Test Document

for

Real-Estate Broker

Version 1.0

Prepared by

Group: 14 **Group Name: Dangling Pointers**

Chinmay Hiran Pillai	**200298**	chinmay20@iitk.ac.in
Shivang Pandey	**200941**	shivangp20@iitk.ac.in
Armeet Singh Luthra	**200185**	armeet20@iitk.ac.in
Arth Banka	**200191**	arth20@iitk.ac.in
Vaishali Rawat	**201080**	vaishalir20@iitk.ac.in
Utpal Dwivedi	**211132**	utpaldwi21@iitk.ac.in
Praveen Raj	**210763**	praveenr21@iitk.ac.in
Abhishek Kumar	**210039**	akumar21@iitk.ac.in
Pravallika Mudunuru	**220814**	pmudunuru22@iitk.ac.in
Gaddam Shiva Leela	**220392**	gaddamshiv22@iitk.ac.in

Course: **CS253**

Mentor TA: ***Vaibhav***

Date: **29/03/2024**

Index

Revisions

Version	Primary Author(s)	Description of Version	Date Completed
1.0	Chinmay Hiran Pillai, Shivang Pandey, Armeet Luthra, Arth Banka, Utpal Dwivedi, Praveen Raj	Testing Document for Real-Estate Broker	29/03/24

1 Introduction

This document outlines the testing strategy employed for the real estate broker webapp. A combination of automated and manual testing approaches were implemented to ensure comprehensive coverage of the application's functionality.

Manual testing for backend components commenced alongside development, with more rigorous automated testing conducted post-implementation. The frontend, integration and system were also manually tested both in parallel with development and more rigorously post-implementation.

The developers themselves were the testers.
Backend (URLs, Models, Serializers, APIs): Chinmay
Frontend Functionality: Armeet
System Testing: Chinmay & Utpal

The Modified Decision/Condition Coverage (MC/DC) criteria was the primary testing strategy employed. This combined approach ensures comprehensive testing of the real estate broker webapp, fostering a high-quality and reliable application.

Tools Utilised:
Backend Testing:
Automation: We utilized a robust automated testing framework, leveraging a combination of Django's built-in test module and pytest.
Coverage: 90 automated tests were written to thoroughly assess the functionality of URLs, models, serializers, and APIs.

Frontend Testing: Armeet manually tested the frontend functionalities of the webapp to ensure a seamless user experience.

System Testing: Chinmay conducted manual system testing to verify successful deployment, adherence to functional requirements, and satisfactory performance in terms of non-functional requirements.

2 Unit Testing

Unit Name: TestModels

This unit tests the models in the api.test_models module.

Test Details:

- **Test Owner**: Chinmay
- **Test Date**: 20/03/2024 - 21/03/2024

Structural Coverage:

- All models tested: UserProfile, Property, Order, Support
- Test coverage criteria: Statement Coverage, Branch Coverage

Test Cases:

1. *Test Case: `test_create_user_profile_with_all_required_fields`*

Description: Create a new user profile with all required fields.
Function Tested: `UserProfile.objects.create()`

Test Steps:
1. Create a user profile with all required fields.
2. Check if the user profile fields match the expected values.

Test Results:
The user profile was created successfully with all expected values.

Structural Coverage:
- Branch Coverage: All branches covered within the `test_create_user_profile_with_all_required_fields` function.
- Statement Coverage: All statements within the function executed.

Additional Comments:
- The test passed successfully.
- This ensures that a user profile can be created with all required fields.

2. *Test Case: `test_property_creation_with_all_fields_filled`*

- **Description**: Property object can be created with all fields filled.
- **Function Tested**: `Property.__init__()`

Test Steps:
1. Create a property object with all fields filled.
2. Check if the property object fields match the expected values.

Test Results:
- The property object was created successfully with all expected values.

Structural Coverage:
- Branch Coverage: All branches covered within the `test_property_creation_with_all_fields_filled` function.
- Statement Coverage: All statements within the function executed.

Additional Comments:
- The test passed successfully.
- This ensures that a property object can be created with all fields filled.

3. *Test Case: `test_create_buy_order_with_valid_inputs`*

- **Description**: Creating a buy order for a property with valid user, property, and price.
- **Function Tested**: `Order.objects.create()`

Test Steps:
1. Create a user profile.
2. Create a property object.
3. Create a buy order with valid inputs.
4. Check if the order fields match the expected values.

Test Results:
- The order was created successfully with all expected values.

Structural Coverage:
- Branch Coverage: All branches covered within the `test_create_buy_order_with_valid_inputs` function.
- Statement Coverage: All statements within the function executed.

Additional Comments:
- The test passed successfully.
- This ensures that a buy order can be created with valid inputs.

4. *Test Case: `test_create_order_with_nonexistent_user`*

- **Description**: Creating an order with a non-existent user.
- **Function Tested**: `Order.objects.create()`

Test Steps:
1. Create a property object.

2. Attempt to create an order with a non-existent user.
3. Check if an `IntegrityError` is raised.

Test Results:
- An `IntegrityError` was raised as expected.

Structural Coverage:
- Branch Coverage: All branches covered within the `test_create_order_with_nonexistent_user` function.
- Statement Coverage: All statements within the function executed.

Additional Comments:
- The test passed successfully.
- This ensures that an order cannot be created with a non-existent user.

5. Test Case: `test_valid_instance`

- **Description**: Creating a new instance of Support with valid name, email, and message.
- **Function Tested**: `Support.__init__()`

Test Steps:
1. Create a support object with valid inputs.
2. Check if the support object fields match the expected values.

Test Results:
- The support object was created successfully with all expected values.

Structural Coverage:
- Branch Coverage: All branches covered within the `test_valid_instance` function.
- Statement Coverage: All statements within the function executed.

Additional Comments:
- The test passed successfully.
- This ensures that a Support instance can be created with valid inputs.

Unit Name: TestSerializers

This unit tests the RegisterSerializer from the api.serializers module.

Test Details:

- Test Owner: Chinmay
- Test Date: 22/03/2024 - 23/03/2024

Structural Coverage:

- Test coverage criteria: Statement Coverage, Branch Coverage

Test Cases:

1. Test Case: test_pan_format_valid

- Description: PAN number follows the format of 5 alphabets followed by 4 numbers followed by another alphabet.
- Function Tested: RegisterSerializer.validate()

Test Steps:

Create a serializer instance.
Create valid PAN number data.
Validate the data using the serializer.

Test Results:

- Expected: The PAN number should be valid.
- Actual: The PAN number was validated successfully.

Structural Coverage:

- Branch Coverage: All branches covered within the test_pan_format_valid function.
- Statement Coverage: All statements within the function executed.

Additional Comments:

- The test passed successfully.
- This ensures that a valid PAN number is accepted.

2. Test Case: test_pan_length_shorter

- Description: PAN number has less than 10 characters.
- Function Tested: RegisterSerializer.validate()

Test Steps:

Create a serializer instance.
Create a PAN number with less than 10 characters.
Validate the data using the serializer.

