Team 8 Design Document: Kadence

Jackson Rosenberg, Avery Schaefer, Nathan Simon, Colston Streit, Raymond Xie
rosenbe7@purdue.edu, schaef35@purdue.edu, simon70@purdue.edu, cstreit@purdue.edu, xie328@purdue.edu

Table of Contents

Purpose

Introduction

Picking music is hard. Whether you're trying to decide on a soundtrack to your workout, discover new bands in your hometown, or just pick something that fits your current mood, finding the perfect playlist is incredibly difficult with just Spotify or Apple Music. Kadence is a playlist creation app designed to bridge that gap, allowing for users to create the perfect soundtrack for whatever they're doing. Kadence will feature multiple revolutionary modes, including a fitness mode that selects songs based on the user's heart rate data from a paired smartwatch, a mood mode that generates a playlist based on how the user is feeling, and a local artist mode that finds new and undiscovered bands right in the user's backyard.

Project Objectives

- Develop a mobile app using Next.js that connects to a user's Spotify or Apple Music account and smart fitness device.
- Create playlist generation "modes", whether based on real-time biometric data (i.e. heart rate) or based on moods to either calm or energize the user.
- Additional mode features include an interval mode (alternating between upbeat and mellow music on specified intervals) and a location-based mode to promote local artists.
- Use recommendation algorithms to mesh user's music taste and genre preferences with a focus on specific playlist types and discovery.
- Design a database to store settings and save preferences for users to be able to sign out and sign back in.

Non-Functional Requirements

- As a user, I would like the app to be cross platform (iOS and Android).
- As a developer, I would like user profile data to be safely and securely stored.
- As a developer, I would like user health / fitness data to be safely and securely stored.
- As a developer, I would like informative error messages to be shown both on the frontend of the app and in the developer console.
- As a developer, I would like the app to be properly documented with an informative README.
- As a user, I would like the app to have an intuitive and consistent user interface.
- As a developer, I would like the repository to have a CI/CD pipeline.

Usability / Performance
Our app should have a clean and responsive UI on both iOS and Android devices to ensure that users can easily understand what our app does and how to navigate it. This includes making it clear visually whenever the app is waiting on a response from the backend so that the user doesn't become discouraged by a lack of feedback. Using Next for cross-platform component building and reuse should make these goals easier to achieve. Finally, these response times should not be so long that the user becomes frustrated - we aim to keep them under 3 seconds wherever possible.

Scalability
Our app should at least be able to handle all of CS 40700 (students and staff) at one time, as this will allow us to feasibly test our product with a reasonably large audience if need be. The low-cost tier of Azure should definitely allow for this and should also fit within our $200 credit. Additionally, we shouldn't need to worry about any specific number of API calls since users of our app will also be users of their chosen music or fitness platforms and we will be using their accounts to get songs or health data. Finally, we plan to write our code in a clean, sustainable way that allows for further extension (if desired) after the conclusion of the semester.

Security
Given that much of our app involves using user fitness or location data as an input to our playlist generation algorithm (depending on which mode is set), it is essential that we keep this data secure. We will do this by ensuring that anytime we make a call to a third-party API, no personal data is sent without first being anonymized. Our backend server will also implement HTTPS to ensure traffic is secure from unwanted eavesdroppers. Additionally, some of the external APIs we will be using include a state parameter in requests, protecting against CSRF attacks. Internally, user profile passwords will be stored in a hashed form to prevent anybody other than the user from seeing their password, including us.

Hosting / Deployment
We plan on using Azure to host our backend server and will likely use Docker or some other compartmentalization tool to simplify the process of downloading dependencies. Additionally, we plan to eventually have a working CI/CD pipeline where our code is only deployed if appropriate tests are passed. The front end of our app will be deployed as an app for iOS and Android. For now, we will not be including a web version.

Functional Requirements

Accounts and Profiles

- As a user, I would like to create an account.
- As a user, I would like to login and logout of my account.
- As a user, I would like to be able to delete my account.
- As a user, I would like to be able to reset my password (using email for recovery if time allows).
- As a user, I would like to view my profile.
- As a user, I would like to have and edit a username.
- As a user, I would like to have and display a profile picture.
- As a user, I would like to update my profile.
- As a user, I would like to display a short bio about myself and my music taste.

Device and Music Connections

- As a user, I would like to link a music platform.
- As a user, I would like to be able to update my music platform.
- As a user, I would like to be able to remove a music platform.
- As a user, I would like to see what music platforms are currently linked.
- As a user, I would like to be able to update my default preferences.
- As a user, I would like to connect to my fitness device.
- As a user, I would like to be able to remove my connected device.
- As a user, I would like to view my connected devices.
- As a user, I would like to be able to choose a default device.
- As a user, I would like to be able to allow / disallow the app to access my fitness data.
- As a user, I would like to be able to rename a connected device.

Song Selection / Playlist Preferences

- As a user, I would like to select the genre of the playlist.
- As a user, I would like to create default preferences on genres, artists, etc. that affect playlist generation.
- As a user, I would like to blacklist songs and artists from my playlists.
- As a user, I would like to be able to create a preference between lyrical and instrumental songs.
- As a user, I would like to choose whether explicit songs are added to the playlist or not.
- As a user, I would like to set a lyric language preference.
- As a user, I would like to set a minimum and maximum song length.
- As a user, I would like to set minimum and maximum time limits on the playlist being generated.

Playlist / Mode Functionality

- As a user, I would like to be able to start playlist generation.
- As a user, I would like to specify whether or not I want to save the playlist right away or allow Kadence to build a queue of songs to listen to first.
- As a user, I would like to be able to provide feedback about the generated playlist.
- As a user, I would like to delete a playlist after initial generation and automatically regenerate / reshuffle.
- As a user, I would like to be able to select different playlist creation modes.
- As a user, I would like to set / edit the short and long interval times in interval mode.
- As a user, I would like to select ramp up or ramp down settings in the fitness mode.
- As a user, I would like to change my default location to explore different local music scenes.
- As a user, I would like to skip a song in a generated playlist / queue if I do not enjoy it.
- As a user, I would like to blacklist additional songs if I strongly dislike them during playback.
- As a user, I would like to save a playlist generated to either Spotify or Apple Music profiles after listening to it.
- As a user, I would like to customize the name of the playlist before I save it
- As a user, I would like to have the ability to alter the playlist (remove songs) after it has been generated (if time permits).

Social

- As a user, I would like to view other people's profile pages.
- As a user, I would like to hide my own profile page from other users by declaring my profile private.
- As a user, I would like to ensure that all of my sensitive fitness information remains private from all users, including friends.
- As a user, I would like other people to find and view my profile by declaring my profile public.
- As a user, I would like to search for other users.
- As a user, I would like to send friend requests to view private profiles.
- As a user, I would like to view and listen to playlists that public users or friends generate.
- As a user, I would like to view what song that a public user or friend is listening to in real-time (if time permits).

