SpottyApp

CSC207-UofT/course-project-tylersgroup

Table of Contents

Specification

Spotify allows users to create playlists in which users can sort and save songs. As avid Spotify users, these customizable playlists inspired us to create a SpottyApp which can create playlists in which the titles of each song form a meaningful sentence.

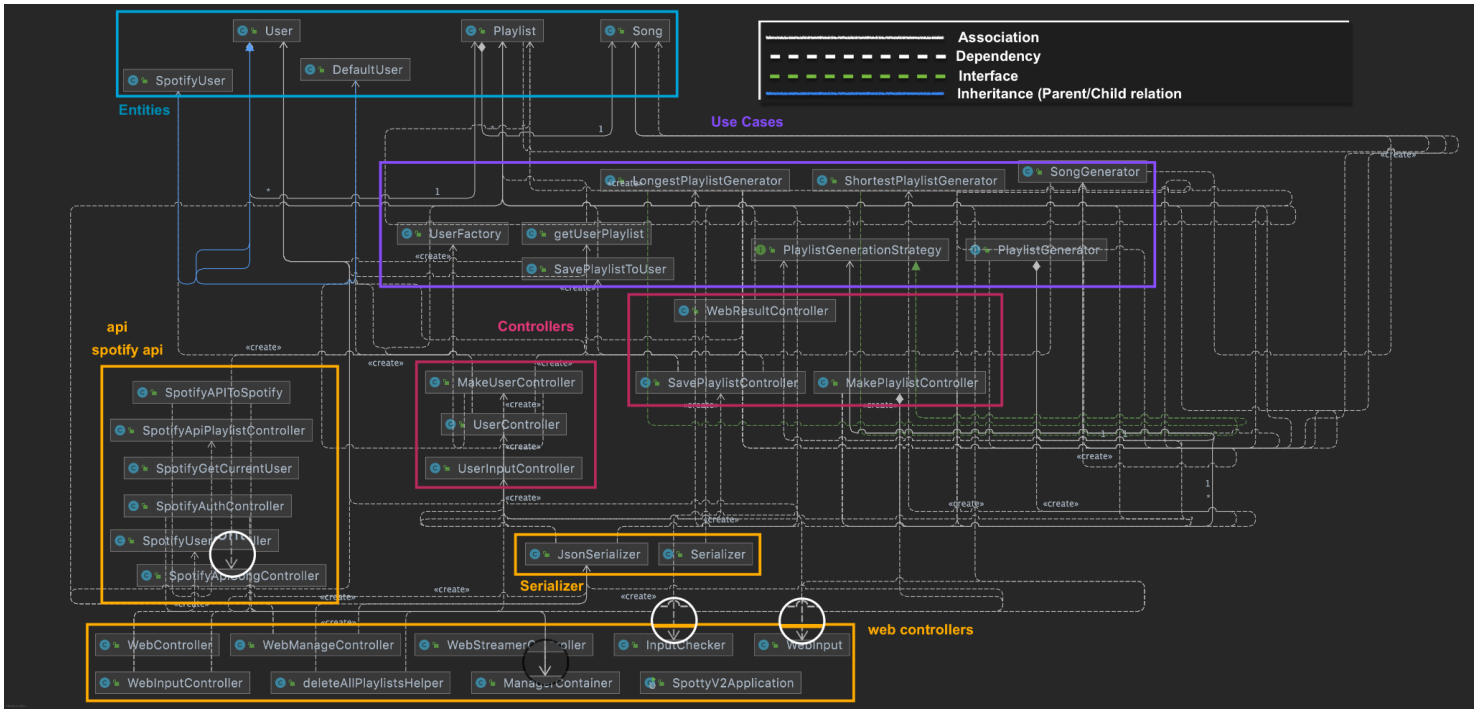
SpottyApp is a playlist generator which generates playlists (group of songs) based on a string input by the user. These playlists can be added to the user's account. This allows SpottyApp users who have Spotify accounts to be able to create custom playlists with ease, explore new songs, and get to know up-and-coming albums without using Spotify.

Simply by entering a meaningful sentence in SpottyApp, it will automatically generate a playlist. This will allow you to dive into the world of music you have never explored before.

- Running the program starts a virtual server, where the user can access the webpage for the web app. The webpage starts on a title screen where the user is prompted to either 1. log in to their spotify account as an existing user, 2. Log in as a default user provided
    - If the user chooses to login with spotify, they are redirected through the spotify api to the spotify OAuth page where they can accept or decline to grant the application access to user information from the requested scopes. If they accept, they are rerouted back to the web app webpage.
    - If the user chooses to login as a default user then they have limited functionality as they will be unable to save generated playlists to their spotify account, however they are able to explore the program without the requirement for a spotify account. They are prompted to log in with provided default user information

- After logging in successfully the program prints a menu where users can choose from:
    - Generate new playlist from string
    - Manage saved playlists
    - Log out + reroute to spotify
- Generate new playlist from string:
    - If the user wants to create a playlist, they are prompted to enter a string of words and choose whether they want to generate the longest possible or shortest possible playlist by number of songs.
        - If the user chooses to generate the longest possible playlist then the program takes small splices (usually single words) of the original string to add as many songs as possible to the playlist
        - If the user chooses to generate the shortest playlist, then the program takes larger splices of the original string by trying to find songs that are combinations of multiple words, and thus the number of songs in the generated playlist is less.
    - The program then finds song titles for each word or group of words. Once song titles are found for each song a playlist is created and the user will be able to choose whether they want to add said playlist to their spotify library.
- Manage saved playlists
    - Once the user has a number of saved playlists they can then edit and manage their playlists, deleting playlists they no longer want and changing the playlist name if they want to.
    - They can also choose to add playlists to their spotify library, and once the playlist has been added, the user has the option to get a url to the added playlist, and share this link with their friends (social functionality).
- Logout + redirect to spotify
    - The logout function logs the user out and redirects them to the spotify homepage where they can then choose to sign in if they wish and login and view the changes the program has made (if they made the choice to save to their spotify)

Class Diagram



Our UML diagram shows the main packages and the relationships between classes. As the image is quite dense with relations, some notable relationships are highlighted below, with explanations to clarify the diagram.

Entities:

- There is an inheritance relationship between the User class, and the two child classes SpotifyUser and DefaultUser
- There are aggregation relations between User and Playlist, and Playlist and Song. Aggregation relations are “has-a” relations, so these relations highlight that User’s have Playlists, and Playlists contain songs.

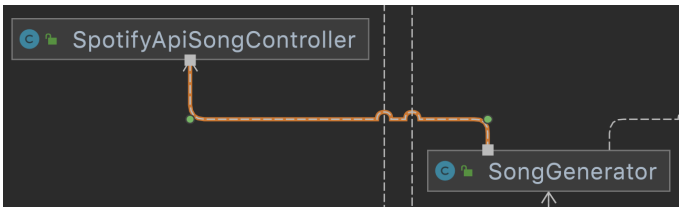
Use Cases

LongestPlaylistGenerator and ShortestPlaylistGenerator both implement the PlaylistGenerationStrategy interface.

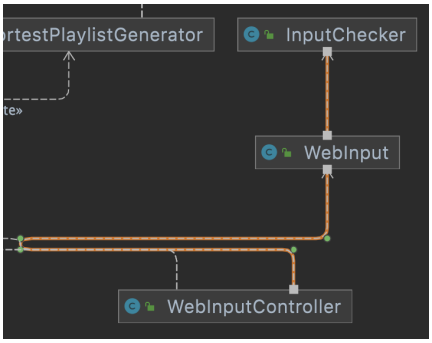
api

- The api package contains three other packages, the Serializer package (database/ data persistence), the Web controllers (web) and the spotify api (
- Although we introduced as many intermediary classes as possible to try and ensure dependencies point inward, there are some dependencies that point outward within the web controllers and spotify api classes.

