SWE Project - 2 | CS6.401.Software Engineering

Members

- Adhiraj Anil Deshmukh (2021121012)
- Arjun Muraleedharan (2020101099)
- Pratham Gupta (2020101080)
- Keyur Ganesh Chaudhari (2020101100)
- Sambasai Reddy Andem (2020101014)

Feature - 1: Better user management

Problem

Our music player has a user management system already in place, which you have already looked at in detail. The current flow of creating a user involves logging into an admin account and creating a new user manually. These new users can add new music to the library. However, this is neither intuitive nor convenient, and thus demands an improvement. Allow for the creation of new users directly from the login page.

Analysis

To tackle this problem, we first had to look at the codebase and find all the relevant files that affect the behaviour of this feature.

To add this feature, we would need to interact with the 2 sub-systems that we have tackled with before in Assignment - 1, namely `Administration` and `User Management`.

User

It includes the following classes:

- `User`: This is an entity class that represents a system user. It contains information about the user, such as their username, password, email, and other attributes.

- `UserDto`: This is a data transfer object (DTO) class that holds information about a user. It is used to transfer data between the application and the database.
- `UserDao`: This is a data access object (DAO) class that provides methods for accessing user data, such as retrieving active users by their username, updating user information, and deleting users. It extends a general-purpose BaseDao class that provides methods for constructing SQL queries.
- `BaseDao`: This is a general-purpose class that works with DTOs and criterion objects. It contains methods for constructing SQL queries and interacting with the database.
- `UserCriteria` and FilterCriteria: These are criterion classes that can be used to set conditions for filtering user data.
- `UserMapper` and UserDtoMapper: These are mapper classes that are used to map data between result sets and DTOs or user entities.
- `Handle`: This is a class that allows you to interact with a JDBI database handle.
- `ThreadLocalContext`: This is a context class for storing the database handle.

The use of DAO and DTO classes provides modularity to the system and helps manage user data in the application. However, there may be a higher learning curve for new users who are not experienced with server administration. The system provides the capacity to alter server settings to meet the requirements of a specific user or business.

Finally, the system also includes classes for handling user authentication and authorization. These classes include `UserResource`, `UserDao` and `UserDto`. The `UserResource` class is responsible for handling user-related requests, such as registering, updating and deleting users. It also supports login, logout and retrieving user information. The `UserDao` class provides access to the user data stored in the database, while the `UserDto` class represents the data of the user.

User Privileges

It includes the following classes:

- `Role`: This is an entity class that represents a user role. It contains information about the role, such as its name and description.
- `Privilege`: This is an entity class that represents a user privilege. It contains information about the privilege, such as its name and description.
- `RolePrivilege`: This is a mapping class that maps privileges to roles. It contains information about which roles have which privileges.
- `PrivilegeMapper`: `PrivilegeMapper` implements a ResultSetMapper for mapping ResultSet data to `Privilege` objects.
- `RoleMapper`: RoleMapper implements a ResultSetMapper for mapping ResultSet data to `Role` objects.
- `RolePrivilegeMapper`: `RolePrivilegeMapper` implements a ResultSetMapper for mapping ResultSet data to `RolePrivilege` objects.

User Resource

- `BaseResource`: This is a general-purpose resource class that handles requests to different types of resources. It uses the Role, Privilege, and RolePrivilege classes to authenticate requests and check if the user has the required privileges to access certain resources
- `DirectoryResources`: This is a resource class that handles requests to directory-related resources. It uses the `BaseResource` class to authenticate requests.
- `UserResource`: This is a resource class that handles requests to user-related resources. It also uses the `BaseResource` class to authenticate requests.

Changes

- Based on the analysis above, we came to the conclusion that we don't need to revamp how the User management or Administration system, works.

  We just can change the privileges required to create a user account from admin to anonymous, and then additionally we can just create a registration page which is linked to the login page at the start of the website when the user is in anonymous state (logged out state).
- The team also felt that no big changes to the Design of these Systems need to be made in order to incorporate these features effectively.

  As the User/Administration Management System already uses **DAO** design patterns, these changes are effectively decoupled in terms of their definition and access instances.
- We have extended **Builder** Design Pattern for the `User` , `UserDto` and `UserCriteria` classes which are used effectively by `UserMapper` and `UserDao` classes to create / update new instances of these classes.

1) Backend

UserResource.java

- This file has all the REST Api implemented related to User Management Subsystem.
- Here the only change we had to make was to allow *Anonymous* users to register a new account, while processing request at `@PUT /user`.
- This required us to remove the restrictions of having the user be *Authenticated* and also remove the restriction of having *Admin* Privilege.

```
11 music-web/src/main/java/com/sismics/music/rest/resource/UserResource.java
@@ -62,11 +62,12 @@ public Response register(
62      @FormParam("password") String password,
63      @FormParam("locale") String localeId,
64      @FormParam("email") String email) {
65  -
66  -   if (!authenticate()) {
67  -     throw new ForbiddenClientException();
68  -   }
69  -   checkPrivilege(Privilege.ADMIN);
70
71      // Validate the input data
72      username = Validation.length(username, "username", 3, 50);
```

2) Frontend

App.js

- Add a new state `register`
- Add templateUrl for the new state `partial/register.html`
- Add Controller to the new state as `Register.js`

Login.js

- Add a funciton `goto_register` which transitions the state from `login` to `regsiter` and redirects to `register.html`

Register.html

- New file added to create a page to implement Registration funcionaity
- The structure is almost very similar to the one implemented in `user.settings.edit.html`

Register.js

- Controller for the `register` state
- Implements the functionality for making calls REST Api for interacting with the backend database for creating new user.

```
'use strict';

/**
 * Register controller.
 */
angular.module('music').controller('Register', function($rootScope, $scope, $state, $dialog, User, Restangular) {

  $scope.checkUsername = function(username) {
    User.checkUsername(username).then(function(data) {
      $scope.reg.valid = data;
    }, function() {
      var title = 'Usercheck failed';
      var msg = 'Username invalid'
      var btns = [{ result:'ok', label: 'OK', cssClass: 'btn-primary' }];

      $dialog.messageBox(title, msg, btns);
    });
  };

  $scope.register = function(user) {

    $scope.reg = user
    $scope.reg.valid = false;

    User.register(user).then(function() {
      //$scope.checkUsername(user.username);
      $state.transitionTo('login');
    }, function() {
      var title = 'Registration failed';
      var msg = 'Username or password invalid'
      var btns = [{ result:'ok', label: 'OK', cssClass: 'btn-primary' }];

      $dialog.messageBox(title, msg, btns);
    });
  };


  $scope.goto_login = function($event) {
    $state.transitionTo('login');
    $event.preventDefault();
  };

});
```

Design Patterns

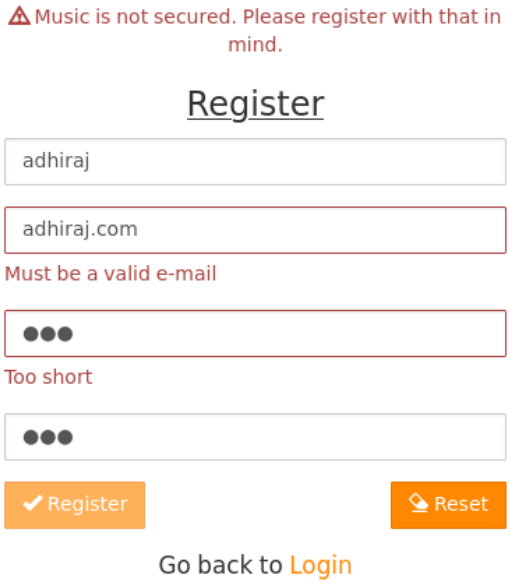
- **DAO** Patterns:
  The Design Pattern is retained from the original codebase, as we found it effective to decouple the creation and access part of the User Management System.

