Discord Bot Automation Assistant

Discord Bot Automation Assistant Test Plan

Oguz Kaan Yildirim

307637

Table Of Contents

INTRDOCUTION

The purpose of this document is to provide a comprehensive test plan for the Discord Bot Automation Assistant project, reflecting recent updates and improvements in the testing process and methodologies. This plan has been revised to incorporate the utilization of pytest and asyncio, enhancing the testing framework to support asynchronous operations and improve test coverage. The intent is to verify the functionality and reliability of the updated components, ensuring that they meet the required standards and perform optimally within the project's ecosystem.

To have a better look at the code/project especially to testing source codes; codes can be found in github. Test files are under UnitTesting Folder.

https://github.com/oguzky7/DiscordBotProject_CISC699/tree/develop/UnitTesting

TEST PLAN OVERVIEW

The revised test plan is designed to systematically assess each component of the project, ensuring robustness and error handling with the introduction of pytest and asyncio. The testing approach categorizes tests into different suites focusing on:

- **Entity Objects Testing:** Ensures that each entity object functions correctly, manages state appropriately, and integrates with other components seamlessly.
- **Control Objects Testing:** Verifies that control objects accurately manage the logic and data flow between the user interface and the data management layers, particularly focusing on asynchronous behavior.
- **Boundary Objects Testing:** Tests the user interface components for correct data capture and validation, and that they correctly pass data to control objects.

  Integration Testing:
- Although the main focus remains on unit testing, this plan includes tests to verify that components work together as expected. This is facilitated by mocking and simulating external dependencies, thus adhering to unit testing principles while ensuring comprehensive integration coverage.

This methodical approach ensures that each component of the Automated Discord Bot Helper is tested thoroughly, thereby minimizing the risk of defects and ensuring a high-quality software product.

Mock and Fake Implementation: Critical to avoid direct database interactions or file system accesses, mock objects and fakes will be used extensively to simulate the external dependencies, ensuring that the tests remain fast, reliable, and repeatable. This approach allows for the testing of error handling and edge cases without the overhead of a live environment.

Each test case described in this plan will outline the expected behavior, the steps to execute the test, the mock or fake data involved, and the anticipated outcomes, ensuring comprehensive coverage of all functionalities. This methodical approach ensures that all aspects of the "Automated Discord Bot Helper" are rigorously tested, thereby minimizing the risk of defects and ensuring a high-quality software product.

By adhering to these guidelines, the test plan aims to validate the functionality thoroughly and reliability of the system, ensuring that it meets all specified requirements and is robust against potential errors or failures.

To have a better look at the code/project especially to testing source codes; codes can be found in github. Test files are under UnitTesting Folder.

https://github.com/oguzky7/DiscordBotProject_CISC699/tree/develop/UnitTesting

TEST CASES

Test Case 0: test_init.py

Tools and Technologies

This test initialization setup incorporates a suite of tools designed to support comprehensive unit testing of the "Discord Bot Automation Assistant" project with enhancements for asynchronous operation handling:

- **Python**: The primary programming language used for both application development and test case formulation.
- **pytest**: Utilized for executing advanced testing frameworks, replacing unittest to leverage its robust fixtures and parameterization capabilities.
- **pytest-asyncio**: A plugin for pytest that provides support for testing asynchronous functions and features.
- **pytest-mock**: Enhances traditional mocking frameworks, offering more flexible and powerful mocking capabilities, crucial for simulating complex object behaviors in a controlled environment.
- **asyncio**: Facilitates asynchronous I/O, event loop, coroutines, and tasks, essential for testing asynchronous operations within the project.

Purpose and Setup

The test_init.py file provides the foundational setup for all other test scripts, ensuring a consistent testing environment across various test cases. This setup is critical for maintaining the integrity and consistency of asynchronous tests throughout the project, reducing redundancy and increasing efficiency.

Implementation Details

- **Mocking Asynchronous Interactions**: With the project significantly interfacing with asynchronous operations, the pytest-asyncio and pytest-mock tools are extensively used. These tools allow for the effective simulation of asynchronous interactions such as database accesses or API communications without actual network operations.

- **Common Test Setup**: Leveraging pytest fixtures, a standardized test environment is established that includes the configuration of mock objects and the setup of the asyncio event loop. This ensures that each test can run independently and concurrently, reducing test time and improving scalability.
- **Isolation and Patching**: Tests are isolated using pytest-mock to patch external dependencies effectively. This isolation helps ensure that each component is tested against predefined responses, safeguarding the tests against external variability and enhancing their reliability.

How It Works

- **Environment Configuration**: At the start of each test, test_init.py configures the necessary environment, setting up mock objects and initializing the asyncio event loop to simulate the system's asynchronous nature.
- **Execution of Asynchronous Operations**: Tests simulate the behavior of asynchronous methods within the system, such as handling web requests or performing database operations, ensuring that each function performs as expected under controlled conditions.
- **Consistent Testing Framework**: By centralizing the test setup, the framework ensures consistency across all tests, minimizing the effort needed to adapt tests to changes in the system architecture or external libraries.

This revised setup not only tests the system's functionality under expected conditions but also prepares it to handle unexpected or edge cases, thereby ensuring the system's robustness and reliability.

Source Code

```
#test_init.py
import sys, os, logging, pytest, asyncio
import subprocess
from unittest.mock import patch, MagicMock
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

#pytest -v > test_results.txt
#Run this command in the terminal to save the test results to a file

async def run_monitoring_loop(control_object, check_function, url, date_str, frequency, iterations=1):
    """Run the monitoring loop for a control object and execute a check function."""
    control_object.is_monitoring = True
    results = []

    while control_object.is_monitoring and iterations > 0:
        try:
            result = await check_function(url, date_str)
        except Exception as e:
            result = f"Failed to monitor: {str(e)}"
        logging.info(f"Monitoring Iteration: {result}")
        results.append(result)
        iterations -= 1
        await asyncio.sleep(frequency)

    control_object.is_monitoring = False
    results.append("Monitoring stopped successfully!")

    return results

def setup_logging():
    """Set up logging without timestamp and other unnecessary information."""
    logger = logging.getLogger()
    if not logger.hasHandlers():
        logging.basicConfig(level=logging.INFO, format='%(message)s')
def save_test_results_to_file(output_file="test_results.txt"):
    """Helper function to run pytest and save results to a file."""
    print("Running tests and saving results to file...")
    output_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), output_file)
    with open(output_path, 'w') as f:
        # Use subprocess to call pytest and redirect output to file
        subprocess.run(['pytest', '-v'], stdout=f, stderr=subprocess.STDOUT)

# Custom fixture for logging test start and end
@pytest.fixture(autouse=True)
def log_test_start_end(request):
    test_name = request.node.name
    logging.info(f"------------------------------------------------------\nStarting test: {test_name}\n")

    # Yield control to the test function
```

```
    yield

    # Log after the test finishes
    logging.info(f"\nFinished test: {test_name}\n----------------------------------------------------")

# Import your control classes
from control.BrowserControl import BrowserControl
from control.AccountControl import AccountControl
from control.AvailabilityControl import AvailabilityControl
from control.PriceControl import PriceControl
from control.BotControl import BotControl

@pytest.fixture
def base_test_case():
    """Base test setup that can be used by all test functions."""
    test_case = MagicMock()
    test_case.browser_control = BrowserControl()
    test_case.account_control = AccountControl()
    test_case.availability_control = AvailabilityControl()
    test_case.price_control = PriceControl()
    test_case.bot_control = BotControl()
    return test_case

if __name__ == "__main__":
    # Save the pytest output to a file in the same folder
    save_test_results_to_file(output_file="test_results.txt")
```

Test Case 1: Add Account

Description

This test case validates the functionality of the account addition process within the Discord bot system. It ensures the add_account function correctly handles both successful additions and various error scenarios using asynchronous testing methods. The test verifies that the system can process and validate account information, manage database interactions appropriately, and provide accurate user feedback.

Steps

1. **Setup and Mock Initialization:**
    - The test initializes by setting up the necessary mocks and test environment using pytest fixtures.
    - The AccountControl object responsible for account management is accessed, and its methods are prepared for mocking.
2. **Entity Layer Interaction:**
    - The test simulates the account addition process at the entity layer by mocking the AccountDAO.add_account method.
    - It sets up expected outcomes for both successful additions and various error scenarios (invalid data, entity errors, and duplicate accounts).
3. **Control Layer Execution:**
    - Executes the add_account method on the control object, passing mock inputs corresponding to different test scenarios.
    - Captures the results from the control layer, which include both the return values and any exceptions raised during the process.
4. **Assertions and Logging:**
    - Each scenario checks that the outcomes at the entity and control layers match the expected results.
    - Logs the start and end of each test, along with detailed information about expected versus actual results, ensuring transparency and traceability of the test execution.

5. **User Feedback Simulation:**
    - The tests simulate user feedback based on the control layer's output, ensuring that the system responds appropriately to the user based on the outcome of the account addition attempt.

Test Data

- **Valid Account Data:**
    - Username: "testuser"
    - Password: "password123"
    - Website: "example.com"
- **Invalid Account Data:**
    - Username: "" (empty)
    - Password: "" (empty)
    - Website: "example.com"

Test Scenarios, Expected Outcomes, Actual Outcomes

```
----------------------------------------------------
Starting test: test_add_account_success

Entity Layer Expected: Account for example.com added successfully.
Entity Layer Received: Account for example.com added successfully.
Unit Test Passed for entity layer.
Control Layer Expected: Account for example.com added successfully.
Control Layer Received: Account for example.com added successfully.
Unit Test Passed for control layer.

Finished test: test_add_account_success
----------------------------------------------------

Starting test: test_add_account_failure_invalid_data

Control Layer Expected: Failed to add account for example.com.
Control Layer Received: Failed to add account for example.com.
Unit Test Passed for control layer invalid data handling.

Finished test: test_add_account_failure_invalid_data
----------------------------------------------------
```

Starting test: test_add_account_failure_entity_error

Control Layer Expected: Control Layer Exception: Database Error
Control Layer Received: Control Layer Exception: Database Error
Unit Test Passed for control layer error handling.

Finished test: test_add_account_failure_entity_error

Starting test: test_add_account_already_exists

Control Layer Expected: Failed to add account for example.com. Account already exists.
Control Layer Received: Failed to add account for example.com. Account already exists.
Unit Test Passed for control layer when account already exists.

Finished test: test_add_account_already_exists

Source Code

Source Code can be found at the end of this document and in GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Test Case 2: Delete Account

Description

This test case validates the functionality of the account deletion process within the Discord bot system. It ensures the delete_account function can correctly handle both successful deletions and various error scenarios, using asynchronous testing methods. The test verifies that the system can manage account deletions, handle errors from the data access object (DAO), and provide accurate feedback based on the deletion outcome.

Steps

1. **Setup and Mock Initialization:**
    - Initializes the test environment using pytest fixtures to set up mocks and other test prerequisites.
    - Accesses the AccountControl object responsible for account management, preparing it for method mocking.
2. **Entity Layer Interaction:**
    - Simulates the account deletion process at the entity layer by mocking the AccountDAO.delete_account method.
    - Configures the mock to return both true (successful deletion) and false (deletion failed) based on the scenario.
3. **Control Layer Execution:**
    - Executes the delete_account method on the control object with predefined account IDs to test both success and failure conditions.
    - Captures and logs the outcomes from the control layer, including both the results of the deletion and any exceptions triggered during the process.
4. **Assertions and Logging:**
    - Verifies that the outcomes at both the entity and control layers match the expected results for each test scenario.
    - Logs detailed information about the setup, execution, and the end results, providing clear documentation of the test flow and outcomes.

5. **User Feedback Simulation:**
    - Simulates user feedback based on the outputs of the delete_account command to ensure that the system responds appropriately to the user after attempting to delete an account.

Test Data

- **Valid Account Data:**
    - Account ID: 1 (exists in the system)
- **Invalid Account Data:**
    - Account ID: 999 (does not exist)

Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test_delete_account_success

Entity Layer Expected: Account with ID 1 deleted successfully.
Entity Layer Received: True
Unit Test Passed for entity layer.

