**Discord Bot Automation Assistant**

**Discord Bot Automation Assistant Test Plan**

**By**

**Oguz Kaan Yildirim**

**M.S. Harrisburg University of Science and Technology**

# Table Of Contents

[Table Of Contents 2](#_Toc178285271)

[INTRDOCUTION 4](#_Toc178285272)

[TEST PLAN OVERVIEW 5](#_Toc178285273)

[TEST CASES 7](#_Toc178285274)

[Test Case 0: test\_init.py 7](#_Toc178285275)

[Description: 7](#_Toc178285276)

[Purpose and Setup 7](#_Toc178285277)

[Implementation Details 8](#_Toc178285278)

[How It Works 8](#_Toc178285279)

[Test Case 1: Add Account 9](#_Toc178285280)

[Description: 9](#_Toc178285281)

[Steps: 9](#_Toc178285282)

[Test Data 10](#_Toc178285283)

[Expected Outcomes 10](#_Toc178285284)

[Mocking Details 10](#_Toc178285285)

[Output 11](#_Toc178285286)

[Source Code 11](#_Toc178285287)

[Test Case 2: Delete Account 12](#_Toc178285288)

[Description: 12](#_Toc178285289)

[Steps: 12](#_Toc178285290)

[Test Data: 13](#_Toc178285291)

[Expected Outcomes: 13](#_Toc178285292)

[Mocking Details: 13](#_Toc178285293)

[Output 14](#_Toc178285294)

[Source Code: 14](#_Toc178285295)

[Test Case 3: Fetch All Accounts 15](#_Toc178285296)

[Description: 15](#_Toc178285297)

[Steps: 15](#_Toc178285298)

[Test Data: 16](#_Toc178285299)

[Expected Outcomes: 16](#_Toc178285300)

[Mocking Details: 17](#_Toc178285301)

[Output 17](#_Toc178285302)

[Source Code: 17](#_Toc178285303)

[Test Case 4: Fetch Account by Website 19](#_Toc178285304)

[Description: 19](#_Toc178285305)

[Steps: 19](#_Toc178285306)

[Test Data: 19](#_Toc178285307)

[Expected Outcomes: 19](#_Toc178285308)

[Output: 20](#_Toc178285309)

[Source Code: 20](#_Toc178285310)

# INTRDOCUTION

The purpose of this document is to outline a comprehensive test plan for the " Discord Bot Automation Assistant" project. This test plan is meticulously designed to ensure the robustness, correctness, and functionality of all components involved in the project, emphasizing the meticulous validation of each unit through structured test cases. The plan incorporates tests for various object types involved in the project—boundary, control, and entity—alongside additional focus on integration points, without accessing real databases or external systems, adhering strictly to unit testing principles using mock objects and fake data.

This document will guide the systematic testing of individual components and their interactions within the system, ensuring that all functionalities meet the specified requirements and behave as expected in various scenarios, both typical and atypical. Each test is crafted to validate specific elements of the system, from the fundamental logic handled by entity and control objects to the data flow managed by boundary objects interfacing with the user.

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 test plan is constructed to systematically validate the performance and reliability of the "Automated Discord Bot Helper," ensuring that each component not only operates in isolation but also performs optimally within the system's ecosystem. The plan is segmented into several suites, each targeting specific components:

**Entity Objects Testing**: Focuses on ensuring that each entity object maintains integrity, correctly manages state, and interacts flawlessly with other components. Tests will include creating, manipulating, and validating state changes within these objects.

**Control Objects Testing**: Aims to verify that control objects accurately orchestrate the flow of data between the user interfaces (boundary objects) and the data management layers. This includes testing the logical conditions and workflows that control objects are responsible for.

**Boundary Objects Testing**: Tests the interfaces that interact with the system users, ensuring data is correctly captured, validated, and passed to the underlying control layers. This suite ensures that all user inputs are handled correctly, simulating various user interaction scenarios.

**Integration of Components**: Although primarily focusing on unit testing, the plan includes a series of tests designed to ensure that components work together as expected under controlled conditions using mocks and stubs instead of real data connections. This approach adheres to the unit testing philosophy while ensuring that interactions between components are tested without crossing into full integration testing.

**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 involves a suite of tools designed to facilitate comprehensive unit testing of the "Discord Service Notifier" project. Key tools include:

* **Python**: The primary programming language used for developing both the application and the test cases.
* **unittest**: A built-in Python framework for constructing and running tests, offering capabilities to setup, execute, and teardown tests.
* **unittest.mock**: Provides a core Mock class removing the need for dependencies during testing. This is crucial for simulating the behavior of complex objects in a controlled environment.
* **AsyncMock**: A subclass of unittest.mock's Mock, designed to test asynchronous functions.
* **discord.py**: A Python library for interacting with Discord, mocked in our tests to simulate interactions without real server connections.
* **CustomTextTestRunner**: An extension of unittest’s TextTestRunner that is tailored to provide customized output formats for test results, enhancing readability and diagnostics.

### Purpose and Setup

The test\_init.py file lays the foundational framework for all other test scripts in the project. It is designed to centralize common setup and teardown processes that are essential across multiple test cases, ensuring consistency and reducing redundancy in the testing codebase.

### Implementation Details

* **Mocking Discord Interactions**: Given that the project interfaces significantly with the Discord API, discord.py is used extensively. However, in the context of unit testing, direct calls to Discord's servers are impractical and potentially disruptive. Instead, the AsyncMock tool is utilized to simulate these interactions. This allows the tests to mimic the behavior of the bot (e.g., sending messages, handling commands) without actual network operations.
* **Common Test Setup**: The BaseTestSetup class provided within test\_init.py is used across all test scripts to establish a consistent environment for each test. This setup includes configuring a mock version of the Discord bot and a testing framework capable of handling asynchronous calls typical in a Discord bot environment.
* **Mocking and Patching**: The use of unittest.mock.patch is pivotal in controlling the scope and impact of external dependencies within tests. By replacing parts of the system under test with mock objects, it ensures that tests run in isolation, thereby increasing their reliability and speed. For instance, patching external API calls to return predetermined responses allows us to test how our system reacts to various external stimuli.

### How It Works

* When a test case is executed, test\_init.py configures the necessary environment by setting up paths and initializing mock objects. This allows each test script to operate independently of the live Discord environment.
* The AsyncMock setup is particularly important for simulating asynchronous methods that interact with the Discord API, such as sending messages or processing commands. Each command's effect is simulated, and its impact is assessed within a controlled test scenario, ensuring the bot reacts as expected in various situations without actual side effects.
* The centralized setup ensures that all test cases start with a consistent, pre-configured environment, minimizing setup duplication across tests and ensuring that any changes to the testing environment need only be made in one place.

This comprehensive setup is not just about ensuring that the bot functions as expected; it's also about ensuring that it does so in a way that is isolated from real-world side effects, consistent across all tests, and robust against changes in external dependencies. This meticulous approach to testing is what helps maintain the reliability and stability of the "Discord Service Notifier" in dynamic real-world scenarios.

## Test Case 1: Add Account

### Description

This test case evaluates the functionality of the “!add\_account” command within the Discord bot. It ensures that the bot can correctly process account addition requests, handling both successful and erroneous scenarios. The test verifies that valid account information is accepted and added to the system and that appropriate error messages are displayed when an account cannot be added.

### Steps

1. **Command Reception:** A mock user message is parsed to simulate the command input (!add\_account username password website).
2. **Boundary Layer Activation:** The !add\_account command is triggered in the bot, and the parsed data is processed.
3. **Control Layer Processing:** The control layer receives the command details from the boundary and attempts to add the account via the AccountDAO.
4. **Database Interaction:** The mocked AccountDAO.add\_account method is called to simulate database interaction (both successful addition and failure scenario).
5. **User Feedback:** Based on the mock database response, the boundary layer sends an appropriate message back to the user indicating the result of the command.

