```
--- main.py ---
from utils.MyBot import start_bot
from utils.Config import Config


# Initialize and run the bot
if __name__ == "__main__":
    print("Bot is starting...")
    start_bot(Config.DISCORD_TOKEN)  # Start the bot using the token from config




--- AccountBoundary.py ---
from discord.ext import commands
from control.AccountControl import AccountControl
from DataObjects.global_vars import GlobalState


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.")


        list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
command and up to 6 variables
        command = list[0]  # First element is the command
        result = self.control.receive_command(command)

        # 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):
         list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
command and up to 6 variables
        command = list[0]  # First element is the command
        website = list[1]  # Second element is the URL

        await ctx.send(f"Command recognized, passing data to control for website {website}.")

        result = self.control.receive_command(command, 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):
        await ctx.send("Command recognized, passing data to control.")

         list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
command and up to 6 variables
```

```
        command = list[0]  # First element is the command
        username = list[1]  # Second element is the username
        password = list[2]  # Third element is the passwrod
        website = list[3]  # Third element is the website

        result = self.control.receive_command(command, 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):

        list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into command and up to 6 variables
        command = list[0]  # First element is the command
        account_id = list[1]  # Second element is the account_id

        await ctx.send(f"Command recognized, passing data to control to delete account with ID {account_id}.")

        result = self.control.receive_command(command, 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
from DataObjects.global_vars import GlobalState

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):
        await ctx.send("Command recognized, passing data to control.")

         list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
command and up to 6 variables

        command = list[0]  # First element is the command
        url = list[1]  # Second element is the URL
        date_str = list[2]  # Third element is the date

        # Pass the command and data to the control layer using receive_command
        result = await self.availability_control.receive_command(command, url, date_str)
        # Send the result back to the user
        await ctx.send(result)



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

         list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
command and up to 6 variables

        command = list[0]  # First element is the command
        url = list[1]  # Second element is the URL
        date_str = list[2]  # Third element is the date
        frequency = list[3] # Fourth element is the frequency

        response = await self.availability_control.receive_command(command, 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_availability(self, ctx):
        """Command to stop monitoring the price."""
        await ctx.send("Command recognized, passing data to control.")
        list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
command and up to 6 variables

        command = list[0]  # First element is the command

            response = await self.availability_control.receive_command(command)        # Pass the
command to the control layer
        await ctx.send(response)



--- BrowserBoundary.py ---
from discord.ext import commands
from control.BrowserControl import BrowserControl
from DataObjects.global_vars import GlobalState

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):
        await ctx.send(f"Command recognized, passing to control object.")

        list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
command and up to 6 variables
        command = list[0]  # First element is the command
```

```
            result = self.browser_control.receive_command(command)         # Pass the updated
user_message 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):
        await ctx.send(f"Command recognized, passing to control object.")

         list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
command and up to 6 variables
        command = list[0]  # First element is the command

        result = self.browser_control.receive_command(command)
        await ctx.send(result)



--- HelpBoundary.py ---
from discord.ext import commands
from control.HelpControl import HelpControl
from DataObjects.global_vars import GlobalState

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.")

         list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
command and up to 6 variables
        command = list[0]  # First element is the command

        response = self.control.receive_command(command)

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



--- LoginBoundary.py ---
from discord.ext import commands
from control.LoginControl import LoginControl
from DataObjects.global_vars import GlobalState

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

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

         list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
command and up to 6 variables
```

```
        command = list[0]  # First element is the command
        website = list[1]


        result = await self.login_control.receive_command(command, website)


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

```
--- NavigationBoundary.py ---
from discord.ext import commands
from control.NavigationControl import NavigationControl
from DataObjects.global_vars import GlobalState


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


         list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
command and up to 6 variables
```

```
        command = list[0]  # First element is the command
        website = list[1]  # Second element is the URL

        result = self.navigation_control.receive_command(command, website) # Pass the parsed variables 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
from DataObjects.global_vars import GlobalState

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):
        """Command to get the price from the given URL."""
        await ctx.send("Command recognized, passing data to control.")

        list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into command and up to 6 variables
        command = list[0]  # First element is the command
        website = list[1]  # Second element is the URL
        result = await self.price_control.receive_command(command, website) # Pass the command to the control layer
        await ctx.send(f"Price found: {result}")


    @commands.command(name='start_monitoring_price')
    async def start_monitoring_price(self, ctx):
        """Command to monitor price at given frequency."""
        list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into command and up to 6 variables
        command = list[0]  # First element is the command
        website = list[1]  # Second element is the URL
        frequency = list[2]

        await ctx.send(f"Command recognized, starting price monitoring at {website} every {frequency} second(s).")

        response = await self.price_control.receive_command(command, website, 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.")
        list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
command and up to 6 variables
        command = list[0]  # First element is the command

        response = await self.price_control.receive_command(command)            # Pass the command
to the control layer

        await ctx.send(response)



--- StopBoundary.py ---
from discord.ext import commands
from control.StopControl import StopControl
from DataObjects.global_vars import GlobalState

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.")

        list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
command and up to 6 variables
        command = list[0]  # First element is the command
        result = await self.control.receive_command(command, 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, *args):
        """Handle all account-related commands and process business logic."""
        print("Data received from boundary:", command)


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


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


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

        elif command == "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()
        result = self.account_dao.add_account(username, password, website)
        self.account_dao.close()

        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()
        try:
            result = self.account_dao.delete_account(account_id)
        except Exception as e:
            print(f"Error deleting account: {e}")
            return "Error deleting account."
        self.account_dao.reset_id_sequence()
        self.account_dao.close()

        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()
        try:
            accounts = self.account_dao.fetch_all_accounts()
        except Exception as e:
            return "Error fetching accounts."
        self.account_dao.close()

        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."""
        try:
            self.account_dao.connect()
            account = self.account_dao.fetch_account_by_website(website)
            self.account_dao.close()

            # Logic to format the result within the control layer
            if account:
                return account
            else:
                return f"No account found for {website}."

        except Exception as e:
            return f"Error: {str(e)}"



--- AvailabilityControl.py ---
import asyncio
from entity.AvailabilityEntity import AvailabilityEntity
from datetime import datetime
from utils.css_selectors import Selectors
```

```
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 == "start_monitoring_availability":
            url = args[0]
            date_str = args[1] if len(args) > 1 else None
            frequency = args[2] if len(args) > 2 and args[2] not in [None, ""] else 15
            return await self.start_monitoring_availability(url, date_str, frequency)

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

        else:
            print("Invalid command.")
            return "Invalid command."
```

```
async def check_availability(self, url: str, date_str=None):
    """Handle availability check and export results."""
    print("Checking availability...")
    # Call the entity to check availability
    try:
        if not url:
            selectors = Selectors.get_selectors_for_url("opentable")
            url = selectors.get('availableUrl')
            if not url:
                return "No URL provided, and default URL for openTable could not be found."
            print("URL not provided, default URL for openTable is: " + url)

        availability_info = await self.availability_entity.check_availability(url, date_str)

    # Prepare the result
        result = f"Checked availability: {availability_info}"
    except Exception as e:
        result = f"Failed to check availability: {str(e)}"
    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."""
    print("Monitoring availability")
    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
            result = await self.check_availability(url, date_str)
            self.results.append(result) # Store the result in the list
            await asyncio.sleep(frequency)  # Wait for the specified frequency before checking again

    except Exception as e:
        error_message = f"Failed to monitor availability: {str(e)}"
```

