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

from modules import database_handling  # Handles database operations

from email.message import EmailMessage  # Used for sending email notifications
from datetime import datetime, timedelta  # Handles date and time operations

import ssl, smtplib, os, bcrypt, secrets, uuid, traceback, zipfile, io, random  # Security, encryption, and file handling

class User:
    """
    Represents a user in the system.
    Used for transferring data between user instances and JSON format.
    """

    def __init__(self, id, email, username, password, salt=bcrypt.gensalt(), last_code=-1, valid_until=None,
verified=False, subscription_level=0, admin_level=0, cookie="", cookie_expiration=-1):
        if id is None:
            self.id = ClientRequests().gen_user_id()  # Generate new user ID if not provided
        else:
            self.id = id
        self.email = email
        self.username = username
        self.password = password
        self.salt = salt
        self.last_code = last_code
        self.valid_until = valid_until if valid_until else str(datetime.now())  # Default to current date
        self.verified = verified
        self.subscription_level = subscription_level
        self.admin_level = admin_level
        self.cookie = cookie
        self.cookie_expiration = cookie_expiration

class File:
    """
    Represents a file in the system.
    Used for transferring data between file instances and JSON format.
    """

    def __init__(self, id, sname, fname, parent, owner_id, size, last_edit=None):
        if id is None:
            self.id = ClientRequests().gen_file_id()  # Generate a unique file ID if not provided
        else:
            self.id = id

        if sname is None:
            self.sname = ClientRequests().gen_file_name()  # Generate a unique stored name
        else:
            self.sname = sname

        self.fname = fname  # Original file name
        self.parent = parent  # Parent directory ID
        self.owner_id = owner_id  # Owner user ID
        self.size = size  # File size in bytes
        self.last_edit = last_edit if last_edit else str(datetime.now())  # Default to current timestamp

class Directory:
    """
    Represents a directory in the system.
    """

    def __init__(self, id, name, parent, owner_id):
        if id is None:
            self.id = ClientRequests().gen_file_id()  # Generate a directory ID if not provided
        else:
            self.id = id
        self.name = name  # Directory name
        self.parent = parent  # Parent directory ID
        self.owner_id = owner_id  # Owner user ID

class ClientRequests:
    """
    Handles client requests related to user authentication, file management,
    and database interactions.
    """

    def __init__(self):
        self.pepper_file = f"{os.path.dirname(os.path.abspath(__file__))}\\pepper.txt"  # Path to the pepper file
        self.server_path = f"{os.path.dirname(os.path.dirname(os.path.abspath(__file__)))}"  # Root server path
        self.gmail = "idancyber3102@gmail.com"  # Email for sending verification emails
        self.gmail_password = "nkjg eaom gzne nyfa"  # SMTP email password (should be stored securely)
        self.get_pepper()  # Load or generate pepper value
        self.db = database_handling.DataBase()  # Initialize database handler

    def get_pepper(self):
        """
```

```
        Retrieves the pepper value used for hashing passwords.
        If the pepper file does not exist, a new one is created.
        """
        if not os.path.isfile(self.pepper_file):
            new_pepper = secrets.token_hex(2000)  # Generate a random 2000-byte hex string
            with open(self.pepper_file, 'wb') as file:
                file.write(new_pepper.encode())  # Write the pepper to the file

        with open(self.pepper_file, 'rb') as file:
            self.pepper = file.read()  # Load the pepper from the file

    def user_exists(self, username):
        """
        Checks if a username is already registered in the database.
        Returns True if the user exists, otherwise False.
        """
        user = self.db.get_user(username)  # Retrieve user record
        return user is not None  # Return True if user exists

    def verified(self, cred):
        """
        Checks if a user account is verified.
        Returns True if verified, otherwise False.
        """
        user = self.db.get_user(cred)
        if user is None:
            return False
        user = User(**user)  # Convert dictionary to User object
        return user.verified

    def email_registered(self, email):
        """
        Checks if an email is already registered.
        Returns True if the email is found in the database.
        """
        user = self.db.get_user(email)
        return user is not None

    def login_validation(self, cred, password):
        """
        Validates user login credentials.
        Returns True if credentials are correct, otherwise False.
        """
        user = self.db.get_user(cred)
        if user is None:
            return False
        user = User(**user)
        return user.password == self.hash_password(password, user.salt)  # Compare stored and hashed passwords