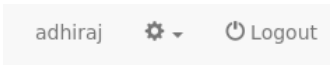
```java
// Update the user
UserDao userDao = new UserDao();
User user = userDao.getActiveByUsername(principal.getName());
if (email != null) {
    user.setEmail(email);
}
if (localeId != null) {
    user.setLocaleId(localeId);
}
if (firstConnection != null && hasPrivilege(Privilege.ADMIN)) {
    user.setFirstConnection(firstConnection);
}

user = userDao.update(user);
```

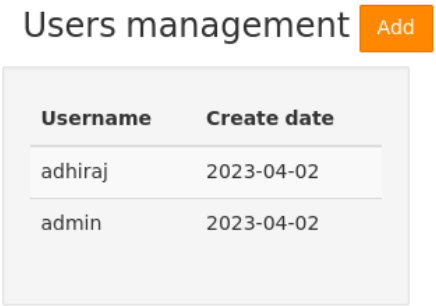
- **Builder** Pattern:
  The Builder Pattern is used for the step-by-step creation of objects like:
  - `User`
  - `UserDto`
  - `UserCriteria`

```java
// 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());

if (localeId == null) {
    // Set the locale from the HTTP headers
    localeId = LocaleUtil.getLocaleIdFromAcceptLanguage(request.getHeader(name: "Accept-Language"));
}
user.setLocaleId(localeId);
```

Screenshots







Feature - 2: Better library management

Problem

In the current codebase, songs added by a user to Music are visible to everyone. This is not ideal - not everyone wants to share all their music. Modify Music so that songs added by a user can be hidden from other users.

Additionally, playlists created by a user are visible only to them. We want to modify Music to allows users to choose whether their created playlists should be visible to other users or not. A public playlist can be interacted with by other users (e.g. songs can be added to it).

Analysis

This feature integration was very backend heavy and forced us to have a good understanding of how multiple subsystems such as `Album`, `Playlist`, `Directory`, `User`, `Admin`, etc. with the `Library` Subsystem interacted with eachother and the database.

Database Interaction

- `Handle`: This is a class that allows you to interact with a JDBI database handle.
- `ThreadLocalContext`: This is a context class for storing the database handle.

Album

This Java class defines the entity of a `Album` that is used in a music application. The class has several instance variables that store information about the album, including its ID, directory ID, artist ID, name, album art, creation date, update date, deletion date, and location. The class provides three static methods for creating, updating, and deleting a named album, which use the `AlbumDao` class. The code also includes a static method `getActiveById()` that returns an active album object by its ID. The method uses an instance of the Handle class from JDBI, a database access library, to execute a SQL query on the `t_album` table, which returns an album object that matches the specified ID and has a null `deleteDate` value.

AlbumDto

The code defines a data transfer object (DTO) for an album. It has several private fields including album ID, name, album art ID, artist ID, artist name, last update date, and user play count. It has getters and setters for each of these fields. The class is designed to store and transfer information about an album between different layers of an application, typically between the database and the service layer. The code does not include any functionality, but it provides a structure for organizing album information and transferring it between components.

AlbumDao

This code defines the `AlbumDao` class for the music application. It provides methods for creating, updating, and querying albums in a database using SQL. The class extends the BaseDao class and defines its own implementation of the getQueryParam method to define a SQL query for retrieving `AlbumDto` objects based on the supplied AlbumCriteria object. The create method inserts a new `Album` object into the database using a generated UUID for the ID field. The update method updates an existing `Album` object in the database using its ID field as the primary key. Finally, the updateDate method updates only the updatedate column of an album in the database using its ID field as the primary key. The class uses the jdbi library to execute SQL statements and map results to objects using the `AlbumMapper` and `AlbumDto` Mapper classes. The class also uses the ThreadLocalContext class to manage database connections.

AlbumArtService

The `AlbumArtService` class provides functionality for importing, deleting, and getting album art images for an album. The service is used to update an album with a unique ID for the album art, import the original image and create resized versions for different purposes such as album covers and thumbnails.

AlbumMapper

This code defines a result set mapper for the `Album` class. It extends a `BaseResultSetMapper` class and implements the `map` method to map the query results to an `Album` object. It also defines the columns to be fetched by overriding the `getColumns` method.

AlbumResource

`AlbumResource` class represents RESTful resources for a playlist app. It provides various API endpoints to manage playlists, such as creating a playlist, updating a playlist, inserting tracks into a playlist, and deleting a playlist.

The class extends another class called BaseResource and contains private methods to authenticate, generate JSON responses, and build JSON output for playlists with tracks. Each API endpoint starts with an authentication check, calls a DAO to interact with the database, and generates the response in JSON format.

The code uses various DAO classes, such as `AlbumDao`, `AlbumTrackDao`, and `TrackDao`, to interact with the database and retrieve the necessary information. Additionally, the code uses `PaginatedLists` , `AlbumCriteria` and `SortCriteria` to support pagination, searching and sorting.

Very similarly, we can also describe the behaviour of `Playlist` and `Directory` Sub-Systems, which were also significant part of our changes to incorporate the features.

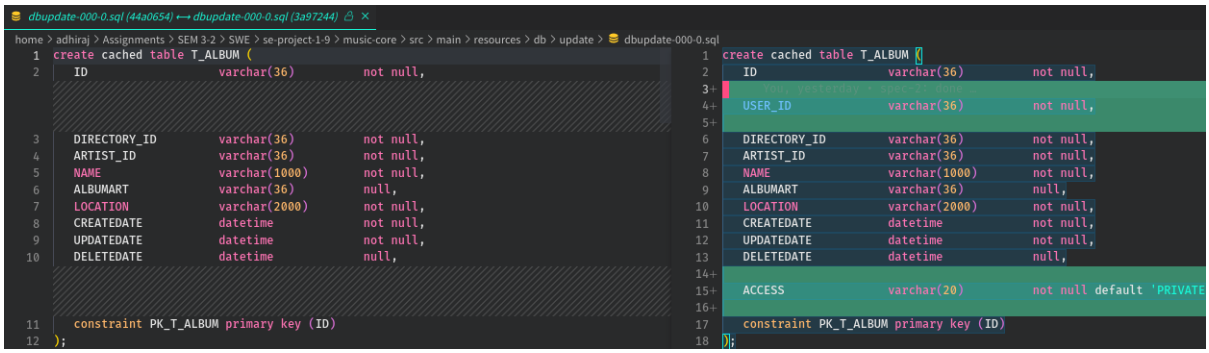
Changes

- To incorporate these features firstly we had to change the SQL database tables `T_Albums` and `T_Playlists` to add an attribute called `ACCESS` which indicated if the Album/playlist was visible to other users.
- `T_Albums` also needed an extra attribute for incorporating the `USER_ID` of the creator account.
- `Model` , `Dto` , `Dao` , `Criteria` classes of Album/Playlist sub-systems had to be changed to incoporate the changes made in the Database Tables.
- Significant changes were made in `Resource` files of these sub-systems to incorporate this attribute and also handle `Criterias` based on which these instances are fetched, searched or sorted from the database.
- Lastly, the frontend had minimal changes to incorporate this feature since we only had to add a button which controlled / indicated the current state of the Playlist/Album as `Private` or `Public`.

> **Note**: All of the changes show below are for *Albums* sub-system, but are also very similarly incorporated for *Playlist* and *Artist* sub-system.

1) Backend

dbupdate-000-0.sql



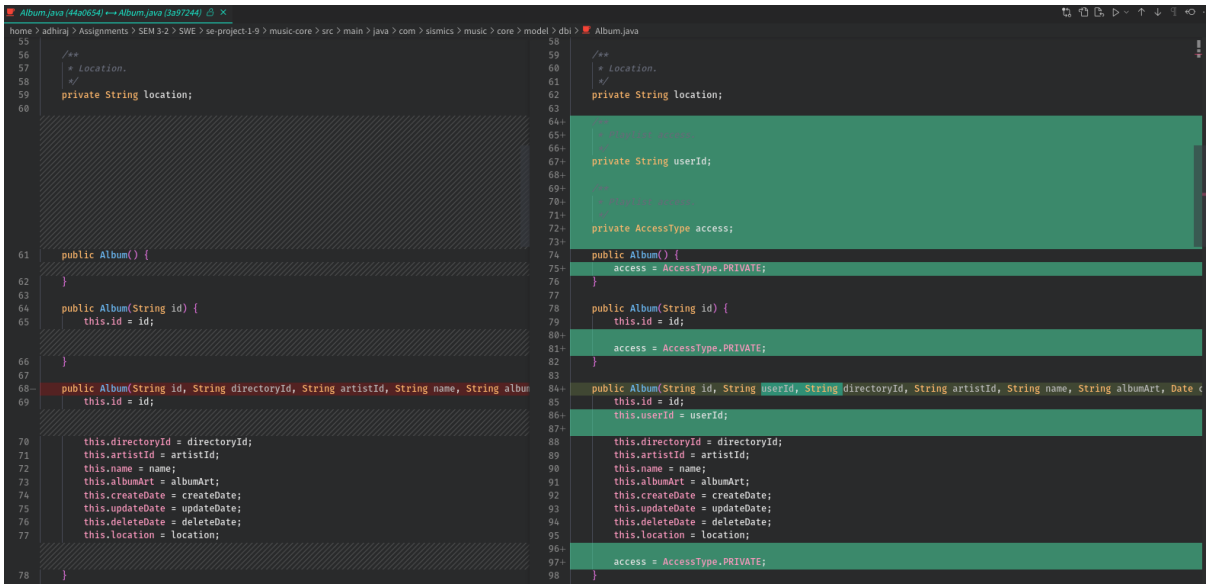
- Add the attributes `user_id` and `access`
- `access` has a default value of `PRIVATE` , this is done so that the we don't have to make several changes to the existing API / Dao when accessing the table attributes.

