TABLE OF CONTENTS

CHAPTER ONE: INTRODUCTION

1.1 Summary and Thesis Outline

CHAPTER TWO: RELATED WORK

2.1 Review of Existing Systems

This page is just to keep structure. I couldn’t make chapter 3 sections start with “3” unless I leave these here.

I’m also using last terms template that used before. I believe this was validating all rules. I still corrected everything I saw

And compared with “CommuniView “ like you suggested

CHAPTER THREE: SYSTEM DESING AND IMPLEMENTATION

This chapter delves into the detailed design and architecture of the Discord bot project, a sophisticated system engineered to facilitate robust interactions within the Discord environment through a series of automated tasks and responses. The design and development of this project are structured around a series of carefully planned assignments, each contributing essential components to the bot's functionality and operational efficiency.

The objective of this chapter is to provide a thorough exposition of the project's requirements, its architectural blueprint, and the intricate design decisions that collectively underpin the bot's functionality. It serves to bridge the theoretical frameworks discussed in previous chapters with the practical implementations that follow, illustrating the transition from conceptual models to executable solutions.

Project Requirements: The chapter begins by revisiting the project's requirements as outlined in earlier coursework, specifically focusing on the use cases developed for the Discord bot. This section will present a UML use case diagram accompanied by detailed textual descriptions, highlighting how these use cases address the specific needs of the system.

Architecture: After the requirements, the architecture section introduces the UML component and deployment diagrams that illustrate the high-level structure and distribution of the system across various platforms and services. This includes detailing how different components interact within the system and how they are deployed to support scalability and reliability.

Design: The design portion of this chapter will explore the UML class diagrams from the project’s development phase. It will include comprehensive descriptions of each class, their associations, and the dynamic interactions within the system. This section aims to showcase the logical structure of the object-oriented approach used in developing the Discord bot.

Object Detailing: Following the class diagrams, a detailed discussion on the attributes, methods, and contracts of each object within the system will be provided. This will include how these objects map to the underlying data store, emphasizing data handling and object persistence.

Technology Stack/Framework: The chapter will also provide an overview of the technology stack and frameworks utilized in the project. It will include diagrams and explanations of how these technologies are implemented within the framework of the project to meet the set objectives efficiently.

Additional Considerations: This section will touch upon the supplementary design considerations such as programming patterns, principles of clean design, and strategies for achieving a zero-defect implementation. It aims to reflect on the theoretical aspects of software design in light of practical, real-world application.

Conclusion: Finally, the chapter will conclude by summarizing the design and architectural choices made during the project's development phase, reflecting on how these decisions align with the project's initial goals and requirements.

3.1 Project Requirements

In this section, we will cover the project requirements, including the use case diagram and detailed descriptions of the use cases. We will also integrate relevant parts from assignments to provide a comprehensive understanding.

3.1.1 Project Help (!project_help)

- **Actor**: User
- **Description**: Provides the user with a list of available commands and descriptions on how to use them.
- **Preconditions**: Bot must be operational and accessible to the user.
- **Trigger**: User sends the "!project_help" command.
- Main Flow:
    1. User requests help by sending "!project_help".
    2. Bot receives the command and fetches a list of all usable commands along with descriptions.
    3. Bot displays the command list to the user.
- **Postconditions**: User receives the information needed to utilize the bot effectively.

3.1.2 Navigate to Website (!navigate_to_website)

- **Actor**: User
- **Description**: Enables the user to command the bot to open a web browser and navigate to a specified URL.
- **Preconditions**: Bot must be operational.

- **Trigger**: User sends the "!navigate_to_website [URL]" command.
- Main Flow:
    1. User inputs the command with a URL.
    2. Bot recognizes the command and extracts the URL.
    3. Bot launches the web browser and navigates to the specified URL.
    4. Bot confirms navigation success to the user.
- **Postconditions**: The browser has opened at the desired web page.

3.1.3 Close Browser (!close_browser)

- **Actor**: User
- **Description**: Allows the user to send a command to the bot to close the currently opened web browser.
- **Preconditions**: A web browser must be opened by the bot.
- **Trigger**: User sends the "!close_browser" command.
- Main Flow:
    1. User sends the command to close the browser.
    2. Bot receives the command and proceeds to close any open browsers.
    3. Bot confirms the closure of the browser.
- **Postconditions**: Any browser opened by the bot is closed.

3.1.4 Login to a Website (!login)

- **Actor**: User
- **Description**: Enables the user to command the bot to log into a web application using provided

credentials.

- **Preconditions**: The target website's login page is accessible.
- **Trigger**: User sends the "!login [website] [username] [password]" command.
- Main Flow:
    1. User inputs the command with website URL, username, and password.
    2. Bot recognizes the command, extracts the details, and navigates to the login page of the website.
    3. Bot inputs the credentials and attempts to log in.
    4. Bot confirms to the user whether the login was successful or if there were any errors.
- **Postconditions**: User is logged into the website if credentials are correct and the website is reachable.

3.1.5 Receive Email (!receive_email)

- **Actor**: User
- **Description**: Commands the bot to send an email with an attached file specified by the user.
- **Preconditions**: Bot must be operational, and the specified file must be present in the system.
- **Trigger**: User sends the "!receive_email [file_name]" command with a valid file name.
- Main Flow:
    1. User inputs the command with the name of the file to be emailed (e.g., "!receive_email fileToEmail.html").
    2. Bot recognizes the command and verifies the presence of the file in the system.
    3. Bot attaches the file to an email and sends it to a predetermined recipient.
    4. Bot confirms to the user that the email has been sent successfully or informs them of any

issues encountered (e.g., file not found or email delivery failure).

- **Postconditions**: The email is sent with the specified attachment if all conditions are met.

3.1.6 Get Price (!get_price)

- **Actor**: User
- **Description**: Retrieves the current price of a product from a specified URL and logs this information to an Excel or HTML file.
- **Preconditions**: Bot must be operational, and the URL must be accessible.
- **Trigger**: User sends the "!get_price [URL]" command.
- Main Flow:
    1. User sends a command with the URL of the product.
    2. Bot recognizes the command, retrieves the current price from the specified URL using web scraping.
    3. Bot logs the price retrieval event to an Excel and HTML file.
    4. Bot displays the price to the user.
- **Postconditions**: The price is displayed to the user and data is logged.

3.1.7 Start Monitoring Price (!start_monitoring_price)

- **Actor**: User
- **Description**: Initiates an ongoing process to monitor price changes at a specified URL, alerting the user via email if there are price changes.
- **Preconditions**: Bot must be operational, and the URL must be accessible.
- **Trigger**: User sends the "!start_monitoring_price [URL] [frequency]" command.

