

```
--- main.py ---
```

```
import discord
```

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

```
from boundary.BotBoundary import BotBoundary
```

```
from boundary.HelpBoundary import HelpBoundary
```

```
from boundary.AccountBoundary import AccountBoundary
```

```
from boundary.BrowserBoundary import BrowserBoundary
```

```
from boundary.LoginBoundary import LoginBoundary
```

```
from boundary.CloseBrowserBoundary import CloseBrowserBoundary
```

```
from boundary.StopBoundary import StopBoundary
```

```
from boundary.NavigationBoundary import NavigationBoundary
```

```
from boundary.PriceBoundary import PriceBoundary
```

```
from boundary.MonitorPriceBoundary import MonitorPriceBoundary
```

```
from boundary.AvailabilityBoundary import AvailabilityBoundary
```

```
from boundary.MonitorAvailabilityBoundary import MonitorAvailabilityBoundary
```

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

```
# Set up the bot's intents
```

```
intents = discord.Intents.default()
```

```
intents.message_content = True # Enable reading message content
```

```
# Initialize the bot with the correct command prefix and intents
```

```
class MyBot(commands.Bot):
```

```
    async def setup_hook(self):
```

```
        await self.add_cog(BotBoundary(self))
```

```
        await self.add_cog(HelpBoundary(self))
```

```
        await self.add_cog(AccountBoundary(self))
```

```
await self.add_cog(BrowserBoundary(self))

await self.add_cog(StopBoundary(self))

await self.add_cog(LoginBoundary(self))

await self.add_cog(CloseBrowserBoundary(self))

await self.add_cog(NavigationBoundary(self))

await self.add_cog(PriceBoundary(self))

await self.add_cog(MonitorPriceBoundary(self))

await self.add_cog(AvailabilityBoundary(self))

await self.add_cog(MonitorAvailabilityBoundary(self))
```

Run the bot

```
if __name__ == "__main__":

    bot = MyBot(command_prefix="!", intents=intents)

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

    bot.run(Config.DISCORD_TOKEN)
```

--- AccountBoundary.py ---

```
from discord.ext import commands

from control.AccountControl import AccountControl
```

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

```
    def __init__(self, bot):

        self.bot = bot

        self.account_control = AccountControl()
```

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

async def fetch_accounts(self, ctx):

    """Fetch and display all accounts."""

    accounts = self.account_control.fetch_accounts()

    # Send each account or the no accounts message to Discord

    for account in accounts:

        await ctx.send(account)
```

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

async def add_account(self, ctx, username: str, password: str):

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

    result = self.account_control.add_account(username, password)

    if result:

        await ctx.send(f"Account for {username} added successfully.")

    else:

        await ctx.send(f"Failed to add account for {username}.")
```

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

async def delete_account(self, ctx, user_id: int):

    """Delete a user account from the database."""

    result = self.account_control.delete_account(user_id)
```

if result:

```
    await ctx.send(f"Account with ID {user_id} deleted successfully.")
```

else:

```
    await ctx.send(f"Failed to delete account with ID {user_id}.")
```

--- AvailabilityBoundary.py ---

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

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

```
class AvailabilityBoundary(commands.Cog): # Make it a Cog to register as a bot command
```

```
    def __init__(self, bot):
```

```
        self.bot = bot
```

```
        self.control = AvailabilityControl()
```

```
@commands.command(name="check_availability") # Register the command with this decorator
```

```
async def check_availability(self, ctx, url: str, date_str=None, time_slot=None):
```

```
    # Call the control and get the results
```

```
    command_name = ctx.command.name
```

```
    result, html_msg, excel_msg = await self.control.handle_availability_check(ctx, url, date_str,
time_slot, command_name)
```

```
    # Send the result first
```

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

```
    # Send HTML and Excel results if available
```

```
    if html_msg:
```

```
    await ctx.send(html_msg)
```

```
if excel_msg:
```

```
    await ctx.send(excel_msg)
```

```
--- BotBoundary.py ---
```

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

```
from control.ChatControl import ChatControl
```

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

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

```
    def __init__(self, bot):
```

```
        self.bot = bot
```

```
        self.chat_control = ChatControl()
```

```
@commands.Cog.listener()
```

```
async def on_ready(self):
```

```
    """Bot startup message when ready."""
```

```
    print(f'Logged in as {self.bot.user.name}')
```

```
    channel = self.bot.get_channel(Config.CHANNEL_ID)
```

```
    if channel:
```

```
        await channel.send("Hi, I'm online!")
```

```
@commands.Cog.listener()
```

```
async def on_message(self, message):
```

```
    """Handle non-prefixed messages and command-prefixed messages."""
```

```
    if message.author == self.bot.user:
```

```
return
```

```
# Handle non-prefixed messages (like greetings)
```

```
if not message.content.startswith('!'):
```

```
    response = self.chat_control.process_non_prefixed_message(message.content)
```