AccessType.java

```java
package com.sismics.music.core.constant;

public enum AccessType {

    PUBLIC("PUBLIC"),
    PRIVATE("PRIVATE");

    private String name;

    AccessType(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

- This additional file is made to define the data type of the attribute `Access` as `AccessType`.
- `AccessType` is a Enum which has 2 defined values indicating `Public` or `Private`.

Album.java / AlbumDto.java



- Incorporate the `access` attribute
- Add `userId` attribute to for identifying the ownership of the Albums.

```java
+    /**
+     * Getter of userId.
+     *
+     * @return id
+     */
+    public String getUserId() {
+        return userId;
+    }
+
+    /**
+     * Setter of id.
+     *
+     * @param id id
+     */
+    public void setUserId(String userId) {
+        this.userId = userId;
+    }
+
+
+    public AccessType getAccess() {
+        return access;
+    }
+
+    public void setAccess(String access) {
+        switch(access) {
+            case "PUBLIC":
+                this.access = AccessType.PUBLIC;
+                break;
+            case "PRIVATE":
+                this.access = AccessType.PRIVATE;
+                break;
+            default:
+                throw new IllegalArgumentException("Invalid access: " + access);
+        }
+    }
+
```

- `setAccess` a way that JavaScript native Data Types can interact with the user defined `AccessType` Data Type.
- `AlbumDto` has very similar changes.

AlbumDtoMapper.java

```java
public class AlbumDtoMapper implements ResultSetMapper<AlbumDto> {
    @Override
    public AlbumDto map(int index, ResultSet r, StatementContext ctx) throws SQLException
        AlbumDto dto = new AlbumDto();
        dto.setId(r.getString("id"));
        dto.setName(r.getString("c0"));
        dto.setAlbumArt(r.getString("albumArt"));
        dto.setArtistId(r.getString("artistId"));
        dto.setArtistName(r.getString("artistName"));
        dto.setUpdateDate(r.getTimestamp("c1"));
        dto.setUserPlayCount(r.getLong("c2"));
        return dto;
    }
}
```

```java
public class AlbumDtoMapper implements ResultSetMapper<AlbumDto> {
    @Override
    public AlbumDto map(int index, ResultSet r, StatementContext ctx) throws SQLException {
        AlbumDto dto = new AlbumDto();
        dto.setId(r.getString("id"));
        dto.setName(r.getString("c0"));
        dto.setAlbumArt(r.getString("albumArt"));
        dto.setArtistId(r.getString("artistId"));
        dto.setArtistName(r.getString("artistName"));
        dto.setUpdateDate(r.getTimestamp("c1"));
        dto.setUserPlayCount(r.getLong("c2"));
+       dto.setUserId(r.getString("userId"));
+       dto.setAccess(r.getString("access"));
        return dto;
    }
}
```

- Also fetches the `userId` and `access` from the database when mapping to a `AlbumDto`

AlbumCriteria.java

```java
+    public AccessType getAccess() {
+        return access;
+    }
+
+    public AlbumCriteria setPublic(boolean flag) {
+        this.access = flag ? AccessType.PUBLIC : AccessType.PRIVATE;
+        return this;
+    }
}
```

- Also incorporates the `access` attributes for the database query criteria.
- Has `setPublic(boolean)` function instead of `setAccess(AccessType)` setter, in order to easily set criteria based on access from the frontend file `AlbumResource` without the knowledge of `AccessType` data type on the frontend part of the

codebase.

AlbumDao

```java
67+         if (criteria.getUserId() ≠ null) {
68+             criteriaList.add("a.user_id = :userId");
69+             parameterMap.put("userId", criteria.getUserId());
70+         }
71+
72+         if (criteria.getAccess() ≠ null) {
73+             criteriaList.add("a.access = :access");
74+             parameterMap.put("access", criteria.getAccess().toString());
75+         }
76+
```

- Addition to `Criteria` fetch requests based on `user_id` and `access` attributes.

```java
80+     /**
81+      * Update access.
82+      *
83+      * @param playlist Playlist to update
84+      */
85+     public static void updateAccess(Album album) {
86+         final Handle handle = ThreadLocalContext.get().getHandle();
87+         handle.createStatement("update t_album " +
88+                 "  set access = :access" +
89+                 "  where id = :id")
90+                 .bind("access", album.getAccess().toString())
91+                 .bind("id", album.getId())
92+                 .execute();
93+     }
94+
```

- `updateAccess` function in the `Dao` to update `access` of a playlist based on the `album_id` key.
- This change is directly not made in the pre-existing `update` function of the `Dao` to decouple some of the logic in how we can update this attribute and also to not make additional changes on the existing code where `update` function is called without the requirement of changes required in `access`

```java
82      Handle handle = ThreadLocalContext.get().getHandle();
83      handle.createStatement("insert into " +
84-             " t_album(id, directory_id, artist_id, name, albumart, createdate, updatedate, locati
85-             " values(:id, :directoryId, :artistId, :name, :albumArt, :createDate, :updateDate, :lo
86              .bind("id", album.getId())
```

```java
113     Handle handle = ThreadLocalContext.get().getHandle();
114     handle.createStatement("insert into " +
115+             " t_album(id, user_id, directory_id, artist_id, name, albumart, createdate, updatedat
116+             " values(:id, :user_id, :directoryId, :artistId, :name, :albumArt, :createDate, :upda
117             .bind("id", album.getId())
118+
119+             .bind("user_id", album.getUserId())
120+
```

- The creation of `Album` instance with `user_id` attribute in the database is also added in the `create` function of the `AlbumDao`.

CollectionService.java

```java
262+                 You, yesterday • spec-2: done …
263+             // Setting the context of logged in user
264+             album.setUserId(AppContext.getInstance().getUserId());
265+
```

- Setting the `userId` form `AppContext`, when `Album` object instance is created in `CollectionService`

AlbumResource.java

```
65-     // Get album info
66-     AlbumDao albumDao = new AlbumDao();
67-     List<AlbumDto> albumList = albumDao.findByCriteria(new AlbumCriteria().setUserId(principal.get
68-     if (albumList.isEmpty()) {
69          return Response.status(Response.Status.NOT_FOUND).build();
70      }
71-     AlbumDto album = albumList.iterator().next();
72
73      JsonObjectBuilder response = Json.createObjectBuilder()
74              .add("id", album.getId())
75              .add("name", album.getName())
76              .add("albumart", album.getAlbumArt() != null)
77              .add("play_count", album.getUserPlayCount());
78
79      response.add("artist", Json.createObjectBuilder()
80              .add("id", album.getArtistId())
81-             .add("name", album.getArtistName()));
```

```
66+     AlbumDto album = null;
67+
68+     AlbumCriteria criteria1 = new AlbumCriteria()
69+         .setUserId(principal.getId())
70+         .setPublic(false)
71+         .setId(id);
72+
73+     AlbumDto album1 = new AlbumDao().findFirstByCriteria(criteria1);
74+
75+     if (album1 != null) {
76+         album = album1;
77+     }
78+
79+     AlbumCriteria criteria2 = new AlbumCriteria()
80+         .setPublic(true)
81+         .setId(id);
82+
83+     AlbumDto album2 = new AlbumDao().findFirstByCriteria(criteria2);
84+
85+     if (album2 != null) {
86+         album = album2;
87+     }
88+
89+     if (album == null) {
90+
91          return Response.status(Response.Status.NOT_FOUND).build();
92      }
93
94      JsonObjectBuilder response = Json.createObjectBuilder()
95              .add("id", album.getId())
96+             .add("userId", album.getUserId())
97              .add("name", album.getName())
98              .add("albumart", album.getAlbumArt() != null)
99              .add("play_count", album.getUserPlayCount());
100
101     response.add("artist", Json.createObjectBuilder()
102             .add("id", album.getArtistId())
103+            .add("name", album.getArtistName()))
104+            .add("access", album.getAccess().toString())
105+            .add("isOwner", (album.getUserId() != null && album.getUserId().equals(principal.getId
```

- `@GET /albums/:id` request function uses `findFirstByCritera` function instead of `findByCriteria` from `BaseDao` just to remove the redundance of using `List` and making the code more modular.
  - We also add one more criteria of featching Albums from other users which have `access` as `PUBLIC`.
  - We also add more items in the `response` object corresponding to `isOwner` and `access` values of the Album, which are useful in the frontend part of the codebase.

