**Discord Bot Automation Assistant**

**Discord Bot Automation Assistant Chapter 3**

**Oguz Kaan Yildirim**

**307637**

# TABLE OF CONTENTS

[Table of Contents 2](#_Toc178958374)

[CHAPTER THREE: PROJECT ORGANIZATION/STRUCTURE 4](#_Toc178958375)

[1. Project Requirements 4](#_Toc178958376)

[1.1 Use Case Diagram 4](#_Toc178958377)

[1.1.1 Actors 4](#_Toc178958378)

[1.1.2 Use Case Groups 4](#_Toc178958379)

[1.1.3 Extensions 5](#_Toc178958380)

[1.2 Use Case Descriptions 6](#_Toc178958381)

[1.2.1 Account Management 6](#_Toc178958382)

[1.2.1.1 Add Account 6](#_Toc178958383)

[1.2.1.2 Fetch All Accounts 6](#_Toc178958384)

[1.2.1.3 Fetch Account by Website 6](#_Toc178958385)

[1.2.1.4 Delete Account 6](#_Toc178958386)

[1.2.2 Browser Operations 7](#_Toc178958387)

[1.2.2.1 Launch Browser 7](#_Toc178958388)

[1.2.2.2 Close Browser 7](#_Toc178958389)

[1.2.2.3 Navigate to Website 7](#_Toc178958390)

[1.2.3 Login 8](#_Toc178958391)

[1.2.3.1 Login to Website 8](#_Toc178958392)

[1.2.4 Price Monitoring 8](#_Toc178958393)

[1.2.4.1 Get Price 8](#_Toc178958394)

[1.2.4.2 Start Monitoring Price 8](#_Toc178958395)

[1.2.4.3 Stop Monitoring Price 9](#_Toc178958396)

[1.2.5 Availability Monitoring 9](#_Toc178958397)

[1.2.5.1 Check Availability 9](#_Toc178958398)

[1.2.5.2 Start Monitoring Availability 9](#_Toc178958399)

[1.2.5.3 Stop Monitoring Availability 10](#_Toc178958400)

[1.2.6 DataExport 10](#_Toc178958401)

[1.2.6.1 DataExport 10](#_Toc178958402)

[1.2.7 Notifications and Emailing 10](#_Toc178958403)

[1.2.7.1 Receive Email 10](#_Toc178958404)

[1.2.7.2 Chat with Bot 11](#_Toc178958405)

# CHAPTER THREE: PROJECT ORGANIZATION/STRUCTURE

This chapter covers the system design and implementation details of the Discord Bot Automation Assistant. We will start with the project requirements, including the use case diagram and descriptions of the use cases. Then, we will present the architecture, including UML component, deployment, and activity diagrams. The design section will include UML package and class diagrams. Finally, we will discuss the technology stack and frameworks used, followed by a short conclusion.

## 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.

### Use Case Diagram

#### Actors

* **User**: Represents the person interacting with the bot.
* **ExternalHelpers**: External systems or tools that assist with browser operations (e.g., Selenium).

#### Use Case Groups

* **Account Management**: Covers account-related commands.
* **Browser Operations**: Covers commands related to launching, closing, and navigating the browser.
* **Login**: Dedicated to logging into websites using stored credentials.
* **Price and Availability Monitoring**: Tracks prices and service availability, respectively.
* **Data Export**: Automatically exports data as part of the monitoring process.
* **Notifications & Email**: Sends email notifications with exported data.
* **Chat with Bot**: Serves as the primary communication interface where users send commands and interact with the bot.

#### Extensions

* **Data Export** is an extension of both **Price Monitoring** and **Availability Monitoring**, as exporting happens after data retrieval.
* **Notifications & Email** is an extension of **Data Export**, as emails are sent after exporting data.
* **Chat with Bot** extends other use cases, as it serves as the interface through which all actions are triggered.

A diagram of a diagram

Description automatically generated

### Use Case Descriptions

#### Account Management

This group of use cases handles managing user accounts in the bot.

