

--- main.py ---

```
from utils.MyBot import start_bot
```

```
from utils.Config import Config
```

```
# Initialize and run the bot
```

```
if __name__ == "__main__":
```

```
    print("Bot is starting...")
```

```
    start_bot(Config.DISCORD_TOKEN) # Start the bot using the token from config
```

```
--- AccountBoundary.py ---
```

```
from discord.ext import commands
```

```
from control.AccountControl import AccountControl
```

```
from DataObjects.global_vars import GlobalState
```

```
class AccountBoundary(commands.Cog):
```

```
    def __init__(self):
```

```
        self.control = AccountControl() # Initialize control object
```

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

```
    async def fetch_all_accounts(self, ctx):
```

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

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

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

```
        result = self.control.receive_command(command)
```

```
        # Send the result (prepared by control) back to the user
```

```
        await ctx.send(result)
```

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

```
    async def fetch_account_by_website(self, ctx):
```

```
        list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
```

command and up to 6 variables

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

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

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

```
result = self.control.receive_command(command, website)
```

```
# Send the result (prepared by control) back to the user
```

```
await ctx.send(result)
```

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

```
async def add_account(self, ctx):
```

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

```
    list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message into
```

command and up to 6 variables

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

```
username = list[1] # Second element is the username
```

```
password = list[2] # Third element is the password
```

```
website = list[3] # Third element is the website
```

```
result = self.control.receive_command(command, username, password, website)
```

```
# Send the result (prepared by control) back to the user
```

```
await ctx.send(result)
```

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

```
async def delete_account(self, ctx):
```

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

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

```
    account_id = list[1] # Second element is the account_id
```

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

```
    result = self.control.receive_command(command, account_id)
```

```
    # Send the result (prepared by control) back to the user
```

```
    await ctx.send(result)
```

```
--- AvailabilityBoundary.py ---
```

```
from discord.ext import commands
```

```
from control.AvailabilityControl import AvailabilityControl
```

```
from DataObjects.global_vars import GlobalState
```

```
class AvailabilityBoundary(commands.Cog):
```

```
    def __init__(self):
```

```
        # Initialize control objects directly
```

```
        self.availability_control = AvailabilityControl()
```

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

```
    async def check_availability(self, ctx):
```

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

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

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

```
        url = list[1] # Second element is the URL
```

```
        date_str = list[2] # Third element is the date
```

```
        # Pass the command and data to the control layer using receive_command
```

```
        result = await self.availability_control.receive_command(command, url, date_str)
```

```
# Send the result back to the user
```

```
await ctx.send(result)
```

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

```
async def start_monitoring_availability(self, ctx):
```

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

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

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

```
    url = list[1] # Second element is the URL
```

```
    date_str = list[2] # Third element is the date
```

```
    frequency = list[3] # Fourth element is the frequency
```

```
    response = await self.availability_control.receive_command(command, url, date_str, frequency)
```

```
# Send the result back to the user
```

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

```
@commands.command(name='stop_monitoring_availability')
```

```
async def stop_monitoring_availability(self, ctx):
```

```
    """Command to stop monitoring the price."""
```

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

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

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

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

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


--- BotBoundary.py ---

```
from discord.ext import commands
```

```
from control.BotControl import BotControl
```

```
from DataObjects.global_vars import GlobalState
```

```
class BotBoundary(commands.Cog):
```

```
    def __init__(self):
```

```
        self.bot_control = BotControl() # Initialize control object
```

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

```
    async def project_help(self, ctx):
```

```
        """Handle help command by sending available commands to the user."""
```

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

```
        try:
```

```
            list = GlobalState.parse_user_message(GlobalState.user_message) # Parse the message
```

```
into command and up to 6 variables
```

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

```
            response = await self.bot_control.receive_command(command) # Call control layer
```

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

```
        except Exception as e:
```

```
            error_msg = f"Error in HelpBoundary: {str(e)}"
```

```
            print(error_msg)
```

```
            await ctx.send(error_msg)
```

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

```

async def stop_bot(self, ctx):

    """Handle stop bot command by shutting down the bot."""

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

    try:

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

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

        result = await self.bot_control.receive_command(command, ctx) # Call control layer to stop
the bot

        print(result) # Send the result to the terminal since the bot will shut down

    except Exception as e:

        error_msg = f"Error in StopBoundary: {str(e)}"

        print(error_msg)

        await ctx.send(error_msg)


@commands.command(name="receive_email")

async def receive_email(self, ctx):

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

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

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

    file_name = list[1] # Second element is the fileName

```

```
    result = await self.bot_control.receive_command(command, file_name) # Pass the command to  
the control layer  
    await ctx.send(result)
```

```
--- BrowserBoundary.py ---
```

```
from discord.ext import commands
```

```
from control.BrowserControl import BrowserControl
```

```
from DataObjects.global_vars import GlobalState
```

```
class BrowserBoundary(commands.Cog):
```

```
    def __init__(self):
```

```
        self.browser_control = BrowserControl() # Initialize Browser control object
```

```
    # Browser-related commands
```

```
    @commands.command(name='launch_browser')
```

```
    async def launch_browser(self, ctx):
```

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

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

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

```
        result = await self.browser_control.receive_command(command) # Pass the updated  
user_message to the control object
```

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

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

```
    async def close_browser(self, ctx):
```

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

```

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

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

result = await self.browser_control.receive_command(command)

await ctx.send(result)

# Login-related commands

@commands.command(name='login')

async def login(self, ctx):

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

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

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

website = list[1]

result = await self.browser_control.receive_command(command, website) # Pass the
command and website to control object

# Send the result back to the user

await ctx.send(result)

# Navigation-related commands

@commands.command(name='navigate_to_website')

async def navigate_to_website(self, ctx):

```

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

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

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

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

```
    result = await self.browser_control.receive_command(command, website) # Pass the parsed  
variables to the control object
```

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

--- PriceBoundary.py ---

```
from discord.ext import commands
```

```
from control.PriceControl import PriceControl
```

```
from DataObjects.global_vars import GlobalState
```

```
class PriceBoundary(commands.Cog):
```

```
    def __init__(self):
```

```
        # Initialize control objects directly
```

```
        self.price_control = PriceControl()
```

```
    @commands.command(name='get_price')
```

```
    async def get_price(self, ctx):
```

```
        """Command to get the price from the given URL."""
```

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

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

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

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

```
        result = await self.price_control.receive_command(command, website) # Pass the command to  
the control layer
```

```
        await ctx.send(f"Price found: {result}")
```

```
    @commands.command(name='start_monitoring_price')
```

```

async def start_monitoring_price(self, ctx):

    """Command to monitor price at given frequency."""

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

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

    website = list[1] # Second element is the URL

    frequency = list[2]


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


    response = await self.price_control.receive_command(command, website, frequency)

    await ctx.send(response)


@commands.command(name='stop_monitoring_price')

async def stop_monitoring_price(self, ctx):

    """Command to stop monitoring the price."""

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


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

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


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

```



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

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

```
#empty init file
```

--- AccountControl.py ---

```
from DataObjects.AccountDAO import AccountDAO
```

```
class AccountControl:
```

```
    def __init__(self):
```

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

```
    def receive_command(self, command, *args):
```

```
        """Handle all account-related commands and process business logic."""
```

```
        print("Data received from boundary:", command)
```

```
        if command == "fetch_all_accounts":
```

```
            return self.fetch_all_accounts()
```

```
        elif command == "fetch_account_by_website":
```

```
            website = args[0] if args else None
```

```
            return self.fetch_account_by_website(website)
```

```
        elif command == "add_account":
```

```
            username, password, website = args if args else (None, None, None)
```

```
            return self.add_account(username, password, website)
```

```
        elif command == "delete_account":
```

```
            account_id = args[0] if args else None
```

```
            return self.delete_account(account_id)
```

else:

result = "Invalid command."

print(result)

return result

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

"""Add a new account to the database."""

self.account_dao.connect()

result = self.account_dao.add_account(username, password, website)

self.account_dao.close()

result_message = f"Account for {website} added successfully." if result else f"Failed to add account for {website}."

print(result_message)

return result_message

def delete_account(self, account_id: int):

"""Delete an account by ID."""

self.account_dao.connect()

try:

result = self.account_dao.delete_account(account_id)

except Exception as e:

print(f"Error deleting account: {e}")

return "Error deleting account."

self.account_dao.reset_id_sequence()

self.account_dao.close()

```
        result_message = f"Account with ID {account_id} deleted successfully." if result else f"Failed to  
delete account with ID {account_id}."
```

```
    print(result_message)
```

```
    return result_message
```

```
def fetch_all_accounts(self):
```

```
    """Fetch all accounts using the DAO."""
```

```
    self.account_dao.connect()
```

```
    try:
```

```
        accounts = self.account_dao.fetch_all_accounts()
```

```
    except Exception as e:
```

```
        return "Error fetching accounts."
```

```
    self.account_dao.close()
```

```
    if accounts:
```

```
        account_list = "\n".join([f"ID: {acc[0]}, Username: {acc[1]}, Password: {acc[2]}, Website:  
{acc[3]}" for acc in accounts])
```

```
        result_message = f"Accounts:\n{account_list}"
```

```
    else:
```

```
        result_message = "No accounts found."
```

```
    print(result_message)
```

```
    return result_message
```

```
def fetch_account_by_website(self, website: str):
```

```
"""Fetch an account by website."""
```

```
try:
```

```
    self.account_dao.connect()
```

```
    account = self.account_dao.fetch_account_by_website(website)
```

```
    self.account_dao.close()
```

```
    # Logic to format the result within the control layer
```

```
    if account:
```

```
        return account
```

```
    else:
```

```
        return f"No account found for {website}."
```

```
except Exception as e:
```

```
    return f"Error: {str(e)}"
```

--- AvailabilityControl.py ---

```
import asyncio
```

```
from entity.AvailabilityEntity import AvailabilityEntity
```

```
from datetime import datetime
```

```
from utils.css_selectors import Selectors
```

```
from utils.exportUtils import ExportUtils
```

```
from utils.configuration import load_config
```

```
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

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

except Exception as e:

error_message = f"Failed to monitor availability: {str(e)}"

```
print(error_message)

return error_message
```

```
return self.results
```

```
def stop_monitoring_availability(self):
```

```
    """Stop monitoring availability."""
```

```
    print("Stopping availability monitoring...")
```

```
    result = None
```

```
    try:
```

```
        if not self.is_monitoring:
```

```
            # If no monitoring session is active
```

```
            result = "There was no active availability monitoring session. Nothing to stop."
```

```
        else:
```

```
            # Stop monitoring and collect results
```

```
            self.is_monitoring = False
```

```
            result = "Results for availability monitoring:\n"
```

```
            result += "\n".join(self.results)
```

```
            result = result + "\n" + "\nMonitoring stopped successfully!"
```

```
            print(result)
```

```
    except Exception as e:
```

```
        # Handle any error that occurs
```

```
        result = f"Error stopping availability monitoring: {str(e)}"
```

```
    return result
```


--- BotControl.py ---

```
import discord
```

```
from utils.email_utils import send_email_with_attachments
```

```
class BotControl:
```

```
    async def receive_command(self, command_data, *args):
```

```
        """Handle commands related to help and stopping the bot."""
```

```
        print("Data received from boundary:", command_data)
```

```
        # Handle help commands
```

```
        if command_data == "project_help":
```

```
            try:
```

```
                help_message = (
```

```
                    "Here are the available commands:\n"
```

```
                    "!project_help - Get help on available commands.\n"
```

```
                    "!fetch_all_accounts - Fetch all stored accounts.\n"
```

```
                    "!add_account 'username' 'password' 'website' - Add a new account to the database.\n"
```

```
                    "!fetch_account_by_website 'website' - Fetch account details by website.\n"
```

```
                    "!delete_account 'account_id' - Delete an account by its ID.\n"
```

```
                    "!launch_browser - Launch the browser.\n"
```

```
                    "!close_browser - Close the browser.\n"
```

```
                    "!navigate_to_website 'url' - Navigate to a specified website.\n"
```

```
                    "!login 'website' - Log in to a website (e.g., !login bestbuy).\n"
```

```
                    "!get_price 'url' - Check the price of a product on a specified website.\n"
```

```
                    "!start_monitoring_price 'url' 'frequency' - Start monitoring a product's price at a specific
```

```
interval (frequency in minutes).\n"
```

```
"!stop_monitoring_price - Stop monitoring the product's price.\n"
```

```
"!check_availability 'url' - Check availability for a restaurant or service.\n"
```

```
"!start_monitoring_availability 'url' 'frequency' - Monitor availability at a specific interval.\n"
```

```
"!stop_monitoring_availability - Stop monitoring availability.\n"
```

```
"!stop_bot - Stop the bot.\n"
```

```
)
```

```
return help_message
```

```
except Exception as e:
```

```
    error_msg = f"Error handling help command: {str(e)}"
```

```
    print(error_msg)
```

```
    return error_msg
```

```
# Handle stop bot commands
```

```
elif command_data == "stop_bot":
```

```
    try:
```

```
        ctx = args[0] if args else None
```

```
        bot = ctx.bot # Get the bot instance from the context
```

```
        await ctx.send("The bot is shutting down...")
```

```
        print("Bot is shutting down...")
```

```
        await bot.close() # Close the bot
```

```
        result = "Bot has been shut down."
```

```
        print(result)
```

```
        return result
```

```
except Exception as e:
```

```
    error_msg = f"Error shutting down the bot: {str(e)}"
```

```
print(error_msg)
```

```
return error_msg
```

```
# Handle receive email commands
```

```
elif command_data == "receive_email":
```

```
    try:
```

```
        file_name = args[0] if args else None
```

```
        if file_name:
```

```
            print(f"Sending email with the file '{file_name}'...")
```

```
            result = send_email_with_attachments(file_name)
```

```
            print(result)
```

```
        else:
```

```
            result = "Please specify a file to send, e.g., !receive_email monitor_price.html"
```

