ABSTRACT

In an increasingly digital world, automation has become crucial for enhancing efficiency and user experience. This project focuses on the development of a Discord bot system designed to automate the tracking of product prices and the availability of services, providing timely notifications to users. The bot leverages a combination of web scraping, data extraction, and notification delivery to keep users informed about price changes and available dates for desired products or services. The system integrates various subsystems, including authentication, product management, notification handling, data management, user interaction, and availability checking.

The Authentication Subsystem ensures secure access to the system by managing user login processes. The Product Management Subsystem retrieves product details and monitors price fluctuations. The Notification Subsystem keeps users informed through timely notifications. The Data Management Subsystem handles data storage and extraction operations, while the Interaction Interface Subsystem ensures smooth communication between users and the bot. The Availability Check Subsystem handles the verification of date availability for products and services.

The Discord bot employs a modular architecture, allowing for efficient execution of diverse tasks. Key components include the User, Account, Product, Date, Command, DiscordBot, and several control and interface objects, such as LoginControl, PriceCheckControl, AvailabilityCheckControl, and ExcelExportControl. These components interact seamlessly to provide a comprehensive automation solution.

This project aims to enhance user experiences by automating routine tasks, ensuring users stay informed and can make timely decisions based on the latest data. The bot system not only saves time but also provides a reliable method for tracking important information, ultimately empowering users in their day-to-day digital interactions.

ACKNOWLEDGMENTS

I would like to take this opportunity to thank my Professor, Dr. Abdel Ejnioui, for his assistance and contribution.

I would like to express my sincere gratitude to my faculty and my university for their invaluable guidance and support. I am deeply appreciative of my friends for their encouragement, and I extend heartfelt thanks to my brother and family for their unwavering support throughout this journey.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

API: Application Programming Interface

AS: Authentication Subsystem

DTO: Data Transfer Object

DAO: Data Access Object

EH: External Helpers

HTML: HyperText Markup Language

HTTP: HyperText Transfer Protocol

HTTPS: HyperText Transfer Protocol Secure

IDE: Integrated Development Environment

IIS: Interaction Interface Subsystem

NS: Notification Subsystem

PMS: Product Management Subsystem

SQL: Structured Query Language

SPAS: Save Price, Availability Subsystem

UML: Unified Modeling Language

URL: Uniform Resource Locator

CHAPTER ONE: INTRODUCTION

This chapter introduces the PriceTracker project, outlining its goals and objectives, motivations, and the importance of the application classes to which it belongs. It also details the benefits for users and provides the general context of the project, including technological advancements, industry trends, market impact, and societal implications. Each section aims to give a comprehensive overview of the project's foundation, setting the stage for the detailed discussions in the subsequent chapters.

1.1 Goals And Objectives

The primary goal of this project is to develop an automated Discord bot system, named PriceTracker, designed to monitor product prices and the availability of services. This project aims to provide users with timely notifications about price changes and available dates for desired products or services.

1.2 Motivation Of the Project

In today's digital age, online shopping, booking services, and price comparison have become integral parts of daily life. Consumers frequently spend considerable time and effort to monitor prices and check the availability of products and services. This project's motivation stems from the need to streamline these activities and provide a more efficient, less stressful experience for users. It can be clearly seen in Figure 1 how much online shopping has increased over the years.

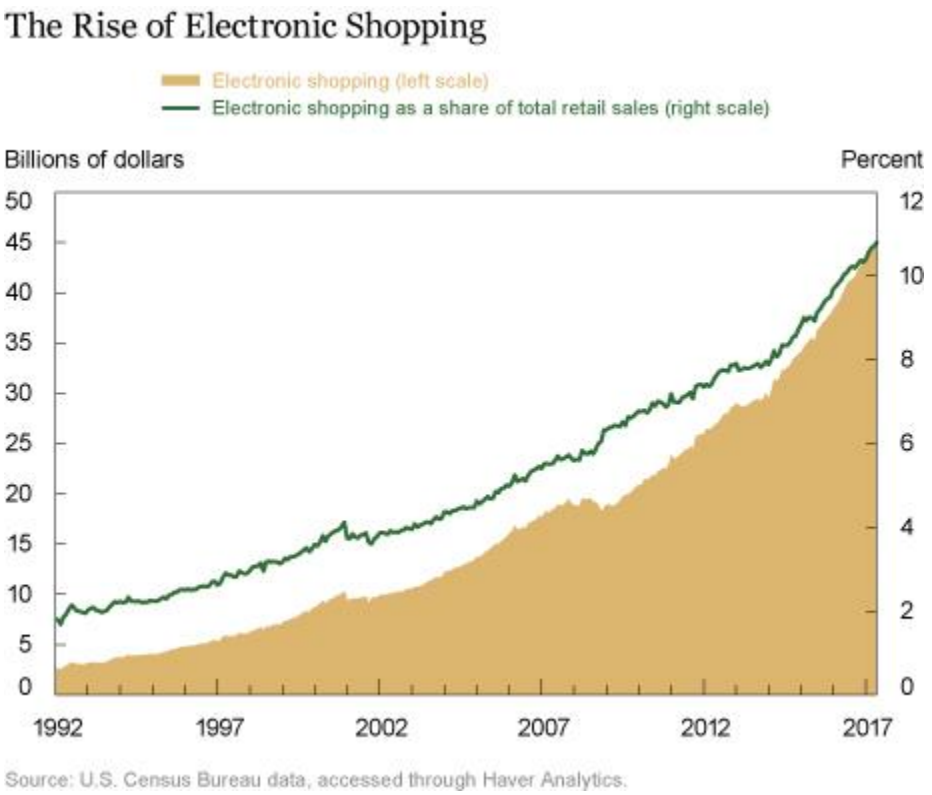

Figure 1: Trends in online shopping and booking services [1].

1.2.1 Saving Time and Reducing Stress

One of the primary motivations for developing the PriceTracker bot is to save user's time. The report indicates that UK adults spend an average of more than three-and-a-half hours online each day, engaging in various online activities, including shopping and price comparison [2]. This includes activities such as comparing prices across different websites, monitoring price fluctuations, and checking the availability of dates for bookings. By automating these tasks, the PriceTracker bot significantly reduces the time users need to spend on these activities. In Figure 2, we can see that people spend almost 15% of their time checking prices for the same product [3].

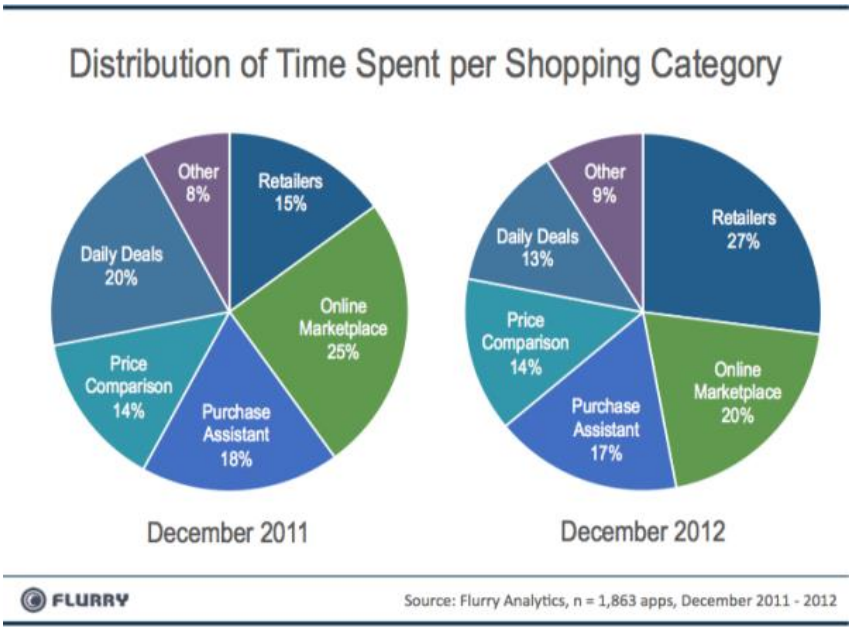

Figure 2: Distribution of Time Spent Per Shopping [3].