Test Results:

- Expected: An error should be raised for a shorter PAN number.
- Actual: An error was raised as expected.

Structural Coverage:

- Branch Coverage: All branches covered within the test_pan_length_shorter function.
- Statement Coverage: All statements within the function executed.

Additional Comments:

- The test passed successfully.
- This ensures that a shorter PAN number is rejected.

3. Test Case: test_pan_length_longer

- Description: PAN number has more than 10 characters.
- Function Tested: RegisterSerializer.validate()

Test Steps:

Create a serializer instance.
Create a PAN number with more than 10 characters.
Validate the data using the serializer.

Test Results:

- Expected: An error should be raised for a longer PAN number.
- Actual: An error was raised as expected.

Structural Coverage:

- Branch Coverage: All branches covered within the test_pan_length_longer function.
- Statement Coverage: All statements within the function executed.

Additional Comments:

- The test passed successfully.
- This ensures that a longer PAN number is rejected.

4. Test Case: test_pan_invalid

- Description: PAN number has alphabets in the wrong position.
- Function Tested: RegisterSerializer.validate()

Test Steps:

Create a serializer instance.
Create an invalid PAN number.
Validate the data using the serializer.

Test Results:

- Expected: An error should be raised for an invalid PAN number.
- Actual: An error was raised as expected.

Structural Coverage:

- Branch Coverage: All branches covered within the test_pan_invalid function.
- Statement Coverage: All statements within the function executed.

Additional Comments:

- The test passed successfully.
- This ensures that an invalid PAN number is rejected.

5. Test Case: test_phone_length_shorter

- Description: Phone number has less than 10 digits.
- Function Tested: RegisterSerializer.validate()

Test Steps:

Create a serializer instance.
Create a phone number with less than 10 digits.
Validate the data using the serializer.

Test Results:

- Expected: An error should be raised for a shorter phone number.
- Actual: An error was raised as expected.

Structural Coverage:

- Branch Coverage: All branches covered within the test_phone_length_shorter function.
- Statement Coverage: All statements within the function executed.

Additional Comments:

- The test passed successfully.
- This ensures that a shorter phone number is rejected.

6. Test Case: test_phone_length_longer

- Description: Phone number has more than 10 digits.
- Function Tested: RegisterSerializer.validate()

Test Steps:

Create a serializer instance.
Create a phone number with more than 10 digits.
Validate the data using the serializer.

Test Results:

- Expected: An error should be raised for a longer phone number.
- Actual: An error was raised as expected.

Structural Coverage:

- Branch Coverage: All branches covered within the test_phone_length_longer function.
- Statement Coverage: All statements within the function executed.

Additional Comments:
The test passed successfully. This ensures that a longer phone number is rejected.

—---------

Unit Name: TestProperties

Test Owner: Chinmay, Shivang

Test Date: 25/03/2024 - 25/03/2024

Structural Coverage:

- Test coverage criteria: Statement Coverage, Branch Coverage

Test Cases:

1. *Test Case: test_get_all_properties*

- Description: GET request to /property returns all properties.
- Function Tested: property_list()

Test Steps:

Create a RequestFactory instance for a GET request to /property.
Create a Property object.
Call property_list() with the request.

Test Results:

- Response status code should be 200.
- Response data should contain 'properties' key.
- 'properties' key should have more than 0 items.

Structural Coverage:

- Branch Coverage: All branches covered within the test_get_all_properties function.
- Statement Coverage: All statements within the function executed.

2. *Test Case: test_valid_property_object*

- Description: Returns a valid property object when given a valid id and a GET request.
- Function Tested: property_by_id()

Test Steps:

Create a RequestFactory instance for a GET request to /properties/{id}.
Create a Property object with an id.
Create a PropertySerializer instance.
Call property_by_id() with the request and id.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response data should match the serialized data.

Structural Coverage:

- Branch Coverage: All branches covered within the test_valid_property_object function.
- Statement Coverage: All statements within the function executed.

3. Test Case: test_invalid_id_404

- Description: Returns a 404 status code when given an invalid id and a GET request.
- Function Tested: property_by_id()

Test Steps:

Create a RequestFactory instance for a GET request to an invalid id.
Call property_by_id() with the request and invalid id.

Test Results:

- Expected: Response status code should be 404.

Structural Coverage:

- Branch Coverage: All branches covered within the test_invalid_id_404 function.
- Statement Coverage: All statements within the function executed.

4. Test Case: test_invalid_id_delete_request

- Description: Returns a 404 status code when given an invalid id and a DELETE request.
- Function Tested: property_by_id()

Test Steps:

Create a RequestFactory instance for a DELETE request to an invalid id.
Call property_by_id() with the request and invalid id.

Test Results:

- Expected: Response status code should be 404.

Structural Coverage:

- Branch Coverage: All branches covered within the test_invalid_id_delete_request function.
- Statement Coverage: All statements within the function executed.

5. Test Case: test_invalid_data_returns_400

- Description: Returns a 400 status code when given invalid data and a PUT request.
- Function Tested: property_by_id()

Test Steps:

Create a RequestFactory instance for a PUT request with invalid data.
Create a Property object with an id.
Call property_by_id() with the request, id, and invalid data.

Test Results:

- Expected: Response status code should be 400.

Structural Coverage:

- Branch Coverage: All branches covered within the test_invalid_data_returns_400 function.
- Statement Coverage: All statements within the function executed.

6. Test Case: test_put_request_with_missing_data

- Description: Returns a 400 status code when given a PUT request with missing data.
- Function Tested: property_by_id()

Test Steps:

Create a RequestFactory instance for a PUT request with missing data.
Create a Property object with an id.
Call property_by_id() with the request and id.

Test Results:

- Expected: Response status code should be 400.

Structural Coverage:

- Branch Coverage: All branches covered within the test_put_request_with_missing_data function.
- Statement Coverage: All statements within the function executed.

7. Test Case: test_valid_property_update

- Description: Updates and returns a valid property object when given a valid id and a PUT request with valid data.
- Function Tested: property_by_id()

Test Steps:

Create a RequestFactory instance for a PUT request with valid data.
Create a Property object with an id.
Create a PropertySerializer instance.
Call property_by_id() with the request, id, and valid data.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response data should match the serialized data.

Structural Coverage:

Branch Coverage: All branches covered within the test_valid_property_update function.

Statement Coverage: All statements within the function executed.

—--------

Unit Name: TestWatchlist

Test Owner: Chinmay

Test Date: 25/03/2024 - 25/03/2024

Structural Coverage:

- Test coverage criteria: Statement Coverage, Branch Coverage

Test Cases:

1. Test Case: test_user_not_found

- Description: Returns HTTP 404 if user with given id does not exist.
- Function Tested: watchlist()

Test Steps:

Create a RequestFactory instance for a GET request to /watchlist/{id} where the user does not exist.
Call watchlist() with the request.

Test Results:

- Expected: Response status code should be 404.

Structural Coverage:

- Branch Coverage: All branches covered within the test_user_not_found function.
- Statement Coverage: All statements within the function executed.