Control Layer Expected: Account with ID 1 deleted successfully.
Control Layer Received: Account with ID 1 deleted successfully.
Unit Test Passed for control layer.

Finished test: test_delete_account_success

Starting test: test_delete_account_not_found

Control Layer Expected: Failed to delete account with ID 999.
Control Layer Received: Failed to delete account with ID 999.
Unit Test Passed for control layer with account not found.

Finished test: test_delete_account_not_found

Starting test: test_delete_account_failure_entity

Control Layer Expected: Error deleting account.
Control Layer Received: Error deleting account.
Unit Test Passed for entity layer error handling.

Finished test: test_delete_account_failure_entity

```
-------------------------------------------------------
Starting test: test_delete_account_failure_control

Control Layer Expected: Control Layer Exception: Control Layer Failed
Control Layer Received: Control Layer Exception: Control Layer Failed
Unit Test Passed for control layer failure.

Finished test: test_delete_account_failure_control
-------------------------------------------------------
```

Source Code

Source Code can be found at the end of this document and in GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Test Case 3: Fetch All Accounts

Description

This test case verifies the functionality of the fetch_all_accounts method in the Discord bot system. It checks the ability of the system to retrieve a list of all accounts, handle scenarios where no accounts exist, and manage database errors effectively using asynchronous testing methods.

Steps

1. **Setup and Mock Initialization:**
    - Initialize testing environment with pytest fixtures.
    - Mock the AccountDAO.fetch_all_accounts method to control database interactions.
2. **Entity Layer Interaction:**
    - Simulate fetching all accounts from the database.
    - Prepare expected outcomes for successful retrieval, no accounts found, and database errors.
3. **Control Layer Execution:**
    - Call the fetch_all_accounts method on the control object.
    - Capture and log the results from the control layer for each scenario.
4. **Assertions and Logging:**
    - Verify that the actual results match the expected outcomes for both entity and control layers.
    - Detailed logging of expected vs. actual results ensures clarity and traceability.
5. **User Feedback Simulation:**
    - Simulate user feedback based on the control layer's output, ensuring appropriate responses are generated for the user based on the method's execution outcome.

Test Data

- **No specific input data required** as the method fetches all existing accounts.

Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test_fetch_all_accounts_success

Entity Layer Expected: Accounts:
ID: 1, Username: user1, Password: pass1, Website: example.com
ID: 2, Username: user2, Password: pass2, Website: test.com
Entity Layer Received: [(1, 'user1', 'pass1', 'example.com'), (2, 'user2', 'pass2', 'test.com')]
Unit Test Passed for entity layer.

Control Layer Expected: Accounts:
ID: 1, Username: user1, Password: pass1, Website: example.com
ID: 2, Username: user2, Password: pass2, Website: test.com
Control Layer Received: Accounts:
ID: 1, Username: user1, Password: pass1, Website: example.com
ID: 2, Username: user2, Password: pass2, Website: test.com
Unit Test Passed for control layer.

Finished test: test_fetch_all_accounts_success

Starting test: test_fetch_all_accounts_no_accounts

Control Layer Expected: No accounts found.
Control Layer Received: No accounts found.
Unit Test Passed for control layer no accounts found.

Finished test: test_fetch_all_accounts_no_accounts

Starting test: test_fetch_all_accounts_failure_entity

Control Layer Expected: Error fetching accounts.
Control Layer Received: Error fetching accounts.
Unit Test Passed for entity layer error handling.

Finished test: test_fetch_all_accounts_failure_entity

Source Code

Source Code can be found at the end of this document and in GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Test Case 4: Fetch Account by Website

Description

This test case checks the fetch_account_by_website functionality, ensuring the system can retrieve account details based on the website, handle scenarios where no account is linked to a website, and manage both entity and control layer errors effectively.

Steps

1. **Setup and Mock Initialization:**
    - Using pytest fixtures to prepare the testing environment.
    - Mock AccountDAO.fetch_account_by_website to simulate database responses.
2. **Entity Layer Interaction:**
    - Simulate the retrieval process for an account associated with a specific website.
    - Set up expected results for successful retrieval, no account found, and simulated database errors.
3. **Control Layer Execution:**
    - Invoke the fetch_account_by_website with varying test inputs.
    - Log the outcomes from the control layer for different test cases.
4. **Assertions and Logging:**
    - Confirm that the results align with pre-defined expectations.
    - Provide detailed logs for expected and actual results for transparency.
5. **User Feedback Simulation:**
    - Simulate system responses to the user based on the fetched results or errors encountered.

Test Data

- **Valid Website:** "example.com"
- **Nonexistent Website:** "nonexistent.com"

Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test_fetch_account_by_website_success

Entity Layer Expected: ('sample_username', 'sample_password')
Entity Layer Received: ('sample_username', 'sample_password')
Unit Test Passed for entity layer.

Control Layer Expected: ('sample_username', 'sample_password')
Control Layer Received: ('sample_username', 'sample_password')
Unit Test Passed for control layer.

Finished test: test_fetch_account_by_website_success

Starting test: test_fetch_account_by_website_no_account

Control Layer Expected: No account found for nonexistent.com.
Control Layer Received: No account found for nonexistent.com.
Unit Test Passed for control layer no account found.

Finished test: test_fetch_account_by_website_no_account

Starting test: test_fetch_account_by_website_failure_entity

Control Layer Expected: Error: Database Error
Control Layer Received: Error: Database Error
Unit Test Passed for entity layer error handling.

Finished test: test_fetch_account_by_website_failure_entity

Starting test: test_fetch_account_by_website_failure_control

Control Layer Expected: Control Layer Exception: Control Layer Error
Control Layer Received: Control Layer Exception: Control Layer Error
Unit Test Passed for control layer error handling.

Finished test: test_fetch_account_by_website_failure_control

Source Code

Source Code can be found at the end of this document and in GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Test Case 5: Launch Browser

Description

This test case verifies the functionality of the browser launch process within the Discord bot system. It focuses on the launch_browser function, ensuring it handles both the initial launch and edge cases, such as attempting to launch when the browser is already running, and properly manages internal errors using asynchronous testing methods.

Steps

1. **Setup and Mock Initialization:**
    - Utilize pytest fixtures to establish the test environment and mock necessary components.
    - Access the BrowserControl object, responsible for managing browser operations, and prepare it for interaction.
2. **Entity Layer Interaction:**
    - Mock the BrowserEntity.launch_browser method to simulate different browser states, such as already running or launch failures.
    - Define expected outcomes for each simulated state, including successful launch, already running, and error scenarios.
3. **Control Layer Execution:**
    - Call the launch_browser method on the BrowserControl object with various conditions simulated by the mocks.
    - Capture the output from the control layer, noting both successful executions and exceptions.
4. **Assertions and Logging:**
    - Verify that the control and entity layer interactions align with the expected outcomes.
    - Log detailed information on the start, execution, and completion of each test case to ensure clarity and traceability.
5. **Error Handling Simulation:**
    - Test the system's response to errors at both the entity and control layers to ensure errors are handled gracefully and appropriate messages are logged.

Test Data

- **No specific input data required** as the method fetches all existing accounts.

Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test_launch_browser_success

Entity Layer Expected: Browser launched.
Entity Layer Received: Browser launched.
Unit Test Passed for entity layer.

Control Layer Expected: Control Object Result: Browser launched.
Control Layer Received: Control Object Result: Browser launched.
Unit Test Passed for control layer.

Finished test: test_launch_browser_success

Starting test: test_launch_browser_already_running

Entity Layer Expected: Browser is already running.
Entity Layer Received: Browser is already running.
Unit Test Passed for entity layer.

Control Layer Expected: Control Object Result: Browser is already running.
Control Layer Received: Control Object Result: Browser is already running.
Unit Test Passed for control layer.

Finished test: test_launch_browser_already_running

Starting test: test_launch_browser_failure_control

Control Layer Expected to Report: Control Layer Exception: Internal error
Control Layer Received: Control Layer Exception: Internal error
Unit Test Passed for control layer error handling.

Finished test: test_launch_browser_failure_control

Starting test: test_launch_browser_failure_entity

Entity Layer Expected Failure: Failed to launch browser: Internal error
Control Layer Received: Control Layer Exception: Failed to launch browser: Internal error
Unit Test Passed for entity layer error handling.

Finished test: test_launch_browser_failure_entity

Source Code

Source Code can be found at the end of this document and in the GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Test Case 6: Close Browser

Description

This test case evaluates the functionality of closing a browser within the Discord bot system. It aims to ensure the close_browser command correctly handles both successful closures and various error scenarios using asynchronous testing methods. The test checks the system's ability to manage browser state changes accurately and provide appropriate feedback based on the outcomes.

Steps

1. **Setup and Mock Initialization:**
    - Initialize necessary mocks and test environment using pytest fixtures.
    - Access the BrowserControl object responsible for browser operations and prepare its methods for mocking.
2. **Entity Layer Interaction:**
    - Simulate the browser closure process at the entity layer by mocking the BrowserEntity.close_browser method.
    - Set up expected outcomes for successful closure, no browser open, and various error scenarios (control and entity layer errors).
3. **Control Layer Execution:**
    - Execute the close_browser method on the control object, capturing results including both return values and exceptions.
4. **Assertions and Logging:**
    - Validate that the outcomes at the entity and control layers match the expected results, using assertions.
    - Log the process start, expected vs. actual results, and test conclusion to ensure transparency and traceability.
5. **User Feedback Simulation:**
    - Simulate user feedback based on the control layer's output to ensure the system responds appropriately based on the outcome of the browser closure attempt.

Test Data

- **No specific input data required** as the method fetches all existing accounts.

Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test_close_browser_success

Entity Layer Expected: Browser closed.
Entity Layer Received: Browser closed.
Unit Test Passed for entity layer.

Control Layer Expected: Control Object Result: Browser closed.
Control Layer Received: Control Object Result: Browser closed.
Unit Test Passed for control layer.

Finished test: test_close_browser_success

Starting test: test_close_browser_not_open

Entity Layer Expected: No browser is currently open.
Entity Layer Received: No browser is currently open.
Unit Test Passed for entity layer.

Control Layer Expected: Control Object Result: No browser is currently open.
Control Layer Received: Control Object Result: No browser is currently open.
Unit Test Passed for control layer.

Finished test: test_close_browser_not_open

Starting test: test_close_browser_failure_control

Control Layer Expected to Report: Control Layer Exception: Unexpected error
Control Layer Received: Control Layer Exception: Unexpected error
Unit Test Passed for control layer error handling.

Finished test: test_close_browser_failure_control

Starting test: test_close_browser_failure_entity

Entity Layer Expected Failure: BrowserEntity_Failed to close browser: Internal error
Control Layer Received: Control Layer Exception: BrowserEntity_Failed to close browser: Internal error
Unit Test Passed for entity layer error handling.

Finished test: test_close_browser_failure_entity

Source Code

Source Code can be found at the end of this document and in GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Test Case 7 : Navigate to Website

Description

This test case evaluates the navigation functionality within the Discord bot's browser management system. It tests the bot's ability to handle valid URLs, respond to invalid URLs, manage browser states, and handle exceptions effectively. The primary goal is to ensure that the bot can navigate to a specified URL using the navigate_to_website command and provide accurate feedback regarding the success or failure of these operations.