```
            print(error_message)
            return error_message

        return self.results


    def stop_monitoring_availability(self):
        """Stop monitoring availability."""
        print("Stopping availability monitoring...")
        result = None
        try:
            if not self.is_monitoring:
                # If no monitoring session is active
                result = "There was no active availability monitoring session. Nothing to stop."
            else:
                # Stop monitoring and collect results
                self.is_monitoring = False
                result = "Results for availability monitoring:\n"
                result += "\n".join(self.results)
                result = result + "\n" + "\nAvailability monitoring stopped successfully!"
                print(result)
        except Exception as e:
            # Handle any error that occurs
            result = f"Error stopping availability monitoring: {str(e)}"

        return result
```

--- BrowserControl.py ---

```
from entity.BrowserEntity import BrowserEntity


class BrowserControl:


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


    def receive_command(self, command_data):
        print("Data Received from boundary object: ", command_data)
        try:
            if command_data == "launch_browser":
                result = self.browser_entity.launch_browser()
            elif command_data == "close_browser":
                result = self.browser_entity.close_browser()
            else:
                result = "Invalid command."
            return f"Control Object Result: {result}"
        except Exception as e:
            error_msg = f"Control Layer Exception: {str(e)}"
            return error_msg
```

--- 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"
                "!start_monitoring_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:
            try:
                # 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."

                result = await self.browser_entity.login(url, username, password)
            except Exception as e:
                result = str(e)
            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, url=None):
        # Validate the command
        print("Data Received from boundary object: ", command)

        if command == "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)

            try:

                result = self.browser_entity.navigate_to_website(url) # Call the entity to perform the actual
operation

            except Exception as e:

                result = str(e)

            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 == "start_monitoring_price":
            url = args[0] if args else None
            frequency = args[1] if len(args) > 1 and args[1] not in [None, ""] else 20
            return await self.start_monitoring_price(url, frequency)

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

        else:
            return "Invalid command."


    async def get_price(self, url: str):
        """Handle fetching the price from the entity."""
        print("getting price...")
```

```
try:
    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

    result = self.price_entity.get_price_from_page(url)
    print(f"Price found: {result}")
    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)

except Exception as e:
    return f"Failed to fetch price: {str(e)}"

return result
```

```
    async def start_monitoring_price(self, url: str, frequency=20):
        """Start monitoring the price at a given interval."""
        print("Starting price monitoring...")
        try:
            if self.is_monitoring:
                return "Already monitoring prices."

            self.is_monitoring = True
            previous_price = None

            while self.is_monitoring:
                current_price = await self.get_price(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)
            await asyncio.sleep(frequency)


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




def stop_monitoring_price(self):
    """Stop the price monitoring loop."""
    print("Stopping price monitoring...")
    result = None
    try:
        if not self.is_monitoring:
            # If no monitoring session is active
            result = "There was no active price monitoring session. Nothing to stop."
        else:
            # Stop monitoring and collect results
            self.is_monitoring = False
            result = "Results for price monitoring:\n"
            result += "\n".join(self.results)
            result = result + "\n" +"\nPrice monitoring stopped successfully!"
            print(result)
    except Exception as e:
        # Handle any error that occurs
        result = f"Error stopping price monitoring: {str(e)}"

    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."""
        try:
            if self.cursor:
                self.cursor.close()
            if self.connection:
                self.connection.close()
                print("Database connection closed.")
        except Exception as error:
            print(f"Error closing the database connection: {error}")



--- global_vars.py ---
import re


class GlobalState:
    user_message = 'default'


    @classmethod
    def reset_user_message(cls):
        """Reset the global user_message variable to None."""
        cls.user_message = None


    @classmethod
    def parse_user_message(cls, message):
        """
        Parses a user message by splitting it into command and up to 6 variables.
        Handles quoted substrings so that quoted parts (e.g., "October 2") remain intact.
        """
        #print(f"User_message before parsing: {message}")
        message = message.replace("!", "").strip()  # Remove "!" and strip spaces
        #print(f"User_message after replacing '!' with empty string: {message}")


        # Simple split by spaces, keeping quoted substrings intact
        parts = re.findall(r'\"[^\"]+\"|\S+', message)
        #print(f"Parts after splitting: {parts}")
        # Ensure we always return 6 variables (command + 5 parts), even if some are empty
        result = [parts[i].strip('"') if len(parts) > i else "" for i in range(6)]  # List comprehension to handle
missing parts

        #print(f"Result: {result}")
        return result  # Return the list (or tuple if needed)


--- 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=15):
        try:
            # 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)

        # Perform date selection (optional)
        if date_str:
            try:
                await asyncio.sleep(3)  # Wait for updates to load
                print(selectors['date_field'])
                date_field = self.browser_entity.driver.find_element(By.CSS_SELECTOR,
selectors['date_field'])
                date_field.click()
                await asyncio.sleep(3)
                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 to load

        # 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."

    except Exception as e:
        return f"Failed to check availability: {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.
    """
    try:
        # 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)
    # Export operations...
except Exception as e:
    return f"priceEntity_Error exporting data: {str(e)}"
```

```
--- 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):
        try:
            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."
                return result
            else:
                result = "Browser is already running."
                return result
        except Exception as e:
            result = f"BrowserEntity_Failed to launch browser: {str(e)}"
            return result

    def close_browser(self):
        try:
            if self.browser_open and self.driver:
                self.driver.quit()
                self.browser_open = False
                return "Browser closed."
            else:
                return "No browser is currently open."
        except Exception as e:
            return f"BrowserEntity_Failed to close browser: {str(e)}"


    def navigate_to_website(self, url):
        try:
            if not self.is_browser_open():
                launch_message = self.launch_browser()
                if "Failed" in launch_message:
                    return launch_message


            if self.driver:
                self.driver.get(url)
                return f"Navigated to {url}"
            else:
                return "Failed to open browser."
        except Exception as e:
            return f"BrowserEntity_Failed to navigate to {url}: {str(e)}"


    async def login(self, url, username, password):
        try:
            navigate_message = self.navigate_to_website(url)
            if "Failed" in navigate_message:
                return navigate_message


                                            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)
```

```
                                        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)

                                        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)

                                                                                     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"BrowserEntity_Failed to log in to {url}: {str(e)}"


--- 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
            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.
        """
        try:
            # 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)

        except Exception as e:

            return f"priceEntity_Error exporting data: {str(e)}"
```

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

#empty init file


--- test_!add_account.py ---

from unittest.mock import patch

import logging, unittest

from test_init import BaseTestSetup, CustomTextTestRunner


class TestAddAccountCommand(BaseTestSetup):


    @patch('DataObjects.global_vars.GlobalState.parse_user_message')

    @patch('DataObjects.AccountDAO.AccountDAO.add_account')

    async def test_add_account_success(self, mock_add_account, mock_parse_user_message):

        """Test the add_account command when it succeeds."""