- Main Flow:
    1. User specifies the URL and frequency of checks.
    2. Bot begins monitoring the price at the given URL at the specified frequency.
    3. For each check, the bot calls the "!get_price" command to log the current price and check for changes.
    4. The bot sends the saved document as an email.
    5. Bot continues to monitor until the "!stop_monitoring_price" command is issued.
- **Postconditions**: Price monitoring is active, logs are being created at each interval, and emails are sent on price changes.

3.1.8 Stop Monitoring Price (!stop_monitoring_price)

- **Actor**: User
- **Description**: Terminates an ongoing price monitoring process and provides a summary of the results.
- **Preconditions**: Price monitoring process must be active.
- **Trigger**: User sends the "!stop_monitoring_price" command.
- Main Flow:
    1. User sends the command to stop monitoring.
    2. Bot receives the command and terminates the ongoing price monitoring.
    3. Bot provides a final summary of monitoring results to the user using the array of results collected during monitoring.
- **Postconditions**: Price monitoring is ceased, and final results are reported to the user.

3.1.9 Check Availability (!check_availability)

- **Actor**: User
- **Description**: Checks the availability of a reservation or booking at a specified URL and logs this information to an Excel or HTML file.
- **Preconditions**: Bot must be operational, and the URL must be accessible.
- **Trigger**: User sends the "!check_availability [URL]" command.
- Main Flow:
    1. User sends a command with the URL where the availability needs to be checked.
    2. Bot recognizes the command, retrieves availability data from the specified URL using web scraping.
    3. Bot logs the availability check event to an Excel and HTML file.
    4. Bot displays the availability status to the user.
- **Postconditions**: The availability status is displayed to the user and data is logged.

3.1.10 Start Monitoring Availability (!start_monitoring_availability)

- **Actor**: User
- **Description**: Initiates an ongoing process to monitor changes in availability at a specified URL, alerting the user via email if there are changes in availability.
- **Preconditions**: Bot must be operational, and the URL must be accessible.
- **Trigger**: User sends the "!start_monitoring_availability [URL] [frequency]" command.
- Main Flow:
    1. User specifies the URL and frequency of checks.
    2. Bot begins monitoring the availability at the given URL at the specified frequency.

3. For each check, the bot calls the "!check_availability" command to log the current availability and check for changes.
4. If an availability change is detected, the bot sends an email with the updated availability information.
5. Bot continues to monitor until the "!stop_monitoring_availability" command is issued.

- **Postconditions**: Availability monitoring is active, logs are being created at each interval, and emails are sent on availability changes.

3.1.11 Stop Monitoring Availability (!stop_monitoring_availability)

- **Actor**: User
- **Description**: Terminates an ongoing availability monitoring process and provides a summary of the results.
- **Preconditions**: Availability monitoring process must be active.
- **Trigger**: User sends the "!stop_monitoring_availability" command.
- Main Flow:
  1. User sends the command to stop monitoring.
  2. Bot receives the command and terminates the ongoing availability monitoring.
  3. Bot provides a final summary of monitoring results to the user using the array of results collected during monitoring.
- **Postconditions**: Availability monitoring is ceased, and results are reported to the user.

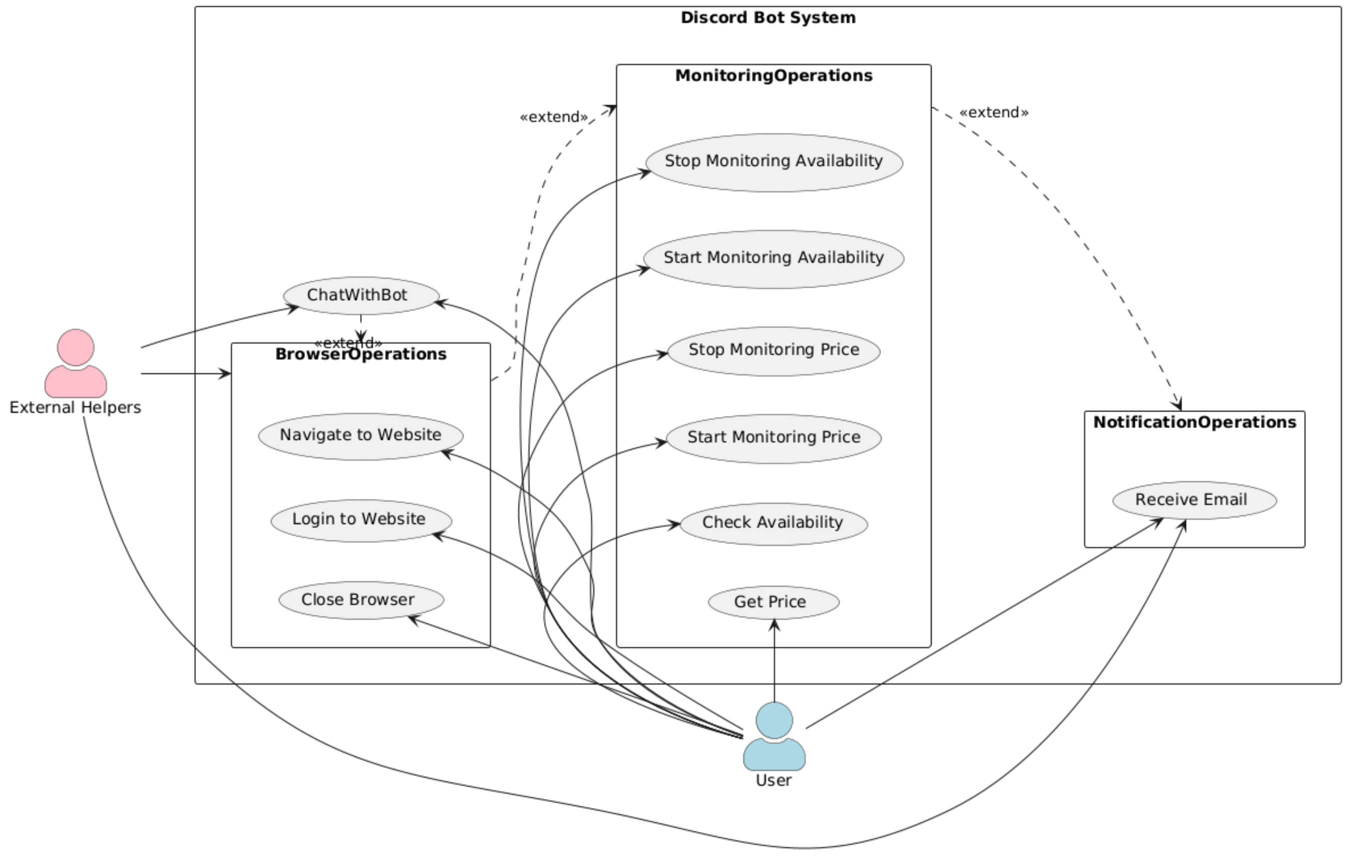


Figure 1: UML use case diagram

3.2 Architecture

The architecture of the Discord bot project forms the backbone of its functionality, providing a robust framework for managing interactions within the Discord environment. This section outlines the system's architectural design, explaining how it supports the operational requirements and enhances the bot's capabilities. By detailing the architectural components and their deployment, this section demonstrates the scalability, reliability, and efficiency of the system.