Steps

1. **Setup and Mock Initialization:**
    - The tests initiate by setting up the necessary environment using pytest fixtures.
    - Mocks are applied to the BrowserEntity.navigate_to_website function to simulate browser interactions without actual web navigation.
2. **Entity Layer Interaction:**
    - The entity layer, which directly interacts with the web browser, is tested by simulating responses for navigation actions. These include successful navigation, URL not found, and internal browser errors.
3. **Control Layer Execution:**
    - The control layer, responsible for managing the flow of data between the user interface and entity layer, is tested for its ability to process commands and handle different outcomes from the entity layer.
4. **Assertions and Logging:**
    - Assertions are used to ensure that the expected outcomes from the entity and control layers match the predefined responses for various scenarios.
    - Detailed logs are recorded for each step of the test to ensure transparency and facilitate debugging.
5. **User Feedback Simulation:**
    - The test simulates user feedback based on the outputs from the control layer, ensuring that messages delivered to the user accurately reflect the outcomes of their navigation commands.
    -

Test Data

- **Valid URL:** "https://example.com"
- **Invalid URL:** "invalid_site"

Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test_navigate_to_website_success

Entity Layer Expected: Navigated to https://example.com
Entity Layer Received: Navigated to https://example.com
Unit Test Passed for entity layer.
Control Layer Expected: Control Object Result: Navigated to https://example.com
Control Layer Received: Control Object Result: Navigated to https://example.com
Unit Test Passed for control layer.

Finished test: test_navigate_to_website_success

Starting test: test_navigate_to_website_invalid_url

Control Layer Expected: URL for invalid_site not found.
Control Layer Received: URL for invalid_site not found.
Unit Test Passed for control layer invalid URL handling.

Finished test: test_navigate_to_website_invalid_url

Starting test: test_navigate_to_website_failure_entity

Control Layer Expected: Control Layer Exception: Failed to navigate
Control Layer Received: Control Layer Exception: Failed to navigate
Unit Test Passed for entity layer error handling.

Finished test: test_navigate_to_website_failure_entity

Starting test: test_navigate_to_website_launch_browser_on_failure

Control Layer Expected: Control Object Result: Navigated to https://example.com
Control Layer Received: Control Object Result: Navigated to https://example.com
Unit Test Passed for control layer with browser launch.

Finished test: test_navigate_to_website_launch_browser_on_failure
---

Starting test: test_navigate_to_website_failure_control

Control Layer Expected: Control Layer Exception: Control Layer Failed
Control Layer Received: Control Layer Exception: Control Layer Failed
Unit Test Passed for control layer failure.

Finished test: test_navigate_to_website_failure_control

Source Code

Source Code can be found at the end of this document and in GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Test Case 8: Login

Description

This test case evaluates the login functionality of the Discord bot system, ensuring that it can successfully log in users with valid credentials, handle cases where no account information is available, manage errors in the entity and control layers effectively, and address scenarios where the URL or selectors are not found. This robust testing guarantees that the system can reliably authenticate users across various conditions, maintaining security and user experience.

Steps

1. **Setup and Mock Initialization:**
    - Initialize the test environment using pytest fixtures to mock necessary components and set up the logging.
    - Access the BrowserControl and AccountControl objects, preparing methods such as login and fetch_account_by_website for mocking to simulate database and browser interactions.
2. **Entity and Control Layer Interaction:**
    - Simulate successful login by mocking the BrowserEntity.login method and the AccountControl.fetch_account_by_website method to return predefined credentials.
    - Handle scenarios where no account information is found, simulating the return of None from the fetch method.
    - Introduce exceptions in the entity layer to test error handling by setting the side_effect of the mock to raise an exception, simulating internal errors during the login process.
3. **Execution and Validation:**
    - Execute the login command through the control object, passing necessary parameters like the website URL.
    - Verify the responses from both the entity and control layers against expected outcomes, using assertions to ensure both layers react appropriately to each scenario.
4. **Logging and Outcome Verification:**
    - Log detailed results of each test case, including expected and actual outcomes for transparency and troubleshooting.

- Ensure the logs capture all pertinent information, aiding in debugging and validation of test results.

5. **Error and Exception Handling:**
    - Test how the control layer manages no account scenarios and various failures, ensuring robust error handling and user feedback accuracy.

Test Data

- **Valid Credentials:**
    - Username: sample_username
    - Password: sample_password
    - Website: http://example.com
- **Invalid Data:**
    - No account found for the website.

Test Scenarios, Expected Outcomes, and Actual Outcomes

Starting test: test_login_success

Entity Layer Expected: Logged in to http://example.com successfully with username: sample_username
Entity Layer Received: Logged in to http://example.com successfully with username: sample_username
Unit Test Passed for entity layer.

Control Layer Expected: Control Object Result: Logged in to http://example.com successfully with username: sample_username
Control Layer Received: Control Object Result: Logged in to http://example.com successfully with username: sample_username
Unit Test Passed for control layer.

Finished test: test_login_success

Starting test: test_login_no_account

Control Layer Expected: No account found for example.com
Control Layer Received: No account found for example.com
Unit Test Passed for missing account handling.

Finished test: test_login_no_account

Starting test: test_login_entity_layer_failure

Control Layer Expected: Control Layer Exception: BrowserEntity_Failed to log in to http://example.com: Internal error
Control Layer Received: Control Layer Exception: BrowserEntity_Failed to log in to http://example.com: Internal

```
error
Unit Test Passed for entity layer failure.

Finished test: test_login_entity_layer_failure
----------------------------------------------------------
----------------------------------------------------------
Starting test: test_login_control_layer_failure

Control Layer Expected: Control Layer Exception: Control layer failure during account fetch.
Control Layer Received: Control Layer Exception: Control layer failure during account fetch.
Unit Test Passed for control layer failure handling.

Finished test: test_login_control_layer_failure
----------------------------------------------------------
----------------------------------------------------------
Starting test: test_login_invalid_url

Control Layer Expected: URL for example not found.
Control Layer Received: URL for example not found.
Unit Test Passed for missing URL/selector handling.

Finished test: test_login_invalid_url
----------------------------------------------------------
```

Source Code

Source Code can be found at the end of this document and in the GitHub repository for a more organized overview: [GitHub - DiscordBotProject_CISC699]

Test Case 9: Get Price

Description

This test case verifies the price retrieval functionality within the Discord bot system. It focuses on the get_price command, ensuring it handles successful price fetches, invalid URLs, and various error conditions effectively. The asynchronous testing approach is employed to test the bot's ability to handle real-time data fetching and error management using the pytest and asyncio libraries.

Steps

1. **Setup and Mock Initialization:**
    - Initialize the testing environment using pytest fixtures to simulate the test context.
    - Prepare the PriceControl object by patching the PriceEntity.get_price_from_page method to control the return values and simulate different testing scenarios.
2. **Entity Layer Interaction:**
    - Mock the price fetching at the entity layer to simulate the retrieval of price information from a webpage.
    - Set expected results for successful price fetches and simulate various error scenarios like invalid URLs or entity failures.
3. **Control Layer Execution:**
    - Execute the get_price command by calling the control object with test inputs for different scenarios.
    - Capture the outputs from the control layer, which includes both the data returned and any exceptions raised during execution.
4. **Assertions and Logging:**
    - Verify that the outcomes at both the entity and control layers match the expected results.
    - Log detailed information about the test execution, documenting both expected and actual results for full transparency.
5. **Error Handling Simulation:**
    - Test how the system handles and logs errors, such as invalid URLs or internal failures, ensuring the user receives appropriate feedback.

Test Data

- **Valid URL Data:** https://example.com/product returns $199.99
- **Invalid URL Data:** invalid_url simulates an error in URL parsing or reachability.
- **Entity Layer Failure:** Simulates a failure in data fetching from the backend.

Test Scenarios, Expected Outcomes, Actual Outcomes

```
Starting test: test_get_price_success

Entity Layer Expected: $199.99
Entity Layer Received: $199.99
Unit Test Passed for entity layer.

Control Layer Expected: $199.99
Control Layer Received: $199.99
Unit Test Passed for control layer.

Finished test: test_get_price_success
----------------------------------------------------------
Starting test: test_get_price_invalid_url

Control Layer Expected: Error fetching price: Invalid URL
Control Layer Received: Error fetching price: Invalid URL
Unit Test Passed for control layer invalid URL handling.

Finished test: test_get_price_invalid_url
----------------------------------------------------------
Starting test: test_get_price_failure_entity

Control Layer Expected: Failed to fetch price: Failed to fetch price
Control Layer Received: Failed to fetch price: Failed to fetch price
Unit Test Passed for entity layer error handling.

Finished test: test_get_price_failure_entity
----------------------------------------------------------
Starting test: test_get_price_failure_control

Control Layer Expected: Control Layer Exception: Control Layer Failed
Control Layer Received: Control Layer Exception: Control Layer Failed
Unit Test Passed for control layer failure.

Finished test: test_get_price_failure_control
----------------------------------------------------------
```

Source Code

Source Code can be found at the end of this document and in GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Test Case 10: Check Availability

Description

This test case evaluates the "check availability" functionality within the Discord bot system, specifically designed to handle various scenarios involving the checking of date availability on websites. The test ensures that the system can correctly query availability, handle errors, and respond appropriately to the user, leveraging asynchronous operations for real-time processing.

Steps

1. **Setup and Mock Initialization:**
    - Utilize pytest fixtures to set up the testing environment and prepare mocks.
    - Access the AvailabilityControl object, which orchestrates the availability checking, and mock its interaction with the entity layer.
2. **Entity Layer Interaction:**
    - Simulate responses from the AvailabilityEntity object through mocking to test different availability scenarios including success, failure, and no availability.
    - Set expected results for the entity layer based on the mocked data.
3. **Control Layer Execution:**
    - Execute the check_availability command with test URLs to simulate real user interaction.
    - Capture and log results from the control layer, which processes the entity layer's data and forms the final output.
4. **Assertions and Logging:**
    - Compare the expected results with actual outcomes at both the entity and control layers to validate correctness.
    - Log detailed information about each test's execution and outcome for transparency.
5. **User Feedback Simulation:**
    - Ensure that the control layer's outputs lead to correct user feedback, simulating real-world operation and interaction within the system.
    -

Test Data

- **Valid URL for Availability Check:** "https://example.com"
- **Invalid URL or No Availability Scenario:** Simulated by appropriate mock responses.

Test Scenarios, Expected and Actual Outcomes

Starting test: test_check_availability_success

Entity Layer Expected: Selected or default date current date is available for booking.
Entity Layer Received: Selected or default date current date is available for booking.
Unit Test Passed for entity layer.

Control Layer Expected: Checked availability: Selected or default date current date is available for booking.
Control Layer Received: Checked availability: Selected or default date current date is available for booking.
Unit Test Passed for control layer.

Finished test: test_check_availability_success

Starting test: test_check_availability_failure_entity

Control Layer Expected: Failed to check availability: Failed to check availability
Control Layer Received: Failed to check availability: Failed to check availability
Unit Test Passed for entity layer error handling.

Finished test: test_check_availability_failure_entity

Starting test: test_check_availability_no_availability

Entity Layer Received: No availability for the selected date.
Control Layer Received: Checked availability: No availability for the selected date.
Unit Test Passed for control layer no availability handling.

Finished test: test_check_availability_no_availability

Starting test: test_check_availability_failure_control

Control Layer Expected: Control Layer Exception: Control Layer Failed
Control Layer Received: Control Layer Exception: Control Layer Failed
Unit Test Passed for control layer failure.

Finished test: test_check_availability_failure_control

Source Code

Source Code can be found at the end of this document and in GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Test Case 11: Start Monitoring Availability

Description

This test case examines the functionality of starting and monitoring availability for a specified URL within the Discord bot system. It tests the start_monitoring_availability method under various conditions, including successful availability checks, handling of entity and control layer errors, and managing monitoring when it is already running. This ensures the system reliably tracks availability status over time and accurately handles interruptions or errors.

Steps

1. **Setup and Mock Initialization:**
    - Initialize the test environment with necessary mocks using pytest fixtures.
    - Access the AvailabilityControl object, preparing to intercept and simulate responses from the AvailabilityEntity.
2. **Entity Layer Interaction:**
    - Simulate responses from the availability checking method using mock_check.
    - Define expected results for successful monitoring, failure scenarios, and already monitoring conditions.
3. **Control Layer Execution:**
    - Execute the start_monitoring_availability command with mocked inputs for different scenarios.
    - Capture results from the control layer, checking both return values and exception handling.
4. **Assertions and Logging:**
    - Verify that results from both entity and control layers match expected outcomes for various test cases.
    - Detailed logs capture each step, providing clarity on the expected vs. actual results for enhanced traceability.
5. **Monitoring Loop Execution:**
    - Utilize the run_monitoring_loop function to simulate continuous monitoring checks.
    - Validate the handling of stopping the monitoring process correctly after one iteration or upon encountering errors.

