Software Design Specification
For AI-Powered Playlist Generator With Spotify Integration

Brianna Ly, Sofia Elenga, Thanh Lu, An Truong

1. System Description

The AI Playlist Generator with Spotify Integration is a cutting-edge software application designed to enhance the music discovery and playlist creation experience for users. This innovative system combines the power of artificial intelligence (AI), machine learning, and seamless integration with Spotify's vast music catalog to provide personalized and engaging playlists tailored to ethem to create, customize, and enjoy playlists tailored to their unique preferences. Key components of the system include user mach user's preferences. The system offers a wide range of functionalities to cater to music enthusiasts, allowing anagement, playlist generation, Spotify integration, collaborative playlist creation, and playback control.

1.1 Purpose

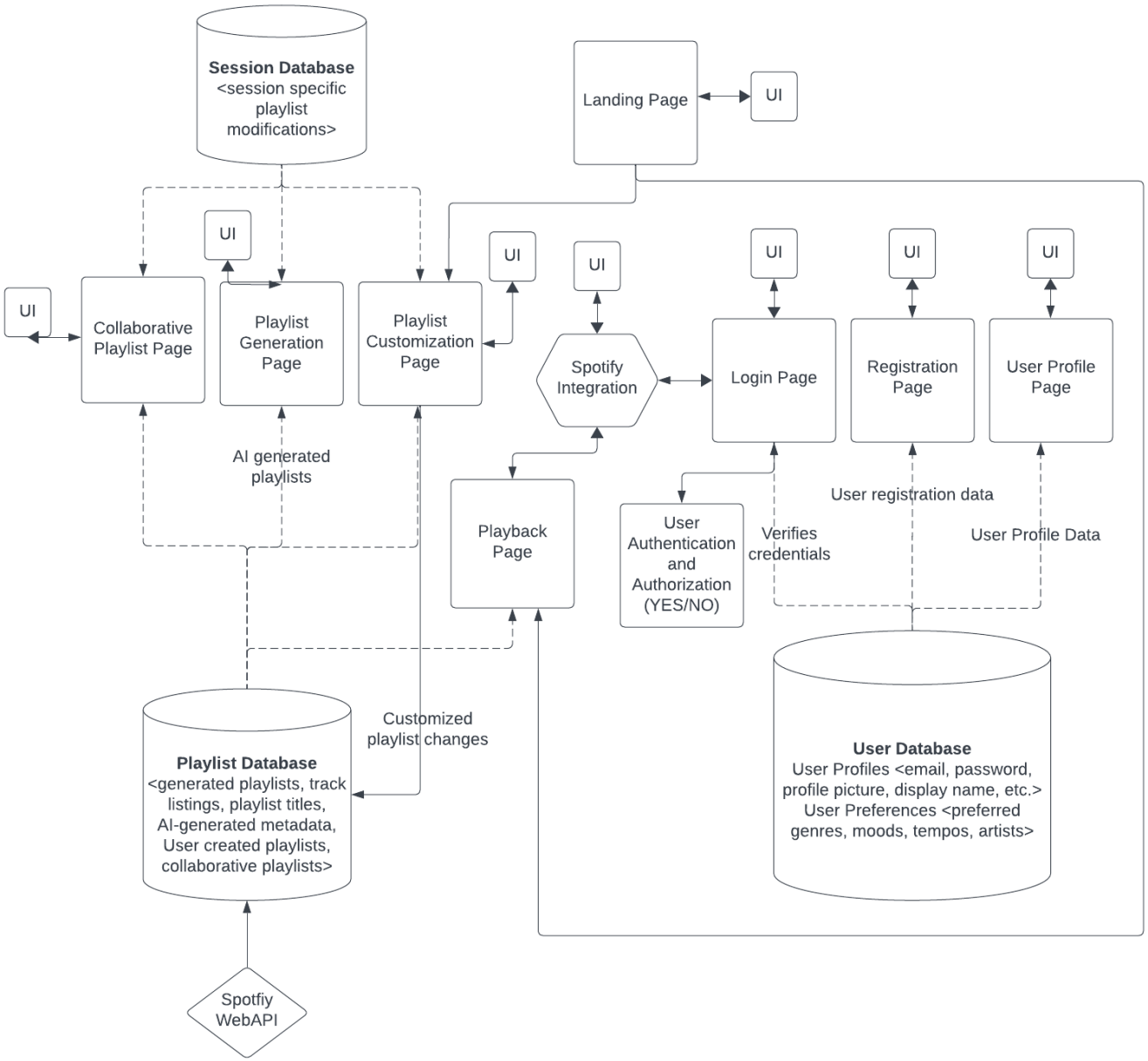
This document outlines the design of the AI-Powered Playlist Generator with Spotify Integration. It provides specific details about the system's architecture, expected input, output, classes, and functions. The interactions between various system components are detailed in figures at the end of the document.

1.2 Scope

This Design Specification serves as a guide for Software Engineering and Software Quality Engineering teams. It defines the system's design for implementing the AI-Powered Playlist Generator with Spotify Integration.

2. Software Architecture Overview

2.1 Architectural Software Diagram



2.1.1 Software Architecture

1. Landing Page:
User Interface (UI): The landing page serves as the entry point to the application. It provides a visually appealing and engaging interface that introduces the system's features and encourages users to explore further.
Database: The landing page itself does not directly interact with any database. It primarily serves as a presentation layer and contains static or dynamically loaded content.

2. Registration/Sign-Up Page:
User Interface (UI): The registration page allows new users to create accounts. It includes input fields for user information such as name, email, password, and profile picture.
Database: User registration data, including usernames, email addresses, and encrypted passwords, is stored in the system's database for authentication and user management.

3. Login Page:
User Interface (UI): The login page presents users with fields to enter their credentials, typically including their username or email, and password.
Database: The login page verifies user credentials by checking against the user database. If valid, the user is granted access to their personalized profile and playlists.

