

PROJECT - 2

TEAM 18

Jugnu Gill - 2022201011

Arun Das - 2022201021

Jatin Sharma - 2022202023

Nikhil Chawla - 2022201045

Harsimran Singh - 202201049


Task - 1

Allowing Registrations from Login Page

Originally new users can only be added by admin by logging into the admin account. The functionality of adding a new user is already implemented in UserResource.java, but there is a check for admin creation. As our aim is to allow anyone to create a new account, we will comment out or remove the privilege check for account creation.

```
// if (!authenticate()) {  
//     throw new ForbiddenClientException();  
// }  
// checkPrivilege(Privilege.ADMIN);
```

We also have to add a button at the login page for registering.

 Music is not secured. Please log in with username and password "**admin**", then change password immediately.

☐ Remember me

✓ Sign in

Register

Now we have to make a html file for registration and its corresponding js file which will make the call to UserResource.java for adding the users, as well as to check if username, email exists before or not, do passwords match, as well as check if all the fields are following the set conventions regarding length, complexity for password, etc.

The registration page will contain the fields for email, username, password and password confirmation. The page will also show if value entered in a field is not following the convention like length of username being too small and more.

Add New User

Username	<input type="text" value="Username"/>
E-mail	<input type="text" value="E-mail"/>
Password	<input type="password" value="Password"/>
Password (confirm)	<input type="password" value="Password (confirm)"/>

 Add

Sign In

After the values are filled, when we click the add button, it calls the register controller, and a new account is created using UserResource.java.

```
angular.module('music').controller('Register', function($scope, $dialog, $state, $stateParams, Restangular) {  
    $scope.register=function(){  
        var promise = null;  
        promise = Restangular  
            .one('user')  
            .put($scope.user);  
    }  
})
```

Now the user can click on the Sign In button and login by entering the credentials of the newly created account.

We will also added a new state called register in the app.js which is routed to when we click on the registration button

```

.state('register', {
  url: '/register',
  views: {
    'page': {
      templateUrl: 'partial/register.html',
      controller: 'Register'
    }
  }
})

```

We will also add the controller in index.html, so that any html file in the system can call the desired controller.

```

<script src="app/controller/AddImport.js" type="text/javascript"></script>
<script src="app/controller/Register.js" type="text/javascript"></script>
<script src="app/directive/AudioPlayer.js" type="text/javascript"></script>

```

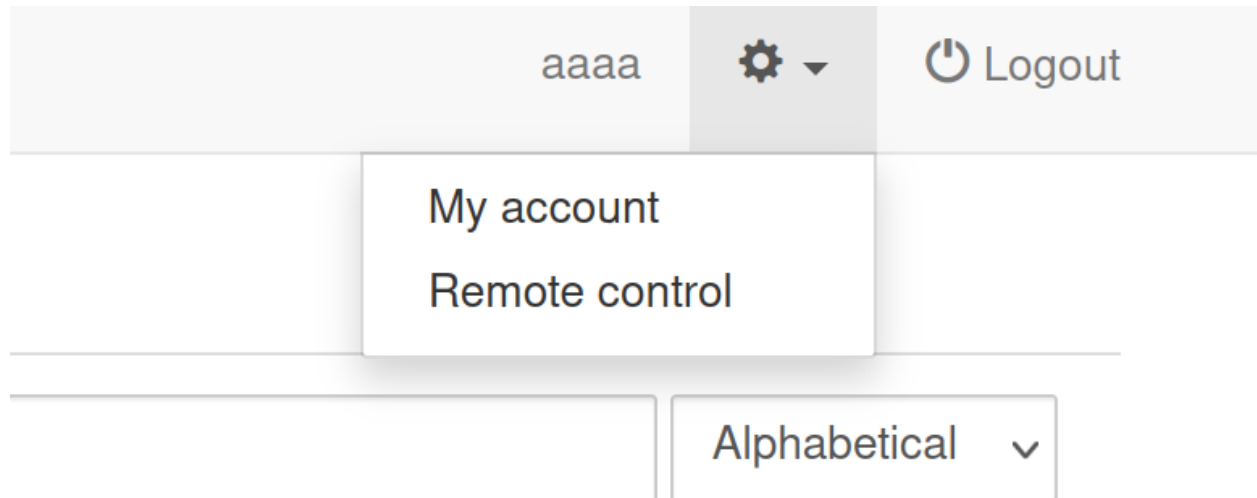
Implementing all this will provide the functionality to create a new account to anyone, the admin can still see the newly created account from the settings dropdown and selecting users.

Users management

Add

Username	Create date
aaaa	2023-03-27
aap	2023-03-27
abc	2023-03-27
abcdefg	2023-03-27
admin	2023-03-27
ppl	2023-03-27
qwe	2023-03-27

Also the newly created users don't have admin access and functionality, and their access is limited to pre determined functionality, like they can change their password but not see all other users registered.



Part-2

Making Music Private

Currently all music is public. This means that the music added by one user is available to everyone. We want to make it so that the music added by one user is visible only to that user.

For this first we need to keep track of who has added the music. So we will create a new column in the T_TRACK table using the ALTER TABLE SQL command. This change is done in the dbupdate-001-0.sql file as shown below.

```
music-core > src > main > resources > db > update > dbupdate-001-0.sql
You, 50 minutes ago | 2 authors (itsmejatins and others)
1 alter table T_USER alter column LASTFMSESSIONTOKEN set default null;
2 update T_USER set LASTFMSESSIONTOKEN = null where LASTFMSESSIONTOKEN = '0';
3 alter table T_TRACK alter column FORMAT type varchar(50);
4 alter table T_TRACK add OWNERID varchar(50);
5 -- alter table T_TRACK drop column OWNERID;
```

Now that we have a new column we need to save the owner id when a user imports a song.

The jaudiotagger library has been used to tag the imported music with the artist name, album name, etc and then this information is passed around. Jaudiotagger is an open-source Java library for reading and manipulating metadata (i.e. information about music files such as title, artist, album, etc.) in various audio file formats such as MP3, FLAC, Ogg Vorbis, and more. It provides a simple API for developers to easily access and modify metadata, as well as handle file and directory operations related to audio files.