Test Data

- **URL for Testing:** https://example.com
- **Date for Availability Check:** 2024-10-01
- **Monitoring Frequency:** Once per session for immediate testing purposes.

Test Scenarios, Expected Outcomes, and Actual Outcomes

```
Starting test: test_start_monitoring_availability_success

Monitoring Iteration: Checked availability: Selected or default date is available for booking.
Control Layer Expected: ['Checked availability: Selected or default date is available for booking.', 'Monitoring stopped successfully!']
Control Layer Received: ['Checked availability: Selected or default date is available for booking.', 'Monitoring stopped successfully!']
Unit Test Passed for control layer.

Finished test: test_start_monitoring_availability_success
----------------------------------------------------------
Starting test: test_start_monitoring_availability_failure_entity
Monitoring Iteration: Failed to check availability: Failed to check availability
Control Layer Expected: ['Failed to check availability: Failed to check availability', 'Monitoring stopped successfully!']
Control Layer Received: ['Failed to check availability: Failed to check availability', 'Monitoring stopped successfully!']
Unit Test Passed for entity layer error handling.

Finished test: test_start_monitoring_availability_failure_entity
----------------------------------------------------------
Starting test: test_start_monitoring_availability_failure_control
Control Layer Expected: Control Layer Exception: Control Layer Failed
Control Layer Received: Control Layer Exception: Control Layer Failed
Unit Test Passed for control layer failure.

Finished test: test_start_monitoring_availability_failure_control
----------------------------------------------------------
Starting test: test_start_monitoring_availability_already_running

Control Layer Expected: Already monitoring availability.
Control Layer Received: Already monitoring availability.
Unit Test Passed for control layer already running handling.

Finished test: test_start_monitoring_availability_already_running
```

Source Code

Source Code can be found at the end of this document and in GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Test Case 12: Stop Monitoring Availability

Description

This test case verifies the functionality of stopping the availability monitoring process within the Discord bot system. It ensures that the stop_monitoring_availability function can correctly handle scenarios where monitoring is active and should be stopped, as well as correctly respond when no monitoring session is active.

Steps

1. **Setup and Mock Initialization:**
    - The test initializes by setting up the necessary mocks and test environment using pytest fixtures.
    - The AvailabilityControl object, which is responsible for monitoring availability, is accessed, and its status attributes are manipulated to simulate different scenarios.
2. **Simulate Monitoring Scenarios:**
    - **Active Monitoring Scenario:** The test simulates an active monitoring session by setting the is_monitoring flag to True and pre-populating results to simulate previous monitoring outputs.
    - **No Active Monitoring Scenario:** The is_monitoring flag is set to False to simulate the scenario where there is no active monitoring session.
3. **Execute Stop Command:**
    - The stop_monitoring_availability method is executed on the AvailabilityControl object, which checks the is_monitoring status and either stops the monitoring or responds that there is nothing to stop.
4. **Assertions and Logging:**
    - The test asserts whether the output from the stop_monitoring_availability method matches the expected result based on the monitoring status.
    - Detailed logs record the expected outcome and the actual outcome from the method execution.

5. **Validate User Feedback:**
    - The test verifies that the user feedback or system response is appropriate for the given scenario, ensuring that the system communicates the correct status of the monitoring process to the user.

Test Data

- **Active Monitoring Data:** Simulated by setting is_monitoring to True and populating the results list with simulated monitoring data.
- **No Active Monitoring Data:** Simulated by setting is_monitoring to False.

Test Scenarios, Expected Outcomes, Actual Outcomes

```
Starting test: test_stop_monitoring_availability_success

Control Layer Expected to contain: Monitoring stopped successfully!
Control Layer Received: Results for availability monitoring:
Checked availability: Selected or default date is available for booking.

Monitoring stopped successfully!
Unit Test Passed for stop monitoring availability.

Finished test: test_stop_monitoring_availability_success
----------------------------------------------------------
Starting test: test_stop_monitoring_availability_no_active_session

Control Layer Expected: There was no active availability monitoring session. Nothing to stop.
Control Layer Received: There was no active availability monitoring session. Nothing to stop.
Unit Test Passed for stop monitoring with no active session.

Finished test: test_stop_monitoring_availability_no_active_session
----------------------------------------------------------
```

Source Code

Source Code can be found at the end of this document and in the GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Test Case 13: Start Monitoring Price

Description

This test case evaluates the functionality of the start_monitoring_price command within the Discord bot. It checks the bot's ability to initiate price monitoring on a given product URL. The test ensures the bot can handle multiple scenarios, including successful monitoring, already running monitoring, and failure cases due to entity or control layer errors.

Steps

1. **Setup and Mock Initialization**:
    - Initialize the necessary mocks and the testing environment using pytest fixtures.
    - Access the PriceControl object and prepare its methods for mocking, particularly focusing on the get_price_from_page method.
2. **Entity Layer Interaction**:
    - Mock the PriceEntity.get_price_from_page method to simulate fetching the current price of a product.
    - Set expected outcomes for successful price retrieval and various failure scenarios such as errors in fetching the price.
3. **Control Layer Execution**:
    - Execute the start_monitoring_price method on the PriceControl object with the mocked price retrieval, simulating different monitoring scenarios.
    - Use the asyncio.sleep method mock to exit monitoring loops after the first iteration to test the command effectively in a test environment.
4. **Assertions and Logging**:
    - For each scenario, verify that the control and entity layers provide outputs matching the expected results, handling both success and error states appropriately.
    - Log detailed information about the process and the results, ensuring clarity and traceability of the test actions.
5. **Simulated User Feedback**:
    - Depending on the outcome of the command execution, simulate the appropriate user feedback, verifying that the system provides correct status updates and error messages.

Test Data

- **Valid URL Data**:
    - URL: "https://example.com/product"
    - Expected Price: "100 USD"
- **Error Scenario Data**:
    - Simulate a fetch error using the Exception to trigger error handling paths.

Test Scenarios, Expected Outcomes, Actual Outcomes

```
Starting test: test_start_monitoring_price_success

Entity Layer Expected: Starting price monitoring. Current price: 100 USD
Control Layer Received: Starting price monitoring. Current price: 100 USD
Unit Test Passed for start_monitoring_price success scenario.

Finished test: test_start_monitoring_price_success
-----------------------------------------------------
Starting test: test_start_monitoring_price_already_running

Control Layer Expected: Already monitoring prices.
Control Layer Received: Already monitoring prices.
Unit Test Passed for already running scenario.

Finished test: test_start_monitoring_price_already_running
-----------------------------------------------------
Starting test: test_start_monitoring_price_failure_in_entity

Control Layer Expected: Starting price monitoring. Current price: Failed to fetch price: Error fetching price
Control Layer Received: Starting price monitoring. Current price: Failed to fetch price: Error fetching price
Unit Test Passed for entity layer failure scenario.

Finished test: test_start_monitoring_price_failure_in_entity
-----------------------------------------------------
Starting test: test_start_monitoring_price_failure_in_control

Control Layer Expected: Control Layer Exception
Control Layer Received: Control Layer Exception: Control Layer Exception
Unit Test Passed for control layer failure scenario.

Finished test: test_start_monitoring_price_failure_in_control
-----------------------------------------------------
```

Source Code

Source Code can be found at the end of this document and in GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Test Case 14: Stop Monitoring Price

Description

This test case verifies the functionality of stopping the price monitoring process within the Discord bot system. It tests the ability of the system to handle both active and inactive monitoring sessions, ensuring that the price monitoring can be halted correctly and that appropriate feedback is provided to the user.

Steps

1. **Setup and Mock Initialization:**
    - The test environment is prepared with necessary mocks and the pytest fixtures are set up.
    - The PriceControl object is accessed, its monitoring state and results are manipulated to simulate different scenarios.
2. **Control Layer Execution:**
    - The stop_monitoring_price method on the PriceControl object is called to stop the monitoring of price changes.
    - The test handles different scenarios: when monitoring is active and when no monitoring session is active.
3. **Assertions and Logging:**
    - The outcomes from the stop_monitoring_price command are captured and compared against the expected results.
    - Logs are recorded for both the expected outcomes and the actual results received from the control layer to ensure traceability and clarity in the test execution.
4. **Error Simulation:**
    - In addition to regular scenarios, an error scenario is tested to simulate a failure in the control layer during the stopping process, ensuring the system's robustness and error handling capabilities.

Test Data

- **Active Monitoring Session:**
    - Monitoring is set to active with results stored as "Price went up!" and "Price went down!".
- **Inactive Monitoring Session:**
    - Monitoring is set to inactive with no results stored.

Test Scenarios, Expected Outcomes, Actual Outcomes

```
Starting test: test_stop_monitoring_price_success

Control Layer Expected: Results for price monitoring:
Price went up!
Price went down!

Price monitoring stopped successfully!
Control Layer Received: Results for price monitoring:
Price went up!
Price went down!

Price monitoring stopped successfully!
Unit Test Passed for stop_monitoring_price success scenario.

Finished test: test_stop_monitoring_price_success
----------------------------------------------------------
Starting test: test_stop_monitoring_price_not_active

Control Layer Expected: There was no active price monitoring session. Nothing to stop.
Control Layer Received: There was no active price monitoring session. Nothing to stop.
Unit Test Passed for stop_monitoring_price when not active.

Finished test: test_stop_monitoring_price_not_active
----------------------------------------------------------
Starting test: test_stop_monitoring_price_failure_in_control

Control Layer Expected: Error stopping price monitoring
Control Layer Received: Error stopping price monitoring
Unit Test Passed for stop_monitoring_price failure scenario.

Finished test: test_stop_monitoring_price_failure_in_control
----------------------------------------------------------
```

Source Code

Source Code can be found at the end of this document and in GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Test Case 15: Stop Bot

Description

This test case verifies the functionality of the "stop_bot" command within the Discord bot system. It is designed to ensure that the bot can be properly shut down upon command and handles failure scenarios when the control layer encounters an error. The tests check the control layer's ability to receive and process the "stop_bot" command, confirming that both successful execution and error handling are managed correctly.

Steps

1. **Setup and Mock Initialization**:
    - Initialize the test environment using pytest fixtures and set up logging for detailed feedback.
    - Access the BotControl object and prepare it for mocking to simulate the reception and processing of the "stop_bot" command.
2. **Mock Command Execution and Response Handling**:
    - Mock the receive_command method of the BotControl object to simulate stopping the bot.
    - Define expected outcomes for successful shutdown and for a simulated failure in the control layer.
3. **Execution and Assertion**:
    - Execute the "stop_bot" command through the mocked control object.
    - Verify that the bot responds as expected under normal conditions and during simulated control layer failures.
4. **Logging and Results Validation**:
    - Log the expected and actual results for both the successful execution and the failure scenario.
    - Use assertions to ensure the test outcomes match the expected results, confirming the bot's behavior is as intended.

Test Data

- No specific input data is required as the test simulates command reception and processing internally.

Test Scenarios, Expected Outcomes, and Actual Outcomes

Starting test: test_stop_bot_success

Control Layer Expected: Bot has been shut down.
Control Layer Received: Bot has been shut down.
Unit Test Passed for control layer stop bot.

Finished test: test_stop_bot_success

Starting test: test_stop_bot_failure_control

Control Layer Expected: Control Layer Exception: Control Layer Failed
Control Layer Received: Control Layer Exception: Control Layer Failed
Unit Test Passed for control layer failure.

Finished test: test_stop_bot_failure_control

Source Code

Source Code can be found at the end of this document and in GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Test Case 16: Project Help

Description

This test case verifies the functionality of the project_help command within the Discord bot system. It tests the bot's ability to provide a list of available commands to the user, ensuring the correct information is relayed and error handling is effectively implemented. The tests confirm that the bot responds accurately under both normal and error conditions, thereby maintaining reliable user interaction.