3.1 User Authentication and Authorization (YES OR NO):
Authentication: When a user chooses to connect their Spotify account, the system initiates an authentication process. The user is redirected to the Spotify login page, where they provide their Spotify credentials (username and password).
Authorization: After successful login, the user is asked to authorize the AI Playlist Generator to access their Spotify account data. This authorization is granted, when the system requests specific scopes (permissions) from Spotify, such as access to the user's playlists and listening history.

4. User Profile Page:
User Interface (UI): The user profile page displays user-specific information such as profile picture, display name, and user preferences. Users can also edit their profiles from this page.
Database: User profile data is retrieved from the database and dynamically displayed on the user profile page. Any changes made to the profile are stored back in the database.

5. Playlist Generation Page:
User Interface (UI): The playlist generation page allows users to initiate the playlist creation process. Users can specify criteria such as genre, mood, tempo, or artist preferences.

Database: The playlist generation page communicates with the backend to trigger AI algorithms. Generated playlists, including track listings and metadata, are temporarily stored in memory during user sessions.

6. Spotify Integration Page:
User Interface (UI): This page provides users with access to Spotify's music catalog. Users can search and select songs from Spotify's extensive library.
Database: The Spotify integration page interacts with the Spotify platform through API calls. It does not directly interact with the system's database but leverages Spotify's database and APIs.

7. Playlist Customization Page:
User Interface (UI): Users can further customize generated playlists by adding, removing, or reordering tracks. They can also apply predefined templates or modify playlist metadata.
Database: Customized playlists and playlist metadata changes are temporarily stored in memory during the user session. The original playlist data remains in the database.

8. Collaborative Playlist Page:
User Interface (UI): Users can collaborate with others on playlist creation. Multiple contributors can add and manage tracks within a playlist.
Database: Collaborative playlist data is managed through the system's database, where each playlist is associated with a list of contributors and track information.

9. Playback Page:
User Interface (UI): The playback page allows users to listen to their generated playlists directly within the application. Users can also choose to open playlists in the Spotify app for playback.
Database: The playback page retrieves playlist data, including track information, from the database or caches it for seamless playback

User Database (2, 3, 4):
Purpose: Stores user profile information, authentication data, and user preferences.
Contents:

- User profiles (including username, email, password, profile picture, display name, etc.)
- User preferences (such as preferred genres, moods, and customization settings)

Playlist Database (5, 7, 8, 9):
Purpose: Stores generated playlists, playlist metadata, and user-created playlists.
Contents:

- Generated playlists (with track listings, playlist titles, and AI-generated metadata)
- User-created playlists (customized by users)

- Collaborative playlists (with contributors and track information)

Session Database (5, 7, 8):
Purpose: Temporarily stores session-specific data, including user interactions and temporary playlist modifications during a user's session.
Contents:

- Session-specific playlist modifications (e.g., customizations made during a session)

Spotify Integration (3, 9):
Purpose: Maintains information related to integrating with Spotify's music catalog and playlists.
Contents:

- Spotify authentication tokens
- Imported Spotify playlists (if users choose to import them)
- Interaction logs with Spotify's APIs

Integration with Spotify APIs:
API Interaction: Once the user authorizes access, the AI Playlist Generator communicates with Spotify's APIs. These APIs provide endpoints for various functions, including:

- Accessing the Spotify music catalog to search for songs and retrieve track information.
- Retrieving the user's playlists, liked songs, and listening history.
- Adding tracks to the user's Spotify playlists.
- Managing user playlists, including creating new playlists or modifying existing ones.

2.2 UML Class Diagram

Continued on Next Page →

Class songs

string songType
string songName
string songArtist
int songDuration
int songBPM

string getSongType() return songType

string getSongName return songName

string getArtist() return songArtist

int getSongDuration return SongDuration

int getSongBPM return songBPM

Class cusPlayList

List<songs> cusPlayList

string getSongType(songs) return songs.songType

string getSongName(songs) return songs.songName

string getSongArtist(songs) return songs.songArtist

int getSongDuration(songs) return songs.songDuration

int getSongBPM(songs) return songs.songBPM

Class PlaylistManager

None

bool addSongToPlaylist(playlistID: int, song: Song)
return true (if success)

bool removeSongFromPlaylist(playlistID: int, songID: int)
return true (if success)

Class TemplateManager

None

Playlist applyTemplate(Playlist playlist , PlaylistTemplate template)
return Playlist

Class CollaborativePlaylist

None

bool addContributor(int playlistID, int userID)
returns true (if successful)

bool removeContributor(int playlistID, int userID)
returns true (if successful)

Class SessionManager

None

bool storeSessionData(int userID, SessionData sessionData:):
return true (if successful)

SessionData getSessionData(int userID)
return SessionData

Class UserAccount

string userName
int userID
string userEmail
string password
string displayName
byte profilePicture
int playListID
List<songs> playList

bool registerUser(string userName,string userEmail, string password
return true (if success)

bool login(string userName, string password)
return true (if success)

bool updateProfile(int userID, UserProfileDetailsuserDetails):
returns true (if success)

UserPreferences getPreferences(int userID)
return UserPreferences

int addPlaylist(int userID, PlaylistDetails playlistDetails)
return PlaylistID

bool customizePlaylist(int userID, int playlistID, PlaylistCustomizationDetails
customizationDetails)
return true (if success)

List<songs> getPlaylists(int userID)
return playList

Class SpotifyAPIManager

None

AuthenticationRequest initiateSpotifyAuthProcess()
return AuthenticationRequest

AuthenticationRequest
requestSpotifyAuthorization(List<Scope>scopes)
return AuthenticationRequest

List<SpotifyTrack> fetchSpotifyCatalog(string query)
return SpotifyTrack

bool addTrackToSpotifyUserPlaylist(string playlistID, string
trackID)
return true (if success)

