```
--- main.py ---
import discord
from discord.ext import commands
from entity.BrowserEntity import BrowserEntity
from boundary.HelpBoundary import HelpBoundary
from boundary.AccountBoundary import AccountBoundary
from boundary.StopBoundary import StopBoundary  # Import StopBoundary
from boundary.LaunchBrowserBoundary import LaunchBrowserBoundary  # Import BrowserBoundary for browser launch
from boundary.CloseBrowserBoundary import CloseBrowserBoundary  # Import CloseBrowserBoundary for closing browser
from boundary.LoginBoundary import LoginBoundary
from boundary.NavigationBoundary import NavigationBoundary  # Import NavigationBoundary for navigating to a URL
from boundary.GetPriceBoundary import GetPriceBoundary
from boundary.MonitorPriceBoundary import MonitorPriceBoundary
from boundary.StopMonitoringPriceBoundary import StopMonitoringPriceBoundary
from control.MonitorPriceControl import MonitorPriceControl
from utils.Config import Config

# Set up the bot's intents
intents = discord.Intents.default()
intents.message_content = True  # Enable reading message content

# Initialize the bot with the correct command prefix and intents
class MyBot(commands.Bot):
    async def setup_hook(self):
        browser_entity = BrowserEntity()
         # Create a single instance of MonitorPriceControl
        monitor_price_control = MonitorPriceControl(browser_entity)
        await self.add_cog(HelpBoundary(self))  # Register HelpBoundary
        await self.add_cog(AccountBoundary(self))  # Register AccountBoundary
        await self.add_cog(StopBoundary(self))  # Register StopBoundary
        await self.add_cog(LaunchBrowserBoundary(self, browser_entity))
        await self.add_cog(NavigationBoundary(self, browser_entity))
        await self.add_cog(CloseBrowserBoundary(self, browser_entity))  # Register
CloseBrowserBoundary to close browser
        await self.add_cog(LoginBoundary(self, browser_entity))
        await self.add_cog(GetPriceBoundary(self, browser_entity))
        await self.add_cog(MonitorPriceBoundary(self, monitor_price_control))
        await self.add_cog(StopMonitoringPriceBoundary(self, monitor_price_control))

    async def on_ready(self):
        # Greet the user when the bot is online
        print(f"Logged in as {self.user}")
        channel = discord.utils.get(self.get_all_channels(), name="general")  # Adjust the channel
name
        if channel:
            await channel.send("Hi, I'm online! Type '!project_help' to see what I can do.")

    async def on_command_error(self, ctx, error):
        """Handle unrecognized commands."""
        if isinstance(error, commands.CommandNotFound):
            await ctx.send("Command not recognized. Type !project_help to see the list of commands.")
```

```
# Run the bot
if __name__ == "__main__":
    bot = MyBot(command_prefix="!", intents=intents)
    print("Bot is starting...")
    bot.run(Config.DISCORD_TOKEN)  # Run the bot with your token
```

--- Tests_URLs.txt ---

```
Working Commands: Test commands

!project_help

!stop_bot

!fetch_all_accounts

!add_account discordtestUser discordTestPass  discordtestWebsite

!fetch_account_by_website discordtestWebsite

!delete_account 4

!stop_bot

!launch_browser
```

!close_browser

!navigate_to_website https://www.google.com/

!login bestbuy

!get_price

!start_monitoring_price

!stop_monitoring_price

!check_availability https://www.opentable.com/r/bar-spero-washington/

!monitor_availability https://www.opentable.com/r/bar-spero-washington/

!monitor_availability https://www.opentable.com/r/bar-spero-washington/ "October 2"

!stop_monitoring_availability

****

Working on it:

!check_availability https://www.opentable.com/r/bar-spero-washington/ "August 22" "8:00 PM"

****

URLs to Test:

https://www.opentable.com/r/bar-spero-washington/

https://www.ebay.com/itm/314411766963?_trkparms=amclksrc%3DITM%26aid%3D777008%26algo%3DPERSONAL.TOPIC%26ao%3D1%26asc%3D20240603121456%26meid%3Da07931f944bc4a5b95376fe64d0ab035%26pid%3D102177%26rk%3D1%26rkt%3D1%26itm%3D314411766963%26pmt%3D1%26noa%3D1%26pg%3D4375194%26algv%3DNoSignalMostWatched%26brand%3DSimpliSafe&_trksid=p4375194.c102177.m166540&_trkparms=parentrq%3A71497a9c1910a8cd54f819a0ffff582e%7Cpageci%3A59d1354a-5f2b-11ef-9c4d-f2c982e61003%7Ciid%3A1%7Cvlpname%3Avlp_homepage

https://www.trendyol.com/puma/rebound-v6-low-p-736020132?boutiqueId=61&merchantId=184734&sav=true

Conclusion:

Control objects: Orchestrate the flow, decide which entities to use, and manage interactions between boundary and entity objects.

Entity objects: Contain the business logic (like logging in, updating prices, managing accounts).