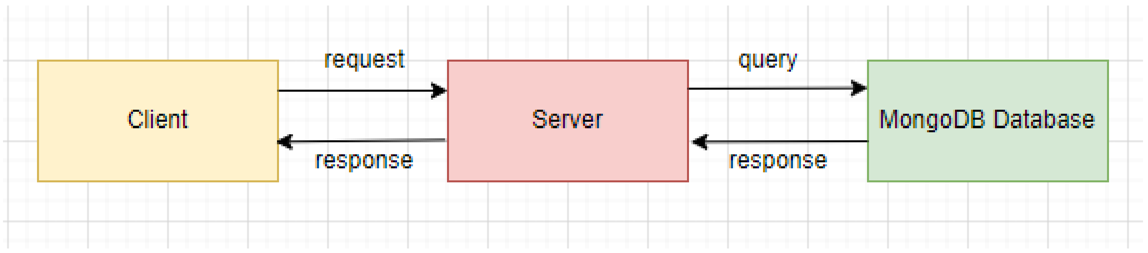
Activity Log (if time permits)

- As a user, I would like to see an activity log of all of my own actions (if time permits).
- As a user, I would like to see an activity log of new playlists my friends are generating and saving to their profiles (if time permits).
- As a user, I would like to see statistics regarding my listening history in each mode and about my playlists (if time permits).

Design Outline

High-Level Overview

At a high level, our application aims to combine data from third-party music and fitness platforms in order to spice up playlist generation. Because each user will have their own account, we will be using a database to store their account information and playlist preferences. Finally, we will have a client which will display the app and make requests to the server, which will respond to requests after fetching relevant data from the database or the third-party music and fitness APIs.



High-level overview of the client-server architecture

System Components

Below are the main components that will compose our app explained in more detail. Our app is fundamentally using the client-server-third party architecture.

Client

Our client will consist of a Next.js application run on a user's mobile device. It will make requests to the server in order to do things like getting or setting the user's profile information, getting or setting the user's playlist preferences, or even just to play a given playlist or skip the current song. This will likely be the "lightest" component of our application, as the server will be doing nearly all of the heavy lifting.

Server

As stated previously, our Node/Express server will truly be doing the heavy lifting of the application. It will take in requests from the client (e.g., to begin playlist generation) before fetching relevant data from / sending requests to the client's selected music and fitness platforms using their APIs. It will then combine that data / perform the actions associated with the client's request before sending a status code back to the client. Additionally, if the client sends a request to modify the user's profile information or preferences, for example, then the server will interface with the database so that the client's modifications to the user's data persist.

Database

Our MongoDB database will be used to store nearly all of the data used in our application, including the user's account information, profile information, and playlist preference information.

The server will interact with the database to obtain this information upon relevant requests from the client.

Music APIs

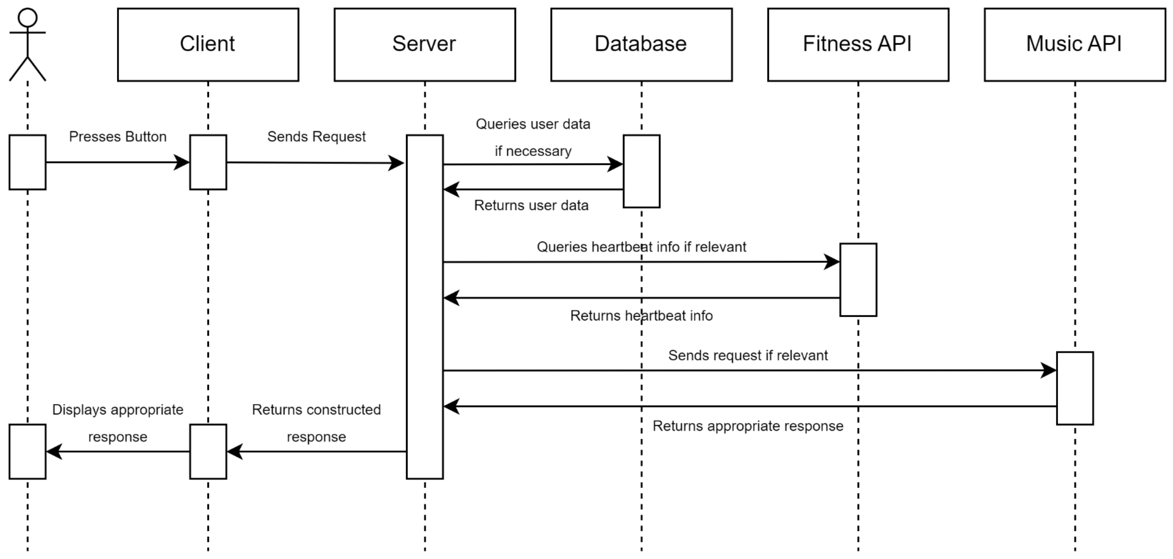
Music APIs (e.g., Spotify or Apple Music) will be used as part of every major functionality of our application. When the server receives a request from the client to create a playlist using the specified preferences, for example, it will repeatedly query the music API in order to find songs that match the user's preferences (taking fitness information into account if relevant). Finally, the server will either add all of these songs to the queue or construct a playlist using the music API before sending back the song collection information to the client to be rendered for the user.

Fitness APIs

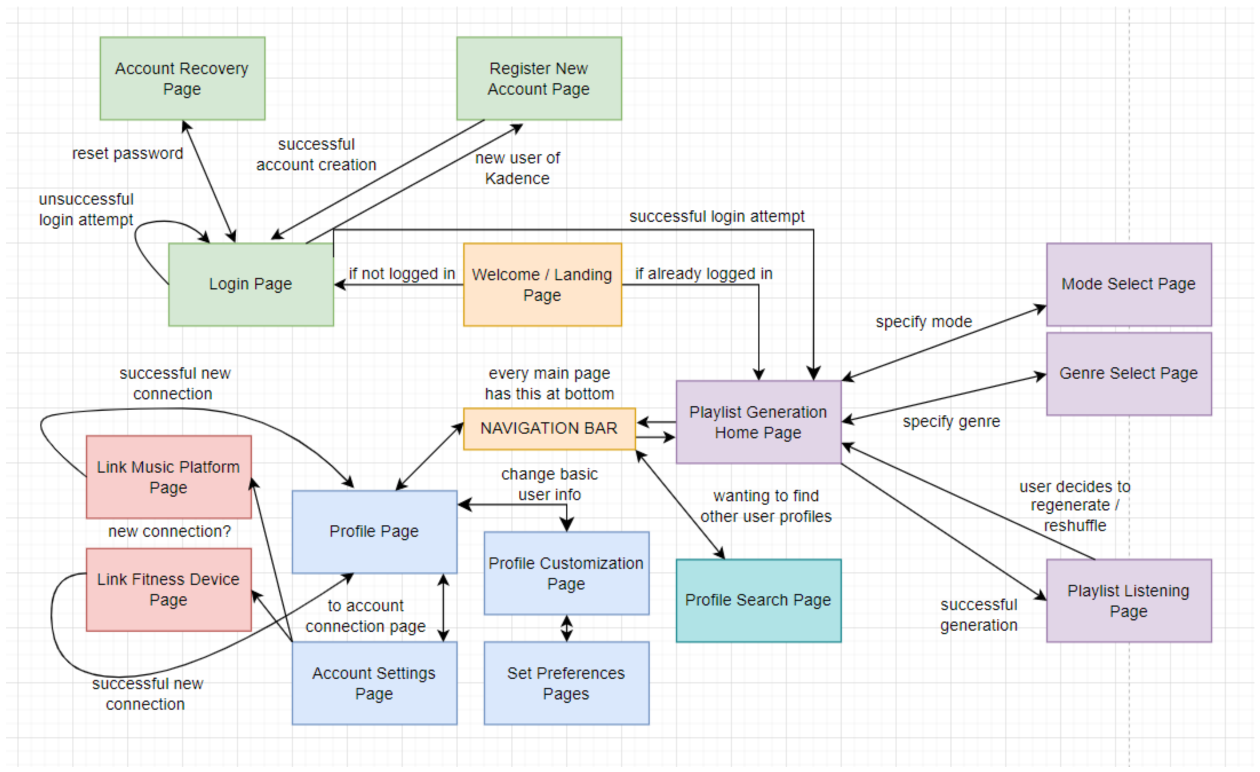
Finally, fitness APIs (e.g., Fitbit) will be used to query heart rate information from the user, who presumably is wearing a device allowing data access through the API. This will be used when the user is running the app in fitness mode, where the next song is chosen based on it both matching the user's playlist preferences and having the same tempo as the user's heart rate. The server will query this information from the fitness API every time a new song needs to be selected while running fitness mode.