- The Spotify Api Controller has a dependency on the SongGenerator class. Given more time we would refactor to implement some interface classes between the layers to handle this, as we did for our other classes that had this dependency in phase 1.



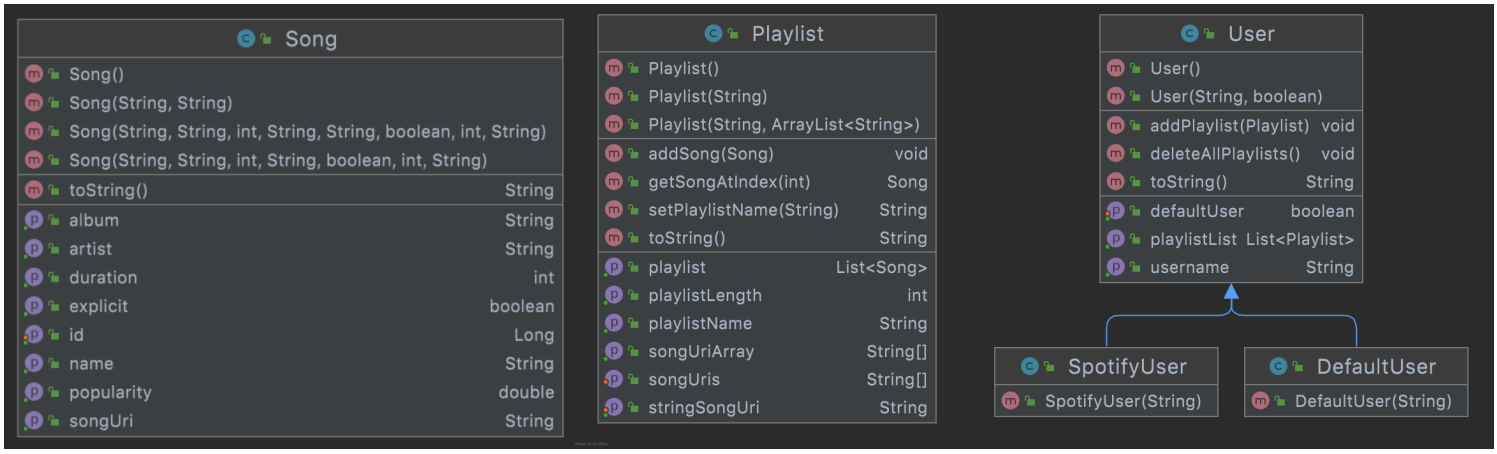
- The InputChecker and the WebInput classes are actually called by the Web input controller which is within the same layer, so although these dependencies seem to be pointing outward (as seen by the arrows), they do not.



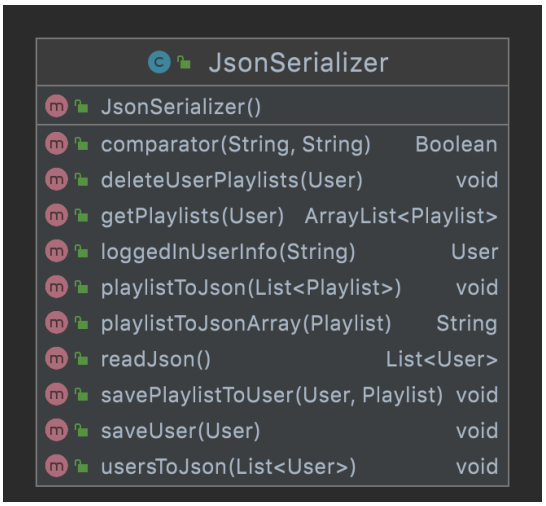
Class Cards

Below are some of our main classes, with their properties (purple), constructors and methods. The relationships within the layers are also shown by the arrows and lines that follow the same colour coding as shown in the full class UML diagram above. Only cards that provide some information about the inner workings of the class have been included, as for example - our controller classes are split and managed in ways that make them self explanatory. Many of these cards are long classes, which is a code smell we could fix with more time by extracting classes

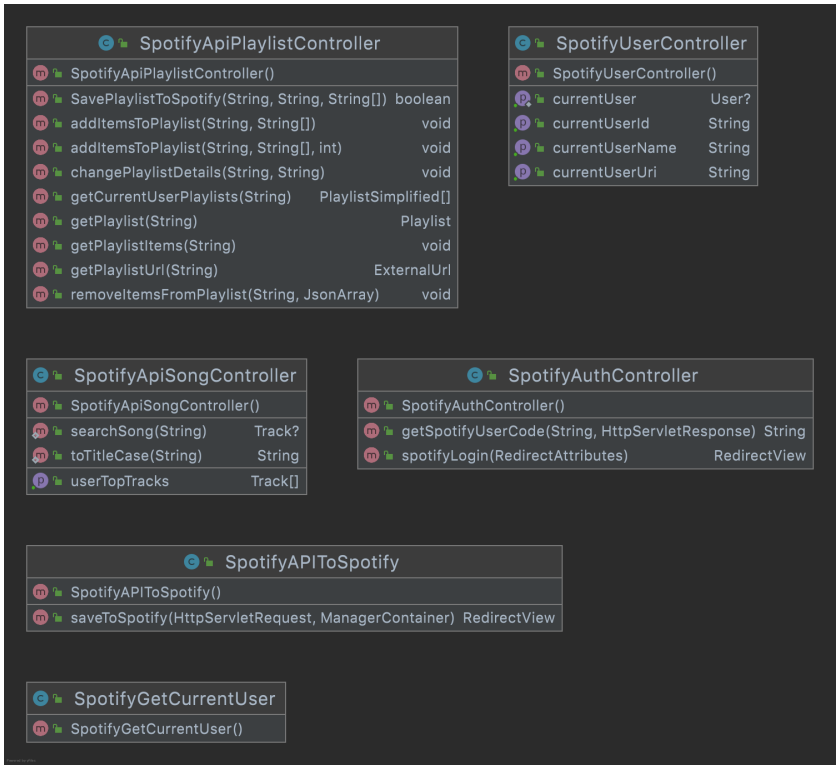
Entities - These cards highlight the attributes of each entity, to give a sense of the functionality implemented by the use cases and controllers



Serializer - The seralizer implements a number of methods used to persist data. We only persist an array of User entities, as they have a “has-a” relation with playlist and song entities, and methods such as comparator and loggedinUserInfo are used to ensure there are no duplicate entities stored in our files



Spotify Api Controllers - Most of the spotify API controllers handle mapping or are REST Controllers making direct requests to the spotify API and databases, making them a part of the outermost layer.



Major design decisions (since phase 1)

1. **Reconfiguring our Web App:** Our Spring configuration from phase 1 did not allow for an integrated database and seamless connection to the API. So, we decided to reconfigure our Spring project. We used Git features to seamlessly integrate this change into our development cycle.
2. **Spotify controllers for API authentication:** We originally intended to store user objects however once we considered the functionality of the spotify API, and that we would need to authenticate through OAuth, we could request the user to allow the app access to User Information. We no longer need to implement an authentication system directly within our code, although we still chose to store the user ID through spotify so we could implement some of our planned extensions as well as to allow us to simplify our data persistence
3. **Implementing a default user and user factory:** Since we want to allow everyone to use our web application whether they have a Spotify account or not, we created a UserFactory class which creates DefaultUser and SpotifyUser classes. These two classes are subclasses of the User class. The UserFactory is called whenever a user logins (as a default user or through the Spotify authentication page), and a corresponding User subclass will be created by UserFactory. This entity is used to store playlists (associate playlists with the user which created them) and for storing to the database.
4. **Data Persistence:** Due to our use of gradle alongside Spring, we ran into many technical issues trying to implement a database during phase 1. Due to time constraints, we decided to implement a different method of data persistence. We considered the use of csv files, but settled on JSON files. We contain all the user entities within a JSON Array object, which we serialize and store to a file that is overwritten whenever User objects are mutated. As User entities act as a container class for associated playlist entities, which contain Song entities, we essentially save all of our created entities within a single JSON file, with a structure that itself implicitly encodes the associations between different objects, allowing us to have all of the functionality we would have used within a database. We use the Jackson library, which uses mapping when serializing and deserializing JSON objects with key value pairs that are the attributes of each entity, which we use when rebuilding entities from our stored data.