    def signup_user(self, user_details):
        """
        Registers a new user in the database.
        Hashes the password and assigns a unique user ID.
        """
        new_user = User(None, *user_details)  # Create a new user object
        new_user.password = self.hash_password(new_user.password, new_user.salt)  # Hash password
        new_user.cookie = self.generate_cookie(new_user.id)  # Generate session cookie
        self.db.add_user(vars(new_user))  # Store user details in the database

    def verify_user(self, email):
        """
        Marks a user as verified based on their email.
        """
        id = self.db.get_user_id(email)  # Retrieve user ID
        self.db.update_user(id, "verified", True)  # Set verified flag to True

    def delete_user(self, id):
        """
        Deletes a user account along with their files and directories.
        """
        files = self.db.get_files(id)  # Get user's files
        for file in files:
            try:
                file = File(**file)  # Convert to File object
                os.remove(self.server_path + "\\cloud\\" + file.sname)  # Remove file from storage
            except:
                print(traceback.format_exc())  # Log error if file deletion fails
                continue

        # Remove user profile picture if it exists
        if os.path.exists(f"{self.server_path}\\user icons\\{id}.ico"):
            os.remove(f"{self.server_path}\\user icons\\{id}.ico")

        self.db.remove_user(id)  # Remove user from the database


    def send_reset_mail(self, email):
```

```
        """
        Sends a password reset email with a randomly generated 6-digit code.
        Stores the code in the database with a 10-minute expiration.
        """
        id = self.db.get_user_id(email)  # Retrieve user ID from email
        code = random.randint(100000, 999999)  # Generate 6-digit reset code
        valid_until = str(timedelta(minutes=10) + datetime.now())  # Set expiration time
        self.db.update_user(id, ["last_code", "valid_until"], [code, valid_until])  # Store code in database

        em = EmailMessage()  # Build email
        em["From"] = self.gmail
        em["To"] = email
        em["Subject"] = "Password reset code"
        body = f"Your password reset code is: {code}\nCode is valid for 10 minutes"
        em.set_content(body)
        self.send_mail(em, email)  # Send email

    def send_verification(self, email):
        """
        Sends an account verification email with a randomly generated 6-digit code.
        Stores the code in the database with a 30-minute expiration.
        """
        id = self.db.get_user_id(email)  # Retrieve user ID from email
        code = random.randint(100000, 999999)  # Generate verification code
        valid_until = str(timedelta(minutes=30) + datetime.now())  # Set expiration time
        self.db.update_user(id, ["last_code", "valid_until"], [code, valid_until])  # Store code in database

        em = EmailMessage()  # Build email
        em["From"] = self.gmail
        em["To"] = email
        em["Subject"] = "Account Verification"
        body = f"Your account verification code is: {code}\nCode is valid for 30 minutes"
        em.set_content(body)
        self.send_mail(em, email)  # Send email

    def send_welcome_mail(self, email):
        """
        Sends a welcome email to a new user.
        """
        em = EmailMessage()  # Build email
        em["From"] = self.gmail
        em["To"] = email
        em["Subject"] = "Welcome!"
        body = f"""Welcome to IdanCloud!
        Currently, you are at the basic subscription level and are welcome to upgrade at any time.
        \nYou have 100 GB of storage, a max file size of 50 MB, an upload speed of 5 MB/s, and a download speed of 10
MB/s.
        \nFor any questions, contact us at {self.gmail}.
        \nIdanCloud Â©2024 - 2025"""
        em.set_content(body)
        self.send_mail(em, email)  # Send email

    def send_mail(self, em, send_to):
        """
        Sends an email using an SMTP secure connection.
        """
        context = ssl.create_default_context()  # Create a secure SSL context
        with smtplib.SMTP_SSL("smtp.gmail.com", 465, context=context) as smtp_server:
            smtp_server.login(self.gmail, self.gmail_password)  # Authenticate
            smtp_server.sendmail(self.gmail, send_to, em.as_string())  # Send email

    def check_code(self, email, code):
        """
        Checks if the provided verification or password reset code is valid.
        Returns:
            "ok" if the code is correct,
            "code" if incorrect,
            "time" if expired.
        """
        user = self.db.get_user(email)
        if user is None:
            return False  # User not found

        user = User(**user)  # Convert dictionary to User object

        if self.str_to_date(user.valid_until) < datetime.now():
            return "time"  # Expired code
        elif not code.isdigit() or int(user.last_code) != int(code) or int(code) < 0:
            return "code"  # Incorrect code
        return "ok"  # Valid code

    def hash_password(self, password, salt):
        """
        Hashes a password using bcrypt along with a pepper for added security.
        """
        return bcrypt.hashpw(password.encode() + self.pepper, salt)  # Hash password
```