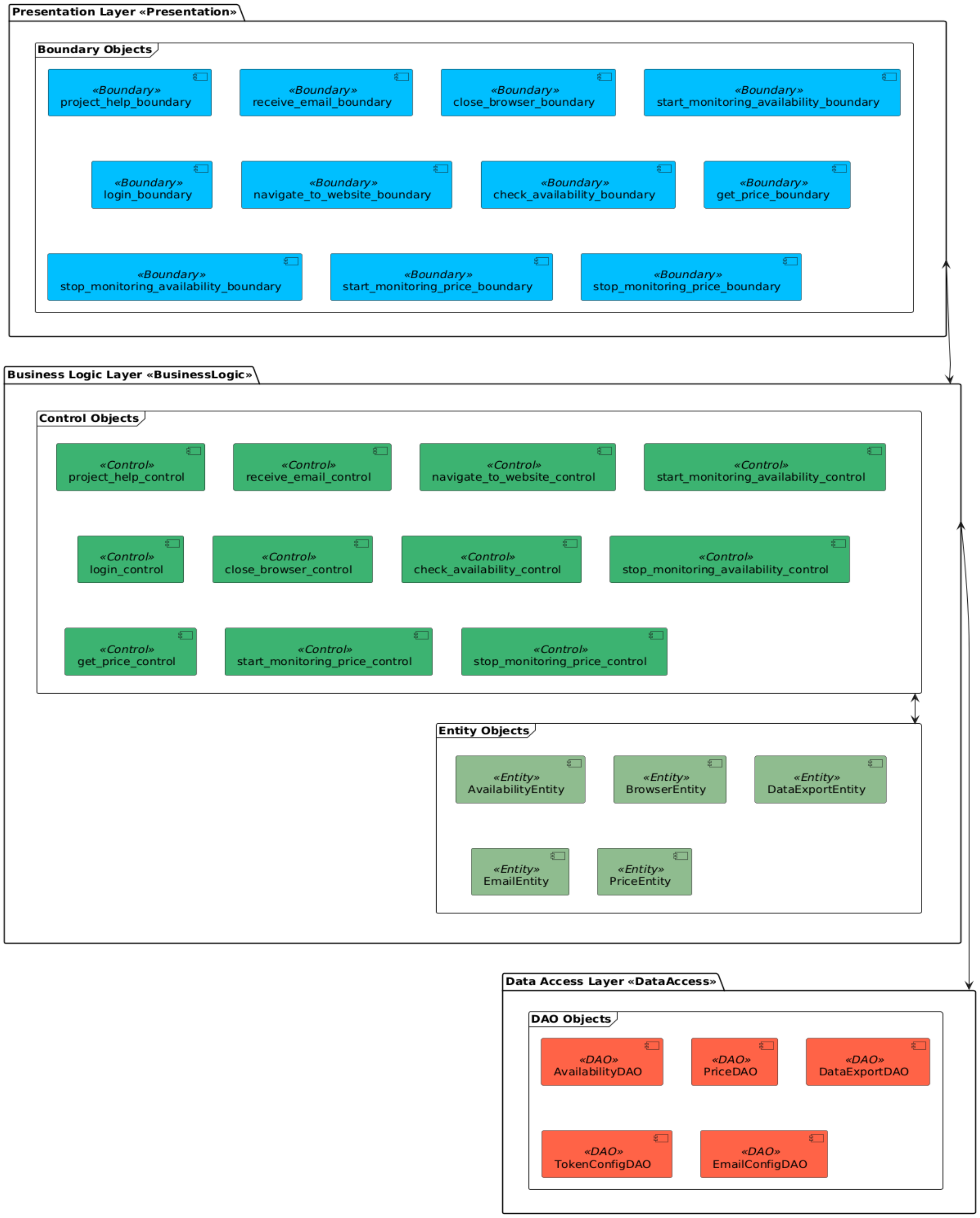

Presentation Layer «Presentation»
Boundary Objects
«Boundary» project_help_boundary
«Boundary» receive_email_boundary
«Boundary» close_browser_boundary
«Boundary» start_monitoring_availability_boundary
«Boundary» login_boundary
«Boundary» navigate_to_website_boundary
«Boundary» check_availability_boundary
«Boundary» get_price_boundary
«Boundary» stop_monitoring_availability_boundary
«Boundary» start_monitoring_price_boundary
«Boundary» stop_monitoring_price_boundary
Business Logic Layer «BusinessLogic»
Control Objects
«Control» project_help_control
«Control» receive_email_control
«Control» navigate_to_website_control
«Control» start_monitoring_availability_control
«Control» login_control
«Control» close_browser_control
«Control» check_availability_control
«Control» stop_monitoring_availability_control
«Control» get_price_control
«Control» start_monitoring_price_control
«Control» stop_monitoring_price_control
Entity Objects
«Entity» AvailabilityEntity
«Entity» BrowserEntity
«Entity» DataExportEntity
«Entity» EmailEntity
«Entity» PriceEntity
Data Access Layer «DataAccess»
DAO Objects
«DAO» AvailabilityDAO
«DAO» PriceDAO
«DAO» DataExportDAO
«DAO» TokenConfigDAO
«DAO» EmailConfigDAO

Figure 2

3.2.1 Entity Objects

These entities act as the data manipulation layer of your architecture, directly interacting with the data sources and external systems to fetch, process, and store the required information. They provide a clean separation of concerns by encapsulating the logic needed to interact with data sources from the rest of the application, ensuring that the control objects can remain focused on

AvailabilityEntity

- **Purpose**: Handles all data operations related to checking and monitoring availability. It directly interacts with external systems or databases to retrieve availability information.
- Key Methods:
  - **check_availability**: Connects to external services to check availability at the given URL on a specified date. It manages direct interactions with web APIs or databases to fetch availability data.
  - **export_data**: Saves or logs availability data to local storage or a database. It might format the data for export to files such as Excel or HTML formats, which are then used for reporting or email notifications.

BrowserEntity

- **Purpose**: Manages all operations that require direct interaction with a web browser, such as opening, navigating, or closing a browser. It encapsulates all functionalities that involve web automation tools like Selenium.
- Key Methods:
  - **launch_browser**: Opens a web browser session with predefined configurations.
  - **navigate_to_website**: Navigates to a specified URL within an open browser session.

        - **close_browser**: Closes the currently open web browser session to free up resources.

DataExportEntity

- **Purpose**: Responsible for exporting data into various formats for storage or transmission. This entity ensures data from operations like price checks or availability monitoring is logged appropriately.
- Key Methods:
    - **export_to_excel**: Formats and writes data to an Excel file, organizing data into sheets and cells according to specified schemas.
    - **export_to_html**: Converts data into HTML format for easy web publication or email attachments.

