SWE Project 2

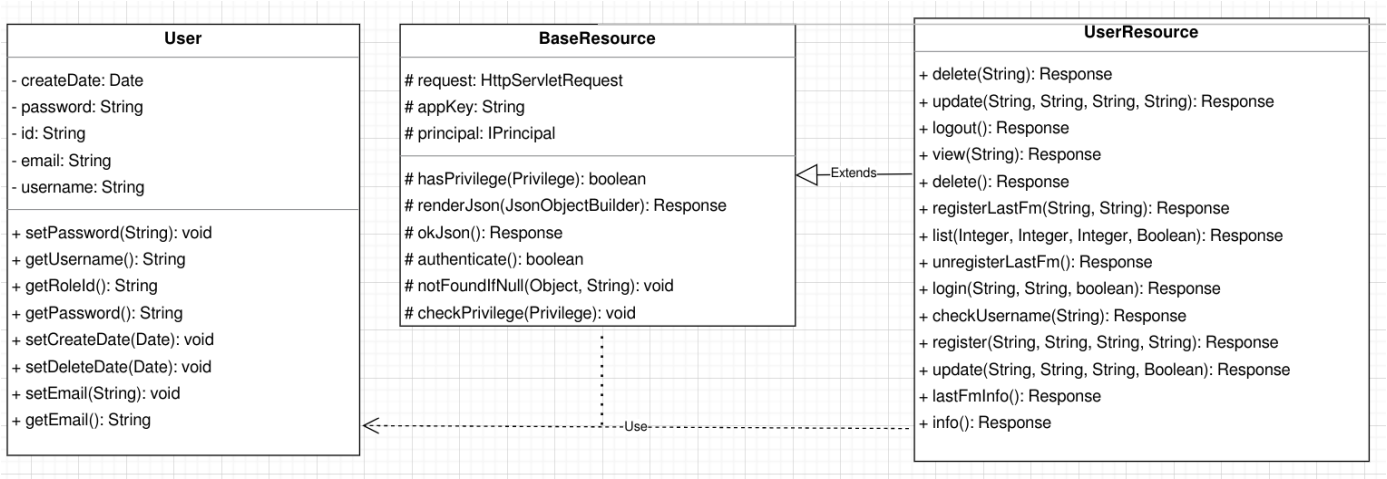
Team 14

Abhijith Anil - 2020101030
Jatin Agarwala - 2020111011
Pratyay Suvarnapathaki - 2020111016
Sneha Raghava Raju - 2020101125
Yash Mehan - 2020111020

Feature 1. Better User Management

1A. The Current System

The User Management Subsystem → Backend



Currently, a new user can only be registered by the admin. The ‘register’ method in the **UserResource** class uses the **checkPrivilege**(Privilege) method of the BaseResource class (UserResource extends from the BaseResource class) to check if the user is an admin before proceeding.

Then an instance of the User class is created and the username, password, etc variables are set using the User methods: setUsername(username), setPassword(password), etc. Then an instance of the UserDao class is created and its create(User) method is called. This method is used to hash the password and check if the username exists. UserDto and UserCriteria are passed as attributes to UserDao.

The Frontend Flow

The frontend is configured in the following manner:

- `…/main/webapp/partials` : HTML pages that define UI components
- `…/main/webapp/controllers`: Collections of JS code that adds interactivity to components. Controllers also invoke the Music REST API to execute operations on the backend.
- `.../main/webapp/app.js`: Defines the association between controllers and partials, in addition to handling UI states and frontend routes.

Relevant points for Feature 1:

- The partial `login.html` is controlled by `login.js` and transitions the UI state to `main.music.albums` upon login.
- Currently, only the admin can register new users. This is implemented in the `settings.user.html` partial, controlled by `SettingsUserEdit.js`, which either adds or edits a user. The former is accomplished by making a PUT API call to the `/user` backend route. This is managed, as stated above, by the `UserResource.java` file

1B. Our Implementation

Frontend Changes

- Created a new partial `myregister.html`, controlled by `MyRegister.js`.
- Added a 'Create an Account' button on `login.html`, which redirects to this new registration page.
- After the prospective new user enters their details and submits the registration form, the `MyRegister` controller makes a PUT request to the newly created registration-specific backend route, `/user/regNew`.
- Upon registration, the user is redirected back to the login page.

Backend Changes

- As mentioned above, a new API route, `/user/regNew` was specified in `UserResource.java`. A PUT request to this route handles user registration.

- The code for this new route is similar as that for a PUT request to `/user`, with the important distinction that the `checkPrivilege()` method is not called herein.

- The second and much more prominent change has been refactoring the user validation process, which now obeys the Chain of Responsibility pattern. The rationale for this change has been to separate concerns between the registration and validation logic.
  To make our point visually:

```
        // Validate the input data
        username = Validation.length(username, "username", 3,
50);
        Validation.alphanumeric(username, "username");
        password = Validation.length(password, "password", 8,
50);
        email = Validation.length(email, "email", 3, 50);
        Validation.email(email, "email");

        // Create the user
        User user = new User();
        user.setRoleId(Constants.DEFAULT_USER_ROLE);
        user.setUsername(username);
        user.setPassword(password);
        user.setEmail(email);
        user.setCreateDate(new Date());
```

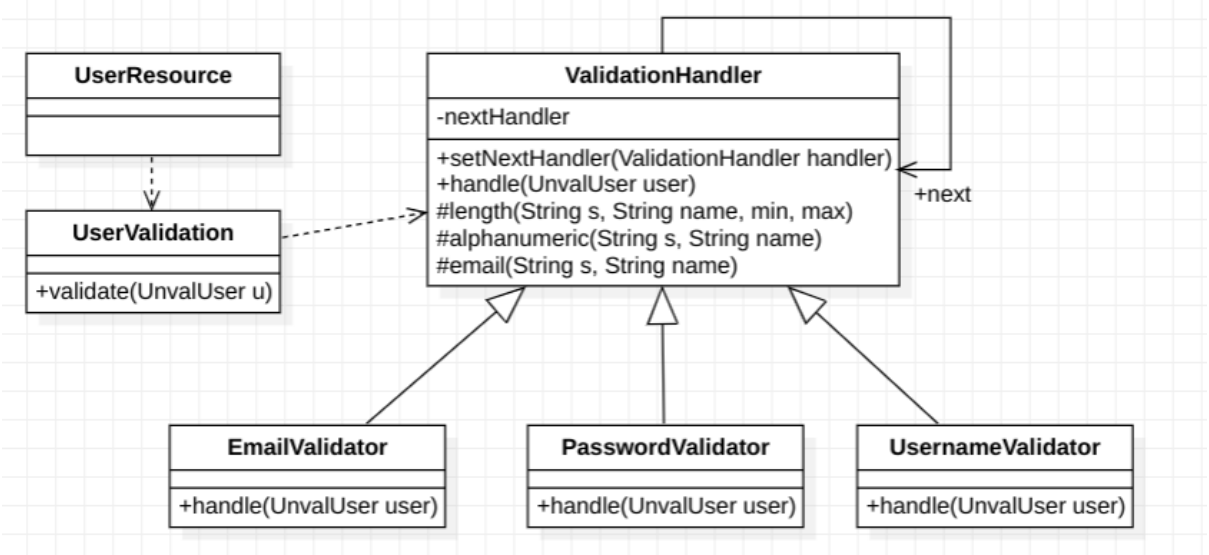
Before

```
UnvalUser unvalUser = new UnvalUser();
unvalUser.setUsername(username);
unvalUser.setPassword(password);
unvalUser.setEmail(email);
UserValidation.validate(unvalUser);
```