```
    def change_password(self, email, new_password):
        """
        Changes a user's password.
        Hashes the new password with a salt and pepper before updating the database.
        """
        id = self.db.get_user_id(email)  # Retrieve user ID
        salt = bcrypt.gensalt()  # Generate new salt
        password = self.hash_password(new_password, salt)  # Hash new password
        self.db.update_user(id, ["salt", "password"], [salt, password])  # Store new credentials

    def get_user_data(self, cred):
        """
        Retrieves user data from the database.
        Used in server-side operations where direct database access is not available.
        """
        return self.db.get_user(cred)


    def get_files(self, owner_id, parent, name_filter=None):
        """
        Retrieves all files belonging to a specific owner within a given parent directory.
        Supports filtering files by name.
        """
        if parent == "":
            files = self.db.get_user_files(owner_id)  # Get all files owned by the user
        else:
            files = self.db.get_files(parent)  # Get files in the specified directory

        parsed_files = []
        for file in files:
            file = File(**file)  # Convert dictionary to File object
            if (name_filter is None or name_filter.lower() in file.fname.lower()) and file.parent == parent and not
self.is_deleted(file.id) and file.sname != owner_id:
                last_edit = self.str_to_date(file.last_edit)  # Convert last edit time to datetime

                if file.owner_id == owner_id:
                    to_add = f"{file.fname}~{last_edit}~{file.size}~{file.id}"
                else:
                    to_add = f"{file.fname}~{last_edit}~{file.size}~{file.id}~
{"".join(self.db.get_user_values(file.owner_id, ["username"]))}"
                    to_add += "~" + "~".join(self.get_perms(owner_id, file.id))  # Append permissions

                parsed_files.append(to_add)  # Add parsed file entry to list

        return parsed_files  # Return list of formatted file information

    def get_directories(self, owner_id, parent, name_filter=None):
        """
        Retrieves all directories belonging to a specific owner within a given parent directory.
        Supports filtering directories by name.
        """
        if parent == "":
            directories = self.db.get_user_directories(owner_id)  # Get all directories owned by the user
        else:
            directories = self.db.get_directories(parent)  # Get directories within the specified parent

        parsed_directories = []
        for directory in directories:
            directory = Directory(**directory)  # Convert dictionary to Directory object

            if (name_filter is None or name_filter.lower() in directory.name.lower()) and directory.parent == parent and
not self.is_deleted(directory.id):
                size = self.directory_size(directory.owner_id, directory.id)  # Get directory size
                last_change = self.get_directory_last_change(directory.id)  # Get last modification date

                if last_change == datetime.min:
                    last_change = ""  # If no changes, return an empty string

                if directory.owner_id == owner_id:
                    to_add = f"{directory.name}~{directory.id}~{last_change}~{size}"
                else:
                    to_add = f"{directory.name}~{directory.id}~{last_change}~{size}~
{"".join(self.db.get_user_values(directory.owner_id, ["username"]))}"
                    to_add += "~" + "~".join(self.get_perms(owner_id, directory.id))  # Append permissions

                parsed_directories.append(to_add)  # Add parsed directory entry to list

        return parsed_directories  # Return list of formatted directory information

    def get_directory_last_change(self, id, lastest_edit=datetime.min):
        """
        Recursively retrieves the latest modification timestamp of a directory.
        Checks both files and subdirectories for the most recent change.
        """
        for directory in self.db.get_directories(id):  # Get all subdirectories
            directory = Directory(**directory)  # Convert dictionary to Directory object
            lastest_edit = max(lastest_edit, self.get_directory_last_change(directory.id, lastest_edit))  # Recursively
```

check for the latest change