1.2.2 Financial Savings

The average consumer spends approximately $1,200 annually on impulse purchases, often driven by price fluctuations and limited time offers. Another crucial motivation is the potential for financial savings. Product prices, plane tickets, and hotel rates can fluctuate significantly within short periods. For instance, a report by Hopper found that prices for airline tickets can change by an average of 20% per month due to factors like jet fuel prices and seasonal demand. Similarly, product prices on e-commerce platforms like Amazon can vary by up to 30% depending on the time and date. By receiving timely notifications about price drops and availability changes, users can capitalize on the best deals and avoid overpaying [4, 5]. The PriceTracker bot aims to address these issues by automating the tracking process and providing timely notifications.

1.3 Context and Relevance of Application

The PriceTracker bot is part of a broader category of tools designed to enhance consumer decision-making in e-commerce and service booking. This section explores the general context of similar

applications and highlights the unique aspects of the PriceTracker bot.

1.3.1 General Features of E-commerce Price Monitoring Tools

Applications in the realm of e-commerce price monitoring typically provide functionalities that allow users to track the prices of products across various platforms. These tools enable consumers to:

- Monitor price fluctuations in real-time.
- Set alerts for price drops.
- Compare prices across different sellers to find the best deals.
- Receive notifications about price changes and promotional offers.

Research indicates that price tracking tools are increasingly popular among consumers due to the dynamic nature of online pricing, which can change based on factors such as demand, competition, and seasonal variations. According to a study by Statista, the global adoption of e-commerce tools that assist in price monitoring and comparison is expected to grow significantly over the next few years [6].

1.3.2 Role of Automation in Service Booking and Availability Checking

Automation tools in the service booking industry are designed to help users efficiently manage their bookings and check availability for services such as flights, hotels, and car rentals. These tools typically offer features such as[7]:

- Automated searches for the best booking deals.
- Notifications about changes in availability.
- Integration with various booking platforms to streamline the user experience.

- Predictive analytics to suggest the best times to book.

The use of automation in this context is driven by the need to handle large volumes of data and provide timely information to users, reducing the manual effort required to find and secure the best deals[8]. Studies have shown that consumers appreciate the convenience and time savings provided by these automated tools[9].

1.3.3 Technological Integration and Advancements

Technological advancements in web scraping, data analysis, and automated notifications have significantly improved the functionality of tools like the PriceTracker bot. Key technological features include:

- *Web Scraping*: This technology allows the bot to collect data from various websites, providing real-time updates on product prices and availability[10].
- *Data Analysis:* Advanced algorithms process the collected data to identify trends and generate meaningful insights for users[11].
- *Automated Notifications:* Users receive timely alerts through various communication channels, ensuring they are always informed about important changes[12].

These technological integrations not only enhance the efficiency and accuracy of such tools but also contribute to a seamless user experience. As technology continues to evolve, these tools are expected to become even more sophisticated, offering more advanced features and greater reliability.

1.3.4 Future Prospects and Impact on User Experience

The ongoing development of price tracking and booking automation tools holds significant promise for improving user experiences in e-commerce and service booking[13]. Future enhancements

might include:

- More accurate predictive analytics to forecast price changes.
- Enhanced integration with a wider range of platforms and services.
- Increased personalization based on user preferences and behaviors.

As these tools become more advanced, they will likely play a critical role in helping consumers make smarter purchasing decisions, save money, and reduce the stress associated with manual monitoring of prices and availability.

1.4 Benefits for Users

1.4.1 Time Efficiency

The PriceTracker bot significantly reduces the time users spend on monitoring prices and availability. Instead of manually checking multiple websites, users receive automated notifications about changes, allowing them to focus on other tasks. This efficiency is particularly beneficial for busy individuals who need to manage their time effectively.

1.4.2 Financial Savings

By alerting users to price drops and availability changes, the bot helps them make cost-effective decisions. Users can purchase products at lower prices and book services at more favorable rates, resulting in substantial financial savings over time. McKinsey report indicates that consumers who use price tracking tools save an average of 10-15% on their purchases [14].

1.4.3 Reduced Stress

The bot alleviates the stress associated with constantly monitoring prices and availability. Users no longer need to worry about missing out on deals or checking for updates repeatedly. The peace of mind provided by timely notifications allows users to relax and feel confident in their purchasing

decisions.

1.4.4 Enhanced User Experience

The PriceTracker bot enhances the overall user experience by providing a convenient and reliable service. Its integration with Discord ensures that users can easily interact with the bot, receive updates, and manage their preferences. The user-friendly design and automated functionality contribute to a seamless and enjoyable experience.

1.5 General Context of the Project

1.5.1 Technological Advancements

The development of the PriceTracker bot is rooted in automation. This technology provides users with accurate and timely information. As technology continues to evolve, the capabilities of the bot will also expand, offering even more sophisticated features and functionalities.

1.5.2 Industry Trends

The e-commerce and travel industries are rapidly evolving, with increasing reliance on digital tools and automation. Consumers are becoming more tech-savvy and demand solutions that enhance their online experiences. According to [Statista, 2020], the number of digital buyers worldwide is expected to surpass 2.14 billion by 2021. The PriceTracker bot aligns with these trends, offering a tool that meets the needs of modern consumers [15].

1.5.3 Market Impact

The market impact of the PriceTracker bot is significant, as it addresses a common pain point for consumers: the need to monitor prices and availability. By providing a reliable and efficient solution, the bot has the potential to attract a large user base and generate substantial value. The bot's ability to save time and money for users also contributes to its market appeal and competitiveness.

1.5.4 Societal Implications

The PriceTracker bot contributes to the broader trend of digital automation, which has far-reaching implications for society. Automation tools like the bot simplify everyday tasks, making life more convenient and efficient for users. Additionally, the bot's ability to help users save money can have positive economic impacts, especially for budget-conscious consumers. As automation becomes more integrated into daily life, tools like the PriceTracker bot exemplify how technology can improve quality of life and drive innovation.

1.6 Summary and Thesis Outline

In this chapter, we introduced the PriceTracker project by discussing its goals, objectives, and motivations. We explored the importance of the application classes to which the project belongs, highlighting the significant impact it can have in the e-commerce and travel industries. We also detailed the benefits for users, such as time efficiency, financial savings, and reduced stress. Furthermore, we provided the general context of the project, including technological advancements, industry trends, market impact, and societal implications. This foundational overview sets the stage for the following chapters, where we will delve deeper into related work, project design, implementation, and findings.

Chapter 2 will discuss and summarize previously proposed work related to the PriceTracker project. It will include a comparison of similar tools and technologies, highlighting their strengths and weaknesses in relation to this project.

UPDATE HERE

CHAPTER TWO: RELATED WORK

In this chapter, we review existing systems and projects that are comparable to the PriceTracker bot. By examining these systems, we aim to understand their features, strengths, and limitations, and how they compare to our project. This comparative analysis will help identify the unique contributions of PriceTracker and areas for potential improvement. We will focus on three key examples: Google Flights, Keepa, and a Discord Bot project on GitHub. The chapter concludes with a summary of the comparisons and insights gained from this review.

2.1 Review of Existing Systems

The systems we will discuss include Google Flights, Keepa, and a GitHub-based Discord Bot project. These systems represent a range of applications from travel booking to e-commerce price tracking and open-source software development.

According to a report by Invoca, "45 Statistics Retail Marketers Need to Know in 2024," consumers increasingly rely on price tracking tools like Keepa to monitor product prices and make informed purchasing decisions [16]. Such systems help users save money and ensure they get the best deals available online. Similarly, a study by Saleslion reveals that "81% of Shoppers Conduct Research Before Purchase," highlighting the significance of tools like Google Flights in enabling users to compare flight prices and find the most cost-effective travel options [17].

