

```
--- 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_!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('control.AvailabilityControl.AvailabilityControl.receive_command')
```

```
    async def test_check_availability_success(self, mock_receive_command):
```

```
        """Test the check_availability command when it succeeds."""
```

```
        logging.info("Starting test: test_check_availability_success")
```

```
        mock_receive_command.return_value = "Available for booking."
```

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

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

```
        await command(self.ctx, "https://example.com", "2024-09-30")
```

```
        expected_message = "Available for booking."
```

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

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

```

@patch('control.AvailabilityControl.AvailabilityControl.receive_command')

async def test_check_availability_error(self, mock_receive_command):
    """Test the check_availability command when it encounters an error."""

    logging.info("Starting test: test_check_availability_error")

    mock_receive_command.return_value = "No availability found."

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

    self.assertIsNotNone(command)

    await command(self.ctx, "https://invalid-url.com", "2024-09-30")

    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('entity.BrowserEntity.BrowserEntity.close_browser')
```

```
    async def test_close_browser_success(self, mock_close_browser):
```

```
        """Test the close_browser command when it succeeds."""
```

```
        logging.info("Starting test: test_close_browser_success")
```

```
        # 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('entity.BrowserEntity.BrowserEntity.close_browser')
```

```
async def test_close_browser_error(self, mock_close_browser):
```

```
    """Test the close_browser command when it encounters an error."""
```

```
    logging.info("Starting test: test_close_browser_error")
```

```
    # 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 ---
```

```
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 = 'testuser', '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.assertIsNone(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\_!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')
```

```
    async def test_get_price_success(self, mock_receive_command):
```

```
        """Test the get_price command when it succeeds."""
```

```
        logging.info("Starting test: test_get_price_success")
```

```
        # 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 with a valid URL
```

```
await command(self.ctx, "https://example.com")
```

```
# Verify the expected message was sent to the user
```

```
expected_message = "Price: $199.99"
```

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

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

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

```
async def test_get_price_error(self, mock_receive_command):
```

```
    """Test the get_price command when it encounters an error."""
```

```
    logging.info("Starting test: test_get_price_error")
```

```
# 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 with an invalid URL
```

```
await command(self.ctx, "https://invalid-url.com")
```

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

```
self.ctx.send.assert_called_with("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.

Tests:

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

```
"""
```

```
class TestLaunchBrowserCommand(BaseTestSetup):
```

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

```
    async def test_launch_browser_success(self, mock_launch_browser):
```

```
"""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."
```

```
# Retrieve the launch_browser command from the bot
```

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

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

```
# Call the command
```

```
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('entity.BrowserEntity.BrowserEntity.launch_browser')
```

```
async def test_launch_browser_error(self, mock_launch_browser):
```

```
"""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")
```

```
# Retrieve the launch_browser command from the bot
```

```

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

self.assertIsNone(command)

# Call the command

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

```
    async def test_login_success(self, mock_receive_command):
```

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

```
        logging.info("Starting test: test_login_success")
```

```
        # 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 with a valid site (e.g., ebay)
```

```
        await command(self.ctx, "ebay")
```

```
        # 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('control.LoginControl.LoginControl.receive_command')

async def test_login_error(self, mock_receive_command):
    """Test the login command when it encounters an error."""
    logging.info("Starting test: test_login_error")

    # 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 with a non-existent site (e.g., nonexistent.com)
    await command(self.ctx, "nonexistent.com")

    # Verify the correct error message is sent
    self.ctx.send.assert_called_with("Failed to login. No account found.")
    logging.info("Verified error handling during login.")

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

--- test_monitor_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('control.AvailabilityControl.AvailabilityControl.receive_command')
```

```
    async def test_monitor_availability_success(self, mock_receive_command):
```

```
        """Test the monitor_availability command when it succeeds."""
```

```
        logging.info("Starting test: test_monitor_availability_success")
```

```
        mock_receive_command.return_value = "Monitoring started for https://example.com."
```

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

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

```
        await command(self.ctx, "https://example.com", "2024-09-30", 15)
```

```
        expected_message = "Monitoring started for https://example.com."
```

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

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

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

```
    async def test_monitor_availability_error(self, mock_receive_command):
```

```
        """Test the monitor_availability command when it encounters an error."""
```

```
logging.info("Starting test: test_monitor_availability_error")
```

```
mock_receive_command.return_value = "Failed to start monitoring."
```

