--- main.py ---

```
from utils.MyBot import MyBot
from utils.Config import Config
import discord

intents = discord.Intents.default()
intents.message_content = True  # Enable reading message content

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

--- AccountBoundary.py ---

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

class AccountBoundary(commands.Cog):
    def __init__(self):
        self.control = AccountControl()  # Initialize control object

    @commands.command(name="fetch_all_accounts")
    async def fetch_all_accounts(self, ctx):
        await ctx.send("Command recognized, passing data to control.")
```

```
    # Pass the command to the control object
    commandToPass = "fetch_all_accounts"
    result = self.control.receive_command(commandToPass)

    # Send the result (prepared by control) back to the user
    await ctx.send(result)


@commands.command(name="fetch_account_by_website")
async def fetch_account_by_website(self, ctx, website: str):
    await ctx.send(f"Command recognized, passing data to control for website {website}.")

    # Pass the command and website to control
    commandToPass = "fetch_account_by_website"
    result = self.control.receive_command(commandToPass, website)

    # Send the result (prepared by control) back to the user
    await ctx.send(result)


@commands.command(name="add_account")
async def add_account(self, ctx, username: str, password: str, website: str):
    await ctx.send("Command recognized, passing data to control.")

    # Pass the command and account details to control
    commandToPass = "add_account"
    result = self.control.receive_command(commandToPass, username, password, website)
```

```
        # Send the result (prepared by control) back to the user
        await ctx.send(result)


    @commands.command(name="delete_account")
    async def delete_account(self, ctx, account_id: int):
            await ctx.send(f"Command recognized, passing data to control to delete account with ID {account_id}.")

        # Pass the command and account ID to control
        commandToPass = "delete_account"
        result = self.control.receive_command(commandToPass, account_id)

        # Send the result (prepared by control) back to the user
        await ctx.send(result)


--- AvailabilityBoundary.py ---
from discord.ext import commands
from control.AvailabilityControl import AvailabilityControl

class AvailabilityBoundary(commands.Cog):
    def __init__(self):
        # Initialize control objects directly
        self.availability_control = AvailabilityControl()
```

```
    @commands.command(name="check_availability")
    async def check_availability(self, ctx, url: str, date_str=None):
        await ctx.send("Command recognized, passing data to control.")

        # Pass the command and data to the control layer using receive_command
        command_to_pass = "check_availability"
        result = await self.availability_control.receive_command(command_to_pass, url, date_str)

        # Send the result back to the user
        await ctx.send(result)

    @commands.command(name="monitor_availability")
    async def monitor_availability(self, ctx, url: str, date_str=None, frequency: int = 15):
        await ctx.send("Command recognized, passing data to control.")

        # Pass the command and data to the control layer using receive_command
        command_to_pass = "monitor_availability"
          response = await self.availability_control.receive_command(command_to_pass, url, date_str,
frequency)

        # Send the result back to the user
        await ctx.send(response)

    @commands.command(name="stop_monitoring_availability")
    async def stop_monitoring(self, ctx):
        await ctx.send("Command recognized, passing data to control.")
```

```
        # Pass the command to the control layer using receive_command
        command_to_pass = "stop_monitoring_availability"
        response = self.availability_control.receive_command(command_to_pass)

        # Send the result back to the user
        await ctx.send(response)
```

--- BrowserBoundary.py ---

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

class BrowserBoundary(commands.Cog):
    def __init__(self):
        self.browser_control = BrowserControl() # Initialize the control object

    @commands.command(name='launch_browser')
    async def launch_browser(self, ctx):
        # Inform the user that the command is recognized
        await ctx.send("Command recognized, passing the data to control object.")

        commandToPass = "launch_browser"
        result = self.browser_control.receive_command(commandToPass)    # Pass data to the control
object
        await ctx.send(result)  # Send the result back to the user

    @commands.command(name="close_browser")
```

```
    async def stop_bot(self, ctx):
        # Inform the user that the command is recognized
        await ctx.send("Command recognized, passing the data to control object.")

        commandToPass = "close_browser"
        result = self.browser_control.receive_command(commandToPass)    # Pass data to the control
object
        await ctx.send(result)  # Send the result back to the user


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

class HelpBoundary(commands.Cog):
    def __init__(self):
        self.control = HelpControl()  # Initialize control object

    @commands.command(name="project_help")
    async def project_help(self, ctx):
        await ctx.send("Command recognized, passing data to control.")

