FileSpace

מערכת לשיתוף קבצים

אורי עומרי כהן

ת.ז: 326090750

שם המורה: ניר דוויק

תאריך הגשה: 1/6/2023

שם החלופה: הגנת סייבר ומערכות הפעלה

בית ספר: ויצ"ו הדסים



תוכן עניינים

[1. מבוא 5](#_Toc136827096)

[1.1 תיאור מערכת ראשוני 5](#_Toc136827097)

[1.2 הסיבות לבחירת הפרויקט 5](#_Toc136827098)

[1.3 הגדרת הלקוח 5](#_Toc136827099)

[1.4 הגדרת יעדים 5](#_Toc136827100)

[1.5 בעיות שהפרויקט פותר 5](#_Toc136827101)

[1.6 פתרונות קיימים 6](#_Toc136827102)

[1.7 תיאור מפורט של המערכת ויכולות המשתמש 6](#_Toc136827103)

[1.8 ניהול סיכונים 6](#_Toc136827104)

[1.9 פירוט היכולות 6](#_Toc136827105)

[1.9.1 הרשמה למערכת 6](#_Toc136827106)

[1.9.2 התחברות למערכת 7](#_Toc136827107)

[1.9.3 הצגת התיקייה 7](#_Toc136827108)

[1.9.4 עדכון מסך חברים 8](#_Toc136827109)

[1.9.5 אתחול מסך שיתופים 8](#_Toc136827110)

[1.9.6 פעולה על פריט 8](#_Toc136827111)

[2. מבנה/ ארכיטקטורת הפרויקט 10](#_Toc136827112)

[2.1 תיאור הארכיטקטורה 10](#_Toc136827113)

[2.1.1 תיאור מילולי 10](#_Toc136827114)

[2.1.2 שרטוט אינטראקציה של המשתמש עם ממשק המשתמש 10](#_Toc136827115)

[2.1.3 שרטוט פעולות שקורות ברקע 10](#_Toc136827116)

[2.2 תיאור הטכנולוגיה 10](#_Toc136827117)

[2.3 אלגוריתמים מרכזיים 11](#_Toc136827118)

[2.3.1 אלגוריתם אימות משתמש 11](#_Toc136827119)

[2.3.2 אלגוריתם סנכרון התיקיות 11](#_Toc136827120)

[2.3.3 אלגוריתם שליחת בקשת חברות 11](#_Toc136827121)

[2.3.4 אישור בקשת חברות 11](#_Toc136827122)

[2.3.5 אלגוריתם ניהול הרשאות 11](#_Toc136827123)

[2.3.6 אלגוריתם ריענון 11](#_Toc136827124)

[2.5 תיאור סביבת הפיתוח 11](#_Toc136827125)

[2.6 תיאור פרוטוקול התקשורת 12](#_Toc136827126)

[2.7 תיאור מסכי המערכת 12](#_Toc136827127)

[LoginWindow 2.7.1 12](#_Toc136827128)

[SignupWindow 2.7.2 13](#_Toc136827129)

[MainWindow 2.7.3 14](#_Toc136827130)

[Screen Flow Diagram 2.7.4 16](#_Toc136827131)

[2.8 מבני הנתונים 16](#_Toc136827132)

[2.9 סקירת חולשות ואיומים 17](#_Toc136827133)

[3 מימוש הפרויקט 18](#_Toc136827134)

[3.1 מודולים מיובאים 18](#_Toc136827135)

[3.2 מחלקות שיצרתי 18](#_Toc136827136)

[File 3.2.1 18](#_Toc136827137)

[Folder 3.2.2 18](#_Toc136827138)

[LoginWindow 3.2.3 19](#_Toc136827139)

[SignupWindow 3.2.4 19](#_Toc136827140)

[MainWindow 3.2.5 20](#_Toc136827141)

[ClientThread 3.2.6 23](#_Toc136827142)

[3.3 קטעי קוד של האלגוריתמים המרכזיים 24](#_Toc136827143)

[3.3.1 קוד אימות משתמש 24](#_Toc136827144)

[3.3.2 קוד סנכרון התיקיות 25](#_Toc136827145)

[3.3.3 קוד ניהול הרשאות 27](#_Toc136827146)

[3.3.4 קוד ריענון 30](#_Toc136827147)

[4 מדריך למשתמש 32](#_Toc136827148)

[4.1 קבצי הפרויקט 32](#_Toc136827149)

[4.2 התקנת המערכת 33](#_Toc136827150)

[4.3 משתמשי המערכת 33](#_Toc136827151)

[5 רפלקציה 36](#_Toc136827152)

[6 קוד הפרויקט 36](#_Toc136827153)

[server.py 6.1 36](#_Toc136827154)

[client\_thread.py 6.2 38](#_Toc136827155)

[file\_classes.py 6.3 51](#_Toc136827156)

[client.py 6.4 54](#_Toc136827157)

[login\_window.py (Auto-Generated by Qt Designer) 6.5 85](#_Toc136827158)

[signup\_window.py (Auto-Generated by Qt Designer) 6.6 87](#_Toc136827159)

[main\_window.py (Auto-Generated by Qt Designer) 6.7 89](#_Toc136827160)

# 1. מבוא

## 1.1 תיאור מערכת ראשוני

מערכת שיתוף הקבצים היא פרויקט שמטרתו לפתח פלטפורמה ידידותית למשתמש לשיתוף קבצים בין המשתמשים. המוצר המוגמר יספק למשתמשים את היכולת להעלות, ליצור, לערוך ולנהל את הקבצים והתיקיות שלהם. למשתמשים תהיה גם היכולת לשתף את הקבצים והתיקיות שלהם זה עם זה, ולקבוע רמות שונות של הרשאות גישה.

## 1.2 הסיבות לבחירת הפרויקט

בחרתי לפתח מערכת שיתוף קבצים מכיוון שאני מכיר מוצרים אחרים לשיתוף קבצים ועניין אותי ליצור מוצר דומה וקל לשימוש. בנוסף, רציתי לפתח מוצר שאוכל להשתמש בו ביום יום ולהוסיף לו פונקציונליות ולשפר אותו גם אחרי הגשת הפרויקט.

## 1.3 הגדרת הלקוח

המערכת מיועדת לאנשים פרטיים, תלמידים, אנשי מקצוע וצוותים הזקוקים לאמצעי יעיל ומאובטח לשיתוף קבצים ושיתוף פעולה בפרויקטים. משתמשי היעד יכולים לנוע מתלמידים העובדים על משימות קבוצתיות ועד לאנשי מקצוע העוסקים בעבודת צוות מרחוק או אנשים שרוצים פלטפורמה מרכזית לניהול הקבצים האישיים שלהם.

## 1.4 הגדרת יעדים

שיתוף פעולה בין משתמשים: המערכת תאפשר שיתוף פעולה יעיל על ידי מתן פלטפורמה למשתמשים לשיתוף קבצים, ולעבוד יחד על פרויקטים.

ניהול קבצים משופר: המערכת תציע ממשק ידידותי למשתמש לארגון וניהול קבצים ותיקיות.

שליטה בהרשאות הגישה: למשתמשים תהיה היכולת להגדיר הרשאות גישה לקבצים המשותפים שלהם, מה שיאפשר להם להעניק גישה לקריאה וכתיבה או גישה לקריאה בלבד לחבריהם, מה שמבטיח אבטחת מידע ופרטיות.

## 1.5 בעיות שהפרויקט פותר

מערכת שיתוף הקבצים פותרת את הבעיות הבאות:

* בעיה: יכולות שיתוף פעולה מוגבלות ושיטות שיתוף קבצים לא יעילות.  
  פתרון: המערכת מספקת פלטפורמה מרכזית המייעלת את שיתוף הקבצים, המובילה להגברת שיתוף הפעולה בין משתמשיה.
* בעיה: סיכוני אבטחת נתונים וגישה לא מורשית לקבצים משותפים.  
  פתרון: המערכת משלבת הצפנה, אימות ובקרת גישה, המבטיחים שלמות נתונים וסודיות.
* בעיה: חוסר עדכונים בזמן אמת ונראות לתיקיות משותפות.  
  פתרון: המערכת מאפשרת למשתמשים לקבל התראות ועדכונים בזמן אמת על שינויים שבוצעו על ידי חבריהם, משפרת את שיתוף הפעולה ומעדכנת את המשתמשים.

## 1.6 פתרונות קיימים

[*Dropbox*](https://www.dropbox.com/official-site?_tk=paid_sem_goog_biz_b&_camp=1027825805&_kw=dropbox%7Ce&_ad=630757798489%7C%7Cc&gclid=Cj0KCQiA2-2eBhClARIsAGLQ2Rl04mIGGtSvOCkfZ3cHwBG73dZtj9A0uV3Z7QrSa8PAJjxZJw0vDQsaAszqEALw_wcB)*: מציעה אחסון מאובטח, שיתוף ועריכה של כמה משתמשים על אותו קובץ בו זמנית.*

[*Google Drive*](https://www.google.com/intl/iw_il/drive/)*: מציעה אחסון ושיתוף בענן שמשתלבים עם שירותי גוגל אחרים כמו Gmail ו-Google Docs.*

[*Microsoft OneDrive*](https://www.microsoft.com/en-us/microsoft-365/onedrive/online-cloud-storage)*: מציעה אחסון ושיתוף בענן שמשתלבים עם שירותי Microsoft אחרים.*

*לשלושת מוצרים אלה יש יתרונות כמו עריכה של כמה משתמשים בזמן אמת ושילוב באפליקציות אחרות. לחלק מהאנשים מוצרים אלו מסובכים מדי לעומת FileSpace שמציעה שימוש פשוט ויעיל.*

## 1.7 תיאור מפורט של המערכת ויכולות המשתמש

בכניסה למערכת, הלקוח יכול להתחבר למשתמש או ליצור משתמש חדש.

לכל משתמש במערכת יש תיקייה משלו, אליה הוא יכול להעלות, ליצור ולשנות קבצים ותיקיות בנוסף לפעולות נוספות על קובץ או תיקיה: מחיקה, העתקה, גזירה והדבקה. משתמש יכול לחפש משתמש אחר ולשלוח לו בקשת חברות, אם הוא מאשר הם הופכים לחברים.

משתמש יכול לתת הרשאות כתיבה וקריאה או קריאה בלבד לכל אחד מהחברים שלו. כל משתמש רואה את התיקיות של המשתמשים שמשתפים איתו ואת השינויים שקורים בהן.   
אם למשתמש יש הרשאות קריאה וכתיבה הוא יכול לבצע את כל הפעולות על התיקייה של המשתמש המשתף, ואם יש לו הרשאות קריאה בלבד הוא יכול רק לראות את השינויים שקורים בתיקייה.

## 1.8 ניהול סיכונים

* אבטחת התקשורת: תכננתי להשתמש בהצפנה אסימטרית כדי להעביר מידע בין הצדדים. בפועל, בגלל שאי אפשר להעביר הודעות ארוכות בהצפנה אסימטרית, השתמשתי בהצפנה אסימטרית כדי להעביר את מפתח ההצפנה הסימטרי מהשרת ללקוח, ושאר ההודעות מוצפנות ומפוענחות עם המפתח הסימטרי.
* SQL Injection: תכננתי לבצע אימות קלט משתמש איפה שקולטים מחרוזות שייעשה בהן שימוש בשאילתת SQL. בפועל, הגבלתי את התווים שאפשר לכלול בשם המשתמש והסיסמה.
* שמות לא תקינים לקבצים/תיקיות: תכננתי לבצע אימות קלט ולהציג הודעה מתאימה על המסך אם השם לא תקין, וזה מה שביצעתי בפועל.

## 1.9 פירוט היכולות

### 1.9.1 הרשמה למערכת

מהות: רישום משתמש חדש במערכת – קליטת שם משתמש, סיסמה ואימות סיסמה.  
אוסף יכולות נדרשות:

* ממשק משתמש – מסך הרשמה
* קליטת שם משתמש, סיסמה ואימות סיסמה
* בדיקה שהסיסמה ואימות הסיסמה זהים
* הצפנה
* שליחה לשרת
* בדיקה מול טבלת המשתמשים בשרת שאין עוד משתמש עם אותו השם
* קבלת תשובה מהשרת
* פענוח
* הצגת התשובה בממשק משתמש

אובייקטים נחוצים: ממשק משתמש, הצפנה/פענוח, תקשורת, בסיס נתונים (טבלת המשתמשים).

### 1.9.2 התחברות למערכת

מהות: התחברות משתמש למערכת – קליטת שם משתמש וסיסמה.  
אוסף יכולות נדרשות:

* ממשק משתמש – מסך התחברות
* קליטת שם משתמש וסיסמה
* בדיקה ששם המשתמש והסיסמה לא ריקים
* הצפנה
* שליחה לשרת
* בדיקה ששם המשתמש קיים ושהסיסמה תואמת מול טבלת המשתמשים בשרת
* קבלת תשובה מהשרת
* פענוח
* הצגת התשובה בממשק משתמש

אובייקטים נחוצים: ממשק משתמש, הצפנה/פענוח, תקשורת, בסיס נתונים (טבלת המשתמשים).

### 1.9.3 הצגת התיקייה

מהות: הצגת התיקייה של המשתמש לאחר התחברות.  
אוסף יכולות נדרשות:

* בדיקה שההתחברות צלחה
* הצפנה
* שליחת בקשה לשרת
* בדיקה בשרת אם קיימת תיקייה
* יצירת תיקייה ריקה בשרת אם לא קיימת תיקייה למשתמש
* יצירת אובייקט תיקייה
* סריאליזציה לאובייקט
* קבלת תשובה מהשרת
* פענוח
* דיסריאליזציה לאובייקט
* יצירת תיקייה בלקוח
* הצגת מיקום התיקייה בממשק המשתמש

אובייקטים נחוצים: ממשק משתמש, הצפנה/פענוח, תקשורת, אובייקט תיקייה עם פעולת יצירת התיקייה.

### 1.9.4 עדכון מסך חברים

מהות: הצגת הרשימות המעודכנות של החברים, בקשות החברות וחברים שהמשתמש משתף איתם.  
אוסף יכולות נדרשות:

* הצפנה
* שליחת בקשה לשרת
* אחזור נתונים מהטבלאות מבסיס הנתונים בשרת
* קבלת תשובה מהשרת
* פענוח
* פיצול לרשימות נפרדות
* הצגת הרשימות בממשק המשתמש

אובייקטים נחוצים: ממשק משתמש, הצפנה/פענוח, תקשורת, בסיס נתונים.

### 1.9.5 אתחול מסך שיתופים

מהות: הצגת התיקיות של המשתמשים שמשתפים עם המשתמש.  
אוסף יכולות נדרשות:

* הצפנה
* שליחת בקשה לשרת
* אחזור המשתמשים שמשתפים עם שולח הבקשה מבסיס הנתונים בשרת.
* יצירת אובייקט התיקייה של כל משתף
* סריאליזציה לאובייקט
* קבלת מספר המשתפים
* קבלת תשובה מהשרת לכל משתמש משתף
* פענוח
* דיסריאליזציה לאובייקט של כל משתמש
* בדיקת ההרשאה לתיקייה
* יצירת התיקייה במיקום התואם להרשאה
* הצגת התיקיות בממשק המשתמש

אובייקטים נחוצים: ממשק משתמש, הצפנה/פענוח, תקשורת, בסיס נתונים, אובייקט תיקייה עם פעולת יצירת התיקייה.

### 1.9.6 פעולה על פריט

מהות: פעולה על קובץ או תיקייה ממחשב הלקוח והשרת.  
איסוף יכולות נדרשות:

* בחירת הפעולה על פריט בממשק המשתמש
* ביצוע הפעולה על הפריט
* עדכון המסך בממשק המשתמש
* יצירת הנתיב/ים החל משם התיקייה
* הצפנה
* שליחת שם הפעולה, והנתיבי/ים
* פענוח
* נעילת התהליך
* ביצוע הפעולה בתיקייה בשרת
* שחרור התהליך

אובייקטים נחוצים: ממשק משתמש, הצפנה/פענוח, תקשורת, בסיס נתונים (טבלת המשתמשים).

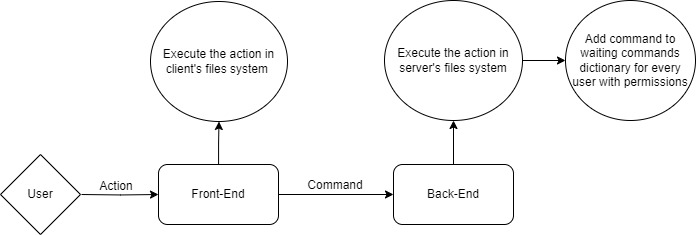
# 2. מבנה/ ארכיטקטורת הפרויקט

## 2.1 תיאור הארכיטקטורה

### 2.1.1 תיאור מילולי

הפרויקט מחולק לשני חלקים: Front-End (ממשק משתמש) וBack-End (צד השרת). ה-Front-End אחראי על האינטראקציה של המשתמש עם התוכנה ועם מערכת הקבצים במחשב שלו. זה כולל משימות כמו הצגת ממשק המשתמש, קלט משתמש וניהול פעולות על קבצים ישירות על הלקוח. ה-Front-End גם משתמש בתהליכים שרצים ברקע המבצעים משימות כמו ריענון והצגת חברים, בקשות חברות, הרשאות שיתוף וקבלת שינויים בתיקיות משותפות. ה-Back-End אחראי על השגת הנתונים הנחוצים (למשל רשימת החברים, תיקיית המשתמש, רשימת בקשות חברות, שינויים בתיקיות משותפות וכו').

### 2.1.2 שרטוט אינטראקציה של המשתמש עם ממשק המשתמש



### 2.1.3 שרטוט פעולות שקורות ברקע

A diagram of a data flow

Description automatically generated with low confidence

## 2.2 תיאור הטכנולוגיה

* הפרויקט נכתב בשפה Python.
* המערכת עובדת במערכת הפעלה Windows וכדי שהיא תעבוד במערכות הפעלה נוספות, יתכן שיהיה צורך בשינויים בחלק מהקוד.
* נעשה שימוש בפרוטוקול התקשורת TCP מכיוון שהוא מבטיח שכל המידע יגיע מצד לצד.
* נעשה שימוש בהצפנות RSA ו-Fernet.
* נעשה שימוש בתהליכים בשביל לאפשר תקשורת מרובת לקוחות וביצוע פעולות ברקע.

## 2.3 אלגוריתמים מרכזיים

### 2.3.1 אלגוריתם אימות משתמש

אלגוריתם זה מטפל בתהליך ההרשמה וההתחברות של משתמשים למערכת. הוא כולל את השלבים: קליטת פרטי משתמש, אימותם מול ה-Database ומתן גישה לאחר אימות מוצלח. הוא כולל הצפנה של המידע העובר בתקשורת והצפנת הסיסמה.

### 2.3.2 אלגוריתם סנכרון התיקיות

אלגוריתם זה מנהל את הסנכרון של התיקיות בין משתמשים. הוא כולל את השלבים: זיהוי פעולה בתיקייה, העברת המידע לשרת, ביצוע הפעולה במערכת הקבצים בשרת ושליחת הפקודה לביצוע הפעולה במשתמשים המחוברים שיש להם הרשאות לתיקייה.

### 2.3.3 אלגוריתם שליחת בקשת חברות

אלגוריתם זה מטפל בשליחה, וקבלה של בקשות חברות בין משתמשים. הוא כולל שלבים לחיפוש משתמשים אחרים, שליחת בקשות חברות והוספת בקשת החברות ל-Database.

### 2.3.4 אישור בקשת חברות

אלגוריתם זה מטפל באישור/דחיה של בקשת חברות.

### 2.3.5 אלגוריתם ניהול הרשאות

אלגוריתם זה מנהל את ההקצאה והאכיפה של הרשאות קריאה וכתיבה בין משתמשים. הוא כולל שלבים להענקה או ביטול של הרשאות, אימות זכויות גישה למשתמש והגבלת פעולות על סמך רמות הרשאה.

### 2.3.6 אלגוריתם ריענון

אלגוריתם זה מעדכן את נתוני צד הלקוח בהתבסס על המידע שהתקבל מהשרת.  
הוא כולל שלבים לאחזור ועיבוד רשימות משתמשים, חברים, בקשות חברות ושיתופים.  
הוא כולל אחזור נתונים מה-Database, שליחת הנתונים ללקוח ועדכון ממשק המשתמש עם הנתונים החדשים.

### 

## 2.5 תיאור סביבת הפיתוח

הפרויקט פותח ב-PyCharm. במהלך העבודה על הפרויקט השתמשתי בGitHub, שמרתי את הקבצים ב-master branch בrepository שיצרתי בשביל הפרויקט. מתי שסיימתי לכתוב קטעי קוד חשובים או לפני שעברתי לעבוד על הפרויקט ממחשב אחר, עשיתי לקבצים ששיניתי commit ו-push.

## 2.6 תיאור פרוטוקול התקשורת

מבנה ההודעה מורכב משני חלקים: אורך ההודעה המוצפנת המוצפן וההודעה המוצפנת.  
האורך המוצפן תמיד יהיה בגודל 100 בתים מכיוון שבהצפנת Fernet, כאשר מצפינים עד 15 בתים, גודלם המוצפן יהיה 100 בתים, ואורך ההודעה המוצפנת אף פעם לא יהיה מספר בעל יותר מ-15 ספרות.

תהליך קריאת ההודעה: קוראים את 100 הבתים הראשונים, מפענחים אותם ומקבלים את אורך ההודעה המוצפנת. לאחר מכן, מקבלים בלוקים בגודל 4096 בתים של ההודעה המוצפנת בלולאה ומחברים אותם עד שמספר הבתים שהתקבלו שווה לאורך ההודעה המוצפנת. אחרי שהלולאה מסתיימת, מפענחים את כל הבתים שהתקבלו ומקבלים את ההודעה המקורית. A picture containing diagram, text, plan, line

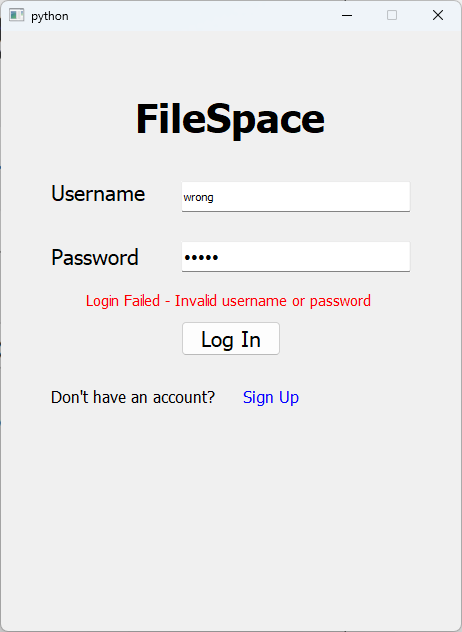
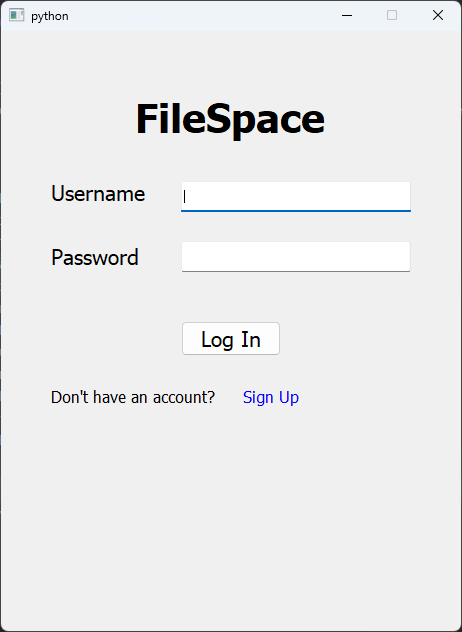
Description automatically generated

## 2.7 תיאור מסכי המערכת

### LoginWindow 2.7.1

אחראי לקלוט שם משתמש וסיסמה, להציג הודעה מתאימה אם יש בעיה בהתחברות, לנתב למסך ההרשמה אם המשתמש לוחץ על "Sign Up" או לנתב למסך הראשי אם ההתחברות צלחה.

המסך ההתחלתי:

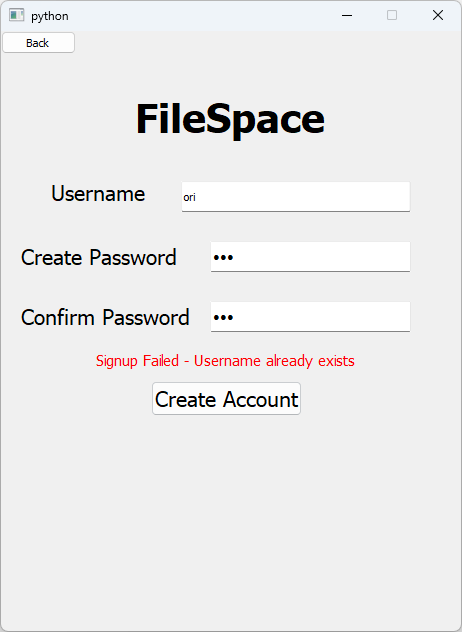


המסך לאחר הכנסת פרטים שגויים:

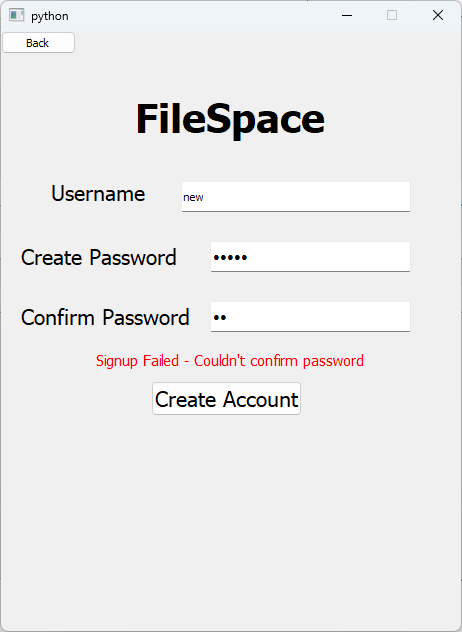
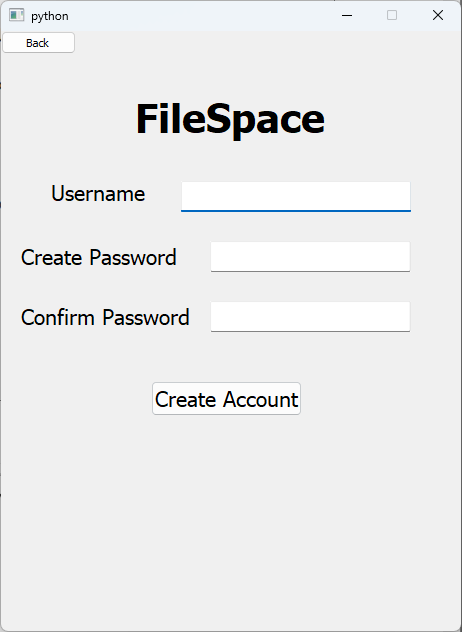
### SignupWindow 2.7.2

אחראי לקלוט שם משתמש, סיסמה ואימות סיסמה, להציג הודעה מתאימה אם יש בעיה בהרשמה, לנתב חזרה למסך ההתחברות אם המשתמש לוחץ על הכפתור "Back" או לנתב למסך הראשי אם ההרשמה צלחה.