```
        files = self.db.get_files(id)  # Get all files in the directory
        if files is None:
            return lastest_edit  # Return lastest_edit if there are no files

        for file in files:
            file = File(**file)  # Convert dictionary to File object
            current_last_change = self.str_to_date(file.last_edit)  # Convert last edit time to datetime

            if current_last_change > lastest_edit:
                lastest_edit = current_last_change  # Update latest edit timestamp

        return lastest_edit  # Return the most recent modification time


    def change_level(self, id, new_level):
        """
        Updates a user's subscription level.
        """
        self.db.update_user(id, "subscription_level", new_level)

    def change_username(self, id, new_username):
        """
        Updates a user's username.
        """
        self.db.update_user(id, "username", new_username)

    def generate_cookie(self, id):
        """
        Generates a unique authentication cookie and stores it in the database.
        """
        cookie = str(secrets.token_hex(256))  # Generate a unique cookie
        while self.db.get_user(cookie) is not None:
            cookie = str(secrets.token_hex(256))  # Ensure uniqueness
        cookie_expiration = str(timedelta(weeks=4) + datetime.now())  # Set expiration time
        self.db.update_user(id, ["cookie", "cookie_expiration"], [cookie, cookie_expiration])  # Store cookie

    def get_cookie(self, id):
        """
        Retrieves a user's authentication cookie.
        """
        return self.db.get_user_values(id, ["cookie"])[0]

    def cookie_expired(self, id):
        """
        Checks if a user's authentication cookie has expired.
        """
        user = self.db.get_user(id)
        if user is None:
            return True  # User not found, treat as expired
        user = User(**user)  # Convert dictionary to User object
        return self.str_to_date(user.cookie_expiration) < datetime.now()  # Compare expiration date

    def get_user_id(self, cred):
        """
        Retrieves a user ID based on email or username.
        """
        return self.db.get_user_id(cred)

    def new_file(self, sname, file_name, parent, owner_id, size):
        """
        Creates a new file entry in the database.
        """
        file = File(None, sname, file_name, parent, owner_id, size)  # Create a File object
        self.db.add_file(vars(file))  # Store file in the database

    def get_file_id(self, file_name):
        """
        Retrieves the file ID based on the file name.
        Returns None if the file does not exist.
        """
        file = self.db.get_file(file_name)  # Fetch file from database
        if file is None:
            return None
        file = File(**file)  # Convert dictionary to File object
        return file.id  # Return file ID

    def get_file_sname(self, file_id):
        """
        Retrieves the stored name (unique filename) for a given file ID.
        """
        file = self.db.get_file(file_id)
        if file is None:
            return None
        file = File(**file)
        return file.sname  # Return stored filename
```

```python
def get_file_fname(self, file_id):
    """
    Retrieves the original filename for a given file ID.
    """
    file = self.db.get_file(file_id)
    if file is None:
        return None
    file = File(**file)
    return file.fname  # Return original filename

def is_file_owner(self, owner_id, file_id):
    """
    Checks if a user is the owner of a given file.
    Returns True if the user owns the file, otherwise False.
    """
    file = self.db.get_file(file_id)
    if file is None:
        return None
    file = File(**file)
    return file.owner_id == owner_id  # Compare owner ID

def is_dir_owner(self, owner_id, dir_id):
    """
    Checks if a user is the owner of a given directory.
    Root directory ("") is always valid.
    """
    if dir_id == "":
        return True  # Allow access to the root directory
    directory = self.db.get_directory(dir_id)
    if directory is None:
        return None
    directory = Directory(**directory)
    return directory.owner_id == owner_id  # Compare owner ID

def rename_file(self, id, new_name):
    """
    Renames a file by updating its original filename in the database.
    """
    self.db.update_file(id, ["fname"], new_name)  # Update filename in the database

def update_file_size(self, file_id, new_size):
    """
    Updates the size of a file in the database.
    """
    self.db.update_file(file_id, "size", new_size)  # Update file size field

def rename_directory(self, id, new_name):
    """
    Renames a directory by updating its name in the database.
    """
    self.db.update_directory(id, ["name"], new_name)  # Update directory name

def delete_file(self, id):
    """
    Deletes a file from the storage and database if it exists.
    """
    sname = self.get_file_sname(id)  # Get stored filename
    if os.path.exists(f"{self.server_path}\\cloud\\{sname}") and self.db.delete_file(id):
        os.remove(f"{self.server_path}\\cloud\\{sname}")  # Delete file from storage