        # Pass the command to the control object
        commandToPass = "project_help"
        response = self.control.receive_command(commandToPass)

        # Send the response back to the user
```

```
        await ctx.send(response)
```

--- LoginBoundary.py ---

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

class LoginBoundary(commands.Cog):
    def __init__(self):
        self.login_control = LoginControl()

    @commands.command(name='login')
    async def login(self, ctx, site: str):
        await ctx.send("Command recognized, passing data to control.")

        # Pass the command and site to control
        commandToPass = "login"
        result = await self.login_control.receive_command(commandToPass, site)

        # Send the result back to the user
        await ctx.send(result)
```

--- NavigationBoundary.py ---

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

```
class NavigationBoundary(commands.Cog):

    def __init__(self):
        self.navigation_control = NavigationControl()                    # Initialize the control object

    @commands.command(name='navigate_to_website')
    async def navigate_to_website(self, ctx, url: str=None):
          await ctx.send("Command recognized, passing the data to control object.")        # Inform the
user that the command is recognized

        commandToPass = "navigate_to_website"
          result = self.navigation_control.receive_command(commandToPass, url)              # Pass the
command and URL to the control object
        await ctx.send(result)                                                  # Send the result back to the user



--- PriceBoundary.py ---
from discord.ext import commands
from control.PriceControl import PriceControl

class PriceBoundary(commands.Cog):
    def __init__(self):
        # Initialize control objects directly
        self.price_control = PriceControl()

    @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, passing data to control.")
        # Pass the command to the control layer
        command_to_pass = "get_price"
        result = await self.price_control.receive_command(command_to_pass, url)
        await ctx.send(result)

    @commands.command(name='start_monitoring_price')
    async def start_monitoring_price(self, ctx, url: str = None, frequency: int = 20):
        """Command to monitor price at given frequency."""
            await ctx.send(f"Command recognized, starting price monitoring at {url} every {frequency}
second(s).")
        # Pass the command and data to the control layer
        command_to_pass = "monitor_price"
        response = await self.price_control.receive_command(command_to_pass, url, frequency)
        await ctx.send(response)

    @commands.command(name='stop_monitoring_price')
    async def stop_monitoring_price(self, ctx):
        """Command to stop monitoring the price."""
        await ctx.send("Command recognized, passing data to control.")
        # Pass the command to the control layer
        command_to_pass = "stop_monitoring_price"
        response = self.price_control.receive_command(command_to_pass)
        await ctx.send(response)
```

--- StopBoundary.py ---

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


class StopBoundary(commands.Cog):
    def __init__(self):
        self.control = StopControl()  # Initialize control object


    @commands.command(name="stop_bot")
    async def stop_bot(self, ctx):
        await ctx.send("Command recognized, passing data to control.")


        # Pass the command to the control object
        commandToPass = "stop_bot"
        result = await self.control.receive_command(commandToPass, ctx)
        print(result)  # Send the result back to the Terminal. since the bot is shut down, it won't be able to send the message back to the user.
```

--- __init__.py ---

```
#empty init file
```

--- AccountControl.py ---

```
from DataObjects.AccountDAO import AccountDAO

class AccountControl:
    def __init__(self):
```

```
    self.account_dao = AccountDAO()  # DAO for database operations

def receive_command(self, command_data, *args):
    """Handle all account-related commands and process business logic."""
    print("Data received from boundary:", command_data)

    if command_data == "fetch_all_accounts":
        return self.fetch_all_accounts()

    elif command_data == "fetch_account_by_website":
        website = args[0] if args else None
        return self.fetch_account_by_website(website)

    elif command_data == "add_account":
        username, password, website = args if args else (None, None, None)
        return self.add_account(username, password, website)

    elif command_data == "delete_account":
        account_id = args[0] if args else None
        return self.delete_account(account_id)

    else:
        result = "Invalid command."
        print(result)
        return result

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

```
        """Add a new account to the database."""
        self.account_dao.connect()  # Establish database connection
            result = self.account_dao.add_account(username, password, website)  # Call DAO to add account
        self.account_dao.close()  # Close the connection

        # Prepare the result and print it
            result_message = f"Account for {website} added successfully." if result else f"Failed to add account for {website}."
        print(result_message)
        return result_message

    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()  # Reset the ID sequence
        self.account_dao.close()  # Close the connection

        # Prepare the result and print it
         result_message = f"Account with ID {account_id} deleted successfully." if result else f"Failed to delete account with ID {account_id}."
        print(result_message)
        return result_message

    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

        # Prepare the result and print it
        if accounts:
            account_list = "\n".join([f"ID: {acc[0]}, Username: {acc[1]}, Password: {acc[2]}, Website: {acc[3]}" for acc in accounts])
            result_message = f"Accounts:\n{account_list}"
        else:
            result_message = "No accounts found."

        print(result_message)
        return result_message

    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)  # Fetch the account details from the DAO
        self.account_dao.close()  # Close the connection

        # Check if the account exists and return the raw data
        if account:
            print(f"Account found for {website}: Username: {account[0]}, Password: {account[1]}")
            return account  # Return the raw account tuple (username, password)
        else:
```

```
            print(f"No account found for {website}.")
            return None  # Return None if no account was found
```

