Discord Bot Automation Assistant

Discord Bot Automation Assistant Test Plan

Oguz Kaan Yildirim

307637

Table Of Contents

INTRDOCUTION

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

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

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

TEST PLAN OVERVIEW

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

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

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

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

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

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

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

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

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

TEST CASES

Test Case 0: test_init.py

Tools and Technologies

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

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

Purpose and Setup

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

Implementation Details

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

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

How It Works

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

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

Source Code

```
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__))))


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

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

    yield

    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")
```

To keep the documentation clean, imports won't be shown here. But all the tests import this class and some other imports depending on the necessity.

from test_init import setup_logging, base_test_case, save_test_results_to_file, log_test_start_end, logging

Also, all comments removed from the code in this documentation.

Original code can be found in GitHub!

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):
        mocker.patch('psycopg2.connect')
        account_dao = base_test_case.account_dao
        account_dao.connection = MagicMock()
        account_dao.cursor = MagicMock()
        return account_dao

    def test_entity_add_account_success(self, account_dao):
        account_dao.cursor.execute = MagicMock()
        account_dao.cursor.rowcount = 1
        account_dao.connection.commit = MagicMock()

        result = account_dao.add_account("test_user", "password123", "example.com")

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

    def test_entity_add_account_fail(self, account_dao):
        account_dao.cursor.execute.side_effect = Exception("Database error")
        account_dao.cursor.rowcount = 0
        account_dao.connection.commit = MagicMock()

        result = account_dao.add_account("fail_user", "fail123", "fail.com")

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

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):
        account_control = base_test_case.account_control
        account_control.account_dao = MagicMock(spec=base_test_case.account_dao)

        mocker.patch.object(account_control.account_dao, 'connect')
        mocker.patch.object(account_control.account_dao, 'close')
        return account_control

    def test_control_add_account_success(self, account_control):
        account_control.account_dao.add_account.return_value = True

        result = account_control.add_account("test_user", "password123", "example.com")
        expected_message = "Account for example.com added successfully."

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

    def test_control_add_account_fail(self, account_control):
        account_control.account_dao.add_account.return_value = False

        result = account_control.add_account("fail_user", "fail123", "fail.com")
        expected_message = "Failed to add account for fail.com."

        assert result == expected_message, "The failure message should match expected output"
```

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):
        mocker.patch('psycopg2.connect')
        account_dao = base_test_case.account_dao
        account_dao.connection = MagicMock()
        account_dao.cursor = MagicMock()
        return account_dao

    def test_entity_delete_account_success(self, account_dao):
        account_dao.cursor.execute = MagicMock()
        account_dao.cursor.rowcount = 1
        account_dao.connection.commit = MagicMock()

        result = account_dao.delete_account(1)
        assert result == True, "Account should be deleted successfully"

    def test_entity_delete_account_fail(self, account_dao):
        account_dao.cursor.execute.side_effect = Exception("Database error")
        account_dao.cursor.rowcount = 0
        account_dao.connection.commit = MagicMock()

        result = account_dao.delete_account(9999)
        assert result == False, "Account should not be deleted"
```

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):
        account_control = base_test_case.account_control
        account_control.account_dao = MagicMock(spec=base_test_case.account_dao)

        mocker.patch.object(account_control.account_dao, 'connect')
        mocker.patch.object(account_control.account_dao, 'close')
        return account_control

    def test_control_delete_account_success(self, account_control):
        account_control.account_dao.delete_account.return_value = True

        result = account_control.delete_account(1)
        expected_message = "Account with ID 1 deleted successfully."

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

    def test_control_delete_account_fail(self, account_control):
        account_control.account_dao.delete_account.return_value = False

        result = account_control.delete_account(9999)
        expected_message = "Failed to delete account with ID 9999."

        assert result == expected_message, "The failure message should match expected output"
```

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()
        return account_dao

    def test_entity_fetch_all_accounts_success(self, account_dao):
        mock_accounts = [(1, "test_user", "password123", "example.com"), (2, "test_user2", "password456",
"example2.com")]
        account_dao.cursor.fetchall.return_value = mock_accounts

        result = account_dao.fetch_all_accounts()

        assert result == mock_accounts, "Should return a list of accounts"

    def test_entity_fetch_all_accounts_fail(self, account_dao):
        account_dao.cursor.fetchall.side_effect = Exception("Database error")

        result = account_dao.fetch_all_accounts()

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

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