We can use the COMMENT field to store the id of the owner so that the FieldKey.COMMENT tag contains the owner id. The picture below shows the snippet of code which sets the various properties of the track.

```
track.setOwner(StringUtils.abbreviate(tag.getFirst(FieldKey.COMMENT), 2000).trim());

// Track title (can be empty string)
track.setTitle(StringUtils.abbreviate(tag.getFirst(FieldKey.TITLE), 2000).trim());

// Track artist (can be empty string)
String artistName = StringUtils.abbreviate(tag.getFirst(FieldKey.ARTIST), 1000).trim();
Artist artist = artistDao.getActiveByName(artistName);
if (artist == null) {
    artist = new Artist();
    artist.setName(artistName);
    artistDao.create(artist);
}
track.setArtistId(artist.getId());
```

Next we have to modify the SQL query which saves the track to the database to include the owner id as well. This portion can be found in the TrackDAO.java file.

```
final Handle handle = ThreadLocalContext.get().getHandle();
handle.createStatement("insert into " +
    " t_track(id, album_id, artist_id, filename, title, titlecorrected, year, genre,
    " values(:id, :albumId, :artistId, :fileName, :title, :titleCorrected, :year, :g
    .bind("id", track.getId())
    .bind("albumId", track.getAlbumId())
    .bind("artistId", track.getArtistId())
    .bind("fileName", track.getFileName())
    .bind("title", track.getTitle())
    .bind("titleCorrected", track.getTitleCorrected())
    .bind("year", track.getYear())
    .bind("genre", track.getGenre())
    .bind("length", track.getLength())
    .bind("bitrate", track.getBitrate())
    .bind("number", track.getOrder())
    .bind("vbr", track.isVbr())
    .bind("format", track.getFormat())
    .bind("createDate", new Timestamp(track.getCreateDate().getTime()))
    .bind("ownerId", track.getOwnerId())
    .execute();
```

Now we also modify the SQL query which fetches the tracks from the database to also fetch the owner id. If we modify this query, we will need to make changes to the TrackDTO.java file and put an attribute called ownerId there. Also TrackDTOMapper.java should contain an extra map that maps the fetched ownerId to the TrackDTO ownerId.

```
@Override
public TrackDto map(int index, ResultSet r, StatementContext ctx) throws SQLException {
    System.out.println("FETCH FROM DB");
    TrackDto dto = new TrackDto();
    dto.setId(r.getString("id"));
    dto.setFileName(r.getString("fileName"));
    dto.setTitle(r.getString("title"));
    dto.setYear(r.getInt("year"));
    dto.setGenre(r.getString("genre"));
    dto.setLength(r.getInt("length"));
    dto.setBitrate(r.getInt("bitrate"));
    dto.setOrder(r.getInt("trackOrder"));
    dto.setVbr(r.getBoolean("vbr"));
    dto.setFormat(r.getString("format"));
    dto.setUserTrackPlayCount(r.getInt("userTrackPlayCount"));
    dto.setUserTrackLike(r.getBoolean("userTrackLike"));
    dto.setArtistId(r.getString("artistId"));
    dto.setArtistName(r.getString("artistName"));
    dto.setAlbumId(r.getString("albumId"));
    dto.setAlbumName(r.getString("albumName"));
    dto.setAlbumArt(r.getString("albumArt"));
    dto.setOwner(r.getString("ownerId"));

    return dto;
}
```

Similar to the TrackDTO file we also need to add an ownerId attribute to the Track.java model.

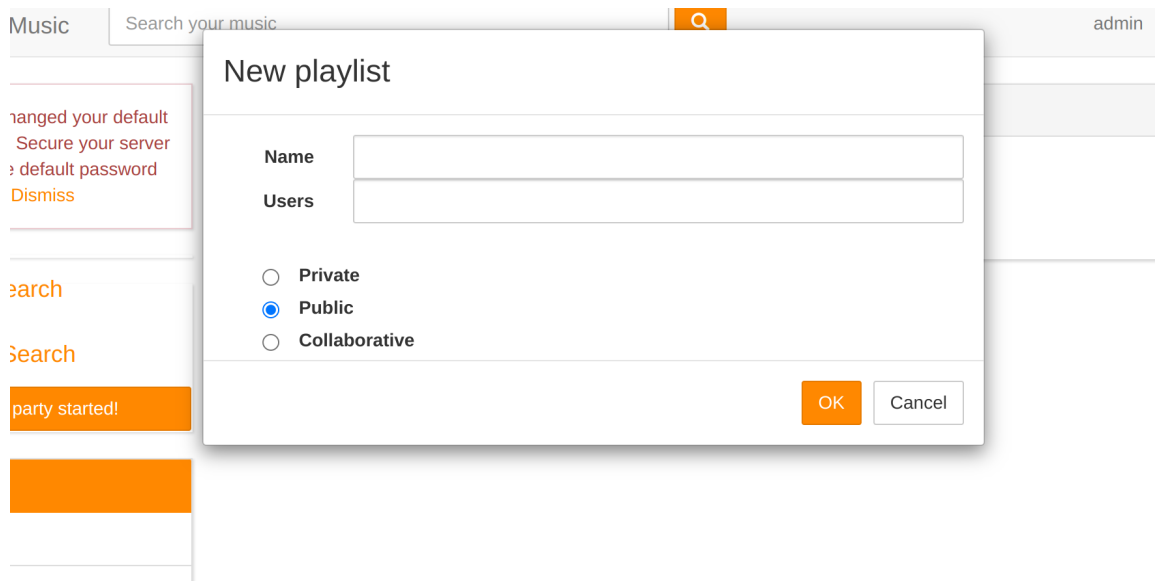
Now the rest of the part is more logic based. We fetch all the tracks and return only the specific tracks to the user as the json using an if condition.

```
for (TrackDto trackDto : trackList) {  
    // System.out.println("BB");  
    // System.out.println(trackDto.getOwner());  
    // System.out.println(trackDto.getTitle());  
    if(trackDto.getOwner() == principal.getId()){  
        tracks.add(Json.createObjectBuilder()  
            .add("order", JsonUtil.nullable(trackDto.getOrder()))  
            .add("id", trackDto.getId())  
            .add("title", trackDto.getTitle())  
            .add("year", JsonUtil.nullable(trackDto.getYear()))  
            .add("genre", JsonUtil.nullable(trackDto.getGenre()))  
            .add("length", trackDto.getLength())  
            .add("bitrate", trackDto.getBitrate())  
            .add("vbr", trackDto.isVbr())  
            .add("format", trackDto.getFormat())  
            .add("filename", trackDto.getFileName())  
            .add("play_count", trackDto.getUserTrackPlayCount())  
            .add("liked", trackDto.isUserTrackLike())  
            .add("artist", Json.createObjectBuilder()  
                .add("id", trackDto.getArtistId())  
                .add("name", trackDto.getArtistName())));  
    }  
}  
response.add("tracks", tracks);  
  
return renderJson(response);
```

