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

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: מציעה אחסון מאובטח, שיתוף ועריכה של כמה משתמשים על אותו קובץ בו זמנית.

Google Drive: מציעה אחסון ושיתוף בענן שמשתלבים עם שירותי גוגל אחרים כמו *Gmail* ו-*Google Docs*.

Microsoft OneDrive: 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).

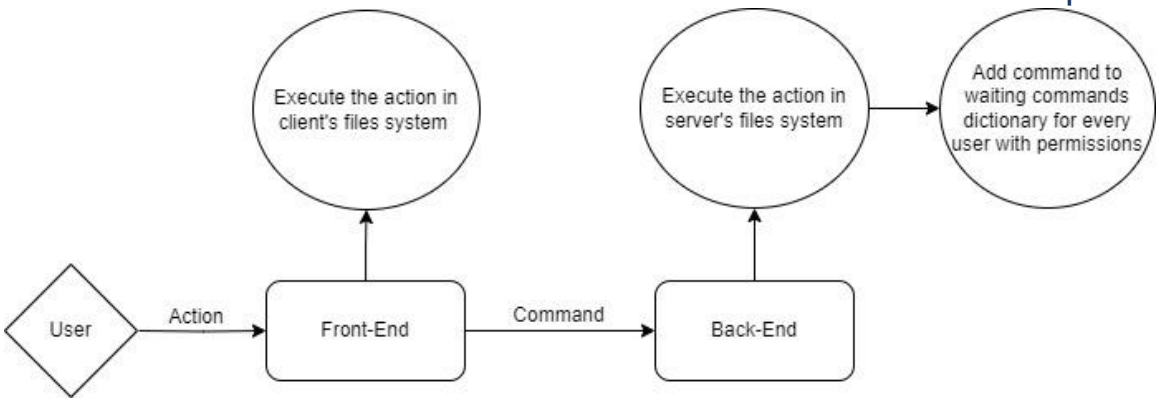
2. מבנה/ ארכיטקטורת הפרויקט

2.1 תיאור הארכיטקטורה

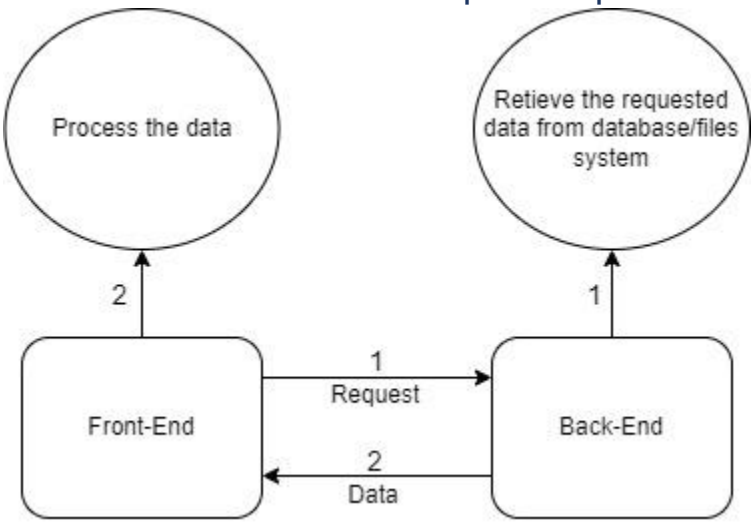
2.1.1 תיאור מילולי

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



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.

