

--- 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_data, *args):
```

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

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

```
        if command_data == "fetch_all_accounts":
```

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

```
        elif command_data == "fetch_account_by_website":
```

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

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

```
        elif command_data == "add_account":
```

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

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

```
        elif command_data == "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
```

```
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":  
    print(f"Monitoring availability at {url} every {frequency} second(s).")  
    url = args[0]  
    date_str = args[1] if len(args) > 1 else None  
    frequency = args[2] if len(args) > 2 else 15  
    return await self.start_monitoring_availability(url, date_str, frequency)
```

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

```
else:  
    return "Invalid command."
```

```
async def check_availability(self, url: str, date_str=None):  
    """Handle availability check and export results."""  
    # Call the entity to check availability  
    availability_info = await self.availability_entity.check_availability(url, date_str)  
  
    # Prepare the result  
    try:
```

```
        result = f"Checked availability: {availability_info}"

    except Exception as e:

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

    print(result)
```

```
# Create a DTO (Data Transfer Object) for export
```

```
data_dto = {

    "command": "check_availability",

    "url": url,

    "result": result,

    "entered_date": datetime.now().strftime('%Y-%m-%d'),

    "entered_time": datetime.now().strftime('%H:%M:%S')

}
```

```
# Export data to Excel/HTML via the entity
```

```
self.availability_entity.export_data(data_dto)

return result
```

```
async def start_monitoring_availability(self, url: str, date_str=None, frequency=15):
```

```
    """Start monitoring availability at a specified frequency."""
```

```
    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
```

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

```
    # Prepare and log the result
```

```
    result = f"Checked availability: {availability_info}"
```

```
    print(result)
```

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

```
    # Create a DTO (Data Transfer Object) for export
```

```
    data_dto = {
```

```
        "command": "start_monitoring_availability",
```

```
        "url": url,
```

```
        "result": result,
```

```
        "entered_date": datetime.now().strftime('%Y-%m-%d'),
```

```
        "entered_time": datetime.now().strftime('%H:%M:%S')
```

```
    }
```

```
    # Export data to Excel/HTML via the entity
```

```
    self.availability_entity.export_data(data_dto)
```

```
    # Wait for the specified frequency before checking again
```

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

```
print(self.results)
```

```
except Exception as e:
```

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

```
    print(error_message)
```

```
    self.results.append(error_message)
```

```
    return error_message
```

```
return self.results
```

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

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

```
    result = None
```

```
    try:
```

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

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

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

```
        else:
```

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

```
            self.is_monitoring = False
```

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

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

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

```
            print(result)
```

```
    except Exception as e:
```

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

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


return result

--- BrowserControl.py ---

from entity.BrowserEntity import BrowserEntity

class BrowserControl:

def __init__(self):

Initialize the entity object inside the control layer

self.browser_entity = BrowserEntity()

def receive_command(self, command_data):

Validate the command

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

if command_data == "launch_browser":

Call the entity to perform the actual operation

try:

result = self.browser_entity.launch_browser()

return result

except Exception as e:

return str(e) # Return the error message

elif command_data == "close_browser":

Call the entity to perform the close operation

try:

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

        return result

    except Exception as e:

        return str(e) # Return the error message
```

```
else:
```

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

```
--- HelpControl.py ---
```

```
class HelpControl:
```

```
    def receive_command(self, command_data):
```

```
        """Handles the command and returns the appropriate message."""
```

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

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

```
            help_message = (
```

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

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

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

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

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

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

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

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

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

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

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

```
"!start_monitoring_price 'url' 'frequency' - Start monitoring a product's price at a specific interval (frequency in minutes).\n"
```

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

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

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

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

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

```
)
```

```
    return help_message
```

```
else:
```

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

```
--- LoginControl.py ---
```

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

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

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

```
class LoginControl:
```

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

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

```
        self.account_control = AccountControl() # Manages account data
```

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

```
        """Handle login command and perform business logic."""
```

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

```
if command_data == "login" and site:
```

```
    try:
```

```
        # Fetch account credentials from the entity
```

```
        account_info = self.account_control.fetch_account_by_website(site)
```

```
        if not account_info:
```

```
            return f"No account found for {site}"
```

