DESIGN REPORT

Feature overview

We have implemented a user friending system, which allows users to add each other as friends. Users must first send a request to the user they wish to add. This protects user privacy by only allowing friending between users if both sides provide consent.

If the other user does not accept, the sending user can cancel their request anytime they wish.

Once users are friends, they can remove each other at any time. They do not need permission from the other user for this, allowing a user to revoke consent towards a friend at any time.

Rationale for including feature

The ability to add each other as friends opens a range of new functionality for the application.

For example, a feed can be implemented which displays friends' activity, like which tracks were liked, or reviewed, or shared between friends. Adding this functionality allows users to share experiences with their friends allowing a larger uptake of the platform.

Implementation

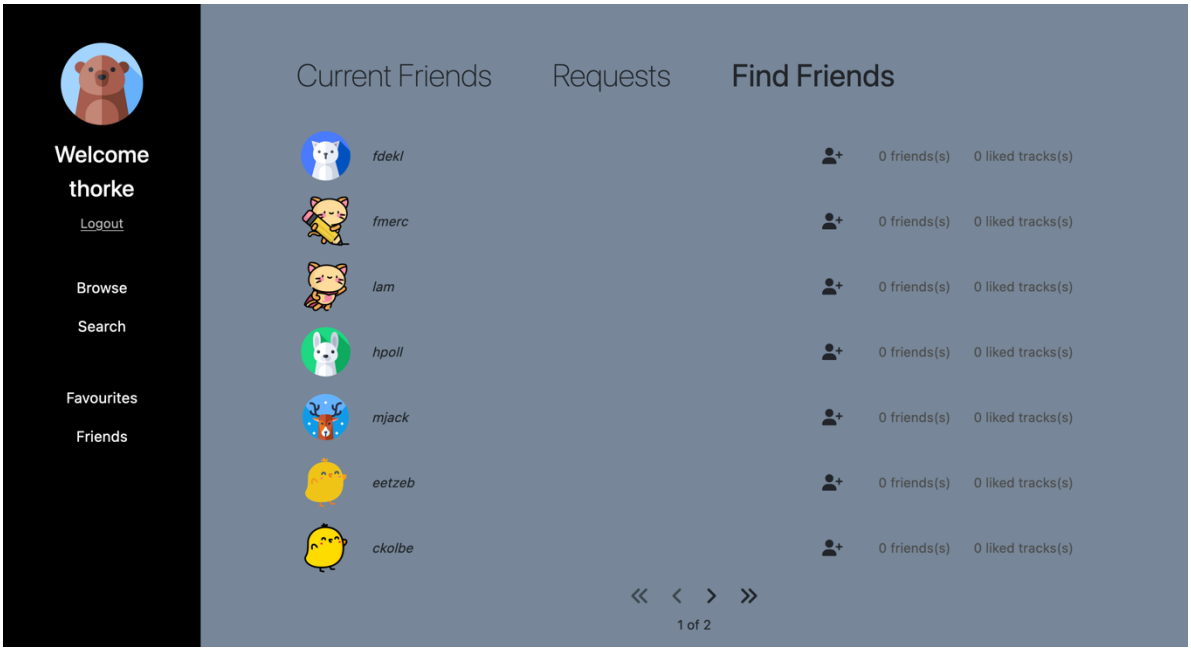
Domain Model

The feature was implemented by creating two new attributes for the User domain model. One holds the requests sent to the user, and another holds the friends they have. The attributes have get methods to view each property, and set methods to add new users.

This method allows easy access to each user's friends and requests, which will be useful for other blueprints that implement functionality dependent on whether two users are friends or not. The example "activity feed" discussed earlier is a scenario where these attributes would be highly useful.

Views Layer

The UI is implemented by creating a 'friends' section in the side menu. This logically partitions the platform functionality, while still providing direct access to each section from any page.



As shown above, the friends section clearly displays current friends, requests, and all other friends a use can send requests to. For each page, the endpoint requests the service layer to provide the users to displayed in each page. This information is displayed through a generic "user list" template, which slightly alters the UI based on which page is being visited.

For example, current friends are look like this:

While current requests look like this:

Using a generic template minimises redundant code, while allowing "hot swapping" of UI elements. The same user list template could be used by any other blueprint that needs to display user information without writing additional code.

It also applies any styling changes throughout the platform maintaining consistency in the UI.

Service Layer

The service layer requests the database for the current user's friends, requests, and all friend-able users. The friend-able users are computed by filtering all users in the system, and removing already friended users, or users that have sent requests to the current user. This logic is implemented in the service layer.

Once the service layer has determined which users should be displayed, it creates data transfer objects which includes all the information required to be displayed. This includes

friend and favourite track counts, and whether the current user has already sent a request to this person. This information is passed to the UI to determine which buttons should be displayed (e.g., send or cancel request, etc).

If no request has been sent, a user looks like this:

This is the same user, after the send request button has been clicked:

The user of data transfer objects allows all required information to be collected and packaged nicely for the UI to display. It avoids cluttering the User model with unnecessary attributes that are only needed by specific endpoints.

Repository Layer

Users can add any other user as a friend. This creates a many-to-many relationship for friends and requests, therefore requires additional association tables between users. Any changes to user attributes must happen in the repository layer. This is important for changes to persist in the database as otherwise any friends added will not be visible outside the scope of that user object.