**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](#_Toc178269463)

[INTRDOCUTION 3](#_Toc178269464)

[TEST PLAN OVERVIEW 4](#_Toc178269465)

[TEST CASES 6](#_Toc178269466)

[Test Case 0: test\_init.py 6](#_Toc178269467)

[Description: 6](#_Toc178269468)

[Purpose and Setup 6](#_Toc178269469)

[Implementation Details 7](#_Toc178269470)

[How It Works 7](#_Toc178269471)

[Test Case 1: Add Account 8](#_Toc178269472)

[Description: 8](#_Toc178269473)

[Steps: 8](#_Toc178269474)

[Test Data 9](#_Toc178269475)

[Expected Outcomes 9](#_Toc178269476)

[Mocking Details 9](#_Toc178269477)

[Source Code: 9](#_Toc178269478)

# 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

### Description:

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.

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 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.

### Mocking Details

* GlobalState.parse\_user\_message is mocked to provide controlled inputs for testing without actual user interaction.
* AccountDAO.add\_account is mocked to simulate database interactions, allowing for testing without actual database access.

### 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.

### Mocking Details:

* GlobalState.parse\_user\_message is mocked to control the input from the user without real user interaction.
* AccountDAO.delete\_account is mocked to simulate database deletion operations, which permits testing without actual database access.

### 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 on.
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.

### Mocking Details:

* **Mock GlobalState.parse\_user\_message** to simulate user command input without real user interaction.
* **Mock AccountDAO.fetch\_all\_accounts** to control database responses without real database access, allowing for the simulation of both success and error conditions.

### 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.

**Mocking Details:**

* GlobalState.parse\_user\_message is mocked to simulate command parsing without actual user input.
* AccountDAO.fetch\_account\_by\_website is mocked to simulate database responses, avoiding direct database access.

### 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))