

```

--- test_!add_account.py ---

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_!check_availability.py ---

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_!close_browser.py ---
```

```
import logging, unittest
```

```
from unittest.mock import patch
```

```
from test_init import BaseTestSetup, CustomTextTestRunner
```

```
"""
```

File: test\_!close\_browser.py

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

The tests validate both successful and error scenarios, ensuring the browser closes properly or errors are handled gracefully.

Tests:

- Positive: Simulates the !close\_browser command and verifies the browser closes correctly.
- Negative: Simulates an error during browser closure and ensures it is handled gracefully.

```
"""
```

```
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")
```

```
self.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_delete_account.py ---
```

```
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("Unit test for delete account starting for positive test:")

        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.")


    @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("Unit test for delete account starting for negative test:")

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.")


if __name__ == "__main__":

    # Custom test runner to highlight the test results

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


--- test_fetch_account_by_website.py ---

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_fetch_all_accounts.py ---

```

```
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.assertIsNone(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\_!get\_price.py ---

```

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_!launch_browser.py ---

import logging, unittest

from unittest.mock import patch

from test_init import BaseTestSetup, CustomTextTestRunner

```



"""

File: test\_!launch\_browser.py

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

The tests validate both successful and error scenarios, ensuring the browser launches properly or errors are handled gracefully.

"""

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

--- test\_!login.py ---

```
import logging, unittest
```

```
from unittest.mock import patch, AsyncMock
```

```
from test_init import BaseTestSetup, CustomTextTestRunner
```

"""

File: test\_!login.py

Purpose: Unit tests for the !login command in the Discord bot.

The tests validate both successful and error scenarios, ensuring the bot correctly logs in to a specified website or handles errors gracefully.

Tests:

- Positive: Simulates the !login command and verifies the login is successful.
- Negative: Simulates an error during login and ensures it is handled gracefully.

"""

```
class TestLoginCommand(BaseTestSetup):
```

```
    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
```

```
    @patch('control.LoginControl.LoginControl.receive_command')
```

```
    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")
```

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

```
mock_parse_user_message.return_value = ["login", "ebay"]
```

```
# Simulate a successful login
```

```
mock_receive_command.return_value = "Login successful."
```

```
# Retrieve the login command from the bot
```

```
command = self.bot.get_command("login")
```

```
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 = "Login successful."
```

```
self.ctx.send.assert_called_with(expected_message)
```

```
logging.info("Verified successful login.")
```

```
@patch('DataObjects.global_vars.GlobalState.parse_user_message')
```

```
@patch('control.LoginControl.LoginControl.receive_command')
```

```
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")
```

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

```
mock_parse_user_message.return_value = ["login", "nonexistent.com"]
```

```
# Simulate a failure during login
```

```
mock_receive_command.return_value = "Failed to login. No account found."
```

```
# Retrieve the login command from the bot
```

```
command = self.bot.get_command("login")
```

```
self.assertIsNotNone(command)
```

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

```
await command(self.ctx)
```

```
# Verify the correct error message is sent
```

```
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_!navigate_to_website.py ---
```

```
import logging, unittest
```

```
from unittest.mock import patch, AsyncMock
```

```
from test_init import BaseTestSetup, CustomTextTestRunner
```

"""

File: test\_!navigate\_to\_website.py

Purpose: This file contains unit tests for the !navigate\_to\_website command in the Discord bot.

The tests validate both successful and error scenarios, ensuring the bot navigates to the website correctly or handles errors.

"""

```
class TestNavigateToWebsiteCommand(BaseTestSetup):
```

```
    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
```

```
    @patch('entity.BrowserEntity.BrowserEntity.navigate_to_website')
```

```
        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")
```

```
        # Mock the parsed message to return the expected command and URL
```

```
        mock_parse_user_message.return_value = ["navigate_to_website", "https://example.com"]
```

```
        # Simulate successful navigation
```

```
        mock_receive_command.return_value = "Navigated to https://example.com."
```

```
        # Retrieve the navigate_to_website command from the bot
```

```
        command = self.bot.get_command("navigate_to_website")
```

```
        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 = "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')

@patch('entity.BrowserEntity.BrowserEntity.navigate_to_website')

    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")


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

mock_parse_user_message.return_value = ["navigate_to_website", "https://invalid-url.com"]


# Simulate a failure during navigation

mock_receive_command.side_effect = Exception("Failed to navigate to the website.")


# Retrieve the navigate_to_website command from the bot

command = self.bot.get_command("navigate_to_website")

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 navigate to the website.") # Error message handled

        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_!project_help.py ---

```

```

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_!start_monitoring_availability.py ---
```

```
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_!start_monitoring_price.py ---

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.assertIsNone(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_!stop_bot.py ---
```

```
import logging, unittest
```

```
from unittest.mock import AsyncMock, 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):
```

```
    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
```

```
    @patch('control.StopControl.StopControl.receive_command', new_callable=AsyncMock)
```

```
    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")
```

```
        # Setup mocks
```

```
        mock_receive_command.return_value = "The bot is shutting down..."
```

```
        mock_parse_user_message.return_value = ["stop_bot"]
```

```
        # Simulate calling the stop_bot command
```

```
        command = self.bot.get_command("stop_bot")
```

```
        self.assertIsNotNone(command, "stop_bot command is not registered.")
```

```
        await command(self.ctx)
```

```
        # Verify the message was sent before shutdown is initiated
```

```
        self.ctx.send.assert_called_once_with("Command recognized, passing data to control.")
```

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

```
        # Ensure bot.close() is 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')

@patch('control.StopControl.StopControl.receive_command', new_callable=AsyncMock)

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")

    # Setup mocks

    exception_message = "Error stopping bot"

    mock_receive_command.side_effect = Exception(exception_message)

    mock_parse_user_message.return_value = ["stop_bot"]

    # Simulate calling the stop_bot command

    command = self.bot.get_command("stop_bot")

    self.assertIsNotNone(command, "stop_bot command is not registered.")

    with self.assertRaises(Exception) as context:

        await command(self.ctx)

    # Verify that the correct error message is sent

    self.ctx.send.assert_called_with('Command recognized, passing data to control.')

    self.assertTrue(exception_message in str(context.exception))

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

    # Verify that the close method was still 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__":
```

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

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

```
--- test_!stop_monitoring_availability.py ---
```

```
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))

```

--- test\_!stop\_monitoring\_price.py ---

```

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\_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
```