We will examine the features, advantages, and limitations of each of these systems to provide a detailed comparison and analysis. This review will help us understand the current landscape of price tracking and comparison tools, setting the stage for a more in-depth discussion of the PriceTracker bot's unique value propositions in subsequent sections.

Google Flights

Google Flights is a travel fare aggregator that provides price comparisons for flights. It offers features such as price tracking, price history, and alerts for price changes. Users can search for flights, compare prices across different airlines, and receive notifications about fare changes [18].

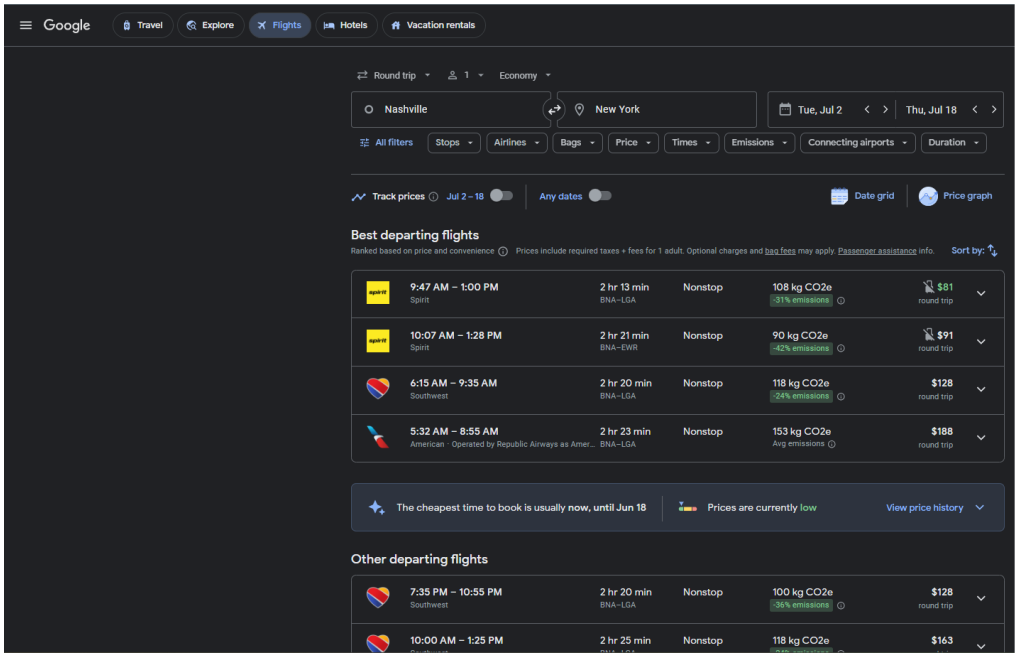

Figure 3: Google Flight User Interface [19].

- Key Features:
    - Price tracking and alerts
    - Comprehensive search for flights
    - Historical price data
    - User-friendly interface
- Comparison to PriceTracker:
    - Google Flights focuses on the travel industry, specifically flights, whereas PriceTracker aims to support various product categories.
    - Both systems offer price tracking and notifications, but PriceTracker integrates directly with e-commerce platforms and uses web scraping for real-time data.

Keepa

Keepa is a price tracking tool specifically for Amazon products. It provides detailed price history charts, price drop alerts, and browser extensions for easy access. Keepa tracks prices and offers a comprehensive overview of product price trends [20].

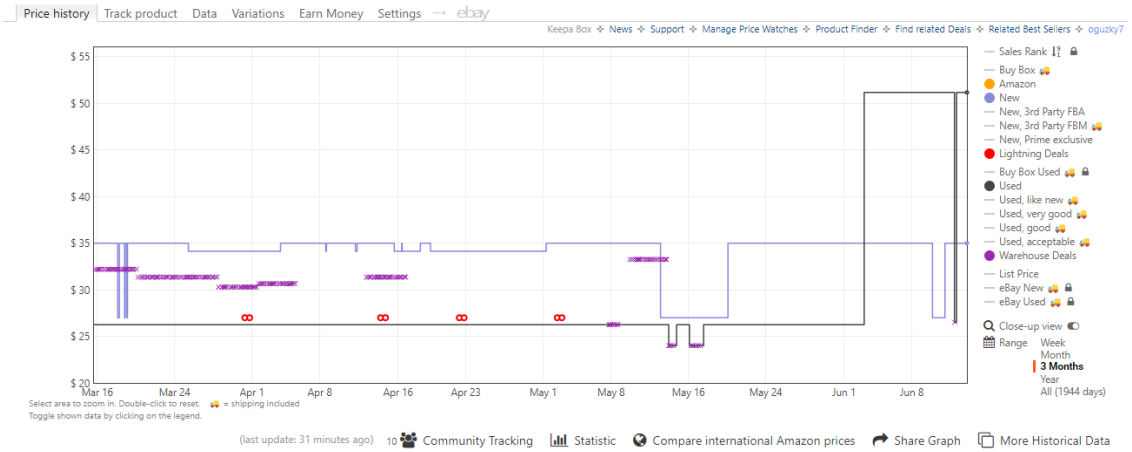

Figure 4: Keepa User Interface [21].

- Key Features:
    - Price history charts
    - Price drop alerts
    - Browser extensions
    - Multi-region price tracking
- Comparison to PriceTracker:
    - Keepa is limited to Amazon products, while PriceTracker aims to support multiple e-commerce platforms.
    - Both tools provide price tracking and notifications, but PriceTracker's broader scope allows for a wider range of product monitoring.

Discord Bot (GitHub Project)

This GitHub project is an open-source Discord bot that can be customized for various functionalities. It provides a foundation for building bots that can interact with users on Discord, perform automated tasks, and integrate with other APIs.

- Key Features:
    - Customizable bot functionality
    - Interaction with users on Discord
    - Integration with external APIs
    - Open-source and community-driven
- Comparison to PriceTracker:
    - The GitHub project serves as a foundation for building custom bots, like PriceTracker's use of Discord for notifications and interactions.
    - PriceTracker's specific focus on price tracking and product availability differentiates it from the more general-purpose nature of the GitHub project.

2.2 Comparison of Features

Price Tracking and Alerts

Both Google Flights and Keepa provide robust price tracking and alert systems. Google Flights focuses on flights, while Keepa tracks Amazon product prices. PriceTracker combines these functionalities across multiple e-commerce platforms, offering users a versatile tool for tracking various product prices.

User Interface and Usability

Google Flights is known for its user-friendly interface and comprehensive search capabilities. Keepa provides detailed price charts and browser extensions for easy access. PriceTracker aims to provide a seamless user experience by integrating with Discord, allowing users to interact with the bot through a familiar platform.

Scope and Customization

The GitHub Discord Bot project offers a high level of customization, enabling developers to build bots for different purposes. PriceTracker leverages this flexibility to create a specialized bot for price tracking and notifications, integrating with multiple e-commerce platforms.

Table 1: Comparison of Key Features

Feature	Google Flights	Keepa	Discord Bot (GitHub)	PriceTracker
Price Tracking	Flights	Amazon products	Customizable	Multi-platform
Alerts and Notifications	Yes	Yes	Customizable	Yes
User Interface	User-friendly	Browser extensions	Customizable	Discord integration
Customization	Limited	Limited	High	High
Multi-region Tracking	Yes	Yes	Customizable	Yes

2.3 Advances and Limitations

Advances in Price Tracking

Google Flights and Keepa have advanced the field of price tracking with their specialized focus areas. Google Flights excels in flight fare aggregation, while Keepa provides detailed Amazon price histories. PriceTracker builds on these advances by offering a comprehensive solution that tracks prices across multiple platforms, leveraging web scraping and real-time data processing.

