--- 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

--- 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]  
 userName = list[2]  
 password = list[3]  
  
 result = await self.browser\_control.receive\_command(command, website, userName, password) # 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

--- AvailabilityControl.py ---

import asyncio  
from entity.AvailabilityEntity import AvailabilityEntity  
from datetime import datetime  
from utils.css\_selectors import Selectors  
from entity.DataExportEntity import ExportUtils  
from utils.configuration import load\_config  
from entity.EmailEntity 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 entity.EmailEntity 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 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  
  
 # Browser-related command handler  
 async def receive\_command(self, command\_data, \*args):  
 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":  
 try:  
 site = args[0]  
 username = args[1]  
 password = args[2]  
 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 entity.DataExportEntity import ExportUtils  
from entity.EmailEntity 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})"  
 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  
  
 send\_email\_with\_attachments("get\_price.html")  
 send\_email\_with\_attachments("check\_availability.xlsx")  
 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

--- 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', 3)  
   
 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)}"

--- DataExportEntity.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}."

--- EmailEntity.py ---

# email\_utils.py  
import smtplib, os  
from email.mime.multipart import MIMEMultipart  
from email.mime.text import MIMEText  
from email.mime.base import MIMEBase  
from email import encoders  
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)}"

--- 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

--- test\_init.py ---

"""  
test\_init.py  
The primary objective is to consolidate all necessary imports in one place.   
We avoid the redundancy of importing modules and dependencies repeatedly   
in each test file. This helps streamline the test setup, making the individual   
test files cleaner and easier to maintain, as they can focus purely on the logic   
being tested rather than handling multiple import statements. This approach also   
helps ensure consistency across all tests by having a single source for the   
required libraries and modules.  
"""  
import sys, os, pytest, logging, asyncio  
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))))  
from unittest.mock import patch, AsyncMock, MagicMock, Mock  
  
from control.AvailabilityControl import AvailabilityControl  
from control.PriceControl import PriceControl  
from control.BrowserControl import BrowserControl  
from control.BotControl import BotControl  
  
from entity.BrowserEntity import BrowserEntity  
from entity.DataExportEntity import ExportUtils  
from entity.PriceEntity import PriceEntity  
from entity.AvailabilityEntity import AvailabilityEntity  
from entity.EmailEntity import send\_email\_with\_attachments  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main()

--- unitTest\_check\_availability.py ---

from test\_init import \*  
"""  
Executable steps for the 'Check\_Availability' use case:  
1. Control Layer Command Reception  
This test will ensure that AvailabilityControl.receive\_command() handles the "check\_availability" command properly, including parsing and validating parameters such as URL and optional date string.  
  
2. Availability Checking  
This test focuses on the AvailabilityEntity.check\_availability() function to verify that it correctly processes the availability check against a provided URL and optional date string. It will ensure that the availability status is accurately determined and returned.  
  
3. Data Logging to Excel  
This test checks that the event data is correctly logged to an Excel file using DataExportEntity.log\_to\_excel(). It will verify that the export includes the correct data formatting, timestamping, and file handling, ensuring data integrity.  
  
4. Data Logging to HTML  
Ensures that the event data is appropriately exported to an HTML file using DataExportEntity.export\_to\_html(). This test will confirm the data integrity and formatting in the HTML output, ensuring it matches expected outcomes.  
"""  
  
  
# Testing the control layer's ability to receive and process the "check\_availability" command  
@pytest.mark.asyncio  
async def test\_control\_layer\_command\_reception():  
 logging.info("Starting test: Control Layer Command Reception for check\_availability command")  
   
 command\_data = "check\_availability"  
 url = "https://example.com/reservation"  
 date\_str = "2023-10-10"  
  
 with patch('control.AvailabilityControl.AvailabilityControl.receive\_command', new\_callable=AsyncMock) as mock\_receive:  
 control = AvailabilityControl()  
 await control.receive\_command(command\_data, url, date\_str)  
   
 logging.info("Verifying that the receive\_command was called with correct parameters")  
 mock\_receive.assert\_called\_with(command\_data, url, date\_str)  
 logging.info("Test passed: Control layer correctly processes 'check\_availability'")  
  
# Testing the availability checking functionality from the AvailabilityEntity  
@pytest.mark.asyncio  
async def test\_availability\_checking():  
 with patch('entity.AvailabilityEntity.AvailabilityEntity.check\_availability', new\_callable=AsyncMock) as mock\_check:  
 # Mock returns a tuple mimicking the real function's output  
 mock\_check.return\_value = ("Checked availability: Availability confirmed",   
 "Data saved to Excel file at ExportedFiles\\excelFiles\\check\_availability.xlsx.",  
 "HTML file saved and updated at ExportedFiles\\htmlFiles\\check\_availability.html.")  
 result = await AvailabilityControl().check\_availability("https://example.com/reservation", "2023-10-10")  
   
 # Properly access the tuple and check the relevant part  
 assert "Availability confirmed" in result[0] # Accessing the first element of the tuple where the status message is  
  
  
# Testing the Excel logging functionality  
@pytest.mark.asyncio  
async def test\_data\_logging\_excel():  
 logging.info("Starting test: Data Logging to Excel for check\_availability command")  
  
 with patch('entity.DataExportEntity.ExportUtils.log\_to\_excel', return\_value="Data saved to Excel file at path.xlsx") as mock\_excel:  
 excel\_result = ExportUtils.log\_to\_excel("check\_availability", "https://example.com", "Available")  
   
 logging.info("Verifying Excel file creation and data logging")  
 assert "path.xlsx" in excel\_result, "Excel data logging did not return expected file path"  
 logging.info("Test passed: Data correctly logged to Excel")  
  