2. Test Case: test_missing_action_field

- Description: Returns HTTP 400 if the action field is missing in request data.
- Function Tested: watchlist()

Test Steps:

Create a RequestFactory instance for a PUT request to /watchlist/{id} without the action field.
Call watchlist() with the request.

Test Results:

- Expected: Response status code should be 400.

Structural Coverage:

- Branch Coverage: All branches covered within the test_missing_action_field function.
- Statement Coverage: All statements within the function executed.

3. Test Case: test_get_watchlist

- Description: Returns user's watchlist if the request method is GET.
- Function Tested: watchlist()

Test Steps:

Create a RequestFactory instance for a GET request to /watchlist/{id}.
Set the user's watchlist.
Call watchlist() with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response data should contain the user's watchlist.

Structural Coverage:

- Branch Coverage: All branches covered within the test_get_watchlist function.
- Statement Coverage: All statements within the function executed.

4. Test Case: test_remove_property_from_watchlist

- Description: Removes property_id from user's watchlist if action is 'remove' and property_id is in watchlist.

- Function Tested: watchlist()

Test Steps:

Create a RequestFactory instance for a PUT request to /watchlist/{id} to remove a property.
Set the user's watchlist.
Call watchlist() with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response data should contain the updated watchlist without the removed property.

Structural Coverage:

- Branch Coverage: All branches covered within the test_remove_property_from_watchlist function.
- Statement Coverage: All statements within the function executed.

5. Test Case: test_add_property_to_watchlist

- Description: Adds property_id to user's watchlist if action is 'add' and property_id is not already in watchlist.
- Function Tested: watchlist()

Test Steps:

Create a RequestFactory instance for a PUT request to /watchlist/{id} to add a property.
Set the user's watchlist.
Call watchlist() with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response data should contain the updated watchlist with the added property.

Structural Coverage:

- Branch Coverage: All branches covered within the test_add_property_to_watchlist function.
- Statement Coverage: All statements within the function executed.

6. Test Case: test_invalid_action

- Description: Returns HTTP 400 if action is not 'add' or 'remove'.
- Function Tested: watchlist()

Test Steps:

Create a RequestFactory instance for a PUT request to /watchlist/{id} with an invalid action.
Call watchlist() with the request.

Test Results:

- Expected: Response status code should be 400.

Structural Coverage:

- Branch Coverage: All branches covered within the test_invalid_action function.
- Statement Coverage: All statements within the function executed.

7. Test Case: test_missing_property_id

- Description: Returns HTTP 400 if property_id field is missing in request data.
- Function Tested: watchlist()

Test Steps:

Create a RequestFactory instance for a PUT request to /watchlist/{id} without the property_id field.
Call watchlist() with the request.

Test Results:

- Expected: Response status code should be 400.

Structural Coverage:

- Branch Coverage: All branches covered within the test_missing_property_id function.
- Statement Coverage: All statements within the function executed.

8. Test Case: test_user_watchlist_remove_no_exceptions

- Description: Does not raise any exceptions if user's watchlist is None and action is 'remove'.
- Function Tested: watchlist()

Test Steps:

Create a RequestFactory instance for a PUT request to /watchlist/{id} to remove a property when watchlist is None.
Call watchlist() with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response data should contain an empty watchlist.

Structural Coverage:

- Branch Coverage: All branches covered within the test_user_watchlist_remove_no_exceptions function.
- Statement Coverage: All statements within the function executed.

—----------

Unit Name: TestPortfolio

Test Owner: Chinmay

Test Date: 25/03/2024 - 25/03/2024

Structural Coverage:

- Test coverage criteria: Statement Coverage, Branch Coverage

Test Cases:

1. Test Case: test_user_not_found

- Description: Returns HTTP 404 if user with given id does not exist.
- Function Tested: portfolio()

Test Steps:

Create a RequestFactory instance for a GET request to /portfolio/{id} where the user does not exist.
Call portfolio() with the request.

Test Results:

- Expected: Response status code should be 404.

Structural Coverage:

- Branch Coverage: All branches covered within the test_user_not_found function.
- Statement Coverage: All statements within the function executed.

2. Test Case: test_missing_action_field

- Description: Returns HTTP 400 if the action field is missing in request data.
- Function Tested: portfolio()

Test Steps:

Create a RequestFactory instance for a PUT request to /portfolio/{id} without the action field.
Call portfolio() with the request.

Test Results:

- Expected: Response status code should be 400.

Structural Coverage:

- Branch Coverage: All branches covered within the test_missing_action_field function.
- Statement Coverage: All statements within the function executed.

3. Test Case: test_get_portfolio

- Description: Returns user's portfolio if the request method is GET.
- Function Tested: portfolio()

Test Steps:

Create a RequestFactory instance for a GET request to /portfolio/{id}.
Set the user's portfolio.
Call portfolio() with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response data should contain the user's portfolio.

Structural Coverage:

- Branch Coverage: All branches covered within the test_get_portfolio function.
- Statement Coverage: All statements within the function executed.

4. Test Case: test_remove_property_from_portfolio

- Description: Removes property_id from user's portfolio if action is 'remove' and property_id is in portfolio.
- Function Tested: portfolio()

Test Steps:

Create a RequestFactory instance for a PUT request to /portfolio/{id} to remove a property.
Set the user's portfolio.
Call portfolio() with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response data should contain the updated portfolio without the removed property.

Structural Coverage:

- Branch Coverage: All branches covered within the test_remove_property_from_portfolio function.
- Statement Coverage: All statements within the function executed.

5. Test Case: test_add_property_to_portfolio

- Description: Adds property_id to user's portfolio if action is 'add' and property_id is not already in portfolio.
- Function Tested: portfolio()

Test Steps:

Create a RequestFactory instance for a PUT request to /portfolio/{id} to add a property.
Set the user's portfolio.
Call portfolio() with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response data should contain the updated portfolio with the added property.

Structural Coverage:

- Branch Coverage: All branches covered within the test_add_property_to_portfolio function.
- Statement Coverage: All statements within the function executed.

6. Test Case: test_invalid_action

- Description: Returns HTTP 400 if action is not 'add' or 'remove'.
- Function Tested: portfolio()

Test Steps:

Create a RequestFactory instance for a PUT request to /portfolio/{id} with an invalid action.
Call portfolio() with the request.

Test Results:

- Expected: Response status code should be 400.

Structural Coverage:

- Branch Coverage: All branches covered within the test_invalid_action function.
- Statement Coverage: All statements within the function executed.

7. Test Case: test_missing_property_id

- Description: Returns HTTP 400 if property_id field is missing in request data.
- Function Tested: portfolio()

Test Steps:

Create a RequestFactory instance for a PUT request to /portfolio/{id} without the property_id field.
Call portfolio() with the request.

Test Results:

- Expected: Response status code should be 400.

Structural Coverage:

- Branch Coverage: All branches covered within the test_missing_property_id function.
- Statement Coverage: All statements within the function executed.

8. Test Case: test_user_portfolio_remove_no_exceptions

- Description: Does not raise any exceptions if user's portfolio is None and action is 'remove'.
- Function Tested: portfolio()

Test Steps:

Create a RequestFactory instance for a PUT request to /portfolio/{id} to remove a property when portfolio is None.
Call portfolio() with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response data should contain an empty portfolio.