        # Simulate parsing user message and extracting command parameters

                mock_parse_user_message.return_value = ["add_account", "testuser", "password123",
"example.com"]

        # Simulate successful account addition in the database

        mock_add_account.return_value = True


        # Triggering the command within the bot

        command = self.bot.get_command("add_account")

        await command(self.ctx)


        # Validate that the success message is correctly sent to the user

        self.ctx.send.assert_called_with("Account for example.com added successfully.")

         logging.info("Verified successful account addition - database addition simulated and feedback
provided.")


    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('DataObjects.AccountDAO.AccountDAO.add_account')
    async def test_add_account_error(self, mock_add_account, mock_parse_user_message):
        """Test the add_account command when it encounters an error."""
        # Setup for receiving command and failing to add account
        mock_parse_user_message.return_value = ["add_account", "testuser", "password123",
"example.com"]
        mock_add_account.return_value = False

        # Command execution with expected failure
        command = self.bot.get_command("add_account")
        await command(self.ctx)

        # Ensuring error feedback is correctly relayed to the user
        self.ctx.send.assert_called_with("Failed to add account for example.com.")
        logging.info("Verified error handling during account addition - simulated database failure and
error feedback.")


if __name__ == "__main__":
    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))



--- test_!check_availability.py ---
import logging, unittest
from unittest.mock import patch
from test_init import BaseTestSetup, CustomTextTestRunner


"""
File: test_!check_availability.py
Purpose: Unit tests for the !check_availability command in the Discord bot.
"""


class TestCheckAvailabilityCommand(BaseTestSetup):


    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('control.AvailabilityControl.AvailabilityControl.receive_command')
    async def test_check_availability_success(self, mock_receive_command,
mock_parse_user_message):
        """Test the check_availability command when it succeeds."""
        logging.info("Starting test: test_check_availability_success")


        # Mock the parsed message to return the expected command and arguments
        mock_parse_user_message.return_value = ["check_availability", "https://example.com",
"2024-09-30"]


        # Simulate successful availability check
        mock_receive_command.return_value = "Available for booking."


        command = self.bot.get_command("check_availability")
        self.assertIsNotNone(command)


        # Call the command without arguments (since GlobalState is mocked)
        await command(self.ctx)

        expected_message = "Available for booking."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful availability check.")

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('control.AvailabilityControl.AvailabilityControl.receive_command')
    async def test_check_availability_error(self, mock_receive_command,
mock_parse_user_message):
        """Test the check_availability command when it encounters an error."""
        logging.info("Starting test: test_check_availability_error")

        # Mock the parsed message to return the expected command and arguments
        mock_parse_user_message.return_value = ["check_availability", "https://invalid-url.com",
"2024-09-30"]

        # Simulate error during availability check
        mock_receive_command.return_value = "No availability found."

        command = self.bot.get_command("check_availability")
        self.assertIsNotNone(command)

        # Call the command without arguments (since GlobalState is mocked)
        await command(self.ctx)

        expected_message = "No availability found."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified error handling during availability check.")


if __name__ == "__main__":
    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))



--- test_!close_browser.py ---
import logging, unittest
from unittest.mock import patch
from test_init import BaseTestSetup, CustomTextTestRunner

"""
File: test_!close_browser.py
Purpose: This file contains unit tests for the !close_browser command in the Discord bot.
The tests validate both successful and error scenarios, ensuring the browser closes properly or
errors are handled gracefully.

Tests:
- Positive: Simulates the !close_browser command and verifies the browser closes correctly.
- Negative: Simulates an error during browser closure and ensures it is handled gracefully.
"""

class TestCloseBrowserCommand(BaseTestSetup):

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')  # Mock the global state
parsing
```

```
    @patch('entity.BrowserEntity.BrowserEntity.close_browser')
    async def test_close_browser_success(self, mock_close_browser, mock_parse_user_message):
        """Test the close_browser command when it succeeds."""
        logging.info("Starting test: test_close_browser_success")

        # Mock the parsed user message
        mock_parse_user_message.return_value = ["close_browser"]

        # Simulate successful browser closure
        mock_close_browser.return_value = "Browser closed."

        # Retrieve the close_browser command from the bot
        command = self.bot.get_command("close_browser")
        self.assertIsNotNone(command)

        # Call the command
        await command(self.ctx)

        # Verify the expected message was sent to the user
        expected_message = "Browser closed."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful browser closure.")

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')  # Mock the global state
parsing
    @patch('entity.BrowserEntity.BrowserEntity.close_browser')
    async def test_close_browser_error(self, mock_close_browser, mock_parse_user_message):
        """Test the close_browser command when it encounters an error."""
        logging.info("Starting test: test_close_browser_error")

        # Mock the parsed user message
        mock_parse_user_message.return_value = ["close_browser"]

        # Simulate a failure during browser closure
        mock_close_browser.side_effect = Exception("Failed to close browser")

        # Retrieve the close_browser command from the bot
        command = self.bot.get_command("close_browser")
        self.assertIsNotNone(command)

        # Call the command
        await command(self.ctx)

        # Verify the correct error message is sent
        self.ctx.send.assert_called_with("Failed to close browser")  # Error message handled
        logging.info("Verified error handling during browser closure.")


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



--- test_!delete_account.py ---
```

```
from unittest.mock import patch
import logging, unittest
from test_init import BaseTestSetup, CustomTextTestRunner


class TestDeleteAccountCommand(BaseTestSetup):
    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('DataObjects.AccountDAO.AccountDAO.delete_account')
    async def test_delete_account_success(self, mock_delete_account, mock_parse_user_message):
        """Test the delete_account command when it succeeds."""
        logging.info("Unit test for delete account starting for positive test:")
        logging.info("Starting test: test_delete_account_success")

        # Mock setup to simulate user input parsing and successful account deletion
        mock_delete_account.return_value = True
        mock_parse_user_message.return_value = ["delete_account", "123"]

        # Triggering the delete account command in the bot
        command = self.bot.get_command("delete_account")
        await command(self.ctx)

        # Checking if the success message was correctly sent to the user
        expected_message = "Account with ID 123 deleted successfully."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful account deletion.")

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('DataObjects.AccountDAO.AccountDAO.delete_account')
    async def test_delete_account_error(self, mock_delete_account, mock_parse_user_message):
        """Test the delete_account command when it encounters an error."""
        logging.info("Unit test for delete account starting for negative test:")
        logging.info("Starting test: test_delete_account_error")

        # Mock setup for testing account deletion failure
        mock_delete_account.return_value = False
        mock_parse_user_message.return_value = ["delete_account", "999"]

        # Executing the delete account command with expected failure
        command = self.bot.get_command("delete_account")
        await command(self.ctx)

        # Checking if the error message was correctly relayed to the user
        expected_message = "Failed to delete account with ID 999."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified error handling during account deletion.")

if __name__ == "__main__":
    # Custom test runner to highlight the test results
    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))



--- test_!fetch_account_by_website.py ---
import logging, unittest
from unittest.mock import patch
from test_init import BaseTestSetup, CustomTextTestRunner


"""
File: test_!fetch_account_by_website.py
Purpose: Unit tests for the !fetch_account_by_website command in the Discord bot.
Tests the retrieval of account details based on website input, handling both found and not found
scenarios.
"""


class TestFetchAccountByWebsiteCommand(BaseTestSetup):


    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('DataObjects.AccountDAO.AccountDAO.fetch_account_by_website')
        async def test_fetch_account_by_website_success(self, mock_fetch_account_by_website,
mock_parse_user_message):
        """Test the fetch_account_by_website command when it succeeds."""
        logging.info("Starting test: test_fetch_account_by_website_success")


         # Mock setup for successful account fetch
        mock_fetch_account_by_website.return_value = ("testuser", "password123")
        mock_parse_user_message.return_value = ["fetch_account_by_website", "example.com"]


        # Command execution
        command = self.bot.get_command("fetch_account_by_website")
        self.assertIsNotNone(command)


        # Expected successful fetch response
        await command(self.ctx)
        expected_message = "testuser", "password123"
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful account fetch.")