לאחר הכנסת שם משתמש קיים:



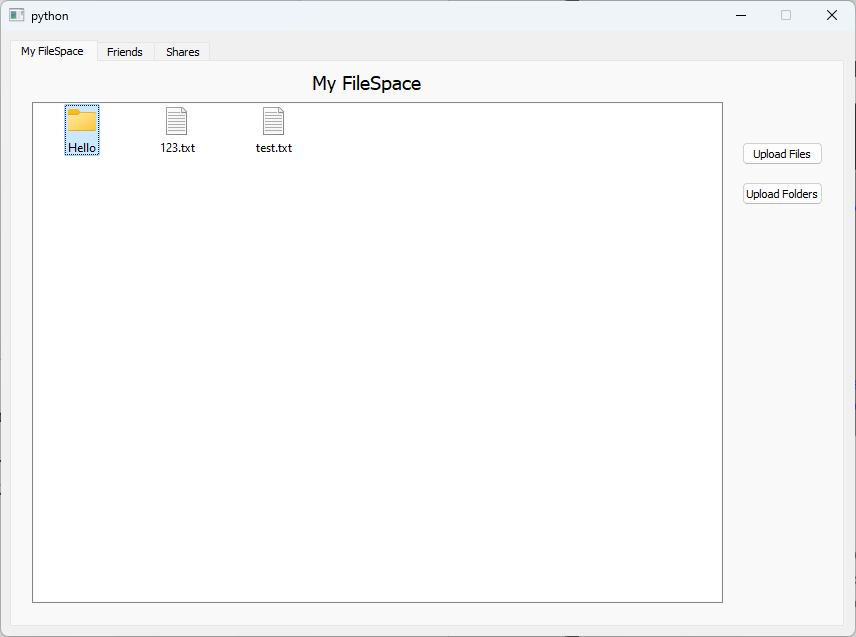
מסך ההרשמה ההתחלתי:



לאחר הכנסת שתי סיסמאות שונות:

### MainWindow 2.7.3

* My FileSpace Tab: כרטיסייה שמציגה את תכולת תיקיית המשתמש. המשתמש יכול לבצע פעולות על הפריטים באמצעות קליק ימני על הפריט ובחירת הפעולה, להיכנס לתיקיה באמצעות דאבל קליק עליה, לפתוח קובץ באמצעות דאבל קליק עליו או בחירת קובץ או תיקייה להעלות בלחיצה על הכפתורים בצד ימין. אחרי שנכנסים לתיקייה אפשר לצאת ממנה באמצעות כפתור "Back" שמופיע.

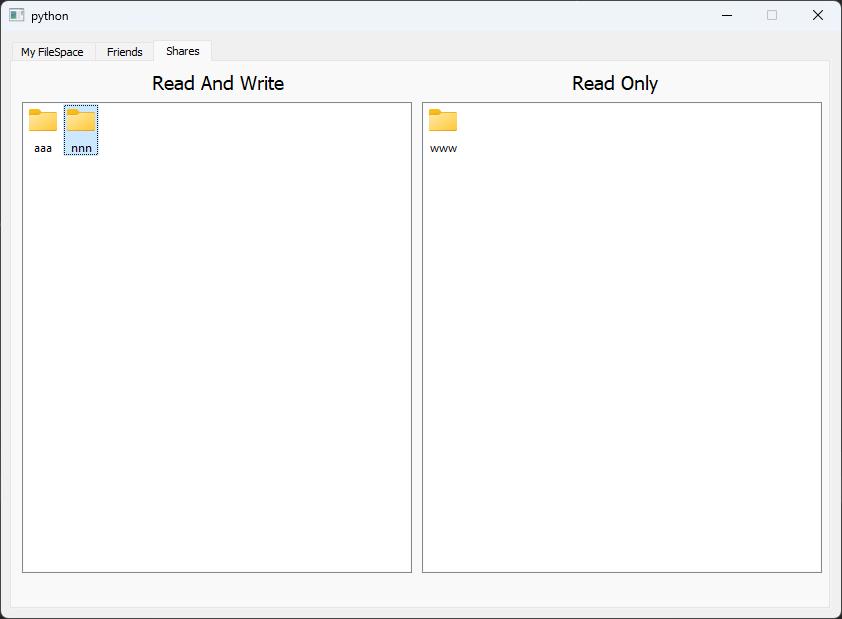


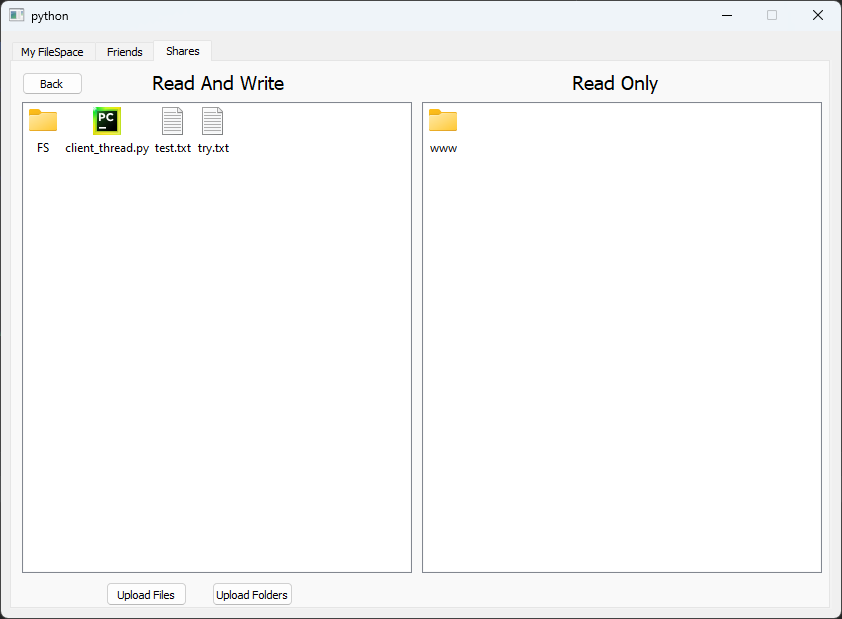
* Friend Tab: כרטיסייה שמאפשרת לחפש משתמשים ולשלוח להם בקשת חברות, מציגה את בקשות החברות ששלחו למשתמש, חברי המשתמש, חברים שהוא משתף איתם עם הרשאות קריאה וכתיבה ו חברים שהוא משתף איתם עם הרשאות קריאה בלבד. המשתמש יכול גם לתת/לשנות/להסיר הרשאות לחברים ולהסיר חברים.

A screenshot of a computer

Description automatically generated

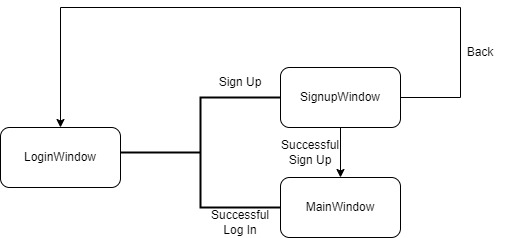
* Shares Tab: כרטיסייה שמציגה את התיקיות שמשתפים עם המשתמש (מחולקות לפי ההרשאות). המשתמש יכול לבצע על הפריטים שבתוך התיקיות של המשתמשים שיש לו הרשאות קריאה וכתיבה אליהן את כל הפעולות שהוא יכול לבצע על התיקייה שלו. בתיקיות שיש לו אליהן הרשאות קריאה הוא רק יכול לראות את השינויים שמתבצעים.





בתוך תיקייה של משתמש שמשתף קריאה וכתיבה:

### Screen Flow Diagram 2.7.4



## 2.8 מבני הנתונים

* Database: לצורך שמירת נתוני היחסים בין המשתמשים השתמשתי בטבלאות בבסיס נתונים MySQL. שם המסד: FileSpace
* טבלת users:
  + username, VARCHAR(255). דוגמה: "User1"
  + password, VARCHAR(255): הסיסמא המוצפנת. דוגמה: "202cb962ac59075b964b07152d234b70"
  + friends, VARCHAR(4095): comma separated string של החברים של המשתמש. דוגמה: "user1,user2,user3"
  + friend\_requests, VARCHAR(4095): comma separated string של המשתמשים ששלחו בקשת חברות למשתמש. דוגמה: "user4,user5,user6"
* טבלת users\_sharing:
  + sharing\_user, VARCHAR(255): המשתמש המשתף. דוגמה: "user1"
  + shared\_user, VARCHAR(255): המשתמש שאליו משתפים. דוגמה: "user2"
  + permission, VARCHAR(255): ההרשאה שהמשתף נתן למשותף. (read/read\_write)
* מערכת הקבצים:
* תיקיית ServerFolder מאחסנת תיקיות בשמות המשתמשים ובהן השרת מבצע שינויים לפריטיהן.
* תיקיית FS בלקוח:
  + תיקיית Folders: מאחסנת תיקיות בשמות המשתמשים שהתחברו באותו המחשב. תיקיות אלו הן התיקיות הראשיות של כל משתמש.
  + תיקיית Read And Write: מאחסנת תיקיות בשמות המשתמשים ובהן התיקיות של המשתמשים שמשתפים עם המשתמש עם הרשאות כתיבה וקריאה.
  + תיקיית Read Only: מאחסנת תיקיות בשמות המשתמשים ובהן התיקיות של המשתמשים שמשתפים עם המשתמש עם הרשאות כתיבה בלבד.
* waiting\_commands dictionary:
* מפתח: שם המשתמש שאליו נשלחת הפעולה.
* ערך: מחרוזת – הפעולה / tuple – הפעולה ומידע של קובץ/תיקייה.

## 2.9 סקירת חולשות ואיומים

* SQL Injection: הגבלתי את התווים שאפשר לכלול בשם המשתמש והסיסמה כך שמשתמש לא יוכל לפגוע בבסיס הנתונים.
* תהליך ההתחברות: קיים אימות מול בסיס הנתונים.
* הצפנה: בתחילת הקשר, השרת והלקוח מייצרים מפתח פרטי ומפתח ציבורי בהצפנת RSA. הם שולחים אחד לשני את המפתח הציבורי שלהם, והשרת שולח מפתח הצפנה Fernet סימטרי ללקוח ובו יעשה שימוש בכל פעולות התקשורת ביניהם.
* העלאת קבצים: התוכן של הקבצים מוצפן בהצפנת Fernet לפני שהם נשמרים בשרת.

# 3 מימוש הפרויקט

## 3.1 מודולים מיובאים

pathlib : מספק פונקציות לעבודה עם נתיבי קבצים וספריות.

cryptography.fernet: מיישם את אלגוריתם ההצפנה של Fernet להצפנת ופענוח נתונים מאובטחים.

rsa: מציע פונקציונליות להצפנת RSA כולל יצירת מפתחות, הצפנה ופענוח.

os: מספק דרך ליצור אינטראקציה עם מערכת ההפעלה, כולל פעולות קבצים וספריות.

shutil: מציע פעולות כמו העתקה, העברה ומחיקת קבצים וספריות.

threading: לאפשר תקשורת מרובת לקוחות וביצוע פעולות ברקע.

pickle.dumps, pickle.loads: סריאליזציה דיסריאליזציה לאובייקטים.

Mysql: מספק ממשק Python לאינטראקציה עם Database של MySQL לצורך שמירת ואחזרת נתונים.

time: מספק מרווחים בין ביצוע הפעולות ברקע.

## 3.2 מחלקות שיצרתי

### File 3.2.1

* תפקיד: מטפל, בפעולות על קבצים.
* תכונות:
* path: הנתיב לקובץ.
* name: שם הקובץ.
* size: גודל הקובץ בבתים.
* rel\_path: הנתיב היחסי של הקובץ (בשימוש במהלך פעולת היצירה).
* Data: הנתונים הבינאריים של הקובץ.
* פעולות:
* create:
* טענת כניסה: parent\_path (אופציונלי) הנתיב ליצירת הקובץ.
* טענת יציאה: יוצרת את הקובץ בנתיב שצוין, מעדכנת אותו אם הוא כבר קיים.

### Folder 3.2.2

* תפקיד: ניהול פעולות על תיקיות ויצירה רקורסיבית של תיקיות ותכניהן.
* תכונות:
* path: הנתיב לתיקייה.
* name: שם התיקייה.
* size: גודל התיקייה בבתים.
* subdirectories: רשימה של אובייקטים Folder משנה בתוך התיקייה.
* files: רשימה של אובייקטים File משנה בתוך התיקייה.
* פעולות:
* create:
* טענת כניסה: parent\_path (אופציונלי) הנתיב ליצירת התיקייה.
* טענת יציאה: יוצרת את התיקייה בנתיב שצוין, מעדכנת אותה אם היא כבר קיימת ומחזירה את אובייקט התיקייה.
* change\_path:
* טענת כניסה: new\_path הנתיב החדש לתיקייה.
* טענת יציאה: משנה את הנתיב של הספרייה, כולל שינוי שם.

### LoginWindow 3.2.3

* תפקיד: מייצרת את חלון ההתחברות של האפליקציה. יורשת מ-QMainWindow ו-Ui\_Login(נוצרה אוטומטית מ-Qt Designer).
* תכונות: עוברות בירושה מ- Ui\_Login
* פעולות:
  + Init:
    - טענת כניסה: self
    - טענת יציאה: מאתחלת את אובייקט ה-LoginWindow, מגדירה את רכיבי ממשק המשתמש, מחברת לחיצות כפתורים לפעולות ומבצעת פעולות התחלתיות על תכונות קיימות.
  + Login:
* טענת כניסה: self
* טענת יציאה: מבצעת את פעולת ההתחברות באמצעות קליטת שם המשתמש והסיסמה משדות הקלט, שליחת בקשת ההתחברות לשרת, קבלת תגובת השרת וטיפול בתגובה בהתאם.
* goto\_signup\_screen:
* טענת כניסה: אין
* מנווטת למסך ההרשמה
* goto\_files:
* טענת כניסה: הנתיב לתיקיית המשתמש
* טענת יציאה: מנווטת למסך האפליקציה הראשי, מעבירה את נתיב תיקיית המשתמש כפרמטר.

### SignupWindow 3.2.4

* תפקיד: מייצרת את חלון ההרשמה של האפליקציה. יורשת מ-QMainWindow ו- Ui\_Signup(נוצרה אוטומטית מ-Qt Designer).
* תכונות: עוברות בירושה מ- Ui\_Signup
* פעולות:
  + Init:
    - טענת כניסה: self
    - טענת יציאה: מאתחלת את אובייקט ה-SignupWindow, מגדירה את רכיבי ממשק המשתמש, מחברת לחיצות כפתורים לפעולות ומבצעת פעולות התחלתיות על תכונות קיימות.
  + signup:
* טענת כניסה: self
* טענת יציאה: מבצעת את פעולת ההרשמה באמצעות קליטת שם המשתמש, סיסמה ואימות סיסמה משדות הקלט, שליחת בקשת ההרשמה לשרת, קבלת תגובת השרת וטיפול בתגובה בהתאם.
* go\_back:
* טענת כניסה: אין.
* טענת יציאה: מנווטת חזרה למסך ההתחברות.
* goto\_files:
* טענת כניסה: הנתיב לתיקיית המשתמש.
* טענת יציאה: מנווטת למסך האפליקציה הראשי, מעבירה את נתיב תיקיית המשתמש כפרמטר.

### MainWindow 3.2.5

* תפקיד: מייצרת את חלון האפליקציה. יורשת מ-QWidget ו- Ui\_MainWindow(נוצרה אוטומטית מ-Qt Designer). מספקת ממשק משתמש לניהול מערכת הקבצים והשיתופים. מעדכנת את השרת על פעולות המשתמש ומקבלת עדכונים מהשרת על שינויים בנתונים.
* תכונות:
* lock: אובייקט של threading. Lock המשמש לסנכרון התהליכים.
* exit: דגל בוליאני המציין אם האפליקציה צריכה לצאת או לא.
* copied\_item\_path: נתיב של הפריט המועתק.
* cut\_item\_path: נתיב של הפריט הנחתך.
* dir\_path: נתיב של תיקיית המשתמש.
* username: שם משתמש המשויך לספרייה.
* read\_write\_path: נתיב של תיקיית הקריאה-כתיבה.
* read\_only\_path: נתיב של תיקיית הקריאה בלבד.
* file\_timestamps: מילון המאחסן חותמות זמן של קבצים.
* directory\_history: רשימה לאחסון היסטוריית ניווט בתיקיית המשתמש.
* read\_write\_directory\_history: רשימה לאחסון היסטוריית ניווט בתיקיית קריאה-כתיבה.
* read\_only\_directory\_history: רשימה לאחסון היסטוריית ניווט בתיקיית קריאה בלבד.
* users: רשימת משתמשים.
* friends: רשימת חברים.
* friend\_requests: רשימת בקשות חברות.
* sharing\_read\_only: רשימה של חברים משותפים לקריאה בלבד.
* sharing\_read\_write: רשימה של חברים משותפים לקריאה-כתיבה.
* shared\_read\_only: רשימת משתמשים ששיתפו את המשתמש עם הרשאת קריאה בלבד.
* shared\_read\_write: רשימת משתמשים ששיתפו את המשתמש עם הרשאות קריאה-כתיבה.
* model: אובייקט QFileSystemModel המייצג את מודל מערכת הקבצים.
* read\_only\_model: אובייקט QFileSystemModel המייצג את מודל מערכת הקבצים לקריאה בלבד.
* read\_write\_model: אובייקט QFileSystemModel המייצג את מודל מערכת הקבצים קריאה-כתיבה.
* watcher: אובייקט QFileSystemWatcher לניטור שינויים בתיקיית המשתמש.
* read\_write\_watcher: אובייקט QFileSystemWatcher לניטור שינויים בתיקיות קריאה-כתיבה.
* פעולות:
* friend\_double\_clicked:
* טענת כניסה: מקבלת פריט נבחר המייצג חבר.
* טענת יציאה: מטפלת בפעולה כאשר חבר נלחץ פעמיים בווידג'ט רשימת החברים ומבצע את הפעולה המתאימה בהתאם לבחירת המשתמש ומעדכנת את השרת.
* share\_friend:
* טענת כניסה: מקבלת שם של חבר לחלוק איתו.
* טענת יציאה: שולחת פקודה לשרת לשתף את תיקיית המשתמש עם ההרשאות שנבחרו לחבר שצוין.
* change\_permission:
  + טענת כניסה: מקבלת את שם המשתמש של חבר ואת ההרשאות הנוכחיות שלו.
  + טענת יציאה: משנה את הרשאות השיתוף של החבר שצוין ומעדכנת את ממשק המשתמש והשרת בהתאם.
* sharing\_to\_double\_clicked:
  + טענת כניסה: מקבלת פריט נבחר המייצג משתמש בווידג'ט באחת מרשימות השיתוף.
  + טענת יציאה: מטפלת בפעולה כאשר משתמש נלחץ פעמיים בווידג'ט רשימת השיתוף ומעדכנת את השרת. (שינוי או הסרת הרשאות)
* remove\_friend:
* טענת כניסה: מקבלת שם של חבר להסרה.
* טענת יציאה: מסירה את החבר שצוין מרשימת החברים ומהווידג'ט של רשימת החברים ומעדכנת את השרת.
* send\_friend\_request:
* טענת כניסה: מקבלת שם של משתמש לשלוח אליו בקשת חברות.
* טענת יציאה: שולחת בקשת חברות למשתמש שצוין ומחזיר את התגובה מהשרת (אישור/כבר נשלח/כבר חבר).
* user\_double\_clicked:
* טענת כניסה: מקבלת פריט נבחר המייצג משתמש בווידג'ט רשימת המשתמשים.
* טענת יציאה: פותחת תיבת הודעה לאישור שליחת בקשת חברות למשתמש הנבחר ושולחת את הבקשה אם היא מאושרת.
* add\_friend:
* טענת כניסה: שם המשתמש להוספה כחבר.
* טענת יציאה: מוסיפה את המשתמש שצוין לרשימת החברים, מעדכנת את הווידג'ט של רשימת החברים על ידי ניקוי והוספה של כל החברים ושולחת פקודה לשרת להוספת החבר.
* remove\_friend\_request:
* טענת כניסה: שם המשתמש שאת בקשת החברות שלו מסירים.
* טענת יציאה: מסירה את שם המשתמש מרשימת בקשות החברים, מעדכנת את הווידג'ט של רשימת בקשות החברים על ידי הסרת שם המשתמש, או מוסיפה פריט ברירת מחדל אם אין בקשות אחרות בווידג'ט. שולחת פקודה לשרת להסיר את בקשת החברות.
* friend\_request\_double\_clicked:
* טענת כניסה: הפריט הנבחר המייצג את בקשת החברות.
* טענת יציאה: מציגה תיבת הודעה ששואלת את המשתמש אם הוא רוצה להוסיף את המשתמש שנבחר כחבר. אם המשתמש בחר להוסיף או לסרב, הבקשה נמחקת, אם הוא בחר להוסיף, המשתמש מתווסף לחברים.
* search\_users:
* טענת כניסה: הטקסט שיש לחפש בשמות מרשימת המשתמשים.
* טענת יציאה: אם טקסט החיפוש ריק, מנקה את הווידג'ט של רשימת תוצאות החיפוש. אחרת, מסננת את רשימת המשתמשים על סמך טקסט החיפוש ומציגה את התוצאות התואמות בווידג'ט רשימת תוצאות החיפוש.
* recursively\_add\_paths:
* טענת כניסה: הנתיב של התיקיה שממנה יש להוסיף את נתיבי קבצים באופן רקורסיבי.
* טענת יציאה: נתיבי הקבצים בתוך נתיב התיקיה שצוין מתווספים באופן רקורסיבי לצופה הקבצים המתאים (או self.watcher או self.read\_write\_watcher), וחותמות הזמן שלהם מאוחסנות במילון file\_timestamps.
* file\_changed:
* טענת כניסה: נתיב הקובץ שנעשה בו שינוי.
* טענת יציאה: אם הקובץ בנתיב שצוין עדיין קיים, חותמת הזמן הנוכחית של הקובץ מושווה עם חותמת הזמן המאוחסנת קודם לכן. אם חותמות הזמן שונות, מה שמציין שהקובץ נערך, נתיב הקובץ והנתונים שלו נשלחים לשרת.
* on\_list\_view\_double\_clicked:
* טענת כניסה: האינדקס של הפריט שנלחץ פעמיים בתיקיית המשתמש.
* טענת יציאה: אם האינדקס מייצג תיקייה, מעדכנת את תצוגת המודל ותצוגת הרשימה כדי להציג את תוכן התיקייה שנלחצה פעמיים. אם האינדקס מייצג קובץ, פותחת את הקובץ.
* on\_read\_write\_list\_view\_double\_clicked:
* טענת כניסה: האינדקס של הפריט שנלחץ פעמיים בתיקיית שיש למשתמש הרשאות קריאה וכתיבה אליה.
* טענת יציאה: אם האינדקס מייצג תיקייה, מעדכנת את תצוגת מודל הקריאה-כתיבה ותצוגת הרשימה כדי להציג את תוכן התיקייה שנלחצה פעמיים. אם האינדקס מייצג קובץ, פותחת את הקובץ.
* on\_read\_only\_list\_view\_double\_clicked:
* טענת כניסה: האינדקס של הפריט שנלחץ פעמיים בתיקיית שיש למשתמש הרשאות קריאה וכתיבה אליה.
* טענת יציאה: אם האינדקס מייצג תיקייה, מעדכנת את תצוגת מודל הקריאה-כתיבה ותצוגת הרשימה כדי להציג את תוכן התיקייה שנלחצה פעמיים. אם האינדקס מייצג קובץ, פותחת את הקובץ.
* create\_context\_menu:
* טענת כניסה: המיקום שבו יש ליצור את תפריט ההקשר. תצוגת הרשימה שבה נוצר תפריט ההקשר.
* טענת יציאה: יוצרת תפריט קליק ימני במיקום שצוין בתצוגת הרשימה הנתונה.
* copy\_item/ cut\_item:
* טענת כניסה: הנתיב של הפריט שנבחר.
* טענת יציאה: שומרת את הנתיב של הפריט לגזירה/העתקה.
* paste\_item:
* טענת כניסה: אין
* טענת יציאה: מדביקה את הפריט שהועתק או חתוך לספרייה הנוכחית ושולחת פקודה לשרת.
* delete\_selected\_item:
* טענת כניסה: הנתיב של הפריט הנבחר.
* טענת יציאה: מוחקת את הפריט שנבחר בנתיב שצוין ושולחת פקודה לשרת.
* create\_new\_file/create\_new\_directory:
* טענת כניסה: אין
* טענת יציאה: יוצרת קובץ/תיקייה חדש/ה בנתיב הנוכחי ושולחת פקודה לשרת.

