

בית הספר הארצי להנדסאים
חיפה, קריית הטכניון

פרויקט גמר

המגמה להנדסת תוכנה

ה טכניוון
מינהל – מכון טכנולוגי לישראל
הטכניון אגף בכיר להכשרה ולפיתוח כה – אדם
מינה"ט – המכון הממשלתי להכשרה טכנולוגית ובעניהם

הנושא: אפליקציה לניהול משימות בקבוצה



שלישי:	אליאן מלך	מנחה:	שיי קויל
מחזור:	47	תאריך:	09/2023

בית הספר הארצי להנדסאים

שם הסטודנט:

אלון מלב

שם המנחה: שלி קויל

מגמה וכיתה: מחשבים

**נושא פרויקט הגמר:
אפליקציה לניהול משימות בקובץ**

תאור הפרויקט:

יצירת אפליקציה שותה למשימות קבוצתיות הוא רעיון שנובע מהתקלות עם הרבה שימוש שעשינו לבצע בתוך קבוצה אם זה בתוך עבודה או בתוך משפחה או אפילו בתור בני אדם עצמאיים, لكن הרעיון הוא לייצר אפליקציה שתאפשר לנוהלים קבוצות אנשים את המשימות הפרטיות והקבוצתיות שלהם תוך כדי התmeshקות עם המיקום במכשיר כדי לשולח התראות על שימושות לביצוע על פי מיקום המשימה.

אם זה לעובדות כמו שליחויות, מסעדות, למשפחות, ואף גם לניהול אישי משתתפי הקבוצה יוכל להיעזר אחד בשני כדי לבצע את המשימות שלהם ובנוסף יוכל לעקוב אחר כמה המשימות שעשוה כל משתתף בקבוצה.

לדוגמה במשפחה של ארבעה נפשות וכולם בעלי רישיון. האמא צריכה לאסוף דואר ויוצרת את המשימה הזו כמשימה חדשה לביצוע בקבוצות המשפחה שלה, כל בני המשפחה יקבלו התראות על המשימה אם מיקומם קרוב למקומות המשימה שנקבע בעת יצירה משימה חדשה, הם יכולים לראות את המשימה, לבצע אותה, ולעדכן בכזעה כדי לעקוב אחר המשימות ואחר רמת הפעולות של המשתתפים בקבוצה.

תחילה רציתי לבנות את האפליקציה באנדרואיד סטודיו ולבסוף החלטתי לבחור בשימוש בFlutter משומן שרציתי לאתגר את עצמי, ללמוד שפה חדשה (Dart) ולהכיר סביבה חדשה שבköד אחד בונה אפליקציה גם لأنדרואיד וגם לאייפון.

תאריך: 15/03/2023

לכבוד
יחידת הפרויקטים מה"ט

הצעה לפרויקט גמר

1. פרטי הסטודנטים

שם הסטודנט	ת.ז. ספרות	כתובת	טלפון נייד	תאריך סיום הלימודים
אלון מלב	208454496	הרצל 12 יהוד מונוסון	0546344996	2023

שם המכללה בית הספר הארצי להנדסאים סמל המכללה : 72201

מסלול ההכשרה : הנדסאים
מגמת לימוד : הנדסת תוכנה 1/47
מקום ביצוע הפרויקט : בית הספר הארצי להנדסאים בקרית טכניון

2. פרטי המנהה האישי

שם המנהה	כתובת	טלפון נייד	תואר	מקום העבודה/תפקיד
שלி קול	גבעת דאונס 13 חיפה	054-7910099	Msc	בה"ס להנדסאים טכניון

חתימת הגורם המקצועי מטעם מה"ט חתימת המנהה האישי חתימת הסטודנט

רחוב מנחים בגין 86 תל אביב ת.ד. 36049. מיקוד 67138 טלפון :
7347644-03 פקס : 7347521-03

1. שם הפרויקט :

"**אפליקציה לניהול משימות בקבוצה**"

2. רקע העבודה ומטרת העבודה:

יצירת אפליקציה שתנהל משימות קבוצתיות הוא רעיון שנובע מהתkalות עם הרבה משימות שעליינו לבצע בתוך קבוצה אם זה בתחום עבודה או בתחום משפחה או אפילו בתחום בני אדם עצמאיים, לכן הרעיון הוא ליצור אפליקציה שתוכל לנוהל לקבוצות אנשים את המשימות הפרטיות והקבוצתיות שלהם תוך כדי התמסחות עם המיקום במקישר בכך לשלוח התראות על משימות לביצוע על פי מיקום המשימה.

אם זה לעובדות כמו שליחויות, מסעדות, למשפחות, ואף גם לניהול אישי משתפי הקבוצה יוכלו להיעזר אחד בשני כדי לבצע את המשימות שלהם ובנוסף יוכלו לעקוב אחר כמה המשימות שעשוה כל משתמש בקבוצה.

לדוגמה במשפחה של ארבעה נפשות וכולם בעלי רישיון. האמא צריכה לאסוף דואר ויוצרת את המשימה הזו כמשימה חדשה לביצוע בקבוצת המשפחה שלה, כל בני המשפחה יקבלו התראות על המשימה אם מיקומם קרוב למיקום המשימה שנקבע בעת יצירת משימה חדשה, הם יוכלו לראות את המשימה, לבצע אותה, ולעדכן בכוצעה כדי לעקוב אחר המשימות ואחר רמת הפעולות של המשתתפים בקבוצה.

3. סקירה מצב קיימש בשוק, אילו בעיות קיימות:

מערכת לניהול משימות פתוח ידני ומוחשב על ידי יומנים/יומניהם אלקטרוניים/רשומות מטלות/וכו או כאפליקציות המוצגות בשוק אך כיום האפליקציות הכח חזקות בתחום איןן כל כך קלות לשימוש ואיןן מתראות על פי מיקום מערכות בסגנון :

- | | |
|---|---|
| 1. /https://www.any.do | 2. /https://trello.com |
| 3. /https://monday.com | |

4. מה הפרויקט אמר לחדר או לשפר :

הפרויקט אמר להציג את הרעיון נוחה וקל לשימוש ובנוסף חוותית למשתמש באמצעות מתן ניקוד למשתמשים ותוך כדי שילוב של יעילות באמצעות שיתוף מיקום והתראה בזמן קרובה למיקום המועד בפיתוח עתידי ת透ספּ אפשרות פרסום בلوح ציבורי על מנת שיתוף משימות עם אנשים בסביבה הקרובה

5. דרישות מערכת ופונקציונליות:

1. דרישות מערכת –

המערכת תהיה אפליקציה תדרש התקנה וחיבור לאינטרנט כדי שימוש, שיתוף מיקום (אופציונאלי) המערכת תכלול בסיס נתונים.

דרישות פונקציונאליות .2

סוג דרישת פונקציונאלית	סוג דרישת פונקציונלית	מקור	תיאור דרישת פונקציונאלית	מספר דרישת
	ממשק		ממשק התהברות ורישום	1
	תפעולית		רישום משתמש	2
	תפעולית		הרשאות מיקום: ושיתוף אנשי קשר האפליקציה בבקשת הרשאות מהמכשיר עליה מותקנת	3
	תפעולית		פעולות שיזור משתמש	4
	תפעולית		התהברות משתמש	5
	ממשק		עמוד קבוצות (עמוד ראשי) בו יוצגו קבוצות של המשתמש ופועלות שתיחת קבוצה	6
	תפעולית		הකפתת התראות על פי מיקום המשימות למשתמש	7
	ממשק		ממשק משתמש (פרטים אישיים, אישור מיקום, וניקוד משוקל של כל הקבוצות)	8
	ממשק		ממשק קבוצה מוצגת רשימת המשימות שניננתו לעדכון ע"י כל משתתפי הקבוצה, אפשרות ביצוע פעולות של ייצרה/עדכון/מחיקת משימה	9
	תפעולית		מתן ניקוד: תינתן נקודה קבוצתית למשתתף שביצע משימה בהצלחה, ויצבר ניקוד כללי לכל משתמש המשתמש יש עלה רמת	10
ביצוע			ביצוע פעולות מהיר ורספונסיבי	11
	תפעולית		הגדרת משימה טקסט חופשי והגדרת מיקום דרוש למשימה על פי מיקומים מוגדרים מראש	12
	תפעולית		תפריט פעולה בצד לעובר בין ממשקים ועמודים	14
ביצוע			התוכנית תרוץ על גירסה lolipop	15
	תפעולית		אפשרות ביצוע פעולות בקבוצה עריכה/הוספה/הסרת משתמש	16

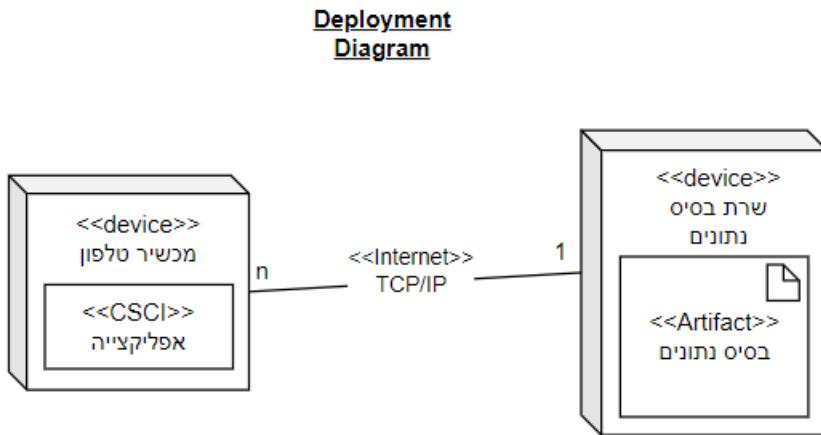
6. בעיות צפויות במהלך הפיתוח ופתרונות (תפעוליות, טכנולוגיות, עומס ועוד)

1. תיאור הבעיה-

זו הפעם הראשונה שאני עובשה פרוייקט בסדר גודל כזה ואני צפוי להתקבל בהרבה בעיות מטעם חוסר ניסיון בתחום אם אלו בעיות תכנון או תכנות כגון קושי בניהול הנתונים, בניית התרשימים ומעקב אחריהם, שימוש בתוסף התממשקות המיקום ועיצוב האפליקציה.

2. פתרונות אפשריים-

פתרונות בעיות הם עבודה, למידה, ניסוי וטיעיה בצד להתמודד עם קשיי התכנון ובנית התרשימים, בעיות העיצוב אטייע עם אנשים שambilנים בעיצוב ואלמד דרך האינטרנט את אופן התכונות ואיך להשתמש בתוסף המיקום.



Deployment Table

פרטי חומרה (NODE):

שם	מהות	עקיבות לדרישות
מכשיר טלפון	בכדי להשתמש באפליקציה נדרש מכשיר טלפון	
שרת בסיס נתונים	מאפשר לגשת לבסיס הנתונים ולנהל מידע	

פרטי תוכנה:

שם	מהות	עקיבות לדרישות
אפליקציה	נותנת את הפלטפורמה לביצוע המטרת של המערכת	
בסיס נתונים	לאחסן ולנהל את המידע	

ממשקים פיזיים:

מרקם	תכנים	תוקף פרוטוקול
חיבור לרשת		TCP/IP

.2. טכנולוגיות בשימוש –

סביבת פיתוח האפליקציה - אנדרואיד סטודיו
בסיס הנתונים – Fire Base

.3. שפות הפתח –

אפשר בגיאו זה שפה המועדף עליי משתמשים בה לפיתוח באנדרואיד משום שהוא השפה העיקרית של מדרנו

.4. תיאור הארכיטקטורה הנבחרת –

בחרתי במודל MVC משום שהוא מאפשר נפרדות של תפקידים ומקל על ניהול רכיבי היישום
Model – בסיס הנתונים.
View – משק המשמש –
Controller – הלוגיקה העסוקית של המערכת

.5. חלוקה לתוכניות ומודולים –

MODEL

User, Group, Mission

VIEW

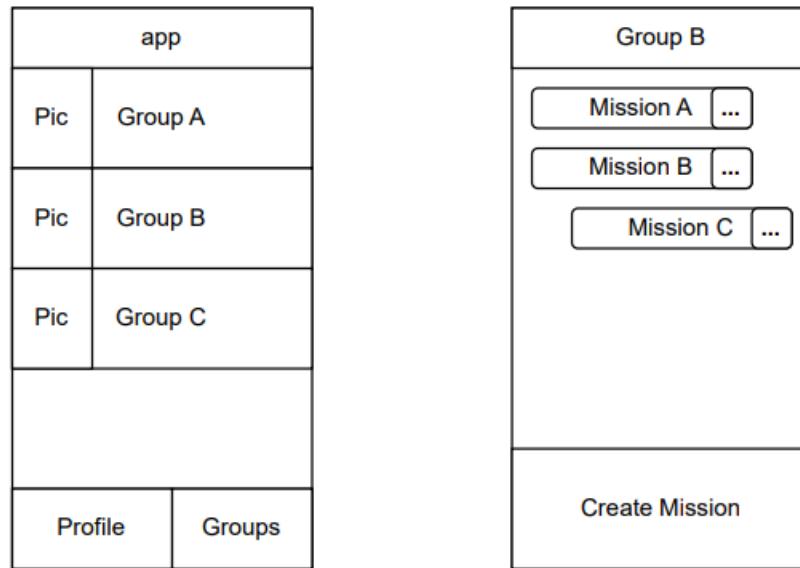
UserOptionsActivity, MainPageActivity, GroupRoonActivity, GroupDetailsActivity,
.LoginActivity, RegisterActivity, MissionDetailsActivity

Controller

.DatabaseActivities

.6. סביבת השירות –

. סביבה וירטואלית. Fire Base



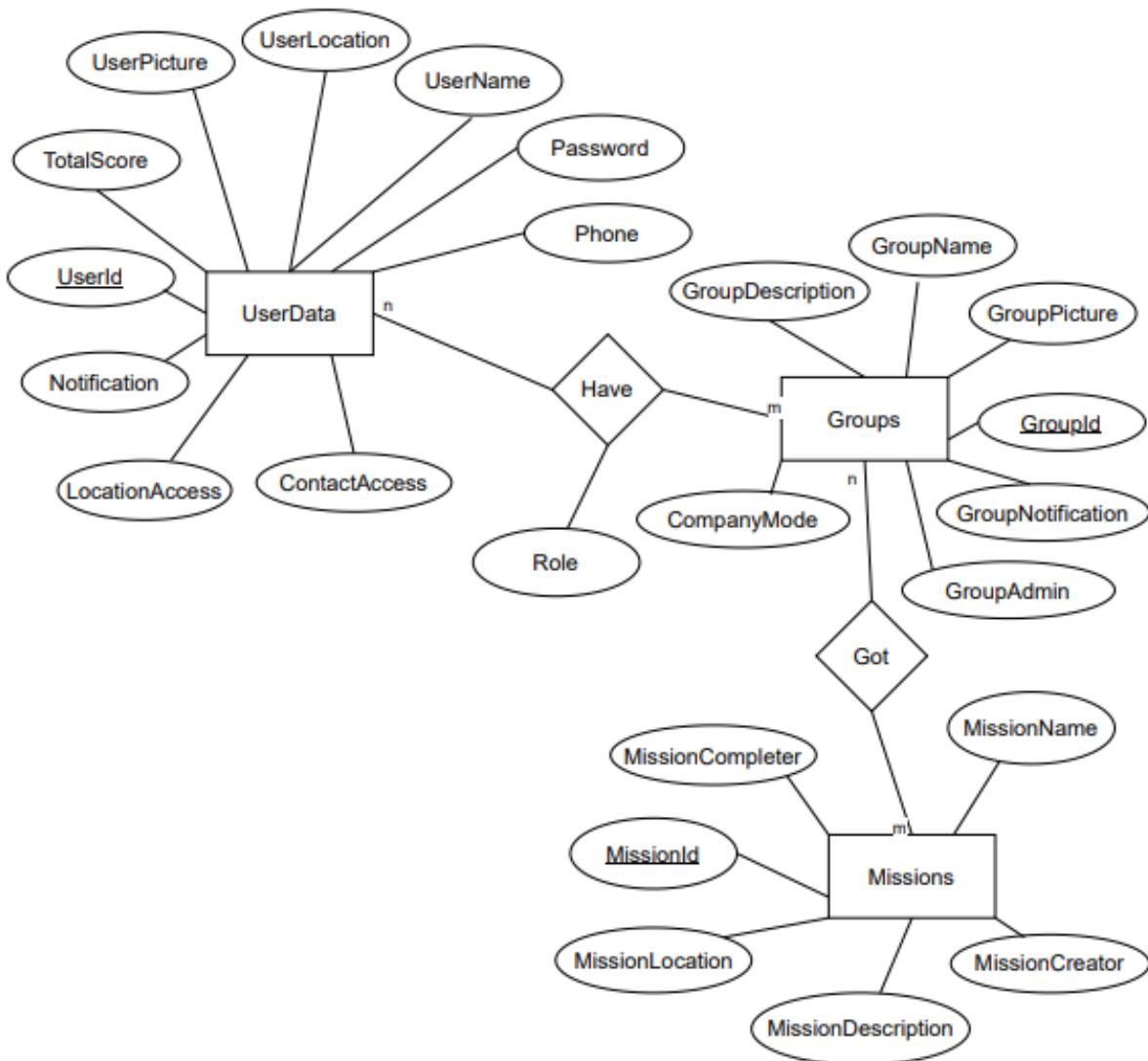
-API .8

google maps

שימוש בחבילות תוכנה .9

לא ידוע ברגע.

ERD



UserData(UserName, Password, UserLocation, ,UserPicture, TotalScore, UserId, phone, Notification, LocationAccess, ContactAccess)

Groups(GroupName, GroupId, GroupDescription, GroupPicture, GroupNotificaion, CompanyMode, GroupAdmin)

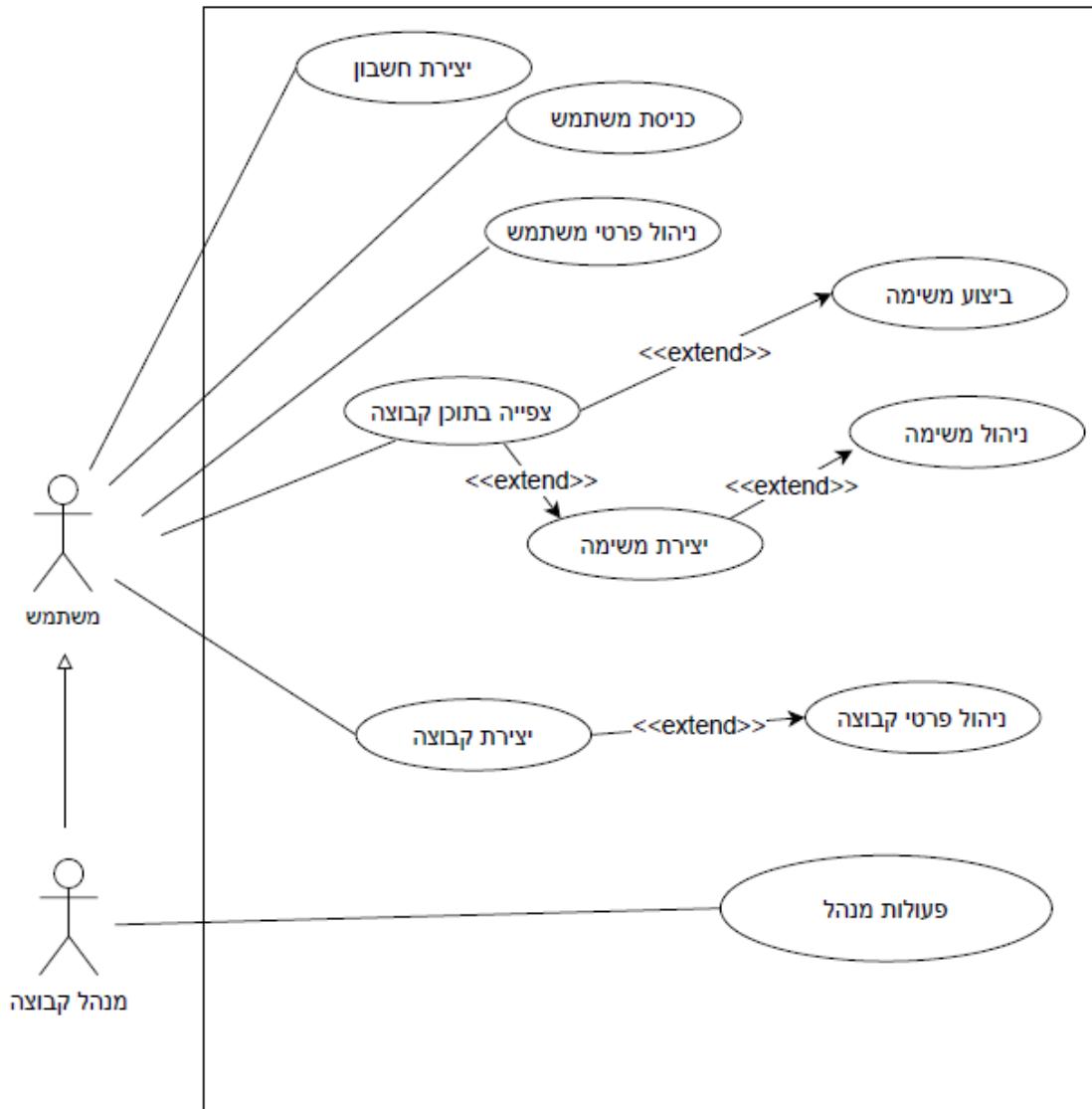
Missions(MissionName, MissionId, MissionDescription, MissionLocation, MissionCreator, MissionComleter)

Cloud(CloudName, CloudDescription, CloudId)

Have(UserId, GroupId, Role)

Got(GroupId, MissionId)

**UseCase
Diagram**



מסמכים נסיבות שימוש

יצירת החשבון ucCreate_Account	שם: <u>מזהה:</u>
יצירת חשבון למשתמש באפליקציה : מילוי פרטים רלוונטיים לייצור משתמש שם משתמש, סיסמא, טלפון	טיואר:
משתמש – יוצר חשבון באפליקציה על מנת להתmesh בה	שחקנים:
	סתטוס:
בכל פעם שמצטרף משתמש חדש לאפליקציה	תדרונות:
1,2,3	עקיבות לדרישות:
נדרש שהייה משתמש חדש באפליקציה	תנאים מקדים:
1. משתמש מכניס פרטי הרשמה 2. המערכת בודקת התאמה 3. המערכת מקימה משתמש DB	תרחיש הצלחה:
משתמש מכניס פרטי הרשמה אחרים ומנסה שוב	חלופות:

כניסת משתמש	<u>שם:</u>
ucLogin_Account	<u>מזהה:</u>
התחברות למשתמש באפליקציה : באמצעות מילוי שם משתמש וסיסמה	<u>טיאור:</u>
משתמש – מתחבר לחשבון באפליקציה על מנת להתmesh בה	<u>שחקנים:</u>
	<u>סתatos:</u>
בכל פעם שמשתמש מתחבר	<u>תדריות:</u>
4,5	<u>עקיבות לדרישות:</u>
נדרש שייהי משתמש רשום	<u>תנאים מקדים:</u>
1. משתמש מזין פרטי התחברות 2. המערכת בודקת תקינות פרטיים אל מול DB 3. המשתמש מחובר לחשבון שלו	<u>תרחיש הצלהה:</u>
משתמש מכניס פרטיים תקינים ומנסה שוב	<u>חלופה:</u>

שם:	ניהול פרטי משתמש
זיהוי:	ucManage_Account
תיאור:	ניהול המשתמש שינוי פרטיים כגון: שם משתמש, סיסמה, תמונה, עדכון הרשאות – מיקום/אנשי קשר, עדכון התראות
תקנים:	משתמש – יכול לעדכן פרטי משתמש
ստארטס:	
תדרות:	בכל פעם שמשתמש ירצה לשנות פרטיים או האגדרות
עקבות לדרישות:	8
תנאים מקדים:	נדרש שייהי מחובר לחשבון משתמש
תרחיש הצלחה:	1. משתמש משנה/معدכן את אחד הפרטים בפרטי המשתמש 2. מערכת בודקת תקינות הפרטים 3. מערכת מעדכנת את פרטי המשתמש DB
חלופה:	משתמש מכניס פרטיים תקינים ומנסה שוב

שם:	יצירת קבוצה
מזהה:	ucCreate_Group
תיאור:	יצירת קבוצה חדשה : יצירת קבוצה עם שם קבוצה ותיאור
שימושים:	משתמש – פתיחה קבוצה באפליקציה על מנת ליצור קבוצה שיתופית למשימות
תדריות:	בכל פעם ששימוש מתחבר מהליבט ליצור קבוצה
עקיפות לדרישות:	6,16,9
תנאים מקדים:	נדרש שהייה מחובר לחשבון משתמש
תרחיש הצלחה:	1.משתמש יוצר קבוצה חדשה 2.מכניס פרטי קבוצה חדשה 3.מערכת מקימה קבוצה לשימוש ולמשתמשים שהווסף
חלופה:	מנסה שוב

שם:	ניהול פרטי קבוצה
מזהה:	ucManage_Group
תיאור:	ניהול קבוצה: ביצוע פעולות בקבוצה כגון שינוי שם קבוצה שינוי תיאור קבוצה שינוי תמונה
שימושים:	משתמש – ביצוע פעולות בקבוצה
սթאטוס:	
תדריות:	בכל פעם שימוש מעוניין לעדכן את הקבוצה
עקיבות לדרישות:	6,9
תנאים מקדים:	נדרש שיהיה משתמש בקבוצה
תרחיש הצלחה:	1. משמש מזמן פרטי קבוצה שמעוניין לשנות 2. המערכת מעדכון פרטי קבוצה
חלופה:	מנסה שוב