```
    return result
```

```
except Exception as e:
```

```
    error_msg = f"Error shutting down the bot: {str(e)}"
```

```
    print(error_msg)
```

```
    return error_msg
```

```
# Default response for invalid commands
```

```
else:
```

```
    try:
```

```
        return "Invalid command."
```

```
    except Exception as e:
```

```
error_msg = f"Error handling invalid command: {str(e)}"
```

```
print(error_msg)
```

```
return error_msg
```


--- BrowserControl.py ---

```
from entity.BrowserEntity import BrowserEntity
```

```
from control.AccountControl import AccountControl # Needed for LoginControl
```

```
from utils.css_selectors import Selectors # Used in both LoginControl and NavigationControl
```

```
import re # Used for URL pattern matching in LoginControl
```

```
class BrowserControl:
```

```
    def __init__(self):
```

```
        self.browser_entity = BrowserEntity() # Initialize the entity object inside the control layer
```

```
        self.account_control = AccountControl() # Manages account data for login use case
```

```
    # Browser-related command handler
```

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

```
        print("Data Received from boundary object: ", command_data)
```

```
    # Handle browser commands
```

```
    if command_data == "launch_browser":
```

```
        try:
```

```
            result = self.browser_entity.launch_browser()
```

```
            return f"Control Object Result: {result}"
```

```
        except Exception as e:
```

```
            return f"Control Layer Exception: {str(e)}"
```

```
    elif command_data == "close_browser":
```

```
        try:
```

```
            result = self.browser_entity.close_browser()
```

```

        return f"Control Object Result: {result}"

except Exception as e:

    return f"Control Layer Exception: {str(e)}"


# Handle login commands

elif command_data == "login" and site:

    try:

        # Fetch account credentials from the account control

        account_info = self.account_control.fetch_account_by_website(site)

        if not account_info:

            return f"No account found for {site}"

        username, password = account_info[0], account_info[1]

        print(f"Username: {username}, Password: {password}")

    # Improved regex to detect URLs even without http/https
    url_pattern = re.compile(r'(https?://)?(www\.)?(\w+)(\.\w{2,})')

    # Check if the input is a full URL or a site name

    if url_pattern.search(site):

        # If it contains a valid domain pattern, treat it as a URL

        if not site.startswith('http'):

            # Add 'https://' if the URL does not include a protocol

            url = f"https://{site}"

        else:

            url = site

```

```
print(f"Using provided URL: {url}")
```

```
else:
```

```
# If not a URL, look it up in the CSS selectors
```

```
selectors = Selectors.get_selectors_for_url(site)
```

```
if not selectors or 'url' not in selectors:
```

```
    return f"URL for {site} not found."
```

```
url = selectors.get('url')
```

```
print(f"URL from selectors: {url}")
```

```
if not url:
```

```
    return f"URL for {site} not found."
```

```
result = await self.browser_entity.login(url, username, password)
```

```
return f"Control Object Result: {result}"
```

```
except Exception as e:
```

```
    return f"Control Layer Exception: {str(e)}"
```

```
# Handle navigation commands
```

```
elif command_data == "navigate_to_website" and site:
```

```
    url_pattern = re.compile(r'(https?://)?(www\.)?(\w+)(\.\w{2,})')
```

```
# Check if the input is a full URL or a site name
```

```
if url_pattern.search(site):
```

```
    # If it contains a valid domain pattern, treat it as a URL
```

```
    if not site.startswith('http'):
```

```
        # Add 'https://' if the URL does not include a protocol
```

```
url = f"https://{site}"
```

```
else:
```

```
url = site
```

```
print(f"Using provided URL: {url}")
```

```
else:
```

```
# If not a URL, look it up in the CSS selectors
```

```
selectors = Selectors.get_selectors_for_url(site)
```

```
if not selectors or 'url' not in selectors:
```

```
    return f"URL for {site} not found."
```

```
url = selectors.get('url')
```

```
print("URL not provided, default URL for Google is: " + url)
```

```
try:
```

```
    result = self.browser_entity.navigate_to_website(url)
```

```
    return f"Control Object Result: {result}"
```

```
except Exception as e:
```

```
    return f"Control Layer Exception: {str(e)}"
```

```
else:
```

```
    return "Invalid command."
```

--- PriceControl.py ---

import asyncio

from datetime import datetime

from entity.PriceEntity import PriceEntity

from utils.configuration import load_config

from utils.css_selectors import Selectors

from utils.exportUtils import ExportUtils

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="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"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
```

```
else:
```

```
result = "Failed to retrieve the price."
```

```
# Add the result to the results list
```

```
self.results.append(result)
```

```
await asyncio.sleep(frequency)
```

```
except Exception as e:
```

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

```
def stop_monitoring_price(self):
```

```
    """Stop the price monitoring loop."""
```

```
    print("Stopping price monitoring...")
```

```
    result = None
```

```
    try:
```

```
        if not self.is_monitoring:
```

```
            # If no monitoring session is active
```

```
            result = "There was no active price monitoring session. Nothing to stop."
```

```
    else:
```

```
        # Stop monitoring and collect results
```

```
        self.is_monitoring = False
```

```
        result = "Results for price monitoring:\n"
```

```
        result += "\n".join(self.results)
```

```
result = result + "\n" + "\nPrice monitoring stopped successfully!"
```

```
print(result)
```

```
except Exception as e:
```

```
# Handle any error that occurs
```

```
result = f"Error stopping price monitoring: {str(e)}"
```

```
return result
```

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

```
#empty init file
```

--- AccountDAO.py ---

import psycopg2

from utils.Config import Config

class AccountDAO:

def __init__(self):

self.dbname = "postgres"

self.user = "postgres"

self.host = "localhost"

self.port = "5432"

self.password = Config.DATABASE_PASSWORD

def connect(self):

"""Establish a database connection."""

try:

self.connection = psycopg2.connect(

dbname=self.dbname,

user=self.user,

password=self.password,

host=self.host,

port=self.port

)

self.cursor = self.connection.cursor()

print("Database Connection Established.")

except Exception as error:

print(f"Error connecting to the database: {error}")

```
self.connection = None
```

```
self.cursor = None
```

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

```
    """Add a new account to the database using structured data."""
```

```
    try:
```

```
        # Combine DTO logic here by directly using the parameters
```

```
        query = "INSERT INTO accounts (username, password, website) VALUES (%s, %s, %s)"
```

```
        values = (username, password, website)
```

```
        self.cursor.execute(query, values)
```

```
        self.connection.commit()
```

```
        print(f"Account {username} added successfully.")
```

```
        return True
```

```
    except Exception as error:
```

```
        print(f"Error inserting account: {error}")
```

```
        return False
```

```
def fetch_account_by_website(self, website):
```

```
    """Fetch account credentials for a specific website."""
```

```
    try:
```

```
        query = "SELECT username, password FROM accounts WHERE LOWER(website) =  
LOWER(%s)"
```

```
        self.cursor.execute(query, (website,))
```

```
        result = self.cursor.fetchone()
```

```
        print(result)
```

```
        return result
```

except Exception as error:

```
print(f"Error fetching account for website {website}: {error}")
```

```
return None
```

def fetch_all_accounts(self):

```
    """Fetch all accounts from the database."""
```

```
    try:
```

```
        query = "SELECT id, username, password, website FROM accounts"
```

```
        self.cursor.execute(query)
```

```
        result = self.cursor.fetchall()
```

```
        print(result)
```

```
        return result
```

except Exception as error:

```
    print(f"Error fetching accounts: {error}")
```

```
    return []
```

def delete_account(self, account_id):

```
    """Delete an account by its ID."""
```

```
    try:
```

```
        self.cursor.execute("DELETE FROM accounts WHERE id = %s", (account_id,))
```

```
        self.connection.commit()
```

```
        if self.cursor.rowcount > 0: # Check if any rows were affected
```

```
            print(f"Account with ID {account_id} deleted successfully.")
```

```
            return True
```

```
    else:
```

```
        print(f"No account found with ID {account_id}.")
```

```
        return False
```

```
    except Exception as error:
```

```
        print(f"Error deleting account: {error}")
```

```
    return False
```

```
def reset_id_sequence(self):
```

```
    """Reset the ID sequence to the maximum ID."""
```

```
    try:
```

```
        reset_query = "SELECT setval('accounts_id_seq', (SELECT MAX(id) FROM accounts))"
```

```
        self.cursor.execute(reset_query)
```

```
        self.connection.commit()
```

```
        print("ID sequence reset successfully.")
```

```
    except Exception as error:
```

```
        print(f"Error resetting ID sequence: {error}")
```

```
def close(self):
```

```
    """Close the database connection."""
```

```
    try:
```

```
        if self.cursor:
```

```
            self.cursor.close()
```

```
        if self.connection:
```

```
            self.connection.close()
```

```
        print("Database connection closed.")
```

```
    except Exception as error:
```

```
        print(f"Error closing the database connection: {error}")
```

```
--- global_vars.py ---
```

```
import re
```

```
class GlobalState:
```

```
    user_message = 'default'
```

```
    @classmethod
```

```
    def reset_user_message(cls):
```

```
        """Reset the global user_message variable to None."""
```

```
        cls.user_message = None
```

```
    @classmethod
```

```
    def parse_user_message(cls, message):
```

```
        """
```

```
        Parses a user message by splitting it into command and up to 6 variables.
```

```
        Handles quoted substrings so that quoted parts (e.g., "October 2") remain intact.
```

```
        """
```

```
        #print(f"User_message before parsing: {message}")
```

```
        message = message.replace("!", "").strip() # Remove "!" and strip spaces
```

```
        #print(f"User_message after replacing '!' with empty string: {message}")
```

```
        # Simple split by spaces, keeping quoted substrings intact
```

```
        parts = re.findall(r"\"[^\"]+\"|\\S+", message)
```

```
        #print(f"Parts after splitting: {parts}")
```

```
        # Ensure we always return 6 variables (command + 5 parts), even if some are empty
```



```
result = [parts[i].strip("") if len(parts) > i else "" for i in range(6)] # List comprehension to handle  
missing parts
```

```
#print(f"Result: {result}")
```

```
return result # Return the list (or tuple if needed)
```

--- AvailabilityEntity.py ---

```
import asyncio
```

```
from entity.BrowserEntity import BrowserEntity
```

```
from utils.css_selectors import Selectors
```

```
from selenium.webdriver.common.by import By
```

```
from selenium.webdriver.support.ui import WebDriverWait
```

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

```
from utils.configuration import load_config
```

```
class AvailabilityEntity:
```

```
    config = load_config()
```

```
    search_element_timeOut = config.get('project_options', {}).get('search_element_timeOut', 15)
```

```
    sleep_time = config.get('project_options', {}).get('sleep_time', 15)
```

```
    def __init__(self):
```

```
        self.browser_entity = BrowserEntity()
```

```
    async def check_availability(self, url: str, date_str=None, timeout=search_element_timeOut):
```

```
        try:
```

```
            # Use BrowserEntity to navigate to the URL
```

```
            self.browser_entity.navigate_to_website(url)
```

```
            # Get selectors for the given URL
```

```
            selectors = Selectors.get_selectors_for_url(url)
```

```

# Perform date selection (optional)

if date_str:

    try:

        await asyncio.sleep(self.sleep_time) # Wait for updates to load

        print(selectors['date_field'])

        date_field = self.browser_entity.driver.find_element(By.CSS_SELECTOR,
selectors['date_field'])

        date_field.click()

        await asyncio.sleep(self.sleep_time)

        date_button = self.browser_entity.driver.find_element(By.CSS_SELECTOR,
f"{selectors['select_date']} button[aria-label*='{date_str}']")

        date_button.click()

    except Exception as e:

        return f"Failed to select the date: {str(e)}"

await asyncio.sleep(self.sleep_time) # Wait for updates to load

# Initialize flags for select_time and no_availability elements
select_time_seen = False
no_availability_seen = False

try:

    # Check if 'select_time' is available within the given timeout
    WebDriverWait(self.browser_entity.driver, timeout).until(

        EC.presence_of_element_located((By.CSS_SELECTOR, selectors['select_time']))

    )

```

```

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

# Logic to determine availability
if select_time_seen:
    return f"Selected or default date {date_str if date_str else 'current date'} is available for
booking."
elif no_availability_seen:
    return "No availability for the selected date."
else:
    return "Unable to determine availability. Please try again."

except Exception as e:
    return f"Failed to check availability: {str(e)}"

```

--- BrowserEntity.py ---

import asyncio

from selenium.webdriver.common.by import By

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected_conditions as EC

from selenium import webdriver

from selenium.webdriver.chrome.service import Service

from utils.configuration import load_config

from utils.css_selectors import Selectors

class BrowserEntity:

 _instance = None

 config = load_config()

 search_element_timeOut = config.get('project_options', {}).get('search_element_timeOut', 15)

 sleep_time = config.get('project_options', {}).get('sleep_time', 15)

 def __new__(cls, *args, **kwargs):

 if not cls._instance:

 cls._instance = super(BrowserEntity, cls).__new__(cls, *args, **kwargs)

 return cls._instance

 def __init__(self):

 self.driver = None

 self.browser_open = False

```
def set_browser_open(self, is_open: bool):
```

```
    self.browser_open = is_open
```

```
def is_browser_open(self) -> bool:
```

```
    return self.browser_open
```

```
def launch_browser(self):
```

```
    try:
```

```
        if not self.browser_open:
```

```
            options = webdriver.ChromeOptions()
```

```
            options.add_argument("--remote-debugging-port=9222")
```

```
            options.add_experimental_option("excludeSwitches", ["enable-automation"])
```

```
            options.add_experimental_option('useAutomationExtension', False)
```

```
            options.add_argument("--start-maximized")
```

```
            options.add_argument("--disable-notifications")
```

```
            options.add_argument("--disable-popup-blocking")
```

```
            options.add_argument("--disable-infobars")
```

```
            options.add_argument("--disable-extensions")
```

```
            options.add_argument("--disable-webgl")
```

```
            options.add_argument("--disable-webrtc")
```

```
            options.add_argument("--disable-rtc-smoothing")
```

```
self.driver = webdriver.Chrome(service=Service(), options=options)
```

```
self.browser_open = True
```

```
result = "Browser launched."
```

```
return result
```

```
else:
```

```
    result = "Browser is already running."
```

```
    return result
```

```
except Exception as e:
```

```
    result = f"BrowserEntity_Failed to launch browser: {str(e)}"
```

```
    return result
```

```
def close_browser(self):
```

```
    try:
```

```
        if self.browser_open and self.driver:
```

```
            self.driver.quit()
```

```
            self.browser_open = False
```

```
            return "Browser closed."
```

```
        else:
```

```
            return "No browser is currently open."
```

```
    except Exception as e:
```

```
        return f"BrowserEntity_Failed to close browser: {str(e)}"
```

```
def navigate_to_website(self, url):
```

```
    try:
```

```
        if not self.is_browser_open():
```

```
            launch_message = self.launch_browser()
```

```
if "Failed" in launch_message:
```

```
    return launch_message
```

```
if self.driver:
```

```
    self.driver.get(url)
```

```
    return f"Navigated to {url}"
```

```
else:
```

```
    return "Failed to open browser."
```

```
except Exception as e:
```

```
    return f"BrowserEntity_Failed to navigate to {url}: {str(e)}"
```

```
async def login(self, url, username, password):
```

```
    try:
```

```
        navigate_message = self.navigate_to_website(url)
```

```
        if "Failed" in navigate_message:
```

```
            return navigate_message
```

```
        email_field = self.driver.find_element(By.CSS_SELECTOR,
```

```
Selectors.get_selectors_for_url(url)['email_field'])
```

```
        email_field.send_keys(username)
```

```
        await asyncio.sleep(self.sleep_time)
```

```
        password_field = self.driver.find_element(By.CSS_SELECTOR,
```

```
Selectors.get_selectors_for_url(url)['password_field'])
```

```
        password_field.send_keys(password)
```

```
        await asyncio.sleep(self.sleep_time)
```



```

        sign_in_button = self.driver.find_element(By.CSS_SELECTOR,
Selectors.get_selectors_for_url(url)['SignIn_button'])

        sign_in_button.click()

        await asyncio.sleep(self.sleep_time)

WebDriverWait(self.driver,
self.search_element_timeout).until(EC.presence_of_element_located((By.CSS_SELECTOR,
Selectors.get_selectors_for_url(url)['homePage'])))

        return f"Logged in to {url} successfully with username: {username}"

except Exception as e:

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

```

--- PriceEntity.py ---

```
from selenium.webdriver.common.by import By
```

```
from entity.BrowserEntity import BrowserEntity
```

```
from utils.css_selectors import Selectors # Import selectors to get CSS selectors for the browser
```

```
class PriceEntity:
```

```
    """PriceEntity is responsible for interacting with the system (browser) to fetch prices
    and handle the exporting of data to Excel and HTML."""
```

```
    def __init__(self):
```

```
        self.browser_entity = BrowserEntity()
```

```
    def get_price_from_page(self, url: str):
```

```
        # Navigate to the URL using BrowserEntity
```

```
        self.browser_entity.navigate_to_website(url)
```

```
        selectors = Selectors.get_selectors_for_url(url)
```

```
        try:
```

```
            # Find the price element on the page using the selector
```

```
            price_element = self.browser_entity.driver.find_element(By.CSS_SELECTOR,
```

```
selectors['price'])
```

```
            result = price_element.text
```

```
            return result
```

```
        except Exception as e:
```

```
            return f"Error fetching price: {str(e)}"
```

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

```
#empty init file
```

```
--- test_init.py ---
```

```
import sys, os, logging, pytest, asyncio
```

```
import subprocess
```

```
from unittest.mock import patch, MagicMock
```

```
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
```

```
from utils.email_utils import send_email_with_attachments
```

```
from utils.exportUtils import ExportUtils
```

```
from control.BrowserControl import BrowserControl
```

```
from control.AccountControl import AccountControl
```

```
from control.AvailabilityControl import AvailabilityControl
```

```
from control.PriceControl import PriceControl
```

```
from control.BotControl import BotControl
```

```
from DataObjects.AccountDAO import AccountDAO
```

```
from entity.AvailabilityEntity import AvailabilityEntity
```

```
from entity.BrowserEntity import BrowserEntity
```

```
from entity.PriceEntity import PriceEntity
```

```
#pytest -v > test_results.txt
```

```
#Run this command in the terminal to save the test results to a file
```

```
async def run_monitoring_loop(control_object, check_function, url, date_str, frequency,
iterations=1):
```

```
    """Run the monitoring loop for a control object and execute a check function."""
```

```
    control_object.is_monitoring = True
```

```
    results = []
```

```
    while control_object.is_monitoring and iterations > 0:
```

try:

 result = await check_function(url, date_str)

except Exception as e:

 result = f"Failed to monitor: {str(e)}"

logging.info(f"Monitoring Iteration: {result}")

results.append(result)

iterations -= 1

await asyncio.sleep(frequency)

control_object.is_monitoring = False

results.append("Monitoring stopped successfully!")

return results

def setup_logging():

 """Set up logging without timestamp and other unnecessary information."""

 logger = logging.getLogger()

 if not logger.handlers():

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

def save_test_results_to_file(output_file="test_results.txt"):

 """Helper function to run pytest and save results to a file."""

 print("Running tests and saving results to file...")

 output_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), output_file)

 with open(output_path, 'w') as f:

```

# Use subprocess to call pytest and redirect output to file
subprocess.run(['pytest', '-v'], stdout=f, stderr=subprocess.STDOUT)

# Custom fixture for logging test start and end

@pytest.fixture(autouse=True)
def log_test_start_end(request):
    test_name = request.node.name

    logging.info(f"-----\nStarting test: {test_name}\n")

    # Yield control to the test function
    yield

    # Log after the test finishes
    logging.info(f"\nFinished test: {test_name}\n-----")

@pytest.fixture
def base_test_case():
    """Base test setup that can be used by all test functions."""
    test_case = MagicMock()
    test_case.browser_control = BrowserControl()
    test_case.account_control = AccountControl()
    test_case.availability_control = AvailabilityControl()
    test_case.price_control = PriceControl()
    test_case.bot_control = BotControl()
    test_case.account_dao = AccountDAO()

```

```
test_case.availability_entity = AvailabilityEntity()

test_case.browser_entity = BrowserEntity()

test_case.price_entity = PriceEntity()

test_case.email_dao = send_email_with_attachments

test_case.export_utils = ExportUtils()

return test_case
```

```
if __name__ == "__main__":
```

```
    # Save the pytest output to a file in the same folder
```

```
    save_test_results_to_file(output_file="test_results.txt")
```

```
--- unitTest_add_account.py ---
```

```
import pytest, os, sys
```

```
from unittest.mock import MagicMock
```

```
from test_init import setup_logging, base_test_case, save_test_results_to_file, log_test_start_end,  
logging
```

```
setup_logging() # Initialize logging if needed
```

```
@pytest.mark.usefixtures("base_test_case")
```

```
class TestAccountDAO:
```

```
    @pytest.fixture
```

```
    def account_dao(self, base_test_case, mocker):
```

```
        # Mock the psycopg2 connection and cursor
```

```
        mocker.patch('psycopg2.connect')
```

```
        account_dao = base_test_case.account_dao
```

```
        account_dao.connection = MagicMock()
```

```
        account_dao.cursor = MagicMock()
```

```
        logging.info("Fake database connection established")
```

```
        return account_dao
```

```
def test_entity_add_account_success(self, account_dao):
```

```
    # Setup the cursor's behavior for successful insertion
```

```
    account_dao.cursor.execute = MagicMock()
```

```
    account_dao.cursor.rowcount = 1
```

```
    account_dao.connection.commit = MagicMock()
```



```
# Test the add_account method for success
```

```
result = account_dao.add_account("test_user", "password123", "example.com")
```

```
# Log the result of the operation
```

```
logging.info(f"AccountDAO.add_account returned {result}")
```

```
logging.info("Expected result: True")
```

```
# Assert and log the final outcome
```

```
assert result == True, "Account should be added successfully"
```

```
logging.info("Test add_account_success passed")
```

```
def test_entity_add_account_fail(self, account_dao):
```

```
    # Setup the cursor's behavior to simulate a failure during insertion
```

```
    account_dao.cursor.execute.side_effect = Exception("Database error")
```

```
    account_dao.cursor.rowcount = 0
```

```
    account_dao.connection.commit = MagicMock()
```

```
    # Perform the test
```

```
    result = account_dao.add_account("fail_user", "fail123", "fail.com")
```

```
# Log the result of the operation
```

```
logging.info(f"AccountDAO.add_account returned {result}")
```

```
logging.info("Expected result: False")
```

```
# Assert and log the final outcome
```

```
assert result == False, "Account should not be added"
```

```
logging.info("Test add_account_fail passed")
```

```
@pytest.mark.usefixtures("base_test_case")
```

```
class TestAccountControl:
```

```
    @pytest.fixture
```

```
    def account_control(self, base_test_case, mocker):
```

```
        # Get the mocked AccountControl from base_test_case
```

```
        account_control = base_test_case.account_control
```

```
        account_control.account_dao = MagicMock(spec=base_test_case.account_dao)
```

```
        # Mock methods used in the control layer's add_account
```

```
        mocker.patch.object(account_control.account_dao, 'connect')
```

```
        mocker.patch.object(account_control.account_dao, 'close')
```

```
        logging.info("Mocked AccountDAO connection and close methods")
```

```
        return account_control
```

```
    def test_control_add_account_success(self, account_control):
```

```
        # Mock successful addition in the DAO layer
```

```
        account_control.account_dao.add_account.return_value = True
```

```
        # Call the control method and check the response
```

```
        result = account_control.add_account("test_user", "password123", "example.com")
```

```
expected_message = "Account for example.com added successfully."
```

```
# Log the response and expectations
```

```
logging.info(f"Control method add_account returned: '{result}'")
```

```
logging.info("Expected message: 'Account for example.com added successfully.'")
```

```
assert result == expected_message, "The success message should match expected output"
```

```
logging.info("Test control_add_account_success passed")
```

```
def test_control_add_account_fail(self, account_control):
```

```
    # Mock failure in the DAO layer
```

```
    account_control.account_dao.add_account.return_value = False
```

```
    # Call the control method and check the response
```

```
    result = account_control.add_account("fail_user", "fail123", "fail.com")
```

```
    expected_message = "Failed to add account for fail.com."
```

```
    # Log the response and expectations
```

```
    logging.info(f"Control method add_account returned: '{result}'")
```

```
    logging.info("Expected message: 'Failed to add account for fail.com.'")
```

```
    assert result == expected_message, "The failure message should match expected output"
```

```
    logging.info("Test control_add_account_fail passed")
```

```
if __name__ == "__main__":
```

```
    pytest.main([__file__]) # Run pytest directly
```

```
--- unitTest_check_availability.py ---
```

```
import pytest, logging
```

```
from unittest.mock import patch
```

```
from test_init import base_test_case, setup_logging, log_test_start_end
```

```
# Enable asyncio for all tests in this file
```

```
pytestmark = pytest.mark.asyncio
```

```
setup_logging()
```

```
# Test for successful availability check (Control and Entity Layers)
```

```
async def test_check_availability_success(base_test_case):
```

```
    with patch('entity.AvailabilityEntity.AvailabilityEntity.check_availability') as mock_check:
```

```
        url = "https://example.com"
```

```
        mock_check.return_value = f"Selected or default date current date is available for booking."
```

```
        expected_entity_result = f"Selected or default date current date is available for booking."
```

```
        expected_control_result = f"Checked availability: Selected or default date current date is  
available for booking."
```

```
# Execute the command
```

```
result = await base_test_case.availability_control.receive_command("check_availability", url)
```

```
# Log and assert the outcomes
```

```
logging.info(f"Entity Layer Expected: {expected_entity_result}")
```

```
logging.info(f"Entity Layer Received: {mock_check.return_value}")
```

```
assert mock_check.return_value == expected_entity_result, "Entity layer assertion failed."
```

```
logging.info("Unit Test Passed for entity layer.\n")
```

```
logging.info(f"Control Layer Expected: {expected_control_result}")

logging.info(f"Control Layer Received: {result}")

assert result == expected_control_result, "Control layer assertion failed."

logging.info("Unit Test Passed for control layer.")
```