### ClientThread 3.2.6

* תפקיד: מייצגת thread לטיפול בחיבור לקוח (יורשת מ-threading.thread).
* תכונות:
* mysql\_connection: אובייקט החיבור לMySQL.
* username: שם המשתמש של הלקוח המחובר ל-thread.
* client\_socket: socket הלקוח.
* client\_address: כתובת הלקוח.
* folder\_path: הנתיב לתיקיית הלקוח.
* friends: רשימת חברי הלקוח
* friend\_requests: רשימת בקשות החברות ללקוח.
* lock: מנעול למניעת "התנגשויות" במהלך שינויי הקבצים.
* fernet: אובייקט ה- Fernetלהצפנה ופענוח מול הלקוח.
* פעולות:
* run:
* טענת כניסה: self.
* טענת יציאה: מתחילה את התהליך ומטפלת בחיבור הלקוח.
* handle\_commands:
* טענת כניסה: self.
* טענת יציאה: מטפל בפקודות שמתקבלות מהלקוח.

## 3.3 קטעי קוד של האלגוריתמים המרכזיים

### 3.3.1 קוד אימות משתמש

קליטת פרטי משתמש, אימותם מול ה-Database ומתן גישה לאחר אימות מוצלח. הצפנה של המידע העובר בתקשורת והצפנת הסיסמה.

Client.py:

def login(self):  
 *"""  
 Performs the login operation.  
 Retrieves the username and password from the input fields, sends the login request to the server,  
 receives the server's response, and handles the response accordingly.* ***:return****: None  
 """* username = self.username\_input.text()  
 password = self.password\_input.text()  
 if username == '' or password == '':  
 return  
 # perform login logic here  
 print(f"Username: {username}")  
 print(f"Password: {password}")  
 # Send the username and password to the server for signup  
 msg = f"login {username} {hashlib.md5(password.encode()).hexdigest()}"  
 send\_data(client\_socket, msg)  
  
 # Receive the server's response  
 response = receive\_data(client\_socket)  
  
 # Check the server's response and show an appropriate message  
 if response == "OK":  
 # Welcome message  
 print(f"Login Successful - Welcome, {username}!")  
 send\_data(client\_socket, "download\_folder")  
 dir\_data = receive\_data(client\_socket, return\_bytes=True)  
 folder = loads(dir\_data)  
 my\_folder = folder.create(os.path.join(DIRECTORY, folder.name))  
  
 self.goto\_files(my\_folder.path)  
 elif response == "FAIL":  
 print("Login Failed - Invalid username or password")  
 self.login\_fail\_label.show()  
 else:  
 fail\_label = create\_fail\_label(self, response, QtCore.QRect(150, 260, 285, 18))  
 fail\_label.show()

server.py:

while True:  
 # Receive the command from the client (login or signup)  
 try:  
 data = self.receive\_data(self.client\_socket)  
 print(f"data: {data}")  
 if data is None:  
 raise ValueError  
 except (OSError, InvalidToken, ValueError) as err:  
 print(f"Connection from {self.client\_address} closed")  
 break  
 command = data.split()[0]  
 print(command)  
  
 # Verify the username and password against the MySQL table  
 mysql\_connection = mysql.connector.connect(\*\*database\_config)  
 mysql\_cursor = mysql\_connection.cursor()  
 if command == "login":  
 # Receive the username and password from the client  
 self.username = data.split()[1]  
 password = data.split()[2]  
 print(f"Username: {self.username} | Password: {password}")  
  
 mysql\_cursor.execute("SELECT \* FROM users WHERE username = %s AND password = %s",  
 (self.username, password))  
 result = mysql\_cursor.fetchone()  
 if result:  
 if self.username in connected\_users:  
 self.send\_data(self.client\_socket, "User already connected")  
 else:  
 waiting\_commands[self.username] = []  
 connected\_users.append(self.username)  
 self.send\_data(self.client\_socket, "OK")  
 self.folder\_path = os.path.join(FOLDER, self.username)  
 self.friends = [] if result[3] is None else result[3].split(',')  
 self.friend\_requests = [] if result[4] is None else result[4].split(',')  
 self.handle\_commands(mysql\_connection,  
 mysql\_cursor) # Call a method to handle subsequent commands  
  
 else:  
 self.send\_data(self.client\_socket, "FAIL")

### 3.3.2 קוד סנכרון התיקיות

זיהוי פעולה בתיקייה, העברת המידע לשרת, ביצוע הפעולה במערכת הקבצים בשרת ושליחת הפקודה לביצוע הפעולה במשתמשים המחוברים שיש להם הרשאות לתיקייה. דוגמה של העלאת תיקייה של לקוח 1 ועדכון הפעולה בלקוח 2 בעל הרשאות כתיבה וקריאה:  
client.py1:

def upload\_folder(self, model):  
 *"""  
 Allows the user to select a folder to upload and sends it to the server.* ***:param*** *model: The model representing the file system view.* ***:return****: None  
 """* directory = QtWidgets.QFileDialog.getExistingDirectory(self, "Select Folder to Upload",  
 QtCore.QDir.homePath())  
 parent\_path = model.rootPath()  
 if directory:  
 # Check if a directory is selected  
 if not os.path.isdir(parent\_path):  
 parent\_path = self.dir\_path  
 directory = Directory(directory)  
 new\_dir = directory.create(os.path.join(parent\_path, directory.name))  
 # Refresh the file system view  
 model.setRootPath(model.rootPath())  
 serialized\_dir = dumps(new\_dir)  
 new\_dir\_path = new\_dir.path  
 if FOLDER in new\_dir\_path:  
 relative\_path = os.path.relpath(new\_dir\_path, DIRECTORY)  
 else:  
 relative\_path = os.path.relpath(new\_dir\_path, self.read\_write\_path)  
  
 # Create a thread and start the network operations  
 thread = threading.Thread(target=self.upload\_directory, args=(relative\_path, serialized\_dir,))  
 thread.start()  
  
@staticmethod  
def upload\_directory(relative\_path, serialized\_dir):  
 *"""  
 Uploads a directory to the server.* ***:param*** *relative\_path: The relative path of the directory.* ***:param*** *serialized\_dir: The serialized representation of the directory.* ***:return****: None  
 """* send\_data(client\_socket, f"upload\_dir ||{relative\_path}")  
 send\_data(client\_socket, serialized\_dir, send\_bytes=True)

Server.py:

elif data.startswith("upload\_dir"):  
 serialized\_dir = self.receive\_data(self.client\_socket, return\_bytes=True)  
 directory = loads(serialized\_dir)  
 rel\_path = data.split("||")[1].strip() # Extract the relative path  
 location = os.path.join(FOLDER, rel\_path) # Create the target location  
 with self.lock:  
 directory.create(location) # Create the directory at the target location  
 print(f"Folder {location} uploaded")  
 modified\_folder = pathlib.Path(rel\_path).parts[0] # Get the modified folder name from the relative path  
 update\_command((data, serialized\_dir), self.username,  
 modified\_folder) # Add the command to waiting\_commands

client.py2:

receive\_commands\_thread = threading.Thread(target=self.handle\_waiting\_commands)  
receive\_commands\_thread.start()

def handle\_waiting\_commands(self):  
 *"""  
 Handles waiting commands by continuously receiving and processing commands.* ***:returns****: None  
 """* while not self.exit:  
 self.receive\_commands()  
 time.sleep(REFRESH\_FREQUENCY)  
  
def receive\_commands(self):  
 *"""  
 Receives and processes current user's waiting commands from the server.* ***:returns****: None  
 """* try:  
 with self.lock:  
 send\_data(client\_socket, "request\_commands")  
 serialized\_commands = receive\_data(client\_socket, return\_bytes=True)  
 commands = loads(serialized\_commands)  
 print(commands)  
 self.read\_write\_watcher.blockSignals(True)  
 self.watcher.blockSignals(True)  
 for command in commands:  
 if type(command) is tuple:  
 if command[0].startswith("upload\_dir"):  
 rel\_path = command[0].split("||")[-1].strip() # Extract the relative path  
 serialized\_dir = command[1]  
 directory = loads(serialized\_dir)  
 if pathlib.Path(rel\_path).parts[0] == self.username:  
 location = os.path.join(DIRECTORY, rel\_path) # If the modified folder is owned by  
 # the user, use the user's main directory  
 else:  
 if pathlib.Path(rel\_path).parts[0] in self.shared\_read\_write:  
 location = os.path.join(self.read\_write\_path, rel\_path) # If the modified folder  
 # is in the shared read-write list, use the read-write path  
 else:  
 location = os.path.join(self.read\_only\_path, rel\_path) # Otherwise, use the  
 # read-only path  
 directory.create(location) # Create the directory at the specified location

### 3.3.3 קוד ניהול הרשאות

להענקה או ביטול של הרשאות, אימות זכויות גישה למשתמש והגבלת פעולות על סמך רמות הרשאה.

Client.py:

def friend\_double\_clicked(self, item):  
 *"""  
 Handles the action when a friend is double-clicked in the friend list widget.* ***:param*** *item: The selected item representing the friend.* ***:return****: None  
 """* friend\_name = item.text()  
 if friend\_name == NO\_FRIENDS:  
 return  
 # Create a dialog box  
 dialog = QMessageBox()  
 dialog.setWindowTitle("Friend Details")  
 dialog.setText(f"Friend: {friend\_name}")  
  
 # Add share and remove buttons  
 share\_button = dialog.addButton("Share", QMessageBox.ActionRole)  
 remove\_button = dialog.addButton("Remove Friend", QMessageBox.ActionRole)  
  
 # Add a cancel button  
 cancel\_button = dialog.addButton(QMessageBox.Cancel)  
  
 # Disable the default OK button  
 dialog.setDefaultButton(cancel\_button)  
  
 # Execute the dialog and handle the button clicked event  
 dialog.exec\_()  
  
 clicked\_button = dialog.clickedButton()  
 if clicked\_button == share\_button:  
 # Share button clicked, call self.share\_friend  
 self.share\_friend(friend\_name)  
 elif clicked\_button == remove\_button:  
 # Remove friend button clicked, call self.remove\_friend  
 self.remove\_friend(friend\_name)  
 elif clicked\_button == cancel\_button:  
 # Cancel button clicked, do nothing or perform any required cleanup  
 print("Canceled")  
  
def share\_friend(self, friend\_name):  
 *"""  
 Send the server a command to share the user's directory with the selected permissions to a friend.* ***:param*** *friend\_name: The name of the friend to share to.* ***:return****: None  
 """* # Create a Directory object to be shared  
 directory = Directory(self.dir\_path)  
 if friend\_name in self.sharing\_read\_write:  
 self.change\_permission(friend\_name, "read\_write")  
 return  
 elif friend\_name in self.sharing\_read\_only:  
 self.change\_permission(friend\_name, "read")  
 return  
 # Show a pop-up message box to ask for permissions  
 message\_box = QMessageBox()  
 message\_box.setWindowTitle("Permission Selection")  
 message\_box.setText(f"What permissions would you like to give {friend\_name}?")  
  
 # Add buttons for read, write, and cancel options  
 read\_button = QPushButton("Read Only")  
 write\_button = QPushButton("Read And Write")  
 cancel\_button = QPushButton("Cancel")  
 message\_box.addButton(read\_button, QMessageBox.ButtonRole.AcceptRole)  
 message\_box.addButton(write\_button, QMessageBox.ButtonRole.AcceptRole)  
 message\_box.addButton(cancel\_button, QMessageBox.ButtonRole.RejectRole)  
 message\_box.setDefaultButton(cancel\_button)  
 # Execute the message box and get the selected button  
 clicked\_button = message\_box.exec\_()  
 # Process the selected button  
 if clicked\_button == 0:  
 print(f"Sharing read-only with friend: {friend\_name}")  
 self.sharing\_read\_only.append(friend\_name)  
 self.sharing\_read\_only\_list\_widget.clear()  
 self.sharing\_read\_only\_list\_widget.addItems(self.sharing\_read\_only)  
 send\_data(client\_socket, f"share||{friend\_name}||read")  
 send\_data(client\_socket, dumps(directory), send\_bytes=True)  
 elif clicked\_button == 1:  
 print(f"Sharing read-write with friend: {friend\_name}")  
 self.sharing\_read\_write.append(friend\_name)  
 self.sharing\_read\_write\_list\_widget.clear()  
 self.sharing\_read\_write\_list\_widget.addItems(self.sharing\_read\_write)  
 send\_data(client\_socket, f"share||{friend\_name}||read\_write")  
 send\_data(client\_socket, dumps(directory), send\_bytes=True)  
  
 else:  
 print("Share canceled")  
  
def change\_permission(self, shared\_user, current\_perm):  
 *"""  
 Changes the sharing permissions for a friend.* ***:param*** *shared\_user: The username of the friend.* ***:param*** *current\_perm: The current permissions of the friend.* ***:return****: None  
 """* message\_box = QMessageBox()  
 message\_box.setWindowTitle("Change Permission")  
 message\_box.setText(f"{shared\_user} currently has {current\_perm} permissions.\nWould you like to change them?")  
 if current\_perm == "read\_write":  
 # Add buttons for read, write, and cancel options  
 read\_button = QPushButton("Read Only")  
 message\_box.addButton(read\_button, QMessageBox.ButtonRole.AcceptRole)  
 elif current\_perm == "read":  
 write\_button = QPushButton("Read And Write")  
 message\_box.addButton(write\_button, QMessageBox.ButtonRole.AcceptRole)  
 remove\_perms\_button = QPushButton("Remove Permissions")  
 cancel\_button = QPushButton("Cancel")  
 message\_box.addButton(remove\_perms\_button, QMessageBox.ButtonRole.RejectRole)  
 message\_box.addButton(cancel\_button, QMessageBox.ButtonRole.RejectRole)  
 message\_box.setDefaultButton(cancel\_button)  
 # Execute the message box and get the selected button  
 clicked\_button = message\_box.exec\_()  
 if clicked\_button == 0:  
 if current\_perm == "read\_write":  
 self.sharing\_read\_write\_list\_widget.takeItem(self.sharing\_read\_write.index(shared\_user))  
 self.sharing\_read\_write.remove(shared\_user)  
 self.sharing\_read\_only.append(shared\_user)  
 self.sharing\_read\_only\_list\_widget.addItem(shared\_user)  
 send\_data(client\_socket, f"share||{shared\_user}||read")  
 print(f"Changed {shared\_user}'s permissions from read and write to read only")  
 else:  
 self.sharing\_read\_only\_list\_widget.takeItem(self.sharing\_read\_only.index(shared\_user))  
 self.sharing\_read\_only.remove(shared\_user)  
 self.sharing\_read\_write.append(shared\_user)  
 self.sharing\_read\_write\_list\_widget.addItem(shared\_user)  
 send\_data(client\_socket, f"share||{shared\_user}||read\_write")  
 print(f"Changed {shared\_user}'s permissions from read only to read and write")  
  
 elif clicked\_button == 1:  
 if current\_perm == "read\_write":  
 self.sharing\_read\_write\_list\_widget.takeItem(self.sharing\_read\_write.index(shared\_user))  
 self.sharing\_read\_write.remove(shared\_user)  
 send\_data(client\_socket, f"share||{shared\_user}||remove")  
 print(f"Removed permissions for {shared\_user}")  
 else:  
 self.sharing\_read\_only\_list\_widget.takeItem(self.sharing\_read\_only.index(shared\_user))  
 self.sharing\_read\_only.remove(shared\_user)  
 send\_data(client\_socket, f"share||{shared\_user}||remove")  
 print(f"Removed permissions for {shared\_user}")  
  
def sharing\_to\_double\_clicked(self, item):  
 *"""  
 Handles the action when a user is double-clicked in the sharing list widget.* ***:param*** *item: The selected item representing the user.* ***:return****: None  
 """* user = item.text()  
 if user in self.sharing\_read\_write:  
 self.change\_permission(user, "read\_write")  
 else:  
 self.change\_permission(user, "read")

server.py:

elif data.startswith("share"):  
 shared\_user = data.split("||")[1]  
 permissions = data.split("||")[2]  
 if shared\_user in get\_users\_user\_is\_sharing\_with(self.username):  
 remove\_row(self.username, shared\_user)  
 else: # user doesn't have the shared folder yet   
 serialized\_dir = self.receive\_data(self.client\_socket, return\_bytes=True)  
 add\_to\_waiting\_commands([shared\_user], (data, serialized\_dir))  
 if permissions != "remove":  
 insert\_user\_sharing(self.username, shared\_user, permissions)  
 print(f"{self.username} has shared his folder with {shared\_user} with {permissions} permissions")  
 print(f"{self.username} is currently sharing to {get\_users\_user\_is\_sharing\_with(self.username)}")

אם ההרשאה ניתנה למשתמש שלא הייתה לו כבר הרשאה:

Client.py2

elif command[0].startswith("share"):  
 permissions = command[0].split("||")[2]  
 serialized\_dir = command[1]  
 directory = loads(serialized\_dir)  
 if permissions == "read":  
 dir\_path = os.path.join(self.read\_only\_path, directory.name)  
 elif permissions == "read\_write":  
 dir\_path = os.path.join(self.read\_write\_path, directory.name)  
 directory.create(dir\_path)

### 3.3.4 קוד ריענון

אחזור ועיבוד רשימות משתמשים, חברים, בקשות חברות ושיתופים.  
אחזור נתונים מה-Database, שליחת הנתונים ללקוח ועדכון ממשק המשתמש עם הנתונים החדשים.  
client.py:

refreshes\_thread = threading.Thread(target=self.handle\_refreshes)  
refreshes\_thread.start()

def handle\_refreshes(self):  
 *"""  
 Handles periodic refreshes by triggering the refresh operation.* ***:returns****: None  
 """* while not self.exit:  
 self.refresh()  
 time.sleep(REFRESH\_FREQUENCY)

def refresh(self):  
 *"""  
 Refreshes the state of the file sharing application by updating the data based on the received information.* ***:returns****: None* ***:raises*** *OSError: If an error occurs during the refresh process.  
 """* try:  
 with self.lock:  
 send\_data(client\_socket, "refresh")  
 data = receive\_data(client\_socket)  
 print(data)  
 self.users = data.split('||')[0].split(',')  
 self.users.remove(os.path.basename(self.dir\_path))  
 friends = data.split('||')[1].split(',')  
 friend\_requests = data.split('||')[2].split(',')  
 sharing\_read = data.split('||')[3].split(',')  
 sharing\_rw = data.split('||')[4].split(',')  
 self.sharing\_read\_only = sharing\_read if sharing\_read != [''] else []  
 self.sharing\_read\_write = sharing\_rw if sharing\_rw != [''] else []  
 shared\_read = data.split('||')[5].split(',')  
 shared\_rw = data.split('||')[6].split(',')  
 self.shared\_read\_only = shared\_read if shared\_read != [''] else []  
 self.shared\_read\_write = shared\_rw if shared\_rw != [''] else []  
 self.friends = friends if friends[0] else []  
 self.friend\_requests = friend\_requests if friend\_requests[0] else []  
 self.friends\_list\_widget.clear()  
 if not self.friends:  
 self.friends\_list\_widget.addItem(NO\_FRIENDS)  
 else:  
 self.friends\_list\_widget.addItems(self.friends)  
 self.friend\_requests\_list\_widget.clear()  
 if not self.friend\_requests:  
 self.friend\_requests\_list\_widget.addItem(NO\_FRIEND\_REQUESTS)  
 else:  
 self.friend\_requests\_list\_widget.addItems(self.friend\_requests)  
 self.sharing\_read\_write\_list\_widget.clear()  
 if self.sharing\_read\_write:  
 self.sharing\_read\_write\_list\_widget.addItems(self.sharing\_read\_write)  
 self.sharing\_read\_only\_list\_widget.clear()  
 if self.sharing\_read\_only:  
 self.sharing\_read\_only\_list\_widget.addItems(self.sharing\_read\_only)  
 for folder in os.listdir(self.read\_write\_path):  
 if folder in self.shared\_read\_only:  
 f = Directory(os.path.join(self.read\_write\_path, folder))  
 f.change\_path(os.path.join(self.read\_only\_path, folder))  
 elif folder not in self.shared\_read\_write:  
 shutil.rmtree(os.path.join(self.read\_write\_path, folder))  
 for folder in os.listdir(self.read\_only\_path):  
 if folder in self.shared\_read\_write:  
 f = Directory(os.path.join(self.read\_only\_path, folder))  
 f.change\_path(os.path.join(self.read\_write\_path, folder))  
 elif folder not in self.shared\_read\_only:  
 shutil.rmtree(os.path.join(self.read\_only\_path, folder))  
 except OSError:  
 self.exit = True

Server.py:

elif data.startswith("refresh"):  
 with self.lock:  
 try:  
 mysql\_connection = mysql.connector.connect(\*\*database\_config)  
 mysql\_cursor = mysql\_connection.cursor()  
 # Execute the query to fetch the updated user list  
 mysql\_cursor.execute("SELECT username FROM users")  
 # Fetch all the usernames from the result  
 rows = mysql\_cursor.fetchall()  
 updated\_users = ','.join([row[0] for row in rows])  
 mysql\_cursor.execute("SELECT friends, friend\_requests FROM users WHERE username = %s",  
 (self.username,))  
 row = mysql\_cursor.fetchone()  
 self.friends = [] if row[0] is None else row[0].split(',')  
 print(f"{self.username} friends: {self.friends}")  
 self.friend\_requests = [] if row[1] is None else row[1].split(',')  
 print(f"{self.username} friend requests: {self.friend\_requests}")  
 friends = ','.join(self.friends)  
 friend\_requests = ','.join(self.friend\_requests)  
 sharing\_read\_only = ','.join(get\_sharing\_read\_only(self.username))  
 sharing\_read\_write = ','.join(get\_sharing\_read\_write(self.username))  
 shared\_read\_only = ','.join(get\_shared\_read\_only(self.username))  
 shared\_read\_write = ','.join(get\_shared\_read\_write(self.username))  
 message = f"{updated\_users}||{friends}||{friend\_requests}||{sharing\_read\_only}||" \  
 f"{sharing\_read\_write}||{shared\_read\_only}||{shared\_read\_write}"  
 self.send\_data(self.client\_socket, message)  
 except Exception as error:  
 print(error, Exception)

# 4 מדריך למשתמש

## 4.1 קבצי הפרויקט

תיקייה כללית FileSpace:

* client.py
* client\_thread.py
* file\_classes.py
* login\_window.py
* main\_window.py
* server.py
* signup\_window.py
* תיקיית FS (לקוח)
  + Folders מכילה את תיקיות המשתמשים.
  + תיקיית Read And Write: מאחסנת תיקיות בשמות המשתמשים ובהן התיקיות של המשתמשים שמשתפים עם המשתמש עם הרשאות כתיבה וקריאה.
  + תיקיית Read Only: מאחסנת תיקיות בשמות המשתמשים ובהן התיקיות של המשתמשים שמשתפים עם המשתמש עם הרשאות כתיבה בלבד.
* תיקיית ServerFolder: מאחסנת את תיקיות המשתמשים בשרת.

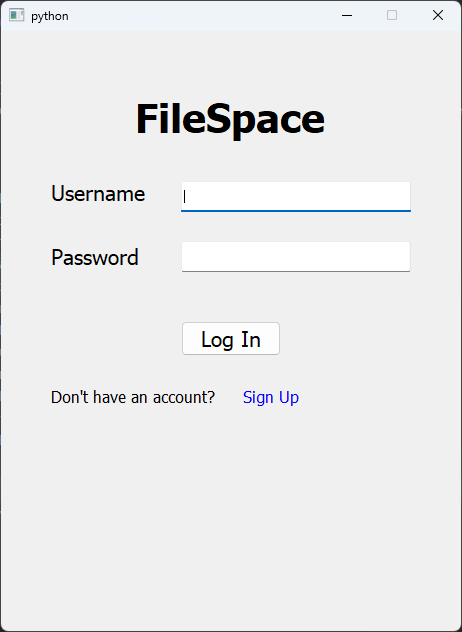
## 4.2 התקנת המערכת

דרושה סביבת Python וחיבור לאינטרנט כדי להריץ את המערכת. במחשב השרת דרושים הקבצים file\_classes.py ,server.py ו- client\_thread.py. במחשב הלקוח דרושים הקבצים client.py, file\_classes.py, login\_window.py, signup\_window.py, main\_window.py. בנוסף, צריךFileSpace Database של MySQL בשרת ולעדכן את הקונפיגורציה בserver.py-ו-client\_thread.py.