--- AvailabilityControl.py ---

```
import asyncio
from entity.AvailabilityEntity import AvailabilityEntity
from datetime import datetime

class AvailabilityControl:
    def __init__(self):
        self.availability_entity = AvailabilityEntity()  # Initialize the entity
        self.is_monitoring = False  # Monitor state
        self.results = []  # List to store monitoring results

    async def receive_command(self, command_data, *args):
        """Handle all commands related to availability."""
        print("Data received from boundary:", command_data)

        if command_data == "check_availability":
            url = args[0]
            date_str = args[1] if len(args) > 1 else None
            return await self.check_availability(url, date_str)

        elif command_data == "monitor_availability":
            print(f"Monitoring availability at {url} every {frequency} second(s).")
```

```
        url = args[0]
        date_str = args[1] if len(args) > 1 else None
        frequency = args[2] if len(args) > 2 else 15
        return await self.start_monitoring_availability(url, date_str, frequency)

    elif command_data == "stop_monitoring_availability":
        return self.stop_monitoring()

    else:
        return "Invalid command."



async def check_availability(self, url: str, date_str=None):
    """Handle availability check and export results."""
    # Call the entity to check availability
    availability_info = await self.availability_entity.check_availability(url, date_str)

    # Prepare the result
    result = f"Checked availability: {availability_info}"
    print(result)

    # Create a DTO (Data Transfer Object) for export
    data_dto = {
        "command": "check_availability",
        "url": url,
        "result": result,
        "entered_date": datetime.now().strftime('%Y-%m-%d'),
```

```
        "entered_time": datetime.now().strftime('%H:%M:%S')
    }

    # Export data to Excel/HTML via the entity
    self.availability_entity.export_data(data_dto)
    return result

async def start_monitoring_availability(self, url: str, date_str=None, frequency=15):
    """Start monitoring availability at a specified frequency."""
    if self.is_monitoring:
        result = "Already monitoring availability."
        print(result)
        return result

    self.is_monitoring = True  # Set monitoring to active
    try:
        while self.is_monitoring:
            # Call entity to check availability
            availability_info = await self.availability_entity.check_availability(url, date_str)

            # Prepare and log the result
            result = f"Checked availability: {availability_info}"
            print(result)
            self.results.append(result)

            # Create a DTO (Data Transfer Object) for export
            data_dto = {
```

```
                "command": "start_monitoring_availability",
                "url": url,
                "result": result,
                "entered_date": datetime.now().strftime('%Y-%m-%d'),
                "entered_time": datetime.now().strftime('%H:%M:%S')
            }

            # Export data to Excel/HTML via the entity
            self.availability_entity.export_data(data_dto)

            # Wait for the specified frequency before checking again
            await asyncio.sleep(frequency)

    except Exception as e:
        error_message = f"Failed to monitor availability: {str(e)}"
        print(error_message)
        self.results.append(error_message)
        return error_message

    return self.results

def stop_monitoring(self):
    """Stop monitoring availability."""
    self.is_monitoring = False  # Set monitoring to inactive
    result = "Monitoring stopped. Collected results:" if self.results else "No data collected."
    print(result)
    return self.results if self.results else [result]
```