שם: מזהה:	פועלות ניהול ucManagePeopleInGroup
תיאור:	ביצוע פועלות ניהול כגון: הוספה/הסרת משתמש בקבוצה לפי קוד משתמש על מנת ליצור קבוצה שיתופית למשימות ו/או האבלת יצרת משימות של משתפים רגילים בקבוצה ב כדי שרק מנהל הקבוצה יוכל לעדכן משימות או מהיקת הקבוצה
שחקנים:	משתמש (מנהל קבוצה) – הוספה/הסרת משתמש באפליקציה ו/או להגביל פעילות של משתמשים בקבוצה או מהיקת קבוצה את הקבוצה
סתאים:	
תדריות:	בכל פעם ששימוש מעוניין להוסיף או להסיר משתמש נוסף לקבוצה או להגביל יצרת משימות
עקיבות לדרישות:	6,16,9 נדרש שייהי מנהל הקבוצה
תנאים מקדים:	
תרחיש הצלחה:	1. מנהל קבוצה מבצע פעולה ניהול 2. מערכת מעכנת פרטי קבוצה
חלופה:	מנסה שוב

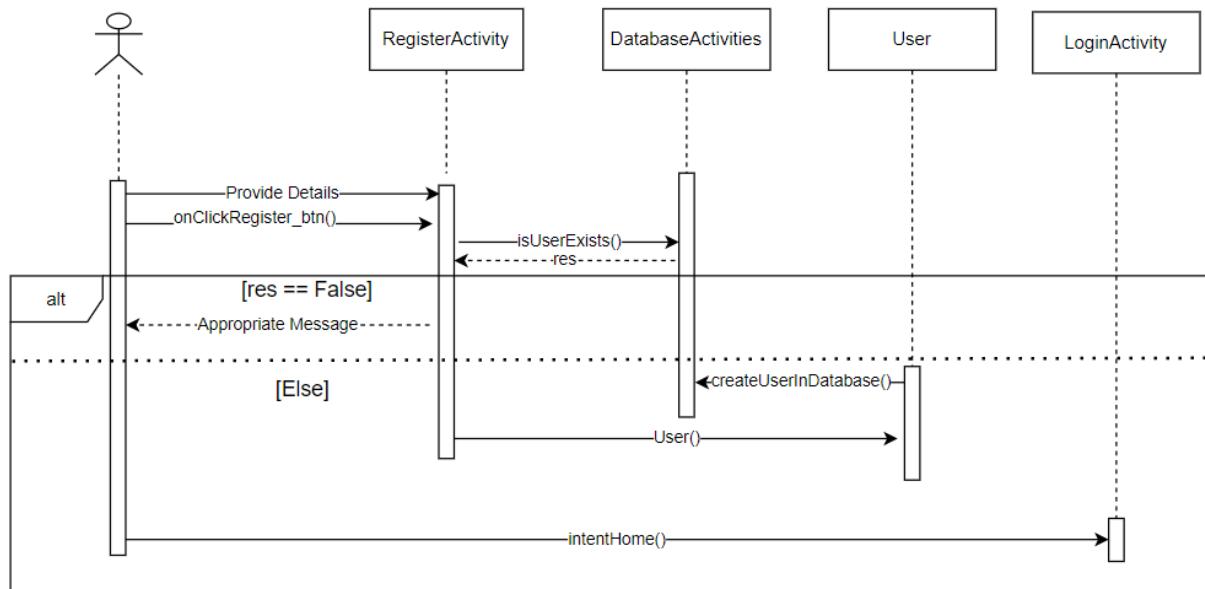
יצירת משימה	שם:
ucCreate_Mission	<u>מזהה:</u>
תיאור: יצירת משימה חדשה : מילוי טקסט חופשי + הגדרת מקום משימה משתמש – יצירה משימה חדשה בקובוצה	שחקנים:
	<u>סתומות:</u>
בכל פעם ששתמש מעוניין ליצור משימה חדשה בקובוצה	<u>תדרות:</u>
9,12,13	<u>עקיבות לדרישות:</u>
נדרש שהיוה משתמש בקובוצה	<u>תנאים מקדים:</u>
1.משתמש יוצר משימה בקובוצה 2.מזון פרטី משימה נדרש 3.מערכת מעכנת משימה בקובוצה	<u>תרחיש הצלחה:</u>
מנסה שוב	<u>חלופה:</u>

שם:	ניהול משימה
מזהה:	ucManage_Mission
תיאור:	ביצוע פעולות במשימה כמו עדכון משימה עדכון מקום משימה מחיקת משימה
תקנים:	משתמש – עדכון משימה קיימת
ստاطוס:	
תדריות:	בכל פעם ששימוש מעוניין לעדכן משימה קיימת שיצר בקובוצה
עקיבות לדרישות:	9,12
תנאים מקדים:	נדרש שהייה יוצר המשימה
תרחיש הצלחה:	1. משתמש מזין פרטי משימה שרוצה לעדכן 2. מערכת מעದכנת משימה
חלופה:	מנסה שוב

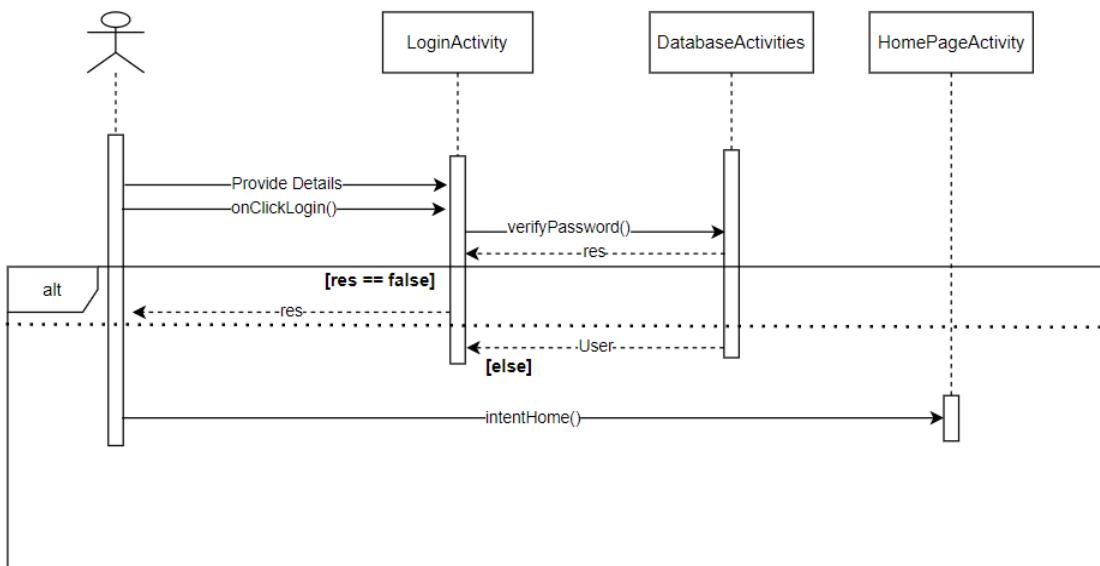
<u>ביצוע משימה</u>	<u>שם:</u>
ucComplete_Mission	<u>מזהה:</u>
השלמת המשימה ותן ניקוד למשתמש המשלים משתמש – השלמת משימה	<u>תיאור:</u> <u>פתרונות:</u>
	<u>סתיפים:</u>
בכל פעם שמשתמש ישלים משימות בסימון יدني	<u>תדריות:</u>
9,10,12	<u>עקיבות לדרישות:</u>
נדרש שלא יהיה יוצר המשימה	<u>תנאים מקדים:</u>
1. מתחמש מסמן משימה כבוצעה 2. מערכת מעדכנת משימה כבוצעה 3. מערכת מעדכנת ניקוד משתמש בקובץה	<u>תרחיש הצלחה:</u>
מנסה שוב	<u>חלופה:</u>

<u>צפיה בתוכן קבוצה</u>	<u>שם:</u>
ucView_Group	<u>מזהה:</u>
<u>צפיה בתוכן הקבוצה בנסיבות הקיימים, וביצוע פעולות בקבוצה</u>	<u>תיאור:</u>
משתמש – צופה בנסיבות	<u>שחקנים:</u>
	<u>סטאטוס:</u>
בכל פעם שמשתמש יכנס לקבוצה	<u>תדרות:</u>
9,12,16	<u>עקיבות לדרישות:</u>
נדרש שייהי משתמש קבוצה	<u>תנאים מקדים:</u>
1. משתמש מקיש על קבוצה אליה מעוניין להכנס 2. מערכת מגישה תוכן הקבוצה לתצוגה ושימוש המשתמש	<u>תרחיש הנסיבות:</u>
מנסה שוב	<u>حلופה</u>