```
293+    @POST
294+    @Path("{id: [a-z0-9\\-]+}/access")
295+    public Response editAccess(
296+            @PathParam("id") String id,
297+            @FormParam("access") String access) {
298+
299+        if (!authenticate()) {
300+            throw new ForbiddenClientException();
301+        }
302+
303+        access = access.toUpperCase();
304+
305+        // get the album
306+        Album album = Album.getActiveById(id);
307+        notFoundIfNull(album, "id");
308+
309+        if (album.getAccess().toString().equals(access)) {
310+            return okJson();
311+        }
312+
313+        album.setAccess(access);
314+
315+        // Static method access is static way
316+        // Calling AlbumDao class directly and not object instance
317+        AlbumDao.updateAccess(album);
318+
319+        // Always return OK
320+        return okJson();
321+    }
322+
323+
```

- Added function for `@POST /albums/:id/access` API request, to handle update requests for `access` attribute for some specific `Album` `id`.

```
277-        AlbumDao albumDao = new AlbumDao();
278-        PaginatedList<AlbumDto> paginatedList = PaginatedLists.create(limit, offset);
279-        SortCriteria sortCriteria = new SortCriteria(sortColumn, asc);
280-        AlbumCriteria albumCriteria = new AlbumCriteria()
281                .setUserId(principal.getId())
282                .setNameLike(search);
283-        albumDao.findByCriteria(paginatedList, albumCriteria, sortCriteria, null);
284
285-        JsonObjectBuilder response = Json.createObjectBuilder();
286-        JsonArrayBuilder items = Json.createArrayBuilder();
287-        for (AlbumDto album : paginatedList.getResultList()) {
288            items.add(Json.createObjectBuilder()
289-                    .add("id", album.getId())
290-                    .add("name", album.getName())
291-                    .add("update_date", album.getUpdateDate().getTime())
292-                    .add("albumart", album.getAlbumArt() != null)
293-                    .add("play_count", album.getUserPlayCount())
294                     .add("artist", Json.createObjectBuilder()
295-                            .add("id", album.getArtistId())
296-                            .add("name", album.getArtistName())));
297-        }
298
299-        response.add("total", paginatedList.getResultCount());
```

```
340+        JsonObjectBuilder response = Json.createObjectBuilder();
341+        JsonArrayBuilder items = Json.createArrayBuilder();
342+
343+        AlbumDao albumDao1 = new AlbumDao();
344+        PaginatedList<AlbumDto> paginatedList1 = PaginatedLists.create(limit, offset);
345+        SortCriteria sortCriteria1 = new SortCriteria(sortColumn, asc);
346+        AlbumCriteria albumCriteria1 = new AlbumCriteria()
347                .setUserId(principal.getId())
348+                .setPublic(false)
349+                .setNameLike(search);
350+
351+        albumDao1.findByCriteria(paginatedList1, albumCriteria1, sortCriteria1, null);
352+
353+        for (AlbumDto album1 : paginatedList1.getResultList()) {
354+            items.add(Json.createObjectBuilder()
355+                    .add("id", album1.getId())
356+                    .add("user_id", album1.getUserId())
357+                    .add("name", album1.getName())
358+                    .add("update_date", album1.getUpdateDate().getTime())
359+                    .add("albumart", album1.getAlbumArt() != null)
360+                    .add("play_count", album1.getUserPlayCount())
361+                    .add("artist", Json.createObjectBuilder()
362+                            .add("id", album1.getArtistId())
363+                            .add("name", album1.getArtistName())));
364+        }
365+
366+        AlbumDao albumDao2 = new AlbumDao();
367+        PaginatedList<AlbumDto> paginatedList2 = PaginatedLists.create(limit, offset);
368+        SortCriteria sortCriteria2 = new SortCriteria(sortColumn, asc);
369+        AlbumCriteria albumCriteria2 = new AlbumCriteria()
370+                .setPublic(true)
371                .setNameLike(search);
372+        albumDao2.findByCriteria(paginatedList2, albumCriteria2, sortCriteria2, null);
373
374+        for (AlbumDto album2 : paginatedList2.getResultList()) {
375             items.add(Json.createObjectBuilder()
376+                    .add("id", album2.getId())
377+                    .add("user_id", album2.getUserId())
378+                    .add("name", album2.getName())
379+                    .add("update_date", album2.getUpdateDate().getTime())
380+                    .add("albumart", album2.getAlbumArt() != null)
381+                    .add("play_count", album2.getUserPlayCount())
382                     .add("artist", Json.createObjectBuilder()
383+                            .add("id", album2.getArtistId())
384+                            .add("name", album2.getArtistName())));
385+        }
386
387+        response.add("total",paginatedList1.getResultCount() + paginatedList2.getResultCount());
```

- Change in the function for `@GET /album` requests to get albums lists, to check different criterias based on `access` and `user_id` to fetch different `Albums` lists fro mthe database.

2) Frontend

Album.html

```
<label ng-show="isOwner">
  <input type="checkbox" ng-model="isPublic" ng-change="changeAccess()" /> Public
</label>

<scope ng-show="!isOwner">
  <b><i>
    [Public Playlist]
  </i></b>
</scope>
```

- Added a checkbox to change access of the current `Album` based on the current user's `user_id`
- Public Playlist test is shown instead of the button when the owner of the public Album is not the current user.

Album.js

```
angular.module('music').controller('Album', function($scope, $state, $stateParams, Restangular, Playli
  // Load album
  Restangular.one('album', $stateParams.id).get().then(function(data) {
    $scope.album = data;
  });
```

```
angular.module('music').controller('Album', function($scope, $state, $statePara
  // Load album
  Restangular.one('album', $stateParams.id).get().then(function(data) {
    $scope.album = data;

    $scope.isPublic = (data.access == "PUBLIC");
    console.log("Album isPublic: " + $scope.isPublic);

    $scope.isOwner = data.isOwner;
    console.log("Album isOwner: " + $scope.isOwner);

    $scope.userId = data.userId;
    console.log("Album userId: " + $scope.userId);
  });


  $scope.changeAccess = function() {
    Restangular.one('album', $stateParams.id).post('access', { access: ($scope.
      console.log("Album isPublic: " + $scope.isPublic);
    });
  };
```

- Added state variables `isPublic` and `isOwner` in the frontend to enable edit buttion of `access` attribute of the Playlist based on criterias.
- `isPublic` determines the state of the checkbox
- `isOwner` determines the visibility of the checkbox to the current user.
- Added `changeAccess` function for sending `POST` request to update `access` variable when checkbox interacted with.

Design Patterns

- **DAO** Patterns:
  The Design Pattern is retained from the original codebase, as it already decouple a lot of logic for accessibility from definition.

```
AlbumDao albumDao1 = new AlbumDao();
PaginatedList<AlbumDto> paginatedList1 = PaginatedLists.create(limit, offset);
SortCriteria sortCriteria1 = new SortCriteria(sortColumn, asc);
AlbumCriteria albumCriteria1 = new AlbumCriteria()
        .setUserId(principal.getId())
        .setPublic(flag:false)
        .setNameLike(search);

albumDao1.findByCriteria(paginatedList1, albumCriteria1, sortCriteria1, filterCriteria:null);

for (AlbumDto album1 : paginatedList1.getResultList()) {
    items.add(Json.createObjectBuilder()
            .add(name:"id", album1.getId())
            .add(name:"user_id", album1.getUserId())
            .add(name:"name", album1.getName())
            .add(name:"update_date", album1.getUpdateDate().getTime())
            .add(name:"albumart", album1.getAlbumArt() ≠ null)
            .add(name:"play_count", album1.getUserPlayCount())
            .add(name:"artist", Json.createObjectBuilder()
                    .add(name:"id", album1.getArtistId())
                    .add(name:"name", album1.getArtistName())));
```

- **Builder** Pattern:
  The Builder Pattern is used for the step-by-step creation of objects like `Album` , `Playlist`, `Directory` etc.