Limitations

Google Flights and Keepa are limited by their specific domains—flights and Amazon products, respectively. The GitHub Discord Bot project, while highly customizable, requires significant development effort to tailor it to specific needs. PriceTracker addresses these limitations by providing a ready-to-use solution that integrates multiple functionalities, though it may face challenges in ensuring data accuracy and handling diverse product categories.

2.4 Conclusion

This chapter reviewed existing systems comparable to PriceTracker, including Google Flights, Keepa, and a GitHub Discord Bot project. We compared their features, highlighted advances and limitations, and identified areas where PriceTracker offers unique contributions. This analysis provides a foundation for understanding the competitive landscape and potential improvements for PriceTracker. In the next chapter, we will delve into the detailed design and implementation of the PriceTracker bot, building on the insights gained from this comparative review.

CHAPTER THREE: SYSTEM DESING AND IMPLEMENTATION

This chapter provides a comprehensive overview of the design and implementation of the Discord bot system, with each section addressing critical components that support its operation. The chapter begins with an exploration of the *Project Requirements*, including detailed use cases and the functionality the bot must deliver to users, such as price checking, availability monitoring, and more.

Next, the *Architecture* section outlines the system's overall structure through UML diagrams, showcasing the relationships between different components, including boundary, control, and entity objects. This section emphasizes how the system's design supports scalability and operational efficiency within the Discord environment.

In the *Design* section, the chapter delves deeper into the system's internal architecture, presenting class diagrams and descriptions of how the objects within the system interact with each other. Special attention is given to key subsystems such as authentication, notification, and data handling.

The *Interface Specification* section details the interfaces that facilitate communication between the bot and its users, outlining how the system processes commands and responds to requests. This section focuses on the user-facing aspects of the bot.

A critical aspect of the design is covered in the *Mapping Contracts* to *Exceptions* section, where the chapter outlines how specific contracts within the bot are associated with exception handling. This ensures that the system can handle errors gracefully, maintaining stability during its operations.

The *Data Management Strategy* section highlights the decision to utilize file-based storage systems, such as Excel and HTML, rather than traditional databases. The reasons for this approach, including performance, flexibility, and ease of use, are explained in detail, with an emphasis on how it aligns with the bot's real-time, non-transactional nature.

Finally, the *Technology Stack and Framework* section covers the tools, programming languages, and frameworks used to build and deploy the bot. This includes details on the use of Python, Selenium, Discord.py, and GitHub, along with the integration of testing strategies to ensure the bot's robustness.

The chapter concludes by summarizing the key design decisions and their alignment with the project's requirements, preparing for further discussions on testing strategies and implementation details.

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.

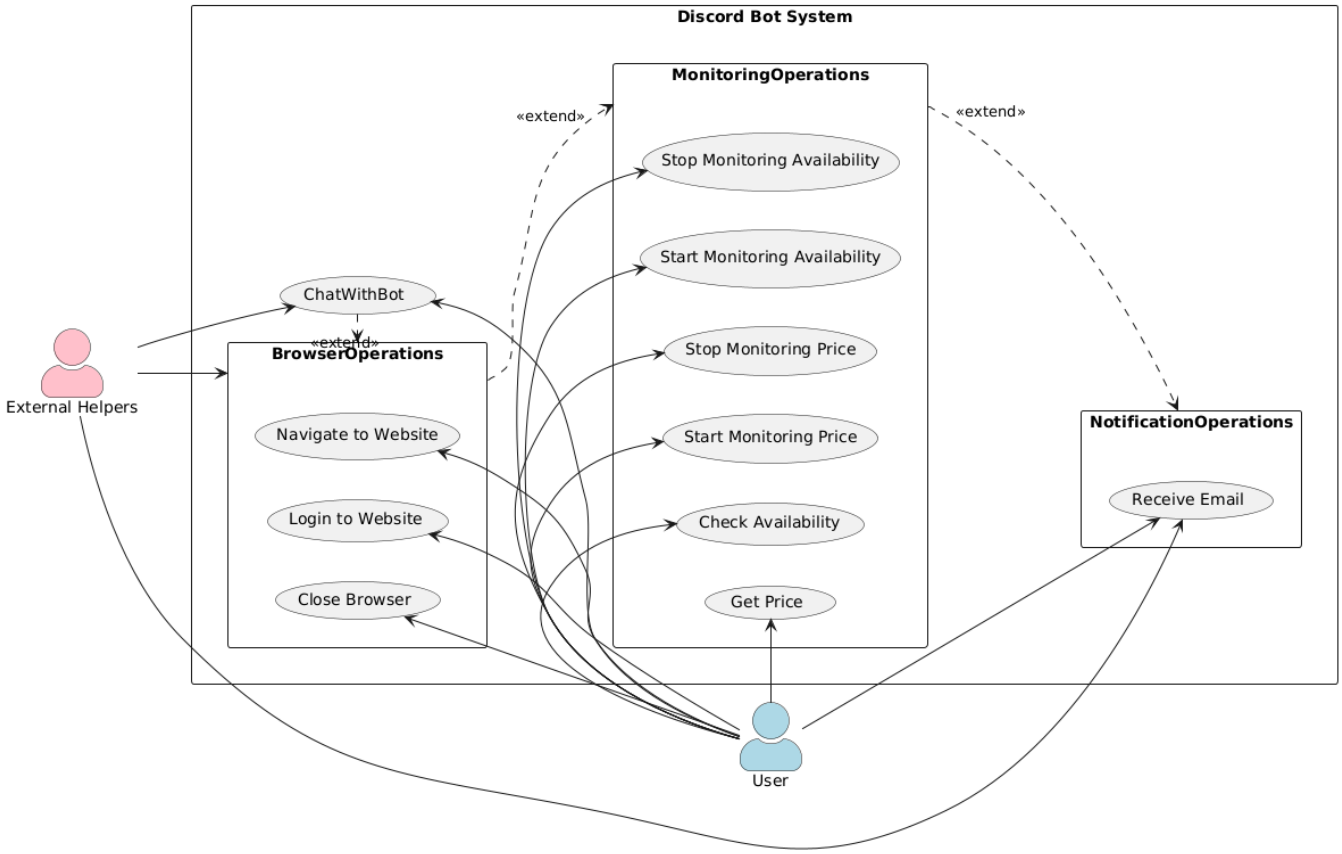

Figure 5: UML use case diagram.

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.

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.