--- BrowserControl.py ---

```
from entity.BrowserEntity import BrowserEntity


class BrowserControl:


    def __init__(self):
        # Initialize the entity object inside the control layer
        self.browser_entity = BrowserEntity()


    def receive_command(self, command_data):
        # Validate the command
        print("Data Received from boundary object: ", command_data)
        if command_data == "launch_browser":
            # Call the entity to perform the actual operation
            result = self.browser_entity.launch_browser()
            return result


        elif command_data == "close_browser":
            # Call the entity to perform the close operation
            result = self.browser_entity.close_browser()
            return result


        else:
            return "Invalid command."
```

```
--- HelpControl.py ---
class HelpControl:
    def receive_command(self, command_data):
        """Handles the command and returns the appropriate message."""
        print("Data received from boundary:", command_data)

        if command_data == "project_help":
            help_message = (
                "Here are the available commands:\n"
                "!project_help - Get help on available commands.\n"
                "!fetch_all_accounts - Fetch all stored accounts.\n"
                "!add_account 'username' 'password' 'website' - Add a new account to the database.\n"
                "!fetch_account_by_website 'website' - Fetch account details by website.\n"
                "!delete_account 'account_id' - Delete an account by its ID.\n"
                "!launch_browser - Launch the browser.\n"
                "!close_browser - Close the browser.\n"
                "!navigate_to_website 'url' - Navigate to a specified website.\n"
                "!login 'website' - Log in to a website (e.g., !login bestbuy).\n"
                "!get_price 'url' - Check the price of a product on a specified website.\n"
                    "!start_monitoring_price 'url' 'frequency' - Start monitoring a product's price at a specific interval (frequency in minutes).\n"
                "!stop_monitoring_price - Stop monitoring the product's price.\n"
                "!check_availability 'url' - Check availability for a restaurant or service.\n"
                "!monitor_availability 'url' 'frequency' - Monitor availability at a specific interval.\n"
                "!stop_monitoring_availability - Stop monitoring availability.\n"
                "!stop_bot - Stop the bot.\n"
```

```
            )


            return help_message
        else:
            return "Invalid command."



--- LoginControl.py ---
from control.AccountControl import AccountControl
from entity.BrowserEntity import BrowserEntity
from utils.css_selectors import Selectors

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

    async def receive_command(self, command_data, site=None):
        """Handle login command and perform business logic."""
        print("Data received from boundary:", command_data)

        if command_data == "login" and site:
            # Fetch account credentials from the entity
            account_info = self.account_control.fetch_account_by_website(site)
            if not account_info:
                return f"No account found for {site}"
```

```
            username, password = account_info[0], account_info[1]
            print(f"Username: {username}, Password: {password}")

            # 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."

            # Perform the login process via the entity
            result = await self.browser_entity.perform_login(url, username, password)
            return result
        else:
            return "Invalid command or site."


--- NavigationControl.py ---
from entity.BrowserEntity import BrowserEntity
from utils.css_selectors import Selectors

class NavigationControl:

    def __init__(self):
        # Initialize the entity object inside the control layer
        self.browser_entity = BrowserEntity()

    def receive_command(self, command_data, url=None):
```

```
        # Validate the command
        print("Data Received from boundary object: ", command_data)
        if command_data == "navigate_to_website":
            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)

            result = self.browser_entity.navigate_to_website(url)              # Call the entity to navigate to
the given URL
            return result
        else:
            return "Invalid command."
```

--- PriceControl.py ---

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

class PriceControl:
    def __init__(self):
        self.price_entity = PriceEntity()  # Initialize PriceEntity for fetching and export
        self.is_monitoring = False  # Monitoring flag
```

```
    self.results = []  # Store monitoring results

async def receive_command(self, command_data, *args):
    """Handle all price-related commands and process business logic."""
    print("Data received from boundary:", command_data)

    if command_data == "get_price":
        url = args[0] if args else None
        return await self.get_price(url)

    elif command_data == "monitor_price":
        url = args[0] if args else None
        frequency = args[1] if len(args) > 1 else 20
        return await self.start_monitoring_price(url, frequency)

    elif command_data == "stop_monitoring_price":
        return self.stop_monitoring()

    else:
        return "Invalid command."

async def get_price(self, url: str):
    """Handle fetching the price from the entity."""
    # If no URL is provided, default to BestBuy
    if not url:
        selectors = Selectors.get_selectors_for_url("bestbuy")
        url = selectors.get('priceUrl')
```

```
        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)

    # Fetch the price from the entity
    price = self.price_entity.get_price_from_page(url)

    data_dto = {
                "command": "monitor_price",
                "url": url,
                "result": price,
                "entered_date": datetime.now().strftime('%Y-%m-%d'),
                "entered_time": datetime.now().strftime('%H:%M:%S')
            }

            # Pass the DTO to PriceEntity to handle export
    self.price_entity.export_data(data_dto)
    return price

async def start_monitoring_price(self, url: str = None, frequency=20):
    """Start monitoring the price at a given interval."""
    if self.is_monitoring:
        return "Already monitoring prices."