Sequence of Events Overview

To give an overview of the information above, for any given request from the client, the server may query the database for the user's information, may query the fitness API to get the user's heart rate, and may query the music API in order to actually carry out the user's request. It will then combine all of the relevant information requested and send it and a status code back to the client. Below is a diagram that details this process for an arbitrary request made by the user:


Client
Server
Database
Fitness API
Music API
Presses Button
Sends Request
Queries user data if necessary
Returns user data
Queries heartbeat info if relevant
Returns heartbeat info
Sends request if relevant
Returns appropriate response
Displays appropriate response
Returns constructed response

Activity / Navigation Flow Diagram



The above diagram represents a high-level overview of all possible navigation paths that a user could take when opening and using the app. The majority of the activities will be organized by a navigation bar found at the bottom of every major page that allows the user to switch between different components of the application seamlessly and quickly. Each major section (accessible via the navigation bar) is designated with a different color as seen in the diagram. Here is a simple breakdown of the color-coding scheme:

- Red: Linking music and fitness devices
- Blue: Profile viewing, preference settings, profile customization
- Teal: Search features
- Purple: Playlist generation
- Green: Authentication
- Orange: General navigation / misc.

Design Issues

Non-Functional Issues

1. **What is the best framework to use for the frontend?**
   Option 1: Flutter
   Option 2: React.js
   Option 3: Next.js

   Justification: Our application will be only available on the mobile platforms of iOS and Android (see Non-Functional Design Issue #5), so we need a mobile-friendly development framework for designing and implementing the frontend. When researching different options, we felt that Next was the best option for us in terms of our team's experience with it in the past, as it is basically an extension upon React. Next is great for creating very consistent-looking user interfaces using a simple component building system and has the major benefit of being cross-platform with mobile apps. Also, upon doing research about the fitness and streaming APIs we plan to integrate, Next supports the ones we want whereas Flutter, Google's cross-platform mobile framework, does not.

2. **What is the best available option to host our application?**
   Option 1: Google Cloud
   Option 2: A traditional server owned by one of our group members
   Option 3: Microsoft Azure
   Option 4: Amazon Web Services

   Justification: According to the GTA overseeing our project's development, the department is allowing every group a hosting budget of 200 dollars on Microsoft Azure for the semester, which should be plenty for the amount of users we plan on being able to service for our testing and demonstration phases. While Google Cloud and AWS have student discounts and even some free options, it is hard to choose them over Azure when we already have the funds provided for us to get the project off the ground. Also, we will not be using a server owned by any of our members as none of us have the technological specs nor the experience to properly set one up for use.

3. **Which database should we implement to store user information?**
   Option 1: PostgreSQL
   Option 2: MongoDB
   Option 3: Oracle RDBMS

   Justification: Evaluating our group's experience in past projects and internships, we realized that our backend team will be the most comfortable designing and engineering the database using MongoDB. In addition to this, MongoDB is document-oriented (JSON) whereas the other two options are more traditional relational database services that are object-oriented and work in some sort of SQL. Although the scripting for queries is unique to MongoDB, we think working with JSON documents will be slightly easier in the overall development of the project as we plan on using the MERN tech stack (MongoDB, Express, React, and Node).

4. **What type of architecture should we use to develop the application?**
   Option 1: Microservices
   Option 2: Client-Server
   Option 3: Model-View-Controller

   Justification: After discussing as a team and examining our overall design schema, we feel as though our project most closely fits into the client-server architecture style. Each of our users will act as a client sending requests to a server that will handle the database management, algorithm for playlist creation, and sending responses back to the client of the playlists they generate. The project requires a lot of computation on the backend, so this architecture will be best for making sure that very little of it will be done on the client-side. Finally, this choice allows the five of us to split into two teams for development once the sprints begin so that both can be working simultaneously without conflict.

5. **Which platforms should we have our application support?**
   Option 1: Web
   Option 2: Mobile (iOS)
   Option 3: Mobile (Android)

   Justification: For both Spotify and Apple Music (the two major music streaming platforms we are choosing to support), a majority of the user base listens to music via either their phone or some sort of tablet device. In order to create the best possible supplemental experience to the streaming platforms, we want the users to be able to have both Kadence and the platform of their choosing in one easy-to-access and easy-to-switch-between place. Because of this design choice, we can focus or UI renderings for common mobile screen resolutions without the tediousness of having to make different mockups and designs for mobile and desktop browsing. Also, because we plan to use Next.js for the development, we do not need to worry about any platform-specific development and can simply have one codebase for both.

6. **How should we deploy and test new code changes?**
   Option 1: Manual testing sequences and building
   Option 2: Continuous integration and continuous deployment

   Justification: We plan on implementing Docker for compartmentalization so that every time a code change is committed, we can automatically build and run a test sequence on the code since this will ultimately help us save time debugging and ensuring the application is working as intended. Although this adds some work hours to the first sprint, we believe the time saved avoiding manually building and testing will be worth the extra effort. CI / CD (as described above) is a common practice in industry as well, so getting more exposure and practice with the technologies will be beneficial to all of our careers.

Functional Issues

1. **What music platform(s) to support?**
   Option 1: Apple Music
   Option 2: Spotify
   Option 3: YouTube Music
   Option 4: Amazon Music

   Justification: In essence, our choice here came down to two factors: market share and richness of API. Spotify and Apple Music make up the vast majority of the music streaming market, which means supporting just these two platforms makes our app usable to most people. We wanted an API that allowed us to interact with music playback and playlist creation features, and both chosen APIs provide this. Furthermore, Spotify provides a rich set of audio analysis metadata about songs/artists, including tempo, valence, level of energy, etc which are pertinent to our algorithm.

2. **What fitness devices to support?**
   Option 1: Garmin
   Option 2: Fitbit
   Option 3: Apple Watch
   Option 4: Samsung

   Justification: FitBit provides a public RESTful web API to retrieve fitness data from their devices. We chose this API because of the simplicity of integrating it into our Node.js backend via HTTP requests.

   In contrast, access to Garmin's Health API requires submitting an application to become an "approved business developer", but since we are not a company (yet), we decided to go with a more openly accessible API. Collecting data from Apple Watches requires the integration of HealthKit, which is Apple's Objective-C/Swift-based framework for health and fitness data. Our team has minimal experience with these languages, so we chose not to support Apple Watches. Similarly, Samsung's Android Health Connect API would clash with our preference for the Next.js / Node.js tech stack.