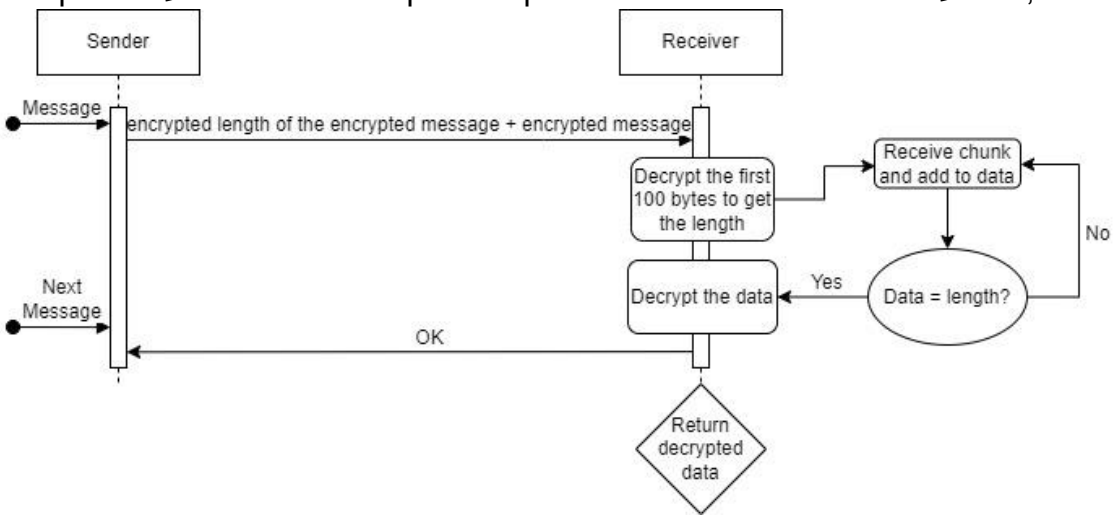
2.5 תיאור סביבת הפיתוח

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.

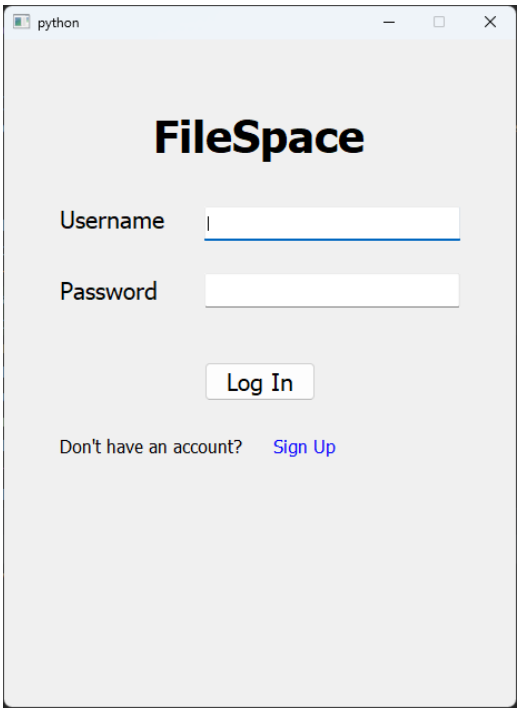


2.7 תיאור מסכי המערכת

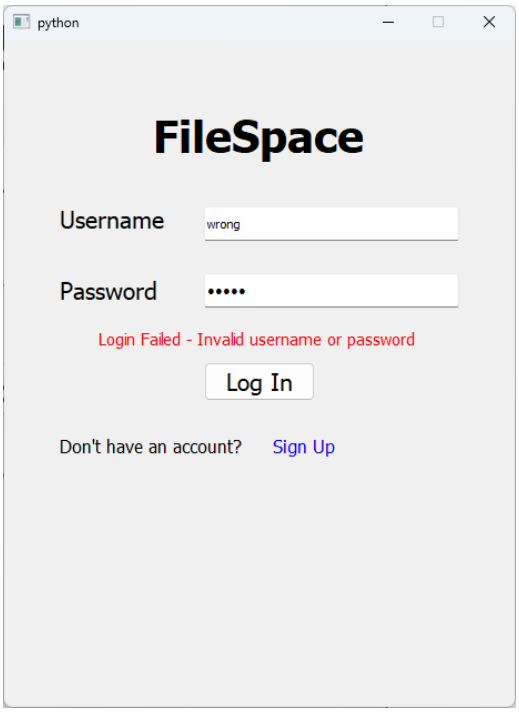
2.7.1 LoginWindow

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:



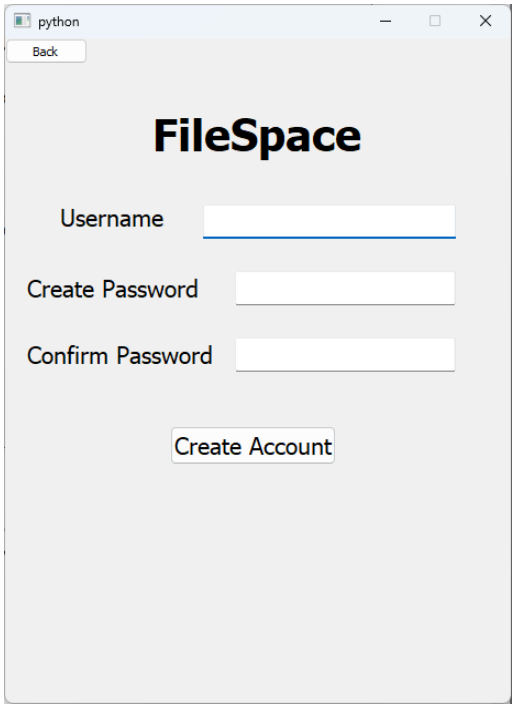
המסך לאחר הכנסת פרטים שגויים:



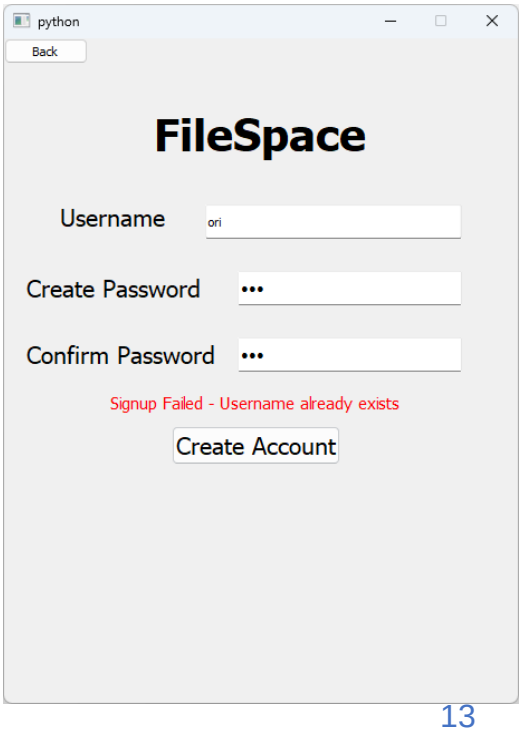
SignupWindow 2.7.2

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.

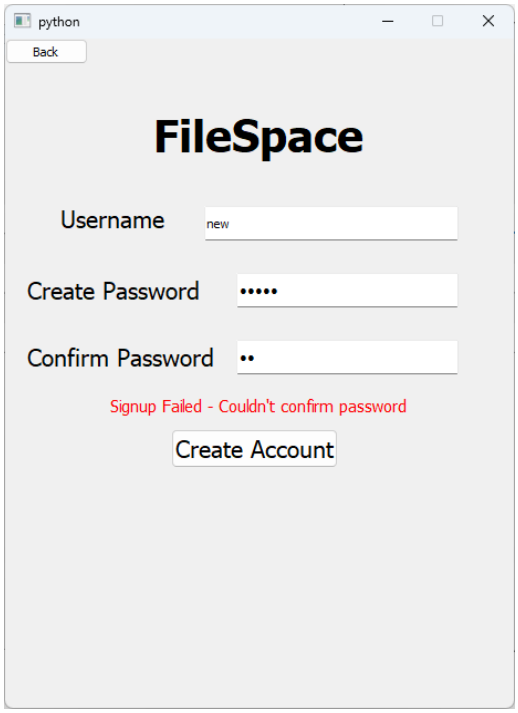
מסך ההרשמה ההתחלתי:



לאחר הכנסת שם משתמש קיים:

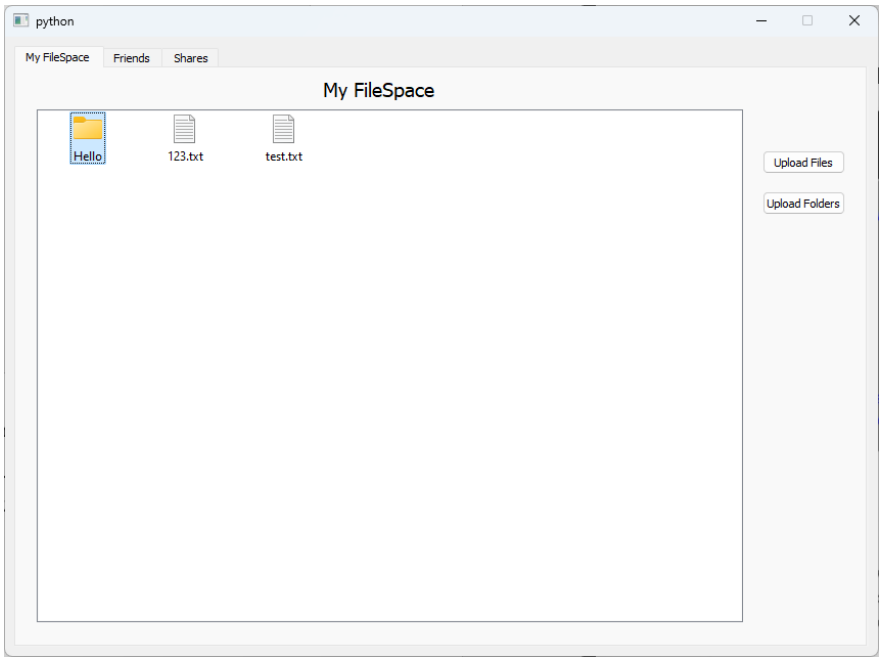


לאחר הכנסת שתי סיסמאות שונות:

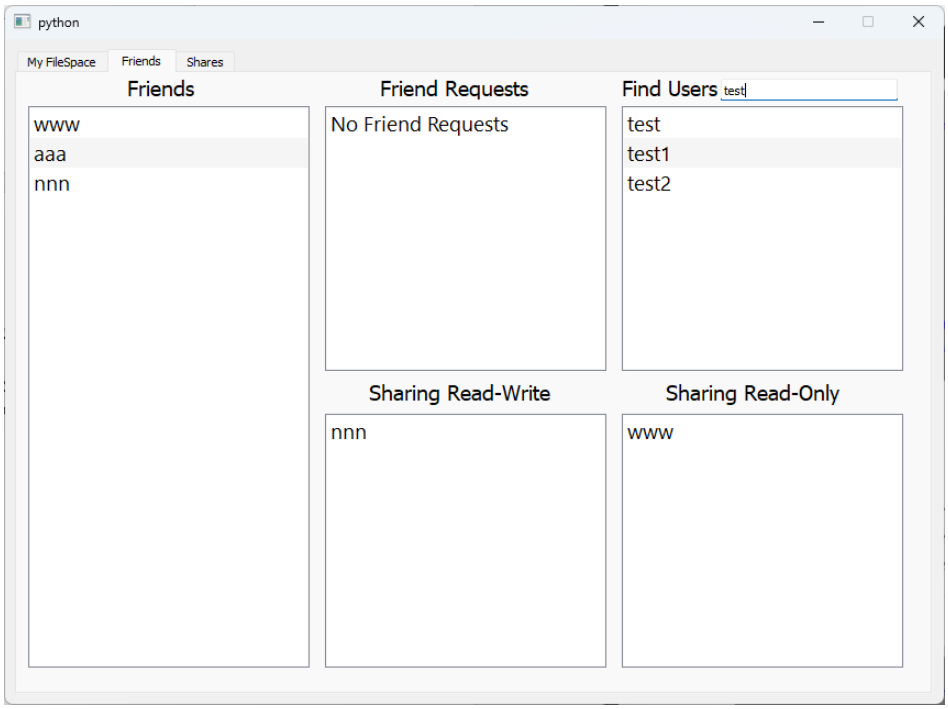


MainWindow 2.7.3

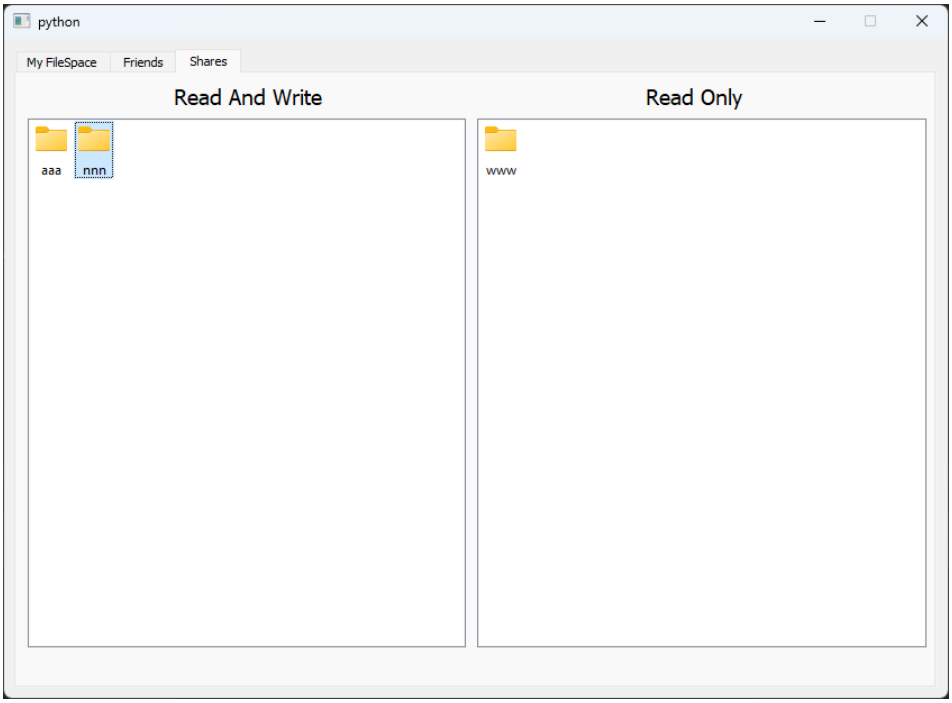
- My FileSpace Tab: כרטיסייה שמציגה את תכולת תיקיית המשתמש. המשתמש יכול לבצע פעולות על הפריטים באמצעות קליק ימני על הפריט ובחירת הפעולה, להיכנס לתיקיה באמצעות דאבל קליק עליה, לפתוח קובץ באמצעות דאבל קליק עליו או בחירת קובץ או תיקייה להעלות בלחיצה על הכפתורים בצד ימין. אחרי שנכנסים לתיקייה אפשר לצאת ממנה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