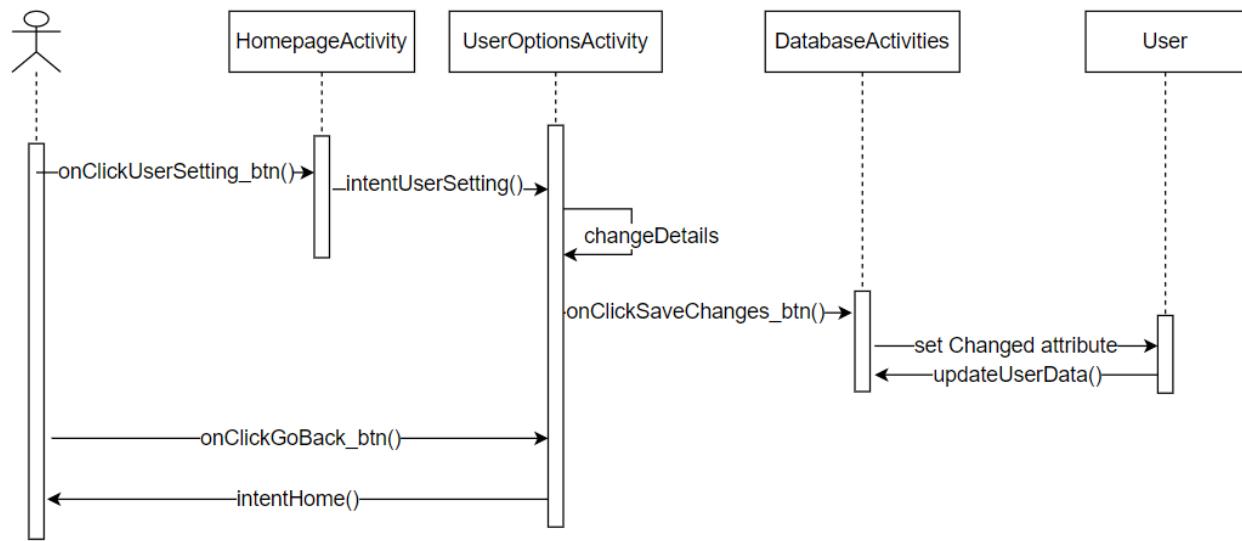
Sequence Diagram
צירוף חשבון



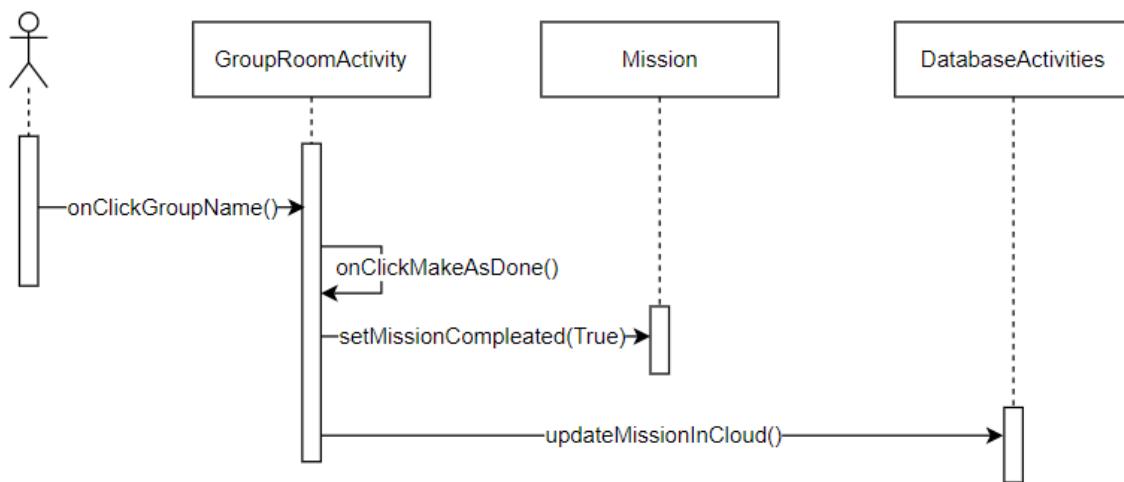
Sequence Diagram
כניסה שת�性



Sequence Diagram
ניהול פרטי משתמש

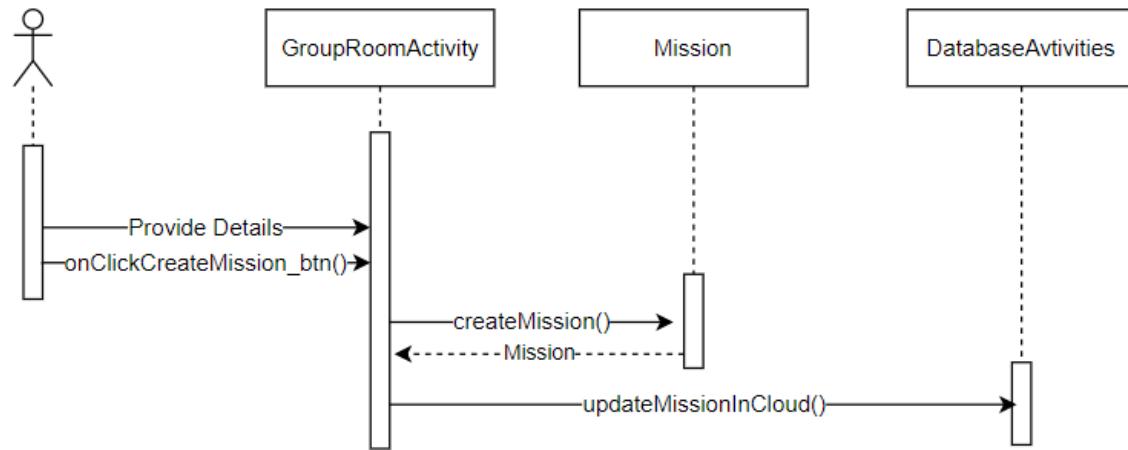


Sequence Diagram
ביצוע משימה



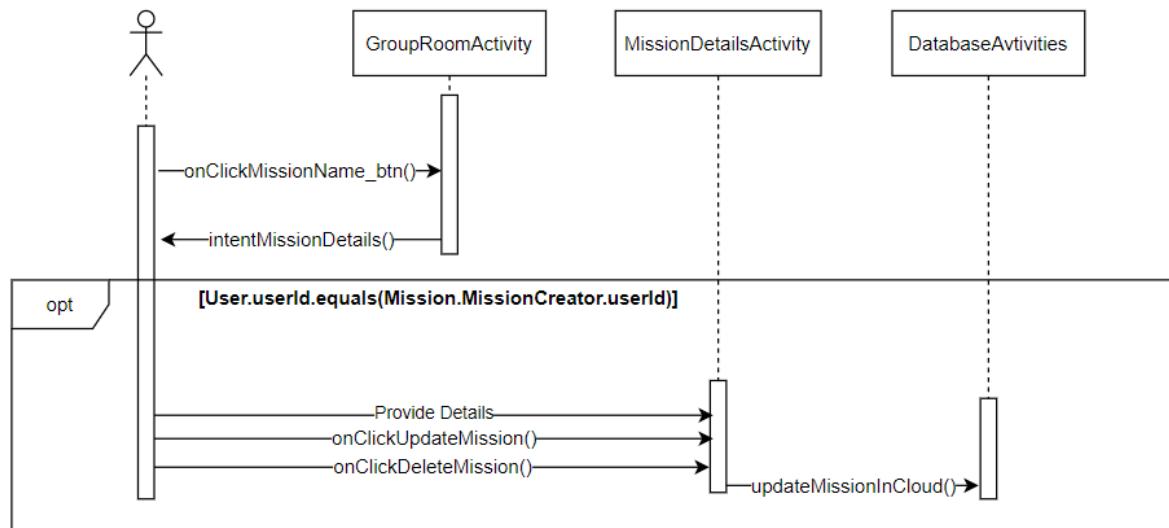
Sequence Diagram

יצירת משימה



Sequence Diagram

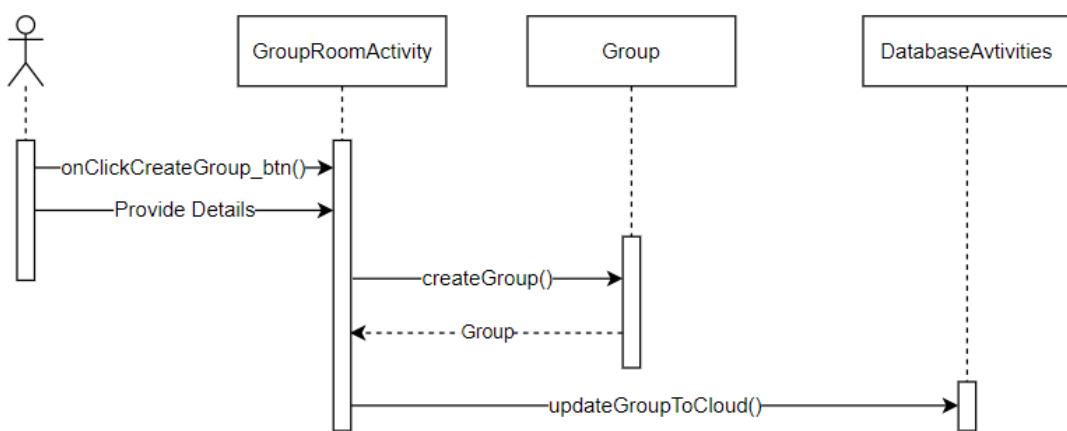
ניהול משימה



Sequence Diagram

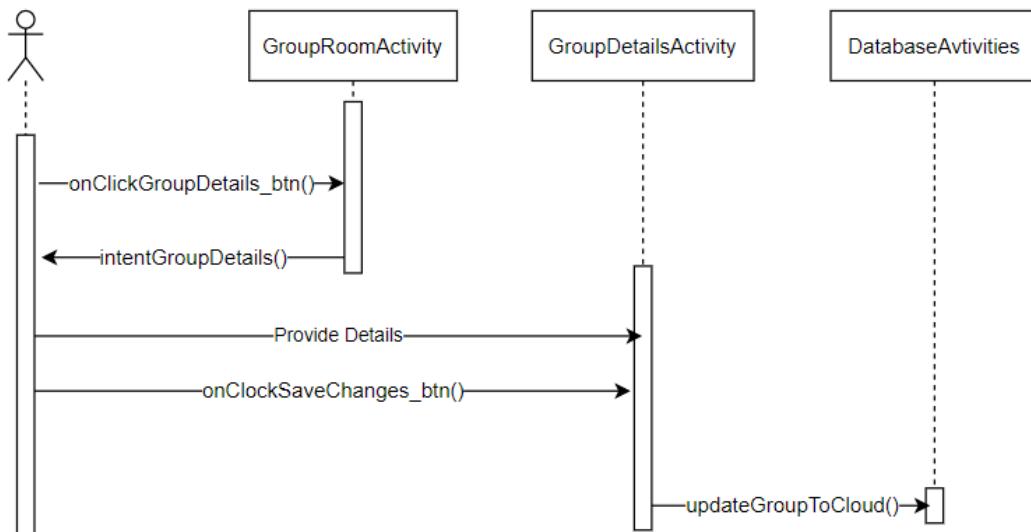
שירות

קבוצה



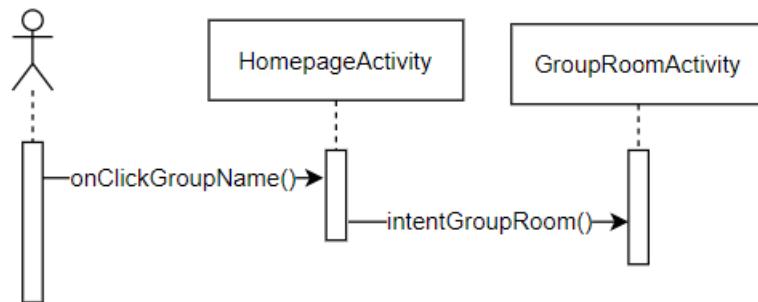
Sequence Diagram

ניהול פרטי קבוצה



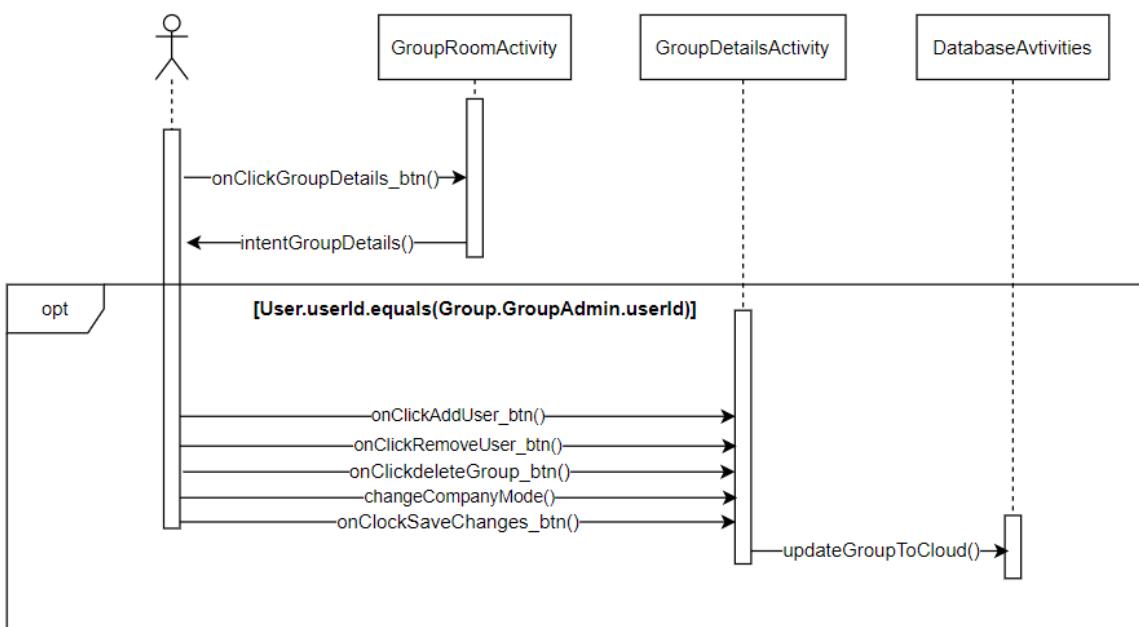
Sequence Diagram

צפיה בתוכן קבוצה

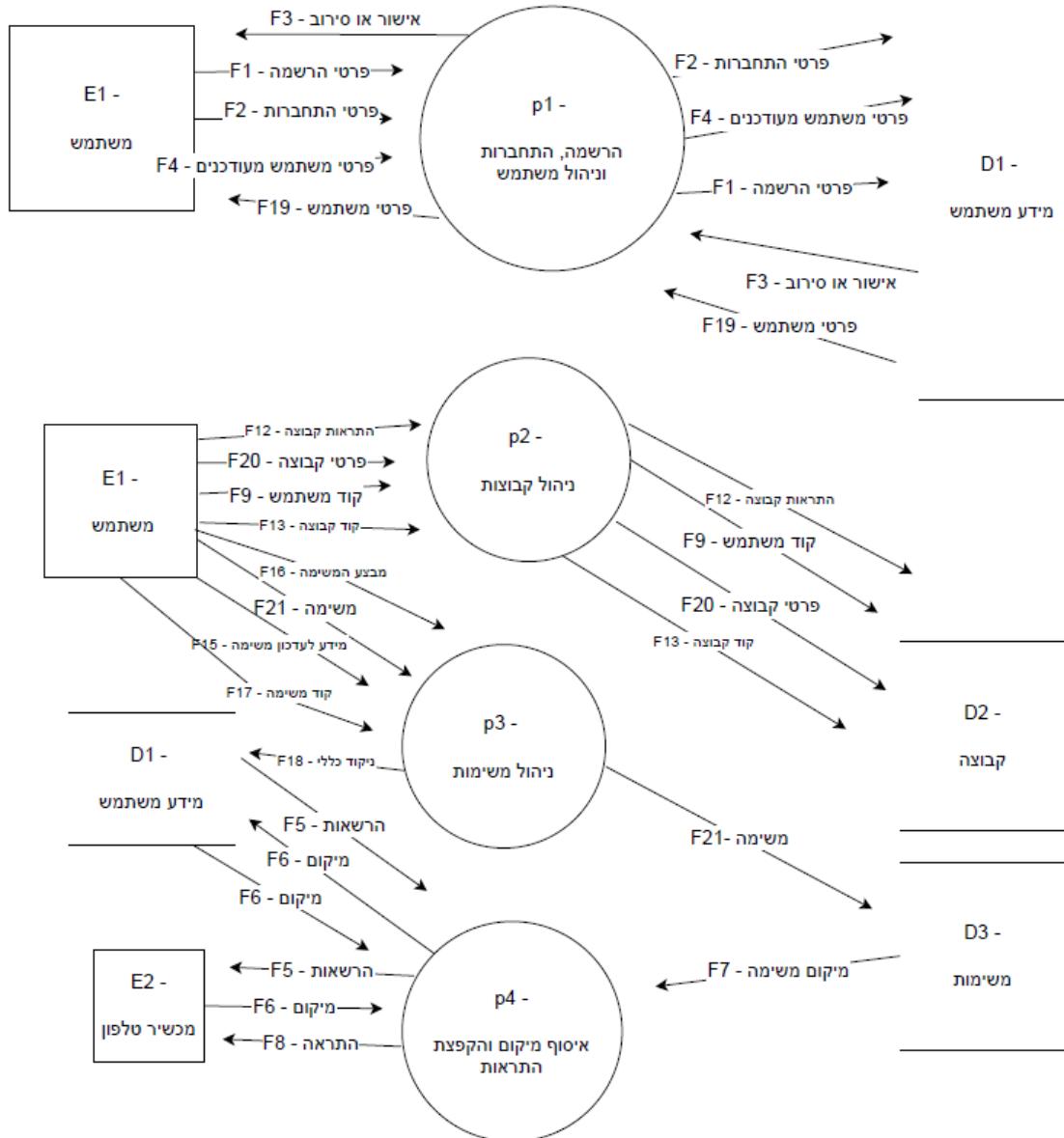


Sequence Diagram

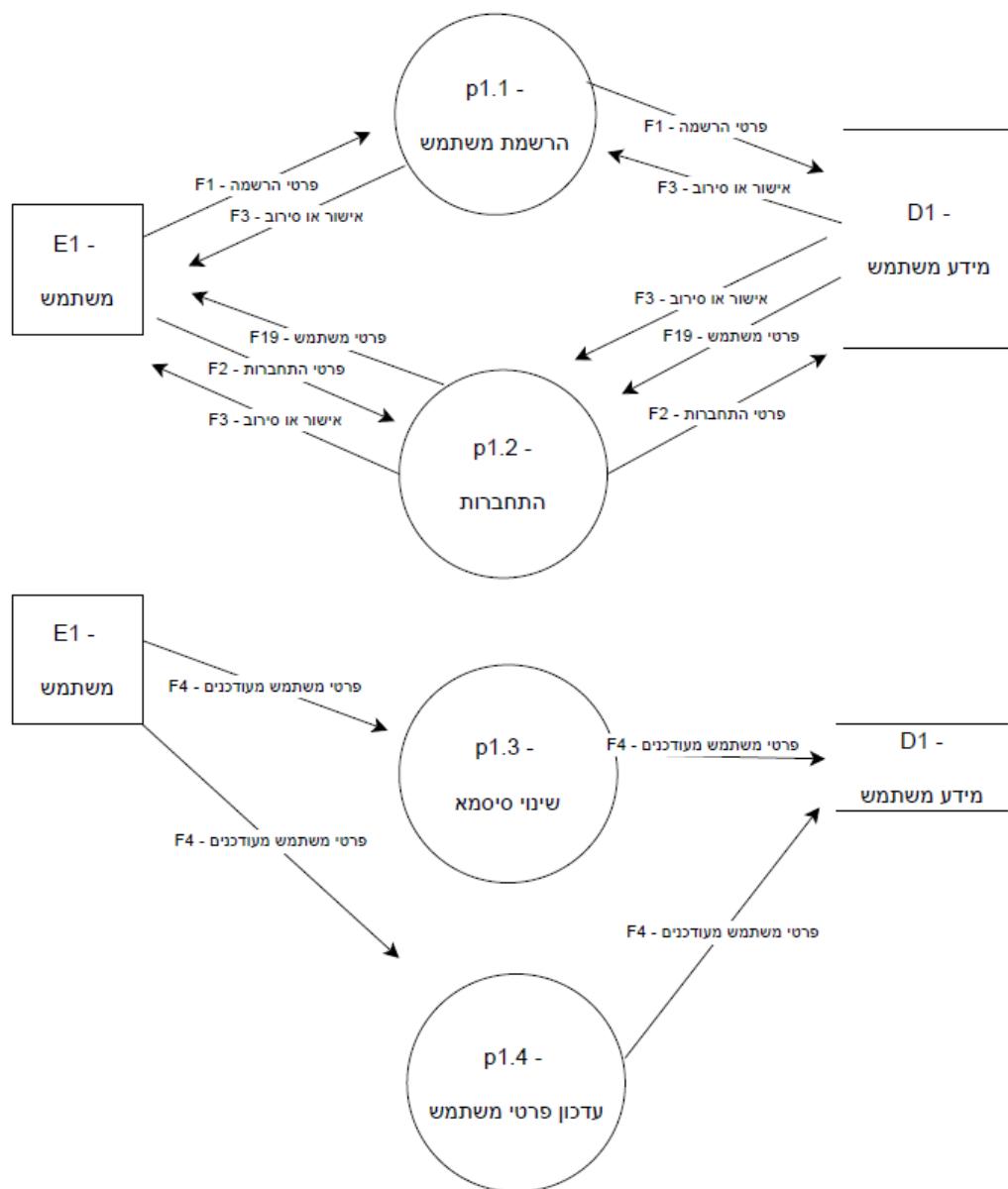
פעולות ניהול



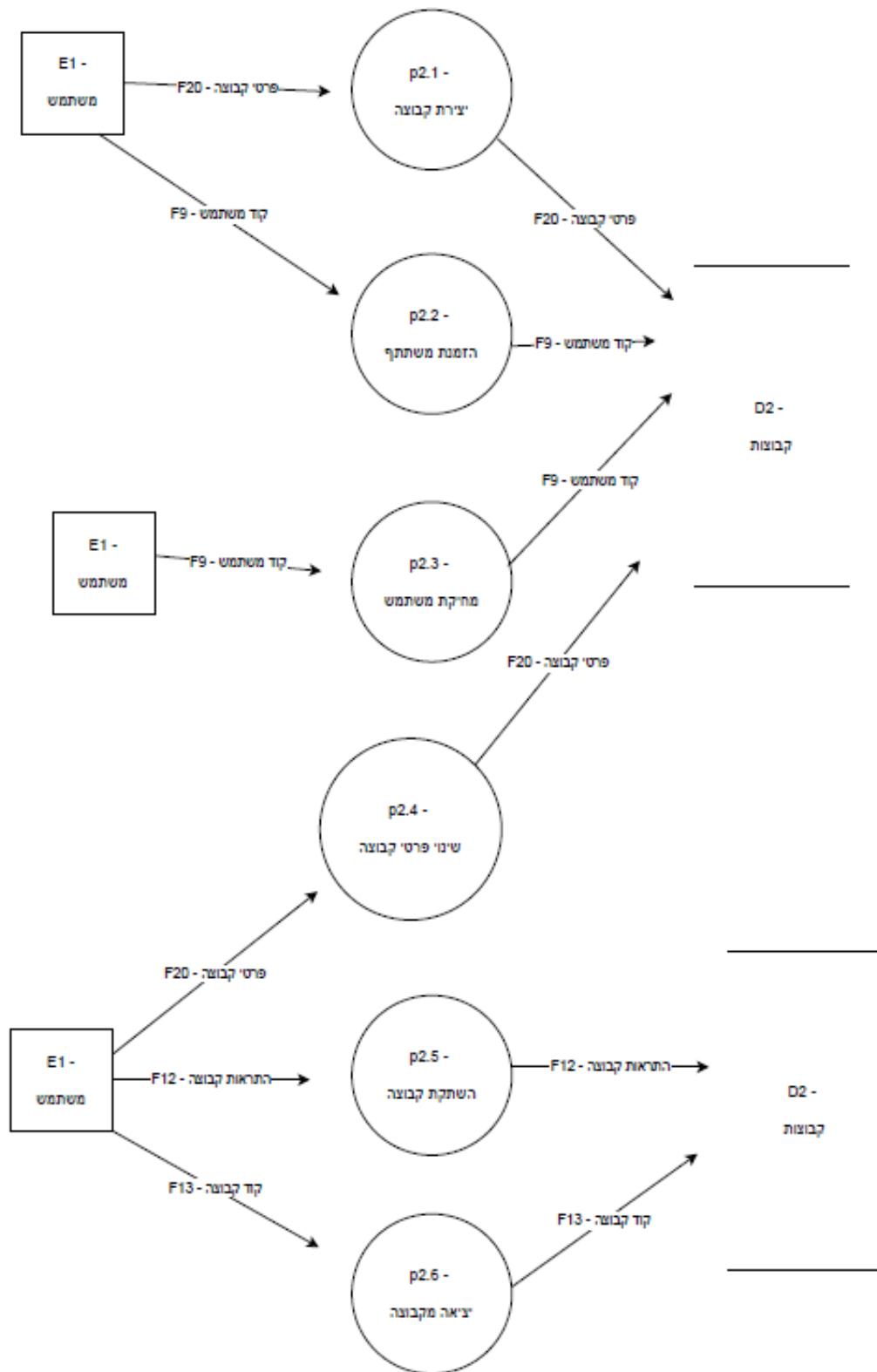
DFD 0



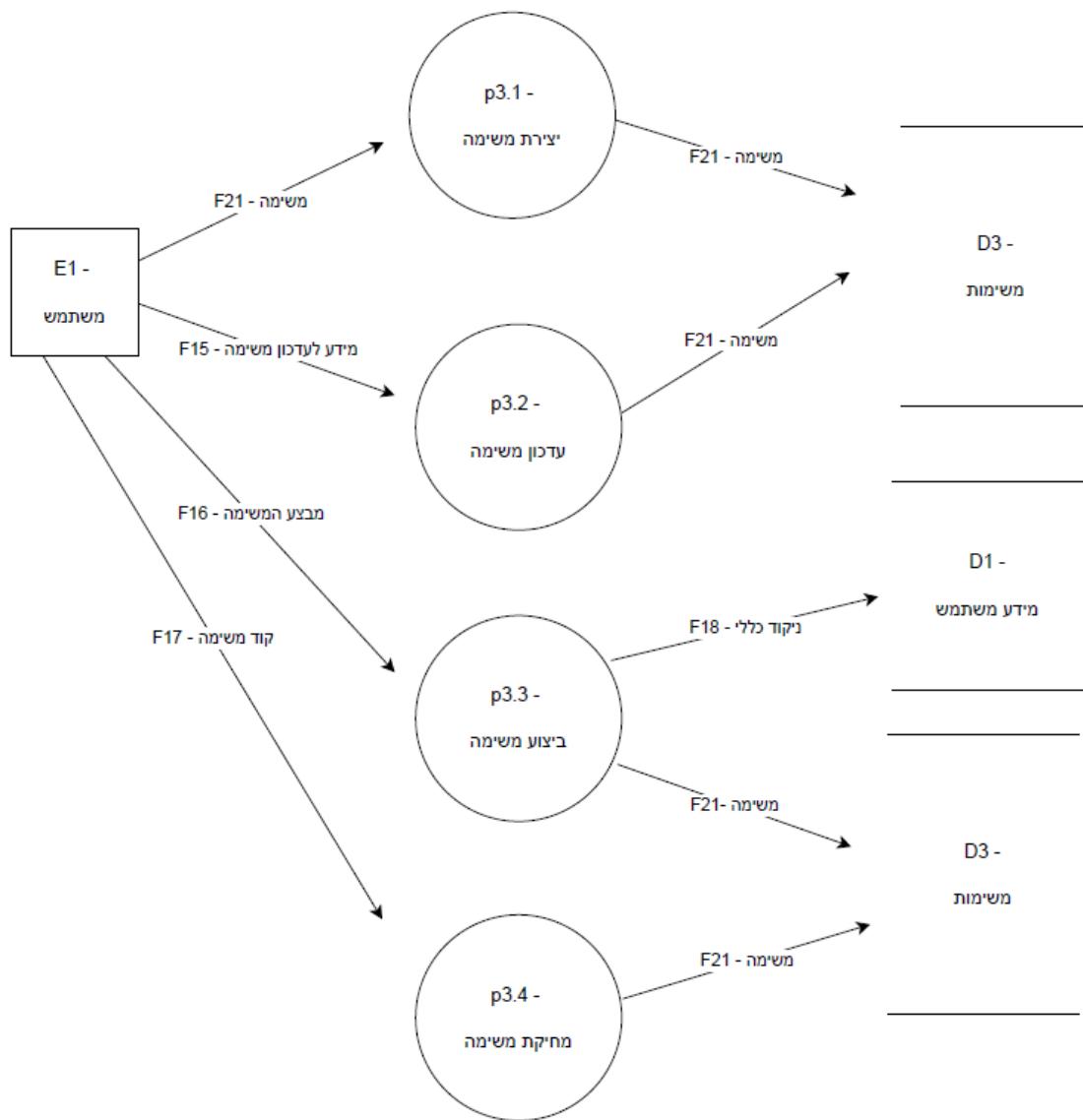
DDF.1.1



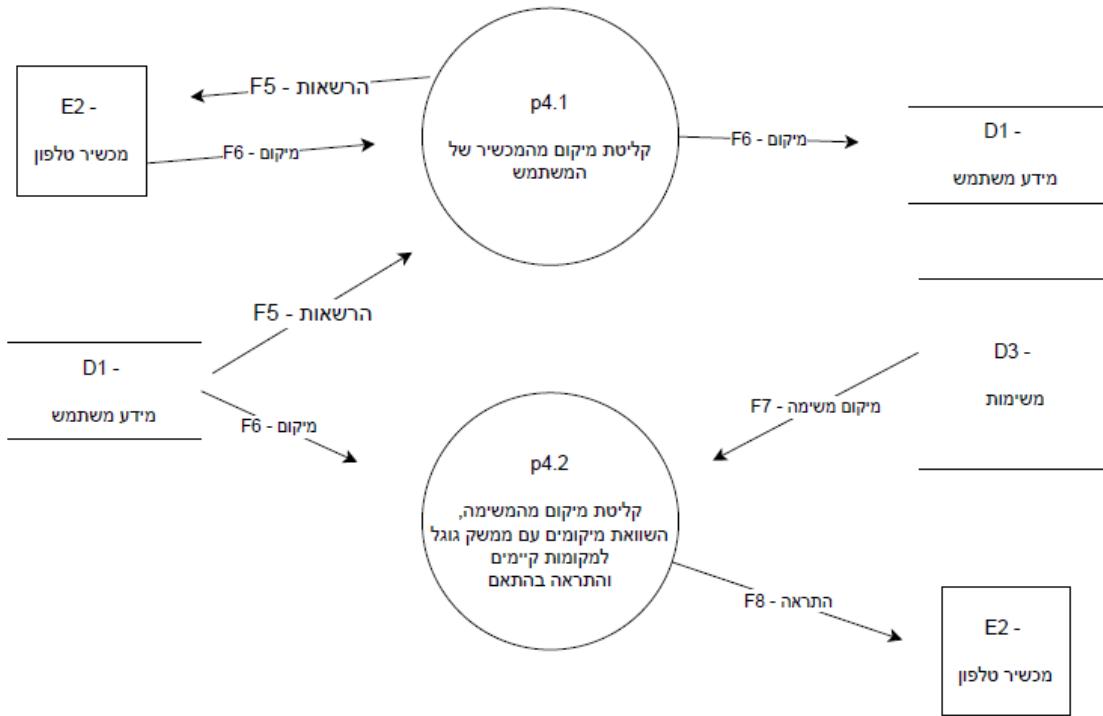
DFD 1.2



DFD 1.3



DFD 1.4



מילון נתונים
תיאור יישוות

תיאור	שם היישות	קוד
משתמש האפליקציה מייצר ומשתמש בקבוצות מייצר ומצבע שימוש	משתמש	E1
המכשיר שעליו יושבת האפליקציה מקבל ומתריע התראות שולח מקום	מכשיר טלפון	E2

תיאור המאגרים

שדות הטבלה	שם המאגר	קוד
, <u>UserId</u> ,UserName, Password, UserLocation, ,UserPicture, TotalScore Phone, Notification, LocationAccess, ContactAccess, Role	מידע משתמש	D1
GroupDescription, GroupPicture, GroupNotificaion, , <u>GroupId</u> ,GroupName Role, CompanyMode, GroupAdmin	קבוצות	D2
MissionDescription, MissionLocation, , <u>MissionId</u> ,MissionName MissionCreator, MissionComleter	משימות	D3

תיאור זרימות

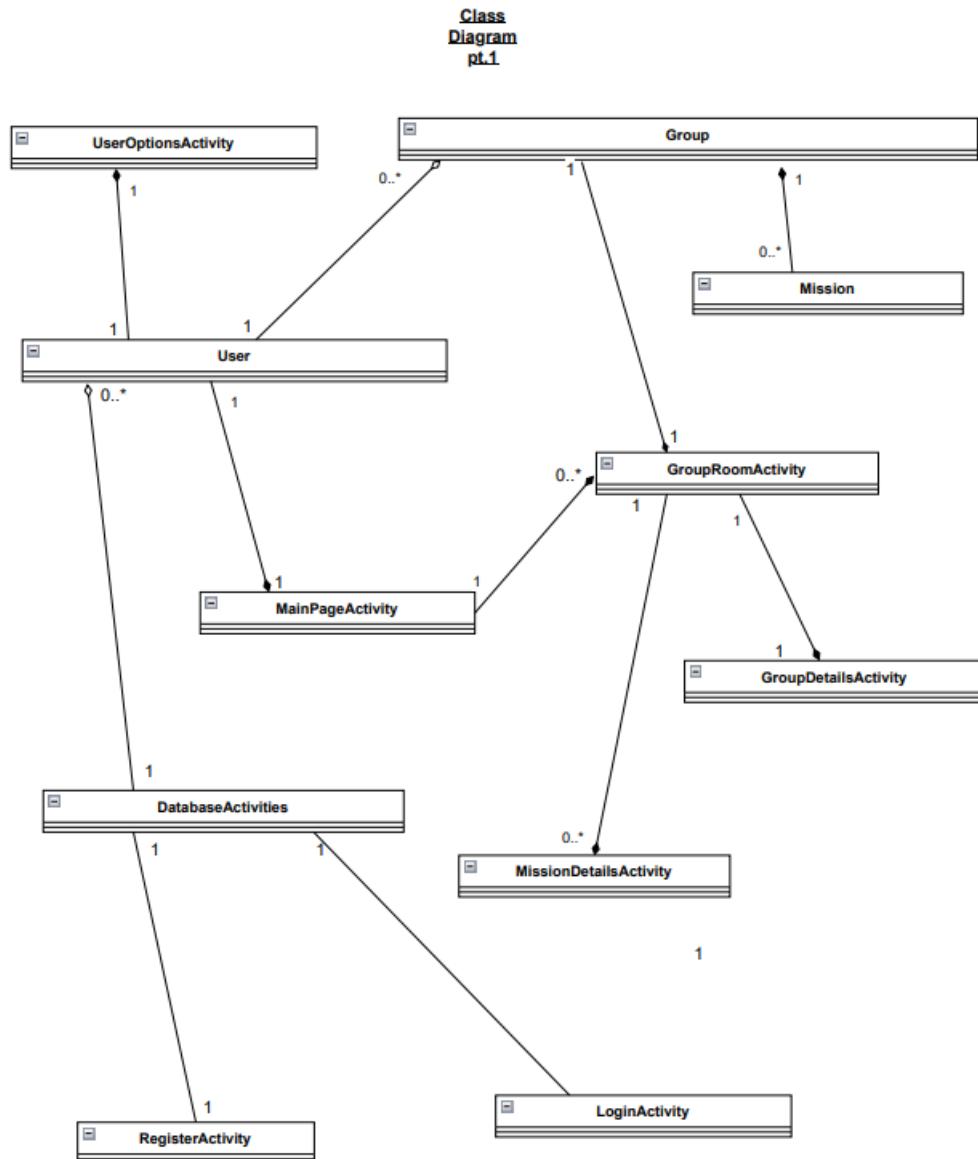
שדות הזרימה	יעד	מקור	קוד + שם זרימה
,UserName, Password, Phone	P1 : הרשמה, התחברות וניהול משתמש	משתמש	F1 : פרטי הרשמה
	מידע משתמש	הרשות, התחברות וניהול משתמש	
	P1.1 : הרשמה משתמש	משתמש	
	מידע משתמש	הרשות P1.1 משתמש	
UserName, Password	P1 : הרשמה, התחברות וניהול משתמש	משתמש	F2 : פרטי התחברות
	מידע משתמש	הרשות, התחברות וניהול משתמש	
	P1.2 : התחברות	משתמש	
	מידע משתמש	התחברות P1.2	

	P1: הרשמה, התחברות וניהול משתמש	מידע משתמש	F3: אישור או סירוב
	P1: הרשמה, התחברות וניהול משתמש	מידע משתמש	
	P1.1: הרשמה משתמש	מידע משתמש	
	P1.1: הרשמה משתמש	מידע משתמש	
	P1.2: התחברות משתמש	מידע משתמש	
	P1.2: התחברות משתמש	מידע משתמש	
UserName, Password, LocationAccess, ,UserPicture ContactAccess, Notification	P1: הרשמה, התחברות וניהול משתמש	משתמש	F4: פרטי משתמש מעודכנים
	P1: הרשמה, התחברות וניהול משתמש	מידע משתמש	
	P1.3: שינוי סיסמא	משתמש	
	P1.3: שינוי סיסמא	שינוי	
	P1.4: עדכון פרטי משתמש	משתמש	
	P1.4: עדכון פרטי משתמש	מידע משתמש	
LocationAccess, ContactAccess	P4: איסוף מיקום והקפצת התראות	מידע משתמש	5F: הרשות
	מכשיר טלפון	איסוף מיקום והקפצת התראות	
UserLocation	P4: איסוף מיקום והקפצת התראות	מכשיר טלפון	F6: מקום
	מידע משתמש	איסוף מיקום והקפצת התראות	
	P4.1: קליטת מיקום מהמכשיר של המשתמש	מכשיר טלפון	
	מידע משתמש	קליטת מיקום מהמכשיר של המשתמש	
	p4.2: קליטת מיקום מהמשימה, השוואת מקומות עם ממשק גугл למקומות קיימים והתראה בהתאם	מידע משתמש	

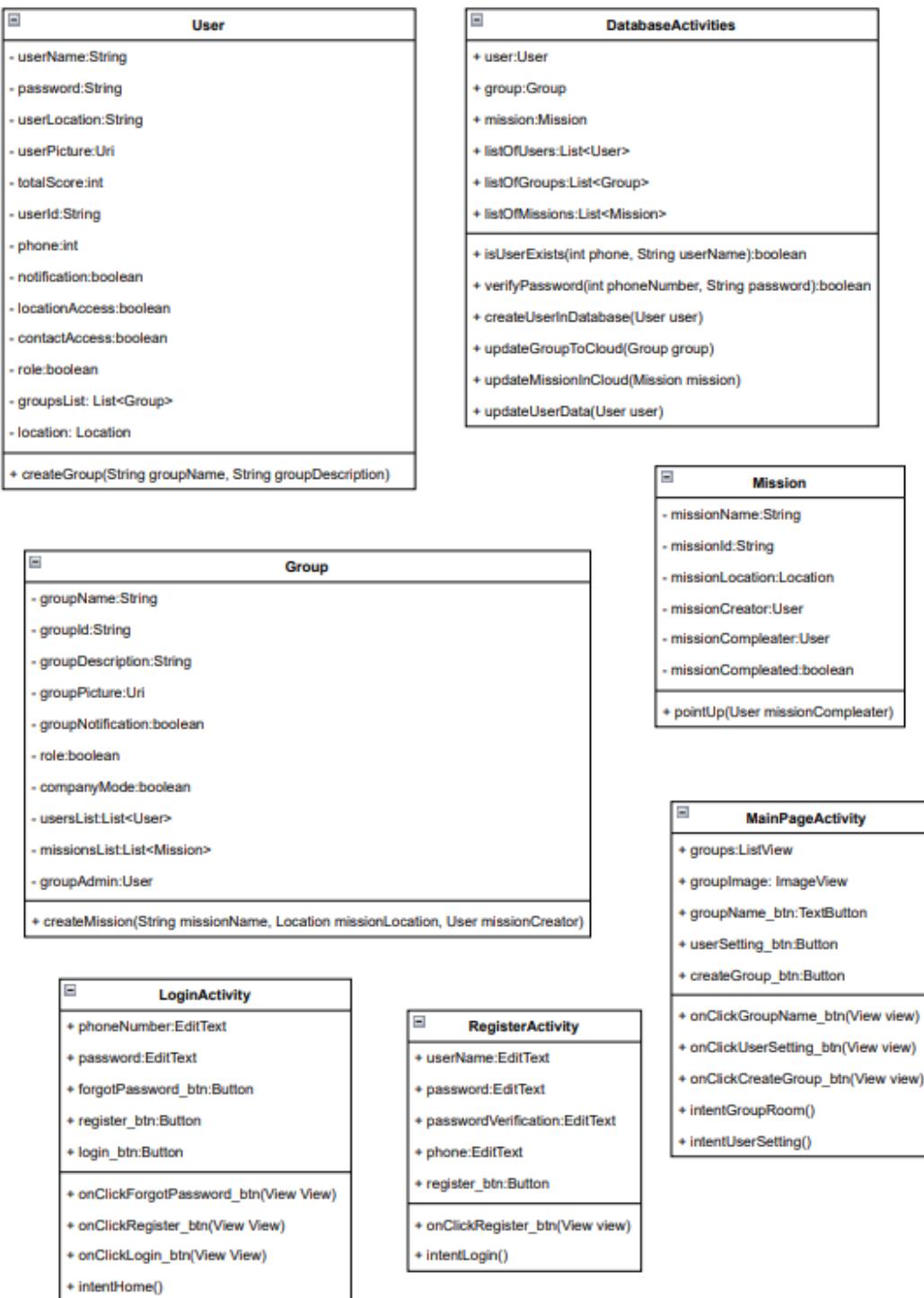
MissionLocation	P4 – איסוף מיקום והקפצת התראות	משימות	F7: מיקום משימה
	p4.2: קליטת מיקום מהמשימה, השוואת מיקומים עם ממשק גוגל למקומות קיימים והתראה בהתאם	משימות	
	מתריאה התראה בהתאם למיקום ופתרונות בקבוצות שהמשתמש משתמש בהן ולמיקום בו המשתמש נמצא	מכשיר טלפון	P4 – איסוף מיקום והקפצת התראות
UserId	P2.2: הזמנת משתמש	משתמש	F9: קוד משתמש
	קבוצות	P2.2: הזמנת משתמש	
	P2.3: מחיקת משתמש	משתמש	
	קבוצות	P2.3: מחיקת משתמש	
GroupName	P2.1: ייצירת קבוצה	משתמש	F10: שם קבוצה
	קבוצות	P2.1: ייצירת קבוצה	
	p2.4: שינוי שם/תיאור קבוצה	משתמש	
	קבוצות	p2.4: שינוי שם/תיאור קבוצה	
GroupDescription	P2.1: ייצירת קבוצה	משתמש	F11: תיאור קבוצה
	קבוצות	P2.1: ייצירת קבוצה	
	p2.4: שינוי שם/תיאור קבוצה	משתמש	
	קבוצות	p2.4: שינוי שם/תיאור קבוצה	
GroupNotification	p2.5: השתקת קבוצה	משתמש	F12: התראות קבוצה
	קבוצות	p2.5: השתקת קבוצה	
GroupId	P2.6: יציאה מהקבוצה	משתמש	F13: קוד קבוצה
	קבוצות	p2.6: יציאה מקבוצה	

