Team Name: Deep Optimum

Team Member UNI:
Beom Joon Baek - bb2763
Bennington Li - wl2750
Yifan Zang - yz3781
Zhuoxuan Li - zl2890

Part 1:

From the current development progress, we are going to have these user stories implemented and tested before our final demo:

(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 ISBN.
- If there are no textbooks I want to find, I can post a request for that textbook.
- 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.
- 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 to 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 transaction is successful, I will get an email notification of the successful Transaction.

Part 2:

In app.py:

1. search():
    - Partition of valid inputs: query parameter “isbn” or “title” followed by any query string.
    - Partition of invalid inputs: query parameter besides “isbn” and “title”
    - No boundary conditions
    - Test case: test_search
2. create_new_post():
    - Partition of valid inputs: a request body with json format that matches the column names of table “Listings”
    - Partition of invalid inputs: all request body with other format
    - No boundary conditions
    - Test case: test_create_new_post
3. post_by_id(listing_id):
    - Partition of valid inputs: any string that serves as listing_id, and a request body with json format that matches the column names of table “Listings”
    - Partition of invalid inputs: all request body with other format
    - No boundary conditions
    - Test case: test_post_by_id
4. create_user():
    - Partition of valid inputs: a request body with json format that matches the column names of table “User_info”
    - Partition of invalid inputs: all request body with other format
    - No boundary conditions
    - Test case: test_create_new_users
5. get_user_posts(uni):
    - Partition of valid inputs: any string that serves as uni
    - Partition of invalid inputs: None

    - No boundary conditions
    - Test case: test_get_user_posts
6. user_by_uni(uni):
    - Partition of valid inputs: any string that serves as uni, and a request body with json format that matches the column names of table “User_info”
    - Partition of invalid inputs: all request body with other format
    - No boundary conditions
    - Test case: test_get_user_posts

7. user_address(uni):
    - Partition of valid inputs: any string that serves as uni
    - Partition of invalid inputs: None
    - No boundary conditions
    - Test case: test_user_address
8. user_orders(uni):
    - Partition of valid inputs: any string that serves as uni
    - Partition of invalid inputs: None
    - No boundary conditions
    - Test case: test_user_orders
9. create_address():
    - Partition of valid inputs: a request body with json format that matches the column names of table “Addresses”
    - Partition of invalid inputs: all request body with other format
    - No boundary conditions
    - Test case: test_create_address
10. address_by_id(address_id):
    - Partition of valid inputs: any string that serves as address_id, and a request body with json format that matches the column names of table “Addresses”
    - Partition of invalid inputs: all request body with other format
    - No boundary conditions
    - Test case: test_address_by_id
11. create_order():
    - Partition of valid inputs: a request body with json format that matches the column names of table “Order_info”
    - Partition of invalid inputs: all request body with other format
    - No boundary conditions
    - Test case: test_create_order
12. order_by_id(order_id):

- Partition of valid inputs: any string that serves as order_id, and a request body with json format that matches the column names of table "Order_info"
- Partition of invalid inputs: all request body with other format
- No boundary conditions
- Test case: test_order_by_id

13. confirm_order(order_id, uni):
    - Partition of valid inputs: any two strings that serves as order_id and uni
    - Partition of invalid inputs: None
    - No boundary conditions
    - Test case: test_confirm_order
14. Create_checkout(order_id):
    - Partition of valid inputs:
        - Both buyer_confirm and seller_confirm is 1
        - Status is "in Progress"
        - Have valid "payment_method_nonce" and transaction results success
        - Query using order_id is successful
    - Partition of invalid inputs:
        - Either buyer_confirm or seller_confrim is 0 (test_checkout_failed_both_confirm)
        - Status is not "In progress" (test_checkout_already_ordered)
        - It does not have valid "payment_method_nonce" and Transaction results in failure (test_checkout_not_success)
        - Query using order_id is not successful (test_checkout_not_id)
    - Boundary conditions: test_payment
        - If the len(res) == 0: this means that query using order_id has not been successful
    - Testing: test_payment.py

In data_tables.py:

This file basically defines the Model for the backend model. Have excluded simple getter and setter methods, a ToString method and helper methods for database connections.