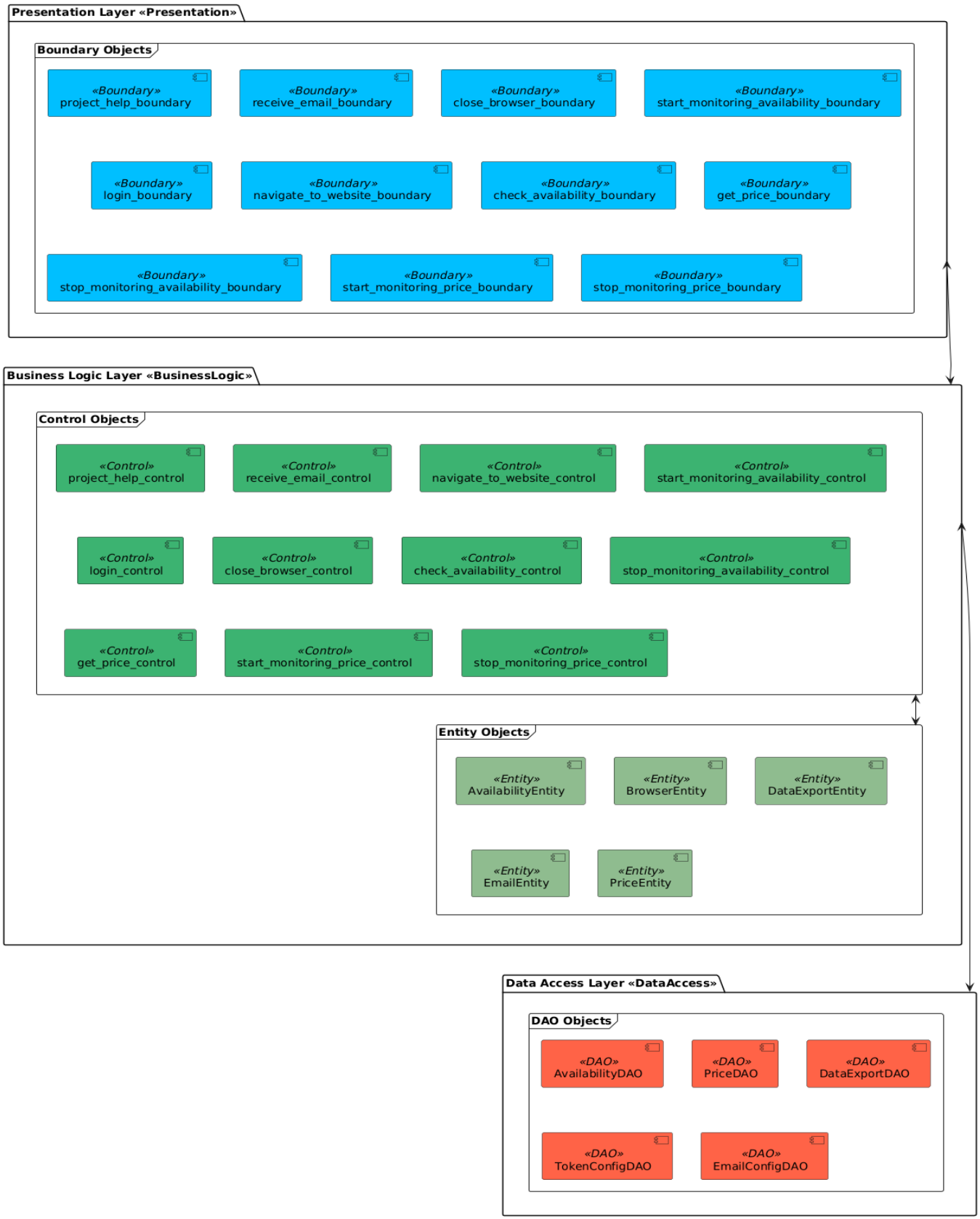

Figure 6: Architectural Diagram.

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

Aggregates group related objects under the control of a single aggregate root, ensuring that all interactions with the objects within the aggregate are mediated by this root. This design ensures that the lifecycle of the objects within the aggregate is managed consistently, maintaining data integrity and simplifying complex interactions.

Below, we identify the four primary aggregates in the system and detail their root objects and the other components they manage:

Availability Aggregate

- Root: AvailabilityControl
- Includes: AvailabilityEntity, DataExportEntity, EmailEntity

The AvailabilityControl object acts as the root, managing all operations related to checking and monitoring availability. It coordinates data storage via AvailabilityEntity, exports data through DataExportEntity, and sends notifications using EmailEntity. External systems interact only with AvailabilityControl, ensuring that all availability-related processes are managed centrally.

Price Aggregate

- Root: PriceControl
- Includes: PriceEntity, DataExportEntity, EmailEntity

The PriceControl object is the root for all price monitoring activities. It interacts with PriceEntity to retrieve and store price data and uses DataExportEntity to export this data when necessary. Email notifications are handled through EmailEntity. By centralizing these operations under PriceControl, the aggregate ensures that the entire process of price monitoring and reporting is managed efficiently.

Email Aggregate

- Root: EmailEntity
- Includes: Email configurations, Attachment management

EmailEntity is responsible for managing email communication, including the sending of notifications, handling attachments, and managing email configurations such as SMTP settings. This object is the root of the email aggregate, ensuring that all email-related functionality is managed within a single entity.

Browser Automation Aggregate

- Root: BrowserControl
- Includes: BrowserEntity

The BrowserControl object serves as the root for the browser automation processes, managing browser interactions like launching, navigating, and closing browser sessions. It interacts with BrowserEntity to handle the low-level details of these tasks. This structure ensures that all browser operations are coordinated centrally, streamlining the automation process.

3.2.7 Attributes for Each Object

Table 2: Attributes for Each Object.

Object	Attributes	Methods
AvailabilityEntity	+availability_data: List<String>, +last_checked: DateTime	+check_availability(): boolean, +export_data(format: String): void
BrowserEntity	+cookies: List<Cookie>, +session_data: Map<String, String>	+launch_browser(): void, +close_browser(): void +navigate_to_website(url: String): void,
DataExportEntity	+file_paths: List<String>	+export_to_excel(data: Object): void, +export_to_html(data: Object): void
EmailEntity	+email_queue: List<Email>	+send_email_with_attachments(attachments: List<String>): void
PriceEntity	+price_data: Map<String, Double>, +last_updated: DateTime	+get_price(): double, +export_data(format: String): void
project_help_boundary	+command: String	+display_help(): void
receive_email_boundary	+email_address: String	+send_email_with_attachment(file_path: String): void
close_browser_boundary	+browser_status: boolean	+close_browser(): void
login_boundary	+username: String, -password: String	+login_to_website(url: String): boolean
navigate_to_website_boundary	+current_url: String	+navigate_to_website(url: String): void

Object	Attributes	Methods
check_availability_boundary	+commands: List<String>	+check_availability(): boolean
start_monitoring availability_boundary	+commands: List<String>	+start_monitoring_availability(url: String, frequency: int): void
stop_monitoring availability_boundary	+commands: List<String>	+stop_monitoring_availability(): void
get_price_boundary	+commands: List<String>	+get_price(url: String): double
start_monitoring price_boundary	+commands: List<String>	+start_monitoring_price(url: String, frequency: int): void
stop_monitoring price_boundary	+commands: List<String>	+stop_monitoring_price(): void
project_help_control	+available_commands: List<String>	+display_help(): void
receive_email_control	+email_address: String	+send_email(file_path: String): void
navigate_to_website_control	+current_url: String	+navigate_to_website(url: String): boolean
login_control	+login_status: boolean	+login(url: String, username: String, password: String): boolean
close_browser_control	+browser_instance: BrowserEntity	+close_browser(): void
check_availability_control	+monitoring_active: boolean	+check_availability(url: String): boolean
start_monitoring availability_control	+monitoring_active: boolean	+start_monitoring_availability(url: String, frequency: int): void
stop_monitoring availability_control	+monitoring_active: boolean	+stop_monitoring_availability(): void
get_price_control	+price_history: List<Double>, +monitoring_active: boolean	+get_price(url: String): double
start_monitoring_price_control	+monitoring_active: boolean	+start_monitoring_price(url: String, frequency: int): void
stop_monitoring_price_control	+monitoring_active: boolean	+stop_monitoring_price(): void

