**Discord Bot Automation Assistant**

**Discord Bot Automation Assistant Test Plan**

**Oguz Kaan Yildirim**

**307637**

# Table Of Contents

[Table Of Contents 2](#_Toc178714678)

[INTRDOCUTION 7](#_Toc178714679)

[TEST PLAN OVERVIEW 8](#_Toc178714680)

[TEST CASES 10](#_Toc178714681)

[Test Case 0: test\_init.py 10](#_Toc178714682)

[Tools and Technologies 10](#_Toc178714683)

[Purpose and Setup 10](#_Toc178714684)

[Implementation Details 10](#_Toc178714685)

[How It Works 11](#_Toc178714686)

[Source Code 12](#_Toc178714687)

[Test Case 1: Add Account 14](#_Toc178714688)

[Description 14](#_Toc178714689)

[Steps 14](#_Toc178714690)

[Test Data 15](#_Toc178714691)

[Entity Tests Code 15](#_Toc178714692)

[Entity Test Output 16](#_Toc178714693)

[Control Tests Code 16](#_Toc178714694)

[Control Test output 17](#_Toc178714695)

[Test Case 2: Delete Account 18](#_Toc178714696)

[Description 18](#_Toc178714697)

[Steps: 18](#_Toc178714698)

[Test Data: 19](#_Toc178714699)

[Entity Tests Code: 19](#_Toc178714700)

[Entity Test Output 19](#_Toc178714701)

[Control Tests Code 20](#_Toc178714702)

[Control Test Output 20](#_Toc178714703)

[Test Case 3: Fetch All Accounts 21](#_Toc178714704)

[Description 21](#_Toc178714705)

[Steps 21](#_Toc178714706)

[Test Data: 22](#_Toc178714707)

[Entity Tests Code 22](#_Toc178714708)

[Entity Test output 23](#_Toc178714709)

[Control Tests Code 23](#_Toc178714710)

[Control Test Output 24](#_Toc178714711)

[Test Case 4: Fetch Account by Website 25](#_Toc178714712)

[Description 25](#_Toc178714713)

[Steps 25](#_Toc178714714)

[Test Data 26](#_Toc178714715)

[Entity Tests Code 26](#_Toc178714716)

[Entity Test output 26](#_Toc178714717)

[Control Tests Code 27](#_Toc178714718)

[Control Test Output 27](#_Toc178714719)

[Test Case 5: Receive Email 28](#_Toc178714720)

[Description 28](#_Toc178714721)

[Steps 28](#_Toc178714722)

[Test Data 29](#_Toc178714723)

[Utility Tests Code 29](#_Toc178714724)

[Utility Test Output 29](#_Toc178714725)

[Control Tests Code: 30](#_Toc178714726)

[Control Test Output 30](#_Toc178714727)

[Test Case 6: Export Data to HTML and Excel 31](#_Toc178714728)

[Description 31](#_Toc178714729)

[Steps 31](#_Toc178714730)

[Test Data 32](#_Toc178714731)

[Utility Tests Code 32](#_Toc178714732)

[Utility Test Output 33](#_Toc178714733)

[Test Case 7: Launch Browser 34](#_Toc178714734)

[Description 34](#_Toc178714735)

[Steps 34](#_Toc178714736)

[Test Data 35](#_Toc178714737)

[Entity Tests Code 35](#_Toc178714738)

[Entity Test output 35](#_Toc178714739)

[Control Tests Code 36](#_Toc178714740)

[Control Test Output 37](#_Toc178714741)

[Test Case 8: Close Browser 38](#_Toc178714742)

[Description 38](#_Toc178714743)

[Steps 38](#_Toc178714744)

[Test Data 39](#_Toc178714745)

[Entity and Control Tests Code 39](#_Toc178714746)

[Entity and Control Test output 40](#_Toc178714747)

[Test Case 9: Navigate to Website 41](#_Toc178714748)

[Description 41](#_Toc178714749)

[Steps 41](#_Toc178714750)

[Test Data 42](#_Toc178714751)

[Entity and Control Tests Code 42](#_Toc178714752)

[Entity and Control Test output 43](#_Toc178714753)

[Test Case 10: Login 45](#_Toc178714754)

[Description 45](#_Toc178714755)

[Steps 45](#_Toc178714756)