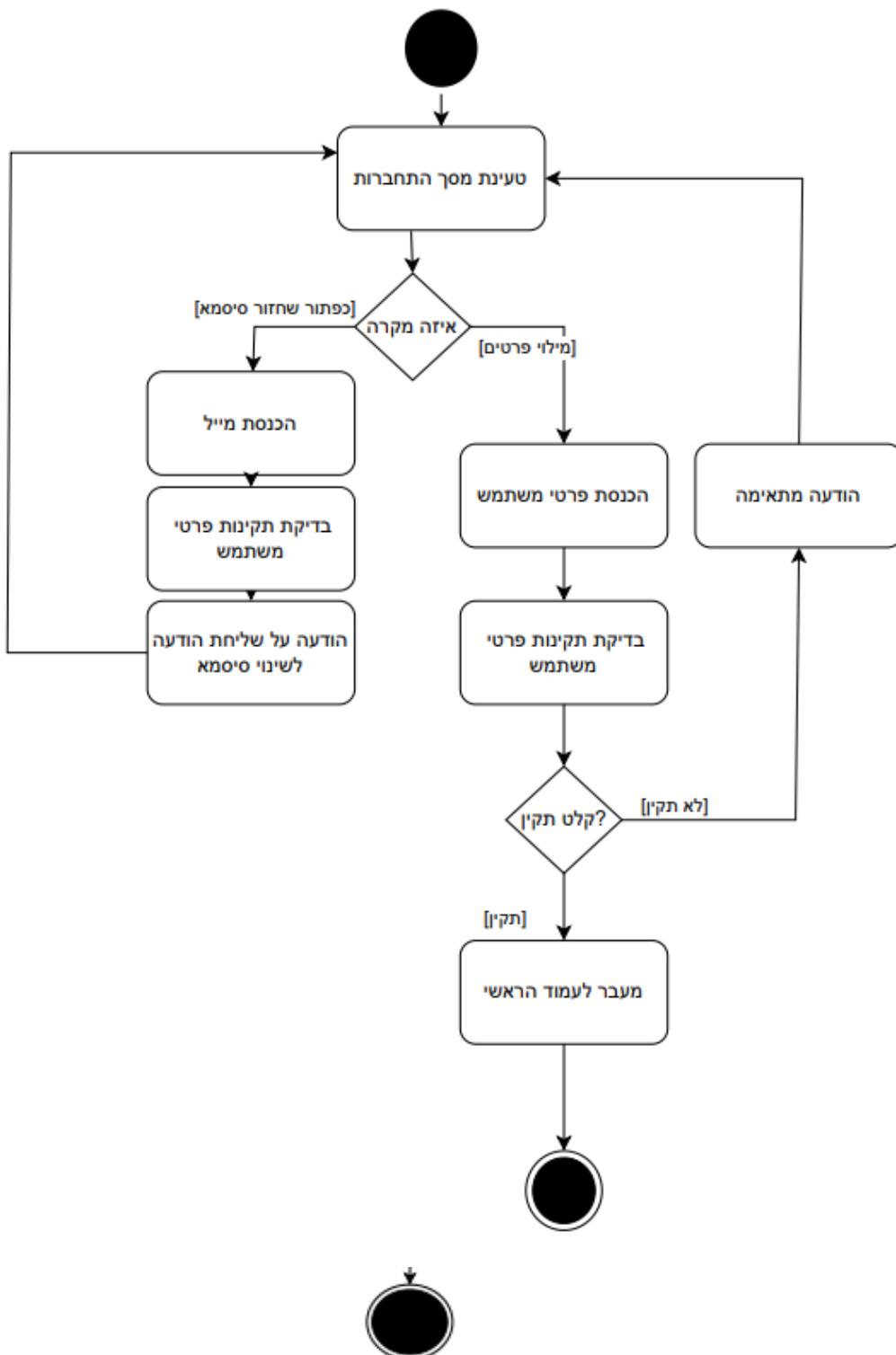
MissionName, MissionDescription, MissionLocation, MissionCreator	P3.1: ייצירת משימה	משתמש	F14: מידע ליצירת משימה
MissionName, MissionDescription, MissionLocation	P3.2: עדכון משימה	משתמש	F15: מידע לעדכון משימה
MissionComleter	P3.3: ביצוע משימה	משתמש	F16: מבצע המשימה
MissionId	P3.4: מהיקת משימה	משתמש	F17: קוד משימה
TotalScore	מידע משתמש P3.3: ביצוע משימה	משתמש F18: ניקוד כללי	
User Name, Password, UserLocation, phone, <u>UserId</u> , UserPicture, TotalScore Notification, LocationAccess, ContactAccess	P1: הרשמה, התחברות וניהול משתמש	משתמש	F19: פרטי משתמש
	מידע משתמש P1: הרשמה, התחברות וניהול משתמש	משתמש	
	מידע משתמש P1: הרשמה, התחברות וניהול משתמש	משתמש	
	משתמש P1: הרשמה, התחברות וניהול משתמש	משתמש	
	P1.2: התחברות	משתמש	
	משתמש P1.2: התחברות	משתמש	
GroupName, GroupId, GroupDescription, GroupPicture, GroupAdmin, GroupNotificaion, Role	P2: ניהול קבוצות	משתמש	F20: פרטי קבוצה
	קבוצות P2: ניהול קבוצות	管理体制	
	קבוצות P2.1: ייצירת קבוצה	管理体制	
MissionName, MissionId, MissionDescription, MissionLocation, MissionCreator, MissionComleter	מידע משתמש P1: הרשמה, התחברות וניהול משתמש	משתמש	F21: משימה
	מידע משתמש P1: הרשמה, התחברות וניהול משתמש	משתמש	
	משתמש P1: הרשמה, התחברות וניהול משתמש	משתמש	
	P1.2: התחברות	משתמש	
	משתמש P1.2: התחברות	משתמש	

	משימות	P4.4 מחייקת משימה	
GroupPicture	P2.4 – שינוי פרטי קובוצה	משתמש	F22 קובוצה תמונה
	קובצות	P2.4 – שינוי פרטי קבוצה	

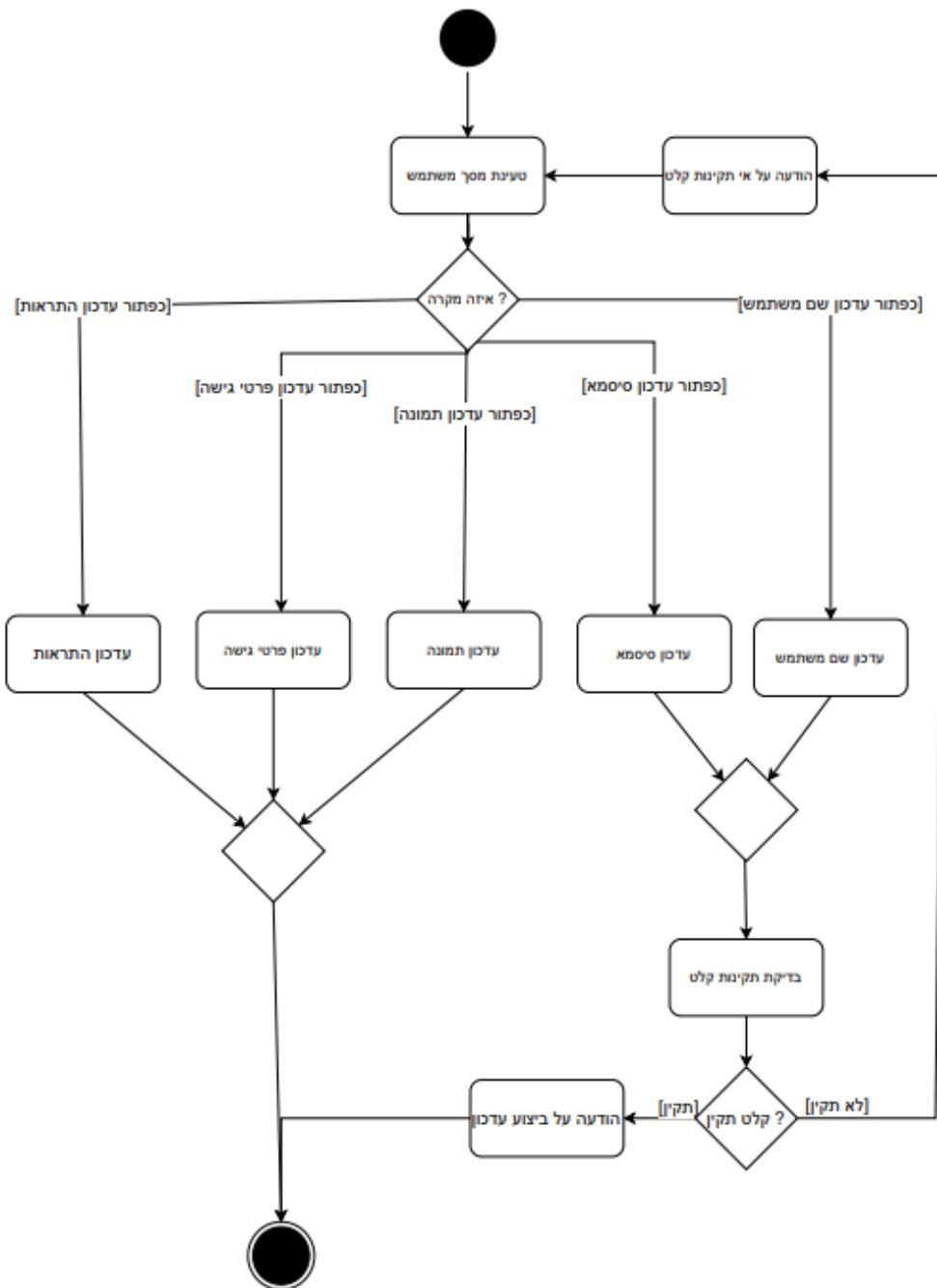


**Class
Diagram
pt.2**

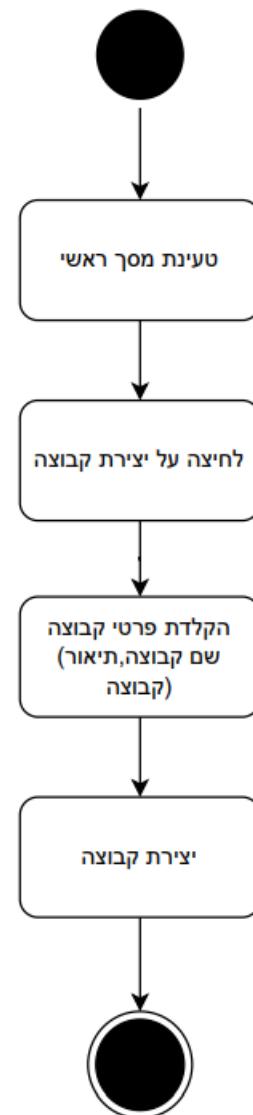


Activity Diagramכניסה משתמש

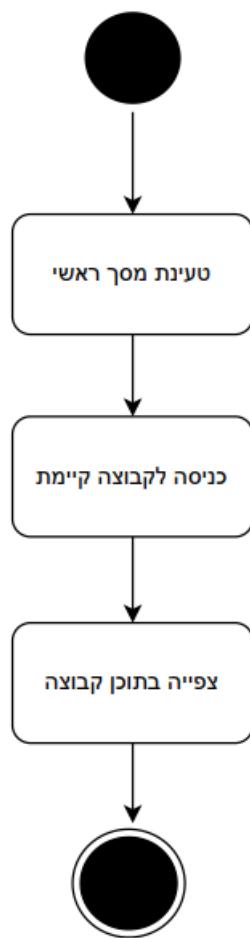
Activity Diagram
ניהול פרטי משתמש



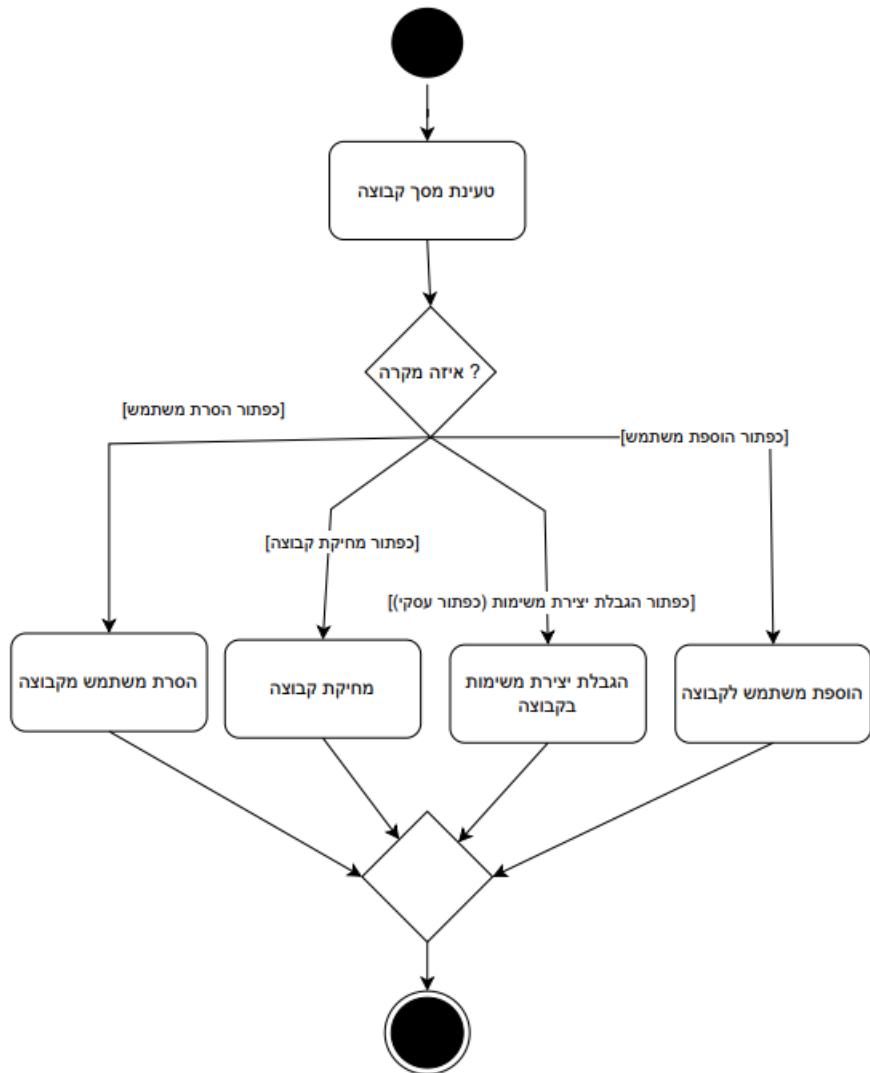
Activity Diagram
יצירת קבוצה



Activity Diagram
צפייה בתוכן קבוצה

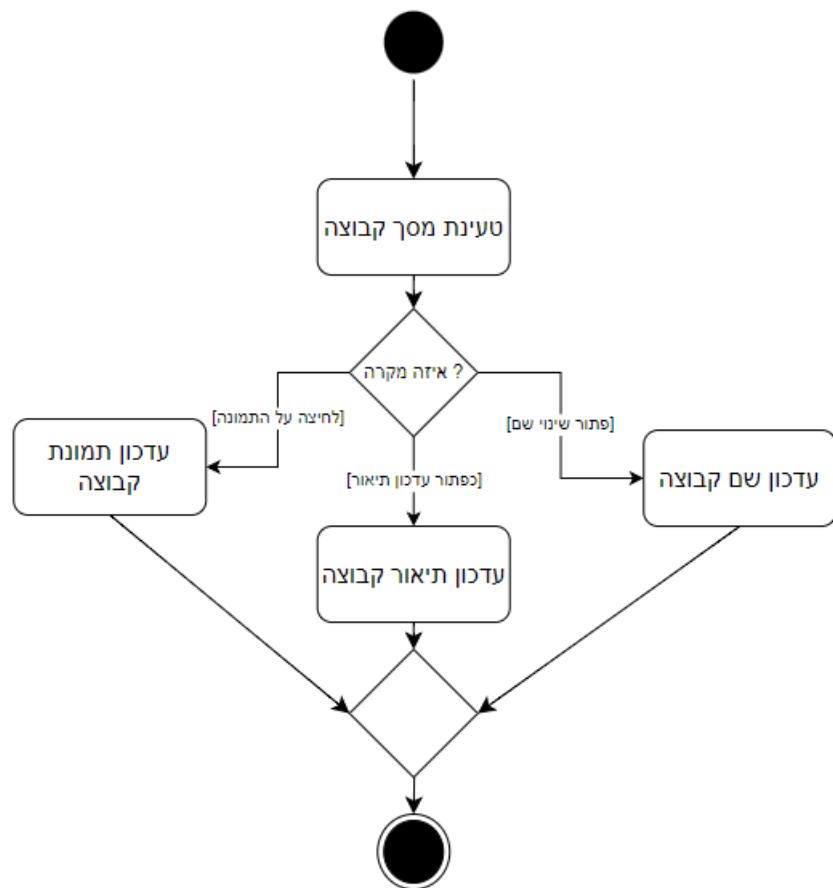


Activity Diagram
פעולות מנהל

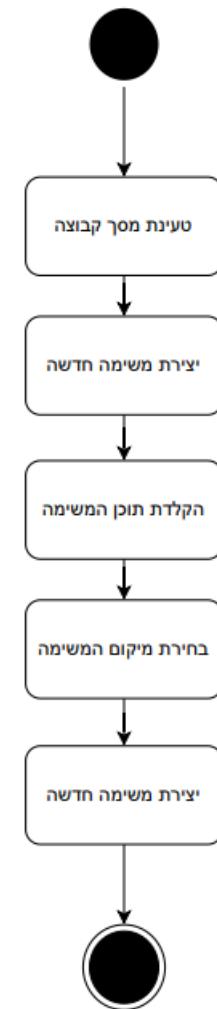


Activity Diagram

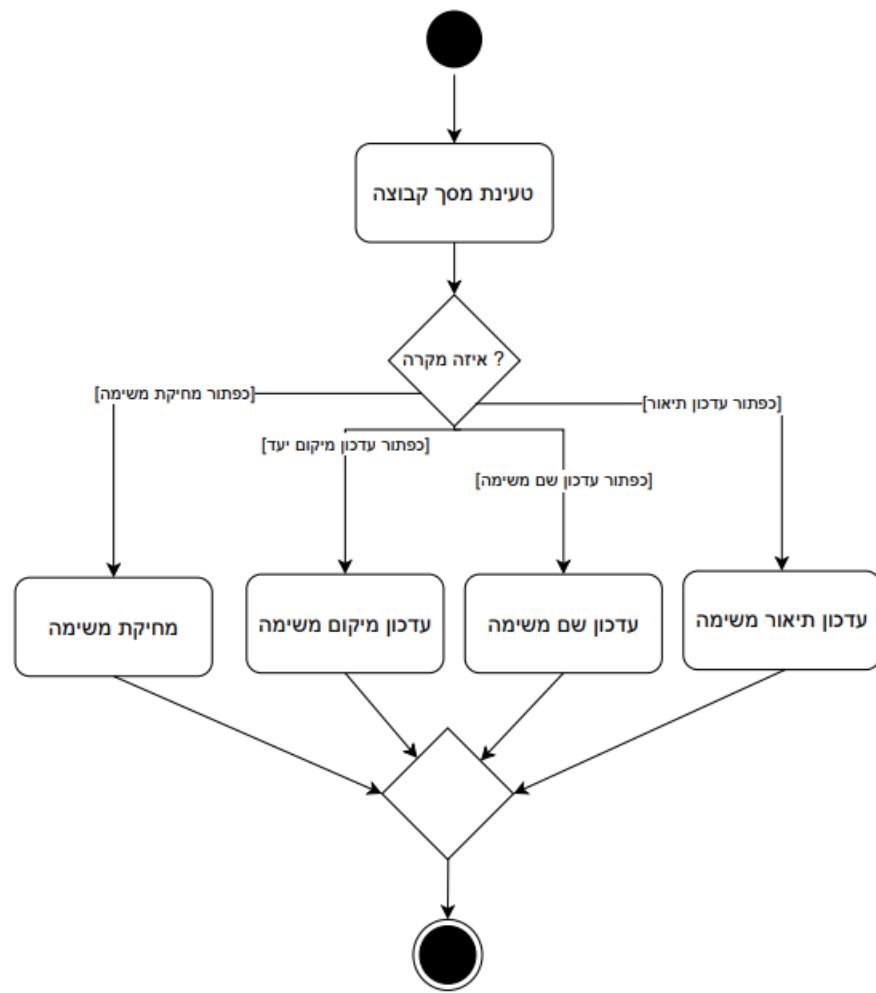
ניהול פרטי קבוצה



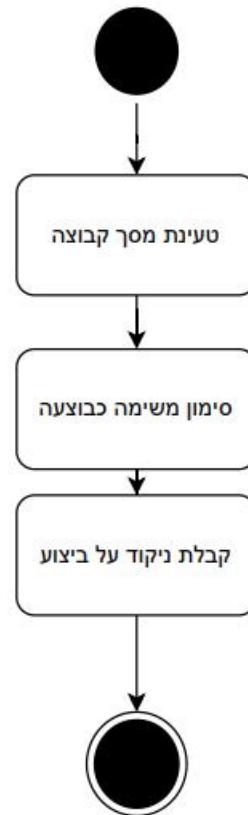
Activity Diagram
יצירת משימה



Activity Diagram
ניהול משימה



Activity Diagram
ביצוע משימה



11. תיאור/התיאchorות לנושאי אבטחת מידע

המערכת תהיה מערכת רגישה מבוססת מידע על משתמשים כדוגמת מיקום, קבוצות, תוכן קבוצות וכן תידרש הגנה גם על בסיס הנתונים וגם על המשתמשים האישיים באמצעות שם משתמש סיסמאות ו암צעי הגנה קיימים אצל נוטני שירותים בסיסי הנתונים

.12 משאבים הנדרשים לפרויקט :

1. מספר שעות המוקדש לפרויקט

300 שעות.

.2 ציוד נדרש -

מחשב, חיבור לאינטרנט, מכשיר אנדרואיד (אופציונלי).

.3 תוכנות נדרשות -

סביבת פיתוח האפליקציה - אנדרואיד סטודיו
בבסיס הנתונים – Fire Base

.4

ידע חדש שנדרש ללמידה לצורך ביצוע הפרויקט –

ללמוד איך לעבוד עם API שונים להתחמץ יותר בסביבת אנדרואיד ושייפה ללמידה של flutter ב כדי שהתוכנה תרוץ בכלל מערכות הפעלה טלפון

.5

ספרות ומקורות מידע –

[/https://developer.android.com](https://developer.android.com) - Developers android

[/https://www.tutorialspoint.com/android](https://www.tutorialspoint.com/android) - Tutorials point

[/https://www.youtube.com](https://www.youtube.com) - YouTube

[/https://www.geeksforgeeks.org](https://www.geeksforgeeks.org) - GeeksForGeeks

.13

תכנית עבודה ושלבים למימוש הפרויקט -

נition הפרויקט בתאריך 19.2.2023,
ממשך משתמש איך מסכים יראו בתאריך 14.4.2023,
כתיבת קוד בתאריך 15.7.2023,
בדיקות תוכנה בתאריך 15.8.2023,
כתיבת ספר פרויקט בתאריך 23.8.2023
הגשת ספר פרויקט בתאריך 1.9.2023

14. תכנון הבדיקות שיבוצעו

כיצד אבדוק את התהיליך	תיאור דרישת	מספר דרישת
טופס רישום שיבוצע בדיקת קלט ולפי זה יתבצע רישום למשתמש וטופס התחברות למשתמש שיבוצע גם תקינות קלט ובדיקה אל מול בסיס נתונים אם משתמש קיים	ממתק התתחברות ורישום	1
כנ"ל בדרישה 1	רישום משתמש	2
בקשת הרשות לפি פעולות קיימות באנדרואיד סטודיו	הרשאות מיקום: ושיתוף אנשי קשר האפליקציה תבקש הרשות מהמכשיר עליה מותקנת	3
אימות מייל ומשלווה שהזור סיסמא	פעולות שיזור משתמש	4
כנ"ל בדרישה 1	התחברות משתמש	5
ווצג עמוד ראשי שיכיל רשימה של כל הקבוצות	עמוד קבוצות (עמוד ראשי) בו יוצגו קבוצות של המשתמש ופעולות פתיחת קבוצה	6
הकפצת התראות לפি אלגוריתם שהייה תלוי במיקום היעד של משימות קיימות והמיקום הנוחכי של כל משתמש	הකפצת התראות על פי מיקום המשימות למשתמש	7
אזור בו המשתמש יוכל לשנות מידע אישי ולעדכו פרטיהם (הגדרות)	ממתק משתמש (פרטים אישיים, אישור מיקום, וניקוד משוקלל של כל הקבוצות)	8

העומד בו ייצג התוכן של כל קבוצה (המשימות הקיימות אפשרויות וצפיה בפרט קבוצה וכו)	משחק קבוצה מוצגת רשות המשימות שניתנו לעדכון ע"י כל משתמש הקבוצה, אפשרות ביצוע פעולות של יצירה/עדכון/מחיקת משימה	9
אלגוריתם אשר יעקוב לכל משתמש בכל קבוצה אחר הניקוד שלו לפי ביצוע המשימות	מתן ניקוד: תינתן נקודה קבוצתית למשתף שביצע משימה בהצלחה, וצבר ניקוד כללי לכל משתמש המשתמש יעלה דמות	10

קוד כמה שיטור תיקון ויעיל	ביצוע פעולות מהיר ורפואי	11
התmeshקות עם מפות קיימות ככל הנראה גוגל ב כדי לקטלג מיקומים קיימים כמו סופרים בת ספר בית וכו'	הגדרת משימה טקסט חופשי והגדרת מקום דרוש למשימה על פי מיקומים מוגדרים מראש	12
תפריט ראשי שהמשתמש יוכל לעבור בין עמודים	תפריט תפעולי ב כדי לעבור בין משקים ועמודים	14
תכנות התחלתי באנדרואיד סטודיו בגרסה לollipop	התוכנית ת clues על גירסה lollipop	15
המשתמש יוכל לבצע פעולות בקבוצה יצירה ועדכון משימות ועדכון פרט קבוצה	אפשרות ביצוע פעולות בקבוצה עריכה/הוספה/הסרת משתמש	16



חתימת המנהה האישי

חתימת הסטודנט

.3. הערות ראש המגמה במכללה

מאשרת

.4. אישור ראש המגמה

שם : _____
חתימה : *כ.יריה כב*
תאריך : 2/7/23

.5. הערות הגורם המקצועי מטעם מה"ט

.6. אישור הגורם המקצועי מטעם מה"ט

שם : _____
חתימה : _____
תאריך : _____

תוכן עניינים

נושא	ע"מ
• מבוא	51
• נושאים הקשורים לניתוח מערכת :	52 - 63
	<ul style="list-style-type: none"> ◦ דיאגרמת קשר Dfd ◦ מילון נתונים ▪ תיאור ישויות ▪ תיאור מאגרים ▪ תיאור זרימות ▪ תיאור תהליכי מבני נתונים • חלופות שפת מימוש • פירוט בדיקות תוכנה ואופן ביצוע
• תיאור אלגוריתמים לפתרון הבעיה	64
• מדריך למשתמש בתוכנה	65 - 74
• תרשימים מחלקות קוד	75 - 196
• סיכום וסיכום	197

מבוא

בעידן המודרני בו אנו חיים, הטכנולוגיה היא חלק בלתי נפרד מהיינו הימויומיים. היא משפרת את איכות החיים שלנו, מקלת علينا ביצוע משימות ומאפשרת לנו להתמודד עםאתגרים שונים בצורה הטובה ביותר.