Test for failure in entity layer (Control should handle it gracefully)

```
async def test_check_availability_failure_entity(base_test_case):

    with patch('entity.AvailabilityEntity.AvailabilityEntity.check_availability',
side_effect=Exception("Failed to check availability")) as mock_check:

        url = "https://example.com"

        expected_control_result = "Failed to check availability: Failed to check availability"


    # Execute the command

    result = await base_test_case.availability_control.receive_command("check_availability", url)


    # Log and assert the outcomes

    logging.info(f"Control Layer Expected: {expected_control_result}")

    logging.info(f"Control Layer Received: {result}")

    assert result == expected_control_result, "Control layer failed to handle entity error correctly."

    logging.info("Unit Test Passed for entity layer error handling.")
```

Test for no availability scenario (control and entity)

```
async def test_check_availability_no_availability(base_test_case):

    with patch('entity.AvailabilityEntity.AvailabilityEntity.check_availability') as mock_check:

        url = "https://example.com"
```

```
mock_check.return_value = "No availability for the selected date."
```

```
expected_control_result = "Checked availability: No availability for the selected date."
```

```
# Execute the command
```

```
result = await base_test_case.availability_control.receive_command("check_availability", url)
```

```
# Log and assert the outcomes
```

```
logging.info(f"Entity Layer Received: {mock_check.return_value}")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
    assert result == expected_control_result, "Control layer failed to handle no availability  
scenario."
```

```
logging.info("Unit Test Passed for control layer no availability handling.")
```

```
# Test for control layer failure scenario
```

```
async def test_check_availability_failure_control(base_test_case):
```

```
    with patch('control.AvailabilityControl.AvailabilityControl.receive_command',
```

```
side_effect=Exception("Control Layer Failed")) as mock_control:
```

```
    url = "https://example.com"
```

```
    expected_control_result = "Control Layer Exception: Control Layer Failed"
```

```
# Execute the command and catch the raised exception
```

```
try:
```

```
    result = await base_test_case.availability_control.receive_command("check_availability", url)
```

```
except Exception as e:
```

```
    result = f"Control Layer Exception: {str(e)}"
```

```
# Log and assert the outcomes
```

```
logging.info(f"Control Layer Expected: {expected_control_result}")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
assert result == expected_control_result, "Control layer assertion failed."
```

```
logging.info("Unit Test Passed for control layer failure.")
```

```
if __name__ == "__main__":
```

```
    pytest.main([__file__])
```



```
--- unitTest_close_browser.py ---
```

```
import pytest, logging
```

```
from unittest.mock import patch
```

```
from test_init import base_test_case, setup_logging, log_test_start_end
```

```
# Enable asyncio for all tests in this file
```

```
pytestmark = pytest.mark.asyncio
```

```
setup_logging()
```

```
async def test_close_browser_success(base_test_case):
```

```
    with patch('entity.BrowserEntity.BrowserEntity.close_browser') as mock_close:
```

```
        # Set up mock and expected outcomes
```

```
        mock_close.return_value = "Browser closed."
```

```
        expected_entity_result = "Browser closed."
```

```
        expected_control_result = "Control Object Result: Browser closed."
```

```
        # Execute the command
```

```
        result = await base_test_case.browser_control.receive_command("close_browser")
```

```
        # Log and assert the outcomes
```

```
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
```

```
        logging.info(f"Entity Layer Received: {mock_close.return_value}")
```

```
        assert mock_close.return_value == expected_entity_result, "Entity layer assertion failed."
```

```
        logging.info("Unit Test Passed for entity layer.\n")
```

```
        logging.info(f"Control Layer Expected: {expected_control_result}")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
assert result == expected_control_result, "Control layer assertion failed."
```

```
logging.info("Unit Test Passed for control layer.")
```

```
async def test_close_browser_not_open(base_test_case):
```

```
    with patch('entity.BrowserEntity.BrowserEntity.close_browser') as mock_close:
```

```
        # Set up mock and expected outcomes
```

```
        mock_close.return_value = "No browser is currently open."
```

```
        expected_entity_result = "No browser is currently open."
```

```
        expected_control_result = "Control Object Result: No browser is currently open."
```

```
        # Execute the command
```

```
        result = await base_test_case.browser_control.receive_command("close_browser")
```

```
        # Log and assert the outcomes
```

```
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
```

```
        logging.info(f"Entity Layer Received: {mock_close.return_value}")
```

```
        assert mock_close.return_value == expected_entity_result, "Entity layer assertion failed."
```

```
        logging.info("Unit Test Passed for entity layer.\n")
```

```
        logging.info(f"Control Layer Expected: {expected_control_result}")
```

```
        logging.info(f"Control Layer Received: {result}")
```

```
        assert result == expected_control_result, "Control layer assertion failed."
```

```
        logging.info("Unit Test Passed for control layer.")
```

```
async def test_close_browser_failure_control(base_test_case):
```

```

        with patch('entity.BrowserEntity.BrowserEntity.close_browser',
side_effect=Exception("Unexpected error")) as mock_close:

    # Set up expected outcome

    expected_result = "Control Layer Exception: Unexpected error"


    # Execute the command

    result = await base_test_case.browser_control.receive_command("close_browser")


    # Log and assert the outcomes

    logging.info(f"Control Layer Expected to Report: {expected_result}")

    logging.info(f"Control Layer Received: {result}")

    assert result == expected_result, "Control layer failed to handle or report the error correctly."

    logging.info("Unit Test Passed for control layer error handling.")

```

```

async def test_close_browser_failure_entity(base_test_case):

    with patch('entity.BrowserEntity.BrowserEntity.close_browser',
side_effect=Exception("BrowserEntity_Failed to close browser: Internal error")) as mock_close:

    # Set up expected outcome

    internal_error_message = "BrowserEntity_Failed to close browser: Internal error"

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


    # Execute the command

    result = await base_test_case.browser_control.receive_command("close_browser")


    # Log and assert the outcomes

    logging.info(f"Entity Layer Expected Failure: {internal_error_message}")

```

```
logging.info(f"Control Layer Received: {result}")
```

```
assert result == expected_control_result, "Control layer failed to report entity error correctly."
```

```
logging.info("Unit Test Passed for entity layer error handling.")
```

```
if __name__ == "__main__":
```

```
    pytest.main([__file__])
```

```
--- unitTest_delete_account.py ---
```

```
import pytest, os, sys
```

```
from unittest.mock import MagicMock
```

```
from test_init import setup_logging, base_test_case, save_test_results_to_file, log_test_start_end,  
logging
```

```
setup_logging() # Initialize logging if needed
```

```
@pytest.mark.usefixtures("base_test_case")
```

```
class TestAccountDAO:
```

```
    @pytest.fixture
```

```
    def account_dao(self, base_test_case, mocker):
```

```
        # Mock the psycopg2 connection and cursor
```

```
        mocker.patch('psycopg2.connect')
```

```
        account_dao = base_test_case.account_dao
```

```
        account_dao.connection = MagicMock()
```

```
        account_dao.cursor = MagicMock()
```

```
        logging.info("Fake database connection established")
```

```
        return account_dao
```

```
def test_entity_delete_account_success(self, account_dao):
```

```
    # Setup the cursor's behavior for successful deletion
```

```
    account_dao.cursor.execute = MagicMock()
```

```
    account_dao.cursor.rowcount = 1
```

```
    account_dao.connection.commit = MagicMock()
```

```
# Test the delete_account method for success

result = account_dao.delete_account(1)


# Log the result of the operation

logging.info(f"AccountDAO.delete_account returned {result}")

logging.info("Expected result: True")


# Assert and log the final outcome

assert result == True, "Account should be deleted successfully"

logging.info("Test delete_account_success passed")


def test_entity_delete_account_fail(self, account_dao):

    # Setup the cursor's behavior to simulate a failure during deletion

    account_dao.cursor.execute.side_effect = Exception("Database error")

    account_dao.cursor.rowcount = 0

    account_dao.connection.commit = MagicMock()


# Perform the test

result = account_dao.delete_account(9999)


# Log the result of the operation

logging.info(f"AccountDAO.delete_account returned {result}")

logging.info("Expected result: False")


# Assert and log the final outcome
```

```
assert result == False, "Account should not be deleted"
```

```
logging.info("Test delete_account_fail passed")
```

```
@pytest.mark.usefixtures("base_test_case")
```

```
class TestAccountControl:
```

```
    @pytest.fixture
```

```
    def account_control(self, base_test_case, mocker):
```

```
        # Get the mocked AccountControl from base_test_case
```

```
        account_control = base_test_case.account_control
```

```
        account_control.account_dao = MagicMock(spec=base_test_case.account_dao)
```

```
        # Mock methods used in the control layer's delete_account
```

```
        mocker.patch.object(account_control.account_dao, 'connect')
```

```
        mocker.patch.object(account_control.account_dao, 'close')
```

```
        logging.info("Mocked AccountDAO connection and close methods")
```

```
        return account_control
```

```
    def test_control_delete_account_success(self, account_control):
```

```
        # Mock successful deletion in the DAO layer
```

```
        account_control.account_dao.delete_account.return_value = True
```

```
        # Call the control method and check the response
```

```
        result = account_control.delete_account(1)
```

```
expected_message = "Account with ID 1 deleted successfully."
```

```
# Log the response and expectations
```

```
logging.info(f"Control method delete_account returned: '{result}'")
```

```
logging.info("Expected message: 'Account with ID 1 deleted successfully.'")
```

```
assert result == expected_message, "The success message should match expected output"
```

```
logging.info("Test control_delete_account_success passed")
```

```
def test_control_delete_account_fail(self, account_control):
```

```
    # Mock failure in the DAO layer
```

```
    account_control.account_dao.delete_account.return_value = False
```

```
    # Call the control method and check the response
```

```
    result = account_control.delete_account(9999)
```

```
    expected_message = "Failed to delete account with ID 9999."
```

```
    # Log the response and expectations
```

```
    logging.info(f"Control method delete_account returned: '{result}'")
```

```
    logging.info("Expected message: 'Failed to delete account with ID 9999.'")
```

```
    assert result == expected_message, "The failure message should match expected output"
```

```
    logging.info("Test control_delete_account_fail passed")
```



```
if __name__ == "__main__":
```

```
    pytest.main([__file__]) # Run pytest directly
```

```
--- unitTest_ExportData.py ---
```

```
import pandas as pd
```

```
import pytest
```

```
from unittest.mock import MagicMock, patch
```

```
from test_init import setup_logging, base_test_case, save_test_results_to_file, log_test_start_end,  
logging
```

```
# Initialize logging
```

```
setup_logging()
```

```
@pytest.mark.usefixtures("base_test_case")
```

```
class TestExportUtils:
```

```
    @pytest.fixture
```

```
    def setup_mocked_paths(self, mocker):
```

```
        mocker.patch('os.path.exists', return_value=False)
```

```
        mocker.patch('os.makedirs') # Mock directory creation
```

```
        mocker.patch('pandas.DataFrame.to_excel') # Mock the Excel export method
```

```
        mocker.patch('builtins.open', mocker.mock_open()) # Mock open for HTML writing
```

```
        logging.info("Mocks for os.path, os.makedirs, pandas.to_excel, and open set up successfully.")
```

```
    def test_positive_html_export(self, base_test_case, setup_mocked_paths):
```

```
        # Test positive case for HTML export
```

```
        result = base_test_case.export_utils.export_to_html("test_command", "http://example.com",  
"Success")
```

```

# Assert and log the result

assert "HTML file saved and updated" in result

logging.info(f"Result: {result}")

logging.info("Test positive HTML export passed successfully.")


def test_positive_excel_export(self, base_test_case, setup_mocked_paths):

    # Mock reading from Excel and test positive case for Excel export

    with patch('pandas.read_excel', return_value=pd.DataFrame(columns=["Timestamp",
"Command", "URL", "Result", "Entered Date", "Entered Time"])):

        result = base_test_case.export_utils.log_to_excel("test_command", "http://example.com",
"Success")


    # Assert and log the result

    assert "Data saved to Excel file" in result

    logging.info(f"Result: {result}")

    logging.info("Test positive Excel export passed successfully.")


def test_negative_html_export(self, base_test_case, setup_mocked_paths):

    # Simulate an error during HTML export by raising an exception

    with patch('builtins.open', side_effect=Exception("Failed to write HTML")):

        try:

            result = base_test_case.export_utils.export_to_html("test_command",
"http://example.com", "Success")

        except Exception as e:

            # Assert that the correct exception was raised and log the result

            assert str(e) == "Failed to write HTML"

