Jammming Feature Request

Global Play: One Playlist to Rule Them All

Polly Siegel, Codecademy Final Project
March 27, 2018

OBJECTIVE

Extend the playlist management capability of Jammming to allow it to manage a user's master playlists. Those playlists can then be uploaded to other streaming services, allowing users to have one common set of playlists that work across all streaming services.

BACKGROUND

In the past few years a number of streaming services have become available in addition to Spotify, including Apple Music, Google Play, Prime Music, and Pandora, to name but a few. Each of these services has its own playlist and favorites management functions. For users who subscribe to or use multiple streaming services, playlist management is a nightmare, since playlists cannot be shared between the services. As a result, users have to manually recreate their playlists on each service they subscribe to. Since not all services have the same music catalog, each streaming-specific version of a given playlist may have only a subset of the songs that another service has.

The Global Play playlist service will allow a user to have a central place to manage streaming-service agnostic "master" playlists, and to generate and upload a streaming service-specific playlist to each subscribed service for a given playlist. The generated playlists will include the subset of songs that that streaming service supports.

TECHNICAL DESIGN AND IMPLEMENTATION

Overview

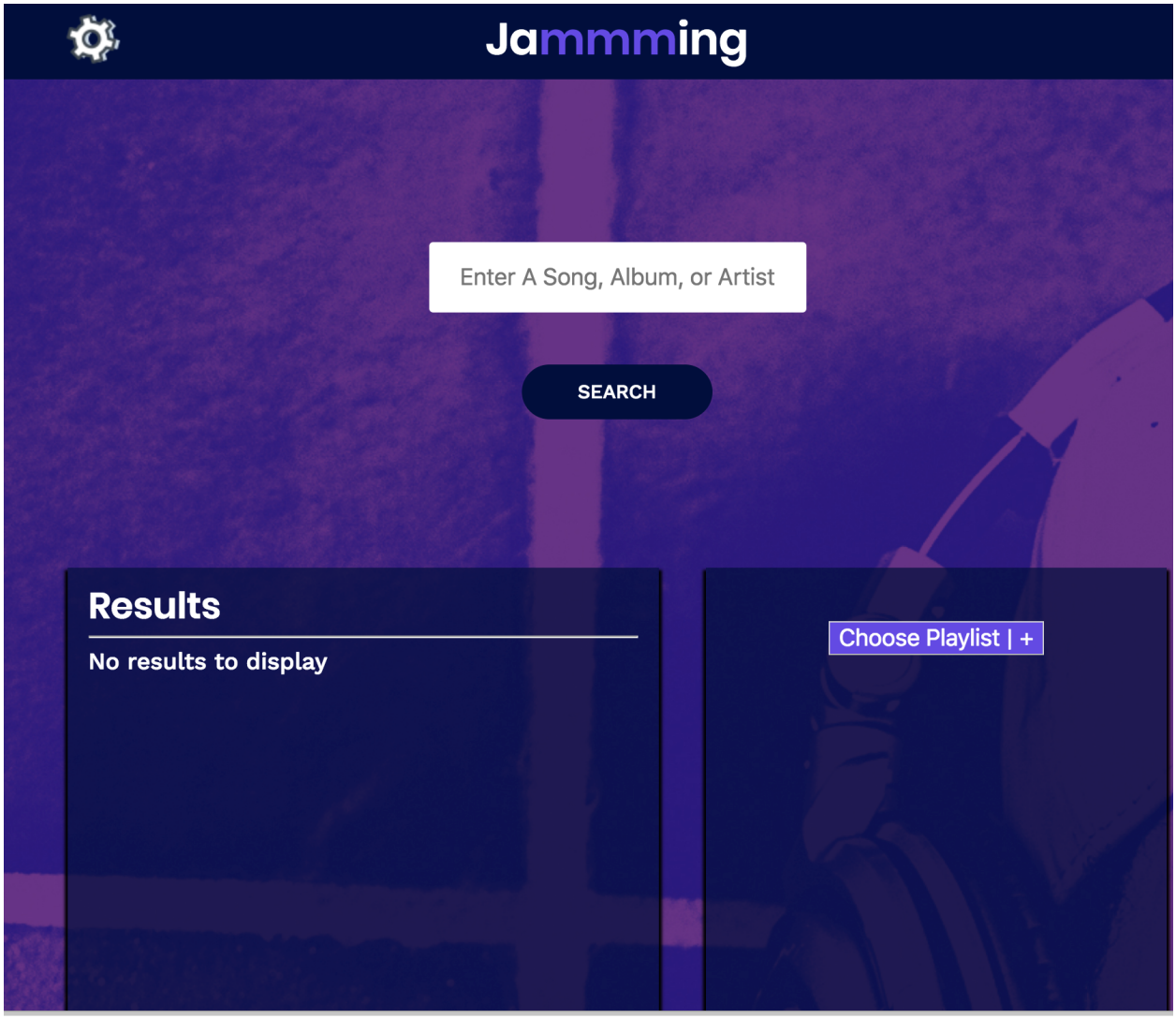
The proposed new feature is a complex extension to Jamming, and as such, there are several new pieces that will need to be added to implement the new functionality:

1. Configuration of and authentication with multiple music services.
2. Retrieval of existing playlists from Spotify.
3. Playlist selection.
4. Identification of track availability from configured music services.
5. Saving of updated playlists to configured music services.

Because of the complexity, development must be done in phases. A phased implementation plan is outlined later in the document.

UI/UX Design

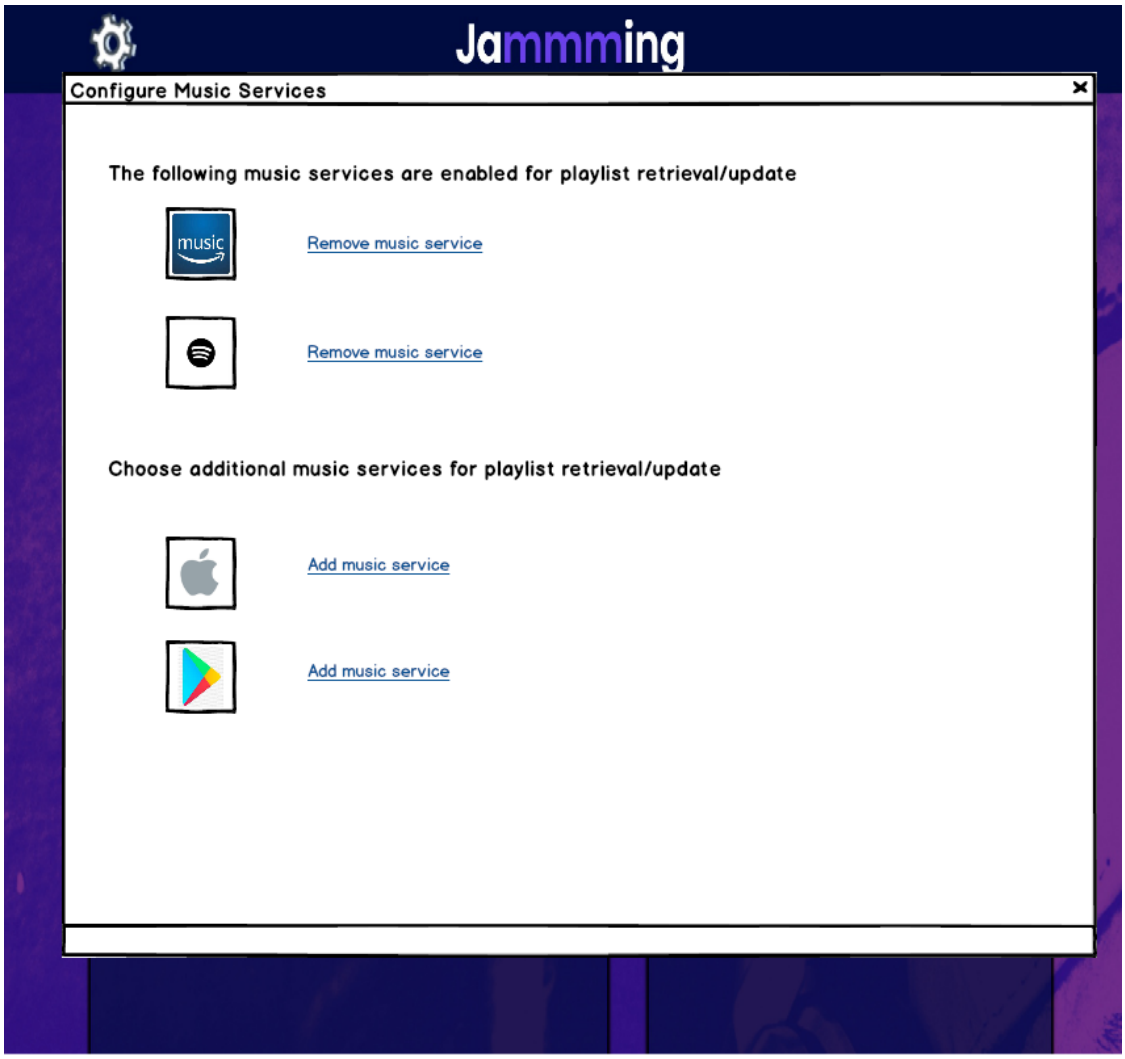
Main Screen Overview



Changes:

- A configuration icon is added in the header. Clicking on this icon allows you to configure the music services you want to connect to
- A choose/add playlist dropdown is added in the right-hand column to allow you to choose from existing playlists or add a new one
- The Save Playlist button is suppressed until a playlist is chosen. Once chosen, the button will appear below the playlist with either SAVE PLAYLIST or UPDATE PLAYLIST depending upon whether a new playlist is being created or not.