### Test Data

* Valid Account Data:
  + Username: "testuser"
  + Password: "password123"
  + Website: "example.com"
* **Invalid Account Data:** Simulated by mocking the database method to return a failure.

### Expected Outcomes

* Success Scenario:
  + The bot responds with "Account for example.com added successfully." indicating that the account has been added to the database.
  + The database method for adding an account is called with correct parameters.
* Error Scenario:
  + The bot responds with "Failed to add account for example.com." indicating an issue with the account addition process.
  + The database method simulates failure without changing the database state.

### Output

A screenshot of a computer program

Description automatically generated

### Source Code

from unittest.mock import patch

import logging, unittest

from test\_init import BaseTestSetup, CustomTextTestRunner

class TestAddAccountCommand(BaseTestSetup):

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('DataObjects.AccountDAO.AccountDAO.add\_account')

    async def test\_add\_account\_success(self, mock\_add\_account, mock\_parse\_user\_message):

        """Test the add\_account command when it succeeds."""

        # Simulate parsing user message and extracting command parameters

        mock\_parse\_user\_message.return\_value = ["add\_account", "testuser", "password123", "example.com"]

        # Simulate successful account addition in the database

        mock\_add\_account.return\_value = True

        # Triggering the command within the bot

        command = self.bot.get\_command("add\_account")

        await command(self.ctx)

        # Validate that the success message is correctly sent to the user

        self.ctx.send.assert\_called\_with("Account for example.com added successfully.")

        logging.info("Verified successful account addition - database addition simulated and feedback provided.")

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('DataObjects.AccountDAO.AccountDAO.add\_account')

    async def test\_add\_account\_error(self, mock\_add\_account, mock\_parse\_user\_message):

        """Test the add\_account command when it encounters an error."""

        # Setup for receiving command and failing to add account

        mock\_parse\_user\_message.return\_value = ["add\_account", "testuser", "password123", "example.com"]

        mock\_add\_account.return\_value = False

        # Command execution with expected failure

        command = self.bot.get\_command("add\_account")

        await command(self.ctx)

        # Ensuring error feedback is correctly relayed to the user

        self.ctx.send.assert\_called\_with("Failed to add account for example.com.")

        logging.info("Verified error handling during account addition - simulated database failure and error feedback.")

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

    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 2: Delete Account

### Description

This test case ensures the proper functioning of the “!delete\_account” command within the Discord bot. It checks that the system can correctly handle requests to delete an account using a given account ID. The test validates both positive and negative scenarios, verifying that accounts are deleted when correct IDs are provided, and appropriate error messages are handled when deletion fails due to incorrect IDs or other issues.

### Steps:

1. **Command Reception:** A mock user message (!delete\_account 123) is parsed to simulate the command input. This includes extracting the command keyword and associated parameters (account ID).
2. **Boundary Layer Activation:** The !delete\_account command is triggered in the bot. The boundary layer interprets the command and prepares data (account ID) for further processing.
3. **Control Layer Processing:** The control layer receives the account ID from the boundary and attempts to delete the account via the AccountDAO. It either confirms the deletion or handles failures.
4. **Database Interaction Simulation:** The mocked AccountDAO.delete\_account method is invoked to simulate the interaction with the database. This method is configured to return success or failure based on the input account ID.
5. **User Feedback Communication:** Depending on the outcome from the mock database interaction, the boundary layer communicates with the user, either confirming successful deletion or explaining the failure.

### Test Data

* **Valid Account ID for Successful Deletion:** "123"
* **Invalid Account ID for Error Scenario:** "999"

### Expected Outcomes:

* **Success Scenario:**
  + Expected Bot Response: "Account with ID 123 deleted successfully."
  + Interaction with AccountDAO.delete\_account is executed with correct parameters.
* **Error Scenario:**
  + Expected Bot Response: "Failed to delete account with ID 999."
  + Interaction with AccountDAO.delete\_account simulates failure without changing the database state.

### Output

A screenshot of a computer program

Description automatically generated

### Source Code

from unittest.mock import patch

import logging, unittest

from test\_init import BaseTestSetup, CustomTextTestRunner

class TestDeleteAccountCommand(BaseTestSetup):

@patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

@patch('DataObjects.AccountDAO.AccountDAO.delete\_account')

async def test\_delete\_account\_success(self, mock\_delete\_account, mock\_parse\_user\_message):

"""Test the delete\_account command when it succeeds."""

logging.info("\n\nUnit test for delete account starting for positive test:\n")

logging.info("Starting test: test\_delete\_account\_success")

# Mock setup to simulate user input parsing and successful account deletion

mock\_delete\_account.return\_value = True

mock\_parse\_user\_message.return\_value = ["delete\_account", "123"]

# Triggering the delete account command in the bot

command = self.bot.get\_command("delete\_account")

await command(self.ctx)

# Checking if the success message was correctly sent to the user

expected\_message = "Account with ID 123 deleted successfully."

self.ctx.send.assert\_called\_with(expected\_message)

logging.info("Verified successful account deletion.\n")

@patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

@patch('DataObjects.AccountDAO.AccountDAO.delete\_account')

async def test\_delete\_account\_error(self, mock\_delete\_account, mock\_parse\_user\_message):

"""Test the delete\_account command when it encounters an error."""

logging.info("\n\nUnit test for delete account starting for negative test:\n")

logging.info("Starting test: test\_delete\_account\_error")

# Mock setup for testing account deletion failure

mock\_delete\_account.return\_value = False

mock\_parse\_user\_message.return\_value = ["delete\_account", "999"]

# Executing the delete account command with expected failure

command = self.bot.get\_command("delete\_account")

await command(self.ctx)

# Checking if the error message was correctly relayed to the user

expected\_message = "Failed to delete account with ID 999."

self.ctx.send.assert\_called\_with(expected\_message)

logging.info("Verified error handling during account deletion.\n")

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

# Custom test runner to highlight the test results

unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 3: Fetch All Accounts

### Description

This test case verifies the functionality of the !fetch\_all\_accounts command within the Discord bot. It ensures that the bot can accurately retrieve and display all stored account details, handling both scenarios where accounts are available and where an unexpected error might occur, such as a database failure.

### Steps

1. **Command Reception**:
   * A mock user message simulating the !fetch\_all\_accounts command is received and parsed to trigger the command within the bot.
2. **Boundary Layer Activation**:
   * The command triggers the bot to process the fetching of account details through its boundary layer, where the command and parameters are parsed and passed.
3. **Control Layer Processing**:
   * The control layer receives the request to fetch all accounts and interacts with the database access object (DAO) to retrieve the data.
4. **Database Interaction**:
   * The AccountDAO.fetch\_all\_accounts method is invoked, which interacts with the database to fetch all stored account information. This interaction is mocked to return either a set list of accounts or to throw an error simulating a database failure.
5. **User Feedback**:
   * The boundary layer processes the results or error from the control layer and sends an appropriate message back to the user detailing the accounts or reporting an error.

### Test Data

* **No specific inputs required** as the command does not need parameters to fetch all accounts.

### Expected Outcomes

* **Success Scenario**:
  + The bot responds with a detailed list of all accounts formatted for user readability.
  + The DAO method for fetching accounts is called, and it successfully retrieves account data.
* **Error Scenario**:
  + The bot responds with an error message indicating failure to fetch account details.
  + The DAO method simulates an exception to mimic database access issues.

### Output

A screenshot of a computer program

Description automatically generated

### Source Code

import logging, unittest

from unittest.mock import patch

from test\_init import BaseTestSetup, CustomTextTestRunner

"""

File: test\_!fetch\_all\_accounts.py

Purpose: Unit tests for the !fetch\_all\_accounts command in the Discord bot.

The tests validate both successful and error scenarios, ensuring accounts are fetched successfully or errors are handled properly.

