**Discord Bot Automation Assistant**

**Discord Bot Automation Assistant Test Plan**

**Oguz Kaan Yildirim**

**307637**

# Table Of Contents

[Table Of Contents 2](#_Toc178631100)

[INTRDOCUTION 6](#_Toc178631101)

[TEST PLAN OVERVIEW 7](#_Toc178631102)

[TEST CASES 9](#_Toc178631103)

[Test Case 0: test\_init.py 9](#_Toc178631104)

[Tools and Technologies 9](#_Toc178631105)

[Purpose and Setup 9](#_Toc178631106)

[Implementation Details 9](#_Toc178631107)

[How It Works 10](#_Toc178631108)

[Source Code 11](#_Toc178631109)

[Test Case 1: Add Account 13](#_Toc178631110)

[Description 13](#_Toc178631111)

[Steps 13](#_Toc178631112)

[Test Data 14](#_Toc178631113)

[Entity Tests Code: 14](#_Toc178631114)

[Entity Test Output: 15](#_Toc178631115)

[Control Tests Code: 15](#_Toc178631116)

[Control Test output 16](#_Toc178631117)

[Test Case 2: Delete Account 17](#_Toc178631118)

[Description 17](#_Toc178631119)

[Steps: 17](#_Toc178631120)

[Test Data: 18](#_Toc178631121)

[Entity Tests Code: 18](#_Toc178631122)

[Entity Test Output: 19](#_Toc178631123)

[Control Tests Code: 19](#_Toc178631124)

[Control Test Output: 20](#_Toc178631125)

[Test Case 3: Fetch All Accounts 21](#_Toc178631126)

[Description 21](#_Toc178631127)

[Steps 21](#_Toc178631128)

[Test Data 22](#_Toc178631129)

[Test Scenarios, Expected Outcomes, Actual Outcomes 22](#_Toc178631130)

[Source Code 23](#_Toc178631131)

[Test Case 4: Fetch Account by Website 23](#_Toc178631132)

[Description 23](#_Toc178631133)

[Steps 23](#_Toc178631134)

[Test Data 24](#_Toc178631135)

[Test Scenarios, Expected Outcomes, Actual Outcomes 24](#_Toc178631136)

[Source Code 25](#_Toc178631137)

[Test Case 5: Launch Browser 25](#_Toc178631138)

[Description 25](#_Toc178631139)

[Steps 25](#_Toc178631140)

[Test Data 26](#_Toc178631141)

[Test Scenarios, Expected Outcomes, Actual Outcomes 26](#_Toc178631142)

[Source Code 27](#_Toc178631143)

[Test Case 6: Close Browser 27](#_Toc178631144)

[Description 27](#_Toc178631145)

[Steps 27](#_Toc178631146)

[Test Data 28](#_Toc178631147)

[Test Scenarios, Expected Outcomes, Actual Outcomes 28](#_Toc178631148)

[Source Code 29](#_Toc178631149)

[Test Case 7 : Navigate to Website 29](#_Toc178631150)

[Description 29](#_Toc178631151)

[Steps 29](#_Toc178631152)

[Test Data 30](#_Toc178631153)

[Test Scenarios, Expected Outcomes, Actual Outcomes 30](#_Toc178631154)

[Source Code 31](#_Toc178631155)

[Test Case 8: Login 32](#_Toc178631156)

[Description 32](#_Toc178631157)

[Steps 32](#_Toc178631158)

[Test Data 33](#_Toc178631159)

[Test Scenarios, Expected Outcomes, and Actual Outcomes 33](#_Toc178631160)

[Source Code 34](#_Toc178631161)

[Test Case 9: Get Price 35](#_Toc178631162)

[Description 35](#_Toc178631163)

[Steps 35](#_Toc178631164)

[Test Data 36](#_Toc178631165)

[Test Scenarios, Expected Outcomes, Actual Outcomes 36](#_Toc178631166)

[Source Code 37](#_Toc178631167)

[Test Case 10: Check Availability 37](#_Toc178631168)

[Description 37](#_Toc178631169)

[Steps 37](#_Toc178631170)

[Test Data 38](#_Toc178631171)

[Test Scenarios, Expected and Actual Outcomes 38](#_Toc178631172)

[Source Code 39](#_Toc178631173)

[Test Case 11: Start Monitoring Availability 39](#_Toc178631174)

[Description 39](#_Toc178631175)

[Steps 39](#_Toc178631176)

[Test Data 40](#_Toc178631177)

[Test Scenarios, Expected Outcomes, and Actual Outcomes 40](#_Toc178631178)

[Source Code 41](#_Toc178631179)

[Test Case 12: Stop Monitoring Availability 41](#_Toc178631180)

[Description 41](#_Toc178631181)

[Steps 41](#_Toc178631182)

[Test Data 42](#_Toc178631183)

[Test Scenarios, Expected Outcomes, Actual Outcomes 42](#_Toc178631184)

[Source Code 42](#_Toc178631185)

[Test Case 13: Start Monitoring Price 43](#_Toc178631186)

[Description 43](#_Toc178631187)

[Steps 43](#_Toc178631188)

[Test Data 44](#_Toc178631189)

[Test Scenarios, Expected Outcomes, Actual Outcomes 44](#_Toc178631190)

[Source Code 45](#_Toc178631191)

[Test Case 14: Stop Monitoring Price 45](#_Toc178631192)

[Description 45](#_Toc178631193)

[Steps 45](#_Toc178631194)

[Test Data 46](#_Toc178631195)

[Test Scenarios, Expected Outcomes, Actual Outcomes 46](#_Toc178631196)

[Source Code 47](#_Toc178631197)

[Test Case 15: Stop Bot 47](#_Toc178631198)

[Description 47](#_Toc178631199)

[Steps 47](#_Toc178631200)

[Test Data 48](#_Toc178631201)

[Test Scenarios, Expected Outcomes, and Actual Outcomes 48](#_Toc178631202)

[Source Code 48](#_Toc178631203)

[Test Case 16: Project Help 49](#_Toc178631204)

[Description 49](#_Toc178631205)

[Steps 49](#_Toc178631206)

[Test Data 50](#_Toc178631207)

[Test Scenarios, Expected Outcomes, Actual Outcomes 50](#_Toc178631208)

[Source Code 51](#_Toc178631209)

[Conclusion 51](#_Toc178631210)

# INTRDOCUTION

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

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

https://github.com/oguzky7/DiscordBotProject\_CISC699/tree/develop/UnitTesting

# TEST PLAN OVERVIEW

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

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

Integration Testing:

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

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

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

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

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

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

https://github.com/oguzky7/DiscordBotProject\_CISC699/tree/develop/UnitTesting

# TEST CASES

## Test Case 0: test\_init.py

### Tools and Technologies

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

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

### Purpose and Setup

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

### Implementation Details

* **Mocking Asynchronous Interactions**: With the project significantly interfacing with asynchronous operations, the pytest-asyncio and pytest-mock tools are extensively used. These tools allow for the effective simulation of asynchronous interactions such as database accesses or API communications without actual network operations.
* **Common Test Setup**: Leveraging pytest fixtures, a standardized test environment is established that includes the configuration of mock objects and the setup of the asyncio event loop. This ensures that each test can run independently and concurrently, reducing test time and improving scalability.
* **Isolation and Patching**: Tests are isolated using pytest-mock to patch external dependencies effectively. This isolation helps ensure that each component is tested against predefined responses, safeguarding the tests against external variability and enhancing their reliability.