    self.is_monitoring = True
    previous_price = None
```

```
        try:
            while self.is_monitoring:
                # Fetch the current price
                if not url:
                    selectors = Selectors.get_selectors_for_url("bestbuy")
                    url = selectors.get('priceUrl')
                    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) for export
            data_dto = {
                "command": "monitor_price",
                "url": url,
                "result": result,
                "entered_date": datetime.now().strftime('%Y-%m-%d'),
                "entered_time": datetime.now().strftime('%H:%M:%S')
            }

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

            await asyncio.sleep(frequency)

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

def stop_monitoring(self):
    """Stop monitoring the price."""
    self.is_monitoring = False
    result = self.results if self.results else ["No data collected."]
    return result
```

--- StopControl.py ---

```
import discord


class StopControl:
    async def receive_command(self, command_data, ctx):
        """Handle the stop bot command."""
        print("Data received from boundary:", command_data)


        if command_data == "stop_bot":
            # Get the bot from the context (ctx) dynamically
            bot = ctx.bot  # This extracts the bot instance from the context
            await ctx.send("The bot is shutting down...")
            print("Bot is shutting down...")
            await bot.close()  # Close the bot
            result = "Bot has been shut down."
            print(result)
            return result
        else:
            result = "Invalid command."
            return result
```

--- __init__.py ---

```
#empty init file
```

--- AccountDAO.py ---

```
import psycopg2
from utils.Config import Config


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, username: str, password: str, website: str):
        """Add a new account to the database using structured data."""
        try:
            # Combine DTO logic here by directly using the parameters
            query = "INSERT INTO accounts (username, password, website) VALUES (%s, %s, %s)"
            values = (username, password, website)
            self.cursor.execute(query, values)
            self.connection.commit()
            print(f"Account {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,))
            result = self.cursor.fetchone()
            print(result)
            return result
        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)
        result = self.cursor.fetchall()
        print(result)
        return result
    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.")
```

--- AvailabilityEntity.py ---

```
import asyncio
from utils.exportUtils import ExportUtils
from entity.BrowserEntity import BrowserEntity
from utils.css_selectors import Selectors
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
```

```
from selenium.webdriver.support import expected_conditions as EC


class AvailabilityEntity:
    def __init__(self):
        self.browser_entity = BrowserEntity()



    async def check_availability(self, url: str, date_str=None, timeout=5):
        # Use BrowserEntity to navigate to the URL
        self.browser_entity.navigate_to_website(url)

        # Get selectors for the given URL
        selectors = Selectors.get_selectors_for_url(url)
        if not selectors:
            return "No valid selectors found for this URL."
        # Perform date and time selection (optional)
        if date_str:
            try:
                                date_field = self.browser_entity.driver.find_element(By.CSS_SELECTOR,
selectors['date_field'])
                date_field.click()
                await asyncio.sleep(1)
                                date_button = self.browser_entity.driver.find_element(By.CSS_SELECTOR,
f"{selectors['select_date']} button[aria-label*='{date_str}']")
                date_button.click()
            except Exception as e:
                return f"Failed to select the date: {str(e)}"
```

```
    await asyncio.sleep(2)  # Wait for updates (adjust this time based on page response)

    # Initialize flags for select_time and no_availability elements
    select_time_seen = False
    no_availability_seen = False
    try:
        # Check if 'select_time' is available within the given timeout
        WebDriverWait(self.browser_entity.driver, timeout).until(
            EC.presence_of_element_located((By.CSS_SELECTOR, selectors['select_time']))
        )
        select_time_seen = True  # If found, set the flag to True
    except:
        select_time_seen = False  # If not found within timeout
    try:
        # Check if 'no_availability' is available within the given timeout
        WebDriverWait(self.browser_entity.driver, timeout).until(
                                        lambda driver: len(driver.find_elements(By.CSS_SELECTOR,
selectors['show_next_available_button'])) > 0
        )
        no_availability_seen = True  # If found, set the flag to True
    except:
        no_availability_seen = False  # If not found within timeout

    # Logic to determine availability
    if select_time_seen:
            return f"Selected or default date {date_str if date_str else 'current date'} is available for
```

```
booking."
        elif no_availability_seen:
            return "No availability for the selected date."
        else:
            return "Unable to determine availability. Please try again."