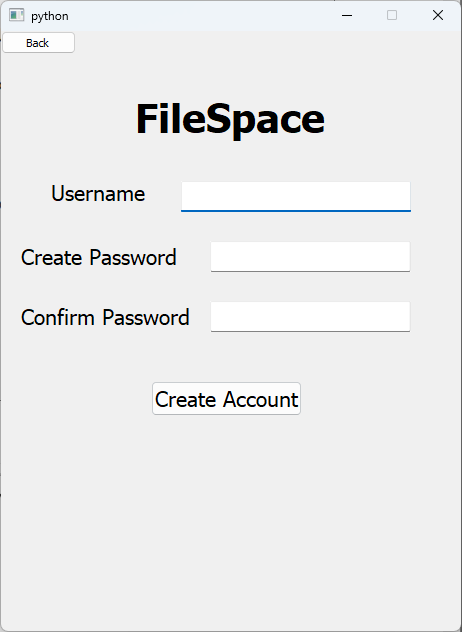
## 4.3 משתמשי המערכת

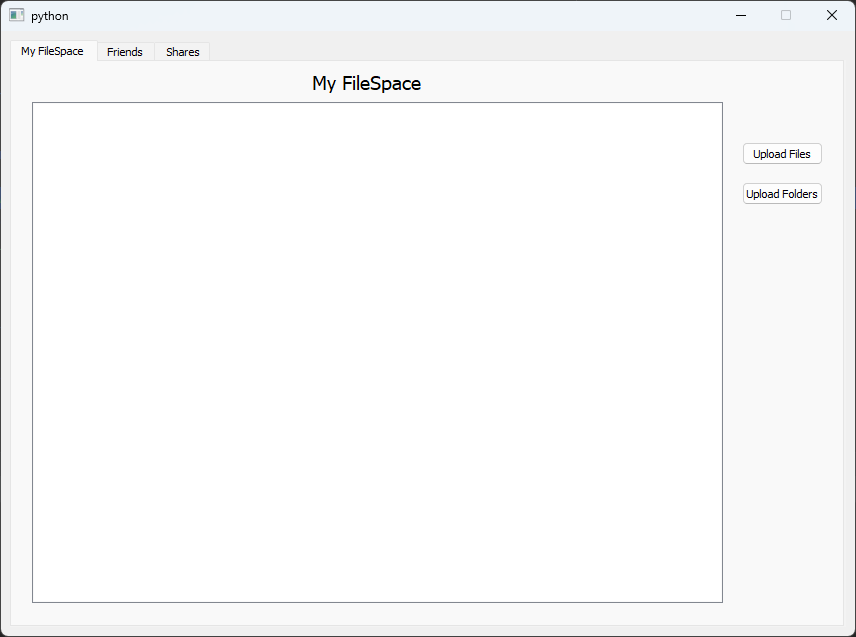
* מנהל המערכת: מפעיל את המערכת ע"י הרצת server.py.
* משתמש קצה: צריך להכניס את ה-IP של השרת ל- client.py ולהריץ את הקובץ.

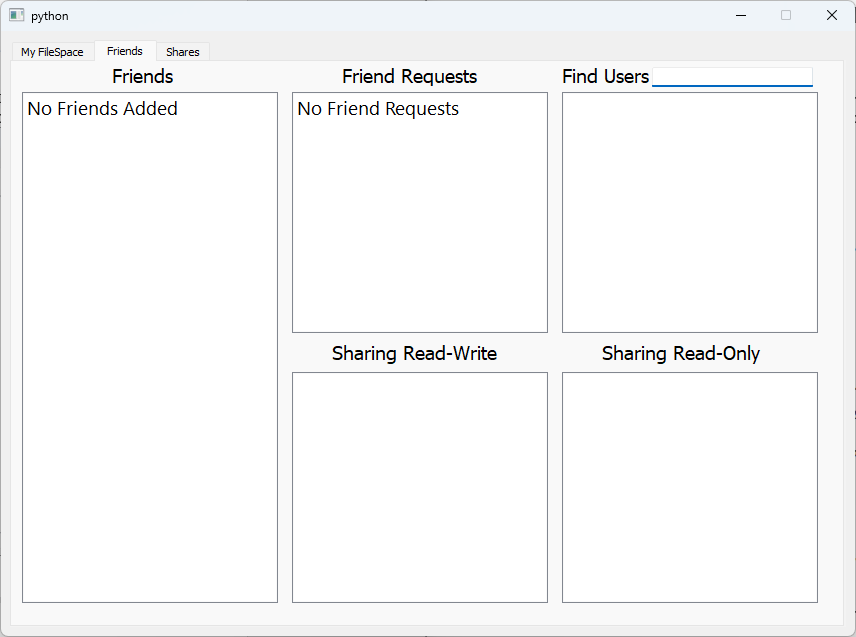
לאחר הרצת client.py יופיע המסך ההתחלתי. כדי ליצור משתמש יש ללחוץ על "Sign Up":

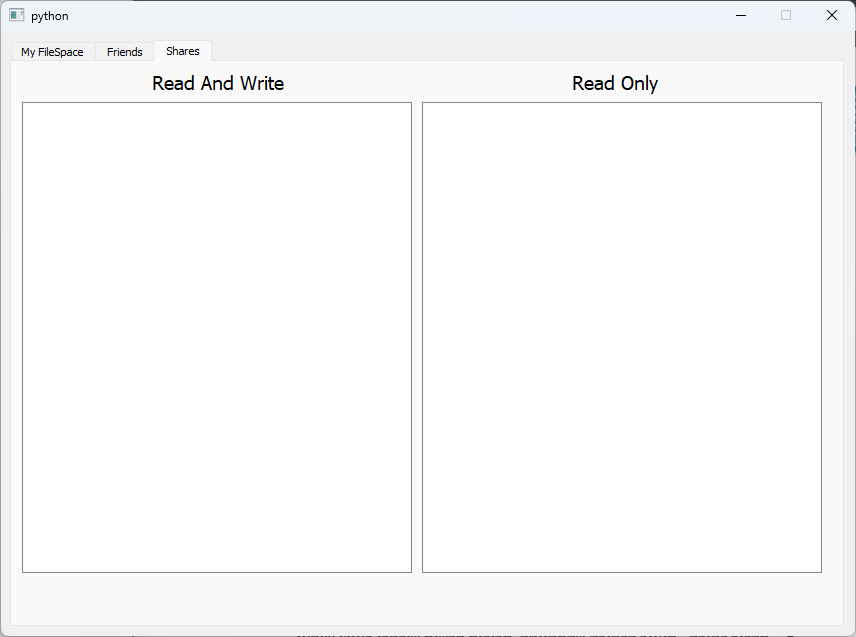


לאחר מכן, המסך ישתנה למסך ההרשמה ההתחלתי:



לאחר הרשמה או התחברות מוצלחת יופיע המסך הראשי של האפליקציה. כרטיסיית "My FileSpace": בכרטיסייה זו אפשר לבצע את הפעולות על תיקיית המשתמש.  


כרטיסיית "Friends": בכרטיסייה זו ניתן לחפש משתמשים, לשלוח להם בקשות חברות ולנהל את מתן ההרשאות לחברים. 

כרטיסיית "Shares": בכרטיסייה זו אפשר לראות את התיקיות המשותפות עם המשתמש והשינויים שקורים בהן ולבצע עליהן פעולות אם הן תחת "Read And Write".

# 5 רפלקציה

* דרך העבודה על הפרויקט הצלחתי להבין יותר לעומק נושאים שלמדנו כמו threading, SQL, יצירת ממשק משתמש, תכנות מונחה עצמים, תקשורת מרובת לקוחות והצפנה.
* היה לי מאתגר להבין איך לקבל עדכונים בלקוח על פעולות שמתרחשות בלקוח אחר כמו שינוי בקובץ, שליחת בקשת חברות, שינוי בהרשאות וכו'. פתרתי את זה בעזרת תהליך שרץ במקביל לתהליך הראשי שבו רץ ה-GUI.
* בהתחלה היה לי קושי ביצירת הGUI בעזרת Tkinter, אז החלפתי ל-PyQt ועיצבתי את ה-GUI דרך Qt Designer וכך חסכתי לעצמי הרבה זמן.
* למדתי איך לכתוב קוד פרויקט בצורה מסודרת יותר ויעילה יותר.
* אם היה לי עוד זמן, הייתי מוסיף אפשרות לחיפוש קבצים/תיקיות, אפשרות לשתף פריטים בודדים ומערכת התראות/הודעות.
* בראייה לאחור, הייתי צריך להתחיל את הפרויקט מוקדם יותר מכיוון שאם היה לי עוד זמן הייתי רוצה לשפר את הקוד ואת ספר הפרויקט.
* במהלך הלמידה במגמה למדתי המון על תכנות בנושאים כמו הגנת סייבר, תכנות מונחה עצמים, הצפנות, יצירת ממשק משתמש, תקשורת מרובת לקוחות וכו', ואני שמח מאוד שלמדתי במגמה זו כי קיבלתי כלים להמשך הלמידה בתחום המחשבים.

# 6 קוד הפרויקט

## server.py 6.1

import socket  
import mysql  
import mysql.connector  
  
from client\_thread import ClientThread  
  
host = "0.0.0.0" # Host IP address where the server will be running  
port = 8080 # Port num to bind the server socket  
  
# Define the MySQL database connection parameters  
database\_config = {  
 "host": "localhost",  
 "user": "root",  
 "password": "OC8305",  
 "database": "FileSpace"  
}  
  
create\_users\_table\_query = """  
CREATE TABLE IF NOT EXISTS users (  
 id INT AUTO\_INCREMENT PRIMARY KEY,  
 username VARCHAR(255),  
 password VARCHAR(255),  
 friends VARCHAR(4095),  
 friend\_requests VARCHAR(4095)  
 )  
"""  
create\_shares\_table\_query = """  
CREATE TABLE IF NOT EXISTS users\_sharing (  
 id INT PRIMARY KEY AUTO\_INCREMENT,  
 sharing\_user VARCHAR(255),  
 shared\_user VARCHAR(255),  
 permission ENUM('read', 'read\_write')  
)  
"""  
  
  
def main():  
 # Create the MySQL table if it doesn't exist  
 mysql\_connection = mysql.connector.connect(\*\*database\_config)  
 mysql\_cursor = mysql\_connection.cursor()  
 try:  
 mysql\_cursor.execute(create\_users\_table\_query)  
 mysql\_cursor.execute(create\_shares\_table\_query)  
 mysql\_connection.commit()  
 except mysql.connector.Error as error:  
 print(f"Error creating MySQL table: {error}")  
 finally:  
 mysql\_cursor.close()  
 mysql\_connection.close()  
  
 # Create a new server socket  
 server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
  
 # Bind the server socket to the host and port  
 server\_socket.bind((host, port))  
  
 # Start listening for incoming connections  
 server\_socket.listen()  
  
 print(f"Server listening on {host}:{port}")  
  
 while True:  
 # Accept incoming connections and start a new thread for each client  
 try:  
 client\_socket, client\_address = server\_socket.accept()  
  
 client\_thread = ClientThread(client\_socket, client\_address)  
 client\_thread.start()  
  
 except ConnectionError as err:  
 print(err)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

## client\_thread.py 6.2

import pathlib  
from cryptography.fernet import Fernet, InvalidToken  
import rsa  
import os  
import shutil  
import threading  
from pickle import dumps, loads  
import mysql  
from file\_classes import Directory  
  
FOLDER = "./ServerFolder"  
CHUNK\_SIZE = 4096  
HEADER\_SIZE = 10  
public\_key, private\_key = rsa.newkeys(1024)  
waiting\_commands = {}  
connected\_users = []  
  
database\_config = {  
 "host": "localhost",  
 "user": "root",  
 "password": "OC8305",  
 "database": "FileSpace"  
}  
  
  
class ClientThread(threading.Thread):  
 SYMMETRIC\_KEY = Fernet.generate\_key()  
  
 def \_\_init\_\_(self, client\_socket, client\_address):  
 *"""  
 Represents a thread for handling client connections.* ***:param*** *client\_socket: The client socket for communication.* ***:param*** *client\_address: The address of the client.  
 """* super().\_\_init\_\_()  
 self.mysql\_connection = None  
 self.username = None  
 self.client\_socket = client\_socket  
 self.client\_address = client\_address  
 self.folder\_path = None  
 self.friends = []  
 self.friend\_requests = []  
 self.lock = threading.Lock()  
 self.fernet = Fernet(self.SYMMETRIC\_KEY)  
  
 def run(self):  
 *"""  
 Starts the client thread and handles the client connection.* ***:return****: None  
 """* mysql\_cursor = None  
 mysql\_connection = None  
 print(f"Connection from {self.client\_address}")  
 self.client\_socket.send(public\_key.save\_pkcs1("PEM"))  
 public\_partner = rsa.PublicKey.load\_pkcs1(self.client\_socket.recv(1024))  
 # Encrypt the symmetric key using the client's public key  
 encrypted\_symmetric\_key = rsa.encrypt(self.SYMMETRIC\_KEY, public\_partner)  
 # Send the encrypted symmetric key to the client  
 self.client\_socket.send(encrypted\_symmetric\_key)  
 try:  
 while True:  
 # Receive the command from the client (login or signup)  
 try:  
 data = self.receive\_data(self.client\_socket)  
 print(f"data: {data}")  
 if data is None:  
 raise ValueError  
 except (OSError, InvalidToken, ValueError):  
 print(f"Connection from {self.client\_address} closed")  
 break  
 command = data.split()[0]  
 print(command)  
  
 # Verify the username and password against the MySQL table  
 mysql\_connection = mysql.connector.connect(\*\*database\_config)  
 mysql\_cursor = mysql\_connection.cursor()  
 if command == "login":  
 # Receive the username and password from the client  
 self.username = data.split()[1]  
 password = data.split()[2]  
 print(f"Username: {self.username} | Password: {password}")  
  
 mysql\_cursor.execute("SELECT \* FROM users WHERE username = %s AND password = %s",  
 (self.username, password))  
 result = mysql\_cursor.fetchone()  
 if result:  
 if self.username in connected\_users:  
 self.send\_data(self.client\_socket, "User already connected")  
 else:  
 waiting\_commands[self.username] = []  
 connected\_users.append(self.username)  
 self.send\_data(self.client\_socket, "OK")  
 self.folder\_path = os.path.join(FOLDER, self.username)  
 self.friends = [] if result[3] is None else result[3].split(',')  
 self.friend\_requests = [] if result[4] is None else result[4].split(',')  
 self.handle\_commands(mysql\_connection,  
 mysql\_cursor) # Call a method to handle subsequent commands  
  
 else:  
 self.send\_data(self.client\_socket, "FAIL")  
 elif command == "signup":  
 # Receive the username and password from the client  
 self.username = data.split()[1]  
 password = data.split()[2]  
 print(f"Username: {self.username} | Password: {password}")  
 # Check if the username already exists in the table  
 mysql\_cursor.execute("SELECT \* FROM users WHERE username = %s", (self.username,))  
 result = mysql\_cursor.fetchone()  
 if result:  
 self.send\_data(self.client\_socket, "FAIL")  
 else:  
 connected\_users.append(self.username)  
 mysql\_cursor.execute("INSERT INTO users (username, password) VALUES (%s, %s)",  
 (self.username, password))  
 mysql\_connection.commit()  
 self.folder\_path = os.path.join(FOLDER, self.username)  
 os.makedirs(self.folder\_path)  
 self.send\_data(self.client\_socket, "OK")  
 self.handle\_commands(mysql\_connection,  
 mysql\_cursor) # Call a method to handle subsequent commands  
 except InvalidToken:  
 # Close the MySQL connection and client socket  
 if mysql\_cursor is not None:  
 mysql\_cursor.close()  
 if mysql\_connection is not None:  
 mysql\_connection.close()  
 self.client\_socket.close()  
  
 connected\_users.remove(self.username)  
 print(f"Connection from {self.client\_address} closed")  
  
 def handle\_commands(self, mysql\_connection, mysql\_cursor):  
 *"""  
 Handles subsequent commands received from the client.* ***:param*** *mysql\_connection: The MySQL connection object.* ***:param*** *mysql\_cursor: The MySQL cursor object.* ***:return****: None  
 """* while True:  
 # Receive the command from the client  
 try:  
 data = self.receive\_data(self.client\_socket)  
 print(data)  
 except (InvalidToken, ConnectionError):  
 self.client\_socket.close()  
 connected\_users.remove(self.username)  
 print(f"Connection from {self.client\_address} closed")  
 break  
  
 if not data:  
 break # Exit the loop if no more data is received  
  
 if data.startswith("download\_folder"):  
 try:  
 folder = Directory(self.folder\_path)  
 except FileNotFoundError:  
 os.makedirs(self.folder\_path)  
 folder = Directory(self.folder\_path)  
 serialized\_dir = dumps(folder)  
 self.send\_data(self.client\_socket, serialized\_dir, send\_bytes=True)  
 print(f"Sent {folder.path} to {self.username}")  
 elif data.startswith("get\_shared\_folders"):  
 self.send\_data(self.client\_socket, str(len(get\_users\_sharing\_with\_user(self.username))))  
 for user in get\_users\_sharing\_with\_user(self.username):  
 print(f"sending {user} to {self.username}")  
 folder = Directory(os.path.join(FOLDER, user))  
 serialized\_dir = dumps(folder)  
 self.send\_data(self.client\_socket, serialized\_dir, send\_bytes=True)  
 elif data.startswith("delete\_item"):  
 rel\_path = data.split("||")[1].strip()  
 item\_path = os.path.join(FOLDER, rel\_path)  
 with self.lock:  
 delete\_item(item\_path)  
 print(f"Deleted {item\_path}")  
 modified\_folder = pathlib.Path(rel\_path).parts[0]  
 update\_command(data, self.username, modified\_folder)  
 elif data.startswith("rename\_item"):  
 rel\_path = data.split("||")[1].strip()  
 item\_path = os.path.join(FOLDER, rel\_path)  
 new\_name = data.split("||")[-1].strip()  
 with self.lock:  
 rename\_item(item\_path, new\_name)  
 print(f"Renamed {item\_path} to {new\_name}")  
 modified\_folder = pathlib.Path(rel\_path).parts[0]  
 update\_command(data, self.username, modified\_folder)  
 elif data.startswith("create\_file"):  
 rel\_path = data.split("||")[1].strip()  
 new\_file\_path = os.path.join(FOLDER, rel\_path)  
 if os.path.exists(new\_file\_path):  
 return  
 with self.lock:  
 # Create the new file  
 with open(new\_file\_path, 'w'):  
 pass # Do nothing, just create an empty file  
 print(f"File {new\_file\_path} created")  
 modified\_folder = pathlib.Path(rel\_path).parts[0]  
 update\_command(data, self.username, modified\_folder)  
 elif data.startswith("create\_folder"):  
 rel\_path = data.split("||")[1].strip()  
 new\_dir\_path = os.path.join(FOLDER, rel\_path)  
 os.makedirs(new\_dir\_path, exist\_ok=True)  
 print(f"Folder {new\_dir\_path} created")  
 modified\_folder = pathlib.Path(rel\_path).parts[0]  
 update\_command(data, self.username, modified\_folder)  
 elif data.startswith("upload\_dir"):  
 serialized\_dir = self.receive\_data(self.client\_socket, return\_bytes=True)  
 directory = loads(serialized\_dir)  
 rel\_path = data.split("||")[1].strip() # Extract the relative path  
 location = os.path.join(FOLDER, rel\_path) # Create the target location  
 with self.lock:  
 directory.create(location) # Create the directory at the target location  
 print(f"Folder {location} uploaded")  
 modified\_folder = pathlib.Path(rel\_path).parts[0] # Get the modified folder name from the relative path  
 update\_command((data, serialized\_dir), self.username,  
 modified\_folder) # Add the command to waiting\_commands  
 elif data.startswith("upload\_file"):  
 serialized\_file = self.receive\_data(self.client\_socket, return\_bytes=True)  
 file = loads(serialized\_file)  
 rel\_path = data.split("||")[1].strip()  
 file\_path = os.path.join(FOLDER, rel\_path)  
 with self.lock:  
 file.create(file\_path)  
 print(f"File {file\_path} uploaded")  
 modified\_folder = pathlib.Path(rel\_path).parts[0]  
 update\_command((data, serialized\_file), self.username, modified\_folder)  
 elif data.startswith("copy"):  
 rel\_copied\_item\_path = data.split("||")[1]  
 rel\_destination\_path = data.split("||")[2]  
 copied\_item\_path = os.path.join(FOLDER, rel\_copied\_item\_path)  
 destination\_path = os.path.join(FOLDER, rel\_destination\_path)  
 with self.lock:  
 # Copy the file or folder  
 if os.path.isfile(copied\_item\_path):  
 try:  
 # Copy a file  
 shutil.copy2(copied\_item\_path, destination\_path)  
 print(f"Copied file {copied\_item\_path} to {destination\_path}")  
 except shutil.SameFileError:  
 pass  
 elif os.path.isdir(copied\_item\_path):  
 try:  
 # Copy a folder  
 shutil.copytree(copied\_item\_path,  
 os.path.join(destination\_path, os.path.basename(copied\_item\_path)))  
 print(f"Copied folder {copied\_item\_path} to {destination\_path}")  
 except FileExistsError:  
 pass  
 modified\_folder = pathlib.Path(rel\_copied\_item\_path).parts[0]  
 update\_command(data, self.username, modified\_folder)  
 elif data.startswith("move"):  
 rel\_cut\_item\_path = data.split("||")[1]  
 rel\_destination\_path = data.split("||")[2]  
 cut\_item\_path = os.path.join(FOLDER, rel\_cut\_item\_path)  
 destination\_path = os.path.join(FOLDER, rel\_destination\_path)  
 with self.lock:  
 try:  
 # Move the file or folder  
 shutil.move(cut\_item\_path, destination\_path)  
 print(f"Moved {cut\_item\_path} to {destination\_path}")  
 except shutil.Error:  
 pass  
 modified\_folder = pathlib.Path(rel\_cut\_item\_path).parts[0]  
 update\_command(data, self.username, modified\_folder)  
 elif data.startswith("file\_edit"):  
 rel\_path = data.split("||")[1]  
 file\_path = os.path.join(FOLDER, rel\_path)  
 file\_data = self.receive\_data(self.client\_socket, return\_bytes=True)  
 # Acquire the lock before opening the file and writing to it  
 with self.lock:  
 with open(file\_path, "wb") as f:  
 f.write(file\_data)  
 modified\_folder = pathlib.Path(rel\_path).parts[0]  
 print(f"mf: {modified\_folder}")  
 update\_command((data, file\_data), self.username, modified\_folder)  
 elif data.startswith("refresh"):  
 with self.lock:  
 try:  
 mysql\_connection = mysql.connector.connect(\*\*database\_config)  
 mysql\_cursor = mysql\_connection.cursor()  
 # Execute the query to fetch the updated user list  
 mysql\_cursor.execute("SELECT username FROM users")  
 # Fetch all the usernames from the result  
 rows = mysql\_cursor.fetchall()  
 updated\_users = ','.join([row[0] for row in rows])  
 mysql\_cursor.execute("SELECT friends, friend\_requests FROM users WHERE username = %s",  
 (self.username,))  
 row = mysql\_cursor.fetchone()  
 self.friends = [] if row[0] is None else row[0].split(',')  
 print(f"{self.username} friends: {self.friends}")  
 self.friend\_requests = [] if row[1] is None else row[1].split(',')  
 print(f"{self.username} friend requests: {self.friend\_requests}")  
 friends = ','.join(self.friends)  
 friend\_requests = ','.join(self.friend\_requests)  
 sharing\_read\_only = ','.join(get\_sharing\_read\_only(self.username))  
 sharing\_read\_write = ','.join(get\_sharing\_read\_write(self.username))  
 shared\_read\_only = ','.join(get\_shared\_read\_only(self.username))  
 shared\_read\_write = ','.join(get\_shared\_read\_write(self.username))  
 message = f"{updated\_users}||{friends}||{friend\_requests}||{sharing\_read\_only}||" \  
 f"{sharing\_read\_write}||{shared\_read\_only}||{shared\_read\_write}"  
 self.send\_data(self.client\_socket, message)  
 except Exception as error:  
 print(error, Exception)  
  
 elif data.startswith("add\_friend"):  
 new\_friend = data.split("||")[1]  
 self.friends.append(new\_friend)  
 friends = ','.join(self.friends)  
 mysql\_cursor.execute("SELECT friends FROM users WHERE username = %s", (new\_friend,))  
 row = mysql\_cursor.fetchone()  
 add\_to\_new\_friend = row[0] + ',' + self.username if row[0] is not None else self.username  
 mysql\_cursor.execute("UPDATE users SET friends = %s WHERE username = %s", (friends, self.username))  
 mysql\_cursor.execute("UPDATE users SET friends = %s WHERE username = %s",  
 (add\_to\_new\_friend, new\_friend))  
 mysql\_connection.commit()  
 print(f"{self.username} has added {new\_friend} as a friend")  
 elif data.startswith("send\_friend\_request"):  
 user = data.split('||')[1]  
 mysql\_cursor.execute("SELECT friend\_requests FROM users WHERE username = %s", (user,))  
 row = mysql\_cursor.fetchone()  
 check\_requests = row[0].split(',') if row[0] is not None else []  
 if self.username not in check\_requests:  
 if row[0] is None:  
 friend\_requests = self.username  
 else:  
 friend\_requests = row[0] + ',' + self.username  
 mysql\_cursor.execute("UPDATE users SET friend\_requests = %s WHERE username = %s",  
 (friend\_requests, user))  
 mysql\_connection.commit()  
 print(f"{self.username} has sent {user} a friend request")  
 self.send\_data(self.client\_socket, "OK")  
 else:  
 self.send\_data(self.client\_socket, "You've already sent this user a friend request")  
 elif data.startswith("rmv\_friend\_request"):  
 user = data.split('||')[1]  
 self.friend\_requests.remove(user)  
 friend\_requests = ','.join(self.friend\_requests) if self.friend\_requests else None  
 mysql\_cursor.execute("UPDATE users SET friend\_requests = %s WHERE username = %s",  
 (friend\_requests, self.username))  
 mysql\_connection.commit()  
 elif data.startswith("remove\_friend"):  
 friend = data.split("||")[1]  
 self.friends.remove(friend)  
 friends = ','.join(self.friends) if self.friends else None  
 mysql\_cursor.execute("SELECT friends FROM users WHERE username = %s", (friend,))  
 row = mysql\_cursor.fetchone()  
 removed\_friend\_friends = row[0].split(',')  
 removed\_friend\_friends.remove(self.username)  
 removed\_friend\_friends = ','.join(removed\_friend\_friends) if removed\_friend\_friends else None  
 mysql\_cursor.execute("UPDATE users SET friends = %s WHERE username = %s", (friends, self.username))  
 mysql\_cursor.execute("UPDATE users SET friends = %s WHERE username = %s",  
 (removed\_friend\_friends, friend))  
 mysql\_connection.commit()  
 print(f"{self.username} and {friend} are no longer friends")  
 elif data.startswith("share"):  
 shared\_user = data.split("||")[1]  
 permissions = data.split("||")[2]  
 if shared\_user in get\_users\_user\_is\_sharing\_with(self.username):  
 remove\_row(self.username, shared\_user)  
 else: # user doesn't have the shared folder yet  
 serialized\_dir = self.receive\_data(self.client\_socket, return\_bytes=True)  
 add\_to\_waiting\_commands([shared\_user], (data, serialized\_dir))  
 if permissions != "remove":  
 insert\_user\_sharing(self.username, shared\_user, permissions)  
 print(f"{self.username} has shared his folder with {shared\_user} with {permissions} permissions")  
 print(f"{self.username} is currently sharing to {get\_users\_user\_is\_sharing\_with(self.username)}")  
 elif data.startswith("request\_commands"):  
 print(waiting\_commands)  
 if self.username in waiting\_commands:  
 commands = dumps(waiting\_commands[self.username])  
 self.send\_data(self.client\_socket, commands, send\_bytes=True)  
 waiting\_commands[self.username] = []  
 else:  
 commands = dumps([])  
 self.send\_data(self.client\_socket, commands, send\_bytes=True)  
  