##### Add Account

* + **Description**: Allows the user to store account credentials for a specific website.
  + **Actor**: User
  + **Precondition**: The user provides valid credentials (username, password, and website).
  + **Postcondition**: The account details are stored in the bot's database for future use.
  + **Command**: !add\_account <username> <password> <website>

##### Fetch All Accounts

* + **Description**: Retrieves all stored accounts for the user.
  + **Actor**: User
  + **Precondition**: The user has accounts stored in the database.
  + **Postcondition**: The bot returns a list of all stored accounts.
  + **Command**: !fetch\_all\_accounts

##### Fetch Account by Website

* + **Description**: Fetches the stored account details for a specific website.
  + **Actor**: User
  + **Precondition**: The user has an account stored for the specified website.
  + **Postcondition**: The bot returns the account details for the specified website.
  + **Command**: !fetch\_account\_by\_website <website>

##### Delete Account

* + **Description**: Removes an account from the database.
  + **Actor**: User
  + **Precondition**: The user specifies the account ID of an existing account.
  + **Postcondition**: The account is deleted from the database.
  + **Command**: !delete\_account <account\_id>

#### Browser Operations

These use cases revolve around the bot automating browser activities.

##### Launch Browser

* + **Description**: Opens a browser window.
  + **Actor**: User
  + **Precondition**: No browser is currently open.
  + **Postcondition**: A browser window is opened, ready for further actions.
  + **Command**: !launch\_browser

##### Close Browser

* + **Description**: Closes an open browser window.
  + **Actor**: User
  + **Precondition**: A browser window is open.
  + **Postcondition**: The browser window is closed.
  + **Command**: !close\_browser

##### Navigate to Website

* + **Description**: Navigates the browser to a specified URL.
  + **Actor**: User
  + **Precondition**: The browser is open, and a valid URL is provided.
  + **Postcondition**: The browser navigates to the provided URL.
  + **Command**: !navigate\_to\_website <url>

#### Login

This use case focuses on logging into a website using stored credentials.

##### Login to Website

* + **Description**: Logs into a website using the credentials stored in the Account Management system.
  + **Actor**: User
  + **Precondition**: The user has stored credentials for the specified website.
  + **Postcondition**: The user is logged into the website within the browser.
  + **Command**: !login <website>

#### Price Monitoring

These use cases allow the bot to track the prices of products and export the data.

##### Get Price

* + **Description**: Fetches the current price of a product from a specified URL.
  + **Actor**: User
  + **Precondition**: A valid product URL is provided.
  + **Postcondition**: The bot returns the current price and logs it in an Excel and HTML file.
  + **Command**: !get\_price <url>

##### Start Monitoring Price

* + **Description**: Starts continuous monitoring of the price for a specified product.
  + **Actor**: User
  + **Precondition**: The user provides a valid URL.
  + **Postcondition**: The bot periodically checks the price and sends notifications if the price changes.
  + **Command**: !start\_monitoring\_price <url>

##### Stop Monitoring Price

* + **Description**: Stops continuous price monitoring.
  + **Actor**: User
  + **Precondition**: The bot is actively monitoring the price.
  + **Postcondition**: The monitoring process is stopped.
  + **Command**: !stop\_monitoring\_price

#### Availability Monitoring

These use cases monitor the availability of services (e.g., restaurant reservations).

##### Check Availability

* + **Description**: Checks the availability of a service for a specific date.
  + **Actor**: User
  + **Precondition**: The user provides a valid URL and date.
  + **Postcondition**: The bot returns availability information and logs it in an Excel and HTML file.
  + **Command**: !check\_availability <url> <date>

##### Start Monitoring Availability

* + **Description**: Starts continuous monitoring of availability for a service.
  + **Actor**: User
  + **Precondition**: The user provides a valid URL and date.
  + **Postcondition**: The bot periodically checks availability and sends notifications when it updates.
  + **Command**: !start\_monitoring\_availability <url> <date>

##### Stop Monitoring Availability

* + **Description**: Stops continuous availability monitoring.
  + **Actor**: User
  + **Precondition**: The bot is actively monitoring availability.
  + **Postcondition**: The monitoring process is stopped.
  + **Command**: !stop\_monitoring\_availability

#### DataExport

This use case automatically exports data retrieved from price and availability monitoring into Excel and HTML files.

