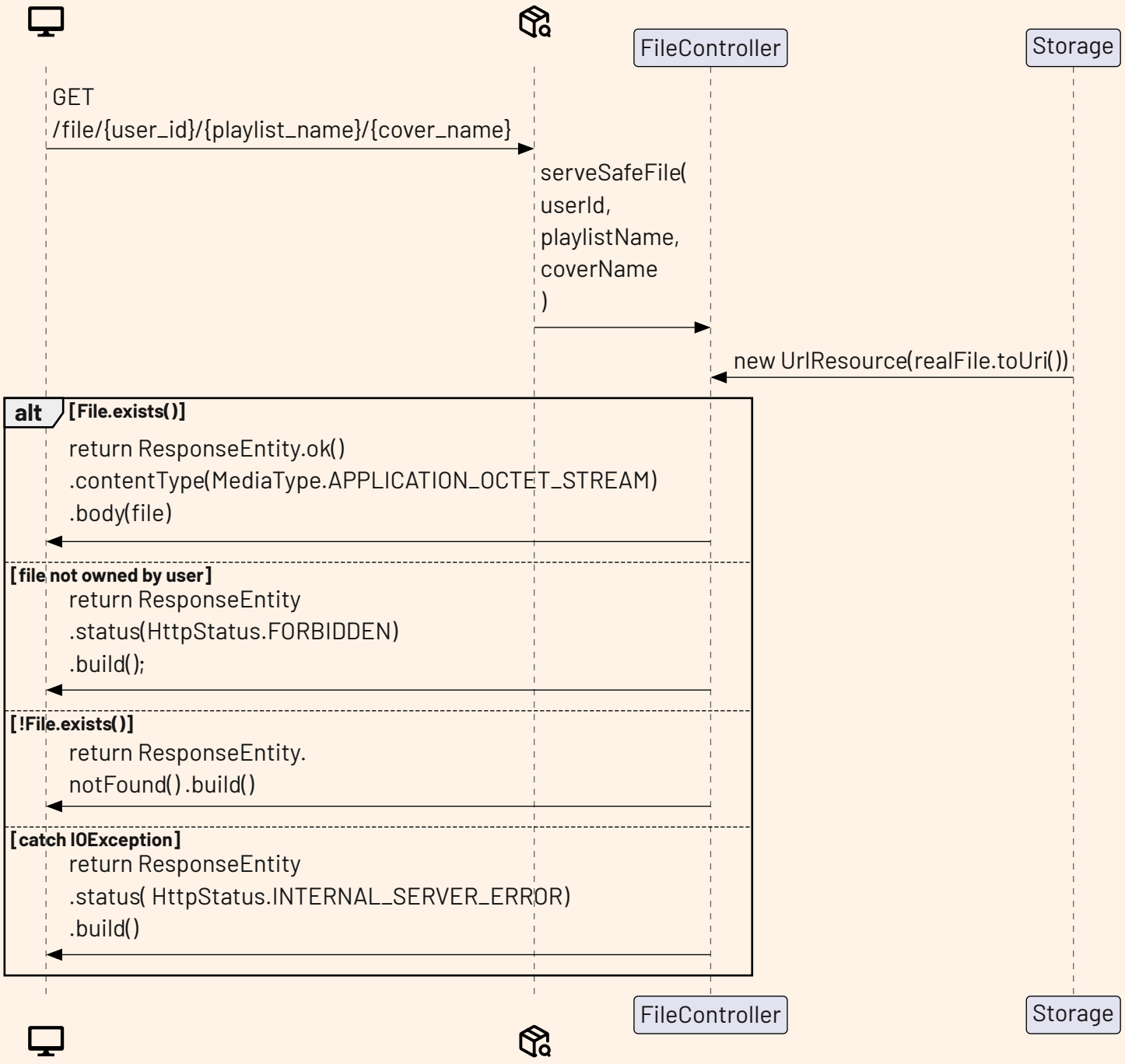
Comment

When the user loads a playlist, the request is handled by a method in PlaylistController which uses playlistTracksDAO to store the new tracks added in the playlist.

If the submitted tracks are not owned by the user, the request is rejected.

If a RuntimeException occurs or if everything went fine, the user is redirected to the home page.

6.16 Get file sequence diagram



Comment

When the user wants to request a file he calls this method which handles the request and if every check is passed, the file is returned.

7

Sequence diagrams RIA

7.1 Disclaimer

Without the integration of thymeleaf ✅ with Springboot, changes occurred in the project structure. For example the login is more manually handled (see).

7.2 SecurityContextHolder

Represented by: 

As the name suggests, this class contains the security context, which means it contains data about the currently authenticated user. And if SecurityContextHolder holder is empty the user is considered not authenticated.

Note that this is the same class used to retrieve the userId inside the controllers.