For an existing album/artist we check if the user has any track belonging to that album/artist. If no such track exists we just don't display that album/artist.

Making playlists public

We first need to make changes to the frontend so that the user can designate a playlist as public and is able to choose which other users can view that. This change is done in the modal.createplaylist.html file.



The list of users who should be able to view should be input in the Users field. It should have the user names of the users separated by a space.

We have taken the approach to make all the playlists public by default and grant access to specific playlists based on whether their names are in the allowedUsers list.

```
// Get the playlist
PlaylistDto playlistDto = new PlaylistDao().findFirstByCriteria(new PlaylistCriteria()
    .setUserId(null) // You, 1 hour ago * Part 2
    .setDefaultPlaylist(false)
    .setId(playlistId));
notFoundIfNull(playlistDto, "Playlist: " + playlistId);
```

For making the playlists public by default we have to make `setUserId` null at most places so that all the playlists are fetched and then we can filter them out.

The rest of the job is very similar to the previous section. We need to keep track of the playlist type and the list of users who have access to the playlist. This can be done similarly to before. If a user is not on the allowedUsers list, the playlist is simply not added to the json file which is returned. This means that the user can't see that playlist.

If a user is on the allowedUsers list he/she is given only read access. This is done by checking if a user is the owner of a playlist when trying to perform a delete/add_track operation. If the user is not an owner, an early return is used to deny the user any modifications.

Design Pattern (Factory Pattern)

Currently the SQL queries are present in the DAO files. For the track, album, artist the fetch queries are present with other important stuff in the DAO file.

For each of the fetch queries we can use the factory pattern. The factory design pattern is a creational pattern in object-oriented programming that provides an interface or

abstract class for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

We will be considering the fetch queries of the track, album and artist files.

```
interface Query{
    public void addCriterion(Criteria criteria, List<String> criteriaList, Map<String, Object> parameterMap);
    public StringBuilder buildString(Criteria criteria);
}

public abstract class QueryCreator{
    public abstract Query createQuery();
    public StringBuilder getQueryParams(Criteria criteria, List<String> criteriaList, Map<String, Object> parameterMap){
        Query query = createQuery();
        StringBuilder sb = query.buildString(criteria);
        query.addCriteria(criteria, criteriaList, parameterMap);
        return sb;
    }
}
```

First we create the Query interface and QueryCreator abstract class. The addCriterion is used to add various filters and criteria to the SQL query. The buildString returns the actual SQL query string.

Now if we want to have the fetch query for the artists we create an ArtistQueryCreator class which builds the SQL string to fetch the details for an artist.

```
public class ArtistQueryCreator extends QueryCreator{
    public ArtistQuery createQuery(){
        ArtistQuery artistQuery = new ArtistQuery();
        return artistQuery;
    }

    private class ArtistQuery implements Query{
        // Adds search criteria
        public void addCriterion(Criteria criteria, List<String> criteriaList, Map<String, Object> parameterMap){
            // Adds search criteria
            criteriaList.add("a.deletedate is null");
            if (criteria.getId() != null) {
                criteriaList.add("a.id = :id");
                parameterMap.put("id", criteria.getId());
            }
            if (criteria.getNameLike() != null) {
                criteriaList.add("lower(a.name) like lower(:nameLike)");
                parameterMap.put("nameLike", "%" + criteria.getNameLike() + "%");
            }
        }

        @Override
        public StringBuilder buildString(Criteria criteria){
            StringBuilder sb = new StringBuilder("select a.id as id, a.name as c0 ");
            sb.append(" from t_artist a ");
            return sb;
        }
    }
}
```

The file using it (ArtistDAO.java) calls it as shown in the image below.

```
List<String> criteriaList = new ArrayList<>();
Map<String, Object> parameterMap = new HashMap<>();

ArtistQueryCreator queryCreator = new ArtistQueryCreator();
StringBuilder sb = queryCreator.getQueryParams(criteria, criteriaList, parameterMap);
return new QueryParam(sb.toString(), criteriaList, parameterMap, null, filterCriteria, new ArtistDtoMapper());
```

As another example we can see the working of the AlbumQueryCreator.

```
public class AlbumQueryCreator extends QueryCreator{
    public AlbumQuery createQuery(){
        AlbumQuery albumQuery = new AlbumQuery();
        return albumQuery;
    }

    private class AlbumQuery implements Query{
        // Adds search criteria
        public void addCriteria(Criteria criteria, List<String> criteriaList, Map<String, Object> parameterMap){
            criteriaList.add("ar.deletedate is null");
            criteriaList.add("a.deletedate is null");
            if (criteria.getId() != null) {
                criteriaList.add("a.id = :id");
                parameterMap.put("id", criteria.getId());
            }
            if (criteria.getDirectoryId() != null) {
                criteriaList.add("a.directory_id = :directoryId");
                parameterMap.put("directoryId", criteria.getDirectoryId());
            }
            if (criteria.getArtistId() != null) {
                criteriaList.add("ar.id = :artistId");
                parameterMap.put("artistId", criteria.getArtistId());
            }
            if (criteria.getNameLike() != null) {
                criteriaList.add("(lower(a.name) like lower(:like) or lower(ar.name) like lower(:like))");
                parameterMap.put("like", "%" + criteria.getNameLike() + "%");
            }
        }

        public StringBuilder buildString(Criteria criteria){
            StringBuilder sb = new StringBuilder("select a.id as id, a.name as c0, a.albumart as albumArt, a.artist_id as artistId ");
            if (criteria.getUserId() == null) {
                sb.append("sum(0) as c2");
            } else {
                sb.append("sum(utr.playcount) as c2");
            }
            sb.append(" from t_album a ");
            sb.append(" join t_artist ar on(ar.id = a.artist_id) ");
            if (criteria.getUserId() != null) {
                sb.append(" left join t_track tr on(tr.album_id = a.id) ");
                sb.append(" left join t_user_track utr on(tr.id = utr.track_id) ");
            }
            return sb;
        }
    }
}
```

```
List<String> criteriaList = new ArrayList<>();
Map<String, Object> parameterMap = new HashMap<>();

AlbumQueryCreator queryCreator = new AlbumQueryCreator();
StringBuilder sb = queryCreator.getQueryParams(criteria, criteriaList, parameterMap);
return new QueryParam(sb.toString(), criteriaList, parameterMap, null, filterCriteria, Lists.newArrayList());
```