EmailEntity

- **Purpose**: Handles the configuration and process of sending emails. This entity works with email servers to facilitate the sending of notifications, alerts, or reports generated by the system.
- Key Methods:
    - **send_email_with_attachments**: Prepares and sends an email with specified attachments. It manages attachments, formats the email content, and interacts with email servers to deliver the message.

PriceEntity

- **Purpose**: Specializes in fetching and monitoring price data from various online sources. It uses web scraping techniques to extract pricing information from web pages.

- Key Methods:
  - **get_price**: Retrieves the current price of a product from a specified URL. It scrapes the web page to find pricing information and returns it to the control layer.
  - **export_data**: Similar to the AvailabilityEntity, it exports price data to various file formats for reporting or further analysis.

3.2.2 Boundary Objects

Each boundary object is specifically designed to parse user commands received via Discord, extracting necessary data before interacting with the appropriate control objects to fulfill the user's requests.

project_help_boundary

Interprets the user's request for help, parses the command, and communicates with the bot control to retrieve and display a list of available commands along with their descriptions.

receive_email_boundary

Handles the command to send an email with an attached file, parses the user's message to determine the file to be attached, and coordinates with the control object to manage the email sending process.

close_browser_boundary

Processes the command to close the web browser, parses the message, and instructs the browser control to end the browser session.

login_boundary

Manages the user's command to log into a website, parsing details like the website URL, username, and password before passing them to the browser control for the login operation.

navigate_to_website_boundary

Captures and parses the user's command to navigate to a specific URL, then communicates with the browser control to perform the navigation.

check_availability_boundary

Receives and parses the user's message to extract necessary data such as the URL and date, then contacts the corresponding control object to check availability at the provided URL.

start_monitoring_availability_boundary

Takes the user's input to begin monitoring availability at a specified URL with certain frequency parameters, parses the message, and forwards the data to the control layer to initiate monitoring.

stop_monitoring_availability_boundary

Captures the command to cease monitoring availability, parses the user's instructions, and passes the command to the control object to stop the monitoring process.

get_price_boundary

Receives the command to retrieve a price from a specified URL, parses the command to extract the URL, and contacts the price control to obtain and return the price.

start_monitoring_price_boundary

Receives the command to start monitoring the price at a specified URL and interval, parses the message for necessary details, and forwards these to the price control to begin the monitoring process.

stop_monitoring_price_boundary

Processes the command to stop price monitoring, parses the user's instructions, and notifies the price control to end the monitoring and summarize the findings.

3.2.3 Control Objects

Each control object acts as a decision-making hub that processes input from its corresponding boundary object, directs operations by interacting with entity objects or utilities (like logging or sending emails), and ultimately returns the outcome to the boundary object for user communication.

project_help _control

Generates and returns a list of all available commands and their descriptions, assisting the user in navigating the bot's functionalities.

receive_email_control

Manages the attachment and sending of an email with specified files, liaising with EmailEntity to perform the email operations.

navigate_to_website_control

Checks if the URL is valid or provided. If everything is valid, then contacts the BrowserEntity to perform the actual action.

login_control

Checks if the URL, username, and password are valid or provided. If everything is valid, then contacts the BrowserEntity to perform the actual login action.

close_browser_control

Checks if there is an open session, then contacts the BrowserEntity to close the browser.

check_availability_control

Checks if the URL is provided or not. If not, takes the default URL from the CSS selectors file. Contacts the AvailabilityEntity to verify availability at a specified URL and date, retrieves the availability status, calls the entity's data export method to save data.

start_monitoring_availability_control

Initiates a monitoring process at defined intervals by repeatedly calling the check_availability method, handling the scheduling and continuation of this process, and calls the receive_email method/control object after obtaining data.

stop_monitoring_availability_control

Ends the monitoring process, summarizes the collected data, and returns the final status to the boundary object for user notification.

get_price_control

Checks if the URL is provided or not. If not, takes the default URL from the CSS selectors file. Contacts the PriceEntity to fetch the price at a specified URL and calls the entity's data export method to save data.

start_monitoring_price _control

Initiates a monitoring process at defined intervals by repeatedly calling the get_price method, handling the scheduling and continuation of this process, and calls the receive_email method/control object after obtaining data.

stop_monitoring_price_control

Terminates the price monitoring process, summarizes the collected data, and communicates the results back to the boundary for user notification.

3.2.4 Data Access Layer Objects

Data access layer objects are essential components of the system architecture, acting as conduits between the user-initiated actions at the frontend and the backend functionalities handled by control objects. By pairing each entity object with a corresponding data access layer object, the system ensures seamless interaction with data. These objects are pivotal in enabling CRUD operations on the data managed by the entities.

AvailabilityDAO

Paired with AvailabilityEntity, the AvailabilityDAO abstracts the complexity of CRUD operations related to checking and monitoring availability data. This DAO ensures efficient data handling, enhancing the reliability of availability checks within the system.

PriceDAO

Paired with PriceEntity, the PriceDAO streamlines the integration of price retrieval and monitoring into the system. It ensures data consistency and reliability by managing CRUD operations focused on pricing information.

DataExportDAO

Paired with DataExportEntity, the DataExportDAO manages CRUD operations for data export tasks. This pairing facilitates the transformation of raw data into structured formats like Excel and HTML, enabling efficient data reporting and accessibility.

TokenConfigDAO

Paired with configuration settings, the TokenConfigDAO handles CRUD operations related to authentication tokens and other configuration parameters. This DAO ensures secure handling and retrieval of sensitive configuration data, crucial for maintaining system integrity and security.

EmailConfigDAO

Paired with EmailEntity, the EmailConfigDAO manages CRUD operations related to email configuration settings. This includes handling email server details, user credentials, and recipient information, ensuring that email functionality is robust and reliable.

3.2.5 Associations Among Objects

- Boundary to Control Associations
    - AvailabilityBoundary communicates with AvailabilityControl.
    - BotBoundary communicates with BotControl.
    - BrowserBoundary communicates with BrowserControl.
    - PriceBoundary communicates with PriceControl.
- Control to Entity Associations
    - AvailabilityControl interacts with AvailabilityEntity, DataExportEntity, and EmailEntity.
    - BotControl interacts with EmailEntity.
    - BrowserControl interacts with BrowserEntity.
    - PriceControl interacts with PriceEntity and DataExportEntity.

3.2.6 Aggregates Among Objects

- Availability Aggregate:
    - Root: AvailabilityEntity
    - Includes: AvailabilityControl (manages AvailabilityEntity and potentially accesses DataExportEntity and EmailEntity for output operations).
- Price Aggregate:
    - Root: PriceEntity
    - Includes: PriceControl (manages PriceEntity and handles data through DataExportEntity).
- Email Aggregate:
    - Root: EmailEntity
    - Includes: Both BotControl and AvailabilityControl may use this for sending emails,

positioning it as a shared resource.

