**Discord Bot Automation Assistant**

**Discord Bot Automation Assistant Test Plan**

**Oguz Kaan Yildirim**

**307637**

# Table Of Contents

[Table Of Contents 2](#_Toc179365184)

[INTRDOCUTION 3](#_Toc179365185)

[TEST CASES 7](#_Toc179365186)

[Test Case 0: test\_init.py 7](#_Toc179365187)

[Description 7](#_Toc179365188)

[Test Case 1: Project Help 8](#_Toc179365189)

[Description 8](#_Toc179365190)

[Test Steps 8](#_Toc179365191)

[Test Data 8](#_Toc179365192)

[Conclusion 9](#_Toc179365193)

# INTRDOCUTION

This document provides an overview of unit testing for a software project aimed at automating the monitoring of product prices and service availability. The goal of the testing is to ensure that all system components function correctly when tested in isolation. This modular approach facilitates the validation of core system functions, including command processing, browser automation, and data export, within a controlled test environment.

The system is composed of modules responsible for interacting with web browsers, processing user commands, retrieving product data from websites, and monitoring availability for services like reservations. These modules have undergone rigorous testing using Python's pytest framework. External systems like websites and Discord commands are simulated using mocks and patches, ensuring expected system behavior in both typical and edge-case scenarios.

This document includes:

* An outline of the testing strategy, scope, and objectives,
* A description of the tools and technologies used during testing,
* Solutions to challenges encountered when testing Discord commands,
* Details on the test setup, implementation, and how the testing framework integrates with the system architecture.

The purpose of the unit testing is to confirm that the system can accurately process user commands, interact with websites, retrieve data, log it, and generate reports. Isolating components during testing enhances confidence in the system's reliability and robustness.

To have a better look at the code/project especially to testing source codes; codes can be found in github. Test files are under UnitTesting Folder.

Unit Tests also has descriptions and executable steps explained in testing.

https://github.com/oguzky7/DiscordBotProject\_CISC699/tree/develop/UnitTesting

Test Plan Overview

Scope

The scope of the unit tests covers all critical aspects of the system, ensuring each component performs its intended function independently. This modular testing approach covers:

* **Command Processing and Core Features:** Verifying that the system properly receives and processes user commands to monitor prices, check availability, and log data efficiently.
* **Browser Interactions:** Ensuring that the system can initiate, navigate, and close browser sessions while effectively interacting with web content.
* **Data Logging and Export:** Validating that price and availability data are logged and exported in structured formats such as Excel and HTML.
* **Error Handling:** Confirming that the system gracefully handles errors (e.g., invalid commands or network issues) and provides appropriate user feedback.

Objectives

The unit testing aims to:

* **Functional Verification:** Ensure that components like command processing, data retrieval, and logging function correctly in isolation.
* **Component Isolation:** By testing each module independently, failures in one area don’t affect others, enabling easier identification of defects.
* **Data Accuracy and Consistency:** Ensure the system processes price and availability data correctly before logging and exporting it.
* **System Reliability:** Test the system’s ability to handle various scenarios, including repeated commands, long-running processes, and invalid inputs.

Strategy

The strategy focuses on modular unit testing, ensuring each part of the system is validated without relying on external dependencies like live websites or browsers. Key elements include:

* **Unit Testing:** Each system module—command processing, web scraping, or data export—is tested independently.
* **Mocking and Simulation:** External systems are simulated using mocks, allowing tests to focus on internal logic without live interactions.
* **Automated Execution:** Tests are automated using pytest, ensuring consistency and enabling integration into CI/CD pipelines for automatic execution upon code changes.

Structure of the Tests

The tests are divided into suites targeting specific components:

* **Control Layer:** Verifies user commands are correctly processed.
* **Entity Layer:** Validates interactions with external systems (e.g., retrieving product prices or checking availability).
* **Data Logging and Export:** Ensures data is logged and exported without errors.

This structure allows the test framework to expand as the system evolves, enabling independent testing of new features without disrupting existing tests.

Expected Outcomes

This modular approach is expected to yield:

* Accurate command processing with correct results,
* Error-free logging and export of data,
* Graceful handling of unexpected situations like invalid commands or network failures,
* Stable performance during long-running tasks, such as continuous monitoring of product prices.

Tools and Technologies

Pytest

The primary framework used for test execution is pytest, which supports both synchronous and asynchronous testing. This is critical since many system operations involve real-time monitoring and asynchronous tasks like web scraping. Integration with mocking tools allows thorough simulation of external dependencies, ensuring isolated and repeatable tests.