The same is also done for tracks by creating a TrackQueryCreator subclass.

Using the factory design pattern has helped in the following aspects-

- 1) Loose Coupling: The factory pattern has made it easier to modify or extend the system without affecting other parts of the code. If we want to add a fetch query for any other object, we just have to create the QueryCreator subclass for it.
- 2) Encapsulation: The factory pattern has encapsulated the SQL queries inside a class making it easier to modify or replace the object creation process without affecting the rest of the code.

- 3) Abstraction: The factory pattern has helped in abstraction as now the DAO files do not need to know the inner workings of how the SQL query is created based on the search criteria and filters.
- 4) Simplified Client Code: The DAOs can now use the QueryCreator as a black box as seen in some of the images above. This simplifies the code a lot.

Design Pattern (Filter or Criteria Pattern)

This pattern is used to filter a set of objects by chaining operations. This pattern had already been used to construct the SQL queries. For each concrete criteria class there is a concrete criteria class which is used to fetch the queries.

```
/**
 * Artist ID.
 */
private String id;

/**
 * Artist name (like).
 */
private String nameLike;

public String getId() {
    return this.id;
}

public String getNameLike() {
    return nameLike;
}

public ArtistCriteria setNameLike(String nameLike) {
    this.nameLike = nameLike;
    return this;
}

public ArtistCriteria setId(String id) {
    this.id = id;
    return this;
}
}
```

The example shown above is the ArtistCriteria class. The factory pattern mentioned above had been incorporated with the criteria pattern to make the code more extensible.

The filter pattern gives us the following advantages-

- It provides a way to filter objects based on certain criteria.
- We can add a new filter any time without affecting the client's code.
- We can select filters dynamically during program execution.

FEATURE 3: ONLINE INTEGRATION

In this task, we were required to implement search and recommend functionalities using the APIs of Spotify and LastFM.

SPOTIFY

Spotify Search

Spotify Search

Numb
Artist: Mount Eminest
URL: <https://open.spotify.com/artist/4oUcWvCNxqNZv4I7BXIE0y>

Numb / Encore
Artist: JAY-Z
URL: <https://open.spotify.com/artist/3nFkdISjzX9mRTtwJOzDYB>

Numbers
Artist: Melanie Martinez
URL: <https://open.spotify.com/artist/63YrD80RY3RNEM2YDpUpO8>

How the api works

- To search with Spotify, we first had to make a developer account in Spotify. From this account we code a client_username and client_secret keys.
- These values were then used to generate a bearer token, which was used for authentication of search and recommendation requests.
- The token for getting a bearer token is: <https://accounts.spotify.com/api/token>
 - the client username and client secret key had to be attached in the http header.
- After hitting this request, we got a response back from Spotify which contained a bearer token.
- Now for searching, we hit the API -
<https://api.spotify.com/v1/search?type=track&q=Numb&limit=10&offset=10>
 - Here type=track means that we want to search tracks that match with our search param.
 - limit=10 means that we need only 10 matching tracks from Spotify. There will be thousands of songs that can match with the search param. To limit the number of results, we use this param.
 - q=Numb means that our search param is numb.
 - The bearer token was attached in the header of http request.

Changes done in the codebase

- spotifySearch.html page was made. Its controller is main.js. Corresponding entry was added in the app.js file. The service endpoint in the back end is UserResource. The code snippet for the same is provide below -

Implementation details

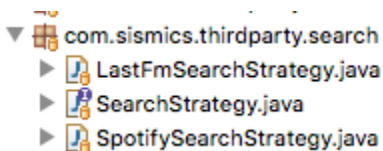
- Strategy design pattern was implemented for search. In the java controller class, based upon the input from the front end (service type), an object of SpotifySearchStrategy or LastFmSearchStrategy was created.

```
@GET
@Path("/searchThirdParty")
public Response search(@QueryParam("sentData") String sentData, @QueryParam("service") String service) {
    SearchStrategy strategy = null;
    if (service.equals("spotify"))
        strategy = new SpotifySearchStrategy();
    else if (service.equals("lastfm"))
        strategy = new LastFmSearchStrategy();

    Response result = null;
    try {
        result = strategy.search(sentData);
    } catch (Exception e) {
        System.out.println("Search third party failed!");
        e.printStackTrace();
    }

    return result;
}
```

- The classes responsible for handling online integration facility were placed in the music-core module



- The logic for searching in Spotify (getting bearer token, sending http request, receiving response, parsing the response to JSON, etc) is written in SpotifySearchStrategy.

```
30 public class SpotifySearchStrategy implements SearchStrategy{
31
32+     private Response renderJson(JsonObjectBuilder response) {}
37
38+     private String getAccessToken() throws IOException {}
69
70+     private JsonNode getSearchResultsSpotify(String search, String token,String type) throws Exception {}
107
109+     public Response search(String sentData) throws Exception {}
142
143 }
```

- For complete implementation details, please refer to the codebase.

Spotify recommendation

- Here we provide recommendation based on two scenarios -
 - You have an existing playlist and you want to get recommendations based on the songs present in the playlist.

