

--- 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."""
```

```
    if self.cursor:
```

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

```
    if self.connection:
```

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

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

```
--- 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)
```

