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

from modules.logger import Logger  # Custom logging module
from modules import networking, receive, gui, dialogs  # Importing necessary modules

import socket, sys, traceback  # Standard libraries

from PyQt6 import QtWidgets  # PyQt6 for GUI handling


class Application:
    """
    Handles the initialization of the PyQt application, networking,
    and GUI setup for client-side operations.
    """

    def __init__(self):
        sys.excepthook = dialogs.global_exception_handler  # Set a global exception handler for unhandled exceptions
        self.qtapp = QtWidgets.QApplication(sys.argv)  # Initialize PyQt application

        self.network = networking.Network()  # Initialize networking module
        self.window = gui.MainWindow(self.qtapp, self.network)  # Initialize main GUI window
        self.start_app()  # Start the application loop

        sys.exit(self.qtapp.exec())  # Start the PyQt event loop and exit when it finishes

    def start_app(self):
        """
        Starts the application by displaying the main window,
        initiating the connection page, and setting up the receive thread.
        """
        self.window.show()
        self.window.not_connected_page(False)  # Show the "not connected" page initially

        self.receive_thread = receive.ReceiveThread(self.network)  # Initialize background thread for receiving data
        self.receive_thread.reply_received.connect(self.handle_reply)  # Connect received replies to handler

        self.window.receive_thread = self.receive_thread  # Attach the receive thread to the main window
        self.window.protocol.connect_server(loop=True)  # Attempt to connect to the server

    def handle_reply(self, reply):
        """
        Handles replies received from the server.
        Parses the response and handles errors or disconnects if necessary.
        """
        try:
            self.network.logtcp('recv', reply)  # Log received data

            to_show = self.window.protocol.protocol_parse_reply(reply)  # Parse the server's reply
            print(to_show)

            if to_show == "Invalid reply from server":
                print(reply)

            # If exit request is acknowledged, disconnect
            if to_show == "Server acknowledged the exit message":
                print('Successfully exited')
                self.network.sock.close()
                sys.exit()

        except socket.error as err:
            print(traceback.format_exc())
            return
        except Exception as err:
            print(traceback.format_exc())
            return

def main():
    """
    Main function to initialize and start the client application.
    Sets up secure connection and GUI for user interaction.
    """
    app = Application()  # Initialize the client application

if __name__ == "__main__":  # Run the main function if the script is executed directly
    sys.stdout = Logger()  # Redirect standard output to the custom logger
    main()
```

```
# 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
```

```
# 2024 Â© Idan Hazay

# Global libraries
import os

# Global variables
SEP = "|"
LEN_FIELD = 4
CHUNK_SIZE = 524288

CLOUD_PATH = f"{os.getcwd()}\\cloud"
USER_ICONS_PATH = f"{os.getcwd()}\\user icons"
```

```
# 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
```

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

import os, traceback
from PyQt6.QtWidgets import QMessageBox, QApplication, QInputDialog
from PyQt6.QtGui import QIcon


def new_name_dialog(title, label, text=""):
    """Display an input dialog for the user to enter a new name."""
    app = QApplication.instance()  # Get existing QApplication instance
    app.setWindowIcon(QIcon(f"{os.path.dirname(os.path.dirname(os.path.abspath(__file__)))}/assets/icon.ico"))  # Set
window icon

    dialog = QInputDialog()
    dialog.setStyleSheet("font-size:18px;")
    dialog.setWindowTitle(title)
    dialog.setLabelText(label)
    dialog.setTextValue(text)
    dialog.resize(400, 300)  # Resize dialog to 400x300

    ok = dialog.exec()  # Show the dialog and wait for user input

    if ok == QInputDialog.DialogCode.Accepted:
        folder_name = dialog.textValue()
        if folder_name and folder_name != text:
            return folder_name  # Return input if it's not empty or unchanged


def show_confirmation_dialog(message):
    """Display a confirmation dialog with Yes/No options."""
    msg_box = QMessageBox()
    msg_box.setStyleSheet("font-size:18px;")
    msg_box.setIcon(QMessageBox.Icon.Question)
    msg_box.setWindowTitle("Confirmation")
    msg_box.setText(message)

    msg_box.setStandardButtons(QMessageBox.StandardButton.Yes | QMessageBox.StandardButton.No)
    msg_box.setDefaultButton(QMessageBox.StandardButton.Yes)
    result = msg_box.exec()  # Show the dialog and wait for user input

    return result == QMessageBox.StandardButton.Yes  # Return True if user clicks Yes


def global_exception_handler(exc_type, exc_value, exc_traceback):
    """Handle uncaught exceptions by displaying an error message."""
    error_message = "".join(traceback.format_exception(exc_type, exc_value, exc_traceback))
    print(f"Unhandled exception:\n{error_message}")

    QMessageBox.critical(
        None,
        "Application Error",
        f"An unexpected error occurred:\n\n{exc_value}",
        QMessageBox.StandardButton.Ok,
    )
```

```python
# 2024 Â© Idan Hazay

# Import required libraries
import hashlib, os, rsa, struct
from modules.config import *
from Crypto import Random
from Crypto.Cipher import AES
from base64 import b64encode, b64decode

class Encryption:
    """
    Provides encryption and decryption methods using AES and RSA.
    """
    def __init__(self):
        self.block_size = AES.block_size  # Block size for AES encryption

    def encrypt(self, plain_text, key):
        """
        Encrypts a plaintext string using AES encryption.
        Pads the plaintext to match the block size before encryption.
        """
        key = hashlib.sha256(key).digest()  # Derive a fixed-length key using SHA-256
        plain_text = self.pad(plain_text)  # Pad the plaintext
        iv = Random.new().read(self.block_size)  # Generate a random IV
        cipher = AES.new(key, AES.MODE_CBC, iv)  # Create AES cipher in CBC mode
        encrypted_text = cipher.encrypt(plain_text)  # Encrypt the plaintext
        return b64encode(iv + encrypted_text)  # Return encoded ciphertext with IV prepended

    def decrypt(self, encrypted_text, key):
        """
        Decrypts a ciphertext string using AES decryption.
        Removes padding after decryption.
        """
        key = hashlib.sha256(key).digest()  # Derive a fixed-length key using SHA-256
        encrypted_text = b64decode(encrypted_text)  # Decode the base64 ciphertext
        iv = encrypted_text[:self.block_size]  # Extract the IV
        cipher = AES.new(key, AES.MODE_CBC, iv)  # Create AES cipher in CBC mode
        plain_text = cipher.decrypt(encrypted_text[self.block_size:])  # Decrypt the ciphertext
        return self.unpad(plain_text)  # Remove padding

    def pad(self, plain_text):
        """
        Pads the plaintext to make its length a multiple of the block size.
        """
        number_of_bytes_to_pad = self.block_size - len(plain_text) % self.block_size
        ascii_string = chr(number_of_bytes_to_pad)
        padding_str = number_of_bytes_to_pad * ascii_string
        return plain_text + padding_str.encode()  # Append padding

    def unpad(self, plain_text):
        """
        Removes padding from the plaintext.
        """
        last_character = plain_text[len(plain_text) - 1:]
        return plain_text[:-ord(last_character)]  # Remove padding

    def create_keys(self):
        """
        Generate RSA public and private keys.
        Save the keys to files for reuse.
        """
        self.public_key, self.private_key = rsa.newkeys(1024)  # Generate RSA keys
        if not os.path.isfile(f"{os.getcwd()}/keys/public.pem"):
            with open(f"{os.getcwd()}/keys/public.pem", "wb") as f:
                f.write(self.public_key.save_pkcs1("PEM"))  # Save public key
        if not os.path.isfile(f"{os.getcwd()}/keys/private.pem"):
            with open(f"{os.getcwd()}/keys/private.pem", "wb") as f:
                f.write(self.private_key.save_pkcs1("PEM"))  # Save private key

    def load_keys(self):
        """
        Load RSA public and private keys from files.
        """
        with open(f"{os.getcwd()}/keys/public.pem", "rb") as f:
            self.public_key = rsa.PublicKey.load_pkcs1(f.read())
        with open(f"{os.getcwd()}/keys/private.pem", "rb") as f:
            self.private_key = rsa.PrivateKey.load_pkcs1(f.read())

    def send_rsa_key(self, sock, tid):
        """
        Send the RSA public key to a client.
        """
        key_to_send = self.public_key.save_pkcs1()  # Serialize public key
        key_len = struct.pack("!l", len(key_to_send))  # Pack key length
        sock.send(key_len + key_to_send)  # Send key length and serialized key

    def recv_shared_secret(self, sock, tid):
```

```
    """
    Receive and decrypt a shared secret from a client.
    """
    key_len_b = b""
    while len(key_len_b) < LEN_FIELD:  # Receive key length
        key_len_b += sock.recv(LEN_FIELD - len(key_len_b))
    key_len = int(struct.unpack("!l", key_len_b)[0])

    key_binary = b""
    while len(key_binary) < key_len:  # Receive the key
        key_binary += sock.recv(key_len - len(key_binary))
    return rsa.decrypt(key_binary, self.private_key)  # Decrypt the shared secret

def rsa_exchange(self, sock, tid):
    """
    Perform an RSA key exchange by sending the public key and receiving a shared secret.
    """
    self.send_rsa_key(sock, tid)  # Send the public key
    return self.recv_shared_secret(sock, tid)  # Receive and return the shared secret
```

```
# 2024 Â© Idan Hazay

from enum import Enum

class Errors(Enum):
    """
    Enumeration for error messages used throughout the application.
    """
    GENERAL = f"ERRR|001|General error"
    UNKNOWN = f"ERRR|002|Code not supported"
    LOGIN_DETAILS = f"ERRR|003|Please check your password and email/username and try again."
    USER_REGISTERED = f"ERRR|004|Username already registered"
    EMAIL_REGISTERED = f"ERRR|005|Email address already registered"
    EMAIL_NOT_REGISTERED = f"ERRR|006|Email is not registered"
    NOT_MATCHING_CODE = f"ERRR|007|Code not matching try again"
    CODE_EXPIRED = f"ERRR|008|Code has expired"
    NOT_VERIFIED = f"ERRR|009|User not verified"
    ALREADY_VERIFIED = f"ERRR|010|Already verified"
    FILE_UPLOAD = f"ERRR|011|File didnt upload correctly"
    NO_DELETE_PERMS = f"ERRR|012|Can't delete this user"
    INVALID_DIRECTORY = f"ERRR|013|Invalid directory"
    FILE_NOT_FOUND = f"ERRR|014|File not found"
    FILE_DOWNLOAD = f"ERRR|015|File didnt download correctly"
    FOLDER_EXISTS = f"ERRR|016|This folder already exists"
    EXISTS = f"ERRR|017|File/Folder with same name already exists"
    SAME_LEVEL = f"ERRR|018|Already at this subscription level"
    INVALID_LEVEL = f"ERRR|019|Invalid subscription level"
    MAX_STORAGE = f"ERRR|019|Max storage reached, try upgrading your subscription"
    SIZE_LIMIT = f"ERRR|020|File exceeded max size of"
    FILE_EXISTS = f"ERRR|021|A file with that name already exists"
    PREVIEW_SIZE = f"ERRR|022|File exceeds max preview size of 10 MB"
    NOT_LOGGED = f"ERRR|023|Can't upload files here"
    INVALID_COOKIE = f"ERRR|024|Cookie is invalid"
    EXPIRED_COOKIE = f"ERRR|025|Cookie is expired"
    NO_PERMS = f"ERRR|026|You don't have the permission to perform this action"
    USER_NOT_FOUND = f"ERRR|027|User with this username/email was not found"
    SELF_SHARE = f"ERRR|028|You cannot share with yourself"
    OWNER_SHARE = f"ERRR|029|You cannot share with owner"
    IN_USE = f"ERRR|030|File is currently in use"
    ALREADY_UPLOADING = f"ERRR|031|File have already started uploading"
    FILE_SIZE = f"ERRR|032|Invalid file size or seek location"

    EMPTY_FIELD = f"ERRR|101|Cannot have an empty field"
    INVALID_CHARS = f"ERRR|102|Invalid chars used"
    INVALID_EMAIL = f"ERRR|103|Invalid email address"
    INVALID_USERNAME = f"ERRR|104|Invalid username\nUsername has to be at least 4 long and contain only chars and numbers"
    PASSWORD_REQ = f"ERRR|105|Password does not meet requirements\nhas to be at least 8 long and contain at least 1 upper
case and number"
    PASSWORDS_MATCH = f"ERRR|106|Passwords do not match"
```

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

from PyQt6.QtCore import QThread, pyqtSignal
import time, uuid, traceback, os
from modules.config import *
from modules.limits import Limits

class FileSending:
    """Handles file upload management, including queuing and thread handling."""
    def __init__(self, window):
        self.window = window
        self.active_threads = []
        self.file_queue = []

    def send_files(self, cmd="FILS", file_id=None, resume_file_id=None, location_infile=0):
        """Starts a new file upload thread if none are active."""
        if len(self.active_threads) >= 1:
            return
        try:
            self.window.file_upload_progress.show()
        except:
            pass
        try:
            self.window.stop_button.setEnabled(True)
            self.window.stop_button.show()
        except:
            pass

        thread = FileSenderThread(cmd, file_id, resume_file_id, location_infile, self.window, self.file_queue)
        self.active_threads.append(thread)

        thread.finished.connect(thread.deleteLater)
        thread.finished.connect(lambda: self.active_threads.remove(thread))
        thread.finished.connect(self.window.finish_sending)

        thread.progress.connect(self.window.update_progress)
        thread.progress_reset.connect(self.window.reset_progress)  # Connect progress signal to progress bar
        thread.message.connect(self.window.set_message)
        thread.error.connect(self.window.set_error_message)

        thread.start()

    def resume_files_upload(self, id, progress):
        """Resumes file upload from the last known progress point."""
        uploading_files = self.window.json.get_files_uploading_data()
        for file_id, details in uploading_files.items():  # Iterate through stored uploading files
            if id == file_id:
                file_path = details.get("file_path")
                if not os.path.exists(file_path):
                    continue
                self.file_queue.extend([file_path])  # Re-add file to queue
                self.window.protocol.send_files(resume_file_id=file_id, location_infile=int(progress))
                break


class FileSenderThread(QThread):
    """Handles file upload operations in a separate thread."""
    finished = pyqtSignal()  # Signal when file sending is complete
    error = pyqtSignal(str)  # Signal for error messages
    progress = pyqtSignal(int)  # Signal for updating progress bar
    progress_reset = pyqtSignal(int)
    message = pyqtSignal(str)  # Signal for updating the status message

    def __init__(self, cmd, file_id, resume_file_id, location_infile, window, file_queue):
        super().__init__()
        self.files_uploaded = []
        self.cmd = cmd
        self.file_id = file_id
        self.resume_file_id = resume_file_id
        self.running = True
        self.location_infile = location_infile
        self.window = window
        self.file_queue = file_queue

    def run(self):
        """Runs the file upload process for each file in the queue."""
        try:
            for file_path in self.file_queue:
                start = time.time()
                bytes_sent = 0

                try:
                    self.window.stop_button.setEnabled(True)
                except:
                    pass
```