List<SpotifyPlaylist> fetchSpotifyUserPlaylists()
return SpotifyPlaylist

Class generatePlayList

string favSongType
string favArtist
int favSongDuration
int favSongBPM
List <songs> recommandSongs

string favSongType(List<songs> cusPlayList) return
favSongType

string favArtist(List<songs> cusPlayList) return favArtist

int favSongDuration(List<songs> cusPlayList) return
favSongDuration

int favSongBPM(List<songs> cusPlayList])
return favSongBPM

Listc <songs> recommandPlaylist(string favSongType,
string favArtist, int favSongDuration, int favSongBPM)
return recommandSongs

Class UIController

None

void displayLoginPage()
void displayRegistrationPage()
void displayUserProfile(int userID)
void displayPlaylistGenerationPage()
void displaySpotifyIntegrationPage()
void displayCustomizationPage(int playlistID)
void displayCollaborativePlaylistPage(int playlistID)
void displayPlaybackPage(int playlistID)

Class DatabaseManager

User newUser

bool storeUser(userDetails: User)
return true (if success)

User getUser(usernameOrEmail: string)
return User

int addPlaylist(int userID, PlaylistDetails playlistDetails)
return PlaylistID

List<songs> getPlaylists(int userID)
return playList

bool storeCollaborativePlaylist(CollaborativePlaylist
playlistDetails: CollaborativePlaylist)
return true (if success)

CollaborativePlaylis getCollaborativePlaylist(int playlistID)
return CollaborativePlaylis

Class PlaybackController

None

bool playPlaylist(int playlistID)
returns true (if successful)

bool pausePlayback()
returns true (if successful)

bool skipTrack()
returns true (if successful)

2.2.1 UML Class Diagram Overview

2.1 UserAccount Class
Attributes:

- userName (string)
- userID (int)
- userEmail (string)
- password (string)
- displayName (string)
- profilePicture (byte)
- playListID (int)
- playList (linked list of Song objects)

2.2 Operations:

- bool registerUser(string userName, string userEmail, string password): Registers a new user with the provided credentials, which would then be saved to the server. The operation will return true if the registration is successful, otherwise a false will be returned along with a failed to register message.
- bool login(string userNameOrEmail, string password): The operation will first asks the user to enter a username or email and a password. It will then compare the entered information with the existing username, email, and password stored in the server and returns true if the login is successful.
- bool updateProfile(int userID, byte profilePicture, string displayName): Updates the user's profile information, such as display name and profile picture. Returns true if the update is successful. If only the profilePicture or displayName was entered, the operation will only update the entered information and retains the other attribute.
- UserPreferences getPreferences(int userID): Returns the user's preferences, such as preferred genres and customization settings.
- int addPlaylist(int userID, CustomPlaylist playlistDetails): Returns the unique identifier of the created playlist.
- bool customizePlaylist(int userID, int playlistID, CustomPlaylist playlistDetails): Customizes an existing playlist based on user preferences. Returns true if customization is successful.
- List<Playlist> getPlaylists(int userID): Returns a list of playlists created by the user.

2.3 Song Class

Each song in the library belongs to the songs class. A songs class contains a songType, songName, and songArtist of type string, as well as songDuration and songBPM of type int. The songs class can perform getters functions such as getSongType, getSongName, getSongArtist, getSongDuration, and getSongBPM without taking any parameters.

Attributes:

- songType (string)
- songName (string)
- songArtist (string)
- songDuration (int)
- songBPM (int)

Operations:

- string getSongType()
- string getSongName()
- string getSongArtist()
- int getSongDuration()
- int getSongBPM()

2.4 CusPlaylist Class

A UserAccount class can have multiple cusPlaylist. A cusPlaylist contains an array of songs which was added by the user. The cusPlaylist can perform many getter operations such as getSongType, getSongName, getSongArtist, getSongDuration, and getSongBPM. These operations take the requested song in the playlist as parameter.

Attributes:

- Array of songs (CusPlaylist can contain multiple songs)

Operations:

- Various getter methods that take a requested song in the playlist as a parameter.

2.5 PlaylistManager Class

To add and remove songs from a playlist, the class PlaylistManager will be used. It contains no attribute but has two operations. The first operation is addSongToPlaylist, which accept a playlistID from the UserAccount and a song of class songs as parameter. It will return true if the song is successfully added to the playlist. The second operation is removeSongFromPlaylist, which also accept a playlistID and a song as parameter. The operation will return true if the song is removed from the playlist.

Operations:

- bool addSongToPlaylist(int playlistID, Song song): Returns true if the song is successfully added to the playlist.
- bool removeSongFromPlaylist(int playlistID, Song song): Returns true if the song is removed from the playlist.

2. 6 PlaybackController Class

User can have control of a song's playback by using the class PlaybackController. The class contains no attribute but has the following operations:

Operations:

- bool playPlaylist(int playlistID): Returns true if the playback of the selected playlist is initiated or resumed.
- bool pausePlayback(): Returns true if the current playback is paused.
- bool skipTrack(): Returns true if the system skips to the next song in the playlist.

2.7 SpotifyAPIManager Class

To integrate the system with Spotify, the class SpotifyAPIManager will be used. It contains no attributes and has the following operations:

Operations:

- AuthenticationRequest initiateSpotifyAuthProcess(): Returns an authentication request.
- AuthorizationRequest requestSpotifyAuthorization(List<Scope> scopes): Returns an authorization request.
- List<SpotifyTrack> fetchSpotifyCatalog(string query): Returns a list of matching tracks from Spotify's catalog.
- bool addTrackToSpotifyUserPlaylist(int playlistID, int trackID): Returns true if a track is added to the user's Spotify playlist.
- List<SpotifyPlaylist> fetchSpotifyUserPlaylists(): Returns a list of the user's Spotify playlists.

2. 8 GeneratePlaylist Class

The AI can generate a custom playlist for the user by using the generatePlayList class. This class contains a favSongType and favArtist of type string; a favSongDuration; a favSongBPM of type int; and a linked list of class songs named recommendSongs. The SortSongs class can performs the following operations based on an array of cusPlayList as parameter.