--- AccountBoundary.py ---

```
from discord.ext import commands
from control.AccountControl import AccountControl


class AccountBoundary(commands.Cog):
    def __init__(self, bot):
        self.bot = bot
        self.control = AccountControl()

    @commands.command(name="fetch_all_accounts")
    async def fetch_all_accounts(self, ctx):
        """Fetch all accounts from the database."""
        await ctx.send("Command recognized, taking action: Fetching all accounts.")
        accounts = self.control.fetch_all_accounts()
        if accounts:
                account_list = "\n".join([f"ID: {acc[0]}, Username: {acc[1]}, Password: {acc[2]}, Website:
{acc[3]}" for acc in accounts])
            await ctx.send(f"Accounts:\n{account_list}")
        else:
            await ctx.send("No accounts found.")

    @commands.command(name="fetch_account_by_website")
    async def fetch_account_by_website(self, ctx, website: str):
        """Fetch an account by website."""
        await ctx.send(f"Command recognized, taking action: Fetching account for website {website}.")
        account = self.control.fetch_account_by_website(website)
        if account:
            await ctx.send(f"Account for {website}: Username: {account[0]}, Password: {account[1]}")
        else:
            await ctx.send(f"No account found for website {website}.")

    @commands.command(name="add_account")
    async def add_account(self, ctx, username: str, password: str, website: str):
        """Add a new account."""
        await ctx.send("Command recognized, taking action: Adding a new account.")
        result = self.control.add_account(username, password, website)
        if result:
            await ctx.send(f"Account for {website} added successfully.")
        else:
            await ctx.send(f"Failed to add account for {website}.")

    @commands.command(name="delete_account")
    async def delete_account(self, ctx, account_id: int):
        """Delete an account by ID."""
        await ctx.send(f"Command recognized, taking action: Deleting account with ID {account_id}.")
        result = self.control.delete_account(account_id)
        if result:
            await ctx.send(f"Account with ID {account_id} deleted successfully.")
        else:
            await ctx.send(f"Failed to delete account with ID {account_id}.")



--- CloseBrowserBoundary.py ---
from discord.ext import commands
from control.CloseBrowserControl import CloseBrowserControl
from entity.BrowserEntity import BrowserEntity


class CloseBrowserBoundary(commands.Cog):
    def __init__(self, bot, browser_entity):
        self.bot = bot
         self.close_browser_control = CloseBrowserControl(browser_entity)  # Pass the browser_entity
to the control

    @commands.command(name='close_browser')
    async def close_browser(self, ctx):
        await ctx.send("Command recognized, taking action to close the browser.")
        result = self.close_browser_control.close_browser()
        await ctx.send(result)



--- GetPriceBoundary.py ---
from discord.ext import commands
from control.GetPriceControl import GetPriceControl


class GetPriceBoundary(commands.Cog):
    def __init__(self, bot, browser_entity):
        self.bot = bot
        self.price_control = GetPriceControl(browser_entity)

    @commands.command(name='get_price')
    async def get_price(self, ctx, url: str=None):
        """Command to get the price from the given URL."""
        await ctx.send("Command recognized, taking action.")
        response = await self.price_control.get_price(url)
        await ctx.send(response)



--- HelpBoundary.py ---
from discord.ext import commands
from control.HelpControl import HelpControl


class HelpBoundary(commands.Cog):  # Cog to register with the bot
    def __init__(self, bot):
        self.bot = bot
        self.control = HelpControl()  # Initialize control object

    @commands.command(name="project_help")
    async def project_help(self, ctx):
        """Send a message with all the available commands."""
        await ctx.send("Command recognized, taking action.")  # Acknowledge the command
        help_message = self.control.get_help_message()  # Get help message from control
        await ctx.send(help_message)  # Send help message to Discord



--- LaunchBrowserBoundary.py ---
from discord.ext import commands
from control.LaunchBrowserControl import LaunchBrowserControl
```

```
class LaunchBrowserBoundary(commands.Cog):
    def __init__(self, bot, browser_entity):
        self.bot = bot
        self.launch_browser_control = LaunchBrowserControl(browser_entity)  # Pass the browser_entity to the control

    @commands.command(name='launch_browser')
    async def launch_browser(self, ctx):
        await ctx.send("Command recognized, taking action.")
        result = self.launch_browser_control.launch_browser()
        await ctx.send(result)



--- LoginBoundary.py ---
from discord.ext import commands
from control.LoginControl import LoginControl

class LoginBoundary(commands.Cog):
    def __init__(self, bot, browser_entity):
        self.bot = bot
        self.login_control = LoginControl(browser_entity)  # Pass browser_entity to control

    @commands.command(name='login')
    async def login(self, ctx, site: str):
        await ctx.send("Command recognized, taking action.")
        result = await self.login_control.login(site)
        await ctx.send(result)
```

```
--- MonitorPriceBoundary.py ---
from discord.ext import commands
from control.MonitorPriceControl import MonitorPriceControl


class MonitorPriceBoundary(commands.Cog):
    def __init__(self, bot, monitor_price_control):