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('DataObjects.AccountDAO.AccountDAO.fetch_account_by_website')
    async def test_fetch_account_by_website_error(self, mock_fetch_account_by_website,
mock_parse_user_message):
        """Test the fetch_account_by_website command when it encounters an error."""
        logging.info("Starting test: test_fetch_account_by_website_error")

        # Mock setup for failure in finding account
        mock_fetch_account_by_website.return_value = None
        mock_parse_user_message.return_value = ["fetch_account_by_website", "nonexistent.com"]

        # Command execution for nonexistent account
        command = self.bot.get_command("fetch_account_by_website")
        self.assertIsNotNone(command)

        # Expected error message response
        await command(self.ctx)
        expected_message = "No account found for nonexistent.com."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified error handling for nonexistent account.")

if __name__ == "__main__":
    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))


--- test_!fetch_all_accounts.py ---
import logging, unittest
from unittest.mock import patch
from test_init import BaseTestSetup, CustomTextTestRunner

"""
File: test_!fetch_all_accounts.py
Purpose: Unit tests for the !fetch_all_accounts command in the Discord bot.
The tests validate both successful and error scenarios, ensuring accounts are fetched successfully
or errors are handled properly.
"""


class TestFetchAllAccountsCommand(BaseTestSetup):

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('DataObjects.AccountDAO.AccountDAO.fetch_all_accounts')
            async def test_fetch_all_accounts_success(self, mock_fetch_all_accounts,
mock_parse_user_message):
        """Test the fetch_all_accounts command when it succeeds."""
        logging.info("Starting test: test_fetch_all_accounts_success")

        # Mock the DAO function to simulate database returning account data
        mock_fetch_all_accounts.return_value = [("1", "testuser", "password", "example.com")]
        # Mock the message parsing to simulate command input handling
        mock_parse_user_message.return_value = ["fetch_all_accounts"]
        # Retrieve the command function from the bot commands
        command = self.bot.get_command("fetch_all_accounts")
        # Ensure the command is properly registered and retrieved
        self.assertIsNotNone(command)
        # Execute the command and pass the context object
        await command(self.ctx)

        # Define expected user message output
        expected_message = "Accounts:\nID: 1, Username: testuser, Password: password, Website: example.com"
        # Assert the expected output was sent to the user
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful fetch.")

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('DataObjects.AccountDAO.AccountDAO.fetch_all_accounts')
    async def test_fetch_all_accounts_error(self, mock_fetch_all_accounts, mock_parse_user_message):
        """Test the fetch_all_accounts command when it encounters an error."""
        logging.info("Starting test: test_fetch_all_accounts_error")

        # Mock the DAO function to raise an exception simulating a database error
        mock_fetch_all_accounts.side_effect = Exception("Database error")
        # Mock the message parsing to simulate command input handling
        mock_parse_user_message.return_value = ["fetch_all_accounts"]
        # Retrieve the command function from the bot commands
        command = self.bot.get_command("fetch_all_accounts")
        # Ensure the command is properly registered and retrieved
        self.assertIsNotNone(command)
        # Execute the command and pass the context object
        await command(self.ctx)

        # Assert the correct error message was sent to the user
        self.ctx.send.assert_called_with("Error fetching accounts.")
        logging.info("Verified error handling.")

if __name__ == "__main__":
    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))



--- test_!get_price.py ---
import logging, unittest
from unittest.mock import patch
from test_init import BaseTestSetup, CustomTextTestRunner

"""
File: test_!get_price.py
Purpose: This file contains unit tests for the !get_price command in the Discord bot.
The tests validate both successful and error scenarios, ensuring that the price is fetched correctly or
errors are handled.
"""
```

```
class TestGetPriceCommand(BaseTestSetup):
    @patch('control.PriceControl.PriceControl.receive_command')
    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    async def test_get_price_success(self, mock_parse_message, mock_receive_command):
        """Test the get_price command when it succeeds."""
        logging.info("Starting test: test_get_price_success")

        # Simulate parsing of user input
        mock_parse_message.return_value = ["get_price", "https://example.com"]

        # Simulate successful price fetch
        mock_receive_command.return_value = "Price: $199.99"

        # Retrieve the get_price command from the bot
        command = self.bot.get_command("get_price")
        self.assertIsNotNone(command)

        # Call the command without passing URL (since parsing handles it)
        await command(self.ctx)

        # Verify the expected message was sent to the user
        self.ctx.send.assert_called_with("Price found: Price: $199.99")
        logging.info("Verified successful price fetch.")


    @patch('control.PriceControl.PriceControl.receive_command')
    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    async def test_get_price_error(self, mock_parse_message, mock_receive_command):

        """Test the get_price command when it encounters an error."""

        logging.info("Starting test: test_get_price_error")


        # Simulate parsing of user input

        mock_parse_message.return_value = ["get_price", "https://invalid-url.com"]


        # Simulate a failure during price fetch

        mock_receive_command.return_value = "Failed to fetch price"


        # Retrieve the get_price command from the bot

        command = self.bot.get_command("get_price")

        self.assertIsNotNone(command)


        # Call the command without passing additional URL argument (parsing handles it)

        await command(self.ctx)


        # Verify the correct error message is sent

        self.ctx.send.assert_called_with("Price found: Failed to fetch price")

        logging.info("Verified error handling during price fetch.")


if __name__ == "__main__":

    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))




--- test_!launch_browser.py ---

import logging, unittest
```

```
from unittest.mock import patch
from test_init import BaseTestSetup, CustomTextTestRunner


"""
File: test_!launch_browser.py
Purpose: This file contains unit tests for the !launch_browser command in the Discord bot.
The tests validate both successful and error scenarios, ensuring the browser launches properly or
errors are handled gracefully.
"""


class TestLaunchBrowserCommand(BaseTestSetup):


    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('entity.BrowserEntity.BrowserEntity.launch_browser')
    async def test_launch_browser_success(self, mock_launch_browser,
mock_parse_user_message):
        """Test the launch_browser command when it succeeds."""
        logging.info("Starting test: test_launch_browser_success")


        # Simulate successful browser launch
        mock_launch_browser.return_value = "Browser launched."
        # Mock the parsed message to return the expected command
        mock_parse_user_message.return_value = ["launch_browser"]


        # Retrieve the launch_browser command from the bot
        command = self.bot.get_command("launch_browser")
        self.assertIsNotNone(command)
        # Call the command without arguments (since GlobalState is mocked)
        await command(self.ctx)

        # Verify the expected message was sent to the user
        expected_message = "Browser launched."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful browser launch.")