Unittest.mock

The unittest.mock library is key to isolating system components from external dependencies, such as web browsers and Discord commands. The system uses Mock and AsyncMock to simulate responses from these services, enabling tests to focus on internal logic.

* **Mocking External Systems:** Mocks simulate browser actions and Discord command inputs, allowing test isolation.

pytest-asyncio

As many system operations are asynchronous, pytest-asyncio manages async code in tests, ensuring that operations like price monitoring are tested properly.

Mocked Selenium

Selenium, a tool for browser automation, is mocked during unit testing to focus on internal logic without requiring real browser instances.

Purpose and Setup

Purpose of Unit Testing

The purpose of unit testing is to validate that each component of the system functions correctly in isolation, focusing on command processing, web interactions, and data export operations.

Challenges in Testing Discord Commands

Testing Discord commands posed a challenge due to the lack of native testing tools for discord.py. To address this, unit tests simulate command inputs and directly interact with the control layer methods, ensuring that the system logic is properly tested without relying on live Discord command handling.

Setup of the Testing Environment

The environment is designed to ensure isolated component testing without live system dependencies:

* **Mocking and Patching:** The unittest.mock library simulates external systems like browsers and Discord commands, ensuring independent testing of each component.
* **Asynchronous Testing:** pytest-asyncio allows proper testing of asynchronous tasks like price monitoring.
* **Test Isolation:** Each test runs independently, ensuring faster execution and easier debugging.

Implementation Details

The structure of the tests mirrors the system’s modular architecture, with distinct test suites for each layer:

* **Control Layer Tests:** Validate command processing to ensure inputs are handled correctly.
* **Entity Layer Tests:** Confirm the core functionality of price retrieval, availability checking, and data export.
* **Logging and Export Tests:** Ensure data is correctly formatted and saved to Excel and HTML files.

By focusing on these areas, the unit tests confirm that the system’s logic functions as intended, and any external dependencies are properly mocked for accurate results.

# 

# Unit Tests for Use Cases

Every use case has multiple unit tests in them for every step in main flow.

## test\_init.py

### Description

The test\_init file serves two main purposes. First, it consolidates all necessary imports to avoid redundant import statements across multiple test files, improving maintainability and consistency. Second, it provides functionality to run all unit tests at once by executing test\_init.py.

A screen shot of a computer program

Description automatically generated

If specific tests need to be run individually, each test file has the if \_\_name\_\_ == "\_\_main\_\_": pytest.main([\_\_file\_\_]) block, allowing users to run that specific test file independently.

This setup streamlines both the import process and test execution, making it easy to run tests collectively or individually based on the needs of the project.

A computer screen with text

Description automatically generated

## !project\_help

**Description**  
This test ensures that the BotControl.receive\_command() method processes the !project\_help command correctly and returns the appropriate help message listing available commands.

**Test Steps**

The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**  
   This test will ensure that BotControl.receive\_command() handles the !project\_help command correctly and returns the expected help message.

**Test Data**

* Command: "!project\_help"
* Expected Output: "Here are the available commands:..."

A computer screen shot of a black background

Description automatically generated

A screen shot of a computer program

Description automatically generated

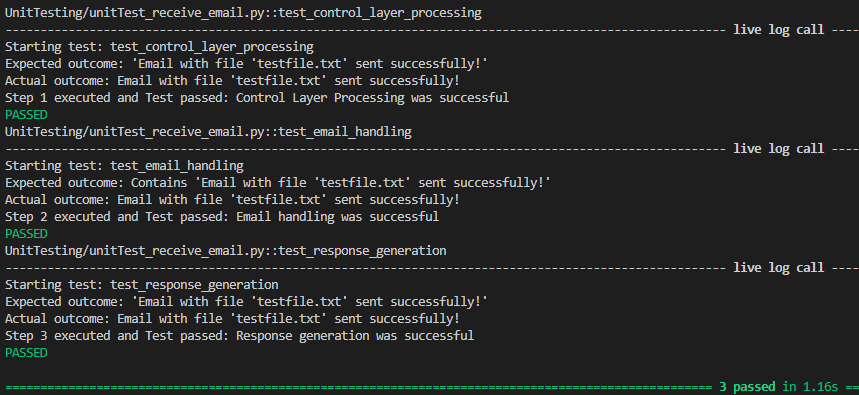
Figure 1

## !receive\_email <file name>

**Description**  
This test ensures that the BotControl.receive\_command() method processes the !receive\_email command correctly by passing the file name to the email handler, sending the email, and generating the correct response.

