--- 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)

--- BotBoundary.py ---

from discord.ext import commands  
from control.BotControl import BotControl  
from DataObjects.global\_vars import GlobalState  
  
class BotBoundary(commands.Cog):  
 def \_\_init\_\_(self):  
 self.bot\_control = BotControl() # Initialize control object  
  
 @commands.command(name="project\_help")  
 async def project\_help(self, ctx):  
 """Handle help command by sending available commands to the user."""  
 await ctx.send("Command recognized, passing data to control.")  
 try:  
 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.bot\_control.receive\_command(command) # Call control layer  
 await ctx.send(response) # Send the response back to the user  
 except Exception as e:  
 error\_msg = f"Error in HelpBoundary: {str(e)}"  
 print(error\_msg)  
 await ctx.send(error\_msg)  
  
 @commands.command(name="stop\_bot")  
 async def stop\_bot(self, ctx):  
 """Handle stop bot command by shutting down the bot."""  
 await ctx.send("Command recognized, passing data to control.")  
 try:  
 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.bot\_control.receive\_command(command, ctx) # Call control layer to stop the bot  
 print(result) # Send the result to the terminal since the bot will shut down  
 except Exception as e:  
 error\_msg = f"Error in StopBoundary: {str(e)}"  
 print(error\_msg)  
 await ctx.send(error\_msg)  
  
   
 @commands.command(name="receive\_email")  
 async def receive\_email(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  
 file\_name = list[1] # Second element is the fileName  
  
 result = await self.bot\_control.receive\_command(command, file\_name) # Pass the command to the control layer  
 await ctx.send(result)

--- 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 Browser control object  
  
 # Browser-related commands  
 @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 = await 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 close\_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 = await self.browser\_control.receive\_command(command)  
 await ctx.send(result)  
  
 # Login-related commands  
 @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.browser\_control.receive\_command(command, website) # Pass the command and website to control object  
   
 # Send the result back to the user  
 await ctx.send(result)  
  
 # Navigation-related commands  
 @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 = await self.browser\_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)

--- \_\_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  
from utils.exportUtils import ExportUtils  
from utils.configuration import load\_config  
from utils.email\_utils import send\_email\_with\_attachments  
  
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":  
 config = load\_config()  
 availability\_monitor\_frequency = config.get('project\_options', {}).get('availability\_monitor\_frequency', 15)  
  
 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 availability\_monitor\_frequency  
 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)  
  
 try:  
 # Call the Excel export method from ExportUtils  
 excelResult = ExportUtils.log\_to\_excel(  
 command="check\_availability",  
 url=url,  
 result=result,  
 entered\_date=datetime.now().strftime('%Y-%m-%d'), # Pass the optional entered\_date  
 entered\_time=datetime.now().strftime('%H:%M:%S') # Pass the optional entered\_time  
 )  
 print(excelResult)  
 htmlResult = ExportUtils.export\_to\_html(  
 command="check\_availability",  
 url=url,  
 result=result,  
 entered\_date=datetime.now().strftime('%Y-%m-%d'), # Pass the optional entered\_date  
 entered\_time=datetime.now().strftime('%H:%M:%S') # Pass the optional entered\_time  
 )  
 print(htmlResult)  
  
 except Exception as e:  
 return f"AvailabilityControl\_Error exporting data: {str(e)}"   
 return result, excelResult, htmlResult  
  
  
 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  
 send\_email\_with\_attachments("check\_availability.html")  
 send\_email\_with\_attachments("check\_availability.xlsx")  
 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" + "\nMonitoring stopped successfully!"  
 print(result)  
 except Exception as e:  
 # Handle any error that occurs  
 result = f"Error stopping availability monitoring: {str(e)}"  
   
 return result

--- BotControl.py ---

import discord  
from utils.email\_utils import send\_email\_with\_attachments  
  
class BotControl:  
 async def receive\_command(self, command\_data, \*args):  
 """Handle commands related to help and stopping the bot."""  
 print("Data received from boundary:", command\_data)  
  
 # Handle help commands  
 if command\_data == "project\_help":  
 try:  
 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  
 except Exception as e:  
 error\_msg = f"Error handling help command: {str(e)}"  
 print(error\_msg)  
 return error\_msg  
  