        self.bot = bot
        self.monitor_price_control = monitor_price_control  # Use shared instance


    @commands.command(name='start_monitoring_price')
    async def start_monitoring_price(self, ctx, url: str = None, frequency: int = 20):
          await ctx.send(f"Command recognized, starting price monitoring at {url} every {frequency}
second(s).")
        response = await self.monitor_price_control.start_monitoring_price(ctx, url, frequency)
        await ctx.send(response)



--- NavigationBoundary.py ---
import discord
from discord.ext import commands
from control.NavigationControl import NavigationControl


class NavigationBoundary(commands.Cog):
    def __init__(self, bot, browser_entity):
        self.bot = bot
        self.navigation_control = NavigationControl(browser_entity)

    @commands.command(name='navigate_to_website')
    async def navigate_to_website(self, ctx, url: str = None):
        await ctx.send("Command recognized, taking action.")
        result = self.navigation_control.navigate_to_website(url)
        await ctx.send(result)



--- StopBoundary.py ---
from discord.ext import commands
from control.StopControl import StopControl

class StopBoundary(commands.Cog):
    def __init__(self, bot):
        self.bot = bot
        self.control = StopControl()

    @commands.command(name="stop_bot")
    async def stop_bot(self, ctx):
        """Shut down the bot."""
        await ctx.send("Command recognized, taking action: Shutting down the bot.")
        await self.control.stop_bot(ctx, self.bot)  # Call the control's method to stop the bot



--- StopMonitoringPriceBoundary.py ---
from discord.ext import commands
from control.MonitorPriceControl import MonitorPriceControl


class StopMonitoringPriceBoundary(commands.Cog):
    def __init__(self, bot, monitor_price_control):
        self.bot = bot
        self.monitor_price_control = monitor_price_control  # Use shared instance

    @commands.command(name='stop_monitoring_price')
    async def StopMonitoringPrice(self, ctx):
        """Command to stop monitoring the price."""
        await ctx.send("Command recognized, taking action.")
        response = self.monitor_price_control.stop_monitoring()
        await ctx.send(response)



--- __init__.py ---
#empty init file


--- AccountControl.py ---
from DataObjects.AccountDAO import AccountDAO
from DataObjects.AccountDTO import AccountDTO  # Assuming the DTO file is in the dto folder


class AccountControl:
    def __init__(self):
        self.account_dao = AccountDAO()


    def add_account(self, username: str, password: str, website: str):
```

```
    """Add a new account to the database using DTO."""
    self.account_dao.connect()  # Establish database connection
    account_dto = AccountDTO(username, password, website)
    result = self.account_dao.add_account(account_dto)
    self.account_dao.close()  # Close the connection
    return result



def delete_account(self, account_id: int):
    """Delete an account by ID."""
    self.account_dao.connect()  # Establish database connection
    result = self.account_dao.delete_account(account_id)
    self.account_dao.reset_id_sequence()
    self.account_dao.close()  # Close the connection
    return result



def fetch_all_accounts(self):
    """Fetch all accounts using the DAO."""
    self.account_dao.connect()  # Establish database connection
    accounts = self.account_dao.fetch_all_accounts()  # Fetch accounts from DAO
    self.account_dao.close()  # Close the connection
    return accounts if accounts else None


def fetch_account_by_website(self, website: str):
    """Fetch an account by website."""
    self.account_dao.connect()  # Establish database connection
```

```
        account = self.account_dao.fetch_account_by_website(website)

        self.account_dao.close()  # Close the connection

        return account if account else None




--- CloseBrowserControl.py ---

class CloseBrowserControl:

    def __init__(self, browser_entity):

        self.browser_entity = browser_entity


    def close_browser(self):

        return self.browser_entity.close_browser()




--- GetPriceControl.py ---

from entity.PriceEntity import PriceEntity

from utils.css_selectors import Selectors


class GetPriceControl:

    def __init__(self, browser_entity):

        self.price_entity = PriceEntity(browser_entity)


    async def get_price(self, url: str):

        # Fetch the url using the correct CSS selector

        if not url:

            selectors = Selectors.get_selectors_for_url("bestbuy")

            url = selectors.get('priceUrl')  # Get the price URL
            if not url:
                return "No URL provided, and default URL for BestBuy could not be found."
            print("URL not provided, default URL for BestBuy is: " + url)

        # Step 3: Call the entity to get the price
        price = self.price_entity.get_price_from_page(url)
        return price
```

```
--- HelpControl.py ---
class HelpControl:
    def get_help_message(self):
        """Returns a list of available bot commands."""
        return (
            "Here are the available commands:\n"
            "!project_help - Get help on available commands.\n"
            "!login 'website' - Log in to a website.\n"
            "!launch_browser - Launch the browser.\n"
            "!close_browser - Close the browser.\n"
            "!navigate_to_website - Navigate to a website.\n"
            "!get_price - Check the price of a product.\n"
            "!monitor_price - monitor a product price.\n"
            "!stop_monitoring - Stop monitoring a product.\n"