```
    await message.channel.send(response)
```

```
@commands.Cog.listener()
```

```
async def on_command_error(self, ctx, error):
```

```
    """Handle unrecognized commands."""
```

```
    if isinstance(error, commands.CommandNotFound):
```

```
        # Handle unknown command
```

```
        response = self.chat_control.handle_unrecognized_command()
```

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

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

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

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

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

```
    def __init__(self, bot):
```

```
        self.bot = bot
```

```
        self.browser_control = BrowserControl()
```

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

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

```
"""Command to launch the browser."""
```

```
incognito = "incognito" in args
```

```
response = self.browser_control.launch_browser(ctx.author, incognito)
```

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

```
--- CloseBrowserBoundary.py ---
```

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

```
from control.CloseBrowserControl import CloseBrowserControl
```

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

```
    def __init__(self, bot):
```

```
        self.bot = bot
```

```
        self.close_browser_control = CloseBrowserControl()
```

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

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

```
        """Command to close the browser."""
```

```
        response = self.close_browser_control.close_browser()
```

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

```
--- HelpBoundary.py ---
```

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

```
from control.HelpControl import HelpControl
```

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

```

def __init__(self, bot):

    self.bot = bot

    self.help_control = HelpControl()

@commands.command(name='project_help')

async def project_help(self, ctx):

    """Handles the project_help command."""

    help_message = self.help_control.get_help_message()

    await ctx.send(help_message)

```

--- LoginBoundary.py ---

```

from discord.ext import commands

from control.LoginControl import LoginControl

class LoginBoundary(commands.Cog):

    def __init__(self, bot):

        self.bot = bot

        self.login_control = LoginControl()

@commands.command(name='login')

async def login(self, ctx, site: str, *args):

    """Command to log into a website using stored credentials."""

    incognito = "incognito" in args

    retries = next((int(arg) for arg in args if arg.isdigit()), 1)

    response = await self.login_control.login(site, incognito, retries)

    await ctx.send(response)

```



```
--- MonitorAvailabilityBoundary.py ---
```

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

```
from control.MonitorAvailabilityControl import MonitorAvailabilityControl
```

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

```
    def __init__(self, bot):
```

```
        self.bot = bot
```

```
        self.control = MonitorAvailabilityControl()
```

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

```
    async def start_monitoring(self, ctx, url: str, date_str=None, time_slot=None, frequency: int = 15):
```

```
        """Command to start monitoring availability at a given frequency (default: 15 seconds)."""
```

```
        # Check if frequency is given as the second argument (but no date or time provided)
```

```
        if date_str and date_str.isdigit() and not time_slot:
```

```
            frequency = int(date_str)
```

```
            date_str = None # Reset date_str because it was actually the frequency
```

```
        await ctx.send(f"Starting availability monitoring every {frequency} second(s).")
```

```
        await self.control.start_monitoring(ctx, url, date_str, time_slot, frequency)
```

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

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

```
        """Command to stop monitoring availability."""
```

```
self.control.stop_monitoring()

await ctx.send("Stopped monitoring availability.")
```

--- MonitorPriceBoundary.py ---

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

```
from control.MonitorPriceControl import MonitorPriceControl
```

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

```
    def __init__(self, bot):
```

```
        self.bot = bot
```

```
        self.monitor_price_control = MonitorPriceControl()
```

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

```
    async def monitor_price_command(self, ctx, url: str, frequency: int = 1):
```

```
        """Command to start monitoring the price of a product."""
```

```
        await self.monitor_price_control.start_monitoring(ctx, url, frequency)
```

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

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

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

```
        await self.monitor_price_control.stop_monitoring(ctx)
```

--- NavigationBoundary.py ---

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

```
from control.NavigationControl import NavigationControl
```

```

class NavigationBoundary(commands.Cog):

    def __init__(self, bot):

        self.bot = bot

        self.navigation_control = NavigationControl()

    @commands.command(name='navigate_to_website')

    async def navigate_to_website(self, ctx, url: str):

        """Command to navigate to a specified URL."""

        response = self.navigation_control.navigate_to_url(url)

        await ctx.send(response)

```

--- PriceBoundary.py ---

```

from discord.ext import commands

from control.PriceControl import PriceControl

class PriceBoundary(commands.Cog):

    def __init__(self, bot):

        self.bot = bot

        self.price_control = PriceControl()

    @commands.command(name='get_price')