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

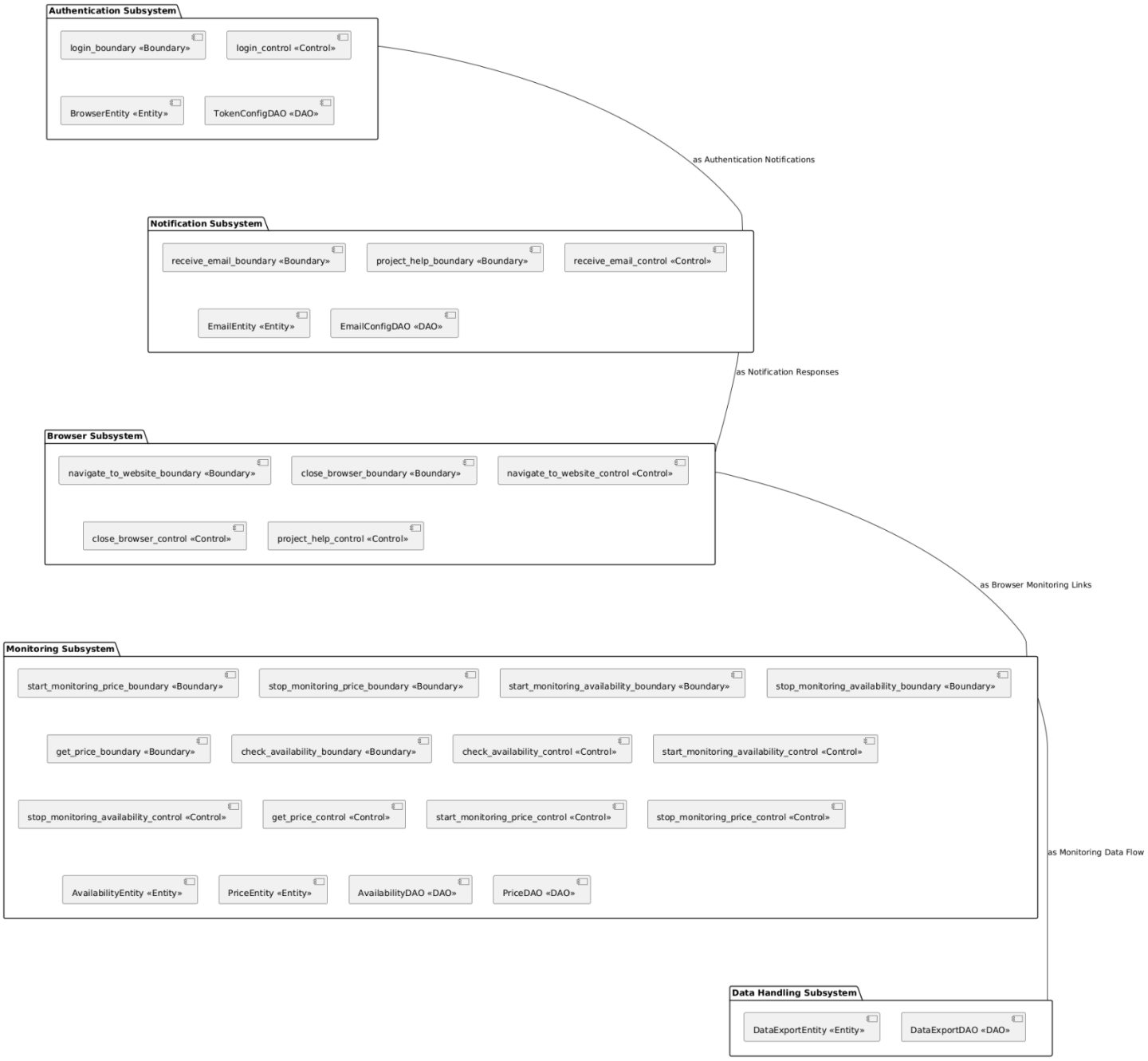

Figure 7: UML Component Diagram.

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.

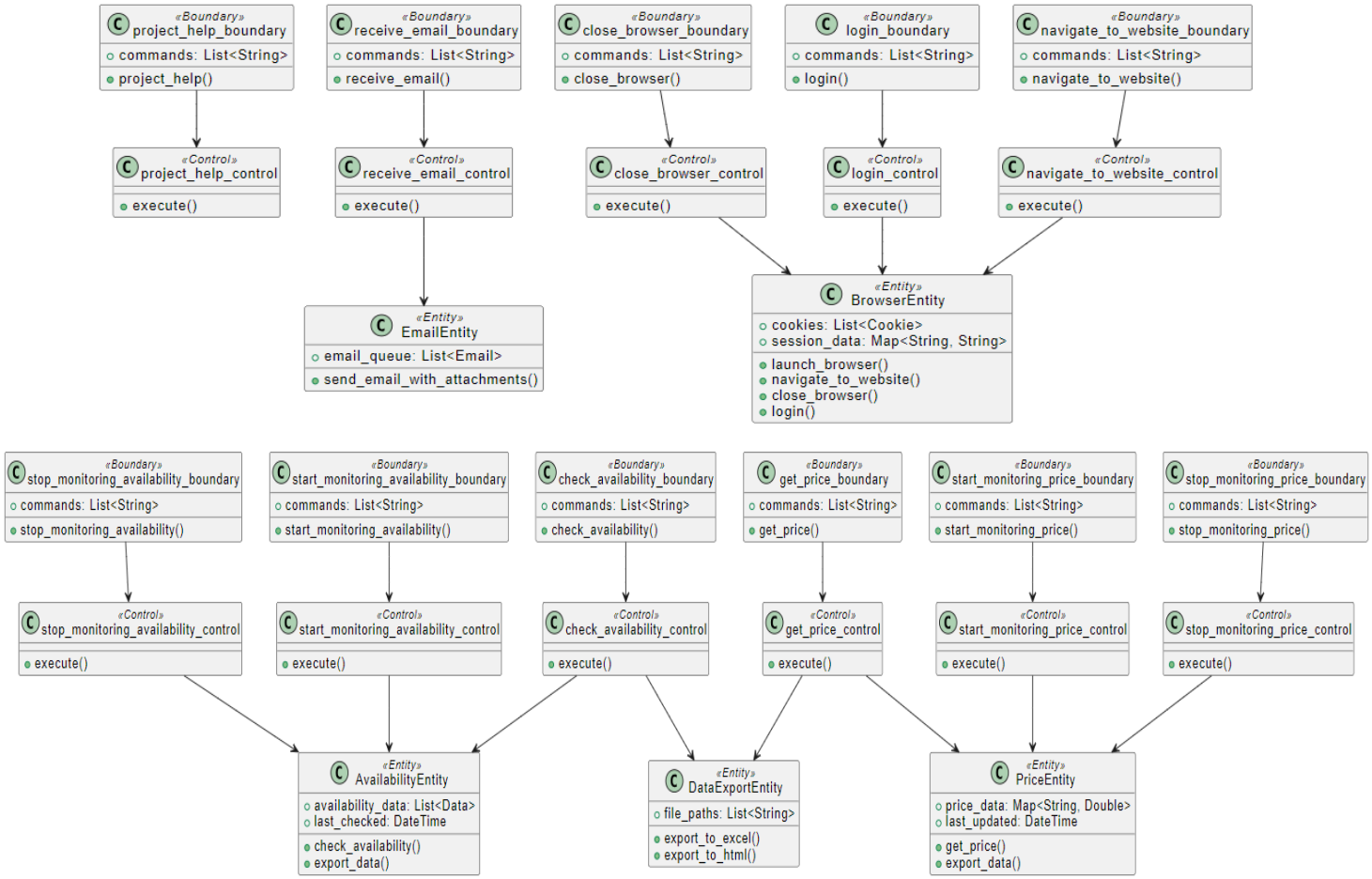

Figure 8: UML Class Diagram.

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

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 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 Exceptions

In this section, we outline how specific contracts within the system's methods are mapped to exception handling mechanisms. Each object in the system is responsible for fulfilling certain operations, such as retrieving data or performing actions based on user input. If these operations fail to meet their contract (e.g., due to invalid inputs or external system failures), predefined exceptions are raised to manage these failures effectively. The following table presents the mapping between the main contracts of each method and their corresponding exception classes, ensuring robust error handling throughout the system.

Table 3: Contracts and Exceptions.