            "!check_availability - Check the availability in a restaurant.\n"
            "!monitor_availability - Monitor the availability in a restaurant.\n"
            "!stop_monitoring_availability - Stop monitoring availibility.\n"
            "!stop_bot - Stop the bot.\n"

        )



# "##!receive_notifications - Receive notifications for price changes.\n"
# "##!extract_data - Export data to Excel or HTML.\n"

--- LaunchBrowserControl.py ---
class LaunchBrowserControl:
    def __init__(self, browser_entity):
        self.browser_entity = browser_entity

    def launch_browser(self):
        return self.browser_entity.launch_browser()



--- LoginControl.py ---
from entity.BrowserEntity import BrowserEntity
from control.AccountControl import AccountControl
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from utils.css_selectors import Selectors
import asyncio

class LoginControl:
    def __init__(self, browser_entity):
        self.browser_entity = browser_entity  # Manages browser state
        self.account_control  = AccountControl()  # Manages account data


    async def login(self, site: str):
        # Step 1: Fetch account credentials from the entity object
        account_info = self.account_control.fetch_account_by_website(site)
        if not account_info:
            return f"No account found for {site}"

        # account_info is a tuple (username, password), so access it by index
        username, password = account_info[0], account_info[1]
        print(f"Username: {username}, Password: {password}")

        # Step 3: Get the URL from the CSS selectors
        url = Selectors.get_selectors_for_url(site).get('url')
        print(url)
        if not url:
            return f"URL for {site} not found."

        # Step 4: Navigate to the URL and perform login (handled by the entity object)
        result = await self.browser_entity.perform_login(url, username, password)
        return result

--- MonitorPriceControl.py ---
import asyncio
```

```
from datetime import datetime
from entity.PriceEntity import PriceEntity
from utils.css_selectors import Selectors


class MonitorPriceControl:
    """MonitorPriceControl handles the business logic of monitoring the price over time
    and instructs PriceEntity to fetch prices and export data."""

    def __init__(self, browser_entity):
        self.price_entity = PriceEntity(browser_entity)  # Initialize PriceEntity for data fetching and
export
        self.is_monitoring = False  # Control flag for monitoring state
        self.results = []  # List to store results during monitoring

    async def start_monitoring_price(self, ctx, url: str = None, frequency=20):
        """Start monitoring the price at a given interval and provide updates to the user via Discord.

        ctx: Context from Discord command.
        url: URL of the product page to monitor.
        frequency: Time interval (in seconds) between price checks.
        """
        if self.is_monitoring:
            return "Already monitoring prices."
        self.is_monitoring = True  # Set monitoring state to true
        previous_price = None  # Track the last price fetched

        try:
```

```
    while self.is_monitoring:
        # Fetch the current price from PriceEntity
        if not url:
            selectors = Selectors.get_selectors_for_url("bestbuy")
            url = selectors.get('priceUrl')  # Get the price URL
            if not url:
                return "No URL provided, and default URL for BestBuy could not be found."
            print("URL not provided, default URL for BestBuy is: " + url)
        current_price = self.price_entity.get_price_from_page(url)

        # Determine price changes and prepare the result
        result = ""
        if current_price:
            if previous_price is None:
                result = f"Starting price monitoring. Current price: {current_price}"
            elif current_price > previous_price:
                result = f"Price went up! Current price: {current_price} (Previous: {previous_price})"
            elif current_price < previous_price:
                result = f"Price went down! Current price: {current_price} (Previous:
{previous_price})"
            else:
                result = f"Price remains the same: {current_price}"
            previous_price = current_price
        else:
            result = "Failed to retrieve the price."

        # Add the result to the results list
```

```
            self.results.append(result)

            # Create a DTO (Data Transfer Object) to organize the data for export
            data_dto = {
                "command": "start_monitoring_price",  # Command executed
                "url": url,  # URL of the product being monitored
                "result": result,  # Result of the price check
                "entered_date": datetime.now().strftime('%Y-%m-%d'),  # Current date
                "entered_time": datetime.now().strftime('%H:%M:%S')  # Current time
            }

            # Pass the DTO to PriceEntity to handle export to Excel and HTML
            self.price_entity.export_data(data_dto)

            await asyncio.sleep(frequency)  # Wait for the next check based on frequency

        except Exception as e:
            self.results.append(f"Failed to monitor price: {str(e)}")

    def stop_monitoring(self):
        """Stop the price monitoring loop."""
        self.is_monitoring = False  # Set monitoring state to false
        # Return the full list of results gathered during monitoring
        return self.results if self.results else ["No data collected."]
```

