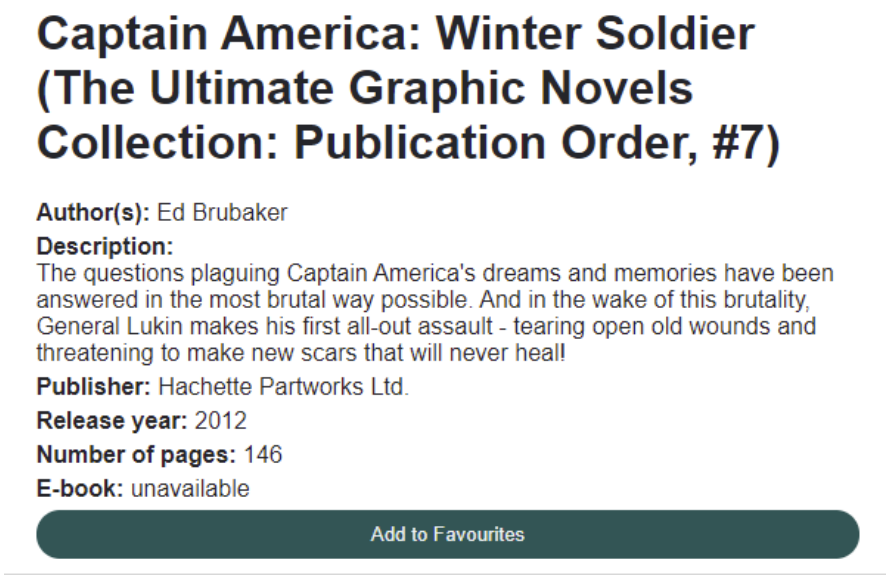
Favourite books function

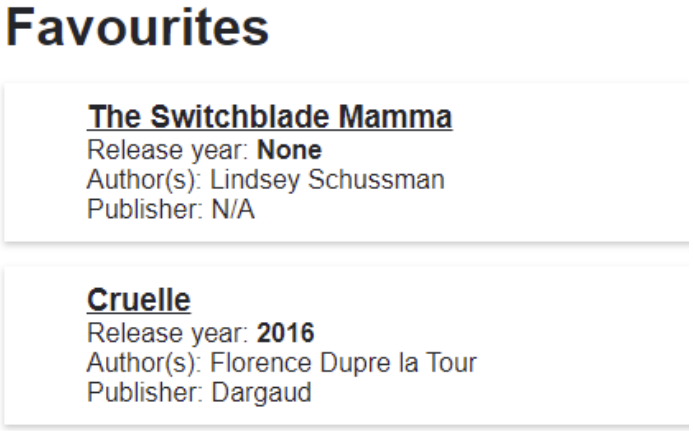
Introduction

The user must be logged in for the favourites function to be usable. If the user tries to access their favourites page without logging in, they will be redirected to the login page.

When logged in, the library app allows the user to add books to their favourites list. The user is able to view a webpage that shows the books previously favourited. Through this webpage, the user can access webpages of their favourited books which shows more detailed information and reviews. The favourites page also allows the user to remove books from their favourites list without having to access the book webpage.



Add to Favourites on a book webpage.



A user's favourite books list webpage.

The favourites list of each user is unique. The favourites list of a user is stored when they logout.

Design decisions

Like the rest of the app, we added our new function with consideration to the web application architecture.

We wanted each user to have a unique list of favourite books. To create this relationship, we added a list ‘favourite_books’ to the User class in our domain model. This list will contain Book objects that the user favourites. Methods for getting the user’s favourites and adding / removing favourites were also added, so that they can be used by the repository.

The repository pattern (abstract repository interface and memory repository) is dependent on the domain model classes. In our repository pattern, we add similar methods which call upon the new methods we created in the domain model and returns the value. The repository methods make it easier for the service layer to interact with the domain model (such as user favourites), as they allow the service layer to access the domain model using primitive data such as book_id instead of a Book object. Since the service layer often sends and retrieves primitive data, this helps prevents unnecessary conversions between primitive types / domain model classes.

The service layer is dependent on the domain model and repository pattern. It is responsible for getting / updating data from the repository and sending it to the view layer as primitive data (usually dict form). In the service layer, we created methods that call methods in the repository (which gets / alters data in the domain model / User object). The methods both parses (as parameters) and returns primitive data. This is so the view layer has no dependence on the domain model, so changes to the domain model doesn’t directly affect the view layer. Any changes that affect the domain model / repository pattern won’t affect the view layer; only the service layer will need to be changed.

Finally, the view layer is able to use the data received from the service layer’s methods to render html templates. For example, the view layer can show the users favourite books on an html page by calling a service layer method that gets a users favourite books (from the repo which gets it from the User object in the domain model). Likewise the view layer can process button clicks for when the user wants to add / remove favourite books (by calling the service layer methods).