Discord Bot Automation Assistant

Discord Bot Automation Assistant Test Plan

Oguz Kaan Yildirim

307637

Table Of Contents

INTRDOCUTION

This document provides an overview of unit testing for a software project aimed at automating the monitoring of product prices and service availability. The goal of the testing is to ensure that all system components function correctly when tested in isolation. This modular approach facilitates the validation of core system functions, including command processing, browser automation, and data export, within a controlled test environment.

The system is composed of modules responsible for interacting with web browsers, processing user commands, retrieving product data from websites, and monitoring availability for services like reservations. These modules have undergone rigorous testing using Python's pytest framework. External systems like websites and Discord commands are simulated using mocks and patches, ensuring expected system behavior in both typical and edge-case scenarios.

This document includes:

- An outline of the testing strategy, scope, and objectives,
- A description of the tools and technologies used during testing,
- Solutions to challenges encountered when testing Discord commands,
- Details on the test setup, implementation, and how the testing framework integrates with the system architecture.

The purpose of the unit testing is to confirm that the system can accurately process user commands, interact with websites, retrieve data, log it, and generate reports. Isolating components during testing enhances confidence in the system's reliability and robustness.

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.

Unit Tests also has descriptions and executable steps explained in testing.

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

Test Plan Overview

Scope

The scope of the unit tests covers all critical aspects of the system, ensuring each component performs its intended function independently. This modular testing approach covers:

- **Command Processing and Core Features:** Verifying that the system properly receives and processes user commands to monitor prices, check availability, and log data efficiently.
- **Browser Interactions:** Ensuring that the system can initiate, navigate, and close browser sessions while effectively interacting with web content.
- **Data Logging and Export:** Validating that price and availability data are logged and exported in structured formats such as Excel and HTML.
- **Error Handling:** Confirming that the system gracefully handles errors (e.g., invalid commands or network issues) and provides appropriate user feedback.

Objectives

The unit testing aims to:

- **Functional Verification:** Ensure that components like command processing, data retrieval, and logging function correctly in isolation.
- **Component Isolation:** By testing each module independently, failures in one area don't affect others, enabling easier identification of defects.
- **Data Accuracy and Consistency:** Ensure the system processes price and availability data correctly before logging and exporting it.
- **System Reliability:** Test the system's ability to handle various scenarios, including repeated commands, long-running processes, and invalid inputs.

Strategy

The strategy focuses on modular unit testing, ensuring each part of the system is validated without relying on external dependencies like live websites or browsers. Key elements include:

- **Unit Testing:** Each system module—command processing, web scraping, or data export—is tested independently.
- **Mocking and Simulation:** External systems are simulated using mocks, allowing tests to focus on internal logic without live interactions.
- **Automated Execution:** Tests are automated using pytest, ensuring consistency and enabling integration into CI/CD pipelines for automatic execution upon code changes.

Structure of the Tests

The tests are divided into suites targeting specific components:

- **Control Layer:** Verifies user commands are correctly processed.
- **Entity Layer:** Validates interactions with external systems (e.g., retrieving product prices or checking availability).
- **Data Logging and Export:** Ensures data is logged and exported without errors.

This structure allows the test framework to expand as the system evolves, enabling independent testing of new features without disrupting existing tests.

Expected Outcomes

This modular approach is expected to yield:

- Accurate command processing with correct results,
- Error-free logging and export of data,
- Graceful handling of unexpected situations like invalid commands or network failures,
- Stable performance during long-running tasks, such as continuous monitoring of product prices.

Tools and Technologies

Pytest

The primary framework used for test execution is pytest, which supports both synchronous and asynchronous testing. This is critical since many system operations involve real-time monitoring and asynchronous tasks like web scraping. Integration with mocking tools allows thorough simulation of external dependencies, ensuring isolated and repeatable tests.

Unittest.mock

The unittest.mock library is key to isolating system components from external dependencies, such as web browsers and Discord commands. The system uses Mock and AsyncMock to simulate responses from these services, enabling tests to focus on internal logic.

- **Mocking External Systems:** Mocks simulate browser actions and Discord command inputs, allowing test isolation.

pytest-asyncio

As many system operations are asynchronous, pytest-asyncio manages async code in tests, ensuring that operations like price monitoring are tested properly.

Mocked Selenium

Selenium, a tool for browser automation, is mocked during unit testing to focus on internal logic without requiring real browser instances.

Purpose and Setup

Purpose of Unit Testing

The purpose of unit testing is to validate that each component of the system functions correctly in isolation, focusing on command processing, web interactions, and data export operations.

Challenges in Testing Discord Commands

Testing Discord commands posed a challenge due to the lack of native testing tools for discord.py. To address this, unit tests simulate command inputs and directly interact with the control layer methods, ensuring that the system logic is properly tested without relying on live Discord command handling.

Setup of the Testing Environment

The environment is designed to ensure isolated component testing without live system dependencies:

- **Mocking and Patching:** The unittest.mock library simulates external systems like browsers and Discord commands, ensuring independent testing of each component.
- **Asynchronous Testing:** pytest-asyncio allows proper testing of asynchronous tasks like price monitoring.
- **Test Isolation:** Each test runs independently, ensuring faster execution and easier debugging.

Implementation Details

The structure of the tests mirrors the system's modular architecture, with distinct test suites for each layer:

- **Control Layer Tests:** Validate command processing to ensure inputs are handled correctly.
- **Entity Layer Tests:** Confirm the core functionality of price retrieval, availability checking, and data export.
- **Logging and Export Tests:** Ensure data is correctly formatted and saved to Excel and HTML files.

By focusing on these areas, the unit tests confirm that the system's logic functions as intended, and any external dependencies are properly mocked for accurate results.

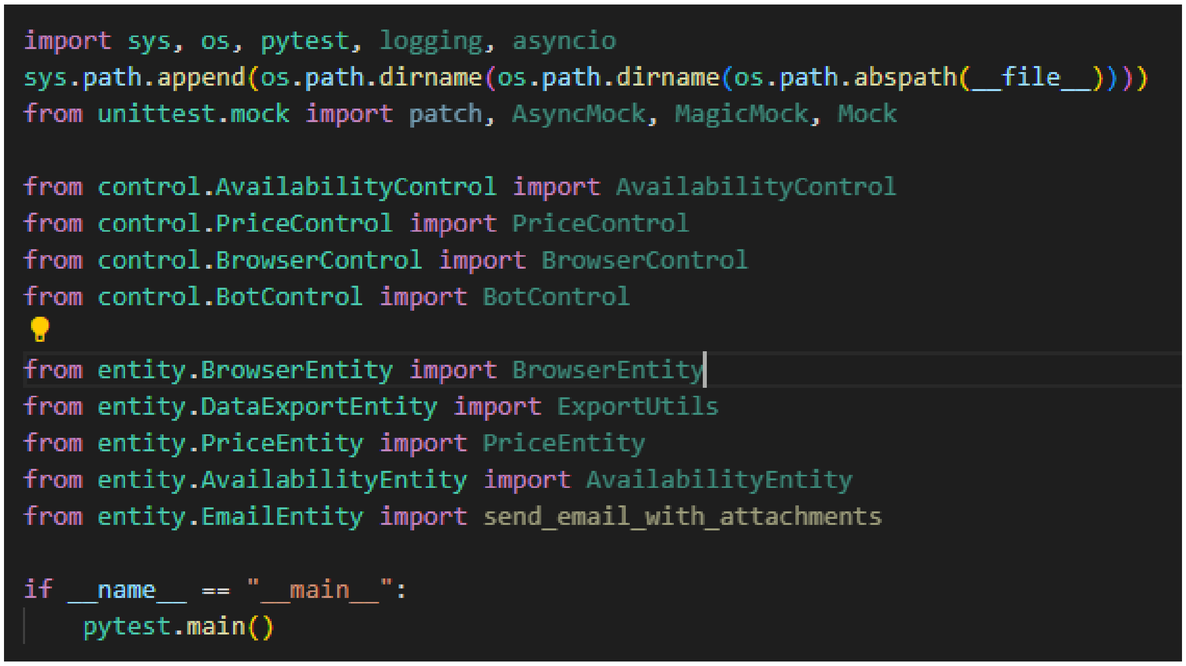
Unit Tests for Use Cases

Every use case has multiple unit tests in them for every step in main flow.

test_init.py

Description

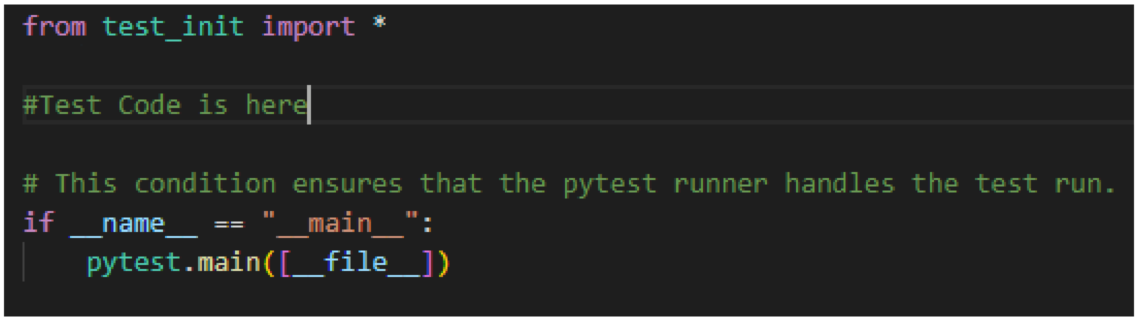
The test_init file serves two main purposes. First, it consolidates all necessary imports to avoid redundant import statements across multiple test files, improving maintainability and consistency. Second, it provides functionality to run all unit tests at once by executing test_init.py.



```python
import sys, os, pytest, logging, asyncio
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from unittest.mock import patch, AsyncMock, MagicMock, Mock

from control.AvailabilityControl import AvailabilityControl
from control.PriceControl import PriceControl
from control.BrowserControl import BrowserControl
from control.BotControl import BotControl

from entity.BrowserEntity import BrowserEntity
from entity.DataExportEntity import ExportUtils
from entity.PriceEntity import PriceEntity
from entity.AvailabilityEntity import AvailabilityEntity
from entity.EmailEntity import send_email_with_attachments

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

If specific tests need to be run individually, each test file has the if __name__ == "__main__": pytest.main([__file__]) block, allowing users to run that specific test file independently.

This setup streamlines both the import process and test execution, making it easy to run tests collectively or individually based on the needs of the project.



