Book Shop

Our Cool New Feature by: Tim and Ben

Our extended feature is to implement a book inventory with the data we already have provided. This implements a 'BookInventory' object in the domain model which holds data for pricing and stock count alongside the 'Book' objects. With this extended data we added some features to the home page, book pages and implemented the use of a user cart and purchase.

> *We decided that our implementation for the BookInventory would be to generate random pricing and stock counts for each book as this data has not been provided within the json files.*

Book Prices and stock:

As part of our new book shop feature, we implemented the use of prices and stock within a book inventory object. This book inventory object is stored within the memory repository with an added method to retrieve it when needed. With our implementation, upon running the application these prices and stock counts are randomized before being populated into the book inventory in the memory repository. Originally the memory repository would be populated by adding books to a list and a dictionary through the populate method in the memory_repository.py file. With our new feature, books are populated into the book inventory alongside the book index and catalogue. Accessing the new price and stock count numbers can be done by using the get book inventory method. In the service layer of our webapp, originally the book objects from the book catalogue are turned into dictionary objects that get passed to the front end. With our new feature we developed services to access the book inventory from the repository and then use the find methods to get the book price and stock. On the front end of the web app, the book price is shown when browsing the catalogue pages. Each book also has a In stock/ out of stock header with corresponding css colors green and red. This color changes when a book goes out of stock. When clicking to view a book, the user will be able to see the exact number of items in stock of that particular book. There is also the implementation of a sale banner and a discounted price with the original price crossed out underneath.

Discounted and Featured Books

Our implementation of the book shop also includes the feature of discounts on the home page. This is also randomized when populating the memory repository. Three books are

chosen at random to be discounted by 50 percent. When implementing this data however, we made sure to separate it from its original price within the book inventory as a different attribute. This allowed us to pass both to the front end with discounted books having the original price crossed out underneath it. In the services layer we developed a method that searches the book inventory. Any books with the discount attribute set to 0 are passed as normal price, the three random discounted books will have a value of 50. Something to note with the implementation is the decision to use integers instead of say 0.5 as a prevention of type errors when passing these values to the front end through the services layer. The web app also randomly features three books from the catalogue at the bottom of the web page within the footer. This is done through the service layer also to pick any random three books and display their price and in stock/ out of stock banner. The user can then choose to click on any three of the displayed books which will direct them to the corresponding book page. With our implementation of this in the footer, we decided not to show this on the login, register, shopping cart, and purchased books pages.

Book Shopping cart:

The book shopping cart allows a user to have their own place to store books that they are going to purchase. The user is able to add and remove books from the cart and they can view what books are in their cart and how many copies of each book they will be buying. The shopping cart HTML page also displays any discounts that have been applied to books as well as the total price of their shopping cart. I decided to create a new class called Shopping Cart as this would allow me to implement custom methods and also how I stored information related to the cart. ShoppingCart only has one attribute self.__books which is a dictionary where the keys are the book id and the value is the amount of copies of that book being bought. My partner and I chose this way of storing the books because it saved space and also allowed us to reuse code already present in the memory repo. For example all we would have to do in order to get the book is pass the book_id to the repo which can then get the book and information on that book if needed. Having the value stored with the book id key as the amount of copies of a book also helped when calculating the total price of the shoppingcart. The Shopping cart and the User have a composition association relationship as the shopping cart only exists within the User object so when the user object ceases to exist so does the shopping cart. We chose this relationship as it made sense that each user on initialization would have their own individual shopping cart. Then we created separate methods in the user class and shopping cart class that follows the single responsibility pattern. The shopping cart simply is able to handle books and add them, remove them or completely clear the cart. While the user only needs to worry about what book they want to add to or remove from the cart. So in conclusion, the user is able to interact with a ShoppingCart object to add and remove books they would like to purchase.

Purchase:

To give Users the ability to purchase we decided to give the user an attribute called purchased books which is a dictionary that uses the book ids as a key and then the amount of copies of that book purchased as the value stored. We decided to do it this way because the feature of being able to purchase books only requires implementing a few methods only so creating an entire class for it felt unnecessary and the purchased books dictionary only exists when a user does (similar idea to the shopping cart). Also this follows the single responsibility principle because only the user is able to purchase books, therefore it makes sense to implement all purchase functions (apart from the service layer purchase functions) inside the user class. So the purchase feature allows a user when they go to their shopping cart, to have the option to press the purchase books button. Once they press the button first it copies all the books in the shopping cart (since the data stored is the same (key(book_id): value(number of copies))) and stores it inside the purchased_books attribute of the user. Once all the data has been copied, we call the method adjuct_book_stock located in the BookInventory class using the service layer methods, to change the stock count of the books purchased by the user using the book_id and number of quantity data that we have. The change in stock is reflected throughout the web page and is updated. Then the user's shopping cart is cleared and is now empty again. We chose to adjust the stock when user's purchase books instead of when they added the book to cart because in a real world perspective of first come first serve, whoever hits that purchase button first gets the book so if you're too slow you miss out on the book. Another reason is that the user potentially could add books to their cart that they may want to purchase but never actually purchase the book (classic window shopping). So if we choose to adjust the stock when a user adds a book to cart the stock of that book will be held hostage until the user removes the book from cart. With these details in mind, our purchase feature gives users the option to purchase books from their shopping cart.