### How It Works

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

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

### Source Code

#test\_init.py

import sys, os, logging, pytest, asyncio

import subprocess

from unittest.mock import patch, MagicMock

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))))

#pytest -v > test\_results.txt

#Run this command in the terminal to save the test results to a file

async def run\_monitoring\_loop(control\_object, check\_function, url, date\_str, frequency, iterations=1):

    """Run the monitoring loop for a control object and execute a check function."""

    control\_object.is\_monitoring = True

    results = []

    while control\_object.is\_monitoring and iterations > 0:

        try:

            result = await check\_function(url, date\_str)

        except Exception as e:

            result = f"Failed to monitor: {str(e)}"

        logging.info(f"Monitoring Iteration: {result}")

        results.append(result)

        iterations -= 1

        await asyncio.sleep(frequency)

    control\_object.is\_monitoring = False

    results.append("Monitoring stopped successfully!")

    return results

def setup\_logging():

    """Set up logging without timestamp and other unnecessary information."""

    logger = logging.getLogger()

    if not logger.hasHandlers():

        logging.basicConfig(level=logging.INFO, format='%(message)s')

def save\_test\_results\_to\_file(output\_file="test\_results.txt"):

    """Helper function to run pytest and save results to a file."""

    print("Running tests and saving results to file...")

    output\_path = os.path.join(os.path.dirname(os.path.abspath(\_\_file\_\_)), output\_file)

    with open(output\_path, 'w') as f:

        # Use subprocess to call pytest and redirect output to file

        subprocess.run(['pytest', '-v'], stdout=f, stderr=subprocess.STDOUT)

# Custom fixture for logging test start and end

@pytest.fixture(autouse=True)

def log\_test\_start\_end(request):

    test\_name = request.node.name

    logging.info(f"------------------------------------------------------\nStarting test: {test\_name}\n")

    # Yield control to the test function

    yield

    # Log after the test finishes

    logging.info(f"\nFinished test: {test\_name}\n------------------------------------------------------")

# Import your control classes

from control.BrowserControl import BrowserControl

from control.AccountControl import AccountControl

from control.AvailabilityControl import AvailabilityControl

from control.PriceControl import PriceControl

from control.BotControl import BotControl

from DataObjects.AccountDAO import AccountDAO

from entity.AvailabilityEntity import AvailabilityEntity

from entity.BrowserEntity import BrowserEntity

from entity.PriceEntity import PriceEntity

@pytest.fixture

def base\_test\_case():

    """Base test setup that can be used by all test functions."""

    test\_case = MagicMock()

    test\_case.browser\_control = BrowserControl()

    test\_case.account\_control = AccountControl()

    test\_case.availability\_control = AvailabilityControl()

    test\_case.price\_control = PriceControl()

    test\_case.bot\_control = BotControl()

    test\_case.account\_dao = AccountDAO()

    test\_case.availability\_entity = AvailabilityEntity()

    test\_case.browser\_entity = BrowserEntity()

    test\_case.price\_entity = PriceEntity()

    return test\_case

if \_\_name\_\_ == "\_\_main\_\_":

    # Save the pytest output to a file in the same folder

    save\_test\_results\_to\_file(output\_file="test\_results.txt")

## Test Case 1: Add Account

### Description

This test case assesses the functionality of adding user accounts through the system's Entity and Control layers. It ensures that both layers can handle successful account additions and appropriately react to errors and exceptions during the process.

### Steps

1. **Setup and Mock Initialization**:
   * Initialize test environments with pytest fixtures, mocking necessary components for database and method interactions.
   * Prepare AccountDAO for entity layer tests and AccountControl for control layer tests with appropriate method mocks.
2. **Entity Layer Interaction**:
   * Mock successful and unsuccessful database operations by manipulating the cursor's behavior and the commit operation.
   * Handle exceptions to test the entity's resilience to database errors.
3. **Control Layer Interaction**:
   * Simulate the control layer's handling of account addition using mocked success and failure responses from the AccountDAO.
   * Verify the correct handling of error scenarios and the accurate construction of response messages.
4. **Execution and Validation**:
   * Execute account addition methods across both layers and validate the responses against expected outcomes using assertions.
5. **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 Credentials**:
  + Username: test\_user
  + Password: password123
  + Website: http://example.com
* **Invalid Data**:
  + No account found for the website.

### Entity Tests Code

@pytest.mark.usefixtures("base\_test\_case")

class TestAccountDAO:

    @pytest.fixture

    def account\_dao(self,base\_test\_case, mocker):

        # Mock the psycopg2 connection and cursor

        mocker.patch('psycopg2.connect')

        account\_dao = base\_test\_case.account\_dao

        account\_dao.connection = MagicMock()

        account\_dao.cursor = MagicMock()

        logging.info("Fake database connection established")

        return account\_dao

    def test\_entity\_add\_account\_success(self, account\_dao):

        # Setup the cursor's behavior for successful insertion

        account\_dao.cursor.execute = MagicMock()

        account\_dao.cursor.rowcount = 1

        account\_dao.connection.commit = MagicMock()

        # Test the add\_account method for success

        result = account\_dao.add\_account("test\_user", "password123", "example.com")

        # Log the result of the operation

        logging.info(f"AccountDAO.add\_account returned {result}")

        logging.info("Expected result: True")

         # Assert and log the final outcome

        assert result == True, "Account should be added successfully"

        logging.info("Test add\_account\_success passed")

    def test\_entity\_add\_account\_fail(self, account\_dao):

        # Setup the cursor's behavior to simulate a failure during insertion

        account\_dao.cursor.execute.side\_effect = Exception("Database error")

        account\_dao.cursor.rowcount = 0

        account\_dao.connection.commit = MagicMock()

         # Perform the test

        result = account\_dao.add\_account("fail\_user", "fail123", "fail.com")

        # Log the result of the operation

        logging.info(f"AccountDAO.add\_account returned {result}")

        logging.info("Expected result: False")

