Software Engineering: Project - 2

Team - 19

Overview

In the previous project, we looked at how the sismics music server application is structured, what are the various sub-systems, what are some of the design and code smells, what are the functionalities it offers and what are some of its limitations.

This project deals with enhancing the features of the sismics music server application. In doing so, we employed some of the design patterns when needed. We will discuss all the new functionalities we engineered and the process of creating them in this report.

Task-1: Improved Account Creation Workflow

The existing system allows new user account creation only through the admin account. This puts unnecessary strain on the admin and is very inefficient. Our solution is to remove this restriction and make it possible for everyone to register for a new account.

Updated Flow of Account Creation

- User visits the homepage of the application/website.
- If he/she is already logged in, the user is taken to the homepage, else the user will be redirected to login page.
- If the user is registered, he will fill in the form and click on the login button. Else, he can click on don't have an account button/link, which takes him to register page where he can create a new account.

Steps involved in achieving the updated flow

- `register.html` and `register.js` controller are created.
- `index.html` is informed about the new controller.
- `app.js` is updated with the new controller-view information.
- `register.html` includes a function call which is implemented in `register.js` which sends a `PUT` request to java backend along with the form data.
- `PUT` request-handler in `UserResource.java` is modified to ignore the isAdmin check while registering the user.

Code Changes

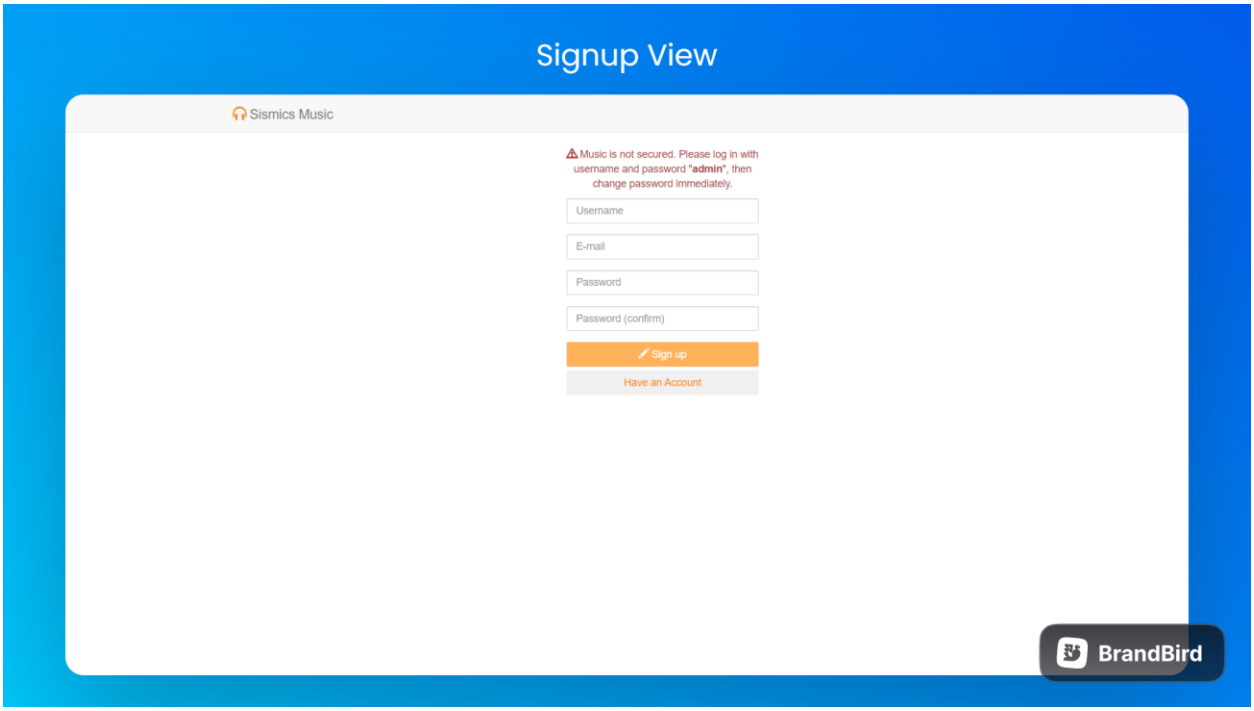
- [ADD] signup.html
- [ADD] signup.js
- [MODIFY] app.js
- [MODIFY] index.html
- [MODIFY] UserResource.java

Screenshots

Signup.js

```
"use strict";

/**
 * Signup controller.
 */
angular.module("music").controller("Signup", function ($rootScope, $scope, $state,
Restangular) {
  $scope.signup = function () {
    Restangular.one("user")
      .put($scope.user)
      .then(function () {
        $state.transitionTo("login");
      });
  };
});
```



Task-2: Extending Library Management Functionality

Sub-task 1

Making the playlists public, private and collaborative.

Existing System

Users can create playlists by clicking the “ADD TO PLAYLIST” button which displays the option to either create a playlist or add songs to already existing playlists, but they are private.