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

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

        response = await self.price_control.get_price(ctx, url)

        await ctx.send(response)

```

--- StopBoundary.py ---

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

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

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

```
    def __init__(self, bot):
```

```
        self.bot = bot
```

```
        self.bot_control = BotControl(bot)
```

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

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

```
        """Handles the stop command and gracefully shuts down the bot."""
```

```
        await ctx.send("Stopping the bot...")
```

```
        await self.bot_control.stop_bot()
```

--- __init__.py ---

```
#empty init file
```

--- AccountControl.py ---

```
from entity.AccountEntity import AccountEntity
```

```
class AccountControl:
```

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

```
        self.account_entity = AccountEntity()
```

```

def add_account(self, username, password, webSite):

    self.account_entity.connect()

    self.account_entity.add_account(username, password, webSite)

    self.account_entity.close()


def fetch_accounts(self):

    """Fetch all accounts and return them."""

    self.account_entity.connect()

    accounts = self.account_entity.fetch_accounts()


    if accounts:

        account_messages = []

        for account in accounts:

            message = f"ID: {account[0]}, Username: {account[1]}, Password: {account[2]}, Website:
{account[3]}"

            print(message) # For terminal output

            account_messages.append(message)

        self.account_entity.close()

        return account_messages

    else:

        print("No accounts found.") # For terminal output

        self.account_entity.close()

        return ["No accounts found."]


def fetch_account_by_website(self, website):

```

```
"""Fetch the username and password where the website matches."""
```

```
self.account_entity.connect()
```

```
account = self.account_entity.fetch_account_by_website(website) # Call the entity method
```

```
self.account_entity.close()
```

```
return account
```

```
def delete_account(self, account_id):
```

```
    self.account_entity.connect()
```

```
    self.account_entity.delete_account(account_id)
```

```
    self.account_entity.reset_id_sequence()
```

```
    self.account_entity.close()
```

```
--- AvailabilityControl.py ---
```

```
from utils.export_utils import ExportUtils
```

```
from entity.AvailabilityEntity import AvailabilityEntity # Fixing this import
```

```
from datetime import datetime # Importing datetime module
```

```
class AvailabilityControl:
```

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

```
        self.availability_entity = AvailabilityEntity() # Initialize the entity
```

```
        async def handle_availability_check(self, ctx, url, date_str=None, time_slot=None,
command_name="NameNotProvided"):
```

```
            # Perform availability check by calling the entity
```

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

```

# Get the current timestamp

current_timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')


# If date and time were entered, populate them; otherwise, use current timestamp
entered_date = date_str if date_str else datetime.now().strftime('%Y-%m-%d')
entered_time = time_slot if time_slot else datetime.now().strftime('%H:%M:%S')


# Log results to HTML and Excel

data = [{'Timestamp': current_timestamp, 'Command': command_name, 'URL': url,
        'Result': availability_info, 'Entered Date': entered_date, 'Entered Time': entered_time}]


html_msg = ""
excel_msg = ""


try:
    html_msg = ExportUtils.export_to_html(data, command_name)
except Exception as e:
    html_msg = f"Failed to export to HTML: {str(e)}"


try:
    excel_msg = ExportUtils.log_to_excel(command_name, url, availability_info)
except Exception as e:
    excel_msg = f"Failed to export to Excel: {str(e)}"


# Return availability info and export results

return availability_info, html_msg, excel_msg

```

--- BotControl.py ---

```
import asyncio
```

```
class BotControl:
```

```
    def __init__(self, bot):
```

```
        self.bot = bot
```

```
    async def send_greeting(self):
```

```
        """Sends a greeting when the bot comes online."""
```

```
        channel = self.bot.get_channel(self.bot.config.CHANNEL_ID)
```

```
        if channel:
```

```
            await channel.send("Hi, I'm online! type '!project_help' to see what I can do")
```

```
    async def stop_bot(self):
```

```
        """Stops the bot gracefully, ensuring all connections are closed."""
```

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

```
        await self.bot.close()
```

--- BrowserControl.py ---

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

```
from control.AccountControl import AccountControl # Use AccountControl for consistency
```

```
class BrowserControl:
```

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



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

```
self.account_control = AccountControl() # Use AccountControl to fetch credentials
```

```
def launch_browser(self, user, incognito=False):
```

```
    return self.browser_entity.launch_browser(incognito=incognito, user=user)
```

--- ChatControl.py ---

ChatControl in control/ChatControl.py

```
class ChatControl:
```

```
    def process_non_prefixed_message(self, message):
```

```
        """Process non-prefixed messages like 'hi', 'hello'."""
```

```
        if message.lower() in ["hi", "hello"]:
```

```
            return "Hello! How can I assist you today? Type !project_help for assistance."
```