"""

class TestFetchAllAccountsCommand(BaseTestSetup):

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('DataObjects.AccountDAO.AccountDAO.fetch\_all\_accounts')

    async def test\_fetch\_all\_accounts\_success(self, mock\_fetch\_all\_accounts, mock\_parse\_user\_message):

        """Test the fetch\_all\_accounts command when it succeeds."""

        logging.info("Starting test: test\_fetch\_all\_accounts\_success")

        # Mock the DAO function to simulate database returning account data

        mock\_fetch\_all\_accounts.return\_value = [("1", "testuser", "password", "example.com")]

        # Mock the message parsing to simulate command input handling

        mock\_parse\_user\_message.return\_value = ["fetch\_all\_accounts"]

        # Retrieve the command function from the bot commands

        command = self.bot.get\_command("fetch\_all\_accounts")

        # Ensure the command is properly registered and retrieved

        self.assertIsNotNone(command)

        # Execute the command and pass the context object

        await command(self.ctx)

        # Define expected user message output

        expected\_message = "Accounts:\nID: 1, Username: testuser, Password: password, Website: example.com"

        # Assert the expected output was sent to the user

        self.ctx.send.assert\_called\_with(expected\_message)

        logging.info("Verified successful fetch.")

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('DataObjects.AccountDAO.AccountDAO.fetch\_all\_accounts')

    async def test\_fetch\_all\_accounts\_error(self, mock\_fetch\_all\_accounts, mock\_parse\_user\_message):

        """Test the fetch\_all\_accounts command when it encounters an error."""

        logging.info("Starting test: test\_fetch\_all\_accounts\_error")

        # Mock the DAO function to raise an exception simulating a database error

        mock\_fetch\_all\_accounts.side\_effect = Exception("Database error")

        # Mock the message parsing to simulate command input handling

        mock\_parse\_user\_message.return\_value = ["fetch\_all\_accounts"]

        # Retrieve the command function from the bot commands

        command = self.bot.get\_command("fetch\_all\_accounts")

        # Ensure the command is properly registered and retrieved

        self.assertIsNotNone(command)

        # Execute the command and pass the context object

        await command(self.ctx)

        # Assert the correct error message was sent to the user

        self.ctx.send.assert\_called\_with("Error fetching accounts.")

        logging.info("Verified error handling.")

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

    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 4: Fetch Account by Website

### Description

This test case evaluates the functionality of the !fetch\_account\_by\_website command within the Discord bot. It confirms that the bot can accurately retrieve account details associated with a specific website, handling both scenarios where the account exists and where it does not.

### Steps

1. **Command Reception**: Simulate the command input for fetching an account by a specific website.
2. **Boundary Layer Activation**: The !fetch\_account\_by\_website command is invoked in the bot, processing the user's request.
3. **Control Layer Processing**: The control layer queries the DAO using the specified website to retrieve account details.
4. **Data Retrieval**: The mocked fetch\_account\_by\_website method is called to simulate retrieving data from the database.
5. **User Feedback**: Depending on the simulation result, an appropriate message is sent back to the user indicating either success or failure in fetching the account.

### Test Data

* **Valid Website**: "example.com"
* **Non-existent Website**: "nonexistent.com"

### Expected Outcomes:

* **Success Scenario**: The bot should respond with the account details ("testuser", "password123") for a valid website.
* **Error Scenario**: The bot should respond with "No account found for nonexistent.com." for a website that does not exist in the database.

### Output

A screenshot of a computer program

Description automatically generated

### Source Code

import logging, unittest

from unittest.mock import patch

from test\_init import BaseTestSetup, CustomTextTestRunner

"""

File: test\_!fetch\_account\_by\_website.py

Purpose: Unit tests for the !fetch\_account\_by\_website command in the Discord bot.

Tests the retrieval of account details based on website input, handling both found and not found scenarios.

"""

class TestFetchAccountByWebsiteCommand(BaseTestSetup):

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('DataObjects.AccountDAO.AccountDAO.fetch\_account\_by\_website')

    async def test\_fetch\_account\_by\_website\_success(self, mock\_fetch\_account\_by\_website, mock\_parse\_user\_message):

        """Test the fetch\_account\_by\_website command when it succeeds."""

        logging.info("Starting test: test\_fetch\_account\_by\_website\_success")

         # Mock setup for successful account fetch

        mock\_fetch\_account\_by\_website.return\_value = ("testuser", "password123")

        mock\_parse\_user\_message.return\_value = ["fetch\_account\_by\_website", "example.com"]

        # Command execution

        command = self.bot.get\_command("fetch\_account\_by\_website")

        self.assertIsNotNone(command)

        # Expected successful fetch response

        await command(self.ctx)

        expected\_message = "testuser", "password123"

        self.ctx.send.assert\_called\_with(expected\_message)

        logging.info("Verified successful account fetch.")

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('DataObjects.AccountDAO.AccountDAO.fetch\_account\_by\_website')

    async def test\_fetch\_account\_by\_website\_error(self, mock\_fetch\_account\_by\_website, mock\_parse\_user\_message):

        """Test the fetch\_account\_by\_website command when it encounters an error."""

        logging.info("Starting test: test\_fetch\_account\_by\_website\_error")

        # Mock setup for failure in finding account

        mock\_fetch\_account\_by\_website.return\_value = None

        mock\_parse\_user\_message.return\_value = ["fetch\_account\_by\_website", "nonexistent.com"]

        # Command execution for nonexistent account

        command = self.bot.get\_command("fetch\_account\_by\_website")

        self.assertIsNotNone(command)

        # Expected error message response

        await command(self.ctx)

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

        self.ctx.send.assert\_called\_with(expected\_message)

        logging.info("Verified error handling for nonexistent account.")

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

    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 5: Launch Browser

### Description

This test case assesses the functionality of the !launch\_browser command within the Discord bot. It ensures that the bot can effectively initiate a browser session, accommodating both successful and erroneous scenarios. The test verifies that the command triggers the correct actions in the system and that any potential errors during the launch process are managed gracefully.

### Steps

1. **Command Reception**: A mock user message is parsed to simulate the input command (!launch\_browser).
2. **Boundary Layer Activation**: The !launch\_browser command is activated within the bot, and the parsed data is processed.
3. **Control Layer Processing**: The control layer instructs the BrowserEntity to initiate a browser launch.
4. **Execution**: The mocked method BrowserEntity.launch\_browser is called to simulate the action of launching the browser.
5. **User Feedback**: Depending on the simulated outcome (success or failure), the boundary layer sends an appropriate message back to the user detailing the result of the command.

### Test Data

* Command Input: !launch\_browser

### Expected Outcomes:

* **Success Scenario**:
  + The bot responds with "Browser launched.", indicating successful initiation of the browser.
  + The method for launching the browser is invoked with the correct parameters.
* **Error Scenario**:
  + The bot responds with "Failed to launch browser", indicating an issue during the browser launch process.
  + The method simulates a launch failure, triggering the appropriate error handling.

### Output

A screenshot of a computer program

Description automatically generated

### Source Code

import logging, unittest

from unittest.mock import patch

from test\_init import BaseTestSetup, CustomTextTestRunner

class TestLaunchBrowserCommand(BaseTestSetup):

@patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

@patch('entity.BrowserEntity.BrowserEntity.launch\_browser')

async def test\_launch\_browser\_success(self, mock\_launch\_browser, mock\_parse\_user\_message):

"""Test the launch\_browser command when it succeeds."""

logging.info("Starting test: test\_launch\_browser\_success")

# Simulate successful browser launch

mock\_launch\_browser.return\_value = "Browser launched."

# Mock the parsed message to return the expected command

mock\_parse\_user\_message.return\_value = ["launch\_browser"]

# Retrieve the launch\_browser command from the bot

command = self.bot.get\_command("launch\_browser")

self.assertIsNotNone(command)

# Call the command without arguments (since GlobalState is mocked)

await command(self.ctx)

# Verify the expected message was sent to the user

expected\_message = "Browser launched."

self.ctx.send.assert\_called\_with(expected\_message)

logging.info("Verified successful browser launch.")

@patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

@patch('entity.BrowserEntity.BrowserEntity.launch\_browser')

async def test\_launch\_browser\_error(self, mock\_launch\_browser, mock\_parse\_user\_message):

"""Test the launch\_browser command when it encounters an error."""

logging.info("Starting test: test\_launch\_browser\_error")

# Simulate a failure during browser launch

mock\_launch\_browser.side\_effect = Exception("Failed to launch browser")

# Mock the parsed message to return the expected command

mock\_parse\_user\_message.return\_value = ["launch\_browser"]

# Retrieve the launch\_browser command from the bot

command = self.bot.get\_command("launch\_browser")

self.assertIsNotNone(command)

# Call the command without arguments (since GlobalState is mocked)

await command(self.ctx)

# Verify the correct error message is sent

self.ctx.send.assert\_called\_with("Failed to launch browser") # Error message handled

logging.info("Verified error handling during browser launch.")

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

unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 6: Close Browser

### Description

This test case evaluates the functionality of the !close\_browser command within the Discord bot. It ensures that the bot can correctly terminate a browser session, accommodating both successful and erroneous scenarios. The test verifies that the command triggers the correct actions in the system for closing the browser and that any potential errors during the closure process are managed gracefully.

### Steps

1. **Command Reception**: A mock user message is parsed to simulate the input command (!close\_browser).
2. **Boundary Layer Activation**: The !close\_browser command is activated within the bot, and the parsed data is processed.
3. **Control Layer Processing**: The control layer instructs the BrowserEntity to terminate the browser session.
4. **Execution**: The mocked method BrowserEntity.close\_browser is called to simulate the action of closing the browser.
5. **User Feedback**: Depending on the simulated outcome (success or failure), the boundary layer sends an appropriate message back to the user detailing the result of the command.

### Test Data

* Command Input: !close\_browser

### Expected Outcomes

* **Success Scenario**:
  + The bot responds with "Browser closed.", indicating successful termination of the browser session.
  + The method for closing the browser is invoked with the correct parameters.
* **Error Scenario**:
  + The bot responds with "Failed to close browser", indicating an issue during the browser closure process.
  + The method simulates a closure failure, triggering the appropriate error handling.

### Output

A computer screen with white text

Description automatically generated

### Source Code

import logging, unittest

from unittest.mock import patch

from test\_init import BaseTestSetup, CustomTextTestRunner

class TestCloseBrowserCommand(BaseTestSetup):

@patch('DataObjects.global\_vars.GlobalState.parse\_user\_message') # Mock the global state parsing

@patch('entity.BrowserEntity.BrowserEntity.close\_browser')

async def test\_close\_browser\_success(self, mock\_close\_browser, mock\_parse\_user\_message):

"""Test the close\_browser command when it succeeds."""

logging.info("Starting test: test\_close\_browser\_success")

# Mock the parsed user message

mock\_parse\_user\_message.return\_value = ["close\_browser"]

# Simulate successful browser closure

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

# Retrieve the close\_browser command from the bot

command = self.bot.get\_command("close\_browser")

self.assertIsNotNone(command)

# Call the command

await command(self.ctx)

# Verify the expected message was sent to the user

expected\_message = "Browser closed."

self.ctx.send.assert\_called\_with(expected\_message)

logging.info("Verified successful browser closure.")

@patch('DataObjects.global\_vars.GlobalState.parse\_user\_message') # Mock the global state parsing

@patch('entity.BrowserEntity.BrowserEntity.close\_browser')

async def test\_close\_browser\_error(self, mock\_close\_browser, mock\_parse\_user\_message):

"""Test the close\_browser command when it encounters an error."""

logging.info("Starting test: test\_close\_browser\_error")

# Mock the parsed user message

mock\_parse\_user\_message.return\_value = ["close\_browser"]

# Simulate a failure during browser closure

mock\_close\_browser.side\_effect = Exception("Failed to close browser")

# Retrieve the close\_browser command from the bot

command = self.bot.get\_command("close\_browser")

this.assertIsNotNone(command)

# Call the command

await command(self.ctx)

# Verify the correct error message is sent

self.ctx.send.assert\_called\_with("Failed to close browser") # Error message handled

logging.info("Verified error handling during browser closure.")

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

# Use the custom test runner to display 'Unit test passed'

unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 7: Navigate to Website

### Description

This test case evaluates the functionality of the !navigate\_to\_website command within the Discord bot. It ensures that the bot can correctly initiate navigation to a specified website URL, handling both successful outcomes and erroneous scenarios. The test verifies that valid URLs trigger the correct system behavior and that errors are managed effectively if the navigation fails.

### Steps

1. **Command Reception**: A mock user message is parsed to simulate the input command (!navigate\_to\_website url).
2. **Boundary Layer Activation**: The !navigate\_to\_website command is activated within the bot, and the parsed data including the URL is processed.
3. **Control Layer Processing**: The control layer instructs the BrowserEntity to navigate to the specified URL.
4. **Execution**: The mocked method BrowserEntity.navigate\_to\_website is called to simulate the action of navigating to the URL.
5. **User Feedback**: Depending on the simulated outcome (success or error), the boundary layer sends an appropriate message back to the user detailing the result of the command.

### Test Data

* Valid URL: https://example.com
* Invalid URL: <https://invalid-url.com>

### Expected Outcomes

* **Success Scenario**:
  + The bot responds with "Navigated to <https://example.com>.", indicating successful navigation to the website.
  + The navigation method is invoked with the correct URL.
* **Error Scenario**:
  + The bot responds with "Failed to navigate to the website.", indicating an issue during the navigation process.
  + The method simulates a navigation failure, triggering the appropriate error handling.

### Output

A screenshot of a computer program

Description automatically generated

### Source Code

import logging, unittest

from unittest.mock import patch, AsyncMock

from test\_init import BaseTestSetup, CustomTextTestRunner

class TestNavigateToWebsiteCommand(BaseTestSetup):

@patch('DataObjects.global\_vars.GlobalState.parse\_user\_message') # Mock the parsing of user message

@patch('entity.BrowserEntity.BrowserEntity.navigate\_to\_website') # Mock the browser navigation

async def test\_navigate\_to\_website\_success(self, mock\_receive\_command, mock\_parse\_user\_message):

"""Test the navigate\_to\_website command when it succeeds."""

logging.info("Starting test: test\_navigate\_to\_website\_success")

# Simulate successful website navigation

mock\_receive\_command.return\_value = "Navigated to https://example.com."

# Setup the expected command and URL

mock\_parse\_user\_message.return\_value = ["navigate\_to\_website", "https://example.com"]

# Retrieve the navigate\_to\_website command from the bot

command = self.bot.get\_command("navigate\_to\_website")

self.assertIsNotNone(command)

# Execute the command with the mocked URL

await command(self.ctx)

# Verify that the navigation message was sent to the user correctly

expected\_message = "Navigated to https://example.com."

self.ctx.send.assert\_called\_with(expected\_message)

logging.info("Verified successful website navigation.")

@patch('DataObjects.global\_vars.GlobalState.parse\_user\_message') # Mock the parsing of user message

@patch('entity.BrowserEntity.BrowserEntity.navigate\_to\_website') # Mock the browser navigation

async def test\_navigate\_to\_website\_error(self, mock\_receive\_command, mock\_parse\_user\_message):

"""Test the navigate\_to\_website command when it encounters an error."""

logging.info("Starting test: test\_navigate\_to\_website\_error")

# Simulate a failure during website navigation

mock\_receive\_command.side\_effect = Exception("Failed to navigate to the website.")

# Setup the expected command and invalid URL

mock\_parse\_user\_message.return\_value = ["navigate\_to\_website", "https://invalid-url.com"]

# Execute the command with the mocked invalid URL

command = self.bot.get\_command("navigate\_to\_website")

this.assertIsNotNone(command)

# Execute the command with the error scenario

await command(self.ctx)

# Verify that the error message was handled and sent correctly

self.ctx.send.assert\_called\_with("Failed to navigate to the website.")

logging.info("Verified error handling during website navigation.")