Configuration Screen

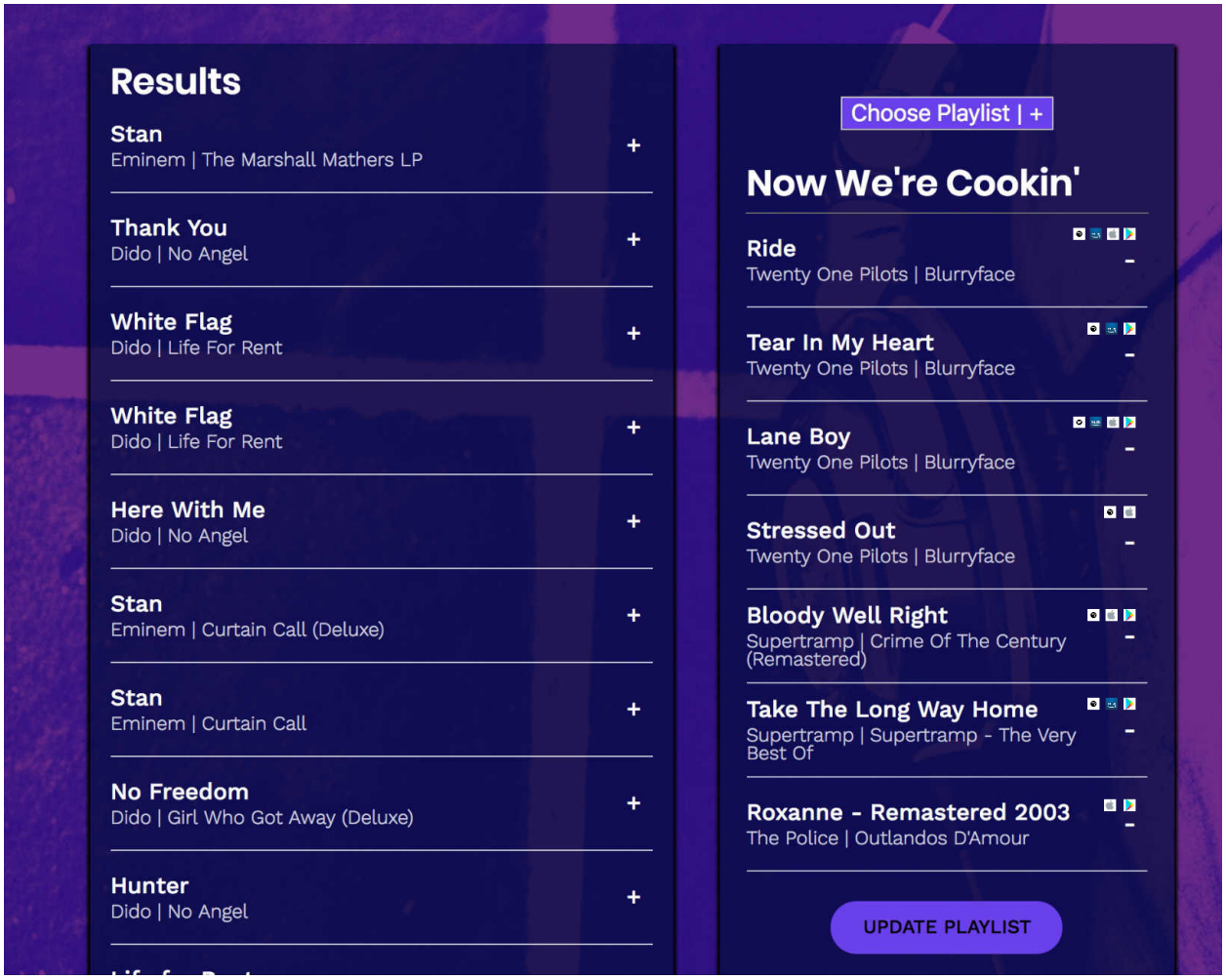


UX:

- When the gear icon is clicked, the configuration screen pops up as an overlay on desktop; on mobile, the configuration screen replaces the main screen (with a back arrow (<) to take you back to the main screen)
    - Styling/design (colors/fonts) are similar to the main screen -- mockup demonstrates functionality only
- Music services that are already enabled are indicated as being enabled
- Music services that are not enabled include a link that will take the user through a dialog to enable the service
    - This dialog will prompt you to log into the music service to establish your credentials