 else:  
 self.send\_data(self.client\_socket, "Invalid command")  
  
 def send\_data(self, sock, msg, send\_bytes=False):  
 *"""  
 Sends data over a socket connection.* ***:param*** *sock: The socket object representing the connection.* ***:param*** *msg: The message to be sent.* ***:param*** *send\_bytes: A boolean flag indicating whether the message is already bytes (default is False).* ***:returns****: None  
 """* try:  
 if not send\_bytes:  
 msg = msg.encode()  
 msg = self.fernet.encrypt(msg)  
 msg\_len = str(len(msg)).encode()  
 sock.send(self.fernet.encrypt(msg\_len)) # Exactly 100 bytes  
 sock.send(msg)  
 sock.recv(1024)  
 except (ConnectionResetError, OSError) as err:  
 print(err)  
 connected\_users.remove(self.username)  
 sock.close()  
  
 def receive\_data(self, sock, return\_bytes=False):  
 *"""  
 Receives data over a socket connection.* ***:param*** *sock: The socket object representing the connection.* ***:param*** *return\_bytes: A boolean flag indicating whether the received data should be returned as bytes  
 (default is False).* ***:returns****: The received data.  
 """* try:  
 data = b''  
 msg\_len = int(self.fernet.decrypt(sock.recv(100)).decode())  
 while len(data) < msg\_len:  
 chunk = sock.recv(CHUNK\_SIZE)  
 data += chunk  
 data = self.fernet.decrypt(data)  
 if not return\_bytes:  
 data = data.decode()  
 sock.send(self.fernet.encrypt("OK".encode()))  
 return data  
 except (ConnectionResetError, OSError) as err:  
 print(err)  
 connected\_users.remove(self.username)  
 sock.close()  
  
  
def delete\_item(item\_path):  
 if os.path.isfile(item\_path):  
 # Delete a file  
 os.remove(item\_path)  
 elif os.path.isdir(item\_path):  
 # Delete a folder and its contents  
 shutil.rmtree(item\_path)  
  
  
def rename\_item(item\_path, new\_name):  
 try:  
 new\_path = os.path.join(os.path.dirname(item\_path), new\_name)  
 os.rename(item\_path, new\_path)  
 except Exception as err:  
 print(err.args[1])  
  
  
def get\_users\_user\_is\_sharing\_with(username):  
 *"""  
 Retrieves a list of users that the given username is sharing with.* ***:param*** *username: The username for which the shared users are retrieved.* ***:return****: A list of users that the given username is sharing with.* ***:raises*** *mysql.connector.Error: If there is an error retrieving the shared users from the database.  
 """* try:  
 mysql\_connection = mysql.connector.connect(\*\*database\_config)  
 mysql\_cursor = mysql\_connection.cursor()  
 query = "SELECT shared\_user FROM users\_sharing WHERE sharing\_user = %s"  
 values = (username,)  
 mysql\_cursor.execute(query, values)  
 rows = mysql\_cursor.fetchall()  
 mysql\_connection.close()  
 users\_list = []  
 for row in rows:  
 users\_list.append(row[0])  
 return users\_list  
 except mysql.connector.Error as error:  
 print(f"Error retrieving users {username} is sharing with: {error}")  
  
  
def get\_users\_sharing\_with\_user(username):  
 try:  
 mysql\_connection = mysql.connector.connect(\*\*database\_config)  
 mysql\_cursor = mysql\_connection.cursor()  
 query = "SELECT sharing\_user, permission FROM users\_sharing WHERE shared\_user = %s"  
 values = (username,)  
 mysql\_cursor.execute(query, values)  
 rows = mysql\_cursor.fetchall()  
 mysql\_connection.close()  
 users\_list = []  
 for row in rows:  
 users\_list.append(row[0])  
 return users\_list  
 except mysql.connector.Error as error:  
 print(f"Error retrieving users sharing with {username}: {error}")  
  
  
def insert\_user\_sharing(sharing\_user, shared\_user, permission):  
 *"""  
 Inserts user sharing information into the database.* ***:param*** *sharing\_user: The user sharing the folder.* ***:param*** *shared\_user: The user receiving the shared folder.* ***:param*** *permission: The permission level for the shared folder.* ***:return****: True if the user sharing information was inserted successfully, False otherwise.  
 """* try:  
 # Connect to the database  
 mysql\_connection = mysql.connector.connect(\*\*database\_config)  
 mysql\_cursor = mysql\_connection.cursor()  
  
 query = "INSERT INTO users\_sharing (sharing\_user, shared\_user, permission) VALUES (%s, %s, %s)"  
 values = (sharing\_user, shared\_user, permission)  
 mysql\_cursor.execute(query, values)  
 mysql\_connection.commit()  
 mysql\_connection.close()  
 print("User sharing information inserted successfully.")  
 return True  
  
 except mysql.connector.Error as error:  
 print(f"Error inserting user sharing information: {error}")  
  
  
def remove\_row(sharing\_user, shared\_user):  
 *"""  
 Removes a row from the users\_sharing table in the database.* ***:param*** *sharing\_user: The user sharing the folder.* ***:param*** *shared\_user: The user receiving the shared folder.* ***:return****: None  
 """* # Establish a connection to the MySQL server  
 mysql\_connection = mysql.connector.connect(\*\*database\_config)  
 mysql\_cursor = mysql\_connection.cursor()  
 try:  
 # Create a cursor object to execute SQL queries  
 mysql\_cursor = mysql\_connection.cursor()  
  
 # Prepare the DELETE statement  
 delete\_query = "DELETE FROM users\_sharing WHERE sharing\_user = %s AND shared\_user = %s"  
  
 # Execute the DELETE statement with the username as a parameter  
 mysql\_cursor.execute(delete\_query, (sharing\_user, shared\_user))  
  
 # Commit the changes to the database  
 mysql\_connection.commit()  
  
 # Print a message to indicate successful deletion  
 print("Row deleted successfully")  
  
 except mysql.connector.Error as error:  
 # Handle any potential errors  
 print(f"Error deleting row: {error}")  
  
 finally:  
 # Close the cursor and connection  
 mysql\_cursor.close()  
 mysql\_connection.close()  
  
  
def get\_permissions(sharing\_user, shared\_user):  
 *"""  
 Retrieves the permission level for shared folder between two users.* ***:param*** *sharing\_user: The user sharing the folder.* ***:param*** *shared\_user: The user receiving the shared folder.* ***:return****: The permission level.  
 """* # Establish a connection to the MySQL server  
 mysql\_connection = mysql.connector.connect(\*\*database\_config)  
 mysql\_cursor = mysql\_connection.cursor()  
 try:  
 # Create a cursor object to execute SQL queries  
 mysql\_cursor = mysql\_connection.cursor()  
 # Prepare the SELECT statement  
 select\_query = "SELECT permission FROM users\_sharing WHERE sharing\_user = %s AND shared\_user = %s"  
 # Execute the SELECT statement with the sharing\_user and shared\_user as parameters  
 mysql\_cursor.execute(select\_query, (sharing\_user, shared\_user))  
 # Fetch the first row returned by the query  
 row = mysql\_cursor.fetchone()  
 if row:  
 # Return the value of the 'permissions' column  
 return row[0]  
 else:  
 # Return a default value if no matching row is found  
 return None  
 except mysql.connector.Error as error:  
 # Handle any potential errors  
 print(f"Error retrieving permissions: {error}")  
 finally:  
 # Close the cursor and connection  
 mysql\_cursor.close()  
 mysql\_connection.close()  
  
  
def get\_sharing\_read\_only(username):  
 *"""  
 Retrieves a list of users with read-only access to folder shared by the specified user.* ***:param*** *username: The username of the user.* ***:return****: A list of users with read-only access.  
 """* sharing\_read\_only = []  
 users = get\_users\_user\_is\_sharing\_with(username)  
 for shared\_user in users:  
 if get\_permissions(username, shared\_user) == "read":  
 sharing\_read\_only.append(shared\_user)  
 return sharing\_read\_only  
  
  
def get\_sharing\_read\_write(username):  
 *"""  
 Retrieves a list of users with read-write access to folder shared by the specified user.* ***:param*** *username: The username of the user.* ***:return****: A list of users with read-write access.  
 """* sharing\_read\_write = []  
 users = get\_users\_user\_is\_sharing\_with(username)  
 for shared\_user in users:  
 if get\_permissions(username, shared\_user) == "read\_write":  
 sharing\_read\_write.append(shared\_user)  
 return sharing\_read\_write  
  
  
def get\_shared\_read\_only(username):  
 *"""  
 Retrieves a list of users who have shared their folder with the specified user with read-only access.* ***:param*** *username: The username of the user.* ***:return****: A list of users who have shared their folder with read-only access.  
 """* shared\_read\_only = []  
 users = get\_users\_sharing\_with\_user(username)  
 for sharing\_user in users:  
 if get\_permissions(sharing\_user, username) == "read":  
 shared\_read\_only.append(sharing\_user)  
 return shared\_read\_only  
  
  
def get\_shared\_read\_write(username):  
 *"""  
 Retrieves a list of users who have shared their folder with the specified user with read-write access.* ***:param*** *username: The username of the user.* ***:return****: A list of users who have shared their folder with read-write access.  
 """* shared\_read\_write = []  
 users = get\_users\_sharing\_with\_user(username)  
 for sharing\_user in users:  
 if get\_permissions(sharing\_user, username) == "read\_write":  
 shared\_read\_write.append(sharing\_user)  
 return shared\_read\_write  
  
  
def add\_to\_waiting\_commands(users, command):  
 *"""  
 Adds a command to the waiting commands for each user.* ***:param*** *users: A list of users.* ***:param*** *command: The command to be added to the waiting commands.* ***:returns****: None  
 """* for user in users:  
 if user in connected\_users:  
 if user in waiting\_commands:  
 waiting\_commands[user].append(command)  
 else:  
 waiting\_commands[user] = [command]  
  
  
def update\_command(command, username, modified\_folder):  
 *"""  
 Adds the command that was given by the user to the list of commands of each user that has any permission to the  
 modified folder.* ***:param*** *command: The command to be added.* ***:param*** *username: The user that sent the command.* ***:param*** *modified\_folder: The folder that was modified.* ***:returns****: None  
 """* if modified\_folder == username:  
 add\_to\_waiting\_commands(get\_users\_user\_is\_sharing\_with(username), command)  
 else:  
 users = get\_users\_user\_is\_sharing\_with(modified\_folder)  
 users.remove(username)  
 users.append(modified\_folder)  
 add\_to\_waiting\_commands(users, command)

## file\_classes.py 6.3

import os  
import shutil  
  
FOLDER = r'.\ServerFolder'  
  
  
class File:  
 def \_\_init\_\_(self, path):  
 *"""  
 Initialize a File object.* ***:param*** *path:: The path to the file.  
 """* self.path = path  
 self.name = os.path.basename(path)  
 self.size = os.path.getsize(path)  
 self.rel\_path = None  
 with open(self.path, "rb") as f:  
 data = f.read()  
 self.data = data  
  
 def create(self, parent\_path=None):  
 *"""  
 Creates the file in the parent path.* ***:param*** *parent\_path: path to the file. Defaults to None.  
 """* if "ServerFolder" in self.path:  
 self.rel\_path = os.path.relpath(self.path, "./ServerFolder")  
 self.path = os.path.join(FOLDER, self.rel\_path)  
 elif r"Desktop\FS" in self.path:  
 self.rel\_path = self.path.split("Desktop/FS")[-1]  
 self.path = os.path.join(FOLDER, self.rel\_path)  
  
 if parent\_path is None:  
 parent\_path = self.path  
 if os.path.exists(parent\_path):  
 file = File(parent\_path)  
 if file.data != self.data:  
 with open(parent\_path, 'wb'):  
 pass  
 with open(parent\_path, 'wb') as f:  
 f.write(self.data)  
 else:  
 with open(parent\_path, 'wb') as f:  
 f.write(self.data)  
  
  
class Directory:  
 def \_\_init\_\_(self, path):  
 *"""  
 Initialize a Directory object.* ***:param*** *path: The path to the directory.  
 """* self.path = path  
 self.name = os.path.basename(path)  
 self.subdirectories = []  
 self.files = []  
 self.size = 0 # Initialize the size attribute to zero  
  
 for entry in os.listdir(path):  
 full\_path = os.path.join(path, entry)  
 if os.path.isdir(full\_path):  
 subdirectory = Directory(full\_path)  
 self.subdirectories.append(subdirectory)  
 self.size += subdirectory.size # Add subdirectory size to the current directory's size  
 else:  
 file = File(full\_path)  
 self.files.append(file)  
 self.size += file.size # Add file size to the current directory's size  
  
 def create(self, parent\_path=None):  
 *"""  
 Recursively create the directory and its contents.* ***:param*** *parent\_path:The parent path for the directory. Defaults to None.  
 """* if parent\_path is None:  
 parent\_path = self.path  
  
 os.makedirs(parent\_path, exist\_ok=True)  
  
 # Remove non-matching files and subdirectories  
 existing\_files = [file.name for file in self.files]  
 existing\_subdirectories = [subdir.name for subdir in self.subdirectories]  
  
 for item\_name in os.listdir(parent\_path):  
 item\_path = os.path.join(parent\_path, item\_name)  
  
 if os.path.isfile(item\_path) and item\_name not in existing\_files:  
 os.remove(item\_path)  
  
 if os.path.isdir(item\_path) and item\_name not in existing\_subdirectories:  
 shutil.rmtree(item\_path)  
  
 for file in self.files:  
 file.create(os.path.join(parent\_path, file.name))  
  
 for subdirectory in self.subdirectories:  
 subdirectory.create(os.path.join(parent\_path, subdirectory.name))  
 return Directory(parent\_path)  
  
 def change\_path(self, new\_path):  
 *"""  
 Change the path of the directory.* ***:param*** *new\_path: The new path for the directory.* ***:returns****: None  
 """* os.makedirs(os.path.dirname(new\_path), exist\_ok=True)  
 shutil.move(self.path, os.path.dirname(new\_path))  
 os.chdir(os.path.dirname(new\_path))  
 os.rename(self.name, os.path.basename(new\_path))  
 self.path = new\_path  
 self.name = os.path.basename(new\_path)  
  
 def search\_files(self, keyword): # Not used yet  
 *"""  
 Search for files containing the specified keyword in their names within the directory and its subdirectories.* ***:param*** *keyword: The keyword to search for in file names.* ***:return****: A list of matching File objects.  
 """* matching\_files = []  
 for file in self.files:  
 if keyword.upper() in file.name.upper():  
 matching\_files.append(file)  
  
 for subdirectory in self.subdirectories:  
 matching\_files.extend(subdirectory.search\_files(keyword))  
  
 return matching\_files

## client.py 6.4

# *TODO when adding a friend, the folders don't update until the client is reopened - send a friend's after he was added*import hashlib  
import os  
import pathlib  
import shutil  
import socket  
import threading  
import time  
from pickle import loads, dumps  
import rsa  
from cryptography.fernet import Fernet  
from login\_window import Ui\_Login  
from signup\_window import Ui\_Signup  
from PyQt5.QtWidgets import QApplication, QMainWindow, QLineEdit, QWidget, QInputDialog, QMessageBox, QPushButton, \  
 QFileSystemModel  
from PyQt5 import QtCore, QtWidgets, QtGui  
from PyQt5.QtCore import Qt, QFileSystemWatcher  
import sys  
from file\_classes import File, Directory  
from main\_window import Ui\_MainWindow  
  
SERVER\_IP = '127.0.0.1'  
PORT = 8080  
DIRECTORY = "./FS/Folders"  
FOLDER = "Folders"  
READ\_ONLY\_SHARES = "./FS/Read Only"  
READ\_WRITE\_SHARES = "./FS/Read and Write"  
CHUNK\_SIZE = 4096  
KEYS\_TO\_DISABLE = [Qt.Key\_Space, Qt.Key\_Period, Qt.Key\_Slash, Qt.Key\_Comma, Qt.Key\_Semicolon, Qt.Key\_Colon, Qt.Key\_Bar,  
 Qt.Key\_Backslash, Qt.Key\_BracketLeft, Qt.Key\_BracketRight, Qt.Key\_ParenLeft, Qt.Key\_ParenRight,  
 Qt.Key\_BraceLeft, Qt.Key\_BraceRight, Qt.Key\_Apostrophe, Qt.Key\_QuoteDbl, Qt.Key\_Equal, Qt.Key\_Plus,  
 Qt.Key\_Minus, Qt.Key\_Percent, Qt.Key\_Question]  
REFRESH\_FREQUENCY = 5  
NO\_FRIENDS = "No Friends Added"  
NO\_FRIEND\_REQUESTS = "No Friend Requests"  
  
  
def send\_data(sock, msg, send\_bytes=False):  
 *"""  
 Sends data over a socket connection.* ***:param*** *sock: The socket object representing the connection.* ***:param*** *msg: The message to be sent.* ***:param*** *send\_bytes: A boolean flag indicating whether the message is already bytes (default is False).* ***:returns****: None  
 """* print(msg)  
 if not send\_bytes:  
 msg = msg.encode()  
 msg = fernet.encrypt(msg)  
 msg\_len = str(len(msg)).encode()  
 sock.send(fernet.encrypt(msg\_len)) # Exactly 100 bytes  
 sock.send(msg)  
 sock.recv(1024)  
  
  
def receive\_data(sock, return\_bytes=False):  
 *"""  
 Receives data over a socket connection.* ***:param*** *sock: The socket object representing the connection.* ***:param*** *return\_bytes: A boolean flag indicating whether the received data should be returned as bytes  
 (default is False).* ***:returns****: The received data.  
 """* data = b''  
 msg\_len = int(fernet.decrypt(sock.recv(100)).decode())  
 while len(data) < msg\_len:  
 chunk = sock.recv(CHUNK\_SIZE)  
 data += chunk  
 data = fernet.decrypt(data)  
 if not return\_bytes:  
 data = data.decode()  
 sock.send(fernet.encrypt("OK".encode()))  
 return data  
  
  
def disable\_keys(field):  
 *"""  
 Disables key events for a QLineEdit field if they're in KEYS\_TO\_DISABLE.* ***:param*** *field: The field object for which the key events should be disabled.* ***:returns****: None  
 """* field.keyPressEvent = lambda event: event.ignore() if event.key() in KEYS\_TO\_DISABLE else QLineEdit.keyPressEvent(  
 field, event)  
  
  
def delete\_item(item\_path):  
 *"""  
 Deletes a file or folder.* ***:param*** *item\_path: The path of the item to be deleted.* ***:returns****: None  
 """* if os.path.isfile(item\_path):  
 # Delete a file  
 os.remove(item\_path)  
 elif os.path.isdir(item\_path):  
 # Delete a folder and its contents  
 shutil.rmtree(item\_path)  
  
  
def rename\_item(item\_path, new\_name):  
 *"""  
 Renames a file or folder.* ***:param*** *item\_path: The path of the item to be renamed.* ***:param*** *new\_name: The new name for the item.* ***:returns****: None* ***:raises*** *Exception: If an error occurs during the renaming process.  
 """* ok = True  
 try:  
 new\_path = os.path.join(os.path.dirname(item\_path), new\_name)  
 os.rename(item\_path, new\_path)  
 except OSError as err:  
 ok = False  
 item = "file"  
 if os.path.isdir(item\_path):  
 item = "folder"  
 QMessageBox.warning(widget, "Error", f"Invalid {item} name.")  
 print(err)  
 return ok  
  
  
def create\_fail\_label(parent, text, geometry):  
 *"""  
 Creates a label for displaying failure messages.* ***:param*** *parent: The parent widget where the label will be placed.* ***:param*** *text: The text to be displayed in the label.* ***:param*** *geometry: The geometry (position and size) of the label.* ***:returns****: The created QLabel object.  
 """* fail\_label = QtWidgets.QLabel(parent)  
 fail\_label.setGeometry(geometry)  
 fail\_label.setText(text)  
 fail\_label.hide()  
 font = QtGui.QFont()  
 font.setPointSize(11)  
 fail\_label.setFont(font)  
 fail\_label.setStyleSheet("color: rgb(255, 0, 0)")  
 return fail\_label  
  
  
def open\_file(item\_path):  
 *"""  
 Opens a file using the default system application.* ***:param*** *item\_path: The path of the file to be opened.* ***:returns****: None  
 """* if os.path.isfile(item\_path):  
 os.startfile(item\_path)  
  
  
class MainWindow(QWidget, Ui\_MainWindow):  
 *"""  
 Represents the main window of the application. Inherits from QWidget and Ui\_MainWindow.  
 """* def \_\_init\_\_(self, dir\_path):  
 *"""  
 Initializes the MainWindow object.  
  
