CSCI 201L Group Project

Group Members:
Front-end (Swift) - Nolan Earl, Pavly Habashy
Back-end (Java) - Paul (PJ) Hernandez, Samuel He
Database (SQL & Java) - Chengyi (Jeff) Chen

TrackChanges

CSCI 201L Group Project High-Level Requirements: Track Changes

High Level Idea: We need to create a iOS application interface where users who are fond of music can immerse themselves in an online social community comprised of their friends. The application will allow users to follow/unfollow users and artists to keep track on what their latest music trends are. As the user grows their music profile, the application will also give users recommended music content and users to follow to grow their profiles further. Ultimately, as more users and artists use the application, the app will serve as an online platform for music enthusiasts that share their love for music.

App Pages:

Log In

- On the Login page users will be able to login with their ~~Apple Music or~~ Spotify account. After selecting to sign in with ~~Apple Music or~~ Spotify the users will also be taken to the Home page but will have all functionality enabled. Alternatively, users will have the option to continue as a guest which will also grant them access to the app but with limited functionality. After selecting to continue as a guest users will be taken to the home page but instead of being able to follow friends, will only be able to follow artists and their activity.

Profile

- Display Picture
    - When users access the profile page, they will be able to see their profile picture on the top left-hand corner. Upon clicking on the profile picture, a modal should appear, allowing them to change their profile picture, either pulling the image from their Facebook library or from their iCloud library.
- Social Status on App
    - The area beside their profile picture should display 2 numbers - their number of Followers and the number of Following.
    - Upon clicking onto the number of Followers, they will be redirected to another view, listing their Followers and a button inside each row of follower giving the user the option to follow / unfollow them. Upon clicking on the number of Following, they will be redirected to another view which lists all the users they follow.
- Settings
    - Users should be able to click on a settings button which redirects them to another view, listing out changes that they can make to their profile, which includes the following - “Change Username”, “Change Primary Email”, “Change Phone number”, “Deactivate account (Bold and in Red)”, “Edit Interest Groups”.
- Content
    - The area below should first feature the songs and artists that they listen to most frequently, followed by blocks of their latest posts.

Search/Discover

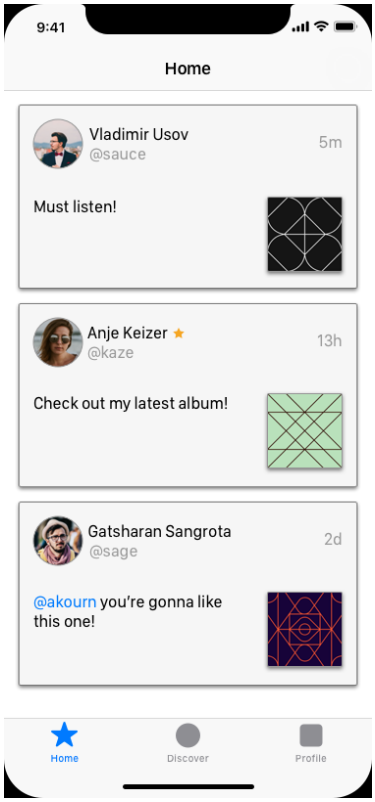
- On top of this page, there will be a search bar for users to look up friends, artists, albums, and songs. The search function will also give the user a drop down menu of recommendations based on the user’s query (while he or she is typing). Once, the user picks something, whether that be a song, artist, album, or a friend, a page will display the information based on the user’s selection.
    - If a user picks a friend, display the friend’s profile.
    - If a user picks a song, play that song.
    - If a user picks an album, go to the album page.
    - If a user picks an artist, go to the artist page.

- Below the search page, there will be a ‘discover page’ that should allow users to see randomly displayed recommended content in the form of a collection view. Users can see new songs, artists, albums, and new friends. If they click on any item, they will be directed to that page.

Home / Social Feed

- Scrollable page filled with news pushed by followed artists, prominent artists in selected interest group
    - How users interact with events/news/songs
        - Event will have title, description, and picture
            - If there are linked pages, users can click on link to open the page in designated browser
            - Option to play any music in event
                - Time stamp and pause/play feature
        - Like event and comment
            - Users can like comments
        - User can share the news with their friends
    - Artists that post will have their name and profile picture displayed on the top left side of the event
        - User can click on the name of the artist to go to their artist page
    - News is ordered chronologically
        - Most recent events are on top, oldest at bottom
        - Maximum event number is 100, when events have reached capacity remove oldest element to make room

Early Renders:



HIGHLIGHTED PARTS ARE THINGS WE KNOW ARE GOING TO BE IN THE APP FOR SURE.

UI/UX (6 hours)

- Splash screen of the logo which transitions to the login page. From there there will be two buttons displayed. One will be to login with an ~~Apple Music account~~ or alternatively login with a Spotify account.

- The home page will have a feed of posts based on the accounts the user is following. This will include shared artists, songs, and albums and additional posts by users. There will be buttons on the posts for a user to share to the post to the feed of users who follow them or up vote/down vote a post. There will be a start button for a user to listen to the content of a music post.

- A discover page allows the user to view trending music, new artists, and new music content

- A profile page where a user can see their basic profile information, a count of who they are following and who is following them. From this page they can access a complete list of who is following them and a list of who they are following. Additionally they can view their past posts and shares via a history log.

Database (4 hours)

- The database will consist of **nine tables** – a User table, a Follow table, a Post table, a PostUpvote table, a PostDownvote table, a Song table, a SongUpvote table, a SongDownvote table, and a UserSong table.
- The User table will be used for validating users and consist of userID (~~Apple Music API~~ ~~and~~ Spotify Music API no?), username, password, fname, lname, and timestamp representing the last login time.
- The Follow table will be used for storing the followerIDs (Apple Music API) foreign key to represent the followers corresponding to a particular user, userID foreign key, and the combination of them will be used as the Primary key for the table.
- The Post table will be used to store the postID primary key, userID foreign key, along with postupvoteID foreign key, postdownvoteID foreign key, postcontent (stores a serialized JSON object containing the associated image urls and text in the post), and a timestamp representing when the user posted.
- The PostUpvote table will contain the postID as a foreign key, along with the userID of the person that upvoted the post, and together they will form the primary key for that row. Iterating through this with a given postID will produce the total number of upvotes for that post.
- The PostDownvote table will contain the postID as a foreign key, along with the userID of the person that downvoted the post, and together they will form the primary key for that row. Iterating through this with a given postID will produce the total number of downvotes for that post.
- The Song table will be used to store the songID primary key, along with songupvoteID foreign key songdownvoteID foreign key.
- The SongUpvote table will be used to store the songID foreign key, along with the userID of the person that upvoted the song, and together they will form the primary key for that row. Iterating through this with a given songID will produce the total number of upvotes for that song.
- The SongDownvote table will be used to store the songID foreign key, along with the userID of the person that downvoted the song, and together they will form the primary key for that row. Iterating through this with a given songID will produce the total number of downvotes for that song.
- The UserSong table contains all the songs that the users have added onto their personal album for easy access from their profile page and will contain the userID foreign key, the songID foreign key, and together they will form the primary key for the row. There will also be a timestamp corresponding to when the song was last accessed by the user.