[Test Data 46](#_Toc178714757)

[Entity and Control Tests Code 46](#_Toc178714758)

[Entity and Control Test output 47](#_Toc178714759)

[Test Case 11: Get Price 49](#_Toc178714760)

[Description 49](#_Toc178714761)

[Steps 49](#_Toc178714762)

[Test Data 50](#_Toc178714763)

[Entity and Control Tests Code 50](#_Toc178714764)

[Entity and Control Test output 51](#_Toc178714765)

[Test Case 12: Check Availability 52](#_Toc178714766)

[Description 52](#_Toc178714767)

[Steps 52](#_Toc178714768)

[Test Data 53](#_Toc178714769)

[Entity and Control Tests Code 53](#_Toc178714770)

[Entity and Control Test output 54](#_Toc178714771)

[Test Case 13: Start Monitoring Availability 55](#_Toc178714772)

[Description 55](#_Toc178714773)

[Steps 55](#_Toc178714774)

[Test Data 56](#_Toc178714775)

[Entity and Control Tests Code 56](#_Toc178714776)

[Entity and Control Test output 57](#_Toc178714777)

[Test Case 14: Stop Monitoring Availability 58](#_Toc178714778)

[Description 58](#_Toc178714779)

[Steps 58](#_Toc178714780)

[Test Data 59](#_Toc178714781)

[Entity and Control Tests Code 59](#_Toc178714782)

[Entity and Control Test output 59](#_Toc178714783)

[Test Case 15: Start Monitoring Price 60](#_Toc178714784)

[Description 60](#_Toc178714785)

[Steps 60](#_Toc178714786)

[Test Data 61](#_Toc178714787)

[Entity and Control Tests Code 61](#_Toc178714788)

[Entity and Control Test output 62](#_Toc178714789)

[Test Case 16: Stop Monitoring Price 63](#_Toc178714790)

[Description 63](#_Toc178714791)

[Steps 63](#_Toc178714792)

[Test Data 64](#_Toc178714793)

[Entity and Control Tests Code 64](#_Toc178714794)

[Entity and Control Test output 64](#_Toc178714795)

[Test Case 17: Stop Bot 66](#_Toc178714796)

[Description 66](#_Toc178714797)

[Steps 66](#_Toc178714798)

[Test Data 67](#_Toc178714799)

[Entity and Control Tests Code 67](#_Toc178714800)

[Entity and Control Test output 67](#_Toc178714801)

[Test Case 18: Project Help 68](#_Toc178714802)

[Description 68](#_Toc178714803)

[Steps 68](#_Toc178714804)

[Test Data 69](#_Toc178714805)

[Entity and Control Tests Code 69](#_Toc178714806)

[Entity and Control Test output 69](#_Toc178714807)

[Conclusion 70](#_Toc178714808)

# 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.

# TEST CASES

## Test Case 0: 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.

A screen shot of a computer program

Description automatically generated

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.

A computer screen with text

Description automatically generated

## Test Case 1: Project Help

### Description

This test ensures that the BotControl.receive\_command() method processes the project\_help command correctly by returning the appropriate help message.

## Test Steps

1. **Mock Command**: Use patch to mock the BotControl.receive\_command method, simulating its behavior.
2. **Expected Message**: Define the expected output of the project\_help command as a string listing the available commands.
3. **Invoke Command**: Create an instance of BotControl and simulate the project\_help command being passed to the receive\_command method.
4. **Assert test**: Check if the result matches the expected message.

## Test Data

* **Command**: "project\_help"
* **Expected Output**: "Here are the available commands:..."

## Test Case 6: Export Data to HTML and Excel

### Description

This test case validates the functionality of exporting data to both HTML and Excel files through the system's utility layer. It ensures that both successful and failed export scenarios are correctly handled, logging outcomes and identifying errors when encountered.