1. Get_info(self, table_name, template, get_similar=False, order_by=None, is_or=False)
    a. Partition of valid inputs:
        i. Table_name is not None or empty string
            1. Querying database does not result in error
                a. Either get_similar is True (test_get_info_similar)

                i. There are multiple similar values : test_get_info_similar_multple_values
                ii. There are multiple fields with similar: test_get_info_similar_multple_fields
            b. Get_similar is False (test_get_info)
    b. Partition of invalid inputs:
        i. Table_name is None or empty string (test_get_info_empty_name)
        ii. Querying database results in error (test_get_info_empty_result)
    c. Boundary condition: None
2. delete_info(table_name, template):
    a. Partition of valid inputs:
        i. Table_name is not none or empty string
            1. Deleting query is a valid query (test_delete_info)
    b. Partition of invalid inputs:
        i. Table_name is none or empty string
        ii. Deleting query is not a valid query (test_delete_info_fail)
    c. Boundary conditions: There are no boundary conditions
3. Import_from_csv(table_name, filepath)
    a. Partition of valid inputs
        i. Valid filepath, accurate table_name
    b. Partition of invalid inputs
        i. Valid filepath, wrong table name
        ii. Valid filepath, empty table name
        iii. Wrong filepath
    c. Boundary conditions: Only boundary conditions might be in certain configurations of operating systems or error, there might be cases when there might be filenames with two same names. In that case, the first file to be queried would be imported.
    d. Testing: testing for all partitions in test_import_from_csv

In dbutils.py

1. get_sql_from_file(connect_info)
    a. Partition of valid inputs:
        i. Connection using default connection (test_get_connection)
        ii. Connection using path (test_get_sql_from_file)
    b. Partition of invalid inputs:
        i. Invalid connection due to non-existing path (test_get_sql_from_Non_exist_file)
    c. Boundary conditions: For querying the database, the query is either in the database or not in the database. It is an either/or situation.

2. run_multiple_sql_statements(statements, fetch=True, cur=None, conn=None, commit=True)
    a. Partition of valid inputs:
        i. Valid connection, valid statement, valid query results (test_run_multiple_sql_statements)
    b. Partition of invalid inputs
        i. Connection is None (test_run_multiple_sql_statements_no_conn)
        ii. Statement is None (test_run_multiple_sql_statements_no_statement)
        iii. Query results in errors
    c. Boundary conditions: For querying the database, the query is either in the database or not in the database. It is an either/or situation. Likewise, for both connection and statement, it is categorical in that both are valid or not valid.
3. template_to_where_clause(template, is_like=False, is_or=False)
    a. Partition in valid inputs:
        i. Template is not none or empty:
            1. Is_like is true (test_template_to_where_clause)
            2. Is_like is false
            3. Is_or is true
            4. Is_or is false
    b. Partition in invalid inputs:
        i. Template is None or empty (test_template_to_where_clause_no_template)
    c. Boundary conditions: None. Pretty much all categorical.
    d. Testing: For above three valid partitions (except is_like is true), those cases are tested in test_template_to_where_clause_nkeys_n_vals, test_template_to_where_clause_1key_n_vals
4. create_select(table_name, template, fields=None, order_by=None, limit=None, offset=None, is_select=True, is_like=False, is_or=False)
    a. Partition in valid inputs:
        i. Is_select is True (test_create_select, test_create_select_single)
        ii. Is_select is False (test_create_delete)
        iii. Fields is not None (test_create_select_w_field_list)
        iv. Fields is None (test_create_select, test_create_select_single)
        v. Order_by is True (test_create_select_is_like_order_by)
        vi. Order_by is False (test_create_select, test_create_select_single)
        vii. Limit is True (test_create_select_is_like_limit_by)
        viii. Limit is False (test_create_select, test_create_select_single)

        ix. Is_like is True: (test_create_select_with_like, test_create_select_with_like_single_field, test_create_select_with_like_three_fields)

    b. Partition in invalid inputs:
    c. Boundary conditions: None

Backend test suite: https://github.com/Deep-Optimum/BackEnd/tree/main/tests
Frontend test suite:
https://github.com/Deep-Optimum/Front-End-Demo/blob/main/src/App.test.js