Backend-logic [Create Read Update Delete] (8 hours)

- Prerequisite technical tools: Eclipse IDE for Java EE, Apache Tomcat Server 9.0, MySQL 5.x, MySQL Workbench, MySQL server, Postman or any other REST Clients
- ~~Backend is Java Servlet, which will make requests to the MySQL server~~
- iOS will talk to the backend via HTTP connection - YES, we'll be using a websockets which are HTTP connections. Our application will consist of both client side and server side code. All client side code will

be written in Swift, and all server side code will be written in Java. We will use StarScream Websocket library in Swift to send messages back and forth to the Java backend. The frontend Swift code will control the View of the application.

- Frontend will forward database requests to the Java Servlet, which will get or store data from/to the MySQL database and, when needed, will forward data to the frontend
    - User information (name, profile picture url, songs), post ordering (use queue data structure with max capacity of 100), new post creation, upvotes, downvotes, albums, songs, and followers of a user will all be stored in MySQL database
    - On login user information will all be stored in database
    - Servlet will request the user's information when opening profile page
    - User creating a post will be stored in database and upvotes and downvotes from other users will also be stored in database
    - Servlet will request posts, upvotes, downvotes when the user opens the home page
    - Servlet will request another user's information when user clicks on another user's profile picture or name (the information that is requested will vary depending on whether the user is following the "clicked" user
    - For the search bar, depending on the search functionality specified, the servlet will request the appropriate data from the database
    - When opening the discover page, servlet will request data depending on what is randomly displayed; songs and friends with latest timestamp will be requested from the server to implement the "new friends and songs" functionality

CSCI 201L Group Project Detailed Design | TrackChanges

Software Requirements

Front-End

- A deployment of iOS 9 or higher is required.
- Spotify iOS SDK (To access database of songs and artists and enable Social Login): https://github.com/spotify/ios-sdk

Back-End

- MySQL database management system (MySQL Workbench)
- Java Server
    - ServerSocket API (To listen to port for requests from iOS client)
    - GSON API (To parse JSON data)
    - Java Database Connectivity API (To Create, Read, Update, Delete data from the SQL database using Java)
- Starscream - a conforming WebSocket client library in Swift

Hardware Requirements

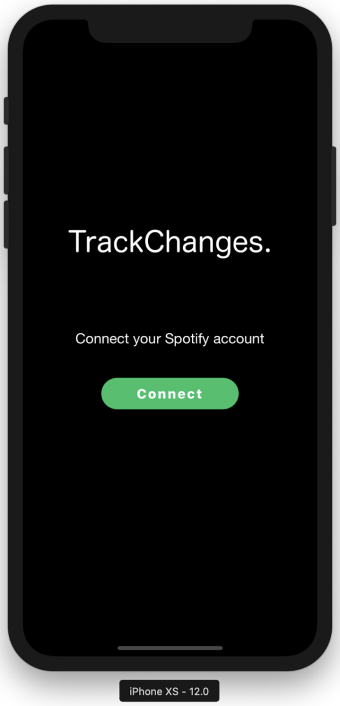
- A physical iOS device is needed to install the Spotify app
- A laptop running the SQL database as well as the Java server to handle requests

Front-End

LoginViewController.swift

This file handles all UI elements on the login page. This is where the user can choose one of the two access levels to the application. Each level is represented by a button that the user choose from.

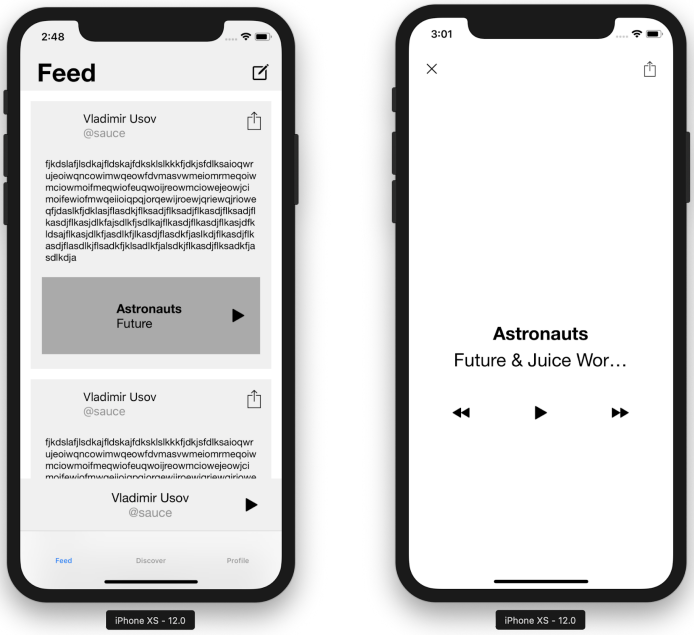
The first button will be a '*Connect to Spotify*' button. This will bring up a Safari View Controller with the Spotify login page. Upon signing in and authorizing *TrackChanges* to access the user's account, the user's token will be acquired and a `User` object will be created with the information retrieved from the user's token. From there, a back-end `verifyUser` function will be called to check if the user exists. If the user exists, the back-end function will update their information. If the user doesn't exist, the function will add them to the database. The view is now directed to the application's Feed View Controller.



The second button will be a 'Guest Mode' button (Not in the figure above). The view will be immediately directed to the *FeedViewController.swift* page.

FeedViewController.swift

This file handles all the UI elements on the feed page. Upon entering this page, a request will be sent to the server to retrieve all posts from the user through a back-end `getFeed` function will be called. The function will return a response of a list of posts from the users that the logged in user follows, which this function will iterate through and populate the Feed page. When the user scrolls to the bottom of the feed page, another request will be sent to the server to respond with 15 later posts from the user's following which will be lazy loaded and appended to the bottom of the screen.

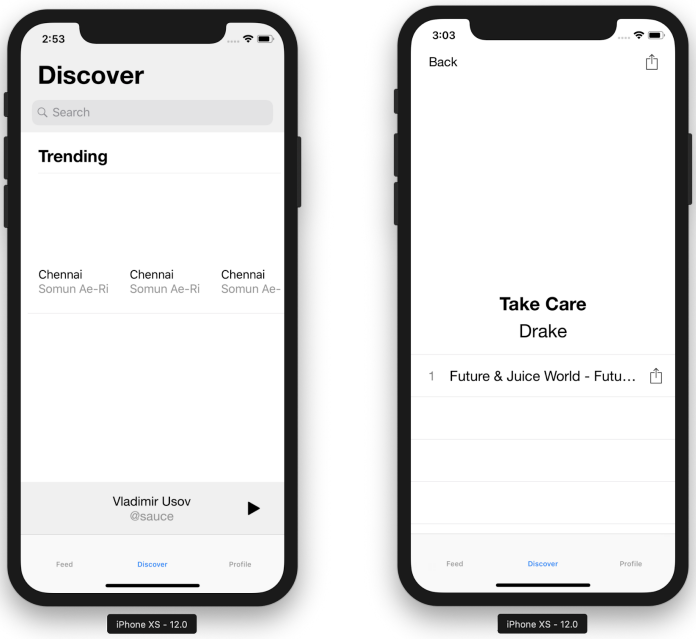


The user can choose to create a post by tapping the compose button at the top right, share a post from their feed by tapping the share button, or play a song from a post by tapping the play button. There is a mini-player on the bottom of the page that shows the current playing song along with a play/pause button. Tapping anywhere on the mini-player should slide up the Now Playing screen. The user can also like/unlike posts (not displayed above). Back-end functions such as `addPost`, `sharePost`, `likePost`, and `unlikePost` will be called here.