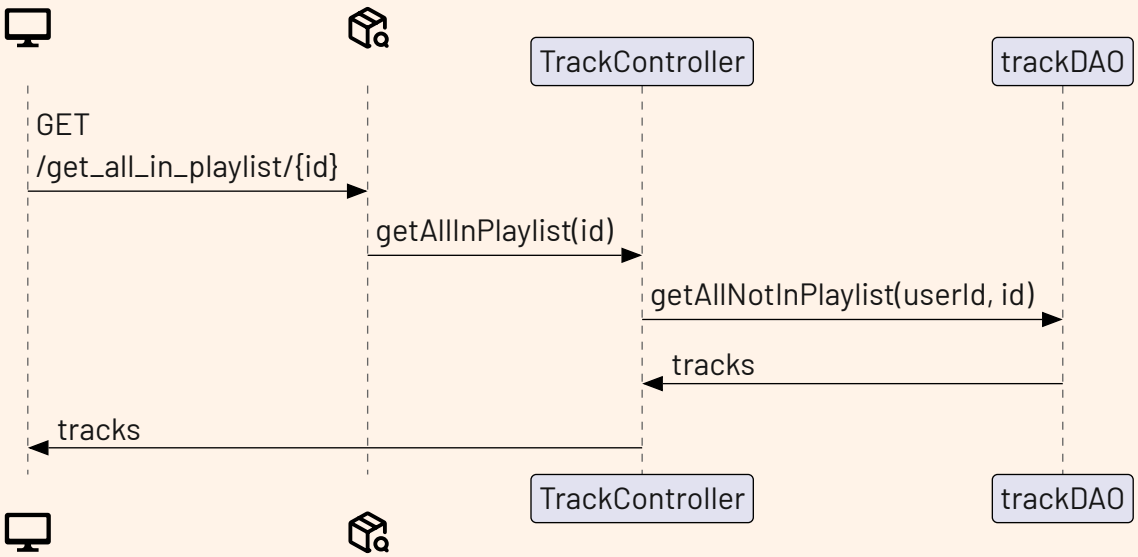
7.3 Basic mappings

The first 6 mappings are very basic and they all act in the same way:

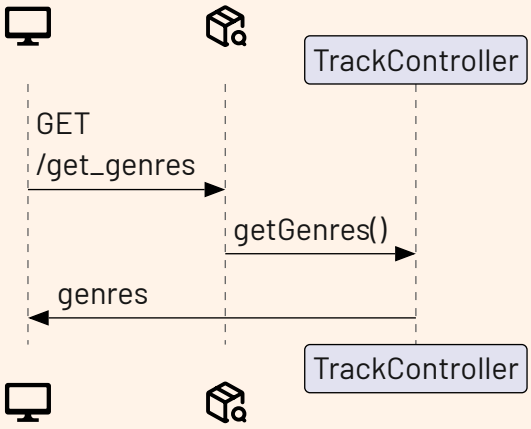
1. the client creates a GET request
2. the request arrives to the correct mapping
3. the mapping obtains the requested data from a specific DAO
4. the mapping returns the obtained data to the client

For this reason there won't be any comments under them.

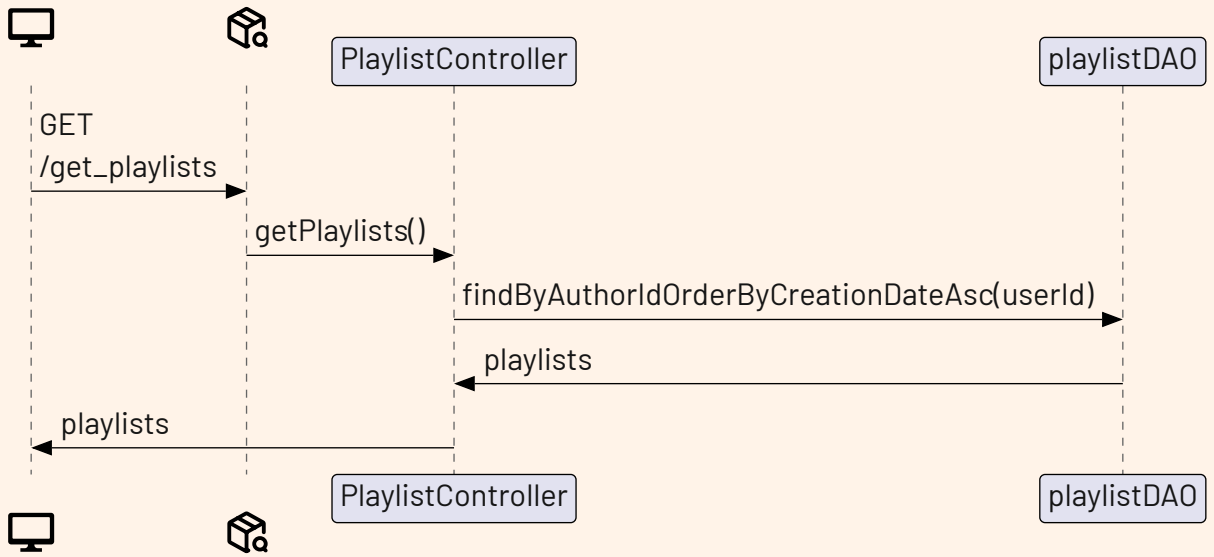
7.4 GetAllNotInPlaylist sequence diagram



