--- test_!project_help.py ---

```
import sys, os, discord, logging, unittest
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from unittest.mock import AsyncMock, patch, call
from utils.MyBot import MyBot
"""
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.
"""
# Setup logging configuration
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Expected help message
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"
    "!monitor_availability 'url' 'frequency' - Monitor availability at a specific interval.\n"
    "!stop_monitoring_availability - Stop monitoring availability.\n"
    "!stop_bot - Stop the bot.\n"
)



class CustomTextTestResult(unittest.TextTestResult):
    """Custom test result to output 'Unit test passed' instead of 'ok'."""
    def addSuccess(self, test):
        super().addSuccess(test)
        self.stream.write("Unit test passed\n")  # Custom success message
        self.stream.flush()

class CustomTextTestRunner(unittest.TextTestRunner):
    """Custom test runner that uses the custom result class."""
    resultclass = CustomTextTestResult

class TestProjectHelpCommand(unittest.IsolatedAsyncioTestCase):

    async def asyncSetUp(self):
```

```
        """Setup the bot and mock context before each test."""
        logging.info("Setting up the bot and mock context for testing...")
        intents = discord.Intents.default()  # Create default intents
        intents.message_content = True  # Ensure the bot can read message content
        self.bot = MyBot(command_prefix="!", intents=intents)  # Initialize the bot with intents
        self.ctx = AsyncMock()  # Mock context (ctx)
        self.ctx.send = AsyncMock()  # Mock the send method to capture responses

        # Call setup_hook to ensure commands are registered
        await self.bot.setup_hook()

    async def test_project_help_success(self):
        """Test the project help command when it successfully returns the help message."""
        logging.info("Starting test: test_project_help_success")

        # Simulate calling the project_help command
        logging.info("Simulating the project_help command call.")
        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)

        # Check both the control message and help message were sent
        expected_calls = [
            call('Command recognized, passing data to control.'),  # First message sent by the bot
            call(help_message)  # Second message: the actual help message
        ]
```

```
        self.ctx.send.assert_has_calls(expected_calls, any_order=False)  # Ensure the messages are sent in the correct order
        logging.info("Verified that both the control and help messages were sent.")

    async def test_project_help_error(self):
        """Test the project help command when it encounters an error during execution."""
        logging.info("Starting test: test_project_help_error")

        # Simulate calling the project_help command and an error occurring
        logging.info("Simulating the project_help command call.")
            self.ctx.send.side_effect = Exception("Error during project_help execution.")  # Simulate an error

        command = self.bot.get_command("project_help")
            self.assertIsNotNone(command, "project_help command is not registered.")  # Ensure the command is registered

        # Act & Assert: Expect the exception to be raised
        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_!stop_bot.py ---
import sys, os, discord, logging, unittest
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from unittest.mock import MagicMock, AsyncMock, call, patch
from utils.MyBot import MyBot

# Setup logging configuration
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

"""
File: test_!stop_bot.py
Purpose: This file contains unit tests for the !stop_bot command in the Discord bot.
The tests validate both successful and error scenarios, ensuring the bot correctly shuts down or handles errors during shutdown.
Tests:
- Positive: Simulates the !stop_bot command and verifies the bot shuts down correctly.
- Negative: Simulates an error during shutdown and ensures it is handled gracefully.
"""

class CustomTextTestResult(unittest.TextTestResult):
    """Custom test result to output 'Unit test passed' instead of 'ok'."""
    def addSuccess(self, test):
        super().addSuccess(test)
        self.stream.write("Unit test passed\n")  # Custom success message
        self.stream.flush()
```

```
class CustomTextTestRunner(unittest.TextTestRunner):
    """Custom test runner that uses the custom result class."""
    resultclass = CustomTextTestResult

class TestStopBotCommand(unittest.IsolatedAsyncioTestCase):

    async def asyncSetUp(self):
        """Setup the bot and mock context before each test."""
        logging.info("Setting up the bot and mock context for testing...")
        intents = discord.Intents.default()  # Create default intents
        intents.message_content = True  # Ensure the bot can read message content
        self.bot = MyBot(command_prefix="!", intents=intents)  # Initialize the bot with intents
        self.ctx = AsyncMock()  # Mock context (ctx)
        self.ctx.send = AsyncMock()  # Mock the send method to capture responses
        self.ctx.bot = self.bot  # Mock the bot property in the context

        # Call setup_hook to ensure commands are registered
        await self.bot.setup_hook()

    async def test_stop_bot_success(self):
        """Test the stop bot command when it successfully shuts down."""
        logging.info("Starting test: test_stop_bot_success")

        # Patch the bot's close method on the ctx.bot (since bot is retrieved from ctx dynamically)
        with patch.object(self.ctx.bot, 'close', new_callable=AsyncMock) as mock_close:
            # Simulate calling the stop_bot command
```

```
        logging.info("Simulating the stop_bot command call.")
        command = self.bot.get_command("stop_bot")
            self.assertIsNotNone(command, "stop_bot command is not registered.")  # Ensure the command is registered
        await command(self.ctx)

        # Check if the correct messages were sent
        expected_calls = [
            call('Command recognized, passing data to control.'),  # First message sent by the bot
            call('The bot is shutting down...')  # Second message confirming the shutdown
        ]
            self.ctx.send.assert_has_calls(expected_calls, any_order=False)  # Ensure the messages are sent in the correct order
        logging.info("Verified that both expected messages were sent to the user.")

        # Check if bot.close() was called on the ctx.bot
        mock_close.assert_called_once()
        logging.info("Verified that the bot's close method was called once.")

    async def test_stop_bot_error(self):
        """Test the stop bot command when it encounters an error during shutdown."""
        logging.info("Starting test: test_stop_bot_error")

        # Patch the bot's close method to raise an exception
        with patch.object(self.ctx.bot, 'close', new_callable=AsyncMock) as mock_close:
            mock_close.side_effect = Exception("Error stopping bot")  # Simulate an error
```

```
        # Simulate calling the stop_bot command
        logging.info("Simulating the stop_bot command call.")
        command = self.bot.get_command("stop_bot")
            self.assertIsNotNone(command, "stop_bot command is not registered.")  # Ensure the command is registered

        # Act & Assert: Expect the exception to be raised
        with self.assertRaises(Exception):
            await command(self.ctx)

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

        # Ensure ctx.send was still called with the shutdown message before the error occurred
        self.ctx.send.assert_called_with("The bot is shutting down...")
        logging.info("Verified that 'The bot is shutting down...' message was sent despite the error.")

        # Verify that the close method was still attempted
        mock_close.assert_called_once()
        logging.info("Verified that the bot's close method was called even though it raised an error.")

if __name__ == "__main__":
    # Use the custom test runner to display 'Unit test passed'
    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))
```