```
  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)

      mocker.patch.object(account_control.account_dao, 'connect')
      mocker.patch.object(account_control.account_dao, 'close')
      return account_control

  def test_control_fetch_all_accounts_success(self, account_control):
      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

      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"

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

  def test_control_fetch_all_accounts_fail(self, account_control):
      account_control.account_dao.fetch_all_accounts.return_value = []

      result = account_control.fetch_all_accounts()

      expected_message = "No accounts found."
      assert result == expected_message, "The message should indicate no accounts found"
```

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

```
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()
    return account_dao

def test_entity_fetch_account_success(self, account_dao):
    account_dao.cursor.execute = MagicMock()
    account_dao.cursor.fetchone.return_value = ("test_user", "password123")

    result = account_dao.fetch_account_by_website("example.com")

    assert result == ("test_user", "password123"), "Account should be fetched successfully"
def test_entity_fetch_account_fail(self, account_dao):
    account_dao.cursor.execute = MagicMock()
    account_dao.cursor.fetchone.return_value = None
    result = account_dao.fetch_account_by_website("fail.com")

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

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):
        account_control = base_test_case.account_control
        account_control.account_dao = MagicMock(spec=base_test_case.account_dao)

        mocker.patch.object(account_control.account_dao, 'connect')
        mocker.patch.object(account_control.account_dao, 'close')
        return account_control

    def test_control_fetch_account_success(self, account_control):
        account_control.account_dao.fetch_account_by_website.return_value = ("test_user", "password123")

        result = account_control.fetch_account_by_website("example.com")
        expected_message = ("test_user", "password123")

        assert result == expected_message, "The fetch result should match expected output"

    def test_control_fetch_account_fail(self, account_control):
        account_control.account_dao.fetch_account_by_website.return_value = None

        result = account_control.fetch_account_by_website("fail.com")
        expected_message = "No account found for fail.com."

        assert result == expected_message, "The failure message should match expected output"
```

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: Receive Email

Description

This test case evaluates the functionality of receiving email commands in the system's Entity and Control layers. It verifies whether both layers can handle the success and failure scenarios of sending an email with attachments and appropriately respond to errors during the process.

Steps

1. **Setup and Mock Initialization**:
    - Initialize the test environment with pytest fixtures, mocking the necessary components for the email sending process.
    - Prepare EmailDAO for the entity layer tests and EmailControl for control layer tests, with appropriate method mocks using smtplib and receive_command.
2. **Entity Layer Interaction**:
    - Simulate successful and failed email sending operations by manipulating the return values and exceptions raised during execution.
    - Test email sending by calling the send_email_with_attachments method with various file inputs.
3. **Control Layer Interaction**:
    - Simulate the control layer's handling of the receive_email command using mocked success and failure responses from EmailDAO.
    - Verify the correct handling of both successful and unsuccessful email sending scenarios and the accurate construction of response messages.
4. **Execution and Validation**:
    - Execute email sending 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 to provide traceability for expected and actual results and to aid in debugging.

Test Data

- **Valid Input**:
  - File Name: monitor_price.html
  - Expected Response: "Email with file 'monitor_price.html' sent successfully!"
- **Invalid Input**:
  - File Name: non_existent_file.html
  - Expected Response: "File 'non_existent_file.html' not found in either excelFiles or htmlFiles."

Utility Tests Code

```
def email_dao(self, base_test_case, mocker):
    email_dao = base_test_case.email_dao
    mocker.patch('smtplib.SMTP')
    return email_dao

def test_entity_send_email_success(self, email_dao):
    email_dao.return_value = "Email with file 'monitor_price.html' sent successfully!"
    result = email_dao('monitor_price.html')
    assert result == "Email with file 'monitor_price.html' sent successfully!"

def test_entity_send_email_fail(self, email_dao):
    email_dao.return_value = "File 'non_existent_file.html' not found."
    result = email_dao('non_existent_file.html')
    assert result == "File 'non_existent_file.html' not found in either excelFiles or htmlFiles."
```

Utility Test Output

```
Starting test: test_entity_send_email_success

Mocked EmailDAO with send_email_with_attachments method
Test send_email_success result: Email with file 'monitor_price.html' sent successfully!
Test send_email_success passed