##### DataExport

* + **Description**: Exports data to Excel and HTML formats after retrieving price or availability information.
  + **Actor**: System (triggered by other use cases)
  + **Precondition**: Price or availability data has been retrieved.
  + **Postcondition**: Data is exported and stored locally.

#### Notifications and Emailing

These use cases handle sending notifications and emails based on monitoring tasks.

##### Receive Email

* + **Description**: Sends an email containing the exported data (HTML or Excel file) to the user.
  + **Actor**: User
  + **Precondition**: The user provides the name of the exported file.
  + **Postcondition**: The specified file is sent to the user's email.
  + **Command**: !receive\_email <file\_name>

##### Chat with Bot

* + **Description**: Enables users to interact with the bot via Discord, sending commands and receiving results or notifications.
  + **Actor**: User
  + **Precondition**: The bot is online and listening for commands.
  + **Postcondition**: The bot responds to the user's commands with relevant results, errors, or notifications.
  + **Command**: !<command> (e.g., !get\_price, !start\_monitoring\_price, etc.)

## Architecture Section

1. **1. Introduction to Architecture**

The architecture of the **Discord Bot Automation Assistant** follows a layered pattern, consisting of three main layers: Boundary Layer, Control Layer, and Entity Layer. Each layer is designed with clear responsibilities, adhering to the principles of modularity and separation of concerns, which improves maintainability and scalability.

This system is structured to manage various features, such as account management, browser automation, price and availability monitoring, and data exportation. It also includes communication with external services, such as websites and databases.

1. **2. Component Diagram**
2. **Title: UML Component Diagram for Discord Bot Automation Assistant**

The **Component Diagram** visualizes the overall architecture by dividing the system into three layers: Boundary, Control, and Entity. Here is an explanation of each layer:

* **Boundary Layer**: This layer is responsible for handling interactions with the user via Discord. Commands such as !get\_price, !check\_availability, or !add\_account are received by the respective boundary classes, which forward the commands to the control layer.
  + Classes include AccountBoundary, BrowserBoundary, AvailabilityBoundary, and PriceBoundary.
* **Control Layer**: This layer manages the business logic of the system. It processes commands from the boundary layer and interacts with the entity layer to fulfill requests.
  + Classes include AccountControl, BrowserControl, AvailabilityControl, and PriceControl.
* **Entity Layer**: This layer is responsible for interacting with external systems, such as the database, websites, and file exports. It includes the core logic for data persistence and web interaction.
  + Classes include AccountDAO (for database operations), BrowserEntity (for browser automation), PriceEntity, and AvailabilityEntity.

The interaction between these layers ensures that the system operates efficiently by maintaining a clear separation between user interaction, business logic, and data handling.

plaintext

Copy code

@startuml

[Discord User] as DU

package "Boundary Layer" as BL{

[AccountBoundary]

[BrowserBoundary]

[AvailabilityBoundary]

[PriceBoundary]

}

package "Control Layer" as CL{

[AccountControl]

[AvailabilityControl]

[BrowserControl]

[PriceControl]

}

package "Entity Layer" as EL{

[AccountDAO]

[BrowserEntity]

[PriceEntity]

[AvailabilityEntity]

}

DU --> [AccountBoundary]

DU --> [BrowserBoundary]

DU --> [AvailabilityBoundary]

DU --> [PriceBoundary]

[AccountBoundary] --> [AccountControl]

[BrowserBoundary] --> [BrowserControl]

[AvailabilityBoundary] --> [AvailabilityControl]

[PriceBoundary] --> [PriceControl]

[AccountControl] --> [AccountDAO]

[AvailabilityControl] --> [AvailabilityEntity]

[BrowserControl] --> [BrowserEntity]

[PriceControl] --> [PriceEntity]

[AccountDAO] --> [Postgres Database]

[BrowserEntity] --> [ExportUtils]

[PriceEntity] --> [ExportUtils]

[AvailabilityEntity] --> [ExportUtils]

@enduml

This diagram shows how user commands flow through the boundary and control layers before reaching the entity layer, where interaction with the database, browser, or export utilities happens.