- Upon successful addition, the link will change to Music service enabled

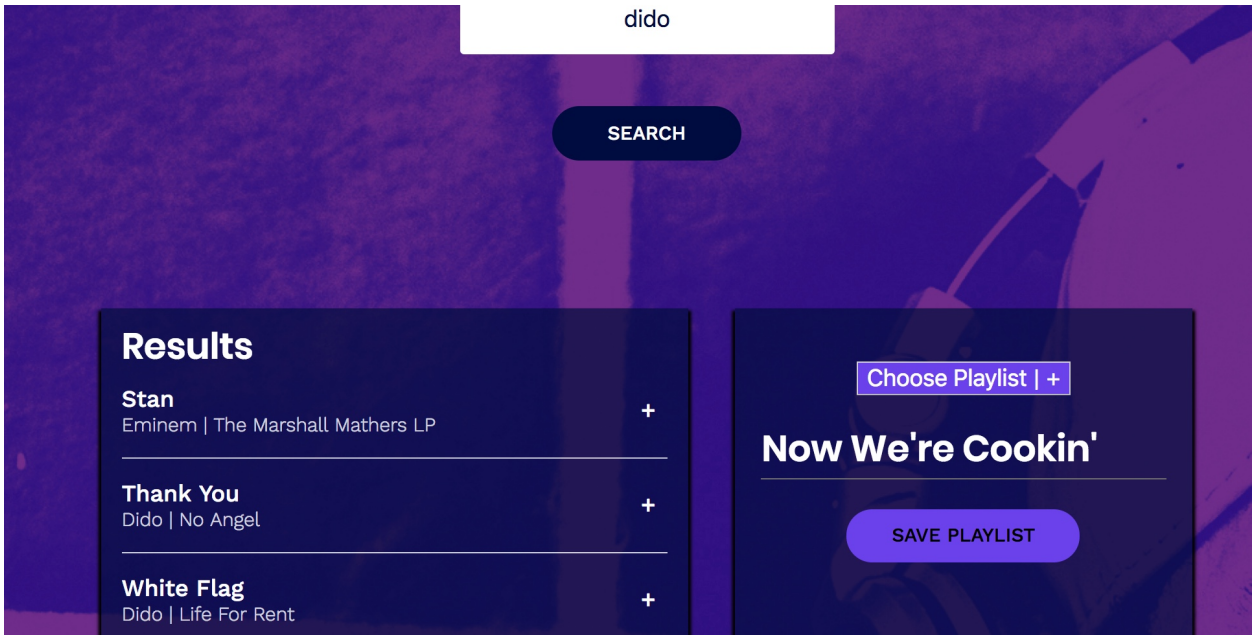
Update Playlist Functionality



UX Design:

- When an existing playlist is chosen from the Choose Playlist dropdown, the chosen playlist displays, along with the existing songs in the playlist
- Next to each track, above the - icon, icons appear for each configured music service that has the track available.
- Clicking on update playlist will save the playlist to all configured music services.

Save Playlist Functionality



UX Design:

- When the + icon is chosen on the Choose Playlist dropdown, the user can type in a new playlist name
- After adding tracks, the user can click on Save Playlist to save the playlist

UX Flow

The revised flow is illustrated in this video **https://youtu.be/XzJZ2XnojlY**

UI

Styling and CSS for new screens and elements should use the same fonts, font sizes, font weights, colors, and layouts as the existing implementation. Flexbox should be used for placement of any new divs to ensure that the revised application remains responsive.

FE (React/JS) Implementation

New Data Structures/Modifications to Existing Data Structures

New state will need to be added to Apps.js, and existing state modified to support the requirements as outlined in the table below:

Name	New/Update	Type	Use
`playlists`	New	Array of playlist names (strings)	Stores playlists. Initially populated from existing stored playlists
`musicService`	New	{musicServiceID, musicServiceName, musicServiceIcon, musicServiceMethod, configured}	Information to be stored for each music service. musicServiceID is a unique integer value starting at 0, and is used to provide a stable index into the music services the app supports. musicServiceName is a string identifying the service for display (e.g. “Google Play Music”) musicServiceIcon is a path to the graphical icon for that service musicServiceMethod is the method to call to invoke the service. (Note that this may need to get expanded into multiple methods once we get into implementation of additional services.) configured is true or false depending upon whether the user has configured this service
`musicServices`	New	Array of musicService objects	Used to access and display information about supported music services. (Used by the Configuration component and by the Track Component.)
`playlistTrack`	Update	Augment to include music services	Store track information for each music service subscribed to. For each track, add map from music service to music service-specific track id.
`trackUris`	Update	Augment to multidimensional array indexed by music service	Add map from music service to URI ({musicService: uri}) for each track. Store track URIs for each music service, similar to playlistTrack

New Components

Configuration Component

We will need to add a new Configuration component, which will be called from App.js:

```
<div>
 <Configuration onClick=/>
 <h1>Ja<span className="highlight">mmm</span>ing</h1>
   ...
```

Configuration Component Structure

The configuration component will perform the following functions:

1. Render the configuration (gear) icon in the header
2. Render a configuration overlay which is normally hidden (display: none), until the configuration icon receives an onClick event.
3. On onClick event, change the display of the configuration overlay form to display, with the z-index set such that the overlay will be on top of the main screen.

Configuration Header Icon

Within Configuration.js, we'll need to render the configuration icon in the page header. We'll need to include the styling for the icon in Component.css.