# Testing the HTML export functionality  
@pytest.mark.asyncio  
async def test\_data\_logging\_html():  
 logging.info("Starting test: Data Export to HTML for check\_availability command")  
   
 with patch('entity.DataExportEntity.ExportUtils.export\_to\_html', return\_value="Data exported to HTML file at path.html") as mock\_html:  
 html\_result = ExportUtils.export\_to\_html("check\_availability", "https://example.com", "Available")  
   
 logging.info("Verifying HTML file creation and data export")  
 assert "path.html" in html\_result, "HTML data export did not return expected file path"  
 logging.info("Test passed: Data correctly exported to HTML")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_close\_browser.py ---

from test\_init import \*  
"""  
Executable steps for the !close\_browser use case:  
1. Control Layer Processing  
This test ensures that BrowserControl.receive\_command() handles the "!close\_browser" command correctly.  
  
2. Browser Closing  
This test focuses on the BrowserEntity.close\_browser() method to ensure it executes the browser closing process.  
  
3. Response Generation  
This test validates that the control layer correctly interprets the response from the browser closing step and returns the appropriate result to the boundary layer.  
"""  
  
# Test for Control Layer Processing  
@pytest.mark.asyncio  
async def test\_control\_layer\_processing():  
 logging.info("Starting test: Control Layer Processing for close\_browser")  
  
 with patch('entity.BrowserEntity.BrowserEntity.close\_browser') as mock\_close:  
 # Configure the mock to return different responses based on the browser state  
 mock\_close.side\_effect = ["Browser closed successfully.", "No browser is currently open."]  
 browser\_control = BrowserControl()  
  
 # First call simulates the browser being open and then closed  
 result = await browser\_control.receive\_command("close\_browser")  
 assert result == "Control Object Result: Browser closed successfully."  
 logging.info(f"Test when browser is initially open and then closed: Passed with '{result}'")  
  
 # Second call simulates the browser already being closed  
 result = await browser\_control.receive\_command("close\_browser")  
 assert result == "Control Object Result: No browser is currently open."  
 logging.info(f"Test when no browser is initially open: Passed with '{result}'")  
  
  
# Test for Browser Closing  
  
def test\_browser\_closing():  
 logging.info("Starting test: Browser Closing")  
  
 # Patching the webdriver.Chrome directly at the point of instantiation  
 with patch('selenium.webdriver.Chrome', new\_callable=MagicMock) as mock\_chrome:  
 mock\_driver = mock\_chrome.return\_value # Mock the return value which acts as the driver  
 mock\_driver.quit = MagicMock() # Mock the quit method of the driver  
  
 browser\_entity = BrowserEntity()  
 browser\_entity.browser\_open = True # Ensure the browser is considered open  
 browser\_entity.driver = mock\_driver # Set the mock driver as the browser entity's driver  
  
 result = browser\_entity.close\_browser()  
  
 mock\_driver.quit.assert\_called\_once() # Check if quit was called on the driver instance  
 logging.info("Expected outcome: Browser quit method called.")  
 logging.info(f"Actual outcome: {result}")  
  
 assert result == "Browser closed."  
 logging.info("Test passed: Browser closing was successful")  
  
  
# Test for Response Generation  
@pytest.mark.asyncio  
async def test\_response\_generation():  
 logging.info("Starting test: Response Generation for close\_browser")  
   
 with patch('control.BrowserControl.BrowserControl.receive\_command', new\_callable=AsyncMock) as mock\_receive:  
 mock\_receive.return\_value = "Browser closed successfully."  
   
 browser\_control = BrowserControl()  
 result = await browser\_control.receive\_command("close\_browser")  
   
 logging.info("Expected outcome: 'Browser closed successfully.'")  
 logging.info(f"Actual outcome: {result}")  
   
 assert result == "Browser closed successfully."  
 logging.info("Step 3 executed and Test passed: Response generation was successful")  
  
# This condition ensures that the pytest runner handles the test run.  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_get\_price.py ---

from test\_init import \*  
import pytest  
import logging  
from unittest.mock import patch, AsyncMock  
  
"""  
Executable steps for the 'get\_price' use case:  
1. Control Layer Processing:  
 This test ensures that `PriceControl.receive\_command()` correctly processes the 'get\_price' command,   
 including proper URL parameter handling and delegation to the `get\_price` method.  
  
2. Price Retrieval:  
 This test verifies that `PriceEntity.get\_price\_from\_page()` retrieves the correct price from the webpage,   
 simulating the fetching process accurately.  
  
3. Data Logging to Excel:  
 This test ensures that the price data is correctly logged to an Excel file using `DataExportEntity.log\_to\_excel()`,   
 ensuring that data is recorded properly.  
  
4. Data Logging to HTML:  
 This test ensures that the price data is correctly exported to an HTML file using `DataExportEntity.export\_to\_html()`,   
 validating the data export process.  
  