In guest mode, the feed will show posts from popular artists. The user will be allowed to play access the equivalent of the free tier of a Spotify account. The Profile tab will be hidden along with the 'Compose' and 'Share' buttons in the Feed tab.

DiscoverViewController.swift

On the Discover tab, we will populate a `UITableView` of content that will be based on current music trends, rising artists, new songs, etc. Clicking on one of the album covers or song covers will take the user to a page that shows an entire album where they will be able to select the songs on the album and play them.

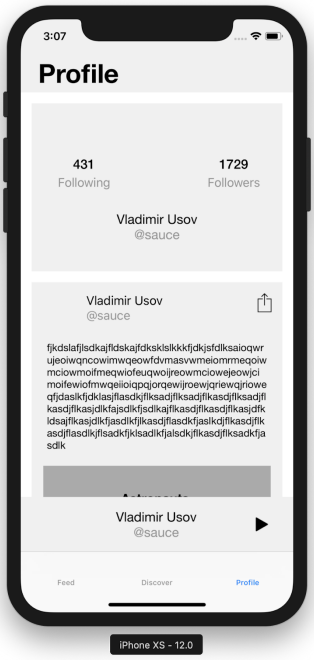


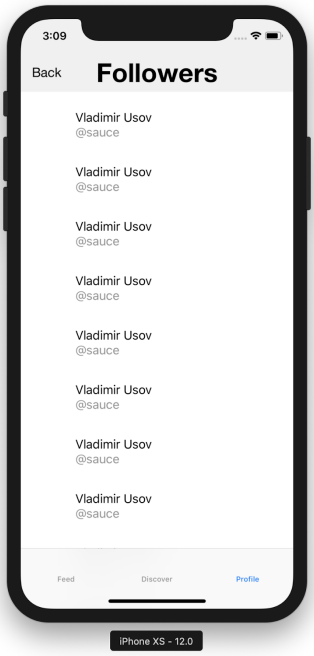
On the Discover page, a user can also search for content throughout the app. If a user is searching for music content, we will call the Spotify API to get results for the search term and display the results to the user on a Search Results page. If a user is searching for other users, we will query the database to return the set of users who match the search term. Clicking on one of the table view cells of the user results will take the current user to that person's profile page. The profile page will have a button for the current user to follow if they have not already. Clicking follow will update the database to add the current user to that person's "following". Conversely, a user who is following someone can click the button which will query the database and remove from the current user that person's "following". Back-end functions such as `follow` and `unfollow` will be called in this view.

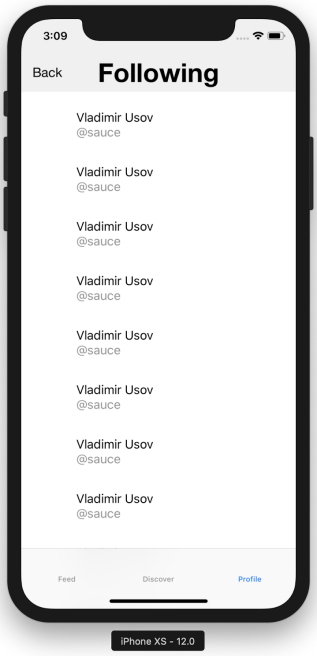
ProfileViewController.swift

On this page, the user's profile picture, number of followers and following, and their username is displayed. On the user's profile page, the user can click a button to see their followers and another to see who they are following. Each of these pages (on click of a table view cell) link to the same profile page of that person as described earlier. The same functionality will be available as above as well. A list of user's previous posts and shares will be displayed on this page. Back-end functions `getPosts`, `deletePost`, `getFollowers` and `getFollowing` will be called in this view.

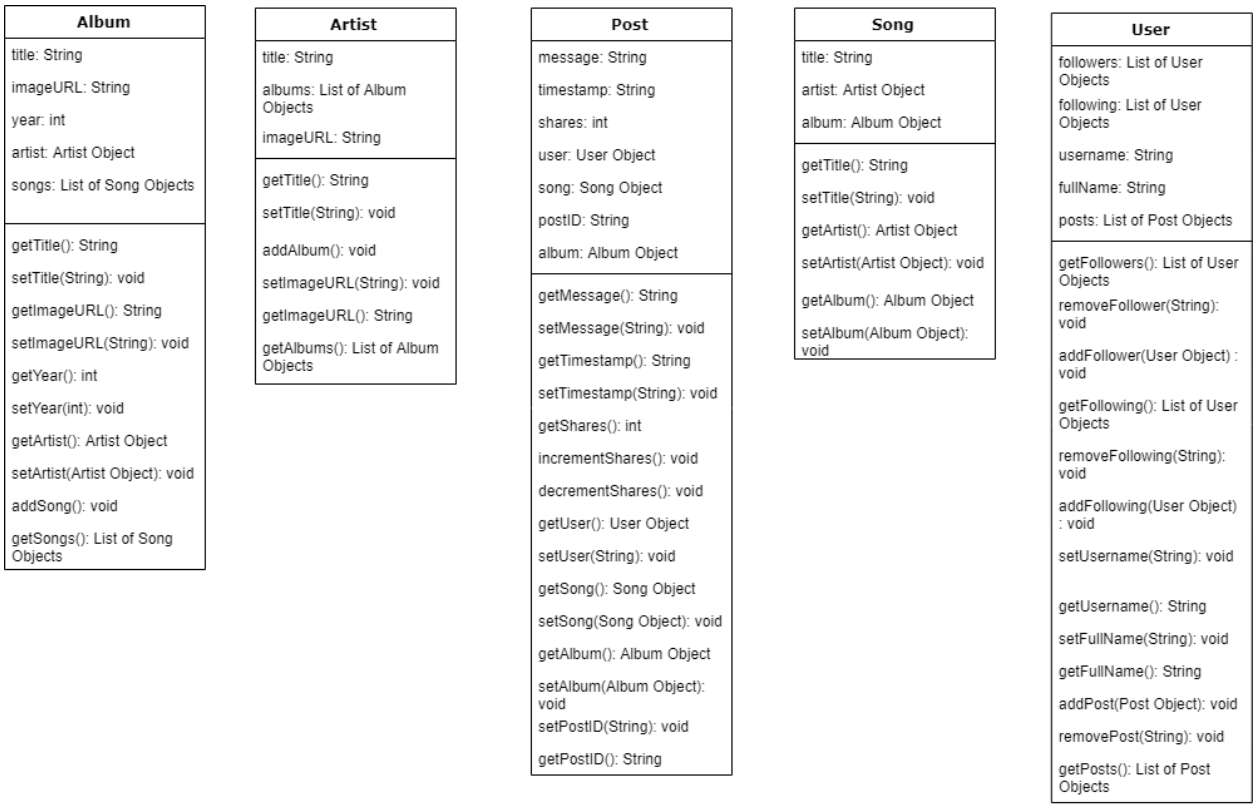
As stated above, this page does not exist in guest mode.