Steps

1. **Setup and Mock Initialization:**
    - The testing environment is prepared with necessary mocks and configurations using pytest fixtures.
    - The BotControl object, which handles command receptions and processing, is targeted for testing with method mock setups.
2. **Command Execution and Mock Interaction:**
    - The project_help command is simulated to invoke the bot's functionality for fetching and displaying available commands.
    - The method BotControl.receive_command is mocked to return a predetermined list of commands, simulating successful retrieval and an error scenario to validate error handling.
3. **Assertions and Result Validation:**
    - The test asserts whether the returned value from the command matches the expected list of commands.
    - Each test logs detailed information about the expected outcomes and actual results, ensuring that the function's integrity is maintained across updates.
4. **Error Simulation and Handling Test:**
    - The error scenario simulates a failure in command processing to ensure that the bot correctly handles and reports errors.

Test Data

- **Command Inputs:** No direct user inputs are required; the test internally triggers the project_help command.

Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test_project_help_success
Control Layer Expected: Here are the available commands:
!project_help - Get help on available commands.
!fetch_all_accounts - Fetch all stored accounts.
!add_account 'username' 'password' 'website' - Add a new account to the database.
!fetch_account_by_website 'website' - Fetch account details by website.
!delete_account 'account_id' - Delete an account by its ID.
!launch_browser - Launch the browser.
!close_browser - Close the browser.
!navigate_to_website 'url' - Navigate to a specified website.
!login 'website' - Log in to a website (e.g., !login bestbuy).
!get_price 'url' - Check the price of a product on a specified website.
!start_monitoring_price 'url' 'frequency' - Start monitoring a product's price at a specific interval (frequency in minutes).
!stop_monitoring_price - Stop monitoring the product's price.
!check_availability 'url' - Check availability for a restaurant or service.
!start_monitoring_availability 'url' 'frequency' - Monitor availability at a specific interval.
!stop_monitoring_availability - Stop monitoring availability.
!stop_bot - Stop the bot.

Control Layer Received: Here are the available commands:
!project_help - Get help on available commands.
!fetch_all_accounts - Fetch all stored accounts.
!add_account 'username' 'password' 'website' - Add a new account to the database.
!fetch_account_by_website 'website' - Fetch account details by website.
!delete_account 'account_id' - Delete an account by its ID.
!launch_browser - Launch the browser.
!close_browser - Close the browser.
!navigate_to_website 'url' - Navigate to a specified website.
!login 'website' - Log in to a website (e.g., !login bestbuy).
!get_price 'url' - Check the price of a product on a specified website.
!start_monitoring_price 'url' 'frequency' - Start monitoring a product's price at a specific interval (frequency in minutes).
!stop_monitoring_price - Stop monitoring the product's price.
!check_availability 'url' - Check availability for a restaurant or service.
!start_monitoring_availability 'url' 'frequency' - Monitor availability at a specific interval.
!stop_monitoring_availability - Stop monitoring availability.
!stop_bot - Stop the bot.

Unit Test Passed for project help.

Finished test: test_project_help_success

Starting test: test_project_help_failure

Control Layer Expected: Error handling help command: Error handling help command
Control Layer Received: Error handling help command: Error handling help command
Unit Test Passed for error handling in project help.

Finished test: test_project_help_failure

Source Code

Source Code can be found at the end of this document and in GitHub repository for a more organized overview: https://github.com/oguzky7/DiscordBotProject_CISC699

Conclusion

The test plan detailed in this document provides a comprehensive and structured approach to ensuring the robustness and reliability of the Discord Bot Automation Assistant. Through methodical testing of each component—covering Entity, Control, and Use Case scenarios—the plan ensures that all functionalities are verified against predefined expectations and real-world use conditions.

The inclusion of advanced testing techniques using pytest and asyncio further enhances the capability to simulate and evaluate asynchronous operations which are critical to the bot's performance. This document not only serves as a testament to the thorough testing processes implemented but also acts as a valuable guide for maintaining and scaling the system. With clear documentation of test data, scenarios, and outcomes, the test plan ensures transparency and repeatability, essential for ongoing development and maintenance of the system. The linkage of source code via GitHub integrates well with modern development practices, providing ease of access and review, thus maintaining a high standard of quality assurance for the project.