After

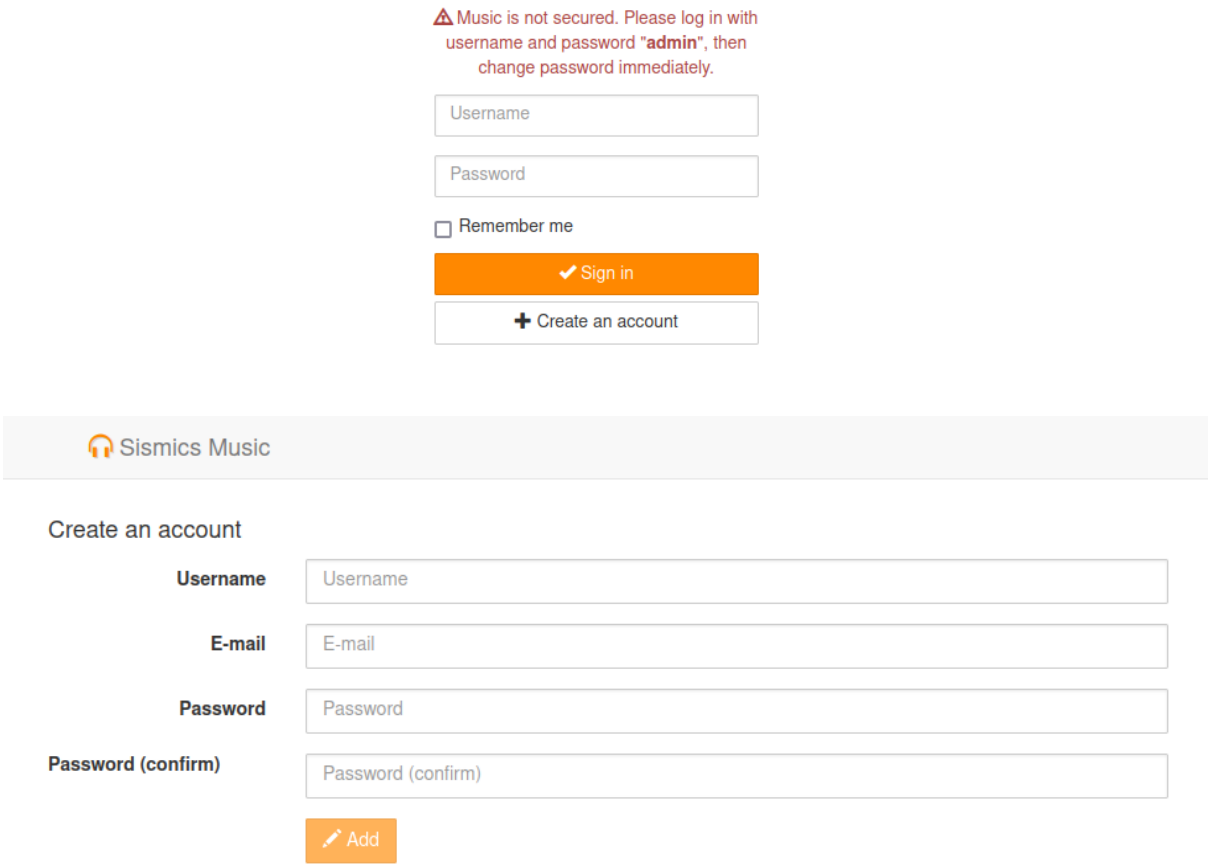
Design Pattern - Chain of Responsibility

- The Chain of Responsibility pattern has been used to codify the different stages of validation performed before a new user is registered. This modularises the process and makes it easy for other authentication methods (such as OAuth) to be added in the future. Here's how:



- The `ValidationHandler` defines core 'chaining' functionality (setting the 'next handler'. Also, this parent class features some basic primitive string validation methods that are common to all the currently implemented ConcreteHandlers.
- The ConcreteHandlers currently comprise a validator each for Email, Password and Username.
- This modular nature can accomodate other authentication methods too. After all, the only changes needed are adding the corresponding attributes to UnvalUser and operating on them by implementing a new ConcreteHandler.
- Additionally, the `validate()` method in `UserValidation` is what instantiates the handlers and defines the chain of responsibility.
- Upon validation, the new user object is created by instantiating the `User` class, just as normal.

1C. Demonstration



Feature 2. Better Library Management

2A. Playlists

1. Introduction

Originally, any playlist any user makes, is visible to every other user. The aim is:

- have two access control levels: public and private,
- For the owner to be able to specify the access control after creation of a playlist

While maintaining the existing functionalities of playlists.

2. Preliminaries

- The originally existing `T_PLAYLIST` table in the schema (`dbupdate-000-0.sql`) does capture the `USER_ID` of the user who made the playlist. This will come in handy while filtering the database for private playlists.
- `Playlist` class exists for the playlist entity, and has fields for name of the playlist, user ID of the creator, the unique ID of the playlist etc.
- `PlaylistCriteria` class exists already to contain details for what criteria are desirable while playlist filtering.
- `PlaylistDao` class exists, and has functions to build SQL queries and interact with the database. This is the only class to access the database, and all communication must happen via this class. The output of the queries is packaged as an object of the `PlaylistDto` class.
- `PlaylistMapper` is an implementation of the `ResultSetMapper` interface which maps a result set (output from the database) to a DTO.
- `PlaylistResource` class has the functions to deal with the REST backend routes.

3. Backend Methodology

1. Since there is a need for multiple access control levels, and for extensibility purposes, an enum is an appropriate choice, for the reasons of readability and ruling out human errors while coding and enforcing only valid values. Hence a file dedicated to enum `PlaylistVisibilityEnum` was created in the directory for constants and configs.
2. Modifications in database schema. The table `T_PLAYLIST` was modified to have an additional column `VISIBILITY`, which is public by default, and non null.

3. A new member variable `visibility` of type `PlaylistVisibilityEnum` was added in the `Playlist` class. Supporting getter and setter were added. The constructors of this class were updated to have the default value of `visibility` to `PlaylistVisibilityEnum.PUBLIC` at the time of object instantiation.
4. Since `PlaylistCriteria` class' objects are supposed to contain data to be passed to filters, a similar `visibility` variable and its getter and setter need to be added.
5. Since DAOs access the DB, the query building needs to be appropriately modified. `VISIBILITY` from the `T_PLAYLIST` table is `SELECT`ed as well. Similar care has to be taken for the `WHERE` clause as well. The `criteria` object contains such information.
6. Users will need to update visibility as well, post making the playlist as well. Hence the need for an `updateVisibility(Playlist)` function.
7. Introducing the `visibility` member variable and its getter and setter in the `PlaylistDto` class. This setter will be used to feed in values in the DTO from `PlaylistMapper`'s `map()` implementation.
8. `PlaylistResource` Class needs a function `changeVisibility()` which will actually handle the update visibility request from the user from the frontend. This makes up an object of `PlaylistCriteria`, feeds it into a `PlaylistDto`, and calls `Playlist.setVisibility()` function on it, which deals with the DAO and further the DB under the hood.

4. Frontend Methodology

The frontend was implemented through 3 main files: html, controller and resource.
For playlist visibility radio buttons were added to the playlist view, allowing users to set the playlist as 'public' or 'private'.

The layout of the playlist frontend is implemented in **playlist.html** inside webapp/src/partial. Using the **ng-change** functionality in angular, when the user selects public or private, the change in value triggers a call to a function called **changeVisibility(value)** with value (i.e public or private) passed as a parameter.