3.2.7 Attributes for Each Object

- Boundary Objects Attributes:
    - BotBoundary: commands !stop_bot, !project_help, !receive_email
    - BrowserBoundary: commands !navigate_to_website, !login, !close_browser,
    - AvailabilityBoundary: Commands: !check_availability, !start_monitoring_availability, !stop_monitoring_availability
    - PriceBoundary: commands !get_price, !start_monitoring_price, !stop_monitoring_price
- Control Objects Attributes:
    - AvailabilityControl: monitoring_active (boolean), scheduled_tasks (list of tasks).
    - BotControl: active_sessions (number of active bot sessions).
    - BrowserControl: browser_instance (current instance of the browser).
    - PriceControl: price_history (historical prices), monitoring_active (boolean).
- Entity Objects Attributes:
    - AvailabilityEntity: availability_data, last_checked.
    - BrowserEntity: cookies, session_data.
    - DataExportEntity: file_paths (locations of saved data).
    - EmailEntity: email_queue (emails waiting to be sent).
    - PriceEntity: price_data, last_updated.

3.3 Design

This section delves into the design framework of the Discord bot, illustrating how complex interactions are managed within a structured environment. Through UML class diagrams and detailed descriptions, we explore the bot's operational logic and its component interactions.

The design section outlines the configuration of boundary objects, control objects, and entity objects, each integral to the bot's functionality. Boundary objects serve as the interface between user commands and the system's logic, ensuring inputs are accurately processed. Control objects, the decision-making core, manage the operational flow, orchestrating responses and actions efficiently. Entity objects, responsible for data manipulation, ensuring data integrity and availability, crucial for the bot's operations.

3.3.1 Authentication Subsystem

- **Boundary Objects:** login_boundary
- **Control Objects:** login_control
- **Entity Objects:** BrowserEntity
- **DAO Objects:** TokenConfigDAO

3.3.2 Notification Subsystem

- **Boundary Objects:** receive_email_boundary, project_help_boundary
- **Control Objects:** receive_email_control
- **Entity Objects:** EmailEntity
- **DAO Objects:** EmailConfigDAO

3.3.3 Browser Subsystem

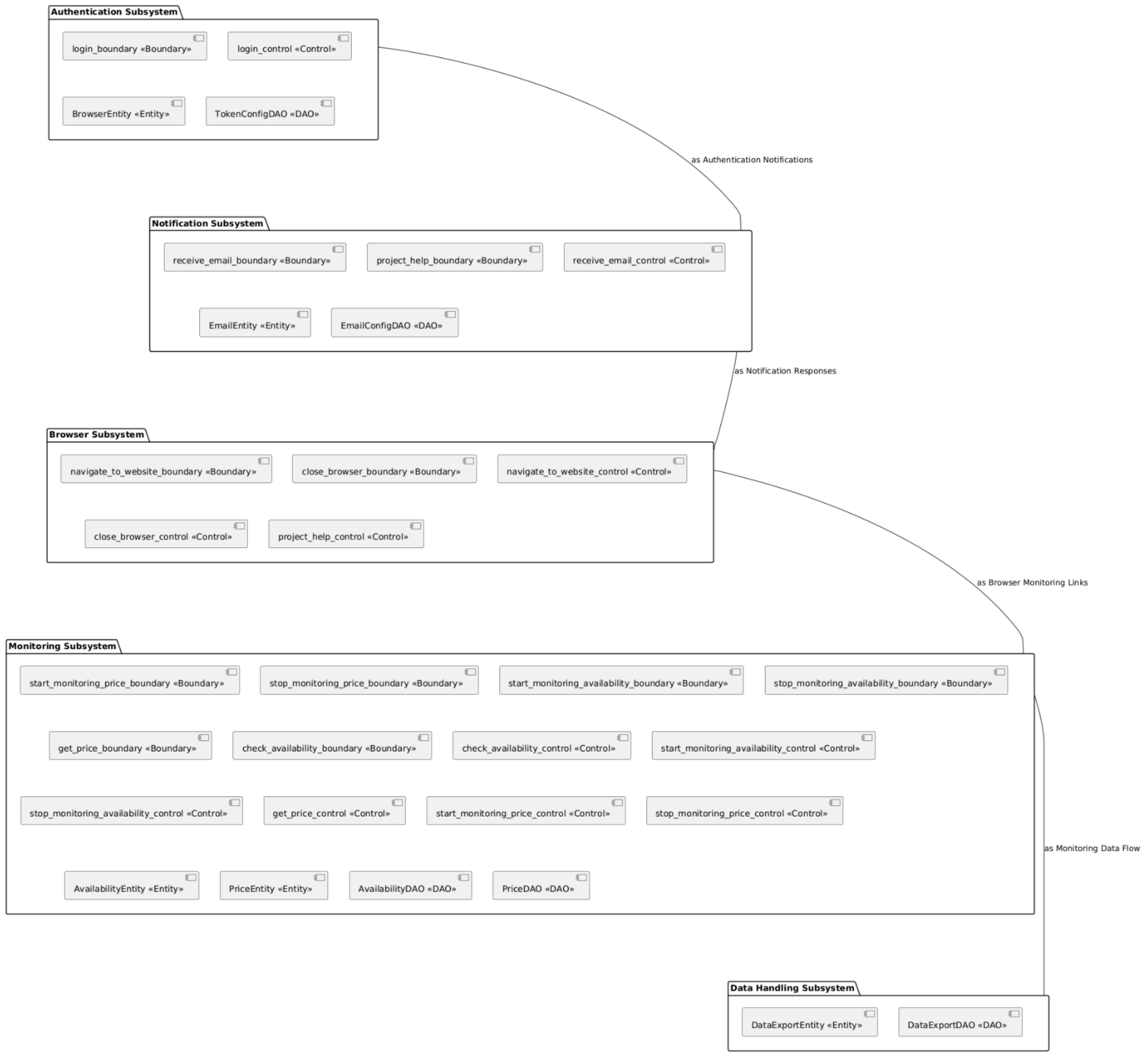
- **Boundary Objects:** navigate_to_website_boundary, close_browser_boundary
- **Control Objects:** navigate_to_website_control, close_browser_control, project_help_control
- **Entity Objects:** BrowserEntity

3.3.4 Monitoring Subsystem

- **Boundary Objects:** start_monitoring_price_boundary, stop_monitoring_price_boundary start_monitoring_availability_boundary, stop_monitoring_availability_boundary, get_price_boundary, check_availability_boundary.
- **Control Objects:** check_availability_control, start_monitoring_availability_control, stop_monitoring_availability_control, get_price_control, start_monitoring_price_control, stop_monitoring_price_control
- **Entity Objects:** AvailabilityEntity, PriceEntity
- **DAO Objects:** AvailabilityDAO, PriceDAO