def create_folder(self, name, parent, owner_id):
    """
    Creates a new folder (directory) and stores it in the database.
    """
    directory = Directory(None, name, parent, owner_id)  # Create directory object
    self.db.add_directory(vars(directory))  # Store in database

def valid_directory(self, directory_id, user_id):
    """
    Checks if a directory is valid and accessible by a user.
    Returns True if the user is the owner or the directory is shared with them.
    """
    directory = self.db.get_directory(directory_id)
    if directory is None:
        return False  # Directory does not exist
    directory = Directory(**directory)
    return directory.owner_id == user_id or self.is_shared(user_id, directory_id)

def is_shared(self, user_id, directory_id):
    """
    Checks if a directory is shared with a user.
    Recursively checks parent directories if necessary.
    """
    directory = self.db.get_directory(directory_id)
    if directory is None:
        return False  # Directory not found
```

```
        directory = Directory(**directory)
        shared_dir = self.db.get_share_file(directory_id, user_id)

        while shared_dir is None:
            directory = self.db.get_directory(directory.parent)  # Move up in directory hierarchy
            if directory is None:
                return False  # Stop if root is reached
            directory = Directory(**directory)
            directory_id = directory.id
            shared_dir = self.db.get_share_file(directory_id, user_id)

        return True  # Directory is shared

    def get_dir_name(self, id):
        """
        Retrieves the name of a directory based on its ID.
        """
        if id == "":
            return ""  # Root directory has no name
        directory = self.db.get_directory(id)
        if directory is None:
            return None
        directory = Directory(**directory)
        return directory.name  # Return directory name

    def get_parent_directory(self, id):
        """
        Retrieves the parent directory ID of a given directory.
        """
        if id == "":
            return ""  # Root directory has no parent
        directory = self.db.get_directory(id)
        if directory is None:
            return None
        directory = Directory(**directory)
        return directory.parent  # Return parent directory ID

    def get_file_parent_directory(self, id):
        """
        Retrieves the parent directory ID of a given file.
        """
        if id == "":
            return ""  # No parent for invalid file
        file = self.db.get_file(id)
        if file is None:
            return None
        file = File(**file)
        return file.parent  # Return parent directory ID of the file

    def get_full_path(self, id):
        """
        Constructs the full directory path from the root to the given directory.
        """
        if id == "":
            return ""  # Root directory
        path = [""]  # Initialize path list

        directory = self.db.get_directory(id)
        if directory is None:
            return None  # Directory not found
        directory = Directory(**directory)
        path.append(directory.name)  # Add directory name

        while directory.parent != "":
            directory = self.db.get_directory(directory.parent)
            if directory is None:
                return None  # Parent directory not found
            directory = Directory(**directory)
            path.append(directory.name)  # Append parent directory name

        path = "\\".join(path[::-1])  # Reverse and join the path
        return path  # Return full directory path

    def delete_directory(self, id):
        """
        Recursively deletes a directory and all its subdirectories and files.
        """
        sub_dirs = self.db.get_sub_directories(id)  # Get all subdirectories
        if sub_dirs:
            for sub_dir in sub_dirs:
                self.delete_directory(sub_dir["id"])  # Recursively delete subdirectories

        files = self.db.get_directory_files(id)  # Get all files in directory
        if self.db.delete_directory(id):  # Delete directory in database
            for file in files:
                try:
                    file = File(**file)  # Convert to File object
```

```
                os.remove(self.server_path + "\\cloud\\" + file.sname)  # Delete from storage
            except:
                print(traceback.format_exc())  # Log deletion failure
                continue


    def directory_size(self, user_id, id):
        """
        Calculates the total size of a directory, including its files and subdirectories.
        """
        total = 0
        files = self.db.get_user_directory_files(user_id, id)  # Retrieve all files in the directory

        for file in files:
            try:
                file = File(**file)  # Convert dictionary to File object
                file_path = self.server_path + "\\cloud\\" + file.sname  # Construct full file path
                if os.path.exists(file_path):
                    total += os.path.getsize(file_path)  # Add file size to total
            except:
                print(traceback.format_exc())  # Log error
                continue