Modified System

Now users can create public playlists as well. While making a playlist, they can choose the option "public". Implemented the above feature using “DECORATOR PATTERN”.

Decorator pattern

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Benefits of using Decorator pattern:

- **Enhances code reusability**: The Decorator pattern allows you to add new functionality to an existing object without modifying its original code. This helps in keeping the code modular and makes it easier to reuse existing code.
- **Provides flexibility**: With the Decorator pattern, you can add or remove behavior from an object at runtime. This flexibility allows you to create complex behavior by combining multiple decorators.
- **Simplifies code maintenance:** When you use the Decorator pattern, you can add or remove functionality by simply adding or removing decorators. This simplifies code maintenance and reduces the chances of introducing bugs.
- **Allows incremental feature development**: The Decorator pattern allows you to add new features to an application incrementally without impacting the existing functionality. This helps in avoiding large code refactoring efforts and reduces the risk of introducing new bugs.
- **Supports open/closed principle**: The Decorator pattern follows the open/closed principle, which states that software entities should be open for extension but closed for modification. With the Decorator pattern, you can extend the functionality of an object without modifying its original code, thus adhering to the open/closed principle.

Code Changes

```java
package com.sismics.music.core.model.dbi;

6 usages  4 implementations
public interface PlaylistInterface {

    2 implementations
    public String getId();
    2 implementations
    public void setId(String id);
    2 implementations
    public String getUserId();
    2 implementations
    public String getName();
    2 implementations
    public void setName(String name);
    2 implementations
    public void setUserId(String userId);

}
```

Figure 1 Interface component

```java
public class Playlist implements PlaylistInterface{
    /**
     * Playlist ID.
     */
    7 usages
    private String id;

    public Playlist() {
    }

    /**
     * User ID.
     */
    5 usages
    private String userId;

    /**
     * Playlist name.
     */
    4 usages
    private String name;

    public Playlist(String id) { this.id = id; }

    public Playlist(String id, String userId) {
        this.id = id;
        this.userId = userId;
    }
    public Playlist(String id, String userId,String name) {
        this.id = id;
        this.userId = userId;
```

Figure 2 Concrete Component

```java
package com.sismics.music.core.model.dbi;

import com.google.common.base.Objects;

2 usages  2 inheritors
public class BaseDecorator implements PlaylistInterface{
    protected  PlaylistInterface wrapper;
    2 usages
    public BaseDecorator(PlaylistInterface wrapper) { this.wrapper = wrapper; }

    @Override
    public String getId() { return wrapper.getId(); }


    @Override
    public void setId(String id) {
        wrapper.setId(id);

    }
    @Override
    public String getUserId() { return wrapper.getUserId(); }

    @Override
    public String getName() { return wrapper.getName(); }

    @Override
    public void setName(String name) {
        wrapper.setName(name);

    }
    @Override
    public void setUserId(String userId) { wrapper.setUserId(userId); }
```

Figure 3 Base Decorator

```java
package com.sismics.music.core.model.dbi;

17 usages
public class PlaylistDecorator extends BaseDecorator{
    3 usages
    private boolean type;

    2 usages
    public PlaylistDecorator(PlaylistInterface wrapper,boolean type) {
        super(wrapper);
        this.type=type;
    }

    public boolean getType() { return type; }

    public void setType(boolean type) { this.type = type; }

}
```

