```
# 2024 Â© Idan Hazay
# Import required libraries

import os, sqlite3, traceback  # SQLite3 for database handling, traceback for error logging
from datetime import datetime, timedelta  # Used for handling date operations

class DataBase:
    """
    Handles all database operations, including user authentication, file management,
    directory management, and permission control.
    """

    def __init__(self):
        self.database = f"
{os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))}\\server\\database\\database.db"  # Path to
the SQLite database
        self.users_table = "Users"  # Table for storing user accounts
        self.files_table = "Files"  # Table for storing file records
        self.permissions_table = "Permissions"  # Table for access control settings
        self.directories_table = "Directories"  # Table for directory structure
        self.deleted_table = "Deleted"  # Table for tracking deleted files (soft delete)

    def create_tables(self):
        """
        Creates necessary database tables if they do not exist.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        #cursor.execute(f"DROP TABLE {users_table}")
        #cursor.execute(f"DROP TABLE {files_table}")
        #cursor.execute(f"DROP TABLE {directories_table}")
        #cursor.execute(f"DROP TABLE {permissions_table}")
        cursor.execute(f"DROP TABLE {self.deleted_table}")
        #cursor.execute(f"CREATE TABLE IF NOT EXISTS {users_table} (id TEXT PRIMARY KEY, email TEXT UNIQUE, username TEXT
UNIQUE, password TEXT, salt TEXT, last_code INTEGER, valid_until TEXT, verified BOOL, subscription_level INT, admin_level
INT, cookie TEXT UNIQUE, cookie_expiration TEXT)")
        #cursor.execute(f"CREATE TABLE IF NOT EXISTS {files_table} (id TEXT PRIMARY KEY, sname TEXT UNIQUE, fname TEXT,
parent TEXT, owner_id TEXT, size TEXT, last_edit TEXT)")
        #cursor.execute(f"CREATE TABLE IF NOT EXISTS {directories_table} (id TEXT PRIMARY KEY, name TEXT, parent TEXT,
owner_id TEXT)")
        #cursor.execute(f"CREATE TABLE IF NOT EXISTS {permissions_table} (id TEXT PRIMARY KEY,  file_id TEXT, owner_id
TEXT, user_id TEXT, read BOOL, write BOOL, del BOOL, rename BOOL, download BOOL, share BOOL)")
        cursor.execute(f"CREATE TABLE IF NOT EXISTS {self.deleted_table} (id TEXT PRIMARY KEY, owner_id TEXT,
time_to_delete TEXT)")
        conn.commit()
        conn.close()

    def get_user_id(self, cred):
        """
        Retrieves user ID based on username or email.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT id FROM {self.users_table} WHERE username = ? OR email = ?", (cred, cred))
        row = cursor.fetchone()
        conn.close()
        if row is None:
            return None  # User not found
        return row[0]  # Return user ID

    def add_user(self, user_dict):
        """
        Adds a new user to the database.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        columns = ', '.join(user_dict.keys())  # Extract column names
        values = ', '.join(['?'] * len(user_dict))  # Create placeholders for values
        sql = f"INSERT INTO {self.users_table} ({columns}) VALUES ({values})"

        try:
            cursor.execute(sql, list(user_dict.values()))
            conn.commit()
        except sqlite3.IntegrityError:
            print(traceback.format_exc())  # Log database integrity error
            print("Key values already exist in table")
        conn.close()

    def remove_user(self, id):
        """
        Removes a user and associated records from the database.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"DELETE FROM {self.users_table} WHERE id = ?", (id,))
        cursor.execute(f"DELETE FROM {self.files_table} WHERE owner_id = ?", (id,))
        cursor.execute(f"DELETE FROM {self.directories_table} WHERE owner_id = ?", (id,))
```

```
        cursor.execute(f"DELETE FROM {self.permissions_table} WHERE owner_id = ?", (id,))
        cursor.execute(f"DELETE FROM {self.permissions_table} WHERE user_id = ?", (id,))
        conn.commit()
        conn.close()

    def update_user(self, id, fields, new_values):
        """
        Updates user details in the database.