⚠ You haven't changed your default admin password. Secure your server by changing the default password now. [Dismiss](#)

Spotify Search

Last FM Search

🎧 Get the party started!

▶ Now playing

📶 My music

+ Add music

Latest albums

Most listened albums

Search in playlist

playlist private

Fav songs

Fav songs

Spotify RecommendationLastFM Recommendation

Search Based Recommendation

▶ Play all ⚡ Shuffle + Add all 🗑 Delete

	Title	Artist	Album	🕒	📖
📶	Stressed Out	Twenty One Pilots	Blurryface	3:22	📖
📶	Malang	Ved Sharma	Malang (2020)	4:47	📖
📶	Ride	Twenty One Pilots	Mw Single Hits	3:47	📖
📶	Baller	Shubh	New Punjabi Song 2022 - P...	2:33	📖

Recommendation from Spotify

NO WARNING

Artist: LE SINNER

URL: <https://open.spotify.com/artist/7ia9MJ4e6Te5oJwgJM29g5>

High Highs to Low Lows

Artist: Lolo Zouaï

URL: <https://open.spotify.com/artist/2qDIR2WicW3IkGqJWg9VJ>

Dope

Artist: Asakura

URL: <https://open.spotify.com/artist/6Twdw4cy4bu2vRXuCne8uU>

Myself

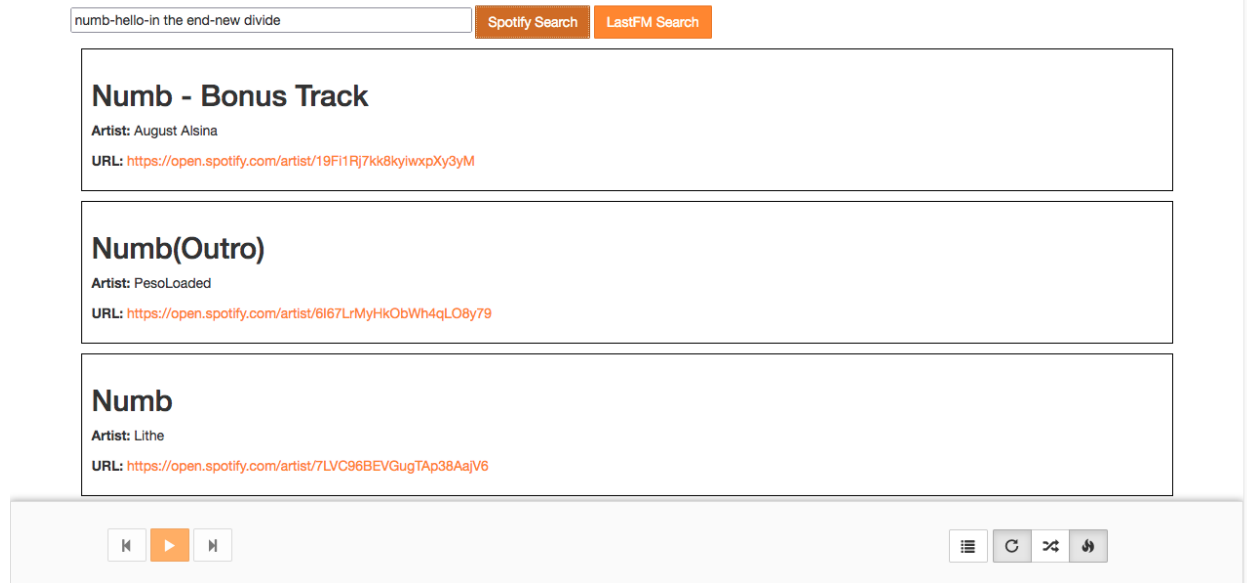
Artist: Duñe

⏮ ⏪ ⏩ ⏭

📶 ⌂ ⌂ ⌂ ⌂

- You give a list of songs separated by hyphen, and spotify will recommend you some songs based on your input.

Search Based Recommendation



How the api works

- To recommend songs from Spotify, we first had to make a developer account in Spotify. From this account we code a client_username and client_secret keys.
- These values were then used to generate a bearer token, which was used for authentication of search and recommendation requests.
- The token for getting a bearer token is: <https://accounts.spotify.com/api/token>
 - the client username and client secret key had to be attached in the http header.
- After hitting this request, we got a response back from Spotify which contained a bearer token
- To get songs recommended from Spotify we need artist seed id.
- Unlike in the search API of Spotify, firstly we need the artist seed from the artists present in our playlist and then using that we can get songs recommended from Spotify.
- Now for getting artist seed, we hit the API - <https://api.spotify.com/v1/search?type=artist&q=Sidhu+Moosewala&limit=10&offset=10>
Here,
 - type=artist means that we want to search artist that match with our search param.
 - limit=10 means that we need only 10 matching artists from Spotify.
 - q= Sidhu Moosewala means that our search param is Sidhu Moosewala.
 - The bearer token was attached in the header of http request.
- Above api will provide us the Artist seed
- Now for getting recommendation, we hit the API -

https://api.spotify.com/v1/recommendations?&seed_artists=5r3wPya2PpeTTsXsGhQU8O&limit=10

Here,

- limit=10 means that we need only 10 matching artists from Spotify.
- In seed_artists, we pass the seed_artist we got from above API
- The bearer token was attached in the header of http request.

Changes done in the codebase

- For recommending songs based on the playlist, we made changes in playlist.html to provide you options to get recommendations based on the service you choose (spotify or LastFm).
 - The corresponding controller was Playlist.js
- For recommending songs based on search input, we made a new file recomdSearch.html.
 - The corresponding controller is Main.js

Implementation details

- The controller in the backend is UserResource. Strategy design pattern was used. Based on the input from the front end, SpotifyPlaylistRecommendStrategy or LastFmPlaylistRecommendStrategy was created.