        # Recursively calculate size of subdirectories
        child_dirs = self.db.get_directories(id)
        for child_dir in child_dirs:
            total += self.directory_size(user_id, child_dir["id"])

        return total  # Return total directory size

    def get_user_storage(self, id):
        """
        Calculates total storage used by a user.
        """
        return self.directory_size(id, "")  # Get size of all files and directories owned by the user

    def clean_db(self, files_uploading):
        """
        Cleans the database by removing orphaned files and directories.
        Ensures that files stored in the cloud exist in the database and vice versa.
        """
        for name in os.listdir(self.server_path + "\\cloud"):
            try:
                if (
                    self.db.get_file(name) is None and
                    self.db.get_user(name) is None and
                    not any(obj.name == name for obj in files_uploading.values())
                ):
                    os.remove(self.server_path + "\\cloud\\" + name)  # Remove orphaned files
            except:
                print(traceback.format_exc())
                continue

        db_files = self.db.get_all_files()
        for file in db_files:
            try:
                file_path = self.server_path + "\\cloud\\" + file["sname"]
                if not os.path.exists(file_path) or (self.db.get_directory(file["parent"]) is None and file["parent"] !=
""):
                    self.db.delete_file(file["id"])  # Remove file from database
                elif self.is_deleted(file["id"]) and self.str_to_date(self.db.get_deleted_time(file["id"])[0]) <
datetime.now():
                    self.db.delete_file(file["id"])  # Permanently delete expired files
            except:
                print(traceback.format_exc())
                continue

        db_directories = self.db.get_all_directories()
        for directory in db_directories:
            try:
                if self.db.get_user(directory["owner_id"]) is None or (self.db.get_directory(directory["parent"]) is None
and directory["parent"] != ""):
                    self.db.delete_directory(directory["id"])  # Remove orphaned directories
            except:
                print(traceback.format_exc())
                continue

    def get_admin_table(self):
        """
        Returns admin table with info on users
        """
        table = ""
        table = self.db.get_all_users()
        return table

    def get_user_total_files(self, user_id):
        """
```

```
        Returns total number of files user has
        """
        return len(self.db.get_user_files(user_id))

    def get_share_options(self, file_id, user_cred):
        """
        Retrieves sharing permissions for a specific user on a file.
        """
        user_id = self.db.get_user_id(user_cred)  # Get user ID
        return self.db.get_share_file(file_id, user_id)  # Return sharing details

    def share_file(self, file_id, user_cred, perms):
        """
        Grants or updates sharing permissions for a file or directory.
        """
        user_id = self.db.get_user_id(user_cred)  # Retrieve user ID
        share = self.db.get_share_file(file_id, user_id)  # Get existing sharing record

        if user_id is None:
            return  # User not found

        if share is None:
            id = self.gen_perms_id()  # Generate a new sharing record ID
            file = self.db.get_file(file_id)
            directory = self.db.get_directory(file_id)

            if file is not None:
                file = File(**file)
                self.db.create_share(id, file.owner_id, file_id, user_id, perms)  # Create a new share entry
            elif directory is not None:
                directory = Directory(**directory)
                self.db.create_share(id, directory.owner_id, file_id, user_id, perms)  # Create a new share entry
        else:
            self.db.update_sharing_premissions(file_id, user_id, perms)  # Update existing permissions

    def get_share_files(self, user_id, parent, name_filter=None):
        """
        Retrieves all shared files accessible by a user.
        """
        files = self.db.get_all_share_files(user_id)  # Get all shared files
        parsed_files = []

        for file in files:
            file = File(**file)
            if (name_filter is None or name_filter.lower() in file.fname.lower()) and not self.is_deleted(file.id) and
file.sname != user_id:
                try:
                    last_edit = self.str_to_date(file.last_edit)  # Convert last edit time to datetime
                    owner_name = "".join(self.db.get_user_values(file.owner_id, ["username"]))
                    permissions = "~".join(self.get_perms(user_id, file.id))
                    parsed_files.append(f"{file.fname}~{last_edit}~{file.size}~{file.id}~{owner_name}~{permissions}")
                except:
                    continue

        return parsed_files  # Return list of shared files

    def get_share_directories(self, user_id, parent, name_filter=None):
        """
        Retrieves all shared directories accessible by a user.
        """
        directories = self.db.get_all_share_directories(user_id)
        parsed_directories = []