Front-end Class Structure diagram



Back-End

Server.java

The Server class handles communication from the frontend(Swift) and the backend(Java). It will use the ServerSocket class in Java to bound to a port that the frontend sends to. The Server class will also contain a data structure to hold each server thread from each client. This will most probably be in the form of a Vector. This class will allow us to see the connections made and the data being transferred between clients.

In terms of the data being transferred, we'll be using Starscream to pass serialized JSON objects from the frontend to the backend, and vice versa.

Functions:

- `public Server(int port):`
  The constructor will take in a port number chosen by the administrators. We will use this port number to instantiate a ServerSocket to bind to this port. After we successfully bind to the port, we will have an infinite while loop 'listening' in for connections from clients. Once, a connection is established and the client Socket accepted, the ServerThread gets added to the data structure.

- `public boolean verifyUser():`
  Upon logging in to the app, the user will have to be verified via the Database.java class. This function just calls the Database.java functions based on whether the user exists or not.

- `public void broadcast(Object o, ServerThread currentST):`
  This broadcast function will be mainly used for the *FeedViewController.swift.* Ideally, each client should be pushing their posts/content using the ServerThread class. Posts/Content should be serializable so we can pass it off to the Server. This function will iterate through the ServerThreads data structure and display it on each of the client's feed page.

- `public static void main(String []args):`
  Main method that will be used to run the server from our local machine.

ServerThread.java

This ServerThread class will extend from the Java's Thread class.

Private Variables:

- `private ObjectInputStream ois;`
- `private ObjectOutputStream oos;`
- `private Server s;`

Functions:

- `public ServerThread(Socket s, Server s)`
  The constructor will take in the socket and the server. The constructor will also instantiate an ObjectInputStream and ObjectOutputStream so that data objects can read and written. We will be expecting a JSON object from the Front-End, which would be parsed via GSON. After parsing the JSON object, the "`requestType`" parameter (should be specified before serialization), will then be extracted upon deserialization of the JSON object. This "`requestType`" parameter would then be responsible for calling the appropriate handler for the request in *Database.java*.

- `public void writeData(Object o)`
  This function will take in the Object that is being sent for broadcast and will be written in the output stream.

- `public void run()`
  This function will override the run method in Java's thread class. It will have an infinite while loop that is constantly reading from the output stream and broadcasting the updates on the threads.

Database.java

This Database class handles all the functions to Create, Read, Update, Delete data from the SQL database using a JDBC Driver.

Variables:

- `private static final String DATABASE_CONNECTION_URL:`
  This contains the function will store the global url we will use to connect to the SQL database that is storing all the data necessary for the application, for example:
  `"jdbc:mysql://localhost:3306/CalendarApp?user=root&password=&useSSL=false";`

Classes:

- `public class User, Artist, Album, Song, Post:`
  These classes will be a template for all the data we need to store for a user, song, post respectively as noted in the Database schema below and will be passed as the class required when parsing the JSON data from the front-end

Functions:

- `private boolean addUser(User newUser):`
  This function will be responsible for adding new users into the database with `"INSERT"` statements after a connection using the JDBC DriverManager is established. Insertion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return `"True"` if user is successfully added and `"False"` otherwise.

- `private boolean updateUser(User user):`
  This function will be responsible for updating an existing user in the database with `"UPDATE"` statements after a connection using the JDBC DriverManager is established. Update will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return `"True"` if user is successfully updated and `"False"` otherwise.

- `private boolean deactivateUser(String user_id):`
  This function will be responsible for deactivating existing users in the database by setting `"user_is_active"` to `"False"` after a connection using the JDBC DriverManager is established. Deactivation will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return `"True"` if user is successfully deactivated and `"False"` otherwise.

- `private boolean deleteUser(String user_id):`
  This function will be responsible for deleting existing users from the database permanently using the `"DELETE"` statement after a connection using the JDBC DriverManager is established. Deletion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return `"True"` if user is successfully deleted and `"False"` otherwise.

- `private boolean addArtist(Artist newArtist):`
  This function will be responsible for adding new artists into the database with `"INSERT"` statements after a connection using the JDBC DriverManager is established. Insertion will also be surrounded by Try, Catch

blocks to ensure a minimum level of error handling. Function will return "`True`" if user is successfully added and "`False`" otherwise.

- `private boolean updateArtist(Artist artist):`
  This function will be responsible for updating an existing artist in the database with "`UPDATE`" statements after a connection using the JDBC DriverManager is established. Update will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "`True`" if user is successfully updated and "`False`" otherwise.

- `private boolean deleteArtist(String artist_id):`
  This function will be responsible for deleting existing artists, along with all their albums, songs, and all posts that included those albums and songs from the database permanently using the "`DELETE`" statement after a connection using the JDBC DriverManager is established. Deletion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "`True`" if artist is successfully deleted and "`False`" otherwise.

- `private boolean follow(String user_id, String follower_id):`
  This function will be responsible for adding a new follower relationship into the database with "`INSERT`" statements after a connection using the JDBC DriverManager is established. Insertion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "`True`" if follower is successfully added and "`False`" otherwise.

- `private boolean unfollow(String user_id, String follower_id):`
  This function will be responsible for deleting a follower relationship permanently using the "`DELETE`" statement after a connection using the JDBC DriverManager is established. Deletion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "`True`" if relationship is successfully deleted and "`False`" otherwise.

- `private String[] getFollowers(String user_id):`
  This function will be responsible for retrieving all the followers of a user using the "`SELECT`" statement after a connection using the JDBC DriverManager is established. Deletion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return an array of "`user_id`"(s) corresponding to each follower. Size of array will be the number of followers a user has.

- `private String[] getFollowing(String user_id):`
  This function will be responsible for retrieving all the users that the current user is following using the "`SELECT`" statement after a connection using the JDBC DriverManager is established. Deletion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return an array of "`user_id`"(s) corresponding to each user the current user is following. Size of array will be the number of users the user specified is following.

- `private boolean addAlbum(Album newAlbum):`
  This function will be responsible for adding new albums, album artist(s), and all the songs in it into the database with "`INSERT`" statements after a connection using the JDBC DriverManager is established. Insertion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "`True`" if album is successfully added and "`False`" otherwise.

- `private boolean deleteAlbum(String album_id):`

This function will be responsible for deleting existing albums, and all posts that included those albums and the songs inside the album from the database permanently using the "DELETE" statement after a connection using the JDBC DriverManager is established. Deletion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "True" if album is successfully deleted and "False" otherwise.

- `private boolean addSong(Song newSong, String album_id):`
  This function will be responsible for adding new songs and its corresponding artist(s), as well as the updating the "AlbumSong" relationship (storing the fact that the song belongs to the album whose "album_id" is specified) in the database with "INSERT" statements after a connection using the JDBC DriverManager is established. Insertion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "True" if song is successfully added and "False" otherwise.

- `private boolean addSong(Song newSong):`
  This function will be responsible for adding new songs and its corresponding artist(s) into the database with "INSERT" statements after a connection using the JDBC DriverManager is established. Insertion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "True" if song is successfully added and "False" otherwise.

- `private boolean likeSong(String song_id, String user_id):`
  This function will be responsible for tracking which users like a particular song in the database with "INSERT" statements to the "AlbumSong" table after a connection using the JDBC DriverManager is established. Insertion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "True" if addition is successful and "False" otherwise.