```java
album = new Album();

// Getting the context of logged in user
album.setUserId(AppContext.getInstance().getUserId());

album.setArtistId(albumArtist.getId());
album.setDirectoryId(rootDirectory.getId());
album.setName(albumName);
album.setLocation(file.getParent().toString());

if (albumArtFile ≠ null) {
    // TODO Remove this, albumarts are scanned separately
    AppContext.getInstance().getAlbumArtService().importAlbumArt(album, albumArtFile, copyOriginal: false);
}

Date updateDate = getDirectoryUpdateDate(parentPath);
album.setCreateDate(updateDate);
album.setUpdateDate(updateDate);

albumDao.create(album);
```

- **Chain of Responsibility** Pattern:
  This pattern is mostly used by us when creation API request functions when interacting with different subsystems step-by-step and then throwing an error at any point of failure and abandoning the request.

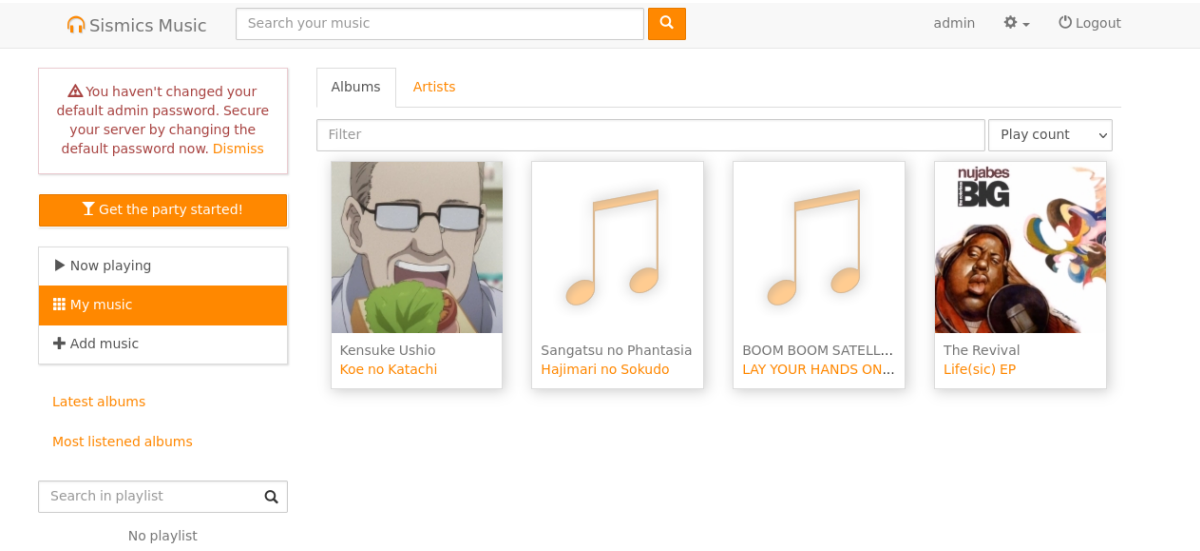
```java
    @PUT
    @Path("{id: [a-z0-9\\-]+}/albumart")
    @Consumes("multipart/form-data")
    public Response updateAlbumart(
            @PathParam("id") String id,
            @FormDataParam("file") FormDataBodyPart fileBodyPart) {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }

        // Get the album
        Album album = Album.getActiveById(id);
        notFoundIfNull(album, message:"id");

        // Validate input data
        Validation.required(fileBodyPart, name:"file");
        Validation.required(fileBodyPart.getFormDataContentDisposition().getFileName(), name:"filename");

        JsonObjectBuilder response = Json.createObjectBuilder().add(name:"status", value:"ok");
        File importFile = null;
        try {
            importFile = FormDataUtil.getAsTempFile(fileBodyPart);

            // Update the album art
            final AlbumArtService albumArtService = AppContext.getInstance().getAlbumArtService();
            String oldAlbumArtId = album.getAlbumArt();
            try {
                albumArtService.importAlbumArt(album, importFile, copyOriginal:true);
            } catch (NonWritableException e) {
                // The album art could't be copied to the album folder
                response.add(name:"message", value:"AlbumArtNotCopied");
            } catch (Exception e) {
                log.error(msg:"The provided URL is not an image", e);
                throw new ClientException(type:"ImageError", message:"The provided URL is not an image");
            }
            AlbumDao.update(album);

            // Delete the previous album art
            if (oldAlbumArtId ≠ null) {
                albumArtService.deleteAlbumArt(oldAlbumArtId);
            }
        } catch (Exception e) {
            throw new ServerException(type:"ImportError", e.getMessage(), e);
        } finally {
            if (importFile ≠ null) {
                importFile.delete();
            }
        }

