

--- 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
```



```
--- 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": "#restProfileSideBarDtpDayPicker-label",
    "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)
```