Feature Request: Multiple Playlists

July 7th 20XX

OBJECTIVE

To give users the ability to update all of their Spotify playlists and save them to Spotify.

BACKGROUND

Currently, Jammming supports the ability to create one new playlist at a time and save it to Spotify. However, updating an existing playlist is not supported. With over 5 million playlists created or edited daily on Spotify, this is key functionality to support.

This feature accomplishes the following:

- Displays a list of the current user's playlists
- Allows a user to select one of their existing playlists. The application will load the name of that playlist and the tracks of that playlist in the playlist panel.
- The user may update the name and/or the tracks of the playlist and click "SAVE PLAYLIST". When the user selects "SAVE PLAYLIST", Jammming will save the current version of the playlist to the user's Spotify account.
- If the user selects a different playlist while the current playlist has unsaved changes, Jammming will load and display the new playlist, and will not save the changes to the old playlist. If the user returns to the old playlist, Jammming will display Spotify's version of the playlist, not the edited version.

TECHNICAL DESIGN

Retrieve and Display Playlists

A new component, **PlaylistList**, should be created. This component, on render, will retrieve a list of the current user's playlists.

We will need to initialize *state* for *PlaylistList* to contain a key for *playlists* that defaults to an empty array.

To retrieve playlists, we will create a new method, **Spotify.getUserPlaylists()**, that hits the https://api.spotify.com/v1/users/{user_id}/playlists endpoint. As we can see, this endpoint requires the user ID.

To retrieve the user ID, we will hit the https://api.spotify.com/v1/me endpoint, as we currently do in *Spotify.savePlaylist()*. We will refactor this request to a new method called **Spotify.getCurrentUserId()** (to avoid duplicate code). At the top of *Spotify.js*, we will instantiate a variable called **userId** with no value. Then inside *Spotify.getCurrentUserId()*, we will check to see if *userId*'s value is already set (from a previous call to the function). If it is, we will create and return a promise that will resolve to that value. Otherwise we will make the call to the */me* endpoint and return that request's promise.

Once our *.getCurrentUserId()* is written, we should use it in both *Spotify.savePlaylist()* and our new method, *Spotify.getUserPlaylists()*.

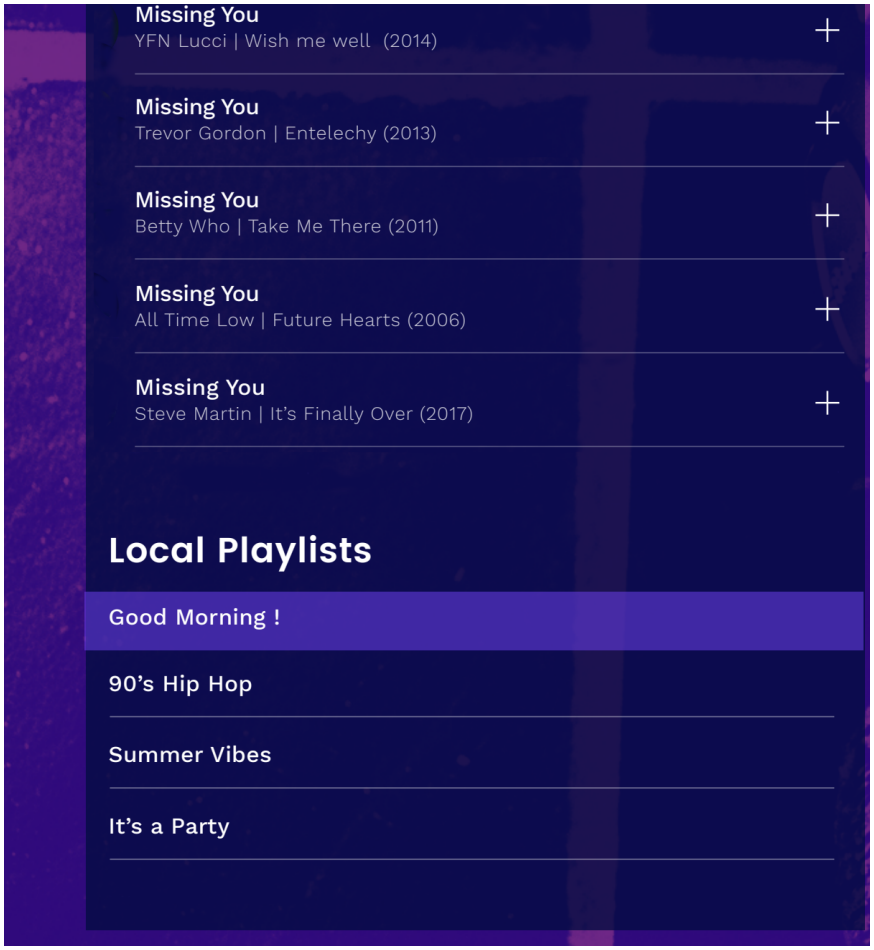
In *Spotify.getUserPlaylist()*, once the user ID has been retrieved, we can make our call to the */users/{user_id}/playlists* endpoint. Upon completion of this request, we will update the current playlist's state to an array of the returned playlists. Rather than storing the entire playlists, we should create and store objects for each playlist that contain the **playlistId** and **name** of each playlist (the API will not return that playlist's tracks, thus we will need to use the playlist ID later to retrieve those tracks).