        """
        if type(fields) != list:
            fields = [fields]  # Ensure fields are in list format
        if type(new_values) != list:
            new_values = [new_values]  # Ensure new values are in list format

        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        sql = f"UPDATE {self.users_table} SET "
        sql += ", ".join(f"{field} = ?" for field in fields)  # Generate update query dynamically
        sql += " WHERE id = ?"

        try:
            cursor.execute(sql, tuple(new_values + [id]))
            conn.commit()
        except sqlite3.IntegrityError:
            print("Key values already exist in table")
        conn.close()

    def get_user_values(self, id, fields):
        """
        Retrieves specific user fields from the database.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        sql = f"SELECT {', '.join(fields)} FROM {self.users_table} WHERE id = ?"
        cursor.execute(sql, (id,))
        row = cursor.fetchone()
        conn.close()
        return row

    def row_to_dict_user(self, row):
        """
        Converts a database row into a user dictionary.
        """
        return {
            "id": row[0], "email": row[1], "username": row[2], "password": row[3],
            "salt": row[4], "last_code": row[5], "valid_until": row[6],
            "verified": bool(row[7]), "subscription_level": int(row[8]),
            "admin_level": int(row[9]), "cookie": row[10], "cookie_expiration": row[11]
        }

    def row_to_dict_file(self, row):
        """
        Converts a database row into a file dictionary.
        """
        file_dict = {"id": row[0], "sname": row[1], "fname": row[2], "parent": row[3],
                     "owner_id": row[4], "size": row[5], "last_edit": row[6]}
        return file_dict

    def row_to_dict_directory(self, row):
        """
        Converts a database row into a directory dictionary.
        """
        directory_dict = {"id": row[0], "name": row[1], "parent": row[2], "owner_id": row[3]}
        return directory_dict

    def get_user(self, cred):
        """
        Retrieves user data from the database.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.users_table} WHERE username = ? OR email = ? OR id = ? OR cookie = ?", (cred,
cred, cred, cred))
        row = cursor.fetchone()
        conn.close()
        return self.row_to_dict_user(row) if row else None  # Convert row to dictionary if user exists


    def update_file(self, id, fields, new_values):
        """
        Updates file attributes in the database.
        Automatically updates the last edit timestamp.
        """
        if type(fields) != list:
            fields = [fields]  # Ensure fields are a list
        if type(new_values) != list:
            new_values = [new_values]  # Ensure values are a list
```

```
        fields.append("last_edit")  # Automatically update the last modified timestamp
        new_values.append(str(datetime.now()))  # Set current timestamp

        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        sql = f"UPDATE {self.files_table} SET " + ", ".join(f"{field} = ?" for field in fields) + " WHERE id = ?"

        try:
            cursor.execute(sql, tuple(new_values + [id]))  # Execute the update query
            conn.commit()
        except sqlite3.IntegrityError:
            print("Key values already exist in table")  # Log integrity constraint error
        conn.close()

    def get_file(self, cred):
        """
        Retrieves a file from the database using file ID or stored name.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.files_table} WHERE id = ? OR sname = ?", (cred, cred))
        row = cursor.fetchone()
        conn.close()
        return self.row_to_dict_file(row) if row else None  # Convert row to dictionary if file exists

    def get_user_files(self, owner_id):
        """
        Retrieves all files owned by a specific user.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.files_table} WHERE owner_id = ?", (owner_id,))
        ans = cursor.fetchall()
        conn.close()
        return [self.row_to_dict_file(file) for file in ans]  # Convert each row to dictionary

    def get_files(self, parent):
        """
        Retrieves all files within a specific directory.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.files_table} WHERE parent = ?", (parent,))
        ans = cursor.fetchall()
        conn.close()
        return [self.row_to_dict_file(file) for file in ans]  # Convert each row to dictionary

    def add_file(self, file_dict):
        """
        Adds a new file entry to the database.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        columns = ', '.join(file_dict.keys())  # Extract column names
        values = ', '.join(['?'] * len(file_dict))  # Create placeholders for values
        sql = f"INSERT INTO {self.files_table} ({columns}) VALUES ({values})"

        try:
            cursor.execute(sql, list(file_dict.values()))  # Execute the insert query
            conn.commit()
        except sqlite3.IntegrityError:
            print("Key values already exist in table")  # Log integrity constraint error
        conn.close()

    def delete_file(self, id):
        """
        Moves a file to the deleted table (soft delete) instead of permanently removing it.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()

        cursor.execute(f"SELECT * FROM {self.deleted_table} WHERE id = ?", (id,))
        ans = cursor.fetchall()

        if not ans:  # If the file isn't already marked as deleted
            sql = f"INSERT INTO {self.deleted_table} (id, owner_id, time_to_delete) VALUES (?, ?, ?)"