```python
from test_init import *

#Test Code is here

# This condition ensures that the pytest runner handles the test run.
if __name__ == "__main__":
    pytest.main([__file__])
```

1. !project_help

Description

This test ensures that the BotControl.receive_command() method processes the !project_help command correctly and returns the appropriate help message listing available commands.

Test Steps

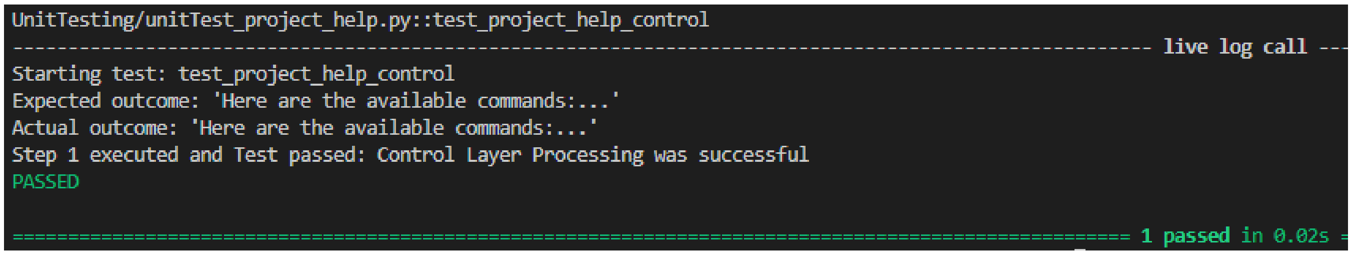
The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**

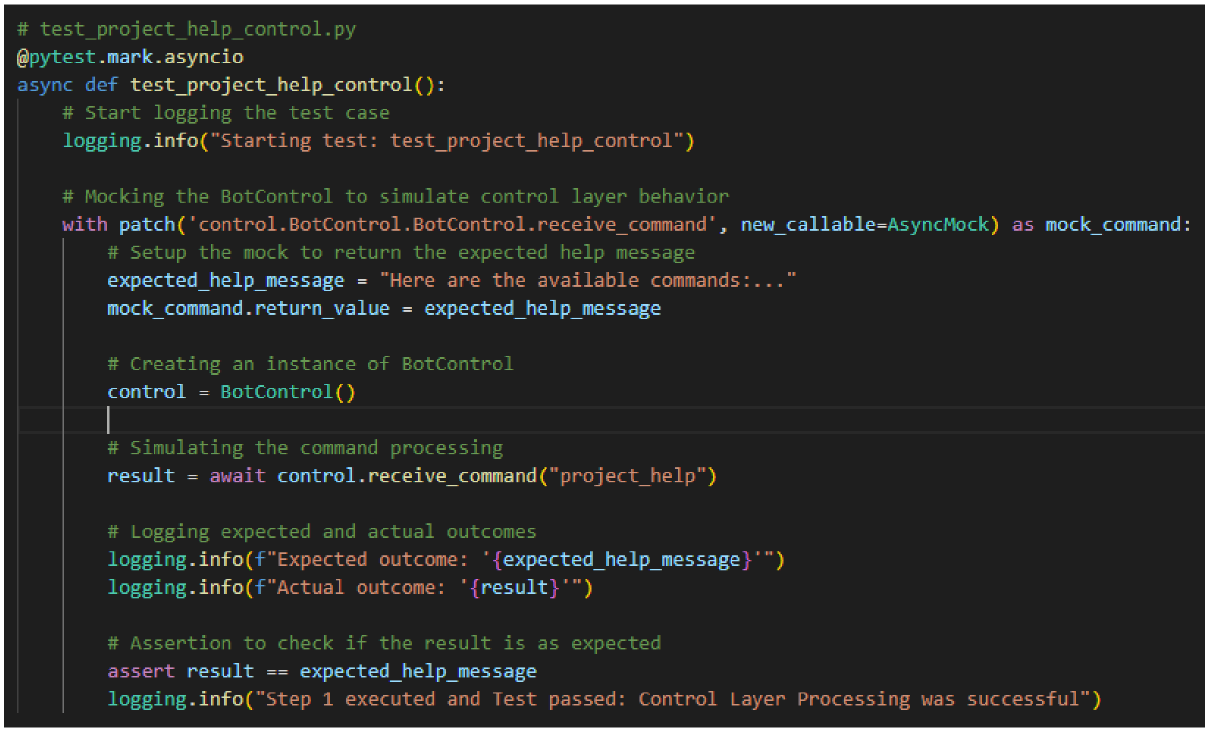
   This test will ensure that BotControl.receive_command() handles the !project_help command correctly and returns the expected help message.

Test Data

- Command: "!project_help"
- Expected Output: "Here are the available commands:..."



```
UnitTesting/unitTest_project_help.py::test_project_help_control
------------------------------------------------------------------------------------------------- live log call --
Starting test: test_project_help_control
Expected outcome: 'Here are the available commands:...'
Actual outcome: 'Here are the available commands:...'
Step 1 executed and Test passed: Control Layer Processing was successful
PASSED

=============================================================================================== 1 passed in 0.02s =
```



```python
# test_project_help_control.py
@pytest.mark.asyncio
async def test_project_help_control():
    # Start logging the test case
    logging.info("Starting test: test_project_help_control")

    # Mocking the BotControl to simulate control layer behavior
    with patch('control.BotControl.BotControl.receive_command', new_callable=AsyncMock) as mock_command:
        # Setup the mock to return the expected help message
        expected_help_message = "Here are the available commands:..."
        mock_command.return_value = expected_help_message

        # Creating an instance of BotControl
        control = BotControl()

        # Simulating the command processing
        result = await control.receive_command("project_help")

        # Logging expected and actual outcomes
        logging.info(f"Expected outcome: '{expected_help_message}'")
        logging.info(f"Actual outcome: '{result}'")

        # Assertion to check if the result is as expected
        assert result == expected_help_message
        logging.info("Step 1 executed and Test passed: Control Layer Processing was successful")
```

Figure 1

2. !receive_email <file name>

Description

This test ensures that the BotControl.receive_command() method processes the !receive_email command correctly by passing the file name to the email handler, sending the email, and generating the correct response.

Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**

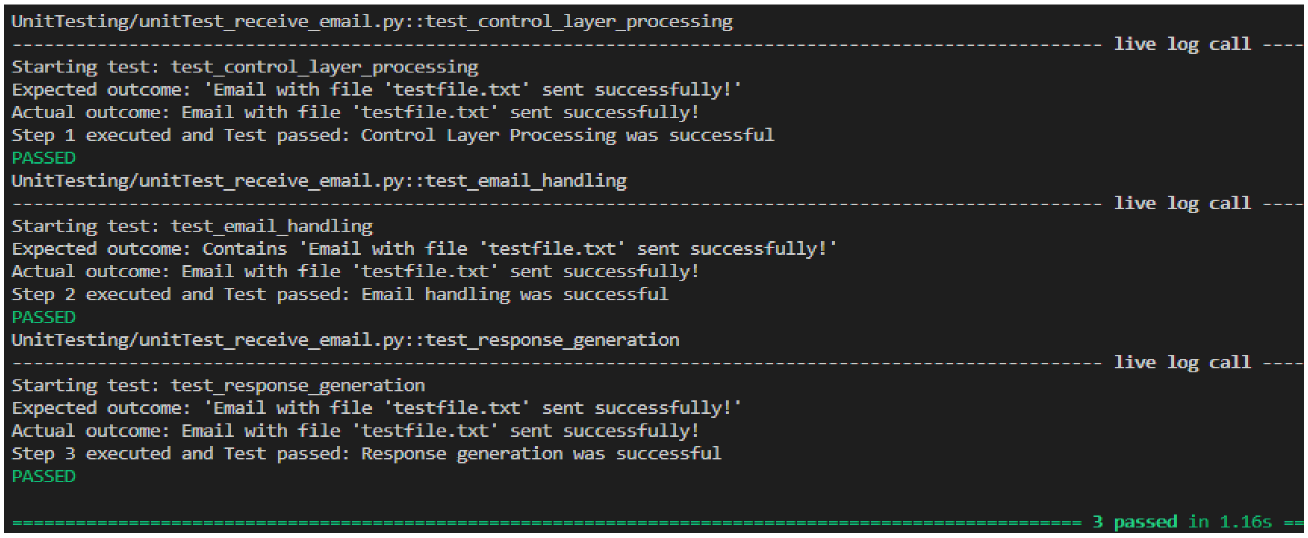
   This test ensures that BotControl.receive_command() handles the !receive_email command correctly, including proper parameter passing and validation.

2. **Email Handling**

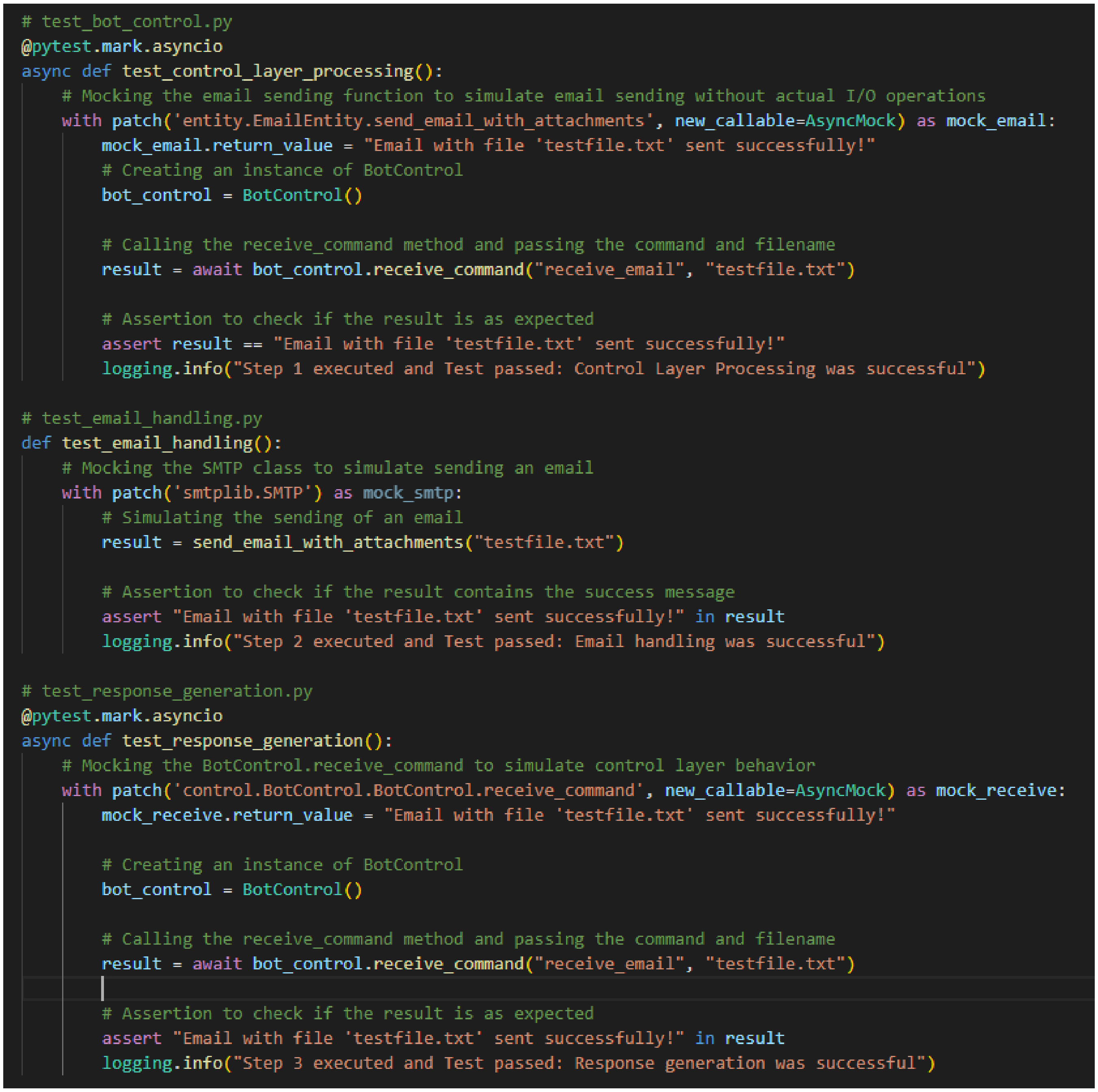
   This test focuses on the EmailEntity.send_email_with_attachments() function to ensure it processes the request to send the email with the attached file.

3. **Response Generation**

   This test validates that the control layer correctly interprets the response from the email handling step and returns the appropriate result to the boundary layer.