```

```
logging.info(f"Expected exception caught: {str(e)}")
```

```
logging.info("Test negative HTML export passed with expected exception.")
```

```
def test_negative_excel_export(self, base_test_case, setup_mocked_paths):
```

```
    # Simulate an error during Excel export by raising an exception
```

```
    with patch('pandas.DataFrame.to_excel', side_effect=Exception("Failed to write Excel")):
```

```
        try:
```

```
            result = base_test_case.export_utils.log_to_excel("test_command", "http://example.com",
```

```
"Success")
```

```
        except Exception as e:
```

```
            # Assert that the correct exception was raised and log the result
```

```
            assert str(e) == "Failed to write Excel"
```

```
            logging.info(f"Expected exception caught: {str(e)}")
```

```
            logging.info("Test negative Excel export passed with expected exception.")
```

```
if __name__ == '__main__':
```

```
    logging.info("Starting pytest for TestExportUtils...")
```

```
    pytest.main([__file__])
```

```
--- unitTest_fetch_account_by_website.py ---
```

```
import pytest, os, sys
```

```
from unittest.mock import MagicMock
```

```
from test_init import setup_logging, base_test_case, save_test_results_to_file, log_test_start_end,  
logging
```

```
setup_logging() # Initialize logging if needed
```

```
@pytest.mark.usefixtures("base_test_case")
```

```
class TestAccountDAOFetchByWebsite:
```

```
    @pytest.fixture
```

```
    def account_dao(self, base_test_case, mocker):
```

```
        # Mock the psycopg2 connection and cursor
```

```
        mocker.patch('psycopg2.connect')
```

```
        account_dao = base_test_case.account_dao
```

```
        account_dao.connection = MagicMock()
```

```
        account_dao.cursor = MagicMock()
```

```
        logging.info("Fake database connection established")
```

```
        return account_dao
```

```
def test_entity_fetch_account_success(self, account_dao):
```

```
    # Setup the cursor's behavior for successful fetch
```

```
    account_dao.cursor.execute = MagicMock()
```

```
    account_dao.cursor.fetchone.return_value = ("test_user", "password123")
```

```
# Test the fetch_account_by_website method for success
```

```
result = account_dao.fetch_account_by_website("example.com")
```

```
# Log the result of the operation
```

```
logging.info(f"AccountDAO.fetch_account_by_website returned {result}")
```

```
logging.info("Expected result: ('test_user', 'password123')")
```

```
# Assert and log the final outcome
```

```
assert result == ("test_user", "password123"), "Account should be fetched successfully"
```

```
logging.info("Test fetch_account_success passed")
```

```
def test_entity_fetch_account_fail(self, account_dao):
```

```
    # Setup the cursor's behavior to simulate failure
```

```
    account_dao.cursor.execute = MagicMock()
```

```
    account_dao.cursor.fetchone.return_value = None
```

```
    # Perform the test
```

```
    result = account_dao.fetch_account_by_website("fail.com")
```

```
    # Log the result of the operation
```

```
    logging.info(f"AccountDAO.fetch_account_by_website returned {result}")
```

```
    logging.info("Expected result: None")
```

```
    # Assert and log the final outcome
```

```
    assert result is None, "No account should be fetched"
```

```
    logging.info("Test fetch_account_fail passed")
```

```
@pytest.mark.usefixtures("base_test_case")
```

```
class TestAccountControlFetchByWebsite:
```

```
    @pytest.fixture
```

```
    def account_control(self, base_test_case, mocker):
```

```
        # Get the mocked AccountControl from base_test_case
```

```
        account_control = base_test_case.account_control
```

```
        account_control.account_dao = MagicMock(spec=base_test_case.account_dao)
```

```
        # Mock methods used in the control layer's fetch_account_by_website
```

```
        mocker.patch.object(account_control.account_dao, 'connect')
```

```
        mocker.patch.object(account_control.account_dao, 'close')
```

```
        logging.info("Mocked AccountDAO connection and close methods")
```

```
        return account_control
```

```
    def test_control_fetch_account_success(self, account_control):
```

```
        # Mock successful fetch in the DAO layer
```

```
        account_control.account_dao.fetch_account_by_website.return_value = ("test_user",
```

```
"password123")
```

```
        # Call the control method and check the response
```

```
        result = account_control.fetch_account_by_website("example.com")
```

```
        expected_message = ("test_user", "password123")
```

```
# Log the response and expectations
```

```
logging.info(f"Control method fetch_account_by_website returned: '{result}'")
```

```
logging.info("Expected message: ('test_user', 'password123')")
```

```
# Assert the success message
```

```
assert result == expected_message, "The fetch result should match expected output"
```

```
logging.info("Test control_fetch_account_success passed")
```

```
def test_control_fetch_account_fail(self, account_control):
```

```
    # Mock failure in the DAO layer
```

```
    account_control.account_dao.fetch_account_by_website.return_value = None
```

```
    # Call the control method and check the response
```

```
    result = account_control.fetch_account_by_website("fail.com")
```

```
    expected_message = "No account found for fail.com."
```

```
# Log the response and expectations
```

```
logging.info(f"Control method fetch_account_by_website returned: '{result}'")
```

```
logging.info("Expected message: 'No account found for fail.com.'")
```

```
# Assert the failure message
```

```
assert result == expected_message, "The failure message should match expected output"
```

```
logging.info("Test control_fetch_account_fail passed")
```



```
if __name__ == "__main__":  
    pytest.main([__file__]) # Run pytest directly
```

```
--- unitTest_fetch_all_accounts.py ---
```

```
import pytest, os, sys
```

```
from unittest.mock import MagicMock
```

```
from test_init import setup_logging, base_test_case, save_test_results_to_file, log_test_start_end,  
logging
```

```
setup_logging() # Initialize logging if needed
```

```
@pytest.mark.usefixtures("base_test_case")
```

```
class TestAccountDAO:
```

```
    @pytest.fixture
```

```
    def account_dao(self, base_test_case, mocker):
```

```
        mocker.patch('psycopg2.connect')
```

```
        account_dao = base_test_case.account_dao
```

```
        account_dao.connection = MagicMock()
```

```
        account_dao.cursor = MagicMock()
```

```
        logging.info("Fake database connection established")
```

```
        return account_dao
```

```
def test_entity_fetch_all_accounts_success(self, account_dao):
```

```
    # Mock successful fetch operation
```

```
        mock_accounts = [(1, "test_user", "password123", "example.com"), (2, "test_user2",  
"password456", "example2.com")]
```

```
        account_dao.cursor.fetchall.return_value = mock_accounts
```

```
# Test fetch_all_accounts method
```

```
result = account_dao.fetch_all_accounts()
```

```
logging.info(f"AccountDAO.fetch_all_accounts returned {result}")
```

```
logging.info("Expected result: a list of accounts")
```

```
# Assert and log the final outcome
```

```
assert result == mock_accounts, "Should return a list of accounts"
```

```
logging.info("Test fetch_all_accounts_success passed")
```

```
def test_entity_fetch_all_accounts_fail(self, account_dao):
```

```
    # Mock failed fetch operation
```

```
    account_dao.cursor.fetchall.side_effect = Exception("Database error")
```

```
# Test fetch_all_accounts method
```

```
result = account_dao.fetch_all_accounts()
```

```
logging.info(f"AccountDAO.fetch_all_accounts returned {result}")
```

```
logging.info("Expected result: an empty list due to failure")
```

```
# Assert and log the final outcome
```

```
assert result == [], "Should return an empty list due to failure"
```

```
logging.info("Test fetch_all_accounts_fail passed")
```

```
@pytest.mark.usefixtures("base_test_case")
```

```
class TestAccountControl:
```

```
    @pytest.fixture
```

```
    def account_control(self, base_test_case, mocker):
```

```
        account_control = base_test_case.account_control
```

```
        account_control.account_dao = MagicMock(spec=base_test_case.account_dao)
```

```
        # Mock methods used in the control layer's fetch_all_accounts
```

```
        mocker.patch.object(account_control.account_dao, 'connect')
```

```
        mocker.patch.object(account_control.account_dao, 'close')
```

```
        logging.info("Mocked AccountDAO connection and close methods")
```

```
        return account_control
```

```
    def test_control_fetch_all_accounts_success(self, account_control):
```

```
        # Mock successful fetch in the DAO layer
```

```
        mock_accounts = [(1, "test_user", "password123", "example.com"), (2, "test_user2",  
"password456", "example2.com")]
```

```
        account_control.account_dao.fetch_all_accounts.return_value = mock_accounts
```

```
        # Call the control method and check the response
```

```
        result = account_control.fetch_all_accounts()
```

```
        expected_message = "Accounts:\nID: 1, Username: test_user, Password: password123,  
Website: example.com\nID: 2, Username: test_user2, Password: password456, Website:  
example2.com"
```

```
        logging.info(f"Control method fetch_all_accounts returned: '{result}'")
```

```
logging.info(f"Expected message: '{expected_message}')
```

```
# Assert and log the final outcome
```

```
assert result == expected_message, "The fetched accounts list should match expected output"
```

```
logging.info("Test control_fetch_all_accounts_success passed")
```

```
def test_control_fetch_all_accounts_fail(self, account_control):
```

```
    # Mock failed fetch in the DAO layer
```

```
    account_control.account_dao.fetch_all_accounts.return_value = []
```

```
    # Call the control method and check the response
```

```
    result = account_control.fetch_all_accounts()
```

```
    expected_message = "No accounts found."
```

```
    logging.info(f"Control method fetch_all_accounts returned: '{result}')
```

```
    logging.info(f"Expected message: '{expected_message}')
```

```
# Assert and log the final outcome
```

```
assert result == expected_message, "The message should indicate no accounts found"
```

```
logging.info("Test control_fetch_all_accounts_fail passed")
```

```
if __name__ == "__main__":
```

```
    pytest.main([__file__]) # Run pytest directly
```

```
--- unitTest_get_price.py ---

import pytest, logging

from unittest.mock import patch

from test_init import base_test_case, setup_logging, log_test_start_end


# Enable asyncio for all tests in this file

pytestmark = pytest.mark.asyncio

setup_logging()


async def test_get_price_success(base_test_case):

    # Simulate a successful price retrieval

    with patch('entity.PriceEntity.PriceEntity.get_price_from_page') as mock_get_price:

        url = "https://example.com/product"

        mock_get_price.return_value = "$199.99"

        expected_entity_result = "$199.99"

        expected_control_result = "$199.99"


    # Execute the command

    result = await base_test_case.price_control.receive_command("get_price", url)


    # Log and assert the outcomes

    logging.info(f"Entity Layer Expected: {expected_entity_result}")

    logging.info(f"Entity Layer Received: {mock_get_price.return_value}")

    assert mock_get_price.return_value == expected_entity_result, "Entity layer assertion failed."

    logging.info("Unit Test Passed for entity layer.\n")
```

```
logging.info(f"Control Layer Expected: {expected_control_result}")

logging.info(f"Control Layer Received: {result}")

assert result == expected_control_result, "Control layer assertion failed."

logging.info("Unit Test Passed for control layer.")
```

```
async def test_get_price_invalid_url(base_test_case):
```

```
    # Simulate an invalid URL case
```

```
    with patch('entity.PriceEntity.PriceEntity.get_price_from_page') as mock_get_price:
```

```
        invalid_url = "invalid_url"
```

```
        mock_get_price.return_value = "Error fetching price: Invalid URL"
```

```
        expected_control_result = "Error fetching price: Invalid URL"
```

```
    # Execute the command
```

```
    result = await base_test_case.price_control.receive_command("get_price", invalid_url)
```

```
    # Log and assert the outcomes
```

```
    logging.info(f"Control Layer Expected: {expected_control_result}")
```

```
    logging.info(f"Control Layer Received: {result}")
```

```
    assert result == expected_control_result, "Control layer assertion failed."
```

```
    logging.info("Unit Test Passed for control layer invalid URL handling.\n")
```

```
async def test_get_price_failure_entity(base_test_case):
```

```
    # Simulate an entity layer failure when fetching the price
```

```
    with patch('entity.PriceEntity.PriceEntity.get_price_from_page', side_effect=Exception("Failed to
fetch price")) as mock_get_price:
```

```
        url = "https://example.com/product"
```

```
expected_control_result = "Failed to fetch price: Failed to fetch price"
```

```
# Execute the command
```

```
result = await base_test_case.price_control.receive_command("get_price", url)
```

```
# Log and assert the outcomes
```

```
logging.info(f"Control Layer Expected: {expected_control_result}")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
assert result == expected_control_result, "Control layer failed to handle entity error correctly."
```

```
logging.info("Unit Test Passed for entity layer error handling.")
```

```
async def test_get_price_failure_control(base_test_case):
```

```
    # Simulate a control layer failure
```

```
    with patch('control.PriceControl.PriceControl.receive_command', side_effect=Exception("Control Layer Failed")) as mock_control:
```

```
        url = "https://example.com/product"
```

```
        expected_control_result = "Control Layer Exception: Control Layer Failed"
```

```
    # Execute the command and catch the raised exception
```

```
    try:
```

```
        result = await base_test_case.price_control.receive_command("get_price", url)
```

```
    except Exception as e:
```

```
        result = f"Control Layer Exception: {str(e)}"
```

```
# Log and assert the outcomes
```

```
logging.info(f"Control Layer Expected: {expected_control_result}")
```



```
logging.info(f"Control Layer Received: {result}")
```

```
assert result == expected_control_result, "Control layer assertion failed."
```

```
logging.info("Unit Test Passed for control layer failure.")
```

```
if __name__ == "__main__":
```

```
    pytest.main([__file__])
```

```
--- unitTest_launch_browser.py ---
```

```
import pytest, logging
```

```
from unittest.mock import patch
```

```
from test_init import base_test_case, log_test_start_end, setup_logging
```

```
# Enable asyncio for all tests in this file
```

```
pytestmark = pytest.mark.asyncio
```

```
setup_logging()
```

```
async def test_launch_browser_success(base_test_case):
```

```
    with patch('entity.BrowserEntity.BrowserEntity.launch_browser') as mock_launch:
```

```
        # Setup mock return and expected outcomes
```

```
        mock_launch.return_value = "Browser launched."
```

```
        expected_entity_result = "Browser launched."
```

```
        expected_control_result = "Control Object Result: Browser launched."
```

```
        # Execute the command
```

```
        result = await base_test_case.browser_control.receive_command("launch_browser")
```

```
        # Log and assert the outcomes
```

```
        logging.info(f"Entity Layer Expected: {expected_entity_result}")
```

```
        logging.info(f"Entity Layer Received: {mock_launch.return_value}")
```

```
        assert mock_launch.return_value == expected_entity_result, "Entity layer assertion failed."
```

```
        logging.info("Unit Test Passed for entity layer.\n")
```

```
        logging.info(f"Control Layer Expected: {expected_control_result}")
```

```
logging.info(f"Control Layer Received: {result}")

assert result == expected_control_result, "Control layer assertion failed."

logging.info("Unit Test Passed for control layer.")
```

```
async def test_launch_browser_already_running(base_test_case):

    with patch('entity.BrowserEntity.BrowserEntity.launch_browser', return_value="Browser is already
running.") as mock_launch:
```

```
        expected_entity_result = "Browser is already running."

        expected_control_result = "Control Object Result: Browser is already running."
```

```
        result = await base_test_case.browser_control.receive_command("launch_browser")
```

```
        logging.info(f"Entity Layer Expected: {expected_entity_result}")

        logging.info(f"Entity Layer Received: {mock_launch.return_value}")

        assert mock_launch.return_value == expected_entity_result, "Entity layer assertion failed."

        logging.info("Unit Test Passed for entity layer.\n")
```

```
        logging.info(f"Control Layer Expected: {expected_control_result}")

        logging.info(f"Control Layer Received: {result}")

        assert result == expected_control_result, "Control layer assertion failed."

        logging.info("Unit Test Passed for control layer.")
```

```
async def test_launch_browser_failure_control(base_test_case):

    with patch('entity.BrowserEntity.BrowserEntity.launch_browser', side_effect=Exception("Internal
error")) as mock_launch:
```

```
        expected_result = "Control Layer Exception: Internal error"
```

```
result = await base_test_case.browser_control.receive_command("launch_browser")
```

```
logging.info(f"Control Layer Expected to Report: {expected_result}")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
    assert result == expected_result, "Control layer failed to handle or report the entity error correctly."
```

```
logging.info("Unit Test Passed for control layer error handling.")
```

```
async def test_launch_browser_failure_entity(base_test_case):
```

```
    with patch('entity.BrowserEntity.BrowserEntity.launch_browser', side_effect=Exception("Failed to launch browser: Internal error")) as mock_launch:
```

```
        expected_control_result = "Control Layer Exception: Failed to launch browser: Internal error"
```

```
result = await base_test_case.browser_control.receive_command("launch_browser")
```

```
logging.info(f"Entity Layer Expected Failure: Failed to launch browser: Internal error")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
assert result == expected_control_result, "Control layer failed to report entity error correctly."
```

```
logging.info("Unit Test Passed for entity layer error handling.")
```

```
if __name__ == "__main__":
```

```
    pytest.main([__file__])
```

```
--- unitTest_login.py ---
```

```
import pytest
```

```
import logging
```

```
from unittest.mock import patch, MagicMock
```

```
from test_init import base_test_case, setup_logging, log_test_start_end
```

```
# Enable asyncio for all tests in this file
```

```
pytestmark = pytest.mark.asyncio
```

```
setup_logging()
```

```
async def test_login_success(base_test_case):
```

```
    """Test that the login is successful when valid credentials are provided."""
```

```
    # Patch methods
```

```
    with patch('entity.BrowserEntity.BrowserEntity.login') as mock_login:
```

```
        with patch('control.AccountControl.AccountControl.fetch_account_by_website') as
```

```
mock_fetch_account:
```

```
    # Setup mock return values
```

```
    mock_login.return_value = "Logged in to http://example.com successfully with username:
```

```
sample_username"
```

```
    mock_fetch_account.return_value = ("sample_username", "sample_password")
```

```
    expected_entity_result = "Logged in to http://example.com successfully with username:
```

```
sample_username"
```

```
    expected_control_result = f"Control Object Result: {expected_entity_result}"
```

```
# Execute the command
```

```
        result = await base_test_case.browser_control.receive_command("login",  
site="example.com")
```

```
# Assert results and logging
```

```
logging.info(f"Entity Layer Expected: {expected_entity_result}")
```

```
logging.info(f"Entity Layer Received: {mock_login.return_value}")
```

```
assert mock_login.return_value == expected_entity_result, "Entity layer assertion failed."
```

```
logging.info("Unit Test Passed for entity layer.\n")
```

```
logging.info(f"Control Layer Expected: {expected_control_result}")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
assert result == expected_control_result, "Control layer assertion failed."
```

```
logging.info("Unit Test Passed for control layer.")
```

```
async def test_login_no_account(base_test_case):
```

```
    """Test that the control layer handles the scenario where no account is found for the website."""
```

```
        with patch('control.AccountControl.AccountControl.fetch_account_by_website') as
```

```
mock_fetch_account:
```

```
    # Setup mock to return no account
```

```
    mock_fetch_account.return_value = None
```

```
    expected_result = "No account found for example.com"
```

```
# Execute the command
```

```
result = await base_test_case.browser_control.receive_command("login", site="example.com")
```

```
# Assert results and logging
```

```
logging.info(f"Control Layer Expected: {expected_result}")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
assert result == expected_result, "Control layer failed to handle missing account correctly."
```

```
logging.info("Unit Test Passed for missing account handling.")
```

```
async def test_login_entity_layer_failure(base_test_case):
```

```
    """Test that the control layer handles an exception raised in the entity layer."""
```

```
    with patch('entity.BrowserEntity.BrowserEntity.login') as mock_login:
```

```
        with patch('control.AccountControl.AccountControl.fetch_account_by_website') as
```

```
mock_fetch_account:
```

```
    # Setup mocks
```

```
    mock_login.side_effect = Exception("BrowserEntity_Failed to log in to http://example.com:  
Internal error")
```

```
    mock_fetch_account.return_value = ("sample_username", "sample_password")
```

```
    expected_result = "Control Layer Exception: BrowserEntity_Failed to log in to  
http://example.com: Internal error"
```

```
    # Execute the command
```

```
    result = await base_test_case.browser_control.receive_command("login",  
site="example.com")
```

```
    # Assert results and logging
```

```
    logging.info(f"Control Layer Expected: {expected_result}")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
assert result == expected_result, "Control layer failed to handle entity layer exception."
```

```
logging.info("Unit Test Passed for entity layer failure.")
```

```
async def test_login_control_layer_failure(base_test_case):
```

```
    """Test that the control layer handles an unexpected failure or exception."""
```

```
        with patch('control.AccountControl.AccountControl.fetch_account_by_website') as
```

```
mock_fetch_account:
```

```
    # Simulate an exception being raised in the control layer
```

```
    mock_fetch_account.side_effect = Exception("Control layer failure during account fetch.")
```

```
    expected_result = "Control Layer Exception: Control layer failure during account fetch."
```

```
    # Execute the command
```

```
    result = await base_test_case.browser_control.receive_command("login", site="example.com")
```

```
    # Assert results and logging
```

```
    logging.info(f"Control Layer Expected: {expected_result}")
```

```
    logging.info(f"Control Layer Received: {result}")
```

```
    assert result == expected_result, "Control layer failed to handle control layer exception."
```

```
    logging.info("Unit Test Passed for control layer failure handling.")
```

```
async def test_login_invalid_url(base_test_case):
```

```
    """Test that the control layer handles the scenario where the URL or selectors are not found."""
```

```
        with patch('control.AccountControl.AccountControl.fetch_account_by_website') as
```