        # Assert and log the final outcome

        assert result == False, "Account should not be added"

        logging.info("Test add\_account\_fail passed")

### Entity Test Output

Starting test: test\_entity\_add\_account\_success

Fake database connection established

AccountDAO.add\_account returned True

Expected result: True

Test add\_account\_success passed

Finished test: test\_entity\_add\_account\_success

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

Starting test: test\_entity\_add\_account\_fail

Fake database connection established

AccountDAO.add\_account returned False

Expected result: False

Test add\_account\_fail passed

Finished test: test\_entity\_add\_account\_fail

### Control Tests Code

@pytest.mark.usefixtures("base\_test\_case")

class TestAccountControl:

    @pytest.fixture

    def account\_control(self, base\_test\_case, mocker):

        # Get the mocked AccountControl from base\_test\_case

        account\_control = base\_test\_case.account\_control

        account\_control.account\_dao = MagicMock(spec=base\_test\_case.account\_dao)

        # Mock methods used in the control layer's add\_account

        mocker.patch.object(account\_control.account\_dao, 'connect')

        mocker.patch.object(account\_control.account\_dao, 'close')

        logging.info("Mocked AccountDAO connection and close methods")

        return account\_control

    def test\_control\_add\_account\_success(self, account\_control):

        # Mock successful addition in the DAO layer

        account\_control.account\_dao.add\_account.return\_value = True

        # Call the control method and check the response

        result = account\_control.add\_account("test\_user", "password123", "example.com")

        expected\_message = "Account for example.com added successfully."

        # Log the response and expectations

        logging.info(f"Control method add\_account returned: '{result}'")

        logging.info("Expected message: 'Account for example.com added successfully.'")

        assert result == expected\_message, "The success message should match expected output"

        logging.info("Test control\_add\_account\_success passed")

    def test\_control\_add\_account\_fail(self, account\_control):

        # Mock failure in the DAO layer

        account\_control.account\_dao.add\_account.return\_value = False

        # Call the control method and check the response

        result = account\_control.add\_account("fail\_user", "fail123", "fail.com")

        expected\_message = "Failed to add account for fail.com."

        # Log the response and expectations

        logging.info(f"Control method add\_account returned: '{result}'")

        logging.info("Expected message: 'Failed to add account for fail.com.'")

        assert result == expected\_message, "The failure message should match expected output"

        logging.info("Test control\_add\_account\_fail passed")

### Control Test output

Starting test: test\_control\_add\_account\_success

Mocked AccountDAO connection and close methods

Control method add\_account returned: 'Account for example.com added successfully.'

Expected message: 'Account for example.com added successfully.'

Test control\_add\_account\_success passed

Finished test: test\_control\_add\_account\_success

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

Starting test: test\_control\_add\_account\_fail

Mocked AccountDAO connection and close methods

Control method add\_account returned: 'Failed to add account for fail.com.'

Expected message: 'Failed to add account for fail.com.'

Test control\_add\_account\_fail passed

Finished test: test\_control\_add\_account\_fail

## Test Case 2: Delete Account

### Description

This test case assesses the functionality of deleting user accounts through the system's Entity and Control layers. It ensures that both layers can handle successful account deletions and appropriately react to errors and exceptions during the process.

### Steps:

1. **Setup and Mock Initialization:**
   * Initialize test environments with pytest fixtures, mocking necessary components for database and method interactions.
   * Prepare AccountDAO for entity layer tests and AccountControl for control layer tests with appropriate method mocks.
2. **Entity Layer Interaction:**
   * Mock successful and unsuccessful database operations by manipulating the cursor's behavior and the commit operation.
   * Handle exceptions to test the entity's resilience to database errors.
3. **Control Layer Interaction:**
   * Simulate the control layer's handling of account deletion using mocked success and failure responses from AccountDAO.
   * Verify the correct handling of error scenarios and the accurate construction of response messages.
4. **Execution and Validation:**
   * Execute account deletion methods across both layers and validate the responses against expected outcomes using assertions.
5. **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:**
  + Account ID: 1
* **Invalid Data:**
  + Account ID: 9999 (non-existent account)

### Entity Tests Code:

@pytest.mark.usefixtures("base\_test\_case")

class TestAccountDAO:

@pytest.fixture

def account\_dao(self, base\_test\_case, mocker):

# Mock the psycopg2 connection and cursor

mocker.patch('psycopg2.connect')

account\_dao = base\_test\_case.account\_dao

account\_dao.connection = MagicMock()

account\_dao.cursor = MagicMock()

logging.info("Fake database connection established")

return account\_dao

def test\_entity\_delete\_account\_success(self, account\_dao):

# Setup the cursor's behavior for successful deletion

account\_dao.cursor.execute = MagicMock()

account\_dao.cursor.rowcount = 1

account\_dao.connection.commit = MagicMock()

# Test the delete\_account method for success

result = account\_dao.delete\_account(1)

# Log the result of the operation

logging.info(f"AccountDAO.delete\_account returned {result}")

logging.info("Expected result: True")

# Assert and log the final outcome

assert result == True, "Account should be deleted successfully"

logging.info("Test delete\_account\_success passed")

def test\_entity\_delete\_account\_fail(self, account\_dao):

# Setup the cursor's behavior to simulate a failure during deletion

account\_dao.cursor.execute.side\_effect = Exception("Database error")

account\_dao.cursor.rowcount = 0

account\_dao.connection.commit = MagicMock()

# Perform the test

result = account\_dao.delete\_account(9999)

# Log the result of the operation

logging.info(f"AccountDAO.delete\_account returned {result}")

logging.info("Expected result: False")

# Assert and log the final outcome

assert result == False, "Account should not be deleted"

logging.info("Test delete\_account\_fail passed")

### Entity Test Output

Starting test: test\_entity\_delete\_account\_success

Fake database connection established

AccountDAO.delete\_account returned True

Expected result: True

Test delete\_account\_success passed

Finished test: test\_entity\_delete\_account\_success

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

Starting test: test\_entity\_delete\_account\_fail

Fake database connection established

AccountDAO.delete\_account returned False

Expected result: False

Test delete\_account\_fail passed

Finished test: test\_entity\_delete\_account\_fail

### Control Tests Code

@pytest.mark.usefixtures("base\_test\_case")

class TestAccountControl:

@pytest.fixture

def account\_control(self, base\_test\_case, mocker):

# Get the mocked AccountControl from base\_test\_case

account\_control = base\_test\_case.account\_control

account\_control.account\_dao = MagicMock(spec=base\_test\_case.account\_dao)

# Mock methods used in the control layer's delete\_account

mocker.patch.object(account\_control.account\_dao, 'connect')

mocker.patch.object(account\_control.account\_dao, 'close')

logging.info("Mocked AccountDAO connection and close methods")

return account\_control

def test\_control\_delete\_account\_success(self, account\_control):

# Mock successful deletion in the DAO layer

account\_control.account\_dao.delete\_account.return\_value = True

# Call the control method and check the response

result = account\_control.delete\_account(1)

expected\_message = "Account with ID 1 deleted successfully."