 # Handle stop bot commands  
 elif command\_data == "stop\_bot":  
 try:  
 ctx = args[0] if args else None  
 bot = ctx.bot # Get 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  
 except Exception as e:  
 error\_msg = f"Error shutting down the bot: {str(e)}"  
 print(error\_msg)  
 return error\_msg  
  
  
 # Handle receive email commands  
 elif command\_data == "receive\_email":  
 try:  
 file\_name = args[0] if args else None  
 if file\_name:  
 print(f"Sending email with the file '{file\_name}'...")  
 result = send\_email\_with\_attachments(file\_name)  
 print(result)  
 else:  
 result = "Please specify a file to send, e.g., !receive\_email monitor\_price.html"  
 return result  
 except Exception as e:  
 error\_msg = f"Error shutting down the bot: {str(e)}"  
 print(error\_msg)  
 return error\_msg  
  
  
 # Default response for invalid commands  
 else:  
 try:  
 return "Invalid command."  
 except Exception as e:  
 error\_msg = f"Error handling invalid command: {str(e)}"  
 print(error\_msg)  
 return error\_msg

--- BrowserControl.py ---

from entity.BrowserEntity import BrowserEntity  
from control.AccountControl import AccountControl # Needed for LoginControl  
from utils.css\_selectors import Selectors # Used in both LoginControl and NavigationControl  
import re # Used for URL pattern matching in LoginControl  
  
class BrowserControl:  
 def \_\_init\_\_(self):  
 self.browser\_entity = BrowserEntity() # Initialize the entity object inside the control layer  
 self.account\_control = AccountControl() # Manages account data for login use case  
  
 # Browser-related command handler  
 async def receive\_command(self, command\_data, site=None, url=None):  
 print("Data Received from boundary object: ", command\_data)  
   
 # Handle browser commands  
 if command\_data == "launch\_browser":  
 try:  
 result = self.browser\_entity.launch\_browser()  
 return f"Control Object Result: {result}"  
 except Exception as e:  
 return f"Control Layer Exception: {str(e)}"  
   
 elif command\_data == "close\_browser":  
 try:  
 result = self.browser\_entity.close\_browser()  
 return f"Control Object Result: {result}"  
 except Exception as e:  
 return f"Control Layer Exception: {str(e)}"  
  
 # Handle login commands  
 elif command\_data == "login" and site:  
 try:  
 # Fetch account credentials from the account control  
 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}")  
  
 # Improved regex to detect URLs even without http/https  
 url\_pattern = re.compile(r'(https?://)?(www\.)?(\w+)(\.\w{2,})')  
  
 # Check if the input is a full URL or a site name  
 if url\_pattern.search(site):  
 # If it contains a valid domain pattern, treat it as a URL  
 if not site.startswith('http'):  
 # Add 'https://' if the URL does not include a protocol  
 url = f"https://{site}"  
 else:  
 url = site  
 print(f"Using provided URL: {url}")  
 else:  
 # If not a URL, look it up in the CSS selectors  
 selectors = Selectors.get\_selectors\_for\_url(site)  
 if not selectors or 'url' not in selectors:  
 return f"URL for {site} not found."  
 url = selectors.get('url')  
 print(f"URL from selectors: {url}")  
  
 if not url:  
 return f"URL for {site} not found."  
  
 result = await self.browser\_entity.login(url, username, password)  
 return f"Control Object Result: {result}"  
 except Exception as e:  
 return f"Control Layer Exception: {str(e)}"  
   
 # Handle navigation commands  
 elif command\_data == "navigate\_to\_website" and site:  
 url\_pattern = re.compile(r'(https?://)?(www\.)?(\w+)(\.\w{2,})')  
  
 # Check if the input is a full URL or a site name  
 if url\_pattern.search(site):  
 # If it contains a valid domain pattern, treat it as a URL  
 if not site.startswith('http'):  
 # Add 'https://' if the URL does not include a protocol  
 url = f"https://{site}"  
 else:  
 url = site  
 print(f"Using provided URL: {url}")  
 else:  
 # If not a URL, look it up in the CSS selectors  
 selectors = Selectors.get\_selectors\_for\_url(site)  
 if not selectors or 'url' not in selectors:  
 return f"URL for {site} not found."  
 url = selectors.get('url')  
   
 print("URL not provided, default URL for Google is: " + url)  
  
 try:  
 result = self.browser\_entity.navigate\_to\_website(url)  
 return f"Control Object Result: {result}"  
 except Exception as e:  
 return f"Control Layer Exception: {str(e)}"  
  
 else:  
 return "Invalid command."