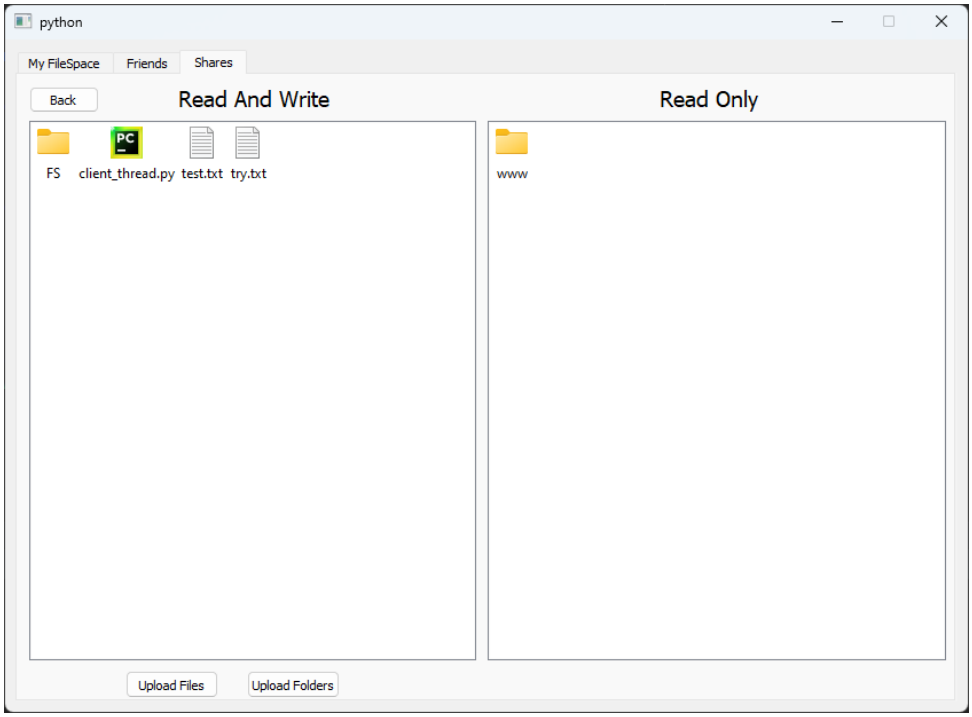
- Friend Tab: כרטיסייה שמאפשרת לחפש משתמשים ולשלוח להם בקשת חברות, מציגה את בקשות החברות ששלחו למשתמש, חברי המשתמש, חברים שהוא משתף איתם עם הרשאות קריאה וכתיבה ו חברים שהוא משתף איתם עם הרשאות קריאה בלבד. המשתמש יכול גם לתת/לשנות/להסיר הרשאות לחברים ולהסיר חברים.



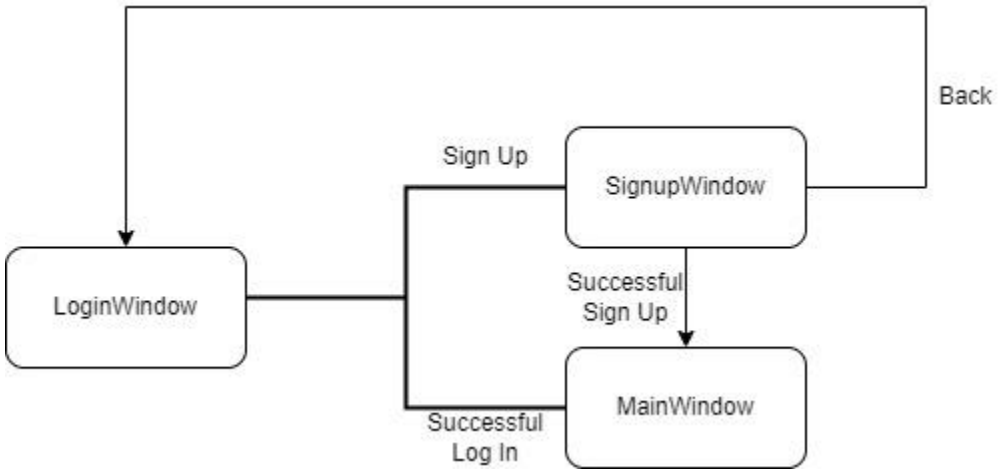
- Shares Tab: כרטיסייה שמציגה את התיקיות שמשתפים עם המשתמש (מחולקות לפי ההרשאות). המשתמש יכול לבצע על הפריטים שבתוך התיקיות של המשתמשים שיש לו הרשאות קריאה וכתיבה אליהן את כל הפעולות שהוא יכול לבצע על התיקייה שלו. בתיקיות שיש לו אליהן הרשאות קריאה הוא רק יכול לראות את השינויים שמתבצעים.