5. Response Assembly and Output:  
 This test confirms that the control layer assembles and outputs the correct response, including price information,   
 Excel and HTML paths, ensuring the completeness of the response.  
"""  
  
# Test 1: Control Layer Processing  
@pytest.mark.asyncio  
async def test\_control\_layer\_processing():  
 logging.info("Starting test: Control Layer Processing for 'get\_price' command")  
  
 # Mock the `get\_price` method to avoid browser interaction  
 with patch('control.PriceControl.PriceControl.get\_price', new\_callable=AsyncMock) as mock\_get\_price:  
 # Set the return value for `get\_price` method  
 mock\_get\_price.return\_value = ("100.00", "Data saved to Excel file at path.xlsx", "Data exported to HTML at path.html")  
  
 # Mock the PriceControl.receive\_command method  
 price\_control = PriceControl()  
  
 # Simulate the command processing  
 result = await price\_control.receive\_command("get\_price", "https://example.com/product")  
  
 # Validate the return values  
 logging.info("Verifying that the receive\_command correctly processed the 'get\_price' command")  
  
 # Unpack the result for clearer assertions  
 price, excel\_path, html\_path = result  
   
 # Validate the return values match what we mocked  
 assert price == "100.00", f"Expected price '100.00', got {price}"  
 assert excel\_path == "Data saved to Excel file at path.xlsx", f"Expected Excel path 'path.xlsx', got {excel\_path}"  
 assert html\_path == "Data exported to HTML at path.html", f"Expected HTML path 'path.html', got {html\_path}"  
  
 logging.info("Test passed: Control layer processing correctly handles 'get\_price'")  
  
  
# Test 2: Price Retrieval  
@pytest.mark.asyncio  
async def test\_price\_retrieval():  
 logging.info("Starting test: Price Retrieval from webpage")  
  
 # Mock the `get\_price\_from\_page` method to simulate price retrieval without browser interaction  
 with patch('entity.PriceEntity.PriceEntity.get\_price\_from\_page', return\_value="100.00") as mock\_price:  
 price\_control = PriceControl()  
   
 # Call the `get\_price` method  
 result = await price\_control.get\_price("https://example.com/product")  
  
 logging.info("Expected fetched price: '100.00'")  
 assert "100.00" in result, f"Expected price '100.00', got {result}"  
 logging.info("Test passed: Price retrieval successful and correct")  
  
  
# Test 3: Data Logging to Excel  
@pytest.mark.asyncio  
async def test\_data\_logging\_excel():  
 logging.info("Starting test: Data Logging to Excel")  
  
 # Mock the `get\_price` method to avoid browser interaction  
 with patch('control.PriceControl.PriceControl.get\_price', new\_callable=AsyncMock) as mock\_get\_price:  
 # Set return value for `get\_price` method  
 mock\_get\_price.return\_value = ("100.00", "Data saved to Excel file at path.xlsx", "Data exported to HTML file at path.html")  
  
 # Mock the log\_to\_excel method to simulate Excel data logging  
 with patch('entity.DataExportEntity.ExportUtils.log\_to\_excel', return\_value="Data saved to Excel file at path.xlsx") as mock\_excel:  
 price\_control = PriceControl()  
   
 # Call the `get\_price` method, which is now mocked  
 \_, excel\_result, \_ = await price\_control.get\_price("https://example.com/product")  
  
 logging.info("Verifying Excel file creation and data logging")  
 assert "path.xlsx" in excel\_result, f"Expected Excel path 'path.xlsx', got {excel\_result}"  
 logging.info("Test passed: Data correctly logged to Excel")  
  
  
# Test 4: Data Export to HTML  
@pytest.mark.asyncio  
async def test\_data\_logging\_html():  
 logging.info("Starting test: Data Export to HTML")  
  
 # Mock the `get\_price` method to avoid browser interaction  
 with patch('control.PriceControl.PriceControl.get\_price', new\_callable=AsyncMock) as mock\_get\_price:  
 # Set return value for `get\_price` method  
 mock\_get\_price.return\_value = ("100.00", "Data saved to Excel file at path.xlsx", "Data exported to HTML file at path.html")  
  
 # Mock the export\_to\_html method to simulate HTML export  
 with patch('entity.DataExportEntity.ExportUtils.export\_to\_html', return\_value="Data exported to HTML file at path.html") as mock\_html:  
 price\_control = PriceControl()  
  
 # Call the `get\_price` method, which is now mocked  
 \_, \_, html\_result = await price\_control.get\_price("https://example.com/product")  
  
 logging.info("Verifying HTML file creation and data export")  
 assert "path.html" in html\_result, f"Expected HTML path 'path.html', got {html\_result}"  
 logging.info("Test passed: Data correctly exported to HTML")  
  
  
# Test 5: Response Assembly and Output  
@pytest.mark.asyncio  
async def test\_response\_assembly\_and\_output():  
 logging.info("Starting test: Response Assembly and Output")  
  
 # Mock the `get\_price` method to simulate price retrieval  
 with patch('control.PriceControl.PriceControl.get\_price', new\_callable=AsyncMock) as mock\_get\_price:  
 mock\_get\_price.return\_value = ("100.00", "Data saved to Excel file at path.xlsx", "Data exported to HTML at path.html")  
  
 price\_control = PriceControl()  
  
 # Call `receive\_command` with `get\_price` command  
 result = await price\_control.receive\_command("get\_price", "https://example.com/product")  
  
 # Unpack the result  
 price, excel\_path, html\_path = result  
  
 logging.info("Checking response contains price, Excel, and HTML paths")  
 assert price == "100.00", f"Price did not match expected value, got {price}"  
 assert "path.xlsx" in excel\_path, f"Excel path did not match, got {excel\_path}"  
 assert "path.html" in html\_path, f"HTML path did not match, got {html\_path}"  
  
 logging.info("Test passed: Correct response assembled and output")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_login.py ---

from test\_init import \*  
"""  
Executable steps for the !login command use case:  
1. Control Layer Processing  
This test will ensure that BotControl.receive\_command() handles the "!login" command correctly, including proper parameter passing and validation.  
  
2. Website Interaction  
This test will focus on the BrowserEntity.login() function to ensure it processes the request to log into the website using the provided credentials.  
  