3.3.5 Data Handling Subsystem

- **Entity Objects:** DataExportEntity
- **DAO Objects:** DataExportDAO



Authentication Subsystem
login_boundary «Boundary»
login_control «Control»
BrowserEntity «Entity»
TokenConfigDAO «DAO»
as Authentication Notifications
Notification Subsystem
receive_email_boundary «Boundary»
project_help_boundary «Boundary»
receive_email_control «Control»
EmailEntity «Entity»
EmailConfigDAO «DAO»
as Notification Responses
Browser Subsystem
navigate_to_website_boundary «Boundary»
close_browser_boundary «Boundary»
navigate_to_website_control «Control»
close_browser_control «Control»
project_help_control «Control»
as Browser Monitoring Links
Monitoring Subsystem
start_monitoring_price_boundary «Boundary»
stop_monitoring_price_boundary «Boundary»
start_monitoring_availability_boundary «Boundary»
stop_monitoring_availability_boundary «Boundary»
get_price_boundary «Boundary»
check_availability_boundary «Boundary»
check_availability_control «Control»
start_monitoring_availability_control «Control»
stop_monitoring_availability_control «Control»
get_price_control «Control»
start_monitoring_price_control «Control»
stop_monitoring_price_control «Control»
AvailabilityEntity «Entity»
PriceEntity «Entity»
AvailabilityDAO «DAO»
PriceDAO «DAO»
as Monitoring Data Flow
Data Handling Subsystem
DataExportEntity «Entity»
DataExportDAO «DAO»

Figure 3: UML Component Diagram

3.4 Interface Specification

This section delves into the interface and class structure of the Discord Bot system, focusing on the interactions and functionalities enabled through the system's architecture. The components, from bot interaction handling to data management and task automation, are elaborated through both visual representation and detailed descriptions.

UML Class Diagram Overview

The UML class diagram, shown in Figure 8, visually represents the architecture of the Discord Bot system, illustrating how various classes are interconnected and interact within the system. This diagram helps to understand the structural relationships and the flow of data across the system, providing insights into the object-oriented design utilized.

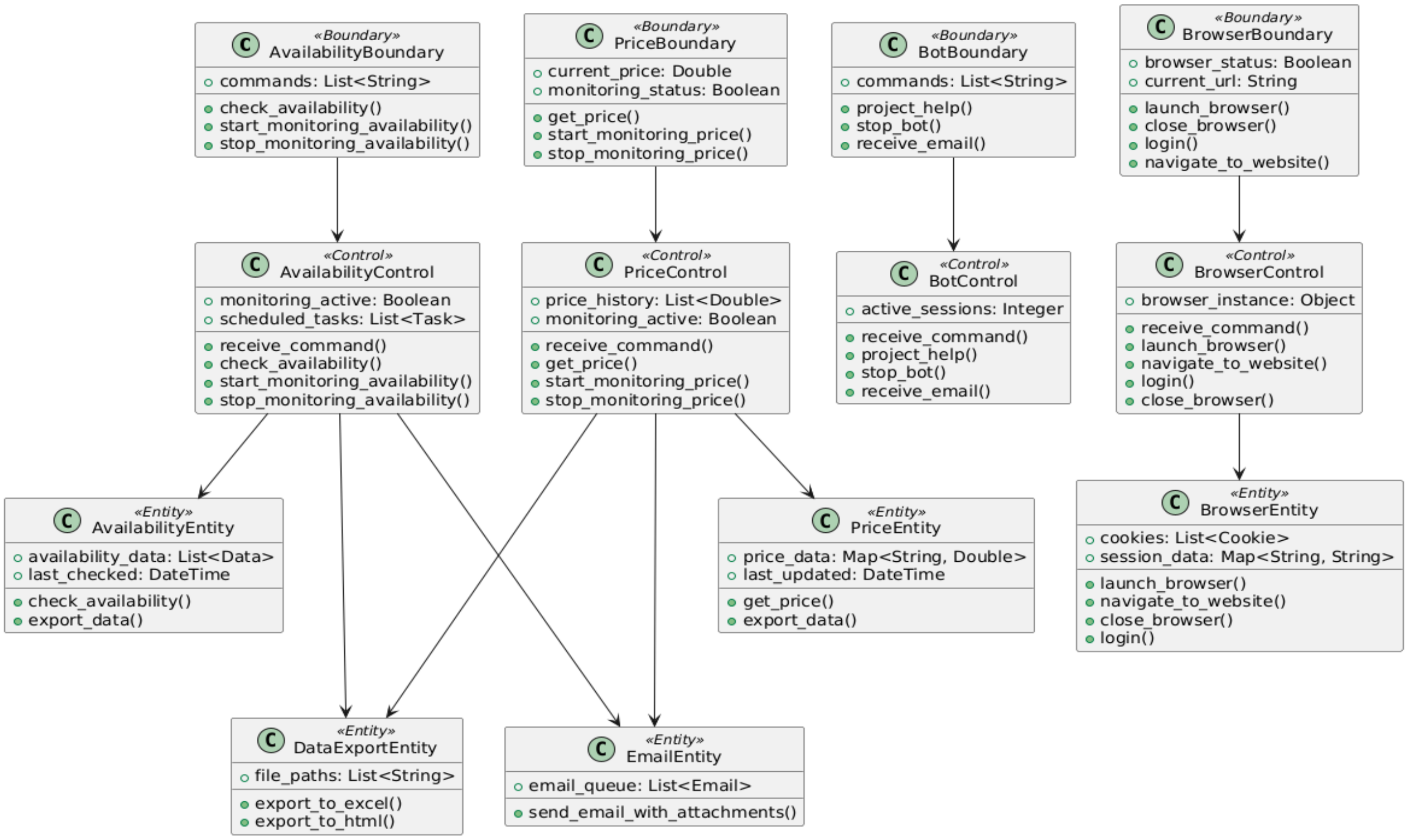


Figure 4: UML Class Diagram

The diagram serves as a high-level depiction of the system's structure. However, it does not encompass all the details of the system’s components due to the complexity and the breadth of the system. To address this, a detailed tabular description for each class in the architecture is provided below. These tables explore the attributes and methods of each class, specifying their visibility, types, and functionality in detail.

3.5 Mapping Contracts to Exception Classes

In the Discord Bot system, ensuring the integrity and correctness of operations is paramount. This is achieved by defining and enforcing contracts through Python’s exception handling mechanisms. These contracts are in the form of preconditions, postconditions, and invariants within the bot's command handling and data processing functionalities.

Preconditions are used to ensure that all necessary conditions are met before executing a command. For example, in the `BrowserControl` class, before navigating to a website, the URL must be validThis checks if the URL conforms to a valid format, raising a `ValueError` if the precondition is not satisfied.