```
UnitTesting/unitTest_receive_email.py::test_control_layer_processing
------------------------------------------------------------------------------------------------------ live log call ----
Starting test: test_control_layer_processing
Expected outcome: 'Email with file 'testfile.txt' sent successfully!'
Actual outcome: Email with file 'testfile.txt' sent successfully!
Step 1 executed and Test passed: Control Layer Processing was successful
PASSED
UnitTesting/unitTest_receive_email.py::test_email_handling
------------------------------------------------------------------------------------------------------ live log call ----
Starting test: test_email_handling
Expected outcome: Contains 'Email with file 'testfile.txt' sent successfully!'
Actual outcome: Email with file 'testfile.txt' sent successfully!
Step 2 executed and Test passed: Email handling was successful
PASSED
UnitTesting/unitTest_receive_email.py::test_response_generation
------------------------------------------------------------------------------------------------------ live log call ----
Starting test: test_response_generation
Expected outcome: 'Email with file 'testfile.txt' sent successfully!'
Actual outcome: Email with file 'testfile.txt' sent successfully!
Step 3 executed and Test passed: Response generation was successful
PASSED

==================================================================================================== 3 passed in 1.16s ==
```


```python
# test_bot_control.py
@pytest.mark.asyncio
async def test_control_layer_processing():
    # Mocking the email sending function to simulate email sending without actual I/O operations
    with patch('entity.EmailEntity.send_email_with_attachments', new_callable=AsyncMock) as mock_email:
        mock_email.return_value = "Email with file 'testfile.txt' sent successfully!"
        # Creating an instance of BotControl
        bot_control = BotControl()

        # Calling the receive_command method and passing the command and filename
        result = await bot_control.receive_command("receive_email", "testfile.txt")

        # Assertion to check if the result is as expected
        assert result == "Email with file 'testfile.txt' sent successfully!"
        logging.info("Step 1 executed and Test passed: Control Layer Processing was successful")

# test_email_handling.py
def test_email_handling():
    # Mocking the SMTP class to simulate sending an email
    with patch('smtplib.SMTP') as mock_smtp:
        # Simulating the sending of an email
        result = send_email_with_attachments("testfile.txt")

        # Assertion to check if the result contains the success message
        assert "Email with file 'testfile.txt' sent successfully!" in result
        logging.info("Step 2 executed and Test passed: Email handling was successful")

# test_response_generation.py
@pytest.mark.asyncio
async def test_response_generation():
    # Mocking the BotControl.receive_command to simulate control layer behavior
    with patch('control.BotControl.BotControl.receive_command', new_callable=AsyncMock) as mock_receive:
        mock_receive.return_value = "Email with file 'testfile.txt' sent successfully!"

        # Creating an instance of BotControl
        bot_control = BotControl()

        # Calling the receive_command method and passing the command and filename
        result = await bot_control.receive_command("receive_email", "testfile.txt")

        # Assertion to check if the result is as expected
        assert "Email with file 'testfile.txt' sent successfully!" in result
        logging.info("Step 3 executed and Test passed: Response generation was successful")
```

3. !navigate_to_website

Description

This test ensures that the BotControl.receive_command() method processes the !navigate_to_website command correctly by extracting the URL, navigating to the specified website, and returning the appropriate result.

Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**

   This test ensures that BotControl.receive_command() handles the !navigate_to_website command correctly by extracting the URL and passing it to the browser control.

2. **Browser Navigation**

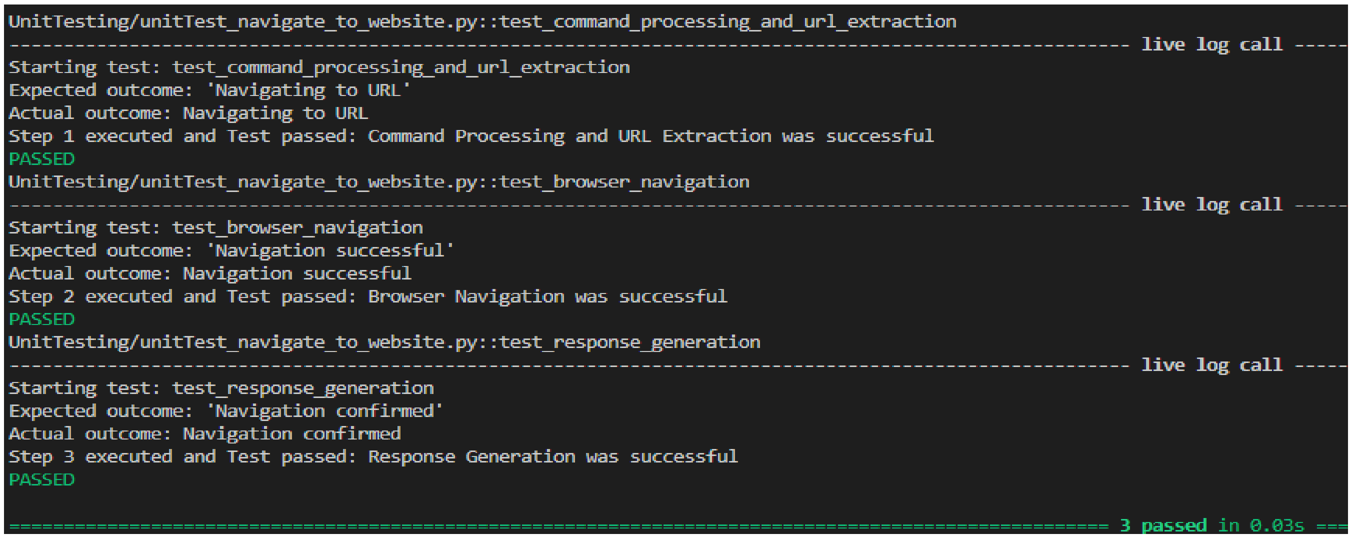
   This test ensures that the BrowserEntity.navigate_to_website() function processes the navigation request to the specified URL correctly.

3. **Response Generation**

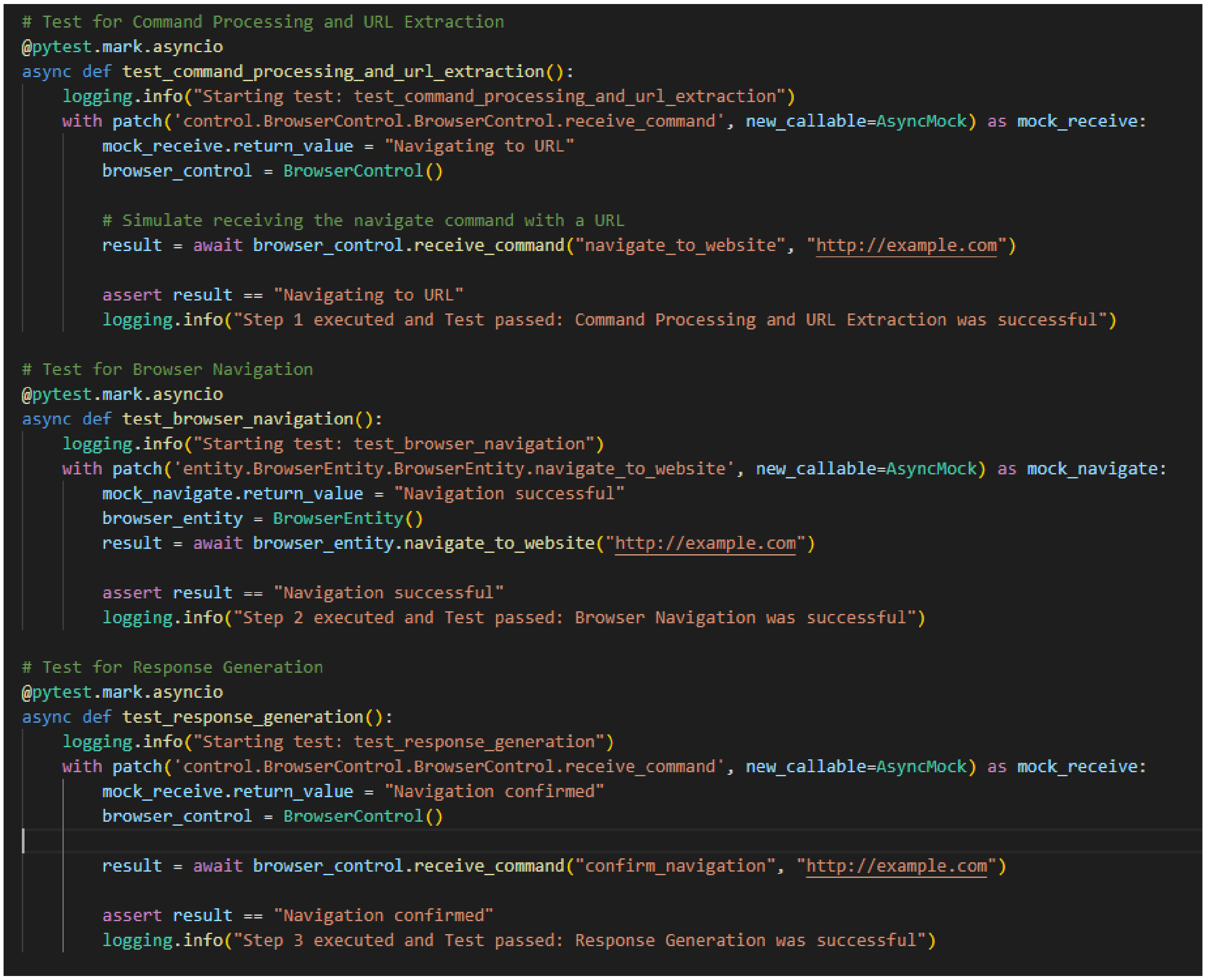
   This test validates that the control layer correctly returns the appropriate result after the browser interaction is completed.

Test Data

- Command: "!navigate_to_website"
- Test URL: "http://example.com"
- Expected Output: "Navigation successful"



```
UnitTesting/unitTest_navigate_to_website.py::test_command_processing_and_url_extraction
------------------------------------------------------------------------------------------ live log call -----
Starting test: test_command_processing_and_url_extraction
Expected outcome: 'Navigating to URL'
Actual outcome: Navigating to URL
Step 1 executed and Test passed: Command Processing and URL Extraction was successful
PASSED
UnitTesting/unitTest_navigate_to_website.py::test_browser_navigation
------------------------------------------------------------------------------------------ live log call -----
Starting test: test_browser_navigation
Expected outcome: 'Navigation successful'
Actual outcome: Navigation successful
Step 2 executed and Test passed: Browser Navigation was successful
PASSED
UnitTesting/unitTest_navigate_to_website.py::test_response_generation
------------------------------------------------------------------------------------------ live log call -----
Starting test: test_response_generation
Expected outcome: 'Navigation confirmed'
Actual outcome: Navigation confirmed
Step 3 executed and Test passed: Response Generation was successful
PASSED

======================================================================================== 3 passed in 0.03s ===
```



```python
# Test for Command Processing and URL Extraction
@pytest.mark.asyncio
async def test_command_processing_and_url_extraction():
    logging.info("Starting test: test_command_processing_and_url_extraction")
    with patch('control.BrowserControl.BrowserControl.receive_command', new_callable=AsyncMock) as mock_receive:
        mock_receive.return_value = "Navigating to URL"
        browser_control = BrowserControl()

        # Simulate receiving the navigate command with a URL
        result = await browser_control.receive_command("navigate_to_website", "http://example.com")

        assert result == "Navigating to URL"
        logging.info("Step 1 executed and Test passed: Command Processing and URL Extraction was successful")

# Test for Browser Navigation
@pytest.mark.asyncio
async def test_browser_navigation():
    logging.info("Starting test: test_browser_navigation")
    with patch('entity.BrowserEntity.BrowserEntity.navigate_to_website', new_callable=AsyncMock) as mock_navigate:
        mock_navigate.return_value = "Navigation successful"
        browser_entity = BrowserEntity()
        result = await browser_entity.navigate_to_website("http://example.com")

        assert result == "Navigation successful"
        logging.info("Step 2 executed and Test passed: Browser Navigation was successful")