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('entity.BrowserEntity.BrowserEntity.launch_browser')
    async def test_launch_browser_error(self, mock_launch_browser, mock_parse_user_message):
        """Test the launch_browser command when it encounters an error."""
        logging.info("Starting test: test_launch_browser_error")

        # Simulate a failure during browser launch
        mock_launch_browser.side_effect = Exception("Failed to launch browser")
        # Mock the parsed message to return the expected command
        mock_parse_user_message.return_value = ["launch_browser"]

        # Retrieve the launch_browser command from the bot
        command = self.bot.get_command("launch_browser")
        self.assertIsNotNone(command)

        # Call the command without arguments (since GlobalState is mocked)
        await command(self.ctx)
```

```
        # Verify the correct error message is sent
        self.ctx.send.assert_called_with("Failed to launch browser")  # Error message handled
        logging.info("Verified error handling during browser launch.")


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



--- test_!login.py ---
import logging, unittest
from unittest.mock import patch, AsyncMock
from test_init import BaseTestSetup, CustomTextTestRunner

"""
File: test_!login.py
Purpose: Unit tests for the !login command in the Discord bot.
The tests validate both successful and error scenarios, ensuring the bot correctly logs in to a specified website or handles errors gracefully.

Tests:
- Positive: Simulates the !login command and verifies the login is successful.
- Negative: Simulates an error during login and ensures it is handled gracefully.
"""

class TestLoginCommand(BaseTestSetup):
```

```
@patch('DataObjects.global_vars.GlobalState.parse_user_message')
@patch('control.LoginControl.LoginControl.receive_command')
async def test_login_success(self, mock_receive_command, mock_parse_user_message):
    """Test the login command when it succeeds."""
    logging.info("Starting test: test_login_success")

    # Mock the parsed message to return the expected command and arguments
    mock_parse_user_message.return_value = ["login", "ebay"]

    # Simulate a successful login
    mock_receive_command.return_value = "Login successful."

    # Retrieve the login command from the bot
    command = self.bot.get_command("login")
    self.assertIsNotNone(command)

    # Call the command without arguments (since GlobalState is mocked)
    await command(self.ctx)

    # Verify the expected message was sent to the user
    expected_message = "Login successful."
    self.ctx.send.assert_called_with(expected_message)
    logging.info("Verified successful login.")

@patch('DataObjects.global_vars.GlobalState.parse_user_message')
@patch('control.LoginControl.LoginControl.receive_command')
async def test_login_error(self, mock_receive_command, mock_parse_user_message):
        """Test the login command when it encounters an error."""
        logging.info("Starting test: test_login_error")

        # Mock the parsed message to return the expected command and arguments
        mock_parse_user_message.return_value = ["login", "nonexistent.com"]

        # Simulate a failure during login
        mock_receive_command.return_value = "Failed to login. No account found."

        # Retrieve the login command from the bot
        command = self.bot.get_command("login")
        self.assertIsNotNone(command)

        # Call the command without arguments (since GlobalState is mocked)
        await command(self.ctx)

        # Verify the correct error message is sent
        expected_message = "Failed to login. No account found."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified error handling during login.")

if __name__ == "__main__":
    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))



--- test_!navigate_to_website.py ---
import logging, unittest
```

```
from unittest.mock import patch, AsyncMock
from test_init import BaseTestSetup, CustomTextTestRunner


"""
File: test_!navigate_to_website.py
Purpose: This file contains unit tests for the !navigate_to_website command in the Discord bot.
The tests validate both successful and error scenarios, ensuring the bot navigates to the website
correctly or handles errors.
"""


class TestNavigateToWebsiteCommand(BaseTestSetup):


    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('entity.BrowserEntity.BrowserEntity.navigate_to_website')


            async def test_navigate_to_website_success(self, mock_receive_command,
mock_parse_user_message):
        """Test the navigate_to_website command when it succeeds."""
        logging.info("Starting test: test_navigate_to_website_success")


        # Mock the parsed message to return the expected command and URL
        mock_parse_user_message.return_value = ["navigate_to_website", "https://example.com"]


        # Simulate successful navigation
        mock_receive_command.return_value = "Navigated to https://example.com."


        # Retrieve the navigate_to_website command from the bot
        command = self.bot.get_command("navigate_to_website")
        self.assertIsNotNone(command)

        # Call the command without arguments (since GlobalState is mocked)
        await command(self.ctx)

        # Verify the expected message was sent to the user
        expected_message = "Navigated to https://example.com."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful website navigation.")

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('entity.BrowserEntity.BrowserEntity.navigate_to_website')
    async def test_navigate_to_website_error(self, mock_receive_command,
mock_parse_user_message):
        """Test the navigate_to_website command when it encounters an error."""
        logging.info("Starting test: test_navigate_to_website_error")

        # Mock the parsed message to return the expected command and URL
        mock_parse_user_message.return_value = ["navigate_to_website", "https://invalid-url.com"]

        # Simulate a failure during navigation
        mock_receive_command.side_effect = Exception("Failed to navigate to the website.")

        # Retrieve the navigate_to_website command from the bot
        command = self.bot.get_command("navigate_to_website")
        self.assertIsNotNone(command)
        # Call the command without arguments (since GlobalState is mocked)
        await command(self.ctx)

        # Verify the correct error message is sent
        self.ctx.send.assert_called_with("Failed to navigate to the website.")  # Error message handled
        logging.info("Verified error handling during website navigation.")