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

# Use the custom test runner to display 'Unit test passed'

unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 8: Login

### Description

This test case evaluates the functionality of the !login command within the Discord bot, specifically targeting the ability to log in to a website. It ensures that the bot can correctly handle user login requests, managing both successful logins and error scenarios where login details are incorrect or the website is not accessible.

### Steps

1. **Command Reception**: The user command !login bestbuy is received and parsed to simulate user interaction.
2. **Boundary Layer Processing**: The parsed command triggers the !login function in the boundary layer, which then passes the request to the control layer.
3. **Control Layer Execution**: The control layer processes the login request by interacting with the LoginControl, which attempts to authenticate the user at the BestBuy website.
4. **Response Handling**: Depending on the response from the LoginControl, the boundary layer sends an appropriate message back to the user, indicating the outcome of the login attempt.

### Test Data

* Successful Login: Username and password are correct.
* Failed Login: Username is correct, but the password is incorrect or the URL is wrong.

### Expected Outcomes

* **Successful Login**:
  + The bot responds with "Login successful." indicating that the user has been successfully logged in.
  + The system logs this as a successful login attempt.
* **Failed Login**:
  + The bot responds with "Failed to login. No account found." indicating a failure in the login process due to incorrect credentials or website issues.
  + The system logs this as a failed login attempt.

### Output

A screen shot of a computer program

Description automatically generated

### Source Code

import logging, unittest

from unittest.mock import patch, AsyncMock

from test\_init import BaseTestSetup, CustomTextTestRunner

class TestLoginCommand(BaseTestSetup):

@patch('DataObjects.global\_vars.GlobalState.parse\_user\_message') # Mock parsing of user command

@patch('control.LoginControl.LoginControl.receive\_command') # Mock the command receiver in the Login control

async def test\_login\_success(self, mock\_receive\_command, mock\_parse\_user\_message):

"""Test the login command when it succeeds."""

logging.info("Starting test: test\_login\_success")

# Simulate parsing of user message for successful login

mock\_parse\_user\_message.return\_value = ["login", "bestbuy"]

# Simulate a successful login response

mock\_receive\_command.return\_value = "Login successful."

# Execute the login command

command = self.bot.get\_command("login")

await command(self.ctx)

# Verify successful login message

expected\_message = "Login successful."

self.ctx.send.assert\_called\_with(expected\_message)

logging.info("Verified successful login.")

@patch('DataObjects.global\_vars.GlobalState.parse\_user\_message') # Mock parsing of user command

@patch('control.LoginControl.LoginControl.receive\_command') # Mock the command receiver in the Login control

async def test\_login\_error(self, mock\_receive\_command, mock\_parse\_user\_message):

"""Test the login command when it encounters an error."""

logging.info("Starting test: test\_login\_error")

# Simulate parsing of user message for failed login

mock\_parse\_user\_message.return\_value = ["login", "nonexistent.com"]

# Simulate a login failure response

mock\_receive\_command.side\_effect = Exception("Failed to login. No account found.")

# Execute the login command

command = self.bot.get\_command("login")

await command(self.ctx)

# Verify error message handling

expected\_message = "Failed to login. No account found."

self.ctx.send.assert\_called\_with(expected\_message)

logging.info("Verified error handling during login.")

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

unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 9: Stop Bot

### Description

This test case evaluates the functionality of the !stop\_bot command within the Discord bot. It ensures that the bot can correctly process shutdown requests, managing both successful shutdowns and handling errors during the shutdown process. This command is crucial for safely terminating the bot's operations without leaving processes hanging.

### Steps

1. **Command Reception**: A mock user command !stop\_bot is simulated to trigger the bot shutdown process.
2. **Boundary Layer Processing**: Upon receiving the command, the boundary layer passes the shutdown request to the control layer.
3. **Control Layer Execution**: The control layer attempts to execute the shutdown by calling the bot's internal shutdown methods.
4. **Response Handling**: Depending on the execution outcome, the boundary layer sends an appropriate message back to the user, either confirming the shutdown or reporting an error.

### Test Data

* No specific input data other than the command trigger.

### Expected Outcomes

* **Successful Shutdown**:
  + The bot responds with "The bot is shutting down..." indicating that the shutdown process has been initiated and completed successfully.
  + The bot's internal close methods are invoked once.
* **Error Scenario**:
  + The bot responds with an error message "Error stopping bot", indicating there was an issue during the shutdown process.
  + The system logs the error and still attempts to close the bot properly.

### Output

A computer screen shot of a program

Description automatically generated

### Source Code

import logging, unittest

from unittest.mock import AsyncMock, patch

from test\_init import BaseTestSetup, CustomTextTestRunner

class TestStopBotCommand(BaseTestSetup):

@patch('DataObjects.global\_vars.GlobalState.parse\_user\_message') # Mocks the parsing of user commands

@patch('control.StopControl.StopControl.receive\_command', new\_callable=AsyncMock) # Mocks the stop control command receiver

async def test\_stop\_bot\_success(self, mock\_receive\_command, mock\_parse\_user\_message):

"""Test the stop\_bot command when it successfully shuts down."""

logging.info("Starting test: test\_stop\_bot\_success")

# Mocks returning a success message

mock\_receive\_command.return\_value = "The bot is shutting down..."

# Simulates receiving the stop command

mock\_parse\_user\_message.return\_value = ["stop\_bot"]

# Triggers the stop\_bot command

command = self.bot.get\_command("stop\_bot")

await command(self.ctx)

# Asserts that the shutdown message was communicated

self.ctx.send.assert\_called\_once\_with("The bot is shutting down...")

logging.info("Verified that the shutdown message was sent to the user.")

# Confirms that the bot's close method was called

mock\_receive\_command.assert\_called\_once()

logging.info("Verified that the bot's close method was called once.")

@patch('DataObjects.global\_vars.GlobalState.parse\_user\_message') # Mocks the parsing of user commands

@patch('control.StopControl.StopControl.receive\_command', new\_callable=AsyncMock) # Mocks the stop control command receiver

async def test\_stop\_bot\_error(self, mock\_receive\_command, mock\_parse\_user\_message):

"""Test the stop\_bot command when it encounters an error during shutdown."""

logging.info("Starting test: test\_stop\_bot\_error")

# Sets up an exception to simulate an error during shutdown

exception\_message = "Error stopping bot"

mock\_receive\_command.side\_effect = Exception(exception\_message)

mock\_parse\_user\_message.return\_value = ["stop\_bot"]

# Triggers the stop\_bot command expecting an error

command = self.bot.get\_command("stop\_bot")

await command(self.ctx)

# Asserts that the initial command was recognized before the error

self.ctx.send.assert\_called\_with('Command recognized, passing data to control.')

logging.info("Verified error handling during bot shutdown.")

# Checks that an error was raised and the close method was attempted

mock\_receive\_command.assert\_called\_once\_with("stop\_bot", self.ctx)

logging.info("Verified that the bot's close method was attempted even though it raised an error.")

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

unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 10: Project Help

### Description

This test case evaluates the functionality of the !project\_help command within the Discord bot. It aims to ensure that the bot correctly provides a comprehensive help message listing all available commands, facilitating user interaction by guiding them through each command's purpose and usage. This test also checks the bot's ability to handle potential errors during the execution of the help command.

### Steps

1. **Command Reception**: A mock user command !project\_help is simulated to trigger the help message process.
2. **Boundary Layer Processing**: Upon receiving the command, the boundary layer handles the retrieval and formatting of the help message.
3. **Response Generation**: The formatted help message is sent back to the user, providing detailed instructions on how to use each command available in the bot.
4. **Error Handling**: The test also simulates an error scenario where the help message cannot be sent, ensuring that the bot can gracefully handle such exceptions.

### Test Data

* No specific input data other than the command trigger.