Structural Coverage:

- Branch Coverage: All branches covered within the test_user_portfolio_remove_no_exceptions function.
- Statement Coverage: All statements within the function executed.

—----------

Unit Name: TestUserData

Test Owner: Chinmay

Test Date: 25/03/2024 - 25/03/2024

Structural Coverage:

- Test coverage criteria: Statement Coverage, Branch Coverage

Test Cases:

1. Test Case: test_get_user_returns_user_with_given_id

- Description: Returns user with given id when a GET request is made to /users/{id}.
- Function Tested: get_user()

Test Steps:

Create a RequestFactory instance for a GET request to /users/{id}.
Call get_user() with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response data should contain the user's details including name, email, phone, funds, money_invested, pnl, portfolio, watchlist, and pan.

Structural Coverage:

- Branch Coverage: All branches covered within the test_get_user_returns_user_with_given_id function.
- Statement Coverage: All statements within the function executed.

2. Test Case: test_get_user_with_invalid_id_returns_404

- Description: Returns 404 when a GET request is made to /users/{id} with an invalid id.
- Function Tested: get_user()

Test Steps:

Create a RequestFactory instance for a GET request to /users/{id} with an invalid id.
Call get_user() with the request.

Test Results:

- Expected: Response status code should be 404.

Structural Coverage:

- Branch Coverage: All branches covered within the test_get_user_with_invalid_id_returns_404 function.
- Statement Coverage: All statements within the function executed.

Unit Name: TestMarketOrder

Test Owner: Chinmay, Shivang

Test Date: 25/03/2024 - 25/03/2024

Structural Coverage:

- Test coverage criteria: Statement Coverage, Branch Coverage

Test Cases:

1. Test Case: test_buy_market_order

- Description: Places a market buy order for a property.
- Function Tested: marketOrder()

Test Steps:

Create a RequestFactory instance for a PUT request to /marketorder.
Call marketOrder() with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: User's portfolio should include the bought property.

Structural Coverage:

- Branch Coverage: All branches covered within the test_buy_market_order function.
- Statement Coverage: All statements within the function executed.

2. Test Case: test_sell_market_order

- Description: Places a market sell order for a property.
- Function Tested: marketOrder()

Test Steps:

Create a RequestFactory instance for a PUT request to /marketorder.
Call marketOrder() with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: User's portfolio should not include the sold property.

Structural Coverage:

- Branch Coverage: All branches covered within the test_sell_market_order function.
- Statement Coverage: All statements within the function executed.

3. Test Case: test_no_buy_orders

- Description: Attempts to place a market sell order for a property with no available buy orders.
- Function Tested: marketOrder()

Test Steps:

Create a RequestFactory instance for a PUT request to /marketorder.
Call marketOrder() with the request.

Test Results:

- Expected: Response status code should be 400.
- Expected: Error message should indicate no buy orders available.

Structural Coverage:

- Branch Coverage: All branches covered within the test_no_buy_orders function.
- Statement Coverage: All statements within the function executed.

4. Test Case: test_no_sell_orders

- Description: Attempts to place a market buy order for a property with no available sell orders.
- Function Tested: marketOrder()

Test Steps:

Create a RequestFactory instance for a PUT request to /marketorder.
Call marketOrder() with the request.

Test Results:

- Expected: Response status code should be 400.
- Expected: Error message should indicate no sell orders available.

Structural Coverage:

- Branch Coverage: All branches covered within the test_no_sell_orders function.
- Statement Coverage: All statements within the function executed.

5. Test Case: test_sell_unowned_prop

- Description: Attempts to sell a property that is not in the user's portfolio.
- Function Tested: marketOrder()

Test Steps:

Create a RequestFactory instance for a PUT request to /marketorder.
Call marketOrder() with the request.

Test Results:

- Expected: Response status code should be 400.
- Expected: Error message should indicate the property is not in the portfolio.

Structural Coverage:

- Branch Coverage: All branches covered within the test_sell_unowned_prop function.
- Statement Coverage: All statements within the function executed.

6. Test Case: test_owned_prop

- Description: Attempts to buy a property that is already owned by the user.
- Function Tested: marketOrder()

Test Steps:

Create a RequestFactory instance for a PUT request to /marketorder.
Call marketOrder() with the request.

Test Results:

- Expected: Response status code should be 400.
- Expected: Error message should indicate the user can't buy their own property.

Structural Coverage:

- Branch Coverage: All branches covered within the test_owned_prop function.
- Statement Coverage: All statements within the function executed.

7. Test Case: test_insufficient_funds

- Description: Attempts to buy a property with insufficient funds.
- Function Tested: marketOrder()

Test Steps:

Create a RequestFactory instance for a PUT request to /marketorder.
Call marketOrder() with the request.

Test Results:

- Expected: Response status code should be 400.
- Expected: Error message should indicate insufficient funds.

Structural Coverage:

- Branch Coverage: All branches covered within the test_insufficient_funds function.
- Statement Coverage: All statements within the function executed

Unit Name: TestLimitOrder

Test Owner: Chinmay, Shivang

Test Date: 25/03/2024 - 25/03/2024

Test Cases:

1. Test Case: test_insufficient_funds

- Description: Attempts to create a limit order with insufficient funds.
- Function Tested: `limitOrder()`

Test Steps:

Create a `RequestFactory` instance for a POST request to `/limitorder`.
Call `limitOrder()` with the request.

Test Results:

- Expected: Response status code should be 400.
- Expected: Error message should indicate insufficient funds.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_insufficient_funds` function.
- Statement Coverage: All statements within the function executed.

2. Test Case: test_valid_post_request

- Description: Creates a valid limit order with all required fields.
- Function Tested: `limitOrder()`

Test Steps:

Create a `RequestFactory` instance for a POST request to `/limitorder`.
Call `limitOrder()` with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: User's portfolio should include the bought property.
- Expected: User's funds should reflect the purchase.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_valid_post_request` function.
- Statement Coverage: All statements within the function executed.

3. Test Case: test_valid_sell_request

- Description: Creates a valid limit sell order with all required fields.
- Function Tested: `limitOrder()`

Test Steps:

Create a `RequestFactory` instance for a POST request to `/limitorder`.
Call `limitOrder()` with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: User's portfolio should not include the sold property.
- Expected: User's funds should reflect the sale.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_valid_sell_request` function.
- Statement Coverage: All statements within the function executed.

4. Test Case: test_invalid_sell_request

- Description: Attempts to create an invalid limit sell order for a property not in the user's portfolio.
- Function Tested: `limitOrder()`

Test Steps:

Create a `RequestFactory` instance for a POST request to `/limitorder`.

Call `limitOrder()` with the request.

Test Results:

- Expected: Response status code should be 400.
- Expected: Error message should indicate the property is not in the portfolio.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_invalid_sell_request` function.
- Statement Coverage: All statements within the function executed.