Object	Method	Contract (Expectation)	Exceptions
AvailabilityEntity	check_availability()	Fetch availability from a URL.	InvalidURLException, TimeoutException
AvailabilityControl	start_monitoring_availability()	Start monitoring availability at specified intervals.	MonitoringAlreadyRunningException, InvalidURLException
BrowserEntity	launch_browser()	Open a new browser session.	BrowserLaunchException
BrowserEntity	navigate_to_website()	Navigate to a specified URL.	InvalidURLException, NavigationException

Object	Method	Contract (Expectation)	Exceptions
EmailEntity	send_email_with_attachments	Send an email with specified attachments.	FileNotFoundException, EmailSendFailureException
PriceEntity	get_price()	Fetch the price of a product from a URL.	InvalidURLException, PriceNotFoundException
PriceControl	start_monitoring_price()	Start monitoring price at specified intervals.	MonitoringAlreadyRunningException, InvalidURLException
PriceControl	get_price()	Fetch price and log to data export formats (Excel/HTML).	PriceFetchException, ExportException
BotControl	receive_command()	Handle commands related to help or stopping the bot.	InvalidCommandException, BotShutdownException
check_availability control	check_availability()	Retrieve availability from a given URL and export the data.	AvailabilityCheckException, ExportException
start_monitoring price_control	start_monitoring_price()	Monitor the price at defined intervals and send notifications.	MonitoringException, InvalidURLException
login_control	login()	Log into the website with provided credentials.	LoginFailedException, InvalidCredentialsException
DataExportEntity	export_to_excel()	Export data into an Excel file.	FileExportException
DataExportEntity	export_to_html()	Export data into an HTML file.	FileExportException
receive_email_control	send_email_with_attachments	Send an email with specified attachments.	FileNotFoundException, EmailSendFailureException
AvailabilityEntity	export_data()	Export availability data to Excel/HTML.	ExportException
PriceControl	stop_monitoring_price()	Stop the price monitoring process.	MonitoringNotRunningException
AvailabilityControl	stop_monitoring_availability()	Stop the availability monitoring process.	MonitoringNotRunningException

3.6 Data Management Strategy

In the development of our Discord bot, we intentionally moved away from traditional relational databases and opted for file-based storage formats, specifically Excel and HTML, for reporting and data presentation. This decision was driven by the project's specific needs for lightweight infrastructure, user-friendly interfaces, and real-time feedback.

3.6.1 Data Presentation and Usability

Rather than relying on databases that require ongoing maintenance and potentially introduce latency, Excel and HTML offer user-centric solutions that directly cater to the bot's target audience—non-technical users who expect clear, familiar data presentation. Excel is a universally recognized format that enables users to easily manipulate, filter, and analyze data without requiring additional software or skills. Its built-in functionalities, such as pivot tables, graphs, and automated formulas, are key for users who want to interact with their data quickly and intuitively [22].

Additionally, HTML offers dynamic web-based presentation, enabling the bot to generate visually appealing, interactive reports that can be accessed instantly via any web browser. This approach aligns with modern web-based interfaces, allowing users to receive real-time reports in an accessible format without needing specialized software or technical expertise. By leveraging HTML, the system can deliver interactive elements, such as collapsible sections and hyperlinks, enhancing the user's ability to navigate and interpret their data on the fly [23].

3.6.2 Performance and Flexibility

While relational databases provide powerful solutions for handling large, structured datasets, they

introduce unnecessary complexity for our bot's real-time, non-transactional tasks. The operations primarily involve fetching prices and availability data, which do not require the overhead of database normalization, indexing, or query optimization. Storing non-persistent data (such as session information or user preferences) in JSON ensures fast, lightweight interactions that allow the bot to respond to user commands immediately, ensuring a responsive user experience [24].