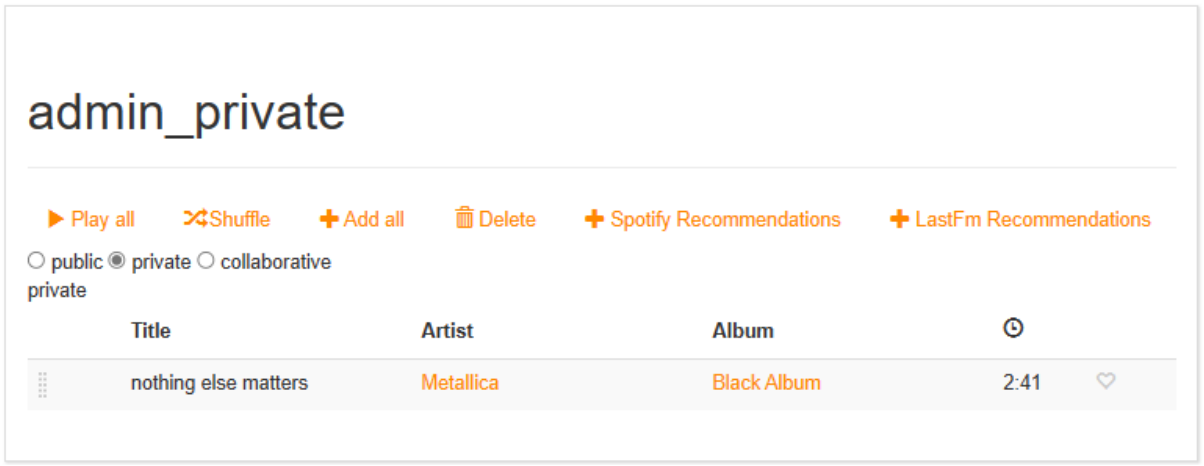
The **changeVisibility** function is implemented in the controller file: **Playlist.js**, found in **webapp/src/app/controller**. The restangular service of Angular.js is used to send a POST request to the endpoint **/playlist/:id/visibility**. Here id is the playlist ID and {visibility: value} is the request payload that contains the new visibility value.

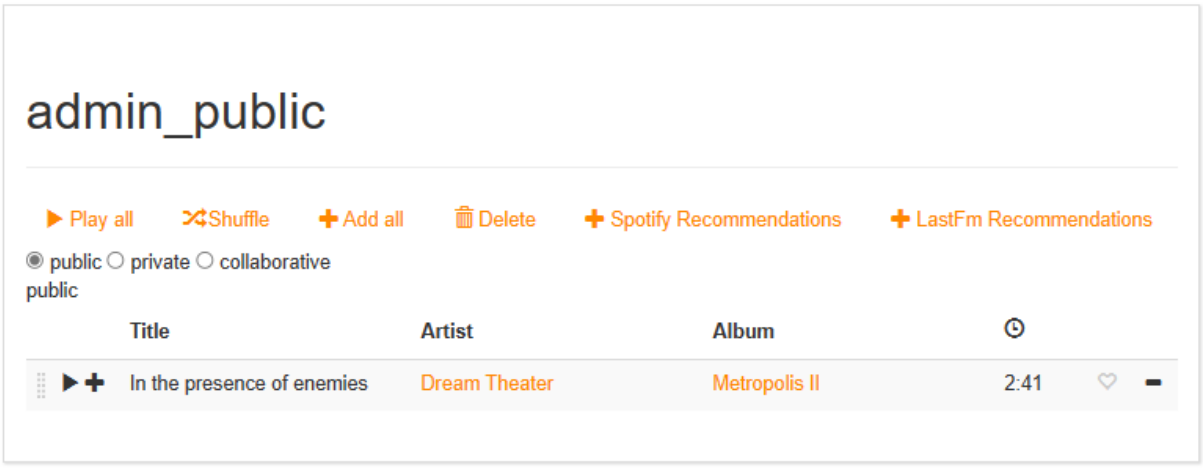
The POST request is defined in **PlaylistResource.java**. This has a function called **changeVisibility()** that actually handles the playlist visibility change.

5. Design Pattern

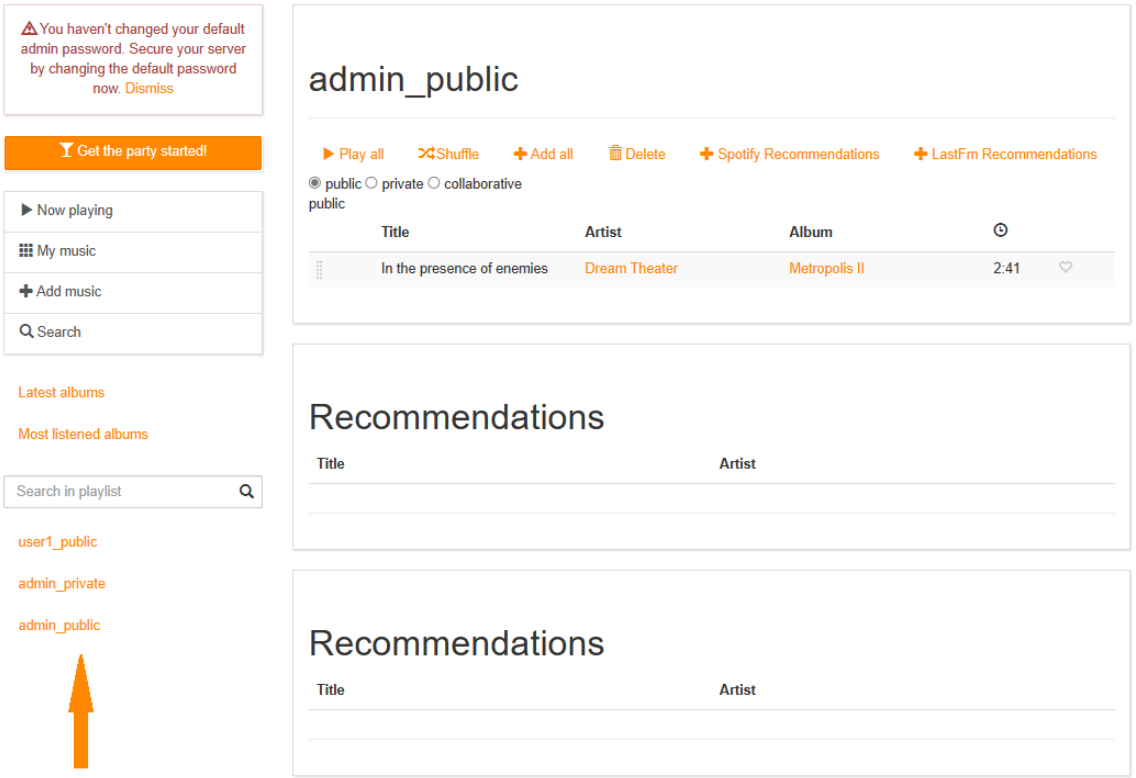
The design choice using DAO classes to interact with the database and DTOs to transfer data around to multiple classes can be considered as a design pattern. However, none of the design patterns taught in class like Chain of Responsibilities, Observer, Factory, etc. were found to fit in this task.

