```
--- test_!add_account.py ---
# File: test_!add_account.py
# Purpose: Unit tests for the !add_account command.

from unittest.mock import patch
import logging, unittest
from test_init import BaseTestSetup, CustomTextTestRunner  # Import the shared setup

"""
File: test_!add_account.py
Purpose: This file contains unit tests for the !add_account command in the Discord bot.
The tests validate both successful and error scenarios, ensuring the account is added successfully
or errors are handled properly.
Tests:
- Positive: Simulates the !add_account command and verifies the account is added correctly.
- Negative: Simulates an error while adding the account.
"""

class TestAddAccountCommand(BaseTestSetup):

    @patch('DataObjects.AccountDAO.AccountDAO.add_account')
    async def test_add_account_success(self, mock_add_account):
        """Test the add_account command when it succeeds."""
        logging.info("Starting test: test_add_account_success")

        # Mock the DAO method to simulate successful account addition
        mock_add_account.return_value = True
```

```
        command = self.bot.get_command("add_account")
        self.assertIsNotNone(command)
        await command(self.ctx, "testuser", "password123", "example.com")

        expected_message = "Account for example.com added successfully."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful account addition.")

    @patch('DataObjects.AccountDAO.AccountDAO.add_account')
    async def test_add_account_error(self, mock_add_account):
        """Test the add_account command when it encounters an error."""
        logging.info("Starting test: test_add_account_error")

        # Mock the DAO method to simulate an error during account addition
        mock_add_account.return_value = False

        command = self.bot.get_command("add_account")
        await command(self.ctx, "testuser", "password123", "example.com")

        self.ctx.send.assert_called_with("Failed to add account for example.com.")
        logging.info("Verified error handling during account addition.")

if __name__ == "__main__":
    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))
```

```
--- test_!delete_account.py ---
import logging, unittest
from unittest.mock import patch
from test_init import BaseTestSetup, CustomTextTestRunner

"""
File: test_!delete_account.py
Purpose: This file contains unit tests for the !delete_account command in the Discord bot.
The tests validate both successful and error scenarios, ensuring the bot deletes the account properly or handles errors.
Tests:
- Positive: Simulates the !delete_account command and verifies the account is deleted successfully.
- Negative: Simulates an error during account deletion and ensures it is handled gracefully.
"""

class TestDeleteAccountCommand(BaseTestSetup):

    @patch('DataObjects.AccountDAO.AccountDAO.delete_account')
    async def test_delete_account_success(self, mock_delete_account):
        """Test the delete_account command when it succeeds."""
        logging.info("Starting test: test_delete_account_success")
        mock_delete_account.return_value = True  # Simulate successful deletion

        command = self.bot.get_command("delete_account")
        self.assertIsNotNone(command)

        await command(self.ctx, '123')  # Simulate passing account ID '123'
```

```
        expected_message = "Account with ID 123 deleted successfully."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful account deletion.")

    @patch('DataObjects.AccountDAO.AccountDAO.delete_account')
    async def test_delete_account_error(self, mock_delete_account):
        """Test the delete_account command when it encounters an error."""
        logging.info("Starting test: test_delete_account_error")
        mock_delete_account.return_value = False  # Simulate failure in deletion

        command = self.bot.get_command("delete_account")
        self.assertIsNotNone(command)

        await command(self.ctx, '999')  # Simulate passing a non-existent account ID '999'

        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.")

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



--- test_!fetch_account_by_website.py ---
import unittest, logging
```

```
from unittest.mock import patch
from test_init import BaseTestSetup, CustomTextTestRunner

class TestFetchAccountByWebsiteCommand(BaseTestSetup):

    @patch('DataObjects.AccountDAO.AccountDAO.fetch_account_by_website')
    async def test_fetch_account_by_website_success(self, mock_fetch_account_by_website):
        """Test the fetch_account_by_website command when it succeeds."""
        logging.info("Starting test: test_fetch_account_by_website_success")
        mock_fetch_account_by_website.return_value = ('testuser', 'password123')

        command = self.bot.get_command("fetch_account_by_website")
        self.assertIsNotNone(command)

        await command(self.ctx, 'example.com')

            expected_message = 'Account found for example.com: Username: testuser, Password: password123'
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful account fetch.")

    @patch('DataObjects.AccountDAO.AccountDAO.fetch_account_by_website')
    async def test_fetch_account_by_website_error(self, mock_fetch_account_by_website):
        """Test the fetch_account_by_website command when it encounters an error."""
        logging.info("Starting test: test_fetch_account_by_website_error")
        mock_fetch_account_by_website.return_value = None
```

```
        command = self.bot.get_command("fetch_account_by_website")
        self.assertIsNotNone(command)

        await command(self.ctx, 'nonexistent.com')

        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_!fetch_all_accounts.py ---
# File: test_!fetch_all_accounts.py
# Purpose: Unit tests for the !fetch_all_accounts command.
from unittest.mock import patch
import logging, unittest
from test_init import BaseTestSetup, CustomTextTestRunner

class TestFetchAllAccountsCommand(BaseTestSetup):
    @patch('DataObjects.AccountDAO.AccountDAO.fetch_all_accounts')
    async def test_fetch_all_accounts_success(self, mock_fetch_all_accounts):
        """Test the fetch_all_accounts command when it succeeds."""
        logging.info("Starting test: test_fetch_all_accounts_success")

        mock_fetch_all_accounts.return_value = [("1", "testuser", "password", "example.com")]
```

```
        command = self.bot.get_command("fetch_all_accounts")
        self.assertIsNotNone(command)
        await command(self.ctx)

        # Correct the expected message
          expected_message = "Accounts:\nID: 1, Username: testuser, Password: password, Website:
example.com"
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful fetch.")


    @patch('DataObjects.AccountDAO.AccountDAO.fetch_all_accounts')  # Correct path
    async def test_fetch_all_accounts_error(self, mock_fetch_all_accounts):
        """Test the fetch_all_accounts command when it encounters an error."""
        logging.info("Starting test: test_fetch_all_accounts_error")

        # Simulate an error
        mock_fetch_all_accounts.side_effect = Exception("Database error")

        command = self.bot.get_command("fetch_all_accounts")
        await command(self.ctx)

        # Verify that the correct error message is sent
        self.ctx.send.assert_called_with("Error fetching accounts.")
        logging.info("Verified error handling.")
```

```
if __name__ == "__main__":
    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))



--- test_!project_help.py ---
import logging, unittest
from test_init import BaseTestSetup, CustomTextTestRunner
from unittest.mock import call
"""
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 TestStopBotCommand(BaseTestSetup):


    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.")

# 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"
```

```
)

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



--- test_!stop_bot.py ---
import logging, unittest
from unittest.mock import AsyncMock, call, patch
from test_init import BaseTestSetup, CustomTextTestRunner

"""
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 TestStopBotCommand(BaseTestSetup):

    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))



--- test_init.py ---
# Purpose: This file contains common setup code for all test cases.
import sys, os, discord, logging, unittest
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from unittest.mock import AsyncMock
from utils.MyBot import MyBot

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

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 BaseTestSetup(unittest.IsolatedAsyncioTestCase):
    """Base setup class for initializing bot and mock context for all tests."""
```

```
    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()
        intents.message_content = True
        self.bot = MyBot(command_prefix="!", intents=intents)
        self.ctx = AsyncMock()
        self.ctx.send = AsyncMock()
        self.ctx.bot = self.bot  # Mock the bot property in the context
        await self.bot.setup_hook()  # Ensure commands are registered
```

--- __init__.py ---

```
#empty init file
```