3. Response Generation  
This test will validate that the control layer correctly interprets the response from the website interaction step and returns the appropriate result to the boundary layer.  
"""  
  
# test\_bot\_control\_login.py  
@pytest.mark.asyncio  
async def test\_control\_layer\_login():  
 logging.info("Starting test: Control Layer Processing for Login")  
   
 with patch('entity.BrowserEntity.BrowserEntity.login', new\_callable=AsyncMock) as mock\_login:  
 mock\_login.return\_value = "Login successful!"  
 browser\_control = BrowserControl()  
  
 result = await browser\_control.receive\_command("login", "example.com", "user", "pass")  
   
 logging.info(f"Expected outcome: Control Object Result: Login successful!")  
 logging.info(f"Actual outcome: {result}")  
   
 assert result == "Control Object Result: Login successful!"  
 logging.info("Step 1 executed and Test passed: Control Layer Processing for Login was successful")  
  
@pytest.fixture  
def browser\_entity\_setup(): # Fixture to setup the BrowserEntity for testing  
 with patch('selenium.webdriver.Chrome') as mock\_browser: # Mocking the Chrome browser  
 entity = BrowserEntity() # Creating an instance of BrowserEntity  
 entity.driver = Mock() # Mocking the driver  
 entity.driver.get = Mock() # Mocking the get method  
 entity.driver.find\_element = Mock() # Mocking the find\_element method  
 return entity  
  
def test\_website\_interaction(browser\_entity\_setup):  
 logging.info("Starting test: Website Interaction for Login")   
   
 browser\_entity = browser\_entity\_setup # Setting up the BrowserEntity  
 browser\_entity.login = Mock(return\_value="Login successful!") # Mocking the login method  
   
 result = browser\_entity.login("http://example.com", "user", "pass") # Calling the login method  
   
 logging.info("Expected to attempt login on 'http://example.com'")   
 logging.info(f"Actual outcome: {result}")  
   
 assert "Login successful!" in result # Assertion to check if the login was successful  
 logging.info("Step 2 executed and Test passed: Website Interaction for Login was successful")  
  
# test\_response\_generation.py  
@pytest.mark.asyncio  
async def test\_response\_generation():  
 logging.info("Starting test: Response Generation for Login")  
   
 with patch('control.BrowserControl.BrowserControl.receive\_command', new\_callable=AsyncMock) as mock\_receive:  
 mock\_receive.return\_value = "Login successful!"  
 browser\_control = BrowserControl()  
  
 result = await browser\_control.receive\_command("login", "example.com", "user", "pass")  
   
 logging.info("Expected outcome: 'Login successful!'")  
 logging.info(f"Actual outcome: {result}")  
   
 assert "Login successful!" in result  
 logging.info("Step 3 executed and Test passed: Response Generation for Login was successful")  
  
# This condition ensures that the pytest runner handles the test run.  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_navigate\_to\_website.py ---

from test\_init import \*  
# Define executable steps from the provided use case  
"""  
Executable steps for the navigate\_to\_website command:  
1. Command Processing and URL Extraction  
 - Ensure that the command is correctly processed and the URL is extracted and passed accurately to the control layer.  
  
2. Browser Navigation  
 - Verify that the browser control object receives the command and correctly triggers navigation to the URL.  
  
3. Response Generation  
 - Check that the correct response about navigation success or failure is generated and would be passed back to the boundary.  
"""  
  
# Test for Command Processing and URL Extraction  
@pytest.mark.asyncio  
async def test\_command\_processing\_and\_url\_extraction():  
 logging.info("Starting test: test\_command\_processing\_and\_url\_extraction")  
 with patch('control.BrowserControl.BrowserControl.receive\_command', new\_callable=AsyncMock) as mock\_receive:  
 mock\_receive.return\_value = "Navigating to URL"  
 browser\_control = BrowserControl()  
  
 # Simulate receiving the navigate command with a URL  
 result = await browser\_control.receive\_command("navigate\_to\_website", "http://example.com")  
   
 logging.info(f"Expected outcome: 'Navigating to URL'")  
 logging.info(f"Actual outcome: {result}")  
  
 assert result == "Navigating to URL"  
 logging.info("Step 1 executed and Test passed: Command Processing and URL Extraction was successful")  
  
# Test for Browser Navigation  
@pytest.mark.asyncio  
async def test\_browser\_navigation():  
 logging.info("Starting test: test\_browser\_navigation")  
 with patch('entity.BrowserEntity.BrowserEntity.navigate\_to\_website', new\_callable=AsyncMock) as mock\_navigate:  
 mock\_navigate.return\_value = "Navigation successful"  
 browser\_entity = BrowserEntity()  
 result = await browser\_entity.navigate\_to\_website("http://example.com")  
  
 logging.info("Expected outcome: 'Navigation successful'")  
 logging.info(f"Actual outcome: {result}")  
  
 assert result == "Navigation successful"  
 logging.info("Step 2 executed and Test passed: Browser Navigation was successful")  
  
# Test for Response Generation  
@pytest.mark.asyncio  
async def test\_response\_generation():  
 logging.info("Starting test: test\_response\_generation")  
 with patch('control.BrowserControl.BrowserControl.receive\_command', new\_callable=AsyncMock) as mock\_receive:  
 mock\_receive.return\_value = "Navigation confirmed"  
 browser\_control = BrowserControl()  
  
 result = await browser\_control.receive\_command("confirm\_navigation", "http://example.com")  
  
 logging.info("Expected outcome: 'Navigation confirmed'")  
 logging.info(f"Actual outcome: {result}")  
  
 assert result == "Navigation confirmed"  
 logging.info("Step 3 executed and Test passed: Response Generation was successful")  
  