3. **Does a user need to create an account to use our app?**
   Option 1: No account required
   Option 2: User logs in through music platform account
   Option 3: Kadence account creation required; music platform accounts are connected

   Justification: Having to maintain accounts for every single service represents a big hassle to most people. Although not requiring an account to use Kadence would be convenient for new users, it would limit the depth of our app's features. We realized that to provide a personalized experience using Kadence, we would have to store preference data (such as favorite genres, artists, disliked songs, etc) associated with each user. Spotify and Apple Music's APIs do not provide this data, which meant that signing in through a music platform account alone would not suffice. Deciding to require an account also opens the door for us to implement social features as well.

4. **How are user preferences for playlist generation collected?**
   Option 1: Adaptively as user listens
   Option 2: Configuration (with defaults)
   Option 3: Survey upon first time opening app

   Justification: We needed to decide when and where to collect info about a user's music preferences, as this would shape how repeated interaction with the playlist generation feature would work. An adaptive approach where the user provides feedback song-by-song would allow them to curate playlist generation at a fine-grained level, but we realized that the initial time needed to teach the app your preferences could be a tedious user experience. We then thought of "first time launch" experiences where an app will take the user through a survey to discover their preferences, but we realized that users are likely to have different preferences for things like genre or mood every time they create a playlist. Thus, we settled with the most simple option of giving the user the ability to configure their preference settings before they generate their playlists.

5. **What interactions exist between users of the app?**
   Option 1: None
   Option 2: Social features
   Option 3: Competitive modes (e.g. who listened to the most music, who exercised most)

   Justification: At first we weren't sure there needed to be any interaction between users of our app. Upon reflecting on our group's poor experience using Spotify's barebone social features however, we decided it would be interesting if users could add other users as friends to see their playlists. Taking inspiration from letterboxd's activity feed, we also thought it would be engaging for users to see what playlists and songs friends have been listening to in a single unified view.

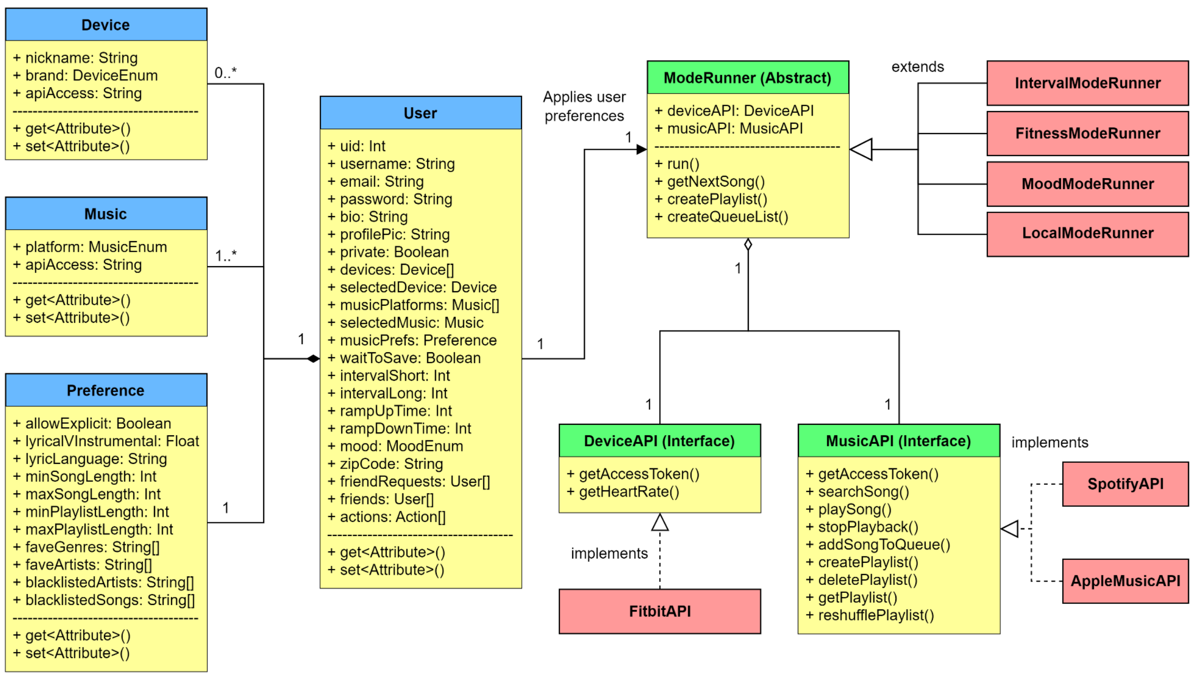
6. **Does the user listen to music in-app or on their music platform?**
   Option 1: Playlist generation done all at once, then user listens in music platform
   Option 2: Playlist generation done on the fly, user listens in-app
   Option 3: Playlist generation done on the fly, user listens in music platform

   Justification: Initially we envisioned that after selecting a mode and hitting "generate", an entire playlist would be generated all at once. However, we realized this approach would limit our ability to incorporate user feedback as they listened. If a user dislikes the current song, we want to prevent similar songs from entering the queue.

   What if a user doesn't know exactly how long they want to work out or study? This uncertainty over playlist length is also handled much better by generating songs on the fly. It followed that music listening had to be within our app instead of in the music platform's app, because otherwise there would be no simple way for our app to collect user feedback information or track the songs that had been heard already.

Design Details

Class Design



Above is our class diagram, indicating the major classes we expect to have as part of our application. We could have included all of the algorithmic Preference fields as part of the User class, but we opted to separate them purely to make the diagram cleaner. Similarly, the Music and Device classes just contain information related to specific user fitness devices or user music platforms and were made separate for clarity since users can link multiple devices and music platforms. Note that every user needs at least one music platform since not having a music platform connected makes our app pointless. Additionally, users are not required to link a fitness device if they don't want to use any of the heartbeat-detection features of our app (though that is our primary differentiating feature).

We have multiple interfaces / abstract classes relating to actual functionality, including ModeRunner which will run the song selection / playlist creation process in a given mode, DeviceAPI which will wrap around the various fitness APIs we might use, and MusicAPI which will wrap around the various music platform APIs we might use. Currently we aim to have four ModeRunner implementations (one each for the interval, fitness, mood, and local modes), one DeviceAPI implementation (for Fitbit devices), and two MusicAPI implementations (Spotify and Apple Music). However, this is subject to change.

Descriptions of Classes and Interaction Between Classes

All of the information that is actually used in the app will be stored in the User class on a per-user basis. We will then have various other helper classes that will be used to wrap around the various music platform APIs and fitness API. Finally, we will have a main runner class that combines these to run the playlist generation algorithm for a given mode. Below, we've gone into more detail regarding each class.

User

This is the User class, which will contain all user preferences and account information. Note that the fields of the Preference, Music, and Device classes from the diagram are contained within the list below. This is simply because all of this information will be stored in our MongoDB database on a per-user basis.