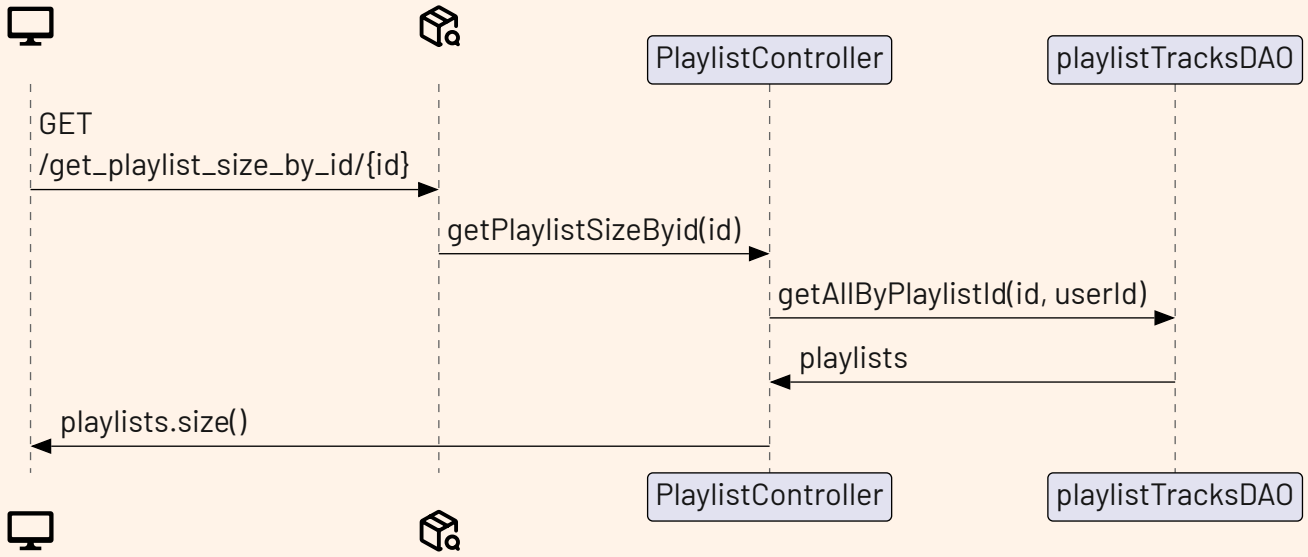
7.5 GetGenres sequence diagram



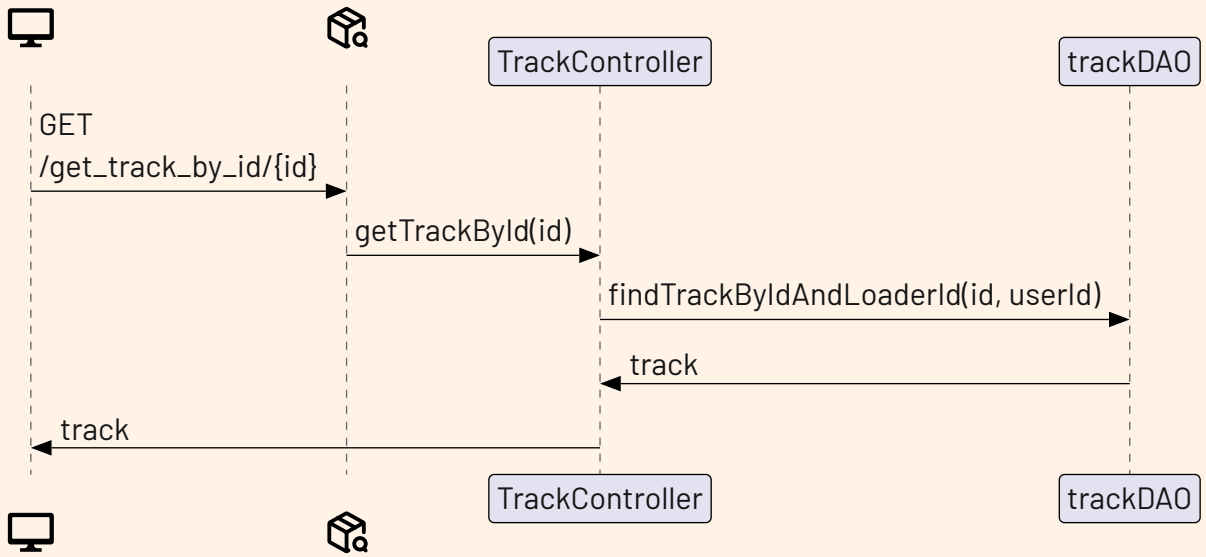
7.6 GetPlaylist sequence diagram



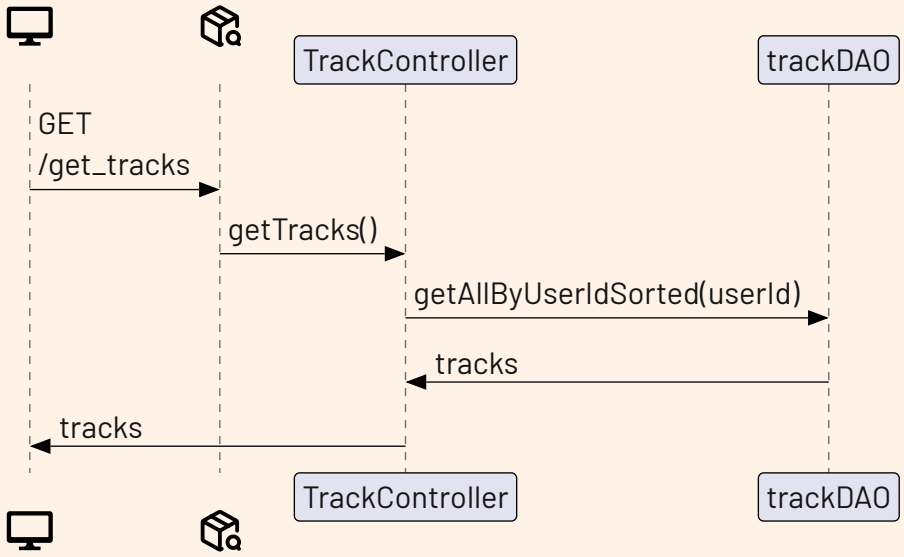
7.7 GetPlaylistSizeByid sequence diagram



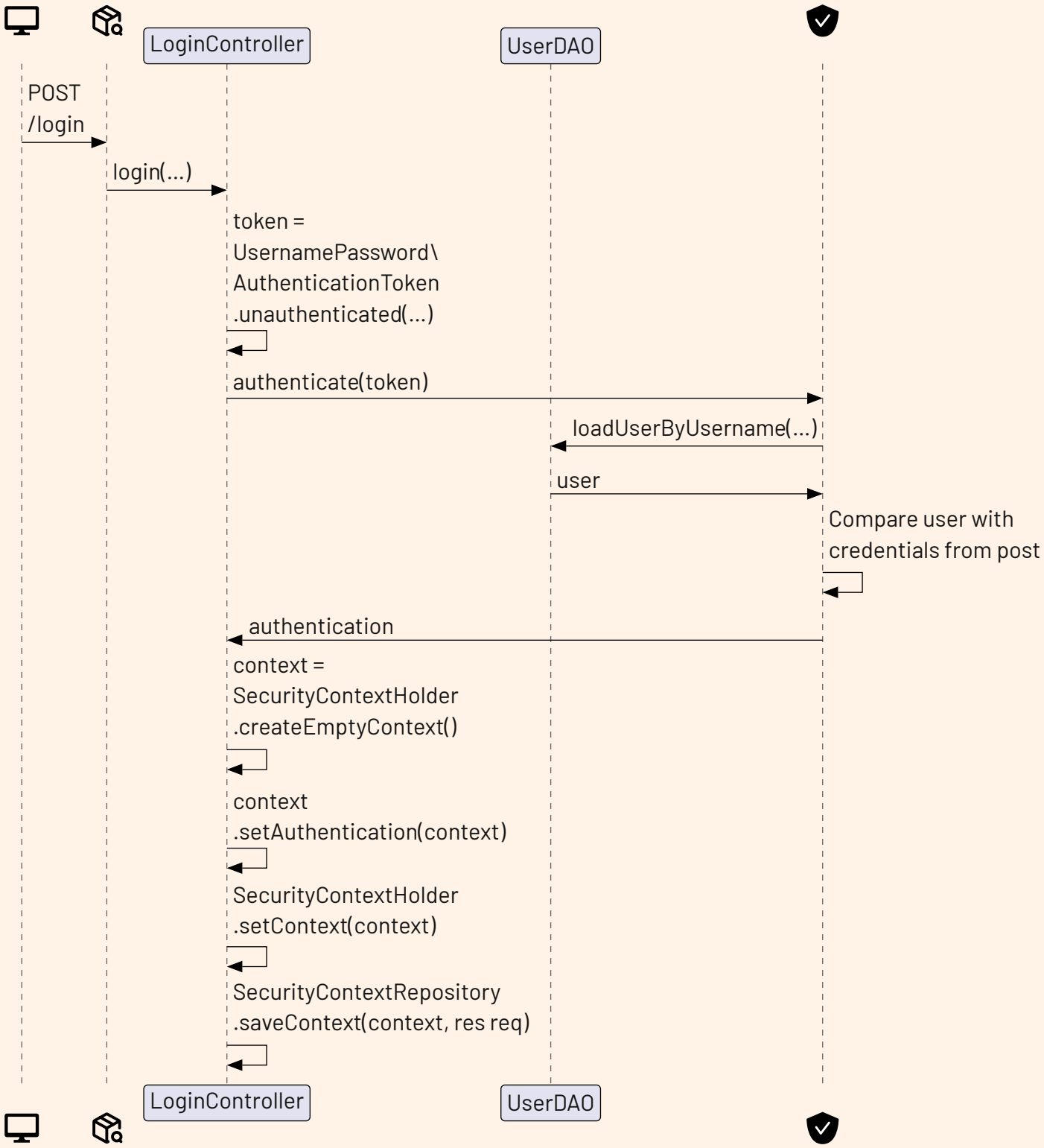
7.8 GetTrackById sequence diagram



7.9 GetTracks sequence diagram



7.10 Login sequence diagram



Comment

Like in the html version the user sends a post request to the /login mapping: this time its implementation is handled more manually.