--- NavigationControl.py ---

```
from entity.BrowserEntity import BrowserEntity
from utils.css_selectors import Selectors


class NavigationControl:
    def __init__(self, browser_entity):
        self.browser_entity = browser_entity


    def navigate_to_website(self, url: str = None):
        if not url:
            selectors = Selectors.get_selectors_for_url("google")
            url = selectors.get('url')
            if not url:
                return "No URL provided, and default URL for google could not be found."
            print("URL not provided, default URL for Google is: " + url)
        return self.browser_entity.navigate_to_url(url)



--- StopControl.py ---
import discord


class StopControl:
    async def stop_bot(self, ctx, bot):
        """Stop the bot gracefully."""
        await ctx.send("The bot is shutting down...")
        await bot.close()  # Close the bot
```

```
--- __init__.py ---

#empty init file


--- AccountDAO.py ---

import psycopg2

from utils.Config import Config

from DataObjects.AccountDTO import AccountDTO


class AccountDAO:

    def __init__(self):

        self.dbname = "postgres"

        self.user = "postgres"

        self.host = "localhost"

        self.port = "5432"

        self.password = Config.DATABASE_PASSWORD


    def connect(self):

        """Establish a database connection."""

        try:

            self.connection = psycopg2.connect(

                dbname=self.dbname,

                user=self.user,

                password=self.password,

                host=self.host,

                port=self.port

            )

            self.cursor = self.connection.cursor()
            print("Database Connection Established.")
        except Exception as error:
            print(f"Error connecting to the database: {error}")
            self.connection = None
            self.cursor = None

    def add_account(self, account_dto: AccountDTO):
        """Add a new account to the database using DTO."""
        try:
            query = "INSERT INTO accounts (username, password, website) VALUES (%s, %s, %s)"
            values = (account_dto.username, account_dto.password, account_dto.website)
            self.cursor.execute(query, values)
            self.connection.commit()
            print(f"Account {account_dto.username} added successfully.")
            return True
        except Exception as error:
            print(f"Error inserting account: {error}")
            return False



    def fetch_account_by_website(self, website):
        """Fetch account credentials for a specific website."""
        try:
                query = "SELECT username, password FROM accounts WHERE LOWER(website) =
LOWER(%s)"
            self.cursor.execute(query, (website,))
            return self.cursor.fetchone()
        except Exception as error:
            print(f"Error fetching account for website {website}: {error}")
            return None

    def fetch_all_accounts(self):
        """Fetch all accounts from the database."""
        try:
            query = "SELECT id, username, password, website FROM accounts"
            self.cursor.execute(query)
            return self.cursor.fetchall()
        except Exception as error:
            print(f"Error fetching accounts: {error}")
            return []



    def delete_account(self, account_id):
        """Delete an account by its ID."""
        try:
            self.cursor.execute("DELETE FROM accounts WHERE id = %s", (account_id,))
            self.connection.commit()
            if self.cursor.rowcount > 0:  # Check if any rows were affected
                print(f"Account with ID {account_id} deleted successfully.")
                return True
            else:
                print(f"No account found with ID {account_id}.")
                return False
        except Exception as error:
            print(f"Error deleting account: {error}")
            return False

    def reset_id_sequence(self):
        """Reset the ID sequence to the maximum ID."""
        try:
            reset_query = "SELECT setval('accounts_id_seq', (SELECT MAX(id) FROM accounts))"
            self.cursor.execute(reset_query)
            self.connection.commit()
            print("ID sequence reset successfully.")
        except Exception as error:
            print(f"Error resetting ID sequence: {error}")



    def close(self):
        """Close the database connection."""
        if self.cursor:
            self.cursor.close()
        if self.connection:
            self.connection.close()
            print("Database connection closed.")



--- AccountDTO.py ---
# dto/DataExportDTO.py

class AccountDTO:
    def __init__(self, username, password, website):
        self.username = username
        self.password = password
        self.website = website



--- BrowserEntity.py ---
import asyncio
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from utils.css_selectors import Selectors



class BrowserEntity:
    def __init__(self):
        self.driver = None
        self.browser_open = False



    def set_browser_open(self, is_open: bool):
        self.browser_open = is_open



    def is_browser_open(self) -> bool:
```

```
    return self.browser_open



def launch_browser(self):
    if not self.browser_open:
        options = webdriver.ChromeOptions()
        options.add_argument("--remote-debugging-port=9222")
        options.add_experimental_option("excludeSwitches", ["enable-automation"])
        options.add_experimental_option('useAutomationExtension', False)
        options.add_argument("--start-maximized")
        options.add_argument("--disable-notifications")
        options.add_argument("--disable-popup-blocking")
        options.add_argument("--disable-infobars")
        options.add_argument("--disable-extensions")
        options.add_argument("--disable-webgl")
        options.add_argument("--disable-webrtc")
        options.add_argument("--disable-rtc-smoothing")

        self.driver = webdriver.Chrome(service=Service(), options=options)
        self.browser_open = True
        return "Browser launched."
    else:
        return "Browser is already running."



def close_browser(self):
    if self.browser_open and self.driver:
        self.driver.quit()
        self.browser_open = False
        return "Browser closed."
    else:
        return "No browser is currently open."