 Sets up the UI elements, initializes variables, connects signals to slots, & performs necessary configurations.* ***:param*** *dir\_path: The path of the current directory.* ***:return****: None  
 """* super().\_\_init\_\_()  
 self.lock = threading.Lock()  
 self.exit = False  
 self.copied\_item\_path = None  
 self.cut\_item\_path = None  
 self.dir\_path = dir\_path  
 self.username = os.path.basename(dir\_path)  
 self.read\_write\_path = os.path.join(READ\_WRITE\_SHARES, self.username)  
 self.read\_only\_path = os.path.join(READ\_ONLY\_SHARES, self.username)  
 self.file\_timestamps = {}  
 self.directory\_history = [] # List to store directory navigation history  
 self.read\_write\_directory\_history = [self.read\_write\_path]  
 self.read\_only\_directory\_history = [self.read\_only\_path]  
 self.users = []  
 self.friends = []  
 self.friend\_requests = []  
 self.sharing\_read\_only = []  
 self.sharing\_read\_write = []  
 self.shared\_read\_only = []  
 self.shared\_read\_write = []  
 os.makedirs(self.read\_only\_path, exist\_ok=True)  
 os.makedirs(self.read\_write\_path, exist\_ok=True)  
 self.setupUi(self)  
 self.setWindowTitle("FileSpace")  
 self.model = QFileSystemModel()  
 self.model.setRootPath(dir\_path)  
 self.list\_view.setModel(self.model)  
 self.list\_view.setRootIndex(self.model.index(dir\_path))  
 self.list\_view.setIconSize(QtCore.QSize(32, 32))  
 self.list\_view.setGridSize(QtCore.QSize(96, 96))  
 self.list\_view.setViewMode(QtWidgets.QListView.IconMode)  
 self.directory\_history.append(self.dir\_path)  
 self.tabs.setCurrentIndex(0)  
 self.upload\_files\_button.clicked.connect(lambda: self.upload\_file(self.model))  
 self.upload\_folders\_button.clicked.connect(lambda: self.upload\_folder(self.model))  
 self.list\_view.doubleClicked.connect(self.on\_list\_view\_double\_clicked)  
 self.list\_view.setContextMenuPolicy(QtCore.Qt.CustomContextMenu)  
 self.list\_view.customContextMenuRequested.connect(lambda event: self.create\_context\_menu(event, self.list\_view))  
 self.go\_back\_button.hide()  
 self.go\_back\_button.clicked.connect(self.go\_back)  
 self.rw\_go\_back\_button.hide()  
 self.rw\_go\_back\_button.clicked.connect(self.rw\_go\_back)  
 self.r\_go\_back\_button.hide()  
 self.r\_go\_back\_button.clicked.connect(self.r\_go\_back)  
 self.upload\_files\_shares\_button.hide()  
 self.upload\_files\_shares\_button.clicked.connect(lambda: self.upload\_file(self.read\_write\_model))  
 self.upload\_folders\_shares\_button.hide()  
 self.upload\_folders\_shares\_button.clicked.connect(lambda: self.upload\_folder(self.read\_write\_model))  
 self.friends\_list\_widget.itemDoubleClicked.connect(self.friend\_double\_clicked)  
 self.friend\_requests\_list\_widget.itemDoubleClicked.connect(self.friend\_request\_double\_clicked)  
 self.sharing\_read\_write\_list\_widget.itemDoubleClicked.connect(self.sharing\_to\_double\_clicked)  
 self.sharing\_read\_only\_list\_widget.itemDoubleClicked.connect(self.sharing\_to\_double\_clicked)  
 self.search\_bar.textChanged.connect(self.search\_users)  
 self.search\_results\_list.itemDoubleClicked.connect(self.user\_double\_clicked)  
 self.read\_only\_model = QFileSystemModel()  
 self.read\_only\_model.setRootPath(self.read\_only\_path)  
 self.read\_only\_list\_view.setModel(self.read\_only\_model)  
 self.read\_only\_list\_view.setRootIndex(self.read\_only\_model.index(self.read\_only\_path))  
 self.read\_only\_list\_view.doubleClicked.connect(self.on\_read\_only\_list\_view\_double\_clicked)  
 self.read\_write\_model = QFileSystemModel()  
 self.read\_write\_model.setRootPath(self.read\_write\_path)  
 self.read\_write\_list\_view.setModel(self.read\_write\_model)  
 self.read\_write\_list\_view.setRootIndex(self.read\_write\_model.index(self.read\_write\_path))  
 self.read\_write\_list\_view.setContextMenuPolicy(QtCore.Qt.CustomContextMenu)  
 self.read\_write\_list\_view.customContextMenuRequested.connect(  
 lambda event: self.create\_context\_menu(event, self.read\_write\_list\_view))  
 refreshes\_thread = threading.Thread(target=self.handle\_refreshes)  
 refreshes\_thread.start()  
 self.get\_shared\_folders()  
 self.watcher = QFileSystemWatcher()  
 self.watcher.addPath(self.dir\_path)  
 self.recursively\_add\_paths(self.dir\_path) # Add subdirectories to watcher recursively  
 self.watcher.fileChanged.connect(self.file\_changed)  
 self.read\_write\_watcher = QFileSystemWatcher()  
 self.read\_write\_watcher.addPath(self.read\_write\_path)  
 self.recursively\_add\_paths(self.read\_write\_path)  
 self.read\_write\_watcher.fileChanged.connect(self.file\_changed)  
 receive\_commands\_thread = threading.Thread(target=self.handle\_waiting\_commands)  
 receive\_commands\_thread.start()  
 self.read\_write\_list\_view.doubleClicked.connect(self.on\_read\_write\_list\_view\_double\_clicked)  
  
 def get\_shared\_folders(self):  
 *"""  
 Retrieves and creates shared folders based on the received data.* ***:returns****: None  
 """* with self.lock:  
 send\_data(client\_socket, "get\_shared\_folders")  
 num = int(receive\_data(client\_socket))  
 for i in range(num):  
 dir\_data = receive\_data(client\_socket, return\_bytes=True)  
 folder = loads(dir\_data)  
 if folder.name in self.shared\_read\_write:  
 folder.create(os.path.join(self.read\_write\_path, folder.name))  
 elif folder.name in self.shared\_read\_only:  
 folder.create(os.path.join(self.read\_only\_path, folder.name))  
  
 def handle\_waiting\_commands(self):  
 *"""  
 Handles waiting commands by continuously receiving and processing commands.* ***:returns****: None  
 """* while not self.exit:  
 self.receive\_commands()  
 time.sleep(REFRESH\_FREQUENCY)  
  
 def handle\_refreshes(self):  
 *"""  
 Handles periodic refreshes by triggering the refresh operation.* ***:returns****: None  
 """* while not self.exit:  
 self.refresh()  
 time.sleep(REFRESH\_FREQUENCY)  
  
 def receive\_commands(self):  
 *"""  
 Receives and processes current user's waiting commands from the server.* ***:returns****: None  
 """* try:  
 with self.lock:  
 send\_data(client\_socket, "request\_commands")  
 serialized\_commands = receive\_data(client\_socket, return\_bytes=True)  
 commands = loads(serialized\_commands)  
 print(commands)  
 self.read\_write\_watcher.blockSignals(True)  
 self.watcher.blockSignals(True)  
 for command in commands:  
 if type(command) is tuple:  
 if command[0].startswith("upload\_dir"):  
 rel\_path = command[0].split("||")[-1].strip() # Extract the relative path  
 serialized\_dir = command[1]  
 directory = loads(serialized\_dir)  
 if pathlib.Path(rel\_path).parts[0] == self.username:  
 location = os.path.join(DIRECTORY, rel\_path) # If the modified folder is owned by  
 # the user, use the user's main directory  
 else:  
 if pathlib.Path(rel\_path).parts[0] in self.shared\_read\_write:  
 location = os.path.join(self.read\_write\_path, rel\_path) # If the modified folder  
 # is in the shared read-write list, use the read-write path  
 else:  
 location = os.path.join(self.read\_only\_path, rel\_path) # Otherwise, use the  
 # read-only path  
 directory.create(location) # Create the directory at the specified location  
 elif command[0].startswith("upload\_file"):  
 rel\_path = command[0].split("||")[-1].strip()  
 serialized\_file = command[1]  
 file = loads(serialized\_file)  
 if pathlib.Path(rel\_path).parts[0] == self.username:  
 location = os.path.join(DIRECTORY, rel\_path)  
 watcher\_path = self.dir\_path  
 else:  
 if pathlib.Path(rel\_path).parts[0] in self.shared\_read\_write:  
 location = os.path.join(self.read\_write\_path, rel\_path)  
 watcher\_path = self.read\_write\_path  
 else:  
 location = os.path.join(self.read\_only\_path, rel\_path)  
 file.create(location)  
 self.recursively\_add\_paths(watcher\_path)  
 elif command[0].startswith("file\_edit"):  
 rel\_path = command[0].split("||")[-1]  
 file\_data = command[1]  
 if pathlib.Path(rel\_path).parts[0] == self.username:  
 file\_path = os.path.join(DIRECTORY, rel\_path)  
 else:  
 if pathlib.Path(rel\_path).parts[0] in self.shared\_read\_write:  
 file\_path = os.path.join(self.read\_write\_path, rel\_path)  
 else:  
 file\_path = os.path.join(self.read\_only\_path, rel\_path)  
 with open(file\_path, "wb") as f:  
 f.write(file\_data)  
 elif command[0].startswith("share"):  
 permissions = command[0].split("||")[2]  
 serialized\_dir = command[1]  
 directory = loads(serialized\_dir)  
 if permissions == "read":  
 dir\_path = os.path.join(self.read\_only\_path, directory.name)  
 elif permissions == "read\_write":  
 dir\_path = os.path.join(self.read\_write\_path, directory.name)  
 directory.create(dir\_path)  
 elif command.startswith("delete\_item"):  
 rel\_path = command.split("||")[1].strip()  
 if pathlib.Path(rel\_path).parts[0] == self.username:  
 item\_path = os.path.join(DIRECTORY, rel\_path)  
 else:  
 if pathlib.Path(rel\_path).parts[0] in self.shared\_read\_write:  
 item\_path = os.path.join(self.read\_write\_path, rel\_path)  
 else:  
 item\_path = os.path.join(self.read\_only\_path, rel\_path)  
 delete\_item(item\_path)  
 print(f"Deleted {item\_path}")  
 elif command.startswith("rename\_item"):  
 rel\_path = command.split("||")[1].strip()  
 print(os.path.join(self.read\_write\_path, rel\_path))  
 if pathlib.Path(rel\_path).parts[0] == self.username:  
 item\_path = os.path.join(DIRECTORY, rel\_path)  
 else:  
 if pathlib.Path(rel\_path).parts[0] in self.shared\_read\_write:  
 item\_path = os.path.join(self.read\_write\_path, rel\_path)  
 else:  
 item\_path = os.path.join(self.read\_only\_path, rel\_path)  
 new\_name = command.split("||")[-1].strip()  
  
 rename\_item(item\_path, new\_name)  
 print(f"Renamed {item\_path} to {new\_name}")  
 elif command.startswith("create\_file"):  
 rel\_path = command.split("||")[1].strip()  
 if pathlib.Path(rel\_path).parts[0] == self.username:  
 new\_file\_path = os.path.join(DIRECTORY, rel\_path)  
 watcher\_path = self.dir\_path  
 else:  
 if pathlib.Path(rel\_path).parts[0] in self.shared\_read\_write:  
 new\_file\_path = os.path.join(self.read\_write\_path, rel\_path)  
 watcher\_path = self.read\_write\_path  
 else:  
 new\_file\_path = os.path.join(self.read\_only\_path, rel\_path)  
 if os.path.exists(new\_file\_path):  
 return  
 # Create the new file  
 with open(new\_file\_path, 'w'):  
 pass # Do nothing, just create an empty file  
 self.recursively\_add\_paths(watcher\_path)  
 print(f"File {new\_file\_path} created")  
 elif command.startswith("create\_folder"):  
 rel\_path = command.split("||")[1].strip()  
 if pathlib.Path(rel\_path).parts[0] == self.username:  
 new\_dir\_path = os.path.join(DIRECTORY, rel\_path)  
 else:  
 if pathlib.Path(rel\_path).parts[0] in self.shared\_read\_write:  
 new\_dir\_path = os.path.join(self.read\_write\_path, rel\_path)  
 else:  
 new\_dir\_path = os.path.join(self.read\_only\_path, rel\_path)  
 os.makedirs(new\_dir\_path, exist\_ok=True)  
 print(f"Folder {new\_dir\_path} created")  
 elif command.startswith("copy"):  
 rel\_copied\_item\_path = command.split("||")[1]  
 rel\_destination\_path = command.split("||")[2]  
 if pathlib.Path(rel\_copied\_item\_path).parts[0] == self.username:  
 copied\_item\_path = os.path.join(DIRECTORY, rel\_copied\_item\_path)  
 destination\_path = os.path.join(DIRECTORY, rel\_destination\_path)  
 else:  
 if pathlib.Path(rel\_copied\_item\_path).parts[0] in self.shared\_read\_write:  
 copied\_item\_path = os.path.join(self.read\_write\_path, rel\_copied\_item\_path)  
 destination\_path = os.path.join(self.read\_write\_path, rel\_destination\_path)  
 else:  
 copied\_item\_path = os.path.join(self.read\_only\_path, rel\_copied\_item\_path)  
 destination\_path = os.path.join(self.read\_only\_path, rel\_destination\_path)  
  
 # Copy the file or folder  
 if os.path.isfile(copied\_item\_path):  
 try:  
 # Copy a file  
 shutil.copy2(copied\_item\_path, destination\_path)  
 print(f"Copied file {copied\_item\_path} to {destination\_path}")  
 except shutil.SameFileError:  
 pass  
 elif os.path.isdir(copied\_item\_path):  
 try:  
 # Copy a folder  
 shutil.copytree(copied\_item\_path,  
 os.path.join(destination\_path, os.path.basename(copied\_item\_path)))  
 print(f"Copied folder {copied\_item\_path} to {destination\_path}")  
 except FileExistsError:  
 pass  
  
 elif command.startswith("move"):  
 rel\_cut\_item\_path = command.split("||")[1]  
 rel\_destination\_path = command.split("||")[2]  
 if pathlib.Path(rel\_cut\_item\_path).parts[0] == self.username:  
 cut\_item\_path = os.path.join(DIRECTORY, rel\_cut\_item\_path)  
 destination\_path = os.path.join(DIRECTORY, rel\_destination\_path)  
 else:  
 if pathlib.Path(rel\_cut\_item\_path).parts[0] in self.shared\_read\_write:  
 cut\_item\_path = os.path.join(self.read\_write\_path, rel\_cut\_item\_path)  
 destination\_path = os.path.join(self.read\_write\_path, rel\_destination\_path)  
 else:  
 cut\_item\_path = os.path.join(self.read\_only\_path, rel\_cut\_item\_path)  
 destination\_path = os.path.join(self.read\_only\_path, rel\_destination\_path)  
 try:  
 # Move the file or folder  
 shutil.move(cut\_item\_path, destination\_path)  
 if os.path.isfile(destination\_path):  
 self.recursively\_add\_paths(self.read\_write\_path)  
 print(f"Moved {cut\_item\_path} to {destination\_path}")  
 except shutil.Error:  
 pass  
 self.read\_write\_watcher.blockSignals(False)  
 self.watcher.blockSignals(False)  
 print(f"commands:{commands}")  
 except OSError as err:  
 print(err)  
 self.exit = True  
  
 def refresh(self):  
 *"""  
 Refreshes the state of the file sharing application by updating the data based on the received information.* ***:returns****: None* ***:raises*** *OSError: If an error occurs during the refresh process.  
 """* try:  
 with self.lock:  
 send\_data(client\_socket, "refresh")  
 data = receive\_data(client\_socket)  
 print(data)  
 self.users = data.split('||')[0].split(',')  
 self.users.remove(os.path.basename(self.dir\_path))  
 friends = data.split('||')[1].split(',')  
 friend\_requests = data.split('||')[2].split(',')  
 sharing\_read = data.split('||')[3].split(',')  
 sharing\_rw = data.split('||')[4].split(',')  
 self.sharing\_read\_only = sharing\_read if sharing\_read != [''] else []  
 self.sharing\_read\_write = sharing\_rw if sharing\_rw != [''] else []  
 shared\_read = data.split('||')[5].split(',')  
 shared\_rw = data.split('||')[6].split(',')  
 self.shared\_read\_only = shared\_read if shared\_read != [''] else []  
 self.shared\_read\_write = shared\_rw if shared\_rw != [''] else []  
 self.friends = friends if friends[0] else []  
 self.friend\_requests = friend\_requests if friend\_requests[0] else []  
 self.friends\_list\_widget.clear()  
 if not self.friends:  
 self.friends\_list\_widget.addItem(NO\_FRIENDS)  
 else:  
 self.friends\_list\_widget.addItems(self.friends)  
 self.friend\_requests\_list\_widget.clear()  
 if not self.friend\_requests:  
 self.friend\_requests\_list\_widget.addItem(NO\_FRIEND\_REQUESTS)  
 else:  
 self.friend\_requests\_list\_widget.addItems(self.friend\_requests)  
 self.sharing\_read\_write\_list\_widget.clear()  
 if self.sharing\_read\_write:  
 self.sharing\_read\_write\_list\_widget.addItems(self.sharing\_read\_write)  
 self.sharing\_read\_only\_list\_widget.clear()  
 if self.sharing\_read\_only:  
 self.sharing\_read\_only\_list\_widget.addItems(self.sharing\_read\_only)  
 for folder in os.listdir(self.read\_write\_path):  
 if folder in self.shared\_read\_only:  
 f = Directory(os.path.join(self.read\_write\_path, folder))  
 f.change\_path(os.path.join(self.read\_only\_path, folder))  
 elif folder not in self.shared\_read\_write:  
 shutil.rmtree(os.path.join(self.read\_write\_path, folder))  
 for folder in os.listdir(self.read\_only\_path):  
 if folder in self.shared\_read\_write:  
 f = Directory(os.path.join(self.read\_only\_path, folder))  
 f.change\_path(os.path.join(self.read\_write\_path, folder))  
 elif folder not in self.shared\_read\_only:  
 shutil.rmtree(os.path.join(self.read\_only\_path, folder))  
 except OSError:  
 self.exit = True  
  
 def friend\_double\_clicked(self, item):  
 *"""  
 Handles the action when a friend is double-clicked in the friend list widget.* ***:param*** *item: The selected item representing the friend.* ***:return****: None  
 """* friend\_name = item.text()  
 if friend\_name == NO\_FRIENDS:  
 return  
 # Create a dialog box  
 dialog = QMessageBox()  
 dialog.setWindowTitle("Friend Details")  
 dialog.setText(f"Friend: {friend\_name}")  
  
 # Add share and remove buttons  
 share\_button = dialog.addButton("Share", QMessageBox.ActionRole)  
 remove\_button = dialog.addButton("Remove Friend", QMessageBox.ActionRole)  
  
 # Add a cancel button  
 cancel\_button = dialog.addButton(QMessageBox.Cancel)  
  
 # Disable the default OK button  
 dialog.setDefaultButton(cancel\_button)  
  
 # Execute the dialog and handle the button clicked event  
 dialog.exec\_()  
  
 clicked\_button = dialog.clickedButton()  
 if clicked\_button == share\_button:  
 # Share button clicked, call self.share\_friend  
 self.share\_friend(friend\_name)  
 elif clicked\_button == remove\_button:  
 # Remove friend button clicked, call self.remove\_friend  
 self.remove\_friend(friend\_name)  
 elif clicked\_button == cancel\_button:  
 # Cancel button clicked, do nothing or perform any required cleanup  
 print("Canceled")  
  
 def share\_friend(self, friend\_name):  
 *"""  
 Send the server a command to share the user's directory with the selected permissions to a friend.* ***:param*** *friend\_name: The name of the friend to share to.* ***:return****: None  
 """* # Create a Directory object to be shared  
 directory = Directory(self.dir\_path)  
 if friend\_name in self.sharing\_read\_write:  
 self.change\_permission(friend\_name, "read\_write")  
 return  
 elif friend\_name in self.sharing\_read\_only:  
 self.change\_permission(friend\_name, "read")  
 return  
 # Show a pop-up message box to ask for permissions  
 message\_box = QMessageBox()  
 message\_box.setWindowTitle("Permission Selection")  
 message\_box.setText(f"What permissions would you like to give {friend\_name}?")  
  
 # Add buttons for read, write, and cancel options  
 read\_button = QPushButton("Read Only")  
 write\_button = QPushButton("Read And Write")  
 cancel\_button = QPushButton("Cancel")  
 message\_box.addButton(read\_button, QMessageBox.ButtonRole.AcceptRole)  
 message\_box.addButton(write\_button, QMessageBox.ButtonRole.AcceptRole)  
 message\_box.addButton(cancel\_button, QMessageBox.ButtonRole.RejectRole)  
 message\_box.setDefaultButton(cancel\_button)  
 # Execute the message box and get the selected button  
 clicked\_button = message\_box.exec\_()  
 # Process the selected button  
 if clicked\_button == 0:  
 print(f"Sharing read-only with friend: {friend\_name}")  
 self.sharing\_read\_only.append(friend\_name)  
 self.sharing\_read\_only\_list\_widget.clear()  
 self.sharing\_read\_only\_list\_widget.addItems(self.sharing\_read\_only)  
 send\_data(client\_socket, f"share||{friend\_name}||read")  
 send\_data(client\_socket, dumps(directory), send\_bytes=True)  
 elif clicked\_button == 1:  
 print(f"Sharing read-write with friend: {friend\_name}")  
 self.sharing\_read\_write.append(friend\_name)  
 self.sharing\_read\_write\_list\_widget.clear()  
 self.sharing\_read\_write\_list\_widget.addItems(self.sharing\_read\_write)  
 send\_data(client\_socket, f"share||{friend\_name}||read\_write")  
 send\_data(client\_socket, dumps(directory), send\_bytes=True)  
  
 else:  
 print("Share canceled")  
  
 def change\_permission(self, shared\_user, current\_perm):  
 *"""  
 Changes the sharing permissions for a friend.* ***:param*** *shared\_user: The username of the friend.* ***:param*** *current\_perm: The current permissions of the friend.* ***:return****: None  
 """* message\_box = QMessageBox()  
 message\_box.setWindowTitle("Change Permission")  
 message\_box.setText(f"{shared\_user} currently has {current\_perm} permissions.\nWould you like to change them?")  
 if current\_perm == "read\_write":  
 # Add buttons for read, write, and cancel options  
 read\_button = QPushButton("Read Only")  
 message\_box.addButton(read\_button, QMessageBox.ButtonRole.AcceptRole)  
 elif current\_perm == "read":  
 write\_button = QPushButton("Read And Write")  
 message\_box.addButton(write\_button, QMessageBox.ButtonRole.AcceptRole)  
 remove\_perms\_button = QPushButton("Remove Permissions")  
 cancel\_button = QPushButton("Cancel")  
 message\_box.addButton(remove\_perms\_button, QMessageBox.ButtonRole.RejectRole)  
 message\_box.addButton(cancel\_button, QMessageBox.ButtonRole.RejectRole)  
 message\_box.setDefaultButton(cancel\_button)  
 # Execute the message box and get the selected button  
 clicked\_button = message\_box.exec\_()  
 if clicked\_button == 0:  
 if current\_perm == "read\_write":  
 self.sharing\_read\_write\_list\_widget.takeItem(self.sharing\_read\_write.index(shared\_user))  
 self.sharing\_read\_write.remove(shared\_user)  
 self.sharing\_read\_only.append(shared\_user)  
 self.sharing\_read\_only\_list\_widget.addItem(shared\_user)  
 send\_data(client\_socket, f"share||{shared\_user}||read")  
 print(f"Changed {shared\_user}'s permissions from read and write to read only")  
 else:  
 self.sharing\_read\_only\_list\_widget.takeItem(self.sharing\_read\_only.index(shared\_user))  
 self.sharing\_read\_only.remove(shared\_user)  
 self.sharing\_read\_write.append(shared\_user)  
 self.sharing\_read\_write\_list\_widget.addItem(shared\_user)  
 send\_data(client\_socket, f"share||{shared\_user}||read\_write")  
 print(f"Changed {shared\_user}'s permissions from read only to read and write")  
  
 elif clicked\_button == 1:  
 if current\_perm == "read\_write":  
 self.sharing\_read\_write\_list\_widget.takeItem(self.sharing\_read\_write.index(shared\_user))  
 self.sharing\_read\_write.remove(shared\_user)  
 send\_data(client\_socket, f"share||{shared\_user}||remove")  
 print(f"Removed permissions for {shared\_user}")  
 else:  
 self.sharing\_read\_only\_list\_widget.takeItem(self.sharing\_read\_only.index(shared\_user))  
 self.sharing\_read\_only.remove(shared\_user)  
 send\_data(client\_socket, f"share||{shared\_user}||remove")  
 print(f"Removed permissions for {shared\_user}")  
  
 def sharing\_to\_double\_clicked(self, item):  
 *"""  
 Handles the action when a user is double-clicked in the sharing list widget.* ***:param*** *item: The selected item representing the user.* ***:return****: None  
 """* user = item.text()  
 if user in self.sharing\_read\_write:  
 self.change\_permission(user, "read\_write")  
 else:  
 self.change\_permission(user, "read")  
  
 def remove\_friend(self, friend\_name):  
 *"""  
 Removes a friend from the friends list and from the friends list widget.* ***:param*** *friend\_name: The name of the friend to remove.* ***:return****: None  
 """* if friend\_name == NO\_FRIENDS:  
 return  
 print(f"Removing friend: {friend\_name}")  
 self.friends.remove(friend\_name)  
 if self.friends:  
 index = self.friends\_list\_widget.currentRow()  
 self.friends\_list\_widget.takeItem(index)  
 else:  
 self.friends\_list\_widget.clear()  
 self.friends\_list\_widget.addItem(NO\_FRIENDS)  
 send\_data(client\_socket, f"remove\_friend ||{friend\_name}")  
  
 def send\_friend\_request(self, user):  
 *"""  
 Sends a friend request to a user.* ***:param*** *user: The name of the user to send the friend request to.* ***:return****: The response from the server (OK/already sent/already friend).* ***:rtype****: str  
 """* if user not in self.friends:  
 send\_data(client\_socket, f"send\_friend\_request||{user}")  
 print(f"Sent a friend request to {user}")  
 response = receive\_data(client\_socket)  
 else:  
 response = "This user is already your friend"  
 return response  
  
 def user\_double\_clicked(self, item):  
 *"""  
 Opens a message box to send a friend request when a user is double-clicked in the user list widget.* ***:param*** *item: The selected item representing the user.* ***:return****: None  
 """* user = item.text()  
 message\_box = QMessageBox()  
 message\_box.setWindowTitle("Send Friend Request")  
 message\_box.setText(f"Send {user} a friend request?")  
  