Figure 4 Concrete Decorator

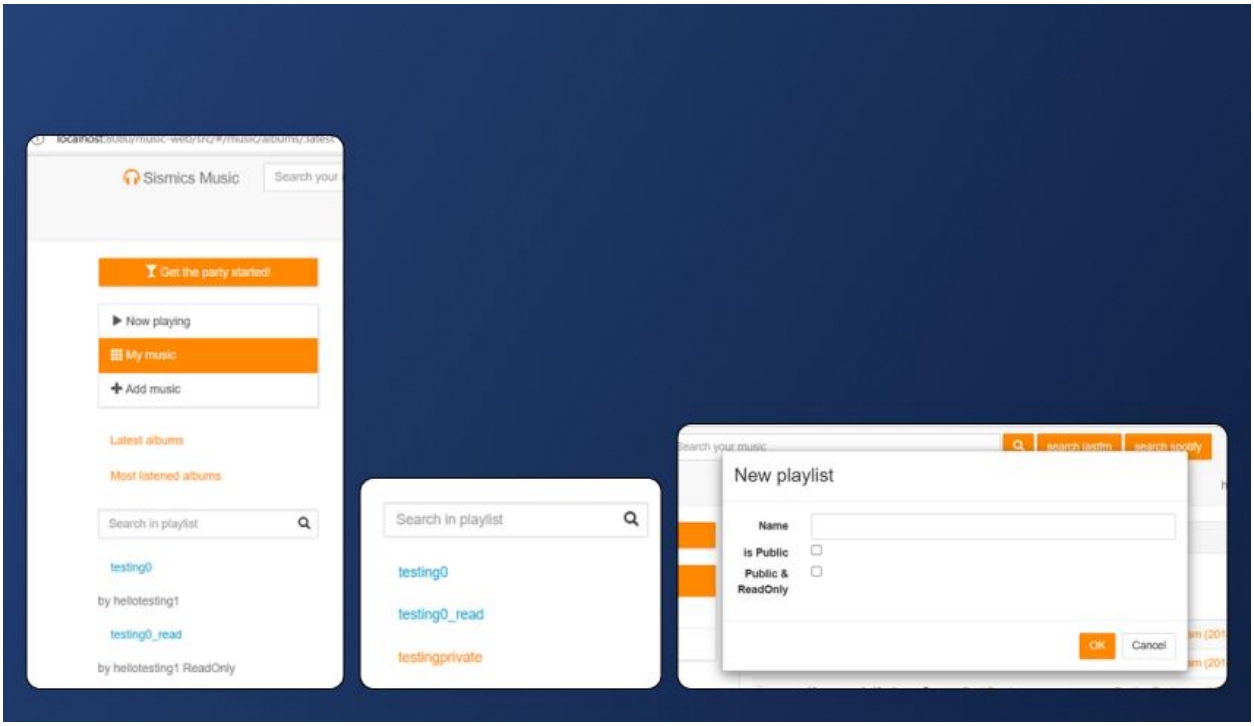
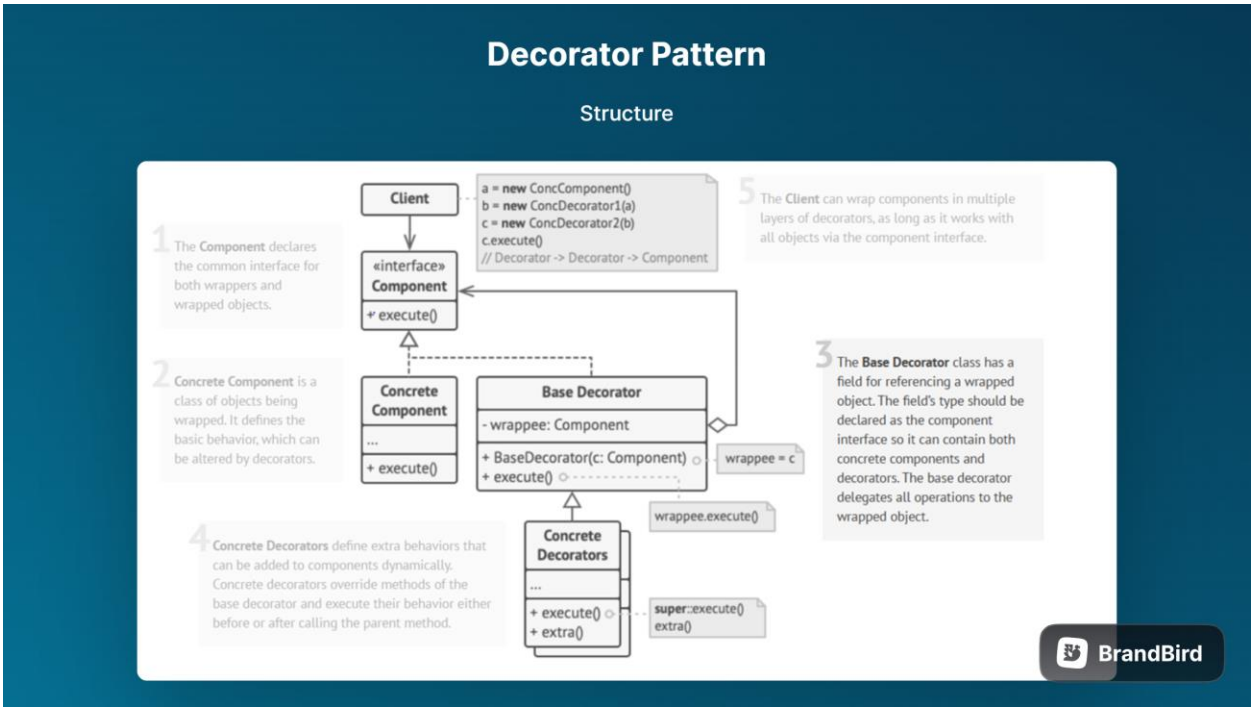


Figure 5 Demo of creation and display of playlists.



Sub-task 2

Making songs added by a user hidden from other users.

Existing System

In the existing system when a user adds any songs, a part of the user uploaded, all users can be able to view and listen to the songs. However, some users may not appreciate this feature. Therefore, to address this concern, the code base has been modified to ensure that songs uploaded by a user remain private to that user only. This means that other users will not be able to view or listen to these songs, thereby providing greater privacy and control over one's music library. With this change, users can enjoy their music without having to worry about unwanted access or sharing.