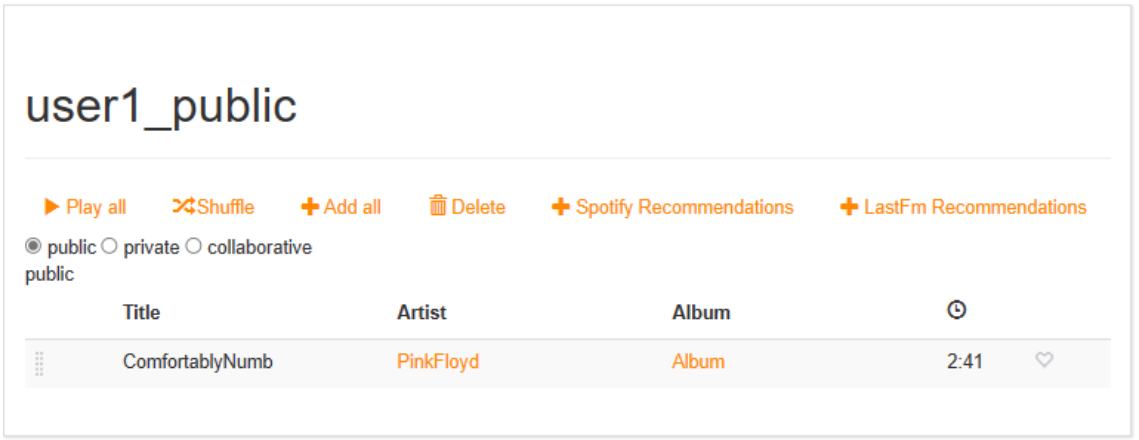
6. Demo

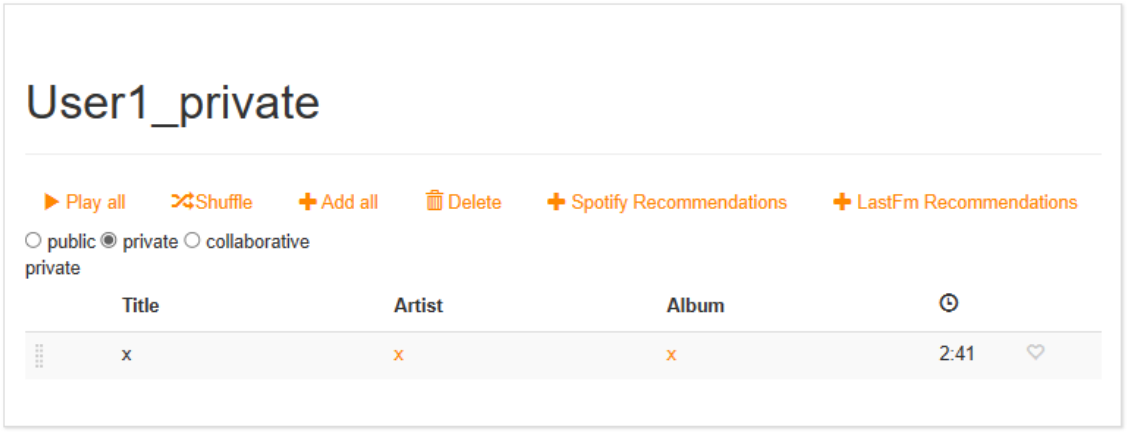


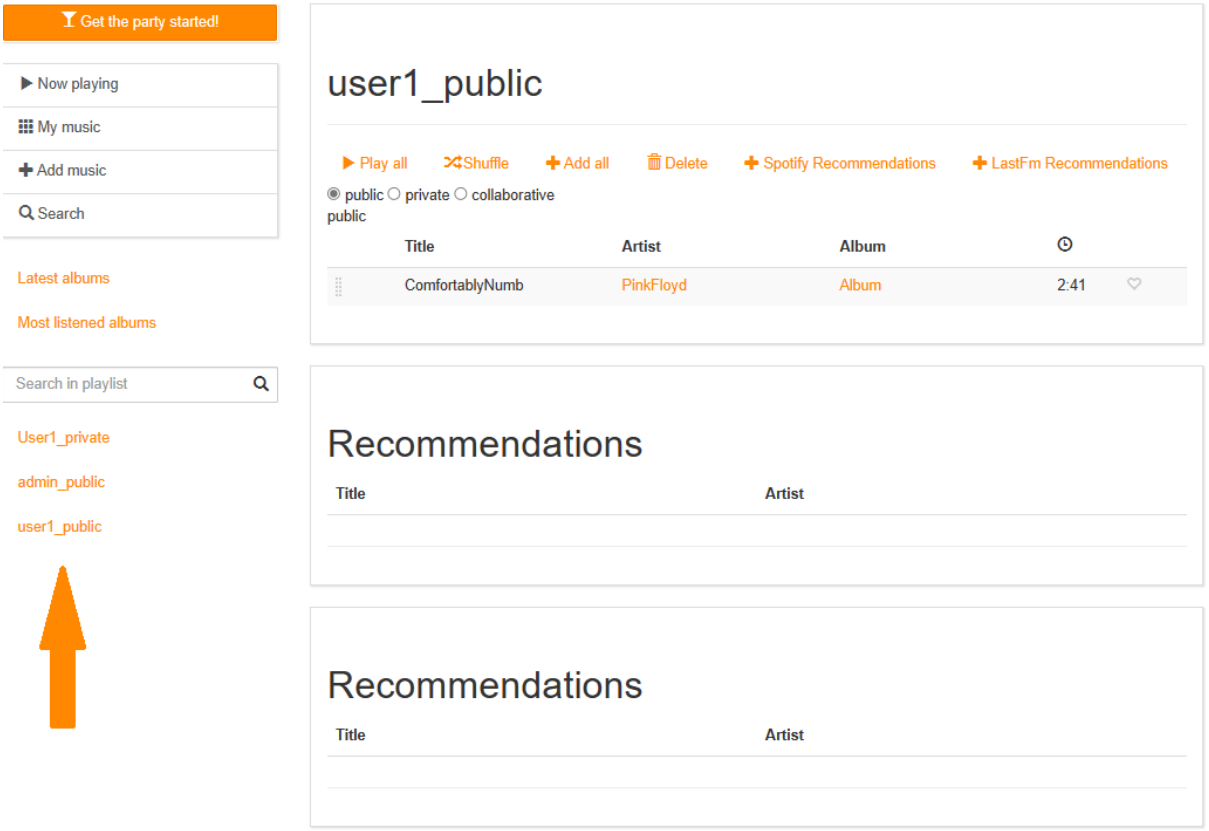


Playlists view for user1:


You haven't changed your default admin password. Secure your server by changing the default password now. Dismiss
Get the party started!
Now playing
My music
Add music
Search
Latest albums
Most listened albums
Search in playlist
user1_public
admin_private
admin_public
admin_public
Play all
Shuffle
Add all
Delete
Spotify Recommendations
LastFm Recommendations
public private collaborative
public
Title
Artist
Album
In the presence of enemies
Dream Theater
Metropolis II
2:41
Recommendations
Title
Artist
Recommendations
Title
Artist


user1_public
Play all
Shuffle
Add all
Delete
Spotify Recommendations
LastFm Recommendations
public private collaborative
public
Title
Artist
Album
ComfortablyNumb
PinkFloyd
Album
2:41


User1_private
Play all
Shuffle
Add all
Delete
Spotify Recommendations
LastFm Recommendations
public private collaborative
private
Title
Artist
Album
x
x
x
2:41



2B. Tracks

1 Introduction

Making tracks private was slightly more involved because all the user imports/uploads are tracks, and the user's music is grouped by albums/artists while displaying. The user does not have an option to view an exhaustive list of all tracks. Another potential issue was to decide the behaviour when private tracks are put in public playlists. Theoretically private tracks in a public playlists should not show up for the non-owners, because the "privateness" of a track is a more "binding" trait and that trait being attached to a track makes it more "granular".

But this issue does not arise as the specification asks for all tracks to be private. So, the only time a user can see someone else's uploaded tracks is when the owner puts it in a public playlist.

2. Preliminaries

It is necessary to highlight the entire process of uploading and indexing, for this section borrows heavily from this process. All the routes that the frontend send REST requests to are mapped in `*Resource.java` files. For example, The landing page after login in My Music, which sends a request to `/api/album/list`. This calls the `public Response list(...)` function in `AlbumResource.java`

The importing process is as follows:

1. The backend receives a request in `/import/upload`, which calls the `public Response upload(...)` function. This function, calls another function which after basic file type and byte stream checks, copies the audio file from the specific source location into `/tmp/importaudio/upload<radnom_number>.mp3`
2. There exists a class `CollectionWatchService`, which makes the parent directory a `watch`able event. It uses multithreading and employs one thread continuously to `watch`. When a new file is added into the parent directory, the `watchEvent` is triggered and one iteration of the `run()` function runs. This iteration calls the `indexFile()` function from `CollectionService`
3. The `indexFile()` goes through the parent directory and indexes the files present there again. For a newly found track (which was freshly imported), it instantiates a `trackDao` and pushes it into the `T_TRACK` table in the database. This function also reads the metadata of the track and ascertains the album and the artist name. On that basis, new entries are made into the `T_ALBUM` table.

What makes this feature challenging is that the user uploads tracks, so the userID of the uploader can be captured during the time of uploading. But during display time, the tracks are never listed, the albums are listed. When the user clicks the icons for a specific album, only then the `T_TRACKS` table is queried for the specific albumID. This calls for the AlbumDao to have access to the UserID as well. The userID is present as the `principal.getId()` function in the `BaseResource.java` class, and its children classes. It is not accessible to `CollectionService`., and hence the insertion into Albums table cannot have the UserID.