- Each user will have a unique identifier (uid).
- Each user will have login credentials, including a username, an email address, and a password.
- Each user can have a bio for their profile.
- Each user can optionally choose a profile picture to display on their profile.
- Each user can choose to make their profile public or private.
- Each user can have fitness devices paired with their account, each with
    - A nickname, so the user can easily identify it.
    - A brand, to declare what type of device it is.
    - An API authentication token of some kind so we can access data,
- Each user will declare a specific device to be the default device.
- Each user can have music platforms paired with their account, each with
    - A platform name so we know which API to use.
    - An API authentication token of some kind so we can access data.
- Each user will declare a specific music platform to be played from.
- Each user will have default preferences for how songs are chosen, including
    - Whether explicit songs can be included.
    - Whether lyrical or instrumental songs are preferred (or don't care). This will be represented as a float in the range [0, 1] where 0 = complete preference for lyrical over instrumental and 1 = complete preference for instrumental over lyrical.
    - What language the lyrics of songs should be (or don't care).
    - A minimum and maximum song length (or don't care).
    - A minimum and maximum playlist length (or don't care).
    - A list of favorite genres.
    - A list of favorite artists.
    - A list of blacklisted artists.
    - A list of blacklisted songs.
- Each user can choose whether, when they begin playlist generation, the playlist should immediately be saved to their account or songs should be added to the queue (with the option to save the playlist to their account later).

- Each user can store their default settings for each mode, including
    - Short and long interval times for interval mode.
    - Optional ramp up and ramp down timing settings for fitness mode.
    - Current mood for mood mode (from a set number of options).
    - Zip code for local mode.
- Each user may have multiple incoming friend requests to respond to.
- Each user will have a list of friends, who are other users.
- (If time allows) For each user, a list of their actions (playlist creations, etc.) will be stored and be viewable as an activity log displayed on their profile.

DeviceAPI (Interface)
An implementation of this interface will essentially serve as a wrapper around a given fitness API (e.g., Fitbit) in order to obtain wellness information from the user.

- Each implementation will have a method to get/refresh our access token.
- Each implementation will have a method to get the user's current heart rate.

MusicAPI (Interface)
An implementation of this interface will essentially serve as a wrapper around a given music platform API (e.g., Spotify or Apple Music) in order to play songs and create/delete playlists on the user's account.

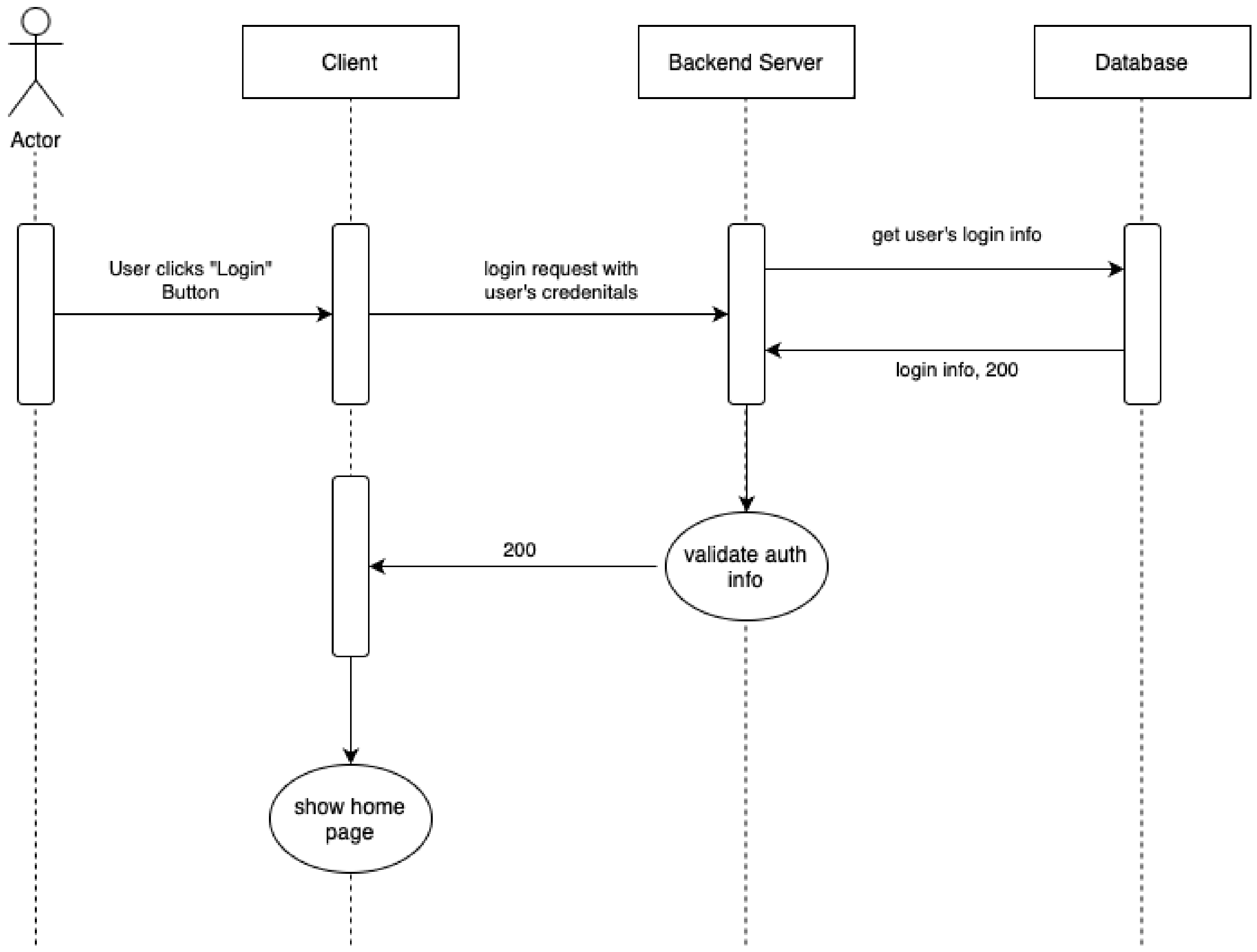
- Each implementation will have a method to get/refresh our access token.
- Each implementation will have a method to search for a song in various ways:
    - By artist
    - By song title
    - By genre
    - By tempo (for fitness mode)
- Each implementation will have a method to play a song from a given user's music account.
- Each implementation will have a method to stop playback on a given user's music account.
- Each implementation will have a method to add a song to a given user's current song queue.
- Each implementation will have a method to create a playlist with a given name on a given user's music account.
- Each implementation will have a method to delete a playlist created by Kadence on a given user's music account.
- Each implementation will have a method to get a playlist created by Kadence from a given user's music account.
- Each implementation will have a method to reshuffle a playlist created by Kadence on a given user's music account.
- Each implementation will have a method to obtain the current time position of the song that is currently playing on the user's music account.

ModeRunner (Abstract Class)
This will essentially be the runner class of the song selection / playlist creation part of our application. Each implementation of this interface (one per mode) will take in a DeviceAPI and a MusicAPI and will, based on the user's preferences as well as the functionality of the mode that the implementation represents, return the correct song(s) to play.

- Each ModeRunner will take in a DeviceAPI implementation to connect to the user's fitness device.
- Each ModeRunner will take in a MusicAPI implementation to connect to the user's chosen music platform.
- Each implementation will have a main method that runs the app in the given mode.
- Each implementation will have a method to select the next song based on the logistics of the mode (for example, the fitness mode will take into consideration the user's heart rate as well as the ramp up and ramp down settings specified by the user).
- Each implementation will have a method to create a playlist based on the mode and user preferences.
- Each implementation will have a method to get multiple songs and add them to the queue based on the mode and user preferences. (This is in case the user doesn't want to save the generated songs to a playlist right away.)