Postconditions confirm that certain conditions hold true after executing a block of code. For instance, when a user requests to stop monitoring availability, the system must confirm that monitoring has ceased. This ensures the monitoring status is appropriately updated, representing a critical postcondition.

Invariants are conditions that must always hold true, regardless of the system’s state. In the context of our bot, one invariant could be that the bot’s response must always be delivered to the correct Discord channel. This is handled implicitly by the Discord commands framework but is critical to the consistent operation of the bot.

Exceptions are mapped to meaningful error messages that provide clarity and feedback to the user. For example, if a command fails due to an invalid parameter, the bot responds with a specific error message. This segment of the bot ensures that any `ValueError` raised (due to URL validation failing the precondition) results in a user-friendly message being sent back to the user, rather than the bot failing silently or crashing.

In addition to localized exception handling, global exception handlers catch and log any unanticipated errors, ensuring that the bot remains operational and that all exceptions are accounted for. This approach to exception handling not only enhances the reliability of the bot but also improves user interaction by providing immediate feedback on the nature of issues encountered during operation.

3.6 Data Management Strategy (CISC695_Assignment9)

In the latest iteration of our Discord bot, a strategic decision was made to deviate from conventional relational database systems, favoring a more agile and less resource-intensive file-based data storage mechanism. This pivot was driven by a comprehensive analysis of the project’s unique requirements, namely, the need for speed, flexibility, and handling predominantly non-transactional data.

System Requirements and Flexibility

The dynamic nature of interactions within the Discord environment necessitates a data management system that can swiftly adapt to changes without the latency often associated with relational databases. The bot's operations, primarily non-transactional and ephemeral data exchanges (such as session data or temporary preferences), benefit significantly from the agility offered by a file-based approach.

Reduced Complexity and Overhead

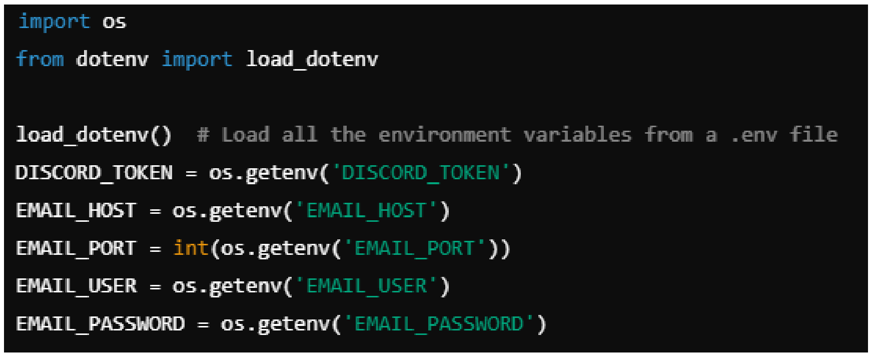
Managing a traditional database involves significant setup, maintenance, and overhead costs. By employing file-based storage, the system sidesteps complexities related to database schema design, data normalization, and transaction management. This reduction in overhead not only simplifies deployment but also enhances the system's overall performance.

Scalability and Security

The chosen strategy simplifies scaling operations horizontally by distributing file storage across multiple nodes or services, without the need for complex database replication strategies. Security is enhanced as sensitive information, such as authentication tokens and SMTP settings. These configurations are loaded into the environment at runtime, isolating sensitive details from the core application logic and minimizing exposure to security vulnerabilities.

Environmental Variables and Security

Tokens and credentials are stored securely within environment files, parsed and loaded at runtime using files like .py or .json. This ensures that sensitive data is not hardcoded into the application’s source code, providing an added layer of security by segregating configuration from deployment.



```python
import os
from dotenv import load_dotenv

load_dotenv()  # Load all the environment variables from a .env file
DISCORD_TOKEN = os.getenv('DISCORD_TOKEN')
EMAIL_HOST = os.getenv('EMAIL_HOST')
EMAIL_PORT = int(os.getenv('EMAIL_PORT'))
EMAIL_USER = os.getenv('EMAIL_USER')
EMAIL_PASSWORD = os.getenv('EMAIL_PASSWORD')
```

Figure 5:Transient and Persistent Data Handling

JSON for Transient Data

Transient data such as user preferences or session states are stored in JSON files. This format is particularly advantageous for its human-readable format and ease of integration with Python, allowing for quick reads and writes. It effectively addresses the need for storing non-sensitive, session-specific data which does not require long-term persistence.

HTML and Excel for Data Reporting

Our system utilizes Excel and HTML formats primarily for the purpose of data reporting and presentation. Excel files are used to organize and analyze data because of their familiar interface and built-in analytical tools which facilitate quick reviews and manipulations by end-users. HTML is employed to format information in a structured and visually appealing manner, particularly useful for generating reports that are distributed via email. These formats allow the bot to provide immediate, readable feedback and detailed reports directly to users, enhancing their interaction and experience.

Data Flow and Real-Time Processing

The data flow is designed to maximize system responsiveness and ensure efficient processing of user commands:

- **Command Processing:** User commands, such as checking prices or setting alerts, are immediately parsed and processed. Depending on the command, this could involve fetching data from web APIs or preparing data for presentation.
- **Immediate Feedback and Output:** Once a command is processed, the system provides feedback directly to the user. For detailed reports or data outputs, HTML is used to present the information clearly and interactively in the user's web browser, while Excel might be used to send detailed data files that users can download and explore.

Advantages of the Current Approach

This approach, using HTML for presentation and Excel for data structuring, leverages their strengths without relying on them for data persistence. It simplifies deployment and avoids the complexities associated with database management and maintenance. Moreover, it allows the bot to operate efficiently with minimal setup, making it ideal for environments where quick deployment and ease of use are priorities.

By focusing on immediate data processing and user feedback, the system ensures that all interactions are handled swiftly and effectively, providing a seamless user experience. This method perfectly suits the interactive nature of a Discord bot, which is designed to offer quick responses and engage users in real-time.

3.7 Technology Stack and Framework

This section delves into the technology stack and frameworks that power the Discord bot, focusing on the tools and technologies that facilitate rapid development, seamless user interaction, and efficient data management.

3.7.1 Programming Languages and Frameworks

Python

- **Role**: Primary programming language for developing the bot.
- **Features**: Chosen for its readability, robust standard library, and extensive support through third-party libraries, Python underpins all major functionalities of the bot, from data scraping to process automation and interaction handling.

Selenium

- **Role**: Automates web browsers to extract real-time product prices and availability.
- **Capabilities**: Simulates human interactions with web pages, allowing the bot to perform complex navigations and data extraction tasks, critical for accurate price monitoring.

Discord.py

- **Role**: Handles communications with the Discord API.
- **Functionality**: Manages user interactions, receives commands, sends notifications, and embeds the bot seamlessly within Discord communities.