Attributes:

- favSongType (string)
- favArtist (string)
- favSongDuration (int)
- favSongBPM (int)
- recommendSongs (linked list of Song objects)

Operations:

- Various getter methods for attributes.
- LinkedList<Song> recommendSongs(): Returns a list of recommended songs based on attributes.

2.9 SessionManager Class

The SessionManager class will be used to manage each login session on a devide. It contains no attribute but has two operations. The first operation is storeSessionData, which accepts a unique userID of type int and a sessionData as parameter. The operation will return true if it manages to store the data on the device. The second operation is getSessionData, which uses the userID as parameter and return the sessionData that has already been stored using the first operation.

Operations:

- bool storeSessionData(int userID, SessionData sessionData): Returns true if data is successfully stored on the device.
- SessionData getSessionData(int userID): Returns stored session data.

2.10 CollaborativePlaylist Class

Operations:

- bool addContributor(int playlistID, int userID): Returns true if the user is successfully added as a contributor.
- bool removeContributor(int playlistID, int userID): Returns true if the user is removed from accessing the playlist.

2.11 UIController Class

The class UIContoller will handle all information and interfaces being displayed to the user. The class has no attributes and has the following operations:

Operations:

- void displayLoginPage(): Display the login page on the device.
- void displayRegistrationPage(): Display the registration page after the user choose to sign up for a new account.
- void displayUserProfile(int userID): Displays the user's profile page based on the userID provided.
- void displayPlaylistGenerationPage(): Display the playlists that is generated by the AI.
- void displaySpotifyIntegrationPage(): Display the Spotify integration page.
- void displayCustomizationPage(int playlistID): Display the playlist customization page, which allows the user to add/remove songs or change the playlist’s name.
- void displayCollaborativePlaylistPage(int playlistID): void: Display the collaborative playlist page and the name of all users contributing to the playlists.
- void displayPlaybackPage(int playlistID): void: Display the playback page, which include the song’s name, the song’s length, the pause and unpause button, and the skip button

2.12 DatabaseManager Class

To manage all the users, playlists, and the collaborations between playlists, the class DatabaseManager is used. It contains no attributes and has the following operations:

Operations:

- bool storeUser(UserAccount userDetails): Returns true if user data is successfully stored in the database.
- UserAccount getUser(string usernameOrEmail): Retrieves user data from the database based on the username or email.
- int storePlaylist(CustomPlaylist playlistDetails): Returns the unique identifier of the stored playlist.
- CustomPlaylist getPlaylist(int playlistID): Retrieves playlist data from the database.
- bool storeCollaborativePlaylist(CollaborativePlaylist playlistDetails): Returns true if collaborative playlist data is successfully stored.
- CollaborativePlaylist getCollaborativePlaylist(int playlistID): Retrieves collaborative playlist data from the database based on its identifier.

2.13 TemplateManager Class

Operations:

- CustomPlaylist applyTemplate(CustomPlaylist playlist, Template template): Applies a template to a playlist and returns the updated playlist.
    - 1To change the template of a playlist, the class TemplateManager will be used. It contains no attribute and has an operation named applyTemplate. The operation takes a cusPlaylist and a template as parameter and will return the same playlist with the updated template.

3. Development Plan and Timeline

3.1 Software Gantt Chart

Continued on Next Page →

AI Playlist Generator With Spotify Integration Schedule

ID	Task name	Description	Resources	Duration	Priority	Start Date	Finish Date
1	**AI Playlist Generator With Spotify Integration**			107.0 d.	None	1/3/22	5/31/22
2	**Phase 1: Requirement Analysis**			21.0 d.	None	1/3/22	1/31/22
3	Gather User Needs	Conduct interviews, surveys, and research to gather user requirements and preferences. Document user stories and use cases.	Business Analyst / Project Manager	6.0 d.	1	1/3/22	1/10/22
4	Define System Scope	Determine the boundaries and limitations of the AI Playlist Generator system. Establish the scope of functionalities to be developed.	Project Manager / System Architect	8.0 d.	1	1/11/22	1/20/22
5	Identify Key Features	Identify the essential features and capabilities required for the AI Playlist Generator, considering user feedback and market trends.	Product Owner / Business Analyst	7.0 d.	1	1/21/22	1/31/22
6	**Phase 2: System Design**			19.0 d.	None	2/1/22	2/25/22
7	UI/UX Design	Create wireframes and mockups for the user interface (UI) and design the user experience (UX) to ensure an intuitive and visually appealing design.	UI/UX Designer	9.0 d.	2	2/1/22	2/11/22
8	Database Design	Design the database schema, including tables, relationships, and data storage structures required to support playlist generation and user profiles.	Database Architect	15.0 d.	1	2/1/22	2/21/22
9	API Design	Define the structure and endpoints of the application programming interfaces (APIs) needed for frontend-backend communication.	System Architect / Backend Developer	19.0 d.	1	2/1/22	2/25/22
10	**Phase 3: Frontend Development**			31.0 d.	None	2/25/22	4/8/22
11	Develop UI Components	Implement the UI components, including screens, buttons, forms, and interactive elements, based on the UI/UX design. Ensure responsiveness and cross-browser compatibility.	Frontend Developer / UI/UX Designer	20.0 d.	1	2/25/22	3/24/22
12	Implement User Flows	Develop the logic for user navigation and interaction within the UI, ensuring smooth user experiences.	Frontend Developer	7.0 d.	2	3/25/22	4/4/22
13	Testing and Debugging	Conduct unit testing and debugging of frontend components to identify and resolve issues and ensure proper functionality.	Quality Assurance Tester / Frontend Developer	6.0 d.	3	4/1/22	4/8/22
14	**Phase 4: Backend Development**			43.0 d.	None	2/25/22	4/26/22
15	Implement APIs	Develop the backend APIs required for user management, playlist generation, and data retrieval.	Backend Developer	23.0 d.	2	2/25/22	3/29/22
16	Set Up Database	Create and configure the database system, including tables, indexes, and data storage, as per the database design.	Backend Developer	24.0 d.	1	2/25/22	3/30/22
17	Implement Algorithms	Develop and implement AI and machine learning algorithms for playlist generation based on user preferences.	AI/ML Engineer / Backend Developer	43.0 d.	1	2/25/22	4/26/22
18	**Phase 5: Integration With Spotify**			15.0 d.	None	4/8/22	4/28/22
19	Authenticate with Spotify	Implement authentication mechanisms to securely connect the AI Playlist Generator with the Spotify platform.	System Integrator	15.0 d.	1	4/8/22	4/28/22
20	Access Spotify Catalog	Develop features that allow users to browse and select songs from Spotify's music catalog.	Backend Developer / System Integrator	7.0 d.	2	4/8/22	4/18/22
21	Playlist Integration	Establish mechanisms to integrate user-generated playlists with the user's Spotify account.	Backend Developer / System Integrator	6.0 d.	2	4/8/22	4/15/22
22	**Phase 6: Testing and Quality Assurance**			11.0 d.	1	4/29/22	5/13/22
23	System Testing	Conduct comprehensive system testing to verify the functionality of all features. Identify and address integration issues, performance bottlenecks, and security vulnerabilities.	Quality Assurance Team	11.0 d.	2	4/29/22	5/13/22
24	**Phase 7: Maintenance and Updates**			13.0 d.	None	5/13/22	5/31/22
25	Ongoing Maintenance	Establish an ongoing maintenance plan to address issues, apply security patches, and perform routine updates. Monitor system performance and scalability.	System Adminstrators	13.0 d.	2	5/13/22	5/31/22