5. Test Case: test_missing_action_field

- Description: Attempts to create a limit order with a missing 'action' field.
- Function Tested: `limitOrder()`

Test Steps:

Create a `RequestFactory` instance for a POST request to `/limitorder`.
Call `limitOrder()` with the request.

Test Results:

- Expected: Response status code should be 400.
- Expected: Error message should indicate the 'action' field is required.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_missing_action_field` function.
- Statement Coverage: All statements within the function executed.

6. Test Case: test_missing_prop_id_field

- Description: Attempts to create a limit order with a missing 'property_id' field.
- Function Tested: `limitOrder()`

Test Steps:

Create a `RequestFactory` instance for a POST request to `/limitorder`.
Call `limitOrder()` with the request.

Test Results:

- Expected: Response status code should be 400.
- Expected: Error message should indicate the 'property_id' field is required.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_missing_prop_id_field` function.
- Statement Coverage: All statements within the function executed.

7. Test Case: test_missing_user_id_field

- Description: Attempts to create a limit order with a missing 'user_id' field.
- Function Tested: `limitOrder()`

Test Steps:

Create a `RequestFactory` instance for a POST request to `/limitorder`.
Call `limitOrder()` with the request.

Test Results:

- Expected: Response status code should be 400.
- Expected: Error message should indicate the 'user_id' field is required.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_missing_user_id_field` function.
- Statement Coverage: All statements within the function executed.

8. Test Case: test_sell_unowned_prop

- Description: Attempts to sell a property that is not in the user's portfolio.
- Function Tested: `limitOrder()`

Test Steps:

Create a `RequestFactory` instance for a POST request to `/limitorder`.
Call `limitOrder()` with the request.

Test Results:

- Expected: Response status code should be 400.
- Expected: Error message should indicate the property is not in the portfolio.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_sell_unowned_prop` function.
- Statement Coverage: All statements within the function executed.

Unit Name: TestLoginRegister

Test Owner: Chinmay

Test Date: 25/03/2024 - 25/03/2024

Test Cases:

1. Test Case: test_post_register_invalid_data_returns_400

- Description: Attempts to register a new user with invalid data.
- Function Tested: `register()`

Test Steps:

Create a `RequestFactory` instance for a POST request to `/register`.
Call `register()` with the request.

Test Results:

- Expected: Response status code should be 400.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_post_register_invalid_data_returns_400` function.
- Statement Coverage: All statements within the function executed.

2. Test Case: test_register_with_valid_data

- Description: Registers a new user with valid data.
- Function Tested: `register()`

Test Steps:

Create a `RequestFactory` instance for a POST request to `/register`.
Call `register()` with the request.

Test Results:

- Expected: Response status code should be 201.
- Expected: Message in response should confirm successful user registration.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_register_with_valid_data` function.
- Statement Coverage: All statements within the function executed.

3. Test Case: test_login_with_valid_credentials

- Description: Logs in a user with valid credentials.
- Function Tested: `login()`

Test Steps:

Create a test user in the database.
Create a `RequestFactory` instance for a POST request to `/login`.
Call `login()` with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Message in response should confirm successful login.
- Expected: User ID should be included in the response.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_login_with_valid_credentials` function.
- Statement Coverage: All statements within the function executed.

4. Test Case: test_login_with_invalid_credentials_returns_401

- Description: Attempts to log in with invalid credentials.
- Function Tested: `login()`

Test Steps:

Create a `RequestFactory` instance for a POST request to `/login`.
Call `login()` with the request.

Test Results:

- Expected: Response status code should be 401.
- Expected: Error message in response should indicate invalid credentials.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_login_with_invalid_credentials_returns_401` function.
- Statement Coverage: All statements within the function executed.

Unit Name: TestFunds

Test Owner: Chinmay

Test Date: 25/03/2024 - 25/03/2024

Test Cases:

1. Test Case: test_get_user_funds

- Description: Retrieves user funds with a GET request.
- Function Tested: `funds()`

Test Steps:

Create a `RequestFactory` instance for a GET request to `/funds/{user_id}`.
Call `funds()` with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response should contain 'funds' field.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_get_user_funds` function.
- Statement Coverage: All statements within the function executed.

2. Test Case: test_put_request_adds_funds_to_user_account

- Description: Adds funds to a user account with a PUT request.
- Function Tested: `funds()`

Test Steps:

Create a `RequestFactory` instance for a PUT request to `/funds/{user_id}` with action 'add'.

Call `funds()` with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response should contain updated 'funds' field reflecting the added amount.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_put_request_adds_funds_to_user_account` function.
- Statement Coverage: All statements within the function executed.

3. Test Case: test_put_request_withdraws_funds_from_user_account

- Description: Withdraws funds from a user account with a PUT request.
- Function Tested: `funds()`

Test Steps:

Create a `RequestFactory` instance for a PUT request to `/funds/{user_id}` with action 'withdraw'.
Call `funds()` with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response should contain updated 'funds' field reflecting the withdrawn amount.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_put_request_withdraws_funds_from_user_account` function.
- Statement Coverage: All statements within the function executed.

4. Test Case: test_put_request_with_excess_withdraw_amount

- Description: Attempts to withdraw an excess amount from a user account.
- Function Tested: `funds()`

Test Steps:

Create a `RequestFactory` instance for a PUT request to `/funds/{user_id}` with action 'withdraw' and an excess amount.

Call `funds()` with the request.

Test Results:

- Expected: Response status code should be 400.
- Expected: Error message in response should indicate insufficient funds.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_put_request_with_excess_withdraw_amount` function.
- Statement Coverage: All statements within the function executed.

Unit Test Document: TestOrderBook

Test Owner: Chinmay

Test Date: 25/03/2024 - 25/03/2024

Test Cases:

1. Test Case: test_get_top_5_buy_orders_for_property

- Description: Retrieves the top 5 buy orders for a property with a GET request.
- Function Tested: `buy_orders()`

Test Steps:

Create a `RequestFactory` instance for a GET request to `/orders/buy/{property_id}`.
Call `buy_orders()` with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response should contain a list of buy orders for the property with a length less than or equal to 5.
- Expected: Each order in the response should have 'order_type' as 'buy' and 'prop' matching the property_id.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_get_top_5_buy_orders_for_property` function.
- Statement Coverage: All statements within the function executed.

2. Test Case: test_get_top_5_sell_orders_for_property

- Description: Retrieves the top 5 sell orders for a property with a GET request.
- Function Tested: `sell_orders()`

Test Steps:

Create a `RequestFactory` instance for a GET request to `/orders/sell/{property_id}`.
Call `sell_orders()` with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response should contain a list of sell orders for the property with a length less than or equal to 5.
- Expected: Each order in the response should have 'order_type' as 'sell' and 'prop' matching the property_id.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_get_top_5_sell_orders_for_property` function.
- Statement Coverage: All statements within the function executed.

3. Test Case: test_get_invalid_buy_orders

- Description: Attempts to retrieve buy orders for an invalid property id.
- Function Tested: `buy_orders()`

Test Steps:

Create a `RequestFactory` instance for a GET request to an invalid URL `/orders/buy/{invalid_id}`.
Call `buy_orders()` with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response should contain an empty list indicating no buy orders found for the invalid property id.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_get_invalid_buy_orders` function.
- Statement Coverage: All statements within the function executed.