### Expected Outcomes

* **Successful Help Message Retrieval**:
  + The bot responds with a detailed help message listing all available commands and their descriptions.
* **Error Scenario**:
  + The bot encounters an error while attempting to send the help message and handles the exception properly, ensuring that the system remains stable.

### Output

A computer screen with text on it

Description automatically generated

### Source Code

import logging, unittest

from unittest.mock import patch, AsyncMock, call

from test\_init import BaseTestSetup, CustomTextTestRunner

"""

File: test\_!project\_help.py

Purpose: This file contains unit tests for the !project\_help command in the Discord bot.

The tests validate both successful and error scenarios, ensuring the bot provides the correct help message and handles errors properly.

Tests:

- Positive: Simulates the !project\_help command and verifies the correct help message is sent.

- Negative: Simulates an error scenario and ensures the error is handled gracefully.

"""

class TestProjectHelpCommand(BaseTestSetup):

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    async def test\_project\_help\_success(self, mock\_parse\_user\_message):

        """Test the project help command when it successfully returns the help message."""

        logging.info("Starting test: test\_project\_help\_success")

        mock\_parse\_user\_message.return\_value = ["project\_help"]  # Mock the command parsing to return the command

        # Simulate calling the project\_help command

        command = self.bot.get\_command("project\_help")

        self.assertIsNotNone(command, "project\_help command is not registered.")  # Ensure the command is registered

        await command(self.ctx)

        # Define the expected help message from the module

        help\_message = (

                "Here are the available commands:\n"

                "!project\_help - Get help on available commands.\n"

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

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

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

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

                "!launch\_browser - Launch the browser.\n"

                "!close\_browser - Close the browser.\n"

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

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

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

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

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

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

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

                "!stop\_monitoring\_availability - Stop monitoring availability.\n"

                "!stop\_bot - Stop the bot.\n"

            )

        # Check if the correct help message was sent

        self.ctx.send.assert\_called\_with(help\_message)

        logging.info("Verified that the correct help message was sent.")

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    async def test\_project\_help\_error(self, mock\_parse\_user\_message):

        """Test the project help command when it encounters an error during execution."""

        logging.info("Starting test: test\_project\_help\_error")

        mock\_parse\_user\_message.return\_value = ["project\_help"]  # Mock the command parsing to return the command

        # Simulate an error when sending the message

        self.ctx.send.side\_effect = Exception("Error during project\_help execution.")

        command = self.bot.get\_command("project\_help")

        self.assertIsNotNone(command, "project\_help command is not registered.")  # Ensure the command is registered

        with self.assertRaises(Exception):

            await command(self.ctx)

        logging.info("Verified that an error occurred and was handled.")

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

    # Use the custom test runner to display 'Unit test passed'

    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 11: Get Price

### Description

This test case evaluates the functionality of the !get\_price command within the Discord bot. It ensures that the bot can correctly fetch and display the price of a product from a specified URL or gracefully handle any errors that occur during the process. This test helps confirm that users receive accurate and timely pricing information, enhancing the bot's utility in online shopping assistance.

### Steps

1. **Command Reception**: The command !get\_price along with a product URL is simulated to trigger the price retrieval process.
2. **Boundary Layer Parsing**: The command and URL are parsed and validated before passing to the control layer.
3. **Control Layer Processing**: The control layer attempts to fetch the price from the provided URL using a mock function that simulates this action.
4. **Response Formation**: Depending on the outcome of the fetch process, the bot constructs a response indicating either the price of the product or an error message.

### Test Data

* **Successful Price Retrieval**: URL - "<https://example.com>"
* **Error Scenario**: URL - "<https://invalid-url.com>"

### Expected Outcomes

* **Successful Scenario**:
  + The bot responds with "Price found: Price: $199.99", indicating that the price retrieval was successful.
* **Error Scenario**:
  + The bot responds with "Price found: Failed to fetch price", indicating an issue with the price retrieval process.

### Output

A screen shot of a computer program

Description automatically generated

### Source Code

import logging, unittest

from unittest.mock import patch

from test\_init import BaseTestSetup, CustomTextTestRunner

"""

File: test\_!get\_price.py

Purpose: This file contains unit tests for the !get\_price command in the Discord bot.

The tests validate both successful and error scenarios, ensuring that the price is fetched correctly or errors are handled.

"""

class TestGetPriceCommand(BaseTestSetup):

    @patch('control.PriceControl.PriceControl.receive\_command')

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    async def test\_get\_price\_success(self, mock\_parse\_message, mock\_receive\_command):

        """Test the get\_price command when it succeeds."""

        logging.info("Starting test: test\_get\_price\_success")

        # Simulate parsing of user input

        mock\_parse\_message.return\_value = ["get\_price", "https://example.com"]

        # Simulate successful price fetch

        mock\_receive\_command.return\_value = "Price: $199.99"

        # Retrieve the get\_price command from the bot

        command = self.bot.get\_command("get\_price")

        self.assertIsNotNone(command)

        # Call the command without passing URL (since parsing handles it)

        await command(self.ctx)

        # Verify the expected message was sent to the user

        self.ctx.send.assert\_called\_with("Price found: Price: $199.99")

        logging.info("Verified successful price fetch.")

    @patch('control.PriceControl.PriceControl.receive\_command')

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    async def test\_get\_price\_error(self, mock\_parse\_message, mock\_receive\_command):

        """Test the get\_price command when it encounters an error."""

        logging.info("Starting test: test\_get\_price\_error")

        # Simulate parsing of user input

        mock\_parse\_message.return\_value = ["get\_price", "https://invalid-url.com"]

        # Simulate a failure during price fetch

        mock\_receive\_command.return\_value = "Failed to fetch price"

        # Retrieve the get\_price command from the bot

        command = self.bot.get\_command("get\_price")

        self.assertIsNotNone(command)

        # Call the command without passing additional URL argument (parsing handles it)

        await command(self.ctx)

        # Verify the correct error message is sent

        self.ctx.send.assert\_called\_with("Price found: Failed to fetch price")

        logging.info("Verified error handling during price fetch.")

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

    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 12: Start Monitoring Price

### Description

This test case evaluates the functionality of the !start\_monitoring\_price command within the Discord bot. It checks whether the bot can successfully initiate price monitoring for a specified product URL at given intervals, and if it can properly handle cases where the monitoring cannot be initiated due to errors.

### Steps

1. **Command Reception**: Simulate the reception of the !start\_monitoring\_price command along with a URL and monitoring interval.
2. **Command Parsing**: The incoming message is parsed to extract the necessary parameters (URL and interval).
3. **Monitoring Initiation**: Attempt to start monitoring the specified URL at the provided interval.
4. **Feedback Provision**: Provide feedback to the user indicating whether the monitoring was successfully started or if an error occurred.

### Test Data

* **Successful Monitoring**:
  + URL: "<https://example.com>"
  + Interval: 20 minutes
* **Monitoring Initiation Failure**:
  + URL: "<https://invalid-url.com>"
  + Interval: 20 minutes

### Expected Outcomes

* **Success Scenario**: The user should receive a message stating "Monitoring started for <https://example.com>." This confirms that the monitoring process was initiated correctly.
* **Error Scenario**: The user should receive a message stating "Failed to start monitoring." This indicates that there was an issue with initiating the monitoring, which could be due to an invalid URL or other internal issues.

### Output

A screen shot of a computer program

Description automatically generated

### Source Code

import logging, unittest

from unittest.mock import patch, AsyncMock

from test\_init import BaseTestSetup, CustomTextTestRunner

"""

File: test\_!start\_monitoring\_price.py

Purpose: This file contains unit tests for the !start\_monitoring\_price command in the Discord bot.

The tests validate both successful and error scenarios, ensuring that the bot starts monitoring prices or handles errors gracefully.

Tests:

- Positive: Simulates the !start\_monitoring\_price command and verifies the monitoring is initiated successfully.

- Negative: Simulates an error during the initiation of price monitoring and ensures it is handled gracefully.