        return renderJson(response);
```

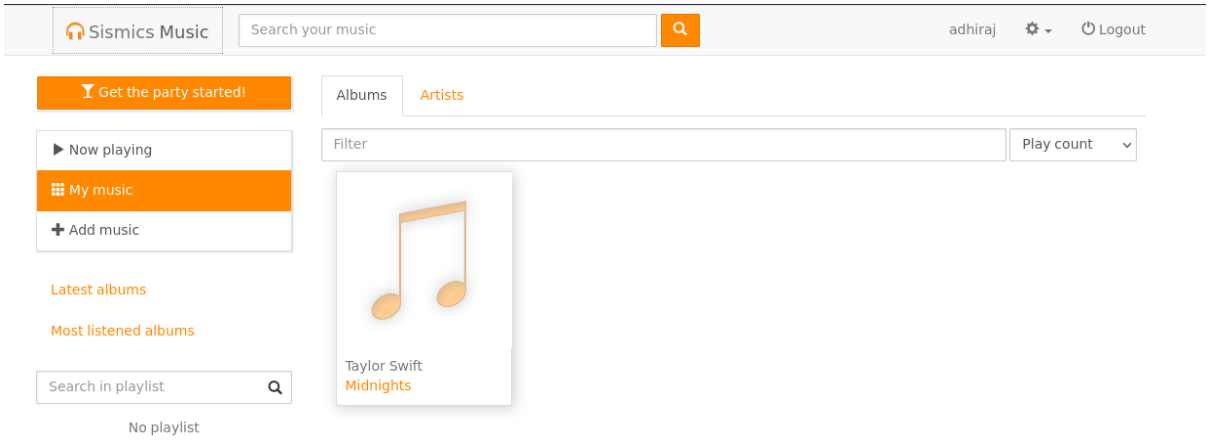
- **Adapter** Pattern:
  Adapter Pattern has been used by us so that `access` attribute which has a User-Defined Data Type, can be interacted with Javascript native Data Types without changing the backend storage type.

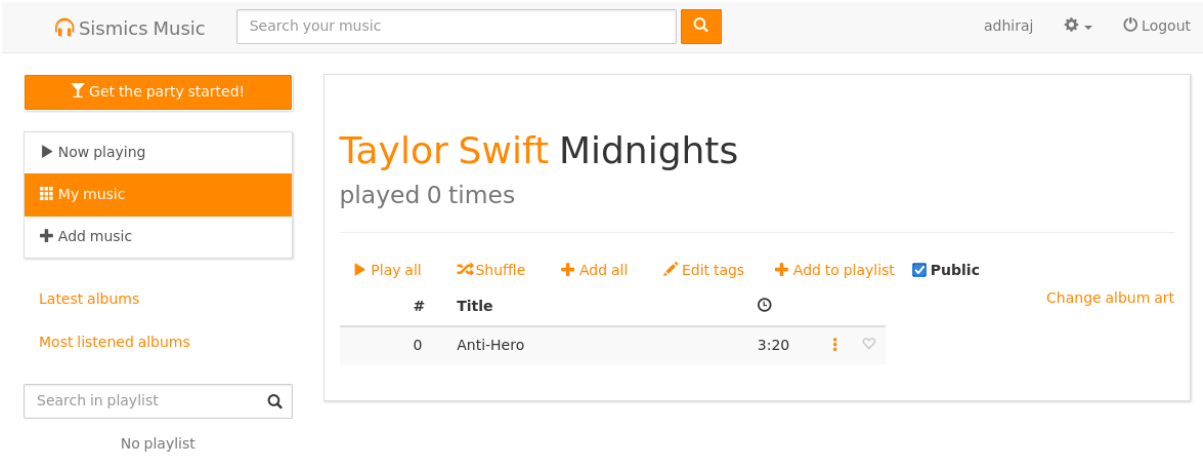
```java
    public AlbumCriteria setPublic(boolean flag) {
        this.access = flag ? AccessType.PUBLIC : AccessType.PRIVATE;
        return this;
    }
```

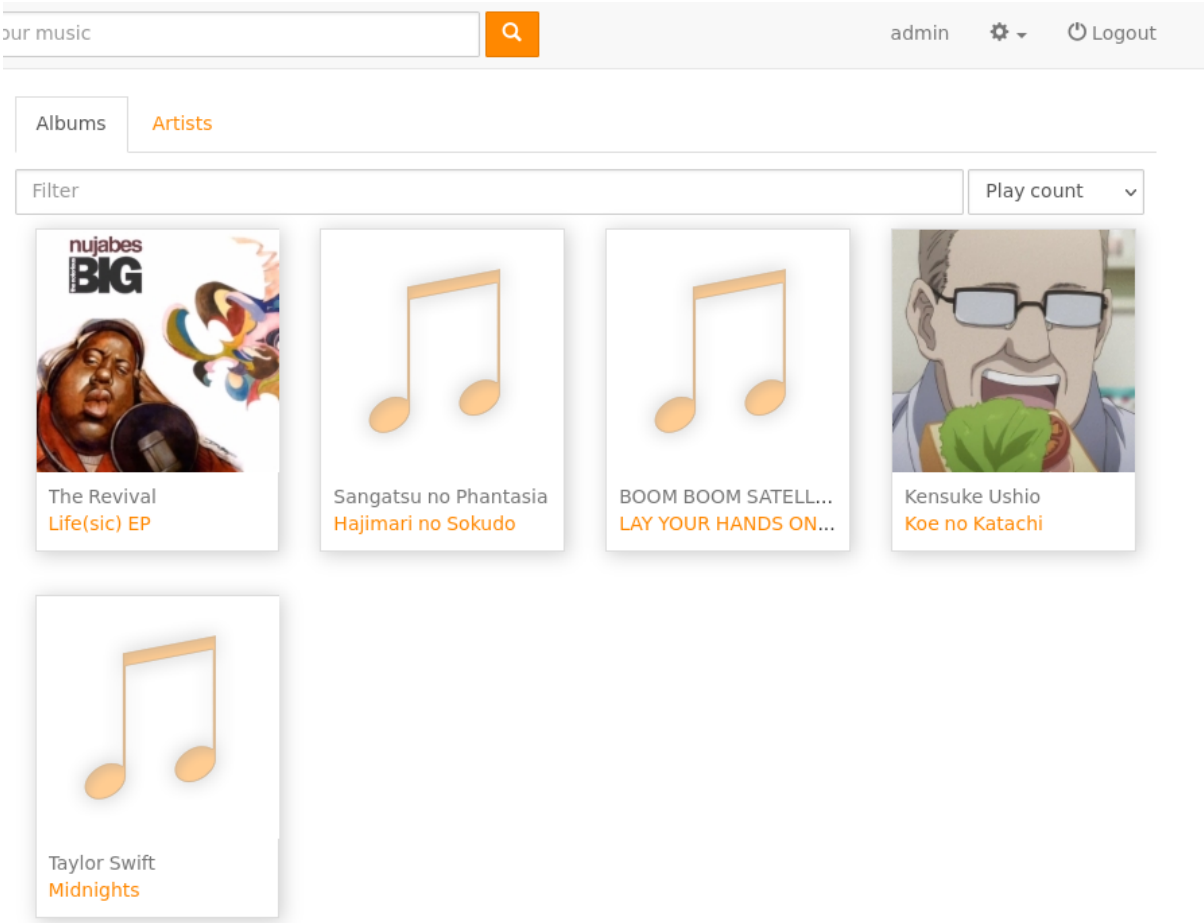
```java
public void setAccess(String access) {
    switch(access) {
        case "PUBLIC":
            this.access = AccessType.PUBLIC;
            break;
        case "PRIVATE":
            this.access = AccessType.PRIVATE;
            break;
        default:
            throw new IllegalArgumentException("Invalid access: " + access);
    }
}
```

Screenshots









Feature - 3: Online Integration

Problem

Music already support some degree of online integration. However, it is limited to only one service - LastFM. We want to extend what you can do with LastFM, and also incorporate integration with Spotify.

In particular, Music should support the following features -

1. Search for songs: LastFM and Spotify both allow you to search for songs. Use their APIs to allow users to search for songs in Music. Note that it should be possible for the user to choose which service to use.

2. Recommend songs based on playlist: Music allows users to create playlists. Spotify and Lastfm allow users to get recommendations similar to provided songs. Putting two and two together, we want to allow Music users to get recommendations from these services based on existing playlists. Concretely, given a list of songs, we want recommended songs from Spotify or LastFm (based on users choice). A simple string representation of these recommendations is enough.

Changes

1) Backend

i) LastFM

Both searching and recommendations can be performed via the in-built `search` and `recommend` functions of the lastfm package for a particular track. Therefore wrappers for these functions, taking the query and limit (for search) and the track name, artist name and limit (for recommendations) were added to `LastFmService.java`.

```
public Collection<de.umass.lastfm.Track> searchTrack(String query, int limit) {
        System.out.println("Searching for track: " + query);
        return de.umass.lastfm.Track.search(null,query,limit, ConfigUtil.getConfigStringValue(ConfigType.LAST_FM_API_KEY));
    }

    public Collection<de.umass.lastfm.Track> recommend(String artist, String name, int limit) {
        return de.umass.lastfm.Track.getSimilar(artist,name,ConfigUtil.getConfigStringValue(ConfigType.LAST_FM_API_KEY), limit);
    }
```

These wrapper functions are then called by the function corresponding to the routes `externalsearch/lastfmsearch` (for search - in `ExternalSearchResource.java`) and `playlist<playlist_id>/lastfmrecommendations` (for recommendations - in `PlaylistResource.java` . The returned results are the arrays of tracks, which are then displayed in the frontend.

```
@Path("/externalsearch")
public class ExternalSearchResource {
    private static final Logger log = LoggerFactory.getLogger(ExternalSearchResource.class);

    @GET
    @Path("/lastfmsearch")
    public JsonObject lastfmsearch(@QueryParam("query") String query) {
        // Parse the query and extract song name
        // String[] queryParts = query.split(":");
        // String songName = queryParts[1];
        // songName = songName.substring(1, songName.length() - 2);
        // System.out.println(songName);
        String songName = query;
        final LastFmService lastFmService = new LastFmService();
        Collection<de.umass.lastfm.Track> tracks = lastFmService.searchTrack(songName, 10);
        JsonArrayBuilder arrayBuilder = Json.createArrayBuilder();
        for (de.umass.lastfm.Track track : tracks) {
            JsonObjectBuilder objectBuilder = Json.createObjectBuilder();
            objectBuilder.add("name", track.getName());
            objectBuilder.add("artist", track.getArtist());
            arrayBuilder.add(objectBuilder);
        }
        return Json.createObjectBuilder().add("tracks", arrayBuilder).build();
    }
}
```

```
@GET
    @Path("{id: [a-z0-9\\-]+}/lastfmrecommendation")
    public JsonObject recommendationLastfm(@PathParam("id") String playlistId) {

        JsonObject playlist = getPlaylistTracks(playlistId);
        final LastFmService lastFmService = AppContext.getInstance().getLastFmService();
        // iterate over all the tracks in the playlist,ie, playlist.tracks using a for loop
        // for each track, get the artist and album
        JsonArray tracks = playlist.getJsonArray("tracks");
        JsonArrayBuilder rectrackArray = Json.createArrayBuilder();
        for (int i=0;i<tracks.size();i++) {
            JsonObject track = tracks.getJsonObject(i);
```

```
            String artist = track.getJsonObject("artist").getString("name");
            String trackname = track.getString("title");
            Collection<de.umass.lastfm.Track> recom=lastFmService.recommend(artist,trackname,1);
            for (de.umass.lastfm.Track t : recom) {
                rectrackArray.add(Json.createObjectBuilder()
                        .add("name", t.getName())
                        .add("artist", t.getArtist()));
            }
        }
        JsonObject response = Json.createObjectBuilder()
                .add("status", "ok")
                .add("tracks", rectrackArray)
                .build();
        System.out.println("response: " + response);
        return response;
    }
```

ii) Spotify

We have used the `spotify-web-api-java` artifact for this purpose. This is a Java wrapper/client for the Spotify Web API.

