Individual Reflection (Individual Submission) _Hninn Ei Khaing Group4

Roles and Responsibilities

The overarching goal of my role in the AD project is to develop Java backend that serves multiple functionalities centred around music recommendation that based on user's location and Time Type (Night, Morning , Afternoon, Evening).

1. Default Playlist Generation:

I've created a "Default Playlist" table in our database that holds 200 song URIs. These songs are either popular or have been paid to be included. When a user logs in, our system generates a default playlist for them. Regular users get a playlist with 6 randomly picked songs from this table, while premium users get 12. This approach ensures that users frequently replay tracks, indicating the playlist's appeal.

SongController to get Default playlist

```
 @GetMapping("/default-song")
      public ResponseEntity<List<String>> generateDefaultSong(@RequestParam("userId") int userId) {
         List<String> playlist = new ArrayList<>();
          if (userService.isUserPremium(userId)) {
            playlist = defaultSongService.findDefaultSongs(12);
              } else {
             playlist = defaultSongService.findDefaultSongs(6);
              }
         return ResponseEntity.ok(playlist);
   }
```

UserService to know is User premium or not

```
public boolean isUserPremium(int userId){
      return userRepository.isUserPremium(userId);
   }
```

UserRepository to know is User premium or not

```
@Query("SELECT u.premium FROM User u WHERE u.id = :userId")
boolean isUserPremium(@Param("userId") int userId);
```

DefaultSongService to generate playlist

```
 public List<String> findDefaultSongs(int limit){
      return defaultSongRepository.findRandomSongs(limit);
   }
```

DefaultSongRepository to generate playlist

```
@Query(value = "SELECT df.uri FROM defaultsong df ORDER BY RAND() LIMIT :limit", nativeQuery = true)
List<String> findRandomSongs(@Param("limit") int limit);
```

2. Generate Song recommendation based on location and time type:

First of all, I create an ReceivedLocation object that include longitude, latitude and user so that I can expect a JSON payload (@RequestBody ReceivedLocation location) containing location details.

```
class ReceivedLocation{
   private double latitude;
   private double longitude;
   private String username;
   public double getLatitude() {
      return latitude;
   }
```

```
    public void setLatitude(double latitude) {
        this.latitude = latitude;
    }
    public double getLongitude() {
        return longitude;
    }
    public void setLongitude(double longitude) {
        this.longitude = longitude;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
}
```

- **Song Recommendation based on Location:**

Develop an algorithm that leverages user location data to generate a curated list of recommended songs. I utilize method to capture the user's current longitude and latitude. After acquiring this geographical data, I cross-reference it with the user's song history to determine if the current location matches any previously recorded locations. If a match is found, the system is programmed to generate a set of song recommendations that align with the user's musical preferences at that specific location.

Integrate this feature with existing location APIs to provide real-time song suggestions that are geographically relevant.

Geocoding utility to convert longitude and latitude into readable address.

```
public class GeocodingUtility {
    private static final String GOOGLE_MAPS_API_KEY = "AIzaSyDiiIF6bHn6O5JnAk7ZZgQX0lOOLLsmIXY";

    public static String getAddressFromCoordinates(double latitude, double longitude) {
        try {
            GeoApiContext context = new GeoApiContext.Builder()
                    .apiKey(GOOGLE_MAPS_API_KEY)
                    .build();
            GeocodingResult[] results = GeocodingApi.newRequest(context)
                    .latlng(new com.google.maps.model.LatLng(latitude, longitude))
                    .await();

            if (results != null && results.length > 0) {
                return results[0].formattedAddress;
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

User Location Management: Upon receiving the user's geolocation coordinates from the frontend, I leverage Google's API to convert these coordinates into a precise address. Then, I query our database to find the corresponding location_id associated with this address. With the obtained location_id, I search our SongHistory table to find any existing records that match this location. If such records are found, I then proceed to generate a personalized song playlist for the user. To achieve this, I utilize a Python-based recommendation service. I send the following parameters to this service: userId, location_id, timeType, and number. In response, the service returns a list of song URIs that form the user's recommended playlist. This playlist is then made available to the user, thus providing a location-based, personalized listening experience.

```
@PostMapping("/by-location")
  public ResponseEntity<List<String>> generateSongByLocation(@RequestBody ReceivedLocation location) {
    try {
      int locationId, timeType = -1, number;

      double latitude = location.getLatitude();
      double longitude = location.getLongitude();
      String address = GeocodingUtility.getAddressFromCoordinates(latitude, longitude);
      locationId = locationService.findLocationIdByAddress(address);
      User user = userService.findUserByUsername(location.getUsername());
      int userId = user.getId();
      if (userService.isUserPremium(userId)) {
        number = 24;
      } else {
        number = 12;
      }
      List<String> playlist = pythonService.senddatatoPython(userId, locationId, timeType, number);

      return ResponseEntity.ok(playlist);
    } catch (Exception e) {
      // log the exception and return an error response
      return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(Collections.emptyList());
    }
  }
```

LocationService to get locationId