def navigate_to_url(self, url):
        # Ensure the browser is launched before navigating
        if not self.is_browser_open():
            launch_message = self.launch_browser()
            print(launch_message)

        # Navigate to the URL if browser is open
        if self.driver:
            self.driver.get(url)
            return f"Navigated to {url}"
        else:
            return "Failed to open browser."


async def perform_login(self, url, username, password):
    # Navigate to the website
    self.navigate_to_url(url)
    await asyncio.sleep(3)

    # Enter the username
```

```
        email_field = self.driver.find_element(By.CSS_SELECTOR,
Selectors.get_selectors_for_url(url)['email_field'])
        email_field.send_keys(username)
        await asyncio.sleep(3)

        # Enter the password
        password_field = self.driver.find_element(By.CSS_SELECTOR,
Selectors.get_selectors_for_url(url)['password_field'])
        password_field.send_keys(password)
        await asyncio.sleep(3)

        # Click the login button
        sign_in_button = self.driver.find_element(By.CSS_SELECTOR,
Selectors.get_selectors_for_url(url)['SignIn_button'])
        sign_in_button.click()
        await asyncio.sleep(5)

        # Wait for the homepage to load
        try:
            WebDriverWait(self.driver,
30).until(EC.presence_of_element_located((By.CSS_SELECTOR,
Selectors.get_selectors_for_url(url)['homePage'])))
            return f"Logged in to {url} successfully with username: {username}"
        except Exception as e:
            return f"Failed to log in: {str(e)}"
```

```
--- PriceEntity.py ---

from selenium.webdriver.common.by import By

from utils.exportUtils import ExportUtils  # Import ExportUtils for handling data export

from utils.css_selectors import Selectors  # Import selectors to get CSS selectors for the browser

from datetime import datetime  # Import for date and time


class PriceEntity:

    """PriceEntity is responsible for interacting with the system (browser) to fetch prices

    and handle the exporting of data to Excel and HTML."""


    def __init__(self, browser_entity):

        self.browser_entity = browser_entity  # Browser entity to handle web page interaction



    def get_price_from_page(self, url: str):

        # Navigate to the URL using BrowserEntity

        self.browser_entity.navigate_to_url(url)

        selectors = Selectors.get_selectors_for_url(url)

        try:

            # Find the price element on the page using the selector

                        price_element  =  self.browser_entity.driver.find_element(By.CSS_SELECTOR,
selectors['price'])

            return price_element.text  # Return the price text

        except Exception as e:

            return f"Error fetching price: {str(e)}"
    def export_data(self, dto):
        """Export price data to both Excel and HTML using ExportUtils.

          dto: This is a Data Transfer Object (DTO) that contains the command, URL, result, date, and
time.
        """
        # Extract the data from the DTO
        command = dto.get('command')
        url = dto.get('url')
        result = dto.get('result')
        entered_date = dto.get('entered_date')  # Optional, could be None
        entered_time = dto.get('entered_time')  # Optional, could be None

        # Call the Excel export method from ExportUtils
        ExportUtils.log_to_excel(
            command=command,
            url=url,
            result=result,
            entered_date=entered_date,  # Pass the optional entered_date
            entered_time=entered_time   # Pass the optional entered_time
        )

        # Call the HTML export method from ExportUtils
        ExportUtils.export_to_html(
            command=command,
            url=url,
            result=result,
            entered_date=entered_date,  # Pass the optional entered_date
            entered_time=entered_time   # Pass the optional entered_time
        )
```

```
--- __init__.py ---
#empty init file
```

```
--- project_structure.py ---
import os

def list_files_and_folders(directory, output_file):
    with open(output_file, 'w') as f:
        for root, dirs, files in os.walk(directory):
            # Ignore .git and __pycache__ folders
            dirs[:] = [d for d in dirs if d not in ['.git', '__pycache__']]

            f.write(f"Directory: {root}\n")
            for dir_name in dirs:
                f.write(f"  Folder: {dir_name}\n")
            for file_name in files:
                f.write(f"  File: {file_name}\n")

# Update the directory path to your project folder
project_directory = "D:/HARRISBURG/Harrisburg Master's Fifth Term Late Summer/CISC 699/DiscordBotProject_CISC699"
```

```
output_file = os.path.join(project_directory, "other/project_structure.txt")


# Call the function to list files and save output to .txt
list_files_and_folders(project_directory, output_file)


print(f"File structure saved to {output_file}")




--- project_text.py ---
import os
from fpdf import FPDF


# Directory where the project files are located
directory = r"D:\HARRISBURG\Harrisburg Master's Fifth Term Late Summer\CISC
699\DiscordBotProject_CISC699"
output_pdf_path = os.path.join(directory, "other/project_text.pdf")


# Function to retrieve all text from files, ignoring .git and __pycache__ directories
def extract_project_text(directory):
    project_text = ""
    for root, dirs, files in os.walk(directory):
        # Ignore .git and __pycache__ directories
        dirs[:] = [d for d in dirs if d not in ['.git', '__pycache__']]


        for file in files:
            if file.endswith('.py') or file.endswith('.txt') or file.endswith('.md'):  # Only considering relevant
file types
```

```
                file_path = os.path.join(root, file)
                try:
                    with open(file_path, 'r', encoding='utf-8') as f:
                        project_text += f"--- {file} ---\n"
                        project_text += f.read() + "\n\n"
                except Exception as e:
                    print(f"Could not read file {file_path}: {e}")

    return project_text


# Function to generate a PDF with the extracted text
def create_pdf(text, output_path):
    pdf = FPDF()
    pdf.set_auto_page_break(auto=True, margin=15)
    pdf.add_page()
    pdf.set_font("Arial", size=12)

    # Ensure proper encoding handling