# Log the response and expectations

logging.info(f"Control method delete\_account returned: '{result}'")

logging.info("Expected message: 'Account with ID 1 deleted successfully.'")

assert result == expected\_message, "The success message should match expected output"

logging.info("Test control\_delete\_account\_success passed")

def test\_control\_delete\_account\_fail(self, account\_control):

# Mock failure in the DAO layer

account\_control.account\_dao.delete\_account.return\_value = False

# Call the control method and check the response

result = account\_control.delete\_account(9999)

expected\_message = "Failed to delete account with ID 9999."

# Log the response and expectations

logging.info(f"Control method delete\_account returned: '{result}'")

logging.info("Expected message: 'Failed to delete account with ID 9999.'")

assert result == expected\_message, "The failure message should match expected output"

logging.info("Test control\_delete\_account\_fail passed")

### Control Test Output

Starting test: test\_control\_delete\_account\_success

Mocked AccountDAO connection and close methods

Control method delete\_account returned: 'Account with ID 1 deleted successfully.'

Expected message: 'Account with ID 1 deleted successfully.'

Test control\_delete\_account\_success passed

Finished test: test\_control\_delete\_account\_success

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

Starting test: test\_control\_delete\_account\_fail

Mocked AccountDAO connection and close methods

Control method delete\_account returned: 'Failed to delete account with ID 9999.'

Expected message: 'Failed to delete account with ID 9999.'

Test control\_delete\_account\_fail passed

Finished test: test\_control\_delete\_account\_fail

## Test Case 3: Fetch All Accounts

### Description

This test case evaluates the functionality of fetching all user accounts from the system through the Entity and Control layers. It ensures that the system can retrieve all accounts from the database and appropriately handle situations where no accounts exist or an error occurs.

### Steps

1. **Setup and Mock Initialization:**
   * Initialize test environments with pytest fixtures, mocking necessary components for database and method interactions.
   * Prepare AccountDAO for entity layer tests and AccountControl for control layer tests with appropriate method mocks.
2. **Entity Layer Interaction:**
   * Mock successful and unsuccessful database operations by manipulating the cursor's behavior.
   * Handle exceptions to test the entity's resilience to database errors.
3. **Control Layer Interaction:**
   * Simulate the control layer's handling of fetching accounts using mocked success and failure responses from the AccountDAO.
   * Verify the correct handling of error scenarios and the accurate construction of response messages.
4. **Execution and Validation:**
   * Execute the fetch\_all\_accounts methods across both layers and validate the responses against expected outcomes using assertions.
5. **Logging and Outcome Verification:**
   * Capture detailed logs of the test execution, providing traceability for expected and actual results and aiding in debugging.

### Test Data:

* **Success Scenario:**
  + Accounts: [(1, "test\_user", "password123", "example.com"), (2, "test\_user2", "password456", "example2.com")]
* **Failure Scenario:**
  + No accounts found.

### Entity Tests Code

@pytest.mark.usefixtures("base\_test\_case")

class TestAccountDAO:

@pytest.fixture

def account\_dao(self, base\_test\_case, mocker):

mocker.patch('psycopg2.connect')

account\_dao = base\_test\_case.account\_dao

account\_dao.connection = MagicMock()

account\_dao.cursor = MagicMock()

logging.info("Fake database connection established")

return account\_dao

def test\_entity\_fetch\_all\_accounts\_success(self, account\_dao):

# Mock successful fetch operation

mock\_accounts = [(1, "test\_user", "password123", "example.com"), (2, "test\_user2", "password456", "example2.com")]

account\_dao.cursor.fetchall.return\_value = mock\_accounts

# Test fetch\_all\_accounts method

result = account\_dao.fetch\_all\_accounts()

logging.info(f"AccountDAO.fetch\_all\_accounts returned {result}")

logging.info("Expected result: a list of accounts")

# Assert and log the final outcome

assert result == mock\_accounts, "Should return a list of accounts"

logging.info("Test fetch\_all\_accounts\_success passed")

def test\_entity\_fetch\_all\_accounts\_fail(self, account\_dao):

# Mock failed fetch operation

account\_dao.cursor.fetchall.side\_effect = Exception("Database error")

# Test fetch\_all\_accounts method

result = account\_dao.fetch\_all\_accounts()

logging.info(f"AccountDAO.fetch\_all\_accounts returned {result}")

logging.info("Expected result: an empty list due to failure")

# Assert and log the final outcome

assert result == [], "Should return an empty list due to failure"

logging.info("Test fetch\_all\_accounts\_fail passed")

### Entity Test output

Starting test: test\_entity\_fetch\_all\_accounts\_success

Fake database connection established

AccountDAO.fetch\_all\_accounts returned [(1, 'test\_user', 'password123', 'example.com'), (2, 'test\_user2', 'password456', 'example2.com')]

Expected result: a list of accounts

Test fetch\_all\_accounts\_success passed

Finished test: test\_entity\_fetch\_all\_accounts\_success

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

Starting test: test\_entity\_fetch\_all\_accounts\_fail

Fake database connection established

AccountDAO.fetch\_all\_accounts returned []

Expected result: an empty list due to failure

Test fetch\_all\_accounts\_fail passed

Finished test: test\_entity\_fetch\_all\_accounts\_fail

### Control Tests Code

@pytest.mark.usefixtures("base\_test\_case")

class TestAccountControl:

@pytest.fixture

def account\_control(self, base\_test\_case, mocker):

account\_control = base\_test\_case.account\_control

account\_control.account\_dao = MagicMock(spec=base\_test\_case.account\_dao)

# Mock methods used in the control layer's fetch\_all\_accounts

mocker.patch.object(account\_control.account\_dao, 'connect')

mocker.patch.object(account\_control.account\_dao, 'close')

logging.info("Mocked AccountDAO connection and close methods")

return account\_control

def test\_control\_fetch\_all\_accounts\_success(self, account\_control):

# Mock successful fetch in the DAO layer

mock\_accounts = [(1, "test\_user", "password123", "example.com"), (2, "test\_user2", "password456", "example2.com")]

account\_control.account\_dao.fetch\_all\_accounts.return\_value = mock\_accounts

# Call the control method and check the response

result = account\_control.fetch\_all\_accounts()

expected\_message = "Accounts:\nID: 1, Username: test\_user, Password: password123, Website: example.com\nID: 2, Username: test\_user2, Password: password456, Website: example2.com"

logging.info(f"Control method fetch\_all\_accounts returned: '{result}'")

logging.info(f"Expected message: '{expected\_message}'")