Clean Architecture

Dependency Rule

Code organisation: We organised our code into packages that represent the layers of Clean Architecture. This means that we are able to ensure that we are not making calls unless they are across adjacent layers. The notable packages are Entities, UseCases, and Controllers.

1. Domain Layer → Entities
2. Application layer → Use Cases
3. Adapter Layer → Controllers
4. Infrastructure Layer (Frameworks and Drivers layer) → api

Abstraction Principle

Our “resources” directory is where we have the web “views” layer. This means we have placed abstractions of the objects in the inner layers and implementation of our interfaces in the outer layers. Our innermost layers contain business rules and logic and are the most abstract and implementation details are in the outer layer which is the most concrete.

SOLID Design Principles

Single Responsibility Principle:

All of our classes are created in terms of their singular functional responsibility, and are named as such. Some example are:

- PlaylistGenerator has the single responsibility to generate playlists, not dependent on what type of song objects or strategy algorithms (web based or UI based) are being used.
- SpotifyApiController being split into two smaller classes more specific to their responsibilities as controllers for playlists and songs.

Open/Closed Principle:

Our high level modules do not depend on lower level modules, but rather on abstractions, which do not depend on details. For example:

- Even while implementing data persistence, we added new methods to our existing entity and use case classes which did not require any changes to the structure of our objects. We did not need to change our inner levels, besides adding to them because these abstractions do not depend on details.

Liskov Substitution Principle:

Objects that are subtypes can be substituted for the objects they are subtypes of without altering the desired properties. For example:

- PlaylistGenerationStrategy interface implemented by shortestPlaylistGenerator and longestPlaylistGenerator. The playlistGenerationStrategy class specifies two methods, generatePlaylist and generatePlaylistWeb, which are both implemented in ways specific to generation of shorter playlists, and longer playlists. These classes could replace generatePlaylist with no change in functionality.
- DefaultUser and SpotifyUser are two subclasses of User. These two subclasses are created using UserFactory. Any code that takes in a User as an argument can be replaced by DefaultUser and SpotifyUser

Interface Segregation Principle:

We made sure that the classes/interfaces we've created contain methods that are grouped together by relevance, and as a result, the user is not forced to implement any irrelevant methods. For example:

- SpotifyApiController was split into SpotifyPlaylistController, spotifyUserController and SpotifySongController to group methods together and separate them in such a way that makes more sense
- Playlist generation strategy interface contains the relevant methods only that are used by shortestPlaylistGenerator and longestPlaylistGenerator both.

Dependency Inversion Principle:

Our high level modules do not depend on low level modules but rather abstractions. As explained above our code organisation and packaging strategy ensures we adhere to this principle. There are several places in the code where this is displayed, for example:

- Playlists are generated in the Playlistgenerator class, which uses the interface PlaylistGenerationStrategy when generating playlists, which acts as an abstraction layer between the classes.
- In phase 1, we mentioned that we didn’t introduce many abstraction layers to separate the low and high level classes, which we fixed in phase 2 by introducing specialized controllers which apply the use cases.
    - The controller class, UserReadWriter, implements the ReadWriter interface
    - More controllers such as makeplaylistcontroller, saveplaylistcontroller, makeUserController have been implemented which call on use case classes.

Common Closure Principle

Our packaging strategy means we have separated our packages by the layers of clean architecture. The common closure principle states that packages (or components) should only have one reason to change and so within our api class we decided to separate each functional component into their own packages, to keep our code organised.

Common Reuse Principle

This principle states that classes in the same component should be reused together, similar to interface segregation in clean architecture. The packages ensure that classes that are used or called together are contained within the same components.

Acyclic Dependency Principle

Our class diagram (although hard to read in screenshot format) shows there are no dependency cycles within our code

Refactoring

Examples of refactoring from phase 1 to phase 2

1. **Problem:** Many methods had missing or outdated javadoc.
   Solution: Updated these javadoc.
2. **Problem:** The SpotifyApiController has low cohesion as it contains methods which deal with songs and playlists.
   Solution: The SpotifyApiController is split into several new controllers: SpotifySongController, spotifyUserController, and SpotifyPlaylistController. These controllers have higher cohesion because they manage only songs, users, or playlists respectively.
3. **Problem:** There were unused methods and classes from phase 1 since we were implementing two interfaces (as we still had our command line interface)
   Solution: We decided to discontinue our text based interface in favour of our web interface, so we deleted all CLI related classes.
4. **Problem:** Entity classes contained variables that we thought we'd use later in the project but ended up not needing, for example: Song objects had the variable "genre" indicating the genre of a song but we later found out that the spotify api does not actually keep track of the genre.
   Solution: Removed any redundant variables from our entity classes. However, not all unused variables were removed for the purpose of further extension to the program, which is a code smell in phase 2 but for a good purpose.
5. **Problem:** The Web controllers did not all adhere to the open/closed and single responsibility principle and were a large class
   Solution: The Web Controllers are refactored by splitting the WebInputController class into smaller classes (WebManageController, WebResultController)

Use of GitHub Features

We used GitHub features extensively in our refactoring process. During phase 1 we did not name our branches correctly, or fully use the functionality of git such as creating sub-branches. In phase 2, due the reconfiguration of our Spring application, we learnt how to sub-branch, stash and apply changes effectively as well as how to use git commands in the terminal in intelliJ effectively

Packaging and Code Organization

Our packaging strategy is grouping classes in the same layer of Clean Architecture into one package.

- **api** → This package is responsible for using the Spotify Web API and data persistence. It contains 3 packages within it:
  - Serializer: contains classes which serialize user and playlist data, saving them to a JSON file.
  - spotifyApi:contains classes which make calls to Spotify to gain access to user data
    - spotifyApiSpotifyAuthController – controller for authorization of access to user's spotify account. It is the key to accessing Spotify's own database.
    - Other controllers for spotifyUser, spotifyPlaylist, spotifySong.
  - Web: Web contains classes which controls the webpage of SpottyApp
- **Controllers** → This package includes all the controllers. Notable ones include MakePlaylistController and MakeUserController.
- **UseCases** → This package includes all the use cases. Notable ones include
  - ShortestPlaylistGenerator and LongestPlaylistGenerator, which are responsible for generating a short or long playlist;
  - Song generator is used by the two playlist generators mentioned above, to find a song from an existing collection of songs;
  - UserFactory class creates two subclasses of User: DefaultUser class and SpotifyUser class
- **Entities →** There are several entity classes in this package. The notable ones are Song, Playlist, User.
  - User class has two subclasses: SpotifyUser and DefaultUser
  - These classes are for data storage and contain basic methods for manipulation (e.g. changing attribute values).

Lastly we have a resources package which contains HTML and CSS files, responsible for creating the interface for the web application.

Design Patterns

- Dependency Injection:
  To Prevent hard-coding the Song type of assigned values within an instance of the Playlist class we have used the SongGenerator class to add created Song objects into the Playlist class. This allows for us to subclass song types and if we were to, for example, extend functionality to include podcasts from Spotify, the code would still work effectively. This allows for dependency inversion
- Strategy - Longest vs shortest playlist generator:
  We implemented two different methods of playlist generation as options to the user. For this we implemented a strategy design pattern interface to create two algorithm choices to decide if the want a longer or shorter playlist (by number of songs)
- Singleton:
  The UserManager and PlaylistManager classes in the UseCases package are both singleton classes.
    - Singleton design pattern is used here because we want one and only one manager for users and playlists each. Moreover, this design provides global access to these managers, which means that there are other use cases to call methods of these classes. In addition, it allows lazy initialization, which means we call, and initialize, these manager classes only when we need them.
    - One possible downside is that since singleton classes can be accessed globally, we must pay extra attention as to not call this use case class in higher level classes.
- Simple Factory
    - The UserFactory class is used to create two subclasses of User: DefaultUser and SpotifyUser. DefaultUser represents users of our program who don’t have a Spotify account; SpotifyUser represents users who have a Spotify account.
    - When the user logins either as a Default User or a Spotify User, a corresponding subclass of User will be created via the UserFactory. This entity is used for associating playlists generated and is stored in the database along with the playlists.