1. **3. Deployment Diagram**
2. **Title: Deployment Diagram of the Discord Bot Automation Assistant**

The **Deployment Diagram** illustrates the physical deployment of the system components across different nodes.

* **User Node**: Represents the user who interacts with the bot through the Discord client.
* **Discord Bot Server**: This node contains the various control classes (AccountControl, BrowserControl, etc.) that handle user commands and execute business logic.
* **Database Server**: Contains the Postgres Database, which stores user accounts and other relevant data.
* **External Systems Node**: Represents websites that are interacted with by the bot, such as for price and availability monitoring.

plaintext

Copy code

@startuml

node "User" {

[DiscordClient]

}

node "Discord Bot Server" {

[AccountControl]

[BrowserControl]

[AvailabilityControl]

[PriceControl]

}

node "Database Server" {

[Postgres Database]

}

node "External Systems" {

[Websites]

}

[DiscordClient] --> [AccountControl]

[DiscordClient] --> [BrowserControl]

[DiscordClient] --> [AvailabilityControl]

[DiscordClient] --> [PriceControl]

[AccountControl] --> [Postgres Database]

[BrowserControl] --> [Websites]

[AvailabilityControl] --> [Websites]

[PriceControl] --> [Websites]

@enduml

This deployment architecture shows the flow of information from the user’s Discord client to the Discord bot server, which interacts with both the database and external systems for processing user requests.

1. **4. Activity Diagram**
2. **Title: Activity Diagram for "Check Price" Use Case**

The **Activity Diagram** provides a detailed flow for the "Check Price" use case. Here, the user sends a command (!get\_price), and the bot processes this command, retrieves the price, exports the data, and notifies the user of any price changes.

* **Command Handling**: The bot receives the command from the user and forwards it to the PriceBoundary, which then interacts with the PriceControl.
* **Price Retrieval**: PriceControl retrieves the price from the PriceEntity.
* **Data Export**: After retrieving the price, the data is exported to both Excel and HTML formats.
* **Notification**: If a price change is detected, notifications are sent to the user via Discord and email.

plaintext

Copy code

@startuml

start

:User sends !get\_price command;

:Discord Bot receives command;

if (Command is valid?) then (yes)

:Parse command;

:Forward command to PriceBoundary;

:PriceBoundary forwards to PriceControl;

:PriceControl requests price from PriceEntity;

:PriceEntity navigates to website and retrieves price;

if (Price found?) then (yes)

:PriceEntity returns price to PriceControl;

:PriceControl forwards price to PriceBoundary;

:PriceBoundary sends price to User via Discord;

:PriceControl exports data to Excel and HTML;

if (Price has changed?) then (yes)

:PriceControl sends email notification with price change;

:PriceControl sends notification to User via Discord;

else (no)

:No price change detected;

endif

else (no)

:PriceEntity returns error;

:PriceBoundary sends error to User;

endif

else (no)

:Send error message to User;

endif

stop

@enduml

1. **5. Code and Structure Validation**

Each part of this architecture is directly mapped to the actual implementation:

* **Boundary Layer**: The boundary classes (e.g., AccountBoundary, BrowserBoundary, PriceBoundary) receive and parse user commands before forwarding them to the control layer. The code shows how these boundary classes handle user input and pass the necessary data to the control classes​(project\_text).
* **Control Layer**: The control classes (e.g., AccountControl, PriceControl, etc.) manage the business logic. For instance, PriceControl handles fetching the price from PriceEntity, processes the data, and determines if a notification should be sent based on price changes​(project\_text)​(UML\_ComponentDiagram).
* **Entity Layer**: The entity classes (e.g., AccountDAO, BrowserEntity, PriceEntity) interact with external systems. AccountDAO handles database interactions, while PriceEntity retrieves prices from websites and exports the data to files​(project\_text).

The architecture is built to handle interactions between users and the bot, manage the flow of commands, process business logic, interact with external systems, and notify the user as needed.