# Test for Response Generation
@pytest.mark.asyncio
async def test_response_generation():
    logging.info("Starting test: test_response_generation")
    with patch('control.BrowserControl.BrowserControl.receive_command', new_callable=AsyncMock) as mock_receive:
        mock_receive.return_value = "Navigation confirmed"
        browser_control = BrowserControl()

        result = await browser_control.receive_command("confirm_navigation", "http://example.com")

        assert result == "Navigation confirmed"
        logging.info("Step 3 executed and Test passed: Response Generation was successful")
```

4. !login

Description

This test ensures that the BotControl.receive_command() method processes the !login command correctly by passing the website, username, and password to the browser and verifying the login process.

Test Steps

The main flow for this use case is as follows, and we will test the following steps:

1. **Control Layer Processing**

   This test ensures that BotControl.receive_command() handles the !login command correctly, including proper parameter passing and validation.

2. **Website Interaction**

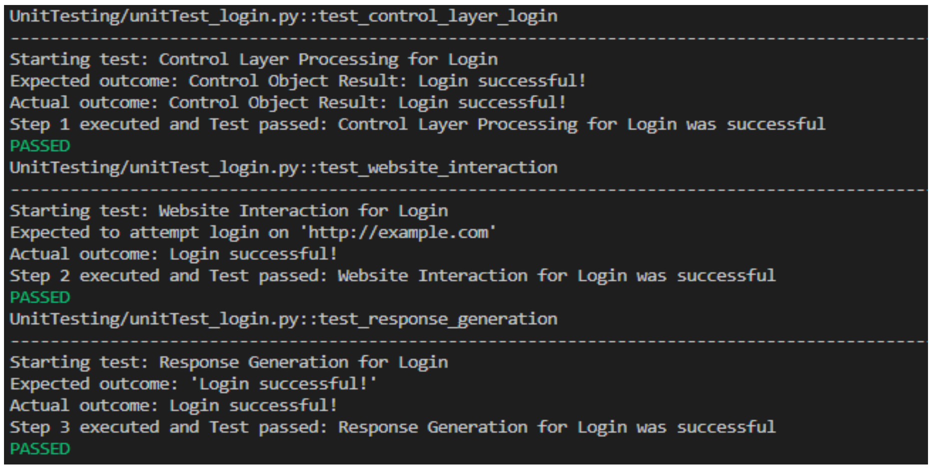
   This test focuses on the BrowserEntity.login() function to ensure it processes the request to log in to the website using the provided credentials.

3. **Response Generation**

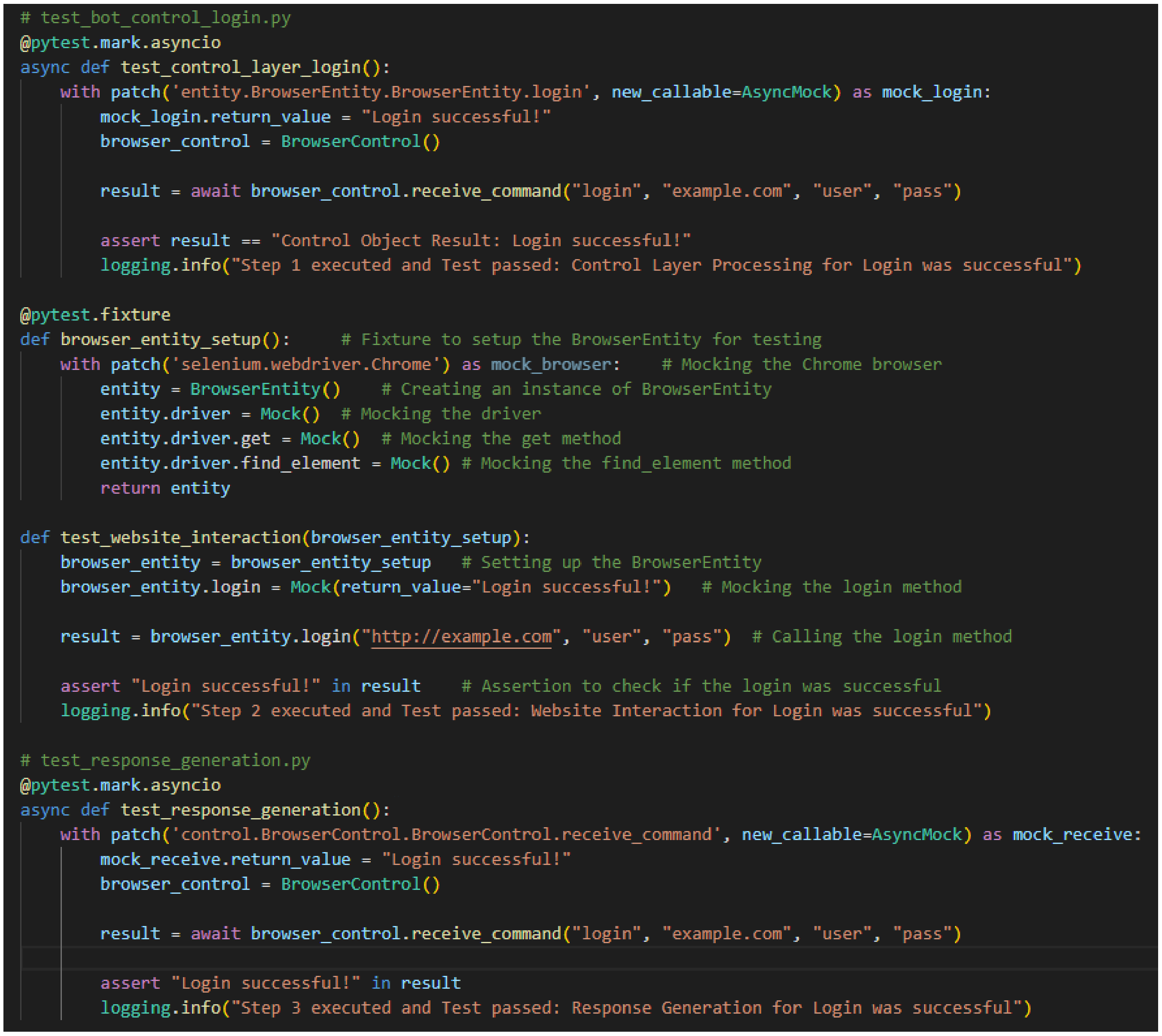
   This test validates that the control layer correctly interprets the response from the website interaction step and returns the appropriate result to the boundary layer.

Test Data

- Command: "!login"
- Test website: "http://example.com"
- Test username: "user"
- Test password: "pass"
- Expected Output: "Login successful"



```
UnitTesting/unitTest_login.py::test_control_layer_login
------------------------------------------------------------------------------------------
Starting test: Control Layer Processing for Login
Expected outcome: Control Object Result: Login successful!
Actual outcome: Control Object Result: Login successful!
Step 1 executed and Test passed: Control Layer Processing for Login was successful
PASSED
UnitTesting/unitTest_login.py::test_website_interaction
------------------------------------------------------------------------------------------
Starting test: Website Interaction for Login
Expected to attempt login on 'http://example.com'
Actual outcome: Login successful!
Step 2 executed and Test passed: Website Interaction for Login was successful
PASSED
UnitTesting/unitTest_login.py::test_response_generation
------------------------------------------------------------------------------------------
Starting test: Response Generation for Login
Expected outcome: 'Login successful!'
Actual outcome: Login successful!
Step 3 executed and Test passed: Response Generation for Login was successful
PASSED
```



```python
# test_bot_control_login.py
@pytest.mark.asyncio
async def test_control_layer_login():
    with patch('entity.BrowserEntity.BrowserEntity.login', new_callable=AsyncMock) as mock_login:
        mock_login.return_value = "Login successful!"
        browser_control = BrowserControl()

        result = await browser_control.receive_command("login", "example.com", "user", "pass")

        assert result == "Control Object Result: Login successful!"
        logging.info("Step 1 executed and Test passed: Control Layer Processing for Login was successful")

@pytest.fixture
def browser_entity_setup():     # Fixture to setup the BrowserEntity for testing
    with patch('selenium.webdriver.Chrome') as mock_browser:    # Mocking the Chrome browser
        entity = BrowserEntity()    # Creating an instance of BrowserEntity
        entity.driver = Mock()  # Mocking the driver
        entity.driver.get = Mock()  # Mocking the get method
        entity.driver.find_element = Mock() # Mocking the find_element method
        return entity

def test_website_interaction(browser_entity_setup):
    browser_entity = browser_entity_setup   # Setting up the BrowserEntity
    browser_entity.login = Mock(return_value="Login successful!")   # Mocking the login method

    result = browser_entity.login("http://example.com", "user", "pass")  # Calling the login method

    assert "Login successful!" in result    # Assertion to check if the login was successful
    logging.info("Step 2 executed and Test passed: Website Interaction for Login was successful")

# test_response_generation.py
@pytest.mark.asyncio
async def test_response_generation():
    with patch('control.BrowserControl.BrowserControl.receive_command', new_callable=AsyncMock) as mock_receive:
        mock_receive.return_value = "Login successful!"
        browser_control = BrowserControl()

        result = await browser_control.receive_command("login", "example.com", "user", "pass")