```
        else:
```

```
            return "I didn't recognize that. Type !project_help to see available commands."
```

```
    def handle_unrecognized_command(self):
```

```
        """Handle unrecognized command from on_command_error."""
```

```
        return "I didn't recognize that command. Type !project_help for assistance."
```

--- CloseBrowserControl.py ---

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

```
class CloseBrowserControl:
```

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

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

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

```
    return self.browser_entity.close_browser()
```

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

```
class HelpControl:
```

```
    def get_help_message(self):
```

```
        """Returns a list of available bot commands."""
```

```
        return (
```

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

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

```
            "!login 'website' - Log in to a website.\n"
```

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

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

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

```
            "!monitor_price - Track a product price.\n"
```

```
            "!get_price - Check the price of a product.\n"
```

```
            "!check_availability - Check the availability of a product.\n"
```

```
            "!stop_monitoring - Stop tracking a product.\n"
```

```
            "##!receive_notifications - Receive notifications for price changes.\n"
```

```
            "##!extract_data - Export data to Excel or HTML.\n"
```

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

```
)
```

--- LoginControl.py ---

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

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

```
class LoginControl:
```

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

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

```
        self.account_control = AccountControl()
```

```
    async def login(self, site, incognito=False, retries=1):
```

```
        # Fetch credentials using AccountControl
```

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

```
        if account:
```

```
            username, password = account
```

```
            return await self.browser_entity.login(site, username, password, incognito, retries)
```

```
        else:
```

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

--- MonitorAvailabilityControl.py ---

```
import asyncio
```

```
from control.AvailabilityControl import AvailabilityControl # Reuse existing control
```

```
import logging
```

```
class MonitorAvailabilityControl:
```

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

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

```

self.monitoring_task = None # To store the running task

self.logger = logging.getLogger("MonitorAvailabilityControl")

async def start_monitoring(self, ctx, url, date_str=None, time_slot=None, frequency=15):
    """Start monitoring availability at the given frequency (in seconds)."""

    # If a task is already running, notify the user

    command_name = "monitor_availability"

    if self.monitoring_task:

        await ctx.send("Monitoring is already running.")

        return

    # Define the monitoring loop

    async def monitor():

        while True:

            try:

                # Reuse the existing check_availability method from AvailabilityControl

                result, html_msg, excel_msg = await

self.availability_control.handle_availability_check(ctx, url, date_str, time_slot, command_name)

                # Send availability result to the user

                await ctx.send(result)

                # Send HTML and Excel results if available

                if html_msg:

                    await ctx.send(html_msg)

                if excel_msg:

                    await ctx.send(excel_msg)

```

```
except Exception as e:
```

```
    self.logger.error(f"Failed to check availability for {url}: {e}")
```

```
    await ctx.send(f"Error: {str(e)}")
```

```
    await asyncio.sleep(frequency) # Wait for the next interval (in seconds)
```

```
# Start the task in the background
```

```
self.monitoring_task = asyncio.create_task(monitor())
```

```
def stop_monitoring(self):
```

```
    """Stop the ongoing monitoring task."""
```

```
    if self.monitoring_task:
```

```
        self.monitoring_task.cancel() # Stop the task
```

```
        self.monitoring_task = None
```

```
--- MonitorPriceControl.py ---
```

```
import asyncio
```

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

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

```
import logging
```

```
class MonitorPriceControl:
```

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

```
        self.price_entity = PriceEntity()
```

```
        self.is_monitoring = False # Control flag for monitoring state
```

```
self.logger = logging.getLogger("MonitorPriceControl")
```

```
async def start_monitoring(self, ctx, url, frequency=20):
```

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

```
    if ctx.channel.id == Config.CHANNEL_ID:
```

```
        if self.is_monitoring:
```

```
            await ctx.send("Already monitoring prices.")
```

```
            return
```

```
        self.is_monitoring = True
```

```
        await ctx.send(f"Monitoring price every {frequency} second(s).")
```

```
        previous_price = None
```

```
    try:
```

```
        while self.is_monitoring:
```

```
            current_price = self.price_entity.get_price(url)
```

```
            if current_price:
```

```
                if previous_price is None:
```

```
                    await ctx.send(f"Starting price monitoring. Current price is: {current_price}")
```

```
                elif current_price > previous_price:
```

```
                    await ctx.send(f"Price went up! Current price: {current_price} (Previous: {previous_price})")
```

```
                elif current_price < previous_price:
```

```
                    await ctx.send(f"Price went down! Current price: {current_price} (Previous: {previous_price})")
```