3. Backend Methodology

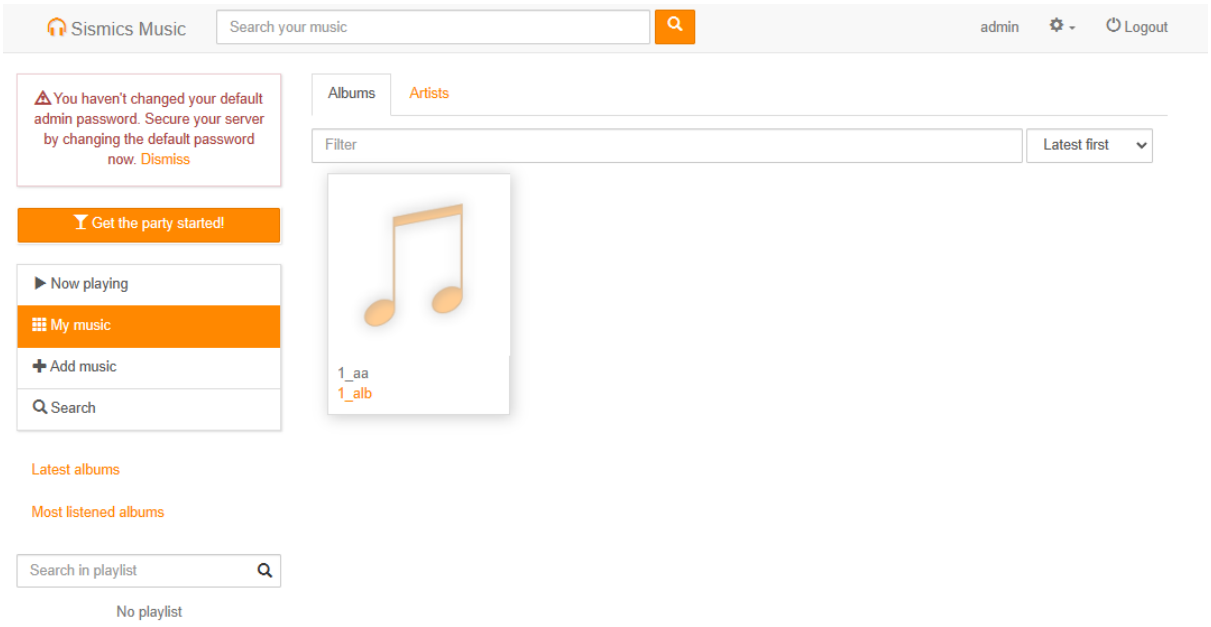
1. For the userID to be available for Album creation, we need to have the userID available to most functions in the backend. It is intuitive that since there is one instance of the `AppContext` class, there can be a private variable for `userID` which is initialised to be `null` but can be set and unset via `login()` and `logout` functions.
2. Changes to database schema: added field `USER_ID` to `T_ALBUM` and `T_TRACK`
3. Modified the track related classes `TrackDao`, `TrackMapper`, `Track` to accommodate the new field.

4. Modified the album related fields `AlbumDao`, `AlbumDto`, `AlbumDtoMapper` `Album`, `AlbumResource` to accommodate the new field. (the roles and how one should modify these files are elucidated in the playlist section)
5. Modifying the readTrackMetadata function which calls the creation of a new row in the Album table, to get the userID from `AppContext` and send it to `AlbumDao`'s create function.

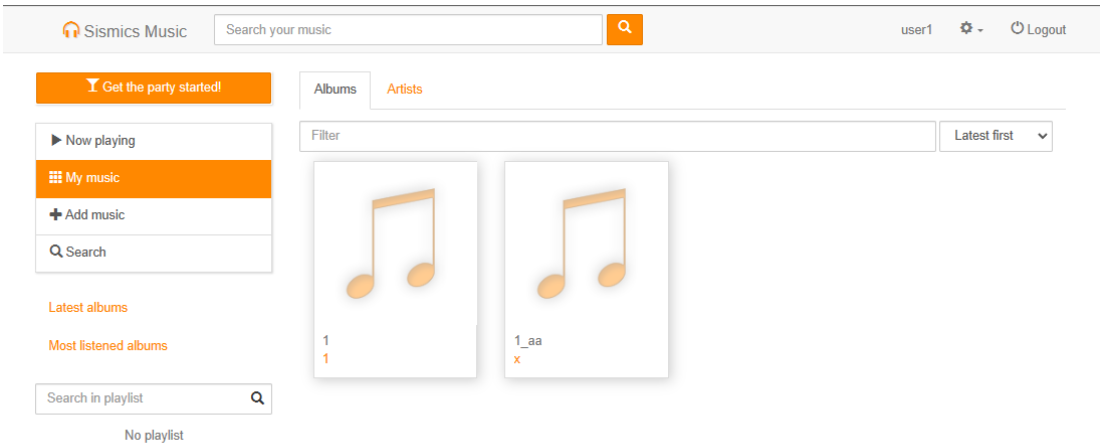
The frontend was not changed for this task. When the `AlbumReosource`'s `list()` function is called, it queries the `T_ALBUM` and filters the response with the current user's `userID`. And so, songs not belonging to user A are not sent to user B's frontend.

4. Demo

Admin's My Music page:



User 1's My music Page:



A Note On Spotify

The Online Integration through Spotify requires the use of Spotify Web API calls to access the data present in Spotify. However the Spotify API calls require us to pass an OAuth Access Token. The problem that arises is that OAuth Tokens expire in an hour. Due to this, we have to ensure that our app can create new access tokens when making API calls, instead of relying on tokens created once by us.

The Spotify documentation has 4 OAuth flows recommended for different use cases. These include:

1. Authorization code
2. Authorization code with PKCE
3. Client credentials
4. Implicit grant

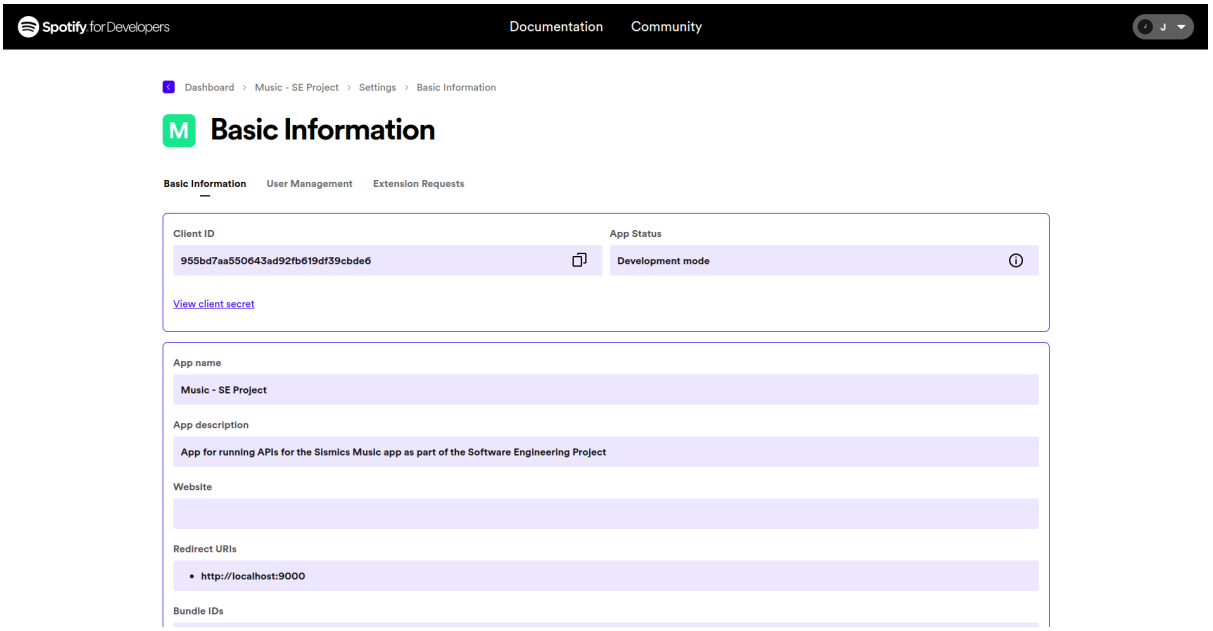
Since Spotify does not require users to login to access general data, Search and Recommendations can be carried without having a Spotify user's login credentials. This in turn also implies that we do not need to access a specific user's resources. This allows users of our app to use Spotify features without having to login. Based on these specifications and requirements, the best OAuth flow is the **Client Credentials** flow.