--- PriceControl.py ---

import asyncio  
from datetime import datetime  
from entity.PriceEntity import PriceEntity  
from utils.configuration import load\_config  
from utils.css\_selectors import Selectors  
from utils.exportUtils import ExportUtils  
from utils.email\_utils import send\_email\_with\_attachments  
  
  
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":  
 config = load\_config()  
 price\_monitor\_frequency = config.get('project\_options', {}).get('price\_monitor\_frequency', 15)  
 url = args[0] if args else None  
 frequency = args[1] if len(args) > 1 and args[1] not in [None, ""] else price\_monitor\_frequency  
 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}")  
 except Exception as e:  
 return f"Failed to fetch price: {str(e)}"  
   
 try:  
 # Call the Excel export method from ExportUtils  
 excelResult = ExportUtils.log\_to\_excel(  
 command="get\_price",  
 url=url,  
 result=result,  
 entered\_date=datetime.now().strftime('%Y-%m-%d'), # Pass the optional entered\_date  
 entered\_time=datetime.now().strftime('%H:%M:%S') # Pass the optional entered\_time  
 )  
 print(excelResult)  
 htmlResult = ExportUtils.export\_to\_html(  
 command="get\_price",  
 url=url,  
 result=result,  
 entered\_date=datetime.now().strftime('%Y-%m-%d'), # Pass the optional entered\_date  
 entered\_time=datetime.now().strftime('%H:%M:%S') # Pass the optional entered\_time  
 )  
 print(htmlResult)  
  
 except Exception as e:  
 return f"PriceControl\_Error exporting data: {str(e)}"   
   
 return result, excelResult, htmlResult  
  
  
 async def start\_monitoring\_price(self, url: str, frequency=10):  
 """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})"  
 send\_email\_with\_attachments("get\_price.html")  
 send\_email\_with\_attachments("check\_availability.xlsx")  
 elif current\_price < previous\_price:  
 result = f"Price went down! Current price: {current\_price} (Previous: {previous\_price})"  
 send\_email\_with\_attachments("get\_price.html")  
 send\_email\_with\_attachments("check\_availability.xlsx")  
 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

--- \_\_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 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  
from utils.configuration import load\_config  
  
class AvailabilityEntity:  
  
 config = load\_config()  
 search\_element\_timeOut = config.get('project\_options', {}).get('search\_element\_timeOut', 15)  
 sleep\_time = config.get('project\_options', {}).get('sleep\_time', 15)  
  
 def \_\_init\_\_(self):  
 self.browser\_entity = BrowserEntity()  
  
  
 async def check\_availability(self, url: str, date\_str=None, timeout=search\_element\_timeOut):  
 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(self.sleep\_time) # 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(self.sleep\_time)  
 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(self.sleep\_time) # 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)}"

--- 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.configuration import load\_config  
from utils.css\_selectors import Selectors  
  
  
class BrowserEntity:  
 \_instance = None  
 config = load\_config()  
 search\_element\_timeOut = config.get('project\_options', {}).get('search\_element\_timeOut', 15)  
 sleep\_time = config.get('project\_options', {}).get('sleep\_time', 15)  
   
 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(self.sleep\_time)  
  
 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(self.sleep\_time)  
  
 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(self.sleep\_time)  
  
 WebDriverWait(self.driver, self.search\_element\_timeOut).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.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)}"

--- \_\_init\_\_.py ---

#empty init file

--- configuration.py ---

import json  
  
#class configuration:  
def load\_config():  
 """Loads the configuration file and returns the settings."""  
 try:  
 with open('config.json', 'r') as config\_file:  
 config\_data = json.load(config\_file)  
 return config\_data  
 except FileNotFoundError:  
 print("Configuration file not found. Using default settings.")  
 return {}  
 except json.JSONDecodeError:  
 print("Error decoding JSON. Please check the format of your config.json file.")  
 return {}

