Part 1:

https://github.com/Deep-Optimum

Part 2:

I copied and pasted our acceptance testing from T2. Due to various reasons, we still have not built a viable product yet. The frontend is still separate from the backend right now and they are tested separately and all functionality works. I actually spent a lot of time writing tests but oftentimes got frustrated because of how the unit test is set up in python. Each unit test is isolated from each other and it is very cumbersome to test databases and endpoints. However, if I don't use unittest configuration to run my test, everything works. Since we had to submit coverage and unit test reports, I had to make stupid changes to all the unit tests. So far everything works fine. The only thing I am concerned about is concurrency. Whether the database can perform transactions among few users still remains a mystery.

(1) As a **student buyer**, I **want** to be able to purchase cheap and used textbooks locally for my classes at Columbia **so that** I do not have to pay extremely high prices on Amazon or Ebay and wait for a week..

My conditions of satisfaction are:

- I can search textbooks by keywords or by category.
- If there are no textbooks I want to find, I can post a request for that textbook.
- I can browse textbooks by category or by distance.
- If there are textbooks I want to buy, I want to be able to contact the seller.
- I want to be able to see the condition of the books via pictures/videos.
- If I decide to purchase the textbook, I want to have buyer protection; meaning I want my money protected. If the seller did not fulfill the agreement, I want to be refunded.
- If the transaction is successful, I will get email notification just for the record.

(2) As a **student seller,** I **want** to be able to post my textbooks that I don't need anymore and sell them locally **so that** I can make the best use of them.

My conditions of satisfaction are:

- I can categorize the textbooks I am selling.
- I can give text-based descriptions along with pictures of the textbooks that I am selling.
- I can set a price for the item.
- I can chat with buyers and set up time and location for local meetups.

(3) As a **student seller**, I **want** to be able to receive my payments securely and promptly after the transaction of the textbook so that I can get paid.

My conditions of satisfaction are:

- After chatting with the buyer to coordinate a time to meet up, I expect the buyer to show up at the location.
- If the buyer does not show up at the location, I can cancel the transaction and have my textbook re-listed on the website.
- If both the buyer and the seller meet in-person and if the buyer agree to the transaction after seeing the textbook in-person, I scan the buyer's QR code. That will finalize the transaction and I get paid for the textbook.
- If the buyer decides that the physical condition of the book is not good or that the book is somehow wrong, the buyer can cancel the transaction. That will have the textbook re-listed on the website.
- If the transaction is successful, I will get an email notification of the successful transaction.

(4) As an **admin**, I want to be able to **get the total amount of transactions** on the website so that I can use it to track my revenue from the website.

My conditions of satisfaction are:

- I can see an aggregate of all the previous transactions on the website.
- I can see the monthly revenue of the website. If the monthly revenue is down from the previous month, it shows the figure in red. If the monthly revenue is up, then it shows in green.

1. Student buyer acceptance testing

Common cases:

(a) User inputs the keywords of the textbook in the text field, or chooses predefined categories for textbook selections.

  (i) If found, then a list of relevant items should be returned and shown on the website. Assert if the same list of relevant items from the database query appears on the website.

  (ii) If not found, it should return a message "Nothing is found" with the corresponding error code. Assert the returned message.

  (iii) Each item in the return list must include at least one photo, and the user should be able to look at the photos/videos of the item. Assert if any of this is present on the website.

(iv) The user should be able to contact the seller and chat with the seller for the item. Assert if a message sent from one end appears on both ends.

(v) The user can pay via paypal. The test fails if the user cannot pay via paypal.

(vi) After goods have been exchanged, the user should be able to provide a QR code for the seller to scan. Assert if the QR code is present.

(vii) Once the QR code has been scanned, the transaction will be marked complete and funds will be transferred to the seller. We can test whether the funds will show up in the seller's account after the QR code has been scanned. The test fails if this cannot be done.

(viii) After the transaction, the item should be marked "unavailable" and the post should be archived. Assert if this is true.

(b) User shares the location via the browser or enters an address and chooses to view the "textbook" nearby.

(i) If there are textbooks available nearby, then a list of relevant items should be returned and shown on the website. We can assert if the list returned by the database is present on the website.

(ii) If not found, it should return a message "Nothing is found" with the corresponding error code. We can assert the returned message.

(iii) The rest is the same as part a.

(c) User inputs the name of the item but cannot find the item.

(i) The user will see a message "Nothing is found." If this cannot be done, the test fails. We can assert the returned message.

(ii) The user can post a "Want to buy" request via a "Want to buy" button on the website. If this cannot be done, the test fails.

(iii) The user categorizes the textbook, attaches a picture, and puts up a price and a location. We can assert if this is saved on the database.

(iv) Each want-to-buy post must include a title, a product description and a want-to-buy-price associated with the item. If this cannot be done, the test fails.

(v) After the request has been sent, the listing should show up in the database and be present on the website. If not, the test fails.

(vi) After posting, the posting should be shown under the "Want to buy" category and the general public can see that post. If this cannot be done, the test fails.

Uncommon cases:

(a) During the meetup, the buyer was not satisfied with the condition of the test book and wanted to cancel the transaction.

(i) The listing will be reposted on the website. We can assert if this is true.

(ii) The funds will be automatically refunded back to the buyer. We can assert the message returned by the paypal API.

2. **Student seller posting related acceptance testing**

The user input for student sellers will be a post of some kind of merchandise.

Common cases

(a) The seller wants to create a new post to sell one or a set of textbooks.

(i) The seller will go to "create a new post" tab of the website. If there is no such tab, the test fails.

(ii) The seller will enter a page to add required details about the textbook by clicking on the tab. If the page fails to load, the test fails.