Spotify documentations describes the Client Credentials OAuth flow in the following manner:

"For some applications running on the backend, such as CLIs or daemons, the system authenticates and authorizes the app rather than a user. For these scenarios, Client credentials is the typical choice. This flow does not include user authorization, so only endpoints that do not request user information (e.g. user profile data) can be accessed."

In order to carry out this flow, an App needs to be made in the Spotify Developers website to gain the required Client ID and Client Secret.

Therefore, the following app "Music - SE Project" was made.



Now, before we make a Spotify Web API call from the Music App, we make a separate call using the Client Credentials to get an OAuth token. This OAuth token is used for the Authorization of the API call, thus allowing the user to access Spotify's features like Search and Recommendations without having to login, which simultaneously allows the API calls to be made without manual intervention from the developers of the Music app.

Feature 3.1: Search

3.1.A. Current System

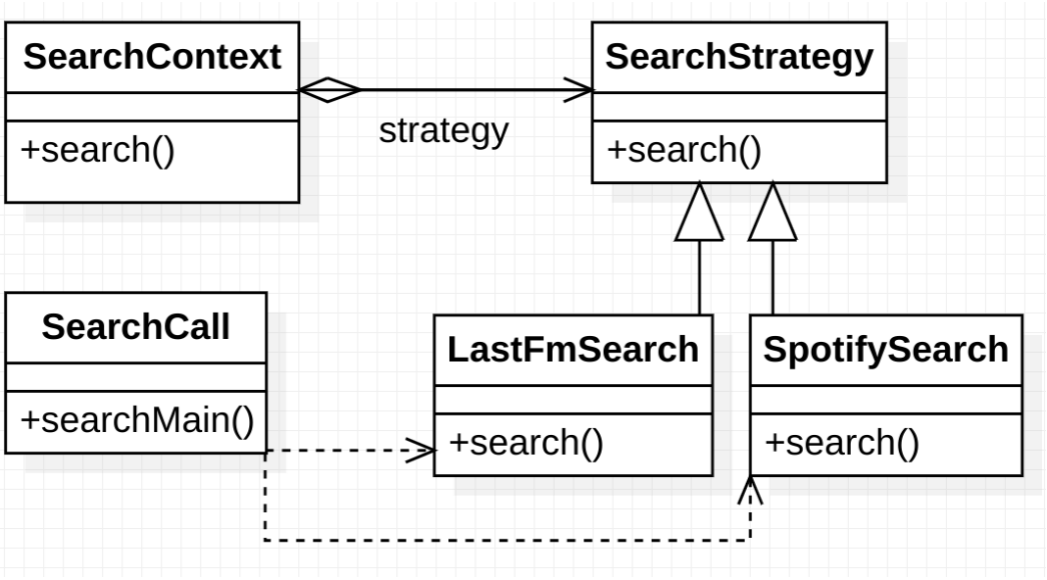
The current Search system in Music can search only for the songs that have been uploaded manually by the user.

3.1.B Our Implementation -

Design Pattern

The Design Pattern used to implement Search in Music is Strategy Pattern.
The Strategy Pattern was chosen because there are multiple algorithms that need to be implemented for Searching for a song. The Strategy pattern allows us to change the algorithm based on the client's needs.

The implementation of the Design Pattern can be understood from the UML diagram shown below:



The functionalities of the classes are as follows:

SearchStrategy:

- It provides the interface that is common to all algorithms. Here, it is the search() function that takes as input String trackName, and returns a JsonObject
- The class object is used by SearchContext

SpotifySearch:

- Implements the search() function through the Spotify API

SearchContext:

- Configures the required strategy with a SearchContext object
- Contains within itself the reference to the corresponding SearchStrategy
- Defines the interface to call the corresponding SearchStrategy and provide it with the required data (String trackName)

SearchCall:

- Called when the search button is clicked on the web app interface
- Acts as a pseudo main for the Search Strategy
- Contains the searchMain() function that creates the SearchStrategy and SearchContext objects and calls the searchContext.search() function

Spotify

The Spotify SearchStrategy has been implemented in the `public JsonObject search(String trackName)` method of `class SpotifySearch` which implements `SearchStrategy`.

The Spotify Search API requires the following parameters:

- Query: The name of the track being searched for
- Type: The type being search for (here, it is track)
- Limit: The maximum number of search results (here, it is 10)
- OAuth Token: Access token needed to make API Calls

First, the OAuth Token is generated using the Client Credentials flow. Once the OAuth token has been procured, the Spotify Web API call for Search is made using HttpURLConnection. The output is read using a BufferedReader, which is then converted into a single string. This is stored in the variable `response`, which is then converted to a JSON object using the `javax.json` library functions. This jsonObject is finally returned for further processing and display in the front end.

LastFM

Similar to Spotify, the Last Fm Search Strategy is also implemented in the `public JsonObject search(String trackName)` method of `class LastfmSearch` which implements `SearchStrategy`.

To make the search call easier by passing nothing but the trackname and the number of songs to return, an additional method `searchTrack(String trackName, int limit)` was created in the class LastfmService. This method calls the `Track.search()` method that is implemented by the api, returning the top limit search results for the trackname passed to it.

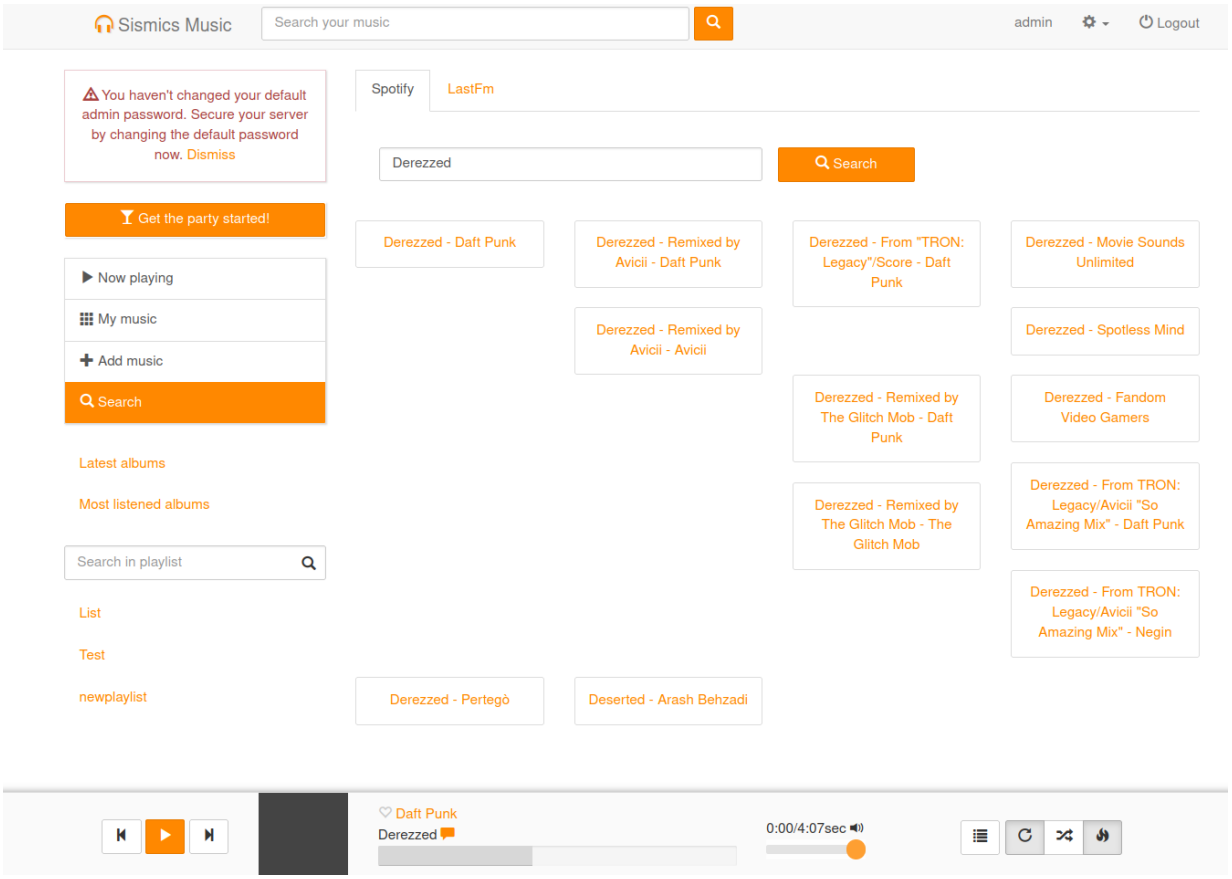
The `search()` method in the `Track` class requires the following parameters:

- Artist: The name of the artist who made the track (we pass null as the artist, as we are searching based on the name of the song)
- Track: The name of the song to be searched for (searchTrack() passes trackName as the track parameter)
- Limit: The number of search results to return.
- apiKey: The Lastfm api key to be used for authentication.

Frontend

In general the frontend for the features was implemented through 3 main files: html, controller and resource. For external search a search button was created that redirects the user to a page with two tabs; one for Spotify search and one for LastFm search.

Spotify Search



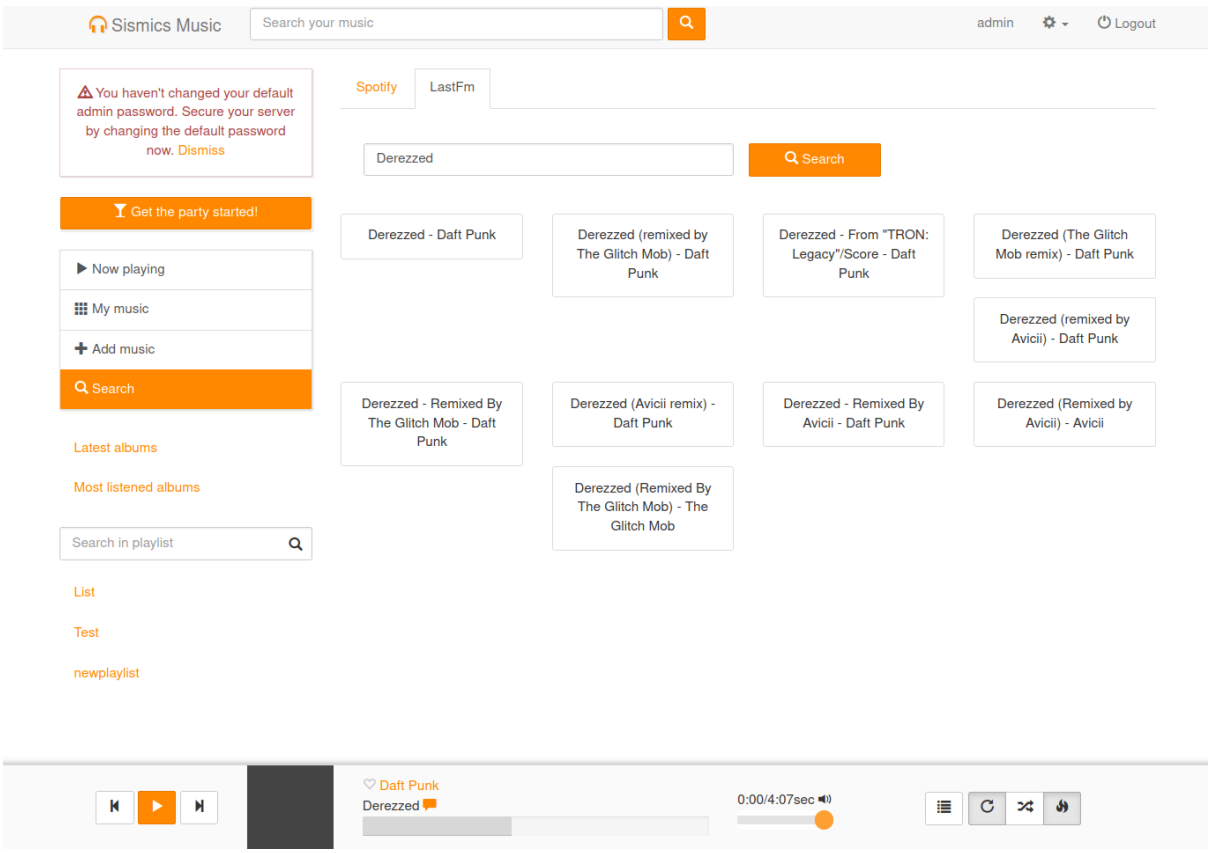
As shown above the Spotify search returns the tracks (name and artist) as well as a link to the track on Spotify.

The layout of the spotify search frontend is implemented in **externalsearch.spotify.html** inside webapp/src/partial. Using the **ng-click** functionality in angular, when the user enters their search query and clicks the search

button the **spotifySearch(query)** function is called with the query passed as a parameter.

The **spotifySearch** function is implemented in the controller file: **Spotify.js**, found in **webapp/src/app/controller**. The restangular service of Angular.js is used to call the GET request. The path for the request is routed at **/external/spotify-search**. Inside the spotifySearch function the results of the data returned by the api call are stored in a variable called **results**, which is then used in **externalsearch.spotify.html** to display the tracks.
The GET request is defined in **ExternalSearchResource.java**. This has a function called **spotifySearch()** that passes the query to **SearchCall** by calling **searchMain(0, query)**. Here '0' indicates that the search is Spotify (1 for LastFm) and passed the query string.

LastFm Search



The LastFm search is implemented in the same way as the Spotify search. The relevant files are: **externalsearch.lastfm.html, LastFm.js, ExternalSearchResource.java.**

Feature 3.2: Recommendations

3.2.A. Current System

There was currently no Recommendations system implemented by the Music app.

However, we were asked to implement the recommendations feature for the playlists in the app. The current system for Playlists, therefore, becomes a key part for the implementation of this feature. The current Playlist system was implemented in the `class PlaylistResource` that was extended for the implementation of this feature.

The `PlaylistResource` class had a method named `listTrack()` which returned a response having all the tracks in a particular playlist. To facilitate getting recommendations, we created a modification of this method named playlistTracks(String playlistId) `playlistTracks(String playlistId)` which returns a JSON object of all the tracks in the playlist with the corresponding id. We used the JSON object returned by this method in the `getRecommendation()` method, iteratively going through each track in the playlist and getting its top recommended track from Lastfm or Spotify api. Note that these recommendations depend on the details of the song like Title, Artist etc, which are provided to the app by the user when uploading songs to Music. It is not possible to extract recommendations from the content of the song. Therefore, songs details need to be provided accurately for proper functioning of the recommendations feature.

It is also interesting to note that the Recommendations provided by LastFM and Spotify have significant differences. A primary difference being that the recommendations by Spotify are different every time the API call is made, whereas it is the same every time for LastFM. Since we had to extend an already existing file instead of making new ones, and due to the differences in the mechanism by which LastFm and Spotify recommend songs, we decided that it would be better to not use a design pattern even though we did so for the Search feature.