mock_fetch_account:

with patch('utils.css_selectors.Selectors.get_selectors_for_url') as mock_get_selectors:

Setup mocks

mock_fetch_account.return_value = ("sample_username", "sample_password")

mock_get_selectors.return_value = {'url': None} # Simulate missing URL

expected_result = "URL for example not found."

Execute the command

result = await base_test_case.browser_control.receive_command("login", site="example")

Assert results and logging

logging.info(f"Control Layer Expected: {expected_result}")

logging.info(f"Control Layer Received: {result}")

assert result == expected_result, "Control layer failed to handle missing URL or selectors."

logging.info("Unit Test Passed for missing URL/selector handling.")

if __name__ == "__main__":

pytest.main([__file__])

```

--- unitTest_navigate_to_website.py ---

import pytest, logging

from unittest.mock import patch

from test_init import base_test_case, setup_logging, log_test_start_end


# Enable asyncio for all tests in this file

pytestmark = pytest.mark.asyncio

setup_logging()


async def test_navigate_to_website_success(base_test_case):

    with patch('entity.BrowserEntity.BrowserEntity.navigate_to_website') as mock_navigate:

        # Setup mock return and expected outcomes

        url = "https://example.com"

        mock_navigate.return_value = f"Navigated to {url}"

        expected_entity_result = f"Navigated to {url}"

        expected_control_result = f"Control Object Result: Navigated to {url}"


        # Execute the command

        result = await base_test_case.browser_control.receive_command("navigate_to_website",
site=url)


        # Log and assert the outcomes

        logging.info(f"Entity Layer Expected: {expected_entity_result}")

        logging.info(f"Entity Layer Received: {mock_navigate.return_value}")

        assert mock_navigate.return_value == expected_entity_result, "Entity layer assertion failed."

```

```
logging.info("Unit Test Passed for entity layer.\n")
```

```
logging.info(f"Control Layer Expected: {expected_control_result}")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
assert result == expected_control_result, "Control layer assertion failed."
```

```
logging.info("Unit Test Passed for control layer.")
```

```
async def test_navigate_to_website_invalid_url(base_test_case):
```

```
    with patch('entity.BrowserEntity.BrowserEntity.navigate_to_website') as mock_navigate:
```

```
        # Setup mock return and expected outcomes
```

```
        invalid_site = "invalid_site"
```

```
        mock_navigate.return_value = f"URL for {invalid_site} not found."
```

```
        expected_control_result = f"URL for {invalid_site} not found."
```

```
        # Execute the command
```

```
        result = await base_test_case.browser_control.receive_command("navigate_to_website",
site=invalid_site)
```

```
        # Log and assert the outcomes
```

```
logging.info(f"Control Layer Expected: {expected_control_result}")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
assert result == expected_control_result, "Control layer assertion failed."
```

```
logging.info("Unit Test Passed for control layer invalid URL handling.\n")
```

```

async def test_navigate_to_website_failure_entity(base_test_case):

    with patch('entity.BrowserEntity.BrowserEntity.navigate_to_website',
side_effect=Exception("Failed to navigate")) as mock_navigate:

        # Setup expected outcomes

        url = "https://example.com"

        expected_control_result = "Control Layer Exception: Failed to navigate"


        # Execute the command

        result = await base_test_case.browser_control.receive_command("navigate_to_website",
site=url)


        # Log and assert the outcomes

        logging.info(f"Control Layer Expected: {expected_control_result}")

        logging.info(f"Control Layer Received: {result}")

        assert result == expected_control_result, "Control layer failed to handle entity error correctly."

        logging.info("Unit Test Passed for entity layer error handling.")

```

```

async def test_navigate_to_website_launch_browser_on_failure(base_test_case):

    # This test simulates a scenario where the browser is not open and needs to be launched first.

    with patch('entity.BrowserEntity.BrowserEntity.is_browser_open', return_value=False), \
        patch('entity.BrowserEntity.BrowserEntity.launch_browser', return_value="Browser
launched."), \

        patch('entity.BrowserEntity.BrowserEntity.navigate_to_website') as mock_navigate:

        # Setup expected outcomes