Existing Design Issues (Code Smells)

- **Security Vulnerabilities**
  Users without Spotify accounts use one Spotify account we create. This account-sharing behavior somewhat violates the policy of Spotify and is not in good practice. It is strongly recommended that every user create their own spotify account (even if it is a free account) to use our app. This results in minimal privacy issues as they login through the Spotify authentication page and users know explicitly what permission the app requires.
- **Class Method not used by Itself** - Inappropriate intimacy
  SpotifyApiSongController class contains a String helper method toTitleCase, which is used in the searchSong method within the class to manipulate the spotifySearch algorithm better, toTitleCase is not necessarily a method used by the SpotifyApiSongController objects directly. Could be moved to a helper method class.
- **Long Entity Classes** - Large class
  Entity classes contain methods that aren't currently implemented, but could be implemented in the future for any extensions. However, in the time being they are just taking up space in the classes without having any purpose.
- **Long Methods**
  The playlist generator algorithms are both quite long. This is due to the complexity of the process. However, it could still be shortened using extract method.
- **Inline Styling of HTML**
  We designed the web application interface using HTML and CSS. However, the CSS stylesheets can't link with the HTML files. We instead copied the CSS code into the HTML files and used Inline Styles in HTML instead.

Progress Report

Group member	Summary of work for phase 2	Pull request links	Comments
Tyler	Created the command line interface (CLI) during phase 1, which is later removed for simplicity during phase 2 Cleaned up code smells, bad architecture structure etc; Removed the CLI package and related controllers Implemented a User Factory which produces a DefaultUser and SpotifyUser, both of which are subclasses of User Improved the interface of the webapp via CSS and HTML	https://github.com/CSC207-UofT/course-project-tylersgroup/pull/148 User Factory and 2 User subclasses, DefaultUser and SpotifyUser are created https://github.com/CSC207-UofT/course-project-tylersgroup/pull/158 https://github.com/CSC207-UofT/course-project-tylersgroup/pull/162 https://github.com/CSC207-UofT/course-project-tylersgroup/pull/166 https://github.com/CSC207-UofT/course-project-tylersgroup/pull/169 Created the Mystery Green Terminal Styling for the interface of the webapp https://github.com/CSC207-UofT/course-project-tylersgroup/pull/142 Removed unnecessary components and classes, including the cli package	The interface for the webapp should be more simple and straightforward while being more elegant. User Factory plays a huge role in deciding whether the user of our program can save/access playlists, in addition to generating new playlists
Max	Implemented the delete all playlists feature. Helped link the spotify api to save playlists to a user's spotify	https://github.com/CSC207-UofT/course-project-tylersgroup/pull/151 Web streamer class for "Matrix Mode". https://github.com/CSC2	The saving playlists to spotify pull requests only worked with the serializer class. The fix to make it work with the JsonSerializer was pushed directly to

	account to the front end using spring boot controllers. Implemented the “Matrix mode” using streaming response body data types in spring boot. Implemented the default user back end handling.	07-UofT/course-project-tylersgroup/pull/174 Default user behaviour. https://github.com/CSC207-UofT/course-project-tylersgroup/pull/179 Saving playlists to spotify.	phase-2 due to time constraints.
Minori	Implemented Json serializer that serialize and deserialize User entity and its associated playlists and corresponding songs from/to a Json file	https://github.com/CSC207-UofT/course-project-tylersgroup/pull/161/files Implemented 1) LoggedInUserInfo method which returns a saved User if the current user is in our json file, or returns a newly created User if they do not exist in our file. 2) saveUser which saves user to our file, making sure we overwrite the existing user to avoid duplicates https://github.com/CSC207-UofT/course-project-tylersgroup/pull/183 https://github.com/CSC207-UofT/course-project-tylersgroup/pull/184 Made changes to SpotifyAuthController and JsonSerializer to debug de/serializaiton.	Data persistence took longer than it should have because of some misunderstanding and minor error. For example, prior to the change in SpotifyAuthController, I was not sure why the program was giving back an empty User every time a user logs in. Turned out SpotifyAuthController was calling MakeUserController which created a brand new User instead of taking the information from the existing User in Json file.
June	Optimized our playlist generating algorithms to result in a smoother user	https://github.com/CSC207-UofT/course-project-tylersgroup/pull/157 ^playlist generator	“Optimizing” an algorithm sounds really cool but in truth it was not too complex (mostly because our

	experience. Increased the efficiency by over 100% (“I kinda like you and I wanted to tell you you might not” previously took 160-170 seconds and it takes around 60 seconds after the optimization.) Also improved the longest playlist generator algorithm so it has a greater % of song matches. (Difficult to quantify as it greatly depends on the input. “I want to see you” would have previously given 40-60% matches, but after the changes we get a 100% match!). Implemented ID persistence between the browser and server using client side JS and SpringBoot controller mapping on the backend. Did a bunch of client side JS stuff in general.	algo improvements https://github.com/CSC207-UofT/course-project-tylersgroup/pull/143/files ^id persistence	previous algo was very very inefficient). You’ll see what I mean once you look at the code. Also, the code quality in the pull requests aren’t exactly “production ready”, however, I assure you that we cleaned up our code base gradually over a few pull requests.
Sara	Worked alongside Minori to implement the JSON serialization. Solved a number of implementation issues, as well as	https://github.com/CSC207-UofT/course-project-tylersgroup/pull/140 Implementing the Json serializer https://github.com/CSC2	I worked alongside minori on the data persistence side of the project. I spent a lot of time reading documentation and trying to decide which

	read documentation to effectively consider which libraries and techniques we should use.	07-UofT/course-project-tylersgroup/pull/120 Initial implementation for database stuff	libraries contained the functionality we wanted to implement to create a single JSON file, that stored all of our information in an effective way. This meant cycling through a number of library choices, and reading through documentation to decide which methods to use, as I had some understanding of this from my research on databasing in phase1. Effectively me and minori worked an equal amount on data-persistence, although we used minoris device to collaboratively write and debug the data persistence code.
Danika	Implemented saving playlists generated by our app into the user's spotify library, allowing extension for sharing functionality.	https://github.com/CSC207-UofT/course-project-tylersgroup/pull/153 Contains implementation for saving playlists to spotify user's library, edits to entity classes made for storing song uris.	This pull request contains mostly the correct code, but with a minor issue where I was calling an object of the class within that class, which was honestly a brain fart on my part but overall the saving functionality is working as it should be in this pull request. Working with the spotify api has been a bit difficult in the sense that it's hard to know how the api is working as it's hidden from the user, which makes sense in terms of clean architecture.