```
                file_name = self.file_id if self.file_id else file_path.split("/")[-1]  # Extract file name
                file_id = uuid.uuid4().hex
                self.window.uploading_file_id = file_id

                if self.resume_file_id is None:
                    print("start upload:", file_id)
                    start_string = f"{self.cmd}|{file_name}|{self.window.user['cwd']}|{os.path.getsize(file_path)}|
{file_id}"
                    self.window.protocol.send_data(start_string.encode())
                    self.window.json.update_json(True, file_id, file_path)
                else:
                    file_id = self.resume_file_id

                if not os.path.isfile(file_path):
                    self.error.emit("File path was not found")
                    return

                size = os.path.getsize(file_path)
                left = size % CHUNK_SIZE
                sent = self.location_infile
                self.progress.emit(sent)
                self.progress_reset.emit(size)
                self.message.emit(f"{file_name} is being uploaded")

                try:
                    with open(file_path, 'rb') as f:
                        f.seek(self.location_infile)  # Resume from last known position
                        for i in range((size - self.location_infile) // CHUNK_SIZE):
                            if not self.running:
                                break

                            location_infile = f.tell()
                            data = f.read(CHUNK_SIZE)

                            current_time = time.time()
                            elapsed_time = current_time - start

                            if elapsed_time >= 1.0:
                                start = current_time
                                bytes_sent = 0

                            self.window.protocol.send_data(f"FILD|{file_id}|{location_infile}|".encode() + data)
                            bytes_sent += len(data)
                            sent += CHUNK_SIZE

                            self.progress_reset.emit(size)
                            self.message.emit(f"{file_name} is being uploaded")
                            self.progress.emit(sent)  # Update progress bar

                            # Ensure upload speed limit
                            if bytes_sent >= (Limits(self.window.user["subscription_level"]).max_upload_speed - 1) *
1_000_000:
                                time_to_wait = 1.0 - elapsed_time
                                if time_to_wait > 0:
                                    time.sleep(time_to_wait)

                        if not self.running:
                            self.running = True
                            continue

                        location_infile = f.tell()
                        data = f.read(left)
                        if data != b"":
                            self.window.protocol.send_data(f"FILE|{file_id}|{location_infile}|".encode() + data)
                            self.progress_reset.emit(size)
                            self.message.emit(f"{file_name} is being uploaded")
                            self.progress.emit(sent)  # Final progress update

                except:
                    print(traceback.format_exc())
                    return

                finally:
                    self.window.json.update_json(True, file_id, file_path, remove=True)

            if self.file_id is not None:
                os.remove(file_path.split("/")[-1])  # Remove temp files after upload completion
            self.finished.emit()

        except:
            print(traceback.format_exc())
            print(type(self.file_queue))


class File:
    """Represents a file being downloaded or uploaded."""
    def __init__(self, window, save_location, id, size, is_view=False, file_name=None):
```

```
        self.save_location = save_location
        self.id = id
        self.size = size
        self.is_view = is_view
        self.file_name = file_name
        self.start_download()
        self.window = window

    def start_download(self):
        """Prepares a file for download by creating an empty placeholder."""
        if not os.path.exists(self.save_location):
            with open(self.save_location, 'wb') as f:
                f.write(b"\0")  # Create an empty file
                f.flush()

    def add_data(self, data, location_infile):
        """Writes received data to the file at the correct position."""
        try:
            self.window.file_upload_progress.show()
        except:
            pass

        self.window.update_progress(location_infile)
        self.window.reset_progress(self.size)
        self.window.set_message(f"File {self.file_name} is downloading")

        try:
            with open(self.save_location, 'r+b') as f:
                f.seek(location_infile)
                f.write(data)
                f.flush()

                self.window.json.update_json(False, self.id, self.save_location, remove=True)
                self.window.json.update_json(False, self.id, self.save_location, file=self, progress=location_infile)
        except:
            self.uploading = False

    def delete(self):
        """Deletes the downloaded file if it exists."""
        if os.path.exists(self.save_location):
            os.remove(self.save_location)
```

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

from PyQt6.QtWidgets import QApplication, QDialog, QVBoxLayout, QLabel, QTextEdit, QPushButton, QHBoxLayout
from PyQt6.QtGui import QPixmap, QIcon
from PyQt6.QtCore import Qt

import os
from docx import Document


class FileViewer:
    """Displays files (text, images, and documents) in a PyQt dialog."""
    def __init__(self, file_path, title):
        self.file_path = file_path
        self.title = title
        self.file_viewer_dialog()

    def open_in_native_app(self):
        """Opens the file with the system's default application."""
        try:
            os.startfile(self.file_path)
        except Exception as e:
            print(f"Error opening file in native app: {e}")

    def file_viewer_dialog(self):
        """Creates and displays a file viewer dialog based on file type."""
        app = QApplication.instance()  # Get existing QApplication instance
        if app is None:
            app = QApplication([])  # Create a new instance if needed

        file_extension = os.path.splitext(self.file_path)[1].lower()
        dialog = QDialog()
        dialog.setStyleSheet("font-size:15px;")
        layout = QVBoxLayout()
        dialog.resize(600, 400)

        # Set window icon if available
        icon_path = f"{os.path.dirname(os.path.dirname(os.path.abspath(__file__)))}/assets/icon.ico"
        if os.path.isfile(icon_path):
            dialog.setWindowIcon(QIcon(icon_path))

        dialog.setWindowTitle(self.title)

        close_button = QPushButton('Close', dialog)
        close_button.clicked.connect(dialog.close)
        layout.addWidget(close_button)

        content_widget = None

        # Display images
        if file_extension in ['.jpg', '.jpeg', '.png', '.bmp', '.gif']:
            content_widget = QLabel(dialog)
            pixmap = QPixmap(self.file_path)
            content_widget.setPixmap(pixmap.scaled(600, 800, Qt.AspectRatioMode.KeepAspectRatio))  # Maintain aspect ratio
            layout.insertWidget(0, content_widget)

        # Display .docx files
        elif file_extension == '.docx':
            content_widget = QTextEdit(dialog)
            content_widget.setReadOnly(True)
            try:
                doc = Document(self.file_path)
                full_text = '\n'.join([paragraph.text for paragraph in doc.paragraphs])  # Extract text from all
paragraphs
                content_widget.setPlainText(full_text)
            except Exception as e:
                content_widget.setPlainText(f"Error reading document: {str(e)}")
            layout.insertWidget(0, content_widget)

        else:
            # Attempt to open unsupported file types as plain text
            try:
                with open(self.file_path, 'r', encoding='utf-8') as f:
                    content = f.read()
                content_widget = QTextEdit(dialog)
                content_widget.setReadOnly(True)
                content_widget.setPlainText(content)
                layout.insertWidget(0, content_widget)

            except Exception as e:
                # If opening as text fails, display a fallback message
                content_widget = QLabel(dialog)
                content_widget.setText(f"Cannot open {os.path.basename(self.file_path)[5:]} in file viewer.\nTry opening
it in its default app.")
                content_widget.setStyleSheet("font-size: 20px")
                content_widget.setAlignment(Qt.AlignmentFlag.AlignCenter)
```

```
        layout.insertWidget(0, content_widget)

        # Add a button to open in the default system application
        open_native_button = QPushButton(f"Open {os.path.splitext(self.file_path)[1]} in default app", dialog)
        open_native_button.clicked.connect(self.open_in_native_app)
        layout.addWidget(open_native_button)

dialog.setLayout(layout)
dialog.exec()
```

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

from PyQt6 import QtWidgets, uic
from PyQt6.QtWidgets import QWidget, QDialog, QApplication, QLabel, QVBoxLayout, QPushButton, QCheckBox, QGroupBox,
QFileDialog, QLineEdit, QGridLayout, QScrollArea, QHBoxLayout, QSpacerItem, QSizePolicy, QMenu
from PyQt6.QtGui import QIcon, QDragEnterEvent, QDropEvent, QMoveEvent, QResizeEvent, QContextMenuEvent
from PyQt6.QtCore import QSize, Qt

import os, time

from modules.config import *
from modules.limits import Limits

from modules import helper, protocol, file_send, dialogs, file_viewer

class MainWindow(QtWidgets.QMainWindow):
    """Main application window handling UI, user interactions, and event management."""
    def __init__(self, app, network):
        super().__init__()
        self.app = app
        self.network = network
        self.protocol = protocol.Protocol(self.network, self)
        self.file_sending = file_send.FileSending(self)

        self.window_geometry = WINDOW_GEOMERTY
        self.save_sizes()
        self.setGeometry(self.window_geometry)

        # Set initial size and position
        self.original_width, self.original_height = self.width(), self.height()
        s_width, s_height = app.primaryScreen().geometry().width(), app.primaryScreen().geometry().height()
        self.resize(s_width * 3 // 4, s_height * 2 // 3)
        self.move(s_width // 8, s_height // 6)

        # Enable fast rendering attributes
        self.setAttribute(Qt.WidgetAttribute.WA_OpaquePaintEvent)
        self.setAttribute(Qt.WidgetAttribute.WA_PaintOnScreen, True)

        # Initialize UI state variables
        self.scroll_progress = 0
        self.current_files_amount = ITEMS_TO_LOAD
        self.last_load = time.time()
        self.scroll_size = SCROLL_SIZE

        self.user = {"email": "guest", "username": "guest", "subscription_level": 0, "cwd": "", "parent_cwd": "",
"cwd_name": "", "admin_level": 0}
        self.json = helper.JsonHandle()

        self.search_filter = None
        self.share, self.deleted = False, False
        self.sort, self.sort_direction = "Name", True
        self.remember = False

        self.files, self.directories = [], []
        self.files_downloading = {}
        self.currently_selected = []
        self.uploading_file_id = ""
        self.used_storage = 0
        self.items_amount = 0

        self.original_sizes = {}
        self.scroll = None

        self.start()

    def start(self):
        """Applies initial styling and sets the application icon."""
        try:
            with open(f"{os.getcwd()}/gui/css/style.css", 'r') as f:
                self.app.setStyleSheet(f.read())
        except:
            print(traceback.format_exc())

        if os.path.isfile(f"{os.getcwd()}/assets/icon.ico"):
            self.setWindowIcon(QIcon(f"{os.getcwd()}/assets/icon.ico"))

    def keyPressEvent(self, event):
        """Handles keypress events for shortcuts and file operations."""
        if event.key() == Qt.Key.Key_Delete and self.currently_selected:
            self.protocol.delete()
        elif event.key() == Qt.Key.Key_R and event.modifiers() & Qt.KeyboardModifier.ControlModifier:
            if self.user["username"] != "guest":
                self.user_page()
        elif event.key() == Qt.Key.Key_A and event.modifiers() & Qt.KeyboardModifier.ControlModifier:
            if self.scroll:
                for button in self.scroll.widget().findChildren(FileButton):
```

```
                if button.id and button not in self.currently_selected:
                    self.select_item(button)
        elif event.key() == Qt.Key.Key_S and event.modifiers() & Qt.KeyboardModifier.ControlModifier:
            if self.user["username"] != "guest":
                self.protocol.search()
        super().keyPressEvent(event)

    def save_sizes(self):
        """Stores the initial sizes and font sizes of all widgets for dynamic resizing."""
        for widget in self.findChildren(QWidget):
            font_size = widget.font().pointSize()
            self.original_sizes[widget] = {
                'geometry': widget.geometry(),
                'font_size': font_size
            }

    def moveEvent(self, event: QMoveEvent):
        """Updates the window geometry when moved."""
        self.window_geometry = self.geometry()

    def resizeEvent(self, event):
        """Dynamically resizes widgets based on the new window size."""
        new_width, new_height = self.width(), self.height()
        width_ratio, height_ratio = new_width / self.original_width, new_height / self.original_height

        for widget in self.findChildren(QWidget):
            if widget in self.original_sizes:
                original_geometry = self.original_sizes[widget]['geometry']
                original_font_size = self.original_sizes[widget]['font_size']

                new_x = int(original_geometry.x() * width_ratio) if width_ratio != 1 else original_geometry.x()
                new_width = int(original_geometry.width() * width_ratio) if width_ratio != 1 else
original_geometry.width()

                new_y = int(original_geometry.y() * height_ratio) if height_ratio != 1 else original_geometry.y()
                new_height = int(original_geometry.height() * height_ratio) if height_ratio != 1 else
original_geometry.height()

                self.window_geometry = self.geometry()
                widget.setGeometry(new_x, new_y, new_width, new_height)
                widget.updateGeometry()

                new_font_size = max(int(original_font_size * (width_ratio + height_ratio) / 2), 8)
                font = widget.font()
                font.setPointSize(new_font_size)
                widget.setFont(font)

                if isinstance(widget, QPushButton):
                    icon = widget.icon()
                    if not icon.isNull():
                        base = 60 if widget.text() == "" else 16
                        new_icon_size = int(base * (width_ratio + height_ratio) / 2)
                        widget.setIconSize(QSize(new_icon_size, new_icon_size))

        # Adjust scroll area size and file buttons
        try:
            if self.scroll:
                for button in self.scroll.widget().findChildren(FileButton):
                    for i in range(button.layout().count()):
                        label = button.layout().itemAt(i).widget()
                        if isinstance(label, QLabel):
                            font = label.font()
                            font.setPointSize(max(int(9 * (width_ratio + height_ratio) / 2), 8))
                            label.setFont(font)
                    button.setMinimumHeight(int(30 * height_ratio))
                self.scroll_size = [int(850 * width_ratio), int(340 * height_ratio)]
                self.scroll.setFixedSize(self.scroll_size[0], self.scroll_size[1])
        except:
            pass


    def main_page(self):
        """Loads the main page UI."""
        try:
            temp = self.window_geometry
            ui_path = f"{os.getcwd()}/gui/ui/main.ui"
            helper.update_ui_size(ui_path, self.window_geometry.width(), self.window_geometry.height())
            uic.loadUi(ui_path, self)

            self.save_sizes()

            self.signup_button.clicked.connect(self.signup_page)
            self.signup_button.setIcon(QIcon(ASSETS_PATH + "\\new_account.svg"))

            self.login_button.clicked.connect(self.login_page)
            self.login_button.setIcon(QIcon(ASSETS_PATH + "\\login.svg"))
```