Planned start Date: January 3, 2022
Planned completion Date: May 31, 2022
Total construction Period: 107 days

Test Plan

Unit Testing - GeneratePlaylist Class

Targeted Class: GeneratePlaylist
Features Tested: The recommendation of songs based on user preferences.
Test Sets/Vectors:

- Test 1: Verify that songs are recommended based on the favorite song type.
- Test 2: Verify that songs are recommended based on the favorite artist.
- Test 3: Verify that songs are recommended based on the favorite song duration.
- Test 4: Verify that songs are recommended based on the favorite song BPM.

Scope: Unit tests will focus on individual methods and attributes of the GeneratePlaylist class. These tests ensure that the class can accurately recommend songs based on user preferences.

TESTS THAT PASS

```
public class GeneratePlaylistTest {

    @Test 1
    public void testRecommendSongsWithValidUserPreferences() {
        GeneratePlaylist generatePlaylist = new GeneratePlaylist();
        UserPreferences userPreferences = new UserPreferences();
        userPreferences.setFavSongType("Pop");
        userPreferences.setFavArtist("Artist1");
        userPreferences.setFavSongDuration(180);
        userPreferences.setFavSongBPM(120);
     Song[] recommendedSongs =
     generatePlaylist.recommendSongs(userPreferences);

     assertTrue(recommendedSongs.length > 0, "At least one song
     should be recommended with valid user preferences.");
    }
}
```

TESTS THAT FAIL

```
public class GeneratePlaylistTest {

    @Test 2
    public void testRecommendSongsWithInvalidFavoriteGenre() {
        GeneratePlaylist generatePlaylist = new GeneratePlaylist();
        UserPreferences userPreferences = new UserPreferences();
        userPreferences.setFavSongType("NonexistentGenre");
     Song[] recommendedSongs =
     generatePlaylist.recommendSongs(userPreferences);
```

```
      assertEquals(0, recommendedSongs.length, "No songs should be
      recommended due to an invalid favorite genre.");
    }

    @Test 3
    public void testRecommendSongsWithInvalidUserPreferences() {
        GeneratePlaylist generatePlaylist = new GeneratePlaylist();
        UserPreferences userPreferences = new UserPreferences();
        userPreferences.setFavSongType(null);
        userPreferences.setFavArtist(null);
        userPreferences.setFavSongDuration(-1);
        userPreferences.setFavSongBPM(-1);
      Song[] recommendedSongs =
      generatePlaylist.recommendSongs(userPreferences);

      assertEquals(0, recommendedSongs.length, "No songs should be
      recommended due to invalid user preferences.");
    }
}
```

Functional Testing - SpotifyAPIManager Class

Targeted Class: SpotifyAPIManager
Features Tested: Interaction with the Spotify API for authentication, authorization, and catalog retrieval.
Test Sets/Vectors:

- Test 1: Verify that the Spotify API authentication process is initiated successfully.
- Test 2: Verify that the Spotify API authorization request is made with specified scopes.
- Test 3: Verify that songs are fetched from Spotify's catalog based on a search query.
- Test 4: Verify that a track is successfully added to the user's Spotify playlist.
- Test 5: Verify that the user's Spotify playlists are retrieved.

Scope: Functional tests ensure that the SpotifyAPIManager class can interact correctly with Spotify's APIs, including authentication, authorization, and data retrieval.