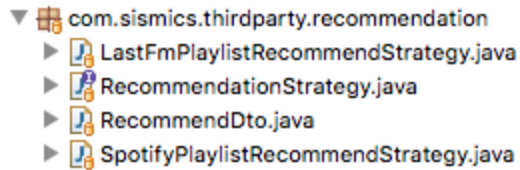
```
@GET
@Path("/recommendThirdParty")
public Response recommend(@QueryParam("A") String artists, @QueryParam("T") String titles,
    @QueryParam("service") String service) throws Exception {

    RecommendationStrategy strategy = null;
    RecommendDto dto = new RecommendDto();
    if (service.equals("spotify")) {
        strategy = new SpotifyPlaylistRecommendStrategy();
        dto.setArtists(artists);
    } else if (service.equals("lastfm")) {
        strategy = new LastFmPlaylistRecommendStrategy();
        dto.setTitles(titles);
        dto.setArtists(artists);
    }

    Response response = null;
    try {
        response = strategy.recommend(dto);
    } catch (Exception e) {
        System.out.println("Recommendation from third party failed!");
        e.printStackTrace();
    }

    return response;
}
```

- The classes responsible for recommendation were placed in the music-core module. These are shown below -



- DTO pattern employed to facilitate exchange of search parameters information between different strategy objects. This was necessary because the API of LastFm requires different information from Spotify recommendation APIs.

```
3 public class RecommendDto {
4     private String artists;
5     private String titles;
6
7     public String getArtists() {
8         return artists;
9     }
10
11    public void setArtists(String artists) {
12        this.artists = artists;
13    }
14
15    public String getTitles() {
16        return titles;
17    }
18
19    public void setTitles(String title) {
20        this.titles = title;
21    }
22 }
23 }
```

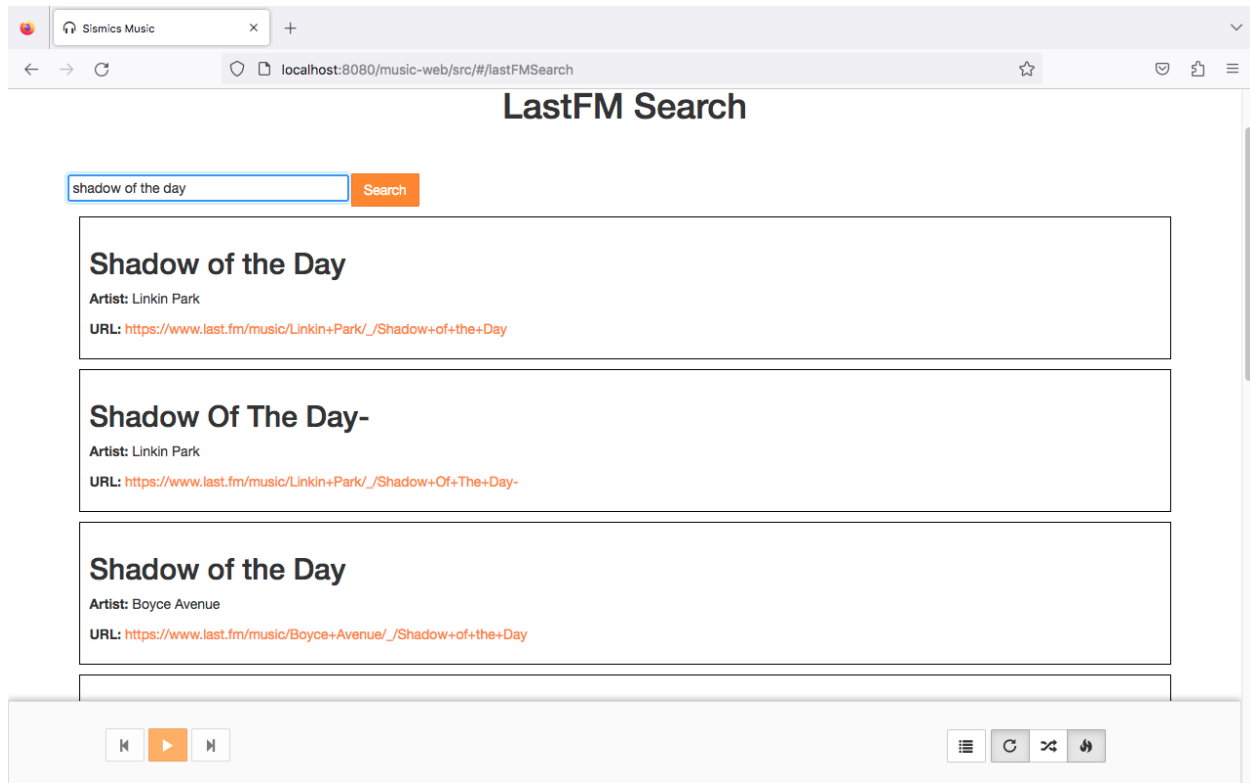
- The logic for getting the recommendation results like establishing connection, etc was implemented in SpotifyPlaylistRecommendStrategy.java

```
0 public class SpotifyPlaylistRecommendStrategy implements RecommendationStrategy {
1
2     private Response renderJson(JsonObjectBuilder response) {}
3
4     private String getAccessToken() throws IOException {}
5
6     private JsonNode getSearchResultsSpotify(String search, String token, String type) throws Exception {}
7
8     private static JsonNode getSpotifyRecommendation(String artistSeed, String token) throws Exception {}
9
10    public Response recommend(RecommendDto dto) throws Exception {}
11 }
12 }
```

- When we need to recommend songs based on search input, the search APIs of spotify were used.

LASTFM

LastFM Search



How the api works

- Similar to spotify APIs, here also we have to create a API account by going to the Last FM website and by doing that we get credentials which help us in using any API service of LASTFM.
- To search for a song we have used this URL:
https://ws.audioscrobbler.com/2.0/?method=track.search&track=Believe&api_key=?&format=json&limit=10

Here,

- type=track means that we want to search tracks that match with our search param.
- limit=10 means that we need only 10 matching tracks from LastFM. There will be thousands of songs that can match with the search param. To limit the number of results, we use this param.
- track=Believe means that our search param is Believe song
- The api key which we got from by creating API account will be mentioned in api_key param

Changes done in the codebase

- lastFMSearch.html page was created. Its controller is in main.js. Corresponding entry was added in the app.js file. The service endpoint in the back end is UserResource.