(iii) The seller will be able to enter the name of the book, a text description about the book, at least one photo of the book, category of the book, and price of the book. If any of those fields is missing, the test fails.

(iv) The seller will be able to complete the post so that it is visible to every user. If the post cannot be seen after completion, the test fails.

(v) The seller should still be able to edit the post before it is purchased. If editing is not allowed, the test fails.

(b) After the textbook is sold, i.e. the seller has received payment from the buyer, the post should be automatically closed so that no buyers can see it anymore. If the post is still visible to any user, the test fails.

Uncommon cases

(a) The seller decides to stop selling one of the textbooks he posted before it is purchased.

(i) The seller will be able to edit his post. If editing is not allowed, the test fails.

(ii) The seller will find a button to remove the post so that no buyers can see it anymore. If any user can still see the post after removal, the test fails.

3. Student seller transaction related acceptance testing

The input is the update to the *product* database that a particular item for sale has been *in_transaction*. The buyer has already paid the cost of the textbook + fees through the Paypal API. That money is deposited in the website's Paypal account temporarily before the transaction succeeds.

Common case:

1. After chatting with the buyer to coordinate a time to meet up, the buyer and the seller agree to set up a location and the time to meet up.

    a. If it is impossible for the buyer and the seller to agree to meet, then the CANCEL TRANSACTION process starts.
    b. Assert that TIMER process has started when the buyer has paid the money for the textbooks. If TIMER process runs out, then assert that CANCEL TRANSACTION process starts.
    c. Assert that CHAT process has started between the two parties
2. The buyer and the seller have met in person.
    a. If the buyer does not like the physical condition of the book or for other reasons, the buyer can cancel the transaction. Assert CANCEL TRANSACTION process starts.
    b. If the buyer does like the item, the seller can scan a QR code that was generated for this transaction (using OpenCV API). Assert that QR CODE VERIFICATION PROCESS is successful.
    c. Assert that the PAYMENT process to give money to the seller has started.
    d. Assert that the EMAIL process using SUCCESSFUL TRANSACTION template has been sent.

Uncommon case:

1. CANCEL TRANSACTION process is called.
    a. Assert that the REFUND process for the buyer has started.
    b. Assert that the EMAIL process using CANCELLED TRANSACTION template has been sent.
    c. Assert that the textbook in transaction is re-listed on the website.

4. Admin related acceptance testing

The user input is the period of which they want to see the revenue, which could be monthly or total aggregate. Depending on the input of the user, the system will show the number corresponding to the monthly, yearly, or total revenue of the website.

Common case

(a) User chooses *Monthly* revenue. It returns a view of monthly revenue for the past 12 months from the *Transaction* database.

Assert that: there are twelve returned values corresponding to monthly revenue.

Assert that months with lower revenue compared to previous month have color variable “Red.”

Assert that the months with higher revenue compared to previous month have color variable “Blue.”

Assert that screen title shows “Monthly Revenue.”

(b) User chooses *Total* revenue. It returns a view of total aggregate revenue from the *transaction* database

Assert that there is only one positive value returned.

Assert that the screen title shows “Total Revenue.”

Uncommon cases

(a) The admin does not have access to the revenue database. The system returns an error message.

In acceptance testing, we can suppose a case where the user does not have access to the database (it does not exist). Then we can assert that the error message code we received is the same as the error message code for not having access to the database (or database not found).

Part 3:

You are required to use an automated build tool and/or package manager, suitable for your programming language and platform, that invokes an automated unit testing tool that, in turn, invokes your unit test cases. There are no specific requirements imposed on your unit test cases other than you have at least one test case for each method other than constructors, getters/setters, and helpers.   Some of the test cases will presumably invoke constructors, getters, setters, and/or helpers as part of their operation. (There will be more specific requirements imposed on unit testing as part of the second iteration assignment, which will also include coverage goals.)

Submit the link to the folder(s) within your github repository containing all the test cases that are automatically invoked by your unit testing tool.

https://github.com/Deep-Optimum/BackEnd

https://github.com/Deep-Optimum/Front-End

All the tests are here. I had a really nice project structure for the backend but had a lot of issues with importing and I got really frustrated and so I just temporarily moved all the files under the same directory so that I can use the terminal to compile all tests.

Submit the link to the file(s) in your repository that configure the build tool and/or package manager and the automated unit testing tool.

You can follow the following instruction to get

https://github.com/Deep-Optimum/Front-End/blob/main/README.md
https://github.com/Deep-Optimum/BackEnd/blob/main/README.md

Part 4:

You are required to use an automated style checker and an automated bug finder (the style checker might be included in the bug finder, but the bug finder must check for possible errors that are not simply violating style rules). Ideally, the build tool or package manager above would automatically invoke these tools in addition to the unit testing tool, but this is not required for this assignment. (The second interation assignment will require continuous integration that automatically invokes all these tools.)

Include the link(s) to the folder(s) within your repository containing the reports from your automated style checker and automated bug finder. Note this means you need to configure these tools to produce outputs that you can store as files in your repository.

In most cases, there should be at least two reports stored for each tool, at least one that shows errors and at least one that is clean(er). You should fix as many errors as possible from the earlier report(s), which should be reflected in the later and final report(s). Static analysis bug finders sometimes report possible errors that could not really occur at runtime, i.e., false positives, so instead of "fixing" such cases, you should figure out why it was reported and write an explanation. The final report should correspond to the current code, not some earlier version of the code. It is unlikely, but not impossible, that these tools find no errors, but they should still produce reports.

We use pylint for style check and the bug finder is already builtin with pycharm and pylint. Right now, we are still not able to connect both frontend and backend so we cannot actually build the project.. Each part works perfectly fine on its own end though.

https://github.com/Deep-Optimum/BackEnd/blob/main/.coverage