# This condition ensures that the pytest runner handles the test run.  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_project\_help.py ---

from test\_init import \*  
"""  
Executable steps for the project\_help use case:  
1. Control Layer Processing  
This test will ensure that BotControl.receive\_command() handles the "project\_help" command correctly, including proper parameter passing.  
"""  
  
# test\_project\_help\_control.py  
@pytest.mark.asyncio  
async def test\_project\_help\_control():  
 # Start logging the test case  
 logging.info("Starting test: test\_project\_help\_control")  
   
 # Mocking the BotControl to simulate control layer behavior  
 with patch('control.BotControl.BotControl.receive\_command', new\_callable=AsyncMock) as mock\_command:  
 # Setup the mock to return the expected help message  
 expected\_help\_message = "Here are the available commands:..."  
 mock\_command.return\_value = expected\_help\_message  
   
 # Creating an instance of BotControl  
 control = BotControl()  
   
 # Simulating the command processing  
 result = await control.receive\_command("project\_help")  
   
 # Logging expected and actual outcomes  
 logging.info(f"Expected outcome: '{expected\_help\_message}'")  
 logging.info(f"Actual outcome: '{result}'")  
   
 # Assertion to check if the result is as expected  
 assert result == expected\_help\_message  
 logging.info("Step 1 executed and Test passed: Control Layer Processing was successful")  
  
# This condition ensures that the pytest runner handles the test run.  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_receive\_email.py ---

from test\_init import \*  
"""  
Executable steps for the receive\_email use case:  
1. Control Layer Processing  
This test will ensure that BotControl.receive\_command() handles the "receive\_email" command correctly, including proper parameter passing.  
  
2. Email Handling  
This test will focus on the EmailEntity.send\_email\_with\_attachments() function to ensure it processes the request and handles file operations and email sending as expected.  
  
3. Response Generation  
This test will validate that the control layer correctly interprets the response from the email handling step and returns the appropriate result to the boundary layer.  
"""  
  
# test\_bot\_control.py  
@pytest.mark.asyncio  
async def test\_control\_layer\_processing():  
 # Start logging the test case  
 logging.info("Starting test: test\_control\_layer\_processing")  
   
 # Mocking the email sending function to simulate email sending without actual I/O operations  
 with patch('entity.EmailEntity.send\_email\_with\_attachments', new\_callable=AsyncMock) as mock\_email:   
 mock\_email.return\_value = "Email with file 'testfile.txt' sent successfully!"   
 # Creating an instance of BotControl  
 bot\_control = BotControl()  
   
 # Calling the receive\_command method and passing the command and filename  
 result = await bot\_control.receive\_command("receive\_email", "testfile.txt")  
   
 # Logging expected and actual outcomes  
 logging.info(f"Expected outcome: 'Email with file 'testfile.txt' sent successfully!'")  
 logging.info(f"Actual outcome: {result}")  
   
 # Assertion to check if the result is as expected  
 assert result == "Email with file 'testfile.txt' sent successfully!"  
 logging.info("Step 1 executed and Test passed: Control Layer Processing was successful")  
  
  
# test\_email\_handling.py  
def test\_email\_handling():  
 # Start logging the test case  
 logging.info("Starting test: test\_email\_handling")  
   
 # Mocking the SMTP class to simulate sending an email  
 with patch('smtplib.SMTP') as mock\_smtp:  
 # Simulating the sending of an email  
 result = send\_email\_with\_attachments("testfile.txt")  
   
 # Logging expected and actual outcomes  
 logging.info("Expected outcome: Contains 'Email with file 'testfile.txt' sent successfully!'")  
 logging.info(f"Actual outcome: {result}")  
   
 # Assertion to check if the result contains the success message  
 assert "Email with file 'testfile.txt' sent successfully!" in result  
 logging.info("Step 2 executed and Test passed: Email handling was successful")  
  
  
# test\_response\_generation.py  
@pytest.mark.asyncio  
async def test\_response\_generation():  
 # Start logging the test case  
 logging.info("Starting test: test\_response\_generation")  
   
 # Mocking the BotControl.receive\_command to simulate control layer behavior  
 with patch('control.BotControl.BotControl.receive\_command', new\_callable=AsyncMock) as mock\_receive:  
 mock\_receive.return\_value = "Email with file 'testfile.txt' sent successfully!"  
   
 # Creating an instance of BotControl  
 bot\_control = BotControl()  
   
 # Calling the receive\_command method and passing the command and filename  
 result = await bot\_control.receive\_command("receive\_email", "testfile.txt")  
   
 # Logging expected and actual outcomes  
 logging.info("Expected outcome: 'Email with file 'testfile.txt' sent successfully!'")  
 logging.info(f"Actual outcome: {result}")  
   
 # Assertion to check if the result is as expected  
 assert "Email with file 'testfile.txt' sent successfully!" in result  
 logging.info("Step 3 executed and Test passed: Response generation was successful")  
  
  
# This condition ensures that the pytest runner handles the test run.  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])  
  
  
  
"""  
@pytest.mark.asyncio  
async def test\_handle\_receive\_email():  
 # Explanation: Patching the 'receive\_command' to simulate control layer behavior without actual execution.  
 with patch('control.BotControl.BotControl.receive\_command', new\_callable=AsyncMock) as mock\_receive\_command:  
 # Expected return value from the mocked method  
 mock\_receive\_command.return\_value = "Email with file 'monitor\_price.html' sent successfully!"  
  