"""

class TestStartMonitoringPriceCommand(BaseTestSetup):

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('control.PriceControl.PriceControl.receive\_command')

    async def test\_start\_monitoring\_price\_success(self, mock\_receive\_command, mock\_parse\_user\_message):

        """Test the start\_monitoring\_price command when it succeeds."""

        logging.info("Starting test: test\_start\_monitoring\_price\_success")

        # Mock the parsed message to return the expected command and parameters

        mock\_parse\_user\_message.return\_value = ["start\_monitoring\_price", "https://example.com", "20"]

        # Simulate successful price monitoring start

        mock\_receive\_command.return\_value = "Monitoring started for https://example.com."

        # Retrieve the start\_monitoring\_price command from the bot

        command = self.bot.get\_command("start\_monitoring\_price")

        self.assertIsNotNone(command)

        # Call the command without explicit parameters due to mocked GlobalState

        await command(self.ctx)

        # Verify the expected message was sent to the user

        expected\_message = "Monitoring started for https://example.com."

        self.ctx.send.assert\_called\_with(expected\_message)

        logging.info("Verified successful price monitoring start.")

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('control.PriceControl.PriceControl.receive\_command')

    async def test\_start\_monitoring\_price\_error(self, mock\_receive\_command, mock\_parse\_user\_message):

        """Test the start\_monitoring\_price command when it encounters an error."""

        logging.info("Starting test: test\_start\_monitoring\_price\_error")

        # Mock the parsed message to simulate the command being executed with an invalid URL

        mock\_parse\_user\_message.return\_value = ["start\_monitoring\_price", "https://invalid-url.com", "20"]

        # Simulate a failure during price monitoring start

        mock\_receive\_command.return\_value = "Failed to start monitoring"

        # Retrieve the start\_monitoring\_price command from the bot

        command = self.bot.get\_command("start\_monitoring\_price")

        self.assertIsNotNone(command)

        # Call the command without explicit parameters due to mocked GlobalState

        await command(self.ctx)

        # Verify the correct error message is sent

        expected\_message = "Failed to start monitoring"

        self.ctx.send.assert\_called\_with(expected\_message)

        logging.info("Verified error handling during price monitoring start.")

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

    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 13: Stop Monitoring Price

### Description

This test case assesses the functionality of the !stop\_monitoring\_price command within the Discord bot. It ensures that the bot can properly terminate price monitoring sessions and handles scenarios where no monitoring session exists or an error occurs during the stop process.

### Steps

1. **Command Reception**: Simulate the reception of the !stop\_monitoring\_price command.
2. **Command Parsing**: The incoming message is parsed to determine if it's the correct command.
3. **Monitoring Termination**: Attempt to stop the price monitoring session based on the received command.
4. **Feedback Provision**: Provide feedback to the user indicating whether the price monitoring was successfully stopped or if no session was active.

### Test Data

* **No Active Session**: The scenario where no active price monitoring session exists.
* **Active Session Exists**: The scenario where an active price monitoring session is successfully terminated.

### Expected Outcomes

* **No Active Session Scenario**: The user should receive a message stating "There was no active price monitoring session. Nothing to stop.", indicating that no session was active to be stopped.
* **Successful Termination Scenario**: The user should receive a message stating "Price monitoring stopped successfully.", confirming that the active monitoring session was terminated correctly.

### Output

A computer code on a black background

Description automatically generated

### Source Code

import logging, unittest

from unittest.mock import patch, AsyncMock

from test\_init import BaseTestSetup, CustomTextTestRunner

"""

File: test\_!stop\_monitoring\_price.py

Purpose: This file contains unit tests for the !stop\_monitoring\_price command in the Discord bot.

The tests validate both successful and error scenarios, ensuring that the bot stops monitoring prices or handles errors gracefully.

"""

class TestStopMonitoringPriceCommand(BaseTestSetup):

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('control.PriceControl.PriceControl.receive\_command')

    async def test\_stop\_monitoring\_price\_success\_with\_results(self, mock\_receive\_command, mock\_parse\_user\_message):

        """Test the stop\_monitoring\_price command when monitoring was active and results are returned."""

        logging.info("Starting test: test\_stop\_monitoring\_price\_success\_with\_results")

        # Simulate stopping monitoring and receiving results

        mock\_parse\_user\_message.return\_value = ["stop\_monitoring\_price"]

        mock\_receive\_command.return\_value = "Results for price monitoring:\nPrice: $199.99\nPrice monitoring stopped successfully!"

        # Retrieve the stop\_monitoring\_price command from the bot

        command = self.bot.get\_command("stop\_monitoring\_price")

        self.assertIsNotNone(command)

        # Call the command

        await command(self.ctx)

        # Verify the expected message was sent to the user

        expected\_message = "Results for price monitoring:\nPrice: $199.99\nPrice monitoring stopped successfully!"

        self.ctx.send.assert\_called\_with(expected\_message)

        logging.info("Verified successful stop with results.")

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('control.PriceControl.PriceControl.receive\_command')

    async def test\_stop\_monitoring\_price\_error(self, mock\_receive\_command, mock\_parse\_user\_message):

        """Test the stop\_monitoring\_price command when it encounters an error."""

        logging.info("Starting test: test\_stop\_monitoring\_price\_error")

        # Simulate a failure during price monitoring stop

        mock\_parse\_user\_message.return\_value = ["stop\_monitoring\_price"]

        mock\_receive\_command.return\_value = "Error stopping price monitoring"

        # Retrieve the stop\_monitoring\_price command from the bot

        command = self.bot.get\_command("stop\_monitoring\_price")

        self.assertIsNotNone(command)

        # Call the command

        await command(self.ctx)

        # Verify the correct error message is sent

        expected\_message = "Error stopping price monitoring"

        self.ctx.send.assert\_called\_with(expected\_message)

        logging.info("Verified error handling during price monitoring stop.")

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

    # Use the custom test runner to display 'Unit test passed'

    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 14: Check Availability

### Description

This test case evaluates the !check\_availability command of the Discord bot, which checks the availability of services on specified dates at given URLs. It ensures that the bot can accurately verify availability or report errors when availability data cannot be retrieved.

### Steps

1. **Command Parsing**: Simulate the user input to parse the !check\_availability command along with a URL and date.
2. **Command Execution**: Execute the command to check availability based on the provided URL and date.
3. **Result Handling**: Assess the bot's response, whether it confirms availability or reports no availability.
4. **Feedback to User**: Ensure the bot provides the correct feedback to the user based on the command's execution results.

### Test Data

* **URL (Valid)**: "https://example.com"
* **Date**: "2024-09-30"
* **URL (Invalid)**: <https://invalid-url.com>

### Expected Outcomes

* **Successful Check**: For valid URLs, the bot should confirm availability, e.g., "Available for booking."
* **Error Handling**: For invalid URLs, the bot should respond with "No availability found."

### Output

A computer screen with white text

Description automatically generated

### Source Code

import logging, unittest

from unittest.mock import patch

from test\_init import BaseTestSetup, CustomTextTestRunner

"""

File: test\_!check\_availability.py

Purpose: Unit tests for the !check\_availability command in the Discord bot.

"""

class TestCheckAvailabilityCommand(BaseTestSetup):

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('control.AvailabilityControl.AvailabilityControl.receive\_command')

    async def test\_check\_availability\_success(self, mock\_receive\_command, mock\_parse\_user\_message):

        """Test the check\_availability command when it succeeds."""

        logging.info("Starting test: test\_check\_availability\_success")

        # Mock the parsed message to return the expected command and arguments

        mock\_parse\_user\_message.return\_value = ["check\_availability", "https://example.com", "2024-09-30"]

        # Simulate successful availability check

        mock\_receive\_command.return\_value = "Available for booking."

        command = self.bot.get\_command("check\_availability")

        self.assertIsNotNone(command)

        # Call the command without arguments (since GlobalState is mocked)

        await command(self.ctx)

        expected\_message = "Available for booking."

        self.ctx.send.assert\_called\_with(expected\_message)

        logging.info("Verified successful availability check.")