Project Accessibility Report

Principles of Universal design

- **Equitable Use**
  - Every person who has access to the Internet can use our program, even if they do not have a Spotify account: they can simply login as a “default user”, which will allow them to generate their own Spotify playlists.
  - We use semantic HTML allowing screen-readers to work effectively
- **Flexibility in Use**
  - There are different methods of playlist generation.
  - All functionality that accesses spotify directly is implemented as a separate choice within the management screen, such as saving and deleting playlists
  - Buttons are large and easy to see, with instructions throughout the page to guide the user how to use our program.
  - We have a responsive UI that is mobile-friendly, so the program is accessible on the go and on different screen formats.
- **Simple and Intuitive Use**
  - Straightforward instructions are included on each page of the web application, guiding the users when they are using the program.
  - The user is given effective prompting and feedback while they are using the web application. For example, after the playlist is generated.
- **Perceptible Information**
  - Only important information is shown on the screen to the user.
  - The interface is accessible on different devices, for example, mobile phones, desktop computers and laptops and scales accordingly to display information clearly.
- **Tolerance for Error**
  - When the user wants to generate a playlist based on their sentence input, the input is restricted to minimize error. Symbols and emoticons are forbidden inputs and the user is aware of this.
- **Low Physical Effort**
  - The styling of the web application uses a high-contrast design with large fonts, so users can use the application without much effort.
- **Size and Space for Approach and Use**

- All elements in the interface are centered so mobile users can easily access them, whether they are left- or right-handed.
- This web application is easy to use in any situations

Target Audience of SpottyApp:

Our program is primarily marketed towards Spotify users only at this stage. Other Spotify third party apps also have integration with other music streaming platforms such as Apple Music, and so if we consider this project on a larger scale we may have generalised our target audience to users of music streaming sites.

Our program is less likely to be used by users with free spotify accounts. Free Spotify accounts can only create up to 15 playlists and so a playlist generator is likely to be used by people who want to create a number of playlists easily or perhaps explore new music. Furthermore users without a spotify account are extremely unlikely to use our program.

Using SpottyApp

Quick Start

Step 0: Starting the program from IntelliJ

Pull from https://github.com/CSC207-UofT/course-project-tylersgroup/tree/phase-2 to your local machine via IntelliJ. Run SpottyV2Application and type "localhost:8080" in your preferred browser to start SpottyApp.

Step 1: Login through Spotify

- Click "Login via Spotify" to login through the official Spotify authentication page to gain full access to SpottyApp (Strongly Recommended)
- Click "Continue without Spotify" if you do not have a Spotify account. You can only access the playlist generator page if you use this option.

If you do not have a Spotify account, it is strongly recommended that you create a free Spotify account on https://www.spotify.com/us/.

See more details about logging in in the "Full Instructions" section.

Step 2: Using SpottyApp

Follow the instructions on the screen and explore SpottyApp!

Full Instructions

Step 0: Starting the program from IntelliJ

Pull from https://github.com/CSC207-UofT/course-project-tylersgroup/tree/phase-2 to your local machine via IntelliJ. Run SpottyV2Application and type “localhost:8080” in your preferred browser to start SpottyApp.

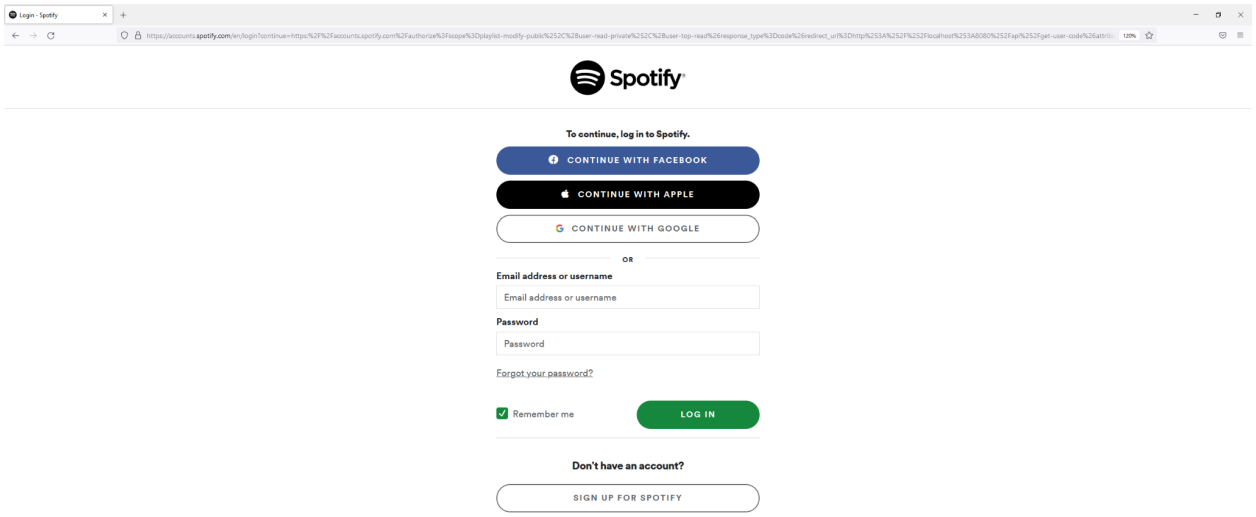
Step 1: Starting the web application

This is the Index page of SpottyApp.

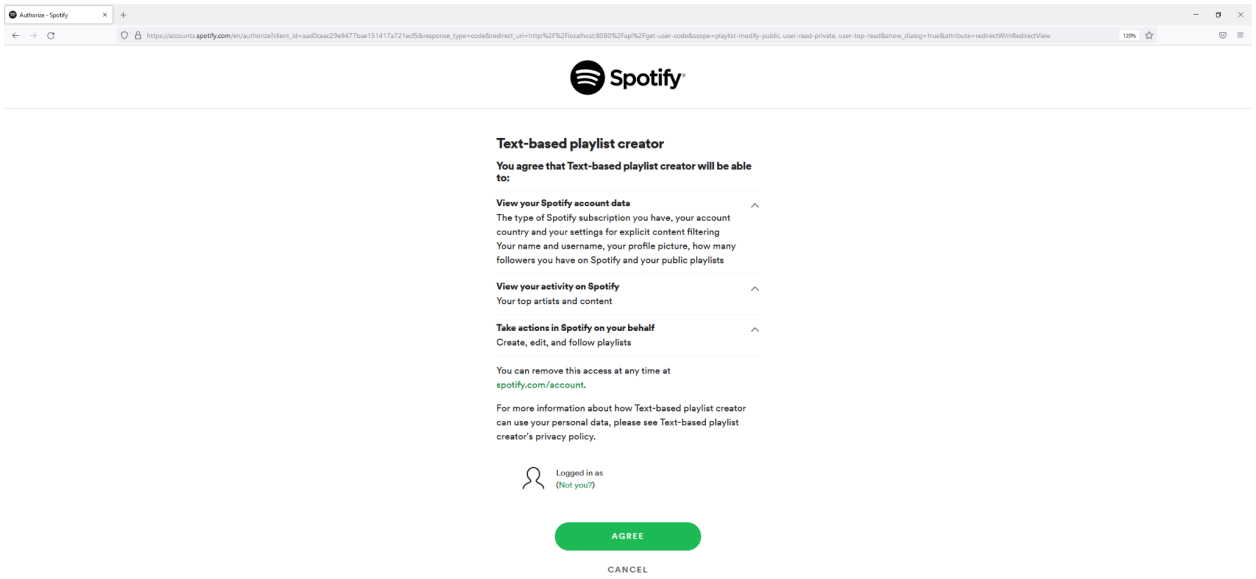
- Click “Login via Spotify” to login through the official Spotify authentication page to gain full access to SpottyApp (Strongly Recommended)
- Click “Continue without Spotify” if you do not have a Spotify account. You can only access the playlist generator page if you use this option.

If you do not have a Spotify account, it is strongly recommended that you create a free Spotify account on https://www.spotify.com/us/.