```
            self.exit_button.clicked.connect(self.protocol.exit_program)
            self.exit_button.setIcon(QIcon(ASSETS_PATH + "\\exit.svg"))

            self.setGeometry(temp)
            self.force_update_window()
        except:
            print(traceback.format_exc())

    def signup_page(self):
        """Loads the signup page UI."""
        try:
            temp = self.window_geometry
            ui_path = f"{os.getcwd()}/gui/ui/signup.ui"
            helper.update_ui_size(ui_path, self.window_geometry.width(), self.window_geometry.height())
            uic.loadUi(ui_path, self)

            self.save_sizes()

            self.password.setEchoMode(QLineEdit.EchoMode.Password)
            self.confirm_password.setEchoMode(QLineEdit.EchoMode.Password)

            self.password_toggle.clicked.connect(lambda: self.toggle_password(self.password))
            self.confirm_password_toggle.clicked.connect(lambda: self.toggle_password(self.confirm_password))

            self.signup_button.clicked.connect(lambda: self.protocol.signup(
                self.email.text(), self.username.text(), self.password.text(), self.confirm_password.text()))
            self.signup_button.setShortcut("Return")
            self.signup_button.setIcon(QIcon(ASSETS_PATH + "\\new_account.svg"))

            self.login_button.clicked.connect(self.login_page)
            self.login_button.setStyleSheet("background-color:transparent;color:royalblue;text-decoration:
underline;border:none;")

            self.back_button.clicked.connect(self.main_page)
            self.back_button.setIcon(QIcon(ASSETS_PATH + "\\back.svg"))

            self.setGeometry(temp)
            self.force_update_window()
        except:
            print(traceback.format_exc())

    def login_page(self):
        """Loads the login page UI."""
        try:
            temp = self.window_geometry
            ui_path = f"{os.getcwd()}/gui/ui/login.ui"
            helper.update_ui_size(ui_path, self.window_geometry.width(), self.window_geometry.height())
            uic.loadUi(ui_path, self)

            self.save_sizes()

            self.password.setEchoMode(QLineEdit.EchoMode.Password)
            self.password_toggle.clicked.connect(lambda: self.toggle_password(self.password))

            self.forgot_password_button.clicked.connect(self.forgot_password)
            self.forgot_password_button.setStyleSheet("background-color:transparent;color:royalblue;text-decoration:
underline;border:none;")

            self.signup_button.clicked.connect(self.signup_page)
            self.signup_button.setStyleSheet("background-color:transparent;color:royalblue;text-decoration:
underline;border:none;")

            self.login_button.clicked.connect(lambda: self.protocol.login(
                self.credi.text(), self.password.text(), self.remember_check.isChecked()))
            self.login_button.setShortcut("Return")
            self.login_button.setIcon(QIcon(ASSETS_PATH + "\\login.svg"))

            self.back_button.clicked.connect(self.main_page)
            self.back_button.setIcon(QIcon(ASSETS_PATH + "\\back.svg"))

            self.setGeometry(temp)
            self.force_update_window()
        except:
            print(traceback.format_exc())

    def forgot_password(self):
        """Loads the password recovery page UI."""
        try:
            temp = self.window_geometry
            ui_path = f"{os.getcwd()}/gui/ui/forgot_password.ui"
            helper.update_ui_size(ui_path, self.window_geometry.width(), self.window_geometry.height())
            uic.loadUi(ui_path, self)

            self.save_sizes()

            self.send_code_button.clicked.connect(lambda: self.protocol.reset_password(self.email.text()))
            self.send_code_button.setShortcut("Return")
```

```
        self.send_code_button.setIcon(QIcon(ASSETS_PATH + "\\send.svg"))

        self.back_button.clicked.connect(self.login_page)
        self.back_button.setIcon(QIcon(ASSETS_PATH + "\\back.svg"))

        self.setGeometry(temp)
        self.force_update_window()
    except:
        print(traceback.format_exc())

def verification_page(self, email):
    """Loads the account verification page UI."""
    try:
        temp = self.window_geometry
        ui_path = f"{os.getcwd()}/gui/ui/verification.ui"
        helper.update_ui_size(ui_path, self.window_geometry.width(), self.window_geometry.height())
        uic.loadUi(ui_path, self)

        self.save_sizes()

        self.verify_button.clicked.connect(lambda: self.protocol.verify(email, self.code.text()))
        self.verify_button.setShortcut("Return")
        self.verify_button.setIcon(QIcon(ASSETS_PATH + "\\verify.svg"))

        self.send_again_button.clicked.connect(lambda: self.protocol.send_verification(email))
        self.send_again_button.setIcon(QIcon(ASSETS_PATH + "\\again.svg"))

        self.back_button.clicked.connect(self.main_page)
        self.back_button.setIcon(QIcon(ASSETS_PATH + "\\back.svg"))

        self.setGeometry(temp)
        self.force_update_window()
    except:
        print(traceback.format_exc())

def send_verification_page(self):
    """Loads the send verification email page UI."""
    try:
        temp = self.window_geometry
        ui_path = f"{os.getcwd()}/gui/ui/send_verification.ui"
        helper.update_ui_size(ui_path, self.window_geometry.width(), self.window_geometry.height())
        uic.loadUi(ui_path, self)

        self.save_sizes()

        self.send_code_button.clicked.connect(lambda: self.protocol.send_verification(self.email.text()))
        self.send_code_button.setShortcut("Return")
        self.send_code_button.setIcon(QIcon(ASSETS_PATH + "\\send.svg"))

        self.back_button.clicked.connect(self.main_page)
        self.back_button.setIcon(QIcon(ASSETS_PATH + "\\back.svg"))

        self.setGeometry(temp)
        self.force_update_window()
    except:
        print(traceback.format_exc())

def user_page(self):
    """Loads and updates the user page UI."""
    self.update_user_page()
    self.run_user_page()

def update_user_page(self):
    """Fetches updated file and directory listings for the user page."""
    self.files, self.directories = None, None
    self.protocol.get_used_storage()

    if self.user["cwd"] == "" and self.deleted:
        self.protocol.get_deleted_files(self.search_filter)
        self.protocol.get_deleted_directories(self.search_filter)
    elif self.user["cwd"] == "" and self.share:
        self.protocol.get_cwd_shared_files(self.search_filter)
        self.protocol.get_cwd_shared_directories(self.search_filter)
    else:
        self.protocol.get_cwd_files(self.search_filter)
        self.protocol.get_cwd_directories(self.search_filter)


def run_user_page(self):
    """Loads and sets up the user page UI."""
    try:
        temp = self.window_geometry

        # Load user management and file navigation UI
        ui_path = f"{os.getcwd()}/gui/ui/account_managment.ui"
        helper.update_ui_size(ui_path, self.window_geometry.width(), self.window_geometry.height())
        uic.loadUi(ui_path, self)
```

```
    ui_path = f"{os.getcwd()}/gui/ui/user.ui"
    helper.update_ui_size(ui_path, self.window_geometry.width(), self.window_geometry.height())
    uic.loadUi(ui_path, self)

    # Enable or disable file dropping based on user mode
    self.setAcceptDrops(not (self.share or self.deleted))
    if self.share:
        self.sort_widget.addItem(" Owner")

    self.set_cwd()

    # Hide progress indicators if no active uploads
    if not self.file_sending.active_threads:
        self.file_upload_progress.hide()
        self.stop_button.hide()

    self.currently_selected = []
    self.main_text.setText(f"Welcome {self.user['username']}")

    # Set storage limit
    self.storage_remaining.setMaximum(Limits(self.user["subscription_level"]).max_storage)
    self.set_used_storage()

    self.sort_widget.currentIndexChanged.connect(lambda: self.change_sort(self.sort.currentText()[1:]))

    # Configure UI buttons
    self.search_button.setIcon(QIcon(ASSETS_PATH + "\\search.svg"))
    self.search_button.setText(f" Search Filter: {self.search_filter}")
    self.search_button.clicked.connect(self.protocol.search)
    self.search_button.setStyleSheet("background-color:transparent;border:none;")

    self.refresh.setIcon(QIcon(ASSETS_PATH + "\\refresh.svg"))
    self.refresh.setText(" ")
    self.refresh.clicked.connect(self.user_page)
    self.refresh.setStyleSheet("background-color:transparent;border:none;")

    self.shared_button.clicked.connect(self.protocol.change_share)
    self.shared_button.setIcon(QIcon(ASSETS_PATH + "\\share.svg"))

    self.recently_deleted_button.clicked.connect(self.protocol.change_deleted)
    self.recently_deleted_button.setIcon(QIcon(ASSETS_PATH + "\\delete.svg"))

    self.user_button.clicked.connect(lambda: self.manage_account())
    self.logout_button.clicked.connect(self.protocol.logout)
    self.logout_button.setIcon(QIcon(ASSETS_PATH + "\\logout.svg"))
    self.upload_button.setIcon(QIcon(ASSETS_PATH + "\\upload.svg"))

    # Adjust upload button behavior based on view mode
    if self.deleted:
        try:
            self.upload_button.setIcon(QIcon(USER_ICON))
        except:
            pass
        self.upload_button.setText(" Your files")
        self.upload_button.clicked.connect(self.protocol.change_deleted)
        self.recently_deleted_button.hide()
        self.shared_button.hide()

    elif self.share:
        try:
            self.upload_button.setIcon(QIcon(USER_ICON))
        except:
            pass
        self.upload_button.setText(" Your files")
        self.upload_button.clicked.connect(self.protocol.change_share)
        self.shared_button.hide()
        self.recently_deleted_button.hide()

    else:
        self.upload_button.clicked.connect(lambda: self.file_dialog())

    self.user_button.setIconSize(QSize(self.user_button.size().width(), self.user_button.size().height()))
    self.user_button.setStyleSheet("padding:0px;border-radius:5px;border:none;background-color:transparent")

    try:
        self.user_button.setIcon(QIcon(USER_ICON))
    except:
        pass

    self.stop_button.clicked.connect(self.stop_upload)
    self.stop_button.setIcon(QIcon(ASSETS_PATH + "\\stop.svg"))

    self.setGeometry(temp)
    self.force_update_window()

except:
```

```
            print(traceback.format_exc())

    def draw_cwd(self):
        """Creates the file and directory listing in the user interface."""
        try:
            central_widget = self.centralWidget()
            outer_layout = QVBoxLayout()
            outer_layout.addStretch(1)

            # Create a scrollable area for files and directories
            scroll = QScrollArea()
            self.scroll = scroll
            scroll.setWidgetResizable(True)
            scroll.setVerticalScrollBarPolicy(Qt.ScrollBarPolicy.ScrollBarAlwaysOn)

            scroll_container_widget = QWidget()
            scroll_layout = QGridLayout()
            scroll_layout.setSpacing(5)

            # Add column headers
            if self.deleted:
                button = FileButton(self, ["File Name", "Deleted In", "Size"])
            elif self.share:
                button = FileButton(self, ["File Name", "Last Change", "Size", "Owner"])
            else:
                button = FileButton(self, ["File Name", "Last Change", "Size"])

            button.setStyleSheet("background-color:#001122;border-radius: 3px;border:1px solid darkgrey;")
            scroll_layout.addWidget(button)

            # Populate file entries
            for file in self.files:
                file = file.split("~")
                file_name, date, size, file_id = file[0], file[1][:-7], helper.format_file_size(int(file[2])), file[3]
                perms = file[5:]

                if self.share:
                    button = FileButton(self, f" {file_name} | {date} | {size} | {file[4]}".split("|"), file_id,
shared_by=file[4], perms=perms, size=int(file[2]), name=file_name)
                else:
                    button = FileButton(self, f" {file_name} | {date} | {size}".split("|"), file_id, size=int(file[2]),
name=file_name)

                button.clicked.connect(lambda checked, btn=button: self.select_item(btn))
                scroll_layout.addWidget(button)

            # Populate directory entries
            for directory in self.directories:
                directory = directory.split("~")
                dir_name, dir_id, last_change, size = directory[0], directory[1], directory[2][:-7],
helper.format_file_size(int(directory[3]))
                perms = directory[5:]

                if self.share:
                    button = FileButton(self, f" {dir_name} | {last_change} | {size} | {directory[4]}".split("|"), dir_id,
is_folder=True, shared_by=directory[2], perms=perms, size=int(directory[3]), name=dir_name)
                else:
                    button = FileButton(self, f" {dir_name} | {last_change} | {size}".split("|"), dir_id, is_folder=True,
size=int(directory[3]), name=dir_name)

                button.clicked.connect(lambda checked, btn=button: self.select_item(btn))
                scroll_layout.addWidget(button)

            # Handle empty directory
            if not self.directories and not self.files:
                button = FileButton(self, ["No files or folders in this directory"])
                button.setStyleSheet("background-color:red;border-radius: 3px;border:1px solid darkgrey;")
                scroll_layout.addWidget(button)

            # Add "Back" button if not at root
            if self.user["cwd"]:
                button = FileButton(self, ["Back"])
                button.clicked.connect(lambda: self.protocol.move_dir(self.user["parent_cwd"]))
                scroll_layout.addWidget(button)

            # Finalize scroll area
            scroll_container_widget.setLayout(scroll_layout)
            scroll.setWidget(scroll_container_widget)
            scroll.setFixedSize(850, 340)

            # Add scroll area to the layout
            spacer = QSpacerItem(20, 20, QSizePolicy.Policy.Minimum, QSizePolicy.Policy.Expanding)
            outer_layout.addItem(spacer)

            center_layout = QHBoxLayout()
            center_layout.addStretch(1)
            center_layout.addWidget(scroll)
```

```
            center_layout.addStretch(1)

            outer_layout.addLayout(center_layout)
            outer_layout.addStretch(1)
            central_widget.setLayout(outer_layout)

        except:
            print(traceback.format_exc())


    def scroll_changed(self, value):
        """Handles scroll event to dynamically load more files if near the bottom."""
        self.scroll_progress = value
        total_scroll_height = self.scroll.verticalScrollBar().maximum()
        if total_scroll_height == 0: return
        if self.scroll_progress / total_scroll_height > 0.95 and len(self.directories) + len(self.files) <
int(self.items_amount):
            self.current_files_amount += ITEMS_TO_LOAD
            self.user_page()

    def manage_account(self):
        """Loads the account management page."""
        try:
            temp = self.window_geometry
            ui_path = f"{os.getcwd()}/gui/ui/account_managment.ui"
            helper.update_ui_size(ui_path, self.window_geometry.width(), self.window_geometry.height())
            uic.loadUi(ui_path, self)
            self.save_sizes()

            self.forgot_password_button.clicked.connect(lambda: self.protocol.reset_password(self.user["email"]))
            self.forgot_password_button.setIcon(QIcon(ASSETS_PATH + "\\key.svg"))

            self.delete_account_button.clicked.connect(lambda: self.protocol.delete_user(self.user["email"]))
            self.delete_account_button.setIcon(QIcon(ASSETS_PATH + "\\delete.svg"))

            self.upload_icon_button.clicked.connect(lambda: self.protocol.upload_icon())
            self.upload_icon_button.setIcon(QIcon(ASSETS_PATH + "\\profile.svg"))

            self.subscriptions_button.clicked.connect(self.subscriptions_page)
            self.subscriptions_button.setIcon(QIcon(ASSETS_PATH + "\\upgrade.svg"))

            self.change_username_button.clicked.connect(self.protocol.change_username)
            self.change_username_button.setIcon(QIcon(ASSETS_PATH + "\\change_user.svg"))

            if self.user["admin_level"] > 0:
                self.admin_button.setIcon(QIcon(ASSETS_PATH + "\\admin.svg"))
                self.admin_button.clicked.connect(self.admin_page)
                self.admin_button.setStyleSheet("background-color:transparent;border:none;")
            else:
                self.admin_button.hide()

            self.back_button.clicked.connect(self.user_page)
            self.back_button.setIcon(QIcon(ASSETS_PATH + "\\back.svg"))

            self.setGeometry(temp)
            self.force_update_window()
        except:
            print(traceback.format_exc())

    def admin_page(self):
        """Loads the admin page."""
        try:
            if self.user["admin_level"] <= 0:
                self.user_page()
                return

            temp = self.window_geometry
            ui_path = f"{os.getcwd()}/gui/ui/admin.ui"
            helper.update_ui_size(ui_path, self.window_geometry.width(), self.window_geometry.height())
            uic.loadUi(ui_path, self)
            self.save_sizes()

            self.protocol.admin_data()

            self.back_button.clicked.connect(self.manage_account)
            self.back_button.setIcon(QIcon(ASSETS_PATH + "\\back.svg"))

            self.setGeometry(temp)
            self.force_update_window()
        except:
            print(traceback.format_exc())

    def subscriptions_page(self):
        """Loads the subscription management page."""
        try:
            temp = self.window_geometry
            ui_path = f"{os.getcwd()}/gui/ui/subscription.ui"
```

```
            helper.update_ui_size(ui_path, self.window_geometry.width(), self.window_geometry.height())
            uic.loadUi(ui_path, self)
            self.save_sizes()

            self.protocol.get_used_storage()

            self.back_button.clicked.connect(self.manage_account)
            self.back_button.setIcon(QIcon(ASSETS_PATH + "\\back.svg"))

            self.free_button.clicked.connect(lambda: self.protocol.subscribe(0))
            self.basic_button.clicked.connect(lambda: self.protocol.subscribe(1))
            self.premium_button.clicked.connect(lambda: self.protocol.subscribe(2))
            self.professional_button.clicked.connect(lambda: self.protocol.subscribe(3))