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

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

```
await command(self.ctx, "https://invalid-url.com", "2024-09-30", 15)
```

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

```
import logging, unittest
```

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

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

```
    async def test_navigate_to_website_success(self, mock_navigate_to_website):
```

```
        """Test the navigate_to_website command when it succeeds."""
```

```
        logging.info("Starting test: test_navigate_to_website_success")
```

```
        # Simulate successful navigation
```

```
        mock_navigate_to_website.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
```

```
        await command(self.ctx, "https://example.com")
```

```
        # 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('entity.BrowserEntity.BrowserEntity.navigate_to_website')
```

```
    async def test_navigate_to_website_error(self, mock_navigate_to_website):
```

```
        """Test the navigate_to_website command when it encounters an error."""
```

```
        logging.info("Starting test: test_navigate_to_website_error")
```

```

# Simulate a failure during navigation
mock_navigate_to_website.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
await command(self.ctx, "https://invalid-url.com")

# 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 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"

"!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"

)

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

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

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

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

import logging, unittest

from unittest.mock import patch

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.

"""

```
class TestStartMonitoringPriceCommand(BaseTestSetup):
```

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

```
    async def test_start_monitoring_price_success(self, mock_receive_command):
```

```
        """Test the start_monitoring_price command when it succeeds."""
```

```
        logging.info("Starting test: test_start_monitoring_price_success")
```

```
        # 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 with a valid URL and frequency
```

```
        await command(self.ctx, "https://example.com", 20)
```

```
        # 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('control.PriceControl.PriceControl.receive_command')

async def test_start_monitoring_price_error(self, mock_receive_command):
    """Test the start_monitoring_price command when it encounters an error."""
    logging.info("Starting test: test_start_monitoring_price_error")

    # Simulate a failure during price monitoring start
    mock_receive_command.return_value = "Failed to start monitoring"

    # Retrieve the start_monitoring_price command from the bot
    command = self.bot.get_command("start_monitoring_price")
    self.assertIsNotNone(command)

    # Call the command with an invalid URL
    await command(self.ctx, "https://invalid-url.com", 20)

    # Verify the correct error message is sent
    self.ctx.send.assert_called_with("Failed to start monitoring")
    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, 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_!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('control.AvailabilityControl.AvailabilityControl.receive_command')
```

```
    async def test_stop_monitoring_availability_no_active_session(self, mock_receive_command):
```

```
        """Test the stop_monitoring_availability command when no active session exists."""
```

```
        logging.info("Starting test: test_stop_monitoring_availability_no_active_session")
```

```
        mock_receive_command.return_value = "There was no active availability monitoring session."
```

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

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

```
        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('control.AvailabilityControl.AvailabilityControl.receive_command')
```

```
    async def test_stop_monitoring_availability_success(self, mock_receive_command):
```

```
        """Test the stop_monitoring_availability command when it succeeds."""
```

```
        logging.info("Starting test: test_stop_monitoring_availability_success")
```

```
        mock_receive_command.return_value = "Availability monitoring stopped successfully."
```

```

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

self.assertIsNotNone(command)

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

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.

"""

```

class TestStopMonitoringPriceCommand(BaseTestSetup):

```

```

    @patch('control.PriceControl.PriceControl.receive_command')

```



```

async def test_stop_monitoring_price_no_active_session(self, mock_receive_command):

    """Test the stop_monitoring_price command when no active monitoring session exists."""

    logging.info("Starting test: test_stop_monitoring_price_no_active_session")

    # Simulate scenario with no active price monitoring session

    mock_receive_command.return_value = "There was no active price monitoring session.
Nothing to stop."

    # 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 = "There was no active price monitoring session. Nothing to stop."

    self.ctx.send.assert_called_with(expected_message)

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

@patch('control.PriceControl.PriceControl.receive_command')

async def test_stop_monitoring_price_success_with_results(self, mock_receive_command):

    """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_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('control.PriceControl.PriceControl.receive_command')
```

```
async def test_stop_monitoring_price_error(self, mock_receive_command):
```

```
    """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_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

self.ctx.send.assert_called_with("Error stopping price monitoring")

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


if __name__ == "__main__":

    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
```