    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
        excelResult = 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
    )
    print(excelResult)

    # Call the HTML export method from ExportUtils
    htmlResult = 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
    )
    print(htmlResult)
```

--- 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:
```

```
_instance = None

def __new__(cls, *args, **kwargs):
    if not cls._instance:
        cls._instance = super(BrowserEntity, cls).__new__(cls, *args, **kwargs)
    return cls._instance

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
            result = "Browser launched."
            print(result)
            return result
        else:
            result = "Browser is already running."
            print(result)
            return result


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

```
        return result



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

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



async def perform_login(self, url, username, password):
    # Navigate to the website
    self.navigate_to_website(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'])))

            result = f"Logged in to {url} successfully with username: {username}"
            print(result)
            return result
        except Exception as e:
            result = f"Failed to log in: {str(e)}"
```

```
            print(result)
            return result



--- PriceEntity.py ---
from selenium.webdriver.common.by import By
from entity.BrowserEntity import BrowserEntity
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

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):
        self.browser_entity = BrowserEntity()

    def get_price_from_page(self, url: str):
        # Navigate to the URL using BrowserEntity
        self.browser_entity.navigate_to_website(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'])
            result = price_element.text
```

```
            print(f"Price found: {result}")
            return result
        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
        excelResult = 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
        )
```

```
        print(excelResult)

        # Call the HTML export method from ExportUtils
        htmlResult = 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
        )
        print(htmlResult)
```

--- __init__.py ---