- `private boolean unlikeSong(String song_id, String user_id):`
  This function will be responsible for deleting the relationship of the user liking the song permanently using the "DELETE" statement after a connection using the JDBC DriverManager is established. Deletion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "True" if relationship is successfully deleted and "False" otherwise.

- `private boolean deleteSong(String song_id):`
  This function will be responsible for deleting existing songs, the "AlbumSong" relationships,and all posts that included those songs inside the album from the database permanently using the "DELETE" statement after a connection using the JDBC DriverManager is established. Deletion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "True" if song is successfully deleted and "False" otherwise.

- `private boolean addPost(Post newPost):`
  This function will be responsible for adding a new post in the database with "INSERT" statements after a connection using the JDBC DriverManager is established (and also updates the "PostAlbum" and "PostSong" table if required). Insertion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "True" if post is successfully added and "False" otherwise.

- `private Post[] getPosts(String user_id):`

This function will be responsible for retrieving the posts from the database with "SELECT" statements after a connection using the JDBC DriverManager is established. Retrieval will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return an array of Post objects and null if no posts are found.

- `private Post[] getFeed(String user_id):`
  This function will be responsible for retrieving the posts from the users that the user specified is following through the database with "SELECT" statements after a connection using the JDBC DriverManager is established. Retrieval will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return an array of Post objects and null if no posts are found.

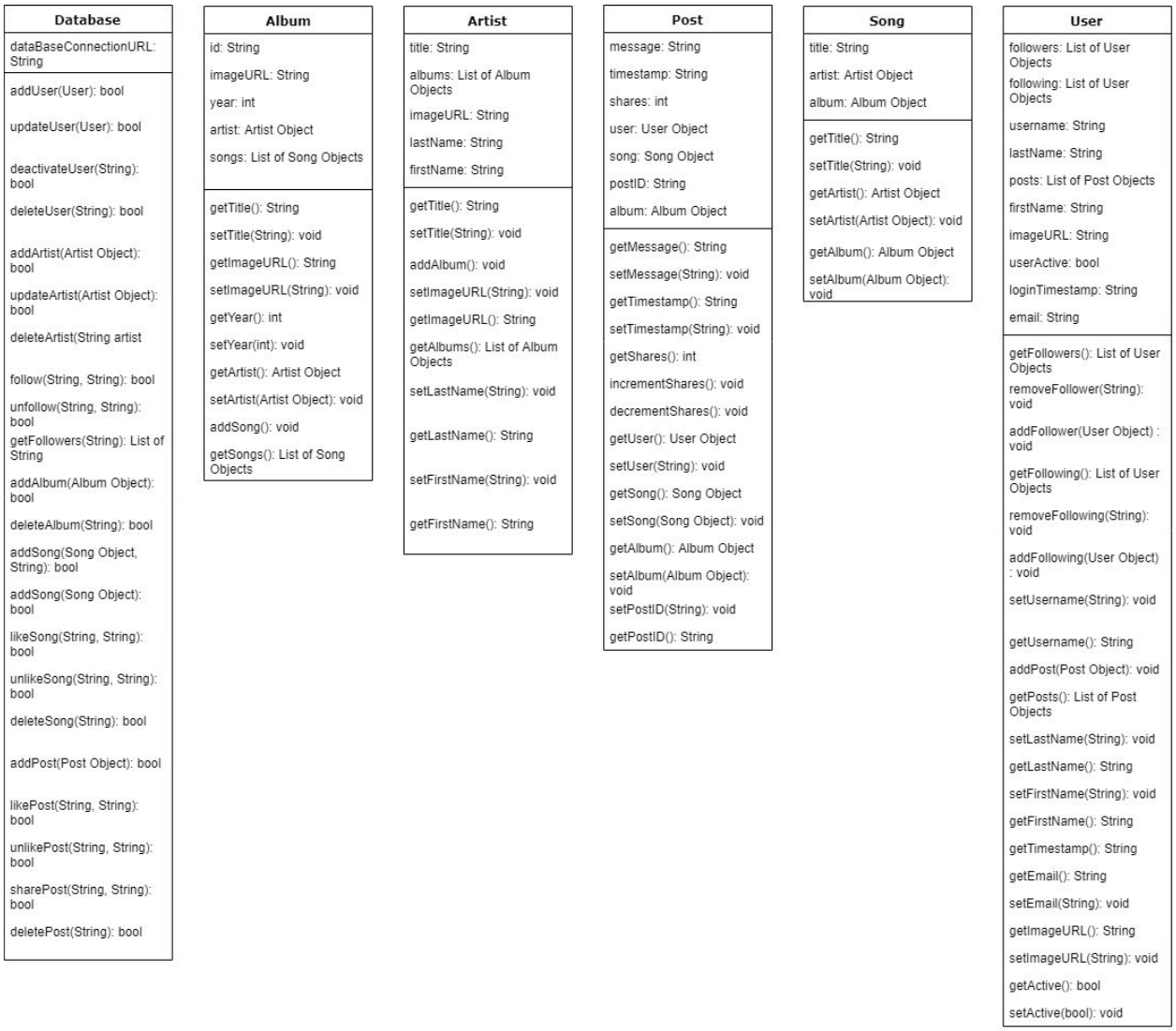
- `private boolean likePost(String post_id, String user_id):`
  This function will be responsible for tracking which users like a particular post in the database with "INSERT" statements to the "PostLike" table after a connection using the JDBC DriverManager is established. Insertion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "True" if addition is successful and "False" otherwise.

- `private boolean unlikePost(String post_id, String user_id):`
  This function will be responsible for deleting the relationship of the user liking the post permanently using the "DELETE" statement after a connection using the JDBC DriverManager is established. Deletion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "True" if relationship is successfully deleted and "False" otherwise.

- `private boolean sharePost(String post_id, String user_id):`
  This function will be responsible for tracking which users shared a particular post in the database with "INSERT" statements to the "PostShare" table and also adds the same post to "Post" table (but under current user) after a connection using the JDBC DriverManager is established. Insertion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "True" if addition is successful and "False" otherwise.

- `private boolean deletePost(String post_id):`
  This function will be responsible for deleting existing songs, the "AlbumSong" relationships,and all posts that included those songs inside the album from the database permanently using the "DELETE" statement after a connection using the JDBC DriverManager is established. Deletion will also be surrounded by Try, Catch blocks to ensure a minimum level of error handling. Function will return "True" if song is successfully deleted and "False" otherwise.