        assert "Login successful!" in result
        logging.info("Step 3 executed and Test passed: Response Generation for Login was successful")
```

5. !close_browser

Description

This test ensures that the BotControl.receive_command() method processes the !close_browser command correctly by handling browser closure and returning the appropriate response.

Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**

   This test ensures that BotControl.receive_command() handles the !close_browser command correctly.

2. **Browser Closing**

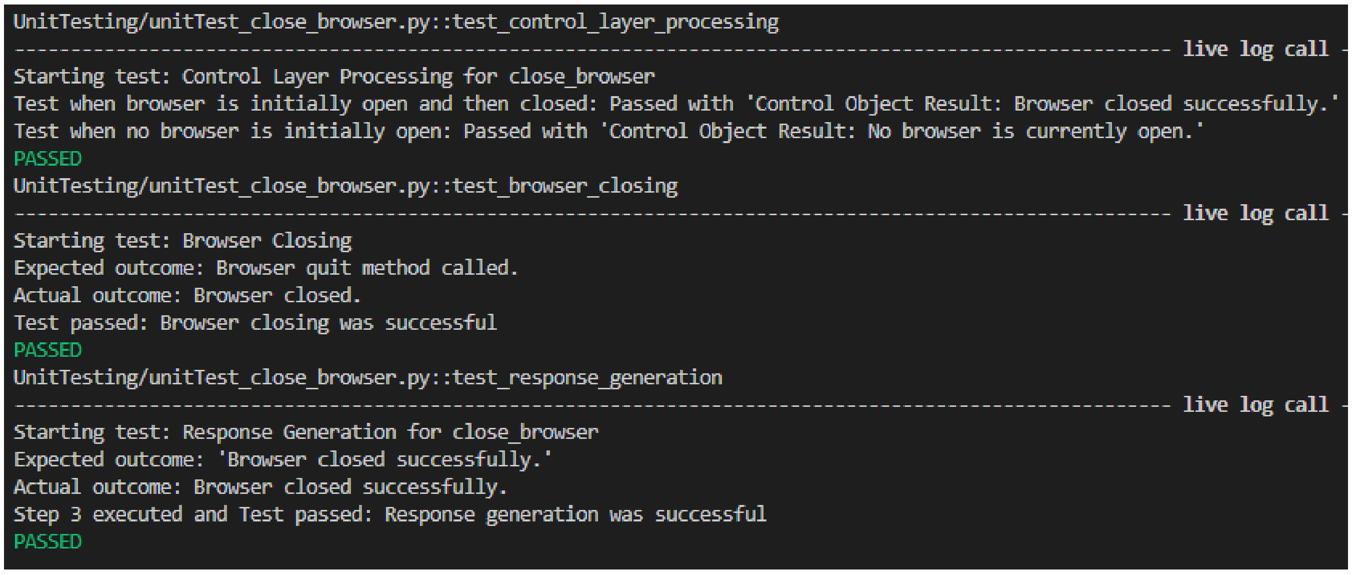
   This test ensures that the BrowserEntity.close_browser() function successfully closes the browser.

3. **Response Generation**

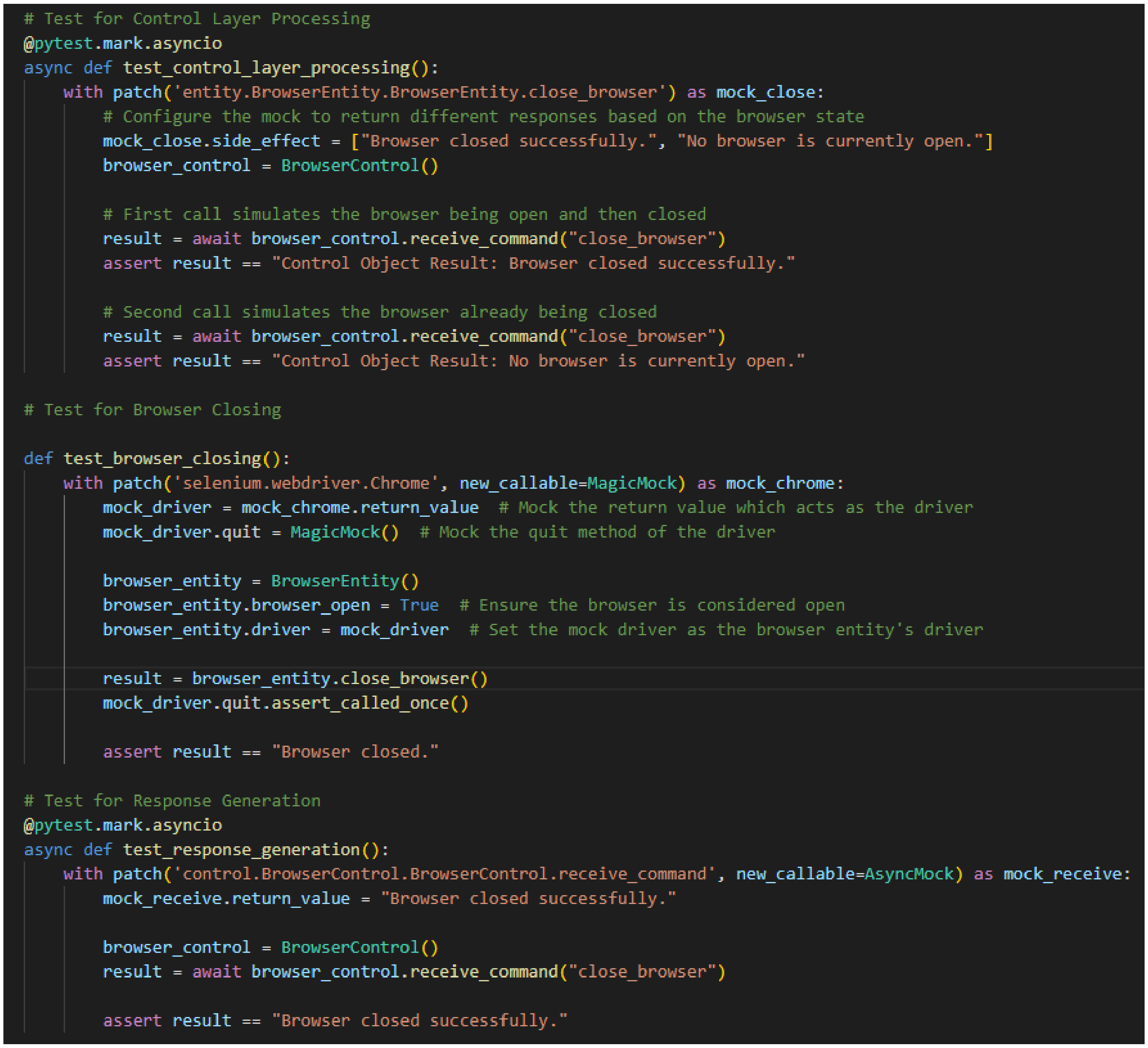
   This test validates that the control layer correctly interprets the browser closure and returns the appropriate result to the boundary layer.

Test Data

- Command: "!close_browser"
- Expected Output: "Browser closed successfully"



```
UnitTesting/unitTest_close_browser.py::test_control_layer_processing
---------------------------------------------------------------------------------------------------- live log call -
Starting test: Control Layer Processing for close_browser
Test when browser is initially open and then closed: Passed with 'Control Object Result: Browser closed successfully.'
Test when no browser is initially open: Passed with 'Control Object Result: No browser is currently open.'
PASSED
UnitTesting/unitTest_close_browser.py::test_browser_closing
---------------------------------------------------------------------------------------------------- live log call -
Starting test: Browser Closing
Expected outcome: Browser quit method called.
Actual outcome: Browser closed.
Test passed: Browser closing was successful
PASSED
UnitTesting/unitTest_close_browser.py::test_response_generation
---------------------------------------------------------------------------------------------------- live log call -
Starting test: Response Generation for close_browser
Expected outcome: 'Browser closed successfully.'
Actual outcome: Browser closed successfully.
Step 3 executed and Test passed: Response generation was successful
PASSED
```



```python
# Test for Control Layer Processing
@pytest.mark.asyncio
async def test_control_layer_processing():
    with patch('entity.BrowserEntity.BrowserEntity.close_browser') as mock_close:
        # Configure the mock to return different responses based on the browser state
        mock_close.side_effect = ["Browser closed successfully.", "No browser is currently open."]
        browser_control = BrowserControl()

        # First call simulates the browser being open and then closed
        result = await browser_control.receive_command("close_browser")
        assert result == "Control Object Result: Browser closed successfully."

        # Second call simulates the browser already being closed
        result = await browser_control.receive_command("close_browser")
        assert result == "Control Object Result: No browser is currently open."

# Test for Browser Closing

def test_browser_closing():
    with patch('selenium.webdriver.Chrome', new_callable=MagicMock) as mock_chrome:
        mock_driver = mock_chrome.return_value  # Mock the return value which acts as the driver
        mock_driver.quit = MagicMock()  # Mock the quit method of the driver

        browser_entity = BrowserEntity()
        browser_entity.browser_open = True  # Ensure the browser is considered open
        browser_entity.driver = mock_driver  # Set the mock driver as the browser entity's driver

        result = browser_entity.close_browser()
        mock_driver.quit.assert_called_once()

        assert result == "Browser closed."

# Test for Response Generation
@pytest.mark.asyncio
async def test_response_generation():
    with patch('control.BrowserControl.BrowserControl.receive_command', new_callable=AsyncMock) as mock_receive:
        mock_receive.return_value = "Browser closed successfully."

        browser_control = BrowserControl()
        result = await browser_control.receive_command("close_browser")

        assert result == "Browser closed successfully."
```

6. !get_price <website>

Description

This test ensures that the BotControl.receive_command() method processes the !get_price command correctly by extracting the website URL, retrieving the price, and logging the data to both Excel and HTML formats.

Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**

   This test ensures that BotControl.receive_command() handles the !get_price command correctly, including URL parameter handling.

2. **Price Retrieval**

   This test ensures that the PriceEntity.get_price_from_page() function retrieves the correct price from the webpage.

3. **Data Logging to Excel**

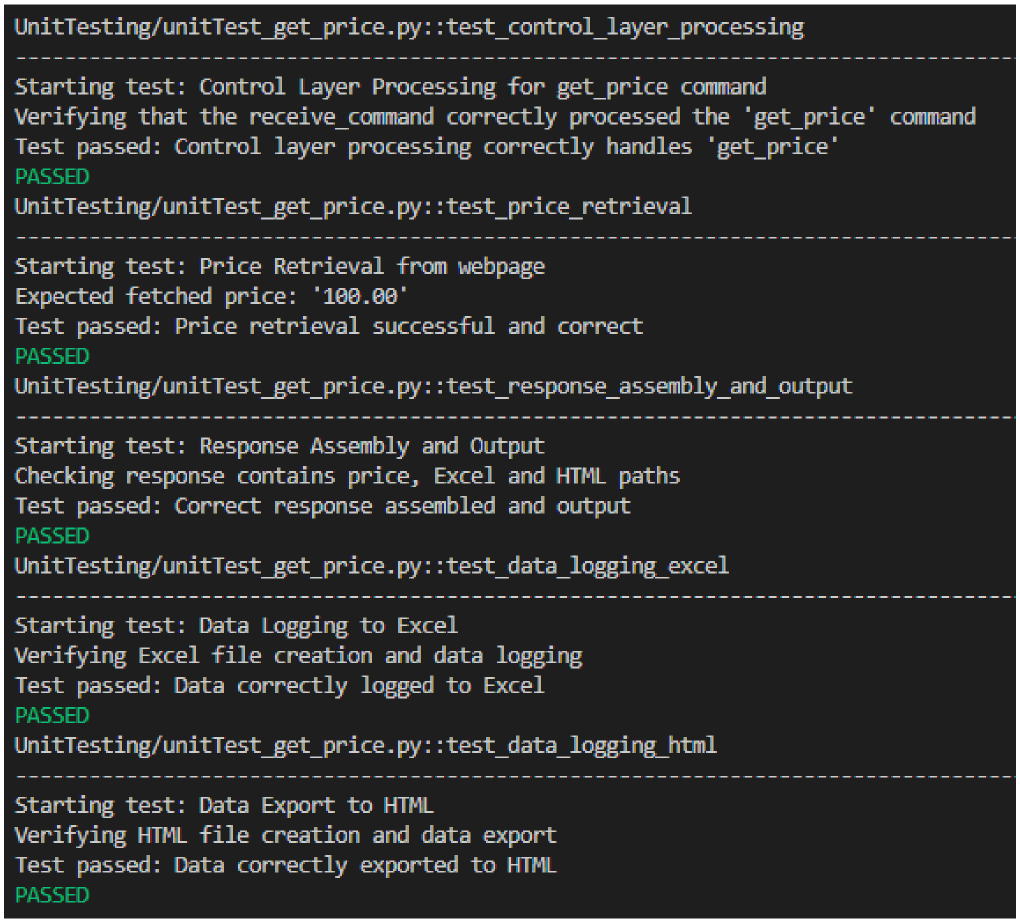
   This test verifies that the retrieved price data is correctly logged to an Excel file.

4. **Data Logging to HTML**

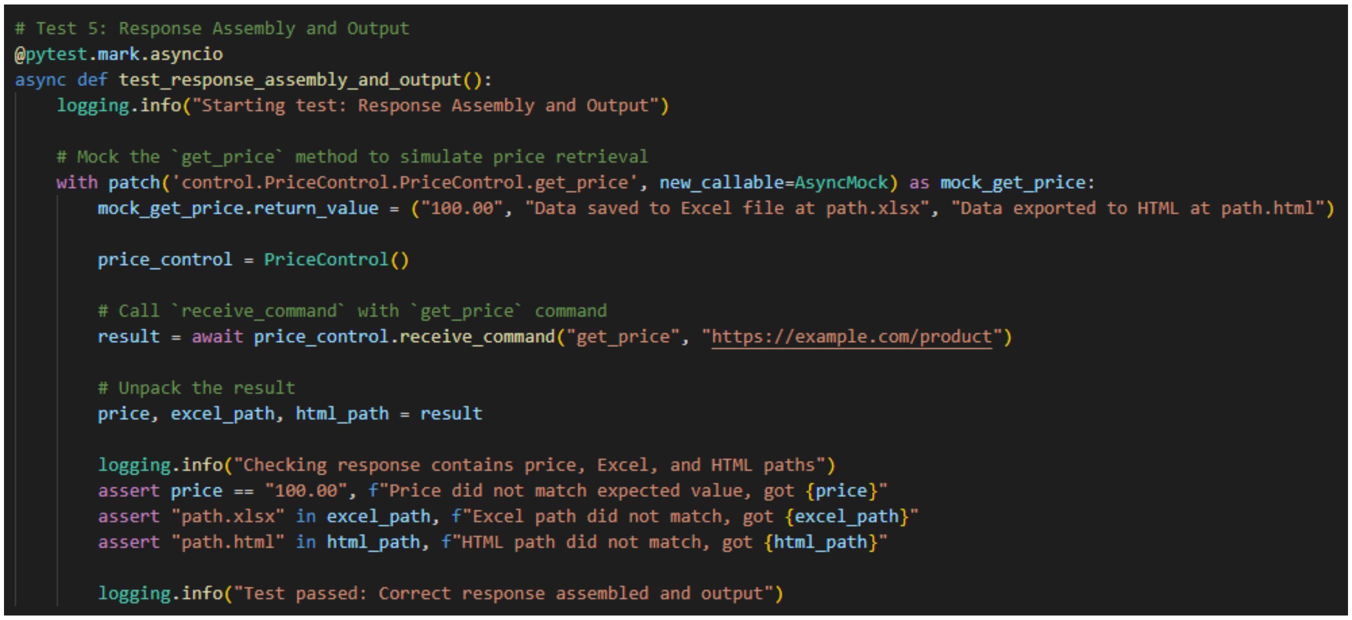
   This test ensures that the price data is correctly exported to an HTML file.

Test Data

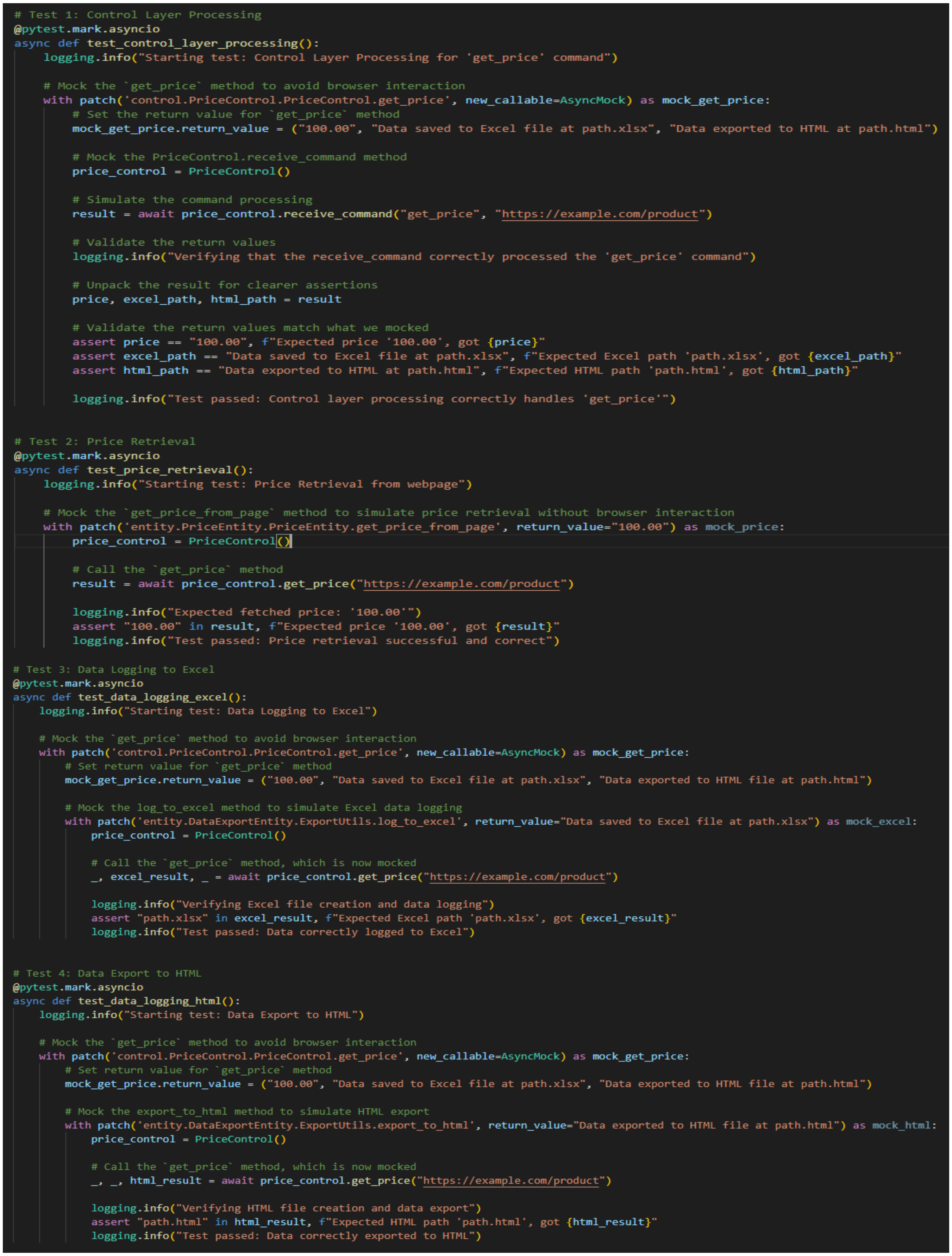
- Command: "!get_price"
- Test website: "http://example.com/product"
- Expected Output: "100.00"