TESTS THAT PASS

```
public class SpotifyAPIManagerTest {

    @Test 1
    public void testFetchSpotifyCatalogWithValidQuery() {
        SpotifyAPIManager spotifyAPIManager = new
SpotifyAPIManager();
        String query = "SongName";
      SpotifyTrack[] spotifyTracks =
      spotifyAPIManager.fetchSpotifyCatalog(query);
```

```
     assertTrue(spotifyTracks.length > 0, "At least one Spotify
     track should be fetched with a valid query.");
    }

    @Test 2
    public void testAddTrackToValidSpotifyUserPlaylist() {
        SpotifyAPIManager spotifyAPIManager = new
SpotifyAPIManager();
        int playlistId = 1;  // Valid playlist ID
        int trackId = 12345;  // Valid track ID
     boolean result =
     spotifyAPIManager.addTrackToSpotifyUserPlaylist(playlistId,
     trackId);

     assertTrue(result, "Adding a track to a valid playlist should
     be successful.");
    }
}
```

TESTS THAT FAIL

```
public class SpotifyAPIManagerTest {

    @Test 1
    public void testFetchSpotifyCatalogWithInvalidQuery() {
        SpotifyAPIManager spotifyAPIManager = new
SpotifyAPIManager();
        String query = "InvalidQuery";
     SpotifyTrack[] spotifyTracks =
     spotifyAPIManager.fetchSpotifyCatalog(query);

     assertEquals(0, spotifyTracks.length, "No Spotify tracks should
     be fetched due to an invalid query.");
    }

    @Test 2
    public void testAddTrackToInvalidSpotifyUserPlaylist() {
        SpotifyAPIManager spotifyAPIManager = new
SpotifyAPIManager();
        int playlistId = -1;  // Invalid playlist ID
        int trackId = 12345;  // Valid track ID
     boolean result =
     spotifyAPIManager.addTrackToSpotifyUserPlaylist(playlistId,
     trackId);
```

```
      assertFalse(result, "Adding a track to an invalid playlist
      should fail.");
    }
}
```

System Testing - DatabaseManager Class

Targeted Class: DatabaseManager
Features Tested: Storing and retrieving user and playlist data in the database.
Test Sets/Vectors:

- Test 1: Verify that user data is successfully stored in the database.
- Test 2: Verify that user data can be retrieved from the database based on username or email.
- Test 3: Verify that playlist data is successfully stored in the database.
- Test 4: Verify that playlist data can be retrieved from the database based on the playlist identifier.
- Test 5: Verify that collaborative playlist data is successfully stored in the database.
- Test 6: Verify that collaborative playlist data can be retrieved from the database based on the playlist identifier.

Scope: System tests validate the complete functionality of the DatabaseManager class, ensuring that user and playlist data can be stored and retrieved accurately from the database.

TESTS THAT PASS/FAIL

```
public class DatabaseManagerTest {

    @Test 1
    public void testRetrieveNonexistentUserData() {
        DatabaseManager databaseManager = new DatabaseManager();
      UserAccount retrievedUser =
      databaseManager.getUser("nonexistent_user");

      assertNull(retrievedUser, "Retrieving nonexistent user data
      should return null.");
    }

    @Test 2
    public void testStoreUserWithDuplicateUsername() {
        DatabaseManager databaseManager = new DatabaseManager();
      UserAccount user1 = new UserAccount("duplicate_user",
      "user1@example.com", "securepass1");
      UserAccount user2 = new UserAccount("duplicate_user",
      "user2@example.com", "securepass2");
        boolean result1 = databaseManager.storeUser(user1);
        boolean result2 = databaseManager.storeUser(user2);
```

```
        assertTrue(result1, "User 1 should be successfully stored.");
        assertFalse(result2, "User 2 should fail to store due to
        duplicate username.");
    }
}
```

Class Song (Unit)						
Test Case	Song Type	Song Name	Artist	Duration	BPM	Classes Tested
1	"Hip-hop"	"Save your tears"	"The Weeknd"	249	118	(1,3,5,7,9)
2	"[rap, R&B]"	"<God's plan, Skyline>"	"<Drake, Kalid>"	900	1001	(1,3,4,8,10)
3	"<>!+"	123	0	Rihana	120	(1,4,6,8,9)
4	[1,11,12,14]	""	"123"	"+-=>"	-100	(1,3,5,8,10)
5	TRUE	"{}"	[A,B,C],Nick	FALSE	[1,2,3],12	(2,3,6,8,10)
Class Number		Valid input				
1	Valid SongType	string				
2	InVadid Song Type	!string				
3	getSongName	string				
4	InVadid Song name	!string				
5	getArtist	string				
6	Invalid Artist	!string				
7	getSongDuration	0 < duration && duration <= 600				
8	Invalid Duration	0 >= duration || duration > 600				
9	getSongBPM	0 < BPM && BPM <= 999				
10	Invalid BPM	0 >= BPM || BPM > 999				
Class cusPlayList (functional)						
TestCase	addSongToPlaylist	removeSongFromPlaylis	Playlist applyTemplate	Classes Tested		
1	(12, "God's Plan")	(12, "God's Plan")	(myfav, black.JPG)	(1,3,5)		
2	(abc,123)	(abv,"Skyline")	(myfav)	(2,4,6)		
3	(-1,"")	("","")	[myfav, chill, gym]	(2,3,6)		
4	("","")	(-12, ["abc,cbv"])	("", black.JPG)	(1,4,6)		
5	(12, [abc,afc,dfg])	("","abc")	(123, 123.JPG)	(2,3,5)		
Class Number						
1	addSongToPlaylist	(int>0, song)				
2	Invalid addSongToPlaylist	(int<0, !song)				
3	removeSongFromPlaylist	(int>0, song)				
4	Invalid removeSongFromPlaylist	(int<0, !song)				
5	Playlist applyTemplate	(Playlist, PlaylistTemplate)				
6	Invalid Playlist applyTemplate	!(Playlist playlist, PlaylistTemplate)				
Class UserAccount (System)						
TestCase	registerUser	login	addPlaylist	getPlaylists	Cases Tested	

1	(tobi, tobi@gmail.com, "123456")	(tobi, "123456")	(1,12, (name:study, background:black))	1	(1,3,5,7)	
2	("","","avcdf")	({}[], "12345")	(a,av,(name:chill, background:white))	abc	(1,4,6,8)	
3	("","",{}[]:)	("","")	(1,12,12)	[1,3,4,5]	(2,3,6,8)	
4	([a,c],[a,b,c],{}[])	({}[],[1,2,3,4])	(1,-12, (name:study, background:black))	0	(2,4,6,8)	
5	([1,2,4],12,124)	([a,b,c],1,3)	("","","")	-99	(2,4,5,8)	
Class Number						
1	registerUser	(string, string, string)				
2	Invalid registerUser	!((string, string, string)				
3	login	(string, string)				
4	Invalid login	!(string, string)				
5	addPlaylist	(int>0, int>0, PlaylistCustomizationDetails)				
6	Invalid addPlaylist	(int<0, int<0, PlaylistCustomizationDetails)				
7	getPlaylists	>0				
8	Invalid getPlaylists	<0				