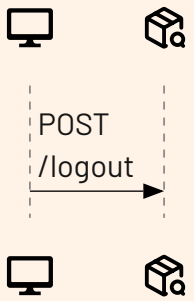
First, a token is generated from the username and password received from the request.

The token is then verified by SecurityContextHolder which calls loadUserByUsername() implemented by the same UserDetailsService from the HTML version.

SecurityContextHolder returns an Authentication object holding the results of the validation of the credentials.

The next chain of calls is basically boilerplate to finally store a valid context containing the authentication, inside SecurityContextRepository. This way the authentication is effectively stored.

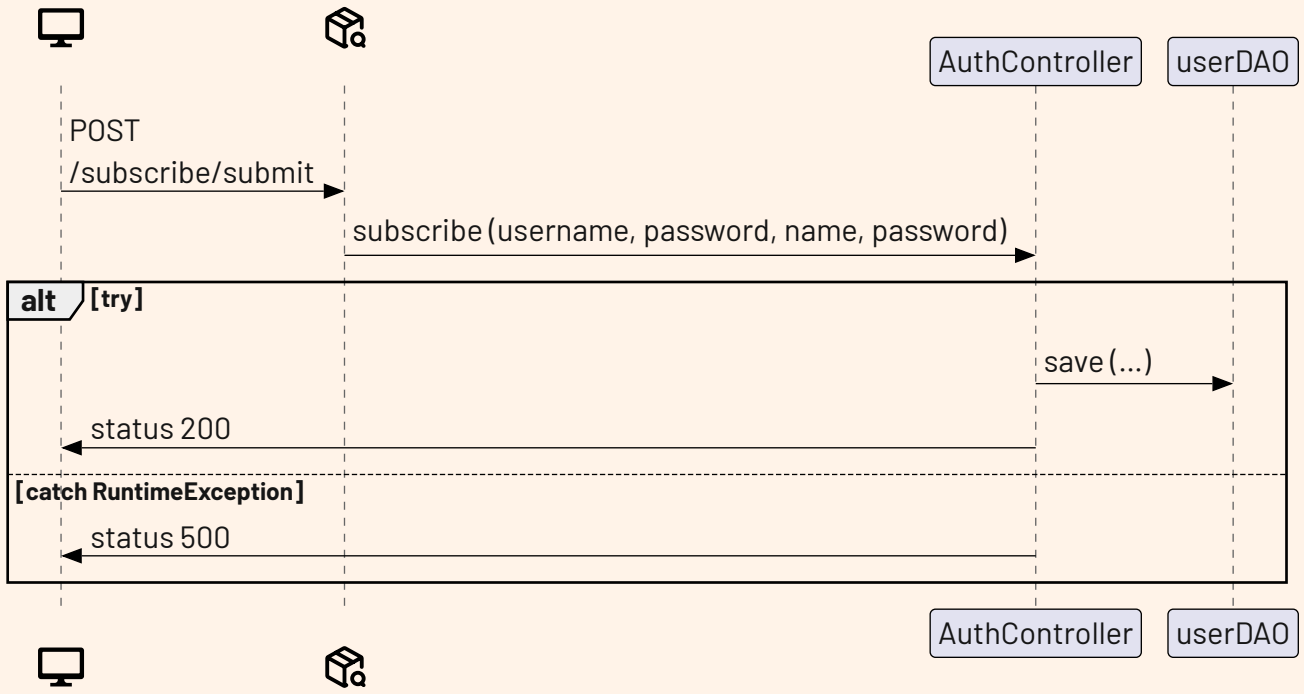
7.11 Logout sequence diagram



Comment

In order to perform a logout, Springboot requires the client to call the POST /logout mapping which internally invalidates session and all the other data associated to the user (of course the data inside the DB are preserved).