4. Test Case: test_get_sell_orders_with_invalid_id_returns_empty_list

- Description: Attempts to retrieve sell orders for an invalid property id.
- Function Tested: `sell_orders()`

Test Steps:

Create a `RequestFactory` instance for a GET request to an invalid URL `/orders/sell/{invalid_id}`.
Call `sell_orders()` with the request.

Test Results:

- Expected: Response status code should be 200.
- Expected: Response should contain an empty list indicating no sell orders found for the invalid property id.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_get_sell_orders_with_invalid_id_returns_empty_list` function.
- Statement Coverage: All statements within the function executed.

Unit Test Document: TestSupport

Test Owner: Chinmay, Shivang

Test Date: 25/03/2024

Test Cases:

1. Test Case: test_post_request_to_support_with_valid_data_sends_message_to_support

- Description: Tests sending a message to support with valid data.
- Function Tested: `support()`

Test Steps:

Create a valid data dictionary containing 'name', 'email', and 'message'.
Create a `RequestFactory` instance for a POST request to `/support/` with the valid data.
Call `support()` with the request.

Test Results:

- Expected: Response status code should be 201.
- Expected: Response data should be {'message': 'Message sent successfully'} indicating successful message sent.

Structural Coverage:

- Branch Coverage: All branches covered within the `test_post_request_to_support_with_valid_data_sends_message_to_support` function.
- Statement Coverage: All statements within the function executed.

3 Integration Testing

1. Navbar: links

Module Details: The navbar has links for going to the homepage, support page, funds, wishlist page, register /login page, portfolio page, The aim is to check it's working.
Test Owner: Armeet Luthra
Test Date: 28/03/24 to 29/03/24
Test Results Expected: When not logged in, clicking on support, wishlist, portfolio,, register, funds page redirects to sign-in page. When logged in, respective pages are opened. Clicking on home page link transfers to home page in both scenarios.

Additional Comments: *Expected behaviour observed*

2. Navbar: Responsiveness

*[***Module Details:** The navbar has links for going to homepage, support page, funds, wishlist page, register /login page, portfolio page, The aim is to check behaviour when screen size is minimised

Test Owner: Armeet Luthra
Test Date: 28/03/24 to 29/03/24
Test Results Expected: On smaller screens (<600px) the navbar should collapse to an icon, which when clicked makes the various links available.

Additional Comments: *Expected behaviour observed*

3. Homepage: Property listing

Module Details: The homepage has properties currently listed for sale. The aim is to check behaviour on various actions.
Test Owner: Armeet Luthra
Test Date: 28/03/24 to 29/03/24
Test Results Expected: On hovering over each property card, it gets highlighted. On clicking view details, transfer to particular property page should happen.

Additional Comments: Expected behaviour observed

4. **Property Page: Details about property**

Module Details: Details of property along with option to place market/limit order, add to wishlist is there. The aim is to check behaviour on clicking various options.

Test Owner: Armeet Luthra
Test Date: 28/03/24 to 29/03/24
Test Results Expected: If property is not in user portfolio or funds are insufficient, sell order cannot be placed and error dialog box will appear. If property is already bought or funds are insufficient then buy order cannot be placed. Buy cannot be placed if no sell order is there, same is for sell orders. Market buy matches with maximum sell bid and Market sell with minimum buy bid. Once order is matched top 5 buy/sell bids are updated. On clicking add to wishlist, property is added to wishlist page.

Additional Comments: *Expected behaviour observed and appropriate dialog box appears in case of exceptions.*

5. **Portfolio Page: Holdings**

Module Details: The portfolio page shows current properties owned by the user logged in.

Test Owner: Armeet Luthra
Test Date: 28/03/24 to 29/03/24
Test Results Expected: List of properties owned by the user to be displayed along with their details. On clicking property details, redirection to the designated property page happens.

Additional Comments: *Expected behaviour observed*

6. **Wishlist page: Properties to buy**

Module Details: The wishlist page shows properties watchlisted by the user logged in.

Test Owner: Armeet Luthra
Test Date: 28/03/24 to 29/03/24
Test Results Expected: List of properties marked as watchlisted by the user to be displayed along with their details. On clicking property details, redirection to the designated property page happens.

Additional Comments: *Expected behaviour observed*

7. **Funds Page: Wallet amount**

Module Details: The funds page shows the current wallet balance, option to add money and withdraw money from it. The aim is to check behaviour for edge cases.

Test Owner: Armeet Luthra
Test Date: 28/03/24 to 29/03/24
Test Results Expected: If the withdrawn amount is greater than the current balance, an error should be displayed. Any amount can be added to the wallet.
Additional Comments: *Expected behaviour observed*

8. **Register/Login page**

Module Details: The login page has the option to enter username and password to login and link to go to the register page for registration of new users.

Test Owner: Armeet Luthra

Test Date: 28/03/24 to 29/03/24
Test Results Expected: If format of details is not met or details already exist, then error should be displayed. On successful login, redirection to last opened page. On successful registration redirected to login page.
Additional Comments: *Expected behaviour observed*

9. **Support page**

Module Details: The support page has a list of frequently asked questions along with an option to leave a message. The aim is to check if FAQs are properly displayed and a user is able to submit the message.

Test Owner: Armeet Luthra
Test Date: 28/03/24 to 29/03/24
Test Results Expected: On clicking FAQs, a list of questions is to be displayed and on submitting the message a success dialog box should be shown..
Additional Comments: *Expected behaviour observed*

10. **Overall responsiveness**

Module Details: The aim is to check behaviour of every component when screen size is minimised

Test Owner: Armeet Luthra

Test Date: 28/03/24 to 29/03/24

Test Results Expected: <900px, <600px are two breakpoints and components should rearrange for better visual appeal and user experience.
Additional Comments: *Expected responsiveness observed*

4 System Testing

Requirement: Deployment

- *The system must build and deploy by following the instruction presented in the README.md file*

Test Owner: Chinmay, Utpal
Test Date: *[15/03/2024] - [24/03/2024]*
Test Results: *Upon cloning and deploying a new local repository as per the instructions, it was observed that the Django API server failed to run despite it working as expected in the local working repositories of the developers. Upon further testing, debugging and checking previous commits, it was discovered that the cause of the failure was the accidental deletion of the Django migrations folder by one of the team members. The fatal error went undetected for a week since all the migrations had been cached and stored in the "__pycache__" folder of the developers. The cache is included in the .gitignore file and hence isn't pushed to the online GitHub repository. Since Django prefers to utilise the cached migrations, the local working repositories were working the same despite losing vital files due to the cache staying intact. The deletion of the "migrations/__init__.py" file prevented "python manage.py makemigration" from re-making the migrations.*
Additional Comments: *The issue was fixed by reverting the HEAD of the new local repository to a commit before the accidental deletion, copy-pasting the migration folder files into the other local working repository that is at the latest commit and pushing the changes.*