אחד הביעות המרכזיות שאנו בתור אנחנו עוסקים נתקלים בה היא ניהול הייעיל של משימות ומטלות, במיוחד כאשר מדובר במשימות קבוצתיות.

אם הייתה לך פעם רגשך שאתה מתבקש לנהל את הזמן שלך?

אם חווית סיטואציות בהן הייתה לך משימה לביצוע ושכחת אותה?

אם פעמי רגשך שאתה לא בטוח במה המשימות שאתה צריך לבצע?

אם התשובה היא כן, אז הפרויקט הזה הוא בשביבך.

יצירת אפליקציה לניהול משימות קבוצתיות היא הפתרון לבעה הנ"ל. הרעיון מגיע מה הצורך המתמיד לנהל ולבצע משימות בקבוצה, בין אם במסגרת העבודה, המשפחה או אפילו בני אדם עצמאיים.

האפליקציה מציעה לקבוצות אנשים לנוהל את המשימות הפרטיות והקבוצתיות שלהם בצורה יעילה וחכמתית, תוך התמסוקות עם המיקום במכשיר. זה מאפשר למשתמשים לקבל התראות על משימות לביצוע בהתאם למיקום המשימה, ובכך להגבר את העילות ולהסוך זמן.

האפליקציה מתאימה למגוון עבודות, כמו שליחויות, מסעדות, משפחות ו אף לניהול אישי.

הפתרון הוא שימושתפי הקבוצה יכולם להיעזר אחד בשני, ביצע את המשימות שלהם ביעילות ולעקוב אחר המשימות של כל אחד. לדוגמה, במשפחה שבה כולם בעלי רישיון נהיגה, האם יכולה ליזור משימה לאסיפת דואר. כאשר אחד מבני המשפחה מתקרב למקום הדואר, הוא יקבל התראה ויכול לבצע את המשימה בקלות.

אחד ה יתרונות המרכזיים של האפליקציה זה מערכת הניקוד בה.

בכל פעם שמשתמש מבצע משימה, הוא מקבל ניקוד. הניקוד משמש כאמצעי מוטיבציה למשתמשים לבצע את המשימות שלהם בצורה יעילה וזמן. בנוסף, הניקוד מאפשר למשתמשים לראות את ההתקדמות שלהם ולהציג את ההישגים שהם מגיעים אליהם. הניקוד יכול לשמש גם כאמצעי תחרותי בין המשתמשים בקבוצה.

לדוגמה, במשפחה או בקבוצה עובדה, המשתמשים יכולים להתחרות ביניהם על מי יצבור את הניקוד

הגבוה ביותר בסוף השבוע או החודש. זה יכול להויסף אלמנט של كيف ותחרות לביצוע המשימות ולהגבר את המוטיבציה להשלמתן.

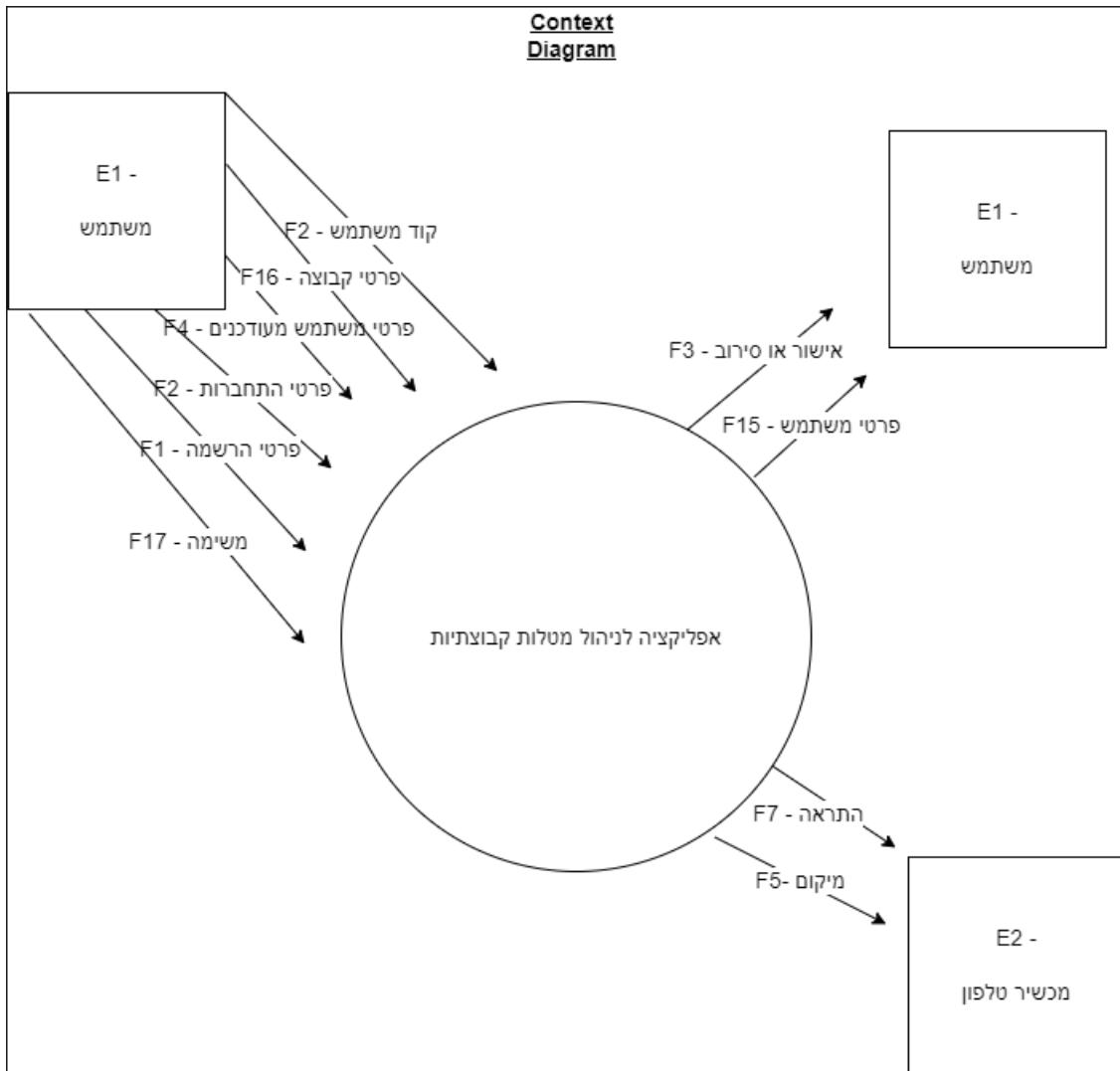
נושאים הקשורים לניתוח מערכת

1. טבלת דרישות שהשתנו

מספר דרישת מקורית	תיאור דרישת מקורית	סיבה לשינוי (דרישה החדשה, נוסח השנתנה, דרישת בוטלה)	תיאור חדש של דרישת

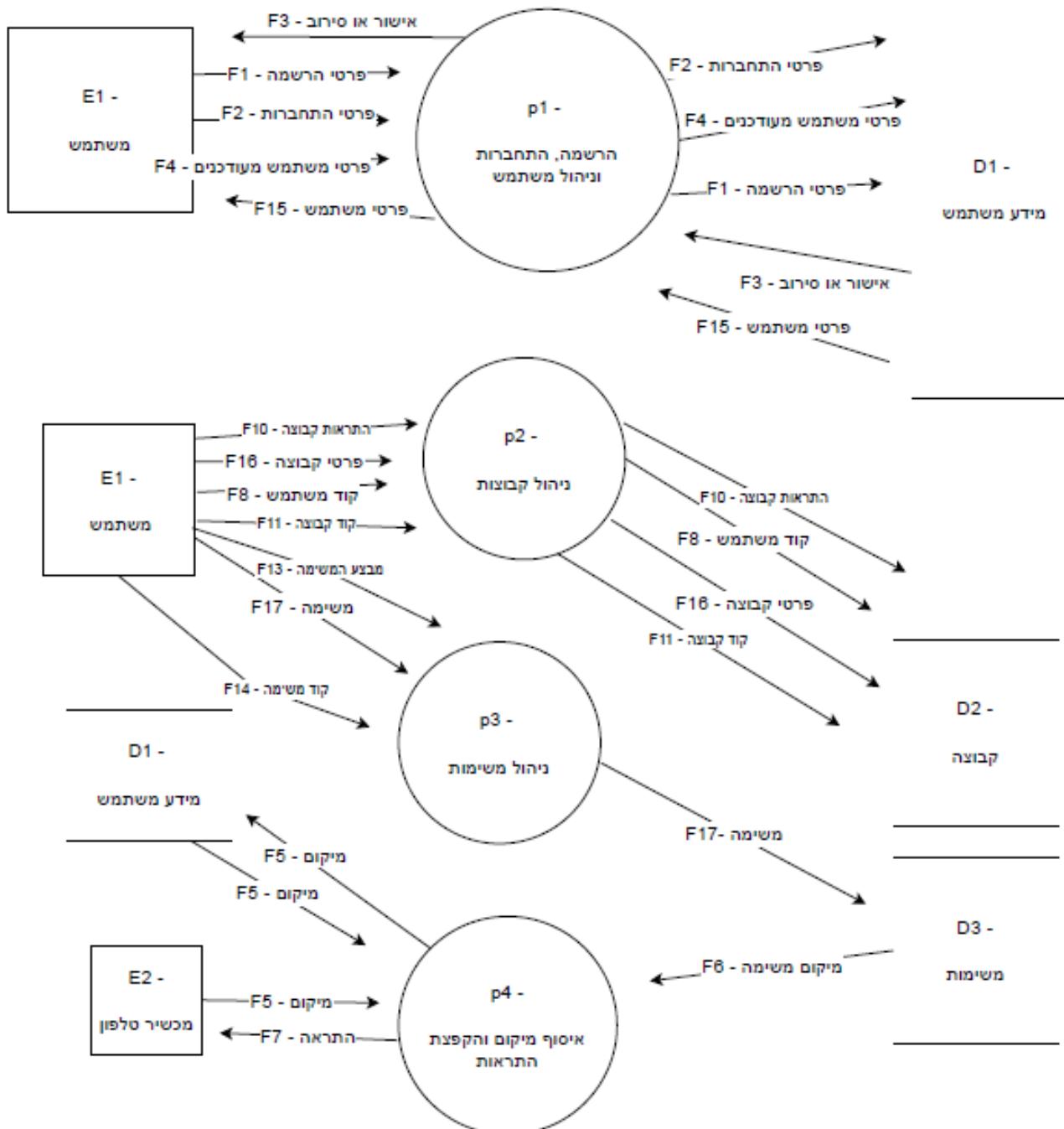
16	עריכה/הוספה/הסרת משתמש	אפשרות ביצוע פעולות בקבוצה עריכאה/הוספה/הסרת משתמש ע"פ רשות החברים	תיאור חדש של דרישת
נוספה החברים	חיפוש והוספה חבר לרשותם	בכדי להוסיף חבר לקבוצה קודם כל צריך לחפש ולהוסיףו לרשותם החברים על פי האימייל	תיאור חדש של דרישת
נוספה חדש	התראה ליוצר המשימה ואישור על ידו על ביצוע המשימה	לאחר שעדכנה משימה כבוצעה צריכה אישור מיווצר המשימה כדי להתעדכן כבוצעה	תיאור חדש של דרישת

.2 דיאגרמת הקשר (Context Diagram)

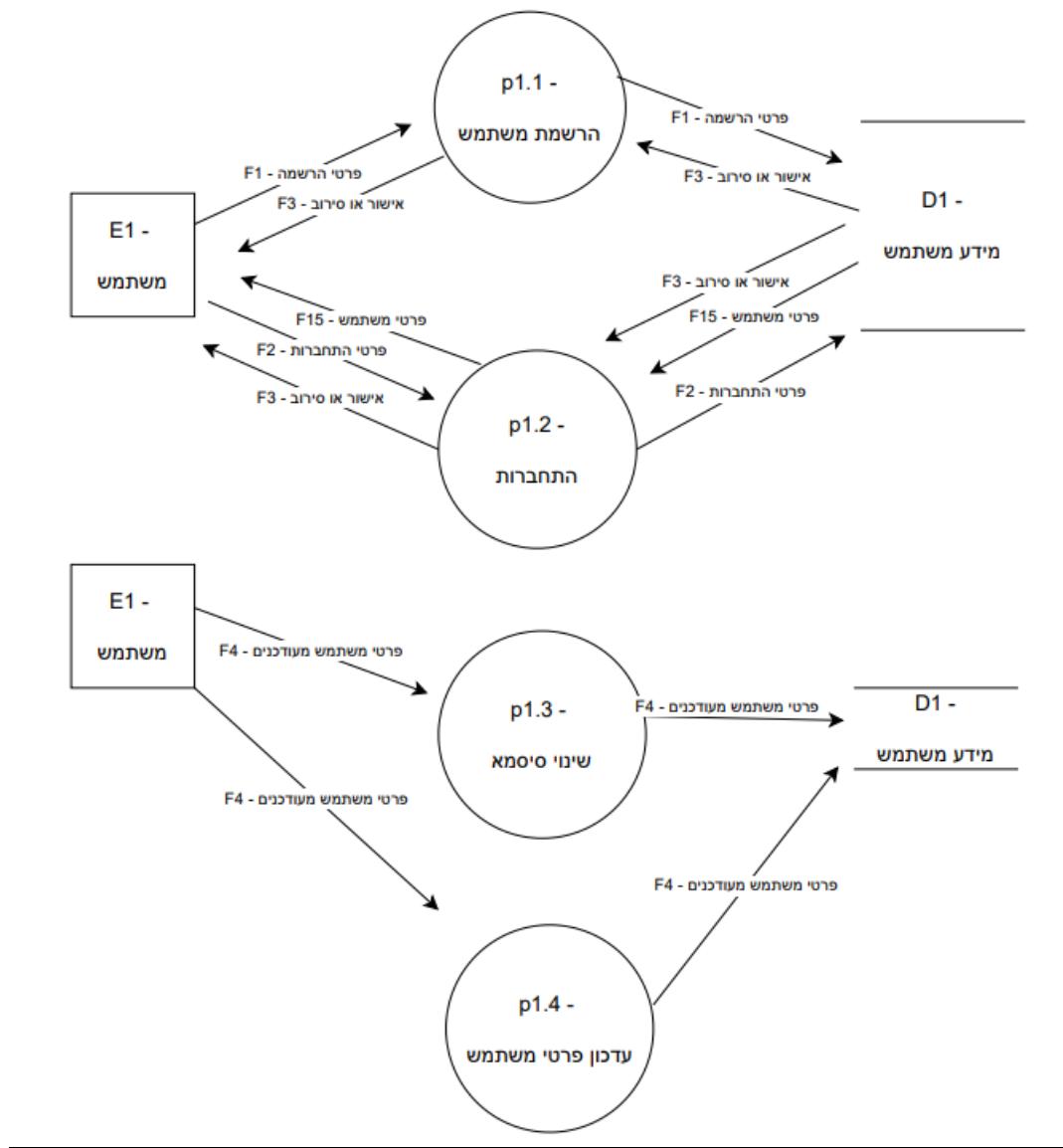


.3 ניתוח התהליכיים העיקריים.

DFD 0



DFD 1.1



מילון נתונים
תיאור ישויות

תיאור	שם היחסות	קוד
משתמש האפליקציה מייצר ומשתמש בקבוצות מייצר ומבצע משימות	משתמש	E1
המכשור שעליו ושבת האפליקציה מקבל ומתריאה התראות שלוח מקום	מכשור טלפון	E2

תיאור המאגרים

שדות הטבלה	שם המאגר	קוד
date, email, image, ,name, token, userId	מידע משתמש	D1
date, groupId, businessMode, creatorUid, friendsPoints, groupName, image, lastTaskCreator, taskCount, taskCreationDate	קבוצות	D2
date, taskId, assignedTo, assingedToUid, creatorUid, dateCreated, deadline, deadlineTime, isTask, location, status, taskIndex, taskInfo, whoDid	משימות	D3

תיאור זרימות

שדות/zורימה	יעד	מקור	קוד + שם זרימה
name, password, email	P1: הרשמה, התהבותות וניהול משתמש	משתמש	F1: פרטי הרשמה
	מידע משתמש	הרשות, התהבותות וניהול משתמש	
	P1.1: הרשמה משתמש	משתמש	
	מידע משתמש	P1.1: הרשמה משתמש	
email, password	P1: הרשמה, התהבותות וניהול משתמש	משתמש	F2: פרטי התהבותות
	מידע משתמש	הרשות, התהבותות וניהול משתמש	
	P1.2: התהבותות	משתמש	
	מידע משתמש	P1.2: התהבותות	
הודעה מתאימה בקשר לניסיון רישום/התהבותות	P1: הרשמה, התהבותות וניהול משתמש	מידע משתמש או	F3: אישור או סירוב

	P1: הרשמה, התחברות וניהול משתמש	משתמש	
	P1.1: הרשמה	משתמש	
	P1.1: הרשמה	משתמש	
	P1.2: התחברות	מיידע משתמש	
	P1.2: התחברות	משתמש	
,Email, name, password, image	P1: הרשמה, התחברות וניהול משתמש	משתמש	F4: פרטי משתמש מעודכנים
	P1: הרשמה, התחברות וניהול משתמש	מיידע משתמש	
	P1.3: שינוי סיסמא	משתמש	
	P1.3: שינוי סיסמא	מיידע משתמש	
	P1.4: עדכון פרטי	משתמש	
	P1.4: עדכון פרטי	מיידע משתמש	
location	P4: איסוף מיקום והקפצת התראות	מכשיר טלפון	F5: מקום
	P4: איסוף מיקום והקפצת התראות	מיידע משתמש	
	P4.1: קליטת מיקום מהמכשיר של המשתמש	מכשיר טלפון	
	P4.1: קליטת מיקום מהמכשיר של המשתמש	מיידע משתמש	
	p4.2: קליטת מיקום מהמשימה, השוואת מיקומים עם ממושך גוגל למקומות קיימים והתראה בהתאם	מיידע משתמש	
location	P4 – איסוף מיקום והקפצת התראות	משימות	F6: מקום משימה
	p4.2: קליטת מיקום מהמשימה, השוואת מיקומים עם ממושך גוגל למקומות	משימות	

		קיימים והתראה בהתאם		
מטריה ההתראה בהתאם למשימות שקיימות בקבוצות שהמשתמש משתמש בהן ולמקום בו המשמש נמצא	מכשיר טלפון	P4 – איסוף מקום והקפצת התראות	F7 : התראה	
UserId	P2.2 : הזמנה משתתף	משתמש	F8 : קוד משתמש	
	קבוצות	P2.2 : הזמנה משתתף		
	P2.3 : מהיקת משתמש	משתמש		
	קבוצות : מהיקת משתמש	P2.3 : מהיקת משתמש		
GroupName	P2.1 : ייצירת קבוצה	משתמש	F9 : שם קבוצה	
	קבוצות : ייצרת קבוצה	P2.1 : ייצרת קבוצה		
	p2.4 : שינוי שם/תיקור קבוצה	משתמש		
	קבוצות : שינוי שם/תיקור קבוצה	p2.4 : שינוי שם/תיקור קבוצה		
GroupNotification	p2.5 : השתקת קבוצה	משתמש	F10 : התראות קבוצה	
	קבוצות : השתקת קבוצה	p2.5 : השתקת קבוצה		
GroupId	P2.6 : יציאה מהקבוצה	משתמש	F11 : קוד קבוצה	
	קבוצות : יציאה מקבוצה	P2.6 : יציאה מקבוצה		
taskInfo,, location, creatorUid	P3.1 : ייצירת משימה	משתמש	F12 : מידע ליצירת משימה	
whoDid	P3.3 : ביצוע משימה	משתמש		F13 : מבצע המשימה
taskId	P3.4 : מהיקת משימה	משתמש	F14 : קוד משימה	
date, email, image, ,name, token, userId	P1 : הרשמה, התחברות וניהול משתמש	משתמש		F15 : פרטי משתמש
	מידע משתמש	P1 : הרשמה, התחברות וניהול משתמש		

	P1: הרשמה, התחברות וניהול משתמש	מידע משתמש	
	משתמש	P1: הרשמה, התחברות וניהול משתמש	
	P1.2: התחברות	מידע משתמש	
	משתמש	P1.2: התחברות	
date, groupId, businessMode, creatorUid, friendsPoints, groupName, image, lastTaskCreator, taskCount, taskCreationDate	P2: ניהול קבוצות	משתמש	F16: פרטី קבוצה
	קבוצות	P2: ניהול קבוצות	
	קבוצות	P2.1: יצירת קבוצה	
date, taskId, assignedTo, assingedToUid, creatorUid, dateCreated, deadline, deadlineTime, isTask, location, status, taskIndex, taskInfo, whoDid	P1: הרשמה, התחברות וניהול משתמש	מידע משתמש	F17: משימה
	P1: הרשמה, התחברות וניהול משתמש	מידע משתמש	
	משתמש	P1: הרשמה, התחברות וניהול משתמש	
	P1.2: התחברות	מידע משתמש	
	משתמש	P1.2: התחברות	
	משימות	P4.4: מהיקת משימה	
image	P2.4 – שינוי פרטי קבוצה	משתמש	F18: תמונה קבוצה
	קבוצות	P2.4 – שינוי פרטי קבוצה	

מבנה הנתונים .3



Users(date, email, image, name, token, uid)

Friends(friendRef, uid)

CheckList(date, uid, done, groupName, groupUid, taskInfo, taskUid, whoDid, whoDidName)

UserGroups(countSeen, uid, taskCreationDate, groupRef)

Groups(date, uid, businessMode, creatorUid, friendsPoints, groupName, image, lastTaskCreator, taskCount, taskCreationDate)

Tasks(date, uid, assignedTo, assingedToUid, creatorUid, dateCreated, deadline, deadlineTime, isTask, location, status, taskIndex, taskInfo, whoDid)

4. חלופות שפת מימוש -
java – android studio

kotlin – android studio

dart - VSCode

שפת מימוש הפרויקט נבחרה להיות flutter במסגרת dart בהירה זו הוחלה עקב רצון לאטגר את עצמי בשפה חדשה ומשום שהמסגרת מכנה בתכנון אחד יצירה של אפליקציה גם לאייפון וגם لأنדרואיד. במסגרת פרויקט זה נחשפי לשימוש רחב בסביבת הפיתוח VSCode שמצאתי אותה נוחה יותר.android studio.

הבחירה לכתחוב ב dart לבסוף הייתה בהירה מזוינה מאוד תחברתי לסביבת הפיתוח, למדתי שפה חדשה ומסגרת חדשה שמקיפה גם אנדרואיד וגם אייפון שגם הצלחתו למצואו אותה יותר ופשטה יותר. למימוש מאשר ב java דרך android studio שלבסוף מיצרת אפליקציה لأنדרואיד בלבד.