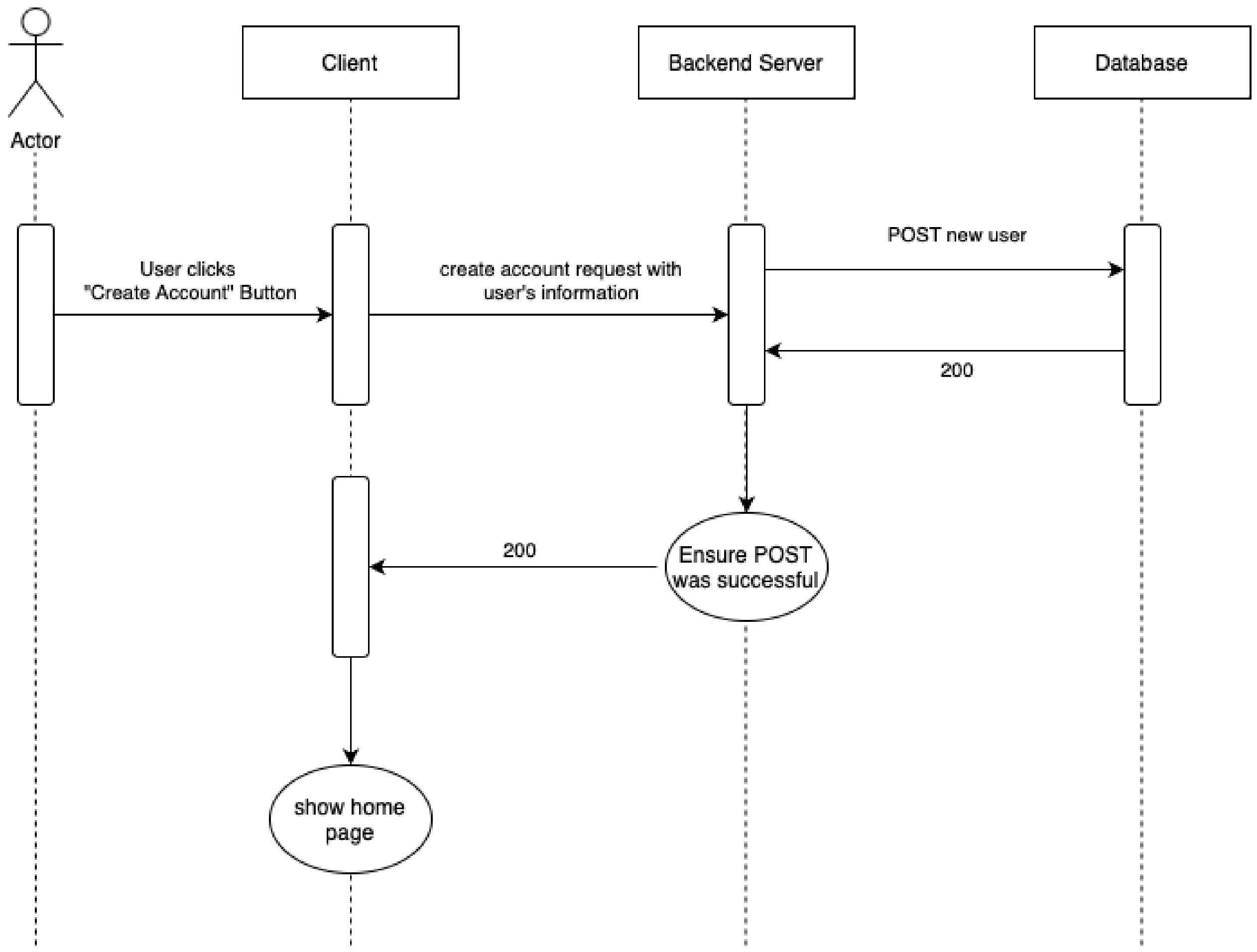
Sequence Diagrams

User Logging Into Account



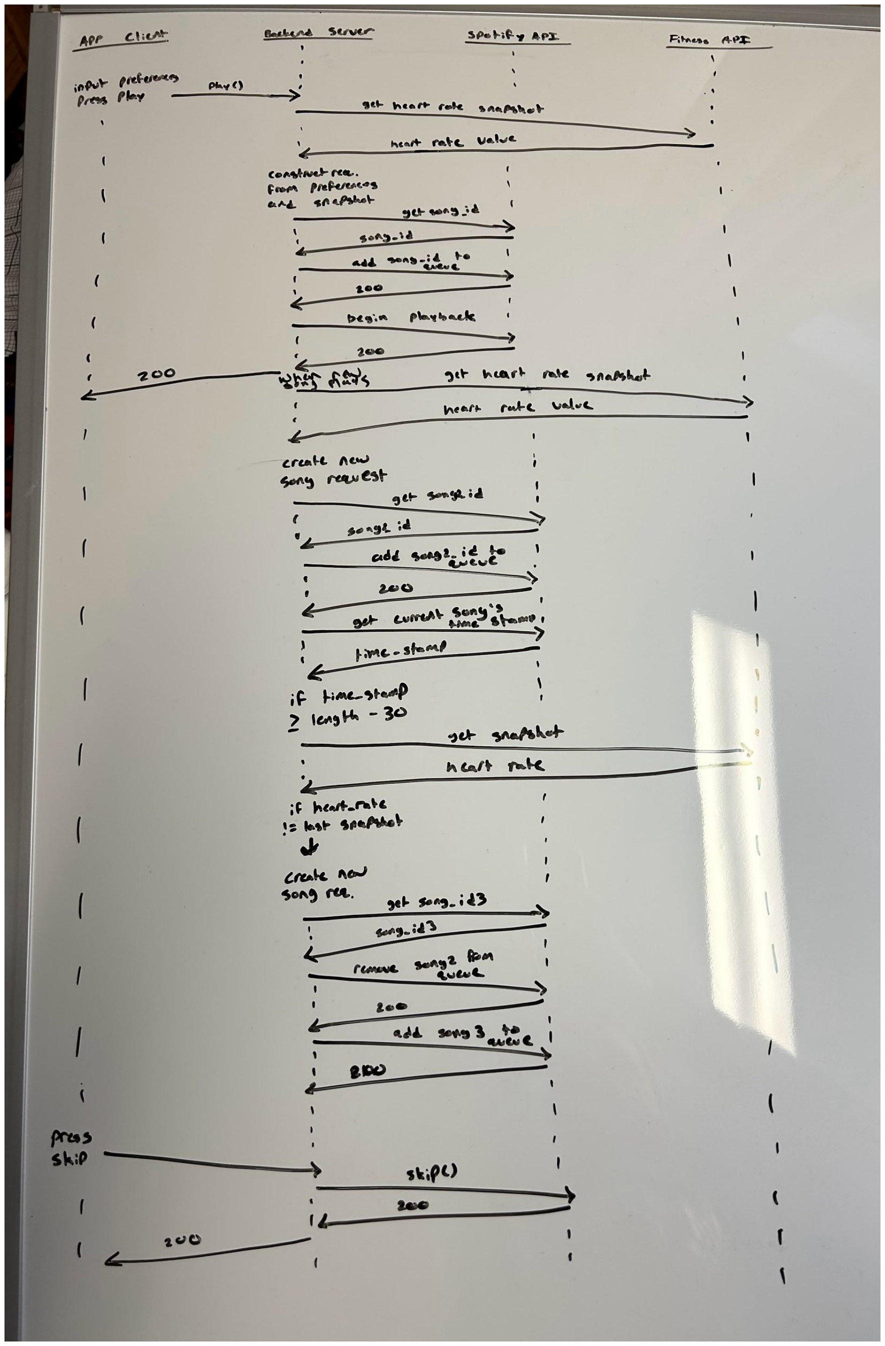
This diagram shows a user logging into their account. The user will input their credentials and press the login button, which will formulate that information into a request and send it to the backend. The backend will then, after pulling the user's login information from the database, compare the submitted credentials to the credentials stored in the database. If the credentials are correct, the server will return a response to the frontend with a 200 response code. Then, the frontend client will display the home page.