            cursor.execute(sql, [id, self.get_file(id)["owner_id"], str(timedelta(days=30) + datetime.now())])
            conn.commit()
            conn.close()
            return False  # File is now marked as deleted
        else:
            cursor.execute(f"DELETE FROM {self.files_table} WHERE id = ?", (id,))  # Permanently delete the file
            cursor.execute(f"DELETE FROM {self.permissions_table} WHERE file_id = ?", (id,))  # Remove access permissions
            cursor.execute(f"DELETE FROM {self.deleted_table} WHERE id = ?", (id,))  # Remove entry from deleted table
            conn.commit()
            conn.close()
```

```
        return True  # File has been permanently deleted

    def get_all_files(self):
        """
        Retrieves all files stored in the database.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.files_table}")
        ans = cursor.fetchall()
        conn.close()
        return [self.row_to_dict_file(file) for file in ans]  # Convert rows to dictionaries

    def add_directory(self, directory_dict):
        """
        Adds a new directory entry to the database.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        columns = ', '.join(directory_dict.keys())  # Extract column names
        values = ', '.join(['?'] * len(directory_dict))  # Create placeholders for values
        sql = f"INSERT INTO {self.directories_table} ({columns}) VALUES ({values})"  # Construct SQL query

        try:
            cursor.execute(sql, list(directory_dict.values()))  # Execute the insert query
            conn.commit()
        except sqlite3.IntegrityError:
            print("Key values already exist in table")  # Log constraint violation
        conn.close()

    def get_user_directories(self, owner_id):
        """
        Retrieves all directories owned by a specific user.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.directories_table} WHERE owner_id = ?", (owner_id,))
        ans = cursor.fetchall()
        conn.close()
        return [self.row_to_dict_directory(directory) for directory in ans]  # Convert rows to dictionaries

    def get_directories(self, parent):
        """
        Retrieves all directories within a specific parent directory.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.directories_table} WHERE parent = ?", (parent,))
        ans = cursor.fetchall()
        conn.close()
        return [self.row_to_dict_directory(directory) for directory in ans]  # Convert rows to dictionaries

    def get_directory(self, id):
        """
        Retrieves directory information based on its ID.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.directories_table} WHERE id = ?", (id,))
        row = cursor.fetchone()
        conn.close()
        return self.row_to_dict_directory(row) if row else None  # Convert row to dictionary if directory exists

    def delete_directory(self, id):
        """
        Moves a directory to the deleted table instead of permanently removing it.
        If already marked as deleted, the directory and its contents are permanently removed.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.deleted_table} WHERE id = ?", (id,))
        ans = cursor.fetchall()

        if not ans:  # If the directory is not already marked as deleted
            sql = f"INSERT INTO {self.deleted_table} (id, owner_id, time_to_delete) VALUES (?, ?, ?)"
            cursor.execute(sql, [id, self.get_directory(id)["owner_id"], str(timedelta(days=30) + datetime.now())])
            conn.commit()
            conn.close()
            return False  # Directory is now marked as deleted
        else:
            cursor.execute(f"DELETE FROM {self.directories_table} WHERE id = ?", (id,))  # Delete directory
            cursor.execute(f"DELETE FROM {self.files_table} WHERE parent = ?", (id,))  # Delete files inside the directory
            cursor.execute(f"DELETE FROM {self.permissions_table} WHERE file_id = ?", (id,))  # Remove permissions
            cursor.execute(f"DELETE FROM {self.deleted_table} WHERE id = ?", (id,))  # Remove deletion record
            conn.commit()
            conn.close()
            return True  # Directory has been permanently deleted
```

```
    def update_directory(self, id, fields, new_values):
        """
        Updates directory attributes in the database.