            # Disable and highlight the currently selected subscription level
            sub_buttons = {
                "0": self.free_button,
                "1": self.basic_button,
                "2": self.premium_button,
                "3": self.professional_button
            }
            if self.user["subscription_level"] in sub_buttons:
                sub_buttons[self.user["subscription_level"]].setDisabled(True)
                sub_buttons[self.user["subscription_level"]].setText("Selected")
                sub_buttons[self.user["subscription_level"]].setStyleSheet("background-color:dimgrey")

            self.storage_remaining.setMaximum(Limits(self.user["subscription_level"]).max_storage)
            self.set_used_storage()

            self.setGeometry(temp)
            self.force_update_window()
        except:
            print(traceback.format_exc())

    def recovery(self, email):
        """Loads the password recovery page."""
        try:
            temp = self.window_geometry
            ui_path = f"{os.getcwd()}/gui/ui/recovery.ui"
            helper.update_ui_size(ui_path, self.window_geometry.width(), self.window_geometry.height())
            uic.loadUi(ui_path, self)
            self.save_sizes()

            self.password.setEchoMode(QLineEdit.EchoMode.Password)
            self.confirm_password.setEchoMode(QLineEdit.EchoMode.Password)

            self.password_toggle.clicked.connect(lambda: self.toggle_password(self.password))
            self.confirm_password_toggle.clicked.connect(lambda: self.toggle_password(self.confirm_password))

            self.reset_button.clicked.connect(lambda: self.protocol.password_recovery(email, self.code.text(),
self.password.text(), self.confirm_password.text()))
            self.reset_button.setShortcut("Return")
            self.reset_button.setIcon(QIcon(ASSETS_PATH + "\\reset.svg"))

            self.send_again_button.clicked.connect(lambda: self.protocol.reset_password(email))
            self.send_again_button.setIcon(QIcon(ASSETS_PATH + "\\again.svg"))

            self.back_button.clicked.connect(self.manage_account)
            self.back_button.setIcon(QIcon(ASSETS_PATH + "\\back.svg"))

            self.setGeometry(temp)
            self.force_update_window()
        except:
            print(traceback.format_exc())

    def not_connected_page(self, connect=True):
        """Loads the not connected page and attempts reconnection."""
        try:
            temp = self.window_geometry
            ui_path = f"{os.getcwd()}/gui/ui/not_connected.ui"
            helper.update_ui_size(ui_path, self.window_geometry.width(), self.window_geometry.height())
            uic.loadUi(ui_path, self)
            self.save_sizes()

            self.ip.setText(self.protocol.ip)
            self.port.setText(str(self.protocol.port))

            self.connect_button.clicked.connect(lambda: self.protocol.connect_server(self.ip.text(), self.port.text(),
loop=True))
            self.connect_button.setShortcut("Return")
            self.connect_button.setIcon(QIcon(ASSETS_PATH + "\\connect.svg"))

            self.exit_button.clicked.connect(helper.force_exit)
            self.exit_button.setIcon(QIcon(ASSETS_PATH + "\\exit.svg"))

            self.setGeometry(temp)
            self.force_update_window()
```

```
            if connect:
                self.protocol.connect_server(loop=True)

        except:
            print(traceback.format_exc())

    def select_item(self, btn):
        """Handles selection of files and folders."""
        item_id = btn.id
        item_name = btn.name

        if btn in self.currently_selected and len(self.currently_selected) == 1:
            if btn.is_folder:
                self.protocol.move_dir(item_id)
                self.reset_selected()
            else:
                self.protocol.view_file(item_id, item_name, btn.file_size)
        elif helper.control_pressed() and btn not in self.currently_selected:
            self.currently_selected.append(btn)
        elif helper.control_pressed() and btn in self.currently_selected:
            self.currently_selected.remove(btn)
        else:
            self.reset_selected()
            self.currently_selected = [btn]

        # Update UI styling for selection
        for label in btn.lables:
            label.setObjectName("selected" if btn in self.currently_selected else ("folder-label" if btn.is_folder else
"file-label"))

        # Refresh UI stylesheet
        current_stylesheet = self.app.styleSheet()
        self.app.setStyleSheet("")
        self.app.setStyleSheet(current_stylesheet)

        self.force_update_window()



    def finish_sending(self):
        """Clears the file queue and hides upload-related UI elements."""
        self.file_sending.file_queue = []
        try:
            self.stop_button.setEnabled(False)
            self.stop_button.hide()
        except:
            pass
        try:
            self.file_upload_progress.hide()
        except:
            pass

    def update_progress(self, value):
        """Updates the file upload progress bar."""
        try:
            self.file_upload_progress.show()
        except:
            pass
        try:
            self.stop_button.setEnabled(True)
            self.stop_button.show()
        except:
            pass
        try:
            self.file_upload_progress.setValue(value)
        except:
            pass

    def reset_progress(self, value):
        """Resets the file upload progress bar."""
        try:
            self.file_upload_progress.show()
        except:
            pass
        try:
            self.stop_button.setEnabled(True)
            self.stop_button.show()
        except:
            pass
        try:
            self.file_upload_progress.setMaximum(value)
        except:
            pass

    def reset_selected(self):
        """Deselects all selected items."""
```

```
        for btn in self.currently_selected:
            for label in btn.lables:
                try:
                    label.setObjectName("folder-label" if btn.is_folder else "file-label")
                except RuntimeError:
                    if label in self.currently_selected:
                        self.currently_selected.remove(label)
        self.currently_selected = []

    def confirm_account_deletion(self, email):
        """Prompts the user to confirm account deletion."""
        confirm_email = dialogs.new_name_dialog("Delete Account", "Enter account email:")
        if email == confirm_email:
            return True
        else:
            self.set_error_message("Entered email does not match account email")

    def activate_file_view(self, file_id):
        """Opens the file viewer and checks for modifications."""
        save_path = self.files_downloading[file_id].save_location
        file_hash = helper.compute_file_md5(save_path)
        file_viewer.FileViewer(save_path, "File Viewer")

        if file_hash != helper.compute_file_md5(save_path):
            save = dialogs.show_confirmation_dialog("Do you want to save changes?")
            if save:
                self.file_sending.file_queue.append(save_path)
                self.file_sending.send_files("UPFL", file_id)
            else:
                os.remove(save_path)
        else:
            os.remove(save_path)

    def toggle_password(self, text):
        """Toggles password visibility in password fields."""
        text.setEchoMode(QLineEdit.EchoMode.Password if text.echoMode() == QLineEdit.EchoMode.Normal else
QLineEdit.EchoMode.Normal)

    def file_dialog(self):
        """Opens a file dialog to select files for uploading."""
        try:
            file_paths, _ = QFileDialog.getOpenFileNames(self, "Open File", "", "All Files (*);;Text Files (*.txt)")
            if file_paths:
                self.file_sending.file_queue.extend(file_paths)
                self.file_sending.send_files()
        except:
            print(traceback.format_exc())

    def dragEnterEvent(self, event: QDragEnterEvent):
        """Handles drag enter event to allow file dropping."""
        if event.mimeData().hasUrls():
            event.acceptProposedAction()

    def dropEvent(self, event: QDropEvent):
        """Handles file drop event and queues files for upload."""
        if event.mimeData().hasUrls():
            file_paths = [url.toLocalFile() for url in event.mimeData().urls()]
            self.file_sending.file_queue.extend(file_paths)
            self.file_sending.send_files()

    def change_sort(self, new_sort):
        """Changes the sorting method and reloads the file list."""
        if self.sort == new_sort:
            self.sort_direction = not self.sort_direction
        self.sort = new_sort
        self.user_page()

    def set_error_message(self, msg):
        """Displays an error message in the UI."""
        try:
            if hasattr(self, "message"):
                self.message.setStyleSheet("color: red;")
                self.message.setText(msg)
        except:
            pass

    def set_message(self, msg):
        """Displays a success message in the UI."""
        try:
            if hasattr(self, "message"):
                self.message.setStyleSheet("color: lightgreen;")
                self.message.setText(msg)
        except:
            pass

    def set_cwd(self):
        """Updates the displayed current working directory path."""
```

```
        if hasattr(self, "cwd"):
            self.cwd.setStyleSheet("color: yellow;")
            if self.share:
                self.cwd.setText(f"Shared > {" > ".join(self.user['cwd_name'].split('\\'))}"[:-3])
            elif self.deleted:
                self.cwd.setText(f"Deleted > {" > ".join(self.user['cwd_name'].split('\\'))}"[:-3])
            else:
                self.cwd.setText(f"{self.user['username']} > {" > ".join(self.user['cwd_name'].split('\\'))}"[:-3])

    def set_used_storage(self):
        """Updates the displayed storage usage."""
        self.storage_remaining.setValue(int(self.used_storage))
        self.storage_label.setText(f"Storage used ({helper.format_file_size(self.used_storage * 1_000_000)} /
{Limits(self.user['subscription_level']).max_storage // 1000} GB):")

    def stop_upload(self):
        """Stops the current file upload."""
        self.stop_button.setEnabled(False)
        if self.file_sending.active_threads:
            self.file_sending.active_threads[0].running = False
        self.protocol.send_data(b"STOP|" + self.uploading_file_id.encode())

    def force_update_window(self):
        """Forces a UI update to apply changes."""
        size = self.size()
        resize_event = QResizeEvent(size, size)
        self.resizeEvent(resize_event)

    def update_current_files(self):
        """Refreshes the file list UI based on the current sorting method."""
        self.sort_widget.currentIndexChanged.disconnect()

        sort_map = {"Name": 0, "Date": 1, "Type": 2, "Size": 3, "Owner": 4}
        if self.sort in sort_map:
            self.sort_widget.setCurrentIndex(sort_map[self.sort])

        self.save_sizes()
        self.draw_cwd()
        self.sort_widget.currentIndexChanged.connect(lambda: self.change_sort(self.sort_widget.currentText()[1:]))
        self.scroll.verticalScrollBar().setMaximum(self.scroll_progress)
        self.scroll.verticalScrollBar().setValue(self.scroll_progress)
        self.scroll.verticalScrollBar().valueChanged.connect(self.scroll_changed)


    def share_file(self, file_id, user_cred, file_name, read="False", write="False", delete="False", rename="False",
download="False", share="False"):
        """Displays a dialog to set file sharing permissions."""
        temp_app = QApplication.instance()
        if temp_app is None:
            temp_app = QApplication([])

        dialog = QDialog()
        dialog.setWindowTitle("File Share Options")
        dialog.setStyleSheet("font-size:15px;")
        dialog.resize(600, 400)

        # Group checkboxes for permission settings
        permissions_group = QGroupBox(f"File sharing permissions for {file_name} with {user_cred}")
        permissions_layout = QGridLayout()

        read_cb = QCheckBox("Read")
        read_cb.setChecked(read == "True")
        permissions_layout.addWidget(read_cb, 0, 0)

        write_cb = QCheckBox("Write")
        write_cb.setChecked(write == "True")
        permissions_layout.addWidget(write_cb, 0, 1)

        delete_cb = QCheckBox("Delete")
        delete_cb.setChecked(delete == "True")
        permissions_layout.addWidget(delete_cb, 1, 0)

        rename_cb = QCheckBox("Rename")
        rename_cb.setChecked(rename == "True")
        permissions_layout.addWidget(rename_cb, 1, 1)

        download_cb = QCheckBox("Download")
        download_cb.setChecked(download == "True")
        permissions_layout.addWidget(download_cb, 2, 0)

        share_cb = QCheckBox("Share")
        share_cb.setChecked(share == "True")
        permissions_layout.addWidget(share_cb, 2, 1)

        permissions_group.setLayout(permissions_layout)

        # Submit button
```

```
        submit_btn = QPushButton("Submit")
        submit_btn.setShortcut("Return")
        submit_btn.clicked.connect(lambda: self.protocol.send_share_premissions(
            dialog, file_id, user_cred,
            read_cb.isChecked(), write_cb.isChecked(), delete_cb.isChecked(),
            rename_cb.isChecked(), download_cb.isChecked(), share_cb.isChecked()
        ))

        # Layout setup
        button_layout = QHBoxLayout()
        button_layout.addSpacerItem(QSpacerItem(40, 20, QSizePolicy.Policy.Expanding, QSizePolicy.Policy.Minimum))
        button_layout.addWidget(submit_btn)

        main_layout = QVBoxLayout()
        main_layout.addWidget(permissions_group)
        main_layout.addLayout(button_layout)

        dialog.setLayout(main_layout)
        dialog.exec()

    def check_all_perms(self, perm):
        """Checks if all selected items have the specified permission."""
        return all(button.perms[perm] == "True" for button in self.currently_selected)

    def check_all_id(self):
        """Ensures all selected items have valid IDs."""
        return all(button.id is not None for button in self.currently_selected)

    def remove_selected(self, button):
        """Removes a button from the selected list."""
        if button in self.currently_selected:
            self.currently_selected.remove(button)


class FileButton(QPushButton):
    """Represents a file or folder button in the UI."""
    def __init__(self, window, text, id=None, parent=None, is_folder=False, shared_by=None, perms=None, size=0, name=""):
        super().__init__("|".join(text), parent)
        self.id = id
        self.is_folder = is_folder
        self.shared_by = shared_by
        self.perms = perms or ["True", "True", "True", "True", "True", "True"]
        self.file_size = size
        self.name = name
        self.window = window
        self.lables = []

        self.setMinimumHeight(30)
        self.setSizePolicy(QSizePolicy.Policy.Expanding, QSizePolicy.Policy.Expanding)
        button_layout = QHBoxLayout()
        button_layout.setContentsMargins(0, 0, 0, 0)
        button_layout.setSpacing(0)

        # Create labels for file button
        for i, label_text in enumerate(text):
            label = QLabel(label_text)
            if i == 0:
                if self.is_folder:
                    label.setText(f' <img src="{ASSETS_PATH + "\\folder.svg"}" width="20" height="20">'
                                  f'<label> {helper.truncate_label(label, label_text)}</label>')
                elif self.id:
                    icon_path = ASSETS_PATH + "\\file_types\\" + helper.format_file_type(label_text.split("~")
[0].split(".")[-1][:-1]) + ".svg"
                    if not os.path.isfile(icon_path):
                        icon_path = ASSETS_PATH + "\\file.svg"
                    label.setText(f' <img src="{icon_path}" width="16" height="20">'
                                  f' {helper.truncate_label(label, label_text)}')

            if self.id is None:
                label.setAlignment(Qt.AlignmentFlag.AlignCenter)
                if label_text not in ["Back", "No files or folders in this directory"]:
                    sort_key = ["Name", "Date", "Size", "Owner"][i] if i < 4 else None
                    if sort_key:
                        label.mousePressEvent = lambda event, key=sort_key: self.window.change_sort(key)
                        if sort_key == self.window.sort:
                            label.setText(f'<img src="{ASSETS_PATH}\\{"asc.svg" if self.window.sort_direction else
"dsc.svg"}" width="20" height="20">'
                                          f'<label>  {label_text}</label>')

            # Set label styling
            label.setObjectName("folder-label" if self.is_folder else "file-label" if self.id else "back-label" if
label_text == "Back" else "")
            button_layout.addWidget(label, stretch=1)
            self.lables.append(label)