**Test Steps**  
The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**  
   This test ensures that BotControl.receive\_command() handles the !receive\_email command correctly, including proper parameter passing and validation.
2. **Email Handling**  
   This test focuses on the EmailEntity.send\_email\_with\_attachments() function to ensure it processes the request to send the email with the attached file.
3. **Response Generation**  
   This test validates that the control layer correctly interprets the response from the email handling step and returns the appropriate result to the boundary layer.



A screenshot of a computer program

Description automatically generated

## !navigate\_to\_website

**Description**  
This test ensures that the BotControl.receive\_command() method processes the !navigate\_to\_website command correctly by extracting the URL, navigating to the specified website, and returning the appropriate result.

**Test Steps**  
The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**  
   This test ensures that BotControl.receive\_command() handles the !navigate\_to\_website command correctly by extracting the URL and passing it to the browser control.
2. **Browser Navigation**  
   This test ensures that the BrowserEntity.navigate\_to\_website() function processes the navigation request to the specified URL correctly.
3. **Response Generation**  
   This test validates that the control layer correctly returns the appropriate result after the browser interaction is completed.

**Test Data**

* Command: "!navigate\_to\_website"
* Test URL: "http://example.com"
* Expected Output: "Navigation successful"A screenshot of a computer program

  Description automatically generated

A screen shot of a computer program

Description automatically generated

## !login

**Description**  
This test ensures that the BotControl.receive\_command() method processes the !login command correctly by passing the website, username, and password to the browser and verifying the login process.

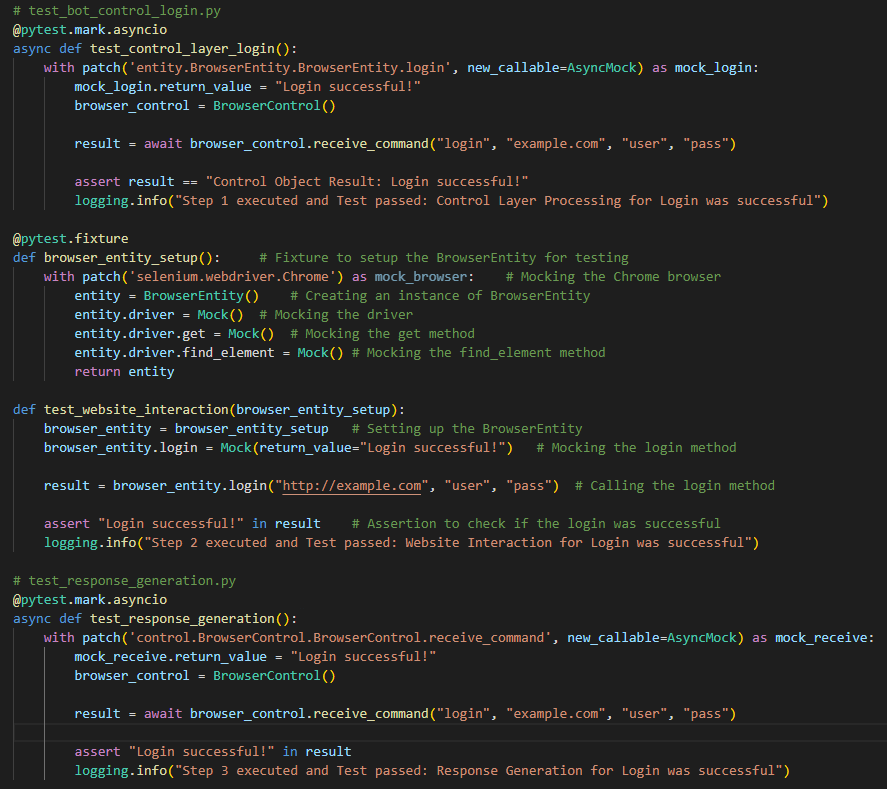
**Test Steps**  
The main flow for this use case is as follows, and we will test the following steps:

1. **Control Layer Processing**  
   This test ensures that BotControl.receive\_command() handles the !login command correctly, including proper parameter passing and validation.
2. **Website Interaction**  
   This test focuses on the BrowserEntity.login() function to ensure it processes the request to log in to the website using the provided credentials.
3. **Response Generation**  
   This test validates that the control layer correctly interprets the response from the website interaction step and returns the appropriate result to the boundary layer.