# Assert and log the final outcome

assert result == expected\_message, "The fetched accounts list should match expected output"

logging.info("Test control\_fetch\_all\_accounts\_success passed")

def test\_control\_fetch\_all\_accounts\_fail(self, account\_control):

# Mock failed fetch in the DAO layer

account\_control.account\_dao.fetch\_all\_accounts.return\_value = []

# Call the control method and check the response

result = account\_control.fetch\_all\_accounts()

expected\_message = "No accounts found."

logging.info(f"Control method fetch\_all\_accounts returned: '{result}'")

logging.info(f"Expected message: '{expected\_message}'")

# Assert and log the final outcome

assert result == expected\_message, "The message should indicate no accounts found"

logging.info("Test control\_fetch\_all\_accounts\_fail passed")

### Control Test Output

Starting test: test\_control\_fetch\_all\_accounts\_success

Mocked AccountDAO connection and close methods

Control method fetch\_all\_accounts returned: 'Accounts:

ID: 1, Username: test\_user, Password: password123, Website: example.com

ID: 2, Username: test\_user2, Password: password456, Website: example2.com'

Expected message: 'Accounts:

ID: 1, Username: test\_user, Password: password123, Website: example.com

ID: 2, Username: test\_user2, Password: password456, Website: example2.com'

Test control\_fetch\_all\_accounts\_success passed

Finished test: test\_control\_fetch\_all\_accounts\_success

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

Starting test: test\_control\_fetch\_all\_accounts\_fail

Mocked AccountDAO connection and close methods

Control method fetch\_all\_accounts returned: 'No accounts found.'

Expected message: 'No accounts found.'

Test control\_fetch\_all\_accounts\_fail passed

Finished test: test\_control\_fetch\_all\_accounts\_fail

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

## Test Case 4: Fetch Account by Website

### Description

This test case assesses the functionality of fetching user accounts based on the website through the system's Entity and Control layers. It ensures that both layers can retrieve accounts correctly and handle cases where no accounts are found.

### Steps

1. **Setup and Mock Initialization:**
   * Initialize test environments with pytest fixtures, mocking necessary components for database and method interactions.
   * Prepare AccountDAO for entity layer tests and AccountControl for control layer tests with appropriate method mocks.
2. **Entity Layer Interaction:**
   * Mock successful and unsuccessful database operations by manipulating the cursor's behavior.
   * Handle exceptions to test the entity's resilience to database errors.
3. **Control Layer Interaction:**
   * Simulate the control layer's handling of account fetching using mocked success and failure responses from AccountDAO.
   * Verify the correct handling of error scenarios and the accurate construction of response messages.
4. **Execution and Validation:**
   * Execute account fetch methods across both layers and validate the responses against expected outcomes using assertions.
5. **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:**
  + Website: http://example.com
* **Invalid Data:**
  + No account found for the website.

### Entity Tests Code

@pytest.mark.usefixtures("base\_test\_case")

class TestAccountDAOFetchByWebsite:

@pytest.fixture

def account\_dao(self, base\_test\_case, mocker):

# Mock the psycopg2 connection and cursor

mocker.patch('psycopg2.connect')

account\_dao = base\_test\_case.account\_dao

account\_dao.connection = MagicMock()

account\_dao.cursor = MagicMock()

logging.info("Fake database connection established")

return account\_dao

def test\_entity\_fetch\_account\_success(self, account\_dao):

# Setup the cursor's behavior for successful fetch

account\_dao.cursor.execute = MagicMock()

account\_dao.cursor.fetchone.return\_value = ("test\_user", "password123")

# Test the fetch\_account\_by\_website method for success

result = account\_dao.fetch\_account\_by\_website("example.com")

# Log the result of the operation

logging.info(f"AccountDAO.fetch\_account\_by\_website returned {result}")

logging.info("Expected result: ('test\_user', 'password123')")

# Assert and log the final outcome

assert result == ("test\_user", "password123"), "Account should be fetched successfully"

logging.info("Test fetch\_account\_success passed")

def test\_entity\_fetch\_account\_fail(self, account\_dao):

# Setup the cursor's behavior to simulate failure

account\_dao.cursor.execute = MagicMock()

account\_dao.cursor.fetchone.return\_value = None

# Perform the test

result = account\_dao.fetch\_account\_by\_website("fail.com")

# Log the result of the operation

logging.info(f"AccountDAO.fetch\_account\_by\_website returned {result}")

logging.info("Expected result: None")

# Assert and log the final outcome

assert result is None, "No account should be fetched"

logging.info("Test fetch\_account\_fail passed")

### Entity Test output

Starting test: test\_entity\_fetch\_account\_success

Fake database connection established

AccountDAO.fetch\_account\_by\_website returned ('test\_user', 'password123')

Expected result: ('test\_user', 'password123')

Test fetch\_account\_success passed

Finished test: test\_entity\_fetch\_account\_success

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

Starting test: test\_entity\_fetch\_account\_fail

Fake database connection established

AccountDAO.fetch\_account\_by\_website returned None

Expected result: None

Test fetch\_account\_fail passed

Finished test: test\_entity\_fetch\_account\_fail

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

### Control Tests Code

@pytest.mark.usefixtures("base\_test\_case")

class TestAccountControlFetchByWebsite:

@pytest.fixture

def account\_control(self, base\_test\_case, mocker):

# Get the mocked AccountControl from base\_test\_case

account\_control = base\_test\_case.account\_control

account\_control.account\_dao = MagicMock(spec=base\_test\_case.account\_dao)

# Mock methods used in the control layer's fetch\_account\_by\_website

mocker.patch.object(account\_control.account\_dao, 'connect')

mocker.patch.object(account\_control.account\_dao, 'close')

logging.info("Mocked AccountDAO connection and close methods")

return account\_control

def test\_control\_fetch\_account\_success(self, account\_control):

# Mock successful fetch in the DAO layer

account\_control.account\_dao.fetch\_account\_by\_website.return\_value = ("test\_user", "password123")

# Call the control method and check the response

result = account\_control.fetch\_account\_by\_website("example.com")

expected\_message = ("test\_user", "password123")

# Log the response and expectations

logging.info(f"Control method fetch\_account\_by\_website returned: '{result}'")

logging.info("Expected message: ('test\_user', 'password123')")

# Assert the success message

assert result == expected\_message, "The fetch result should match expected output"

logging.info("Test control\_fetch\_account\_success passed")

def test\_control\_fetch\_account\_fail(self, account\_control):

# Mock failure in the DAO layer

account\_control.account\_dao.fetch\_account\_by\_website.return\_value = None

# Call the control method and check the response

result = account\_control.fetch\_account\_by\_website("fail.com")

expected\_message = "No account found for fail.com."