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

--- test_!project_help.py ---

```
import logging, unittest
from unittest.mock import patch, AsyncMock, call
from test_init import BaseTestSetup, CustomTextTestRunner

"""
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.
"""
```

```
class TestProjectHelpCommand(BaseTestSetup):

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    async def test_project_help_success(self, mock_parse_user_message):
        """Test the project help command when it successfully returns the help message."""
        logging.info("Starting test: test_project_help_success")
        mock_parse_user_message.return_value = ["project_help"]  # Mock the command parsing to
return the command

        # Simulate calling the project_help command
        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)

        # Define the expected help message from the module
        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"
            "!start_monitoring_availability 'url' 'frequency' - Monitor availability at a specific interval.\n"
            "!stop_monitoring_availability - Stop monitoring availability.\n"
            "!stop_bot - Stop the bot.\n"
        )


        # Check if the correct help message was sent
        self.ctx.send.assert_called_with(help_message)
        logging.info("Verified that the correct help message was sent.")

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    async def test_project_help_error(self, mock_parse_user_message):
        """Test the project help command when it encounters an error during execution."""
        logging.info("Starting test: test_project_help_error")
        mock_parse_user_message.return_value = ["project_help"]  # Mock the command parsing to return the command

        # Simulate an error when sending the message
        self.ctx.send.side_effect = Exception("Error during project_help execution.")
        command = self.bot.get_command("project_help")
            self.assertIsNotNone(command, "project_help command is not registered.")  # Ensure the
command is registered

        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_!start_monitoring_availability.py ---
import logging, unittest
from unittest.mock import patch
from test_init import BaseTestSetup, CustomTextTestRunner

"""
File: test_!monitor_availability.py
Purpose: Unit tests for the !monitor_availability command in the Discord bot.
"""

class TestMonitorAvailabilityCommand(BaseTestSetup):

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('control.AvailabilityControl.AvailabilityControl.receive_command')
    async def test_monitor_availability_success(self, mock_receive_command, mock_parse_user_message):
        """Test the monitor_availability command when it succeeds."""
        logging.info("Starting test: test_monitor_availability_success")

        # Mock the parsed message to return the expected command and arguments
        mock_parse_user_message.return_value = ["start_monitoring_availability", "https://example.com", "2024-09-30", 15]

        # Simulate successful availability monitoring start
        mock_receive_command.return_value = "Monitoring started for https://example.com."

        command = self.bot.get_command("start_monitoring_availability")
        self.assertIsNotNone(command)

        # Call the command without arguments (since GlobalState is mocked)
        await command(self.ctx)

        expected_message = "Monitoring started for https://example.com."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful availability monitoring start.")

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('control.AvailabilityControl.AvailabilityControl.receive_command')
    async def test_monitor_availability_error(self, mock_receive_command, mock_parse_user_message):
        """Test the monitor_availability command when it encounters an error."""
        logging.info("Starting test: test_monitor_availability_error")

        # Mock the parsed message to return the expected command and arguments
        mock_parse_user_message.return_value = ["start_monitoring_availability",
"https://invalid-url.com", "2024-09-30", 15]

        # Simulate an error during availability monitoring
        mock_receive_command.return_value = "Failed to start monitoring."

        command = self.bot.get_command("start_monitoring_availability")
        self.assertIsNotNone(command)

        # Call the command without arguments (since GlobalState is mocked)
        await command(self.ctx)

        expected_message = "Failed to start monitoring."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified error handling during availability monitoring.")

if __name__ == "__main__":
    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))



--- test_!start_monitoring_price.py ---
import logging, unittest
from unittest.mock import patch, AsyncMock
from test_init import BaseTestSetup, CustomTextTestRunner


"""
File: test_!start_monitoring_price.py
Purpose: This file contains unit tests for the !start_monitoring_price command in the Discord bot.
The tests validate both successful and error scenarios, ensuring that the bot starts monitoring prices
or handles errors gracefully.

Tests:
- Positive: Simulates the !start_monitoring_price command and verifies the monitoring is initiated
successfully.
- Negative: Simulates an error during the initiation of price monitoring and ensures it is handled
gracefully.
"""


class TestStartMonitoringPriceCommand(BaseTestSetup):
    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('control.PriceControl.PriceControl.receive_command')
            async def test_start_monitoring_price_success(self, mock_receive_command,
mock_parse_user_message):
        """Test the start_monitoring_price command when it succeeds."""
        logging.info("Starting test: test_start_monitoring_price_success")

        # Mock the parsed message to return the expected command and parameters
          mock_parse_user_message.return_value = ["start_monitoring_price", "https://example.com",
"20"]
```

```
        # Simulate successful price monitoring start
        mock_receive_command.return_value = "Monitoring started for https://example.com."

        # Retrieve the start_monitoring_price command from the bot
        command = self.bot.get_command("start_monitoring_price")
        self.assertIsNotNone(command)

        # Call the command without explicit parameters due to mocked GlobalState
        await command(self.ctx)

        # Verify the expected message was sent to the user
        expected_message = "Monitoring started for https://example.com."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful price monitoring start.")

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('control.PriceControl.PriceControl.receive_command')
    async def test_start_monitoring_price_error(self, mock_receive_command,
mock_parse_user_message):
        """Test the start_monitoring_price command when it encounters an error."""
        logging.info("Starting test: test_start_monitoring_price_error")

        # Mock the parsed message to simulate the command being executed with an invalid URL
        mock_parse_user_message.return_value = ["start_monitoring_price", "https://invalid-url.com",
"20"]

        # Simulate a failure during price monitoring start
        mock_receive_command.return_value = "Failed to start monitoring"

        # Retrieve the start_monitoring_price command from the bot
        command = self.bot.get_command("start_monitoring_price")
        self.assertIsNotNone(command)

        # Call the command without explicit parameters due to mocked GlobalState
        await command(self.ctx)

        # Verify the correct error message is sent
        expected_message = "Failed to start monitoring"
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified error handling during price monitoring start.")

if __name__ == "__main__":
    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))


--- test_!stop_bot.py ---
import logging, unittest
from unittest.mock import AsyncMock, patch
from test_init import BaseTestSetup, CustomTextTestRunner

"""
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 TestStopBotCommand(BaseTestSetup):


    @patch('DataObjects.global_vars.GlobalState.parse_user_message')

    @patch('control.StopControl.StopControl.receive_command', new_callable=AsyncMock)

    async def test_stop_bot_success(self, mock_receive_command, mock_parse_user_message):

        """Test the stop bot command when it successfully shuts down."""

        logging.info("Starting test: test_stop_bot_success")


        # Setup mocks

        mock_receive_command.return_value = "The bot is shutting down..."

        mock_parse_user_message.return_value = ["stop_bot"]


        # Simulate calling the stop_bot command

        command = self.bot.get_command("stop_bot")

        self.assertIsNotNone(command, "stop_bot command is not registered.")

        await command(self.ctx)


        # Verify the message was sent before shutdown is initiated

        self.ctx.send.assert_called_once_with("Command recognized, passing data to control.")

        logging.info("Verified that the shutdown message was sent to the user.")
        # Ensure bot.close() is called
        mock_receive_command.assert_called_once()
        logging.info("Verified that the bot's close method was called once.")

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('control.StopControl.StopControl.receive_command', new_callable=AsyncMock)
    async def test_stop_bot_error(self, mock_receive_command, mock_parse_user_message):
        """Test the stop bot command when it encounters an error during shutdown."""
        logging.info("Starting test: test_stop_bot_error")

        # Setup mocks
        exception_message = "Error stopping bot"
        mock_receive_command.side_effect = Exception(exception_message)
        mock_parse_user_message.return_value = ["stop_bot"]

        # Simulate calling the stop_bot command
        command = self.bot.get_command("stop_bot")
        self.assertIsNotNone(command, "stop_bot command is not registered.")

        with self.assertRaises(Exception) as context:
            await command(self.ctx)

        # Verify that the correct error message is sent
        self.ctx.send.assert_called_with('Command recognized, passing data to control.')
        self.assertTrue(exception_message in str(context.exception))
        logging.info("Verified error handling during bot shutdown.")
```

```
        # Verify that the close method was still attempted
        mock_receive_command.assert_called_once_with("stop_bot", self.ctx)
        logging.info("Verified that the bot's close method was attempted 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))



--- test_!stop_monitoring_availability.py ---
import logging, unittest
from unittest.mock import patch
from test_init import BaseTestSetup, CustomTextTestRunner


"""
File: test_!stop_monitoring_availability.py
Purpose: Unit tests for the !stop_monitoring_availability command in the Discord bot.
"""


class TestStopMonitoringAvailabilityCommand(BaseTestSetup):


    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('control.AvailabilityControl.AvailabilityControl.receive_command')
    async def test_stop_monitoring_availability_no_active_session(self, mock_receive_command,
mock_parse_user_message):
        """Test the stop_monitoring_availability command when no active session exists."""
        logging.info("Starting test: test_stop_monitoring_availability_no_active_session")

        # Mock the parsed message to return the expected command and arguments
        mock_parse_user_message.return_value = ["stop_monitoring_availability"]

        # Simulate no active session scenario
        mock_receive_command.return_value = "There was no active availability monitoring session."

        command = self.bot.get_command("stop_monitoring_availability")
        self.assertIsNotNone(command)

        # Call the command without arguments (since GlobalState is mocked)
        await command(self.ctx)

        expected_message = "There was no active availability monitoring session."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified no active session stop scenario.")

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('control.AvailabilityControl.AvailabilityControl.receive_command')
        async def test_stop_monitoring_availability_success(self, mock_receive_command,
mock_parse_user_message):
        """Test the stop_monitoring_availability command when it succeeds."""
        logging.info("Starting test: test_stop_monitoring_availability_success")

        # Mock the parsed message to return the expected command and arguments
        mock_parse_user_message.return_value = ["stop_monitoring_availability"]
        # Simulate successful stopping of monitoring
        mock_receive_command.return_value = "Availability monitoring stopped successfully."

        command = self.bot.get_command("stop_monitoring_availability")
        self.assertIsNotNone(command)

        # Call the command without arguments (since GlobalState is mocked)
        await command(self.ctx)

        expected_message = "Availability monitoring stopped successfully."
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful availability monitoring stop.")

if __name__ == "__main__":
    unittest.main(testRunner=CustomTextTestRunner(verbosity=2))
```