5. פירוט בדיקות תוכנה ואופן ביצוען -

מס' דרישة	תיאור דרישة	כיצד אבדוק את התהיליך	תוצאה רצiosa	תוצאה מתקבלת
1	ממתק התהברות ורישום	טופס רישום שיבוצע בדיקת קלט ולפי זה יבוצע רישום למשתמש וטופס התהברות למשתמש שיבוצע גם תקינות קלט ובדיקה אל מול בסיס נתונים אם משתמש קיים	הצלחה ברישום ובהתחברות הקמת משתמש בסיס הנתונים	הצלחה בהרשמה והקמת משתמש בסיס הנתונים
2	רישום משתמש	כנ"ל בדרישה 1	הצלחה בהרשמה	הצלחה בהרשמה והקמת משתמש בסיס הנתונים
3	הרשאות מיקום: ושיתוף אנשי קשר האפליקציה תבקש הרשאות מהמכשיר עליה מותקנת	בקשת הרשות לפיצולות קיימות באנדרואיד סטודיו	בהתהברות ראשונית בטלפון הקפצת חלונות לבקשת אישור התראות ומיקום	בהתהברות ראשונית בטלפון הקפצת חלונות לבקשת אישור התראות ומיקום
4	פעולות שייחזור משתמש	אימות מייל ומשLOW שחזר סיסמא	קבלת מייל ושינוי סיסמא	קבלת מייל ושינוי סיסמא
5	התהברות משתמש	כנ"ל בדרישה 1	הצלחה בהתחברות כניסה למסך	הצלחה בהתחברות כניסה למסך

ראשי של האפליקציה	ראשי של האפליקציה			
הציג עמוד הקבוצות אחורי ההתחברות	הציג עמוד הקבוצות אחורי ההתחברות	יוזג עמוד ראשי שיכיל רשימה של כל הקבוצות	עמוד קבוצות (עמוד ראשי) בו יוצגו קבוצות של המשמש ופועלות פיתחת קבוצה	6
במידה והמשתמש נמצוא ליד המיקומים הקיימים במשימות תתקבל בתפקיד המשמש התראה למשתמש שקרוב למיקום ישנה משימה בו למשתמש		הkpצת התראות לפי נמצוא ליד המיקומים הקיימים והמקום הנוחכי של כל המשמש	הkpצת התראות על פי מקום המשימוש למשתמש	7
כניסה למסך הגדירות משתמש לשינוי פרטיים	כניסה למסך הגדירות משתמש לשינוי פרטיים (הגדירות)	כניסה בו המשמש יכול לשנות מידע אישי ולעדכו פרטיים (הגדירות)	מסך משתמש (פרטים אישיים)	8
כניסה למסך הגדירות קבוצה לשינוי פרטי קבוצה	כניסה למסך הגדירות קבוצה לשינוי פרטי קבוצה	העמוד בו יוזג התוכן של כל קבוצה (המשימות הקיימות אפשרויות וצפיה בפרטית הקבוצה וכו')	מסך קבוצה מוצגת, אפשרות ביצוע פעולות של יירה/עדכון/מחיקת משימה	9
לאחר ביצוע ניקוד משתמש עולה מבצע עולה	לאחר ביצוע ניקוד משתמש עולה	אלגוריתם אשר יעקוב לכל שימוש בכל קבוצה אחר הניקוד שלו לפי ביצוע המשימות	מתן ניקוד: תינתן נקודה קבוצתית למשתף שביצע משימה בהצלחה	10
לחפש חבר לפי אימיל למצוואו ולהוסיף לחברים בהצלחה	לחפש חבר לפי אימיל למצואו אותו ולהוסיף לחברים בהצלחה	בכדי להוסיף חבר לקבוצה קודם כל צריך לחפש ולהוסיף לרישימת החברים על פי האימיל	חיפוש והוספה חבר לרישימת החברים	11
מחיקת חבר קיים מהרשימה בהצלחה	מחיקת חבר קיים מהרשימה בהצלחה	האפשרות להסיר חבר מרישימת החברים	ניהול רשימת החברים	12
לאחר להחיצה על ביצוע משימה מעבר למצב המתנה	לאחר להחיצה על ביצוע משימה מעבר למצב המתנה ושליחת	לאחר שעודכנה משימה כבוצעה צריכה אישור מיוצר המשימה כדי להתעדכן כבוצעה	התראה ליוצר המשימה ואישור על ידו על ביצוע המשימה	13

ושליחת התראת ליוצר המשימה וה	התראה ליוצר המשימה וה			
---------------------------------------	--------------------------	--	--	--

תיאור אלגוריתמים לפתרון הבעיה

האפליקציה נבנתה בשפת dart בשימוש בפרויומוורק flutter התממשקות עם google maps API ו firebase NODEJS message שנבנתה בJS בנוסף מושלבת פונקציה לשילוח התראות דרך firebase message

– קיים שימוש ב-MVC –

Model : notificationModel, groupModel, taskModel, userModel
מציגים את מבנה הנתונים

The widgets that display the UI on flutter : View
הנתונים המוצגים באמצעות הויזג'טים באפליקציה

Controller : checkListController, groupController, taskController, userController
notificationController
שלוטים בשימוש המידע דרך בסיס הנתונים לאפליקציה

אלגוריתמים מורכבים

submitForm – פונקציה זו אחראית ליצירת קבוצה חדשה. היא מבצעת משימות שונות כמו הגדרת הקבוצה ב-Firebase, שיר חברי קבוצה, ייצרת משנה ראשונית לקבוצה ויצירת דיאלוגים. המורכבות נובעת מהאופי האсинכרוני של פעולות Firestore ומהצורך לתאם עדכוני נתונים מרובים אינטראקטיבים או במהלך ייצרת הקבוצה. - (ח)>O כאשר הוא מספר החברים שנבחרו.

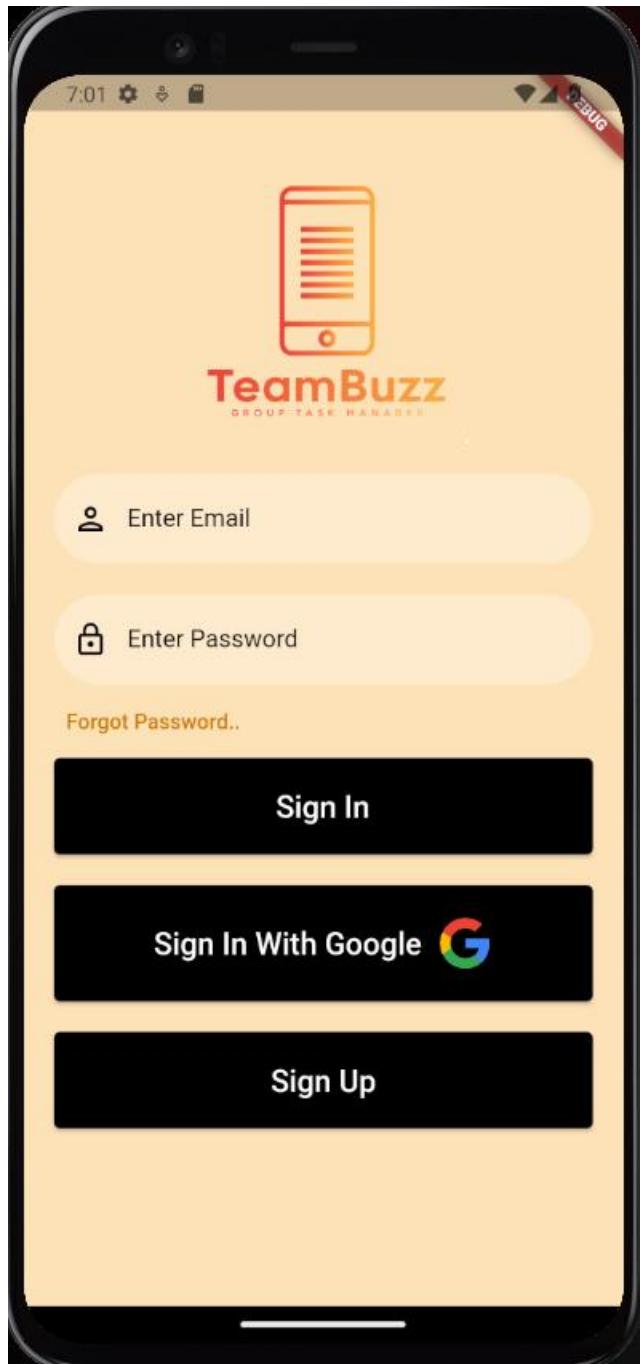
makeNamesArray – פונקציה זו פותרת את הבעיה של אחזר וארגן מידע חברים ביצירת קבוצה היא עוברת על מילון שמכיל UID של כל חברי הקבוצה מושך את שם החבר מה Firestore באמצעות הUID פונקציה זו אחראית על האפשרות להציג את שמות חברי הקבוצה בצדיה להקצותם אליהם משימות. - (ח)O, כאשר הוא מספר החברים.

getTasksLocation – פונקציה זו פותרת את הבעיה של איסוף וציבורה של מיקומי משימות עבור המשתמש על פניו מספר קבוצות. הוא עוקב אחר מבנה היררכי, מנוט בין קבוצות, משימות והקצאות משתמשים כדי לבחור ולאסוף מיקומים על סמך קритריונים ספציפיים.

uploadImageToFireBase – פונקציה אחראית להעלאת תמונות פרופיל משתמש ל-Firebase Storage. הוא יוצר שם קובץ ייחודי לתמונה, יוצר הפניה לאחסון ומבצע את ההעלאה ומבטיח שכל תמונה מזוהה באופן ייחודי. לאחר העלאה מעדכן את כתובת ה-URL של תמונת הפרופיל של המשתמש ומשתתקף בשינוי באפליקציה. פונקציה זו מפשטת את תהליך ניהול התמונות עבור משתמשים – (1)O

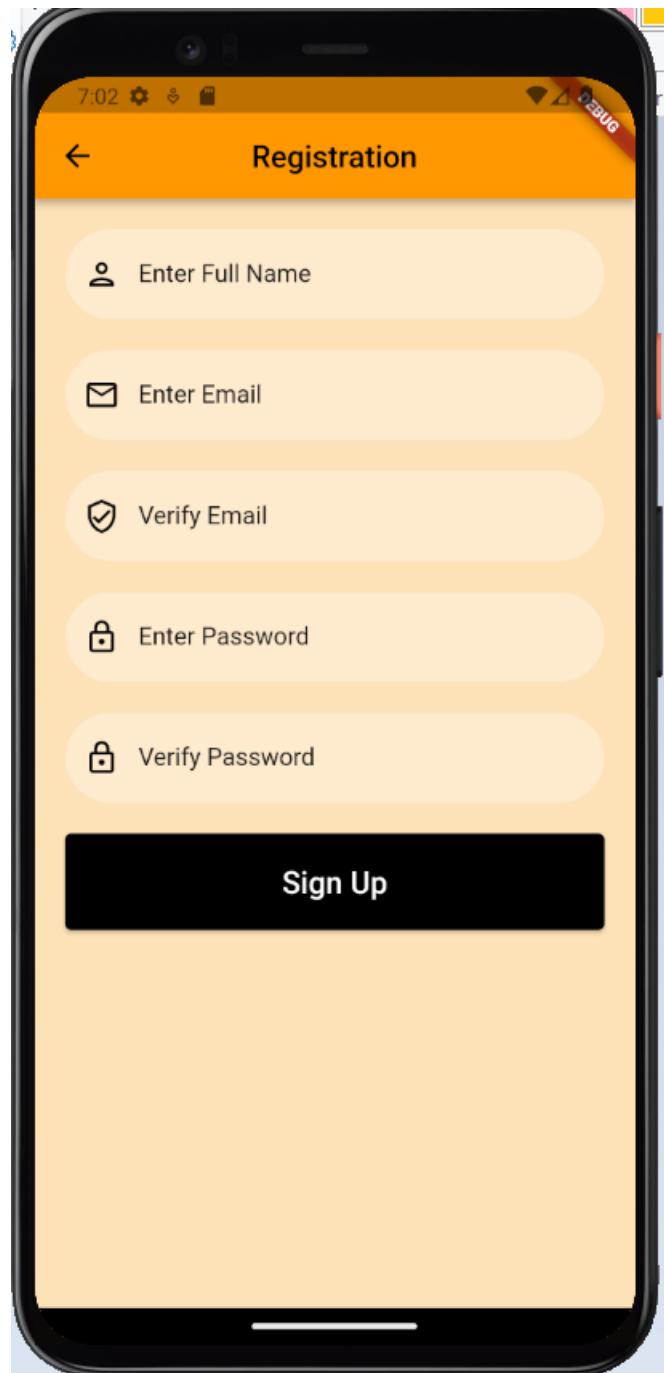
שימוש בשליפת נתונים מבוסיס הנתונים ובנית ויידג'טים באופן דינמי

מדריך למשתמש בתוכנה



מסך התחברות

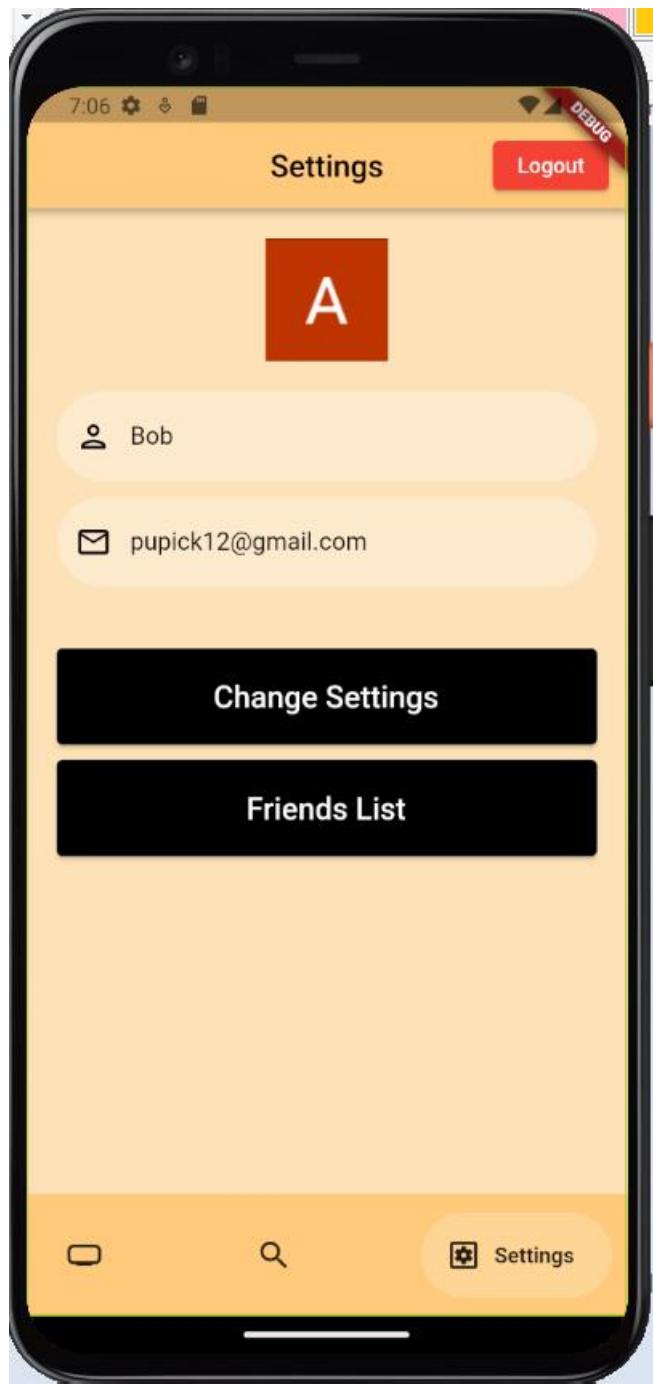
- כפתור שכחתי סיסמה
- כפתור מעבר למסך הרשומה
- כפתור התחברות והתחברות באמצעות גוגל



מסך הרשמה



מסך חיפוש חברים



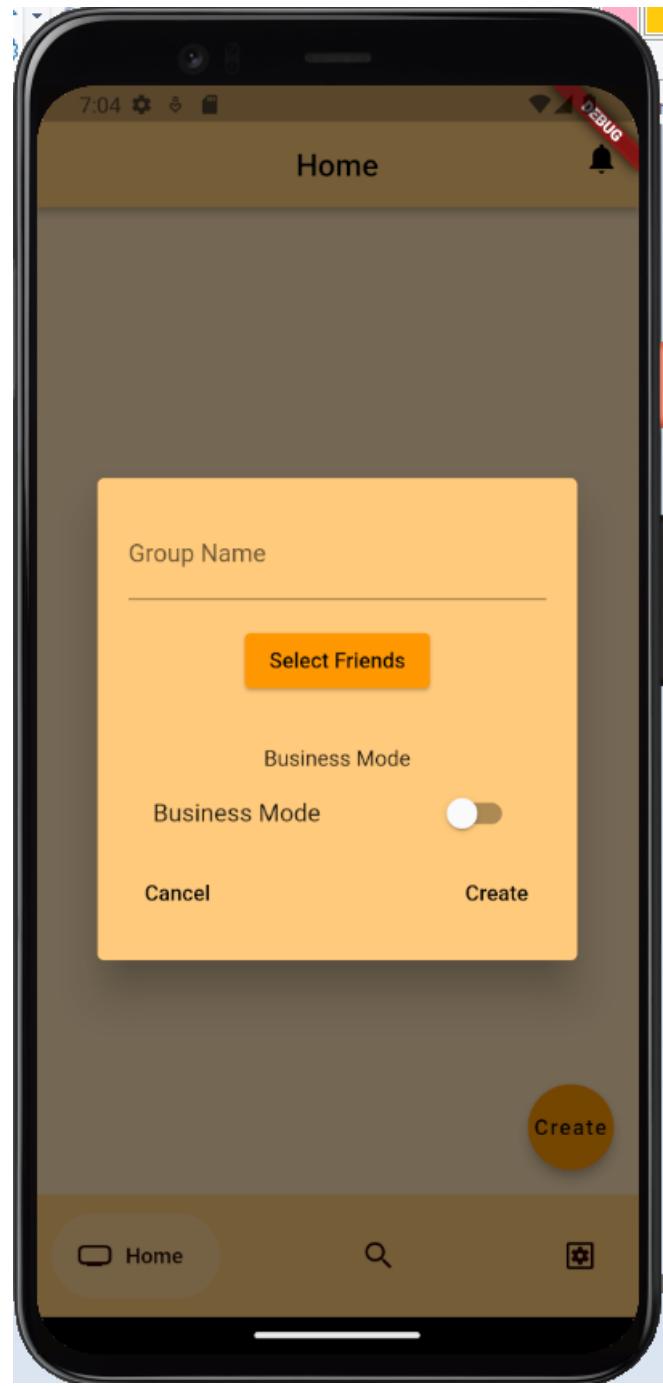
מסך הגדרות משתמש

- כפתור לשינוי מידע (מייל או שם)
- כפתור למעבר למסך רשימת החברים
- לחיצה על התמונה תתן אפשרות להחליפּ תמונה



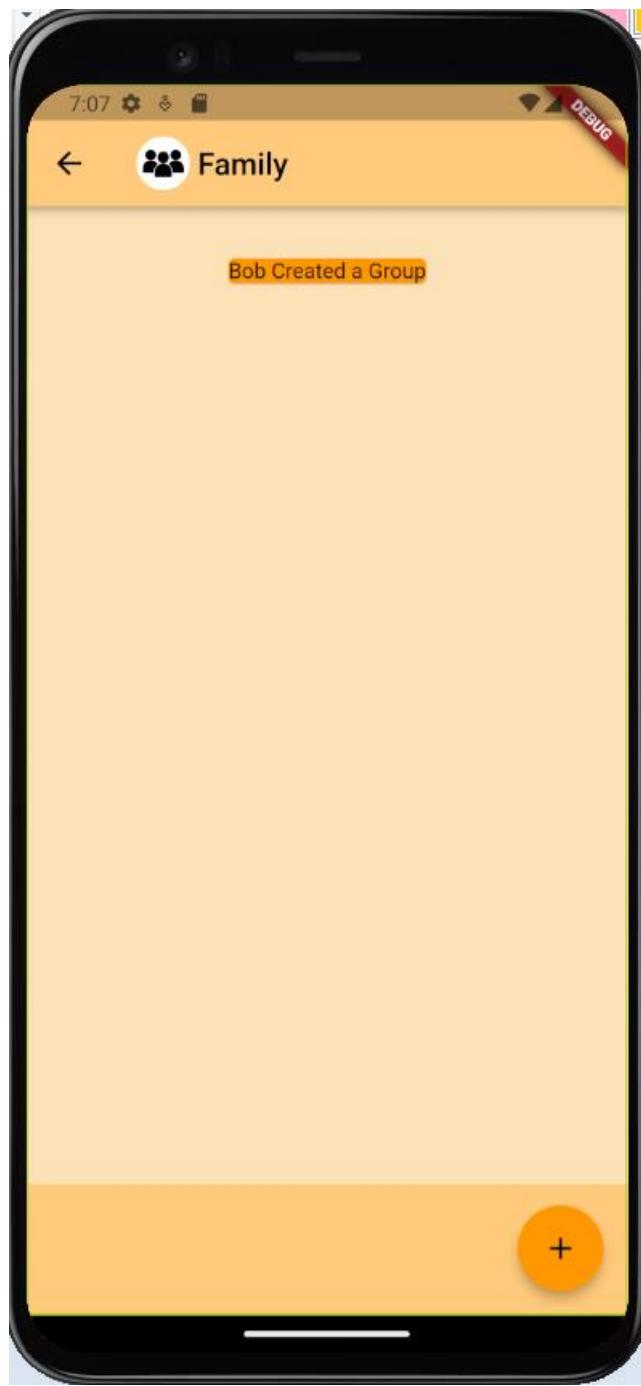
מסך הבית

- מסך בו ייצגו הקבוצות
- כפתור ליצירת קבוצה

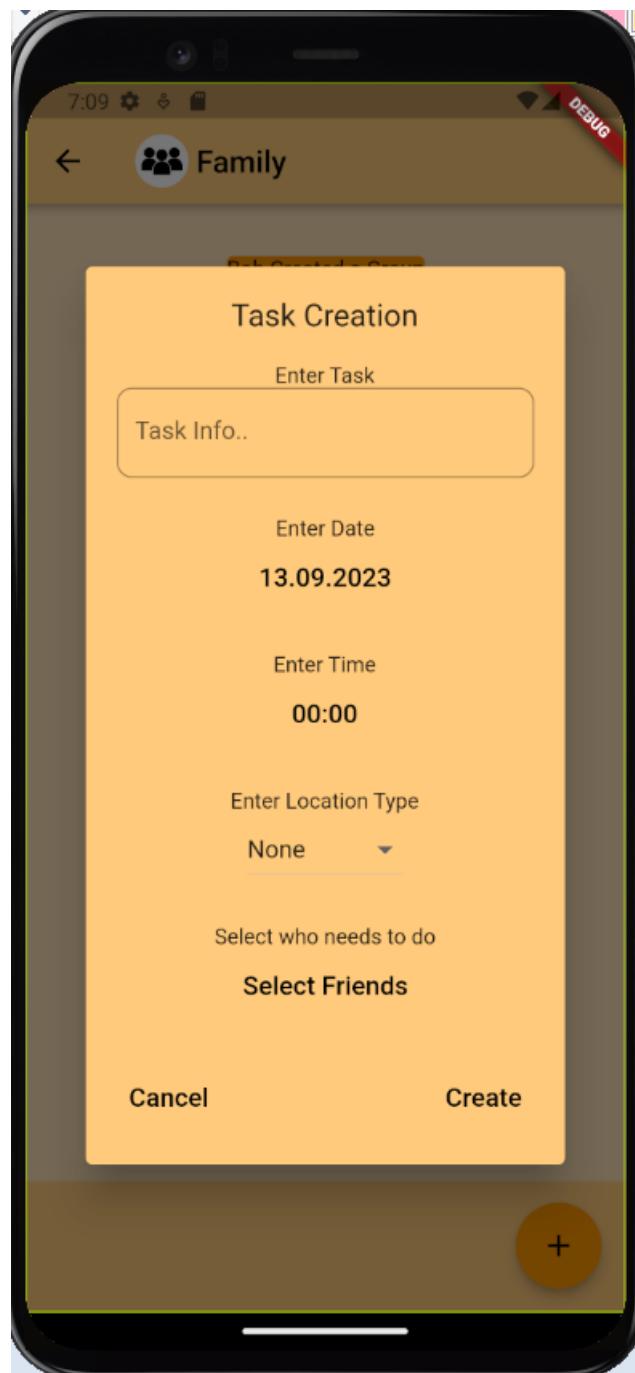


דיאלוג פתיחת קבוצה

- כפטור להוספה חברים לקבוצה
- בחירת סוג קבוצה (מצב עסק או לא)
- כפטור אישור וביטול