Step 2: Login through Spotify

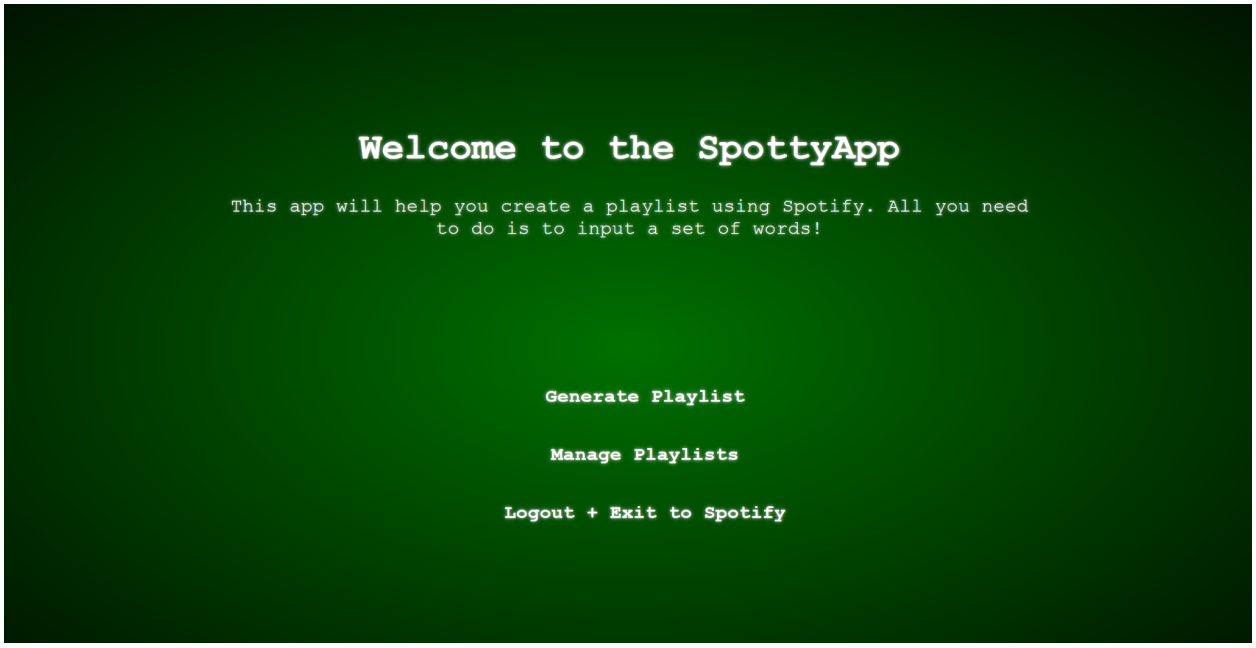


If you choose “Login via Spotify”, simply login with your credentials. If you choose “Continue without Spotify”, simply click the field “Email address of username” and press CTRL+V on Windows or CMD+V on MAC to copy the login details into the field. Do the same for the field “Password”. Finally, click “LOG IN”.



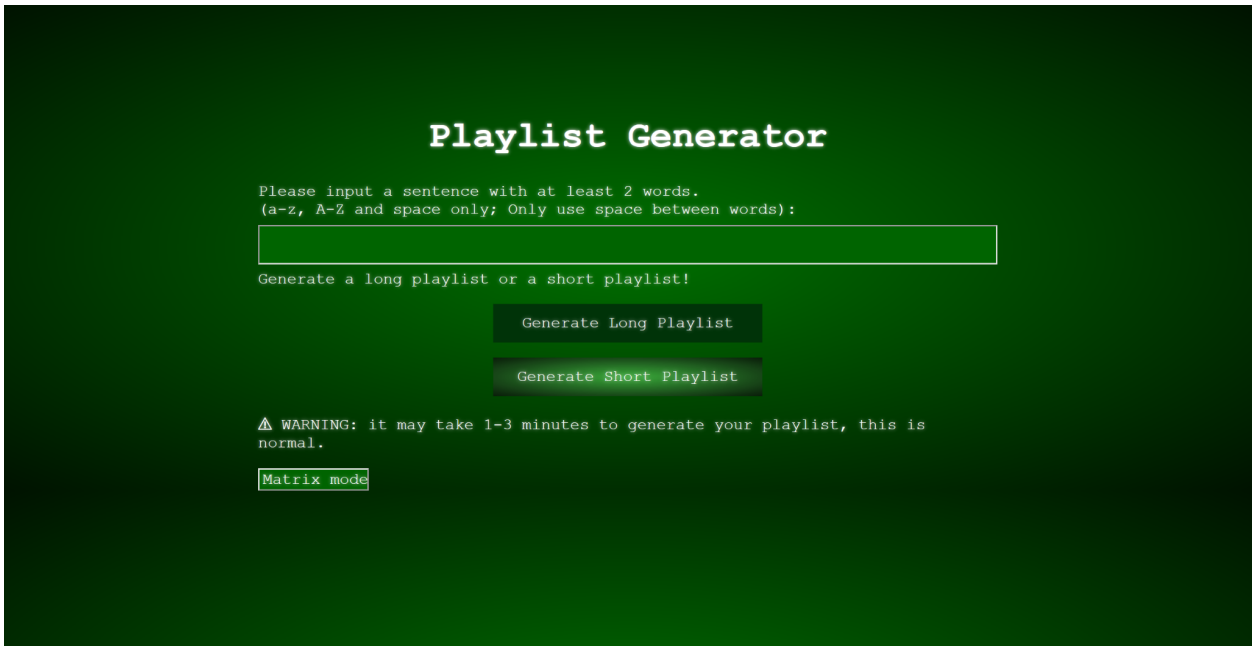
SpottyApp requires your permission to view your Spotify account data in order to generate and save playlists. Click the “AGREE” button if you agree to grant SpottyApp with such permission. Else, click “CANCEL” and close the browser.

Step 3: Using SpottyApp:



If you want to generate a playlist, click the “Generate Playlist” button.
If you want to manage the playlists generated by SpottyApp, click the “Manage Playlists” button.
If you want to logout and exit to Spotify, click the “Logout + Exit to Spotify” button.