```
<dependency>
  <groupId>se.michaelthelin.spotify</groupId>
  <artifactId>spotify-web-api-java</artifactId>
  <version>6.5.4</version>
</dependency>
```

Changes in the database:

1. 4 columns added to T_USER:
    a. SPOTIFYACCESSTOKEN
    b. SPOTIFYREFRESHTOKEN
    c. SPOTIFYREFRESHESIN
    d. SPOTIFYAUTHCODE

    These additions have been made as to access Spotify data through APIs, we need the user's access token, which is in turn generated from an authentication code. Furthermore, this token expires in every fixed interval, and hence a refresh token is needed to provide a new access token every time.

2. 3 rows added to T_CONFIG:
    a. SPOTIFY_API_KEY
    b. SPOTIFY_API_SECRET,
    c. SPOTIFY_REDIRECT_URI

    These rows represent the API key, the secret key, and the redirection URL for the Spotify API.

```diff
7 music-core/src/main/resources/db/update/dbupdate-000-0.sql
@@ -221,6 +221,10 @@ create memory table T_USER (
221 221    EMAIL                varchar(100)        not null,
222 222    MAXBITRATE           integer             null default '0',
223 223    LASTFMSESSIONTOKEN   varchar(100)        null default '0',
    224 +  SPOTIFYACCESSTOKEN   varchar(100)        null default '0',
    225 +  SPOTIFYREFRESHTOKEN  varchar(100)        null default '0',
    226 +  SPOTIFYREFRESHTIME   datetime             null,
    227 +  SPOTIFYAUTHCODE      varchar(100)        null default '0',
224 228    LASTFMACTIVE         bit                 not null default '0',
225 229    FIRSTCONNECTION      bit                 not null default '0',
226 230    CREATEDATE           datetime            not null,
@@ -397,6 +401,9 @@ on delete restrict on update restrict;
397 401
398 402  insert into t_config(id,value) values('LAST_FM_API_KEY','7119a7b5c4455bbe8196934e22358a27');
399 403  insert into t_config(id,value) values('LAST_FM_API_SECRET','30dce5dfdb01b87af6038dd36f696f8a');
    404 + insert into t_config(id,value) values('SPOTIFY_API_KEY', '5d222579ce9241b6aa234b6609071a8f');
    405 + insert into t_config(id,value) values('SPOTIFY_API_SECRET', 'be4ca711e6f244df93e47a95e0ddda06');
    406 + insert into t_config(id,value) values('SPOTIFY_API_REDIRECT_URI', 'http://localhost:8080/');
400 407  insert into t_config(id,value) values('DB_VERSION', '0');
401 408  insert into t_config(id,value) values('LUCENE_DIRECTORY_STORAGE', 'FILE');
402 409  insert into t_privilege(id) values('ADMIN');
```

Changes in music-core java code

- `UserMapper`, `User`, `UserCriteria`, and `Configtype` were accordingly updated.
- The following method was added to `UserDao`:

```
/**
     * Update the user Spotify session tokens.
     *
     * @param user User to update
     * @return Updated user
     */
    public User updateSpotifyTokens(User user) {
        final Handle handle = ThreadLocalContext.get().getHandle();
        handle.createStatement("update t_user u set " +
                " u.spotifyaccesstoken = :spotifyAccessToken, " +
                " u.spotifyrefreshtoken = :spotifyRefreshToken " +
                " u.spotifyrefreshtime = :spotifyRefreshTime " +
                " u.spotifyauthcode = :spotifyAuthCode " +
                " where u.id = :id and u.deletedate is null")
                .bind("id", user.getId())
                .bind("spotifyAccessToken", user.getSpotifyAccessToken())
                .bind("spotifyRefreshToken", user.getSpotifyRefreshToken())
                .bind("spotifyRefreshTime", user.getSpotifyRefreshTime())
                .bind("spotifyAuthCode", user.getSpotifyAuthCode())
                .execute();

        return user;
    }
```

- `SpotifyService` was created analogous to the `LastFmService` which contains all the functionality/services that the music app accesses from Spotify. `AppContext` was accordingly updated to provide and instantiate the Spotify service.
- The following methods are provided by `SpotifyService`:
  - `scheduler()` - Returns a Scheduler object that specifies how often the service should run. In the current implementation, the service is scheduled to run every 3500 seconds (58 minutes).
  - `startUp()` - Loads Client ID, Client Secret key, and the Redirection URI from the database.
  - `runOneIteration()` - Called by the scheduler and contains the logic for what the service should do at each iteration. In the current implementation, this

method calls the `authorizationCodeRefresh_Sync()` for every connected user.

  - `authorizationCodeRefresh_Sync()` - Using the current refresh token, new access, and refresh tokens are retrieved.
  - `authorizationCode_Sync()` - Using the authentication code, an access and refresh token is retrieved.
  - `authorizationCodeUri_Sync()` - An authentication URI is received which contains the “code” as a query parameter.
  - `searchTracks_Sync()` - Returns a paging object (collection) of tracks matching the search query.
  - `searchAlbums_Sync()` - Returns a paging object (collection) of albums matching the search query.
  - `searchArtists_Sync()` - Returns a paging object (collection) of artists matching the search query.

- Some of the `SpotifyService` methods are used by the `UserResource` to link and unlink a user’s Spotify account as follows:

```
    /**
     * Authenticates a user on Spotify.
     *
     * @param spotifyUsername Spotify username
     * @param spotifyPassword Spotify password
     * @return Response
     */
    @PUT
    @Path("spotify")
    public Response registerSpotify(){
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }

        // Get the value of the session token
        final SpotifyService spotifyService = AppContext.getInstance().getSpotifyService();

        UserDao userDao = new UserDao();
        User user = userDao.getActiveById(principal.getId());

        spotifyService.authorizationCodeUri_Sync(user);
        spotifyService.authorizationCode_Sync(user);

        // Update tokens
        userDao.updateSpotifyTokens(user);

        // Always return ok
        JsonObject response = Json.createObjectBuilder()
                .add("status", "ok")
                .build();
        return Response.ok().entity(response).build();
    }

    /**
     * Disconnect the current user from Spotify.
     *
     * @return Response
     */
    @DELETE
    @Path("spotify")
    public Response unregisterSpotify() {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }

        // Remove the session token
        UserDao userDao = new UserDao();
        User user = userDao.getActiveById(principal.getId());
        user.setSpotifyAccessToken(null);
        user.setSpotifyRefreshToken(null);
        user.setSpotifyAuthCode(null);
        userDao.updateSpotifyTokens(user);

        // Always return ok
        JsonObject response = Json.createObjectBuilder()
                .add("status", "ok")
                .build();
        return Response.ok().entity(response).build();
    }
}
```

- Some of the `SpotifyService` methods are used by the `SearchResource` to do query-based searching for tracks, albums, and artists from Spotify analogous to that from the music app:

```
    /**
     * Run a full text search from Spotify.
     *
     * @param query Search query
     */
    @GET
    @Path("{spotify/query: .+}")
    public Response spotifySearch(@PathParam("query") String query) {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }
        final SpotifyService spotifyService = AppContext.getInstance().getSpotifyService();
        User user = new UserDao().getActiveById(principal.getId());
        Paging<com.wrapper.spotify.model_objects.specification.Track> t = spotifyService.searchTracks_Sync(user, query);
        Paging<AlbumSimplified> a = spotifyService.searchAlbums_Sync(user, query);