Implementation details

- Like mentioned in spotify search, service received from the front end will be lastfm, and therefore LastFmSearchStrategy object will be created.
- This will fetch information to the front end in the same way the spotify search object fetched.

```
public class LastFmSearchStrategy implements SearchStrategy {  
+   private Response renderJson(JsonObjectBuilder response) {  
  
+   private static JsonNode getSearchResultsLastFM(String search) throws Exception {  
  
+   public Response search(String searchData) throws Exception {  
  
}
```

LastFM Recommendation:

- Just like we did in spotify recommendation, here we provide recommendation based on two scenarios -
 - You have an existing playlist and you want to get recommendations based on the songs present in the playlist.

⚠ You haven't changed your default admin password. Secure your server by changing the default password now. [Dismiss](#)

Spotify Search

Last FM Search

🎧 Get the party started!

▶ Now playing

📶 My music

+ Add music

Latest albums

Most listened albums

Fav songs

Spotify Recommendation

LastFM Recommendation

Search Based Recommendation

▶ Play all ⚡ Shuffle + Add all 🗑 Delete

	Title	Artist	Album	🕒	
📶	Stressed Out	Twenty One Pilots	Blurryface	3:22	♥
📶	Malang	Ved Sharma	Malang (2020)	4:47	♥
📶	Ride	Twenty One Pilots	Mw Single Hits	3:47	♥
📶	Baller	Shubh	New Punjabi Song 2022 - P...	2:33	♥

playlist private

Fav songs

Recommendation from LastFM

We Rollin

Artist: Shubh

URL: https://www.last.fm/music/Shubh/_/We+Rollin

No Love

Artist: Shubh

URL: https://www.last.fm/music/Shubh/_/No+Love

Gustakhiyan

Artist: The Landers

URL: https://www.last.fm/music/The+Landers/_/Gustakhiyan

25-25

- You give a list of songs separated by hyphen, and LastFm will recommend you some songs based on your input.

Search Based Recommendation

Spotify Search
LastFM Search

Numb

Artist: Linkin Park

URL: https://www.last.fm/music/Linkin+Park/_/Numb

Comfortably Numb

Artist: Pink Floyd

URL: https://www.last.fm/music/Pink+Floyd/_/Comfortably+Numb

My Number

Artist: Foals

URL: https://www.last.fm/music/Foals/_/My+Number

How the api works

- As done in the search part this also needs an api key to hit Recommendation API at the LastFM side.

- To use LastFM API for recommendation API we have used the following URL:
http://ws.audioscrobbler.com/2.0/?method=track.getsimilar&artist=Cherish&track=That+Boi&api_key=?&limit=10
 Here,
 - method=track.similar means that we want to get tracks that are similar to the passed track
 - artist=Cherish means that our search param song is Cherish
 - track - That Boi means that our search param track is That Boi
 - limit=10 means that we need only 10 matching tracks from LastFM. There will be thousands of songs that can match with the search param. To limit the number of results, we use this param.
 - The api key which we got from by creating API account will be mentioned in api_key param
- By hitting this above API we get a response in which we have all the track with artist which are similar to passed track and artist in the above URL

Changes done in the codebase

- For recommending songs based on the playlist, we made changes in playlist.html to provide you options to get recommendations based on the service you choose (spotify or LastFm).
 - The corresponding controller was Playlist.js
- For recommending songs based on search input, we made a new file recomdSearch.html.
 - The corresponding controller is Main.js

Implementation details

- The implementation details are very similar to the Spotify recommendation, just the strategy changes -

```

17 public class LastFmPlaylistRecommendStrategy implements RecommendationStrategy {
18
19     private Response renderJson(JsonObjectBuilder response) {}
22
23     private static JsonNode getLastFMRecommendation(String artist, String title) throws Exception {}
56
58     public Response recommend(RecommendDto dto) throws Exception {}
105
106 }
107

```

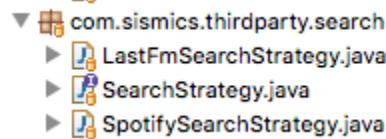
- RecommendDto was used to transfer information between different strategies
- When we need to recommend songs based on search input, the search APIs of LastFm were used.

DESIGN PATTERNS SUMMARIZED

Though all the design patterns can be found in implementation of various functionalities, a brief description is provided below. We also mention some of the design patterns that we identified in the existing code, and some design pattern that we implemented in the existing code -

Search Functionality

- **Strategy pattern** was used to create objects of different search strategies based on the input received from the front end.



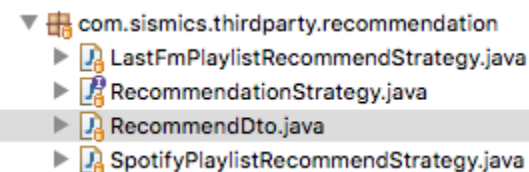
```
@GET
@Path("/searchThirdParty")
public Response search(@QueryParam("sentData") String sentData, @QueryParam("service") String service) {
    SearchStrategy strategy = null;
    if (service.equals("spotify"))
        strategy = new SpotifySearchStrategy();
    else if (service.equals("lastfm"))
        strategy = new LastFmSearchStrategy();

    Response result = null;
    try {
        result = strategy.search(sentData);
    } catch (Exception e) {
        System.out.println("Search third party failed!");
        e.printStackTrace();
    }

    return result;
}
```

Recommend Functionality

- **Strategy pattern** was used to create objects of different search strategies based on the input received from the front end.




```

@GET
@Path("recommendThirdParty")
public Response recommend(@QueryParam("A") String artists, @QueryParam("T") String titles,
    @QueryParam("service") String service) throws Exception {

    RecommendationStrategy strategy = null;
    RecommendDto dto = new RecommendDto();
    if (service.equals("spotify")) {
        strategy = new SpotifyPlaylistRecommendStrategy();
        dto.setArtists(artists);
    } else if (service.equals("lastfm")) {
        strategy = new LastFmPlaylistRecommendStrategy();
        dto.setTitles(titles);
        dto.setArtists(artists);
    }

    Response response = null;
    try {
        response = strategy.recommend(dto);
    } catch (Exception e) {
        System.out.println("Recommendation from third party failed!");
        e.printStackTrace();
    }

    return response;
}