        # Adjust layout spacing
        button_layout.setStretch(0, 2)
```

```
        for i in range(1, len(text)):
            button_layout.setStretch(i, 1)

        self.setLayout(button_layout)


    def contextMenuEvent(self, event: QContextMenuEvent):
        """Creates a context menu for file and folder actions."""
        menu = QMenu(self)

        # Add download option if all selected items are valid and have download permission
        if self.window.check_all_id() and self.window.check_all_perms(4) and not self.window.deleted and
self.window.currently_selected:
            action = menu.addAction(" Download")
            action.triggered.connect(self.window.protocol.download)
            action.setIcon(QIcon(ASSETS_PATH + "\\download.svg"))

        if self.window.check_all_id() and self.window.currently_selected:
            # Add delete option if all selected items have delete permission
            if self.window.check_all_perms(2):
                if (self.window.deleted and self.window.user["cwd"] == "") or not self.window.deleted:
                    action = menu.addAction(" Delete")
                    action.triggered.connect(self.window.protocol.delete)
                    action.setIcon(QIcon(ASSETS_PATH + "\\delete.svg"))

            # Add rename option if a single item is selected and rename is allowed
            if self.window.check_all_perms(3) and not self.window.deleted and len(self.window.currently_selected) == 1:
                action = menu.addAction(" Rename")
                action.triggered.connect(self.rename)
                action.setIcon(QIcon(ASSETS_PATH + "\\change_user.svg"))

            # Add share option if sharing is allowed
            if self.window.check_all_perms(5) and not self.window.deleted:
                action = menu.addAction(" Share")
                action.triggered.connect(self.window.protocol.share_action)
                action.setIcon(QIcon(ASSETS_PATH + "\\share.svg"))

            # Add remove from share option if the user is in shared files view
            if self.window.share and self.window.user["cwd"] == "" and not self.window.deleted:
                action = menu.addAction(" Remove")
                action.triggered.connect(self.window.protocol.remove)
                action.setIcon(QIcon(ASSETS_PATH + "\\remove.svg"))

        # Add new folder option if the user is not in shared or deleted mode
        if not self.window.share and not self.window.deleted:
            action = menu.addAction(" New Folder")
            action.triggered.connect(self.window.protocol.new_folder)
            action.setIcon(QIcon(ASSETS_PATH + "\\new_account.svg"))

        # Add recover option if user is in the deleted files view
        if self.window.deleted and self.window.user["cwd"] == "" and self.window.currently_selected:
            action = menu.addAction(" Recover")
            action.triggered.connect(self.window.protocol.recover)
            action.setIcon(QIcon(ASSETS_PATH + "\\new_account.svg"))

        # Add search option
        action = menu.addAction(" Search")
        action.triggered.connect(self.window.protocol.search)
        action.setIcon(QIcon(ASSETS_PATH + "\\search.svg"))

        # Display the menu at the cursor position
        menu.exec(event.globalPos())

    def rename(self):
        """Prompts the user to enter a new name and sends a rename request."""
        name = self.text().split(" | ")[0][1:]  # Extracts the current file name
        new_name = dialogs.new_name_dialog("Rename", "Enter new file name:", name)
        if new_name:
            self.window.protocol.send_data(b"RENA|" + self.id.encode() + b"|" + name.encode() + b"|" + new_name.encode())
```

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

from datetime import datetime
import xml.etree.ElementTree as ET
from PyQt6.QtCore import Qt
from PyQt6.QtGui import QFontMetrics, QGuiApplication
import hashlib, os, json, sys, re

class JsonHandle:
    """Handles file upload/download tracking in JSON format."""
    def __init__(self):
        self.uploading_files_json = f"{os.getcwd()}/cache/uploading_files.json"
        self.downloading_files_json = f"{os.getcwd()}/cache/downloading_files.json"

    def get_files_uploading_data(self):
        """Retrieves data of currently uploading files."""
        if os.path.exists(self.uploading_files_json):
            with open(self.uploading_files_json, 'r') as f:
                return json.load(f)

    def get_files_downloading_data(self):
        """Retrieves data of currently downloading files."""
        if os.path.exists(self.downloading_files_json):
            with open(self.downloading_files_json, 'r') as f:
                return json.load(f)

    def update_json(self, upload, file_id, file_path, remove=False, file=None, progress=0):
        """Updates the JSON tracking file with file upload/download details."""
        json_path = self.uploading_files_json if upload else self.downloading_files_json
        if not os.path.exists(os.getcwd() + "\\cache"):
            os.makedirs(os.getcwd() + "\\cache")
        if not os.path.exists(json_path):
            with open(json_path, 'w') as f:
                json.dump({}, f)  # Initialize as an empty dictionary

        with open(json_path, 'r') as f:
            files = json.load(f)

        if remove:  # Remove the file entry if needed
            if file_id in files:
                del files[file_id]
        else:
            if file is None:
                files[file_id] = {"file_path": file_path}
            else:
                files[file_id] = {
                    "file_path": file_path,
                    "size": file.size,
                    "is_view": file.is_view,
                    "file_name": file.file_name,
                    "progress": progress
                }

        with open(json_path, 'w') as f:
            json.dump(files, f, indent=4)

def force_exit():
    """Forces the application to exit."""
    sys.exit()

def control_pressed():
    """Checks if the Control key is pressed."""
    modifiers = QGuiApplication.queryKeyboardModifiers()
    return modifiers & Qt.KeyboardModifier.ControlModifier

def build_req_string(code, values=[]):
    """Builds a request string from a command code and a list of values."""
    return f"{code}|{'|'.join(values)}".encode()

def format_file_size(size):
    """Formats file size into a human-readable format."""
    if size < 10_000:
        return f"{size:,} B"
    elif size < 10_000_000:
        return f"{size / 1_000:,.2f} KB"
    elif size < 10_000_000_001:
        return f"{size / 1_000_000:,.2f} MB"
    elif size < 10_000_000_000_001:
        return f"{size / 1_000_000_000:,.2f} GB"
    else:
        return f"{size / 1_000_000_000_000:,.2f} TB"

def parse_file_size(size_str):
    """Parses a human-readable file size string into bytes."""
    units = {"B": 1, "KB": 1_000, "MB": 1_000_000, "GB": 1_000_000_000, "TB": 1_000_000_000_000}
    unit = size_str.split(" ")[1]
```

```
    size = size_str.split(" ")[0]
    return int(float(size) * units[unit]) if unit in units else 0

def str_to_date(str):
    """Converts a string to a datetime object."""
    return datetime.strptime(str, "%Y-%m-%d %H:%M:%S.%f") if str else datetime.min

def update_ui_size(ui_file, new_width, new_height):
    """Updates the window size in a .ui XML file."""
    tree = ET.parse(ui_file)
    root = tree.getroot()

    for widget in root.findall(".//widget[@class='QMainWindow']"):
        geometry = widget.find("property[@name='geometry']/rect")
        if geometry is not None:
            width_elem = geometry.find("width")
            height_elem = geometry.find("height")
            if width_elem is not None and height_elem is not None:
                width_elem.text = str(new_width)
                height_elem.text = str(new_height)

    tree.write(ui_file, encoding='utf-8', xml_declaration=True)

def truncate_label(label, text):
    """Truncates text with an ellipsis if it exceeds the label width."""
    font_metrics = QFontMetrics(label.font())
    max_width = int(label.width() // 1.9)

    return font_metrics.elidedText(text, Qt.TextElideMode.ElideRight, max_width) if font_metrics.horizontalAdvance(text) >
max_width else text

def update_saved_ip_port(new_ip, new_port):
    """Updates the saved IP and port values in the config file."""
    file_path = f"{os.getcwd()}/modules/config.py"
    with open(file_path, "r", encoding="utf-8") as file:
        content = file.read()

    content = re.sub(r'SAVED_IP\s*=\s*["\'].*?["\']', f'SAVED_IP = \"{new_ip}\"', content)  # Replace SAVED_IP
    content = re.sub(r'SAVED_PORT\s*=\s*\d+', f'SAVED_PORT = {new_port}', content)  # Replace SAVED_PORT

    with open(file_path, "w", encoding="utf-8") as file:
        file.write(content)

file_types = {
    "zip": ["rar"],
    "png": ["jpg", "jpeg", "jfif", "gif", "ico"],
    "mp3": ["wav"],
    "code": ["py", "js", "cs", "c", "cpp", "jar"],
    "txt": ["css"]
}

def format_file_type(type):
    """Maps file extensions to standardized categories."""
    for extension, variations in file_types.items():
        if type in variations or type == extension:
            return extension
    return type

def compute_file_md5(file_path):
    """Computes the MD5 checksum of a file."""
    hash_func = hashlib.new('md5')
    with open(file_path, 'rb') as file:
        while chunk := file.read(8192):
            hash_func.update(chunk)

    return hash_func.hexdigest()
```

```
# 2024 Â© Idan Hazay

class Limits:
    """
    Users networking and files limitations, based on subscription
    """
    def __init__(self, level):
        level = int(level)
        if (level == 0):
            self.max_storage = 100_000
            self.max_file_size = 50
            self.max_upload_speed = 5
            self.max_download_speed = 10
        elif (level == 1):
            self.max_storage = 250_000
            self.max_file_size = 100
            self.max_upload_speed = 10
            self.max_download_speed = 20
        elif (level == 2):
            self.max_storage = 500_000
            self.max_file_size = 250
            self.max_upload_speed = 15
            self.max_download_speed = 30
        elif (level == 3):
            self.max_storage = 1_000_000
            self.max_file_size = 500
            self.max_upload_speed = 25
            self.max_download_speed = 50
        else:
            raise Exception


class LimitExceeded(Exception):
    """
    Exception of a limit reached/exceeded
    """
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)
```

```python
# 2024 Â© Idan Hazay

import os, sys, logging

# Configure logging
logging.basicConfig(
    filename=f"{os.path.dirname(os.path.dirname(os.path.abspath(__file__)))}\\app_log.txt",
    level=logging.INFO,
    format='%(asctime)s - %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S'
)


class Logger:
    """
    Class to log all prints into file
    """
    def __init__(self):
        self.terminal = sys.stdout  # Store the original stdout so we can still print to console
        sys.stdout = self  # Redirect sys.stdout to the Logger instance

    def write(self, message):
        if message.strip():  # Log non-empty messages
            logging.info(message.strip())
            self.terminal.write(message + "\n")  # Also write the message to the console

    def flush(self):
        self.terminal.flush()  # Make sure to flush stdout buffer for compatibility
```

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

import struct, traceback, socket  # Struct for data packing, traceback for debugging, socket for networking
from modules import encrypting  # Import encryption module
from modules.config import *  # Import configuration settings

class Network:
    """
    Handles network communication between server and clients.
    Supports encrypted data transmission, logging, and TCP communication.
    """

    def __init__(self, clients, bytes_recieved, bytes_sent, log=False):
        self.log = log  # Enable or disable logging
        self.clients = clients  # Dictionary of connected clients
        self.encryption = encrypting.Encryption()  # Encryption handler
        self.bytes_recieved = bytes_recieved  # Track bytes received per client
        self.bytes_sent = bytes_sent  # Track bytes sent per client

    def logtcp(self, dir, tid, byte_data):
        """
        Logs TCP traffic if logging is enabled.
        """
        if self.log:
            try:
                if str(byte_data[0]) == "0":
                    print("")  # Empty print for readability
            except Exception:
                return  # Ignore exceptions

            if dir == 'sent':
                print(f'{tid} S LOG:Sent     >>> {byte_data}')  # Log sent data
            else:
                print(f'{tid} S LOG:Recieved <<< {byte_data}')  # Log received data

    def send_data(self, sock, tid, bdata):
        """
        Sends data to a client.
        Supports encryption and adds packet length for proper parsing.
        """
        if self.clients[tid].encryption:  # Check if encryption is enabled
            encrypted_data = self.encryption.encrypt(bdata, self.clients[tid].shared_secret)  # Encrypt data
            data_len = struct.pack('!l', len(encrypted_data))  # Pack length as 4-byte integer
            to_send = data_len + encrypted_data  # Combine length header and encrypted data
            to_send_decrypted = str(len(bdata)).encode() + bdata  # Decrypted version for logging
            self.logtcp('sent', tid, to_send)  # Log encrypted data
            self.logtcp('sent', tid, to_send_decrypted)  # Log decrypted data
        else:
            data_len = struct.pack('!l', len(bdata))  # Pack unencrypted data length
            to_send = data_len + bdata  # Combine length and data
            self.logtcp('sent', tid, to_send)  # Log sent data

        try:
            self.bytes_sent[tid] += len(to_send)  # Track bytes sent
            sock.send(to_send)  # Send data
        except ConnectionResetError:
            pass  # Handle client disconnection

    def recv_data(self, sock, tid):
        """
        Receives data from a client.
        Reads packet length first, then retrieves the full message.
        """
        try:
            b_len = b''
            while len(b_len) < LEN_FIELD:  # Ensure full length field is received
                b_len += sock.recv(LEN_FIELD - len(b_len))  # Read remaining bytes

            self.bytes_recieved[tid] += len(b_len)  # Track bytes received
            msg_len = struct.unpack("!l", b_len)[0]  # Extract message length

            if msg_len == b'':
                print('Client seems to have disconnected')  # Detect disconnection

            msg = b''
            while len(msg) < msg_len:  # Keep reading until full message is received
                chunk = sock.recv(msg_len - len(msg))
                self.bytes_recieved[tid] += len(chunk)  # Track bytes received
                if not chunk:
                    print('Server disconnected abnormally.')  # Handle unexpected disconnection
                    break
                msg += chunk

            if tid in self.clients and self.clients[tid].encryption:
                self.logtcp('recv', tid, b_len + msg)  # Log encrypted data
                msg = self.encryption.decrypt(msg, self.clients[tid].shared_secret)  # Decrypt message
```

```
            self.logtcp('recv', tid, str(msg_len).encode() + msg)  # Log decrypted data

        return msg  # Return received message

    except ConnectionResetError:
        return None  # Handle client disconnection
    except Exception as err:
        print(traceback.format_exc())  # Log error