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: "cb962ac59075b964b07152d234b70202"
- ■ 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

3.2.1 File

- 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.

3.2.2 Folder

- 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.

3.2.3 LoginWindow

- 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.

3.2.4 SignupWindow

- 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.

3.2.5 MainWindow

- 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.

3.2.6 ClientThread

- 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-Fernet להצפנה 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(shar
ed_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(shar
ed_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.

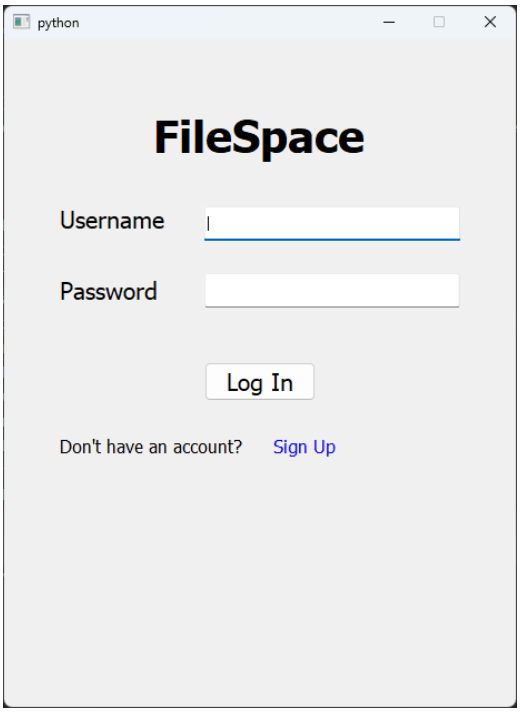
4.2 התקנת המערכת

דרושה סביבת Python וחיבור לאינטרנט כדי להריץ את המערכת. במחשב השרת דרושים הקבצים server.py, file_classes.py ו- client_thread.py. במחשב הלקוח דרושים הקבצים client.py, file_classes.py, login_window.py, signup_window.py, main_window.py. בנוסף, צריך Database FileSpace של MySQL בשרת ולעדכן את הקונפיגורציה ב-server.py ו-client_thread.py.

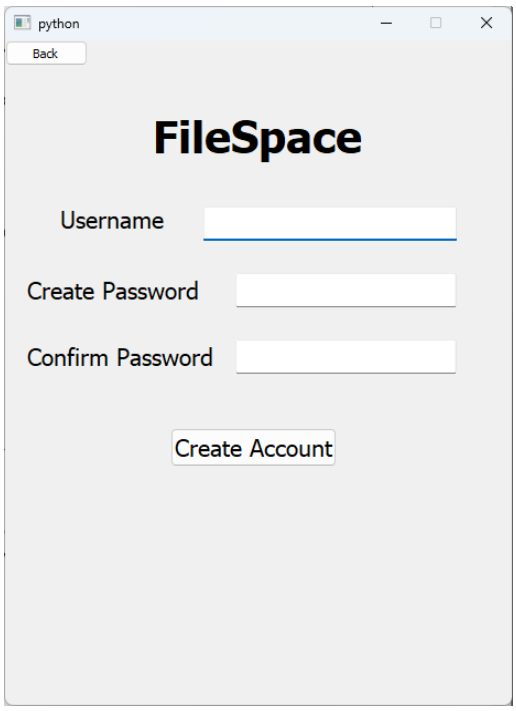
4.3 משתמשי המערכת

- 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.