        JsonObjectBuilder response = Json.createObjectBuilder();
        // tracks
        JsonArrayBuilder tracks = Json.createArrayBuilder();
        for (com.wrapper.spotify.model_objects.specification.Track track : t.getItems()) {
            tracks.add(Json.createObjectBuilder()
                    .add("id", track.getId())
                    .add("title", track.getName())
                    .add("length", track.getDurationMs())
                    .add("album", Json.createObjectBuilder()
                            .add("id", track.getAlbum().getId())
                            .add("name", track.getAlbum().getName())
                            .add("albumart", track.getAlbum().getImages()[0].getUrl()))
                    .add("artist", Json.createObjectBuilder()
                            .add("id", track.getArtists()[0].getId())
                            .add("name", track.getArtists()[0].getName())));
        }
        response.add("tracks", tracks);

        // albums
        JsonArrayBuilder albums = Json.createArrayBuilder();
        for (AlbumSimplified album : a.getItems()) {
            albums.add(Json.createObjectBuilder()
                    .add("id", album.getId())
                    .add("name", album.getName())
                    .add("albumart", album.getImages()[0].getUrl())
                    .add("artist", Json.createObjectBuilder()
                            .add("id", album.getArtists()[0].getId())
                            .add("name", album.getArtists()[0].getName())));
        }

        response.add("albums", albums);

        // artists
        JsonArrayBuilder artists = Json.createArrayBuilder();
        for (ArtistSimplified artist : a.getItems()[0].getArtists()) {
            artists.add(Json.createObjectBuilder()
                    .add("id", artist.getId())
                    .add("name", artist.getName()));
        }
        response.add("artists", artists);
        return renderJson(response);
    }
}
```

1) Frontend

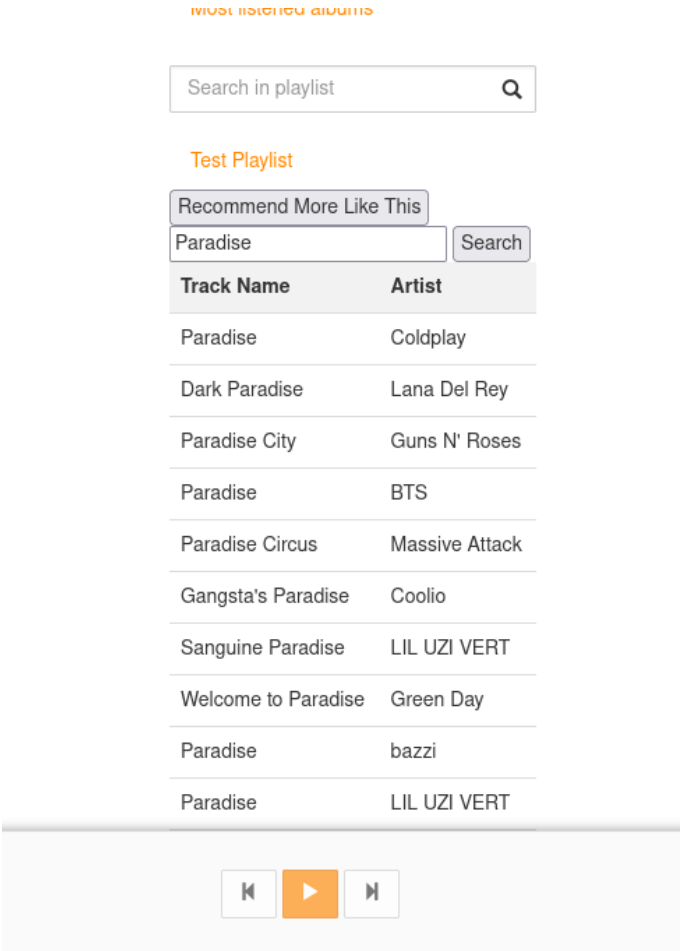
In the frontend, we added a search bar to search specifically for tracks through lastfm. The user enters the name of the track in the bar (which translates to the query in the backend), and the results are displayed in the form of a table below the search bar.

Since recommendations correspond to a particular playlist, each playlist has a `Recommend` button placed below it, that suggests recommendations based on the tracks in the playlist.

```
$scope.lastFmSearch = function(query) {
    Restangular.one("externalsearch/lastfmsearch").get(query).then(function(response) {
      $scope.myData = response;
    });
  };

  $scope.lastFmRecommendations = function(playlistId) {
    console.log(playlistId);
    Restangular.one("playlist/" + playlistId + "/lastfmrecommendation").get().then(function(response) {
      $scope.myData = response;
    });
  };
```

The controllers corresponding to `main.html` are present in `Main.js`. We have added the above two functions that use Restangular to call the backend routes from the frontend. The above two are for lastfm.

We have written similar functions for Spotify.



DESIGN PATTERNS

1. **Builder**

   The Spotify web API wrapper uses the Builder pattern to create complex objects step by step.

```
SpotifyApi spotifyApi = new SpotifyApi.Builder()
        .setClientId(clientId)
        .setClientSecret(clientSecret)
        .setRedirectUri(redirectUri)
        .setAccessToken(user.getSpotifyAccessToken())
```

```
        .setRefreshToken(user.getSpotifyRefreshToken())
        .build();
```

In the given code, the `SpotifyApi` class has a lot of attributes. Instead of having a constructor that takes all of these attributes as arguments, the class uses a Builder pattern to set these attributes before building an object of the SpotifyApi class. This way, you can create an object of the SpotifyApi class with specific attributes without having to pass all of them, overall, helping to simplify the creation of complex objects and providing a flexible way to set the attributes of an object before creating it.

2. **Observer**

   The `AbstractScheduledService` class in the Guava library uses the Observer pattern to notify its subscribers when a scheduled task is executed.

   In the given code, the AbstractScheduledService class from the Guava library acts as the subject, designed to execute scheduled tasks, and it maintains a list of subscribers who are interested in being notified when a task is executed through the `notifySuccessful()` method. Subscribers can then take appropriate action based on the event, such as logging the event or updating their internal state. Through this pattern, we can decouple the subject and observer objects.

3. **Builder**

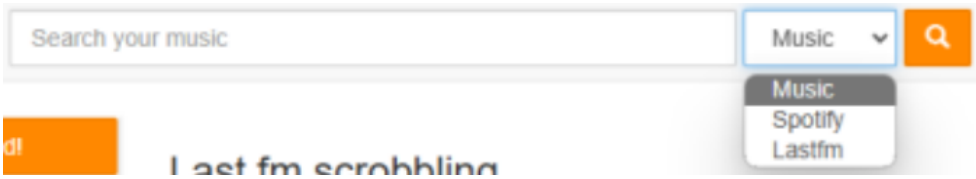
   The `javax.json.JsonObjectBuilder` is used to construct JSON objects using the builder pattern.

```
JsonObjectBuilder response = Json.createObjectBuilder();
        // tracks
        JsonArrayBuilder tracks = Json.createArrayBuilder();
        for (com.wrapper.spotify.model_objects.specification.Track track : t.getItems()) {
            tracks.add(Json.createObjectBuilder()
                    .add("id", track.getId())
                    .add("title", track.getName())
                    .add("length", track.getDurationMs())
                    .add("album", Json.createObjectBuilder()
                            .add("id", track.getAlbum().getId())
                            .add("name", track.getAlbum().getName())
                            .add("albumart", track.getAlbum().getImages()[0].getUrl()))
                    .add("artist", Json.createObjectBuilder()
                            .add("id", track.getArtists()[0].getId())
                            .add("name", track.getArtists()[0].getName())));
        }
        response.add("tracks", tracks);
```

   The add method is used to add properties to the object being built separately and successively by returning the same JsonObjectBuilder instance. Overall, this helps to simplify the creation of complex objects and provides a flexible way to set the attributes of an object before creating it. This makes the code easier to comprehend and maintain.

4. **Strategy**

   While the pattern is not strictly implemented, its essence is reflected in the Search functionality of the 3 music services, viz., LastFm, and Spotify.



   The frontend expects a JSON response in the standard format of track, album, and artist dictionaries. All the 3 services return that using different services(like `LastFmService` and `SpotifyService`) that can be seen to be the concrete strategies implementing `AbstractScheduledService` abstract strategy.

Individual Contributions

- Adhiraj Anil Deshmukh: 1, 2

- Arjun Muraleedharan: 2, 3
- Pratham Gupta: 1, 3
- Keyur Ganesh Chaudhari: 3
- Sambasai Reddy Andem: 3