Codes

```
--- test_init.py ---
import sys, os, logging, pytest, asyncio
import subprocess
from unittest.mock import patch, MagicMock
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

#pytest -v > test_results.txt
#Run this command in the terminal to save the test results to a file

async def run_monitoring_loop(control_object, check_function, url, date_str, frequency, iterations=1):
    """Run the monitoring loop for a control object and execute a check function."""
    control_object.is_monitoring = True
    results = []

    while control_object.is_monitoring and iterations > 0:
        try:
            result = await check_function(url, date_str)
        except Exception as e:
            result = f"Failed to monitor: {str(e)}"
        logging.info(f"Monitoring Iteration: {result}")
        results.append(result)
        iterations -= 1
        await asyncio.sleep(frequency)

    control_object.is_monitoring = False
    results.append("Monitoring stopped successfully!")

    return results

def setup_logging():
    """Set up logging without timestamp and other unnecessary information."""
    logger = logging.getLogger()
    if not logger.hasHandlers():
        logging.basicConfig(level=logging.INFO, format='%(message)s')
def save_test_results_to_file(output_file="test_results.txt"):
    """Helper function to run pytest and save results to a file."""
    print("Running tests and saving results to file...")
    output_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), output_file)
    with open(output_path, 'w') as f:
        # Use subprocess to call pytest and redirect output to file
        subprocess.run(['pytest', '-v'], stdout=f, stderr=subprocess.STDOUT)

# Custom fixture for logging test start and end
@pytest.fixture(autouse=True)
def log_test_start_end(request):
    test_name = request.node.name
    logging.info(f"------------------------------------------------------\nStarting test: {test_name}\n")
```

```
    # Yield control to the test function
    yield

    # Log after the test finishes
    logging.info(f"\nFinished test: {test_name}\n----------------------------------------------------------")

# Import your control classes
from control.BrowserControl import BrowserControl
from control.AccountControl import AccountControl
from control.AvailabilityControl import AvailabilityControl
from control.PriceControl import PriceControl
from control.BotControl import BotControl

@pytest.fixture
def base_test_case():
    """Base test setup that can be used by all test functions."""
    test_case = MagicMock()
    test_case.browser_control = BrowserControl()
    test_case.account_control = AccountControl()
    test_case.availability_control = AvailabilityControl()
    test_case.price_control = PriceControl()
    test_case.bot_control = BotControl()
    return test_case

if __name__ == "__main__":
    # Save the pytest output to a file in the same folder
    save_test_results_to_file(output_file="test_results.txt")
```

```
--- unitTest_add_account.py ---
import pytest, logging
from unittest.mock import patch
from test_init import base_test_case, setup_logging, log_test_start_end, save_test_results_to_file

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()

async def test_add_account_success(base_test_case):
    with patch('control.AccountControl.AccountControl.add_account', return_value="Account for example.com
added successfully.") as mock_add_account:
        # Setup expected outcomes
        username = "test_user"
        password = "test_pass"
        website = "example.com"
        expected_entity_result = "Account for example.com added successfully."
        expected_control_result = "Account for example.com added successfully."

        # Execute the command
        result = base_test_case.account_control.add_account(username, password, website)

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_add_account.return_value}")
        assert mock_add_account.return_value == expected_entity_result, "Entity layer assertion failed."
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer.")

async def test_add_account_failure_invalid_data(base_test_case):
    with patch('control.AccountControl.AccountControl.add_account', return_value="Failed to add account for
example.com.") as mock_add_account:
        # Setup expected outcomes for invalid data scenario
        username = ""  # Invalid username
        password = ""  # Invalid password
        website = "example.com"
        expected_control_result = "Failed to add account for example.com."

        # Execute the command
        result = base_test_case.account_control.add_account(username, password, website)

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer invalid data handling.\n")
```

```
async def test_add_account_failure_entity_error(base_test_case):
    with patch('control.AccountControl.AccountControl.add_account', side_effect=Exception("Database Error"))
as mock_add_account:
        # Setup expected outcomes
        username = "test_user"
        password = "test_pass"
        website = "example.com"
        expected_control_result = "Control Layer Exception: Database Error"

        # Execute the command
        try:
            result = base_test_case.account_control.add_account(username, password, website)
        except Exception as e:
            result = f"Control Layer Exception: {str(e)}"

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer failed to handle entity error correctly."
        logging.info("Unit Test Passed for control layer error handling.")

async def test_add_account_already_exists(base_test_case):
    # This simulates a scenario where an account for the website already exists
    with patch('control.AccountControl.AccountControl.add_account', return_value="Failed to add account for
example.com. Account already exists.") as mock_add_account:
        # Setup expected outcomes
        username = "test_user"
        password = "test_pass"
        website = "example.com"
        expected_control_result = "Failed to add account for example.com. Account already exists."

        # Execute the command
        result = base_test_case.account_control.add_account(username, password, website)

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer when account already exists.")

if __name__ == "__main__":
    pytest.main([__file__])
```

```
--- unitTest_check_availability.py ---
import pytest, logging
from unittest.mock import patch
from test_init import base_test_case, setup_logging, log_test_start_end

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()

# Test for successful availability check (Control and Entity Layers)
async def test_check_availability_success(base_test_case):
    with patch('entity.AvailabilityEntity.AvailabilityEntity.check_availability') as mock_check:
        url = "https://example.com"
        mock_check.return_value = f"Selected or default date current date is available for booking."
        expected_entity_result = f"Selected or default date current date is available for booking."
        expected_control_result = f"Checked availability: Selected or default date current date is available for
booking."

        # Execute the command
        result = await base_test_case.availability_control.receive_command("check_availability", url)

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_check.return_value}")
        assert mock_check.return_value == expected_entity_result, "Entity layer assertion failed."
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer.")

# Test for failure in entity layer (Control should handle it gracefully)
async def test_check_availability_failure_entity(base_test_case):
    with patch('entity.AvailabilityEntity.AvailabilityEntity.check_availability', side_effect=Exception("Failed to
check availability")) as mock_check:
        url = "https://example.com"
        expected_control_result = "Failed to check availability: Failed to check availability"

        # Execute the command
        result = await base_test_case.availability_control.receive_command("check_availability", url)

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer failed to handle entity error correctly."
        logging.info("Unit Test Passed for entity layer error handling.")

# Test for no availability scenario (control and entity)
async def test_check_availability_no_availability(base_test_case):
```

```
    with patch('entity.AvailabilityEntity.AvailabilityEntity.check_availability') as mock_check:
        url = "https://example.com"
        mock_check.return_value = "No availability for the selected date."
        expected_control_result = "Checked availability: No availability for the selected date."

        # Execute the command
        result = await base_test_case.availability_control.receive_command("check_availability", url)

        # Log and assert the outcomes
        logging.info(f"Entity Layer Received: {mock_check.return_value}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer failed to handle no availability scenario."
        logging.info("Unit Test Passed for control layer no availability handling.")

# Test for control layer failure scenario
async def test_check_availability_failure_control(base_test_case):
    with patch('control.AvailabilityControl.AvailabilityControl.receive_command', side_effect=Exception("Control
Layer Failed")) as mock_control:
        url = "https://example.com"
        expected_control_result = "Control Layer Exception: Control Layer Failed"

        # Execute the command and catch the raised exception
        try:
            result = await base_test_case.availability_control.receive_command("check_availability", url)
        except Exception as e:
            result = f"Control Layer Exception: {str(e)}"

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer failure.")

if __name__ == "__main__":
    pytest.main([__file__])
```

```
--- unitTest_close_browser.py ---
import pytest, logging
from unittest.mock import patch
from test_init import base_test_case, setup_logging, log_test_start_end

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()

async def test_close_browser_success(base_test_case):
    with patch('entity.BrowserEntity.BrowserEntity.close_browser') as mock_close:
        # Set up mock and expected outcomes
        mock_close.return_value = "Browser closed."
        expected_entity_result = "Browser closed."
        expected_control_result = "Control Object Result: Browser closed."

        # Execute the command
        result = await base_test_case.browser_control.receive_command("close_browser")

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_close.return_value}")
        assert mock_close.return_value == expected_entity_result, "Entity layer assertion failed."
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer.")

async def test_close_browser_not_open(base_test_case):
    with patch('entity.BrowserEntity.BrowserEntity.close_browser') as mock_close:
        # Set up mock and expected outcomes
        mock_close.return_value = "No browser is currently open."
        expected_entity_result = "No browser is currently open."
        expected_control_result = "Control Object Result: No browser is currently open."

        # Execute the command
        result = await base_test_case.browser_control.receive_command("close_browser")

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_close.return_value}")
        assert mock_close.return_value == expected_entity_result, "Entity layer assertion failed."
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer.")
```

```
async def test_close_browser_failure_control(base_test_case):
    with patch('entity.BrowserEntity.BrowserEntity.close_browser', side_effect=Exception("Unexpected error"))
as mock_close:
        # Set up expected outcome
        expected_result = "Control Layer Exception: Unexpected error"

        # Execute the command
        result = await base_test_case.browser_control.receive_command("close_browser")

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected to Report: {expected_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_result, "Control layer failed to handle or report the error correctly."
        logging.info("Unit Test Passed for control layer error handling.")

async def test_close_browser_failure_entity(base_test_case):
    with patch('entity.BrowserEntity.BrowserEntity.close_browser', side_effect=Exception("BrowserEntity_Failed
to close browser: Internal error")) as mock_close:
        # Set up expected outcome
        internal_error_message = "BrowserEntity_Failed to close browser: Internal error"
        expected_control_result = f"Control Layer Exception: {internal_error_message}"

        # Execute the command
        result = await base_test_case.browser_control.receive_command("close_browser")

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected Failure: {internal_error_message}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer failed to report entity error correctly."
        logging.info("Unit Test Passed for entity layer error handling.")

if __name__ == "__main__":
    pytest.main([__file__])
```

```
--- unitTest_delete_account.py ---
import pytest, logging
from unittest.mock import patch
from test_init import base_test_case, setup_logging, log_test_start_end

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()

async def test_delete_account_success(base_test_case):
    with patch('DataObjects.AccountDAO.AccountDAO.delete_account') as mock_delete:
        # Setup mock return and expected outcomes
        account_id = 1
        mock_delete.return_value = True
        expected_entity_result = "Account with ID 1 deleted successfully."
        expected_control_result = "Account with ID 1 deleted successfully."

        # Execute the command
        result = base_test_case.account_control.delete_account(account_id)

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_delete.return_value}")
        assert mock_delete.return_value == True, "Entity layer assertion failed."
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer.")

async def test_delete_account_not_found(base_test_case):
    with patch('DataObjects.AccountDAO.AccountDAO.delete_account') as mock_delete:
        # Setup mock return and expected outcomes
        account_id = 999
        mock_delete.return_value = False
        expected_control_result = "Failed to delete account with ID 999."

        # Execute the command
        result = base_test_case.account_control.delete_account(account_id)

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer with account not found.\n")

async def test_delete_account_failure_entity(base_test_case):
    with patch('DataObjects.AccountDAO.AccountDAO.delete_account', side_effect=Exception("Failed to delete
account in DAO")) as mock_delete:
```

```
    # Setup expected outcomes
    account_id = 1
    expected_control_result = "Error deleting account."

    # Execute the command
    result = base_test_case.account_control.delete_account(account_id)

    # Log and assert the outcomes
    logging.info(f"Control Layer Expected: {expected_control_result}")
    logging.info(f"Control Layer Received: {result}")
    assert result == expected_control_result, "Control layer failed to handle entity error correctly."
    logging.info("Unit Test Passed for entity layer error handling.")

async def test_delete_account_failure_control(base_test_case):
  # This simulates a failure within the control layer
  with patch('control.AccountControl.AccountControl.delete_account', side_effect=Exception("Control Layer
Failed")) as mock_control:

    # Setup expected outcomes
    account_id = 1
    expected_control_result = "Control Layer Exception: Control Layer Failed"

    # Execute the command and catch the raised exception
    try:
      result = base_test_case.account_control.delete_account(account_id)
    except Exception as e:
      result = f"Control Layer Exception: {str(e)}"

    # Log and assert the outcomes
    logging.info(f"Control Layer Expected: {expected_control_result}")
    logging.info(f"Control Layer Received: {result}")
    assert result == expected_control_result, "Control layer assertion failed."
    logging.info("Unit Test Passed for control layer failure.")

if __name__ == "__main__":
  pytest.main([__file__])
```

```
--- unitTest_fetch_account_by_website.py ---
import pytest
import logging
from unittest.mock import patch
from test_init import base_test_case, setup_logging, log_test_start_end

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()

async def test_fetch_account_by_website_success(base_test_case):
    with patch('DataObjects.AccountDAO.AccountDAO.fetch_account_by_website') as mock_fetch:
        # Setup mock return and expected outcomes
        website = "example.com"
        mock_fetch.return_value = ("sample_username", "sample_password")
        expected_entity_result = ("sample_username", "sample_password")
        expected_control_result = ("sample_username", "sample_password")

        # Execute the command
        result = base_test_case.account_control.fetch_account_by_website(website)

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_fetch.return_value}")
        assert mock_fetch.return_value == expected_entity_result, "Entity layer assertion failed."
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer.")


async def test_fetch_account_by_website_no_account(base_test_case):
    with patch('DataObjects.AccountDAO.AccountDAO.fetch_account_by_website') as mock_fetch:
        # Setup mock return and expected outcomes
        website = "nonexistent.com"
        mock_fetch.return_value = None
        expected_control_result = "No account found for nonexistent.com."

        # Execute the command
        result = base_test_case.account_control.fetch_account_by_website(website)

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer no account found.\n")
```

```
async def test_fetch_account_by_website_failure_entity(base_test_case):
    with patch('DataObjects.AccountDAO.AccountDAO.fetch_account_by_website',
side_effect=Exception("Database Error")) as mock_fetch:
        # Setup expected outcomes
        website = "example.com"
        expected_control_result = "Error: Database Error"

        # Execute the command
        result = base_test_case.account_control.fetch_account_by_website(website)