Back-end Class Structure diagram



Database

Schema

```
/*
-- CSCI 201L TrackChanges
-- Database
*/
DROP DATABASE IF EXISTS TrackChanges;
CREATE DATABASE TrackChanges;
USE TrackChanges;

/* --------------------------- Users Data --------------------------- */
/* Stores user's personal details */
CREATE TABLE User (
	user_id VARCHAR(100) PRIMARY KEY,
	user_login_timestamp TIMESTAMP NOT NULL,
	user_email VARCHAR(100) NOT NULL,
	user_firstname VARCHAR(100) NOT NULL,
	user_lastname VARCHAR(100) NOT NULL,
	user_username VARCHAR(100) NOT NULL,
	user_image_url VARCHAR(100) NOT NULL,
	user_is_active BOOL NOT NULL
);

/* Stores the artist's personal details */
CREATE TABLE Artist (
	artist_id VARCHAR(100) PRIMARY KEY,
	artist_firstname VARCHAR(100) NOT NULL,
	artist_lastname VARCHAR(100) NOT NULL
);

/* Stores followers / following relationship */
CREATE TABLE Follow (
	user_id VARCHAR(100) NOT NULL,
	follower_id VARCHAR(100) NOT NULL,
	PRIMARY KEY (user_id, follower_id),
	FOREIGN KEY Follow_user_id (user_id) REFERENCES User(user_id),
	FOREIGN KEY Follow_follower_id (follower_id) REFERENCES User(user_id)
);

/* -------------------------- Album and Song Data -------------------------- */
/* Stores the album id for identification */
CREATE TABLE Album (
	album_id VARCHAR(100) PRIMARY KEY,
	artist_id VARCHAR(100) NOT NULL,
	FOREIGN KEY Album_artist_id (artist_id) REFERENCES Artist(artist_id)
);

/* Stores the song id to uniquely identify the song */
CREATE TABLE Song (
	song_id VARCHAR(100) PRIMARY KEY,
	artist_id VARCHAR(100) NOT NULL,
	FOREIGN KEY Song_artist_id (artist_id) REFERENCES Artist(artist_id)
);
```

```
/* Stores which songs are included inside an album */
CREATE TABLE AlbumSong (
       album_id VARCHAR(100) NOT NULL,
       song_id VARCHAR(100) NOT NULL,
       PRIMARY KEY (album_id, song_id),
       FOREIGN KEY AlbumSong_album_id (album_id) REFERENCES Album(album_id),
       FOREIGN KEY AlbumSong_song_id (song_id) REFERENCES Song(song_id)
);

/* Tracks which users like which songs */
CREATE TABLE SongLike (
       song_id VARCHAR(100) NOT NULL,
       user_id VARCHAR(100) NOT NULL,
       PRIMARY KEY (song_id, user_id),
       FOREIGN KEY SongLike_song_id (song_id) REFERENCES Song(song_id),
       FOREIGN KEY SongLike_user_id (user_id) REFERENCES User(user_id)
);

/* --------------------------- Post Data --------------------------- */
/* Stores the content of each post and creator of post */
CREATE TABLE Post (
       post_id VARCHAR(100) PRIMARY KEY,
       post_timestamp TIMESTAMP NOT NULL,
       user_id VARCHAR(100) NOT NULL,
       post_message VARCHAR(500) NOT NULL,
       FOREIGN KEY Post_user_id (user_id) REFERENCES User(user_id)
);

/* Stores the number of shares of each post */
CREATE TABLE PostShare (
       post_id VARCHAR(100) NOT NULL,
       user_id VARCHAR(100) NOT NULL,
       PRIMARY KEY (post_id, user_id),
       FOREIGN KEY PostShare_post_id (post_id) REFERENCES Post(post_id),
       FOREIGN KEY PostShare_user_id (user_id) REFERENCES User(user_id)
);

/* Stores the number of likes of each post */
CREATE TABLE PostLike (
       post_id VARCHAR(100) NOT NULL,
       user_id VARCHAR(100) NOT NULL,
       PRIMARY KEY (post_id, user_id),
       FOREIGN KEY PostLike_post_id (post_id) REFERENCES Post(post_id),
       FOREIGN KEY PostLike_user_id (user_id) REFERENCES User(user_id)
);

/* Stores the albums that are included in each post */
CREATE TABLE PostAlbum (
       album_id VARCHAR(100) NOT NULL,
       post_id VARCHAR(100) NOT NULL,
       PRIMARY KEY (album_id, post_id),
       FOREIGN KEY PostAlbum_album_id (album_id) REFERENCES Album(album_id),
       FOREIGN KEY PostAlbum_post_id (post_id) REFERENCES Post(post_id)
);

/* Stores the songs that are included in each post */
```

```
CREATE TABLE PostSong (
       song_id VARCHAR(100) NOT NULL,
       post_id VARCHAR(100) NOT NULL,
       PRIMARY KEY (song_id, post_id),
       FOREIGN KEY PostSong_song_id (song_id) REFERENCES Song(song_id),
       FOREIGN KEY PostSong_post_id (post_id) REFERENCES Post(post_id)
);
```

CSCI 201L Group Project Testing Document: TrackChanges

Black Box Testing

Login

Route 1

- Login screen looks intuitive and as expected.
- Pressing the 'Connect' button should bring up a Safari View Controller to enter their Spotify account info, authorize TrackChanges, and then direct them to the feed screen.

Route 2

- Pressing the 'Guest' button should direct the user to a demo version of the feed screen.

Feed

- Feed screen looks intuitive and as described in the design document.
- If there is no content to show, the feed screen should display 'Nothing here to see! Try following some users first!'.
- Clicking the 'Compose' button on the top right bring up a screen where the user can write a post. A "Post" button should be displayed at the top right and a "Cancel" button should be displayed at the top left.
- If there is content to show, feed view displays the different content appropriately and chronologically.
    - Clicking the 'like' button likes the content and flips the button to 'unlike'.
    - Clicking the 'unlike' button unlikes the content and flips the button to 'like'.
- Clicking the 'Play' button on a song or an album will be reflected in the mini player at the bottom of the screen and the content should start playing.
- Dragging down the feed when at the very top will refresh the page to check for new content in the database.
- When the user scrolls down to the bottom of the page, increments of fifteen earlier posts will be appended to the end as the user scrolls.
- When the user is in Guest mode, 'Compose', 'Share', and 'Like' buttons don't show. In addition, the Profile and Favorites tab (recently added to our project) should not be displayed.
- Clicking 'Feed' tab on the navigation bar should not change anything.
- Clicking 'Discover' tab should take the user to the Discover tab.
- Clicking 'Profile' tab should take the user to the Profile tab.

Search and Discover

- A search bar should be displayed at the top.
- Under the search bar, trending artist, songs, and albums should be displayed.
- When a search word is typed and the return key is pressed, the trending display should be replaced with the search results.
    - The results will be divided in four sections: Songs, Albums, Artists, and Users.
        - Users section does not exist in Guest mode.
    - Tapping a song should play it and reflect it on the mini player at the bottom of the screen.

    - Tapping an artist, album, or user should go to its respective page.
- The navigation bar should behave as described above.

Profile

- This page does not exist in guest mode.
- Profile picture should be displayed. The number of followers and following users should be displayed under it.
- The user's submitted posts should be displayed under that in chronological order.
- Clicking the Following count displays a list of users that the user follows.
- Clicking the Followers count displays a list of the user's followers.
- The navigation bar should behave as described above.

Favorites

- This page does not exist in guest mode.
- This page does not exist in our original design. The purpose of this page is to show the user their liked songs and albums.
- The page is divided into two sections: songs and albums.
- The data for this page is pulled from our database based on what the user has previously liked.

Follow/Unfollow

- Following a user and refreshing the feed page should display the newly followed user's posts in the feed of the logged in user.
- Unfollowing a user and refreshing the feed should remove the posts of the unfollowed user from the feed of the logged in user.

Like/Unlike

- If a user likes a song or an album, the newly liked item should be displayed under the favorites tab under its respective section.