### Steps

1. **Setup and Mock Initialization:**
   * Initialize test environments with pytest fixtures, mocking necessary components for file path verification, directory creation, and file writing.
   * Use fixtures to simulate file existence and successful or failed export operations for both HTML and Excel formats.
2. **Positive HTML Export Test:**
   * Mock successful HTML file writing and validate that the export\_to\_html method correctly saves the file and returns a success message.
3. **Positive Excel Export Test:**
   * Simulate a successful Excel file export by mocking the DataFrame writing operation and validate that the log\_to\_excel method returns a success message indicating that the data was saved.
4. **Negative HTML Export Test:**
   * Simulate an error during HTML file writing (e.g., raising an exception) and ensure that the method raises the expected error and logs the failure correctly.
5. **Negative Excel Export Test:**
   * Simulate an error during Excel file writing and ensure that the method raises the expected error and logs the failure correctly.
6. **Execution and Validation:**
   * Execute the export methods for both HTML and Excel cases across success and failure scenarios. Validate the responses against expected outcomes using assertions.
7. **Logging and Outcome Verification:**
   * Capture detailed logs of the test execution, providing traceability for expected and actual results and aiding in debugging.

### Test Data

* **Valid Data:**
  + Command: "test\_command"
  + URL: "<http://example.com>"
  + Result: "Success"
* **Invalid Data:**
  + File cannot be written due to an error in file access or write permissions.

### Utility Tests Code

@pytest.fixture

def setup\_mocked\_paths(self, mocker):

mocker.patch('os.path.exists', return\_value=False)

mocker.patch('os.makedirs') # Mock directory creation

mocker.patch('pandas.DataFrame.to\_excel') # Mock the Excel export method

mocker.patch('builtins.open', mocker.mock\_open()) # Mock open for HTML writing

def test\_positive\_html\_export(self, base\_test\_case, setup\_mocked\_paths):

result = base\_test\_case.export\_utils.export\_to\_html("test\_command", "http://example.com", "Success")

assert "HTML file saved and updated" in result

print("Positive HTML Export Test Passed")

def test\_positive\_excel\_export(self, base\_test\_case, setup\_mocked\_paths):

with patch('pandas.read\_excel', return\_value=pd.DataFrame(columns=["Timestamp", "Command", "URL", "Result", "Entered Date", "Entered Time"])):

result = base\_test\_case.export\_utils.log\_to\_excel("test\_command", "http://example.com", "Success")

assert "Data saved to Excel file" in result

print("Positive Excel Export Test Passed")

def test\_negative\_html\_export(self, base\_test\_case, setup\_mocked\_paths):

with patch('builtins.open', side\_effect=Exception("Failed to write HTML")):

try:

result = base\_test\_case.export\_utils.export\_to\_html("test\_command", "http://example.com", "Success")

except Exception as e:

assert str(e) == "Failed to write HTML"

print("Negative HTML Export Test Passed with Expected Exception")

def test\_negative\_excel\_export(self, base\_test\_case, setup\_mocked\_paths):

with patch('pandas.DataFrame.to\_excel', side\_effect=Exception("Failed to write Excel")):

try:

result = base\_test\_case.export\_utils.log\_to\_excel("test\_command", "http://example.com", "Success")

except Exception as e:

assert str(e) == "Failed to write Excel"

print("Negative Excel Export Test Passed with Expected Exception")

### Utility Test Output

Starting test: test\_positive\_html\_export

Mocks for os.path, os.makedirs, pandas.to\_excel, and open set up successfully.

Result: HTML file saved and updated at ExportedFiles\htmlFiles\test\_command.html.

Test positive HTML export passed successfully.

Finished test: test\_positive\_html\_export

------------------------------------------------------

Starting test: test\_positive\_excel\_export

Mocks for os.path, os.makedirs, pandas.to\_excel, and open set up successfully.

Result: Data saved to Excel file at ExportedFiles\excelFiles\test\_command.xlsx.

Test positive Excel export passed successfully.

Finished test: test\_positive\_excel\_export

------------------------------------------------------

Starting test: test\_negative\_html\_export

Mocks for os.path, os.makedirs, pandas.to\_excel, and open set up successfully.

Expected exception caught: Failed to write HTML

Test negative HTML export passed with expected exception.

Finished test: test\_negative\_html\_export

------------------------------------------------------

Starting test: test\_negative\_excel\_export

Mocks for os.path, os.makedirs, pandas.to\_excel, and open set up successfully.

Expected exception caught: Failed to write Excel

Test negative Excel export passed with expected exception.