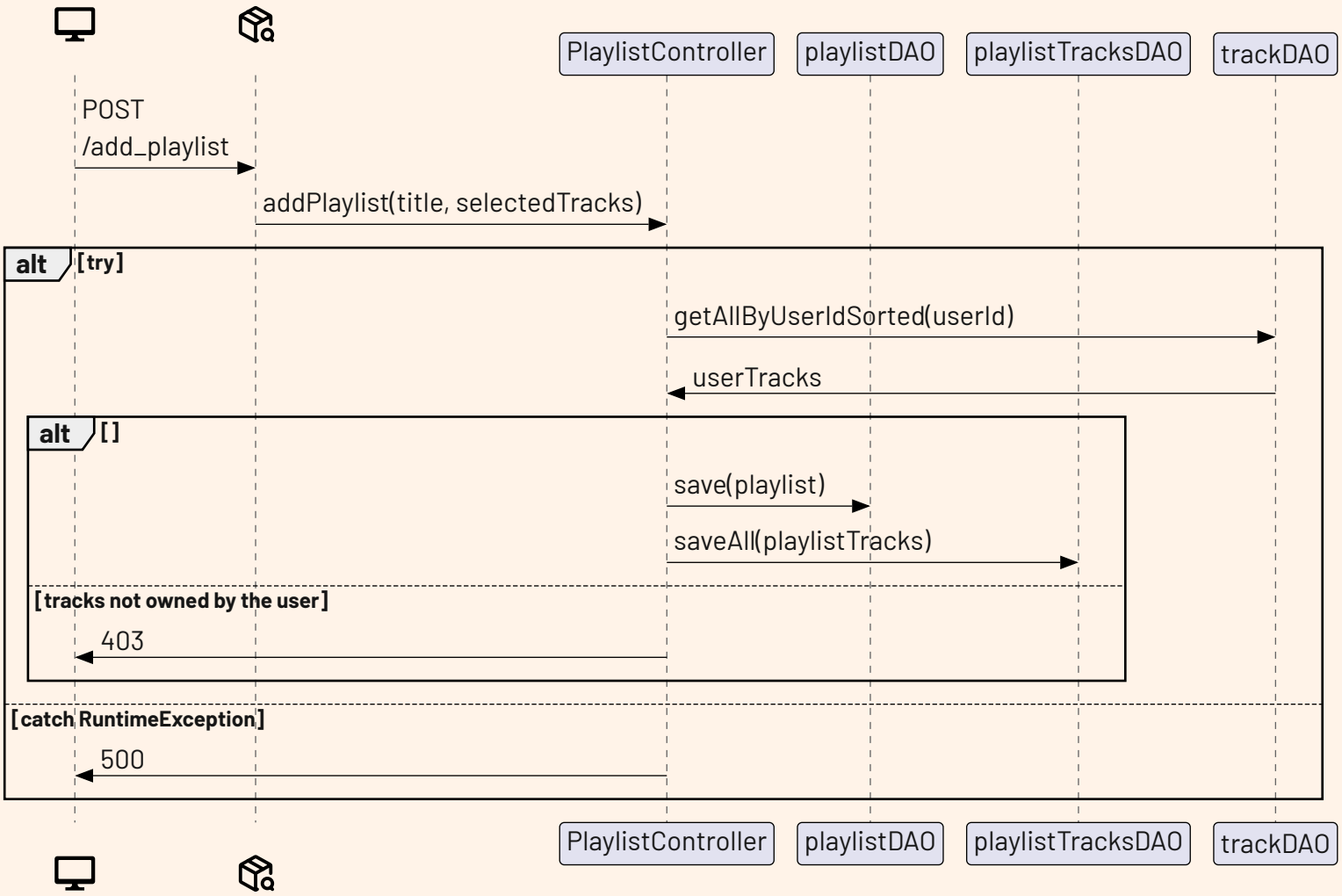
7.12 Subscribe sequence diagram



Comment

The client request is redirected to AuthController by DispatcherServlet: the controller tries to save the user by calling userDAO.save() and if any kind of RuntimeException exception occurs (so SQLExceptions are also considered), a response is sent back with a 200 status. Otherwise the response is sent back with a 500 status.