**Test Data**

* Command: "!login"
* Test website: "http://example.com"
* Test username: "user"
* Test password: "pass"
* Expected Output: "Login successful"A screenshot of a computer program

  Description automatically generated



## !close\_browser

**Description**  
This test ensures that the BotControl.receive\_command() method processes the !close\_browser command correctly by handling browser closure and returning the appropriate response.

**Test Steps**  
The main flow for this use case is as follows, and we have unit tests for the following steps:

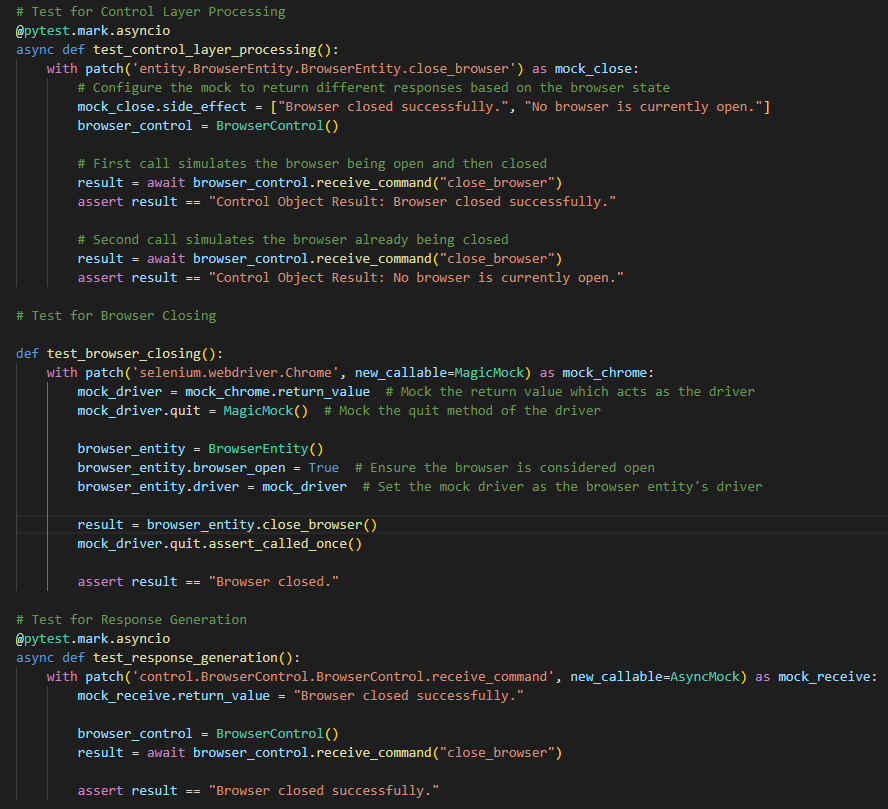
1. **Control Layer Processing**  
   This test ensures that BotControl.receive\_command() handles the !close\_browser command correctly.
2. **Browser Closing**  
   This test ensures that the BrowserEntity.close\_browser() function successfully closes the browser.
3. **Response Generation**  
   This test validates that the control layer correctly interprets the browser closure and returns the appropriate result to the boundary layer.

**Test Data**

* Command: "!close\_browser"
* Expected Output: "Browser closed successfully"

A screenshot of a computer program

Description automatically generated



## !get\_price <website>

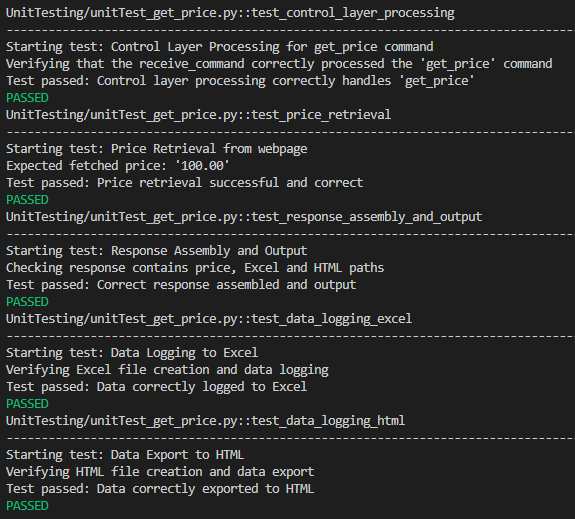
**Description**  
This test ensures that the BotControl.receive\_command() method processes the !get\_price command correctly by extracting the website URL, retrieving the price, and logging the data to both Excel and HTML formats.