```
#empty init file
```

--- projectToText.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, "projectToText.pdf")

# Lists for files and folders to ignore
filesToIgnore = ['ignore_this.py', 'Tests_URLs.txt', 'UseCases.txt', 'Read.md', '*.pdf']  # Example file
```

```
names to ignore
foldersToIgnore = ['ignore_folder', '.git', '__pycache__', 'PersonelTest', 'MockTesting', 'ExportedFiles']  # Folders to ignore

# Function to retrieve all text from files, ignoring specific folders and files
def extract_project_text(directory, ignore_files=None, ignore_folders=None):
    if ignore_files is None:
        ignore_files = []
    if ignore_folders is None:
        ignore_folders = []

    project_text = ""
    for root, dirs, files in os.walk(directory):
        # Ignore specific folders
        dirs[:] = [d for d in dirs if d not in ignore_folders]

        for file in files:
            # Skip ignored files
            if file in ignore_files:
                continue
            # Only considering relevant file types
            if file.endswith('.py'):
                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)

# Function to create PDFs for specific folders
def create_folder_specific_pdfs(directory, ignore_files=None, ignore_folders=None):
    if ignore_files is None:
        ignore_files = []
```

```
    if ignore_folders is None:
        ignore_folders = []

    # Create PDFs for each folder in the project
    for folder in os.listdir(directory):
        folder_path = os.path.join(directory, folder)
        if os.path.isdir(folder_path) and folder not in ignore_folders:
            folder_text = extract_project_text(folder_path, ignore_files, ignore_folders)
            if folder_text:
                folder_pdf_path = os.path.join(folder_path, f"All_files_in_{folder}_folder_toText.pdf")
                create_pdf(folder_text, folder_pdf_path)
                print(f"PDF created for folder {folder} at: {folder_pdf_path}")

# Extract project text and create the main project PDF
project_text = extract_project_text(directory, filesToIgnore, foldersToIgnore)
if project_text:
    create_pdf(project_text, output_pdf_path)
    print(f"Main PDF created with all project's text at: {output_pdf_path}")
else:
    print("No project text found.")

# Create PDFs for each specific folder
create_folder_specific_pdfs(directory, filesToIgnore, foldersToIgnore)



--- 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}")



--- test_!project_help.py ---
import sys, os, discord, logging, unittest
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
```

```
from unittest.mock import AsyncMock, patch, call
from utils.MyBot import MyBot
"""
File: test_!project_help.py
Purpose: This file contains unit tests for the !project_help command in the Discord bot.
The tests validate both successful and error scenarios, ensuring the bot provides the correct help message and handles errors properly.
Tests:
- Positive: Simulates the !project_help command and verifies the correct help message is sent.
- Negative: Simulates an error scenario and ensures the error is handled gracefully.
"""
# Setup logging configuration
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# Expected help message
help_message = (
    "Here are the available commands:\n"
    "!project_help - Get help on available commands.\n"
    "!fetch_all_accounts - Fetch all stored accounts.\n"
    "!add_account 'username' 'password' 'website' - Add a new account to the database.\n"
    "!fetch_account_by_website 'website' - Fetch account details by website.\n"
    "!delete_account 'account_id' - Delete an account by its ID.\n"
    "!launch_browser - Launch the browser.\n"
    "!close_browser - Close the browser.\n"
    "!navigate_to_website 'url' - Navigate to a specified website.\n"
    "!login 'website' - Log in to a website (e.g., !login bestbuy).\n"
    "!get_price 'url' - Check the price of a product on a specified website.\n"
```

```
    "!start_monitoring_price 'url' 'frequency' - Start monitoring a product's price at a specific interval (frequency in minutes).\n"
    "!stop_monitoring_price - Stop monitoring the product's price.\n"
    "!check_availability 'url' - Check availability for a restaurant or service.\n"
    "!monitor_availability 'url' 'frequency' - Monitor availability at a specific interval.\n"
    "!stop_monitoring_availability - Stop monitoring availability.\n"
    "!stop_bot - Stop the bot.\n"
)



class CustomTextTestResult(unittest.TextTestResult):
    """Custom test result to output 'Unit test passed' instead of 'ok'."""
    def addSuccess(self, test):
        super().addSuccess(test)
        self.stream.write("Unit test passed\n")  # Custom success message
        self.stream.flush()

class CustomTextTestRunner(unittest.TextTestRunner):
    """Custom test runner that uses the custom result class."""
    resultclass = CustomTextTestResult

class TestProjectHelpCommand(unittest.IsolatedAsyncioTestCase):

    async def asyncSetUp(self):
        """Setup the bot and mock context before each test."""
        logging.info("Setting up the bot and mock context for testing...")
        intents = discord.Intents.default()  # Create default intents
```

```
        intents.message_content = True  # Ensure the bot can read message content
        self.bot = MyBot(command_prefix="!", intents=intents)  # Initialize the bot with intents
        self.ctx = AsyncMock()  # Mock context (ctx)
        self.ctx.send = AsyncMock()  # Mock the send method to capture responses

        # Call setup_hook to ensure commands are registered
        await self.bot.setup_hook()

    async def test_project_help_success(self):
        """Test the project help command when it successfully returns the help message."""
        logging.info("Starting test: test_project_help_success")

        # Simulate calling the project_help command
        logging.info("Simulating the project_help command call.")
        command = self.bot.get_command("project_help")
            self.assertIsNotNone(command, "project_help command is not registered.")  # Ensure the command is registered
        await command(self.ctx)

        # Check both the control message and help message were sent
        expected_calls = [
            call('Command recognized, passing data to control.'),  # First message sent by the bot
            call(help_message)  # Second message: the actual help message
        ]
         self.ctx.send.assert_has_calls(expected_calls, any_order=False)  # Ensure the messages are sent in the correct order
        logging.info("Verified that both the control and help messages were sent.")
```

```
    async def test_project_help_error(self):
        """Test the project help command when it encounters an error during execution."""
        logging.info("Starting test: test_project_help_error")

        # Simulate calling the project_help command and an error occurring
        logging.info("Simulating the project_help command call.")
            self.ctx.send.side_effect = Exception("Error during project_help execution.")  # Simulate an error

        command = self.bot.get_command("project_help")
            self.assertIsNotNone(command, "project_help command is not registered.")  # Ensure the command is registered

        # Act & Assert: Expect the exception to be raised
        with self.assertRaises(Exception):
            await command(self.ctx)

        logging.info("Verified that an error occurred and was handled.")


if __name__ == "__main__":
    # Use the custom test runner to display 'Unit test passed'
    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))


--- test_!stop_bot.py ---
```

```
import sys, os, discord, logging, unittest
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from unittest.mock import MagicMock, AsyncMock, call, patch
from utils.MyBot import MyBot

# Setup logging configuration
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

"""
File: test_!stop_bot.py
Purpose: This file contains unit tests for the !stop_bot command in the Discord bot.
The tests validate both successful and error scenarios, ensuring the bot correctly shuts down or handles errors during shutdown.
Tests:
- Positive: Simulates the !stop_bot command and verifies the bot shuts down correctly.
- Negative: Simulates an error during shutdown and ensures it is handled gracefully.
"""

class CustomTextTestResult(unittest.TextTestResult):
    """Custom test result to output 'Unit test passed' instead of 'ok'."""
    def addSuccess(self, test):
        super().addSuccess(test)
        self.stream.write("Unit test passed\n")  # Custom success message
        self.stream.flush()

class CustomTextTestRunner(unittest.TextTestRunner):
    """Custom test runner that uses the custom result class."""
```

```
    resultclass = CustomTextTestResult

class TestStopBotCommand(unittest.IsolatedAsyncioTestCase):

    async def asyncSetUp(self):
        """Setup the bot and mock context before each test."""
        logging.info("Setting up the bot and mock context for testing...")
        intents = discord.Intents.default()  # Create default intents
        intents.message_content = True  # Ensure the bot can read message content
        self.bot = MyBot(command_prefix="!", intents=intents)  # Initialize the bot with intents
        self.ctx = AsyncMock()  # Mock context (ctx)
        self.ctx.send = AsyncMock()  # Mock the send method to capture responses
        self.ctx.bot = self.bot  # Mock the bot property in the context

        # Call setup_hook to ensure commands are registered
        await self.bot.setup_hook()

    async def test_stop_bot_success(self):
        """Test the stop bot command when it successfully shuts down."""
        logging.info("Starting test: test_stop_bot_success")

        # Patch the bot's close method on the ctx.bot (since bot is retrieved from ctx dynamically)
        with patch.object(self.ctx.bot, 'close', new_callable=AsyncMock) as mock_close:
            # Simulate calling the stop_bot command
            logging.info("Simulating the stop_bot command call.")
            command = self.bot.get_command("stop_bot")
                self.assertIsNotNone(command, "stop_bot command is not registered.")  # Ensure the
```

```
command is registered
            await command(self.ctx)

            # Check if the correct messages were sent
            expected_calls = [
                call('Command recognized, passing data to control.'),  # First message sent by the bot
                call('The bot is shutting down...')  # Second message confirming the shutdown
            ]
               self.ctx.send.assert_has_calls(expected_calls, any_order=False)  # Ensure the messages
are sent in the correct order
            logging.info("Verified that both expected messages were sent to the user.")

            # Check if bot.close() was called on the ctx.bot
            mock_close.assert_called_once()
            logging.info("Verified that the bot's close method was called once.")

    async def test_stop_bot_error(self):
        """Test the stop bot command when it encounters an error during shutdown."""
        logging.info("Starting test: test_stop_bot_error")

        # Patch the bot's close method to raise an exception
        with patch.object(self.ctx.bot, 'close', new_callable=AsyncMock) as mock_close:
            mock_close.side_effect = Exception("Error stopping bot")  # Simulate an error

            # Simulate calling the stop_bot command
            logging.info("Simulating the stop_bot command call.")
            command = self.bot.get_command("stop_bot")
```

```
            self.assertIsNotNone(command, "stop_bot command is not registered.")  # Ensure the command is registered

        # Act & Assert: Expect the exception to be raised
        with self.assertRaises(Exception):
            await command(self.ctx)

        logging.info("Verified that an error occurred and was handled correctly.")

        # Ensure ctx.send was still called with the shutdown message before the error occurred
        self.ctx.send.assert_called_with("The bot is shutting down...")
        logging.info("Verified that 'The bot is shutting down...' message was sent despite the error.")

        # Verify that the close method was still attempted
        mock_close.assert_called_once()
        logging.info("Verified that the bot's close method was called even though it raised an error.")

if __name__ == "__main__":
    # Use the custom test runner to display 'Unit test passed'
    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))
```