Posting

- When a user makes a post, the post should be displayed under the user's profile tab. If the user deletes the post, it should be removed from the profile tab.

White Box Testing

`private boolean addUser(User newUser):`
Test Case 1: Test the login functionality by specifying a user_id in the JSON User object parsed that does not exist in the database. The user should be taken back to the login page with a notification that the user has not signed up via connecting to “Spotify” as of yet, prompting them to “Sign up” instead or use the “Guest login”.

`private boolean updateUser(User user):`
Test Case 1: Select any user and update user’s follower count by 1 by having an outside user follow him. Should return true after SQL insertion
Test Case 2: Select any user and update user’s following count by 1 by having that user follow someone. Should return true after SQL insertion.

`private boolean deactivateUser(String user_id):`
Test Case 1: Select any user and use their user id to update `user_is_active` to `False` . Should return true after updating this in the database.

`private boolean deleteUser(String user_id):`
Test Case 1: Specify a user with their user_id in the Users table. Delete this user by signing out from the app. Should return true if user_id exists.
Test Case 2: Specify a user with their user_id that is not in the Users table. Delete this user by signing out from the app. Should return false if user_id does not exist.

`private boolean addArtist(Artist newArtist):`
Test Case 1: Read from JSON object parse the artist name that does not exist in the database. Should return true if artist does not exist in the database.
Test Case 2: Read from JSON object parse the artist name that does exist in the database. Should return false if artist already exists in the database.

`private boolean updateArtist(Artist artist):`
Test Case 1: Select artist from JSON object pass from the front end and update artist’s first name. Should return true if updated successfully.
Test Case 2: Select artist from JSON object pass from the front end and update artist’s last name. Should return true if updated successfully.

`private boolean deleteArtist(String artist_id):`
Test Case 1: Specify an artist with their artist_id in the Artist table. Delete this artist by deleting them from the user’s artist list. Should return true if the artist exists.
Test Case 1: Specify an artist with their artist_id in the Artist table. Delete this artist by deleting them from the user’s artist list. Should return false is the artist does not exist.

`private boolean follow(String user_id, String follower_id):`
Test Case 1: Specify a user with their user_id. If the user does not exist in the database, return false.
Test Case 2: Specify a user with their user_id. If the user does exist, but the follower_id specified does not exist in the database, return false.

Test Case 3: Specify a user with their user_id. If both the user_id and follower_id exists, update the user's following count. If update is successful, function should return true.

`private boolean unfollow(String user_id, String follower_id):`
Test Case 1: Specify a user with their user_id. If the user does not exist in the database, return false.
Test Case 2: Specify a user with their user_id. If the user does exist, but the follower_id specified does not exist in the database, return false.
Test Case 3: Specify a user with their user_id. If both the user_id and follower_id exists, update the user's following count. If update is successful, function should return true.

`private String[] getFollowers(String user_id):`
Test Case 1: Specify a `user_id` that does not exist in the `User` table. Function should return a null `String[]`.
Test Case 2: Specify a `user_id` that does exist in the `User` table, but `user_id` is not found in the `Follow` table, meaning that user is not following anyone and so check that `null` is returned.
Test Case 3: Specify a `user_id` that does exist in the `User` table. Function should return the corresponding `String[]` with the follower id's.

`private String[] getFollowing(String user_id):`
Test Case 1: Specify a `user_id` that does not exist in the `User` table and return a `null`.
Test Case 2: Specify a `user_id` that does exist in the `User` table, but `user_id` is not found in the `Follow` table, meaning that user has no followers and so check that `null` is returned.
Test Case 3: Specify a `user_id` that does exist in the `User` table and an array of `user_id` corresponding to the user's Following is returned.

`private boolean addAlbum(Album newAlbum):`
Test Case 1: Insert an `Album` object with an `album_id` that already exists in the database and check if no new album objects are added into the database.
Test Case 2: Insert an `Album` object that contains missing values, such as missing `album_id` or `artist_id`, and check if no new album objects are added into the database as well as return of a `False` value for the function.
Test Case 3: Insert an `Album` object that contains all required values, and check if the new album objects are added into the database as well as return of a `True` value for the function.
Test Case 4: Insert an `Album` object with an `album_id` that exists, but `artist_id` does not exist in the `Artist` table and check that a new song object is added into the `Album` table, and a new artist row with the `artist_id` specified is added into `Artist` table.

`private boolean deleteAlbum(String album_id):`
Test Case 1: Specify an `album_id` that does not exist in the `Album` table and check that no existing `Album` rows are deleted from the database as well as return of a `False` value for the function.
Test Case 2: Specify a `album_id` that does exist in the `Album` table and check that the `Album` row with the corresponding `album_id` is successfully deleted from the database as well as return of a `True` value for the function. Check also that all existing `AlbumSong` relationships with the specified `album_id` are deleted, all existing `Post` rows with the specified `album_id` are deleted as well, and return of a `True` value for the function.

`private boolean addSong(Song newSong):`
Test Case 1: Insert a newSong object with a `song_id` that already exists in the database and check if no new song objects are added into the database.

Test Case 2: Insert a newSong object that contains missing values, such as missing `song_id` or `artist_id`, and check if no new song objects are added into the database as well as return of a `False` value for the function.
Test Case 3: Insert a newSong object that contains all required values, and check if the new song objects are added into the database as well as return of a `True` value for the function.
Test Case 4: Insert a newSong object with a `song_id` that exists, but `artist_id` does not exist in the `Artist` table and check that a new song object is added into the `Song` table, and a new artist row with the `artist_id` specified is added into `Artist` table.

`private boolean likeSong(String song_id, String user_id):`
Test Case 1: Specify a `song_id` that does not exist in the `SongLike` table and check that no new `SongLike` rows are added into the database as well as return of a `False` value for the function.
Test Case 2: Specify a `user_id` that does not exist in the `SongLike` table and check that no new `SongLike` rows are added into the database as well as return of a `False` value for the function.
Test Case 3: Specify a `song_id` and a `user_id` that does not exist in the `SongLike` table and check that a new `SongLike` row is added into the database as well as return of a `True` value for the function.

`private boolean unlikeSong(String song_id, String user_id):`
Test Case 1: Specify a `song_id` that does not exist in the `SongLike` table and check that no existing `SongLike` rows are deleted from the database as well as return of a `False` value for the function.
Test Case 2: Specify a `user_id` that does not exist in the `SongLike` table and check that no existing `SongLike` rows are deleted from the database as well as return of a `False` value for the function.
Test Case 3: Specify a `song_id` and `user_id` that does exist in the `SongLike` table and check the `SongLike` row with the corresponding `song_id` and `user_id` is successfully added to the database as well as return of a `True` value for the function.

`private boolean deleteSong(String song_id):`
Test Case 1: Specify a `song_id` that does not exist in the `Song` table and check that no existing `Song` rows are deleted from the database as well as return of a `False` value for the function.
Test Case 2: Specify a `song_id` that does exist in the `Song` table and check that the `Song` row with the corresponding `song_id` is successfully deleted from the database as well as return of a `True` value for the function. Check also that all existing `AlbumSong` relationships with the specified `song_id` are deleted, all existing `Post` rows with the specified `song_id` are deleted as well, and return of a `True` value for the function.