```
UnitTesting/unitTest_get_price.py::test_control_layer_processing
--------------------------------------------------------------------------------
Starting test: Control Layer Processing for get_price command
Verifying that the receive_command correctly processed the 'get_price' command
Test passed: Control layer processing correctly handles 'get_price'
PASSED
UnitTesting/unitTest_get_price.py::test_price_retrieval
--------------------------------------------------------------------------------
Starting test: Price Retrieval from webpage
Expected fetched price: '100.00'
Test passed: Price retrieval successful and correct
PASSED
UnitTesting/unitTest_get_price.py::test_response_assembly_and_output
--------------------------------------------------------------------------------
Starting test: Response Assembly and Output
Checking response contains price, Excel and HTML paths
Test passed: Correct response assembled and output
PASSED
UnitTesting/unitTest_get_price.py::test_data_logging_excel
--------------------------------------------------------------------------------
Starting test: Data Logging to Excel
Verifying Excel file creation and data logging
Test passed: Data correctly logged to Excel
PASSED
UnitTesting/unitTest_get_price.py::test_data_logging_html
--------------------------------------------------------------------------------
Starting test: Data Export to HTML
Verifying HTML file creation and data export
Test passed: Data correctly exported to HTML
PASSED
```



```python
# Test 5: Response Assembly and Output
@pytest.mark.asyncio
async def test_response_assembly_and_output():
    logging.info("Starting test: Response Assembly and Output")

    # Mock the `get_price` method to simulate price retrieval
    with patch('control.PriceControl.PriceControl.get_price', new_callable=AsyncMock) as mock_get_price:
        mock_get_price.return_value = ("100.00", "Data saved to Excel file at path.xlsx", "Data exported to HTML at path.html")

        price_control = PriceControl()

        # Call `receive_command` with `get_price` command
        result = await price_control.receive_command("get_price", "https://example.com/product")

        # Unpack the result
        price, excel_path, html_path = result

        logging.info("Checking response contains price, Excel, and HTML paths")
        assert price == "100.00", f"Price did not match expected value, got {price}"
        assert "path.xlsx" in excel_path, f"Excel path did not match, got {excel_path}"
        assert "path.html" in html_path, f"HTML path did not match, got {html_path}"

        logging.info("Test passed: Correct response assembled and output")
```



```python
# Test 1: Control Layer Processing
@pytest.mark.asyncio
async def test_control_layer_processing():
    logging.info("Starting test: Control Layer Processing for 'get_price' command")

    # Mock the `get_price` method to avoid browser interaction
    with patch('control.PriceControl.PriceControl.get_price', new_callable=AsyncMock) as mock_get_price:
        # Set the return value for `get_price` method
        mock_get_price.return_value = ("100.00", "Data saved to Excel file at path.xlsx", "Data exported to HTML at path.html")

        # Mock the PriceControl.receive_command method
        price_control = PriceControl()

        # Simulate the command processing
        result = await price_control.receive_command("get_price", "https://example.com/product")

        # Validate the return values
        logging.info("Verifying that the receive_command correctly processed the 'get_price' command")

        # Unpack the result for clearer assertions
        price, excel_path, html_path = result

        # Validate the return values match what we mocked
        assert price == "100.00", f"Expected price '100.00', got {price}"
        assert excel_path == "Data saved to Excel file at path.xlsx", f"Expected Excel path 'path.xlsx', got {excel_path}"
        assert html_path == "Data exported to HTML at path.html", f"Expected HTML path 'path.html', got {html_path}"

        logging.info("Test passed: Control layer processing correctly handles 'get_price'")


# Test 2: Price Retrieval
@pytest.mark.asyncio
async def test_price_retrieval():
    logging.info("Starting test: Price Retrieval from webpage")

    # Mock the `get_price_from_page` method to simulate price retrieval without browser interaction
    with patch('entity.PriceEntity.PriceEntity.get_price_from_page', return_value="100.00") as mock_price:
        price_control = PriceControl()

        # Call the `get_price` method
        result = await price_control.get_price("https://example.com/product")

        logging.info("Expected fetched price: '100.00'")
        assert "100.00" in result, f"Expected price '100.00', got {result}"
        logging.info("Test passed: Price retrieval successful and correct")

# Test 3: Data Logging to Excel
@pytest.mark.asyncio
async def test_data_logging_excel():
    logging.info("Starting test: Data Logging to Excel")

    # Mock the `get_price` method to avoid browser interaction
    with patch('control.PriceControl.PriceControl.get_price', new_callable=AsyncMock) as mock_get_price:
        # Set return value for `get_price` method
        mock_get_price.return_value = ("100.00", "Data saved to Excel file at path.xlsx", "Data exported to HTML file at path.html")

        # Mock the log_to_excel method to simulate Excel data logging
        with patch('entity.DataExportEntity.ExportUtils.log_to_excel', return_value="Data saved to Excel file at path.xlsx") as mock_excel:
            price_control = PriceControl()

            # Call the `get_price` method, which is now mocked
            _, excel_result, _ = await price_control.get_price("https://example.com/product")

            logging.info("Verifying Excel file creation and data logging")
            assert "path.xlsx" in excel_result, f"Expected Excel path 'path.xlsx', got {excel_result}"
            logging.info("Test passed: Data correctly logged to Excel")


# Test 4: Data Export to HTML
@pytest.mark.asyncio
async def test_data_logging_html():
    logging.info("Starting test: Data Export to HTML")

    # Mock the `get_price` method to avoid browser interaction
    with patch('control.PriceControl.PriceControl.get_price', new_callable=AsyncMock) as mock_get_price:
        # Set return value for `get_price` method
        mock_get_price.return_value = ("100.00", "Data saved to Excel file at path.xlsx", "Data exported to HTML file at path.html")

        # Mock the export_to_html method to simulate HTML export
        with patch('entity.DataExportEntity.ExportUtils.export_to_html', return_value="Data exported to HTML file at path.html") as mock_html:
            price_control = PriceControl()