```

```
url = "https://example.com"
```

```
mock_navigate.return_value = f"Navigated to {url}"
```

```
expected_control_result = f"Control Object Result: Navigated to {url}"
```

```
# Execute the command
```

```
result = await base_test_case.browser_control.receive_command("navigate_to_website",  
site=url)
```

```
# Log and assert the outcomes
```

```
logging.info(f"Control Layer Expected: {expected_control_result}")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
assert result == expected_control_result, "Control layer assertion failed."
```

```
logging.info("Unit Test Passed for control layer with browser launch.\n")
```

```
async def test_navigate_to_website_failure_control(base_test_case):
```

```
# This simulates a failure within the control layer
```

```
with patch('control.BrowserControl.BrowserControl.receive_command',  
side_effect=Exception("Control Layer Failed")) as mock_control:
```

```
# Setup expected outcomes
```

```
url = "https://example.com"
```

```
expected_control_result = "Control Layer Exception: Control Layer Failed"
```

```
# Execute the command and catch the raised exception
```

```
try:
```

```
        result = await base_test_case.browser_control.receive_command("navigate_to_website",
site=url)

    except Exception as e:

        result = f"Control Layer Exception: {str(e)}"

# Log and assert the outcomes

logging.info(f"Control Layer Expected: {expected_control_result}")

logging.info(f"Control Layer Received: {result}")

assert result == expected_control_result, "Control layer assertion failed."

logging.info("Unit Test Passed for control layer failure.")

if __name__ == "__main__":

    pytest.main([__file__])
```

```
--- unitTest_project_help.py ---
```

```
import pytest, logging
```

```
from unittest.mock import patch
```

```
from test_init import base_test_case, setup_logging, log_test_start_end
```

```
# Enable asyncio for all tests in this file
```

```
pytestmark = pytest.mark.asyncio
```

```
setup_logging()
```

```
async def test_project_help_success(base_test_case):
```

```
    with patch('control.BotControl.BotControl.receive_command') as mock_help:
```

```
        # Setup mock return and expected outcomes
```

```
        mock_help.return_value = (
```

```
            "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"
```

```
)
```

```
expected_result = mock_help.return_value
```

```
# Execute the command
```

```
result = await base_test_case.bot_control.receive_command("project_help")
```

```
# Log and assert the outcomes
```

```
logging.info(f"Control Layer Expected: {expected_result}")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
assert result == expected_result, "Control layer assertion failed."
```

```
logging.info("Unit Test Passed for project help.\n")
```

```
async def test_project_help_failure(base_test_case):
```

```
    with patch('control.BotControl.BotControl.receive_command', side_effect=Exception("Error  
handling help command")) as mock_help:
```

```
        expected_result = "Error handling help command: Error handling help command"
```

```
# Execute the command and catch the raised exception
```

```
try:
```

```
    result = await base_test_case.bot_control.receive_command("project_help")
```



```
except Exception as e:
```

```
    result = f"Error handling help command: {str(e)}"
```

```
# Log and assert the outcomes
```

```
logging.info(f"Control Layer Expected: {expected_result}")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
assert result == expected_result, "Control layer failed to handle error correctly."
```

```
logging.info("Unit Test Passed for error handling in project help.\n")
```

```
if __name__ == "__main__":
```

```
    pytest.main([__file__])
```

```
--- unitTest_receive_email.py ---
```

```
import pytest
```

```
from unittest.mock import MagicMock
```

```
from test_init import setup_logging, base_test_case, save_test_results_to_file, log_test_start_end,  
logging
```

```
setup_logging()
```

```
@pytest.mark.usefixtures("base_test_case")
```

```
class TestEmailDAO:
```

```
    @pytest.fixture
```

```
    def email_dao(self, base_test_case, mocker):
```

```
        # Use the send_email_with_attachments from base_test_case
```

```
        email_dao = base_test_case.email_dao
```

```
        mocker.patch('smtpplib.SMTP')
```

```
        logging.info("Mocked EmailDAO with send_email_with_attachments method")
```

```
        return email_dao
```

```
    def test_entity_send_email_success(self, email_dao):
```

```
        # Mock successful email sending
```

```
        email_dao.return_value = "Email with file 'monitor_price.html' sent successfully!"
```

```
        # Perform the test
```

```
        result = email_dao('monitor_price.html')
```

```
# Log and assert the result
```

```
assert result == "Email with file 'monitor_price.html' sent successfully!"
```

```
logging.info("Test send_email_success passed")
```

```
def test_entity_send_email_fail(self, email_dao):
```

```
# Mock failure in email sending
```

```
email_dao.return_value = "File 'non_existent_file.html' not found."
```

```
# Perform the test
```

```
result = email_dao('non_existent_file.html')
```

```
# Log and assert the result
```

```
assert result == "File 'non_existent_file.html' not found in either excelFiles or htmlFiles."
```

```
logging.info("Test send_email_fail passed")
```

```
@pytest.mark.usefixtures("base_test_case")
```

```
class TestEmailControl:
```

```
@pytest.fixture
```

```
def email_control(self, base_test_case, mocker):
```

```
# Get the bot control from base_test_case, which should handle the receive_command method
```

```
email_control = base_test_case.bot_control
```

```
email_control.receive_command = MagicMock() # Mock the receive_command method
```

```
logging.info("Mocked EmailControl (BotControl) for control layer")
```

```
return email_control
```

```
def test_control_send_email_success(self, email_control):

    # Mock successful email sending

    email_control.receive_command.return_value = "Email with file 'monitor_price.html' sent successfully!"


    # Call the control method and check the response

    result = email_control.receive_command("receive_email", "monitor_price.html")


    # Log and assert the result

    assert result == "Email with file 'monitor_price.html' sent successfully!"

    logging.info("Test control_send_email_success passed")


def test_control_send_email_fail(self, email_control):

    # Mock failure in email sending

    email_control.receive_command.return_value = "File 'non_existent_file.html' not found."


    # Call the control method and check the response

    result = email_control.receive_command("receive_email", "non_existent_file.html")


    # Log and assert the result

    assert result == "File 'non_existent_file.html' not found."

    logging.info("Test control_send_email_fail passed")
```

```
if __name__ == "__main__":
```

```
    pytest.main([__file__]) # Run pytest directly
```

```
--- unitTest_start_monitoring_availability.py ---
```

```
import pytest, logging
```

```
from unittest.mock import patch
```

```
from test_init import base_test_case, setup_logging, run_monitoring_loop, log_test_start_end
```

```
import asyncio
```

```
# Enable asyncio for all tests in this file
```

```
pytestmark = pytest.mark.asyncio
```

```
setup_logging()
```

```
async def test_start_monitoring_availability_success(base_test_case):
```

```
    with patch('entity.AvailabilityEntity.AvailabilityEntity.check_availability') as mock_check:
```

```
        url = "https://example.com"
```

```
        mock_check.return_value = "Selected or default date is available for booking."
```

```
        expected_control_result = [
```

```
            "Checked availability: Selected or default date is available for booking.",
```

```
            "Monitoring stopped successfully!"
```

```
        ]
```

```
# Run the monitoring loop once
```

```
actual_control_result = await run_monitoring_loop(
```

```
    base_test_case.availability_control,
```

```
    base_test_case.availability_control.check_availability,
```

```
    url,
```

```
    "2024-10-01",
```

1

)

```
logging.info(f"Control Layer Expected: {expected_control_result}")
```

```
logging.info(f"Control Layer Received: {actual_control_result}")
```

```
assert actual_control_result == expected_control_result, "Control layer assertion failed."
```

```
logging.info("Unit Test Passed for control layer.")
```

```
async def test_start_monitoring_availability_failure_entity(base_test_case):
```

```
    with patch('entity.AvailabilityEntity.AvailabilityEntity.check_availability',
```

```
side_effect=Exception("Failed to check availability")):
```

```
    url = "https://example.com"
```

```
    expected_control_result = [
```

```
        "Failed to check availability: Failed to check availability",
```

```
        "Monitoring stopped successfully!"
```

```
    ]
```

```
# Run the monitoring loop once
```

```
actual_control_result = await run_monitoring_loop(
```

```
    base_test_case.availability_control,
```

```
    base_test_case.availability_control.check_availability,
```

```
    url,
```

```
    "2024-10-01",
```

```
    1
```

```
)
```

```

logging.info(f"Control Layer Expected: {expected_control_result}")

logging.info(f"Control Layer Received: {actual_control_result}")

    assert actual_control_result == expected_control_result, "Control layer failed to handle entity
error correctly."

logging.info("Unit Test Passed for entity layer error handling.")

```

```

async def test_start_monitoring_availability_failure_control(base_test_case):

    with patch('control.AvailabilityControl.AvailabilityControl.receive_command',
side_effect=Exception("Control Layer Failed")):

        url = "https://example.com"

        expected_control_result = "Control Layer Exception: Control Layer Failed"

        try:

            result = await

base_test_case.availability_control.receive_command("start_monitoring_availability", url,
"2024-10-01", 5)

        except Exception as e:

            result = f"Control Layer Exception: {str(e)}"

        logging.info(f"Control Layer Expected: {expected_control_result}")

        logging.info(f"Control Layer Received: {result}")

        assert result == expected_control_result, "Control layer assertion failed."

        logging.info("Unit Test Passed for control layer failure.")

```



```

async def test_start_monitoring_availability_already_running(base_test_case):

    with patch('entity.AvailabilityEntity.AvailabilityEntity.check_availability') as mock_check:

        url = "https://example.com"

        base_test_case.availability_control.is_monitoring = True

        expected_control_result = "Already monitoring availability."

        result = await

base_test_case.availability_control.receive_command("start_monitoring_availability", url,

"2024-10-01", 5)

        logging.info(f"Control Layer Expected: {expected_control_result}")

        logging.info(f"Control Layer Received: {result}")

        assert result == expected_control_result, "Control layer failed to handle already running

condition."

        logging.info("Unit Test Passed for control layer already running handling.\n")

if __name__ == "__main__":

    pytest.main([__file__])

```

```
--- unitTest_start_monitoring_price.py ---
```

```
import pytest
```

```
import logging
```

```
from unittest.mock import patch, AsyncMock
```

```
from test_init import base_test_case, setup_logging, log_test_start_end
```

```
# Enable asyncio for all tests in this file
```

```
pytestmark = pytest.mark.asyncio
```

```
setup_logging()
```

```
async def test_start_monitoring_price_success(base_test_case):
```

```
    with patch('entity.PriceEntity.PriceEntity.get_price_from_page', return_value="100 USD") as  
mock_get_price:
```

```
    # Setup expected outcomes
```

```
    url = "https://example.com/product"
```

```
    expected_result = "Starting price monitoring. Current price: 100 USD"
```

```
    # Mocking the sleep method to break out of the loop after the first iteration
```

```
    with patch('asyncio.sleep', side_effect=KeyboardInterrupt):
```

```
        try:
```

```
            # Execute the command
```

```
            base_test_case.price_control.is_monitoring = False
```

```
            result = await base_test_case.price_control.receive_command("start_monitoring_price",  
url, 1)
```

except KeyboardInterrupt:

Force the loop to stop after the first iteration

base_test_case.price_control.is_monitoring = False

Log and assert the outcomes

logging.info(f"Entity Layer Expected: {expected_result}")

logging.info(f"Control Layer Received: {base_test_case.price_control.results[0]}")

assert expected_result in base_test_case.price_control.results[0], "Price monitoring did not start as expected."

logging.info("Unit Test Passed for start_monitoring_price success scenario.\n")

async def test_start_monitoring_price_already_running(base_test_case):

Test when price monitoring is already running

base_test_case.price_control.is_monitoring = True

expected_result = "Already monitoring prices."

Execute the command

result = await base_test_case.price_control.receive_command("start_monitoring_price",
"https://example.com/product", 1)

Log and assert the outcomes

logging.info(f"Control Layer Expected: {expected_result}")

logging.info(f"Control Layer Received: {result}")

assert result == expected_result, "Control layer did not detect that monitoring was already running."

```
logging.info("Unit Test Passed for already running scenario.\n")
```

```
async def test_start_monitoring_price_failure_in_entity(base_test_case):
```

```
    # Mock entity failure during price fetching
```

```
        with patch('entity.PriceEntity.PriceEntity.get_price_from_page', side_effect=Exception("Error fetching price")) as mock_get_price:
```

```
            # Setup expected outcomes
```

```
            url = "https://example.com/product"
```

```
            expected_result = "Starting price monitoring. Current price: Failed to fetch price: Error fetching price"
```

```
            # Mocking the sleep method to break out of the loop after the first iteration
```

```
            with patch('asyncio.sleep', side_effect=KeyboardInterrupt):
```

```
                try:
```

```
                    # Execute the command
```

```
                    base_test_case.price_control.is_monitoring = False
```

```
                    await base_test_case.price_control.receive_command("start_monitoring_price", url, 1)
```

```
                except KeyboardInterrupt:
```

```
                    # Force the loop to stop after the first iteration
```

```
                    base_test_case.price_control.is_monitoring = False
```

```
            # Log and assert the outcomes
```

```
            logging.info(f"Control Layer Expected: {expected_result}")
```

```
            logging.info(f"Control Layer Received: {base_test_case.price_control.results[-1]}")
```

```
    assert expected_result in base_test_case.price_control.results[-1], "Entity layer did not handle failure correctly."
```

```
    logging.info("Unit Test Passed for entity layer failure scenario.\n")
```

```
async def test_start_monitoring_price_failure_in_control(base_test_case):
```

```
    # Mock control layer failure
```

```
        with patch('control.PriceControl.PriceControl.start_monitoring_price',
```

```
side_effect=Exception("Control Layer Exception")) as mock_start_monitoring:
```

```
    # Setup expected outcomes
```

```
    expected_result = "Control Layer Exception"
```

```
    # Execute the command and catch the raised exception
```

```
    try:
```

```
        result = await base_test_case.price_control.receive_command("start_monitoring_price",
```

```
"https://example.com/product", 1)
```

```
    except Exception as e:
```

```
        result = f"Control Layer Exception: {str(e)}"
```

```
    # Log and assert the outcomes
```

```
    logging.info(f"Control Layer Expected: {expected_result}")
```

```
    logging.info(f"Control Layer Received: {result}")
```

```
    assert expected_result in result, "Control layer did not handle the failure correctly."
```

```
    logging.info("Unit Test Passed for control layer failure scenario.\n")
```

```
if __name__ == "__main__":  
    pytest.main([__file__])
```

```
--- unitTest_stop_bot.py ---
```

```
import pytest
```

```
import logging
```

```
from unittest.mock import MagicMock, patch
```

```
from test_init import base_test_case, setup_logging, log_test_start_end
```

```
# Enable asyncio for all tests in this file
```

```
pytestmark = pytest.mark.asyncio
```

```
setup_logging()
```

```
async def test_stop_bot_success(base_test_case):
```

```
    with patch('control.BotControl.BotControl.receive_command') as mock_stop_bot:
```

```
        # Setup mock return and expected outcomes
```

```
        mock_stop_bot.return_value = "Bot has been shut down."
```

```
        expected_entity_result = "Bot has been shut down."
```

```
        expected_control_result = "Bot has been shut down."
```

```
        # Execute the command
```

```
        result = await base_test_case.bot_control.receive_command("stop_bot", ctx=MagicMock())
```

```
        # Log and assert the outcomes
```

```
        logging.info(f"Control Layer Expected: {expected_control_result}")
```

```
        logging.info(f"Control Layer Received: {result}")
```

```
        assert result == expected_control_result, "Control layer assertion failed."
```

```
        logging.info("Unit Test Passed for control layer stop bot.\n")
```

```

async def test_stop_bot_failure_control(base_test_case):

    with patch('control.BotControl.BotControl.receive_command', side_effect=Exception("Control
Layer Failed")) as mock_control:

        # Setup expected outcomes

        expected_control_result = "Control Layer Exception: Control Layer Failed"


        # Execute the command and catch the raised exception

        try:

            result = await base_test_case.bot_control.receive_command("stop_bot", ctx=MagicMock())

        except Exception as e:

            result = f"Control Layer Exception: {str(e)}"


        # Log and assert the outcomes

        logging.info(f"Control Layer Expected: {expected_control_result}")

        logging.info(f"Control Layer Received: {result}")

        assert result == expected_control_result, "Control layer assertion failed."

        logging.info("Unit Test Passed for control layer failure.\n")


if __name__ == "__main__":

    pytest.main([__file__])