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer failed to handle entity error correctly."
        logging.info("Unit Test Passed for entity layer error handling.")


async def test_fetch_account_by_website_failure_control(base_test_case):
    with patch('control.AccountControl.AccountControl.fetch_account_by_website',
side_effect=Exception("Control Layer Error")) as mock_control:
        # Setup expected outcomes
        website = "example.com"
        expected_control_result = "Control Layer Exception: Control Layer Error"

        # Execute the command and catch the raised exception
        try:
            result = base_test_case.account_control.fetch_account_by_website(website)
        except Exception as e:
            result = f"Control Layer Exception: {str(e)}"

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer failed to handle its own error correctly."
        logging.info("Unit Test Passed for control layer error handling.")


if __name__ == "__main__":
    pytest.main([__file__])
```

```
--- unitTest_fetch_all_accounts.py ---
import pytest, logging
from unittest.mock import patch
from test_init import base_test_case, setup_logging, log_test_start_end

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()

async def test_fetch_all_accounts_success(base_test_case):
    with patch('DataObjects.AccountDAO.AccountDAO.fetch_all_accounts') as mock_fetch_all:
        # Setup mock return and expected outcomes
        mock_fetch_all.return_value = [(1, "user1", "pass1", "example.com"), (2, "user2", "pass2", "test.com")]
        expected_entity_result = "Accounts:\nID: 1, Username: user1, Password: pass1, Website: example.com\nID:
2, Username: user2, Password: pass2, Website: test.com"
        expected_control_result = expected_entity_result

        # Execute the command
        result = base_test_case.account_control.receive_command("fetch_all_accounts")

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_fetch_all.return_value}")
        assert mock_fetch_all.return_value == [(1, "user1", "pass1", "example.com"), (2, "user2", "pass2",
"test.com")], "Entity layer assertion failed."
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer.")

async def test_fetch_all_accounts_no_accounts(base_test_case):
    with patch('DataObjects.AccountDAO.AccountDAO.fetch_all_accounts') as mock_fetch_all:
        # Setup mock return and expected outcomes
        mock_fetch_all.return_value = []
        expected_control_result = "No accounts found."

        # Execute the command
        result = base_test_case.account_control.receive_command("fetch_all_accounts")

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer no accounts found.\n")

async def test_fetch_all_accounts_failure_entity(base_test_case):
    with patch('DataObjects.AccountDAO.AccountDAO.fetch_all_accounts', side_effect=Exception("Database
Error")) as mock_fetch_all:
```

```
        # Setup expected outcomes
        expected_control_result = "Error fetching accounts."

        # Execute the command
        result = base_test_case.account_control.receive_command("fetch_all_accounts")

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer failed to handle entity error correctly."
        logging.info("Unit Test Passed for entity layer error handling.")

if __name__ == "__main__":
    pytest.main([__file__])
```

```
--- unitTest_get_price.py ---
import pytest, logging
from unittest.mock import patch
from test_init import base_test_case, setup_logging, log_test_start_end

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()

async def test_get_price_success(base_test_case):
    # Simulate a successful price retrieval
    with patch('entity.PriceEntity.PriceEntity.get_price_from_page') as mock_get_price:
        url = "https://example.com/product"
        mock_get_price.return_value = "$199.99"
        expected_entity_result = "$199.99"
        expected_control_result = "$199.99"

        # Execute the command
        result = await base_test_case.price_control.receive_command("get_price", url)

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_get_price.return_value}")
        assert mock_get_price.return_value == expected_entity_result, "Entity layer assertion failed."
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer.")

async def test_get_price_invalid_url(base_test_case):
    # Simulate an invalid URL case
    with patch('entity.PriceEntity.PriceEntity.get_price_from_page') as mock_get_price:
        invalid_url = "invalid_url"
        mock_get_price.return_value = "Error fetching price: Invalid URL"
        expected_control_result = "Error fetching price: Invalid URL"

        # Execute the command
        result = await base_test_case.price_control.receive_command("get_price", invalid_url)

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer invalid URL handling.\n")

async def test_get_price_failure_entity(base_test_case):
    # Simulate an entity layer failure when fetching the price
    with patch('entity.PriceEntity.PriceEntity.get_price_from_page', side_effect=Exception("Failed to fetch
```

```
price")) as mock_get_price:
        url = "https://example.com/product"
        expected_control_result = "Failed to fetch price: Failed to fetch price"

        # Execute the command
        result = await base_test_case.price_control.receive_command("get_price", url)

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer failed to handle entity error correctly."
        logging.info("Unit Test Passed for entity layer error handling.")

async def test_get_price_failure_control(base_test_case):
    # Simulate a control layer failure
    with patch('control.PriceControl.PriceControl.receive_command', side_effect=Exception("Control Layer
Failed")) as mock_control:
        url = "https://example.com/product"
        expected_control_result = "Control Layer Exception: Control Layer Failed"

        # Execute the command and catch the raised exception
        try:
            result = await base_test_case.price_control.receive_command("get_price", url)
        except Exception as e:
            result = f"Control Layer Exception: {str(e)}"

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer failure.")

if __name__ == "__main__":
    pytest.main([__file__])
```

```
--- unitTest_launch_browser.py ---
import pytest, logging
from unittest.mock import patch
from test_init import base_test_case, log_test_start_end, setup_logging

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()

async def test_launch_browser_success(base_test_case):
    with patch('entity.BrowserEntity.BrowserEntity.launch_browser') as mock_launch:
        # Setup mock return and expected outcomes
        mock_launch.return_value = "Browser launched."
        expected_entity_result = "Browser launched."
        expected_control_result = "Control Object Result: Browser launched."

        # Execute the command
        result = await base_test_case.browser_control.receive_command("launch_browser")

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_launch.return_value}")
        assert mock_launch.return_value == expected_entity_result, "Entity layer assertion failed."
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer.")

async def test_launch_browser_already_running(base_test_case):
    with patch('entity.BrowserEntity.BrowserEntity.launch_browser', return_value="Browser is already running.")
as mock_launch:
        expected_entity_result = "Browser is already running."
        expected_control_result = "Control Object Result: Browser is already running."

        result = await base_test_case.browser_control.receive_command("launch_browser")

        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_launch.return_value}")
        assert mock_launch.return_value == expected_entity_result, "Entity layer assertion failed."
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer.")

async def test_launch_browser_failure_control(base_test_case):
    with patch('entity.BrowserEntity.BrowserEntity.launch_browser', side_effect=Exception("Internal error")) as
```

```
mock_launch:
        expected_result = "Control Layer Exception: Internal error"

        result = await base_test_case.browser_control.receive_command("launch_browser")

        logging.info(f"Control Layer Expected to Report: {expected_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_result, "Control layer failed to handle or report the entity error correctly."
        logging.info("Unit Test Passed for control layer error handling.")


async def test_launch_browser_failure_entity(base_test_case):
    with patch('entity.BrowserEntity.BrowserEntity.launch_browser', side_effect=Exception("Failed to launch
browser: Internal error")) as mock_launch:
        expected_control_result = "Control Layer Exception: Failed to launch browser: Internal error"

        result = await base_test_case.browser_control.receive_command("launch_browser")

        logging.info(f"Entity Layer Expected Failure: Failed to launch browser: Internal error")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer failed to report entity error correctly."
        logging.info("Unit Test Passed for entity layer error handling.")

if __name__ == "__main__":
    pytest.main([__file__])
```

```
--- unitTest_login.py ---
import pytest
import logging
from unittest.mock import patch, MagicMock
from test_init import base_test_case, setup_logging, log_test_start_end

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio

setup_logging()

async def test_login_success(base_test_case):
    """Test that the login is successful when valid credentials are provided."""
    # Patch methods
    with patch('entity.BrowserEntity.BrowserEntity.login') as mock_login:
        with patch('control.AccountControl.AccountControl.fetch_account_by_website') as mock_fetch_account:
            # Setup mock return values
            mock_login.return_value = "Logged in to http://example.com successfully with username:
sample_username"
            mock_fetch_account.return_value = ("sample_username", "sample_password")

            expected_entity_result = "Logged in to http://example.com successfully with username:
sample_username"
            expected_control_result = f"Control Object Result: {expected_entity_result}"

            # Execute the command
            result = await base_test_case.browser_control.receive_command("login", site="example.com")

            # Assert results and logging
            logging.info(f"Entity Layer Expected: {expected_entity_result}")
            logging.info(f"Entity Layer Received: {mock_login.return_value}")
            assert mock_login.return_value == expected_entity_result, "Entity layer assertion failed."
            logging.info("Unit Test Passed for entity layer.\n")

            logging.info(f"Control Layer Expected: {expected_control_result}")
            logging.info(f"Control Layer Received: {result}")
            assert result == expected_control_result, "Control layer assertion failed."
            logging.info("Unit Test Passed for control layer.")

async def test_login_no_account(base_test_case):
    """Test that the control layer handles the scenario where no account is found for the website."""
    with patch('control.AccountControl.AccountControl.fetch_account_by_website') as mock_fetch_account:
        # Setup mock to return no account
        mock_fetch_account.return_value = None

        expected_result = "No account found for example.com"

        # Execute the command
        result = await base_test_case.browser_control.receive_command("login", site="example.com")
```

```
    # Assert results and logging
    logging.info(f"Control Layer Expected: {expected_result}")
    logging.info(f"Control Layer Received: {result}")
    assert result == expected_result, "Control layer failed to handle missing account correctly."
    logging.info("Unit Test Passed for missing account handling.")

async def test_login_entity_layer_failure(base_test_case):
  """Test that the control layer handles an exception raised in the entity layer."""
  with patch('entity.BrowserEntity.BrowserEntity.login') as mock_login:
    with patch('control.AccountControl.AccountControl.fetch_account_by_website') as mock_fetch_account:
      # Setup mocks
      mock_login.side_effect = Exception("BrowserEntity_Failed to log in to http://example.com: Internal
error")
      mock_fetch_account.return_value = ("sample_username", "sample_password")

      expected_result = "Control Layer Exception: BrowserEntity_Failed to log in to http://example.com:
Internal error"

      # Execute the command
      result = await base_test_case.browser_control.receive_command("login", site="example.com")

      # Assert results and logging
      logging.info(f"Control Layer Expected: {expected_result}")
      logging.info(f"Control Layer Received: {result}")
      assert result == expected_result, "Control layer failed to handle entity layer exception."
      logging.info("Unit Test Passed for entity layer failure.")


async def test_login_control_layer_failure(base_test_case):
  """Test that the control layer handles an unexpected failure or exception."""
  with patch('control.AccountControl.AccountControl.fetch_account_by_website') as mock_fetch_account:
    # Simulate an exception being raised in the control layer
    mock_fetch_account.side_effect = Exception("Control layer failure during account fetch.")

    expected_result = "Control Layer Exception: Control layer failure during account fetch."

    # Execute the command
    result = await base_test_case.browser_control.receive_command("login", site="example.com")

    # Assert results and logging
    logging.info(f"Control Layer Expected: {expected_result}")
    logging.info(f"Control Layer Received: {result}")
    assert result == expected_result, "Control layer failed to handle control layer exception."
    logging.info("Unit Test Passed for control layer failure handling.")

async def test_login_invalid_url(base_test_case):
  """Test that the control layer handles the scenario where the URL or selectors are not found."""
  with patch('control.AccountControl.AccountControl.fetch_account_by_website') as mock_fetch_account:
    with patch('utils.css_selectors.Selectors.get_selectors_for_url') as mock_get_selectors:
      # Setup mocks
```

```
        mock_fetch_account.return_value = ("sample_username", "sample_password")
        mock_get_selectors.return_value = {'url': None}  # Simulate missing URL

        expected_result = "URL for example not found."

        # Execute the command
        result = await base_test_case.browser_control.receive_command("login", site="example")

        # Assert results and logging
        logging.info(f"Control Layer Expected: {expected_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_result, "Control layer failed to handle missing URL or selectors."
        logging.info("Unit Test Passed for missing URL/selector handling.")

if __name__ == "__main__":
    pytest.main([__file__])
```

```
--- unitTest_navigate_to_website.py ---
import pytest, logging
from unittest.mock import patch
from test_init import base_test_case, setup_logging, log_test_start_end

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()


async def test_navigate_to_website_success(base_test_case):
    with patch('entity.BrowserEntity.BrowserEntity.navigate_to_website') as mock_navigate:
        # Setup mock return and expected outcomes
        url = "https://example.com"
        mock_navigate.return_value = f"Navigated to {url}"
        expected_entity_result = f"Navigated to {url}"
        expected_control_result = f"Control Object Result: Navigated to {url}"

        # Execute the command
        result = await base_test_case.browser_control.receive_command("navigate_to_website", site=url)

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_navigate.return_value}")
        assert mock_navigate.return_value == expected_entity_result, "Entity layer assertion failed."
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer.")


async def test_navigate_to_website_invalid_url(base_test_case):
    with patch('entity.BrowserEntity.BrowserEntity.navigate_to_website') as mock_navigate:
        # Setup mock return and expected outcomes
        invalid_site = "invalid_site"
        mock_navigate.return_value = f"URL for {invalid_site} not found."
        expected_control_result = f"URL for {invalid_site} not found."

        # Execute the command
        result = await base_test_case.browser_control.receive_command("navigate_to_website", site=invalid_site)

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer invalid URL handling.\n")
```

```
async def test_navigate_to_website_failure_entity(base_test_case):
    with patch('entity.BrowserEntity.BrowserEntity.navigate_to_website', side_effect=Exception("Failed to
navigate")) as mock_navigate:
        # Setup expected outcomes
        url = "https://example.com"
        expected_control_result = "Control Layer Exception: Failed to navigate"

        # Execute the command
        result = await base_test_case.browser_control.receive_command("navigate_to_website", site=url)

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer failed to handle entity error correctly."
        logging.info("Unit Test Passed for entity layer error handling.")


async def test_navigate_to_website_launch_browser_on_failure(base_test_case):
    # This test simulates a scenario where the browser is not open and needs to be launched first.
    with patch('entity.BrowserEntity.BrowserEntity.is_browser_open', return_value=False), \
         patch('entity.BrowserEntity.BrowserEntity.launch_browser', return_value="Browser launched."), \
         patch('entity.BrowserEntity.BrowserEntity.navigate_to_website') as mock_navigate:

        # Setup expected outcomes
        url = "https://example.com"
        mock_navigate.return_value = f"Navigated to {url}"
        expected_control_result = f"Control Object Result: Navigated to {url}"

        # Execute the command
        result = await base_test_case.browser_control.receive_command("navigate_to_website", site=url)

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer with browser launch.\n")


async def test_navigate_to_website_failure_control(base_test_case):
    # This simulates a failure within the control layer
    with patch('control.BrowserControl.BrowserControl.receive_command', side_effect=Exception("Control Layer