Approach Taken

Storing user-id for the songs uploaded

To store the user-id for the songs uploaded, we have made changes to the database table T_Album by adding an extra column named User-Id. This modification required changes to be made in the corresponding files to ensure that the data is stored and retrieved correctly. Specifically, modifications were made to the database queries in the ALBUMDAO file to include the user-id while querying and filtering the albums from the database. These changes ensure that only user-specific data is retrieved and displayed

Uploading Music

Storing the user-id while uploading files is a complicated task. we have made changes to the existing code. The songs that are added are stored using the org.jaudiotagger.tag package in Java. While tagging the files, we have added an extra attribute called CUSTOM1 in the tag field key attribute, which contains the user-id information. Once the file is successfully uploaded, an event is triggered to make changes to the backend of the database, which is available in the Music core package CollectionService.java file. With these modifications, the user-id is automatically attached to the uploaded files, making it easier to track and manage files specific to each user. This file is responsible for creating the Albums and Artists so changes are made accordingly to achieve the task.

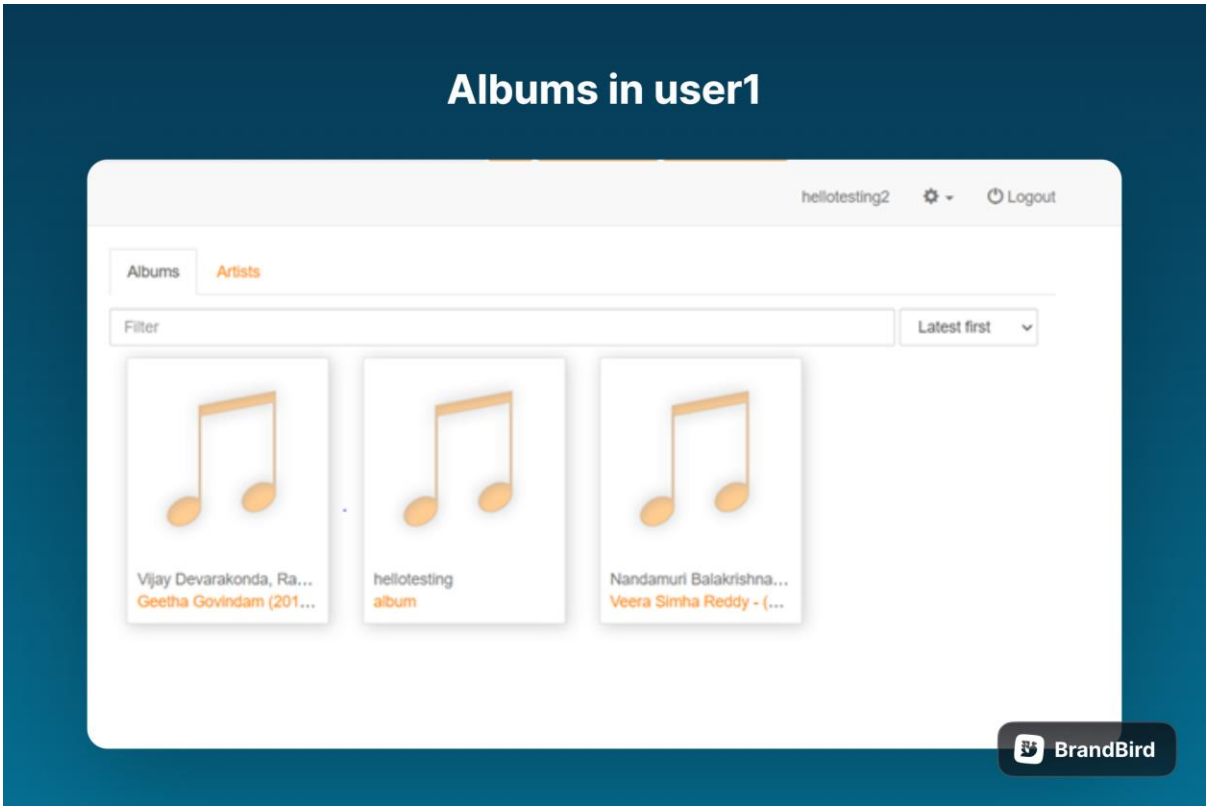
Querying the Backend

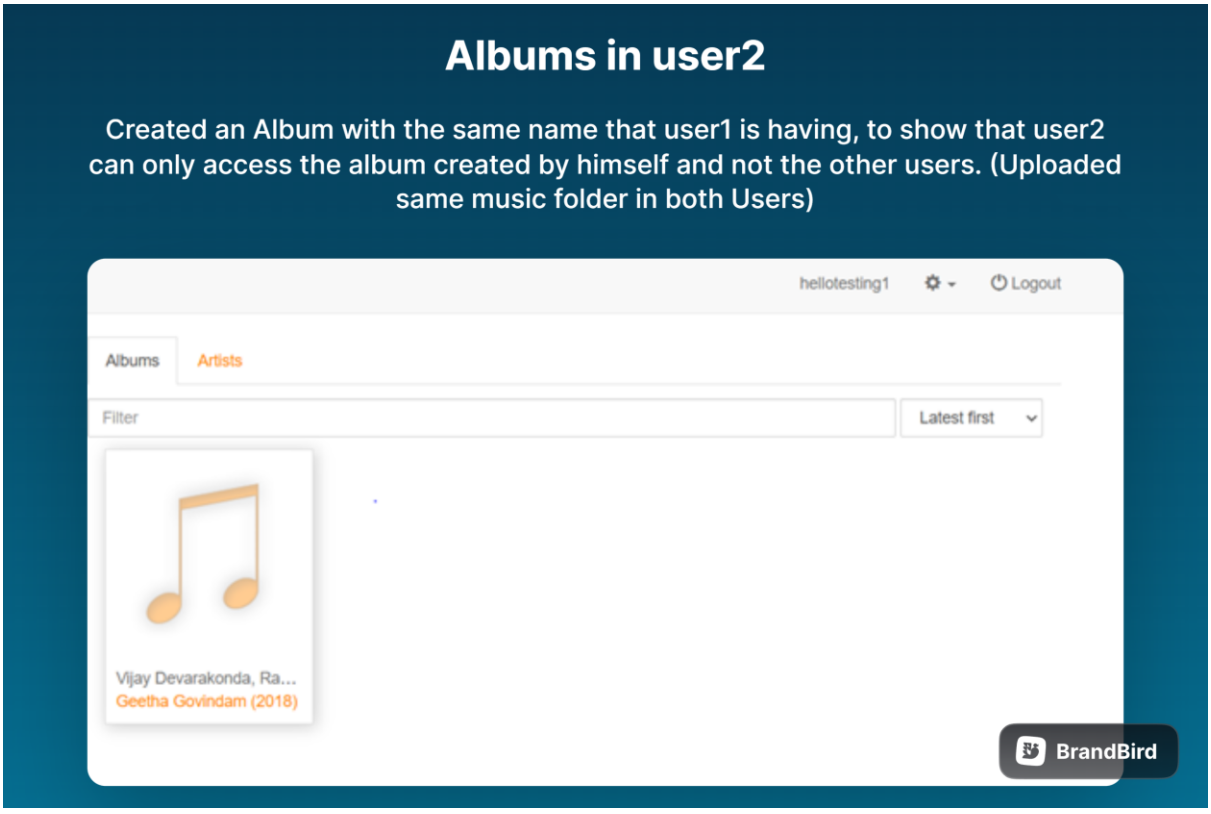
Whenever a request is made from the front-end to the backend, modifications are made in the AlbumResource.java and ArtistResource.java files. Specifically, the current user-id of the logged-in user is included in the database query, ensuring that only user-specific data is retrieved and sent back. As a result, the database queries are tailored to the user, and the retrieved data is restricted to what the user has access to.

Code Changes:

- music-core\src\main\java\com\sismics\music\core\dao\dbi\AlbumDao.java
- music-core\src\main\java\com\sismics\music\core\dao\dbi\dto\AlbumDto.java