```
            else:
```

```
                await ctx.send(f"Price remains the same: {current_price}")
```

```

        previous_price = current_price

    else:

        await ctx.send("Failed to retrieve the price.")

        await asyncio.sleep(frequency) # Wait for the next check

    except Exception as e:

        self.logger.error(f"Failed to monitor price for {url}: {e}")

        await ctx.send(f"Failed to monitor price: {e}")

    else:

        await ctx.send("This command can only be used in the designated channel.")


async def stop_monitoring(self, ctx):

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

    if self.is_monitoring:

        self.is_monitoring = False

        await ctx.send("Price monitoring has been stopped.")

    else:

        await ctx.send("No monitoring process is currently running.")

```

--- NavigationControl.py ---

```

from entity.BrowserEntity import BrowserEntity

```

```

class NavigationControl:

```

```

    def __init__(self):

        self.browser_entity = BrowserEntity()

```

```

    def navigate_to_url(self, url):

```

```
"""Navigate to a specific URL."""
```

```
return self.browser_entity.navigate_to_url(url)
```

```
--- PriceControl.py ---
```

```
import asyncio
```

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

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

```
import logging
```

```
class PriceControl:
```

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

```
        self.price_entity = PriceEntity()
```

```
        self.logger = logging.getLogger("PriceControl")
```

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

```
        """Fetch the current price from the given URL."""
```

```
        if ctx.channel.id == Config.CHANNEL_ID:
```

```
            try:
```

```
                price = self.price_entity.get_price(url)
```

```
                if price:
```

```
                    return f"The current price is: {price}"
```

```
            else:
```

```
                return "Failed to retrieve the price."
```

```
        except Exception as e:
```

```
            self.logger.error(f"Failed to get price for {url}: {e}")
```

```
            return f"Error getting price: {e}"
```


else:

return "This command can only be used in the designated channel."

--- __init__.py ---

#empty init file

--- AccountEntity.py ---

import psycopg2

from utils.Config import Config

class AccountEntity:

def __init__(self):

self.dbname = "postgres"

self.user = "postgres"

self.host = "localhost"

self.port = "5432"

self.password = Config.DATABASE_PASSWORD

def connect(self):

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, password, webSite):

    """Insert a new account into the accounts table."""

    try:

        if self.cursor:

            self.cursor.execute("INSERT INTO accounts (username, password, website) VALUES
(%s, %s, %s)", (username, password, webSite))

            self.connection.commit()

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

        except Exception as error:

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


def fetch_accounts(self):

    """Fetch all accounts from the accounts table."""

    try:

        if self.cursor:

            self.cursor.execute("SELECT * FROM accounts;")

```

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

```
return accounts
```

```
except Exception as error:
```

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

```
    return None
```

```
def delete_account(self, account_id):
```

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

```
    try:
```

```
        if self.cursor:
```

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

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

```
            if account:
```

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

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

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

```
            else:
```

```
                print(f"Account with ID {account_id} not found. No deletion performed.")
```

```
except Exception as error:
```

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

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

```
    """Fetch the username and password where the website matches."""
```

```
    try:
```

```
        self.cursor.execute("SELECT username, password FROM accounts WHERE
```

```
LOWER(website) = LOWER(%s)", (website,))
```

```
    return self.cursor.fetchone() # Returns one matching account
```

```
except Exception as error:
```

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

```
    return None
```

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

```
    """Reset the account ID sequence to the next available value."""
```

```
    try:
```

```
        if self.cursor:
```

```
            self.cursor.execute("SELECT COALESCE(MAX(id), 0) + 1 FROM accounts")
```

```
            next_id = self.cursor.fetchone()[0]
```

```
                self.cursor.execute("ALTER SEQUENCE accounts_id_seq RESTART WITH %s",
```

```
(next_id,))
```

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

```
            print(f"ID sequence reset to {next_id}.")
```

```
except Exception as error:
```

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

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

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

```
    if self.cursor:
```

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

```
    if self.connection:
```

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

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

```
--- AvailabilityEntity.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 utils.css_selectors import Selectors
```

```
from entity.BrowserEntity import BrowserEntity # Import BrowserEntity
```

```
class AvailabilityEntity:
```

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

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

```
    async def check_availability(self, ctx, url, date_str=None, time_slot=None, timeout=5):
```

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

```
        self.browser_entity.navigate_to_url(url)
```

```
        # Wait for page to load (you can tweak the sleep time based on your page loading behavior)
```

```
        await asyncio.sleep(3)
```