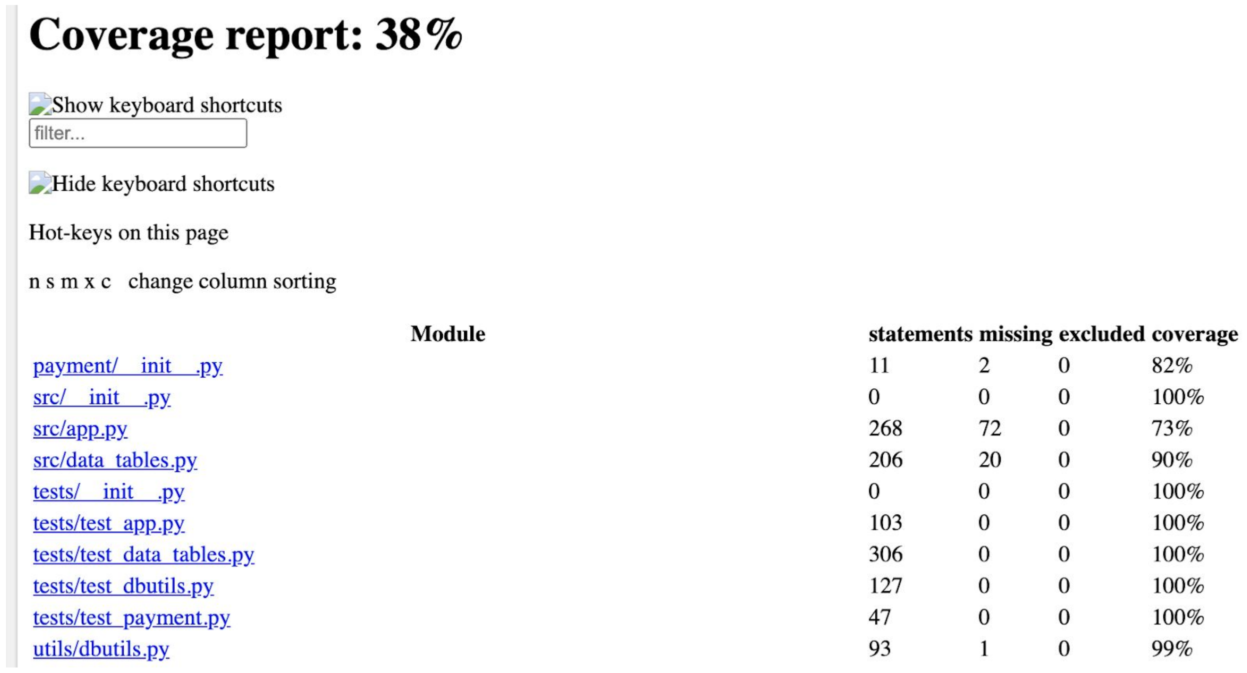
Part 3:

Branch coverage cannot reach 100% because we use try-catch structure in a lot of functions, and it is hard to produce exceptions to cover all the exception branches. For example, all endpoint methods in app.py are implemented with try catch structure, and many of those exceptions cannot be produced in unit tests. Besides app.py, other files achieve pretty high coverage.

Coverage report link: https://github.com/Deep-Optimum/BackEnd/tree/main/htmlcov
For some reason, the command we use to generate reports

```
coverage run -m pytest
coverage html
```

generated html reports for ALL the packages in the entire virtual environment. That is why the total coverage is only 38%. I have deleted all the corresponding html files that were not part of the program. Here is a snapshot of the index.html file; you can see the coverage for all the python files we actually ran. The actual coverage is about **92%.** Upon submission of this report, we have not fixed it. We will try to figure out how to generate reports only for the project later if we can.



Coverage report: 38%

Show keyboard shortcuts

filter...

Hide keyboard shortcuts

Hot-keys on this page

n s m x c   change column sorting

Module	statements	missing	excluded	coverage
payment/__init__.py	11	2	0	82%
src/__init__.py	0	0	0	100%
src/app.py	268	72	0	73%
src/data_tables.py	206	20	0	90%
tests/__init__.py	0	0	0	100%
tests/test_app.py	103	0	0	100%
tests/test_data_tables.py	306	0	0	100%
tests/test_dbutils.py	127	0	0	100%
tests/test_payment.py	47	0	0	100%
utils/dbutils.py	93	1	0	99%

Part 4:

CI Configuration:

Backend Link:

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

Frontend Link:

https://github.com/Deep-Optimum/Front-End-Demo/blob/main/.travis.yml

CI report folder:

Backend and Frontend Link: click the badges in
https://github.com/Deep-Optimum/BackEnd/blob/main/README.md
There are 2 badges, one for the backend and one for the frontend.