        """
        if not isinstance(fields, list):
            fields = [fields]  # Ensure fields are a list
        if not isinstance(new_values, list):
            new_values = [new_values]  # Ensure values are a list

        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        sql = f"UPDATE {self.directories_table} SET " + ", ".join(f"{field} = ?" for field in fields) + " WHERE id = ?"

        try:
            cursor.execute(sql, tuple(new_values + [id]))  # Execute update query
            conn.commit()
        except sqlite3.IntegrityError:
            print("Key values already exist in table")  # Log integrity constraint error
        conn.close()

    def get_directory_files(self, parent_id):
        """
        Retrieves all files within a specific directory.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.files_table} WHERE parent = ?", (parent_id,))
        ans = cursor.fetchall()
        conn.close()
        return [self.row_to_dict_file(file) for file in ans]  # Convert rows to dictionaries

    def get_user_directory_files(self, user_id, parent_id):
        """
        Retrieves all files in a specific directory that belong to a particular user.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.files_table} WHERE owner_id = ? AND parent = ?", (user_id, parent_id))
        ans = cursor.fetchall()
        conn.close()
        return [self.row_to_dict_file(file) for file in ans]  # Convert rows to dictionaries

    def get_sub_directories(self, parent_id):
        """
        Retrieves all subdirectories within a specific parent directory.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.directories_table} WHERE parent = ?", (parent_id,))
        ans = cursor.fetchall()
        conn.close()
        return [self.row_to_dict_directory(directory) for directory in ans]  # Convert rows to dictionaries

    def get_all_directories(self):
        """
        Retrieves all directories stored in the database.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.directories_table}")
        ans = cursor.fetchall()
        conn.close()
        return [self.row_to_dict_directory(directory) for directory in ans]  # Convert rows to dictionaries

    def get_share_file(self, file_id, user_id):
        """
        Retrieves a shared file entry for a specific user.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.permissions_table} WHERE file_id = ? AND user_id = ?", (file_id, user_id))
        row = cursor.fetchone()
        conn.close()
        return row  # Returns the shared file record if found

    def get_all_share_files(self, user_id):
        """
        Retrieves all files shared with a specific user.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT f.* FROM {self.files_table} f JOIN {self.permissions_table} p ON f.id = p.file_id WHERE
p.user_id = ? AND p.read = ?", (user_id, "True"))
        ans = cursor.fetchall()
        conn.close()
        return [self.row_to_dict_file(file) for file in ans]  # Convert rows to dictionaries
```

```
    def get_all_share_directories(self, user_id):
        """
        Retrieves all shared directories accessible by a user.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT d.* FROM {self.directories_table} d JOIN {self.permissions_table} p ON d.id = p.file_id
WHERE p.user_id = ? AND p.read = ?", (user_id, "True"))
        ans = cursor.fetchall()
        conn.close()
        return [self.row_to_dict_directory(directory) for directory in ans]  # Convert rows to dictionaries

    def get_perms(self, id):
        """
        Retrieves permission settings for a specific file or directory.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.permissions_table} WHERE id = ?", (id,))
        row = cursor.fetchone()
        conn.close()
        return row  # Returns permission details if found

    def create_share(self, id, owner_id, file_id, user_id, new_perms):
        """
        Creates a new sharing entry, granting permissions to a user.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        sql = f"INSERT INTO {self.permissions_table} (id, file_id, owner_id, user_id, read, write, del, rename, download,
share) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)"

        try:
            cursor.execute(sql, [id, file_id, owner_id, user_id] + new_perms)  # Insert new permission settings
            conn.commit()
        except sqlite3.IntegrityError:
            print("Key values already exist in table")  # Handle duplicate entry error
        conn.close()

    def update_sharing_premissions(self, file_id, user_id, new_perms):
        """
        Updates sharing permissions for a file or folder.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        sql = f"UPDATE {self.permissions_table} SET read = ?, write = ?, del = ?, rename = ?, download = ?, share = ?