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

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

```
        if not selectors:
```

```
            return "No valid selectors found for this URL."
```

```

# Ensure select_time and no_availability exist in selectors

if 'select_time' not in selectors or 'no_availability' not in selectors:

    return "Missing required selectors for availability check."


# Perform date and time selection (optional)

if date_str:

    try:

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

        date_field.click()

        await asyncio.sleep(1)

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

        date_button.click()

    except Exception as e:

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


if time_slot:

    try:

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

        time_field.clear()

        time_field.send_keys(time_slot)

    except Exception as e:

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


await asyncio.sleep(2) # Wait for updates (adjust this time based on page response)

```

```
# 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(
```

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

```
    )
```

```
    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."
```

```
--- BrowserEntity.py ---
```

```
import asyncio
```

```
from selenium import webdriver
```

```
from selenium.webdriver.chrome.service import Service
```

```
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.css_selectors import Selectors # Import CSS selectors for the website
```

```
class BrowserEntity:
```

```
    _instance = None # Singleton instance
```

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

```
        if cls._instance is None:
```

```
            cls._instance = super(BrowserEntity, cls).__new__(cls)
```

```
            cls._instance.driver = None # Initialize driver to None
```

```
        return cls._instance
```

```
    def launch_browser(self, incognito=False, user=None):
```

```
        if self.driver:
```



```
print("Browser is already running. No need to launch a new one.")
```

```
return "Browser is already running."
```

```
try:
```

```
# Special launch options as per your original implementation
```

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

```
# Add options to avoid crashing and improve performance
```

```
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")
```

```
if incognito:
```

```
    options.add_argument("--incognito")
```

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

```
success_message = "Chrome browser launched successfully in incognito mode." if incognito
```

```
else "Chrome browser launched successfully."
```

```
print(f"Driver initialized: {self.driver}") # Debug: Print the driver
```

```
    return success_message
```

```
except Exception as e:
```

```
    error_message = f"Failed to launch browser: {e}"
```

```
    print(error_message)
```

```
    raise
```

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

```
    if not self.driver:
```

```
        print("Driver is not initialized, launching browser first.") # Debug
```

```
        self.launch_browser()
```

```
    try:
```

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

```
        print(f"Navigated to URL: {url}")
```

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

```
    except Exception as e:
```

```
        raise
```

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

```
    print(f"Closing browser. Current driver: {self.driver}") # Debug: Check the driver status
```

```
    if self.driver:
```

```
        self.driver.quit() # Close the browser session
```

```
        self.driver = None # Set to None after closing
```

```
        print("Browser closed successfully.")
```

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

```
    else:
```

```
print("No browser is currently open.")  
  
return "No browser is currently open."
```

```
async def login(self, site, username, password, incognito=False, retries=1):  
  
    # Get the URL and selectors from css_selectors  
  
    url = Selectors.get_selectors_for_url(site)['url']  
  
    for attempt in range(retries):  
  
        try:  
  
            self.navigate_to_url(url)  
  
            await asyncio.sleep(3)  
  
            # Enter the email address  
  
            email_field = self.driver.find_element(By.CSS_SELECTOR,  
Selectors.get_selectors_for_url(site)['email_field'])  
  
            email_field.click()  
  
            email_field.send_keys(username)  
  
            await asyncio.sleep(3)  
  
            # Enter the password  
  
            password_field = self.driver.find_element(By.CSS_SELECTOR,  
Selectors.get_selectors_for_url(site)['password_field'])  
  
            password_field.click()  
  
            password_field.send_keys(password)  
  
            await asyncio.sleep(3)  
  
            # Click the login button
```

```

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

        sign_in_button.click()

        await asyncio.sleep(5)

        # Wait for the homepage to load after login

        WebDriverWait(self.driver, 30).until(

            EC.presence_of_element_located((By.CSS_SELECTOR,
Selectors.get_selectors_for_url(site)['homePage'])))

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

    except Exception as e:

        if attempt < retries - 1:

            await asyncio.sleep(3)

        else:

            raise e

```

--- NotificationEntity.py ---

--- PriceEntity.py ---

```

import time

from selenium.webdriver.common.by import By

from utils.css_selectors import Selectors

from entity.BrowserEntity import BrowserEntity # Import the browser interaction logic

```

```

class PriceEntity:

    def __init__(self):

        self.browser_entity = BrowserEntity()


    def get_price(self, url):

        """Fetch the price from the provided URL using CSS selectors."""

        selectors = Selectors.get_selectors_for_url(url)

        if not selectors:

            raise ValueError(f"No selectors found for URL: {url}")


        # Navigate to the URL using the browser entity

        self.browser_entity.navigate_to_url(url)

        time.sleep(2) # Wait for the page to load


        try:

            # Use the CSS selector to find the price on the page

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

            price = price_element.text

            print(f"Price found: {price}")

            return price

        except Exception as e:

            print(f"Error finding price: {e}")

            return None