            # Call the `get_price` method, which is now mocked
            _, _, html_result = await price_control.get_price("https://example.com/product")

            logging.info("Verifying HTML file creation and data export")
            assert "path.html" in html_result, f"Expected HTML path 'path.html', got {html_result}"
            logging.info("Test passed: Data correctly exported to HTML")
```

7. !start_monitoring_price <website>

Description

This test ensures that the BotControl.receive_command() method processes the !start_monitoring_price command correctly by initiating price monitoring at regular intervals for the specified website.

Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**

   This test ensures that BotControl.receive_command() handles the !start_monitoring_price command correctly, including proper URL parameter passing.

2. **Price Monitoring Initiation**

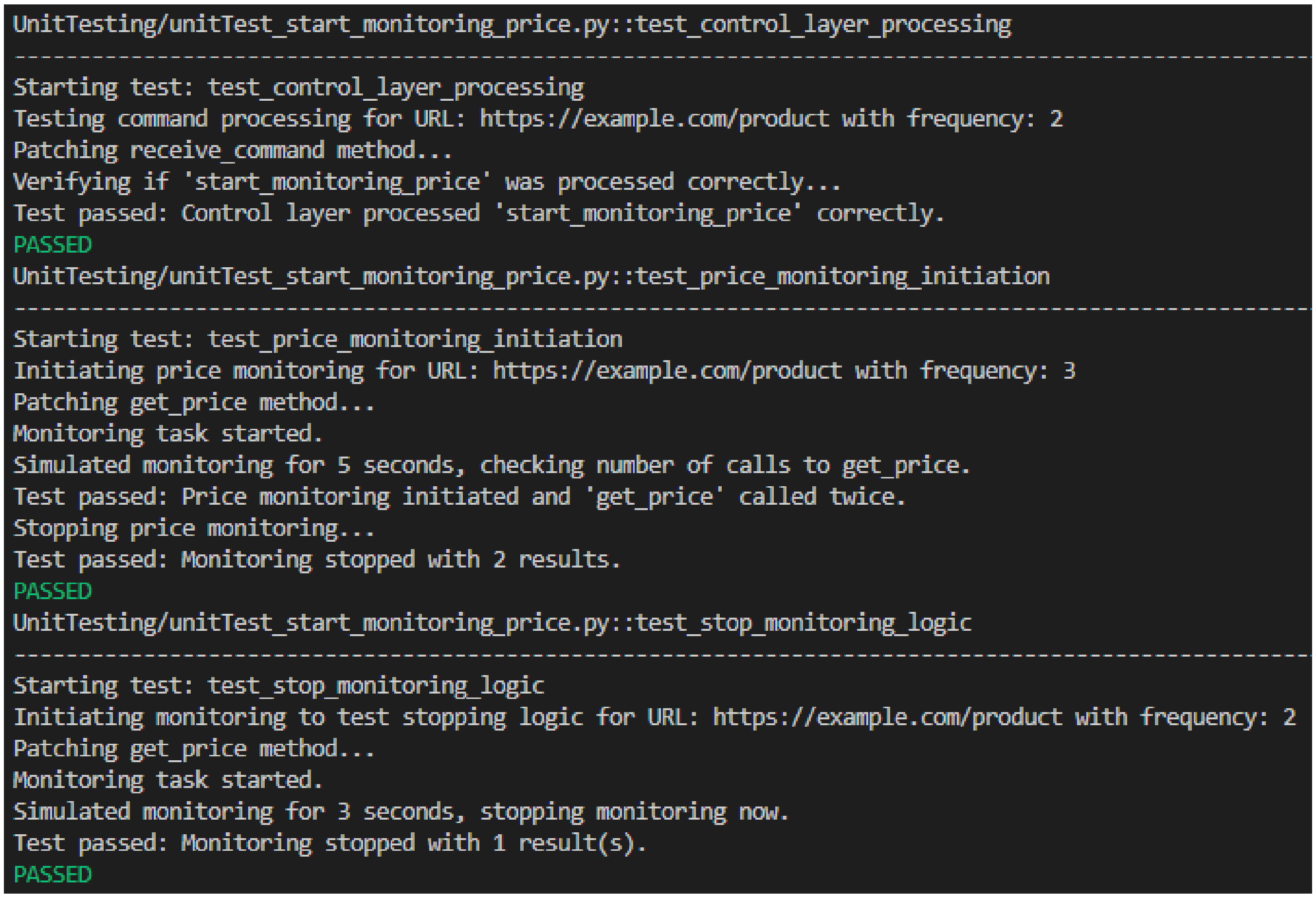
   This test ensures that price monitoring is initiated and repeated at regular intervals by calling the get_price() function.

3. **Stop Monitoring Logic**

   This test confirms that price monitoring can be stopped correctly and the final results are collected.

Test Data

- Command: "!start_monitoring_price"
- Test website: "http://example.com/product"
- Expected Output: "Price monitoring started"



```
UnitTesting/unitTest_start_monitoring_price.py::test_control_layer_processing
------------------------------------------------------------------------------------------------------
Starting test: test_control_layer_processing
Testing command processing for URL: https://example.com/product with frequency: 2
Patching receive_command method...
Verifying if 'start_monitoring_price' was processed correctly...
Test passed: Control layer processed 'start_monitoring_price' correctly.
PASSED
UnitTesting/unitTest_start_monitoring_price.py::test_price_monitoring_initiation
------------------------------------------------------------------------------------------------------
Starting test: test_price_monitoring_initiation
Initiating price monitoring for URL: https://example.com/product with frequency: 3
Patching get_price method...
Monitoring task started.
Simulated monitoring for 5 seconds, checking number of calls to get_price.
Test passed: Price monitoring initiated and 'get_price' called twice.
Stopping price monitoring...
Test passed: Monitoring stopped with 2 results.
PASSED
UnitTesting/unitTest_start_monitoring_price.py::test_stop_monitoring_logic
------------------------------------------------------------------------------------------------------
Starting test: test_stop_monitoring_logic
Initiating monitoring to test stopping logic for URL: https://example.com/product with frequency: 2
Patching get_price method...
Monitoring task started.
Simulated monitoring for 3 seconds, stopping monitoring now.
Test passed: Monitoring stopped with 1 result(s).
PASSED
```

8. !stop_monitoring_price <website>

Description

This test ensures that the BotControl.receive_command() method processes the !stop_monitoring_price command correctly by stopping the monitoring process and generating a final summary of the results.

Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**

   This test ensures that BotControl.receive_command() handles the !stop_monitoring_price command correctly.

2. **Stop Monitoring Logic**

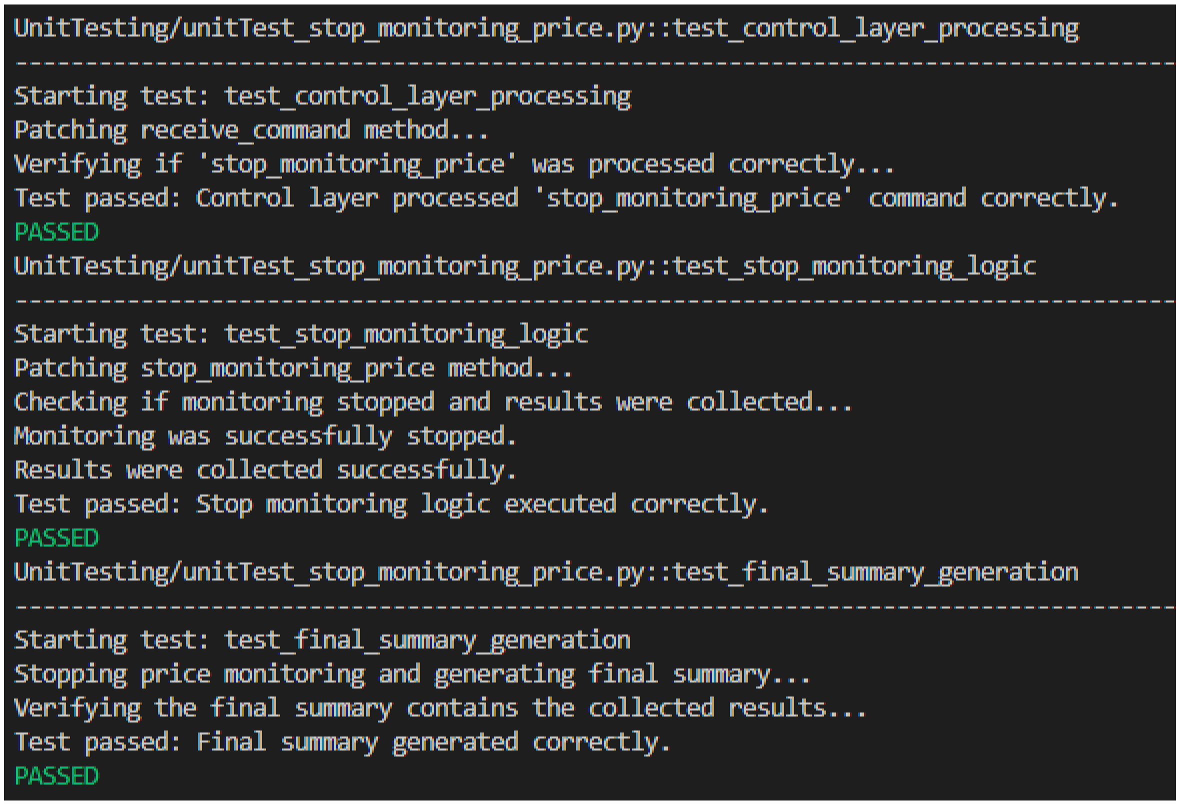
   This test ensures that the monitoring process is stopped and results are collected.

3. **Final Summary Generation**

   This test validates that a final summary of the price monitoring results is generated and returned.

Test Data

- Command: "!stop_monitoring_price"
- Test website: "http://example.com/product"
- Expected Output: "Price monitoring stopped"



```
UnitTesting/unitTest_stop_monitoring_price.py::test_control_layer_processing
------------------------------------------------------------------------------------
Starting test: test_control_layer_processing
Patching receive_command method...
Verifying if 'stop_monitoring_price' was processed correctly...
Test passed: Control layer processed 'stop_monitoring_price' command correctly.
PASSED
UnitTesting/unitTest_stop_monitoring_price.py::test_stop_monitoring_logic
------------------------------------------------------------------------------------
Starting test: test_stop_monitoring_logic
Patching stop_monitoring_price method...
Checking if monitoring stopped and results were collected...
Monitoring was successfully stopped.
Results were collected successfully.
Test passed: Stop monitoring logic executed correctly.
PASSED
UnitTesting/unitTest_stop_monitoring_price.py::test_final_summary_generation
------------------------------------------------------------------------------------
Starting test: test_final_summary_generation
Stopping price monitoring and generating final summary...
Verifying the final summary contains the collected results...
Test passed: Final summary generated correctly.
PASSED
```

9. !check_availability <website>

Description

This test ensures that the BotControl.receive_command() method processes the !check_availability command correctly by checking the availability of the specified service on the website and logging the results.

Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**

   This test ensures that BotControl.receive_command() handles the !check_availability command correctly.

2. **Availability Checking**

   This test ensures that the AvailabilityEntity.check_availability() function checks the availability of the specified service.

3. **Data Logging to Excel**

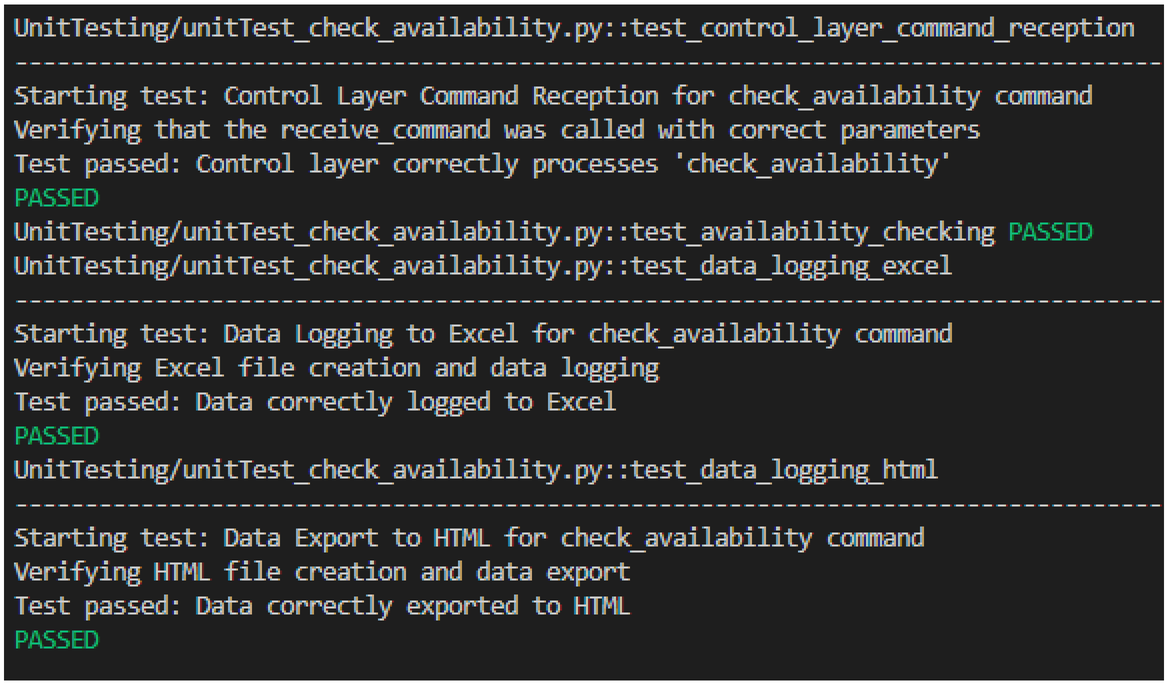
   This test verifies that the availability data is logged to an Excel file.

4. **Data Logging to HTML**

   This test ensures that the availability data is exported to an HTML file.

Test Data

- Command: "!check_availability"
- Test website: "http://example.com/reservation"
- Expected Output: "Availability confirmed"