**Test Steps**  
The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**  
   This test ensures that BotControl.receive\_command() handles the !get\_price command correctly, including URL parameter handling.
2. **Price Retrieval**  
   This test ensures that the PriceEntity.get\_price\_from\_page() function retrieves the correct price from the webpage.
3. **Data Logging to Excel**  
   This test verifies that the retrieved price data is correctly logged to an Excel file.
4. **Data Logging to HTML**  
   This test ensures that the price data is correctly exported to an HTML file.

**Test Data**

* Command: "!get\_price"
* Test website: "http://example.com/product"
* Expected Output: "100.00"



## !start\_monitoring\_price <website>

**Description**  
This test ensures that the BotControl.receive\_command() method processes the !start\_monitoring\_price command correctly by initiating price monitoring at regular intervals for the specified website.

**Test Steps**  
The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**  
   This test ensures that BotControl.receive\_command() handles the !start\_monitoring\_price command correctly, including proper URL parameter passing.
2. **Price Monitoring Initiation**  
   This test ensures that price monitoring is initiated and repeated at regular intervals by calling the get\_price() function.
3. **Stop Monitoring Logic**  
   This test confirms that price monitoring can be stopped correctly and the final results are collected.

**Test Data**

* Command: "!start\_monitoring\_price"
* Test website: "http://example.com/product"
* Expected Output: "Price monitoring started"

## !stop\_monitoring\_price <website>

**Description**  
This test ensures that the BotControl.receive\_command() method processes the !stop\_monitoring\_price command correctly by stopping the monitoring process and generating a final summary of the results.

**Test Steps**  
The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**  
   This test ensures that BotControl.receive\_command() handles the !stop\_monitoring\_price command correctly.
2. **Stop Monitoring Logic**  
   This test ensures that the monitoring process is stopped and results are collected.
3. **Final Summary Generation**  
   This test validates that a final summary of the price monitoring results is generated and returned.

**Test Data**

* Command: "!stop\_monitoring\_price"
* Test website: "http://example.com/product"
* Expected Output: "Price monitoring stopped"

## !check\_availability <website>

**Description**  
This test ensures that the BotControl.receive\_command() method processes the !check\_availability command correctly by checking the availability of the specified service on the website and logging the results.

**Test Steps**  
The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**  
   This test ensures that BotControl.receive\_command() handles the !check\_availability command correctly.
2. **Availability Checking**  
   This test ensures that the AvailabilityEntity.check\_availability() function checks the availability of the specified service.
3. **Data Logging to Excel**  
   This test verifies that the availability data is logged to an Excel file.
4. **Data Logging to HTML**  
   This test ensures that the availability data is exported to an HTML file.

**Test Data**

* Command: "!check\_availability"
* Test website: "http://example.com/reservation"
* Expected Output: "Availability confirmed"

## !start\_monitoring\_availability <website>

**Description**  
This test ensures that the BotControl.receive\_command() method processes the !start\_monitoring\_availability command correctly by initiating service availability monitoring at regular intervals for the specified website.

**Test Steps**  
The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**  
   This test ensures that BotControl.receive\_command() handles the !start\_monitoring\_availability command correctly.
2. **Availability Monitoring Initiation**  
   This test ensures that service availability monitoring is initiated and repeated at regular intervals.
3. **Stop Monitoring Logic**  
   This test confirms that availability monitoring can be stopped correctly and the final results are collected.

**Test Data**

* Command: "!start\_monitoring\_availability"
* Test website: "http://example.com/reservation"
* Expected Output: "Availability monitoring started"

## !stop\_monitoring\_availability <website>

**Description**  
This test ensures that the BotControl.receive\_command() method processes the !stop\_monitoring\_availability command correctly by stopping the monitoring process and generating a final summary of the results.

**Test Steps**  
The main flow for this use case is as follows, and we have unit tests for the following steps:

1. **Control Layer Processing**  
   This test ensures that BotControl.receive\_command() handles the !stop\_monitoring\_availability command correctly.
2. **Stop Monitoring Logic**  
   This test ensures that the monitoring process is stopped and results are collected.
3. **Final Summary Generation**  
   This test validates that a final summary of the availability monitoring results is generated and returned.

**Test Data**

* Command: "!stop\_monitoring\_availability"
* Test website: "http://example.com/reservation"
* Expected Output: "Availability monitoring stopped"

# Conclusion

To be written later