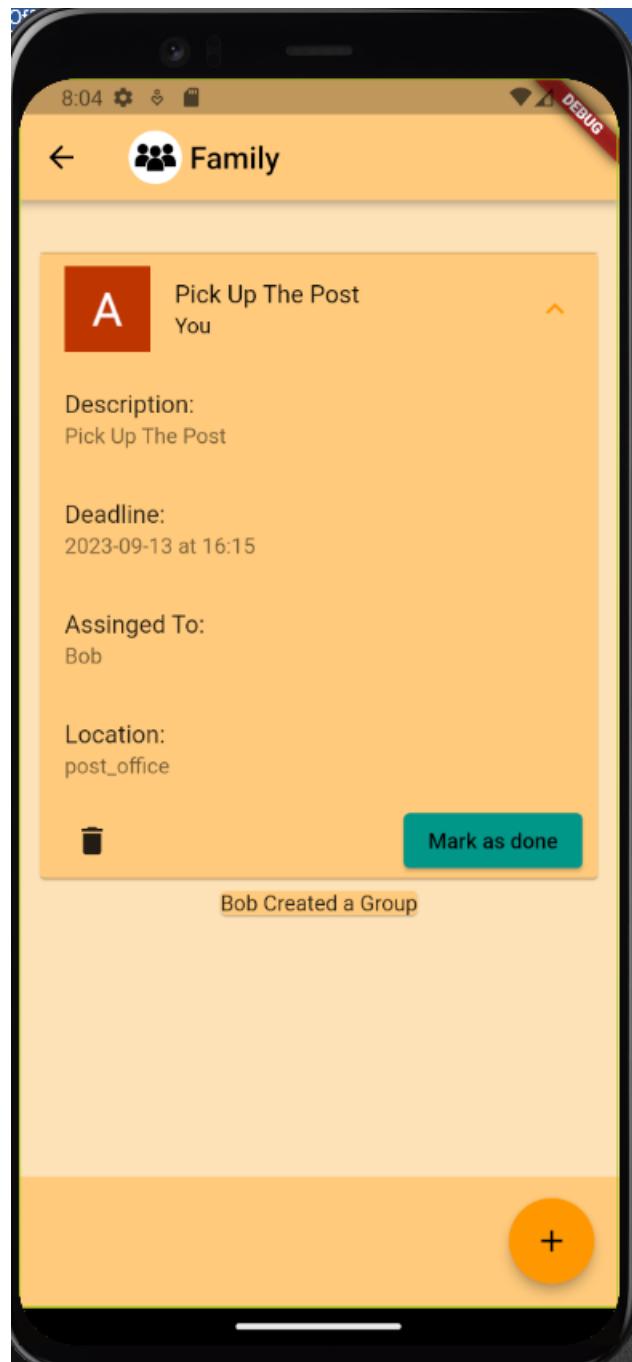


מסמך קבוצה
- כפתור ליצירת משימה



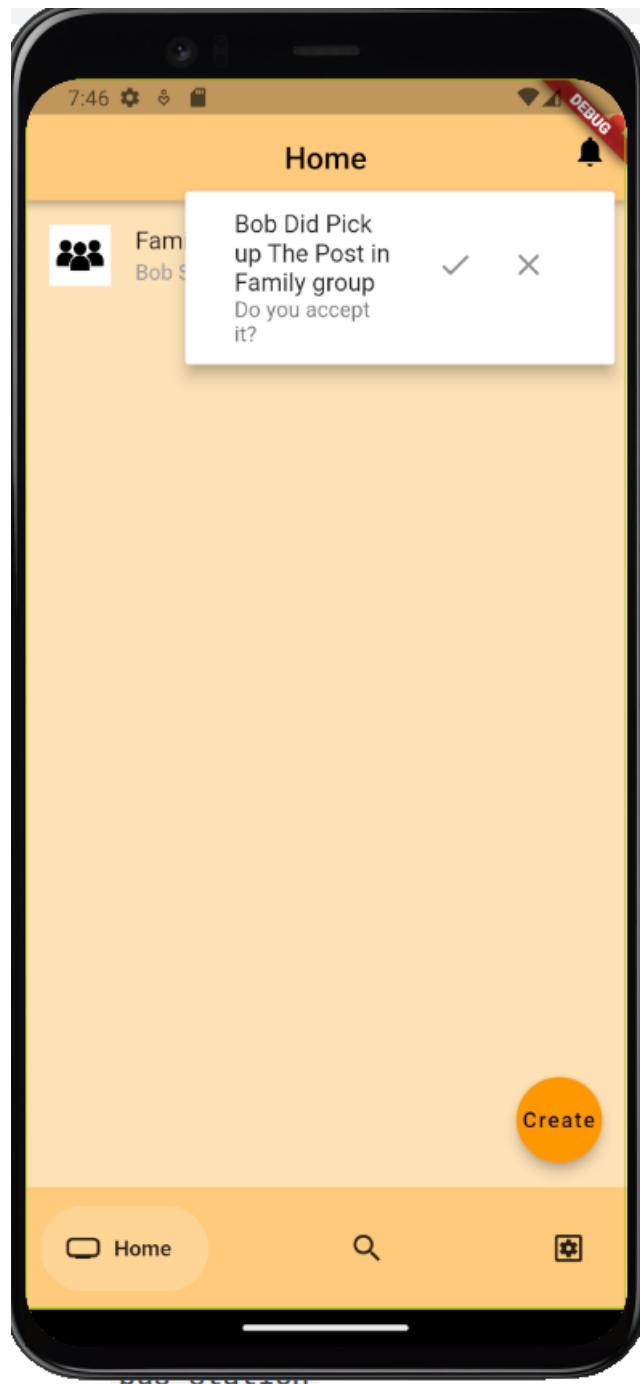
דיאלוג ייצירת משימה

- בחירת תאריך ושעה
- בחירת מקום
- הקדשת המשימה למשתמש/ים ספציפי/ים
- כפטור הקמה וביטול



משימה פתוחה

- תאור המשימה
- כפטור ביצוע המשימה
- ליצר המשימה אפשרות מחיקת משימה



תפריט התראות לאישור ביצוע המשימה

- שימושה מתבצעת מתעדכן התפריט והמשימה ממתינה לאישור או סירוב של יוצר המשימה
- בכך שתתבצע באופן מוחלט

תרשים מחלקות קיד

Data

groupModel

```
// This class represents the data model for a group.

import 'package:cloud_firestore/cloud_firestore.dart';

class GroupModel {
    // Business mode associated with the group.
    String businessMode;

    // UID of the creator of the group.
    String creatorUid;

    // Timestamp indicating when the group was created.
    Timestamp date;

    // Map storing points associated with friends in the group.
    Map<String, dynamic> friendsPoints;

    // Information about the user who created the last task.
    String lastTaskCreator;

    // Count of tasks associated with the group.
    int taskCount;

    // Name assigned to the group.
    String groupName;

    // URL to the group's image.
    String image;

    // Unique identifier for the group.
    String uid;

    // Timestamp indicating when the last task was created in the group.
    Timestamp taskCreationDate;

    // Constructor to initialize a GroupModel instance with given properties.
    GroupModel({
```

```
required this.businessMode,
required this.creatorUid,
required this.date,
required this.friendsPoints,
required this.groupName,
required this.image,
required this.uid,
required this.lastTaskCreator,
required this.taskCreationDate,
required this.taskCount,
});

// Factory constructor that facilitates the creation of a GroupModel object
from a map.
factory GroupModel.fromJson(Map<String, dynamic> json) {
return GroupModel(
  businessMode: json["businessMode"],
  creatorUid: json["creatorUid"],
  date: json["date"],
  friendsPoints: json["friendsPoints"],
  groupName: json["groupName"],
  image: json["image"],
  uid: json["uid"],
  lastTaskCreator: json["lastTaskCreator"],
  taskCreationDate: json["taskCreationDate"],
  taskCount: json["taskCount"],
);
}
}
```

NotificationModel

```
// This class defines the structure of a notification item for the application.

import 'package:cloud_firestore/cloud_firestore.dart';

class NotificationItem {
    final String uid;
    final String title;
    final String content;
    final bool done;
    final Timestamp date;
    final String groupUid;
    final String taskUid;
    final String whoDid;

    // Constructor for the NotificationItem class.
    NotificationItem(this.uid, this.title, this.content, this.done, this.date,
        this.groupUid, this.taskUid, this.whoDid);
}
```

taskModel

```
import 'package:cloud_firestore/cloud_firestore.dart';

// The TaskModel class represents a task's structure.
class TaskModel {
  String creatorUid;
  Timestamp deadline;
  String location;

  String status;
  String taskInfo;
  String deadlineTime;
  String assignedTo;
  String uid;
  String isTask;
  String whoDid;
  List<dynamic> assignedToUid;
  int taskIndex;

  // Constructs a TaskModel instance with all required fields.
  TaskModel({
    required this.creatorUid,
    required this.deadline,
    required this.location,
    required this.status,
    required this.taskInfo,
    required this.deadlineTime,
    required this.assignedTo,
    required this.uid,
    required this.isTask,
    required this.whoDid,
    required this.assignedToUid,
    required this.taskIndex,
  });
}
```

```

// Creates a TaskModel instance from a JSON map.
factory TaskModel.fromJson(Map<String, dynamic> json) {
    return TaskModel(
        creatorUid: json["creatorUid"],
        deadline: json["deadline"],
        location: json["location"],
        status: json["status"],
        taskInfo: json["taskInfo"],
        deadlineTime: json["deadlineTime"],
        assignedTo: json["assignedTo"],
        uid: json["uid"],
        isTask: json["isTask"],
        whoDid: json["whoDid"],
        assignedToUid: json["assignedToUid"],
        taskIndex: json["taskIndex"],
    );
}

// Creates a TaskModel instance from a Firestore document snapshot.
factory TaskModel.fromSnapshot(DocumentSnapshot snapshot) {
    final data = snapshot.data() as Map<String, dynamic>

    // Extracting fields from the snapshot's data map
    final creatorUid = data['creatorUid'];
    final deadline = data['deadline'];
    final location = data['location'];
    final status = data['status'];
    final taskInfo = data['taskInfo'];
    final deadlineTime = data['deadlineTime'];
    final assignedTo = data['assignedTo'];
    final uid = data['uid'];
    final isTask = data['isTask'];
    final whoDid = data['whoDid'];
    final assignedToUid = data['assignedToUid'];
    final taskIndex = data['taskIndex'];
}

```

```
// Constructing and returning a TaskModel instance
    return TaskModel(
        creatorUid: creatorUid,
        deadline: deadline,
        location: location,
        status: status,
        taskInfo: taskInfo,
        deadlineTime: deadlineTime,
        assignedTo: assignedTo,
        uid: uid,
        isTask: isTask,
        whoDid: whoDid,
        assignedToUid: assignedToUid,
        taskIndex: taskIndex);
}
}
```

userModel

```
import 'package:cloud_firestore/cloud_firestore.dart';

// The UserModel class defines the structure for user data.

class UserModel {
    String email;
    String name;
    String image;
    Timestamp date;
    String uid;
    String token;

    // Constructor for the UserModel class
    UserModel({
        required this.email,
        required this.name,
        required this.image,
        required this.date,
        required this.uid,
        required this.token,
    });

    // Factory method to create a UserModel instance from a Firestore document
    // snapshot
    factory UserModel.fromJson(DocumentSnapshot snapshot) {
        return UserModel(
            email: snapshot["email"],
            name: snapshot["name"],
            image: snapshot["image"],
            date: snapshot["date"],
            uid: snapshot["uid"],
            token: snapshot["token"],
        );
    }
}
```

Controllers

checkListController

```
import 'package:cloud_firestore/cloud_firestore.dart';

// This class controls the interactions related to the checkList data between the
// app and the database.
// It provides methods to fetch, update, delete, and set checkList items in
// Firestore.
class CheckListController {
    final FirebaseFirestore _firestore = FirebaseFirestore.instance;

    // Fetches a stream of checklist data for a given user.
    Stream<QuerySnapshot> getCheckListStream(String userId) {
        // Fetch the stream of documents from the checkList collection of the
        // specified user.
        return _firestore
            .collection("users")
            .doc(userId)
            .collection("checkList")
            .snapshots();
    }

    // Updates the 'done' status of a given checkList document.
    Future<void> updateDone(
        bool flag, String userId, String checkListId) async {
        await _firestore
            .collection("users")
            .doc(userId)
            .collection("checkList")
            .doc(checkListId)
            .update({
                "done": flag,
            });
    }
}
```

```

// Deletes a specific checkList document by its UID.
Future<void> deleteCheckList(String userUid, String checkListUid) async {
  await _firestore
    .collection("users")
    .doc(userUid)
    .collection("checkList")
    .doc(checkListUid)
    .delete();
}

//Creates a new checkList document in Firestore.
Future<void> setCheckListNotification(
  String creatorUid,
  String whoDid,
  String whoDidName,
  String taskInfo,
  String groupUid,
  String groupName,
  String taskUid,
) async {
  final docRef = _firestore
    .collection("users")
    .doc(creatorUid)
    .collection("checkList")
    .doc();
  docRef.set({
    "uid": docRef.id,
    "whoDid": whoDid,
    "whoDidName": whoDidName,
    "taskInfo": taskInfo,
    "groupUid": groupUid,
    "groupName": groupName,
    "done": false,
    "taskUid": taskUid,
    "date": DateTime.now(),
  });
}
}

```

groupController

```
import 'package:cloud_firestore/cloud_firestore.dart';

// This class manages interactions with the groups data between the app and
// Firestore.
// It provides methods to fetch, set, update, and delete group-related data.
class GroupController {
    final FirebaseFirestore _firestore = FirebaseFirestore.instance;

    // Fetches a map of friend points for a specific group.
    Future<Map<String, dynamic>> getFriendsList(String groupUid) async {
        await Future.delayed(const Duration(seconds: 0));
        return _firestore
            .collection("groups")
            .doc(groupUid)
            .get()
            .then((doc) => doc.data()?[ "friendsPoints"]);
    }

    // Retrieves a list of task UIDs for a specific group.
    Future<List<String>> fetchTaskUids(String groupUid) async {
        List<String> taskUids = [];
        QuerySnapshot<Object?> querySnapshot = await FirebaseFirestore.instance
            .collection('groups')
            .doc(groupUid)
            .collection('tasks')
            .get();

        for (var doc in querySnapshot.docs) {
            taskUids.add(doc.id);
        }
        return taskUids;
    }

    // Fetches a stream that tracks the count of tasks for a specific group.
    Stream<int> getTaskCountStream(String groupUid) {
        return _firestore
            .collection("groups")
            .doc(groupUid)
            .snapshots()
            .map((doc) => doc.data()?[ "taskCount"] ?? 0);
    }

    // Gets the count of tasks for a specific group.
    Future<int> getTaskCount(String groupUid) async {
        await Future.delayed(Duration(seconds: 0));
    }
}
```

```

    return _firestore
        .collection("groups")
        .doc(groupUid)
        .get()
        .then((doc) => doc.data()?["taskCount"]);
}

// Retrieves a document snapshot for a group using its UID.
Future<DocumentSnapshot<Map<String, dynamic>>> getGroupByUid(
    String groupUid) async {
    return await _firestore.collection('groups').doc(groupUid).get();
}

// Removes a user's group reference based on the provided UIDs.
Future<void> deleteGroupRef(String groupUser, String groupUid) async {
    await _firestore
        .collection("users")
        .doc(groupUser)
        .collection("groups")
        .doc(groupUid)
        .delete();
}

// Deletes a group document using its UID.
Future<void> deleteGroup(String groupUid) async {
    await _firestore.collection("groups").doc(groupUid).delete();
}

// Updates the friend points of a specific group using a map of friend list
data.
Future<void> updateFriendsPoints(
    String groupUid, Map<String, dynamic> friendsListData) async {
    await _firestore.collection("groups").doc(groupUid).update({
        "friendsPoints": friendsListData,
    });
}

```

```

// Retrieves a stream of user groups, ordered by task creation date.
Stream<QuerySnapshot> getUserGroupsOrderedByCreationDate(String userId) {
    return _firestore
        .collection("users")
        .doc(userId)
        .collection("groups")
        .orderBy("taskCreationDate", descending: true)
        .snapshots();
}

// Updates the image URL of a specific group.
Future<void> updateImage(String groupUid, String downloadUrl) async {
    await _firestore.collection("groups").doc(groupUid).update({
        "image": downloadUrl,
    });
}

// Modifies the business mode of a specific group.
Future<void> updateBusinessMode(String groupUid, String mode) async {
    await _firestore.collection('groups').doc(groupUid).update({
        'businessMode': mode,
    });
}

// Creates a new group document and returns its UID.
Future<String> setGroup(
    DocumentReference<Map<String, dynamic>> docRef,
    String groupName,
    String creatorUid,
    Map<String, dynamic> friendsPoints,
    String image,
    String mode,
    String userName,
) async {
    docRef.set({
        "groupName": groupName,
        "creatorUid": creatorUid,
        "friendsPoints": friendsPoints,
        "image": "https://t4.ftcdn.net/jpg/03/78/40/51/240_F_378405187_PyVLw51NVo3KltNlhUOpKfULdkUOUn7j.jpg",
        "date": DateTime.now(),
        "uid": docRef.id,
        "businessMode": mode,
        "lastTaskCreator": userName + " Created a Group",
        "taskCreationDate": DateTime.now(),
        "taskCount": 0,
    });
}

```

```

        return docRef.id;
    }

    // Adds a new group reference document to a user's collection of groups.
    Future<void> setGroupRef(String friendUid, String groupUid, int num) async {
        await _firestore
            .collection("users")
            .doc(friendUid)
            .collection("groups")
            .doc(groupUid)
            .set({
                "groupRef": _firestore.collection("groups").doc(groupUid),
                "taskCreationDate": DateTime.now(),
                "countSeen": num
            });
    }

    // Fetches a stream that tracks the friend points for a user within a specific
    group.
    Stream<int> getFriendPointsStream(String userUid, String groupUid) {
        return FirebaseFirestore.instance
            .collection("users")
            .doc(userUid)
            .collection('groups')
            .doc(groupUid)
            .snapshots()
            .map((document) => document.data()?["friendsPoints"] ?? 0);
    }

    // Retrieves the image URL of a specific group.
    Future<String> getImageUrl(String groupUid) async {
        await Future.delayed(const Duration(seconds: 0));
        return _firestore
            .collection("groups")
            .doc(groupUid)
            .get()
            .then((doc) => doc.data()?["image"]);
    }

    // Fetches the name of a specific group.
    Future<String> getGroupName(String groupUid) async {
        return _firestore
            .collection("groups")
            .doc(groupUid)
            .get()

```

```

        .then((doc) => doc.data()?["groupName"]);
    }

    // Updates the name of a specific group.
    Future<void> updateGroupName(String groupUid, String newGroupName) async {
        await _firestore
            .collection("groups")
            .doc(groupUid)
            .update({"groupName": newGroupName});
    }

    // Retrieves the last task creator's info for a specific group.
    Future<String> getGroupTaskInfo(String groupUid) async {
        await Future.delayed(const Duration(seconds: 0));
        return _firestore
            .collection("groups")
            .doc(groupUid)
            .get()
            .then((doc) => doc.data()?["lastTaskCreator"]);
    }

    // Gets the timestamp of when the last task was created for a specific group.
    Future<Timestamp> getGroupTaskCreationTime(String groupUid) async {
        await Future.delayed(Duration(seconds: 0));
        return _firestore
            .collection("groups")
            .doc(groupUid)
            .get()
            .then((doc) => doc.data()?["taskCreationDate"]);
    }

    // Provides a stream of a group's document updates using its UID.
    Stream<DocumentSnapshot> getGroupStream(String groupUid) {
        return _firestore.collection("groups").doc(groupUid).snapshots();
    }

    // Updates task creation metadata for a specific group.
    Future<void> taskCreationUpdate(
        String groupUid, int taskCount, String userName) async {
        await _firestore.collection("groups").doc(groupUid).update({
            "taskCount": taskCount + 1,
            "lastTaskCreator": userName + " Set A Task",
            "taskCreationDate": DateTime.now(),
        });
    }
}

```

```
// Updates the task creation date for a specific group.  
Future<void> updateCreationDate(String userUid, String groupUid) async {  
    await _firestore  
        .collection("users")  
        .doc(userUid)  
        .collection("groups")  
        .doc(groupUid)  
        .update({  
            "taskCreationDate": DateTime.now(),  
        });  
}  
}  
}
```

notifiactionController

```
// This class serves as a controller for sending notifications through Firebase
Cloud Functions.

import 'package:cloud_functions/cloud_functions.dart';

class NotificationController {
    // Reference to the Firebase Cloud Function sendFCMNotification.
    final HttpsCallable callable =
        FirebaseFunctions.instance.httpsCallable('sendFCMNotification');

    // Sends a notification to a user when they are close to a specified location.
    void sendCloseToLocationNotification(String token, String locationName) {
        callable.call(<String, dynamic>{
            'toToken': token,
            'title': '$locationName is close to you',
            'body': 'You have a task that close to you',
        }).then((response) {
            print(response.data);
        }).catchError((error) {
            print("Error calling the function: $error");
        });
    }

    // Sends a notification when a new task is created in a group.
    void sendCreatedTaskNotification(
        String token, String creatorName, String _taskInfo, String groupName) {
        callable.call(<String, dynamic>{
            'toToken': token,
            'title': '$creatorName Set new Task',
            'body': '$creatorName set ${_taskInfo} in $groupName group',
        }).then((response) {
            print(response.data);
        }).catchError((error) {
            print("Error calling the function: $error");
        });
    }
}
```

taskController

```
import 'package:cloud_firestore/cloud_firestore.dart';

// This class manages interactions with the tasks data between the app and
Firestore.

class TaskController {
    final FirebaseFirestore _firestore = FirebaseFirestore.instance;

    // Updates the status of a specific task within a group.
    Future<void> updateTaskStatus(
        String groupUid, String taskUid, String status) async {
    _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .doc(taskUid)
        .update({"status": status});
}

// Records an activity indicating a change in the group's image.
Future<void> setChangeGroupImageTask(String groupUid, String userName) async {
    _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .doc()
        .set({
    "lastTaskCreator": "$userName Changed the Group Image",
    "isTask": "false",
    "dateCreated": DateTime.now(),
});
}

// Records an activity indicating the addition of a user to the group.
Future<void> setAddToTheGroupTask(String groupUid, String userName) async {
    _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .doc()
        .set({
    "lastTaskCreator": "$userName Added to the Group",
    "isTask": "false",
    "dateCreated": DateTime.now(),
}
```

```

    });
}

// Logs a task when the group name changes.
Future<void> setChangeGroupNameTask(String groupUid, String userName) async {
    _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .doc()
        .set({
            "lastTaskCreator": userName + " changed the Group Name",
            "isTask": "false",
            "dateCreated": DateTime.now(),
        });
}

// Logs a task when the group mode changes.
Future<void> setChangeGroupModeTask(String groupUid, String userName) async {
    _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .doc()
        .set({
            "lastTaskCreator": userName + " changed the Group Mode",
            "isTask": "false",
            "dateCreated": DateTime.now(),
        });
}

// Logs a task when a group is created.
Future<void> setCreateGroupTask(String groupUid, String userName) async {
    await _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .doc()
        .set({
            "lastTaskCreator": userName + " Created a Group",
            "isTask": "false",
            "dateCreated": DateTime.now(),
        });
}

// Logs a task when someone is removed from a group.
Future<void> setRemovedFromGroupTask(
    String groupUid, String friendName) async {

```

```

_firestore
    .collection("groups")
    .doc(groupUid)
    .collection("tasks")
    .doc()
    .set({
        "lastTaskCreator": friendName + " Removed From the Group",
        "isTask": "false",
        "dateCreated": DateTime.now(),
    });
}

// Updates a specific task's status to 'pending'.
Future<void> updateToPending(
    String groupUid,
    String taskUid,
    String userUid,
) async {
    await _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .doc(taskUid)
        .update({
            "status": "TaskStatus.pending",
            "whoDid": userUid,
        });
}

// Fetches a specific task document.
Future<DocumentSnapshot> fetchTaskDocument(
    String groupUid, String taskUid) async {
    return await _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .doc(taskUid)
        .get();
}

// Logs a task when someone quits a group.
Future<void> setQuitGroupTask(String groupUid, String userName) async {
    await _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .doc()
        .set({

```

```

        "lastTaskCreator": userName + " Quited From the Group",
        "isTask": "false",
        "dateCreated": DateTime.now(),
    });
}

// Fetches tasks associated with a specific group.
Stream<QuerySnapshot> getUserTasks(String groupUid) {
    return _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .snapshots();
}

// Fetches tasks associated with a specific group and orders them by creation
// date.
Stream<QuerySnapshot> getTasksOrderedByCreationDate(String groupUid) {
    return _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .orderBy("dateCreated", descending: true)
        .snapshots();
}

// Deletes a specific task within a group.
Future<void> deleteTask(String groupUid, String taskUid) async {
    _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .doc(taskUid)
        .delete();
}

// Fetches the number of tasks within a group.
Future<int> getTaskCount(String groupUid) async {
    return _firestore
        .collection("groups")
        .doc(groupUid)
        .get()
        .then((doc) => doc.data()?["taskCount"]);
}

// Fetches the location of tasks within a group.
Future<String> getTaskLocation(String groupUid, String taskUid) async {

```

```

    await Future.delayed(
        Duration(seconds: 0)); // Simulating an asynchronous operation
    Future<String> location = _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .doc(taskUid)
        .get()
        .then((doc) => doc.data()?["location"]);

    return location;
}

// Fetches the isTask of tasks within a group.
Future<String> isTask(String groupUid, String taskUid) async {
    await Future.delayed(Duration(seconds: 0));
    Future<String> isTask = _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .doc(taskUid)
        .get()
        .then((doc) => doc.data()?["isTask"]);

    return isTask;
}

// Fetches the status of tasks within a group.
Future<String> getStatus(String groupUid, String taskUid) async {
    await Future.delayed(Duration(seconds: 0));
    Future<String> status = _firestore
        .collection("groups")
        .doc(groupUid)
        .collection("tasks")
        .doc(taskUid)
        .get()
        .then((doc) => doc.data()?["status"]);

    return status;
}

// Fetches List of Uid of the users assigned to tasks within a group.
Future<List<dynamic>> getAssignedToUid(
    String groupUid, String taskUid) async {
    await Future.delayed(Duration(seconds: 0));
    Future<List<dynamic>> assignedToUid = _firestore
        .collection("groups")
        .doc(groupUid)