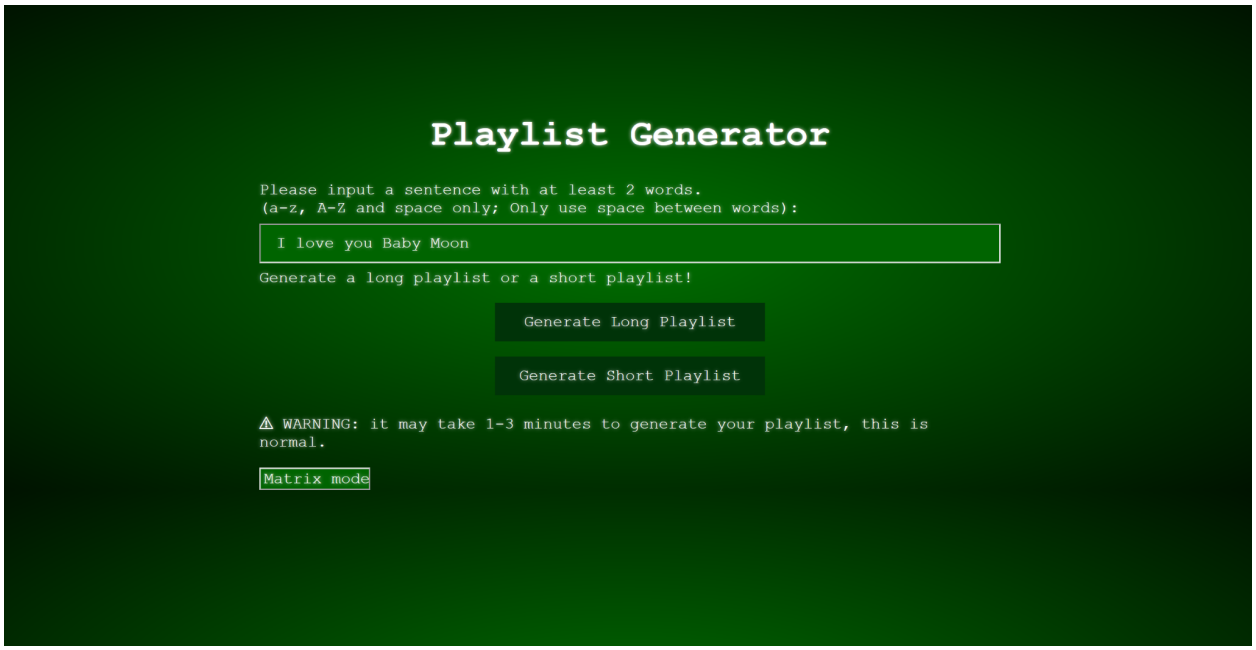
Step 3a: Generating Playlists



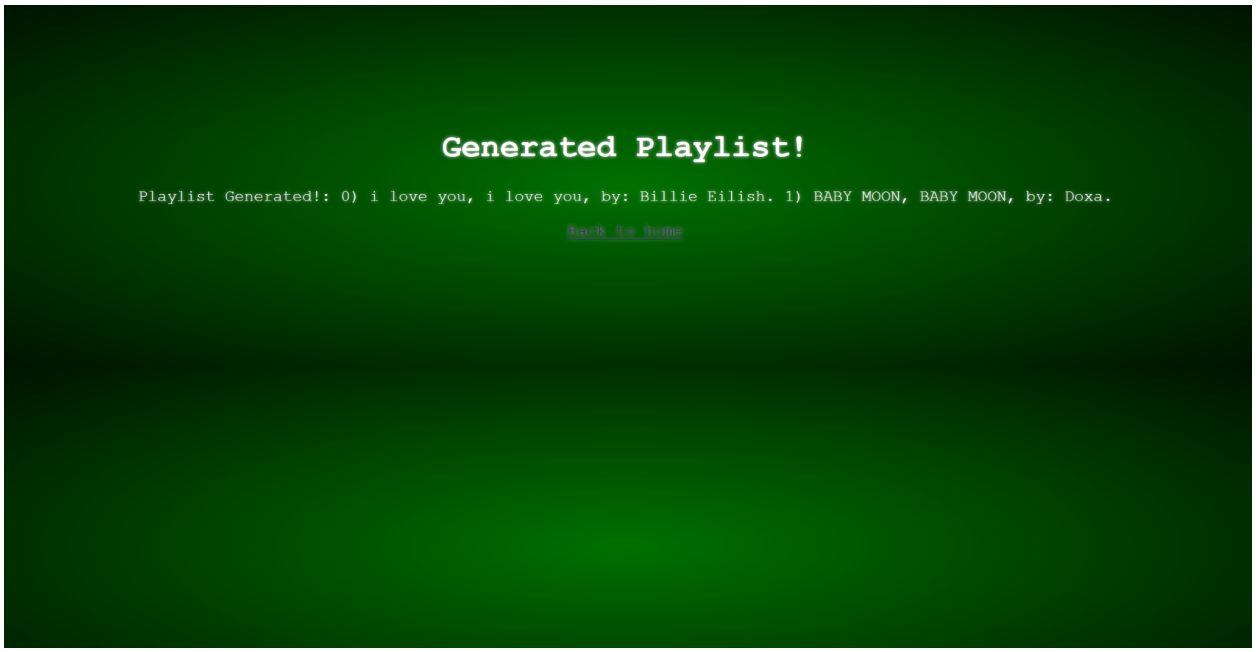
Our playlist generator generates a playlist according to your sentence input and your preference for a long playlist or a short playlist. The titles of the songs in the generated playlist will match your sentence input word by word. A “Long Playlist” refers to a playlist with more songs, while a “Short Playlist” refers to a playlist with fewer songs.
To generate your playlist, simply type a sentence in the input box. Note that the sentence you input must have at least 2 (TWO) words, uses the English alphabets only, and have space in between words (but not at the start or end of the input). Also note that long sentence inputs may require longer time to generate a playlist.
For example, “How are you” is a valid input, but not “Hello”, “ Sunshine Summer” or “Let’s go home!”.

After inputting your sentence, click the “Generate Long Playlist” button or the “Generate Short Playlist” button to generate a long or short playlist.

An example is shown below:

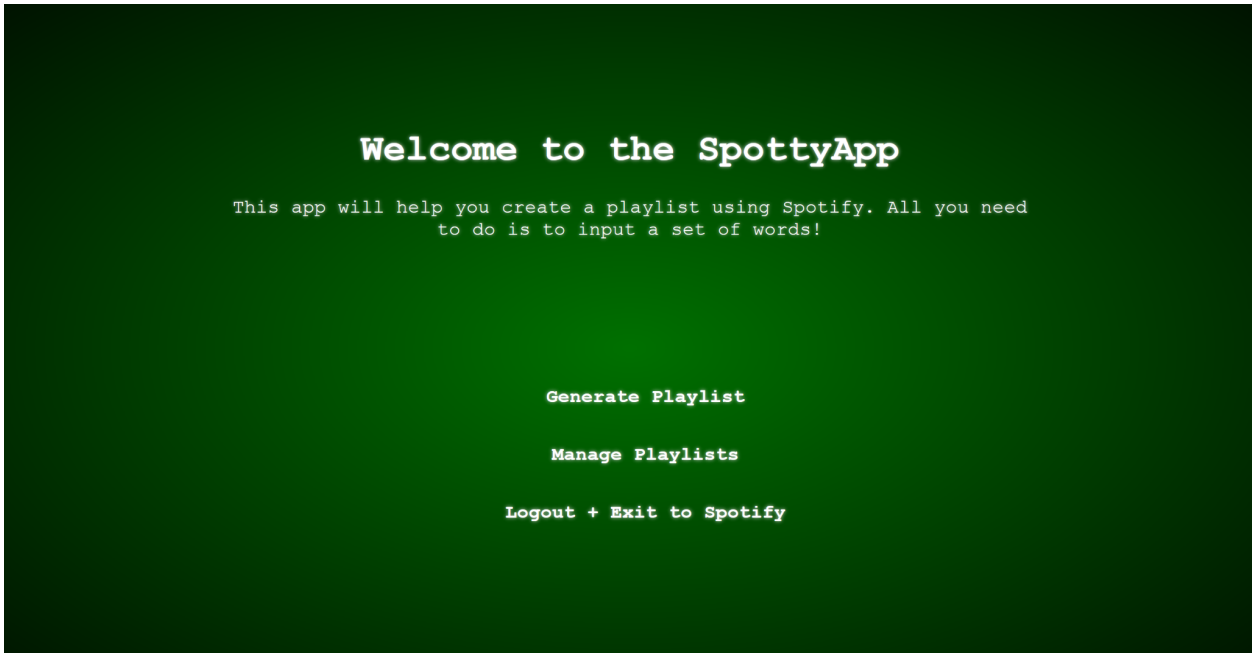


In this example, the sentence input is “I love you Baby Moon”.