--- test_!stop_monitoring_price.py ---

```
import logging, unittest
from unittest.mock import patch, AsyncMock
from test_init import BaseTestSetup, CustomTextTestRunner

"""
File: test_!stop_monitoring_price.py
Purpose: This file contains unit tests for the !stop_monitoring_price command in the Discord bot.
The tests validate both successful and error scenarios, ensuring that the bot stops monitoring prices
```

```
or handles errors gracefully.
"""


class TestStopMonitoringPriceCommand(BaseTestSetup):


    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('control.PriceControl.PriceControl.receive_command')
        async def test_stop_monitoring_price_success_with_results(self, mock_receive_command,
mock_parse_user_message):
            """Test the stop_monitoring_price command when monitoring was active and results are
returned."""
        logging.info("Starting test: test_stop_monitoring_price_success_with_results")

        # Simulate stopping monitoring and receiving results
        mock_parse_user_message.return_value = ["stop_monitoring_price"]
          mock_receive_command.return_value = "Results for price monitoring:\nPrice: $199.99\nPrice
monitoring stopped successfully!"

        # Retrieve the stop_monitoring_price command from the bot
        command = self.bot.get_command("stop_monitoring_price")
        self.assertIsNotNone(command)

        # Call the command
        await command(self.ctx)

        # Verify the expected message was sent to the user
         expected_message = "Results for price monitoring:\nPrice: $199.99\nPrice monitoring stopped
```

```
successfully!"
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified successful stop with results.")

    @patch('DataObjects.global_vars.GlobalState.parse_user_message')
    @patch('control.PriceControl.PriceControl.receive_command')
    async def test_stop_monitoring_price_error(self, mock_receive_command,
mock_parse_user_message):
        """Test the stop_monitoring_price command when it encounters an error."""
        logging.info("Starting test: test_stop_monitoring_price_error")

        # Simulate a failure during price monitoring stop
        mock_parse_user_message.return_value = ["stop_monitoring_price"]
        mock_receive_command.return_value = "Error stopping price monitoring"

        # Retrieve the stop_monitoring_price command from the bot
        command = self.bot.get_command("stop_monitoring_price")
        self.assertIsNotNone(command)

        # Call the command
        await command(self.ctx)

        # Verify the correct error message is sent
        expected_message = "Error stopping price monitoring"
        self.ctx.send.assert_called_with(expected_message)
        logging.info("Verified error handling during price monitoring stop.")
```

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



--- test_init.py ---
# Purpose: This file contains common setup code for all test cases.
import sys, os, discord, logging, unittest
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from unittest.mock import AsyncMock
from utils.MyBot import MyBot

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

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 BaseTestSetup(unittest.IsolatedAsyncioTestCase):
    """Base setup class for initializing bot and mock context for all tests."""


    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()
        intents.message_content = True
        self.bot = MyBot(command_prefix="!", intents=intents)
        self.ctx = AsyncMock()
        self.ctx.send = AsyncMock()
        self.ctx.bot = self.bot  # Mock the bot property in the context
        await self.bot.setup_hook()  # Ensure commands are registered



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


--- BCE_test_close_browser.py ---
from test_init import BaseTestCase, patch, logging, unittest


class TestBrowserFunctionality(BaseTestCase):


    @patch('entity.BrowserEntity.BrowserEntity.close_browser')
    def test_close_browser_success(self, mock_close):
        """Test successful browser close."""
        print("\nTest Started for: test_close_browser_success")
        mock_close.return_value = "Browser closed."
        expected_entity_result = "Browser closed."
        expected_control_result = "Control Object Result: Browser closed."
        result = self.control.receive_command("close_browser")

        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_close.return_value}")
            self.assertEqual(mock_close.return_value, expected_entity_result, "Entity layer assertion
failed.")
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        self.assertEqual(result, expected_control_result, "Control layer assertion failed.")
        logging.info("Unit Test Passed for control layer.\n")

    @patch('entity.BrowserEntity.BrowserEntity.close_browser')
    def test_close_browser_not_open(self, mock_close):
        """Test closing a browser that is not open."""
        print("\nTest Started for: test_close_browser_not_open")
        mock_close.return_value = "No browser is currently open."
        expected_entity_result = "No browser is currently open."
        expected_control_result = "Control Object Result: No browser is currently open."
        result = self.control.receive_command("close_browser")

        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_close.return_value}")
            self.assertEqual(mock_close.return_value, expected_entity_result, "Entity layer assertion
failed.")
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        self.assertEqual(result, expected_control_result, "Control layer assertion failed.")
        logging.info("Unit Test Passed for control layer.\n")

    @patch('entity.BrowserEntity.BrowserEntity.close_browser')
    def test_close_browser_failure(self, mock_close):
        """Test control layer's handling of an unexpected error during browser close."""
        print("\nTest Started for: test_close_browser_failure")
        mock_close.side_effect = Exception("Unexpected error")
        expected_result = "Control Layer Exception: Unexpected error"
        result = self.control.receive_command("close_browser")

        logging.info(f"Control Layer Expected to Report: {expected_result}")
        logging.info(f"Control Layer Received: {result}")
            self.assertEqual(result, expected_result, "Control layer failed to handle or report the error
correctly.")
        logging.info("Unit Test Passed for control layer error handling.\n")


    @patch('entity.BrowserEntity.BrowserEntity.close_browser')
    def test_close_browser_failure_entity(self, mock_close):
        """Test failure to close the browser due to an internal error in the entity layer."""
        print("\nTest Started for: test_close_browser_failure_entity")
        internal_error_message = "BrowserEntity_Failed to close browser: Internal error"
        mock_close.side_effect = Exception(internal_error_message)  # Simulate an exception on error
        expected_control_result = f"Control Layer Exception: {internal_error_message}"

        # Execute command
        result = self.control.receive_command("close_browser")

        # Check if the control layer returns the correct error message
        logging.info(f"Entity Layer Expected Failure: {internal_error_message}")
        logging.info(f"Control Layer Received: {result}")
        self.assertEqual(result, expected_control_result, "Control layer failed to report entity error correctly.")
        logging.info("Unit Test Passed for entity layer error handling.\n")


if __name__ == '__main__':
    unittest.main()