To call this method at time of render, we should add a *.componentWillMount()* lifecycle method to *playlistList* and call *.getUserPlaylists()* within it.

The *.render()* method of *playlistList* should render a list of **playlistListItem** components (another new component), passing down the *ID* and *name* of each playlist to be rendered.

Finally, *App* should render *playlistList*.

This component should look as follows upon implementation:



Select Playlists

In *App.js*, we will add a method for selecting a playlist. This method should retrieve the tracks of the selected playlist and then update state to the retrieved playlist.

In *Spotify.js*, we will add a method called .**getPlaylist(id)** that will retrieve the playlist with the provided ID. This method should call the https://api.spotify.com/v1/users/{user_id}/playlists/{playlist_id}/tracks endpoint and return a promise that will resolve to the retrieved tracks.

In *App.js*, we will add a method called *.selectPlaylist(id)* which will call *Spotify.getPlaylist()* with the provided playlist ID. When the *Spotify.getPlaylist()* call resolves, we will update *playlistName* and *playlistTracks* on *App*'s *state*.

App.selectPlaylist() should then be bound to the current instance of *App* and passed down to the rendered *PlaylistList*.

PlaylistList will then pass this method down to each *PlaylistListItem*.

Finally, each *PlaylistListIem* should add an *onClick* listener to its root rendered JSX element. This listener should call a method which calls the passed down *App.selectPlaylist()* method with the current PlaylistListItem's *id* prop.

Save Playlists

When saving a playlist, we always make a new playlist in the user's account; however, we only want to do that if the playlist doesn't exist. We will check this by storing the playlist's ID (if it has one).

In *App.js*, we will add a key to our state called **playlistId**, which initializes to *null*.

Next we will modify *Spotify.savePlaylist()* to take a third parameter, **id**. In *App.js* we will pass the *state*'s *playlistId* in as this third parameter in our call to *Spotify.savePlaylist()* within *App.savePlaylist()*. We also need to update our *App.selectPlaylist()* method to update its state to the selected playlist's ID.

Finally, in *Spotify.savePlaylist()* we will check to see if an *id* was provided. If so, we should update the playlist name by calling the https://api.spotify.com/v1/users/{user_id}/playlists/{playlist_id} endpoint with the updated name. If not, we can continue creating a new playlist as normal.

In *App.js*, we currently clear the *playlistName* and *playlistTracks* when a playlist has finished saving. We need to additionally clear the value of *playlistId* by setting it back to *null*.

CAVEATS

App Playlist State

With this implementation, we are now storing three separate values about our playlist on our *App* component. This information is almost always linked together and therefore it may be useful to store one object called *playlist* on the *App*'s *state*. This implementation would make passing information easier and potentially improve readability. However, it also makes it less clear which information must be present in *App*'s state. This less structured data provides more room for incorrect implementation of future feature requests. With only three properties being stored, we have decided to continue storing this information as separate *state* values. However, this should be re-evaluated if we decide to store even more playlist information on *App*'s *state*.

Asynchronous Existing Playlist Save Requests

In the initial implementation of Jammming, every playlist save necessitated a playlist to be created with the specified name and then the tracks to be saved to the playlist after that playlist

creation completes. With the added functionality of saving a pre-existing playlist, we no longer need to create a new playlist as the first step every time. As a result, when saving a pre-existing playlist, we could send two simultaneous requests: one to save the updated name, the other to save the updated track list. This would result in a more complicated save playlist flow and would result in marginal returns in time efficiency when saving a playlist. Since the user is not prevented from continuing to interact with Jammming during playlist save, this efficiency would result in no change to user experience. Therefore we will not implement this functionality since the loss in code readability outweighs any gain in user experience. This would be a premature optimization.

Excess Playlist Saves

Currently, Jammming will save the displayed playlist name and tracks on every "SAVE PLAYLIST" click regardless of whether or not data has changed. This could result in unnecessary requests to the Spotify API (especially considering Playlist names are not likely to change often). As noted in the previous caveat, though, extra requests to the Spotify API will not result in any change to the user experience. If we find in the future that users are intentionally abusing this functionality and we run the risk of throttling the Spotify API, we will implement this functionality. However, at this point, this would be another premature optimization.