Functional Requirement Testing:

1. **Requirement:** *Property Display*
   - *The application must display a comprehensive list of all available properties.*
   - *Users should be able to navigate to a page providing further details by clicking on their desired property.*

Test Owner: Chinmay
Test Date: *[15/03/2024] - [29/03/2024]*
Test Details:

1. *Go to "localhost:5137" in the browser and check if the properties are displaying as expected.*
2. *Click on each of the properties.*
3. *They should take you to their corresponding property page.*

Test Results: *All the properties are shown as expected and take you to the corresponding property pages.*
Additional Comments: *Test passed.*

2. **Requirement:** *eKYC*
   • The broker must only allow users to interact with the application after eKYC procedure

Test Owner: Chinmay
Test Date: *[15/03/2024] - [29/03/2024]*
Test Details:

1. *Go to “localhost:5137” in browser and try to sign up without verifying your PAN card.*
2. *Tried utilizing all the features of the website without verification process.*

Test Results: *Was able to interact with the application without completing the eKYC procedure.*
Additional Comments: *Tests failed. (Since setting up eKYC will require legal proceedings, we will not be implementing it. But it should be a part of the final product released to consumers for reporting taxes.)*

3. **Requirement:** *Market Order Option*
   • Users must be allowed to execute market orders to buy/sell at the current best available sell/buy price.
   • If funds are available, the system must automatically subtract them on buy.
   • If the required amount of funds is not available, the system must provide a 'no funds' alert. The buy order must not go through unless the required amount of funds are present.
   • On a successful sell, the system must add funds to the user's account.

Test Owner: Chinmay
Test Date: *[15/03/2024] - [29/03/2024]*
Test Details:

1. *Go to the property details page of any property.*
2. *Click on market order buy or market or sell option(required you to own the property).*
3. *The broker will execute market orders to buy and sell properties at current best available prices*
4. *If you do not own the property and are trying to sell it, you will be prompted with an error and the market order will not execute.*
5. *Verify if funds are automatically subtracted upon buy orders.*
6. *Test if a 'no funds' alert is displayed when insufficient funds are available for a buy order. The market order should not execute in this case.*
7. *Ensure successful sell order results in funds being added to the user's account.*
8. *Ensure successful buy order results in the property being added to the user's portfolio.*

Test Results: *Working as expected*
Additional Comments: *Test passed.*

4. **Requirement:** *Limit Order Option*
   • *The platform should enable users to place buy/sell limit orders at specified prices.*
   • *The bids must get added onto the order book.*
   • *If the required amount of funds is not available, the system must provide a 'no funds' alert. A buy order must not go through unless the required amount of funds are present.*
   • *The system must automatically subtract funds when the buy order goes through.*
   • *On a successful sell, the system must add funds to the user's account.*

Test Owner: Chinmay
Test Date: *[15/03/2024] - [29/03/2024]*
Test Details:

1. *Go to the property details page of any property.*
2. *Place buy and sell limit orders at specified prices.*
3. *There must be a pop up asking you to specify your price for the property.*
4. *Verify if bids get added to the order book.*
5. *Test 'no funds' alert functionality for buy orders with insufficient funds.*
6. *Place limit order buy on a property you don't own and limit order sell on a property that you do own, both at the current best prices.*
7. *Login to another user's account, add the required funds and place a buy market order.*
8. *Ensure this results in the sell limit order going through, it being deleted from the order book, property being removed from their portfolio and funds being added to the original user's account.*
9. *Login to the user's account who owns the property you initially put a limit order on and place a sell market order.*
10. *Ensure this results in the buy limit order going through, it being deleted from the order book, funds being deducted from the original user's account.*

Test Results: *All tests performed as expected.*
Additional Comments: *Test passed.*

5. **Requirement:** *Order Book*
   • *The system should display the current best 5 buy and best 5 sell bids for a specific property*

Test Owner: Chinmay
Test Date: *[15/03/2024] - [29/03/2024]*

Test Details:

1. *Check if the system displays the current best 5 buy and sell bids for a specific property on its property details page.*

Test Results: *The system correctly displayed the best 5 buy and sell bids.*
Additional Comments: *Test passed.*

6. **Requirement:** *Property Details*
   • The platform should show the location, size, price and other necessary details about the properties.
   • The price of every property must be updated every minute so as the avoid stale data.

Test Owner: Chinmay
Test Date: *[15/03/2024] - [29/03/2024]*
Test Details:

1. *Go to "localhost:5137" in the browser and go to explore buying.*
2. *Then select one of the properties whose info you need to view.*
3. *This opens up the details of the property like location, size, price, etc., and also provide various market and limit order options.*

Test Results: *All property details were displayed accurately, and prices were updated as expected.*
Additional Comments: *Test passed.*

7. **Requirement:** *Portfolio Overview*
   • The system must display all properties that the user has purchased.
   • The platform should show the quantity and relevant details for each owned item.

Test Owner: Chinmay
Test Date: *[15/03/2024] - [29/03/2024]*
Test Details:

1. *Go to "localhost:5137" in the browser and login your account.*
2. *Then go to the portfolio tab and you would be able to view all the properties owned by you.*
3. *Also, the addition of new properties to your portfolio is being updated.*

Test Results: *The portfolio overview displayed all owned properties with relevant details.*
Additional Comments:*Test passed.*

8. **Requirement:** *Watchlist Properties*
   • The system should allow users to add any property they are interested in to a watchlist.

• The platform should have a page that shows only the properties watchlisted by the user.

• The platform should display the current prices of the items on the watchlist.

Test Owner: Chinmay

Test Date: *[15/03/2024] - [29/03/2024]*

Test Details:

1. *Go to “localhost:5137” in the browser and login your account.*
2. *Then go to the watchlist tab and you would be able to view all the properties watchlisted by you.*
3. *Also, the addition of watchlisted properties is being updated.*

Test Results: *The watchlist functionality worked correctly, displaying watchlisted properties with current prices.*

Additional Comments: *Test passed.*

9. **Requirement:** *Funds*

• The platform must show the current balance of funds in the user's account.

• The system must enable users to add funds to their account.

• The platform should redirect users to a payment portal to adding funds to their account.

• The system must allow users to withdraw funds from their account.

Test Owner: Chinmay

Test Date: *[15/03/2024] - [29/03/2024]*

Test Details:

1. *Go to “localhost:5137” in the browser and login your account.*
2. *Then go to the funds tab and you would be able to view the total balance.*
3. *Able to add funds with updation of current funds and notify in case of invalid input.*
4. *Able to withdraw funds with updation of current funds and notify in case of invalid input.*

Test Results:*All fund-related functionalities worked as expected.*

Additional Comments: *Test passed.*

10. **Requirement:** *New Properties*

• The system should allow the admin to add new properties to the list of available products.

Test Owner: Chinmay

Test Date: *[15/03/2024] - [29/03/2024]*

Test Details:

1. *Run the server and database and go to 'http://127.0.0.1:8000/admin' and login with the admin id created.*
2. *Admin logged in successfully and was able to view 'propertys' through which he can add new properties along with its details.*

Test Results: *New property was successfully updated on the lists of properties on the frontend.*
Additional Comments: *Test passed.*

11. Requirement: *Customer Support*

• The system should provide basic contact information for customer support that users can reach out for assistance or queries.