@staticmethod
def dhcp_listen(local_ip, port):
    """
    Listens for DHCP discovery requests and responds with server information.
    Used for automatic client-server connection.
    """
    dhcp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)  # Create UDP socket
    dhcp_socket.bind(("", 31026))  # Listen on UDP port 31026

    while True:
        data, addr = dhcp_socket.recvfrom(1024)  # Receive data from clients
        if data.decode() == "SEAR":  # Check if the message is a search request
            response_message = f"SERR|{local_ip}|{port}"  # Construct response with server details
            dhcp_socket.sendto(response_message.encode(), addr)  # Send response to client
```

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

import traceback, time, os
from modules import validity  # Import validation module for input checking
from modules.config import *  # Configuration settings
from modules.limits import Limits  # Subscription limits and restrictions
from modules.errors import Errors  # Error handling
from filelock import FileLock  # File locking to prevent concurrent access

class Protocol:
    """
    Handles the processing of client requests and generates appropriate responses.
    This class interprets incoming messages, validates data, and executes necessary actions.
    """

    def __init__(self, network, clients, cr, files_uploading):
        self.network = network  # Handles network communication
        self.clients = clients  # Stores active client sessions
        self.v = validity.Validation()  # Instance of validation class for input checks
        self.cr = cr  # Handles client database interactions
        self.files_uploading = files_uploading  # Tracks files currently being uploaded
        self.files_in_use = []  # Stores files that are currently being accessed

    def protocol_build_reply(self, request, tid, sock):
        """
        Parses client requests, validates input, and determines the appropriate response.
        Each request has an action code that specifies the intended operation.
        """
        if request is None:
            return None

        fields = request.split(b"|")  # Split the request into parts using "|"
        code = fields[0].decode()  # Extract the action code

        if code not in ["FILD", "FILE"]:  # If not a file-related request, decode to string
            fields = request.decode().split("|")

        if code == 'EXIT':
            reply = 'EXTR'  # Send exit confirmation to client
            self.clients[tid].id = tid  # Reset client ID
            self.clients[tid].user = "dead"  # Mark client as disconnected

        elif code == "LOGN": # Client login request: validate credentials and grant access
            cred, password = fields[1], fields[2]

            if self.v.is_empty(fields[1:]):  # Ensure fields are not empty
                return Errors.EMPTY_FIELD.value
            elif self.v.check_illegal_chars(fields[1:]):  # Check for illegal characters
                return Errors.INVALID_CHARS.value

            if self.cr.login_validation(cred, password):  # Validate credentials
                if not self.cr.verified(cred):  # Ensure account is verified
                    reply = Errors.NOT_VERIFIED.value
                else:
                    user_dict = self.cr.get_user_data(cred)  # Retrieve user data from database
                    self.clients[tid].id = user_dict["id"]
                    self.clients[tid].user = user_dict["username"]
                    self.clients[tid].email = user_dict["email"]
                    self.clients[tid].subscription_level = user_dict["subscription_level"]
                    self.clients[tid].admin_level = user_dict["admin_level"]
                    reply = f"LOGS|{user_dict['email']}|{user_dict['username']}|{int(user_dict['subscription_level'])}|
{self.clients[tid].admin_level}"
            else:
                reply = Errors.LOGIN_DETAILS.value  # Send error if credentials are incorrect

        elif code == "SIGU": # Client signup request: register new users
            email, username, password, confirm_password = fields[1:5]

            if self.v.is_empty(fields[1:]):
                return Errors.EMPTY_FIELD.value
            elif self.v.check_illegal_chars(fields[1:]):
                return Errors.INVALID_CHARS.value
            elif not self.v.is_valid_email(email):
                return Errors.INVALID_EMAIL.value
            elif not self.v.is_valid_username(username) or username == "guest":
                return Errors.INVALID_USERNAME.value
            elif not self.v.is_valid_password(password):
                return Errors.PASSWORD_REQ.value
            elif password != confirm_password:
                return Errors.PASSWORDS_MATCH.value

            if self.cr.user_exists(username):  # Check if username already exists
                reply = Errors.USER_REGISTERED.value
            elif self.cr.email_registered(email):  # Check if email is already used
                reply = Errors.EMAIL_REGISTERED.value
            else:
```

```
        self.cr.signup_user([email, username, password])  # Register new user
        self.cr.send_verification(email)  # Send verification email
        reply = f"SIGS|{email}|{username}|{password}"

elif code == "FOPS": # Request password reset code
    email = fields[1]

    if self.v.is_empty(fields[1:]):
        return Errors.EMPTY_FIELD.value
    elif self.v.check_illegal_chars(fields[1:]):
        return Errors.INVALID_CHARS.value
    elif not self.v.is_valid_email(email):
        return Errors.INVALID_EMAIL.value

    if self.cr.email_registered(email):
        if not self.cr.verified(email):
            reply = Errors.NOT_VERIFIED.value
        else:
            self.cr.send_reset_mail(email)  # Send password reset email
            reply = f"FOPR|{email}"
    else:
        reply = Errors.EMAIL_NOT_REGISTERED.value

elif code == "PASR": # Reset password after receiving verification code
    email, code, new_password, confirm_new_password = fields[1:5]

    if self.v.is_empty(fields[1:]):
        return Errors.EMPTY_FIELD.value
    elif self.v.check_illegal_chars(fields[1:]):
        return Errors.INVALID_CHARS.value
    elif not self.v.is_valid_password(new_password):
        return Errors.PASSWORD_REQ.value
    elif new_password != confirm_new_password:
        return Errors.PASSWORDS_MATCH.value

    res = self.cr.check_code(email, code)  # Validate reset code
    if res == "ok":
        self.cr.change_password(email, new_password)  # Update password in database
        self.clients[tid].user = "guest"
        reply = f"PASS|{email}|{new_password}"
    elif res == "code":
        reply = Errors.NOT_MATCHING_CODE.value
    else:
        reply = Errors.CODE_EXPIRED.value

elif code == "LOGU": # Client logout request
    self.clients[tid].id = tid
    self.clients[tid].user = "guest"
    self.clients[tid].email = "guest"
    self.clients[tid].subscription_level = 0
    self.clients[tid].admin_level = 0
    reply = "LUGR"  # Logout confirmation message

elif code == "SVER": # Resend verification email for unverified accounts
    email = fields[1]

    if self.v.is_empty(fields[1:]):
        return Errors.EMPTY_FIELD.value
    elif self.v.check_illegal_chars(fields[1:]):
        return Errors.INVALID_CHARS.value
    elif not self.v.is_valid_email(email):
        return Errors.INVALID_EMAIL.value

    if self.cr.email_registered(email):
        if self.cr.verified(email):
            reply = Errors.ALREADY_VERIFIED.value
        else:
            self.cr.send_verification(email)
            reply = f"VERS|{email}"
    else:
        reply = Errors.EMAIL_NOT_REGISTERED.value

elif (code == "VERC"):   # Client requests account verification
    email = fields[1]
    code = fields[2]

    if (self.v.is_empty(fields[1:])):
        return Errors.EMPTY_FIELD.value
    elif (self.v.check_illegal_chars(fields[1:])):
        return Errors.INVALID_CHARS.value
    elif (not self.v.is_valid_email(email)):
        return Errors.INVALID_EMAIL.value

    if (self.cr.email_registered(email)):
        res = self.cr.check_code(email, code)
        if (res == "ok"):
            self.cr.verify_user(email)
```

```
                self.cr.send_welcome_mail(email)
                reply = f"VERR|{email}"
            elif (res == "code"):
                reply = Errors.NOT_MATCHING_CODE.value
            else:
                reply = Errors.CODE_EXPIRED.value
        else:
            reply = Errors.EMAIL_NOT_REGISTERED.value

    elif code == "DELU": # Client requests account deletion
        email = fields[1]
        user_id = self.clients[tid].id  # Get the user's ID

        if self.cr.user_exists(user_id):  # Check if user exists
            self.cr.delete_user(user_id)  # Delete user from the database
            self.clients[tid].id = tid  # Reset client ID
            self.clients[tid].user = "guest"  # Mark client as logged out
            reply = f"DELR|{email}"  # Confirm deletion to client
        else:
            reply = Errors.LOGIN_DETAILS.value  # Return error if user does not exist

    elif code == "FILS" or code == "UPFL": # Client requests to start uploading a file
        file_name, parent, size, file_id = fields[1:5]
        size = int(size)  # Convert size to integer

        try:
            if self.is_guest(tid):  # Ensure the user is logged in
                reply = Errors.NOT_LOGGED.value
            elif not self.cr.is_dir_owner(self.clients[tid].id, parent):  # Check directory ownership
                reply = Errors.NO_PERMS.value
            elif size > Limits(self.clients[tid].subscription_level).max_file_size * 1_000_000:
                reply = Errors.SIZE_LIMIT.value + f" {Limits(self.clients[tid].subscription_level).max_file_size} MB"
            elif self.cr.get_user_storage(self.clients[tid].user) >
Limits(self.clients[tid].subscription_level).max_storage * 1_000_000:
                reply = Errors.MAX_STORAGE.value
            elif file_id in self.files_uploading.keys():
                reply = Errors.ALREADY_UPLOADING.value
            else:
                # Generate a new file record
                if code == "UPFL":
                    name = self.cr.get_file_sname(file_name)
                    if os.path.exists(CLOUD_PATH + "\\" + name):
                        os.remove(CLOUD_PATH + "\\" + name)  # Delete existing file
                    self.files_uploading[file_id] = File(name, parent, size, file_id, file_name)
                    self.cr.update_file_size(file_name, size)  # Update size in the database
                    reply = f"UPFR|{file_name}|was updated successfully"
                else:
                    name = self.cr.gen_file_name()  # Generate a unique name
                    self.files_uploading[file_id] = File(name, parent, size, file_id, file_name)
                    reply = f"FISS|{file_name}|Upload started"
        except Exception:
            print(traceback.format_exc())  # Log any errors
            reply = Errors.FILE_UPLOAD.value  # Return upload error

    elif code == "FILD" or code == "FILE": # File chunk received from client
        file_id = fields[1].decode()
        location_infile = int(fields[2].decode())
        data = request[4 + len(file_id) + len(str(location_infile)) + 3:]  # Extract file data

        file = None
        for i in range(5):  # Retry logic to ensure file is in tracking list
            if file_id in self.files_uploading.keys():
                file = self.files_uploading[file_id]
                break
            time.sleep(1)

        if file is None:
            return Errors.FILE_NOT_FOUND.value + "|" + file_id  # Return error if file is missing

        # Permission and storage checks
        if self.is_guest(tid):
            reply = Errors.NOT_LOGGED.value
        elif not self.cr.is_dir_owner(self.clients[tid].id, file.parent):
            reply = Errors.NO_PERMS.value
        elif file.size > Limits(self.clients[tid].subscription_level).max_file_size * 1_000_000:
            reply = Errors.SIZE_LIMIT.value + f" {Limits(self.clients[tid].subscription_level).max_file_size} MB"
        elif self.cr.get_user_storage(self.clients[tid].user) >
Limits(self.clients[tid].subscription_level).max_storage * 1_000_000:
            reply = Errors.MAX_STORAGE.value
        else:
            if location_infile + len(data) > file.size:
                return Errors.FILE_SIZE.value  # Ensure data does not exceed allocated size
            file.add_data(data, location_infile)  # Write data to the file

            if code == "FILE":  # Final chunk received
                if file.name != self.clients[tid].user:
                    self.cr.new_file(file.name, file.file_name, file.parent, self.clients[tid].id, file.size)
```

```
                    reply = f"FILR|{file.file_name}|File finished uploading"
                else:
                    reply = f"ICUP|Profile icon uploaded"

                if file_id in self.files_uploading.keys():
                    del self.files_uploading[file_id]  # Remove from tracking list
            else:
                reply = ""  # Continue uploading


        elif (code == "GETP" or code == "GETD" or code == "GESP" or code == "GESD" or code == "GEDP" or code == "GEDD"): #
Client requests files or directories list (personal/shared/deleted)
            directory = fields[1]  # Directory ID or path
            amount = int(fields[2])  # Number of items to fetch
            sort = fields[3]  # Sorting parameter (Name, Date, Size, etc.)
            sort_direction = fields[4] == "True"  # Sorting order (ascending/descending)

            search_filter = fields[5] if len(fields) == 6 else None  # Optional search filter

            prev_amount = 0  # Keeps track of file count before adding directories

            if (code == "GETP"):  # Get personal files
                items = self.cr.get_files(self.clients[tid].id, directory, search_filter)
                reply = "PATH"
            elif (code == "GETD"):  # Get personal directories
                items = self.cr.get_directories(self.clients[tid].id, directory, search_filter)
                prev_amount = len(self.cr.get_files(self.clients[tid].id, directory, search_filter))  # Track previous
file count
                reply = "PATD"
            elif (code == "GESP"):  # Get shared files
                items = self.cr.get_share_files(self.clients[tid].id, directory, search_filter)
                reply = "PASH"
            elif (code == "GESD"):  # Get shared directories
                items = self.cr.get_share_directories(self.clients[tid].id, directory, search_filter)
                prev_amount = len(self.cr.get_share_files(self.clients[tid].id, directory, search_filter))  # Track
previous file count
                reply = "PASD"
            elif (code == "GEDP"):  # Get deleted files
                items = self.cr.get_deleted_files(self.clients[tid].id, directory, search_filter)
                reply = "PADH"
            elif (code == "GEDD"):  # Get deleted directories
                items = self.cr.get_deleted_directories(self.clients[tid].id, directory, search_filter)
                prev_amount = len(self.cr.get_deleted_files(self.clients[tid].id, directory, search_filter))  # Track
previous file count
                reply = "PADD"

            total = len(items) + prev_amount  # Calculate total items in the directory
            amount -= prev_amount  # Adjust the number of items based on previous count

            if amount > len(items):
                amount = len(items)  # Ensure not exceeding available items
            elif amount < 0:
                amount = 0  # Prevent negative count

            # Sorting logic based on user preference
            if sort == "Name" or ((code == "GETD" or code == "GESD" or code == "GEDD") and sort == "Owner"):
                items = sorted(items, key=lambda x: x.split("~")[0].lower(), reverse=sort_direction)  # Sort by name or
owner

            elif sort == "Date":
                if (code == "GETD" or code == "GESD" or code == "GEDD"):
                    items = sorted(items, key=lambda x: self.cr.str_to_date(x.split("~")[2]), reverse=sort_direction)  #
Sort directories by date
                else:
                    items = sorted(items, key=lambda x: self.cr.str_to_date(x.split("~")[1]), reverse=sort_direction)  #
Sort files by date

            elif sort == "Type" and (code == "GETP" or code == "GESP" or code == "GEDP"):
                items = sorted(items, key=lambda x: x.split("~")[0].split(".")[-1].lower(), reverse=sort_direction)  #
Sort by file extension

            elif sort == "Size":
                if (code == "GETD" or code == "GESD" or code == "GEDD"):
                    items = sorted(items, key=lambda x: int(x.split("~")[3]), reverse=sort_direction)  # Sort directories
by size
                else:
                    items = sorted(items, key=lambda x: int(x.split("~")[2]), reverse=sort_direction)  # Sort files by
size

            elif sort == "Owner" and (code == "GETD" or code == "GESD" or code == "GEDD"):
                items = sorted(items, key=lambda x: x.split("~")[4].lower(), reverse=sort_direction)  # Sort by owner (for
shared content)

            reply += f"|{total}"  # Include total count in response

            for item in items[:amount]:  # Append requested number of items to response
                reply += f"|{item}"
```

```
        elif (code == "MOVD"): # Client requests to move to a different directory
            directory_id = fields[1]  # Extract the target directory ID

            if (self.cr.valid_directory(directory_id, self.clients[tid].id) or directory_id == ""):
                self.clients[tid].cwd = directory_id  # Update current working directory
                reply = f"MOVR|{directory_id}|{self.cr.get_parent_directory(directory_id)}|
{self.cr.get_full_path(directory_id)}|moved successfully"
            else:
                self.clients[tid].cwd = ""  # Reset to root if directory is invalid
                reply = f"MOVR|{''}|{self.cr.get_parent_directory('')}|{self.cr.get_full_path('')}|moved successfully"

        elif (code == "DOWN"): # Client requests to download a file or folder
            file_id = fields[1]

            if "~" in file_id:  # Multiple files requested (zip them)
                name = fields[2]  # Name of the zip file
                ids = file_id.split("~")  # Split multiple file IDs

                for id in ids:
                    if not self.cr.can_download(self.clients[tid].id, id) or self.is_guest(tid):
                        reply = Errors.NO_PERMS.value  # Permission denied
                        return reply
                    elif self.cr.get_file_sname(id) is None and self.cr.get_dir_name(id) is None:
                        reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File not found
                        return reply

                zip_buffer = self.cr.zip_files(ids)  # Create a zip archive of the files
                self.send_zip(zip_buffer, file_id, sock, tid)  # Send the zipped file to the client
                zip_buffer.close()
                reply = f"DOWR|{name}|{file_id}|was downloaded"
            else:
                if not self.cr.can_download(self.clients[tid].id, file_id) or self.is_guest(tid):
                    reply = Errors.NO_PERMS.value  # Permission denied
                    return reply
                elif self.cr.get_dir_name(file_id) is not None:
                    zip_buffer = self.cr.zip_directory(file_id)  # Zip the entire directory
                    self.send_zip(zip_buffer, file_id, sock, tid)  # Send the zip file
                    zip_buffer.close()
                    reply = f"DOWR|{self.cr.get_dir_name(file_id)}|{file_id}|was downloaded"
                    return reply
                elif self.cr.get_file_sname(file_id) is None:
                    reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File not found
                    return reply

                file_path = CLOUD_PATH + "\\" + self.cr.get_file_sname(file_id)  # Get the full file path
                if not os.path.isfile(file_path):
                    reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # Ensure the file exists
                else:
                    try:
                        self.send_file_data(file_path, file_id, sock, tid)  # Send file data
                        reply = f"DOWR|{self.cr.get_file_fname(file_id)}|was downloaded"
                    except Exception:
                        reply = Errors.FILE_DOWNLOAD.value  # Handle file download errors

        elif (code == "NEWF"): # Client requests to create a new folder
            folder_name = fields[1]
            folder_path = self.clients[tid].cwd  # Get current directory

            if not self.cr.is_dir_owner(self.clients[tid].id, folder_path) or self.is_guest(tid):
                reply = Errors.NO_PERMS.value  # Permission denied
            else:
                self.cr.create_folder(folder_name, folder_path, self.clients[tid].id)  # Create the folder
                reply = f"NEFR|{folder_name}|was created"

        elif (code == "RENA"): # Client requests to rename a file or directory