```

- The **DTO pattern** was used to provide information to different strategies, based upon the APIs they use.

```

3 public class RecommendDto {
4     private String artists;
5     private String titles;
6
7     public String getArtists() {}
10
11     public void setArtists(String artists) {}
14
15     public String getTitles() {}
18
19     public void setTitles(String title) {}
22
23 }
24

```

Fetching from Database Functionality

- The **Factory Pattern** was used to create different classes for fetching all the tracks/artists/albums from playlists. This increased the readability of the code and also improved the extensibility as fetching from the database now only requires 3-4 lines of code. An example is shown below.

```

public class ArtistQueryCreator extends QueryCreator{
    public ArtistQuery createQuery(){
        ArtistQuery artistQuery = new ArtistQuery();
        return artistQuery;
    }

    private class ArtistQuery implements Query{
        // Adds search criteria
        public void addCriteria(Criteria criteria, List<String> criteriaList, Map<String, Object> parameterMap){
            // Adds search criteria
            criteriaList.add("a.deletedate is null");
            if (criteria.getId() != null) {
                criteriaList.add("a.id = :id");
                parameterMap.put("id", criteria.getId());
            }
            if (criteria.getNameLike() != null) {
                criteriaList.add("lower(a.name) like lower(:nameLike)");
                parameterMap.put("nameLike", "%" + criteria.getNameLike() + "%");
            }
        }

        @Override
        public StringBuilder buildString(Criteria criteria){
            StringBuilder sb = new StringBuilder("select a.id as id, a.name as c0 ");
            sb.append(" from t_artist a ");
            return sb;
        }
    }
}

```

- The **criteria/filter pattern** was used to make it easier to construct queries for the database and improve extensibility as new queries can easily be added. This pattern was already implemented in the existing codebase.

```

v criteria
  J AlbumCriteria.java
  J ArtistCriteria.java
  J Criteria.java
  J PlaylistCriteria.java
  J TrackCriteria.java
  J UserCriteria.java

```

```

public class ArtistCriteria extends Criteria {
    /**
     * Artist ID.
     */
    private String id;

    /**
     * Artist name (like).
     */
    private String nameLike;

    public String getId() {
        return this.id;
    }

    public String getNameLike() {
        return nameLike;
    }

    public ArtistCriteria setNameLike(String nameLike) {
        this.nameLike = nameLike;
        return this;
    }

    public ArtistCriteria setId(String id) {
        this.id = id;
        return this;
    }
}

```

Visitor pattern in Collection service

- This pattern was already implemented in the existing code. It can be seen in the package - com.sismics.music.core.service.collection;

```
public class CollectionVisitor extends SimpleFileVisitor<Path> {
    * Root directory to visit.
    private Directory rootDirectory;

    * Root path.
    private Path rootPath;

    * Files indexed during the lifetime of this visitor.
    private Set<String> fileNameSet = new HashSet<>();

    * Logger.
    private static final Logger log = LoggerFactory.getLogger(CollectionVisitor.class);

    public CollectionVisitor(Directory rootDirectory) {}

    public FileVisitResult visitFile(Path path, BasicFileAttributes attrs) throws IOException {}

    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws IOException {}

    public FileVisitResult visitFileFailed(Path file, IOException exc) throws IOException {}

    * Index subfolders in the root directory.
    public void index() {}

    /**
     * Getter of fileNameSet.
     *
     * @return fileNameSet
     */
    public Set<String> getFileNameSet() {
        return fileNameSet;
    }
}
```

- Collection service is used by various listeners attached to the event buses to perform appropriate tasks when they consume an object of some event.

Builder pattern in Album

- In the existing code, the Album class had a long constructor. So we made a builder class for it.

```

5 public class AlbumBuilder {
6
7     private Album album = new Album();
8
9
10
11+ public AlbumBuilder setId(String id) {}
12
13
14
15
16
17+ public AlbumBuilder setDirectoryId(String directoryId) {}
18
19
20
21
22
23+ public AlbumBuilder setArtistId(String artistId) {}
24
25
26
27
28
29+ public AlbumBuilder setName(String name) {}
30
31
32
33
34
35+ public AlbumBuilder setAlbumArt(String albumArt) {}
36
37
38
39
40
41+ public AlbumBuilder setCreateDate(Date createDate) {}
42
43
44
45
46
47+ public AlbumBuilder setUpdateDate(Date updateDate) {}
48
49
50
51
52
53+ public AlbumBuilder setDeleteDate(Date deleteDate) {}
54
55
56
57
58
59+ public AlbumBuilder setLocation(String location) {}
60
61
62
63
64+ public Album build()
65 {
66     return album;
67 }
68
69
70 }
71

```

- This builder was used to create the object of album.

```

30+ @Override
31 public Album map(int index, ResultSet r, StatementContext ctx) throws SQLException {
32     final String[] columns = getColumns();
33     int column = 0;
34     AlbumBuilder builder = new AlbumBuilder();
35     builder.setId(r.getString(columns[column++]));
36     builder.setDirectoryId(r.getString(columns[column++]));
37     builder.setArtistId(r.getString(columns[column++]));
38     builder.setName(r.getString(columns[column++]));
39     builder.setAlbumArt(r.getString(columns[column++]));
40     builder.setCreateDate(r.getTimestamp(columns[column++]));
41     builder.setUpdateDate(r.getTimestamp(columns[column++]));
42     builder.setDeleteDate(r.getTimestamp(columns[column++]));
43     builder.setLocation(r.getString(columns[column]));
44
45     return builder.build();
46 }
47 }

```

Adapter pattern in search and recommend

```
public class JsonAdapter {  
    public static JsonNode convertToJson(String response) throws JsonParseException, IOException  
    {  
        ObjectMapper mapper = new ObjectMapper();  
        JsonParser parser = mapper.getFactory().createParser(response);  
  
        JsonNode jsonNode = mapper.readTree(parser);  
        return jsonNode;  
    }  
}
```

- We used adapter pattern to convert the string returned from the servers of spotify and lastfm to json.

CONTRIBUTION

Task 1: Jugnu Gill (2022201011)

Task 2: Arun Das (2022201021) and Nikhil Chawla (2022201045)

Task 3: Harsimran Singh (2022201049) and Jatin Sharma (2022201023)