Create Account


Actor
Client
Backend Server
Database
User clicks "Create Account" Button
create account request with user's information
POST new user
200
200
Ensure POST was successful
show home page

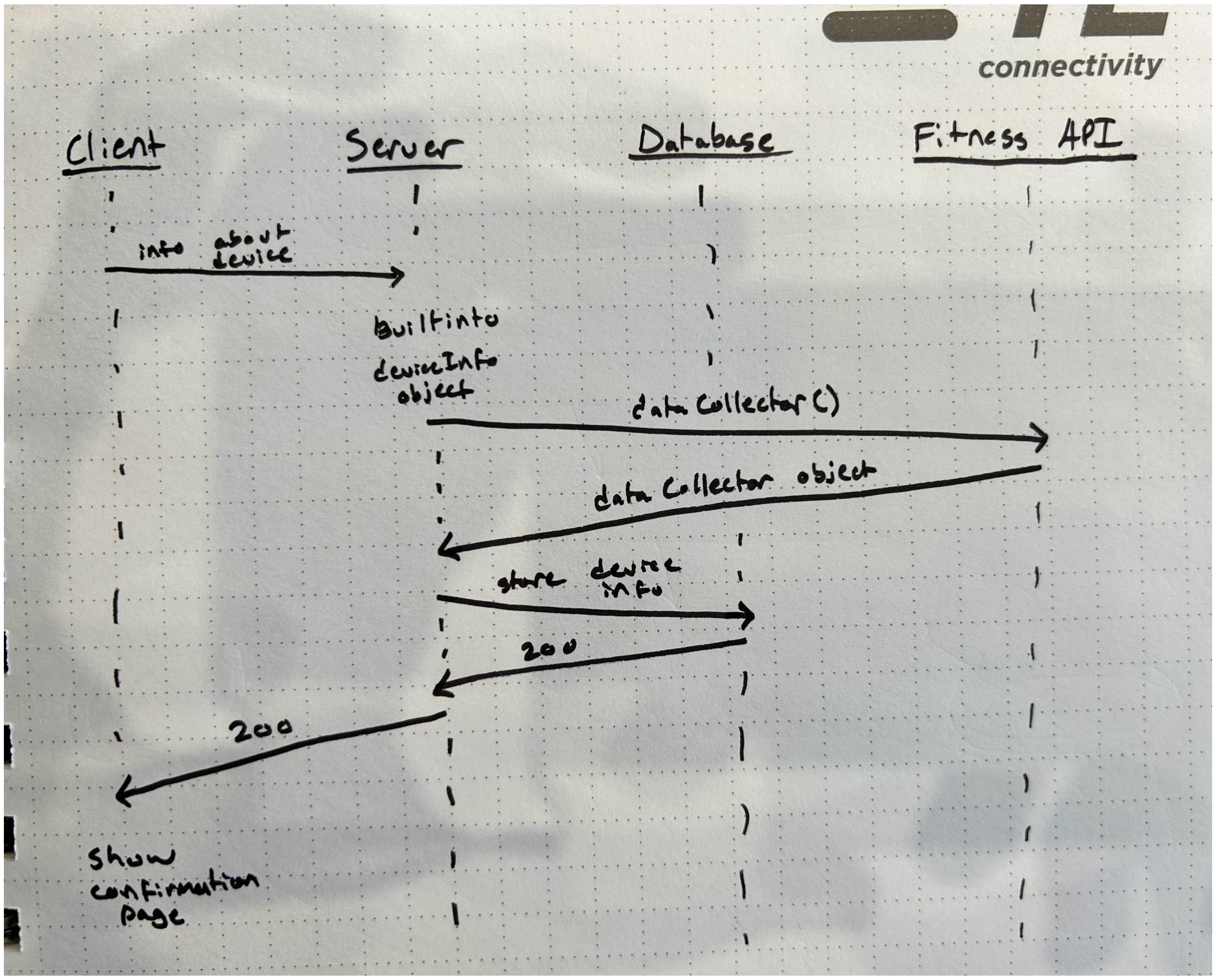
This diagram shows the process for a user to create a new account. After the user enters the information for their new account and they press the “create account” button, the client will send that information to the server, which will forward it to the database in a POST request. After verifying that the database request was successful, the server will then return a 200 code to the frontend client, which will show the home screen.

Basic Fitness Mode Playback Loop



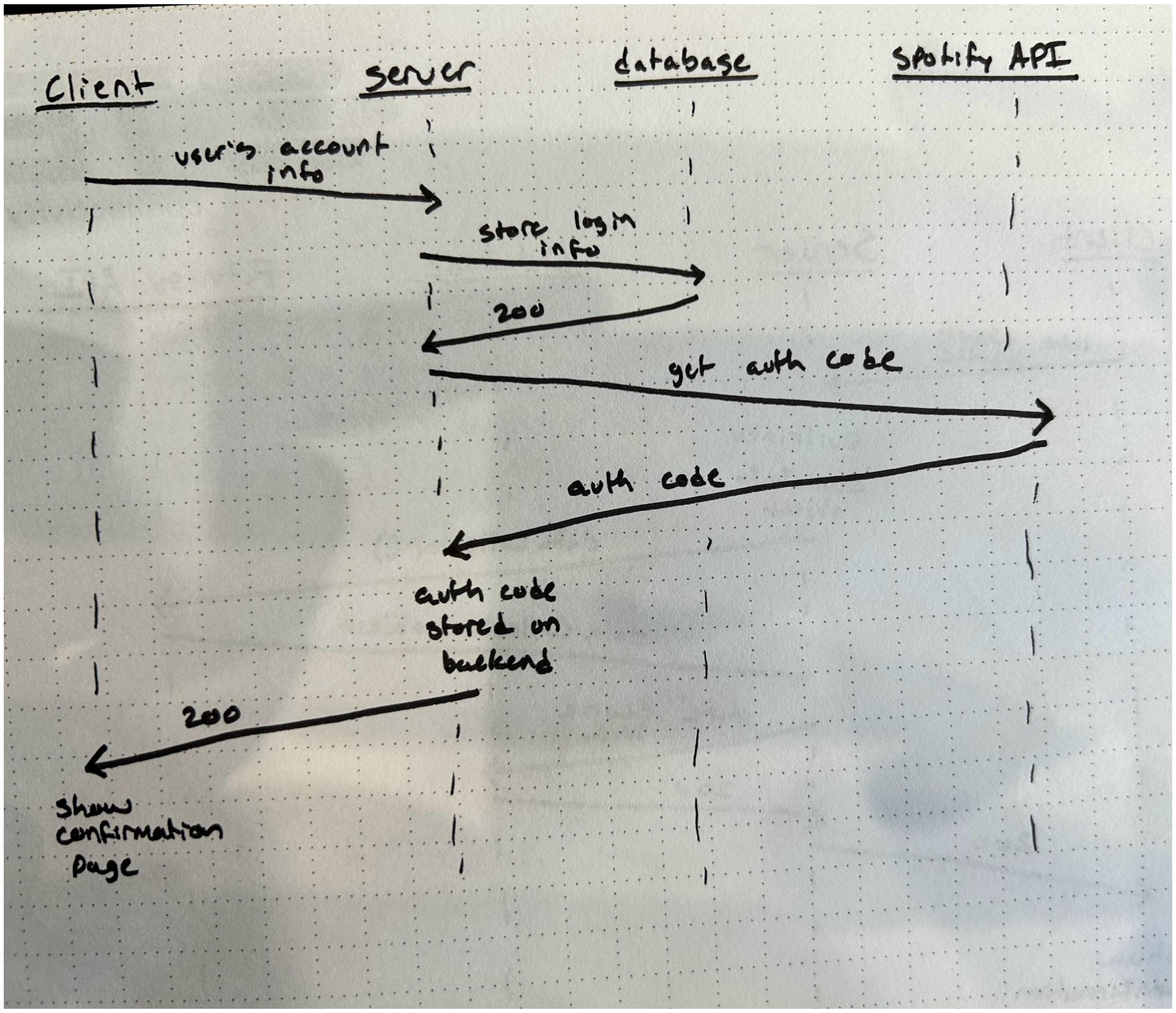
This diagram shows the basic playback for fitness mode, the most complex mode to program. When the mode starts, it gathers the user's fitness data from the paired watch, then queues up and starts two songs based on that data: one to play and one "up next", in case the user skips. It then uses the same process with 30 seconds left in the song, as long as the updated fitness snapshot has a significantly different heart rate value, and after the skip button is pressed.