 # Instantiate BotControl to test the interaction within the control layer  
 control = BotControl()  
  
 # Explanation: This line simulates the control layer receiving the 'receive\_email' command with a filename.  
 result = await control.receive\_command("receive\_email", "monitor\_price.html")  
  
 # Logging the result to understand what happens when the command is processed  
 logging.info(f'Result of receive\_command: {result}')  
  
 # Explanation: Assert that the mocked method returns the expected result  
 assert result == "Email with file 'monitor\_price.html' sent successfully!"  
 # Explanation: Ensure that the method was called exactly once with expected parameters  
 mock\_receive\_command.assert\_called\_once\_with("receive\_email", "monitor\_price.html")  
 """

--- unitTest\_start\_monitoring\_availability.py ---

from test\_init import \*  
"""  
Executable steps for the `start\_monitoring\_availability` use case:  
  
1. Control Layer Processing:  
 This test ensures that `AvailabilityControl.receive\_command()` handles the "start\_monitoring\_availability" command correctly,  
 including proper parameter passing for the URL, date, and frequency.  
  
2. Availability Monitoring Initiation:  
 This test verifies that the control layer starts the monitoring process by calling `check\_availability()` at regular intervals.  
  
3. Stop Monitoring Logic:  
 This test confirms that the monitoring can be stopped correctly using the "stop\_monitoring\_availability" command and that the final results are collected.  
"""  
  
# Test 1: Control Layer Processing  
@pytest.mark.asyncio  
async def test\_control\_layer\_processing():  
 logging.info("Starting test: test\_control\_layer\_processing")  
  
 url = "https://example.com/availability"  
 frequency = 1  
 logging.info(f"Testing command processing for URL: {url} with frequency: {frequency}")  
  
 # Mock the actual command handling to simulate command receipt and processing  
 with patch('control.AvailabilityControl.AvailabilityControl.receive\_command', new\_callable=AsyncMock) as mock\_receive:  
 logging.info("Patching receive\_command method...")  
  
 # Simulate receiving the 'start\_monitoring\_availability' command  
 result = await AvailabilityControl().receive\_command("start\_monitoring\_availability", url, None, frequency)  
  
 logging.info("Verifying if 'start\_monitoring\_availability' was processed correctly...")  
 assert "start\_monitoring\_availability" in str(mock\_receive.call\_args)  
 assert mock\_receive.call\_args[0][1] == url  
 assert mock\_receive.call\_args[0][3] == frequency  
 logging.info("Test passed: Control layer processed 'start\_monitoring\_availability' correctly.")  
  
# Test 2: Availability Monitoring Initiation  
@pytest.mark.asyncio  
async def test\_availability\_monitoring\_initiation():  
 logging.info("Starting test: test\_availability\_monitoring\_initiation")  
  
 availability\_control = AvailabilityControl()  
 url = "https://example.com/availability"  
 frequency = 3  
 logging.info(f"Initiating availability monitoring for URL: {url} with frequency: {frequency}")  
  
 # Mock the check\_availability method to return a constant value  
 with patch.object(availability\_control, 'check\_availability', new\_callable=AsyncMock) as mock\_check\_availability:  
 logging.info("Patching check\_availability method...")  
 mock\_check\_availability.return\_value = "Available"  
  
 # Start the monitoring process (monitoring in a separate task)  
 monitoring\_task = asyncio.create\_task(availability\_control.start\_monitoring\_availability(url, None, frequency))  
 logging.info("Monitoring task started.")  
  
 # Simulate a brief period of monitoring (e.g., for two intervals)  
 await asyncio.sleep(8)  
 logging.info(f"Simulated monitoring for 5 seconds, checking number of calls to check\_availability.")  
  
 # Check if check\_availability was called twice due to the frequency  
 assert mock\_check\_availability.call\_count == 2, f"Expected 2 availability checks, but got {mock\_check\_availability.call\_count}"  
 logging.info("Test passed: Availability monitoring initiated and 'check\_availability' called twice.")  
  
 # Stop the monitoring  
 logging.info("Stopping availability monitoring...")  
 availability\_control.stop\_monitoring\_availability()  
 await monitoring\_task # Wait for the task to stop  
  
 # Ensure monitoring stopped and results were collected  
 assert len(availability\_control.results) == 2  
 logging.info(f"Test passed: Monitoring stopped with {len(availability\_control.results)} results.")  
  
# Test 3: Stop Monitoring Logic  
@pytest.mark.asyncio  
async def test\_stop\_monitoring\_logic():  
 logging.info("Starting test: test\_stop\_monitoring\_logic")  
  
 availability\_control = AvailabilityControl()  
 url = "https://example.com/availability"  
 frequency = 1  
 logging.info(f"Initiating monitoring to test stopping logic for URL: {url} with frequency: {frequency}")  
  
 # Mock check\_availability method  
 with patch.object(availability\_control, 'check\_availability', new\_callable=AsyncMock) as mock\_check\_availability:  
 logging.info("Patching check\_availability method...")  
 mock\_check\_availability.return\_value = "Available"  
  
 # Start monitoring  
 monitoring\_task = asyncio.create\_task(availability\_control.start\_monitoring\_availability(url, None, frequency))  
 logging.info("Monitoring task started.")  
  
 # Simulate monitoring for one interval  
 await asyncio.sleep(2)  
 logging.info("Simulated monitoring for 6 seconds, stopping monitoring now.")  
  
 # Stop the monitoring  
 availability\_control.stop\_monitoring\_availability()  
 await monitoring\_task # Wait for the task to stop  
  
 # Ensure the monitoring has stopped  
 assert availability\_control.is\_monitoring == False  
 assert len(availability\_control.results) >= 1  
 logging.info(f"Test passed: Monitoring stopped with {len(availability\_control.results)} result(s).")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_start\_monitoring\_price.py ---

from test\_init import \*  
"""  
Executable steps for the `start\_monitoring\_price` use case:  
  
1. Control Layer Processing:  
 This test will ensure that `PriceControl.receive\_command()` correctly handles the "start\_monitoring\_price" command,  
 including proper URL and frequency parameter passing.  
  