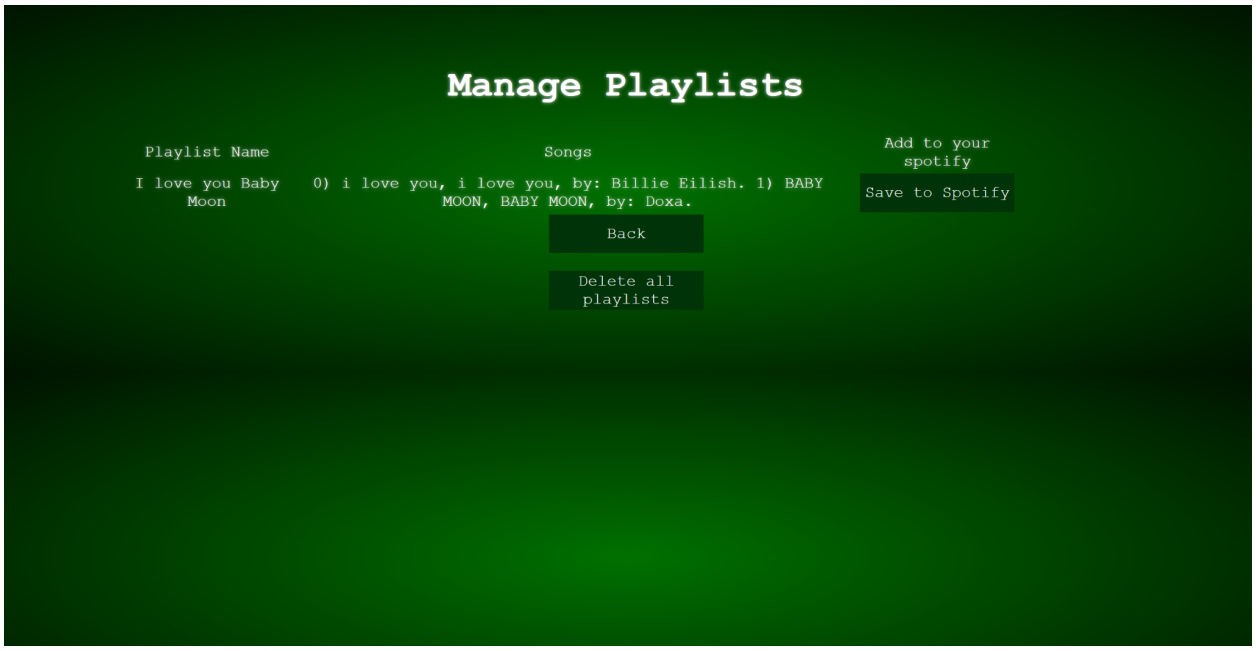


After choosing “Generate Long Playlist”, when the playlist is successfully generated, you will be directed to a new page. Click “Back to home” button to finish generating this playlist.

Step 3b: Managing Playlists



After generating a playlist, you will be directed to the home screen. Click the “Manage Playlists” button to manage the playlists you have generated so far.

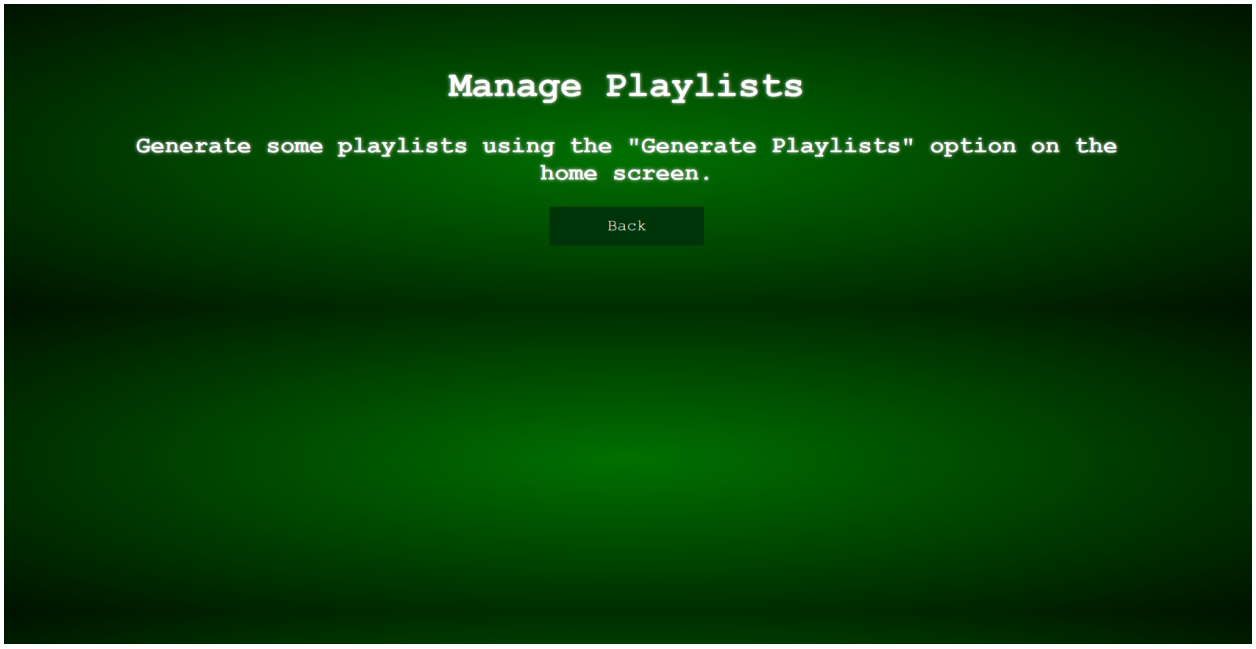


Continuing the example above, after generating a playlist called “I love you Baby Moon”, this playlist will be saved in the Manage Playlists page.

In this page, you can

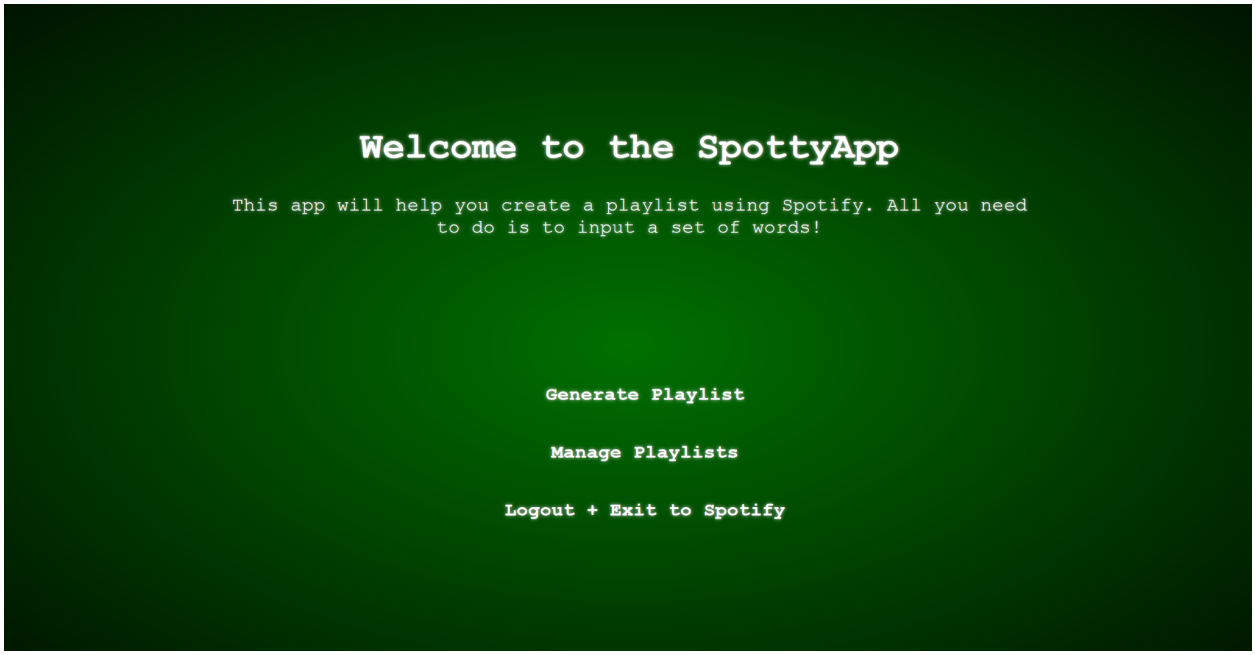
- Save a playlist to your Spotify account using the “Save to Spotify” button.
- Delete all playlists using the “Delete all playlists” button.
- Go back to the home screen using the “Back” button.

By clicking to “Save to Spotify”, the playlist you saved will be saved in your playlists of your Spotify account. You can remove the playlist from your Spotify account using Spotify directly.



If you delete all playlists, all playlists will be deleted and you can generate more playlists by clicking the “Back” button to go back to the home screen, then go to the Generate Playlist page to generate more playlists. See step 3a.

Step 4: Logging Out



To logout of our application, simply click “Logout + Exit to Spotify” button to logout of SpottyApp and go to Spotify.

Thank you for using SpottyApp! Don’t forget to save your playlists to your Spotify account!

References

11, C. C. onS., & Coyier, C. (2018, September 11). *Old timey terminal styling*. CSS. Retrieved December 6, 2021, from https://css-tricks.com/old-timey-terminal-styling/.