Failed")) as mock_control:

        # Setup expected outcomes
        url = "https://example.com"
        expected_control_result = "Control Layer Exception: Control Layer Failed"

        # Execute the command and catch the raised exception
        try:
            result = await base_test_case.browser_control.receive_command("navigate_to_website", site=url)
```

```
        except Exception as e:
            result = f"Control Layer Exception: {str(e)}"

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer failure.")

if __name__ == "__main__":
    pytest.main([__file__])
```

```
--- unitTest_project_help.py ---
import pytest, logging
from unittest.mock import patch
from test_init import base_test_case, setup_logging, log_test_start_end

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()

async def test_project_help_success(base_test_case):
    with patch('control.BotControl.BotControl.receive_command') as mock_help:
        # Setup mock return and expected outcomes
        mock_help.return_value = (
            "Here are the available commands:\n"
            "!project_help - Get help on available commands.\n"
            "!fetch_all_accounts - Fetch all stored accounts.\n"
            "!add_account 'username' 'password' 'website' - Add a new account to the database.\n"
            "!fetch_account_by_website 'website' - Fetch account details by website.\n"
            "!delete_account 'account_id' - Delete an account by its ID.\n"
            "!launch_browser - Launch the browser.\n"
            "!close_browser - Close the browser.\n"
            "!navigate_to_website 'url' - Navigate to a specified website.\n"
            "!login 'website' - Log in to a website (e.g., !login bestbuy).\n"
            "!get_price 'url' - Check the price of a product on a specified website.\n"
            "!start_monitoring_price 'url' 'frequency' - Start monitoring a product's price at a specific interval
(frequency in minutes).\n"
            "!stop_monitoring_price - Stop monitoring the product's price.\n"
            "!check_availability 'url' - Check availability for a restaurant or service.\n"
            "!start_monitoring_availability 'url' 'frequency' - Monitor availability at a specific interval.\n"
            "!stop_monitoring_availability - Stop monitoring availability.\n"
            "!stop_bot - Stop the bot.\n"
        )
        expected_result = mock_help.return_value

        # Execute the command
        result = await base_test_case.bot_control.receive_command("project_help")

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for project help.\n")


async def test_project_help_failure(base_test_case):
    with patch('control.BotControl.BotControl.receive_command', side_effect=Exception("Error handling help
command")) as mock_help:
        expected_result = "Error handling help command: Error handling help command"

        # Execute the command and catch the raised exception
```

```
        try:
            result = await base_test_case.bot_control.receive_command("project_help")
        except Exception as e:
            result = f"Error handling help command: {str(e)}"

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_result, "Control layer failed to handle error correctly."
        logging.info("Unit Test Passed for error handling in project help.\n")

if __name__ == "__main__":
    pytest.main([__file__])
```

```
--- unitTest_start_monitoring_availability.py ---
import pytest, logging
from unittest.mock import patch
from test_init import base_test_case, setup_logging, run_monitoring_loop, log_test_start_end
import asyncio

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()

async def test_start_monitoring_availability_success(base_test_case):
    with patch('entity.AvailabilityEntity.AvailabilityEntity.check_availability') as mock_check:
        url = "https://example.com"
        mock_check.return_value = "Selected or default date is available for booking."

        expected_control_result = [
            "Checked availability: Selected or default date is available for booking.",
            "Monitoring stopped successfully!"
        ]

        # Run the monitoring loop once
        actual_control_result = await run_monitoring_loop(
            base_test_case.availability_control,
            base_test_case.availability_control.check_availability,
            url,
            "2024-10-01",
            1
        )

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {actual_control_result}")
        assert actual_control_result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer.")


async def test_start_monitoring_availability_failure_entity(base_test_case):
    with patch('entity.AvailabilityEntity.AvailabilityEntity.check_availability', side_effect=Exception("Failed to
check availability")):
        url = "https://example.com"
        expected_control_result = [
            "Failed to check availability: Failed to check availability",
            "Monitoring stopped successfully!"
        ]

        # Run the monitoring loop once
        actual_control_result = await run_monitoring_loop(
            base_test_case.availability_control,
            base_test_case.availability_control.check_availability,
            url,
            "2024-10-01",
```

```
        1
    )

    logging.info(f"Control Layer Expected: {expected_control_result}")
    logging.info(f"Control Layer Received: {actual_control_result}")
    assert actual_control_result == expected_control_result, "Control layer failed to handle entity error
correctly."
    logging.info("Unit Test Passed for entity layer error handling.")


async def test_start_monitoring_availability_failure_control(base_test_case):
  with patch('control.AvailabilityControl.AvailabilityControl.receive_command', side_effect=Exception("Control
Layer Failed")):
    url = "https://example.com"
    expected_control_result = "Control Layer Exception: Control Layer Failed"

    try:
      result = await base_test_case.availability_control.receive_command("start_monitoring_availability", url,
"2024-10-01", 5)
    except Exception as e:
      result = f"Control Layer Exception: {str(e)}"

    logging.info(f"Control Layer Expected: {expected_control_result}")
    logging.info(f"Control Layer Received: {result}")
    assert result == expected_control_result, "Control layer assertion failed."
    logging.info("Unit Test Passed for control layer failure.")


async def test_start_monitoring_availability_already_running(base_test_case):
  with patch('entity.AvailabilityEntity.AvailabilityEntity.check_availability') as mock_check:
    url = "https://example.com"
    base_test_case.availability_control.is_monitoring = True
    expected_control_result = "Already monitoring availability."

    result = await base_test_case.availability_control.receive_command("start_monitoring_availability", url,
"2024-10-01", 5)

    logging.info(f"Control Layer Expected: {expected_control_result}")
    logging.info(f"Control Layer Received: {result}")
    assert result == expected_control_result, "Control layer failed to handle already running condition."
    logging.info("Unit Test Passed for control layer already running handling.\n")


if __name__ == "__main__":
  pytest.main([__file__])
```

```
--- unitTest_start_monitoring_price.py ---
import pytest
import logging
from unittest.mock import patch, AsyncMock
from test_init import base_test_case, setup_logging, log_test_start_end

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()


async def test_start_monitoring_price_success(base_test_case):
    with patch('entity.PriceEntity.PriceEntity.get_price_from_page', return_value="100 USD") as mock_get_price:

        # Setup expected outcomes
        url = "https://example.com/product"
        expected_result = "Starting price monitoring. Current price: 100 USD"

        # Mocking the sleep method to break out of the loop after the first iteration
        with patch('asyncio.sleep', side_effect=KeyboardInterrupt):
            try:
                # Execute the command
                base_test_case.price_control.is_monitoring = False
                result = await base_test_case.price_control.receive_command("start_monitoring_price", url, 1)
            except KeyboardInterrupt:
                # Force the loop to stop after the first iteration
                base_test_case.price_control.is_monitoring = False

        # Log and assert the outcomes
        logging.info(f"Entity Layer Expected: {expected_result}")
        logging.info(f"Control Layer Received: {base_test_case.price_control.results[0]}")
        assert expected_result in base_test_case.price_control.results[0], "Price monitoring did not start as
expected."
        logging.info("Unit Test Passed for start_monitoring_price success scenario.\n")


async def test_start_monitoring_price_already_running(base_test_case):
    # Test when price monitoring is already running
    base_test_case.price_control.is_monitoring = True
    expected_result = "Already monitoring prices."

    # Execute the command
    result = await base_test_case.price_control.receive_command("start_monitoring_price",
"https://example.com/product", 1)

    # Log and assert the outcomes
    logging.info(f"Control Layer Expected: {expected_result}")
    logging.info(f"Control Layer Received: {result}")
    assert result == expected_result, "Control layer did not detect that monitoring was already running."
    logging.info("Unit Test Passed for already running scenario.\n")
```

```
async def test_start_monitoring_price_failure_in_entity(base_test_case):
    # Mock entity failure during price fetching
    with patch('entity.PriceEntity.PriceEntity.get_price_from_page', side_effect=Exception("Error fetching price")) as mock_get_price:

        # Setup expected outcomes
        url = "https://example.com/product"
        expected_result = "Starting price monitoring. Current price: Failed to fetch price: Error fetching price"

        # Mocking the sleep method to break out of the loop after the first iteration
        with patch('asyncio.sleep', side_effect=KeyboardInterrupt):
            try:
                # Execute the command
                base_test_case.price_control.is_monitoring = False
                await base_test_case.price_control.receive_command("start_monitoring_price", url, 1)
            except KeyboardInterrupt:
                # Force the loop to stop after the first iteration
                base_test_case.price_control.is_monitoring = False

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_result}")
        logging.info(f"Control Layer Received: {base_test_case.price_control.results[-1]}")
        assert expected_result in base_test_case.price_control.results[-1], "Entity layer did not handle failure correctly."
        logging.info("Unit Test Passed for entity layer failure scenario.\n")


async def test_start_monitoring_price_failure_in_control(base_test_case):
    # Mock control layer failure
    with patch('control.PriceControl.PriceControl.start_monitoring_price', side_effect=Exception("Control Layer Exception")) as mock_start_monitoring:

        # Setup expected outcomes
        expected_result = "Control Layer Exception"

        # Execute the command and catch the raised exception
        try:
            result = await base_test_case.price_control.receive_command("start_monitoring_price", "https://example.com/product", 1)
        except Exception as e:
            result = f"Control Layer Exception: {str(e)}"

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_result}")
        logging.info(f"Control Layer Received: {result}")
        assert expected_result in result, "Control layer did not handle the failure correctly."
        logging.info("Unit Test Passed for control layer failure scenario.\n")
```

```
if __name__ == "__main__":
    pytest.main([__file__])
```

```
--- unitTest_stop_bot.py ---
import pytest
import logging
from unittest.mock import MagicMock, patch
from test_init import base_test_case, setup_logging, log_test_start_end

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()


async def test_stop_bot_success(base_test_case):
    with patch('control.BotControl.BotControl.receive_command') as mock_stop_bot:
        # Setup mock return and expected outcomes
        mock_stop_bot.return_value = "Bot has been shut down."
        expected_entity_result = "Bot has been shut down."
        expected_control_result = "Bot has been shut down."

        # Execute the command
        result = await base_test_case.bot_control.receive_command("stop_bot", ctx=MagicMock())

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer stop bot.\n")



async def test_stop_bot_failure_control(base_test_case):
    with patch('control.BotControl.BotControl.receive_command', side_effect=Exception("Control Layer Failed"))
as mock_control:
        # Setup expected outcomes
        expected_control_result = "Control Layer Exception: Control Layer Failed"

        # Execute the command and catch the raised exception
        try:
            result = await base_test_case.bot_control.receive_command("stop_bot", ctx=MagicMock())
        except Exception as e:
            result = f"Control Layer Exception: {str(e)}"

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        assert result == expected_control_result, "Control layer assertion failed."
        logging.info("Unit Test Passed for control layer failure.\n")


if __name__ == "__main__":
    pytest.main([__file__])
```

```
--- unitTest_stop_monitoring_availability.py ---
import pytest, logging
from unittest.mock import patch
from test_init import base_test_case, setup_logging, log_test_start_end
import asyncio

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()

async def test_stop_monitoring_availability_success(base_test_case):
    # Simulate the case where monitoring is already running
    base_test_case.availability_control.is_monitoring = True
    base_test_case.availability_control.results = ["Checked availability: Selected or default date is available for
booking."]

    # Expected message to be present in the result
    expected_control_result_contains = "Monitoring stopped successfully!"

    # Execute the stop command
    result = base_test_case.availability_control.stop_monitoring_availability()

    # Log and assert the outcomes
    logging.info(f"Control Layer Expected to contain: {expected_control_result_contains}")
    logging.info(f"Control Layer Received: {result}")

    assert expected_control_result_contains in result, "Control layer assertion failed for stop monitoring."
    logging.info("Unit Test Passed for stop monitoring availability.")

async def test_stop_monitoring_availability_no_active_session(base_test_case):
    # Simulate the case where no monitoring session is active
    base_test_case.availability_control.is_monitoring = False
    expected_control_result = "There was no active availability monitoring session. Nothing to stop."

    # Execute the stop command
    result = base_test_case.availability_control.stop_monitoring_availability()

    # Log and assert the outcomes
    logging.info(f"Control Layer Expected: {expected_control_result}")
    logging.info(f"Control Layer Received: {result}")
    assert result == expected_control_result, "Control layer assertion failed for no active session."
    logging.info("Unit Test Passed for stop monitoring with no active session.")

if __name__ == "__main__":
    pytest.main([__file__])
```

```
--- unitTest_stop_monitoring_price.py ---
import pytest
import logging
from unittest.mock import patch, AsyncMock
from test_init import base_test_case, setup_logging, log_test_start_end

# Enable asyncio for all tests in this file
pytestmark = pytest.mark.asyncio
setup_logging()

async def test_stop_monitoring_price_success(base_test_case):
    # Set up monitoring to be active
    base_test_case.price_control.is_monitoring = True
    base_test_case.price_control.results = ["Price went up!", "Price went down!"]

    # Expected result after stopping monitoring
    expected_result = "Results for price monitoring:\nPrice went up!\nPrice went down!\n\nPrice monitoring
stopped successfully!"

    # Execute the command
    result = base_test_case.price_control.stop_monitoring_price()

    # Log and assert the outcomes
    logging.info(f"Control Layer Expected: {expected_result}")
    logging.info(f"Control Layer Received: {result}")
    assert result == expected_result, "Control layer did not return the correct results for stopping monitoring."
    logging.info("Unit Test Passed for stop_monitoring_price success scenario.\n")


async def test_stop_monitoring_price_not_active(base_test_case):
    # Test the case where monitoring is not active
    base_test_case.price_control.is_monitoring = False
    expected_result = "There was no active price monitoring session. Nothing to stop."

    # Execute the command
    result = base_test_case.price_control.stop_monitoring_price()

    # Log and assert the outcomes
    logging.info(f"Control Layer Expected: {expected_result}")
    logging.info(f"Control Layer Received: {result}")
    assert result == expected_result, "Control layer did not detect that monitoring was not active."
    logging.info("Unit Test Passed for stop_monitoring_price when not active.\n")


async def test_stop_monitoring_price_failure_in_control(base_test_case):
    # Simulate failure in control layer during stopping of monitoring
    with patch('control.PriceControl.PriceControl.stop_monitoring_price', side_effect=Exception("Error stopping
price monitoring")) as mock_stop_monitoring:

        # Expected result when the control layer fails
```

```
        expected_result = "Error stopping price monitoring"

        # Execute the command and handle exception
        try:
            result = base_test_case.price_control.stop_monitoring_price()
        except Exception as e:
            result = str(e)

        # Log and assert the outcomes
        logging.info(f"Control Layer Expected: {expected_result}")
        logging.info(f"Control Layer Received: {result}")
        assert expected_result in result, "Control layer did not handle the failure correctly."
        logging.info("Unit Test Passed for stop_monitoring_price failure scenario.\n")


if __name__ == "__main__":
    pytest.main([__file__])
```