- music-core\src\main\java\com\sismics\music\core\dao\dbi\mapper\AlbumDtoMapper.java
- music-core\src\main\java\com\sismics\music\core\dao\dbi\mapper\AlbumMapper.java
- music-core\src\main\java\com\sismics\music\core\model\dbi\Album.java
- music-core\src\main\java\com\sismics\music\core\service\collection\CollectionService.java
- music-core\src\main\java\com\sismics\music\core\service\importaudio\ImportAudioFile.java
- music-core\src\main\java\com\sismics\music\core\service\importaudio\ImportAudioService.java
- music-core\src\main\resources\db\update\dbupdate-001-0.sql
- music-web\src\main\java\com\sismics\music\rest\resource\AlbumResource.java
- music-web\src\main\java\com\sismics\music\rest\resource\ArtistResource.java
- music-web\src\main\java\com\sismics\music\rest\resource\ImportResource.java

Screenshots





Task-3: Third Party Search and Recommendation

The existing system has barebones Lastfm integration such as linking the Lastfm account and scrobbling the tracks. Our task is to extend this connectivity to search and recommendations using both lastfm and Spotify.

Flow of Searching for Tracks

- After successfully signing the application. Users can enter the search query in the input box present in the navigation bar.
- By default, All the songs/albums/artists including the user-uploads, Lastfm and Spotify will be listed.
- Separate buttons for Lastfm and Spotify are provided to narrow down the search results.

Steps Involved for Search Functionality

- Added two buttons in the navigation bar for Lastfm and Spotify search.
- `navigation.js` controller is updated with two command functions which call the concrete commands in `search.js` controller via $rootscope.
- `search.js` controller is updated to make get http requests to the backend on routes – "/search/lastfm" and "/search/spotify" respectively.
- `SearchResource.java` REST file is updated with route-handlers for the above-mentioned routes.
- These handlers utilize the strategy pattern and call the appropriate concrete strategy handlers.
- `search.html` is updated to show the results from the api network calls.