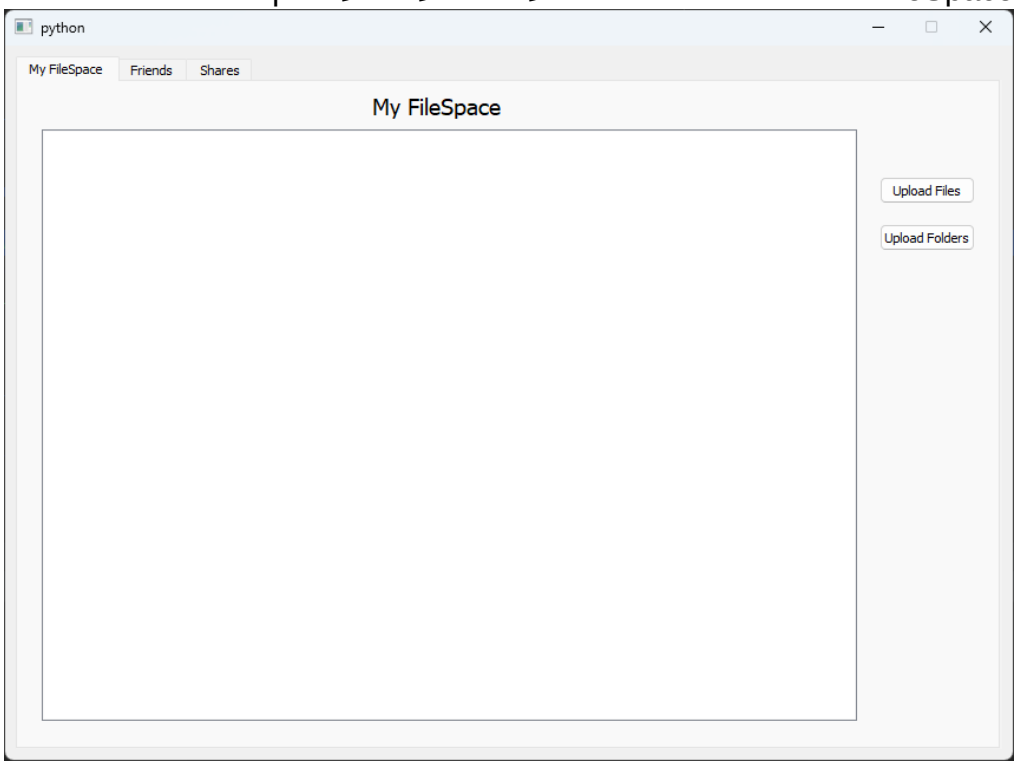
לאחר הרצת client.py יופיע המסך ההתחלתי. כדי ליצור משתמש יש ללחוץ על "Sign Up":



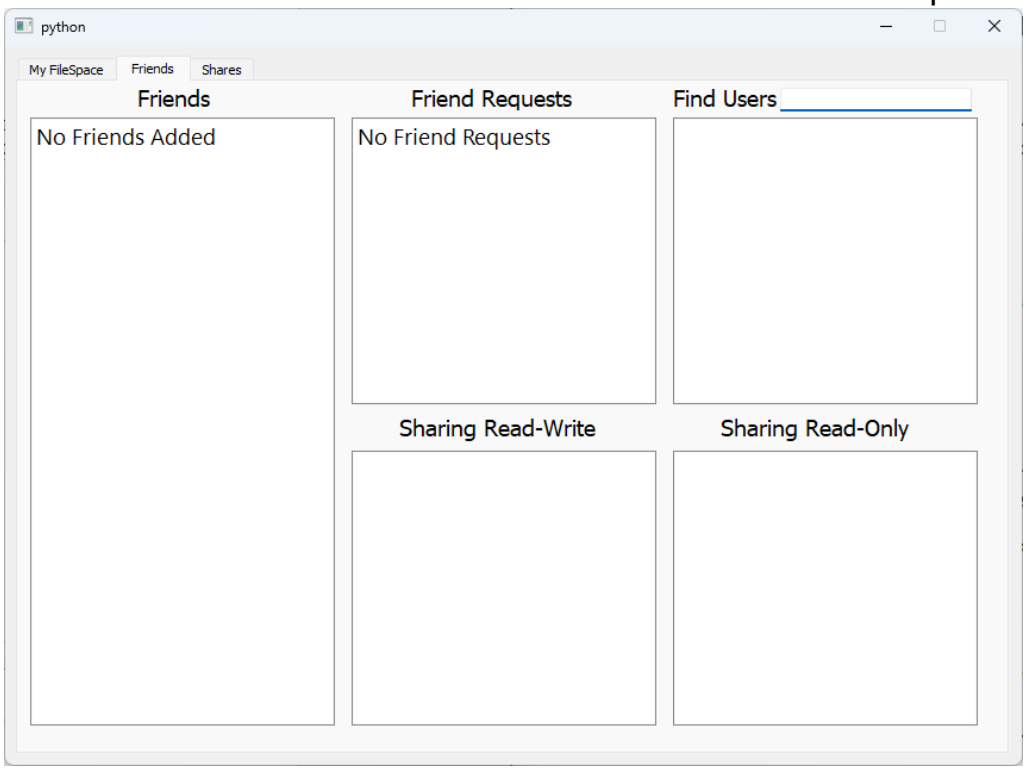
לאחר מכן, המסך ישתנה למסך ההרשמה ההתחלתי:



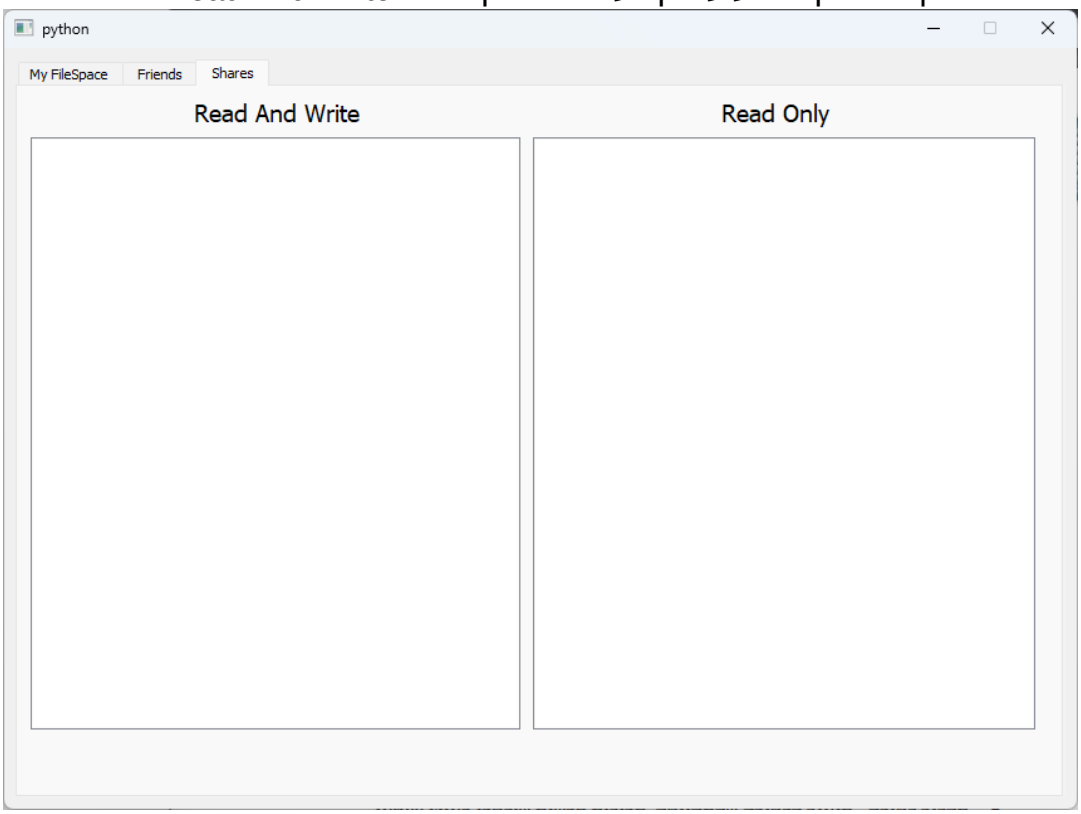
לאחר הרשמה או התחברות מוצלחת יופיע המסך הראשי של האפליקציה. כרטיסיית " My FileSpace": בכרטיסייה זו אפשר לבצע את הפעולות על תיקיית המשתמש.



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

6.1 server.py

```
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_c
licked)

self.friend_requests_list_widget.itemDoubleClicked.connect(self.friend_
request_double_clicked)

self.sharing_read_write_list_widget.itemDoubleClicked.connect(self.shar
ing_to_double_clicked)

self.sharing_read_only_list_widget.itemDoubleClicked.connect(self.shari
ng_to_double_clicked)
        self.search_bar.textChanged.connect(self.search_users)

self.search_results_list.itemDoubleClicked.connect(self.user_double_cli
cked)
        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.r
ead_only_path))
```

```
self.read_only_list_view.doubleClicked.connect(self.on_read_only_list_v
iew_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.CustomContextM
enu)
        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.in
dex(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.inde
x(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.in
dex(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.inde
x(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(dire
ctory_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(direct
ory_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(pare
nt_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.ExtendedSel
ection)
        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.QAbstrac
tScrollArea.AdjustIgnored)
        self.friend_requests_list_widget.setAlternatingRowColors(True)

self.friend_requests_list_widget.setResizeMode(QtWidgets.QListView.Fixe
d)
```

```
self.friend_requests_list_widget.setObjectName("friend_requests_list_wi
dget")
        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_l
ist_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_lis
t_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_butt
on")
        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"))
```