WHERE file_id = ? AND user_id = ?"
        cursor.execute(sql, new_perms + [file_id, user_id])  # Update permission settings
        conn.commit()
        conn.close()

    def get_file_perms(self, user_id, file_id):
        """
        Retrieves specific permission settings for a user on a given file.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT read, write, del, rename, download, share FROM {self.permissions_table} WHERE user_id = ?
AND file_id = ?", (user_id, file_id))
        row = cursor.fetchone()
        conn.close()
        return row  # Returns the permission settings

    def remove_share(self, user_id, id):
        """
        Removes a shared file or directory from a user's access list.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"DELETE FROM {self.permissions_table} WHERE file_id = ? AND user_id = ?", (id, user_id))
        conn.commit()
        conn.close()

    def get_directory_contents(self, directory_id):
        """
        Retrieves all files and subdirectories within a given directory.
        Returns a list of tuples (full_path, relative_path) for zipping.
        """
        contents = []
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()

        # Get all files in the directory
        cursor.execute(f"SELECT sname, fname FROM {self.files_table} WHERE parent = ?", (directory_id,))
        files = cursor.fetchall()
```

```
        for file_id, file_name in files:
            full_path = os.path.join(f"
{os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))}\\server\\cloud", str(file_id))  # Get
absolute file path
            relative_path = file_name  # Set relative path for zip archive
            contents.append((full_path, relative_path))

        # Get all subdirectories
        cursor.execute(f"SELECT id, name FROM {self.directories_table} WHERE parent = ?", (directory_id,))
        subdirectories = cursor.fetchall()

        for subdirectory_id, subdirectory_name in subdirectories:
            # Recursively retrieve subdirectory contents
            subdir_contents = self.get_directory_contents(subdirectory_id)

            for full_path, relative_path in subdir_contents:
                contents.append((full_path, os.path.join(subdirectory_name, relative_path)))  # Maintain folder structure

        return contents  # Return complete list of directory contents



    def get_deleted_files(self, owner_id):
        """
        Retrieves all files marked for deletion that belong to a specific user.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT f.* FROM {self.files_table} f JOIN {self.deleted_table} d ON f.id = d.id WHERE d.owner_id
= ?", (owner_id,))
        ans = cursor.fetchall()
        conn.close()
        return [self.row_to_dict_file(file) for file in ans]  # Convert rows to dictionaries

    def get_deleted_directories(self, owner_id):
        """
        Retrieves all directories marked for deletion that belong to a specific user.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT d.* FROM {self.directories_table} d JOIN {self.deleted_table} del ON d.id = del.id WHERE
del.owner_id = ?", (owner_id,))
        ans = cursor.fetchall()
        conn.close()
        return [self.row_to_dict_directory(directory) for directory in ans]  # Convert rows to dictionaries

    def get_deleted(self, id):
        """
        Checks if a file or directory is marked as deleted.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.deleted_table} WHERE id = ?", (id,))
        row = cursor.fetchone()
        conn.close()
        return row  # Returns the deleted entry if found

    def get_deleted_time(self, id):
        """
        Retrieves the scheduled deletion time for a file or directory.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT time_to_delete FROM {self.deleted_table} WHERE id = ?", (id,))
        row = cursor.fetchone()
        conn.close()
        return row  # Returns the deletion time if found

    def recover(self, id):
        """
        Restores a previously deleted file or directory from the deleted table.
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"DELETE FROM {self.deleted_table} WHERE id = ?", (id,))
        conn.commit()
        conn.close()

    def get_all_users(self):
        """
        Fetch all users in databse
        """
        conn = sqlite3.connect(self.database)
        cursor = conn.cursor()
        cursor.execute(f"SELECT * FROM {self.users_table}")
        ans = cursor.fetchall()
        conn.close()
```

```
return ans
```