    for line in text.split("\n"):
        # Convert the text to UTF-8 and handle unsupported characters
        try:
            pdf.multi_cell(0, 10, line.encode('latin1', 'replace').decode('latin1'))
        except UnicodeEncodeError:
            # Handle any other encoding issues
            pdf.multi_cell(0, 10, line.encode('ascii', 'replace').decode('ascii'))

    pdf.output(output_path)
```

```
# Extract project text and create the PDF

project_text = extract_project_text(directory)

if project_text:

    create_pdf(project_text, output_pdf_path)

    output_pdf_path

    print("PDF file created with all project's as text at: " + output_pdf_path)

else:

    "No project text found."





--- test_addAccount.py ---

import sys, os

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from control.AccountControl import AccountControl


def test_add_account(username, password, website):

    account_control = AccountControl()


    # Adding a new account

    result = account_control.add_account(username, password, website)

    if result:

        print(f"Account for {website} added successfully.")
    else:
        print(f"Failed to add account for {website}.")


if __name__ == "__main__":
    test_add_account("newUser", "newPassword123", "newWebsite")  # Change values to test
```

```
--- test_deleteAccount.py ---
import sys, os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from control.AccountControl import AccountControl

def test_delete_account(account_id):
    account_control = AccountControl()

    result = account_control.delete_account(account_id)
    if result:
        print(f"Account with ID {account_id} deleted successfully.")
    else:
        print(f"Failed to delete account with ID {account_id}.")

if __name__ == "__main__":
    test_delete_account(4)  # You can change the account ID here for testing
```

```
--- test_excel_creation.py ---
```

```
import sys, os

from datetime import datetime


sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from utils.exportUtils import ExportUtils

from DataObjects.DataExportDTO import DataExportDTO  # Importing the DTO


def test_excel_creation():

    # Mock data that simulates the data received from a website

    mock_command = "MOCK_check_availability"

    mock_url = "MOCKURL_https://www.opentable.com/r/bar-spero-washington/"

    mock_result = "MOCK_No availability for the selected date."

    mock_entered_date = datetime.now().strftime('%Y-%m-%d')

    mock_entered_time = datetime.now().strftime('%H:%M:%S')


    # Create DTO object

    data_dto = DataExportDTO(

        command=mock_command,

        url=mock_url,

        result=mock_result,

        entered_date=mock_entered_date,

        entered_time=mock_entered_time

    )


    # Validate the DTO

    try:

        data_dto.validate()
    except ValueError as ve:
        print(f"Validation Error: {ve}")
        return

    # Log data to Excel using the DTO
    result_message = ExportUtils.log_to_excel(
        command=data_dto.command,
        url=data_dto.url,
        result=data_dto.result,
        entered_date=data_dto.entered_date,
        entered_time=data_dto.entered_time
    )

    # Output the result of the Excel file creation
    print(result_message)

if __name__ == "__main__":
    test_excel_creation()



--- test_fetchAccounts.py ---
import sys, os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from control.AccountControl import AccountControl  # Import the control layer directly

def test_fetch_accounts():
    account_control = AccountControl()  # Use AccountControl instead of AccountBoundary

    # Fetching all accounts
    accounts = account_control.fetch_all_accounts()

    if accounts:
        for account in accounts:
            print(f"ID: {account[0]}, Username: {account[1]}, Password: {account[2]}, Website: {account[3]}")
    else:
        print("No accounts found.")

def test_fetch_account_by_website(website):
    account_control = AccountControl()  # Use AccountControl instead of AccountBoundary

    # Fetch the account by website directly
    account = account_control.fetch_account_by_website(website)

    if account:
        username, password = account  # Unpack the returned tuple
        print(f"Website: {website}, Username: {username}, Password: {password}")
    else:
        print(f"No account found for website: {website}")

if __name__ == "__main__":
    test_fetch_accounts()  # Test fetching all accounts
    test_fetch_account_by_website("ebay")  # Test fetching account for a specific website
```

```
--- test_html_creation.py ---

import sys, os

from datetime import datetime

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))

from DataObjects.DataExportDTO import DataExportDTO  # Importing the DTO

from utils.exportUtils import ExportUtils


def test_html_creation():

    # Mock data that simulates the data received from a website

    mock_command = "MOCK_check_availability"

    mock_url = "MOCK_https://www.opentable.com/r/bar-spero-washington/"

    mock_result = "No availability for the selected date."


    # Get the current date and time

    mock_entered_date = datetime.now().strftime('%Y-%m-%d')

    mock_entered_time = datetime.now().strftime('%H:%M:%S')


    # Create DTO object

    data_dto = DataExportDTO(

        command=mock_command,

        url=mock_url,

        result=mock_result,

        entered_date=mock_entered_date,

        entered_time=mock_entered_time

    )
    # Validate the DTO
    try:
        data_dto.validate()
    except ValueError as ve:
        print(f"Validation Error: {ve}")
        return

    # Prepare the data for HTML export
    mock_data = [data_dto.to_dict()]

    # Export data to HTML using the DTO
    result_message = ExportUtils.export_to_html(
        data=mock_data,
        command_name=data_dto.command
    )

    # Output the result of the HTML file creation
    print(result_message)

if __name__ == "__main__":
    test_html_creation()
```

```
--- __init__.py ---
#empty init file
```

```
--- Config.py ---
class Config:
    DISCORD_TOKEN = 'MTI2OTM4MTE4OTA1NjMzNTk3Mw.Gihcfw.nrq0x-JiL65P0LIQTO-rTyyXq0qC-2PSSBuXr8'
    CHANNEL_ID = 1269383349278081054
    DATABASE_PASSWORD = 'postgres'


--- css_selectors.py ---
class Selectors:
    SELECTORS = {
        "google": {
            "url": "https://www.google.com/"
        },
        "ebay": {