            file_id, name, new_name = fields[1:4]

            if self.v.is_empty(fields[1:]):
                return Errors.EMPTY_FIELD.value  # Ensure fields are not empty
            elif not self.cr.can_rename(self.clients[tid].id, file_id):
                reply = Errors.NO_PERMS.value  # Permission denied
            else:
                if self.cr.get_file_fname(file_id) is not None:
                    self.cr.rename_file(file_id, new_name)  # Rename file
                else:
                    self.cr.rename_directory(file_id, new_name)  # Rename directory
                reply = f"RENR|{name}|{new_name}|File renamed successfully"

        elif (code == "GICO"): # sending user icon
            if (os.path.isfile(os.path.join(USER_ICONS_PATH, self.clients[tid].id) + ".ico")): # checl of user has icon
                self.send_file_data(os.path.join(USER_ICONS_PATH, self.clients[tid].id) + ".ico", "user", sock, tid)
            else:
                self.send_file_data(os.path.join(USER_ICONS_PATH, "guest.ico"), "user", sock, tid) # send generic icon
            reply = f"GICR|Sent use profile picture"

        elif (code == "ICOS"): # uploading new user icon
```

```
        size = int(fields[3])
        id = fields[4]
        try:
            self.files_uploading[id] = File(self.clients[tid].id, "", size, id, self.clients[tid].id, icon=True)
            reply = f"ICOR|Profile icon started uploading"

        except Exception:
            print(traceback.format_exc())
            reply = Errors.FILE_UPLOAD.value

    elif (code == "DELF"): # Client requests to delete a file or folder
        file_id = fields[1]

        if not self.cr.can_delete(self.clients[tid].id, file_id):
            reply = Errors.NO_PERMS.value  # Permission denied
        elif file_id in self.files_in_use:
            reply = Errors.IN_USE.value  # File is currently in use
        elif self.cr.get_file_fname(file_id) is not None:
            name = self.cr.get_file_fname(file_id)
            self.cr.delete_file(file_id)  # Delete file from storage
            reply = f"DLFR|{name}|was deleted!"
        elif self.cr.get_dir_name(file_id) is not None:
            name = self.cr.get_dir_name(file_id)
            self.cr.delete_directory(file_id)  # Delete directory
            reply = f"DFFR|{name}|was deleted!"
        else:
            reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File not found


    elif code == "SUBL": # change subscription level
        level = fields[1]
        if (level == self.clients[tid].subscription_level): # check level validity
            reply = Errors.SAME_LEVEL.value
        elif (int(level) < 0 or int(level) > 3):
            reply = Errors.INVALID_LEVEL.value
        else:
            self.cr.change_level(self.clients[tid].id, int(level)) # change it
            self.clients[tid].subscription_level = int(level)
            reply = f"SUBR|{level}|Subscription level updated"

    elif (code == "GEUS"): # Client requests current storage usage
        used_storage = self.cr.get_user_storage(self.clients[tid].id)  # Get userâ€™s used storage
        reply = f"GEUR|{used_storage}"

    elif (code == "CHUN"): # Client requests to change their username
        new_username = fields[1]

        if self.v.is_empty(fields[1:]):
            return Errors.EMPTY_FIELD.value  # Ensure field is not empty
        elif self.v.check_illegal_chars(fields[1:]):
            return Errors.INVALID_CHARS.value  # Check for illegal characters
        elif not self.v.is_valid_username(new_username) or new_username == "guest":
            return Errors.INVALID_USERNAME.value  # Validate username
        elif self.cr.user_exists(new_username):
            reply = Errors.USER_REGISTERED.value  # Username is already taken
        else:
            self.cr.change_username(self.clients[tid].id, new_username)  # Update username in database
            self.clients[tid].user = new_username  # Update username in session
            reply = f"CHUR|{new_username}|Changed username"

    elif code == "VIEW": # send file to view
        file_id = fields[1]
        file_path = CLOUD_PATH + "\\" + self.cr.get_file_sname(file_id)
        if(not self.cr.can_download(self.clients[tid].id, file_id)):
            reply = Errors.NO_PERMS.value + "|" + self.cr.get_file_fname(file_id)
        elif (not os.path.isfile(file_path)):
            reply = Errors.FILE_NOT_FOUND.value + "|" + file_id
        elif (os.path.getsize(file_path) > 10_000_000):
            reply = f"{Errors.PREVIEW_SIZE.value}|{self.cr.get_file_fname(file_id)}"
        elif file_id in self.files_in_use: # if file is already in view dont allow
            reply = Errors.IN_USE.value
        else:
            try:
                self.send_file_data(file_path, file_id, sock, tid) # send the file
                self.files_in_use.append(file_id)
                reply = f"VIER|{self.cr.get_file_fname(file_id)}|was viewed"
            except Exception:
                reply = Errors.FILE_DOWNLOAD.value


    elif (code == "GENC"): # Client requests to generate a new authentication cookie
        if self.is_guest(tid):
            reply = Errors.NOT_LOGGED.value  # Ensure user is logged in
        else:
            self.cr.generate_cookie(self.clients[tid].id)  # Generate new cookie
            reply = f"COOK|{self.cr.get_cookie(self.clients[tid].id)}"
```

```
        elif (code == "COKE"): # Client sends authentication cookie for validation
            cookie = fields[1]
            user_dict = self.cr.get_user_data(cookie)  # Retrieve user data

            if user_dict is None:
                reply = Errors.INVALID_COOKIE.value  # Invalid cookie
            elif self.cr.cookie_expired(user_dict["id"]):
                reply = Errors.EXPIRED_COOKIE.value  # Expired cookie
            else:
                username = user_dict["username"]
                email = user_dict["email"]
                self.clients[tid].id = user_dict["id"]
                self.clients[tid].user = user_dict["username"]
                self.clients[tid].email = user_dict["email"]
                self.clients[tid].subscription_level = user_dict["subscription_level"]
                self.clients[tid].admin_level = user_dict["admin_level"]
                reply = f"LOGS|{email}|{username}|{int(self.clients[tid].subscription_level)}|
{self.clients[tid].admin_level}"

        elif (code == "SHRS"): # Client requests to share a file or folder with another user
            file_id = fields[1]  # File or folder ID
            user_cred = fields[2]  # Email or username of the recipient

            if self.cr.get_file_fname(file_id) is None and self.cr.get_dir_name(file_id) is None:
                reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File or folder does not exist
            elif not self.cr.can_share(self.clients[tid].id, file_id):
                reply = Errors.NO_PERMS.value  # User does not have permission to share this file
            elif user_cred == self.clients[tid].email or user_cred == self.clients[tid].user:
                reply = Errors.SELF_SHARE.value  # Cannot share a file with yourself
            elif self.cr.is_file_owner(self.cr.get_user_id(user_cred), file_id) or
self.cr.is_dir_owner(self.cr.get_user_id(user_cred), file_id):
                reply = Errors.OWNER_SHARE.value  # Cannot share a file with its owner
            elif self.cr.get_user_data(user_cred) is None:
                reply = Errors.USER_NOT_FOUND.value  # Recipient user not found
            else:
                sharing = self.cr.get_share_options(file_id, user_cred)  # Get existing share settings
                if sharing is None:
                    reply = f"SHRR|{file_id}|{user_cred}|{self.cr.get_file_fname(file_id)}"  # File is not yet shared
                else:
                    reply = f"SHRR|{file_id}|{user_cred}|{self.cr.get_file_fname(file_id)}|" + "|".join(sharing[4:])  #
Return existing share settings

        elif (code == "SHRP"): # Client updates sharing permissions for a file or folder
            file_id = fields[1]  # File ID
            user_cred = fields[2]  # Email or username of recipient

            if self.cr.get_file_fname(file_id) is None and self.cr.get_dir_name(file_id) is None:
                reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File does not exist
            elif not self.cr.can_share(self.clients[tid].id, file_id):
                reply = Errors.NO_PERMS.value  # User lacks permission to share
            elif user_cred == self.clients[tid].email or user_cred == self.clients[tid].user:
                reply = Errors.SELF_SHARE.value  # Cannot share with yourself
            elif self.cr.is_file_owner(self.cr.get_user_id(user_cred), file_id) or
self.cr.is_dir_owner(self.cr.get_user_id(user_cred), file_id):
                reply = Errors.OWNER_SHARE.value  # Cannot share with the owner
            elif self.cr.get_user_data(user_cred) is None:
                reply = Errors.USER_NOT_FOUND.value  # Recipient not found
            else:
                self.cr.share_file(file_id, user_cred, fields[3:])  # Update sharing permissions
                reply = f"SHPR|Sharing option with {user_cred} have been updated"

        elif code == "SHRE": # Client requests to remove sharing permissions for a file or folder
            file_id = fields[1]  # File ID
            file_name = self.cr.get_file_fname(file_id) or self.cr.get_dir_name(file_id)  # Get file or folder name

            if file_name is None:
                reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File not found
            else:
                self.cr.remove_share(self.clients[tid].id, file_id)  # Remove sharing permissions
                reply = f"SHRM|{file_name}|Share removed"

        elif code == "RECO": # Client requests to recover a deleted file or folder
            file_id = fields[1]

            if not self.cr.can_delete(self.clients[tid].id, file_id):
                reply = Errors.NO_PERMS.value  # User lacks permission to recover
            elif self.cr.get_file_fname(file_id) is not None:
                name = self.cr.get_file_fname(file_id)  # Get file name
            elif self.cr.get_dir_name(file_id) is not None:
                name = self.cr.get_dir_name(file_id)  # Get folder name

            if name is None:
                reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File not found
            else:
                self.cr.recover(file_id)  # Restore file from deleted state
                reply = f"RECR|{name}|was recovered!"
```

```
        elif code == "VIEE": # Client stops viewing a file
            file_id = fields[1]
            self.files_in_use.remove(file_id)  # Remove file from in-use list
            reply = f"VIRR|{file_id}|stop viewing"

        elif code == "STOP": # Client requests to stop an ongoing file upload
            file_id = fields[1]
            name = self.remove_file_mid_down(file_id)  # Remove file from upload list
            reply = f"STOR|{name}|{file_id}|File upload stopped"

        elif code == "RESU": # Client requests to resume an interrupted file upload
            file_id = fields[1]

            if file_id in self.files_uploading.keys():
                progress = self.files_uploading[file_id].curr_location_infile  # Get current upload progress
                reply = f"RESR|{file_id}|{progress}"  # Send resume position
            else:
                reply = Errors.FILE_NOT_FOUND.value + "|" + file_id  # File not found

        elif code == "RESD": # Client requests to resume an interrupted file download
            id = fields[1]
            progress = int(fields[2])
            if self.cr.get_file_sname(id) != None:
                file_path = CLOUD_PATH + "\\" + self.cr.get_file_sname(id)
                self.send_file_data(file_path, id, sock, tid, progress) # if file send file
            elif self.cr.get_dir_name(id) != None:
                zip_buffer = self.cr.zip_directory(id) # if folder send zip
                self.send_zip(zip_buffer, id, sock, tid)
                zip_buffer.close()
            elif "~" in id:
                ids = id.split("~")
                zip_buffer = self.cr.zip_files(ids)
                self.send_zip(zip_buffer, id, sock, tid)
                zip_buffer.close()
            else:
                reply = Errors.FILE_NOT_FOUND.value + "|" + id
                return reply
            reply = f"RUSR|{id}|{progress}"

        elif code == "UPDT": # Client sends an update message to the server
            msg = fields[1]  # Message content
            reply = f"UPDR|{msg}"  # Send update message back

        elif code == "ADMN": # Client requests admin data
            if self.clients[tid].admin_level > 0:
                reply = f"ADMR"
                for user in self.cr.get_admin_table():
                    id, email, username, verified, subscription_level, admin_level = user[0], user[1], user[2], user[7],
user[8], user[9]
                    files_amount, total_storage_used = self.cr.get_user_total_files(id), self.cr.get_user_storage(id)
                    reply +=f"|{id}~{email}~{username}~{verified}~{subscription_level}~{admin_level}~{files_amount}~
{total_storage_used}"

            else:
                reply = Errors.NO_PERMS.value
        else:
            # Unknown request received
            reply = Errors.UNKNOWN.value  # Return generic error message
            fields = ''

        return reply  # Send response to client

    def send_file_data(self, file_path, id, sock, tid, progress=0):
        """
        Sends a file's content to the client in chunks.
        Supports resuming from a given progress point.
        """
        lock_path = f"{file_path}.lock"  # Lock file path to prevent concurrent access
        lock = FileLock(lock_path)  # Create a lock object

        if not os.path.isfile(file_path):
            raise Exception  # Raise an error if file doesn't exist

        size = os.path.getsize(file_path)  # Get total file size
        left = size % CHUNK_SIZE  # Get remainder bytes outside full chunks
        sent = progress  # Track amount of data sent

        start = time.time()
        bytes_sent = 0

        try:
            with lock:  # Acquire lock before reading the file
                with open(file_path, 'rb') as f:
                    f.seek(progress)  # Move file pointer to resume position

                    for i in range((size - progress) // CHUNK_SIZE):
                        location_infile = f.tell()  # Get current position
```

```
                    data = f.read(CHUNK_SIZE)  # Read file in chunks
                    current_time = time.time()
                    elapsed_time = current_time - start

                    if elapsed_time >= 1.0:  # Reset bandwidth tracking every second
                        start = current_time
                        bytes_sent = 0

                    self.network.send_data(sock, tid, f"RILD|{id}|{location_infile}|".encode() + data)
                    bytes_sent += len(data)  # Update sent bytes counter
                    sent += CHUNK_SIZE

                    # Check if bandwidth limit is exceeded
                    if bytes_sent >= (Limits(self.clients[tid].subscription_level).max_download_speed) * 1_000_000:
                        time_to_wait = 1.0 - elapsed_time
                        if time_to_wait > 0:
                            time.sleep(time_to_wait)  # Pause to respect speed limit

                location_infile = f.tell()
                data = f.read(left)  # Read remaining bytes
                if data != b"":
                    self.network.send_data(sock, tid, f"RILE|{id}|{location_infile}|".encode() + data)  # Send last
chunk
        except:
            print(traceback.format_exc())  # Log error
            if os.path.exists(lock_path):
                os.remove(lock_path)  # Remove lock file if an error occurs
            raise

    def send_zip(self, zip_buffer, id, sock, tid, progress=0):
        """
        Sends a compressed zip file to the client in chunks.
        Supports resuming from a given progress point.
        """
        size = len(zip_buffer.getbuffer())  # Get zip file size
        left = size % CHUNK_SIZE  # Get remainder bytes
        sent = progress  # Track amount sent

        start = time.time()
        bytes_sent = 0

        try:
            zip_buffer.seek(progress)  # Move to the resume position

            for i in range((size - progress) // CHUNK_SIZE):
                location_infile = zip_buffer.tell()
                data = zip_buffer.read(CHUNK_SIZE)  # Read zip file in chunks
                current_time = time.time()
                elapsed_time = current_time - start

                if elapsed_time >= 1.0:  # Reset tracking every second
                    start = current_time
                    bytes_sent = 0

                self.network.send_data(sock, tid, f"RILD|{id}|{location_infile}|".encode() + data)
                bytes_sent += len(data)
                sent += CHUNK_SIZE

                # Check if bandwidth limit is exceeded
                if bytes_sent >= (Limits(self.clients[tid].subscription_level).max_download_speed) * 1_000_000:
                    time_to_wait = 1.0 - elapsed_time
                    if time_to_wait > 0:
                        time.sleep(time_to_wait)  # Pause to respect speed limit

            location_infile = zip_buffer.tell()
            data = zip_buffer.read(left)  # Read last part
            if data != b"":
                self.network.send_data(sock, tid, f"RILE|{id}|{location_infile}|".encode() + data)  # Send final chunk
        except:
            raise  # Raise any encountered exception

    def is_guest(self, tid):
        """
        Checks if a client is a guest user.
        """
        return self.clients[tid].user == "guest"

    def remove_file_mid_down(self, id):
        """
        Cancels an ongoing file upload and deletes the file.
        """
        if id in self.files_uploading.keys():
            name = self.files_uploading[id].file_name
            file_id = self.cr.get_file_id(self.files_uploading[id].name)  # Retrieve actual file ID
            self.cr.delete_file(file_id)  # Delete file from database
            del self.files_uploading[id]  # Remove file from active uploads
            return name  # Return the name of the deleted file
```

```
class File:
    """
    Represents a file being uploaded or downloaded.
    Handles file storage, tracking, and writing data incrementally.
    """

    def __init__(self, name, parent, size, id, file_name, curr_location_infile=0, icon=False):
        self.name = name  # Internal storage name of the file
        self.parent = parent  # Parent directory ID
        self.size = size  # File size in bytes
        self.id = id  # Unique file identifier
        self.file_name = file_name  # Original file name as uploaded
        self.curr_location_infile = curr_location_infile  # Tracks upload/download progress
        self.save_path = USER_ICONS_PATH + "\\" + self.name + ".ico" if icon else CLOUD_PATH + "\\" + self.name  #
Determine save location
        self.start_download()  # Initialize the file for writing

    def start_download(self):
        """
        Prepares the file for writing by creating an empty file of the correct size.
        """
        with open(self.save_path, 'wb') as f:
            f.seek(self.size - 1)  # Move to the last byte of the file
            f.write(b"\0")  # Write a null byte to allocate space
            f.flush()  # Ensure data is written to disk

    def add_data(self, data, location_infile):
        """
        Writes received data to the file at the specified location.
        Uses file locking to prevent simultaneous writes.
        """
        lock_path = f"{self.save_path}.lock"  # Define a lock file path
        lock = FileLock(lock_path)  # Create a lock object

        try:
            with lock:  # Acquire lock before writing
                with open(self.save_path, 'r+b') as f:
                    f.seek(location_infile)  # Move to the correct write position
                    f.write(data)  # Write received data
                    f.flush()  # Ensure data is written to disk
                    self.curr_location_infile = location_infile  # Update progress tracking
        except:
            print(traceback.format_exc())  # Print error if write operation fails
            self.uploading = False  # Mark file as inactive due to failure
        finally:
            try:
                if os.path.exists(lock_path):  # Remove lock file if it exists
                    os.remove(lock_path)
            except:
                pass  # Ignore errors when removing the lock file

    def delete(self):
        """
        Deletes the file from storage.
        Ensures any lock file is removed before deleting the actual file.
        """
        lock_path = f"{self.save_path}.lock"
        if os.path.exists(lock_path):
            os.remove(lock_path)  # Remove lock file if it exists
        if os.path.exists(self.save_path):
            os.remove(self.save_path)  # Delete the actual file
```