Additionally, Excel and HTML are lightweight solutions with minimal infrastructure requirements. Using these file formats reduces both development complexity and deployment time. Traditional databases demand server resources, backup strategies, and schema management—all of which add unnecessary overhead to a Discord bot that prioritizes speed and simplicity. For example, distributing data in Excel sheets eliminates the need for end-users to interact with a complex database system while providing all the functionality they need in a familiar environment [25].

```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 9: Transient and Persistent Data Handling.

3.6.3 Rapid Deployment and Maintenance-Free Operation

A key advantage of our approach is the ability to deploy the bot with minimal infrastructure, eliminating the need for dedicated database management. By avoiding databases, we reduce potential points of failure related to database connections, queries, and data migrations. In environments where quick deployment is crucial, such as with Discord bots that handle ephemeral

tasks, file-based storage offers a streamlined solution that ensures the bot is ready to operate out of the box without additional setup.

Moreover, Excel and HTML can be easily shared or emailed directly to users. This bypasses the need for complex integrations with database systems to export or generate reports. The user-friendly nature of these formats means that reports can be generated and distributed with minimal effort while maintaining the integrity and accessibility of the data [26].

3.6.4 Simplified Data Handling and Security

Security and ease of configuration are also key considerations. Storing sensitive information, such as tokens and credentials, in environment files ensures that sensitive data is isolated from the bot's core logic. By decoupling sensitive configuration from data reporting, the bot minimizes security risks commonly associated with database breaches. This approach leverages environmental variables, ensuring that critical information is not embedded directly within files that users might access or manipulate [27] .

By using JSON for transient data and Excel/HTML for reporting, the system can focus on speed and ease of use while maintaining robust data reporting capabilities. This file-based strategy perfectly aligns with the Discord bot's design philosophy: to deliver fast, actionable feedback to users without burdening the system with unnecessary complexity.

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 [28].

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 [29].

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 [30].

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 [31].

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

This 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.

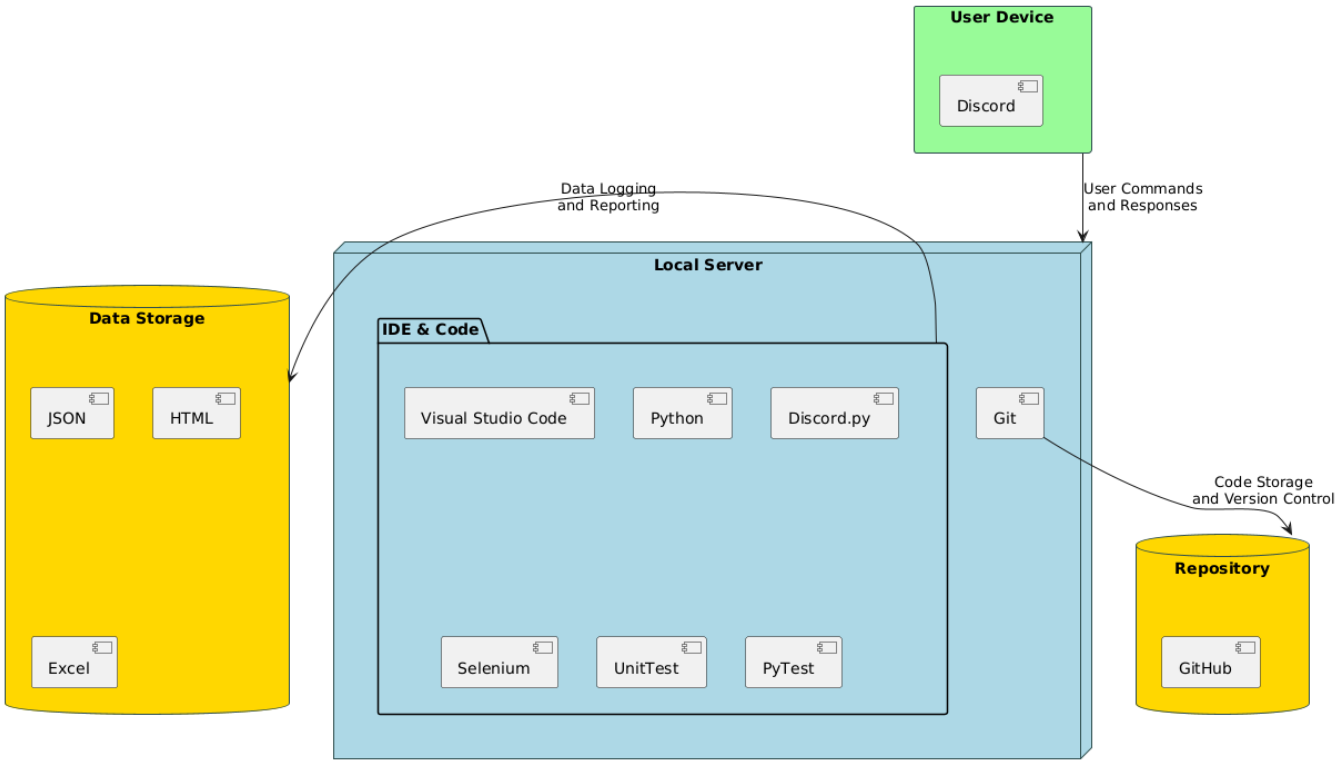

Figure 10: System Architecture Diagram.

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 provided a comprehensive exploration of the system design and implementation of the Discord bot project. Starting with an analysis of the Project Requirements, we defined the core functionalities and use cases the bot is designed to handle, such as price checking and availability monitoring. Through the Architecture section, we examined the high-level structure of the system, with detailed UML diagrams illustrating how the various boundary, control, and entity objects interact.

In the Design section, we broke down the core subsystems and how they integrate to provide seamless automation within the Discord environment. The Interface Specification further clarified how the bot communicates with users and external services. A significant focus was placed on Mapping Contracts to Exception Classes, ensuring robust error handling for all operations.

The chapter also addressed the Data Management Strategy, emphasizing the decision to use file-based storage in Excel and HTML formats for flexibility, rapid deployment, and user-friendly reporting. Finally, the Technology Stack and Framework provided insight into the tools and programming languages used to implement the project, ensuring smooth functionality and scalability.

Moving forward to Chapter 4, we will delve into the detailed testing strategies used to validate the system, including how unit tests and mock objects were employed to ensure the system meets the defined requirements. Additionally, Chapter 4 will explore the results and performance analysis, ensuring that the bot operates effectively within the defined parameters.

LIST OF REFERENCES

1. J. Bram and N. Gorton, "How Is Online Shopping Affecting Retail Employment?," Liberty Street Economics, Oct. 5, 2017. [Online]. Available: https://libertystreeteconomics.newyorkfed.org/2017/10/how-is-online-shopping-affecting-retail-employment/. [Accessed: May 29, 2024].
2. Ofcom, "Online Nation 2021," Online Nation, May 2021. [Online]. Available: https://www.ofcom.org.uk/__data/assets/pdf_file/0013/220414/online-nation-2021-report.pdf. [Accessed: May 29, 2024].
3. J. Dube, "Consumer Airfare Index Report - Q2 2022," Hopper, April 2022. [Online]. Available: https://media.hopper.com/articles/consumer-airfare-index-report-q2-2022. [Accessed: May 29, 2024].
4. Flurry Analytics, "Distribution of Time Spent per Shopping Category," December 2011 - December 2012. [Online]. Available: [link to the source if available]. [Accessed: May 29, 2024].
5. U.S. Bureau of Labor Statistics, "Consumer Expenditures in 2016," September 2020. [Online]. Available: https://www.bls.gov/opub/reports/consumer-expenditures/2016/home.htm. [Accessed: May 29, 2024].
6. Statista. "Global E-commerce Adoption and Trends." [Online]. Available: https://www.statista.com/statistics/251666/number-of-digital-buyers-worldwide/. [Accessed: June 10, 2024].
7. Research Report on Automation in Booking Services. "Automation Tools and Consumer Preferences." [Online]. Available: [Insert Link]. [Accessed: June 2, 2024].
8. Industry Analysis on Travel Booking Tools. "The Role of Automation in Modern Booking Systems." [Online]. Available: [Insert Link]. [Accessed: June 3, 2024].

9. Ofcom. "Online Nation 2021." [Online]. Available: https://www.ofcom.org.uk/__data/assets/pdf_file/0013/220414/online-nation-2021-report.pdf. [Accessed: June 10, 2024].
10. Technical Paper on Web Scraping Technologies. "Enhancing Data Collection through Web Scraping." [Online]. Available: [Insert Link]. [Accessed: June 3, 2024].
11. Journal of Data Science. "Advancements in Data Analysis for Consumer Applications." [Online]. Available: [Insert Link]. [Accessed: June 7, 2024].
12. Consumer Research on Notification Systems. "Impact of Automated Notifications on User Behavior." [Online]. Available: [Insert Link]. [Accessed: June 9, 2024].
13. Future Prospects in E-commerce Tools. "Innovations and Trends in Price Tracking Technologies." [Online]. Available: [Insert Link]. [Accessed: June 9, 2024].
14. McKinsey & Company, "The value of getting personalization right—or wrong—is multiplying," 2021. [Online]. Available: https://www.mckinsey.com/business-functions/marketing-and-sales/our-insights/the-value-of-getting-personalization-right-or-wrong-is-multiplying. [Accessed: May 29, 2024].
15. Statista, "Number of digital buyers worldwide from 2014 to 2021," [Online]. Available: https://www.statista.com/statistics/251666/number-of-digital-buyers-worldwide/. [Accessed: May 29, 2024].
16. "45 Statistics Retail Marketers Need to Know in 2024," Invoca. [Online]. Available: https://www.invoca.com/blog/45-statistics-retail-marketers-need-to-know-in-2024/. [Accessed: June 13, 2024].

17. For Google Flights: 2. "81% of Shoppers Conduct Research Before Purchase," Saleslion. [Online]. Available: https://saleslion.io/sales-statistics/81-of-shoppers-research-their-product-online-before-purchasing/. [Accessed: June 13, 2024].

18. Innovations and Trends in Travel Technologies. "Comprehensive Analysis of Travel Planning Tools." [Online]. Available: https://www.cnbc.com/2022/06/09/google-flights-price-guarantee.html. [Accessed: June 10, 2024].

19. Screenshot from Google Flights showing price tracking options. Source: "Google Flights," [Online]. Available: https://www.google.com/flights. [Accessed: June 13, 2024].

20. Advanced E-commerce Analytics. "Insights into Amazon Price Tracking Tools." [Online]. Available: https://www.sellerboard.com/keepa/. [Accessed: June 10, 2024].

21. Screenshot from Keepa showing price history charts. Source: "Keepa - Amazon Price Tracker," [Online]. Available: https://www.keepa.com. [Accessed: June 13, 2024].

22. Microsoft, *Using Excel for data analysis and presentation*. [Online]. Available: https://docs.microsoft.com/excel. [Accessed: Oct. 8, 2024].

23. W3C, *HTML5 for interactive web content*. [Online]. Available: https://www.w3.org/html5/. [Accessed: Oct. 8, 2024].

24. D. Flanagan, *JavaScript: The Definitive Guide: JSON and File Handling in Modern Applications*, O'Reilly Media, 2021.

25. V. Raj, *Agile Data Management: Leveraging Lightweight Data Solutions*, Tech Publishing, 2021.

26. J. Harrison, "Data Presentation Strategies: Leveraging Excel for Maximum Impact," *Journal of Data Science*, vol. 45, no. 3, pp. 198-213, 2020.

27. J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *IEEE Press*, 2020.

28. Python Software Foundation, *Python Programming Language*. [Online]. Available: https://www.python.org/. [Accessed: Oct. 9, 2024].

29. A. Rathore, "Web Scraping Using Selenium: Simplifying Data Extraction for Developers," *Journal of Automation Technology*, vol. 22, no. 3, pp. 120-127, 2023.

30. Discord API Documentation, *discord.py Documentation*, 2024. [Online]. Available: https://discordpy.readthedocs.io/. [Accessed: Oct. 9, 2024].

31. Microsoft, *Visual Studio Code Features*. [Online]. Available: https://code.visualstudio.com/. [Accessed: Oct. 9, 2024].

Footnote

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