            "url": "https://signin.ebay.com/signin/",
            "email_field": "#userid",
            "continue_button": "[data-testid*='signin-continue-btn']",
            "password_field": "#pass",
            "login_button": "#sgnBt",
            "price": ".x-price-primary span"  # CSS selector for Ebay price
        },
        "bestbuy": {
            "priceUrl": "https://www.bestbuy.com/site/microsoft-xbox-wireless-controller-for-xbox-series-x-xbox-series-s-xbox-one-windows-devices-sky-cipher-special-edition/6584960.p?skuId=6584960",
            "url": "https://www.bestbuy.com/signin/",
            "email_field": "#fld-e",
            #"continue_button": ".cia-form__controls  button",
            "password_field": "#fld-p1",
            "SignIn_button": ".cia-form__controls  button",
            "price": "[data-testid='customer-price'] span",  # CSS selector for BestBuy price
            "homePage": ".v-p-right-xxs.line-clamp"
        },
        "opentable": {
            "url": "https://www.opentable.com/",
            "date_field": "#restProfileSideBarDtpDayPicker-label",
            "time_field": "#restProfileSideBartimePickerDtpPicker",
            "select_date": "#restProfileSideBarDtpDayPicker-wrapper", # button[aria-label*="{}"]
            "select_time": "h3[data-test='select-time-header']",
            "no_availability": "div._8ye6OVzeOuU- span",
            "find_table_button": ".find-table-button",  # Example selector for the Find Table button
            "availability_result": ".availability-result",  # Example selector for availability results
               "show_next_available_button": "button[data-test='multi-day-availability-button']",   # Show
next available button
            "available_dates": "ul[data-test='time-slots'] > li",  # Available dates and times

        }
    }

    @staticmethod
    def get_selectors_for_url(url):
        for keyword, selectors in Selectors.SELECTORS.items():
            if keyword in url.lower():
                return selectors
        return None  # Return None if no matching selectors are found


--- exportUtils.py ---
import os
import pandas as pd
from datetime import datetime


class ExportUtils:

    @staticmethod
    def log_to_excel(command, url, result, entered_date=None, entered_time=None):
        # Determine the file path for the Excel file
        file_name = f"{command}.xlsx"
        file_path = os.path.join("ExportedFiles", "excelFiles", file_name)

        # Ensure directory exists
        os.makedirs(os.path.dirname(file_path), exist_ok=True)

        # Timestamp for current run
        timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

        # If date/time not entered, use current timestamp
        entered_date = entered_date or datetime.now().strftime('%Y-%m-%d')
        entered_time = entered_time or datetime.now().strftime('%H:%M:%S')

        # Check if the file exists and create the structure if it doesn't
        if not os.path.exists(file_path):
                df = pd.DataFrame(columns=["Timestamp", "Command", "URL", "Result", "Entered Date",
"Entered Time"])
            df.to_excel(file_path, index=False)

        # Load existing data from the Excel file
        df = pd.read_excel(file_path)

        # Append the new row
        new_row = {
            "Timestamp": timestamp,
            "Command": command,
            "URL": url,
            "Result": result,
            "Entered Date": entered_date,
            "Entered Time": entered_time
        }

        # Add the new row to the existing data and save it back to Excel
        df = pd.concat([df, pd.DataFrame([new_row])], ignore_index=True)
        df.to_excel(file_path, index=False)

        return f"Data saved to Excel file at {file_path}."

    @staticmethod
    def export_to_html(command, url, result, entered_date=None, entered_time=None):
        """Export data to HTML format with the same structure as Excel."""
```

```
# Define file path for HTML
file_name = f"{command}.html"
file_path = os.path.join("ExportedFiles", "htmlFiles", file_name)


# Ensure directory exists
os.makedirs(os.path.dirname(file_path), exist_ok=True)


# Timestamp for current run
timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')


# If date/time not entered, use current timestamp
entered_date = entered_date or datetime.now().strftime('%Y-%m-%d')
entered_time = entered_time or datetime.now().strftime('%H:%M:%S')


# Data row to insert
new_row = {
    "Timestamp": timestamp,
    "Command": command,
    "URL": url,
    "Result": result,
    "Entered Date": entered_date,
    "Entered Time": entered_time
}


# Check if the HTML file exists and append rows
if os.path.exists(file_path):
    # Open the file and append rows
```

```
        with open(file_path, "r+", encoding="utf-8") as file:
            content = file.read()
            # Look for the closing </table> tag and append new rows before it
            if "</table>" in content:
                new_row_html = f"<tr><td>{new_row['Timestamp']}</td><td>{new_row['Command']}</td><td>{new_row['URL']}</td><td>{new_row['Result']}</td><td>{new_row['Entered Date']}</td><td>{new_row['Entered Time']}</td></tr>\n"
                content = content.replace("</table>", new_row_html + "</table>")
                file.seek(0)  # Move pointer to the start
                file.write(content)
                file.truncate()  # Truncate any remaining content
                file.flush()  # Flush the buffer to ensure it's written
    else:
        # If the file doesn't exist, create a new one with table headers
        with open(file_path, "w", encoding="utf-8") as file:
            html_content = "<html><head><title>Command Data</title></head><body>"
            html_content += f"<h1>Results for {command}</h1><table border='1'>"
            html_content += "<tr><th>Timestamp</th><th>Command</th><th>URL</th><th>Result</th><th>Entered Date</th><th>Entered Time</th></tr>"
            html_content += f"<tr><td>{new_row['Timestamp']}</td><td>{new_row['Command']}</td><td>{new_row['URL']}</td><td>{new_row['Result']}</td><td>{new_row['Entered Date']}</td><td>{new_row['Entered Time']}</td></tr>\n"
            html_content += "</table></body></html>"
            file.write(html_content)
        file.flush()  # Ensure content is written to disk


    return f"HTML file saved and updated at {file_path}."
```