--- __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/",
            "unavailableUrl": "https://www.opentable.com/r/bar-spero-washington/",
            "availableUrl": "https://www.opentable.com/r/the-rux-nashville",
            "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}."



--- MyBot.py ---
import discord
from discord.ext import commands
from boundary.BrowserBoundary import BrowserBoundary
from boundary.NavigationBoundary import NavigationBoundary
from boundary.HelpBoundary import HelpBoundary
from boundary.StopBoundary import StopBoundary
from boundary.LoginBoundary import LoginBoundary
from boundary.AccountBoundary import AccountBoundary
from boundary.AvailabilityBoundary import AvailabilityBoundary
from boundary.PriceBoundary import PriceBoundary



class MyBot(commands.Bot):

    async def setup_hook(self):
        await self.add_cog(BrowserBoundary())
        await self.add_cog(NavigationBoundary())
        await self.add_cog(HelpBoundary())
        await self.add_cog(StopBoundary())
```

```
        await self.add_cog(LoginBoundary())
        await self.add_cog(AccountBoundary())
        await self.add_cog(AvailabilityBoundary())
        await self.add_cog(PriceBoundary())

    async def on_ready(self):
        print(f"Logged in as {self.user}")
            channel = discord.utils.get(self.get_all_channels(), name="general")  # Adjust the channel
name if needed
        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):
        if isinstance(error, commands.CommandNotFound):
            await ctx.send("Command not recognized. Type !project_help to see the list of commands.")
```