 # Add buttons for Yes and No options  
 message\_box.addButton(QMessageBox.Yes)  
 message\_box.addButton(QMessageBox.No)  
  
 # Execute the message box and get the result  
 result = message\_box.exec\_()  
 if result == QMessageBox.Yes:  
 response = self.send\_friend\_request(user)  
 if response != "OK":  
 QMessageBox.warning(self, "Error", response)  
  
 def add\_friend(self, user):  
 *"""  
 Adds a friend to the friend list and sends command to server.* ***:param*** *user: The name of the user to add as a friend.* ***:return****: None  
 """* self.friends.append(user)  
 self.friends\_list\_widget.clear()  
 self.friends\_list\_widget.addItems(self.friends)  
 send\_data(client\_socket, f"add\_friend ||{user}")  
 print(f"added {user}")  
  
 def remove\_friend\_request(self, user):  
 *"""  
 Removes a friend request from the friend request list and sends command to server.* ***:param*** *user: The name of the user whose friend request is to be removed.* ***:return****: None  
 """* self.friend\_requests.remove(user)  
 if self.friend\_requests:  
 index = self.friend\_requests\_list\_widget.currentRow()  
 self.friend\_requests\_list\_widget.takeItem(index)  
 else:  
 self.friend\_requests\_list\_widget.clear()  
 self.friend\_requests\_list\_widget.addItem(NO\_FRIEND\_REQUESTS)  
 send\_data(client\_socket, f"rmv\_friend\_request ||{user}")  
  
 def friend\_request\_double\_clicked(self, item):  
 *"""  
 Handles the action when a friend request is double-clicked in the friend request list.* ***:param*** *item: The selected item representing the friend request.* ***:return****: None  
 """* user = item.text()  
 if user == NO\_FRIEND\_REQUESTS:  
 return  
 message\_box = QtWidgets.QMessageBox()  
 message\_box.setWindowTitle("Add Friend")  
 message\_box.setText(f"Add {user} as a friend?")  
 message\_box.addButton(QtWidgets.QPushButton("Yes"), QMessageBox.YesRole)  
 message\_box.addButton(QtWidgets.QPushButton("No"), QMessageBox.ActionRole)  
 message\_box.addButton(QtWidgets.QPushButton("Close"), QMessageBox.ActionRole)  
  
 message\_box.exec\_()  
 button = message\_box.clickedButton().text()  
  
 if button != "Close":  
 self.remove\_friend\_request(user)  
 if button == "Yes":  
 self.add\_friend(user)  
  
 def search\_users(self, search\_text):  
 *"""  
 Searches for users based on the provided search text and displays the matching results.* ***:param*** *search\_text: The text to search for in the list of users.* ***:return****: None  
 """* if not search\_text:  
 # Clear the current contents of the search results list widget  
 self.search\_results\_list.clear()  
 else:  
 self.search\_results\_list.clear()  
 # Filter the users list based on the search text  
 filtered\_users = [user for user in self.users if search\_text.lower() in user.lower()]  
  
 # Display the matching results in the search results list widget  
 self.search\_results\_list.addItems(filtered\_users)  
  
 def recursively\_add\_paths(self, folder\_path):  
 *"""  
 Recursively adds file paths within the specified folder path to the file watcher and stores their timestamps.  
 :param folder\_path: The path of the folder to recursively add file paths from.* ***:return****: None  
 """* if folder\_path == self.dir\_path:  
 watcher = self.watcher  
 else:  
 watcher = self.read\_write\_watcher  
 watcher.removePaths(watcher.files())  
 for root, dirs, files in os.walk(folder\_path):  
 for file in files:  
 path = os.path.join(root, file)  
 watcher.addPath(path)  
 self.file\_timestamps[path] = os.path.getmtime(path)  
  
 def file\_changed(self, path):  
 *"""  
 Handles the action when a file is changed.* ***:param*** *path: The path of the changed file.* ***:return****: None  
 """* if not os.path.exists(path):  
 # File doesn't exist anymore, skip processing  
 return  
  
 current\_timestamp = os.path.getmtime(path)  
 previous\_timestamp = self.file\_timestamps.get(path)  
 os.path.getmtime(path)  
 if previous\_timestamp and current\_timestamp != previous\_timestamp:  
 print("File edited:", path)  
 file\_data = File(path).data  
 # Send the file path and its data over the socket  
 if FOLDER in path:  
 relative\_path = os.path.relpath(path, DIRECTORY)  
 else:  
 relative\_path = os.path.relpath(path, self.read\_write\_path)  
 send\_data(client\_socket, f"file\_edit ||{relative\_path}")  
 send\_data(client\_socket, file\_data, send\_bytes=True)  
  
 self.file\_timestamps[path] = current\_timestamp  
  
 def on\_list\_view\_double\_clicked(self, index):  
 *"""  
 Handles the action when an item (file or folder) in the list view (user's folder) is double-clicked.* ***:param*** *index: The index of the double-clicked item.* ***:return****: None  
 """* # Check if the selected index represents a directory  
 if self.model.isDir(index):  
 # Get the path of the double-clicked directory  
 directory\_path = self.model.filePath(index)  
 # Set the root path of the model to the double-clicked directory  
 self.model.setRootPath(directory\_path)  
 # Set the root index of the list view to the new root path  
 self.list\_view.setRootIndex(self.model.index(directory\_path))  
 self.go\_back\_button.show()  
 self.directory\_history.append(directory\_path)  
 else:  
 file\_path = self.model.filePath(index)  
 open\_file(file\_path)  
  
 def on\_read\_write\_list\_view\_double\_clicked(self, index):  
 *"""  
 Handles the action when an item (file or folder) in the read-write shared folders list view is double-clicked.* ***:param*** *index: The index of the double-clicked item.* ***:return****: None  
 """* # Check if the selected index represents a directory  
 if self.read\_write\_model.isDir(index):  
 # Get the path of the double-clicked directory  
 directory\_path = self.read\_write\_model.filePath(index)  
 # Set the root path of the model to the double-clicked directory  
 self.read\_write\_model.setRootPath(directory\_path)  
 # Set the root index of the list view to the new root path  
 self.read\_write\_list\_view.setRootIndex(self.read\_write\_model.index(directory\_path))  
 self.rw\_go\_back\_button.show()  
 self.upload\_files\_shares\_button.show()  
 self.upload\_folders\_shares\_button.show()  
 self.read\_write\_directory\_history.append(directory\_path)  
 else:  
 file\_path = self.read\_write\_model.filePath(index)  
 open\_file(file\_path)  
  
 def on\_read\_only\_list\_view\_double\_clicked(self, index):  
 *"""  
 Handles the action when an item (file or folder) in the read-only shared folders list view is double-clicked.* ***:param*** *index: The index of the double-clicked item.* ***:return****: None  
 """* # Check if the selected index represents a directory  
 if self.read\_only\_model.isDir(index):  
 # Get the path of the double-clicked directory  
 directory\_path = self.read\_only\_model.filePath(index)  
 # Set the root path of the model to the double-clicked directory  
 self.read\_only\_model.setRootPath(directory\_path)  
 # Set the root index of the list view to the new root path  
 self.read\_only\_list\_view.setRootIndex(self.read\_only\_model.index(directory\_path))  
 self.r\_go\_back\_button.show()  
 self.read\_only\_directory\_history.append(directory\_path)  
 else:  
 file\_path = self.read\_only\_model.filePath(index)  
 open\_file(file\_path)  
  
 def go\_back(self):  
 *"""  
 Navigates back to the previous directory in the file system.* ***:return****: None  
 """* if len(self.directory\_history) >= 1:  
 # Remove the current directory from the history  
 self.directory\_history.pop()  
 # Get the previous directory path  
 parent\_directory\_path = self.directory\_history[-1]  
 # Set the root path of the model to the parent directory  
 self.model.setRootPath(parent\_directory\_path)  
 # Set the root index of the list view to the new root path  
 self.list\_view.setRootIndex(self.model.index(parent\_directory\_path))  
  
 # Show or hide the "Go Back" button based on the directory history  
 if len(self.directory\_history) <= 1:  
 self.go\_back\_button.hide()  
 else:  
 self.go\_back\_button.show()  
  
 def rw\_go\_back(self):  
 *"""  
 Navigates back to the previous directory in the read-write widget.* ***:return****: None  
 """* if len(self.read\_write\_directory\_history) >= 1:  
 # Remove the current directory from the history  
 self.read\_write\_directory\_history.pop()  
 # Get the previous directory path  
 parent\_directory\_path = self.read\_write\_directory\_history[-1]  
 # Set the root path of the model to the parent directory  
 self.read\_write\_model.setRootPath(parent\_directory\_path)  
 # Set the root index of the list view to the new root path  
 self.read\_write\_list\_view.setRootIndex(self.read\_write\_model.index(parent\_directory\_path))  
  
 # Show or hide the "Go Back" button based on the directory history  
 if len(self.read\_write\_directory\_history) <= 1:  
 self.rw\_go\_back\_button.hide()  
 self.upload\_files\_shares\_button.hide()  
 self.upload\_folders\_shares\_button.hide()  
 else:  
 self.rw\_go\_back\_button.show()  
 self.upload\_files\_shares\_button.show()  
 self.upload\_folders\_shares\_button.show()  
  
 def r\_go\_back(self):  
 *"""  
 Navigates back to the previous directory in the read-only widget.* ***:return****: None  
 """* if len(self.read\_only\_directory\_history) >= 1:  
 # Remove the current directory from the history  
 self.read\_only\_directory\_history.pop()  
 # Get the previous directory path  
 parent\_directory\_path = self.read\_only\_directory\_history[-1]  
 # Set the root path of the model to the parent directory  
 self.read\_only\_model.setRootPath(parent\_directory\_path)  
 # Set the root index of the list view to the new root path  
 self.read\_only\_list\_view.setRootIndex(self.read\_only\_model.index(parent\_directory\_path))  
  
 # Show or hide the "Go Back" button based on the directory history  
 if len(self.read\_only\_directory\_history) <= 1:  
 self.r\_go\_back\_button.hide()  
 else:  
 self.r\_go\_back\_button.show()  
  
 def create\_context\_menu(self, position, list\_view):  
 *"""  
 Creates a context menu at the specified position in the given list view.* ***:param*** *position: The position where the context menu should be created.* ***:param*** *list\_view: The list view where the context menu is being created.* ***:return****: None  
 """* menu = QtWidgets.QMenu()  
 temp = self.model.rootPath()  
 if list\_view == self.read\_write\_list\_view:  
 self.model.setRootPath(self.read\_write\_model.rootPath())  
 selected\_index = list\_view.indexAt(position)  
 if selected\_index.isValid():  
 # Get the selected item's path  
 item\_path = self.model.filePath(selected\_index)  
 if os.path.basename(os.path.dirname(item\_path)) == os.path.basename(self.read\_write\_path) and \  
 list\_view == self.read\_write\_list\_view:  
 # Disable context menu on users' folders in shares tab  
 return  
 if os.path.isfile(item\_path):  
 # Add "Open" action to the context menu  
 open\_action = menu.addAction("Open")  
 open\_action.triggered.connect(lambda: open\_file(item\_path))  
  
 # Add "Rename" action to the context menu  
 rename\_action = menu.addAction("Rename")  
 rename\_action.triggered.connect(lambda: self.rename\_selected\_item(item\_path))  
  
 # Add "Delete" action to the context menu  
 delete\_action = menu.addAction("Delete")  
 delete\_action.triggered.connect(lambda: self.delete\_selected\_item(item\_path))  
  
 # Add "Copy" action to the context menu  
 copy\_action = menu.addAction("Copy")  
 copy\_action.triggered.connect(lambda: self.copy\_item(item\_path))  
  
 # Add "Cut" action to the context menu  
 cut\_action = menu.addAction("Cut")  
 cut\_action.triggered.connect(lambda: self.cut\_item(item\_path))  
  
 # Add "Paste" action to the context menu  
 paste\_action = menu.addAction("Paste")  
 paste\_action.triggered.connect(lambda: self.paste\_item())  
 else:  
 # Add "Paste" action to the context menu  
 paste\_action = menu.addAction("Paste")  
 paste\_action.triggered.connect(lambda: self.paste\_item())  
  
 # Add "New" submenu to the context menu  
 new\_menu = menu.addMenu("New")  
 # Add "Create File" action to the "New" submenu  
 create\_file\_action = new\_menu.addAction("Create File")  
 create\_file\_action.triggered.connect(self.create\_new\_file)  
 # Add "Create Folder" action to the "New" submenu  
 create\_folder\_action = new\_menu.addAction("Create Folder")  
 create\_folder\_action.triggered.connect(self.create\_new\_directory)  
 # Show the context menu at the given position  
 menu.exec\_(list\_view.viewport().mapToGlobal(position))  
 self.model.setRootPath(temp)  
  
 def copy\_item(self, item\_path):  
 *"""  
 Saves the path of the item to copy.* ***:param*** *item\_path: The path of the selected item.* ***:return****: None  
 """* self.copied\_item\_path = item\_path  
 self.cut\_item\_path = None  
  
 def cut\_item(self, item\_path):  
 *"""  
 Saves the path of the item to cut.* ***:param*** *item\_path: The path of the selected item.* ***:return****: None  
 """* self.cut\_item\_path = item\_path  
 self.copied\_item\_path = None  
  
 def paste\_item(self):  
 *"""  
 Pastes the copied or cut item to the current directory and sends command to the server.* ***:return****: None  
 """* destination\_path = self.model.rootPath()  
 if self.copied\_item\_path:  
 ok = True  
 # Copy the file or folder  
 if os.path.isfile(self.copied\_item\_path):  
 try:  
 # Copy a file  
 shutil.copy2(self.copied\_item\_path, destination\_path)  
 except shutil.SameFileError:  
 ok = False  
 elif os.path.isdir(self.copied\_item\_path):  
 try:  
 # Copy a folder  
 shutil.copytree(self.copied\_item\_path,  
 os.path.join(destination\_path, os.path.basename(self.copied\_item\_path)))  
 except FileExistsError:  
 ok = False  
 if ok:  
 if FOLDER in self.copied\_item\_path:  
 rel\_copied\_item\_path = os.path.relpath(self.copied\_item\_path, DIRECTORY)  
 else:  
 rel\_copied\_item\_path = os.path.relpath(self.copied\_item\_path, self.read\_write\_path)  
 if FOLDER in destination\_path:  
 rel\_des\_item\_path = os.path.relpath(destination\_path, DIRECTORY)  
 else:  
 rel\_des\_item\_path = os.path.relpath(destination\_path, self.read\_write\_path)  
 self.recursively\_add\_paths(self.dir\_path)  
 self.recursively\_add\_paths(self.read\_write\_path)  
 send\_data(client\_socket, f"copy ||{rel\_copied\_item\_path}||{rel\_des\_item\_path}")  
 elif self.cut\_item\_path:  
 try:  
 # Move the file or folder  
 shutil.move(self.cut\_item\_path, destination\_path)  
 if FOLDER in self.cut\_item\_path:  
 rel\_cut\_item\_path = os.path.relpath(self.cut\_item\_path, DIRECTORY)  
 else:  
 rel\_cut\_item\_path = os.path.relpath(self.cut\_item\_path, self.read\_write\_path)  
 if FOLDER in destination\_path:  
 rel\_des\_item\_path = os.path.relpath(destination\_path, DIRECTORY)  
 else:  
 rel\_des\_item\_path = os.path.relpath(destination\_path, self.read\_write\_path)  
 send\_data(client\_socket, f"move ||{rel\_cut\_item\_path}||{rel\_des\_item\_path}")  
 self.copied\_item\_path = os.path.join(destination\_path, os.path.basename(self.cut\_item\_path))  
 self.cut\_item\_path = None  
 except shutil.Error:  
 pass  
 # Refresh the file system view  
 self.model.setRootPath(self.model.rootPath())  
  
 def delete\_selected\_item(self, item\_path):  
 *"""  
 Deletes the selected item at the specified path and sends command to the server.* ***:param*** *item\_path: The path of the selected item.* ***:return****: None  
 """* self.model.setRootPath(item\_path)  
 # Delete the item (file or folder)  
 delete\_item(item\_path)  
 # Refresh the file system view  
 self.model.setRootPath(self.model.rootPath())  
 if FOLDER in item\_path:  
 relative\_path = os.path.relpath(item\_path, DIRECTORY)  
 else:  
 relative\_path = os.path.relpath(item\_path, self.read\_write\_path)  
  
 send\_data(client\_socket, f"delete\_item || {relative\_path}")  
  
 def rename\_selected\_item(self, item\_path):  
 *"""  
 Renames the selected item at the specified path and sends command to the server.* ***:param*** *item\_path: The path of the selected item.* ***:return****: None  
 """* # Open a dialog to get the new name  
 self.model.setRootPath(item\_path)  
 new\_name, ok = QInputDialog.getText(self, "Rename Item", "New Name:")  
 if ok and new\_name:  
 if os.path.splitext(new\_name)[-1] == '':  
 new\_name = new\_name + os.path.splitext(item\_path)[-1]  
 # Rename the item  
 valid = rename\_item(item\_path, new\_name)  
 if not valid:  
 return  
 # Refresh the file system view  
 self.model.setRootPath(self.model.rootPath())  
 if FOLDER in item\_path:  
 relative\_path = os.path.relpath(item\_path, DIRECTORY)  
 else:  
 relative\_path = os.path.relpath(item\_path, self.read\_write\_path)  
 send\_data(client\_socket, f"rename\_item || {relative\_path} || {new\_name}")  
  
 def create\_new\_file(self):  
 *"""  
 Creates a new file in the current directory and sends command to the server.* ***:return****: None  
 """* # Open a dialog to get the new file name  
 file\_name, ok = QInputDialog.getText(self, "Create New File", "File Name:")  
 parent\_path = self.model.rootPath()  
 if ok and file\_name:  
 # Check if a directory is selected  
 if not os.path.isdir(parent\_path):  
 parent\_path = self.dir\_path  
 # Construct the path of the new file  
 new\_file\_path = os.path.join(parent\_path, file\_name)  
  
 # Check if a file or directory with the same name already exists  
 if os.path.exists(new\_file\_path):  
 QMessageBox.warning(self, "Error", "A file or directory with the same name already exists.")  
 return  
 try:  
 # Create the new file  
 with open(new\_file\_path, 'w'):  
 pass # Do nothing, just create an empty file  
 self.recursively\_add\_paths(self.dir\_path)  
 except OSError:  
 QMessageBox.warning(self, "Error", "Invalid file name.")  
 return  
  
 # Refresh the file system view  
 self.model.setRootPath(self.model.rootPath())  
 if FOLDER in new\_file\_path:  
 relative\_path = os.path.relpath(new\_file\_path, DIRECTORY)  
 self.recursively\_add\_paths(self.dir\_path)  
 else:  
 relative\_path = os.path.relpath(new\_file\_path, self.read\_write\_path)  
 self.recursively\_add\_paths(self.read\_write\_path)  
 send\_data(client\_socket, f"create\_file || {relative\_path}")  
  
 def create\_new\_directory(self):  
 *"""  
 Creates a new directory in the current directory and sends command to the server.* ***:return****: None  
 """* # Open a dialog to get the new directory name  
 dir\_name, ok = QInputDialog.getText(self, "Create New Directory", "Directory Name:")  
 parent\_path = self.model.rootPath()  
 if ok and dir\_name:  
 # Check if a directory is selected  
 if not os.path.isdir(parent\_path):  
 parent\_path = self.dir\_path  
 # Construct the path of the new directory  
 new\_dir\_path = os.path.join(parent\_path, dir\_name)  
  
 # Check if a file or directory with the same name already exists  
 if os.path.exists(new\_dir\_path):  
 QMessageBox.warning(self, "Error", "A file or directory with the same name already exists.")  
 return  
 try:  
 # Create the new directory  
 os.makedirs(new\_dir\_path)  
 except OSError:  
 QMessageBox.warning(self, "Error", "Invalid folder name.")  
 return  
  
 # Refresh the file system view  
 self.model.setRootPath(self.model.rootPath())  
 if FOLDER in new\_dir\_path:  
 relative\_path = os.path.relpath(new\_dir\_path, DIRECTORY)  
 else:  
 relative\_path = os.path.relpath(new\_dir\_path, self.read\_write\_path)  
 send\_data(client\_socket, f"create\_folder || {relative\_path}")  
  
 def upload\_folder(self, model):  
 *"""  
 Allows the user to select a folder to upload and sends it to the server.* ***:param*** *model: The model representing the file system view.* ***:return****: None  
 """* directory = QtWidgets.QFileDialog.getExistingDirectory(self, "Select Folder to Upload",  
 QtCore.QDir.homePath())  
 parent\_path = model.rootPath()  
 if directory:  
 # Check if a directory is selected  
 if not os.path.isdir(parent\_path):  
 parent\_path = self.dir\_path  
 directory = Directory(directory)  
 new\_dir = directory.create(os.path.join(parent\_path, directory.name))  
 # Refresh the file system view  
 model.setRootPath(model.rootPath())  
 serialized\_dir = dumps(new\_dir)  
 new\_dir\_path = new\_dir.path  
 if FOLDER in new\_dir\_path:  
 relative\_path = os.path.relpath(new\_dir\_path, DIRECTORY)  
 else:  
 relative\_path = os.path.relpath(new\_dir\_path, self.read\_write\_path)  
  
 # Create a thread and start the network operations  
 thread = threading.Thread(target=self.upload\_directory, args=(relative\_path, serialized\_dir,))  
 thread.start()  
  
 @staticmethod  
 def upload\_directory(relative\_path, serialized\_dir):  
 *"""  
 Uploads a directory to the server.* ***:param*** *relative\_path: The relative path of the directory.* ***:param*** *serialized\_dir: The serialized representation of the directory.* ***:return****: None  
 """* send\_data(client\_socket, f"upload\_dir ||{relative\_path}")  
 send\_data(client\_socket, serialized\_dir, send\_bytes=True)  
  
 def upload\_file(self, model):  
 *"""  
 Allows the user to select a file to upload and sends them to the server.* ***:param*** *model: The model representing the file system view.* ***:return****: None  
 """* file\_dialog = QtWidgets.QFileDialog(self, "Select File to Upload")  
 file\_dialog.setFileMode(QtWidgets.QFileDialog.ExistingFiles | QtWidgets.QFileDialog.Directory)  
 file\_dialog.setOption(QtWidgets.QFileDialog.ShowDirsOnly, False) # Show both files and directories  
 parent\_path = model.rootPath()  
 if file\_dialog.exec\_():  
 # Check if a directory is selected  
 if not os.path.isdir(parent\_path):  
 parent\_path = self.dir\_path  
 selected\_files = file\_dialog.selectedFiles()  
 for file\_path in selected\_files:  
 # Upload a single file  
 file = File(file\_path)  
 destination\_path = os.path.join(parent\_path,  
 file.name)  
 shutil.copyfile(file.path, destination\_path)  
 serialized\_file = dumps(file)  
 if FOLDER in destination\_path:  
 relative\_path = os.path.relpath(destination\_path, DIRECTORY)  
 self.recursively\_add\_paths(self.dir\_path)  
 else:  
 relative\_path = os.path.relpath(destination\_path, self.read\_write\_path)  
 self.recursively\_add\_paths(self.read\_write\_path)  
  
 send\_data(client\_socket, f"upload\_file||{relative\_path}")  
 send\_data(client\_socket, serialized\_file, send\_bytes=True)  
  
 # Refresh the file system view  
 model.setRootPath(model.rootPath())  
  
  
class LoginWindow(QMainWindow, Ui\_Login):  
 *"""  
 Represents the login window of the application. Inherits from QMainWindow and UiLogin.  
 Responsible for the login action.  
 """* def \_\_init\_\_(self):  
 *"""  
 Initializes the LoginWindow object.  
 Sets up the UI elements, connects button signals to slots, and performs necessary configurations.* ***:return****: None  
 """* super().\_\_init\_\_()  
 self.setupUi(self)  
 self.login\_fail\_label.hide()  
 self.setWindowTitle("Log In")  
 disable\_keys(self.username\_input)  
 disable\_keys(self.password\_input)  
 self.username\_input.setMaxLength(12)  
 self.password\_input.setMaxLength(12)  
 self.login\_button.clicked.connect(self.login)  
 self.signup\_button.clicked.connect(self.goto\_signup\_screen)  
  
 def login(self):  
 *"""  
 Performs the login operation.  
 Retrieves the username and password from the input fields, sends the login request to the server,  
 receives the server's response, and handles the response accordingly.* ***:return****: None  
 """* username = self.username\_input.text()  
 password = self.password\_input.text()  
 if username == '' or password == '':  
 return  
 # perform login logic here  
 print(f"Username: {username}")  
 print(f"Password: {password}")  
 # Send the username and password to the server for signup  
 msg = f"login {username} {hashlib.md5(password.encode()).hexdigest()}"  
 send\_data(client\_socket, msg)  
  
 # Receive the server's response  
 response = receive\_data(client\_socket)  
  
 # Check the server's response and show an appropriate message  
 if response == "OK":  
 # Welcome message  
 print(f"Login Successful - Welcome, {username}!")  
 send\_data(client\_socket, "download\_folder")  
 dir\_data = receive\_data(client\_socket, return\_bytes=True)  
 folder = loads(dir\_data)  
 my\_folder = folder.create(os.path.join(DIRECTORY, folder.name))  
  
 self.goto\_files(my\_folder.path)  
 elif response == "FAIL":  
 print("Login Failed - Invalid username or password")  
 self.login\_fail\_label.show()  
 else:  
 fail\_label = create\_fail\_label(self, response, QtCore.QRect(150, 260, 285, 18))  
 fail\_label.show()  
  
 @staticmethod  
 def goto\_signup\_screen():  
 *"""  
 Navigates to the signup screen.* ***:return****: None  
 """* widget.addWidget(SignupWindow())  
 widget.setCurrentIndex(widget.currentIndex() + 1)  
  
 @staticmethod  
 def goto\_files(path):  
 *"""  
 Navigates to the main application screen.* ***:param*** *path: The path to the user's directory.* ***:return****: None  
 """* widget.addWidget((MainWindow(path)))  
 widget.setCurrentIndex(widget.currentIndex() + 1)  
 widget.setFixedWidth(854)  
 widget.setFixedHeight(605)  
  
  
class SignupWindow(QMainWindow, Ui\_Signup):  
 *"""  
 Represents the signup window of the application. Inherits from QMainWindow and UiSignup.  
 """* def \_\_init\_\_(self):  
 *"""  
 Initializes the SignupWindow object.  
 Sets up the UI elements, connects button signals to slots, and performs necessary configurations.* ***:return****: None  
 """* super().\_\_init\_\_()  
 self.setupUi(self)  
 self.confirm\_fail\_label.hide()  
 self.signup\_fail\_label.hide()  
 self.setWindowTitle("Sign Up")  
 disable\_keys(self.username\_input)  
 disable\_keys(self.password\_input)  
 disable\_keys(self.confirm\_password\_input)  
  
 self.create\_account\_button.clicked.connect(self.signup)  
 self.back\_button.clicked.connect(self.go\_back)  
  
 def signup(self):  
 *"""  
 Performs the signup operation.  
 Retrieves the username, password, and confirm password from the input fields, validates the inputs,  
 sends the signup request to the server, receives the server's response, and handles the response accordingly.* ***:return****: None  
 """* username = self.username\_input.text()  
 password = self.password\_input.text()  
 confirm\_password = self.confirm\_password\_input.text()  
 if username == '' or password == '' or confirm\_password == '':  
 return  
 if password != confirm\_password:  
 self.signup\_fail\_label.hide()  
 self.confirm\_fail\_label.show()  
 return  
 print("Username:", username)  
 print("Password:", password)  
 # Create a new socket and connect to the server  
 send\_data(client\_socket, f"signup {username} {hashlib.md5(password.encode()).hexdigest()}")  
 # Receive the server's response  
 response = receive\_data(client\_socket)  
  
 # Check the server's response and show an appropriate message  
 if response == "OK":  
 print(f"Signup Successful - Welcome {username}!")  
 path = os.path.join(DIRECTORY, username)  
 os.makedirs(path)  
 self.goto\_files(path)  
  
 else:  
 print("Signup Failed - Username already exists")  
 self.confirm\_fail\_label.hide()  
 self.signup\_fail\_label.show()  
  
 @staticmethod  
 def go\_back():  
 *"""  
 Navigates back to the previous screen.* ***:return****: None  
 """* widget.removeWidget(widget.currentWidget())  
 widget.setCurrentIndex(widget.currentIndex() - 1)  
  