2. Price Monitoring Initiation:  
 This test will verify that the control layer starts the monitoring process by repeatedly calling `get\_price()` at regular intervals.  
  
3. Stop Monitoring Logic:  
 This test confirms that the monitoring can be stopped correctly using the "stop\_monitoring\_price" command and that final results are collected.  
"""  
  
# Test 1: Control Layer Processing for start\_monitoring\_price command  
@pytest.mark.asyncio  
async def test\_control\_layer\_processing():  
 logging.info("Starting test: test\_control\_layer\_processing")  
  
 url = "https://example.com/product"  
 frequency = 2  
 logging.info(f"Testing command processing for URL: {url} with frequency: {frequency}")  
  
 # Mock the actual command handling to simulate command receipt and processing  
 with patch('control.PriceControl.PriceControl.receive\_command', new\_callable=AsyncMock) as mock\_receive:  
 logging.info("Patching receive\_command method...")  
   
 # Simulate receiving the 'start\_monitoring\_price' command  
 result = await PriceControl().receive\_command("start\_monitoring\_price", url, frequency)  
   
 logging.info("Verifying if 'start\_monitoring\_price' was processed correctly...")  
 assert "start\_monitoring\_price" in str(mock\_receive.call\_args)  
 assert mock\_receive.call\_args[0][1] == url  
 assert mock\_receive.call\_args[0][2] == frequency  
 logging.info("Test passed: Control layer processed 'start\_monitoring\_price' correctly.")  
  
# Test 2: Price Monitoring Initiation  
@pytest.mark.asyncio  
async def test\_price\_monitoring\_initiation():  
 logging.info("Starting test: test\_price\_monitoring\_initiation")  
  
 price\_control = PriceControl()  
 url = "https://example.com/product"  
 frequency = 3  
 logging.info(f"Initiating price monitoring for URL: {url} with frequency: {frequency}")  
  
 # Mock the get\_price method to return a constant value  
 with patch.object(price\_control, 'get\_price', new\_callable=AsyncMock) as mock\_get\_price:  
 logging.info("Patching get\_price method...")  
 mock\_get\_price.return\_value = "100.00"  
  
 # Start the monitoring process (monitoring in a separate task)  
 monitoring\_task = asyncio.create\_task(price\_control.start\_monitoring\_price(url, frequency))  
 logging.info("Monitoring task started.")  
  
 # Simulate a brief period of monitoring (e.g., two intervals)  
 await asyncio.sleep(8)  
 logging.info(f"Simulated monitoring for 5 seconds, checking number of calls to get\_price.")  
  
 # Check if get\_price was called twice due to the frequency  
 assert mock\_get\_price.call\_count == 2, f"Expected 2 price checks, but got {mock\_get\_price.call\_count}"  
 logging.info("Test passed: Price monitoring initiated and 'get\_price' called twice.")  
  
 # Stop the monitoring  
 logging.info("Stopping price monitoring...")  
 price\_control.stop\_monitoring\_price()  
 await monitoring\_task # Wait for the task to stop  
  
 # Ensure monitoring stopped and results were collected  
 assert len(price\_control.results) == 2  
 logging.info(f"Test passed: Monitoring stopped with {len(price\_control.results)} results.")  
  
# Test 3: Stop Monitoring Logic  
@pytest.mark.asyncio  
async def test\_stop\_monitoring\_logic():  
 logging.info("Starting test: test\_stop\_monitoring\_logic")  
  
 price\_control = PriceControl()  
 url = "https://example.com/product"  
 frequency = 2  
 logging.info(f"Initiating monitoring to test stopping logic for URL: {url} with frequency: {frequency}")  
  
 # Mock get\_price method  
 with patch.object(price\_control, 'get\_price', new\_callable=AsyncMock) as mock\_get\_price:  
 logging.info("Patching get\_price method...")  
 mock\_get\_price.return\_value = "100.00"  
  
 # Start monitoring  
 monitoring\_task = asyncio.create\_task(price\_control.start\_monitoring\_price(url, frequency))  
 logging.info("Monitoring task started.")  
  
 # Simulate monitoring for one interval  
 await asyncio.sleep(3)  
 logging.info("Simulated monitoring for 3 seconds, stopping monitoring now.")  
  
 # Stop the monitoring  
 price\_control.stop\_monitoring\_price()  
 await monitoring\_task # Wait for the task to stop  
  
 # Ensure the monitoring has stopped  
 assert price\_control.is\_monitoring == False  
 assert len(price\_control.results) >= 1  
 logging.info(f"Test passed: Monitoring stopped with {len(price\_control.results)} result(s).")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_stop\_monitoring\_availability.py ---

from test\_init import \*  
"""  
Executable steps for the 'Stop\_monitoring\_availability' use case:  
  
1. Control Layer Processing:  
 This test ensures that `AvailabilityControl.receive\_command()` correctly handles the "stop\_monitoring\_availability" command.  
  
2. Monitoring Termination:  
 This test verifies that the control layer terminates an ongoing availability monitoring session.  
  
3. Final Results Summary:  
 This test confirms that the control layer returns the correct summary of monitoring results once the process is terminated.  
"""  
  
# Test 1: Control Layer Processing for stop\_monitoring\_availability command  
@pytest.mark.asyncio  
async def test\_control\_layer\_processing():  
 logging.info("Starting test: Control Layer Processing for stop\_monitoring\_availability command")  
  
 with patch('control.AvailabilityControl.AvailabilityControl.receive\_command', new\_callable=AsyncMock) as mock\_receive:  
 # Simulate receiving the 'stop\_monitoring\_availability' command  
 result = await AvailabilityControl().receive\_command("stop\_monitoring\_availability")  
  
 # Verify that the command was processed correctly  
 assert "stop\_monitoring\_availability" in str(mock\_receive.call\_args)  
 logging.info("Test passed: Control layer processed stop\_monitoring\_availability command successfully.")  
  