Add Fitness Device Account



This diagram shows the process for connecting a fitness device account to the user's profile. Once the user inputs their device info, it is sent to the backend and built into the appropriate data object, which is then processed by the fitness device's API and returned to the server as a data collector object. Then, that device's info is stored in the database with the user's profile info. Following that, the frontend client will show a confirmation page.

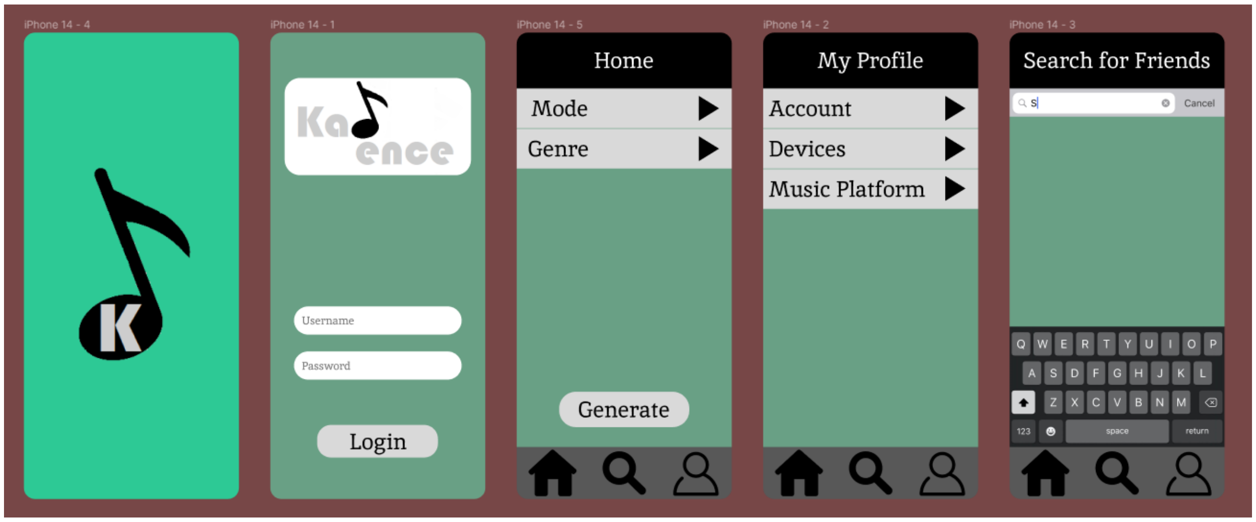
Connect Spotify (or Any Music Service) Account



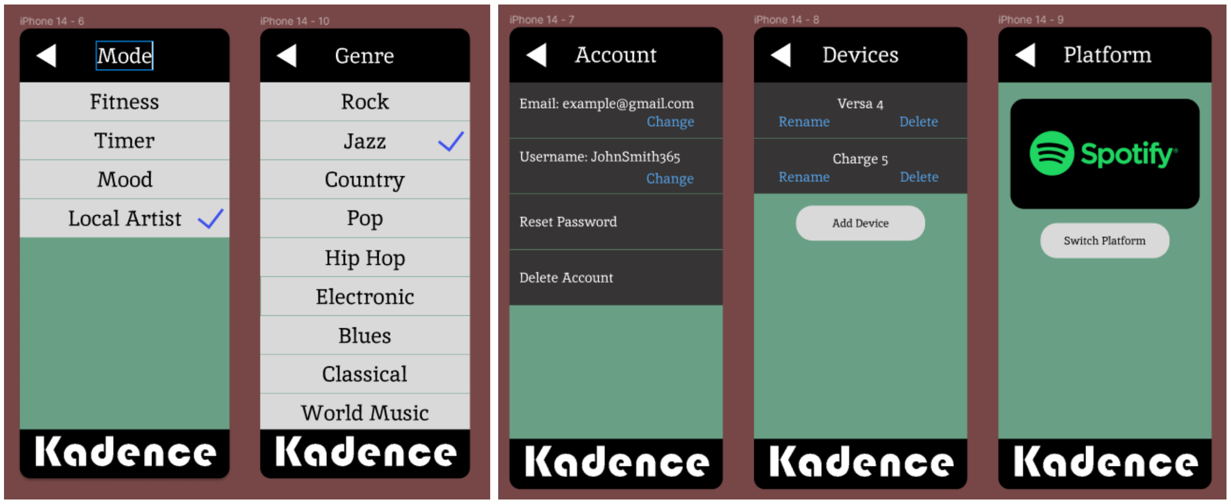
This diagram shows the process of connecting a user’s music account, a spotify account in this case. The user will enter their account info on the frontend page, which is then stored in the user’s profile on the database. Then, the server will use that account info to request an “authorization code”, which is used to authorize requests to the music API. This auth code is then stored locally on the back end, and the server returns a success message to the frontend.

UI Mockups

Below we've included several UI mockups.



Looking at the five screens above, the first will be our loading screen. The second will be our login screen. The third screen will be our home page, where users can navigate to select their mode and genre as well as generate a playlist to listen to. The fourth screen will be our profile / settings page, where users can edit profile information as well as manage their devices and music platforms. The fifth screen will be our search tab, where users can find their friends. Note the navigation bar at the bottom of the last three screens: this will allow users to easily navigate to the various sections of our app.



Finally, the first two screens above are the mood and genre selection pages that are linked to from the home page. The last three screens above are the account, device, and music platform management screens.