`private boolean addPost(Post newPost):`
Test Case 1: Specify a `user_id` in the `Post` object that does not exist in the `User` table and check that no new `Post` rows are added to the database as well as return of a `False` value for the function.
Test Case 2: Specify a `user_id` that does exist in the `User` table and check that the `Post` row with the corresponding `user_id` and `post_message` is successfully added to the database as well as return of a `True` value for the function.
Test Case 3: Specify a `user_id` or `song_id` that does not exist in the `User` and `Song` table respectively and check that no new `Post` rows are added to the database as well as return of a `False` value for the function.
Test Case 4: Specify a `user_id` and `song_id` that both exist in the `User` and `Song` table respectively and check that a new `Post` row is added to the `Post` table and a new `PostSong` row is added to the `PostSong` table as well as return of a `True` value for the function.
Test Case 5: Specify a `user_id` or `album_id` that does not exist in the `User` and `Song` table respectively and check that no new `Post` rows are added to the database as well as return of a `False` value for the function.

Test Case 6: Specify a `user_id` and `album_id` that both exist in the `User` and `Album` table respectively and check that a new `Post` row is added to the `Post` table and a new `PostAlbum` row is added to the `PostAlbum` table as well as return of a `True` value for the function.

`private Post[] getPosts(String user_id):`
Test Case 1: Specify a `user_id` that does not exist in the `User` table and check that no `Post` objects are returned for the function, hence return `null`.
Test Case 2: Specify a `user_id` that does exist in the `User` table and check that the correct `Post` rows are returned after the sql query as well as return of correct `Post` objects for the function that correspond to the `user_id`.

`private Post[] getFeed(String user_id):`
Test Case 1: Specify a `user_id` that doesn't exist. An appropriate exception should be thrown.
Test Case 2: Specify a `user_id` that exists. If no posts are found, check that `null` is returned and otherwise an array of Post objects is returned. If an array of Post objects is returned, check that the array of Post objects is chronologically ascending and the correct posts are inside.

`private boolean likePost(String post_id, String user_id):`
Test Case 1: Specify a `user_id` that doesn't exist and a `post_id` that doesn't exist. An appropriate exception should be thrown.
Test Case 2: Specify a `user_id` that doesn't exist and a `post_id` that exists. An appropriate exception should be thrown.
Test Case 3: Specify a `user_id` that exists and a `post_id` that doesn't exist. An appropriate exception should be thrown.
Test Case 4: Specify a `user_id` that exists and a `post_id` that exists. Check that the a new row is added for the correct user to the "`PostLike`" table. Check that the function returns true.

`private boolean unlikePost(String post_id, String user_id):`
Test Case 1: Specify a `user_id` that doesn't exist and a `post_id` that doesn't exist. An appropriate exception should be thrown.
Test Case 2: Specify a `user_id` that doesn't exist and a `post_id` that exists. An appropriate exception should be thrown.
Test Case 3: Specify a `user_id` that exists and a `post_id` that doesn't exist. An appropriate exception should be thrown.
Test Case 4: Specify a `user_id` that exists and a `post_id` that exists. Check that the correct row is deleted from the "`PostLike`" table. Check that the function returns true.

`private boolean sharePost(String post_id, String user_id):`
Test Case 1: Specify a `user_id` that doesn't exist and a `post_id` that doesn't exist. An appropriate exception should be thrown.
Test Case 2: Specify a `user_id` that doesn't exist and a `post_id` that exists. An appropriate exception should be thrown.
Test Case 3: Specify a `user_id` that exists and a `post_id` that doesn't exist. An appropriate exception should be thrown.
Test Case 4: Specify a `user_id` that exists and a `post_id` that exists. Check that the a new row is added for the correct user to the "`PostShare`" table. Check that the function returns true.

`private boolean deletePost(String post_id):`

Test Case 1: Specify a `user_id` that doesn't exist and a `post_id` that doesn't exist. An appropriate exception should be thrown.
Test Case 2: Specify a `user_id` that doesn't exist and a `post_id` that exists. An appropriate exception should be thrown.
Test Case 3: Specify a `user_id` that exists and a `post_id` that doesn't exist. An appropriate exception should be thrown.
Test Case 4: Specify a `user_id` that exists and a `post_id` that exists. Check that the correct row is deleted from the "`PostShare`" table. Check that the function returns true.

Stress Testing

Test Case 1: Test with multiple users simultaneously logging in. The front-end should have low latency and the backend server should be able to handle the creation of several threads, each verifying the users' login details with the database.

Test Case 2: Test having multiple users scrolling down their feed and querying the database for the earlier posts on their feed. Latency is mostly dependent on how fast the back-end is able to fulfill the query while having multiple threads requesting data.

Test Case 3: Test with multiple users creating posts simultaneously and check if the backend server handles the requests appropriately by using several threads and have the appropriate locking to ensure no concurrency issues.

Test Case 4: Test having multiple users search at the same time. This requires the database to return a number of users to be displayed on the search page, again, check if the backend server handles the requests appropriately by using several threads and have the appropriate locking to ensure no concurrency issues.

Test Case 5: Many users following and unfollowing a user around the same time, again, check if the backend server handles the requests appropriately by using several threads and have the appropriate locking to ensure no concurrency issues.

Test Case 6: Test many users liking a post around the same time. Front-end should have low latency and backend will handle the creation of multiple threads for efficient insertion of liked posts into database. And again, check if the backend server handles the requests appropriately by using several threads and have the appropriate locking to ensure no concurrency issues.

Test Case 7: Test many users sharing a post around the same time. Front-end should have low latency and backend will handle the creation of multiple threads for efficient insertion of shared posts into database. And again, check if the backend server handles the requests appropriately by using several threads and have the appropriate locking to ensure no concurrency issues.

Test Case 8: Test that when the connection to the backend server is lost, the front-end will display the appropriate page of "connection to server is lost".

Test Case 9: Test when a user logs in to the application with two or more distinct devices. Screens on all devices should be updated accordingly and be synchronized to the same session and not display any outdated information.

CSCI 201L Group Project Deployment Document: TrackChanges

Frontend:

Execution Instructions:

1. Download and install XCode.
2. Open TrackChanges.xcworkspace.
3. Make sure the server is on, and build and run the program

Backend:

Server (Database)

Installation of this product is supported on the following operation systems and versions:

- macOS Sierra
- macOS High Sierra
- macOS Mojave
- Windows Server 2016
- Windows Server 2019

Roles, Features, and Packages

The following software packages must be installed on the operating system prior to installation of the software:

- MySQL Community Server 8.0.13
- MySQL Workbench 8.0.13

SQL Server Configuration

Ensure that The MySQL Instance is running

Authentication

- Mixed mode authentication should be enabled
- Add new connection to MySQL server with the following configuration:
    - Hostname: localhost
    - Port: 3306
    - Username: root
    - Password: root

SQL Server Network Configuration

- TCP/IP should be enabled

Execution Instructions:

1. Run the Java Project 'TrackChangesBackend' on the TomCat server in Eclipse with the port number that you choose.
2. Open and run the SQL file 'TrackChanges.sql' in MySQL Workbench.
3. On line 43 of the class Application.java, set your username and password for MySQL (currently set to root).
4. Run unit tests on all functions in Application.java