```

--- PriceHistoryEntity.py ---

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

```
#empty init file
```

```
--- project_structure.py ---
```

```
import os
```

```
def list_files_and_folders(directory, output_file):
```

```
    with open(output_file, 'w') as f:
```

```
        for root, dirs, files in os.walk(directory):
```

```
            # Ignore .git and __pycache__ folders
```

```
            dirs[:] = [d for d in dirs if d not in ['.git', '__pycache__']]
```

```
            f.write(f"Directory: {root}\n")
```

```
            for dir_name in dirs:
```

```
                f.write(f"  Folder: {dir_name}\n")
```

```
            for file_name in files:
```

```
                f.write(f"    File: {file_name}\n")
```

```
# Update the directory path to your project folder
```

```
project_directory = "D:/HARRISBURG/Harrisburg Master's Fifth Term Late Summer/CISC  
699/DiscordBotProject_CISC699"
```

```
output_file = os.path.join(project_directory, "project_structure.txt")
```

```
# Call the function to list files and save output to .txt
```

```
list_files_and_folders(project_directory, output_file)
```

```
print(f"File structure saved to {output_file}")
```

```
--- project_text.py ---
```

```
import os
```

```
from fpdf import FPDF
```

```
# Directory where the project files are located
```

```
directory = r"D:\HARRISBURG\Harrisburg Master's Fifth Term Late Summer\CISC
699\DiscordBotProject_CISC699"
```

```
output_pdf_path = os.path.join(directory, "project_text.pdf")
```

```
# Function to retrieve all text from files, ignoring .git and __pycache__ directories
```

```
def extract_project_text(directory):
```

```
project_text = ""
```

for root, dirs, files in os.walk(directory):

```
# Ignore .git and __pycache__ directories
```

```
dirs[:] = [d for d in dirs if d not in ['.git', '__pycache__']]
```

for file in files:

```
if file.endswith('.py') or file.endswith('.txt') or file.endswith('.md'): # Only considering relevant
```

file types

```
file_path = os.path.join(root, file)
```

try:

```
with open(file_path, 'r', encoding='utf-8') as f:
```

```
project_text += f"--- {file} ---\n"
```

```
        project_text += f.read() + "\n\n"

    except Exception as e:

        print(f"Could not read file {file_path}: {e}")
```

```
    return project_text
```

```
# Function to generate a PDF with the extracted text
```

```
def create_pdf(text, output_path):
```

```
    pdf = FPDF()
```

```
    pdf.set_auto_page_break(auto=True, margin=15)
```

```
    pdf.add_page()
```

```
    pdf.set_font("Arial", size=12)
```

```
# Ensure proper encoding handling
```

```
for line in text.split("\n"):
```

```
    # Convert the text to UTF-8 and handle unsupported characters
```

```
    try:
```

```
        pdf.multi_cell(0, 10, line.encode('latin1', 'replace').decode('latin1'))
```

```
    except UnicodeEncodeError:
```

```
        # Handle any other encoding issues
```

```
        pdf.multi_cell(0, 10, line.encode('ascii', 'replace').decode('ascii'))
```

```
pdf.output(output_path)
```

```
# Extract project text and create the PDF
```



```
project_text = extract_project_text(directory)
```

```
if project_text:
```

```
    create_pdf(project_text, output_pdf_path)
```

```
    output_pdf_path
```

```
    print("pdf file created with all project's as text: " + output_pdf_path)
```

```
else:
```

```
    "No project text found."
```

```
--- Tests_URLs.txt ---
```

```
database password: postgres
```

Working Commands: Test commands

```
!project_help
```

```
!login bestbuy
```

```
!launch_browser
```

```
!close_browser
```

```
!navigate_to_website https://www.google.com/
```

```
!get_price
```

<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?skuld=6584960>

!monitor_price

<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?skuld=6584960>

!stop_monitoring

!check_availability <https://www.opentable.com/r/bar-spero-washington/>

!monitor_availability <https://www.opentable.com/r/bar-spero-washington/>

!stop_monitoring_availability

!check_availability <https://www.opentable.com/r/bar-spero-washington/> "August 22"

!stop_bot

Working on it:

!check_availability <https://www.opentable.com/r/bar-spero-washington/> "August 22" "8:00 PM"

URLs to Test:

<https://www.opentable.com/r/bar-spero-washington/>

https://www.ebay.com/itm/314411766963?_trkparms=amclsrc%3DITM%26aid%3D777008%26algo%3DPERSONAL.TOPIC%26ao%3D1%26asc%3D20240603121456%26meid%3Da07931f944bc4a5b95376fe64d0ab035%26pid%3D102177%26rk%3D1%26rkt%3D1%26itm%3D314411766963%26pmt%3D1%26noa%3D1%26pg%3D4375194%26algv%3DNoSignalMostWatched%26brand%3DSimpliSafe&_trksid=p4375194.c102177.m166540&_trkparms=parentrq%3A71497a9c1910a8cd54f819a0ffff582e%7Cpageci%3A59d1354a-5f2b-11ef-9c4d-f2c982e61003%7Ciid%3A1%7Cvlpname%3Avlp_homepage

<https://www.trendyol.com/puma/rebound-v6-low-p-736020132?boutiqueId=61&merchantId=184734&sav=true>

!get_price

<https://www.trendyol.com/puma/rebound-v6-low-p-736020132?boutiqueId=61&merchantId=184734&sav=true>

--- test_addAccount.py ---

```
import sys, os
```

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

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

```
def test_add_account():
```

```
    account_control = AccountControl()
```

```
# Adding a new account

account_control.add_account("newUser", "newPassword123", "newWebsite")


if __name__ == "__main__":

    test_add_account()
```

--- test_deleteAccount.py ---

```
import sys, os

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

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

```
def test_delete_account():

    account_control = AccountControl()

    account_control.delete_account(4)
```

```
if __name__ == "__main__":

    test_delete_account()
```

--- test_excel_creation.py ---

--- test_fetchAccounts.py ---

```
import sys, os

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

from control.AccountControl import AccountControl

def test_fetch_accounts():

    account_control = AccountControl()

    # Fetching all accounts

    account_control.fetch_accounts()

def test_fetch_account_by_website(website):

    account_control = AccountControl()

    # Fetch the account by website directly

    account = account_control.fetch_account_by_website(website)

    if account:

        username, password = account # Unpack the returned tuple

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

    else:

        print(f"No account found for website: {website}")

if __name__ == "__main__":

    test_fetch_accounts()

    test_fetch_account_by_website("ebay")
```

--- test_html_creation.py ---

--- __init__.py ---

#empty init file

--- Config.py ---

class Config:

DISCORD_TOKEN =

'MTI2OTM4MTE4OTA1NjMzNTk3Mw.Gihcfw.nrq0x-JiL65P0LIQTO-rTyyXq0qC-2PSSBuXr8'

CHANNEL_ID = 1269383349278081054

DATABASE_PASSWORD = 'postgres'

--- css_selectors.py ---

class Selectors:

SELECTORS = {

"trendyol": {

"price": ".featured-prices .prc-dsc" # Selector for Trendyol price

},

"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": {
```

```
    "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/",
```

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

```
    "time_field": "#restProfileSideBarTimePickerDtpPicker",
```

```
    "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
```

--- DiscordUtils.py ---

--- export_utils.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 ({file_path})."
```

```
@staticmethod
```

```
def export_to_html(data, command_name):
```

```
    # Define file path for HTML
```

```
    file_name = f"{command_name}.html" # Only one HTML file per command, will be appended
```

```
    file_path = os.path.join("ExportedFiles", "htmlFiles", file_name)
```

```
    # Ensure the directory exists
```

```
    os.makedirs(os.path.dirname(file_path), exist_ok=True)
```

```
    # Check if the file already 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_rows = ""
```

```
                for row in data:
```

```
                    # Ensure all necessary keys are in the row dictionary
```

```
                    new_rows += f"<tr><td>{row.get('Timestamp', 'N/A')}</td><td>{row.get('Command',  
'N/A')}</td><td>{row.get('URL', 'N/A')}</td><td>{row.get('Result', 'N/A')}</td><td>{row.get('Entered  
Date', 'N/A')}</td><td>{row.get('Entered Time', 'N/A')}</td></tr>\n"
```

```
# Insert new rows before </table>
```

```
content = content.replace("</table>", new_rows + "</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_name}</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>"
```

```
    for row in data:
```

```
        # Ensure all necessary keys are in the row dictionary
```

```
        html_content += f"<tr><td>{row.get('Timestamp', 'N/A')}</td><td>{row.get('Command',  
'N/A')}</td><td>{row.get('URL', 'N/A')}</td><td>{row.get('Result', 'N/A')}</td><td>{row.get('Entered  
Date', 'N/A')}</td><td>{row.get('Entered Time', 'N/A')}</td></tr>\n"
```

```
    html_content += "</table></body></html>"
```

```
    file.write(html_content)
```

```
    file.flush() # Ensure content is written to disk
```

```
    print(f"Created new HTML file at {file_path}.")
```

```
    return f"HTML file saved and updated at {file_path}."
```