Test Owner: Chinmay
Test Date: *[15/03/2024] - [29/03/2024]*
Test Details:

1. *Go the 'localhost:5137' and click on support tab.*
2. *Type the name, email and message and click submit.*
3. *This message successfully can be view in the django terminal by admin by visiting 'http://127.0.0.1:8000/admin' and then going to support tab.*
4. *WebApp also provides some FAQ for users refernce*

Test Results: *The system provided basic contact information for customer support and successfully captured the grievances in the backend by the admin.*
Additional Comments: *Test passed.*

Non-Functional Requirement Testing:

Performance Requirements :

1.Requirement: *Property Listing Page Load Time*

Test Owner: *Chinmay*

Test Date: *[15/03/2024] - [29/03/2024]*

Test Details:

1. *Access the property listing page.*
2. *Measure the time taken for the page to load.*
3. *Verify that the page loads within 3 seconds as per the requirement.*

Test Results*: The property listing page consistently loads within 3 seconds.*

Additional Comments*: Test passed.*

2.Requirement: *Property Purchase Transaction Processing Time*

Test Owner: *Chinmay*

Test Date: *[15/03/2024] - [29/03/2024]*

Test Details:

1. *Initiate property purchase transactions.*
2. *Measure the time taken for the transactions to be processed.*
3. *Ensure that transactions are completed within 5 seconds as specified.*

Test Results: *Property purchase transactions are processed within 5 seconds.*

Additional Comments: *Test passed.*

3.Requirement: *Concurrent Users*

Test Owner: *Utpal, Chinmay*

Test Date: *[15/03/2024] - [29/03/2024]*

Test Details:

1. *Simulate concurrent users browsing property listings.*
2. *Ensure that the system supports at least 50 concurrent users without significant performance degradation.*

Test Results*: The system is tested with 10 concurrent users and is built to support far more than 50 concurrent users. Since test for 50 users manually will be very inefficient, we hence consider the test on 10 users as a sufficient for this.*

Additional Comments: *Test passed.*

4.Requirement: *Database Response Time*

Test Owner: *Utpal*

Test Date: *[15/03/2024] - [29/03/2024]*

Test Details:

1. *Execute database queries.*
2. *Measure the time taken for the queries to return results.*
3. *Verify that database queries return results within 2 seconds as required.*

Test Results: *Database queries consistently return results within 2 seconds.*

Additional Comments: *Test passed.*

5.Requirement: *Atomicity in Database*

Test Owner: *Utpal, Chinmay*

Test Date: *[15/03/2024] - [29/03/2024]*

Test Details:

1. *Test simultaneous purchases of the same property by different buyers.*
2. *Verify that atomicity is enforced in the database to prevent conflicts in property ownership.*

Test Results: *Atomicity is enforced in the database, preventing conflicts in property ownership. This is ensure by the use of MySQL database that is ACID compliant.*

Additional Comments: *Test passed.*

6.Requirement: *Cross-device Compatibility*

Test Owner: *Chinmay*

Test Date: *[15/03/2024] - [29/03/2024]*

Test Details:

1. *Access the application from multiple devices such as mobile, tablet, and laptop with different resolutions.*
2. *Ensure that the application is fully functional without any issues on all devices.*

Test Results: *The application is fully functional on multiple devices with different resolutions.*

Additional Comments: *Test passed.*

Safety and Security Requirements:

1.Requirement: *User Authentication*

Test Owner*: Chinmay*

Test Date*: [15/03/2024] - [29/03/2024]*

Test Details*: Successful authentication with valid credentials.*

Test Results*: Authentication with valid credentials verified.*

Additional Comments: *Test passed.*

2.Requirement: *Data Protection*

Test Owner*: Chinmay*

Test Date*: [15/03/2024] - [29/03/2024]*

Test Details*: Personal information securely transferred using TLS encryption using HTTPS protocol.*

Test Results*: HTTPS has not been implemented yet. It can only be implemented once the web application is deployed.*

Additional Comments: *Test failed.*

3.Requirement:*User Authorization*

Test Owner*: Utpal, Chinmay*

Test Date: *[15/03/2024] - [29/03/2024]*

Test Details: *Role-based access permissions confirmed.*

Test Results*: admin and regular users have different permissions. Regular users are not allowed to login into the admin terminal.*

Additional Comments: *Test passed.*

5 Conclusion

How Effective and exhaustive was the testing?

The testing conducted for the Django application was effective in ensuring basic functionality and handling various scenarios. It covered a wide range of features including user registration, login, portfolio management, market orders, limit orders, fund management, order book, and support messages. Each test case was designed to cover different scenarios such as valid inputs, invalid inputs, edge cases, and error handling. The tests achieved a high level of statement and branch coverage, ensuring that most paths of the code were executed. The testing procedure was a thorough and effective combination of manual and automated tests, ensuring a comprehensive coverage of both functional and structural aspects. The exhaustive nature of the testing is evident, with a coverage rate of approximately 90%. Various scenarios, including valid inputs, invalid inputs, edge cases, and error handling, were meticulously tested.

Which components have not been tested adequately?

Stress and load testing, which evaluates system performance under heavy loads, have not been thoroughly conducted. These tests are crucial for identifying potential bottlenecks, scalability issues, and the system's overall resilience. Implementing comprehensive automated integration tests and stress/load testing would provide a more thorough assessment of the system's reliability and performance in real-world scenarios.

What difficulties have you faced during testing?

During testing, several challenges were encountered. One significant difficulty was ensuring proper setup and teardown of test data to maintain test independence and consistency. Managing complex interactions between different components and databases also posed challenges, requiring careful orchestration of test scenarios. Handling edge cases and boundary conditions effectively was another hurdle, as these scenarios often revealed unexpected behavior. Additionally, ensuring comprehensive coverage without overly redundant tests demanded strategic planning and design. Debugging and diagnosing issues in a timely manner, especially with complex test suites, also presented its own set of challenges. These difficulties required patience, meticulous attention to detail, and creative problem-solving to overcome.

How could the testing process be improved?

The testing process could be improved by implementing automated tests for system testing, which would enhance efficiency and coverage. Additionally, adding detailed

comments to functions during the application implementation phase would aid in writing effective test functions later on. Incorporating more integration tests to ensure proper interaction between different modules and databases would also be beneficial. Furthermore, creating a robust test plan with clear objectives and priorities, along with regular reviews and updates to the test suite, would enhance the overall effectiveness of the testing process.

Appendix A - Group Log

Meeting date and time	Description
22/03/2024 8pm - 10pm	The group met to discuss the roadmap for testing and distribute the parts among the group members.
27/03/2024 9pm - 10pm	Met to review the progress of the test document and make sure that the work done by each member is consistent with the others.
29/03/2024 3pm - 4pm	Met to discuss the final tests and finalise the document and remove any inconsistencies or errors.