Code Changes for Search Functionality

- [MODIFY] index.html
- [MODIFY] search.html
- [MODIFY] navigation.js
- [MODIFY] search.js
- [MODIFY] SearchResource.java
- [ADD] strategies/search/SearchStrategy.java
- [ADD] strategies/search/LastFmSearchStrategy.java
- [ADD] strategies/search/SpotifySearchStrategy.java
- [ADD] core/service/spotify/SpotifyService.java
- [MODIFY] core/service/lastfm/LastFmService.java

Flow of Getting Recommendations from Playlist Tracks

- After successfully signing the application. Users can select the playlist of their liking.
- Two buttons are provided, one for Lastfm recommendations and the other for Spotify recommendations.
- Upon clicking them, we fetch the list of recommended songs based on the current playlist tracks and displayed below.

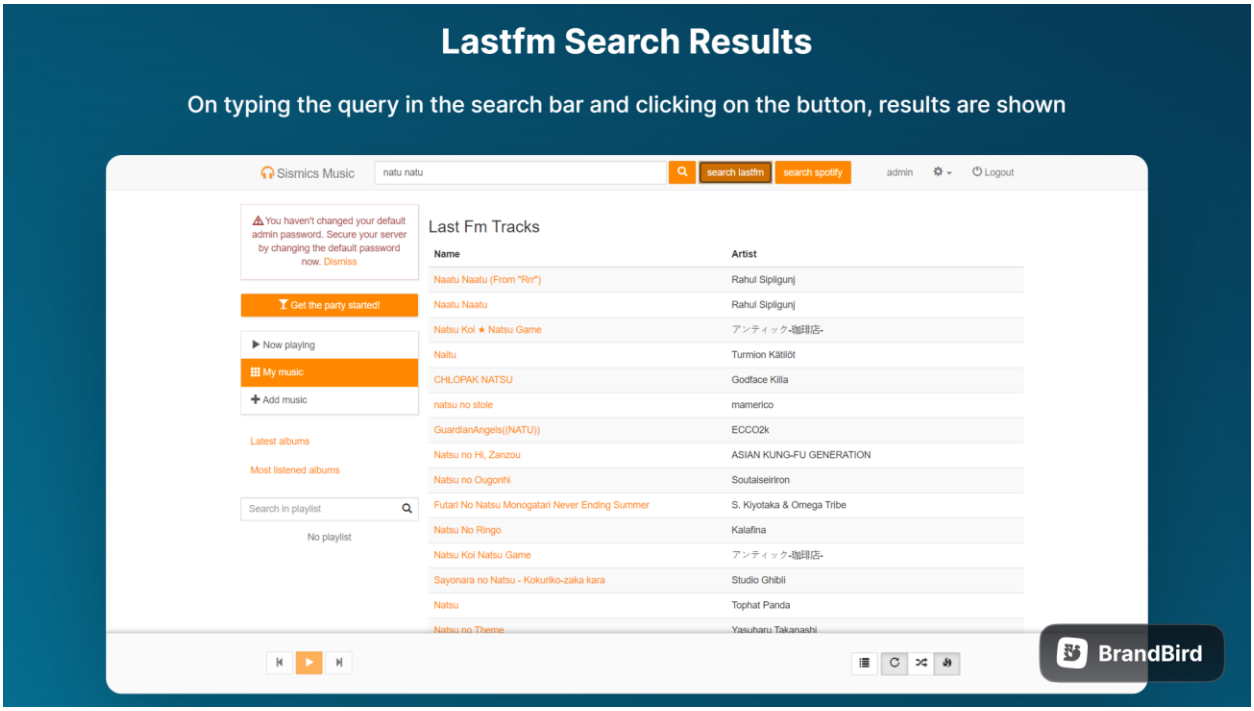
Steps Involved for Recommendation Functionality