# Log the response and expectations

logging.info(f"Control method fetch\_account\_by\_website returned: '{result}'")

logging.info("Expected message: 'No account found for fail.com.'")

# Assert the failure message

assert result == expected\_message, "The failure message should match expected output"

logging.info("Test control\_fetch\_account\_fail passed")

### Control Test Output

Starting test: test\_control\_fetch\_account\_success

Mocked AccountDAO connection and close methods

Control method fetch\_account\_by\_website returned: '('test\_user', 'password123')'

Expected message: ('test\_user', 'password123')

Test control\_fetch\_account\_success passed

Finished test: test\_control\_fetch\_account\_success

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

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

Starting test: test\_control\_fetch\_account\_fail

Mocked AccountDAO connection and close methods

Control method fetch\_account\_by\_website returned: 'No account found for fail.com.'

Expected message: 'No account found for fail.com.'

Test control\_fetch\_account\_fail passed

Finished test: test\_control\_fetch\_account\_fail

## Test Case 5: Launch Browser

### Description

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

### Steps

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

### Test Data

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

### Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test\_launch\_browser\_success

Entity Layer Expected: Browser launched.

Entity Layer Received: Browser launched.

Unit Test Passed for entity layer.

Control Layer Expected: Control Object Result: Browser launched.

Control Layer Received: Control Object Result: Browser launched.

Unit Test Passed for control layer.

Finished test: test\_launch\_browser\_success

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

Starting test: test\_launch\_browser\_already\_running

Entity Layer Expected: Browser is already running.

Entity Layer Received: Browser is already running.

Unit Test Passed for entity layer.

Control Layer Expected: Control Object Result: Browser is already running.

Control Layer Received: Control Object Result: Browser is already running.

Unit Test Passed for control layer.

Finished test: test\_launch\_browser\_already\_running

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

Starting test: test\_launch\_browser\_failure\_control

Control Layer Expected to Report: Control Layer Exception: Internal error

Control Layer Received: Control Layer Exception: Internal error

Unit Test Passed for control layer error handling.

Finished test: test\_launch\_browser\_failure\_control

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

Starting test: test\_launch\_browser\_failure\_entity

Entity Layer Expected Failure: Failed to launch browser: Internal error

Control Layer Received: Control Layer Exception: Failed to launch browser: Internal error

Unit Test Passed for entity layer error handling.

Finished test: test\_launch\_browser\_failure\_entity

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

### Source Code

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

## Test Case 6: Close Browser

### Description

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

### Steps

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

### Test Data

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

### Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test\_close\_browser\_success

Entity Layer Expected: Browser closed.

Entity Layer Received: Browser closed.

Unit Test Passed for entity layer.

Control Layer Expected: Control Object Result: Browser closed.

Control Layer Received: Control Object Result: Browser closed.

Unit Test Passed for control layer.

Finished test: test\_close\_browser\_success

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

Starting test: test\_close\_browser\_not\_open

Entity Layer Expected: No browser is currently open.

Entity Layer Received: No browser is currently open.

Unit Test Passed for entity layer.

Control Layer Expected: Control Object Result: No browser is currently open.

Control Layer Received: Control Object Result: No browser is currently open.

Unit Test Passed for control layer.

Finished test: test\_close\_browser\_not\_open

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

Starting test: test\_close\_browser\_failure\_control

Control Layer Expected to Report: Control Layer Exception: Unexpected error

Control Layer Received: Control Layer Exception: Unexpected error

Unit Test Passed for control layer error handling.

Finished test: test\_close\_browser\_failure\_control

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

Starting test: test\_close\_browser\_failure\_entity

Entity Layer Expected Failure: BrowserEntity\_Failed to close browser: Internal error

Control Layer Received: Control Layer Exception: BrowserEntity\_Failed to close browser: Internal error

Unit Test Passed for entity layer error handling.

Finished test: test\_close\_browser\_failure\_entity

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

### Source Code

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

## Test Case 7 : Navigate to Website

### Description

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

### Steps

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

### Test Data

* **Valid URL:** "<https://example.com>"
* **Invalid URL:** "invalid\_site"

### Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test\_navigate\_to\_website\_success

Entity Layer Expected: Navigated to https://example.com

Entity Layer Received: Navigated to https://example.com

Unit Test Passed for entity layer.

Control Layer Expected: Control Object Result: Navigated to https://example.com

Control Layer Received: Control Object Result: Navigated to https://example.com

Unit Test Passed for control layer.

Finished test: test\_navigate\_to\_website\_success

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

Starting test: test\_navigate\_to\_website\_invalid\_url

Control Layer Expected: URL for invalid\_site not found.

Control Layer Received: URL for invalid\_site not found.

Unit Test Passed for control layer invalid URL handling.

Finished test: test\_navigate\_to\_website\_invalid\_url

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

Starting test: test\_navigate\_to\_website\_failure\_entity

Control Layer Expected: Control Layer Exception: Failed to navigate

Control Layer Received: Control Layer Exception: Failed to navigate

Unit Test Passed for entity layer error handling.

Finished test: test\_navigate\_to\_website\_failure\_entity

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

Starting test: test\_navigate\_to\_website\_launch\_browser\_on\_failure

Control Layer Expected: Control Object Result: Navigated to https://example.com

Control Layer Received: Control Object Result: Navigated to https://example.com

Unit Test Passed for control layer with browser launch.

Finished test: test\_navigate\_to\_website\_launch\_browser\_on\_failure

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

Starting test: test\_navigate\_to\_website\_failure\_control

Control Layer Expected: Control Layer Exception: Control Layer Failed

Control Layer Received: Control Layer Exception: Control Layer Failed

Unit Test Passed for control layer failure.

Finished test: test\_navigate\_to\_website\_failure\_control

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

### Source Code

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

## Test Case 8: Login

### Description

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

### Steps

1. **Setup and Mock Initialization:**
   * Initialize the test environment using pytest fixtures to mock necessary components and set up the logging.
   * Access the BrowserControl and AccountControl objects, preparing methods such as login and fetch\_account\_by\_website for mocking to simulate database and browser interactions.
2. **Entity and Control Layer Interaction:**
   * Simulate successful login by mocking the BrowserEntity.login method and the AccountControl.fetch\_account\_by\_website method to return predefined credentials.
   * Handle scenarios where no account information is found, simulating the return of None from the fetch method.
   * Introduce exceptions in the entity layer to test error handling by setting the side\_effect of the mock to raise an exception, simulating internal errors during the login process.
3. **Execution and Validation:**
   * Execute the login command through the control object, passing necessary parameters like the website URL.
   * Verify the responses from both the entity and control layers against expected outcomes, using assertions to ensure both layers react appropriately to each scenario.
4. **Logging and Outcome Verification:**
   * Log detailed results of each test case, including expected and actual outcomes for transparency and troubleshooting.
   * Ensure the logs capture all pertinent information, aiding in debugging and validation of test results.
5. **Error and Exception Handling:**
   * Test how the control layer manages no account scenarios and various failures, ensuring robust error handling and user feedback accuracy.