Finished test: test\_negative\_excel\_export

------------------------------------------------------

## Test Case 8: 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.

### Entity and Control Tests Code

*These objects are tested together within the same test case(s)*

async def test\_close\_browser\_success(base\_test\_case):

    with patch('entity.BrowserEntity.BrowserEntity.close\_browser') as mock\_close:

        mock\_close.return\_value = "Browser closed."

        expected\_entity\_result = "Browser closed."

        expected\_control\_result = "Control Object Result: Browser closed."

        result = await base\_test\_case.browser\_control.receive\_command("close\_browser")

        assert mock\_close.return\_value == expected\_entity\_result, "Entity layer assertion failed."

        assert result == expected\_control\_result, "Control layer assertion failed."

async def test\_close\_browser\_not\_open(base\_test\_case):

    with patch('entity.BrowserEntity.BrowserEntity.close\_browser') as mock\_close:

        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."

        result = await base\_test\_case.browser\_control.receive\_command("close\_browser")

        assert mock\_close.return\_value == expected\_entity\_result, "Entity layer assertion failed."

        assert result == expected\_control\_result, "Control layer assertion failed."

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:

        expected\_result = "Control Layer Exception: Unexpected error"

        result = await base\_test\_case.browser\_control.receive\_command("close\_browser")

        assert result == expected\_result, "Control layer failed to handle or report the error correctly."

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:

        internal\_error\_message = "BrowserEntity\_Failed to close browser: Internal error"

        expected\_control\_result = f"Control Layer Exception: {internal\_error\_message}"

        result = await base\_test\_case.browser\_control.receive\_command("close\_browser")

        assert result == expected\_control\_result, "Control layer failed to report entity error correctly."

### Entity and Control Test output

*These objects are tested together within the same test case(s)*

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

------------------------------------------------------

## Test Case 9: 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"

### Entity and Control Tests Code

These objects are tested together within the same test case(s)

async def test\_navigate\_to\_website\_success(base\_test\_case):

    with patch('entity.BrowserEntity.BrowserEntity.navigate\_to\_website') as mock\_navigate:

        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}"

        result = await base\_test\_case.browser\_control.receive\_command("navigate\_to\_website", site=url)

        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:

        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."

        result = await base\_test\_case.browser\_control.receive\_command("navigate\_to\_website", site=invalid\_site)

        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:

        url = "https://example.com"

        expected\_control\_result = "Control Layer Exception: Failed to navigate"

        result = await base\_test\_case.browser\_control.receive\_command("navigate\_to\_website", site=url)

        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):

    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:

        url = "https://example.com"

        mock\_navigate.return\_value = f"Navigated to {url}"

        expected\_control\_result = f"Control Object Result: Navigated to {url}"

        result = await base\_test\_case.browser\_control.receive\_command("navigate\_to\_website", site=url)

        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):

    with patch('control.BrowserControl.BrowserControl.receive\_command', side\_effect=Exception("Control Layer Failed")) as mock\_control:

        url = "https://example.com"

        expected\_control\_result = "Control Layer Exception: Control Layer Failed"

        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)}"

        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.")

### Entity and Control Test output

These objects are tested together within the same test case(s)

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

## Test Case 10: 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.

### Entity and Control Tests Code

These objects are tested together within the same test case(s)

async def test\_login\_success(base\_test\_case):

with patch('entity.BrowserEntity.BrowserEntity.login') as mock\_login:

with patch('control.AccountControl.AccountControl.fetch\_account\_by\_website') as mock\_fetch\_account:

mock\_login.return\_value = "Logged in to http://example.com successfully with username: sample\_username"

mock\_fetch\_account.return\_value = ("sample\_username", "sample\_password")

result = await base\_test\_case.browser\_control.receive\_command("login", site="example.com")

assert mock\_login.return\_value == "Logged in to http://example.com successfully with username: sample\_username"

assert result == f"Control Object Result: Logged in to http://example.com successfully with username: sample\_username"

async def test\_login\_no\_account(base\_test\_case):

with patch('control.AccountControl.AccountControl.fetch\_account\_by\_website') as mock\_fetch\_account:

mock\_fetch\_account.return\_value = None

result = await base\_test\_case.browser\_control.receive\_command("login", site="example.com")

assert result == "No account found for example.com"

async def test\_login\_entity\_layer\_failure(base\_test\_case):

with patch('entity.BrowserEntity.BrowserEntity.login') as mock\_login:

with patch('control.AccountControl.AccountControl.fetch\_account\_by\_website') as mock\_fetch\_account:

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")

result = await base\_test\_case.browser\_control.receive\_command("login", site="example.com")

assert result == "Control Layer Exception: BrowserEntity\_Failed to log in to http://example.com: Internal error"

async def test\_login\_control\_layer\_failure(base\_test\_case):

with patch('control.AccountControl.AccountControl.fetch\_account\_by\_website') as mock\_fetch\_account:

mock\_fetch\_account.side\_effect = Exception("Control layer failure during account fetch.")

result = await base\_test\_case.browser\_control.receive\_command("login", site="example.com")

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

async def test\_login\_invalid\_url(base\_test\_case):

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:

mock\_fetch\_account.return\_value = ("sample\_username", "sample\_password")

mock\_get\_selectors.return\_value = {'url': None}

result = await base\_test\_case.browser\_control.receive\_command("login", site="example")

assert result == "URL for example not found."

### Entity and Control Test output

These objects are tested together within the same test case(s)

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 Expected: Control Layer Exception: BrowserEntity\_Failed to log in to http://example.com: Internal error

Control 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

## Test Case 11: 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.

### Entity and Control Tests Code

These objects are tested together within the same test case(s)

async def test\_get\_price\_success(base\_test\_case):

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"

result = await base\_test\_case.price\_control.receive\_command("get\_price", url)

assert mock\_get\_price.return\_value == "$199.99"

assert result == "$199.99"

async def test\_get\_price\_invalid\_url(base\_test\_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"

result = await base\_test\_case.price\_control.receive\_command("get\_price", invalid\_url)

assert result == "Error fetching price: Invalid URL"

async def test\_get\_price\_failure\_entity(base\_test\_case):

with patch('entity.PriceEntity.PriceEntity.get\_price\_from\_page', side\_effect=Exception("Failed to fetch price")):

url = "https://example.com/product"

result = await base\_test\_case.price\_control.receive\_command("get\_price", url)

assert result == "Failed to fetch price: Failed to fetch price"

async def test\_get\_price\_failure\_control(base\_test\_case):

with patch('control.PriceControl.PriceControl.receive\_command', side\_effect=Exception("Control Layer Failed")):

url = "https://example.com/product"

try:

result = await base\_test\_case.price\_control.receive\_command("get\_price", url)

except Exception as e:

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

assert result == "Control Layer Exception: Control Layer Failed"

### Entity and Control Test output

These objects are tested together within the same test case(s)

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

## Test Case 12: 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.

### Entity and Control Tests Code

These objects are tested together within the same test case(s)

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 = "Selected or default date current date is available for booking."

result = await base\_test\_case.availability\_control.receive\_command("check\_availability", url)

assert result == "Checked availability: Selected or default date current date is available for booking."

async def test\_check\_availability\_failure\_entity(base\_test\_case):

with patch('entity.AvailabilityEntity.AvailabilityEntity.check\_availability', side\_effect=Exception("Failed to check availability")):

url = "https://example.com"

result = await base\_test\_case.availability\_control.receive\_command("check\_availability", url)

assert result == "Failed to check availability: Failed to check availability"

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."

result = await base\_test\_case.availability\_control.receive\_command("check\_availability", url)

assert result == "Checked availability: No availability for the selected date."

async def test\_check\_availability\_failure\_control(base\_test\_case):

with patch('control.AvailabilityControl.AvailabilityControl.receive\_command', side\_effect=Exception("Control Layer Failed")):

url = "https://example.com"

try:

result = await base\_test\_case.availability\_control.receive\_command("check\_availability", url)

except Exception as e:

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

assert result == "Control Layer Exception: Control Layer Failed"

### Entity and Control Test output

These objects are tested together within the same test case(s)

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

## Test Case 13: 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.

### Entity and Control Tests Code

These objects are tested together within the same test case(s)

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!"

]

actual\_control\_result = await run\_monitoring\_loop(

base\_test\_case.availability\_control,

base\_test\_case.availability\_control.check\_availability,

url,

"2024-10-01",

1

)

assert actual\_control\_result == expected\_control\_result

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!"