3.2.B Our Implementation -

Spotify

The Spotify Recommendations API requires the following parameters:

- limit: The target size of the list of recommendations
- market: The country in which the content should be available (IN here, for India)
- Seed_artists: Spotify ID of the artist used as seed for recommendations
- Seed_genres: Comma separated list of genres used as seed for recommendations
- Seed_tracks: Spotify ID of the track used as seed for recommendations
- OAuth Token: Access token needed to make API Calls

Since the size of the list of all seeds cannot exceed 5, we are limited to recommendations from single tracks. The manner in which Spotify recommendations have been implemented is similar to the Enhance feature present in Spotify. We use individual songs for generating recommendations, and include the artist, and the artist's genres as seeds for the recommendations. This had to be done since Spotify API does not provide genres for individual tracks, but only for artists. Each API call therefore contains the following:

- 1 seed track
- 1 seed artist
- 1 to 3 seed genres

First, the OAuth Token is generated using the Client Credentials flow. Once the OAuth token has been procured, then the following is done for every track in the playlist:

- The Spotify Web API call for Search is made using HttpURLConnection. This is done to get the information like trackID and artistID for the track.
- The output is read using a BufferedReader, which is then converted into a single string.
- This is stored in the variable `response`, which is then converted to a JSON object using the `javax.json` library functions.
- Then, the Spotify Web API call for getting Artist info is used to get the artist's genres using another HttpURLConnection.
- Finally, the Get Recommendations API call is made by combining all the information gathered above. The recommended track is stored into a

JSONArray, which is a part of the JSONObject finally returned to the Frontend for displaying to the user. Note that the value of the limit parameter is set to 1

- Thus, the number of tracks recommended is equal to the number of tracks present currently in the playlist.

LastFM

To make getting recommendations easier, a method `getRecommendations()` was added to the class LastfmService. This method interacts with the Lastfm api and returns the list of recommended songs. This method has the following parameters:

- Artist: a string variable used to get the name of the artist who made the track
- Trackname: string that has the name of the song to get recommendations for.
- Limit: This parameter is used to denote how many results (tracks) to return as recommendations.

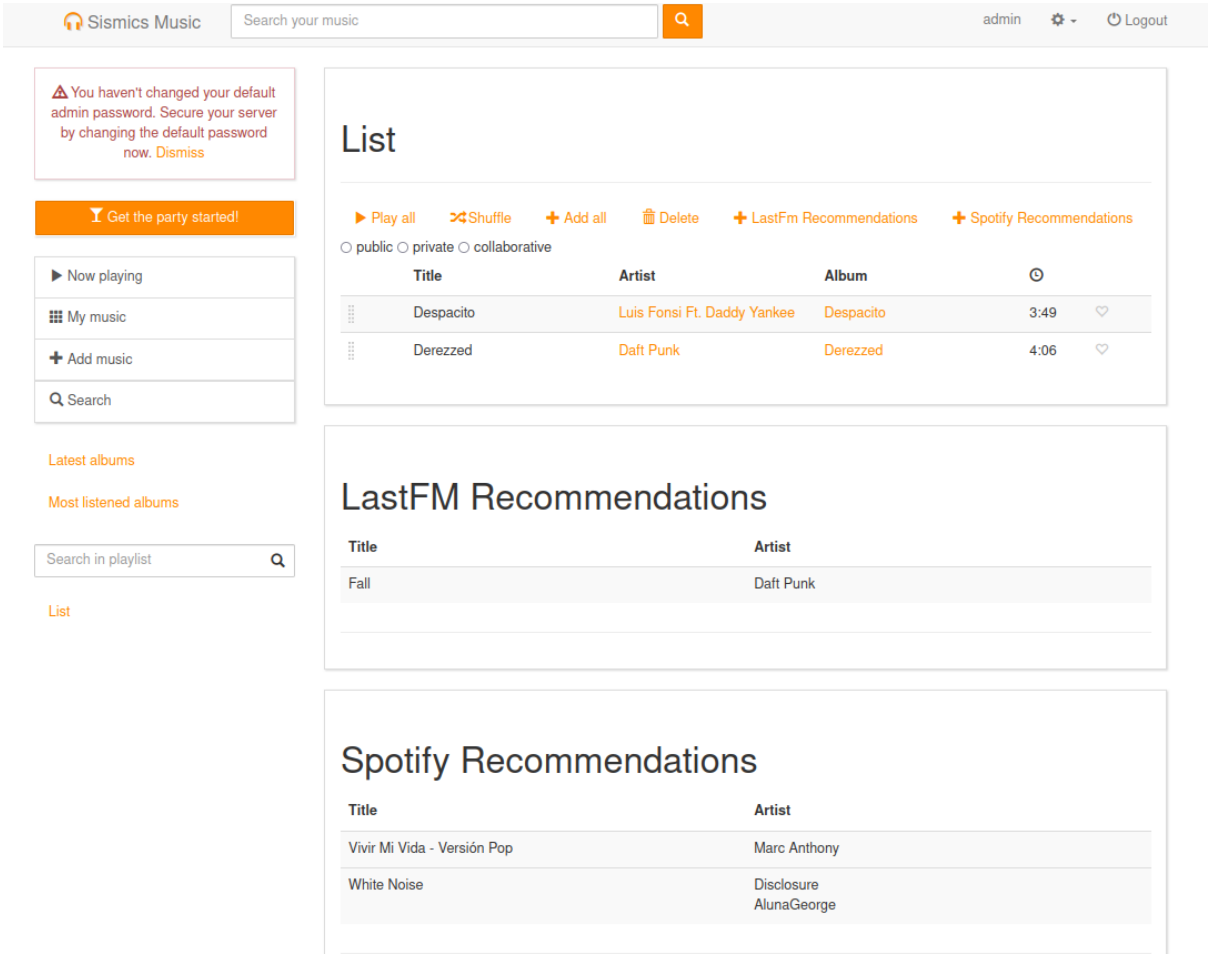
The `getRecommendations()` method uses the Track.`getSimilar()` method of the Lastfm api, which takes the artistName, trackName, apiKey and Limit as its parameters, and returns the 'limit' most similar tracks to the one passed.

Similar to the Spotify recommendations (and in turn, to the Enhance feature in the Spotify app), the tracks in a particular playlist are first obtained using the playlistTracks() method of the PlaylistResource class. Then, we iteratively go through each track in the JSON returned and call getRecommendations() for each of the tracks as `getRecommendations(artist,trackname,1)` returning one recommended track for each track in the playlist.

Frontend

For recommendations a 'Spotify Recommendations' and 'LastFm Recommendations' button was added to the playlist view. When the button is clicked the respective recommendations are displayed in the panels below. As seen below, for the tracks in the playlist 'Test' the recommended track name and artists are displayed.

Spotify Recommendations



The **playlist.html** file was modified to add the buttons and panels to the playlist view. When the **Spotify Recommendations** button is clicked **ng-click** is used to call a function called **spotifyrecommend()**.

The **spotifyrecommend** is defined in the **Playlist.js** controller. Restangular is used to call the GET request for the path **/playlist/spotifyrecommendation**. The results of the request are set to a variable called **spotifyrecommendations** that is used in the **playlist.html** file to display the recommended tracks.

The **PlaylistResource.java** file defines the GET request using the **getRecommendations** function which returns a json object with the results of the GET request.

LastFm Recommendations

The LastFm recommendations frontend is implemented in the same way as the Spotify recommendations. The relevant files are: **playlist.html, Playlist.js, PlaylistResource.java**.

Design Patterns

The Design Patterns in our project, therefore, include the following:

1. **Chain of Responsibility Pattern**: Implemented for Feature 1
2. **Dao Pattern**: Implemented for Feature 2
3. **Strategy Pattern**: Implemented for Feature 3

Besides these, we have also identified the presence of a **Singleton Pattern** in the code. The singleton pattern is a design pattern that restricts the instantiation of a class to one object. This is present in the AppContext class, which contains a single instance object. This ensures that the entire application, and all the classes use the same instance of the AppContext class.

Individual Responsibilities

Feature 1 → Pratyay
Feature 2 → Abhitjih and Yash (backend), Sneha (frontend)
Feature 3 → Jatin (spotify backend), Abhijith (LastFM backend), Sneha (frontend)