7.13 AddPlaylist sequence diagram



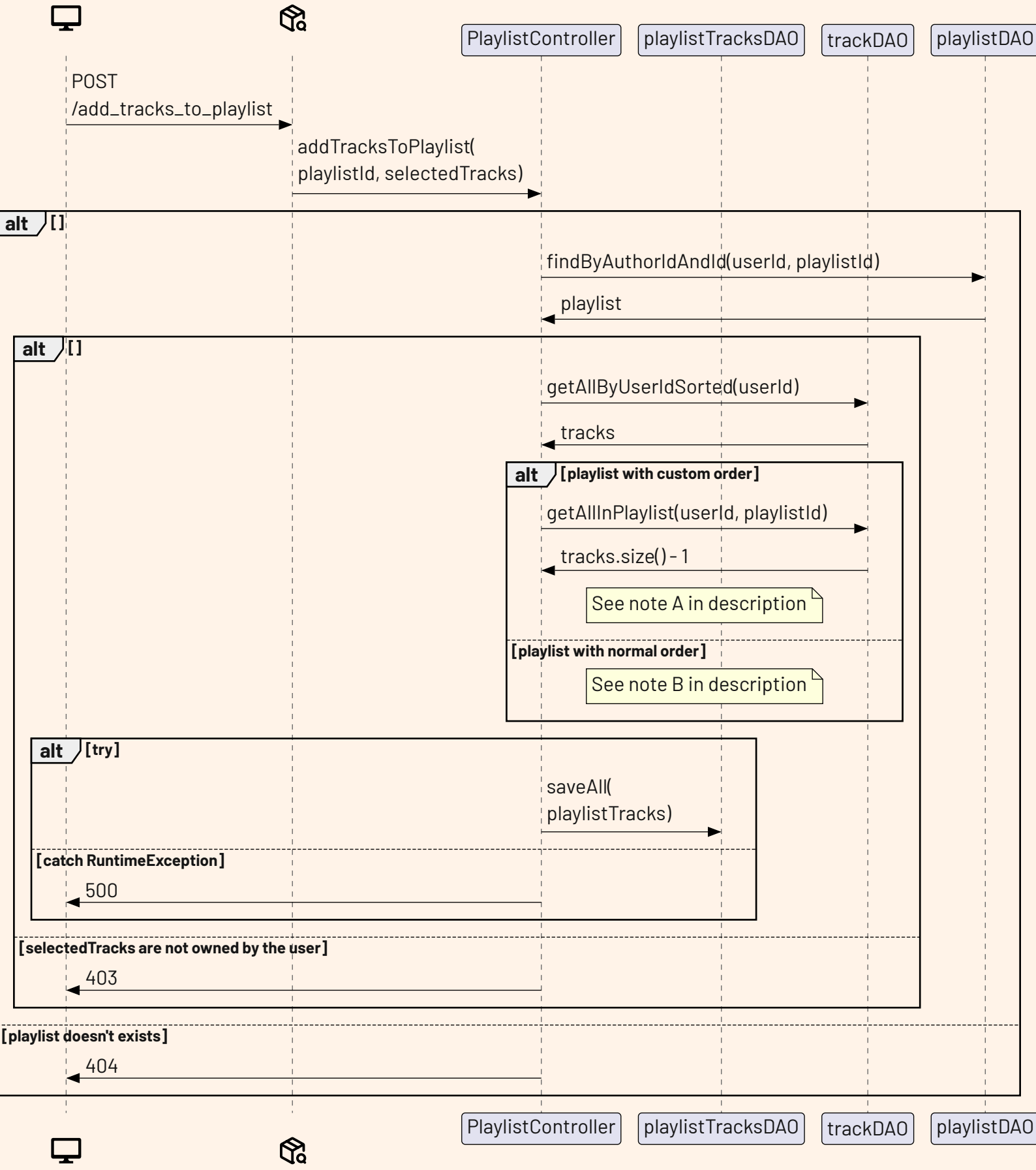
Comment

When the user loads a playlist, the request is handled by a method in PlaylistController which uses trackDAO, playlistDAO and playlistTracksDAO to store the data received.

It also checks if the user owns the ids of the tracks contained in selectedTracks. If at least one of them is not owned by the user, the request is rejected.

If a RuntimeException occurs or if everything went fine, the user is redirected to the home page.

7.14 AddTracksPlaylist sequence diagram



Comment

When the user adds tracks to a playlist, the request is handled by a method in PlaylistController which uses trackDAO, playlistDAO and playlistTracksDAO to store the data received.

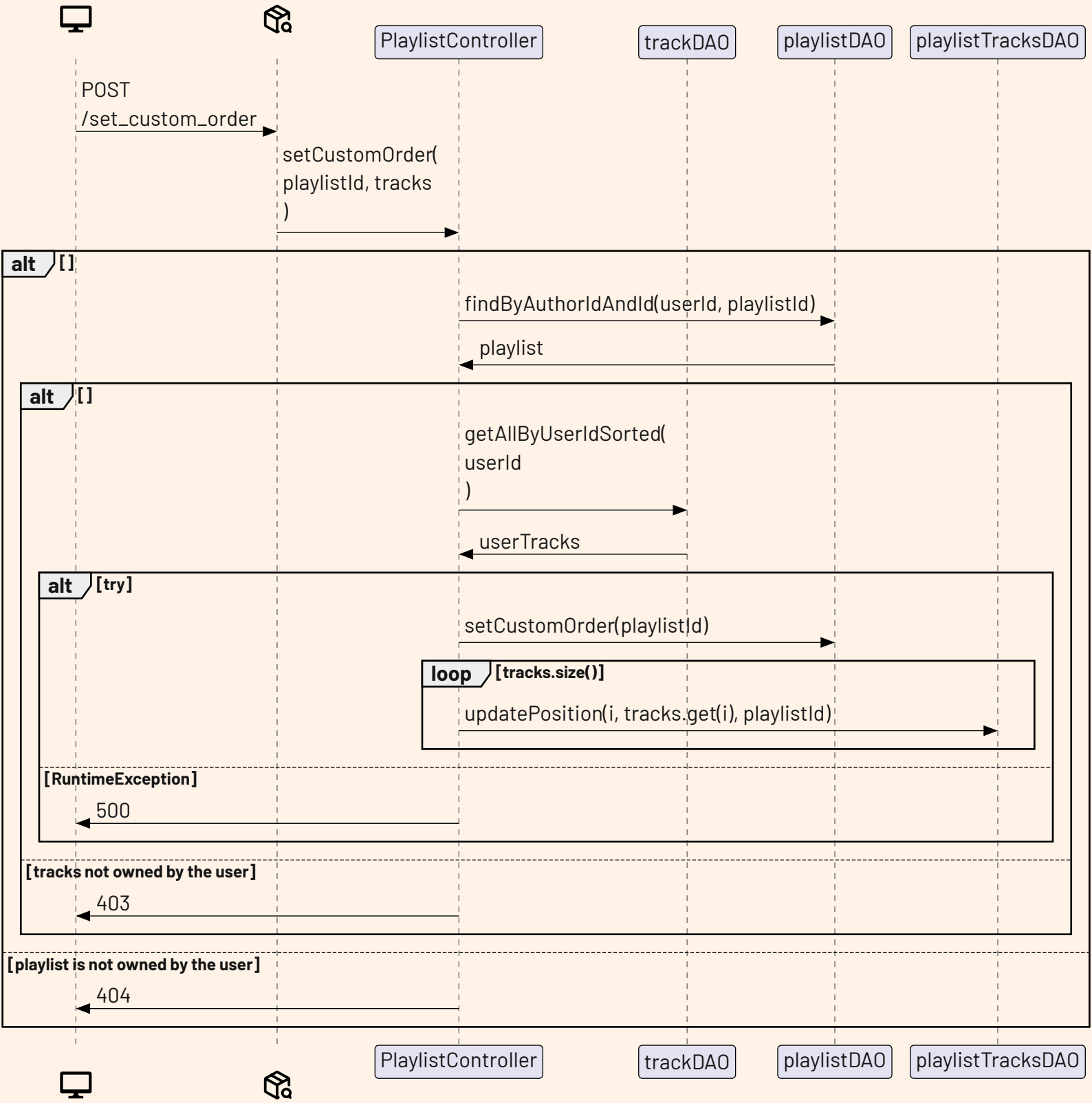
It checks if the user owns the ids of the tracks contained in selectedTracks and if he owns the playlist associated to playlistId. If at least one of them is not owned by the user, the request is rejected.