### Test Data

* **Valid Credentials:**
  + Username: sample\_username
  + Password: sample\_password
  + Website: http://example.com
* **Invalid Data:**
  + No account found for the website.

### Test Scenarios, Expected Outcomes, and Actual Outcomes

Starting test: test\_login\_success

Entity Layer Expected: Logged in to http://example.com successfully with username: sample\_username

Entity Layer Received: Logged in to http://example.com successfully with username: sample\_username

Unit Test Passed for entity layer.

Control Layer Expected: Control Object Result: Logged in to http://example.com successfully with username: sample\_username

Control Layer Received: Control Object Result: Logged in to http://example.com successfully with username: sample\_username

Unit Test Passed for control layer.

Finished test: test\_login\_success

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

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

Starting test: test\_login\_no\_account

Control Layer Expected: No account found for example.com

Control Layer Received: No account found for example.com

Unit Test Passed for missing account handling.

Finished test: test\_login\_no\_account

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

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

Starting test: test\_login\_entity\_layer\_failure

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

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

Unit Test Passed for entity layer failure.

Finished test: test\_login\_entity\_layer\_failure

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

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

Starting test: test\_login\_control\_layer\_failure

Control Layer Expected: Control Layer Exception: Control layer failure during account fetch.

Control Layer Received: Control Layer Exception: Control layer failure during account fetch.

Unit Test Passed for control layer failure handling.

Finished test: test\_login\_control\_layer\_failure

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

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

Starting test: test\_login\_invalid\_url

Control Layer Expected: URL for example not found.

Control Layer Received: URL for example not found.

Unit Test Passed for missing URL/selector handling.

Finished test: test\_login\_invalid\_url

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

### Source Code

Source Code can be found at the end of this document and in the GitHub repository for a more organized overview: [GitHub - DiscordBotProject\_CISC699](https://github.com/oguzky7/DiscordBotProject_CISC699)

## Test Case 9: Get Price

### Description

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

### Steps

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

### Test Data

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

### Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test\_get\_price\_success

Entity Layer Expected: $199.99

Entity Layer Received: $199.99

Unit Test Passed for entity layer.

Control Layer Expected: $199.99

Control Layer Received: $199.99

Unit Test Passed for control layer.

Finished test: test\_get\_price\_success

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

Starting test: test\_get\_price\_invalid\_url

Control Layer Expected: Error fetching price: Invalid URL

Control Layer Received: Error fetching price: Invalid URL

Unit Test Passed for control layer invalid URL handling.

Finished test: test\_get\_price\_invalid\_url

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

Starting test: test\_get\_price\_failure\_entity

Control Layer Expected: Failed to fetch price: Failed to fetch price

Control Layer Received: Failed to fetch price: Failed to fetch price

Unit Test Passed for entity layer error handling.

Finished test: test\_get\_price\_failure\_entity

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

Starting test: test\_get\_price\_failure\_control

Control Layer Expected: Control Layer Exception: Control Layer Failed

Control Layer Received: Control Layer Exception: Control Layer Failed

Unit Test Passed for control layer failure.

Finished test: test\_get\_price\_failure\_control

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

### Source Code

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

## Test Case 10: Check Availability

### Description

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

### Steps

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

### Test Data

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

### Test Scenarios, Expected and Actual Outcomes

Starting test: test\_check\_availability\_success

Entity Layer Expected: Selected or default date current date is available for booking.

Entity Layer Received: Selected or default date current date is available for booking.

Unit Test Passed for entity layer.

Control Layer Expected: Checked availability: Selected or default date current date is available for booking.

Control Layer Received: Checked availability: Selected or default date current date is available for booking.

Unit Test Passed for control layer.

Finished test: test\_check\_availability\_success

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

Starting test: test\_check\_availability\_failure\_entity

Control Layer Expected: Failed to check availability: Failed to check availability

Control Layer Received: Failed to check availability: Failed to check availability

Unit Test Passed for entity layer error handling.

Finished test: test\_check\_availability\_failure\_entity

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

Starting test: test\_check\_availability\_no\_availability

Entity Layer Received: No availability for the selected date.

Control Layer Received: Checked availability: No availability for the selected date.

Unit Test Passed for control layer no availability handling.

Finished test: test\_check\_availability\_no\_availability

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

Starting test: test\_check\_availability\_failure\_control

Control Layer Expected: Control Layer Exception: Control Layer Failed

Control Layer Received: Control Layer Exception: Control Layer Failed

Unit Test Passed for control layer failure.

Finished test: test\_check\_availability\_failure\_control

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

### Source Code

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

## Test Case 11: Start Monitoring Availability

### Description

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

### Steps

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

### Test Data

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

### Test Scenarios, Expected Outcomes, and Actual Outcomes

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

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

Control Layer Expected: ['Checked availability: Selected or default date is available for booking.', 'Monitoring stopped successfully!']

Control Layer Received: ['Checked availability: Selected or default date is available for booking.', 'Monitoring stopped successfully!']

Unit Test Passed for control layer.

Finished test: test\_start\_monitoring\_availability\_success

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

Starting test: test\_start\_monitoring\_availability\_failure\_entity

Monitoring Iteration: Failed to check availability: Failed to check availability

Control Layer Expected: ['Failed to check availability: Failed to check availability', 'Monitoring stopped successfully!']

Control Layer Received: ['Failed to check availability: Failed to check availability', 'Monitoring stopped successfully!']

Unit Test Passed for entity layer error handling.

Finished test: test\_start\_monitoring\_availability\_failure\_entity

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

Starting test: test\_start\_monitoring\_availability\_failure\_control

Control Layer Expected: Control Layer Exception: Control Layer Failed

Control Layer Received: Control Layer Exception: Control Layer Failed

Unit Test Passed for control layer failure.

Finished test: test\_start\_monitoring\_availability\_failure\_control

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

Starting test: test\_start\_monitoring\_availability\_already\_running

Control Layer Expected: Already monitoring availability.

Control Layer Received: Already monitoring availability.

Unit Test Passed for control layer already running handling.

Finished test: test\_start\_monitoring\_availability\_already\_running

### Source Code

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

## Test Case 12: Stop Monitoring Availability

### Description

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

### Steps

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

### Test Data

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

### Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test\_stop\_monitoring\_availability\_success

Control Layer Expected to contain: Monitoring stopped successfully!