HTML:

```
<img className="configIcon" alt="Configure" src="icon_URL">
```

CSS:

```
.configIcon {
   position: absolute;
   left: 210px;
   top: 5px;
}
```

A nicer icon than in the mockup should be chosen or created.

Configuration Screen

The configuration screen should follow the overall structure of the mockup, and be similar in look and feel to the main site. The screen will be rendered by the render() method within Configuration.js, which will use a map function to render each music service through a new MusicService component, e.g.:

```
class Configuration extends React.Component {
 render() {
     return (
       <div className="MusicServices">
         {
           /* Renders the list of configured Music Services */

           <h2 className="configHeading">The following music services are enabled for
playlist retrieval/update</h2>
           this.props.musicServices.map(musicService => {
             return (
               if (musicService.configured) {
                  <MusicService musicService={musicService} />
               }
             );
           })

           /* Renders the list of not-yet-configured Music Services */

           <h2 className="configHeading">Choose additional services for playlist
management</h2>
           this.props.musicServices.map(musicService => {
             return (
               if (!musicService.configured) {
                 <MusicService musicService={musicService} />
               }
             );
           })
         }
       </div>
     );
   }
 }
}
```

Although it's inefficient to traverse the list of music services twice, it's necessary since the UI groups the services by whether or not they're configured. A later optimization would be to split the services out into two separate arrays, one for configured music services and another for unconfigured music services. But since the list of music services will be a small number, for now we'll just go with a single array.

MusicService Component

The MusicService Component will be used to render each music service on the configuration screen, and will be used to update information about the state of each service. The logic is outlined below:

```
class MusicService extends React.Component {
 render() {
     return (
       <div className="MusicService">
         {
           <div className="musicServiceIcon">
              <img src={this.props.musicService.icon} />
           </div>
           <div className="configMessage">
             if ({this.props.musicService.configured}) {
                <a href={this.props.musicService.musicServiceDeauthorizeMethod}>
                  Remove music service
                </a>
             } else {
                <a href={this.props.musicService.musicServiceAuthorizeMethod}>
                  Add music service
                </a>
           </div>
         }
       </div>
     );
   }
 }
}
```

musicServiceInterface base class

We are extending the base implementation of the original Jammming app to include multiple services, we want to define a parent class with methods that will get overridden by music-specific services. Each music service-specific JS file (e.g. Spotify.js) will need to implement the methods defined by the parent. Since the implementation depends on the specific services API, the interface class can't be generalized, but the parent class should serve as a template and documentation for creating the code for new music service implementations.

The parent class should look something like:

```
class musicServiceInterface {
   getAccessToken() {
        console.log("Authenticate with music service");
        console.log('getAccessToken not implemented for this service");
   },
```

```
   search(term) {
      console.log("retrieve track from music service");
      console.log("search not implemented for this service");
   }

   savePlaylist(playlistName, tracks) {

       console.log("savePlaylist not implemented for this service");
   }

}
```

Music Service support

We will create a new JS file and class for each new music service we want to support. Each new music service will extend the musicServiceInterface class, and will implement the methods defined within that class, similar to how Spotify.js is implemented. In the initial implementation, support for the following services will be provided:

Music Service	Task	Class/Fille Name	API Information
Apple Music	Create	AppleMusic/AppleMusic.js	https://developer.apple.com/library/content/documentation/NetworkingInternetWeb/Conceptual/AppleMusicWebServicesReference/
Google Play	Create	GooglePlay/GooglePlay.js	https://www.npmjs.com/package/playmusic
Amazon Prime Music	Create	Prime/Prime.js	https://developer.amazon.com/sdk-download
Spotify	Update	Spotify/Spotify.js	

Not all services are easily supported. Google Play does not offer a third-party API, so we will need to use an open source Google Play API. Amazon Prime Music will take a bit more sleuthing to find an API… we may need to reverse engineer it. Therefore, we will add Apple Music as the first additional service, and then move on from there.

Spotify.js will need to be modified to incorporate the new functionality defined by this upgrade.