Finished test: test_entity_send_email_success
------------------------------------------------------------
Starting test: test_entity_send_email_fail

Mocked EmailDAO with send_email_with_attachments method
Test send_email_fail result: File 'non_existent_file.html' not found in either excelFiles or htmlFiles.
Test send_email_fail passed

Finished test: test_entity_send_email_fail
```

Control Tests Code:

```
@pytest.mark.usefixtures("base_test_case")
class TestEmailControl:

    @pytest.fixture
    def email_control(self, base_test_case, mocker):
        email_control = base_test_case.bot_control
        email_control.receive_command = MagicMock()
        logging.info("Mocked EmailControl (BotControl) for control layer")
        return email_control

    def test_control_send_email_success(self, email_control):
        email_control.receive_command.return_value = "Email with file 'monitor_price.html' sent successfully!"
        result = email_control.receive_command("receive_email", "monitor_price.html")
        logging.info(f"Test control_send_email_success result: {result}")
        assert result == "Email with file 'monitor_price.html' sent successfully!"
        logging.info("Test control_send_email_success passed")

    def test_control_send_email_fail(self, email_control):
        email_control.receive_command.return_value = "File 'non_existent_file.html' not found."
        result = email_control.receive_command("receive_email", "non_existent_file.html")
        logging.info(f"Test control_send_email_fail result: {result}")
        assert result == "File 'non_existent_file.html' not found."
        logging.info("Test control_send_email_fail passed")
```

Control Test Output

```
Starting test: test_control_send_email_success

Mocked EmailControl (BotControl) for control layer
Test control_send_email_success result: Email with file 'monitor_price.html' sent successfully!
Test control_send_email_success passed

Finished test: test_control_send_email_success
-------------------------------------------------------
Starting test: test_control_send_email_fail

Mocked EmailControl (BotControl) for control layer
Test control_send_email_fail result: File 'non_existent_file.html' not found.
Test control_send_email_fail passed

Finished test: test_control_send_email_fail
```

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 7: 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.

Entity Tests Code

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

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

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

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

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

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

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

Entity Test output

```
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
```

Control Tests Code

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

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

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

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

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

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

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

Control Test Output

```
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_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
-----------------------------------------------------------
```

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

Test Case 17: 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.

Entity and Control Tests Code

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

```
async def test_stop_bot_success(base_test_case):
   with patch('control.BotControl.BotControl.receive_command') as mock_stop_bot:
      mock_stop_bot.return_value = "Bot has been shut down."
      expected_result = "Bot has been shut down."
      result = await base_test_case.bot_control.receive_command("stop_bot", ctx=MagicMock())
      assert result == expected_result

async def test_stop_bot_failure_control(base_test_case):
   with patch('control.BotControl.BotControl.receive_command', side_effect=Exception("Control Layer Failed"))
as mock_control:
      expected_result = "Control Layer Exception: Control Layer Failed"
      try:
         result = await base_test_case.bot_control.receive_command("stop_bot", ctx=MagicMock())
      except Exception as e:
         result = f"Control Layer Exception: {str(e)}"
      assert result == expected_result
```

Entity and Control Test output

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

```
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
```

Test Case 18: 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.

Entity and Control Tests Code

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

```
async def test_project_help_success(base_test_case):
   with patch('control.BotControl.BotControl.receive_command') as mock_help:
      mock_help.return_value = (“Here are the available commands:
Help message is too long, not wasting space here”)
      expected_result = mock_help.return_value
      result = await base_test_case.bot_control.receive_command("project_help")
      assert result == expected_result

async def test_project_help_failure(base_test_case):
   with patch('control.BotControl.BotControl.receive_command', side_effect=Exception("Error handling help
command")) as mock_help:
      expected_result = "Error handling help command: Error handling help command"
      try:
         result = await base_test_case.bot_control.receive_command("project_help")
      except Exception as e:
         result = f"Error handling help command: {str(e)}"
      assert result == expected_result
```

Entity and Control Test output

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

```
Starting test: test_project_help_success
Control Layer Expected: Here are the available commands:
Help message is too long, not wasting space here

Control Layer Received: Here are the available commands:
Help message is too long, not wasting space here
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
```

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.