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('control.AvailabilityControl.AvailabilityControl.receive\_command')

    async def test\_check\_availability\_error(self, mock\_receive\_command, mock\_parse\_user\_message):

        """Test the check\_availability command when it encounters an error."""

        logging.info("Starting test: test\_check\_availability\_error")

        # Mock the parsed message to return the expected command and arguments

        mock\_parse\_user\_message.return\_value = ["check\_availability", "https://invalid-url.com", "2024-09-30"]

        # Simulate error during availability check

        mock\_receive\_command.return\_value = "No availability found."

        command = self.bot.get\_command("check\_availability")

        self.assertIsNotNone(command)

        # Call the command without arguments (since GlobalState is mocked)

        await command(self.ctx)

        expected\_message = "No availability found."

        self.ctx.send.assert\_called\_with(expected\_message)

        logging.info("Verified error handling during availability check.")

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

    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 15: Start Monitoring Availability

### Description

This test case verifies the functionality of the !start\_monitoring\_availability command in the Discord bot, which initiates monitoring of availability for specified services on given URLs and dates. The test ensures that monitoring can be started successfully and that errors are properly reported when monitoring cannot be initiated.

### Steps

1. **Command Parsing**: Mock the parsing of user input to simulate the !start\_monitoring\_availability command along with a URL, date, and frequency.
2. **Command Execution**: Execute the command to start monitoring the availability based on the provided details.
3. **Monitoring Setup**: Check that the command initiates monitoring with the correct parameters.
4. **Feedback to User**: Ensure that the user receives accurate feedback regarding the monitoring status.

### Test Data

* **Valid Setup**:
  + URL: "https://example.com"
  + Date: "2024-09-30"
  + Frequency: 15 minutes
* **Invalid Setup**:
  + URL: "https://invalid-url.com"
  + Date: "2024-09-30"
  + Frequency: 15 minutes

### Expected Outcomes

* **Successful Monitoring Start**: The user should receive a message stating that monitoring has started, e.g., "Monitoring started for <https://example.com>."
* **Failed Monitoring Start**: The user should be informed of the failure, e.g., "Failed to start monitoring."

### Output

A computer screen with text

Description automatically generated

### Source Code

import logging, unittest

from unittest.mock import patch

from test\_init import BaseTestSetup, CustomTextTestRunner

"""

File: test\_!monitor\_availability.py

Purpose: Unit tests for the !monitor\_availability command in the Discord bot.

"""

class TestMonitorAvailabilityCommand(BaseTestSetup):

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('control.AvailabilityControl.AvailabilityControl.receive\_command')

    async def test\_monitor\_availability\_success(self, mock\_receive\_command, mock\_parse\_user\_message):

        """Test the monitor\_availability command when it succeeds."""

        logging.info("Starting test: test\_monitor\_availability\_success")

        # Mock the parsed message to return the expected command and arguments

        mock\_parse\_user\_message.return\_value = ["start\_monitoring\_availability", "https://example.com", "2024-09-30", 15]

        # Simulate successful availability monitoring start

        mock\_receive\_command.return\_value = "Monitoring started for https://example.com."

        command = self.bot.get\_command("start\_monitoring\_availability")

        self.assertIsNotNone(command)

        # Call the command without arguments (since GlobalState is mocked)

        await command(self.ctx)

        expected\_message = "Monitoring started for https://example.com."

        self.ctx.send.assert\_called\_with(expected\_message)

        logging.info("Verified successful availability monitoring start.")

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('control.AvailabilityControl.AvailabilityControl.receive\_command')

    async def test\_monitor\_availability\_error(self, mock\_receive\_command, mock\_parse\_user\_message):

        """Test the monitor\_availability command when it encounters an error."""

        logging.info("Starting test: test\_monitor\_availability\_error")

        # Mock the parsed message to return the expected command and arguments

        mock\_parse\_user\_message.return\_value = ["start\_monitoring\_availability", "https://invalid-url.com", "2024-09-30", 15]

        # Simulate an error during availability monitoring

        mock\_receive\_command.return\_value = "Failed to start monitoring."

        command = self.bot.get\_command("start\_monitoring\_availability")

        self.assertIsNotNone(command)

        # Call the command without arguments (since GlobalState is mocked)

        await command(self.ctx)

        expected\_message = "Failed to start monitoring."

        self.ctx.send.assert\_called\_with(expected\_message)

        logging.info("Verified error handling during availability monitoring.")

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

    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))

## Test Case 16: Stop Monitoring Availability

### Description:

This test case evaluates the !stop\_monitoring\_availability command's functionality within the Discord bot to ensure it can properly terminate ongoing availability monitoring sessions. It confirms that the bot can handle scenarios where no monitoring is active and successfully stop active monitoring sessions.

### Steps

1. **Command Parsing**: Mock the parsing of user input to simulate the receipt of the !stop\_monitoring\_availability command.
2. **Command Execution**: Execute the command to stop monitoring availability.
3. **Monitoring Termination**: Check whether the command properly handles the termination of monitoring, both when sessions are active and when none are active.
4. **User Feedback**: Ensure that the user receives accurate feedback regarding the status of monitoring termination.

### Test Data

* No active sessions to stop.
* Active sessions available for stopping.

### Expected Outcomes

* **No Active Session**: The user receives a message, "There was no active availability monitoring session."
* **Successful Termination**: The user receives a message, "Availability monitoring stopped successfully."

### Output

A computer screen with white text

Description automatically generated

### Source Code

import logging, unittest

from unittest.mock import patch

from test\_init import BaseTestSetup, CustomTextTestRunner

"""

File: test\_!stop\_monitoring\_availability.py

Purpose: Unit tests for the !stop\_monitoring\_availability command in the Discord bot.

"""

class TestStopMonitoringAvailabilityCommand(BaseTestSetup):

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('control.AvailabilityControl.AvailabilityControl.receive\_command')

    async def test\_stop\_monitoring\_availability\_no\_active\_session(self, mock\_receive\_command, mock\_parse\_user\_message):

        """Test the stop\_monitoring\_availability command when no active session exists."""

        logging.info("Starting test: test\_stop\_monitoring\_availability\_no\_active\_session")

        # Mock the parsed message to return the expected command and arguments

        mock\_parse\_user\_message.return\_value = ["stop\_monitoring\_availability"]

        # Simulate no active session scenario

        mock\_receive\_command.return\_value = "There was no active availability monitoring session."

        command = self.bot.get\_command("stop\_monitoring\_availability")

        self.assertIsNotNone(command)

        # Call the command without arguments (since GlobalState is mocked)

        await command(self.ctx)

        expected\_message = "There was no active availability monitoring session."

        self.ctx.send.assert\_called\_with(expected\_message)

        logging.info("Verified no active session stop scenario.")

    @patch('DataObjects.global\_vars.GlobalState.parse\_user\_message')

    @patch('control.AvailabilityControl.AvailabilityControl.receive\_command')

    async def test\_stop\_monitoring\_availability\_success(self, mock\_receive\_command, mock\_parse\_user\_message):

        """Test the stop\_monitoring\_availability command when it succeeds."""

        logging.info("Starting test: test\_stop\_monitoring\_availability\_success")

        # Mock the parsed message to return the expected command and arguments

        mock\_parse\_user\_message.return\_value = ["stop\_monitoring\_availability"]

        # Simulate successful stopping of monitoring

        mock\_receive\_command.return\_value = "Availability monitoring stopped successfully."

        command = self.bot.get\_command("stop\_monitoring\_availability")

        self.assertIsNotNone(command)

        # Call the command without arguments (since GlobalState is mocked)

        await command(self.ctx)

        expected\_message = "Availability monitoring stopped successfully."

        self.ctx.send.assert\_called\_with(expected\_message)

        logging.info("Verified successful availability monitoring stop.")

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

    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))