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

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

```
        # Get the URL from the CSS selectors
```

```
        url = Selectors.get_selectors_for_url(site).get('url')
```

```
        print(url)
```

```
        if not url:
```

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

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

```
    except Exception as e:
```

```
        result = str(e)
```

```
    return result
```

```
else:
```

```
    return "Invalid command or site."
```

--- NavigationControl.py ---

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

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

```
class NavigationControl:
```

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

```
        # Initialize the entity object inside the control layer
```

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

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

```
        # Validate the command
```

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

```
        if command_data == "navigate_to_website":
```

```
            if not url:
```

```
                selectors = Selectors.get_selectors_for_url("google")
```

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

```
                if not url:
```

```
                    return "No URL provided, and default URL for google could not be found."
```

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

```
            try:
```

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

```
operation
```

```
            except Exception as e:
```

```
                result = str(e)
```

```
            return result
```

```
        else:
```

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

--- PriceControl.py ---

```
import asyncio
```

```
from datetime import datetime
```

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

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

```
class PriceControl:
```

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

```
        self.price_entity = PriceEntity() # Initialize PriceEntity for fetching and export
```

```
        self.is_monitoring = False # Monitoring flag
```

```
        self.results = [] # Store monitoring results
```

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

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

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

```
        if command_data == "get_price":
```

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

```
            return await self.get_price(url)
```

```
        elif command_data == "start_monitoring_price":
```

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

```
            frequency = args[1] if len(args) > 1 else 20
```

```
            return await self.start_monitoring_price(url, frequency)
```

```
elif command_data == "stop_monitoring_price":
```

```
    return self.stop_monitoring_price()
```

```
else:
```

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

```
async def get_price(self, url: str):
```

```
    """Handle fetching the price from the entity."""
```

```
    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)
```

```
    data_dto = {
```

```
        "command": "monitor_price",
```

```
        "url": url,
```

```
        "result": result,
```

```
        "entered_date": datetime.now().strftime('%Y-%m-%d'),
```

```
        "entered_time": datetime.now().strftime('%H:%M:%S')
    }
```

```
        # Pass the DTO to PriceEntity to handle export
        self.price_entity.export_data(data_dto)
```

```
except Exception as e:
```

```
    result = f"Failed to fetch price: {str(e)}"
```

```
return result
```

```
async def start_monitoring_price(self, url: str = None, frequency=20):
```

```
    """Start monitoring the price at a given interval."""
```

```
    if self.is_monitoring:
```

```
        return "Already monitoring prices."
```

```
    self.is_monitoring = True
```

```
    previous_price = None
```

```
    try:
```

```
        while self.is_monitoring:
```

```
            # Fetch the current price
```

```
            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)

current_price = self.price_entity.get_price_from_page(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."

print(result)

# Add the result to the results list
self.results.append(result)

# Create a DTO (Data Transfer Object) for export
data_dto = {

```

```
        "command": "monitor_price",
        "url": url,
        "result": result,
        "entered_date": datetime.now().strftime('%Y-%m-%d'),
        "entered_time": datetime.now().strftime('%H:%M:%S')
    }
```

```
# Pass the DTO to PriceEntity to handle export
```

```
self.price_entity.export_data(data_dto)
```

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

```
print(self.results)
```

```
except Exception as e:
```

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

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

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

```
    result = None
```

```
    try:
```

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

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

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

```
        else:
```

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

```
            self.is_monitoring = False
```

```

        result = "Results for price monitoring:\n"

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

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

        print(result)

    except Exception as e:

        # Handle any error that occurs

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

    return result

```

--- StopControl.py ---

```
import discord
```

```
class StopControl:
```

```
    async def receive_command(self, command_data, ctx):
```

```
        """Handle the stop bot command."""
```

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

```
        if command_data == "stop_bot":
```

```
            # Get the bot from the context (ctx) dynamically
```

```
            bot = ctx.bot # This extracts the bot instance from the context
```

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

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

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

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

```
print(result)
```

```
return result
```

```
else:
```

```
result = "Invalid command."
```

```
return result
```

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

```
#empty init file
```