--- css\_selectors.py ---

class Selectors:  
 SELECTORS = {  
 "google": {  
 "url": "https://www.google.com/"   
 },  
 "ebay": {  
 "url": "https://signin.ebay.com/signin/",  
 "email\_field": "#userid",  
 "continue\_button": "[data-testid\*='signin-continue-btn']",  
 "password\_field": "#pass",  
 "login\_button": "#sgnBt",  
 "price": ".x-price-primary span" # CSS selector for Ebay price  
 },  
 "bestbuy": {  
 "priceUrl": "https://www.bestbuy.com/site/microsoft-xbox-wireless-controller-for-xbox-series-x-xbox-series-s-xbox-one-windows-devices-sky-cipher-special-edition/6584960.p?skuId=6584960",  
 "url": "https://www.bestbuy.com/signin/",  
 "email\_field": "#fld-e",  
 #"continue\_button": ".cia-form\_\_controls button",  
 "password\_field": "#fld-p1",  
 "SignIn\_button": ".cia-form\_\_controls button",  
 "price": "[data-testid='customer-price'] span", # CSS selector for BestBuy price  
 "homePage": ".v-p-right-xxs.line-clamp"  
 },  
 "opentable": {  
 "url": "https://www.opentable.com/",  
 "unavailableUrl": "https://www.opentable.com/r/bar-spero-washington/",  
 "availableUrl": "https://www.opentable.com/r/the-rux-nashville",  
 "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

--- email\_utils.py ---

# email\_utils.py  
import smtplib  
from email.mime.multipart import MIMEMultipart  
from email.mime.text import MIMEText  
from email.mime.base import MIMEBase  
from email import encoders  
import os  
from utils.Config import Config  
  
def send\_email\_with\_attachments(file\_name=None):  
 try:  
 # Setup the MIME  
 msg = MIMEMultipart()  
 msg['From'] = Config.EMAIL\_USER  
 msg['To'] = Config.EMAIL\_RECEIVER  
 msg['Subject'] = "Exported Files from Discord Bot"  
   
 # Body of the email  
 body = "Attached is the exported file you requested."  
 msg.attach(MIMEText(body, 'plain'))  
  
 # Check if a specific file was requested  
 if file\_name:  
 file\_path = None  
 # Search in both directories  
 for folder in ['excelFiles', 'htmlFiles']:  
 possible\_path = os.path.join('./ExportedFiles', folder, file\_name)  
 if os.path.exists(possible\_path):  
 file\_path = possible\_path  
 break  
  
 if not file\_path:  
 return f"File '{file\_name}' not found in either excelFiles or htmlFiles."  
  
 # Attach the requested file  
 attachment = open(file\_path, "rb")  
 part = MIMEBase('application', 'octet-stream')  
 part.set\_payload(attachment.read())  
 encoders.encode\_base64(part)  
 part.add\_header('Content-Disposition', f"attachment; filename= {file\_name}")  
 msg.attach(part)  
 attachment.close()  
 else:  
 return "Please specify a file to send."  
  
 # Send the email  
 server = smtplib.SMTP(Config.EMAIL\_HOST, Config.EMAIL\_PORT)  
 server.starttls()  
 server.login(Config.EMAIL\_USER, Config.EMAIL\_PASSWORD)  
 text = msg.as\_string()  
 server.sendmail(Config.EMAIL\_USER, Config.EMAIL\_RECEIVER, text)  
 server.quit()  
  
 return f"Email with file '{file\_name}' sent successfully!"  
 except Exception as e:  
 return f"Failed to send email: {str(e)}"

--- 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.AccountBoundary import AccountBoundary  
from boundary.AvailabilityBoundary import AvailabilityBoundary  
from boundary.PriceBoundary import PriceBoundary  
from boundary.BotBoundary import BotBoundary  
from DataObjects.global\_vars import GlobalState  
  
# 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(AccountBoundary())  
 await self.add\_cog(AvailabilityBoundary())  
 await self.add\_cog(PriceBoundary())  
 await self.add\_cog(BotBoundary())  
  
 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)