Modifications to Existing Components/Classes/Methods

App

musicServices

Track

The Track component includes logic to render tracks from the displayed playlist. Track will need to be modified to iterate through the music services that include the track and display the music services which include the track above each track, as indicated in the mockups.

```
render() {
   return (
       <div className="Track">
           /* Display music services that have this track */
           {this.renderMusicIcons()}

           /* Display of existing track information doesn't change */
           <div className="Track-information">
               <h3>{this.props.track.name}</h3>
               <p>{this.props.track.artist} | {this.props.track.album}</p>
           </div>
           <a className="Track-action">{this.renderAction()}</a>
       </div>
   )
};
```

A new method, renderMusicIcons(), will be created that will iterate through the music services associated with the track (through a map function) and display icons associated with services that found this track.

```
renderMusicIcons() {
   /*
    * Either removeTrack or addTrack will be set by the parent, not both.
    */
   return (
     <div className="Music-Services">

         this.props.track.musicService.map(musicService => {
            return (
               <img className="musicServiceIcon" {musicService.icon} />
            )
        });
    )
};
```

(Of course we'll need to add a bind this statement within Track for this new method.)

Track.css must be modified to define the CSS for the Music-Services class, which will format the music services as shown in the mockup.

The TrackList component must be modified to pass musicServices information to Track for use within renderMusicIcons().

Playlist

The Playlist component will need to be modified to implement a drop-down containing the user's defined playlists from the master playlist service. This will require adding a method that retrieves all playlists from the master music service (or database), and store them in the playlists state within App.js. The JSX/HTML code will need to be modified to output a drop-down that is populated with those playlists. The code will need to be modified to handle the user interaction with the dropdown/+ icon, to support the user interaction defined in the mockups. musicService information will need to be passed to the Playlist component as well, so that this information can be made available to child components.

IMPLEMENTATION PHASING

The implementation will be done in phases, as the feature is quite complex:

Phase 1: Add playlist management functionality using Spotify as the sole source. We will use Spotify as the central source for the master playlists, and we will only read playlists from Spotify. We will add the ability to retrieve and update playlists, and save them back to Spotify.

Phase 2: Add additional music services configurations, and the ability for the user to select which services they want to use by default. Add automatic authentication to those music services upon starting the app. Change Save Playlist to save playlists to Spotify plus to any configured music services.

Phase 3: After retrieving a Playlist, retrieve corresponding tracks from configured playlists. Mark the master playlist with which services have those tracks. Display service-specific icons on the playlist next to each track for each service that supports that track. For each service, save playlist will now only save those tracks that the music service supports.

Phase 4: Integrate with an application-specific database. The first time a playlist is read in, the playlist name/tracks will be stored locally and saved for future use. Each time a playlist is modified or a new one is created, it will compare the track list with which music services support that track (as music services can add/delete music from time to time).

CAVEATS

Matching artists/tracks: Assume exact match on artist/track/album. If the music metadata doesn't match from one service to another we won't include the track. Later versions of the feature will attempt to find alternate matches (for example, match track and artist first; if that fails, match track, e.g.: take Your Song from Elton John's Greatest Hits if Your Song on the album Elton John can't be found).

Initial playlist implementation: For the initial implementation of this feature, we'll assume that the user only has playlists in Spotify (or the user will create them from the app). Later on we'll allow the user to load playlists from other music services and use them to create the master for a given playlist.

Application performance: Having to interact with n music services to pull in playlists each time the app is invoked may slow down the application. We will assess the performance once the initial feature has been implemented. If performance is a problem, then we'll store the playlist modification date within the app, and compare the date modified from the external service before pulling the playlist. Playlists will only be pulled when the remote playlist is newer than the local one.

FUTURES

We may find that we need to create our own centralized service or may choose one of the other streaming providers with an extensive music catalog and extensible metadata as the basis for our Global Play playlist service.

The application can be extended to read in playlists from all services, making it easier to migrate playlists from one service to another.