```
public int findLocationIdByAddress(String address){
    return locationRepository.findLocationIdByAddress(address);
  }
```

UserService to get userId

```
public User findUserByUsername(String username)
  {
    return userRepository.findByUsername(username);
  }
```

- **Song Recommendation based on Time Type:**

Design and implement a system that recommends songs based on the specific time of the day. I first identify the current time type, which could be categories like morning, afternoon, evening, or night. I then cross-reference this time type with the user's historical listening data to filter for songs that the user typically listens to during that specific time frame. This approach allows the system to recommend songs that are not only similar in genre or artist but are also aligned with the user's listening habits at that particular time of day. To keep the recommendations fresh and relevant, the algorithm continuously updates based on new user listening data.

```
  @GetMapping("/by-time")
    public ResponseEntity<List<String>> generateSongByTime(@RequestBody ReceivedLocation location) {
    try {
      int locationId=-1, timeType = -1, number;
      LocalTime currentTime = LocalTime.now();
      User user = userService.findUserByUsername(location.getUsername());
      int userId = user.getId();
       if (currentTime.isAfter(LocalTime.of(5, 59)) && currentTime.isBefore(LocalTime.of(12, 0))) {
          timeType = 0;
      } else if (currentTime.isAfter(LocalTime.of(11, 59)) && currentTime.isBefore(LocalTime.of(17, 0))) {
            timeType = 1;
      } else if (currentTime.isAfter(LocalTime.of(16, 59)) && currentTime.isBefore(LocalTime.of(21, 0))) {
            timeType = 2;
      } else {
            timeType = 3;
      }
      if (userService.isUserPremium(userId)) {
         number = 24;
      } else {
         number = 12;
      }
      List<String> playlist = pythonService.senddatatoPython(userId, locationId, timeType, number);
      return ResponseEntity.ok(playlist);
    } catch (Exception e) {
      // log the exception and return an error response
      return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(Collections.emptyList());
    }
  }
```

PythonService to connect with flask

```
public List<String> senddatatoPython(int userId, int locationId, int timeType, int number) {
  String url = "http://localhost:5500/model1?userid=" + userId + "&locationid=" +
            locationId + "&timetype=" + timeType + "&number=" + number;
  ResponseEntity<List> response = restTemplate.getForEntity(url, List.class);

  // Check response status code and body if necessary
  if (response.getStatusCode().value() == 200) {
    return response.getBody();
  } else {
    System.out.println("Request failed with status code: " + response.getStatusCode().value());
    return null;
  }
}
```

Major Challenges & Lesson Learned

In the development of our location-based music recommendation service, we've encountered challenges in converting user location data from latitude and longitude to a specific address using Google's Geolocation API, as well as in sending this geolocation data from the HTML frontend to the Java backend, both of which we plan to resolve through targeted coding solutions.

Challenges

- **Using Google Geolocation API**: Solution - Utilize Google's Geolocation API to perform this conversion. Create a utility class, GeocodingUtility, that handles API requests and responses.
- **HTML to Java Backend Communication**: Solution - Implement an HTML/JavaScript frontend that captures the user's location and sends it to the backend via a RESTful API call.

Technical aspects

- **Coding Bug(s)/Problem(s) Encountered**

Geolocation API Integration: One of the significant challenges we encountered involved integrating the Geolocation API to obtain user locations. We faced difficulty in running the HTML code for geolocation collection and sending the data back to our Java backend in the same application.

Foreign Key Constraints: Another issue was a SQL error (Error Code: 1452) when trying to insert data into the SongHistory table. The problem lay in violating foreign key constraints, specifically relating to the song_id.

- **Codes with Issues**

Performance: The SQL query to update the URI column in the song table was inefficient, resulting in longer execution times.

Reliability: GeocodingUtility class for translating latitude and longitude to address did not have any fallback mechanism in case the Google API call failed.

Good Coding/CICD Habits: Initially, we had hardcoded the Google Maps API key into GeocodingUtility class, which is against best practices. It should be loaded from a secure environment variable.

Elegant Coding: Our first implementation of time-based song selection in generateSongByTime function used multiple if-elsenstatements. This could have been elegantly done using a time range mapping object, thereby making the code cleaner and more maintainable.

Development strategies, communication with the team

In our team's development strategy for this Songs recommendation service, we prioritize collaborative coding and frequent team meet-ups to ensure seamless communication and efficient problem-solving, especially as we work to overcome challenges related to geolocation data processing and API integration.

I was initially uncomfortable when a teammate, responsible for integrating frontend and backend, modified my code.

Resolution:

- Open Communication: We discussed the changes to clear misunderstandings.
- Code Review: I reviewed the changes, understanding they were made for system stability.
- Collaborative Debugging: We worked together to debug and fine-tune the application.

This experience reinforced the importance of team collaboration and open communication in problem-solving. It provided a learning opportunity that contributed to the project's collective success.