        for directory in directories:
            directory = Directory(**directory)
            if (name_filter is None or name_filter.lower() in directory.name.lower()) and not
self.is_deleted(directory.id):
                owner_name = "".join(self.db.get_user_values(directory.owner_id, ["username"]))
                size = self.directory_size(directory.owner_id, directory.id)  # Get directory size
                last_change = self.get_directory_last_change(directory.id)  # Get last modification date

                if last_change == datetime.min:
                    last_change = ""

                permissions = "~".join(self.get_perms(user_id, directory.id))
                parsed_directories.append(f"{directory.name}~{directory.id}~{last_change}~{size}~{owner_name}~
{permissions}")

        return parsed_directories  # Return list of shared directories

    def get_deleted_files(self, user_id, parent, name_filter=None):
        """
        Retrieves all files marked as deleted for a specific user.
        """
        files = self.db.get_deleted_files(user_id)  # Get all deleted files
        parsed_files = []
```

```
        for file in files:
            file = File(**file)
            if name_filter is None or name_filter.lower() in file.fname.lower() and file.sname != user_id:
                last_edit = self.str_to_date(file.last_edit)  # Convert last edit time
                parsed_files.append(f"{file.fname}~{last_edit}~{file.size}~{file.id}")

        return parsed_files  # Return list of deleted files

    def get_deleted_directories(self, user_id, parent, name_filter=None):
        """
        Retrieves all directories marked as deleted for a specific user.
        """
        directories = self.db.get_deleted_directories(user_id)
        parsed_directories = []

        for directory in directories:
            directory = Directory(**directory)
            if name_filter is None or name_filter.lower() in directory.name.lower():
                size = self.directory_size(directory.owner_id, directory.id)  # Get directory size
                last_change = self.db.get_deleted_time(directory.id)[0]  # Get deletion timestamp

                if last_change == datetime.min:
                    last_change = ""

                parsed_directories.append(f"{directory.name}~{directory.id}~{last_change}~{size}")

        return parsed_directories  # Return list of deleted directories

    def is_shared_directory(self, user_id, directory_id):
        """
        Checks if a directory or any of its parent directories is shared with a user.
        Returns the shared directory ID if found, otherwise None.
        """
        directory = self.db.get_directory(directory_id)
        if directory is None:
            return None

        directory = Directory(**directory)
        shared_dir = self.db.get_share_file(directory_id, user_id)

        while shared_dir is None:
            directory = self.db.get_directory(directory.parent)  # Move up in the hierarchy
            if directory is None:
                return None  # Stop if root is reached
            directory = Directory(**directory)
            directory_id = directory.id
            shared_dir = self.db.get_share_file(directory_id, user_id)

        return directory_id  # Return the shared directory ID


    def is_shared_file(self, user_id, file_id):
        """
        Checks if a file is shared with a user.
        Uses the parent directory to verify if it's shared.
        """
        parent = self.get_file_parent_directory(file_id)  # Get the parent directory of the file
        return self.is_shared_directory(user_id, parent)  # Check if the directory is shared

    def remove_share(self, user_id, id):
        """
        Removes sharing permissions for a file or directory.
        """
        self.db.remove_share(user_id, id)  # Remove sharing entry from the database

    def can_read(self, user_id, id):
        """
        Checks if a user has read permissions for a file or directory.
        """
        perms = self.get_perms(user_id, id)
        return self.is_file_owner(user_id, id) or self.is_dir_owner(user_id, id) or (perms is not None and perms[0] ==
"True")

    def can_write(self, user_id, id):
        """
        Checks if a user has write permissions for a file or directory.
        """
        perms = self.get_perms(user_id, id)
        return self.is_file_owner(user_id, id) or self.is_dir_owner(user_id, id) or (perms is not None and perms[1] ==
"True")

    def can_delete(self, user_id, id):
        """
        Checks if a user has delete permissions for a file or directory.
        """
        perms = self.get_perms(user_id, id)
        return self.is_file_owner(user_id, id) or self.is_dir_owner(user_id, id) or (perms is not None and perms[2] ==
```