"""Explanation:

Great! Both test cases (test_stop_bot_success and test_stop_bot_error) have passed successfully. Let me walk you through everything that?s happening in these tests, along with the key elements and logic involved.

Test Overview

You now have two test cases for the !stop_bot command:

test_stop_bot_success: Tests the successful shutdown of the bot when the !stop_bot command is issued.

test_stop_bot_error: Tests what happens when an error occurs during the bot shutdown process.

Key Components

StopBoundary:

The StopBoundary listens for the !stop_bot command from Discord.

When the command is received, it passes the data to StopControl using the method receive_command.

StopControl:

The StopControl handles the actual business logic for shutting down the bot.

It dynamically retrieves the bot from the context (ctx.bot) and calls bot.close() to shut it down.

If any error occurs during the shutdown (in the error test case), it simulates throwing an exception.

test_stop_bot_success (Positive Test Case)

Objective:

To ensure that the bot correctly shuts down when the !stop_bot command is issued and no errors occur.

Steps:

Mock Setup:

The bot.close() method is mocked so that it can be tracked during the test.

Simulate Command:

The stop_bot command is fetched and called with a mocked context (self.ctx).

Assertions:

Verify that the following messages were sent to the user:

"Command recognized, passing data to control."

"The bot is shutting down..."

Ensure that bot.close() was called exactly once, confirming the shutdown process was triggered.

Result:

The bot successfully sends the expected messages and shuts down, and all assertions pass.

test_stop_bot_error (Negative Test Case)

Objective:

To ensure that the bot handles an error during shutdown appropriately and still sends the proper messages.

Steps:

Mock Setup:

The bot.close() method is patched to raise an exception when it is called.

Simulate Command:

The stop_bot command is called, and the error is triggered by bot.close() raising an exception.

Assertions:

The test expects an exception to be raised and checks that the following message was sent before the error occurred:

"The bot is shutting down..."

Verify that bot.close() was still called even though it raised an error.

Result:

The bot properly sends the shutdown message before the error occurs, and the test confirms that the error is correctly handled.

Key Aspects of the Test:

Mocking:

Context (ctx): The ctx object is mocked to simulate Discord?s command context. We mock ctx.send to track the messages sent by the bot.

Bot (bot): The bot?s close() method is patched so we can simulate the shutdown behavior and check if it was called.

Testing Control Flow:

Positive Case: We verify that the bot successfully shuts down and sends the correct messages when there are no errors.

Negative Case: We simulate an error during the shutdown process and ensure the bot still sends the shutdown message and attempts to shut down before the error occurs.

Custom Test Runner:

The CustomTextTestRunner is used to modify the output so that it prints "Unit test passed" instead of the default ok when a test succeeds.

Summary of the Workflow:

Receive Command: The StopBoundary listens for the !stop_bot command.

Pass to Control: The command is passed to the StopControl via the receive_command method.

Shut Down: The StopControl dynamically retrieves the bot from the context and calls bot.close() to shut it down.

Positive Scenario: The bot shuts down successfully, and the test verifies that all the expected messages are sent.

Negative Scenario: An error is simulated during shutdown, and the test ensures the error is handled gracefully while still sending the shutdown message.

What You Have Achieved:

Positive and Negative Testing: You now have both positive (success) and negative (error) test cases for the !stop_bot command.

Comprehensive Validation: You validate that the bot behaves correctly in both normal and error situations, ensuring robustness.

Clear Test Output: The tests provide clear logging to help you understand what is happening at each step, and all assertions are passed.

This structure can be reused and expanded to test other commands and scenarios in your bot project! Let me know if you need any help expanding these tests to other use cases.

"""

--- __init__.py ---

#empty init file