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

import threading
from PyQt6.QtCore import pyqtSignal, QThread

class ReceiveThread(QThread):
        # Define a signal to emit data received from recv_data
        reply_received = pyqtSignal(bytes)

        def __init__(self, network):
            super().__init__()
            self.running = True  # Add a flag to control the thread loop
            self._pause_event = threading.Event()  # Event to manage pausing
            self._pause_event.set()  # Initially, the thread is not paused
            self.network = network

        def run(self):
            while self.running:
                # Wait for the thread to be resumed if paused
                self._pause_event.wait()

                # Simulate receiving data
                reply = self.network.recv_data()  # Assume this method exists and returns bytes
                if reply:
                    self.reply_received.emit(reply)  # Emit the received reply to the main thread

        def pause(self):
            self._pause_event.clear()

        def resume(self):
            self._pause_event.set()
```

```
# 2024 Â© Idan Hazay

from modules import client_requests, networking, protocol
from modules.config import *
from modules.errors import Errors
from modules.logger import Logger

import socket, traceback, time, threading, sys
from requests import get

class Application:
    """
    Main server application handling client connections, requests, and general server functionality.
    """
    def __init__(self, addr):
        self.clients = {}
        self.bytes_recieved = {}
        self.bytes_sent = {}
        self.files_uploading = {}
        self.all_to_die = False
        self.network = networking.Network(self.clients, self.bytes_recieved, self.bytes_sent)
        self.cr = client_requests.ClientRequests()
        self.protocol = protocol.Protocol(self.network, self.clients, self.cr, self.files_uploading)
        self.addr = addr
        self.start()

    def start(self):
        """
        Start the server, listen for client connections, and manage threads for each client.
        """
        threads = []
        self.srv_sock = socket.socket()  # Server socket initialization
        self.srv_sock.bind(self.addr)  # Bind the server to the provided address
        self.srv_sock.listen(20)

        print(f"Server listening on {self.addr}")

        try:
            self.public_ip = get('https://api.ipify.org').content.decode('utf8')  # Fetch public IP
        except Exception:
            self.public_ip = "No IP found"

        try:
            with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
                s.connect(("8.8.8.8", 80))  # Google's DNS server for local IP discovery
                self.local_ip = s.getsockname()[0]
        except:
            self.local_ip = "127.0.0.1"

        print(f"Public server ip: {self.public_ip}, local server ip: {self.local_ip}")

        self.srv_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)  # Enable address reuse
        i = 1

        try:
            self.network.encryption.create_keys()  # Encryption key generation
            self.network.encryption.load_keys()
            scheduler = threading.Thread(target=self.cleaner)  # Start cleanup process
            scheduler.start()
        except:
            print(traceback.format_exc())
            self.srv_sock.close()
            return

        dhcp_listener = threading.Thread(target=self.network.dhcp_listen, args=(self.local_ip, self.addr[1]))  # DHCP
listening thread
        dhcp_listener.start()

        print('Main thread: before accepting ...\n')
        while True:
            cli_sock, addr = self.srv_sock.accept()  # Accept incoming client connection
            t = threading.Thread(target=self.handle_client, args=(cli_sock, str(i), addr))
            t.start()  # Start client thread
            i += 1
            threads.append(t)
            if i > 100000000:
                print('\nMain thread: going down for maintenance')
                break

        self.all_to_die = True  # Stop all client threads
        print('Main thread: waiting to all clients to die')
        for t in threads:
            t.join()  # Ensure all threads finish

        self.srv_sock.close()
        print('Bye ..')
```

```
    def handle_client(self, sock, tid, addr):
        """
        Handle an individual client connection, initialize secure communication, and process client requests.
        """
        try:
            finish = False
            print(f'New Client number {tid} from {addr}')
            self.bytes_sent[tid] = 0
            self.bytes_recieved[tid] = 0
            start = self.network.recv_data(sock, tid)  # Receive initial client data
            code = start.split(b"|")[0]

            self.clients[tid] = Client(tid, "guest", "guest", 0, 0, None, False)  # Initialize client with guest role

            if code == b"RSAR":
                shared_secret = self.network.encryption.rsa_exchange(sock, tid)  # RSA key exchange
            if shared_secret == "":
                return

            self.clients[tid].shared_secret = shared_secret
            self.clients[tid].encryption = True  # Mark client as encrypted

        except Exception:
            print(traceback.format_exc())
            print(f'Client {tid} connection error')
            if tid in self.clients:
                self.clients[tid] = None  # Remove problematic client
            sock.close()
            return

        while not finish and self.clients[tid] is not None:
            if self.all_to_die:
                print('Will close due to main server issue')
                break
            try:
                entire_data = self.network.recv_data(sock, tid)  # Read client data
                t = threading.Thread(target=self.handle_request, args=(entire_data, tid, sock))
                t.start()

            except socket.error as err:
                print(f'Socket Error exit client loop: err: {err}')
                break
            except Exception as err:
                print(f'General Error: {err}')
                print(traceback.format_exc())
                break

        print(f'Client {tid} Exit')
        self.clients[tid] = None  # Mark client as disconnected
        sock.close()

    def handle_request(self, request, tid, sock):
        """
        Parse and handle a client request, sending appropriate responses.
        """
        try:
            to_send = self.protocol.protocol_build_reply(request, tid, sock)  # Build a response for the client
            if to_send is None:
                self.clients[tid] = None  # Mark client as disconnected
                print(f"Client {tid} disconnected")
                return
            to_send = to_send.encode()
            self.network.send_data(sock, tid, to_send)  # Send data back to client

            if to_send == b"EXTR":
                self.clients[tid] = None  # Disconnect client explicitly
                print(f"Client {tid} disconnected")

        except Exception:
            print(traceback.format_exc())
            to_send = Errors.GENERAL.value  # Fallback error response
            self.network.send_data(sock, tid, to_send.encode())

    def cleaner(self):
        """
        Periodically clean up database entries for ongoing file uploads.
        """
        while True:
            self.cr.clean_db(self.files_uploading)  # Remove old or invalid uploads
            time.sleep(100)  # Wait between cleanup operations

class Client:
    """
    Client class for managing individual client states.
    """
    def __init__(self, id, user, email, subscription_level, admin_level, shared_secret, encryption):
        self.id = id
```

```
        self.user = user
        self.email = email
        self.subscription_level = subscription_level
        self.admin_level = admin_level
        self.shared_secret = shared_secret
        self.encryption = encryption
        self.cwd = f"{CLOUD_PATH}\\{self.user}"

def main(addr):
    """
    Main entry point to initialize and run the server application.
    """
    app = Application(addr)

if __name__ == '__main__':
    sys.stdout = Logger()
    port = 3102
    if len(sys.argv) == 2:
        port = sys.argv[1]
    main(("0.0.0.0", int(port)))
```

```python
import os
import shutil
import subprocess
import zipfile

# Define file and folder names
current_folder = os.path.dirname(os.path.abspath(__file__))
main_folder = os.path.dirname(current_folder)

dist_folder = os.path.join(current_folder, 'dist')
build_folder = os.path.join(current_folder, 'build')
spec_file = os.path.join(current_folder, 'client.spec')
old_exe = os.path.join(current_folder, 'IdanCloud.exe')
client_exe_in_dist = os.path.join(dist_folder, 'client.exe')
new_exe = os.path.join(current_folder, 'IdanCloud.exe')

# Define the folders to be zipped
folders_to_include = {
    "assets": os.path.join(main_folder, 'assets'),
    "gui": os.path.join(main_folder, 'gui')
}
zip_file = os.path.join(current_folder, 'IdanCloud.zip')

# Run PyInstaller to package client.pyw
subprocess.run(['pyinstaller', '--onefile', os.path.join(main_folder, 'client.pyw')])

# Remove old IdanCloud.exe if it exists
if os.path.exists(old_exe):
    os.remove(old_exe)

# Move and rename client.exe
if os.path.exists(client_exe_in_dist):
    shutil.move(client_exe_in_dist, new_exe)

# Delete dist and build folders, and client.spec file
if os.path.exists(dist_folder):
    shutil.rmtree(dist_folder)

if os.path.exists(build_folder):
    shutil.rmtree(build_folder)

if os.path.exists(spec_file):
    os.remove(spec_file)

# Delete old IdanCloud.zip if it exists
if os.path.exists(zip_file):
    os.remove(zip_file)

# Create a new zip file containing IdanCloud.exe, assets, and gui folders
with zipfile.ZipFile(zip_file, 'w') as zipf:
    # Add IdanCloud.exe to the zip file
    if os.path.exists(new_exe):
        zipf.write(new_exe, os.path.basename(new_exe))

    # Add the 'assets' and 'gui' folders to the zip file
    for folder_name, folder_path in folders_to_include.items():
        if os.path.exists(folder_path):
            for root, dirs, files in os.walk(folder_path):
                for file in files:
                    file_path = os.path.join(root, file)
                    # Calculate arcname to preserve folder structure relative to its parent directory
                    arcname = os.path.join(folder_name, os.path.relpath(file_path, folder_path))
                    zipf.write(file_path, arcname)

if os.path.exists(new_exe):
    os.remove(new_exe)

print("Process complete! New IdanCloud.zip created.")
```

```
# 2024 Â© Idan Hazay

import re

# Begin validation checking related functions
class Validation:
    """
    Provides validation methods for email, username, password, and input strings.
    """
    def __init__(self):
        self.illegal_chars = {'\'', '"', '>', '<', '~', '`', '|', '\\','/', '}', '{', '[', ']', '+', '=', ';', '(', ')'}
# Set of illegal characters

    @staticmethod
    def is_valid_email(email):
        """
        Validate an email address using a regular expression.
        """
        email_regex = r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$'
        return re.match(email_regex, email) is not None

    @staticmethod
    def is_valid_username(username):
        """
        Validate a username ensuring it is at least 4 characters long and alphanumeric.
        """
        return len(username) >= 4 and username.isalnum()

    @staticmethod
    def is_valid_password(password):
        """
        Validate a password ensuring it is at least 8 characters long, contains uppercase letters, and numbers.
        """
        return len(password) >= 8 and any(char.isupper() for char in password) and any(char.isdigit() for char in
password)

    @staticmethod
    def is_empty(list):
        """
        Check if any string in a list is empty.
        """
        return any(item == "" for item in list)

    def has_illegal_chars(self, input_str):
        """
        Check if a string contains any illegal characters.
        """
        return any(char in self.illegal_chars for char in input_str)

    def check_illegal_chars(self, string_list):
        """
        Check if any string in a list contains illegal characters.
        """
        return any(self.has_illegal_chars(s) for s in string_list)
```