3.7.2 Tools and Platforms

Visual Studio Code

- **Role**: Preferred IDE for writing, testing, and debugging the bot’s code.

- **Advantages**: Offers extensive plugin support, built-in Git control, and integrated terminal, which streamline the coding and version control processes.

Git

- **Role**: Manages source code versions and collaborative features.
- **Benefits**: Essential for tracking code changes, managing branches, and integrating changes from multiple contributors, ensuring consistency and continuity in the development process.

GitHub

- **Role**: Hosts the source code repository and facilitates collaborative features like issue tracking and code reviews.
- **Integration**: Centralizes source control and acts as a platform for continuous integration and deployment strategies.

3.7.3 Data Management and Storage

Tjis project utilizes a combination of configuration files, JSON, and direct file output mechanisms for managing both transient and persistent data:

Configuration Files

- **Role**: Manage operational parameters and sensitive credentials, such as Tokens, SMTP settings.
- **Implementation**: Stored in .py files, these parameters are loaded dynamically into the application environment, enhancing security by segregating configuration from the code.

JSON Files

- **Role**: Handle transient data like session states and user preferences.
- **Advantages**: Offers flexibility and speed in accessing and updating data, ideal for non-sensitive, temporary information.

Excel and HTML

- **Role**: Serve as formats for logging long-term data and generating reports.
- **Functionality**: Facilitates easy distribution and accessibility of data, allowing comprehensive reporting and analysis through automated emails.

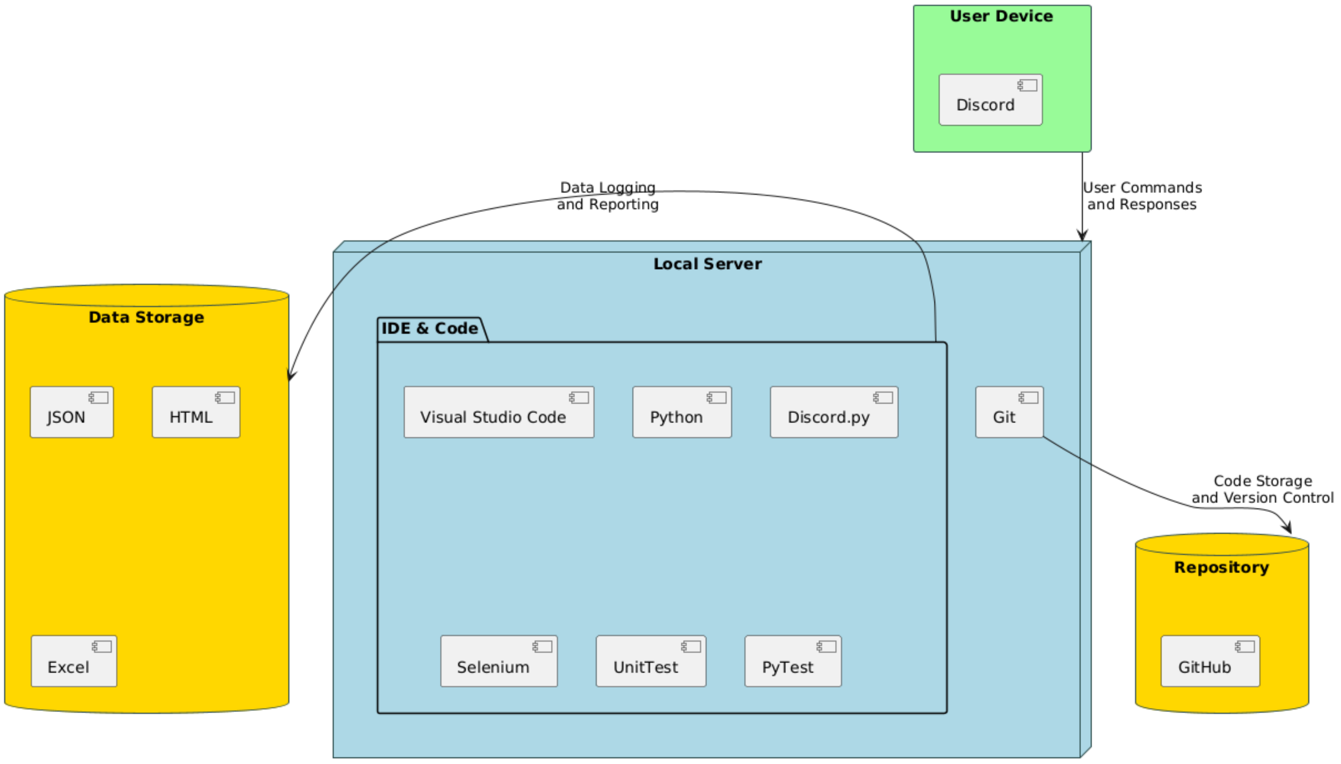


Figure 6: System Architecture Diagram

3.7.4 Testing Strategy

Our project employs a robust testing framework using Python's unittest library and unittest.mock for mocking external dependencies. This strategy ensures that each component of the bot functions as expected under various scenarios. A detailed exploration of our testing approach and methodologies will be presented in the subsequent chapter.

3.8 Conclusion

This chapter has thoroughly examined the structured organization and design of the Discord Bot project, a sophisticated system engineered to facilitate robust interactions within the Discord environment. From detailed use case diagrams to intricate class and component diagrams, we have explored the deep interconnectivity and logical architecture that empower the bot's functionality and operational efficiency.

Project Requirements and Use Cases: Initially, we revisited the bot's foundational requirements, emphasizing practical scenarios it needs to handle, which were illustrated through comprehensive UML diagrams and descriptions. These cases not only demonstrated the bot's responsiveness but also its capability to handle diverse tasks efficiently.

Architectural Design: The architecture section delineated the multi-layered setup of the system, ensuring scalability, reliability, and manageability. By separating concerns across distinct layers—from presentation and business logic to data access—the system's architecture promises enhanced maintainability and easier future expansions.

Detailed Design: In the design segment, we delved into the specifics of each system component. The UML class diagrams provided a clear visualization of the system's structure, showcasing the relationships and responsibilities across various objects within the bot's framework.

Data Management Strategy: Transitioning from a traditional database to a file-based storage system, the project adopts an innovative approach to handle data, which aligns with the dynamic requirements of the Discord environment. This strategy ensures flexibility, rapid data access, and simplifies the system's scalability.

Technology Stack: The chapter also outlined the technology stack that underpins the bot's functionality, highlighting the synergy between various tools and platforms that streamline development and enhance the bot's performance.

As we conclude this chapter, it is evident that the design and architectural decisions made throughout the project are in perfect alignment with the initial goals—creating a responsive, efficient, and scalable bot. Chapter 4 will continue this narrative by focusing on the implementation details, testing strategies using unittest and mock, and how these elements contribute to the robustness of the bot.

Footnote

Code and Text in this documentation has been partially generated with assistance with ChatGPT 4.0.