```
"True")

    def can_rename(self, user_id, id):
        """
        Checks if a user has rename permissions for a file or directory.
        """
        perms = self.get_perms(user_id, id)
        return self.is_file_owner(user_id, id) or self.is_dir_owner(user_id, id) or (perms is not None and perms[3] ==
"True")

    def can_download(self, user_id, id):
        """
        Checks if a user has download permissions for a file or directory.
        """
        perms = self.get_perms(user_id, id)
        return self.is_file_owner(user_id, id) or self.is_dir_owner(user_id, id) or (perms is not None and perms[4] ==
"True")

    def can_share(self, user_id, id):
        """
        Checks if a user has permission to share a file or directory.
        """
        perms = self.get_perms(user_id, id)
        return self.is_file_owner(user_id, id) or self.is_dir_owner(user_id, id) or (perms is not None and perms[5] ==
"True")

    def get_perms(self, user_id, id):
        """
        Retrieves the permission settings for a file or directory.
        Checks shared parent directories if no permissions are found.
        """
        perms = self.db.get_file_perms(user_id, id)
        if perms is None:
            perms = self.db.get_file_perms(user_id, self.is_shared_directory(user_id, id))
        if perms is None:
            perms = self.db.get_file_perms(user_id, self.is_shared_file(user_id, id))
        return perms  # Return permission settings

    def zip_files(self, ids):
        """
        Creates a zip file containing the specified files and directories.
        """
        zip_buffer = io.BytesIO()  # Create an in-memory zip buffer

        with zipfile.ZipFile(zip_buffer, 'w', zipfile.ZIP_DEFLATED) as zf:
            for file_id in ids:
                if self.get_file_sname(file_id) is not None:
                    # It's a file
                    file_path = self.server_path + "\\cloud\\" + self.get_file_sname(file_id)
                    file_name = self.get_file_fname(file_id)
                    zf.write(file_path, file_name)  # Add file to zip archive
                elif self.get_dir_name(file_id) is not None:
                    # It's a directory, use zip_directory to add contents
                    directory_buffer = self.zip_directory(file_id)
                    with zipfile.ZipFile(directory_buffer, 'r') as dir_zip:
                        for name in dir_zip.namelist():
                            dir_name = self.get_dir_name(file_id)
                            zf.writestr(f"{dir_name}/{name}", dir_zip.read(name))  # Maintain folder structure

        zip_buffer.seek(0)  # Reset buffer position
        return zip_buffer  # Return zip file

    def zip_directory(self, directory_id):
        """
        Creates a zip archive containing all files and directories within the specified directory.
        """
        directory_contents = self.db.get_directory_contents(directory_id)  # Retrieve directory contents
        zip_buffer = io.BytesIO()  # Create in-memory zip buffer

        with zipfile.ZipFile(zip_buffer, 'w', zipfile.ZIP_DEFLATED) as zf:
            for full_path, relative_path in directory_contents:
                zf.write(full_path, relative_path)  # Add files to zip archive

        zip_buffer.seek(0)  # Reset buffer position
        return zip_buffer  # Return zip file buffer

    def is_deleted(self, id):
        """
        Checks if a file or directory is marked as deleted.
        """
        return self.db.get_deleted(id) is not None

    def recover(self, id):
        """
        Recovers a previously deleted file or directory.
        """
        self.db.recover(id)  # Remove deletion marker from database
```

```
def gen_user_id(self):
    """
    Generates a unique user ID that does not already exist in the database.
    """
    id = uuid.uuid4().hex
    while self.db.get_user(id) is not None:
        id = uuid.uuid4().hex  # Regenerate if duplicate exists
    return id

def gen_file_id(self):
    """
    Generates a unique file ID that does not already exist in the database.
    """
    id = uuid.uuid4().hex
    while self.db.get_file(id) is not None:
        id = uuid.uuid4().hex  # Regenerate if duplicate exists
    return id

def gen_file_name(self):
    """
    Generates a unique stored filename that does not already exist in the database.
    """
    name = uuid.uuid4().hex
    while self.db.get_file(name) is not None:
        name = uuid.uuid4().hex  # Regenerate if duplicate exists
    return name

def gen_perms_id(self):
    """
    Generates a unique permission ID that does not already exist in the database.
    """
    name = uuid.uuid4().hex
    while self.db.get_perms(name) is not None:
        name = uuid.uuid4().hex  # Regenerate if duplicate exists
    return name

@staticmethod
def str_to_date(str):
    """
    Converts a date string into a datetime object.
    """
    if str == "":
        return datetime.min  # Return minimum datetime if empty
    format = "%Y-%m-%d %H:%M:%S.%f"
    return datetime.strptime(str, format)  # Convert string to datetime
```