The test set 1 include:

- Class Song as unit test
- Class cusPlayList as functional test
- Class UserAccount as system test

1. Unit test:
   We choose the song class because it mainly collects song information.
   1.1. All the valid input
   1.2. Show that using list of names is invalid for Song Name and Artist variables.
   1.3. String is invalid for Duration and integer is not.
   1.4. Blank”” is accepted as a string. BPM can not be negative
   1.5. Boolean variables are not invalid in this class.
2. Functional test:
   We choose cusPlayList class because it's also aggregating with two other class which is PlaylistManager & PlaylistManager)
   2.1. All the valid input
   2.2. Trying all invalid input
   2.3. Negative number is not valid in addSongToPlaylist, Blank”” will be valid in removeSongFromPlaylist.
   2.4. Blank will be accepted as string in addSongToPlaylist, array of string is invalid.
   2.5. Blank will be accepted as string in removeSongFromPlayList, array of string is invalid.
3. System test:
   We choose UserAccount as a system test because it includes many different functions and can be operated as an independent system.
   3.1. Showing all valid input
   3.2. Blank”” can be accepted in registerUser but special sign and brackets is invalid in Login
   3.3. Blank “” is valid in login as a string, array of integer is not a integer
   3.4. getPlaylist only accept integer greater than 0, array of string is invalid in registerUser
   3.5. Negative number is invalid in getPlaylist, array of number is invalid in registerUser class.

4. Data Management Strategy

In our AI-powered playlist Generator with Spotify Integration, effective data management is crucial to ensure the security, integrity, and accessibility of user-related information, song data, and playlists. We employ SQL databases as the primary data storage solution for their relational capabilities and data structure organization. Our data management strategy, leveraging SQL databases for structured data and implementing backup databases for redundancy, aims to strike a balance between data integrity, security, and performance while ensuring data availability and disaster recovery capabilities. This approach aligns with our system's objectives and user data protection requirements.

4.1 Overview

Design Decisions:
Choice of SQL: SQL databases were chosen due to their relational capabilities, structured organization, and data integrity features. This choice aligns with our goal of efficiently managing user data, song information, and playlist customization.
Number of Databases: We employ a separate SQL database for each primary data category, including User Information, Song Data, and Playlists. This organization enhances data separation and simplifies data management, contributing to system scalability.

Backup Databases:
To enhance data redundancy and disaster recovery capabilities, we maintain backup databases for each of the primary databases mentioned above. These backup databases mirror the structure of their respective primary databases and provide synchronized copies of data. This strategy ensures that even in the event of data loss or system failures in the primary databases, critical information can be promptly restored from the backups.

Tradeoffs:
Processing Cost: SQL databases may have a higher processing cost compared to NoSQL databases. However, the tradeoff is justified by the need for structured and relational data management, which is critical for user experience and data security.
Structured vs. Unstructured Data: While NoSQL databases offer advantages in terms of cost and flexibility, they often result in unstructured data that may require more time for information retrieval. SQL's structured approach helps ensure efficient data access.
Data Integrity and Security: SQL databases are chosen for their robust data integrity and security features, including data encryption. This prioritizes user data protection but may involve higher implementation costs.

Alternatives Considered:
NoSQL Databases: NoSQL databases were considered for their cost-effectiveness and flexibility. However, their unstructured nature and limited relational capabilities made them less suitable for our structured data requirements.
Single Database: An alternative could have been using a single SQL database to store all data categories. However, this approach might lead to data complexity, reduced scalability, and difficulties in data retrieval.

4.2 Database Descriptions

4.2.1 User Database:
Choice of SQL Database: The User Database is designed as an SQL database due to its relational capabilities. Data within this database is structured and organized into two primary tables.
Table 1: User Information: This table stores essential user details, including User ID (as the Primary Key), email, phone, username, and other relevant information. These details are critical for user authentication and profile management.
Table 2: Playlist Information: The User Database includes a Foreign Key, Playlist ID, which establishes a relationship with the Playlist Database. This allows users to access the playlists they create and those generated by the AI. The use of a relational database enhances data integrity and security.
Data Encryption: To prioritize user data security, this database employs encryption techniques. Encryption significantly reduces the risk of data loss and unauthorized access, ensuring the confidentiality of sensitive information.
Processing Cost Tradeoff: While SQL databases offer strong data integrity and security, they may incur a higher processing cost compared to NoSQL databases. This tradeoff is justified by the need for relational data management and enhanced security.

4.2.2 Song Database:
Choice of SQL Database: The Song Database can be implemented using either SQL or NoSQL. However, SQL is chosen for its ability to define a schema and organize song data efficiently.
Data Storage: This database primarily stores basic information about songs obtained from the Spotify database, including Song ID (Primary Key), song name, song duration, artist, and related attributes.
Schema Definition: SQL's schema capabilities allow for the structured organization of song data into tables. This schema simplifies data management, making it easier to retrieve and manipulate song information.
Data Control: SQL databases provide greater control over input data, allowing the selection and storage of specific attributes needed for future use.

Tradeoff: While NoSQL databases may offer cost advantages, they often result in unstructured data that requires more time to search and retrieve relevant information. SQL's structured approach is preferred for efficient song data management.

4.2.3 Playlist Database:

Choice of SQL Database: The Playlist Database is implemented using an SQL database due to its direct connection with users and the need for relational data management.

Primary Key and Foreign Key: The Playlist Database employs Playlist ID as the Primary Key, establishing a relationship with User ID, which serves as both the Primary Key and Foreign Key. This connection allows the database to associate playlists with specific users.