If the playlist has a custom order, then all tracks are inserted at the end of the playlist. Otherwise they are simply saved by setting their position to 0, since when a playlist is not custom ordered, that parameter is useless.

***Note A**: by knowing the size of a playlist is possible to create new playlistTracks by assigning to them the right position.*

***Note B**: the list of new playlistTracks is created by assigning a position of 0, since in this situation the position is not relevant.*

7.15 SetCustomOrder sequence diagram



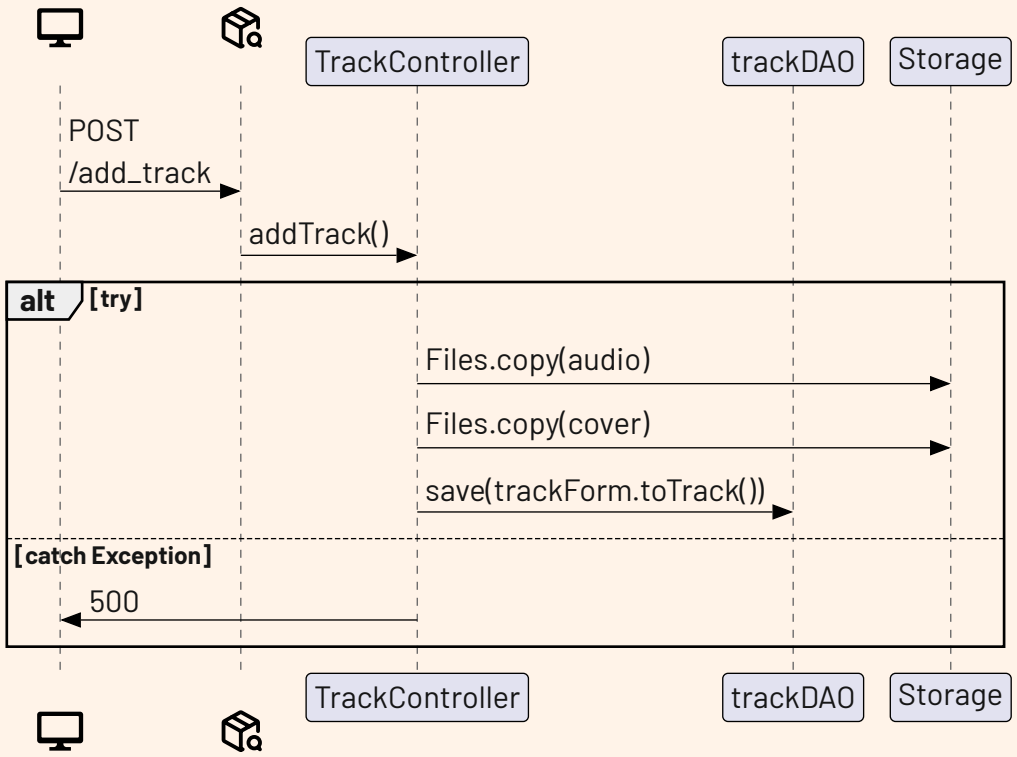
Comment

When the user reorders a playlist, the request is handled by a method in PlaylistController which uses trackDAO and playlistDAO to store the data received.

It checks if the user owns the ids of the tracks contained in selectedTracks and if he owns the playlist associated to playlistId. If at least one of them is not owned by the user, the request is rejected.

If everything went fine, the playlist is flagged with "custom_order" and the tracks receives.

7.16 AddTrack sequence diagram



Comment

When the user loads a track, the request is handled by a method in TrackController where the mime type of the loaded files is checked and if they are right, the track is also added to the database.

If a RuntimeException occurs the server throws an error to the client.