```
UnitTesting/unitTest_check_availability.py::test_control_layer_command_reception
--------------------------------------------------------------------------------
Starting test: Control Layer Command Reception for check_availability command
Verifying that the receive_command was called with correct parameters
Test passed: Control layer correctly processes 'check_availability'
PASSED
UnitTesting/unitTest_check_availability.py::test_availability_checking PASSED
UnitTesting/unitTest_check_availability.py::test_data_logging_excel
--------------------------------------------------------------------------------
Starting test: Data Logging to Excel for check_availability command
Verifying Excel file creation and data logging
Test passed: Data correctly logged to Excel
PASSED
UnitTesting/unitTest_check_availability.py::test_data_logging_html
--------------------------------------------------------------------------------
Starting test: Data Export to HTML for check_availability command
Verifying HTML file creation and data export
Test passed: Data correctly exported to HTML
PASSED
```

10. !start_monitoring_availability <website>

Description

This test ensures that the BotControl.receive_command() method processes the !start_monitoring_availability command correctly by initiating service availability monitoring at regular intervals for the specified website.

Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**

   This test ensures that BotControl.receive_command() handles the !start_monitoring_availability command correctly.

2. **Availability Monitoring Initiation**

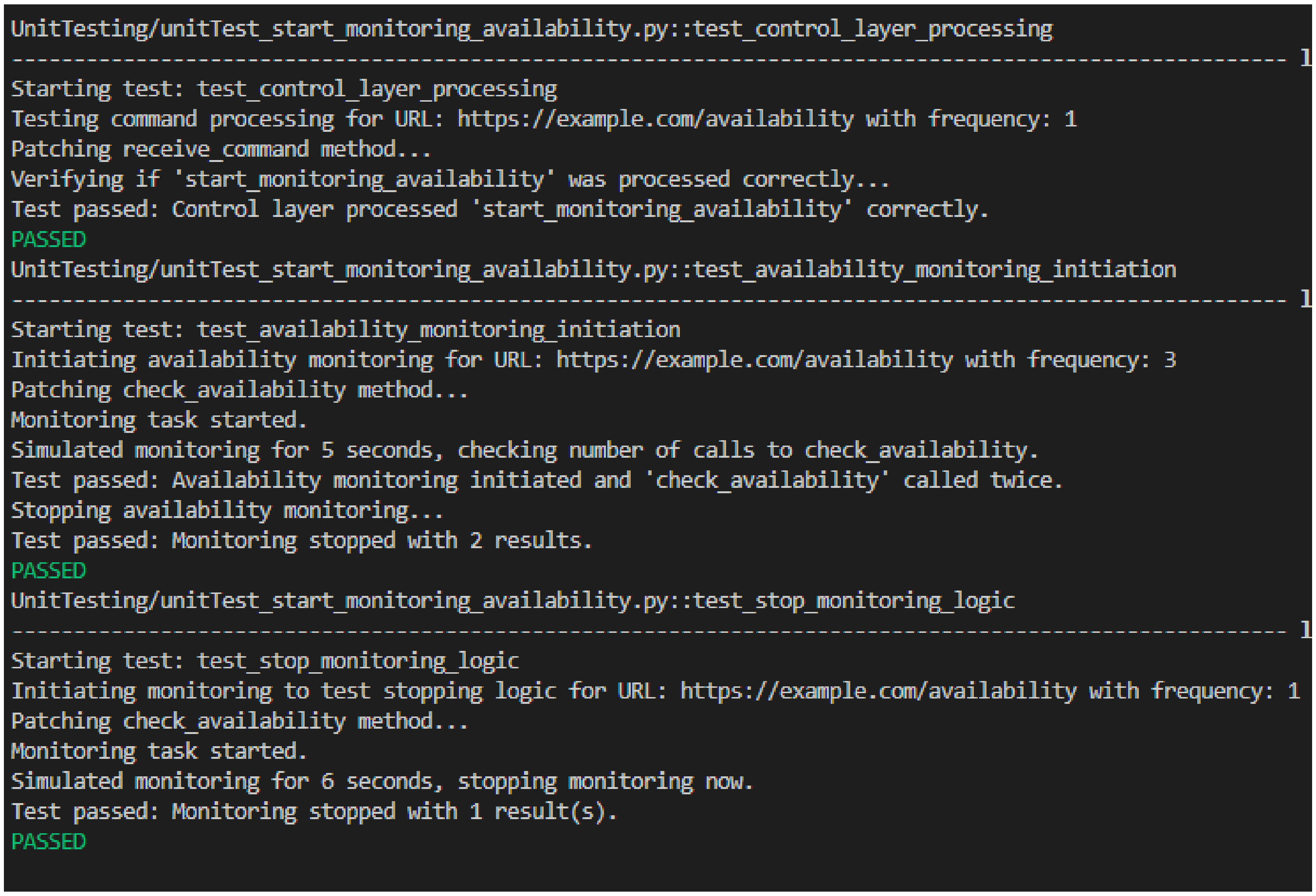
   This test ensures that service availability monitoring is initiated and repeated at regular intervals.

3. **Stop Monitoring Logic**

   This test confirms that availability monitoring can be stopped correctly and the final results are collected.

Test Data

- Command: "!start_monitoring_availability"
- Test website: "http://example.com/reservation"
- Expected Output: "Availability monitoring started"



```
UnitTesting/unitTest_start_monitoring_availability.py::test_control_layer_processing
----------------------------------------------------------------------------------------------- l
Starting test: test_control_layer_processing
Testing command processing for URL: https://example.com/availability with frequency: 1
Patching receive_command method...
Verifying if 'start_monitoring_availability' was processed correctly...
Test passed: Control layer processed 'start_monitoring_availability' correctly.
PASSED
UnitTesting/unitTest_start_monitoring_availability.py::test_availability_monitoring_initiation
----------------------------------------------------------------------------------------------- l
Starting test: test_availability_monitoring_initiation
Initiating availability monitoring for URL: https://example.com/availability with frequency: 3
Patching check_availability method...
Monitoring task started.
Simulated monitoring for 5 seconds, checking number of calls to check_availability.
Test passed: Availability monitoring initiated and 'check_availability' called twice.
Stopping availability monitoring...
Test passed: Monitoring stopped with 2 results.
PASSED
UnitTesting/unitTest_start_monitoring_availability.py::test_stop_monitoring_logic
----------------------------------------------------------------------------------------------- l
Starting test: test_stop_monitoring_logic
Initiating monitoring to test stopping logic for URL: https://example.com/availability with frequency: 1
Patching check_availability method...
Monitoring task started.
Simulated monitoring for 6 seconds, stopping monitoring now.
Test passed: Monitoring stopped with 1 result(s).
PASSED
```

11. !stop_monitoring_availability <website>

Description

This test ensures that the BotControl.receive_command() method processes the !stop_monitoring_availability command correctly by stopping the monitoring process and generating a final summary of the results.

Test Steps

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**

   This test ensures that BotControl.receive_command() handles the !stop_monitoring_availability command correctly.

2. **Stop Monitoring Logic**

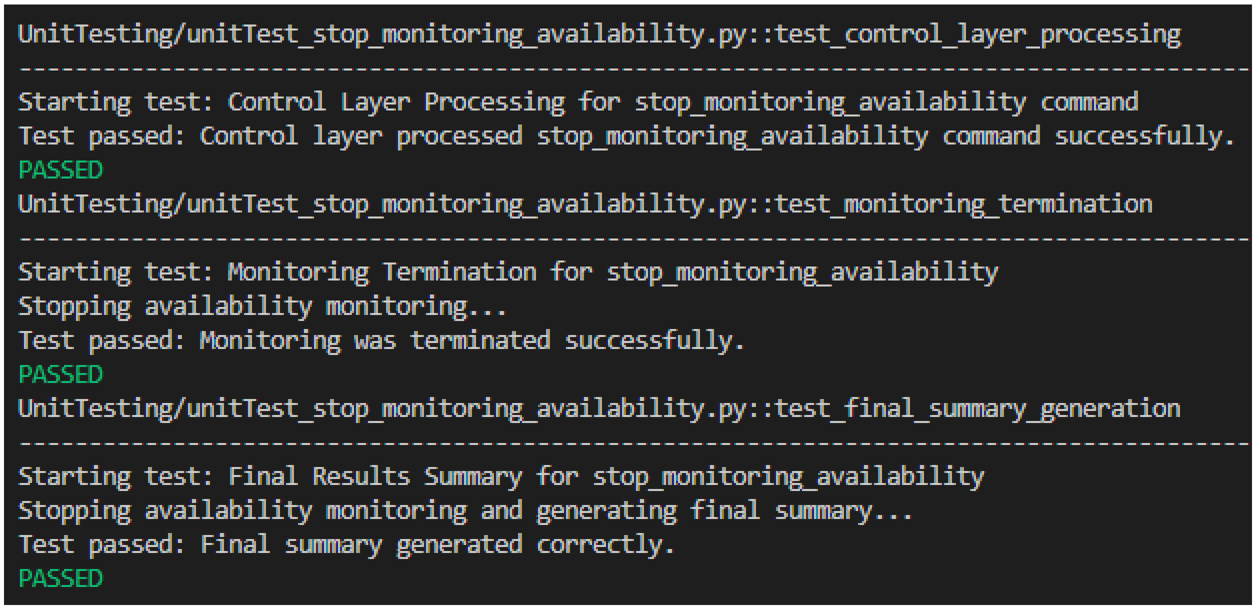
   This test ensures that the monitoring process is stopped and results are collected.

3. **Final Summary Generation**

   This test validates that a final summary of the availability monitoring results is generated and returned.

Test Data

- Command: "!stop_monitoring_availability"
- Test website: "http://example.com/reservation"
- Expected Output: "Availability monitoring stopped"



```
UnitTesting/unitTest_stop_monitoring_availability.py::test_control_layer_processing
------------------------------------------------------------------------------------
Starting test: Control Layer Processing for stop_monitoring_availability command
Test passed: Control layer processed stop_monitoring_availability command successfully.
PASSED
UnitTesting/unitTest_stop_monitoring_availability.py::test_monitoring_termination
------------------------------------------------------------------------------------
Starting test: Monitoring Termination for stop_monitoring_availability
Stopping availability monitoring...
Test passed: Monitoring was terminated successfully.
PASSED
UnitTesting/unitTest_stop_monitoring_availability.py::test_final_summary_generation
------------------------------------------------------------------------------------
Starting test: Final Results Summary for stop_monitoring_availability
Stopping availability monitoring and generating final summary...
Test passed: Final summary generated correctly.
PASSED
```

Conclusion

This project aimed to rigorously test various functionalities of the Discord bot, particularly focusing on control and entity layers across key use cases such as availability checking, price monitoring, and email handling. The test cases were structured to cover command reception, process execution, and result validation, ensuring that each component behaved as expected under both normal and boundary conditions.

Through the execution of the 34 unit tests, various issues were identified, such as ModuleNotFoundErrors and assertion failures, which were addressed promptly. This defect identification helped improve the robustness of the bot, highlighting the importance of unit testing in developing reliable software systems.

The structured approach, using pytest and mocking frameworks, allowed the testing process to be efficient and scalable. While limitations in testing the boundary layers, particularly in simulating real-world Discord interactions, were acknowledged, the tests effectively ensured that the core logic and functionalities performed correctly.

In conclusion, the thorough testing and defect resolution process not only enhanced the bot’s performance but also provided valuable insights into software testing strategies, emphasizing the need for comprehensive test coverage in ensuring product quality.