# Test 2: Monitoring Termination  
@pytest.mark.asyncio  
async def test\_monitoring\_termination():  
 logging.info("Starting test: Monitoring Termination for stop\_monitoring\_availability")  
  
 availability\_control = AvailabilityControl()  
 availability\_control.is\_monitoring = True # Simulate that monitoring is active  
 availability\_control.results = ["Availability at URL was available.", "Availability was checked again."]  
  
 # Simulate monitoring stop  
 logging.info("Stopping availability monitoring...")  
 result = availability\_control.stop\_monitoring\_availability()  
  
 # Verify that monitoring was stopped and flag was set correctly  
 assert availability\_control.is\_monitoring == False  
 logging.info("Test passed: Monitoring was terminated successfully.")  
  
# Test 3: Final Results Summary  
@pytest.mark.asyncio  
async def test\_final\_summary\_generation():  
 logging.info("Starting test: Final Results Summary for stop\_monitoring\_availability")  
  
 availability\_control = AvailabilityControl()  
 availability\_control.is\_monitoring = True # Simulate an ongoing monitoring session  
 availability\_control.results = ["Availability at URL was available.", "Availability was checked again."]  
  
 # Simulate the monitoring stop and ensure results are collected  
 logging.info("Stopping availability monitoring and generating final summary...")  
 result = availability\_control.stop\_monitoring\_availability()  
  
 # Verify that the summary contains the expected results  
 assert "Availability at URL was available." in result  
 assert "Availability was checked again." in result  
 assert "Monitoring stopped successfully!" in result  
 logging.info("Test passed: Final summary generated correctly.")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_file\_\_])

--- unitTest\_stop\_monitoring\_price.py ---

from test\_init import \*  
"""  
Executable steps for the `stop\_monitoring\_price` use case:  
  
1. Control Layer Processing:  
 This test will ensure that `PriceControl.receive\_command()` correctly handles the "stop\_monitoring\_price" command,  
 including the proper termination of the price monitoring process.  
  
2. Stop Monitoring Logic:  
 This test verifies that the control layer stops the price monitoring process and collects the final results correctly.  
  
3. Final Summary Generation:  
 This test confirms that the control layer generates and returns a final summary of the monitoring session, containing the collected price results.  
"""  
  
# Test 1: Control Layer Processing for stop\_monitoring\_price command  
@pytest.mark.asyncio  
async def test\_control\_layer\_processing():  
 logging.info("Starting test: test\_control\_layer\_processing")  
  
 # Mock the actual command handling to simulate command receipt and processing  
 with patch('control.PriceControl.PriceControl.receive\_command', new\_callable=AsyncMock) as mock\_receive:  
 logging.info("Patching receive\_command method...")  
   
 # Simulate receiving the 'stop\_monitoring\_price' command  
 result = await PriceControl().receive\_command("stop\_monitoring\_price")  
   
 logging.info("Verifying if 'stop\_monitoring\_price' was processed correctly...")  
 assert "stop\_monitoring\_price" in str(mock\_receive.call\_args)  
 logging.info("Test passed: Control layer processed 'stop\_monitoring\_price' command correctly.")  
  
# Test 2: Stop Monitoring Logic  
@pytest.mark.asyncio  
async def test\_stop\_monitoring\_logic():  
 logging.info("Starting test: test\_stop\_monitoring\_logic")  
  
 price\_control = PriceControl()  
 price\_control.is\_monitoring = True # Simulate an ongoing monitoring session  
  
 # Mock the stop\_monitoring\_price method  
 with patch.object(price\_control, 'stop\_monitoring\_price', wraps=price\_control.stop\_monitoring\_price) as mock\_stop\_monitoring:  
 logging.info("Patching stop\_monitoring\_price method...")  
  
 # Simulate the stop command  
 result = price\_control.stop\_monitoring\_price()  
  
 logging.info("Checking if monitoring stopped and results were collected...")  
 assert price\_control.is\_monitoring == False  
 logging.info("Monitoring was successfully stopped.")  
 assert len(price\_control.results) >= 0 # Ensuring that results were collected  
 logging.info("Results were collected successfully.")  
 logging.info("Test passed: Stop monitoring logic executed correctly.")  
  
  
# Test 3: Final Summary Generation  
@pytest.mark.asyncio  
async def test\_final\_summary\_generation():  
 logging.info("Starting test: test\_final\_summary\_generation")  
  
 price\_control = PriceControl()  
 price\_control.is\_monitoring = True # Simulate an ongoing monitoring session  
 price\_control.results = ["Price at URL was $100", "Price dropped to $90"] # Mock some results  
  
 # Simulate the monitoring stop and ensure results are collected  
 logging.info("Stopping price monitoring and generating final summary...")  
 result = price\_control.stop\_monitoring\_price()  
  
 # Ensure that the summary contains the expected results  
 logging.info("Verifying the final summary contains the collected results...")  
 assert "Price at URL was $100" in result  
 assert "Price dropped to $90" in result  
 assert "Price monitoring stopped successfully!" in result # Updated to match the actual result  
 logging.info("Test passed: Final summary generated correctly.")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 pytest.main([\_\_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

--- MyBot.py ---

import discord  
from discord.ext import commands  
from boundary.BrowserBoundary import BrowserBoundary  
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(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)