 @staticmethod  
 def goto\_files(path):  
 *"""  
 Navigates to the files screen.* ***:param*** *path: The path to the user's directory.* ***:return****: None  
 """* widget.addWidget((MainWindow(path)))  
 widget.setCurrentIndex(widget.currentIndex() + 1)  
 widget.setFixedWidth(854)  
 widget.setFixedHeight(605)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 client\_socket = None  
 try:  
 # Create a new socket and connect to the server  
 client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
 client\_socket.connect((SERVER\_IP, PORT))  
 public\_key, private\_key = rsa.newkeys(1024)  
 public\_partner = rsa.PublicKey.load\_pkcs1(client\_socket.recv(1024))  
 client\_socket.send(public\_key.save\_pkcs1("PEM"))  
 # Receive the encrypted symmetric key from the server  
 encrypted\_symmetric\_key = client\_socket.recv(1024)  
 # Decrypt the symmetric key using the client's private key  
 symmetric\_key = rsa.decrypt(encrypted\_symmetric\_key, private\_key)  
 # Create a Fernet instance with the symmetric key  
 fernet = Fernet(symmetric\_key)  
 app = QApplication(sys.argv)  
 widget = QtWidgets.QStackedWidget()  
 widget.addWidget(LoginWindow())  
 widget.setFixedHeight(600)  
 widget.setFixedWidth(460)  
 widget.show()  
  
 sys.exit(app.exec\_())  
 except ConnectionRefusedError:  
 print("Server is closed")  
 finally:  
 if client\_socket:  
 client\_socket.close()

## login\_window.py (Auto-Generated by Qt Designer) 6.5

from PyQt5 import QtCore, QtGui, QtWidgets  
  
  
class Ui\_Login(object):  
 def setupUi(self, MainWindow):  
 MainWindow.setObjectName("MainWindow")  
 MainWindow.resize(460, 600)  
 MainWindow.setMouseTracking(True)  
 MainWindow.setTabletTracking(True)  
 self.centralwidget = QtWidgets.QWidget(MainWindow)  
 self.centralwidget.setObjectName("centralwidget")  
 self.title\_label = QtWidgets.QLabel(self.centralwidget)  
 self.title\_label.setGeometry(QtCore.QRect(134, 60, 191, 51))  
 font = QtGui.QFont()  
 font.setPointSize(30)  
 font.setBold(True)  
 font.setItalic(False)  
 font.setWeight(75)  
 self.title\_label.setFont(font)  
 self.title\_label.setMouseTracking(False)  
 self.title\_label.setObjectName("title\_label")  
 self.username\_label = QtWidgets.QLabel(self.centralwidget)  
 self.username\_label.setGeometry(QtCore.QRect(50, 150, 101, 23))  
 font = QtGui.QFont()  
 font.setPointSize(16)  
 self.username\_label.setFont(font)  
 self.username\_label.setObjectName("username\_label")  
 self.password\_label = QtWidgets.QLabel(self.centralwidget)  
 self.password\_label.setGeometry(QtCore.QRect(50, 210, 101, 31))  
 font = QtGui.QFont()  
 font.setPointSize(16)  
 self.password\_label.setFont(font)  
 self.password\_label.setObjectName("password\_label")  
 self.username\_input = QtWidgets.QLineEdit(self.centralwidget)  
 self.username\_input.setGeometry(QtCore.QRect(180, 150, 230, 31))  
 self.username\_input.setObjectName("username\_input")  
 self.password\_input = QtWidgets.QLineEdit(self.centralwidget)  
 self.password\_input.setGeometry(QtCore.QRect(180, 210, 230, 31))  
 self.password\_input.setTabletTracking(False)  
 self.password\_input.setAutoFillBackground(False)  
 self.password\_input.setEchoMode(QtWidgets.QLineEdit.Password)  
 self.password\_input.setClearButtonEnabled(False)  
 self.password\_input.setObjectName("password\_input")  
 self.login\_button = QtWidgets.QPushButton(self.centralwidget)  
 self.login\_button.setGeometry(QtCore.QRect(180, 290, 100, 35))  
 font = QtGui.QFont()  
 font.setPointSize(16)  
 self.login\_button.setFont(font)  
 self.login\_button.setObjectName("login\_button")  
 self.create\_account\_label = QtWidgets.QLabel(self.centralwidget)  
 self.create\_account\_label.setGeometry(QtCore.QRect(50, 350, 171, 30))  
 font = QtGui.QFont()  
 font.setPointSize(12)  
 self.create\_account\_label.setFont(font)  
 self.create\_account\_label.setObjectName("create\_account\_label")  
 self.signup\_button = QtWidgets.QPushButton(self.centralwidget)  
 self.signup\_button.setGeometry(QtCore.QRect(220, 350, 100, 30))  
 font = QtGui.QFont()  
 font.setPointSize(12)  
 self.signup\_button.setFont(font)  
 self.signup\_button.setStyleSheet("color: rgb(0, 0, 255)")  
 self.signup\_button.setFlat(True)  
 self.signup\_button.setObjectName("signup\_button")  
 self.login\_fail\_label = QtWidgets.QLabel(self.centralwidget)  
 self.login\_fail\_label.setEnabled(True)  
 self.login\_fail\_label.setGeometry(QtCore.QRect(85, 260, 285, 18))  
 font = QtGui.QFont()  
 font.setPointSize(11)  
 self.login\_fail\_label.setFont(font)  
 self.login\_fail\_label.setStyleSheet("color:rgb(255, 0, 0)")  
 self.login\_fail\_label.setObjectName("login\_fail\_label")  
 MainWindow.setCentralWidget(self.centralwidget)  
  
 self.retranslateUi(MainWindow)  
 QtCore.QMetaObject.connectSlotsByName(MainWindow)  
  
 def retranslateUi(self, MainWindow):  
 \_translate = QtCore.QCoreApplication.translate  
 MainWindow.setWindowTitle(\_translate("MainWindow", "MainWindow"))  
 self.title\_label.setText(\_translate("MainWindow", "FileSpace"))  
 self.username\_label.setText(\_translate("MainWindow", "Username"))  
 self.password\_label.setText(\_translate("MainWindow", "Password"))  
 self.login\_button.setText(\_translate("MainWindow", "Log In"))  
 self.create\_account\_label.setText(\_translate("MainWindow", "Don\'t have an account?"))  
 self.signup\_button.setText(\_translate("MainWindow", "Sign Up"))  
 self.login\_fail\_label.setText(\_translate("MainWindow", "Login Failed - Invalid username or password"))

## signup\_window.py (Auto-Generated by Qt Designer) 6.6

from PyQt5 import QtCore, QtGui, QtWidgets  
  
  
class Ui\_Signup(object):  
 def setupUi(self, MainWindow):  
 MainWindow.setObjectName("MainWindow")  
 MainWindow.setEnabled(True)  
 MainWindow.resize(460, 600)  
 self.centralwidget = QtWidgets.QWidget(MainWindow)  
 self.centralwidget.setObjectName("centralwidget")  
 self.title\_label = QtWidgets.QLabel(self.centralwidget)  
 self.title\_label.setGeometry(QtCore.QRect(134, 60, 191, 51))  
 font = QtGui.QFont()  
 font.setPointSize(30)  
 font.setBold(True)  
 font.setItalic(False)  
 font.setWeight(75)  
 self.title\_label.setFont(font)  
 self.title\_label.setMouseTracking(False)  
 self.title\_label.setObjectName("title\_label")  
 self.username\_label = QtWidgets.QLabel(self.centralwidget)  
 self.username\_label.setGeometry(QtCore.QRect(50, 150, 101, 23))  
 font = QtGui.QFont()  
 font.setPointSize(16)  
 self.username\_label.setFont(font)  
 self.username\_label.setObjectName("username\_label")  
 self.username\_input = QtWidgets.QLineEdit(self.centralwidget)  
 self.username\_input.setGeometry(QtCore.QRect(180, 150, 230, 31))  
 self.username\_input.setObjectName("username\_input")  
 self.password\_input = QtWidgets.QLineEdit(self.centralwidget)  
 self.password\_input.setGeometry(QtCore.QRect(209, 210, 201, 31))  
 self.password\_input.setTabletTracking(False)  
 self.password\_input.setAutoFillBackground(False)  
 self.password\_input.setEchoMode(QtWidgets.QLineEdit.Password)  
 self.password\_input.setClearButtonEnabled(False)  
 self.password\_input.setObjectName("password\_input")  
 self.password\_label = QtWidgets.QLabel(self.centralwidget)  
 self.password\_label.setGeometry(QtCore.QRect(20, 210, 161, 31))  
 font = QtGui.QFont()  
 font.setPointSize(16)  
 self.password\_label.setFont(font)  
 self.password\_label.setObjectName("password\_label")  
 self.confirm\_password\_label = QtWidgets.QLabel(self.centralwidget)  
 self.confirm\_password\_label.setGeometry(QtCore.QRect(20, 270, 171, 31))  
 font = QtGui.QFont()  
 font.setPointSize(16)  
 self.confirm\_password\_label.setFont(font)  
 self.confirm\_password\_label.setObjectName("confirm\_password\_label")  
 self.confirm\_password\_input = QtWidgets.QLineEdit(self.centralwidget)  
 self.confirm\_password\_input.setGeometry(QtCore.QRect(209, 270, 201, 31))  
 self.confirm\_password\_input.setTabletTracking(False)  
 self.confirm\_password\_input.setAutoFillBackground(False)  
 self.confirm\_password\_input.setText("")  
 self.confirm\_password\_input.setEchoMode(QtWidgets.QLineEdit.Password)  
 self.confirm\_password\_input.setClearButtonEnabled(False)  
 self.confirm\_password\_input.setObjectName("confirm\_password\_input")  
 self.create\_account\_button = QtWidgets.QPushButton(self.centralwidget)  
 self.create\_account\_button.setGeometry(QtCore.QRect(150, 350, 151, 35))  
 font = QtGui.QFont()  
 font.setPointSize(16)  
 self.create\_account\_button.setFont(font)  
 self.create\_account\_button.setObjectName("create\_account\_button")  
 self.back\_button = QtWidgets.QPushButton(self.centralwidget)  
 self.back\_button.setGeometry(QtCore.QRect(0, 0, 75, 23))  
 self.back\_button.setObjectName("pushButton")  
 self.signup\_fail\_label = QtWidgets.QLabel(self.centralwidget)  
 self.signup\_fail\_label.setEnabled(True)  
 self.signup\_fail\_label.setGeometry(QtCore.QRect(95, 320, 285, 18))  
 font = QtGui.QFont()  
 font.setPointSize(11)  
 self.signup\_fail\_label.setFont(font)  
 self.signup\_fail\_label.setStyleSheet("color:rgb(255, 0, 0)")  
 self.signup\_fail\_label.setObjectName("signup\_fail\_label")  
 self.confirm\_fail\_label = QtWidgets.QLabel(self.centralwidget)  
 self.confirm\_fail\_label.setEnabled(True)  
 self.confirm\_fail\_label.setGeometry(QtCore.QRect(95, 320, 285, 18))  
 font = QtGui.QFont()  
 font.setPointSize(11)  
 self.confirm\_fail\_label.setFont(font)  
 self.confirm\_fail\_label.setStyleSheet("color:rgb(255, 0, 0)")  
 self.confirm\_fail\_label.setObjectName("confirm\_fail\_label")  
 MainWindow.setCentralWidget(self.centralwidget)  
  
 self.retranslateUi(MainWindow)  
 QtCore.QMetaObject.connectSlotsByName(MainWindow)  
  
 def retranslateUi(self, MainWindow):  
 \_translate = QtCore.QCoreApplication.translate  
 MainWindow.setWindowTitle(\_translate("MainWindow", "MainWindow"))  
 self.title\_label.setText(\_translate("MainWindow", "FileSpace"))  
 self.username\_label.setText(\_translate("MainWindow", "Username"))  
 self.password\_label.setText(\_translate("MainWindow", "Create Password"))  
 self.confirm\_password\_label.setText(\_translate("MainWindow", "Confirm Password"))  
 self.create\_account\_button.setText(\_translate("MainWindow", "Create Account"))  
 self.back\_button.setText(\_translate("MainWindow", "Back"))  
 self.signup\_fail\_label.setText(\_translate("MainWindow", "Signup Failed - Username already exists"))  
 self.confirm\_fail\_label.setText(\_translate("MainWindow", "Signup Failed - Couldn\'t confirm password"))

## main\_window.py (Auto-Generated by Qt Designer) 6.7

from PyQt5 import QtCore, QtGui, QtWidgets  
  
  
class Ui\_MainWindow(object):  
 def setupUi(self, MainWindow):  
 MainWindow.setObjectName("MainWindow")  
 MainWindow.resize(854, 605)  
 self.horizontalLayout = QtWidgets.QHBoxLayout(MainWindow)  
 self.horizontalLayout.setObjectName("horizontalLayout")  
 self.tabs = QtWidgets.QTabWidget(MainWindow)  
 self.tabs.setEnabled(True)  
 sizePolicy = QtWidgets.QSizePolicy(QtWidgets.QSizePolicy.MinimumExpanding, QtWidgets.QSizePolicy.MinimumExpanding)  
 sizePolicy.setHorizontalStretch(0)  
 sizePolicy.setVerticalStretch(0)  
 sizePolicy.setHeightForWidth(self.tabs.sizePolicy().hasHeightForWidth())  
 self.tabs.setSizePolicy(sizePolicy)  
 self.tabs.setObjectName("tabs")  
 self.myfolder\_tab = QtWidgets.QWidget()  
 self.myfolder\_tab.setObjectName("myfolder\_tab")  
 self.upload\_files\_button = QtWidgets.QPushButton(self.myfolder\_tab)  
 self.upload\_files\_button.setGeometry(QtCore.QRect(730, 80, 81, 23))  
 self.upload\_files\_button.setObjectName("upload\_files\_button")  
 self.upload\_folders\_button = QtWidgets.QPushButton(self.myfolder\_tab)  
 self.upload\_folders\_button.setGeometry(QtCore.QRect(730, 120, 81, 23))  
 self.upload\_folders\_button.setObjectName("upload\_folders\_button")  
 self.go\_back\_button = QtWidgets.QPushButton(self.myfolder\_tab)  
 self.go\_back\_button.setGeometry(QtCore.QRect(5, 10, 61, 23))  
 self.go\_back\_button.setObjectName("go\_back\_button")  
 self.list\_view = QtWidgets.QListView(self.myfolder\_tab)  
 self.list\_view.setGeometry(QtCore.QRect(20, 40, 691, 501))  
 self.list\_view.setAcceptDrops(True)  
 self.list\_view.setDragEnabled(False)  
 self.list\_view.setDragDropOverwriteMode(False)  
 self.list\_view.setDragDropMode(QtWidgets.QAbstractItemView.InternalMove)  
 self.list\_view.setSelectionMode(QtWidgets.QAbstractItemView.ExtendedSelection)  
 self.list\_view.setIconSize(QtCore.QSize(0, 0))  
 self.list\_view.setViewMode(QtWidgets.QListView.IconMode)  
 self.list\_view.setObjectName("list\_view")  
 self.my\_filespace\_label = QtWidgets.QLabel(self.myfolder\_tab)  
 self.my\_filespace\_label.setGeometry(QtCore.QRect(300, 10, 111, 21))  
 font = QtGui.QFont()  
 font.setPointSize(14)  
 self.my\_filespace\_label.setFont(font)  
 self.my\_filespace\_label.setObjectName("my\_filespace\_label")  
 self.tabs.addTab(self.myfolder\_tab, "")  
 self.friends\_tab = QtWidgets.QWidget()  
 self.friends\_tab.setObjectName("friends\_tab")  
 self.friends\_list\_widget = QtWidgets.QListWidget(self.friends\_tab)  
 self.friends\_list\_widget.setGeometry(QtCore.QRect(10, 30, 256, 511))  
 font = QtGui.QFont()  
 font.setPointSize(14)  
 self.friends\_list\_widget.setFont(font)  
 self.friends\_list\_widget.setAlternatingRowColors(True)  
 self.friends\_list\_widget.setObjectName("friends\_list\_widget")  
 self.friends\_label = QtWidgets.QLabel(self.friends\_tab)  
 self.friends\_label.setGeometry(QtCore.QRect(100, 3, 61, 21))  
 font = QtGui.QFont()  
 font.setPointSize(14)  
 self.friends\_label.setFont(font)  
 self.friends\_label.setObjectName("friends\_label")  
 self.friend\_requests\_list\_widget = QtWidgets.QListWidget(self.friends\_tab)  
 self.friend\_requests\_list\_widget.setGeometry(QtCore.QRect(280, 30, 256, 241))  
 font = QtGui.QFont()  
 font.setPointSize(14)  
 self.friend\_requests\_list\_widget.setFont(font)  
 self.friend\_requests\_list\_widget.setSizeAdjustPolicy(QtWidgets.QAbstractScrollArea.AdjustIgnored)  
 self.friend\_requests\_list\_widget.setAlternatingRowColors(True)  
 self.friend\_requests\_list\_widget.setResizeMode(QtWidgets.QListView.Fixed)  
 self.friend\_requests\_list\_widget.setObjectName("friend\_requests\_list\_widget")  
 self.friend\_requests\_label = QtWidgets.QLabel(self.friends\_tab)  
 self.friend\_requests\_label.setGeometry(QtCore.QRect(330, 3, 141, 21))  
 font = QtGui.QFont()  
 font.setPointSize(14)  
 self.friend\_requests\_label.setFont(font)  
 self.friend\_requests\_label.setObjectName("friend\_requests\_label")  
 self.search\_results\_list = QtWidgets.QListWidget(self.friends\_tab)  
 self.search\_results\_list.setGeometry(QtCore.QRect(550, 30, 256, 241))  
 font = QtGui.QFont()  
 font.setPointSize(14)  
 self.search\_results\_list.setFont(font)  
 self.search\_results\_list.setAlternatingRowColors(True)  
 self.search\_results\_list.setObjectName("search\_results\_list")  
 self.search\_users\_label = QtWidgets.QLabel(self.friends\_tab)  
 self.search\_users\_label.setGeometry(QtCore.QRect(550, 3, 91, 21))  
 font = QtGui.QFont()  
 font.setPointSize(14)  
 self.search\_users\_label.setFont(font)  
 self.search\_users\_label.setObjectName("search\_users\_label")  
 self.search\_bar = QtWidgets.QLineEdit(self.friends\_tab)  
 self.search\_bar.setGeometry(QtCore.QRect(640, 5, 161, 20))  
 self.search\_bar.setText("")  
 self.search\_bar.setObjectName("search\_bar")  
 self.sharing\_read\_write\_label = QtWidgets.QLabel(self.friends\_tab)  
 self.sharing\_read\_write\_label.setGeometry(QtCore.QRect(320, 275, 171, 31))  
 font = QtGui.QFont()  
 font.setPointSize(14)  
 self.sharing\_read\_write\_label.setFont(font)  
 self.sharing\_read\_write\_label.setObjectName("sharing\_read\_write\_label")  
 self.sharing\_read\_write\_list\_widget = QtWidgets.QListWidget(self.friends\_tab)  
 self.sharing\_read\_write\_list\_widget.setGeometry(QtCore.QRect(280, 310, 256, 231))  
 font = QtGui.QFont()  
 font.setPointSize(14)  
 self.sharing\_read\_write\_list\_widget.setFont(font)  
 self.sharing\_read\_write\_list\_widget.setAlternatingRowColors(True)  
 self.sharing\_read\_write\_list\_widget.setObjectName("sharing\_read\_write\_list\_widget")  
 self.sharing\_read\_only\_list\_widget = QtWidgets.QListWidget(self.friends\_tab)  
 self.sharing\_read\_only\_list\_widget.setGeometry(QtCore.QRect(550, 310, 256, 231))  
 font = QtGui.QFont()  
 font.setPointSize(14)  
 self.sharing\_read\_only\_list\_widget.setFont(font)  
 self.sharing\_read\_only\_list\_widget.setAlternatingRowColors(True)  
 self.sharing\_read\_only\_list\_widget.setObjectName("sharing\_read\_only\_list\_widget")  
 self.sharing\_read\_only\_label = QtWidgets.QLabel(self.friends\_tab)  
 self.sharing\_read\_only\_label.setGeometry(QtCore.QRect(590, 275, 161, 31))  
 font = QtGui.QFont()  
 font.setPointSize(14)  
 self.sharing\_read\_only\_label.setFont(font)  
 self.sharing\_read\_only\_label.setObjectName("sharing\_read\_only\_label")  
 self.tabs.addTab(self.friends\_tab, "")  
 self.shares\_tab = QtWidgets.QWidget()  
 self.shares\_tab.setObjectName("shares\_tab")  
 self.read\_write\_list\_view = QtWidgets.QListView(self.shares\_tab)  
 self.read\_write\_list\_view.setGeometry(QtCore.QRect(10, 40, 390, 471))  
 self.read\_write\_list\_view.setViewMode(QtWidgets.QListView.IconMode)  
 self.read\_write\_list\_view.setObjectName("read\_write\_list\_view")  
 self.read\_write\_label = QtWidgets.QLabel(self.shares\_tab)  
 self.read\_write\_label.setGeometry(QtCore.QRect(140, 5, 141, 31))  
 font = QtGui.QFont()  
 font.setPointSize(14)  
 self.read\_write\_label.setFont(font)  
 self.read\_write\_label.setObjectName("read\_write\_label")  
 self.read\_only\_list\_view = QtWidgets.QListView(self.shares\_tab)  
 self.read\_only\_list\_view.setGeometry(QtCore.QRect(410, 40, 400, 471))  
 self.read\_only\_list\_view.setViewMode(QtWidgets.QListView.IconMode)  
 self.read\_only\_list\_view.setObjectName("read\_only\_list\_view")  
 self.read\_only\_label = QtWidgets.QLabel(self.shares\_tab)  
 self.read\_only\_label.setGeometry(QtCore.QRect(560, 5, 91, 31))  
 font = QtGui.QFont()  
 font.setPointSize(14)  
 self.read\_only\_label.setFont(font)  
 self.read\_only\_label.setObjectName("read\_only\_label")  
 self.rw\_go\_back\_button = QtWidgets.QPushButton(self.shares\_tab)  
 self.rw\_go\_back\_button.setGeometry(QtCore.QRect(10, 10, 61, 23))  
 self.rw\_go\_back\_button.setObjectName("rw\_go\_back\_button")  
 self.r\_go\_back\_button = QtWidgets.QPushButton(self.shares\_tab)  
 self.r\_go\_back\_button.setGeometry(QtCore.QRect(410, 10, 61, 23))  
 self.r\_go\_back\_button.setObjectName("r\_go\_back\_button")  
 self.upload\_folders\_shares\_button = QtWidgets.QPushButton(self.shares\_tab)  
 self.upload\_folders\_shares\_button.setGeometry(QtCore.QRect(200, 520, 81, 24))  
 self.upload\_folders\_shares\_button.setObjectName("upload\_folders\_shares\_button")  
 self.upload\_files\_shares\_button = QtWidgets.QPushButton(self.shares\_tab)  
 self.upload\_files\_shares\_button.setGeometry(QtCore.QRect(94, 520, 81, 24))  
 self.upload\_files\_shares\_button.setObjectName("upload\_files\_shares\_button")  
 self.read\_write\_label.raise\_()  
 self.read\_write\_list\_view.raise\_()  
 self.read\_only\_list\_view.raise\_()  
 self.read\_only\_label.raise\_()  
 self.rw\_go\_back\_button.raise\_()  
 self.r\_go\_back\_button.raise\_()  
 self.upload\_folders\_shares\_button.raise\_()  
 self.upload\_files\_shares\_button.raise\_()  
 self.tabs.addTab(self.shares\_tab, "")  
 self.horizontalLayout.addWidget(self.tabs)  
  
 self.retranslateUi(MainWindow)  
 self.tabs.setCurrentIndex(2)  
 QtCore.QMetaObject.connectSlotsByName(MainWindow)  
  
 def retranslateUi(self, MainWindow):  
 \_translate = QtCore.QCoreApplication.translate  
 MainWindow.setWindowTitle(\_translate("MainWindow", "FileSpace"))  
 self.upload\_files\_button.setText(\_translate("MainWindow", "Upload Files"))  
 self.upload\_folders\_button.setText(\_translate("MainWindow", "Upload Folders"))  
 self.go\_back\_button.setText(\_translate("MainWindow", "Back"))  
 self.my\_filespace\_label.setText(\_translate("MainWindow", "My FileSpace"))  
 self.tabs.setTabText(self.tabs.indexOf(self.myfolder\_tab), \_translate("MainWindow", "My FileSpace"))  
 self.friends\_label.setText(\_translate("MainWindow", "Friends"))  
 self.friend\_requests\_label.setText(\_translate("MainWindow", "Friend Requests"))  
 self.search\_users\_label.setText(\_translate("MainWindow", "Find Users"))  
 self.sharing\_read\_write\_label.setText(\_translate("MainWindow", "Sharing Read-Write"))  
 self.sharing\_read\_only\_label.setText(\_translate("MainWindow", "Sharing Read-Only"))  
 self.tabs.setTabText(self.tabs.indexOf(self.friends\_tab), \_translate("MainWindow", "Friends"))  
 self.read\_write\_label.setText(\_translate("MainWindow", "Read And Write"))  
 self.read\_only\_label.setText(\_translate("MainWindow", "Read Only"))  
 self.rw\_go\_back\_button.setText(\_translate("MainWindow", "Back"))  
 self.r\_go\_back\_button.setText(\_translate("MainWindow", "Back"))  
 self.upload\_folders\_shares\_button.setText(\_translate("MainWindow", "Upload Folders"))  
 self.upload\_files\_shares\_button.setText(\_translate("MainWindow", "Upload Files"))  
 self.tabs.setTabText(self.tabs.indexOf(self.shares\_tab), \_translate("MainWindow", "Shares"))