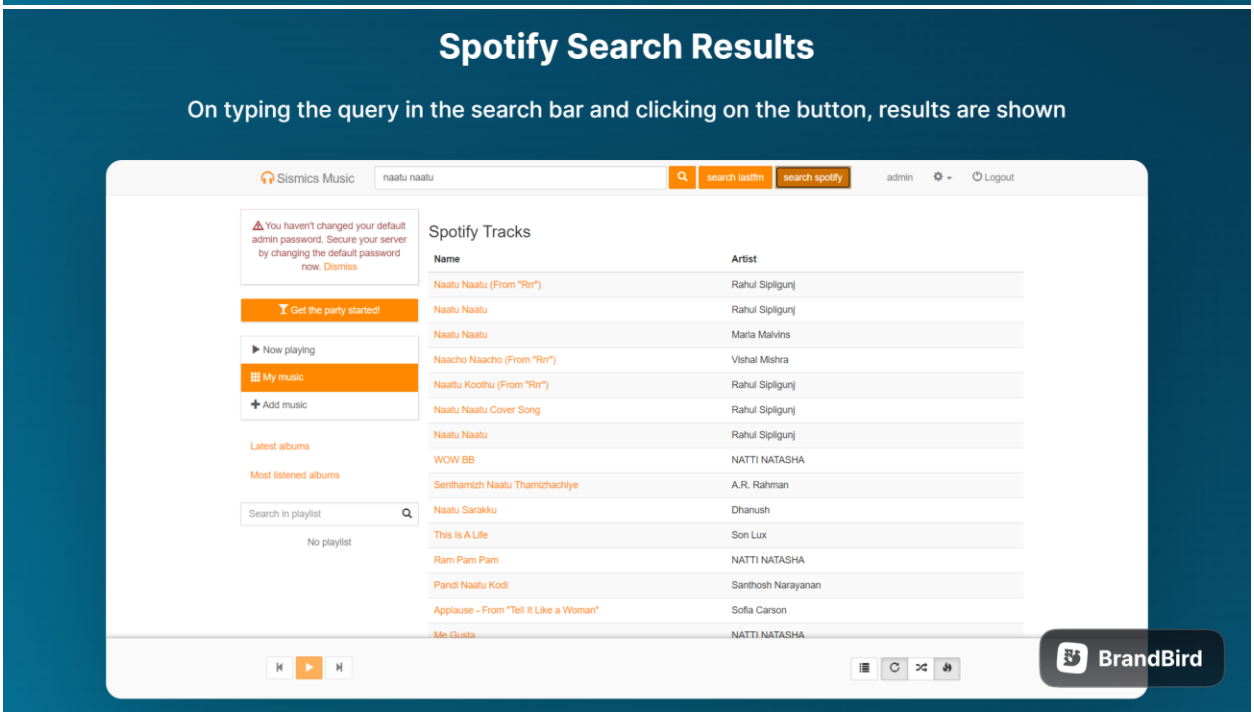
- Added two buttons in the playlist page for Lastfm and Spotify recommendations.
- `playlist.js` controller is updated to make get http requests to the backend on routes – "/playlist/recommend/spotify" and "/playlist/recommend/lastfm" respectively.
- `PlaylistResource.java` REST file is updated with route-handlers for the above-mentioned routes.
- These handlers utilize the strategy pattern and call the appropriate concrete strategy handlers.
- We utilized the Lastfm and Spotify rest-api to fetch recommendations.
- `playlist.html` is updated to show the fetched recommendations below the playlist songs.

Code Changes for Recommendation Functionality

- [MODIFY] playlist.html
- [MODIFY] playlist.js
- [MODIFY] PlaylistResource.java
- [ADD] strategies/recommend/RecommendationStrategy.java
- [ADD] strategies/recommend/LastFmRecommendStrategy.java
- [ADD] strategies/recommend/SpotifyRecommendStrategy.java
- [ADD] core/service/spotify/SpotifyService.java
- [MODIFY] core/service/lastfm/LastFmService.java