Customization Storage: This database stores user playlist customizations, including details such as playlist titles, track listings, and customization settings.

Structured Data: SQL databases organize data into tables, enabling faster loading speed and easier data access. The relational abilities of SQL are essential for retrieving necessary information.

Tradeoff: While SQL databases may come with higher implementation costs, they offer robust relational capabilities, critical for efficient data retrieval and user experience.

The three remaining databases serve as backups to the primary databases mentioned above, providing redundancy and data recovery capabilities to ensure data integrity and availability.

4.2.4 User Database Backup:

Purpose: The User Database Backup is designed to replicate and safeguard user-related data from the primary User Database. It serves as a redundant store of user information, ensuring continuity in user account management and profile data.

Data Mirroring: This database mirrors the structure of the primary User Database, containing tables for User Information and Playlist Information. User ID is set as the primary key for maintaining data integrity.

Synchronization: The data in the User Database Backup remains synchronized with changes made in the primary User Database, ensuring that user accounts and associated playlist details are consistently preserved.

Data Recovery: In the event of data loss or system failures in the primary User Database, the User Database Backup can be used to restore user accounts, playlists, and associated information, minimizing disruption to user services.

4.2.5 Song Database Backup:

Purpose: The Song Database Backup acts as a redundancy layer for song-related data, offering data recovery capabilities for song information retrieved from the Spotify database.

Data Storage: Similar to the primary Song Database, this backup database stores basic song information, including Song ID, song name, duration, artist details, and other relevant attributes.

Data Redundancy: Data in the Song Database Backup is periodically synchronized with the primary Song Database, ensuring that song details remain consistent.
Disaster Recovery: In cases where song data from the primary database is compromised or lost, the Song Database Backup can be employed to restore song-related information, ensuring the availability of a vast music catalog for users.

4.2.6 Playlist Database Backup:
Purpose: The Playlist Database Backup replicates playlist customizations and related data from the primary Playlist Database, offering an additional layer of data protection for user-generated playlists.
Data Replication: This backup database mirrors the structure of the primary Playlist Database, encompassing playlist customization details and the relationship between Playlist IDs and User IDs.
Data Consistency: Synchronization mechanisms keep the data in the Playlist Database Backup up-to-date with changes made in the primary Playlist Database, guaranteeing that user-customized playlists are preserved.
Data Restoration: In scenarios where user-generated playlist data is at risk or lost in the primary database, the Playlist Database Backup serves as a critical resource for restoring customized playlists and ensuring an uninterrupted user experience.

4.2.7 Analytics Database:
Choice of SQL Database: The Analytics Database is implemented as an SQL database due to its ability to manage structured data effectively.
Data Purpose: This database is dedicated to storing and managing data related to user interactions, system performance metrics, and playlist usage statistics.
Tables: The Analytics Database comprises tables such as UserActivityLogs, PlaylistUsageStatistics, SystemPerformanceMetrics, and UserFeedback.
Data Stored: The data stored includes user activity logs (e.g., login times, playlist creation), playlist usage statistics (e.g., popular playlists, most played songs), system performance metrics (e.g., response times, server load), and user feedback (e.g., ratings, comments).
Benefits: The use of SQL allows for efficient data organization, enabling the generation of reports and dashboards for user engagement analysis, system performance evaluation, and user behavior insights.

4.2.8 Logging Database:
Choice of SQL Database: The Logging Database is designed as an SQL database to facilitate structured storage of logs, errors, and user activities.
Data Purpose: This database is responsible for capturing and retaining logs related to system events, errors, and user actions.

Tables: The Logging Database includes tables such as ErrorLogs, UserActivityLogs, and SystemEventLogs.
Data Stored: Data stored in this database encompasses error logs (e.g., exceptions, crashes), user activity logs (e.g., login/logout events, user actions), and system event logs (e.g., system start/stop, maintenance tasks).
Benefits: The SQL structure aids in organizing log data efficiently, making it valuable for troubleshooting, auditing, and monitoring system health and user activities.

4.2.9 Content Management Database:
Choice of SQL Database: The Content Management Database is chosen as an SQL database to manage structured user-generated content effectively.
Data Purpose: This database stores content such as user comments, playlist reviews, and user-contributed playlist details.
Tables: The Content Management Database consists of tables such as UserComments, PlaylistReviews, and UserContributedPlaylists.
Data Stored: Content stored includes user comments on playlists, playlist reviews, and user-contributed playlists with associated information.
Benefits: SQL's table structure facilitates the management and retrieval of user-generated content, enhancing user engagement and community-building features within the platform.

4.2.10 Session Management Database:
Choice of SQL Database: The Session Management Database is implemented as an SQL database to track and manage user sessions across devices.
Data Purpose: This database is responsible for maintaining session data, user session states, and device information.
Tables: The Session Management Database includes tables such as ActiveSessions, UserSessionState, and DeviceInformation.
Data Stored: Data stored encompasses active user sessions, user session states (e.g., logged in/out), and device information associated with each session.
Benefits: SQL's relational capabilities enable consistent user experiences across devices by maintaining session continuity and providing efficient session data retrieval.

4.2.11 Cache Database:
Choice of Database: The Cache Database is used for caching frequently accessed data to optimize system performance.
Data Purpose: This database temporarily stores frequently accessed data to reduce the need for repetitive data retrieval from primary databases.
Data Stored: Data stored includes cached user playlists, AI-generated playlist previews, and other frequently accessed data.

Benefits: By leveraging caching techniques, the Cache Database enhances system responsiveness by minimizing data retrieval overhead, particularly for read-heavy operations.


User Accounts (Encrypted)
User Accounts (Backup, encrypted)
Playlists
Playlists (Backup)
AI Server 1
Songs
Songs (Backup)
User
Encrypted Connection
Load Balancer
AI Server 2
Analytics
AI Server 3
Logging
Content Management
Session Management
Cache