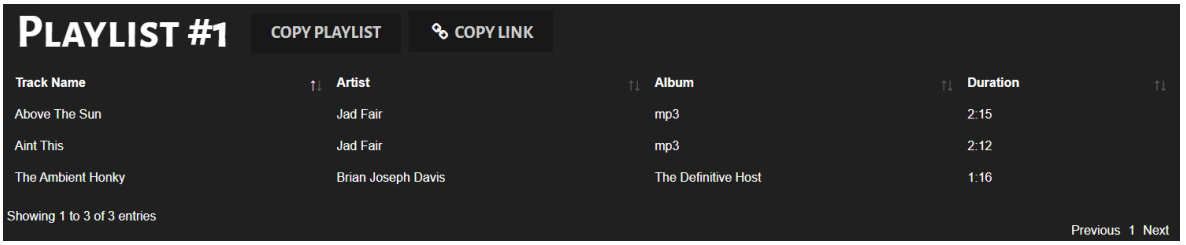
Playlist Features Design Report

Feature Introduction

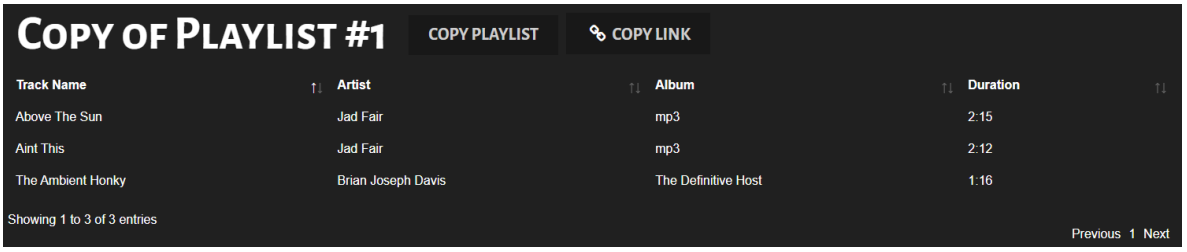
The feature we implemented into our music web application allows users to share playlists with others.

When viewing a playlist, users have the option to copy the playlist into their playlist library. This feature allows users to share playlists with others and for the other users to create a copy of their playlist as their own, allowing them to personalise the playlist by adding more music and accessing the playlist from their own library.

A copy link option has also been added for users' convenience and ease of use, as they can simply click the button and paste the playlist link in a message (or through other means) to send the playlist to others.



Clicking on the 'COPY PLAYLIST' button on the above playlist would result in the following playlist to be created in the user's playlist library:



Key Design Decisions

The Single Responsibility Principle

Throughout the process of implementing this new feature, a fundamental design principle that was applied to every aspect was the Single Responsibility Principle. Every module created in the service layer, repository pattern, and domain models, all had this design principle applied, ensuring that each module only had responsibility over one part of the functionality. Applying this principle, in our case, allowed us to reuse code (modules) for other modules and use cases. This also made the code easy to understand and debug, as the functionality of each module is simple to understand and code.

Application Architecture

Following the Single Responsibility Principle, as well as the structure of the music application beforehand, we ensured that we continued to follow the application architecture design shown in lectures, i.e., the onion architecture. The entire application is split and defined into different layers: the middleware, the request handlers/view layer, the service layer, the repository pattern, and the domain models. These layers were defined to only interact with their adjacent layers, i.e., only the service layer can interact with the repository interface and domain models, only the view layer can interact with the service layer, and only the middleware (i.e., Flask) can interact with the view layer, and vice versa. This was implemented heavily into our code and our features.

Dependency Inversion Principle

The benefits of using the application architecture as defined above follow the Dependency Inversion Principe, which states that high-level modules should not depend on low-level modules - both should depend on abstractions. We ensured that we reinforced this while implementing our feature in the program, meaning only modules from adjacent layers could interact. This is shown in our code, for example, when interacting/accessing data from the database repository. Requests for data first went through a function in the service layer, which would then interact with an abstract repository interface and, finally, the database repository. Applying the application architecture and, as follows, the dependency inversion principle to our application has helped increase the modularity of the program and made the code easier to test and maintain.