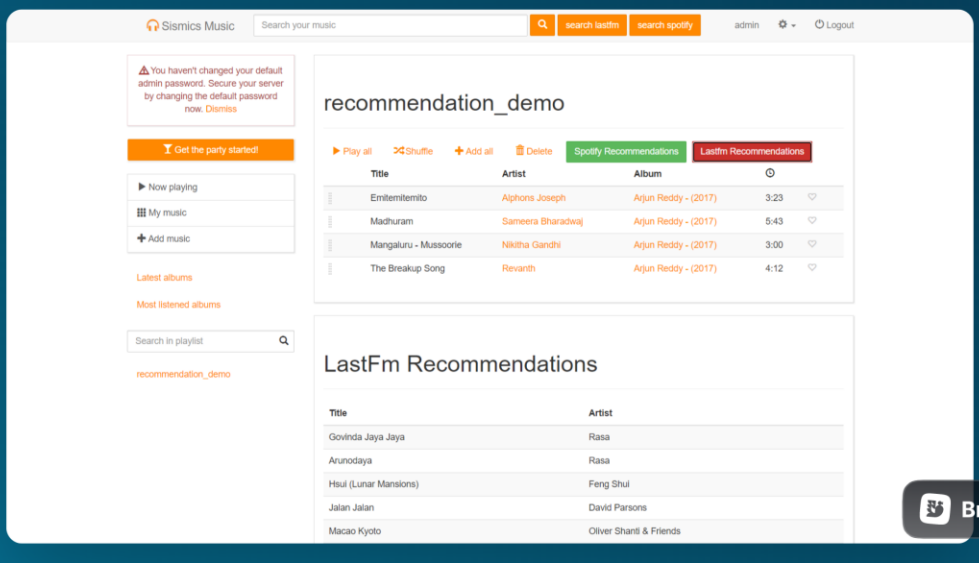
Screenshots





Lastfm Track Recommendations

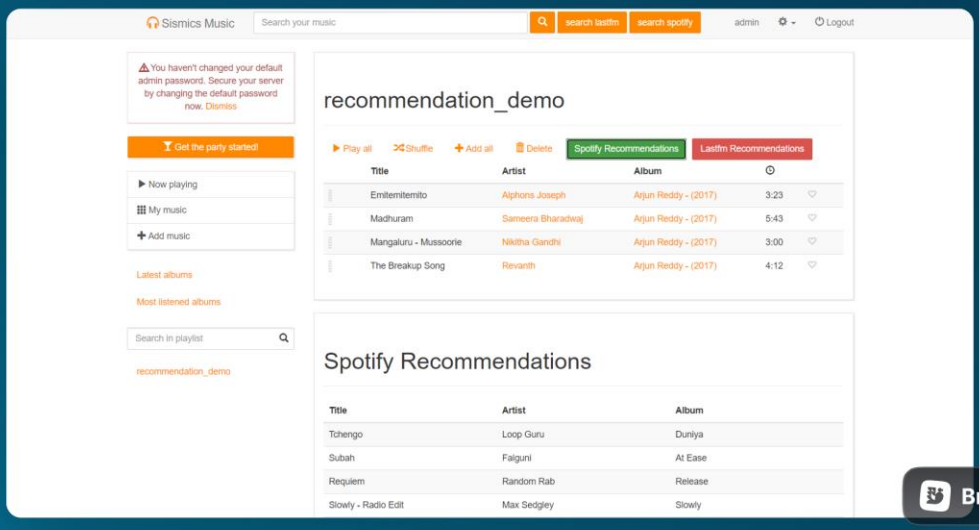
Select a playlist and click on the button to get recommendations



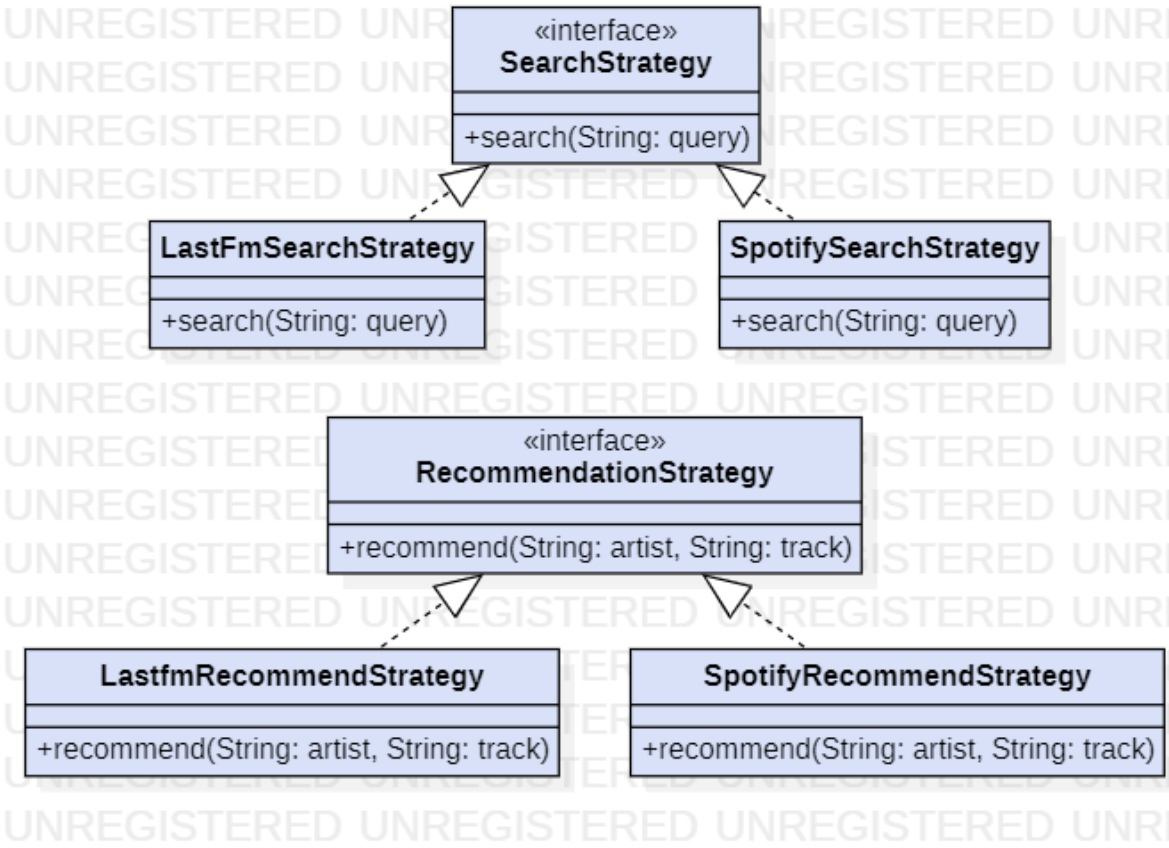
BrandBird

Spotify Track Recommendations

Select a playlist and click on the button to get recommendations



BrandBird


«interface»
SearchStrategy
+search(String: query)
LastFmSearchStrategy
+search(String: query)
SpotifySearchStrategy
+search(String: query)
«interface»
RecommendationStrategy
+recommend(String: artist, String: track)
LastfmRecommendStrategy
+recommend(String: artist, String: track)
SpotifyRecommendStrategy
+recommend(String: artist, String: track)