--- BCE_test_launch_browser.py ---
from test_init import BaseTestCase, patch, logging, unittest

class TestBrowserFunctionality(BaseTestCase):

    @patch('entity.BrowserEntity.BrowserEntity.launch_browser')
    def test_launch_browser_success(self, mock_launch):
        """Test successful browser launch."""
        print("\nTest Started for: test_launch_browser_success")
        mock_launch.return_value = "Browser launched."
        expected_entity_result = "Browser launched."
        expected_control_result = "Control Object Result: Browser launched."
        result = self.control.receive_command("launch_browser")

        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_launch.return_value}")
            self.assertEqual(mock_launch.return_value, expected_entity_result, "Entity layer assertion
failed.")
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        self.assertEqual(result, expected_control_result, "Control layer assertion failed.")
        logging.info("Unit Test Passed for control layer.\n")

    @patch('entity.BrowserEntity.BrowserEntity.launch_browser')
    def test_launch_browser_already_running(self, mock_launch):
        """Test launch browser when already running."""
        print("\nTest Started for: test_launch_browser_already_running")
        mock_launch.return_value = "Browser is already running."
        expected_entity_result = "Browser is already running."
        expected_control_result = "Control Object Result: Browser is already running."
        result = self.control.receive_command("launch_browser")

        logging.info(f"Entity Layer Expected: {expected_entity_result}")
        logging.info(f"Entity Layer Received: {mock_launch.return_value}")
        self.assertEqual(mock_launch.return_value, expected_entity_result, "Entity layer assertion failed.")
        logging.info("Unit Test Passed for entity layer.\n")

        logging.info(f"Control Layer Expected: {expected_control_result}")
        logging.info(f"Control Layer Received: {result}")
        self.assertEqual(result, expected_control_result, "Control layer assertion failed.")
        logging.info("Unit Test Passed for control layer.\n")

    @patch('entity.BrowserEntity.BrowserEntity.launch_browser')
    def test_launch_browser_failure_control(self, mock_launch):
        """Test control layer's handling of the entity layer failure."""
        print("\nTest Started for: test_launch_browser_failure_control")
        mock_launch.side_effect = Exception("Internal error")
        expected_result = "Control Layer Exception: Internal error"
        result = self.control.receive_command("launch_browser")

        logging.info(f"Control Layer Expected to Report: {expected_result}")
        logging.info(f"Control Layer Received: {result}")
        self.assertEqual(result, expected_result, "Control layer failed to handle or report the entity error correctly.")
        logging.info("Unit Test Passed for control layer error handling.\n")

    @patch('entity.BrowserEntity.BrowserEntity.launch_browser')
    def test_launch_browser_failure_entity(self, mock_launch):
        """Test failure to launch browser due to an internal error in the entity layer."""
        print("\nTest Started for: test_launch_browser_failure_entity")
        internal_error_message = "Failed to launch browser: Internal error"
        mock_launch.side_effect = Exception(internal_error_message)  # Simulate an exception on error
        expected_control_result = f"Control Layer Exception: {internal_error_message}"

        # Execute command
        result = self.control.receive_command("launch_browser")

        # Check if the control layer returns the correct error message
        logging.info(f"Entity Layer Expected Failure: {internal_error_message}")
        logging.info(f"Control Layer Received: {result}")
        self.assertEqual(result, expected_control_result, "Control layer failed to report entity error correctly.")
        logging.info("Unit Test Passed for entity layer error handling.\n")


if __name__ == '__main__':
    unittest.main()


--- temporary.py ---
import unittest
from unittest.mock import patch, AsyncMock
import logging
import sys, os, discord, logging, unittest
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
```

```
# Setup logging

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


# Import your classes

from control.BrowserControl import BrowserControl


class TestBrowserFunctionality(unittest.TestCase):


    def setUp(self):

        """Set up BrowserControl and context for each test."""

        self.control = BrowserControl()

        self.ctx = AsyncMock()  # Mocking the context to use in the control object


    @patch('entity.BrowserEntity.BrowserEntity.launch_browser')

    def test_launch_browser_failure_entity(self, mock_launch):

        """Test failure to launch browser due to an internal error in the entity layer."""

        internal_error_message = "Failed to launch browser: Internal error"

        mock_launch.side_effect = Exception(internal_error_message)  # Simulate an exception on
error

        expected_control_result = f"Control Layer Exception: {internal_error_message}"


        # Execute command

        result = self.control.receive_command("launch_browser")


        # Check if the control layer returns the correct error message

        logging.info(f"Entity Layer Expected Failure: {internal_error_message}")
        logging.info(f"Control Layer Received: {result}")
            self.assertEqual(result, expected_control_result, "Control layer failed to report entity error correctly.")
        logging.info("Unit Test Passed for entity layer error handling.")


if __name__ == '__main__':
    unittest.main()



--- test_init.py ---
# test_init.py
import sys
import os
import unittest
from unittest.mock import patch, AsyncMock
import logging

# Ensure all necessary paths are included for modules that tests need to access
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
# Setting up logging without timestamp
logging.basicConfig(level=logging.INFO, format='%(levelname)s - %(message)s')

# Import your BrowserControl class and any other common classes
from control.BrowserControl import BrowserControl

class BaseTestCase(unittest.TestCase):
    """Base test class that can be extended by other test modules."""
    def setUp(self):
        """Set up BrowserControl and context for each test."""
        self.control = BrowserControl()
        self.ctx = AsyncMock()  # Mocking the context to use in the control object


--- Config.py ---
#ignored not pushed to git!
class Config:
                                                                    DISCORD_TOKEN              =
'MTI2OTM4MTE4OTA1NjMzNTk3Mw.GJdUct.-2RsoynZh78VFGdoXdrXWFhFQPbUCHM7V2w-u8'
    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-xb
ox-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",
            "availableUrl2": "https://www.opentable.com/r/hals-the-steakhouse-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
from DataObjects.global_vars import GlobalState  # Import the global variable


# Bot initialization

intents = discord.Intents.default()

intents.message_content = True  # Enable reading message content


class MyBot(commands.Bot):


    def __init__(self, *args, **kwargs):

        super().__init__(*args, **kwargs)


    async def on_message(self, message):

        if message.author == self.user:  # Prevent the bot from replying to its own messages

            return


        print(f"Message received: {message.content}")

        GlobalState.user_message = message.content


        if GlobalState.user_message.lower() in ["hi", "hey", "hello"]:

            await message.channel.send("Hi, how can I help you?")


        elif GlobalState.user_message.startswith("!"):

            print("User message: ", GlobalState.user_message)


        else:

                await message.channel.send("I'm sorry, I didn't understand that. Type !project_help to see
the list of commands.")
```

```
        await self.process_commands(message)
        GlobalState.reset_user_message()  # Reset the global user_message variable
        #print("User_message reset to empty string")

    async def setup_hook(self):
        await self.add_cog(BrowserBoundary())  # Add your boundary objects
        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):
            print("Command not recognized:")
            print(error)
            await ctx.channel.send("I'm sorry, I didn't understand that. Type !project_help to see the list
of commands.")


# Initialize the bot instance

bot = MyBot(command_prefix="!", intents=intents, case_insensitive=True)


def start_bot(token):

    """Run the bot with the provided token."""

    bot.run(token)
```