]

actual\_control\_result = await run\_monitoring\_loop(

base\_test\_case.availability\_control,

base\_test\_case.availability\_control.check\_availability,

url,

"2024-10-01",

1

)

assert actual\_control\_result == expected\_control\_result

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)}"

assert result == expected\_control\_result

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)

assert result == expected\_control\_result

### Entity and Control Test output

These objects are tested together within the same test case(s)

Starting test: test\_start\_monitoring\_availability\_success

Monitoring Iteration: Checked availability: Selected date is available for booking.

Control Expected: ['Checked availability: Selected date is available.', 'Monitoring stopped successfully!']

Control Received: ['Checked availability: Selected date is available.', '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

## Test Case 14: 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.

### Entity and Control Tests Code

These objects are tested together within the same test case(s)

async def test\_stop\_monitoring\_availability\_success(base\_test\_case):

base\_test\_case.availability\_control.is\_monitoring = True

base\_test\_case.availability\_control.results = ["Checked availability: Selected date is available for booking."]

expected\_control\_result\_contains = "Monitoring stopped successfully!"

result = base\_test\_case.availability\_control.stop\_monitoring\_availability()

assert expected\_control\_result\_contains in result

async def test\_stop\_monitoring\_availability\_no\_active\_session(base\_test\_case):

base\_test\_case.availability\_control.is\_monitoring = False

expected\_control\_result = "There was no active availability monitoring session. Nothing to stop."

result = base\_test\_case.availability\_control.stop\_monitoring\_availability()

assert result == expected\_control\_result

### Entity and Control Test output

These objects are tested together within the same test case(s)

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

## Test Case 15: 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.

### Entity and Control Tests Code

These objects are tested together within the same test case(s)

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:

url = "https://example.com/product"

expected\_result = "Starting price monitoring. Current price: 100 USD"

with patch('asyncio.sleep', side\_effect=KeyboardInterrupt):

try:

base\_test\_case.price\_control.is\_monitoring = False

result = await base\_test\_case.price\_control.receive\_command("start\_monitoring\_price", url, 1)

except KeyboardInterrupt:

base\_test\_case.price\_control.is\_monitoring = False

assert expected\_result in base\_test\_case.price\_control.results[0]

async def test\_start\_monitoring\_price\_already\_running(base\_test\_case):

base\_test\_case.price\_control.is\_monitoring = True

expected\_result = "Already monitoring prices."

result = await base\_test\_case.price\_control.receive\_command("start\_monitoring\_price", "https://example.com/product", 1)

assert result == expected\_result

async def test\_start\_monitoring\_price\_failure\_in\_entity(base\_test\_case):

with patch('entity.PriceEntity.PriceEntity.get\_price\_from\_page', side\_effect=Exception("Error fetching price")) as mock\_get\_price:

url = "https://example.com/product"

expected\_result = "Starting price monitoring. Current price: Failed to fetch price: Error fetching price"

with patch('asyncio.sleep', side\_effect=KeyboardInterrupt):

try:

base\_test\_case.price\_control.is\_monitoring = False

await base\_test\_case.price\_control.receive\_command("start\_monitoring\_price", url, 1)

except KeyboardInterrupt:

base\_test\_case.price\_control.is\_monitoring = False

assert expected\_result in base\_test\_case.price\_control.results[-1]

### Entity and Control Test output

These objects are tested together within the same test case(s)

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

## Test Case 16: 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.

### Entity and Control Tests Code

These objects are tested together within the same test case(s)

async def test\_stop\_monitoring\_price\_success(base\_test\_case):

base\_test\_case.price\_control.is\_monitoring = True

base\_test\_case.price\_control.results = ["Price went up!", "Price went down!"]

expected\_result = "Results for price monitoring:\nPrice went up!\nPrice went down!\n\nPrice monitoring stopped successfully!"

result = base\_test\_case.price\_control.stop\_monitoring\_price()

assert result == expected\_result

async def test\_stop\_monitoring\_price\_not\_active(base\_test\_case):

base\_test\_case.price\_control.is\_monitoring = False

expected\_result = "There was no active price monitoring session. Nothing to stop."

result = base\_test\_case.price\_control.stop\_monitoring\_price()

assert result == expected\_result

### Entity and Control Test output

These objects are tested together within the same test case(s)

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

## 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.