Technical Stack & Database Model

Assumptions: This will be a web-application.

Front-End Languages
Technology: JavaScript will be the programming language used as it is the core programming language of the web along-side CSS and HTML.

Libraries/Frameworks
Technology: React will be used as it is light weight, it is simple to test individual components and its hook system allows us to be explicit when we want to re render components making for a better user experience. A simple library like jQuery may also be a good solution here, but maintaining an organized architecture may be more difficult. Frameworks like Angular are less flexible when designing the front-end architecture as they are template driven and may be overkill for such a simple application.

Styles
Technology: Bootstrap will be used for styling as it is light-weight, provides out of the box styles/widgets, and is good for creating responsive/mobile-first web applications without having to define our own breakpoints. This will make for developing a professional looking application quickly. Bootstrap's predefined classes will minimize custom style sheets and encourage consistency throughout the application provided developers take advantage of them where they can.

Back-End Language
Assumptions: the application may need to handle a lot of simple requests in the future.

Technology: Node.JS will be used as the back-end language since the client side will need to interact with the back-end/database frequently while users interact with the application. The primary use for the app will consist of favoriting/unfavoriting songs, thus updating their favorite song list in the db. Node.js' single threaded non-blocking nature works well for this since requests are simple and not computationally heavy. Furthermore, if the number of simultaneous requests increases as usage increases, its async nature will ensure minimal blockage.

Database Model
Assumptions: NoSQL will be used instead of SQL with the following assumptions: 1) we're not sure if the application will be changed or have new data types/data fields since the application is in early stages and all requirements may not be realized.

Technology: MongoDB will be used as the DBMS as it allows for flexible Schemas. This is ideal considering the data we want to capture for each song and user may change as the application is still in early development. Three data models will be created and used to meet the user's requirements: User, FavoriteSongs, and Songs.

Users Collection
Will store registered users so that upon being authenticated, a user will be given access to *their* list of favorite songs.

```
Users
 {
  "_id": ObjectId //unique identifier or primary key
  "email": String
  "userName": String
  "password": String
 }
```

FavoriteSongs Collection
A user's Favorite Song will be stored in the FavoriteSongs collection. Each record in FavoriteSongs will contain the userID (reference/foreign key) of the user who favorited a song, the songID (reference/foreign key) of the song they favorited, and an _id that works as the unique id/primary key for each record in the collection.

```
FavoriteSongs
 {
  "_id": ObjectId //unique identifier or primary key
  "userID": String //reference or foriegn key
  "songID": String //reference or foreign key
 }
```

Song Collection
The purpose of the Songs table is to store a song's information that has been favorited by a user. A song's information will be pulled into each song of the user's list of favorite songs when requested.

```
Songs
 {
  "_id": ObjectId //unique identifier or primary key
  "artist": String
  "album": String
  "genre": String
 }
```

Stored Functions / Stored Procedures

Two Stored Functions will be used by mongoDB:

addFavoriteSong (@userID, @songID) => ()
When a song is added to a user's favorite list, addFavoriteSong() will check if the song is not in the Songs table, the song will be added to the Songs collection with the attributes required by the user: id, artist,

album and genre. The song will also be added to the FavoriteSongs collection with the attributes userID, songID and an auto generated id. NOTE: Depending on how songs will be added (e.g. a star button next to songs on the top 100 songs lists) it may be a good idea to implement 'toggle' logic that removes the record from the FavoriteSongs list if it already exists and adds it to the FavoriteSongs list if it does not already exist.

NOTE: I'm not sure if there is an equivalent of SQL Server's views in MongoDB, if there is that may better suit this function's purpose

getFavoriteSongs (@userID) => all user's favorite songs

When navigating to the FavoriteSongs page, getFavoriteSongs will return an array of the current user's favorite songs. getFavoriteSongs will find the user's favorite songs with logic equivalent to an 'INNER JOINING' of FavoriteSongs and Songs ON songID WHERE userID is the currentUser's ID.

Methodology for Accessing Data

The Billboard Top 100 songs will be requested via a GET request from the client to an API or endpoint within the node.js application that will provide the Billboard Top 100 songs data. The http GET request will be placed in a useEffect hook with no dependencies to ensure we are fetching only once after the components initial mounting.

In a similar way, the list of Favorite Songs will be fetched via an endpoint within the node.js application (which will then fetch the data from the mongoDB) and will act as the initialized state for favoriteSongs. The request will also be placed in a useEffect hook with no dependencies to ensure we are fetching the data only once after the components initial mounting.

Lastly, a PUT request will be sent to update the user's favoriteSongs via an endpoint in the node app any time the favoriteSongs state is updated within the React application (adding a song or removing a song from the list).

Front-End Architecture

We will use components as the building block for our application. Each component will focus on maintaining the single responsibility rule to ensure modularity and loose coupling.

Starting from the top of the hierarchy there will exist an App Component that will contain some sort of navigation (may include login/logout), followed by a Favorite Songs container and a Billboard Top 100 Container. The Favorite Songs/Billboard Top 100 components will act as containers and will be responsible only for fetching and maintaining the Favorite Songs and Billboard Top 100 state respectively. They will pass this data down to their children components: Favorite Songs and Billboard Top 100 Songs. Each will respectively render a ‘list’ of Favorite Song/Top Song components while passing each song's data as props so that it may display each song’s details. Lastly, the Favorite Song/Top Song component will act as the presentation component for data, but also contain setter functions to update state held in their most parent components containing the state they rely on.

- App Component
    - Navigation of sort
    - Favorite Songs Container (will fetch and hold favorite songs state/data)
        - Favorite Songs (will render a list of favorite song components based on favorite songs data)
            - Favorite Song Component (will render a favorite song component based on provided props)
    - Billboard Top 100 Songs Container (will fetch and hold top 100 songs state/data)
        - Billboard Top 100 Songs (will render a list of top song components based on favorite songs data)
            - Top Song Component (will render a top song component based on provided props)

Testing Strategy

The focus of testing will be to ensure components are rendered properly on the applications initial state and to ensure components are rendered correctly after state changes. JEST and React Testing Library will be used to test the React application's front end through Unit Tests and Integration Tests where applicable.

The only state we are able to change in this application after the initial mounting is the state holding the list of the user's favorite songs (we can add a favorite song or remove a favorite song). We can mock the functionality returned by our getFavoriteSong function after a song is favorited and ensure the components are rendered with the expected value given the state returned by the mock function getFavoriteSong function.

Other testing priorities will include testing edge cases such as rendering an extreme number of favorite songs or when a fetch request may fail (e.g. trying to load 10,000+ favorite songs and ensuring the application can handle it) and making sure components are rendered properly provided a minimal number of songs (rendering different verbiage or displays when there are 0 or 1 songs).

Lastly, E2E testing will be implemented with tools Cypress and Testing Playground. This can be used to ensure the application behaves as expected by testing its entire process/flow from beginning to end . An example of this would be to have Cypress emulate an entire user experience by emulating logging in, favoriting a song on the billboard top 100 list and logging out all while asserting our components are updating correctly, our routes are correct, etc through the emulation.