Control Layer Received: Results for availability monitoring:

Checked availability: Selected or default date is available for booking.

Monitoring stopped successfully!

Unit Test Passed for stop monitoring availability.

Finished test: test\_stop\_monitoring\_availability\_success

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

Starting test: test\_stop\_monitoring\_availability\_no\_active\_session

Control Layer Expected: There was no active availability monitoring session. Nothing to stop.

Control Layer Received: There was no active availability monitoring session. Nothing to stop.

Unit Test Passed for stop monitoring with no active session.

Finished test: test\_stop\_monitoring\_availability\_no\_active\_session

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

### Source Code

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

## Test Case 13: Start Monitoring Price

### Description

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

### Steps

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

### Test Data

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

### Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test\_start\_monitoring\_price\_success

Entity Layer Expected: Starting price monitoring. Current price: 100 USD

Control Layer Received: Starting price monitoring. Current price: 100 USD

Unit Test Passed for start\_monitoring\_price success scenario.

Finished test: test\_start\_monitoring\_price\_success

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

Starting test: test\_start\_monitoring\_price\_already\_running

Control Layer Expected: Already monitoring prices.

Control Layer Received: Already monitoring prices.

Unit Test Passed for already running scenario.

Finished test: test\_start\_monitoring\_price\_already\_running

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

Starting test: test\_start\_monitoring\_price\_failure\_in\_entity

Control Layer Expected: Starting price monitoring. Current price: Failed to fetch price: Error fetching price

Control Layer Received: Starting price monitoring. Current price: Failed to fetch price: Error fetching price

Unit Test Passed for entity layer failure scenario.

Finished test: test\_start\_monitoring\_price\_failure\_in\_entity

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

Starting test: test\_start\_monitoring\_price\_failure\_in\_control

Control Layer Expected: Control Layer Exception

Control Layer Received: Control Layer Exception: Control Layer Exception

Unit Test Passed for control layer failure scenario.

Finished test: test\_start\_monitoring\_price\_failure\_in\_control

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

### Source Code

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

## Test Case 14: Stop Monitoring Price

### Description

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

### Steps

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

### Test Data

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

### Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test\_stop\_monitoring\_price\_success

Control Layer Expected: Results for price monitoring:

Price went up!

Price went down!

Price monitoring stopped successfully!

Control Layer Received: Results for price monitoring:

Price went up!

Price went down!

Price monitoring stopped successfully!

Unit Test Passed for stop\_monitoring\_price success scenario.

Finished test: test\_stop\_monitoring\_price\_success

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

Starting test: test\_stop\_monitoring\_price\_not\_active

Control Layer Expected: There was no active price monitoring session. Nothing to stop.

Control Layer Received: There was no active price monitoring session. Nothing to stop.

Unit Test Passed for stop\_monitoring\_price when not active.

Finished test: test\_stop\_monitoring\_price\_not\_active

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

Starting test: test\_stop\_monitoring\_price\_failure\_in\_control

Control Layer Expected: Error stopping price monitoring

Control Layer Received: Error stopping price monitoring

Unit Test Passed for stop\_monitoring\_price failure scenario.

Finished test: test\_stop\_monitoring\_price\_failure\_in\_control

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

### Source Code

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

## Test Case 15: Stop Bot

### Description

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

### Steps

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

### Test Data

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

### Test Scenarios, Expected Outcomes, and Actual Outcomes

Starting test: test\_stop\_bot\_success

Control Layer Expected: Bot has been shut down.

Control Layer Received: Bot has been shut down.

Unit Test Passed for control layer stop bot.

Finished test: test\_stop\_bot\_success

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

Starting test: test\_stop\_bot\_failure\_control

Control Layer Expected: Control Layer Exception: Control Layer Failed

Control Layer Received: Control Layer Exception: Control Layer Failed

Unit Test Passed for control layer failure.

Finished test: test\_stop\_bot\_failure\_control

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

### Source Code

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

## Test Case 16: Project Help

### Description

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

### Steps

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

### Test Data

* **Command Inputs:** No direct user inputs are required; the test internally triggers the project\_help command.

### Test Scenarios, Expected Outcomes, Actual Outcomes

Starting test: test\_project\_help\_success

Control Layer Expected: Here are the available commands:

!project\_help - Get help on available commands.

!fetch\_all\_accounts - Fetch all stored accounts.

!add\_account 'username' 'password' 'website' - Add a new account to the database.

!fetch\_account\_by\_website 'website' - Fetch account details by website.

!delete\_account 'account\_id' - Delete an account by its ID.

!launch\_browser - Launch the browser.

!close\_browser - Close the browser.

!navigate\_to\_website 'url' - Navigate to a specified website.

!login 'website' - Log in to a website (e.g., !login bestbuy).

!get\_price 'url' - Check the price of a product on a specified website.

!start\_monitoring\_price 'url' 'frequency' - Start monitoring a product's price at a specific interval (frequency in minutes).

!stop\_monitoring\_price - Stop monitoring the product's price.

!check\_availability 'url' - Check availability for a restaurant or service.

!start\_monitoring\_availability 'url' 'frequency' - Monitor availability at a specific interval.

!stop\_monitoring\_availability - Stop monitoring availability.

!stop\_bot - Stop the bot.

Control Layer Received: Here are the available commands:

!project\_help - Get help on available commands.

!fetch\_all\_accounts - Fetch all stored accounts.

!add\_account 'username' 'password' 'website' - Add a new account to the database.

!fetch\_account\_by\_website 'website' - Fetch account details by website.

!delete\_account 'account\_id' - Delete an account by its ID.

!launch\_browser - Launch the browser.

!close\_browser - Close the browser.

!navigate\_to\_website 'url' - Navigate to a specified website.

!login 'website' - Log in to a website (e.g., !login bestbuy).

!get\_price 'url' - Check the price of a product on a specified website.

!start\_monitoring\_price 'url' 'frequency' - Start monitoring a product's price at a specific interval (frequency in minutes).

!stop\_monitoring\_price - Stop monitoring the product's price.

!check\_availability 'url' - Check availability for a restaurant or service.

!start\_monitoring\_availability 'url' 'frequency' - Monitor availability at a specific interval.

!stop\_monitoring\_availability - Stop monitoring availability.

!stop\_bot - Stop the bot.

Unit Test Passed for project help.

Finished test: test\_project\_help\_success

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

Starting test: test\_project\_help\_failure

Control Layer Expected: Error handling help command: Error handling help command

Control Layer Received: Error handling help command: Error handling help command

Unit Test Passed for error handling in project help.

Finished test: test\_project\_help\_failure

### Source Code

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

## Conclusion

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

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