```



```
--- unitTest_stop_monitoring_availability.py ---
```

```
import pytest, logging
```

```
from unittest.mock import patch
```

```
from test_init import base_test_case, setup_logging, log_test_start_end
```

```
import asyncio
```

```
# Enable asyncio for all tests in this file
```

```
pytestmark = pytest.mark.asyncio
```

```
setup_logging()
```

```
async def test_stop_monitoring_availability_success(base_test_case):
```

```
    # Simulate the case where monitoring is already running
```

```
    base_test_case.availability_control.is_monitoring = True
```

```
    base_test_case.availability_control.results = ["Checked availability: Selected or default date is  
available for booking."]
```

```
    # Expected message to be present in the result
```

```
    expected_control_result_contains = "Monitoring stopped successfully!"
```

```
    # Execute the stop command
```

```
    result = base_test_case.availability_control.stop_monitoring_availability()
```

```
    # Log and assert the outcomes
```

```
    logging.info(f"Control Layer Expected to contain: {expected_control_result_contains}")
```

```
    logging.info(f"Control Layer Received: {result}")
```

```
    assert expected_control_result_contains in result, "Control layer assertion failed for stop monitoring."
```

```
    logging.info("Unit Test Passed for stop monitoring availability.")
```

```
async def test_stop_monitoring_availability_no_active_session(base_test_case):
```

```
    # Simulate the case where no monitoring session is active
```

```
    base_test_case.availability_control.is_monitoring = False
```

```
    expected_control_result = "There was no active availability monitoring session. Nothing to stop."
```

```
    # Execute the stop command
```

```
    result = base_test_case.availability_control.stop_monitoring_availability()
```

```
    # Log and assert the outcomes
```

```
    logging.info(f"Control Layer Expected: {expected_control_result}")
```

```
    logging.info(f"Control Layer Received: {result}")
```

```
    assert result == expected_control_result, "Control layer assertion failed for no active session."
```

```
    logging.info("Unit Test Passed for stop monitoring with no active session.")
```

```
if __name__ == "__main__":
```

```
    pytest.main([__file__])
```

```
--- unitTest_stop_monitoring_price.py ---
```

```
import pytest
```

```
import logging
```

```
from unittest.mock import patch, AsyncMock
```

```
from test_init import base_test_case, setup_logging, log_test_start_end
```

```
# Enable asyncio for all tests in this file
```

```
pytestmark = pytest.mark.asyncio
```

```
setup_logging()
```

```
async def test_stop_monitoring_price_success(base_test_case):
```

```
    # Set up monitoring to be active
```

```
    base_test_case.price_control.is_monitoring = True
```

```
    base_test_case.price_control.results = ["Price went up!", "Price went down!"]
```

```
    # Expected result after stopping monitoring
```

```
    expected_result = "Results for price monitoring:\nPrice went up!\nPrice went down!\n\nPrice  
monitoring stopped successfully!"
```

```
    # Execute the command
```

```
    result = base_test_case.price_control.stop_monitoring_price()
```

```
    # Log and assert the outcomes
```

```
    logging.info(f"Control Layer Expected: {expected_result}")
```

```
    logging.info(f"Control Layer Received: {result}")
```

```
    assert result == expected_result, "Control layer did not return the correct results for stopping
```

monitoring."

```
logging.info("Unit Test Passed for stop_monitoring_price success scenario.\n")
```

```
async def test_stop_monitoring_price_not_active(base_test_case):
```

```
    # Test the case where monitoring is not active
```

```
    base_test_case.price_control.is_monitoring = False
```

```
    expected_result = "There was no active price monitoring session. Nothing to stop."
```

```
    # Execute the command
```

```
    result = base_test_case.price_control.stop_monitoring_price()
```

```
    # Log and assert the outcomes
```

```
    logging.info(f"Control Layer Expected: {expected_result}")
```

```
    logging.info(f"Control Layer Received: {result}")
```

```
    assert result == expected_result, "Control layer did not detect that monitoring was not active."
```

```
    logging.info("Unit Test Passed for stop_monitoring_price when not active.\n")
```

```
async def test_stop_monitoring_price_failure_in_control(base_test_case):
```

```
    # Simulate failure in control layer during stopping of monitoring
```

```
    with patch('control.PriceControl.PriceControl.stop_monitoring_price', side_effect=Exception("Error  
stopping price monitoring")) as mock_stop_monitoring:
```

```
        # Expected result when the control layer fails
```

```
        expected_result = "Error stopping price monitoring"
```

```
# Execute the command and handle exception
```

```
try:
```

```
    result = base_test_case.price_control.stop_monitoring_price()
```

```
except Exception as e:
```

```
    result = str(e)
```

```
# Log and assert the outcomes
```

```
logging.info(f"Control Layer Expected: {expected_result}")
```

```
logging.info(f"Control Layer Received: {result}")
```

```
assert expected_result in result, "Control layer did not handle the failure correctly."
```

```
logging.info("Unit Test Passed for stop_monitoring_price failure scenario.\n")
```

```
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": "#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
```



```
--- email_utils.py ---
```

```
# email_utils.py
```

```
import smtplib
```

```
from email.mime.multipart import MIMEMultipart
```

```
from email.mime.text import MIMEText
```

```
from email.mime.base import MIMEBase
```

```
from email import encoders
```

```
import os
```

```
from utils.Config import Config
```

```
def send_email_with_attachments(file_name=None):
```

```
    try:
```

```
        # Setup the MIME
```

```
        msg = MIMEMultipart()
```

```
        msg['From'] = Config.EMAIL_USER
```

```
        msg['To'] = Config.EMAIL_RECEIVER
```

```
        msg['Subject'] = "Exported Files from Discord Bot"
```

```
        # Body of the email
```

```
        body = "Attached is the exported file you requested."
```

```
        msg.attach(MIMEText(body, 'plain'))
```

```
        # Check if a specific file was requested
```

```
        if file_name:
```

```
            file_path = None
```

```
            # Search in both directories
```

```
for folder in ['excelFiles', 'htmlFiles']:
```

```
    possible_path = os.path.join('./ExportedFiles', folder, file_name)
```

```
    if os.path.exists(possible_path):
```

```
        file_path = possible_path
```

```
        break
```

```
if not file_path:
```

```
    return f"File '{file_name}' not found in either excelFiles or htmlFiles."
```

```
# Attach the requested file
```

```
attachment = open(file_path, "rb")
```

```
part = MIMEBase('application', 'octet-stream')
```

```
part.set_payload(attachment.read())
```

```
encoders.encode_base64(part)
```

```
part.add_header('Content-Disposition', f"attachment; filename= {file_name}")
```

```
msg.attach(part)
```

```
attachment.close()
```

```
else:
```

```
    return "Please specify a file to send."
```

```
# Send the email
```

```
server = smtplib.SMTP(Config.EMAIL_HOST, Config.EMAIL_PORT)
```

```
server.starttls()
```

```
server.login(Config.EMAIL_USER, Config.EMAIL_PASSWORD)
```

```
text = msg.as_string()
```

```
server.sendmail(Config.EMAIL_USER, Config.EMAIL_RECEIVER, text)
```

```
server.quit()
```

```
    return f"Email with file '{file_name}' sent successfully!"
```

```
except Exception as e:
```

```
    return f"Failed to send email: {str(e)}"
```

```
--- exportUtils.py ---
```

```
import os
```

```
import pandas as pd
```

```
from datetime import datetime
```

```
class ExportUtils:
```

```
    @staticmethod
```

```
    def log_to_excel(command, url, result, entered_date=None, entered_time=None):
```

```
        # Determine the file path for the Excel file
```

```
        file_name = f"{command}.xlsx"
```

```
        file_path = os.path.join("ExportedFiles", "excelFiles", file_name)
```

```
        # Ensure directory exists
```

```
        os.makedirs(os.path.dirname(file_path), exist_ok=True)
```

```
        # Timestamp for current run
```

```
        timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
```

```
        # If date/time not entered, use current timestamp
```

```
        entered_date = entered_date or datetime.now().strftime('%Y-%m-%d')
```

```
        entered_time = entered_time or datetime.now().strftime('%H:%M:%S')
```

```
        # Check if the file exists and create the structure if it doesn't
```

```
        if not os.path.exists(file_path):
```

```
            df = pd.DataFrame(columns=["Timestamp", "Command", "URL", "Result", "Entered Date",
```

```
"Entered Time"])
```

```
df.to_excel(file_path, index=False)
```

```
# Load existing data from the Excel file
```

```
df = pd.read_excel(file_path)
```

```
# Append the new row
```

```
new_row = {
```

```
    "Timestamp": timestamp,
```

```
    "Command": command,
```

```
    "URL": url,
```

```
    "Result": result,
```

```
    "Entered Date": entered_date,
```

```
    "Entered Time": entered_time
```

```
}
```

```
# Add the new row to the existing data and save it back to Excel
```

```
df = pd.concat([df, pd.DataFrame([new_row])], ignore_index=True)
```

```
df.to_excel(file_path, index=False)
```

```
return f"Data saved to Excel file at {file_path}."
```

```
@staticmethod
```

```
def export_to_html(command, url, result, entered_date=None, entered_time=None):
```

```
    """Export data to HTML format with the same structure as Excel."""
```

```
# Define file path for HTML

file_name = f"{command}.html"

file_path = os.path.join("ExportedFiles", "htmlFiles", file_name)


# Ensure directory exists

os.makedirs(os.path.dirname(file_path), exist_ok=True)


# Timestamp for current run

timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')


# If date/time not entered, use current timestamp

entered_date = entered_date or datetime.now().strftime('%Y-%m-%d')
entered_time = entered_time or datetime.now().strftime('%H:%M:%S')


# Data row to insert

new_row = {
    "Timestamp": timestamp,
    "Command": command,
    "URL": url,
    "Result": result,
    "Entered Date": entered_date,
    "Entered Time": entered_time
}


# Check if the HTML file exists and append rows

if os.path.exists(file_path):
```

```
# Open the file and append rows
```

```
with open(file_path, "r+", encoding="utf-8") as file:
```

```
    content = file.read()
```

```
    # Look for the closing </table> tag and append new rows before it
```

```
    if "</table>" in content:
```

```
        new_row_html =
```

```
f"<tr><td>{new_row['Timestamp']}</td><td>{new_row['Command']}</td><td>{new_row['URL']}</td><td>{new_row['Result']}</td><td>{new_row['Entered Date']}</td><td>{new_row['Entered Time']}</td></tr>\n"
```

```
    content = content.replace("</table>", new_row_html + "</table>")
```

```
    file.seek(0) # Move pointer to the start
```

```
    file.write(content)
```

```
    file.truncate() # Truncate any remaining content
```

```
    file.flush() # Flush the buffer to ensure it's written
```

```
else:
```

```
    # If the file doesn't exist, create a new one with table headers
```

```
    with open(file_path, "w", encoding="utf-8") as file:
```

```
        html_content = "<html><head><title>Command Data</title></head><body>"
```

```
        html_content += f"<h1>Results for {command}</h1><table border='1'>"
```

```
            html_content +=
```

```
"<tr><th>Timestamp</th><th>Command</th><th>URL</th><th>Result</th><th>Entered Date</th><th>Entered Time</th></tr>"
```

```
            html_content +=
```

```
f"<tr><td>{new_row['Timestamp']}</td><td>{new_row['Command']}</td><td>{new_row['URL']}</td><td>{new_row['Result']}</td><td>{new_row['Entered Date']}</td><td>{new_row['Entered Time']}</td></tr>\n"
```

```
html_content += "</table></body></html>"
```

```
file.write(html_content)
```

```
file.flush() # Ensure content is written to disk
```

```
return f"HTML file saved and updated at {file_path}."
```


--- MyBot.py ---

import discord

from discord.ext import commands

from boundary.BrowserBoundary import BrowserBoundary

from boundary.AccountBoundary import AccountBoundary

from boundary.AvailabilityBoundary import AvailabilityBoundary

from boundary.PriceBoundary import PriceBoundary

from boundary.BotBoundary import BotBoundary

from DataObjects.global_vars import GlobalState

Bot initialization

intents = discord.Intents.default()

intents.message_content = True # Enable reading message content

class MyBot(commands.Bot):

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

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

async def on_message(self, message):

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

return

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

GlobalState.user_message = message.content

```

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

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

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

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

else:

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

await self.process_commands(message)

GlobalState.reset_user_message() # Reset the global user_message variable

#print("User_message reset to empty string")

async def setup_hook(self):

    await self.add_cog(BrowserBoundary()) # Add your boundary objects

    await self.add_cog(AccountBoundary())

    await self.add_cog(AvailabilityBoundary())

    await self.add_cog(PriceBoundary())

    await self.add_cog(BotBoundary())

async def on_ready(self):

    print(f"Logged in as {self.user}")

    channel = discord.utils.get(self.get_all_channels(), name="general") # Adjust the channel
name if needed

    if channel:

```

```
await channel.send("Hi, I'm online! Type '!project_help' to see what I can do.")
```

```
async def on_command_error(self, ctx, error):
```

```
    if isinstance(error, commands.CommandNotFound):
```

```
        print("Command not recognized:")
```

```
        print(error)
```

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

```
# Initialize the bot instance
```

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

```
def start_bot(token):
```

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

```
    bot.run(token)
```