```

```
.collection("tasks")
.doc(taskUid)
.get()
.then((doc) => doc.data()["assignedToUid"]);

return assignedToUid;
}
```

```
// Adds a new task with provided details in a group.
Future<void> setTask(
    String groupUid,
    String creatorUid,
    String taskInfo,
    Set selectedFriends,
    String location,
    DateTime deadline,
    String deadlineString,
    String status,
    int taskCount,
    List<dynamic> selectedFriendsUid) async {
final docRef =
    _firestore.collection("groups").doc(groupUid).collection("tasks").doc();
docRef.set({
    "uid": docRef.id,
    "creatorUid": creatorUid,
    "assignedTo": selectedFriends.join(", "),
    "taskInfo": taskInfo,
    "location": location,
    "deadlineTime": deadlineString,
    "status": status,
    "deadline": deadline,
    "dateCreated": DateTime.now(),
    "isTask": "true",
    "whoDid": "",
    "assignedToUid": selectedFriendsUid,
    "taskIndex": taskCount + 1,
});
}
}
```

userController

```
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:firebase_messaging/firebase_messaging.dart';

// The `UserController` class provides methods to interact with Firebase
Firestore
// and Firebase Messaging services, specifically for user-related operations.
class UserController {
    final FirebaseFirestore _firestore = FirebaseFirestore.instance;
    final FirebaseMessaging _firebaseMessaging = FirebaseMessaging.instance;

    // Fetch and set the user token
    Future<void> setUser(User user) async {
        String? token = await _firebaseMessaging.getToken();

        await _firestore.collection("users").doc(user.uid).set({
            "email": user.email,
            "name": user.displayName,
            "image": user.photoURL,
            "uid": user.uid,
            "date": DateTime.now(),
            "token": token,
        });
    }

    // Fetch and set the user token
    Future<void> setUserBySignIn(User user, String name, String image) async {
        String? token = await _firebaseMessaging.getToken();

        await _firestore.collection("users").doc(user.uid).set({
            "email": user.email,
            "name": name,
            "image": image,
            "uid": user.uid,
            "date": DateTime.now(),
            "token": token,
        });
    }

    // Update the user's messaging token in Firestore
    Future<void> setToken(String userUid, String newToken) async {
        _firestore.collection("users").doc(userUid).update({'token': newToken});
    }
    // Delete a friend reference document for the current user
```

```

Future<void> deleteFriendDocReference(
    String currentUserId, String friendId) async {
    _firestore
        .collection("users")
        .doc(currentUserId)
        .collection("friends")
        .doc(friendId)
        .delete();
}
// Search and return users by email in Firestore

Future<QuerySnapshot<Map<String, dynamic>>> searchUsersByEmail(
    String searchText) async {
    return await _firestore
        .collection("users")
        .where("email", isGreaterThanOrEqualTo: searchText)
        .where("email", isLessThanOrEqualTo: searchText + '\uf8ff')
        .get();
}
// Retrieve a user's friends as a query snapshot from Firestore

Future<QuerySnapshot<Object?>> getFriendQuerySnapshot(
    String currentUserId) async {
    QuerySnapshot querySnapshot = await _firestore
        .collection("users")
        .doc(currentUserId)
        .collection("friends")
        .get();

    return querySnapshot;
}
// Retrieve the document snapshot of a user's friend from Firestore

Future<DocumentSnapshot<Object?>> getDocumentSnapshot(
    String userUid, String friendUid) async {
    return await _firestore
        .collection("users")
        .doc(userUid)
        .collection("friends")
        .doc(friendUid)
        .get();
}
// Set a friend reference for a user in Firestore

Future<void> setFriend(String userUid, String friendUid) async {
    await _firestore
        .collection("users")

```

```

    .doc(userUid)
    .collection("friends")
    .doc(friendUid)
    .set({"friendRef": _firebase.collection("users").doc(friendUid)}));
}
// Set a group reference for a user in Firestore

Future<void> setGroupRef(String userUid, String groupUid) async {
  await _firebase
    .collection("users")
    .doc(userUid)
    .collection("groups")
    .doc(groupUid)
    .set({
      "groupRef": _firebase.collection("groups").doc(groupUid),
      "taskCreationDate": DateTime.now(),
      "countSeen": 0,
    });
}
// Update the seen count of a group for a user in Firestore

Future<void> updateCountSeen(
  String userUid, String groupUid, int seen) async {
  await _firebase
    .collection("users")
    .doc(userUid)
    .collection("groups")
    .doc(groupUid)
    .update({
      "countSeen": seen,
    });
}
// Retrieve the seen count of a group for a user from Firestore

Future<int> getCountSeen(
  String userUid,
  String groupUid,
) async {
  await Future.delayed(
    Duration(seconds: 0)); // Simulating an asynchronous operation
  Future<int> countSeen = _firebase
    .collection("users")
    .doc(userUid)
    .collection("groups")
    .doc(groupUid)
    .get()
    .then((doc) => doc.data()?[ "countSeen"]);
}

```

```

        return countSeen;
    }
    // Retrieve the messaging token of a user from Firestore

    Future<String> getToken(
        String userId,
    ) async {
        await Future.delayed(
            Duration(seconds: 0)); // Simulating an asynchronous operation
        Future<String> token = _firestore
            .collection("users")
            .doc(userId)
            .get()
            .then((doc) => doc.data()?["token"]);

        return token;
    }
    // Fetch all group UIDs associated with a user from Firestore

    Future<List<String>> fetchGroupUids(String userId) async {
        List<String> groupUids = [];

        QuerySnapshot<Object?> querySnapshot = await FirebaseFirestore.instance
            .collection('users')
            .doc(userId)
            .collection('groups')
            .get();

        for (var doc in querySnapshot.docs) {
            groupUids.add(doc.id);
        }

        return groupUids;
    }
    // Stream the seen count of a group for a user from Firestore

    Stream<int> getCountSeenStream(String userId, String groupUid) {
        return _firestore
            .collection("users")
            .doc(userId)
            .collection("groups")
            .doc(groupUid)
            .snapshots() // This gives a Stream<DocumentSnapshot>
            .map((doc) =>
                doc.data()?["countSeen"] ?? 0); // Convert DocumentSnapshot to int
    }
    // Stream a user's document snapshot from Firestore

```

```

Stream<DocumentSnapshot> getUserStream(String friendUid) {
    return _firestore.collection("users").doc(friendUid).snapshots();
}
// Update a user's profile image URL in Firestore

Future<void> updateImage(String userId, String downloadUrl) async {
    await _firestore.collection("users").doc(userId).update({
        "image": downloadUrl,
    });
}
// Retrieve a user's profile image URL from Firestore

Future<String> getImageUrl(String userId) async {
    await Future.delayed(
        Duration(seconds: 0)); // Simulating an asynchronous operation
    Future<String> downloadUrlImg = _firestore
        .collection("users")
        .doc(userId)
        .get()
        .then((doc) => doc.data()?[ "image"]);
}

return downloadUrlImg;
}
// Retrieve a user's name from Firestore

Future<String> getUserName(String userId) async {
    Future<String> userName = _firestore
        .collection("users")
        .doc(userId)
        .get()
        .then((doc) => doc.data()?[ "name"]);
    return userName;
}
// Retrieve a user's email from Firestore

Future<String> getEmail(String userId) async {
    Future<String> email = _firestore
        .collection("users")
        .doc(userId)
        .get()
        .then((doc) => doc.data()?[ "email"]);
    return email;
}
// Update a user's email in Firestore

Future<void> updateEmail(String userId, String newEmail) async {
    await _firestore.collection("users").doc(userId).update({
        "email": newEmail,
    });
}

```

```
    });
}
// Update a user's name in Firestore

Future<void> updateUserName(String userUid, String newName) async {
    await _firestore.collection("users").doc(userUid).update({
        "name": newName,
    });
}
// Retrieve a user's document snapshot from Firestore

Future<DocumentSnapshot<Map<String, dynamic>>> getUserDocumentSnapshot(
    String userUid) async {
    DocumentSnapshot<Map<String, dynamic>> user =
        await _firestore.collection("users").doc(userUid).get();
    return user;
}
// Delete a group reference for a user from Firestore

Future<void> deleteGroup(String friendUid, String groupUid) async {
    await _firestore
        .collection("users")
        .doc(friendUid)
        .collection("groups")
        .doc(groupUid)
        .delete();
}
}
```

View

mainApp

```
import 'package:flutter/material.dart';
import 'package:firebase_core/firebase_core.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:task_manage_app/screens/auth_screen.dart';
import 'package:task_manage_app/models/user_model.dart';
import 'package:task_manage_app/screens/mainly_screen.dart';
import 'package:firebase_crashlytics/firebase_crashlytics.dart';

final GlobalKey<NavigatorState> navigatorKey = GlobalKey<NavigatorState>();

// This is the main function where the app starts its execution.
void main() async {
    // Ensure the flutter framework is properly initialized
    WidgetsFlutterBinding.ensureInitialized();

    // Initialize Firebase app
    await Firebase.initializeApp();

    // Enable Firebase Crashlytics
    await FirebaseCrashlytics.instance.setCrashlyticsCollectionEnabled(true);

    // Run the main application widget
    runApp(const MyApp());
}

// MyApp is the root widget of the application.
class MyApp extends StatelessWidget {
    const MyApp({super.key});

    // This function checks if a user is currently signed in or not.
    // if signed in, it fetches the user's data and prepares the MainlyScreen,
    // otherwise, it prepares the AuthScreen.
    Future<Widget> userSignedIn() async {
        User? user = FirebaseAuth.instance.currentUser;
        if (user != null) {
            DocumentSnapshot userData = await FirebaseFirestore.instance
                .collection("users")
                .doc(user.uid)
                .get();
            UserModel userModel = UserModel.fromJson(userData);
            return MainlyScreen(

```

```
        userModel,
        key: mainScreenKey,
    );
} else {
    return AuthScreen();
}
}

// Builds the main MaterialApp widget, wrapping around either the MainlyScreen
or the AuthScreen.
@Override
Widget build(BuildContext context) {
    return MaterialApp(
        title: 'Flutter Task Manager',
        theme: ThemeData(
            primarySwatch: Colors.orange,
        ),
        home: FutureBuilder(
            future: userSignedIn(),
            builder: (context, AsyncSnapshot<Widget> snapshot) {
                if (snapshot.hasData) {
                    return snapshot.data!;
                }
                return Scaffold(
                    body: Center(child: CircularProgressIndicator()),
                );
            },
        ),
    );
}
```

authScreen

```
import "package:cloud_firestore/cloud_firestore.dart";
import "package:firebase_auth/firebase_auth.dart";
import "package:flutter/material.dart";
import "package:google_sign_in/google_sign_in.dart";
import "package:task_manage_app/main.dart";
import "package:task_manage_app/models/user_controller.dart";
import "package:task_manage_app/screens/registration_screen.dart";
import "package:task_manage_app/widgets/reusable_widgets.dart";

// This screen handles user authentication, offering both email/password and
Google sign-in options.
class AuthScreen extends StatefulWidget {
  @override
  State<AuthScreen> createState() => _AuthScreenState();
}

class _AuthScreenState extends State<AuthScreen> {
  GoogleSignIn googleSignIn = GoogleSignIn();
  final TextEditingController _passwordTextController = TextEditingController();
  final TextEditingController _emailTextController = TextEditingController();
  String _pass = '';
  String _email = '';
  final _userController = UserController();

  @override
  void initState() {
    super.initState();
    // Setting up listeners for text changes in the email and password fields.
    _emailTextController.addListener(_onChange);
    _passwordTextController.addListener(_onChange);
  }

  @override
  void dispose() {
    // Cleaning up the text controllers to prevent memory leaks.
    _emailTextController.dispose();
    _passwordTextController.dispose();
    super.dispose();
  }

  // Updates the _pass and _email variables based on the user's text input.
  void _onChange() {
    setState(() {
      _pass = _passwordTextController.text;
      _email = _emailTextController.text;
    });
  }
}
```

```

    });
}
// Handles password reset functionality for users who've forgotten their
password.

Future<void> resetPassword(String email) async {
try {
  await FirebaseAuth.instance.sendPasswordResetEmail(email: email);
  // if email exists sends email for changing password
  ScaffoldMessenger.of(context).showSnackBar(
    const SnackBar(
      content: Text('Email Sent'),
      duration: Duration(seconds: 2),
    ),
  );
  dispose();
} on FirebaseAuthException catch (e) {
  // if email not found will pop a message
  if (e.code == 'user-not-found') {
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(
        content: Text('Email Not Found'),
        duration: Duration(seconds: 2),
      ),
    );
  }
} catch (e) {
  // if something else will pop message
  ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(
      content: Text(e.toString()),
      duration: const Duration(seconds: 2),
    ),
  );
}
}
// Allows users to sign in using their email and password.

Future<void> signInWithEmailAndPassFunction() async {
try {
  // firebase sign in function
  await FirebaseAuth.instance.signInWithEmailAndPassword(
    email: _emailTextController.text,
    password: _passwordTextController.text);
  User? user = FirebaseAuth.instance.currentUser;
  if (user != null) {
    if (user.emailVerified) {
      // will navigate to the main app
    }
  }
}
}

```

```

        Navigator.pushAndRemoveUntil(
            context,
            MaterialPageRoute(builder: (context) => MyApp()),
            (route) => false);
    } else {
        // will show dialog for verify the email
        showDialog(
            context: context,
            builder: (context) =>
                reusable_AlertDialog("Verify Email First", context));
    }
}
} catch (error, stackTrace) {
    // when wrong password or email will display it
    showDialog(
        context: context,
        builder: (context) =>
            reusable_AlertDialog("Wrong Email or password", context));
}
}
// Handles user sign-in functionality using Google account.

Future signInFunction() async {
    try {
        GoogleSignInAccount? googleUser = await googleSignIn.signIn();
        if (googleUser == null) {
            // if exit
            return null;
        }
        final googleAuth = await googleUser
            .authentication; // creates authentication user on firebase
        final credential = GoogleAuthProvider.credential(
            accessToken: googleAuth.accessToken, idToken: googleAuth.idToken);
        UserCredential userCredential =
            await FirebaseAuth.instance.signInWithCredential(credential);
        DocumentSnapshot userExist =
            await _userController.getUserDocumentSnapshot(userCredential
                .user!.uid); // try to get the user info from firebase

        if (userExist.exists) {
            print("User already exists in Database");
        } else {
            // if not exists so it set one
            _userController.setUser(userCredential.user!);
        }
        // then will navigate to the main app
        Navigator.pushAndRemoveUntil(context,
            MaterialPageRoute(builder: (context) => MyApp()), (route) => false);
    }
}

```

```

} catch (error, stackTrace) {
    print("${error.toString()} ${stackTrace.toString()}");
}
}

final sizedBoxHight = SizedBox(
    height: 20,
);
@Override
Widget build(BuildContext context) {
    return Scaffold(
        // Setting the background color of the authentication screen.
        backgroundColor: Color.fromARGB(255, 253, 225, 183),
        body: SingleChildScrollView(
            child: Column(
                children: <Widget>[
                    // Spacing elements to structure the layout.
                    sizedBoxHight,
                    sizedBoxHight,
                    sizedBoxHight,
                    // Displaying the app's logo.
                    Image.asset(
                        "assets/images/taskmanagerlogo.png",
                        width: 190,
                    ),
                    sizedBoxHight,
                    Padding(
                        padding:
                            const EdgeInsets.symmetric(vertical: 0, horizontal: 20),
                        child: Column(children: [
                            // Text input field for email.
                            reusableTextField("Enter Email", Icons.person_outlined, false,
                                _emailTextController),
                            sizedBoxHight,
                            // Text input field for password.
                            reusableTextField("Enter Password", Icons.lock_outline, true,
                                _passwordTextController),
                            // Button to trigger password reset.
                            Row(
                                children: [
                                    TextButton(
                                        onPressed: () {
                                            showDialog(
                                                context: context,
                                                builder: (context) => resetPasswordDialog(
                                                    resetPassword,
                                                    context,
                                                    _emailTextController));
                                        }
                                    )
                                ],
                            ),
                        ],
                    ),
                ],
            ),
        ),
    );
}

```

```

        },
        child: const Text('Forgot Password..',
            style: TextStyle(
                color: Color.fromARGB(255, 207, 124, 0))),
        ),
    ],
),
// Sign-in button for email/password authentication.
ElevatedButton(
    onPressed: () async {
        await signInWithEmailAndPassFunction();
    },
    child: Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
            Text(
                "Sign In",
                style: TextStyle(fontSize: 20, color: Colors.white),
            )
        ],
    ),
    style: ButtonStyle(
        backgroundColor:
            MaterialStateProperty.all(Colors.black),
        padding: MaterialStateProperty.all(
            EdgeInsets.symmetric(vertical: 20)),
    ),
    sizedBoxHight,
// Sign-in button for Google authentication.
ElevatedButton(
    onPressed: () async {
        await signInFunction();
    },
    child: Row(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
            Text(
                "Sign In With Google",
                style: TextStyle(fontSize: 20, color: Colors.white),
            ),
            SizedBox(
                width: 10,
            ),
            Image.asset(
                "assets/images/googleLogo.png",
                height: 36,
            )
        ],
    ),

```

```
        ),
        style: ButtonStyle(
            backgroundColor:
                MaterialStateProperty.all(Colors.black),
            padding: MaterialStateProperty.all(
                EdgeInsets.symmetric(vertical: 20))),
        ),
        sizedBoxHight,
        // Button to navigate to the sign-up screen.
        reusable_button("Sign Up", () {
            Navigator.push(
                context,
                MaterialPageRoute(
                    builder: (context) => RegistrationScreen())));
        },
        sizedBoxHight,
    )));
],
),
);
);
;
);
}
}
```

registrationScreen

```
// Import necessary libraries and packages.  
import "package:firebase_messaging/firebase_messaging.dart";  
import 'package:flutter/material.dart';  
import "package:cloud_firestore/cloud_firestore.dart";  
import "package:firebase_auth/firebase_auth.dart";  
import "package:google_sign_in/google_sign_in.dart";  
import "package:task_manage_app/models/user_controller.dart";  
import "package:task_manage_app/screens/auth_screen.dart";  
import "package:task_manage_app/widgets/reusable_widgets.dart";  
  
// Define a StatefulWidget for the RegistrationScreen.  
class RegistrationScreen extends StatefulWidget {  
    @override  
    State<RegistrationScreen> createState() => _RegistrationScreenState();  
}  
  
// Define the state for the RegistrationScreen widget.  
class _RegistrationScreenState extends State<RegistrationScreen> {  
    // Create instances of necessary classes and controllers.  
    GoogleSignIn googleSignIn = GoogleSignIn();  
    FirebaseFirestore firestore = FirebaseFirestore.instance;  
    TextEditingController _emailTextController = TextEditingController();  
    TextEditingController _nameTextController = TextEditingController();  
    TextEditingController _passwordTextController = TextEditingController();  
    TextEditingController _verifyEmailTextController = TextEditingController();  
    TextEditingController _verifyPassTextController = TextEditingController();  
  
    // Initialize variables for user input.  
    String _pass = "";  
    String _email = "";  
    String _verify_pass = "";  
    String _vetify_email = "";  
    String _name = "";  
    String? _token;  
  
    // Create a UserController instance to manage user data.  
    UserController _userController = UserController();  
  
    // Initialize the widget state.  
    void initState() {  
        super.initState();  
  
        // Add listeners to user input controllers.  
        _emailTextController.addListener(_onChange);  
        _passwordTextController.addListener(_onChange);
```

```

        _nameTextController.addListener(_onChange);
        _verifyEmailTextController.addListener(_onChange);
        _verifyPassTextController.addListener(_onChange);
    }

    // Dispose of controllers when the widget is disposed.
    void disposeControllers() {
        super.dispose();
    }

    // Update state variables when the controller values change.
    void _onChange() {
        setState(() {
            _pass = _passwordTextController.text;
            _email = _emailTextController.text;
            _verify_pass = _verifyPassTextController.text;
            _vetify_email = _verifyEmailTextController.text;
            _name = _nameTextController.text;
        });
    }

    // Function to handle user registration.
    Future<void> signInFunction() async {
        try {
            bool flag_email = false;
            bool flag_pass = false;

            // Check if any of the input fields are empty.
            if (_pass == "" ||
                _email == "" ||
                _verify_pass == "" ||
                _vetify_email == "" ||
                _name == "") {
                showDialog(
                    context: context,
                    builder: (context) => reusable_AlertDialog(
                        "One Or Few Of The Fields Are Empty", context));
            }
        }

        // Check if email and password verification match.
        if (_vetify_email != _email) {
            flag_email = true;
        }
        if (_verify_pass != _pass) {
            flag_pass = true;
        }
        // Display an error message if email or password verification fails.
    }
}

```

```

if (flag_pass || flag_email) {
    String str = "";
    str = flag_pass ? "Incorrect Password" : "Incorrect Email";
    showDialog(
        context: context,
        builder: (context) => reusable_AlertDialog(str, context));
    return;
} else {
    // Create a user with email and password.
    await FirebaseAuth.instance
        .createUserWithEmailAndPassword(email: _email, password: _pass);
    User? user = await FirebaseAuth.instance.currentUser;
    if (user == null) {
        return;
    }
    // Check if the user already exists in the database.
    DocumentSnapshot userExist =
        await firestore.collection("users").doc(user.uid).get();
    if (userExist.exists) {
        print("User already exists in Database");
    } else {
        // Get and set the FCM token.
        FirebaseMessaging.instance.getToken().then((String? token) {
            assert(token != null);
            print('FCM Token: $token');

            setState(() {
                _token = token;
            });
        });
    }

    // Set the user data using the UserController.
    _userController.setUserBySignIn(
        user,
        _name,
        "https://www.freepnglogos.com/uploads/rodan-and-fields-png-
logo/rodan--fields-icons-png-logo-34.png",
    );
    // Send email verification to the user.
    user.sendEmailVerification();

    // Show a success message and navigate to the AuthScreen.
    showDialog(
        context: context,
        builder: (context) => reusable_AlertDialog(
            "User Created Please Verify Email: ${user.email}",
            context)).then((value) {
        Navigator.pushAndRemoveUntil(

```

```

        context,
        MaterialPageRoute(builder: (context) => AuthScreen()),
        (route) => false);
    });
}
}
} catch (error, stackTrace) {
showDialog(
    context: context,
    builder: (context) =>
        reusable_AlertDialog("This Email Already In Use", context));
}
}
@Override
Widget build(BuildContext context) {
// Build the RegistrationScreen scaffold.
return Scaffold(
    backgroundColor: Color.fromARGB(255, 253, 225, 183),
    appBar: AppBar(
        title: Text("Registration"),
        centerTitle: true,
    ),
    body: Padding(
        padding: const EdgeInsets.symmetric(vertical: 0, horizontal: 20),
        child: SingleChildScrollView(
            child: Column(
                children: [
                    SizedBox(
                        height: 20,
                    ),
                    reusableTextField("Enter Full Name", Icons.person_outlined, false,
                        _nameTextController),
                    SizedBox(
                        height: 20,
                    ),
                    reusableTextField("Enter Email", Icons.email_outlined, false,
                        _emailTextController),
                    SizedBox(
                        height: 20,
                    ),
                    reusableTextField("Verify Email", Icons.verified_user_outlined,
                        false, _verifyEmailTextController),
                    SizedBox(
                        height: 20,
                    ),
                    reusableTextField("Enter Password", Icons.lock_outlined, true,
                        _passwordTextController),
                    SizedBox(

```

```
        height: 20,
    ),
    reusableTextField("Verify Password", Icons.lock_outlined, true,
        _verifyPassTextController),
    SizedBox(
        height: 20,
    ),
    ElevatedButton(
        onPressed: () async {
            await signInFunction();
        },
        child: Row(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
                Text(
                    "Sign Up",
                    style: TextStyle(fontSize: 20, color: Colors.white),
                )
            ],
        ),
        style: ButtonStyle(
            backgroundColor: MaterialStateProperty.all(Colors.black),
            padding: MaterialStateProperty.all(
                EdgeInsets.symmetric(vertical: 20))),
        ),
    ],
),
),
),
),
);
}
}
```

mainlyScreen

```
// Import necessary packages.
import "package:firebase_messaging/firebase_messaging.dart";
import "package:flutter/material.dart";
import "package:geolocator/geolocator.dart";
import "package:google_maps_flutter/google_maps_flutter.dart";
import "package:google_nav_bar/google_nav_bar.dart";
import "package:task_manage_app/models/group_controller.dart";
import "package:task_manage_app/models/notification_controller.dart";
import "package:task_manage_app/models/task_controller.dart";
import "package:task_manage_app/models/user_controller.dart";
import "package:task_manage_app/models/user_model.dart";
import "package:task_manage_app/screens/home_screen.dart";
import "package:task_manage_app/screens/options_screen.dart";
import "package:task_manage_app/screens/search_screen.dart";
import 'dart:convert';
import 'dart:async';
import 'package:http/http.dart' as http;

// Create a global key for accessing the MainlyScreenState.
 GlobalKey<MainlyScreenState> mainScreenKey = GlobalKey<MainlyScreenState>();

// MainlyScreen is the main screen of the application after login.
class MainlyScreen extends StatefulWidget {
    UserModel user;
    MainlyScreen(this.user, {Key? key}) : super(key: key);

    @override
    State<MainlyScreen> createState() => _MainlyScreenState();
}

class _MainlyScreenState extends State>MainlyScreen> {
    int pageIndex = 0;
    late Position currentPosition;
    late LatLng currentLatLng;
    FirebaseMessaging messaging = FirebaseMessaging.instance;
    UserController _userController = UserController();
    TaskController _taskController = TaskController();
    GroupController _groupController = GroupController();
    List<String> taskLocations = [];
    NotificationController _notificationController = NotificationController();

    Timer? locationTimer;

    // Request notification permissions.
    Future<void> notificationPermission() async {


```

```

NotificationSettings settings = await messaging.requestPermission(
    alert: true,
    announcement: false,
    badge: true,
    carPlay: false,
    criticalAlert: false,
    provisional: false,
    sound: true,
);

print('User granted permission: ${settings.authorizationStatus}');
}

// Set up token refresh for push notifications.
void setupTokenRefresh(UserModel user) {
    FirebaseMessaging.instance.onTokenRefresh.listen((newToken) {
        _userController.setToken(widget.user.uid, newToken);
        print("token changed");
    });
}

@Override
void initState() {
    super.initState();
    setupTokenRefresh(widget.user);
    notificationPermission().then((value) {
        getLocationPermission();
        getCurrentLocation();
        locationTimer = Timer.periodic(Duration(minutes: 30), (timer) {
            getCurrentLocation();
        });
    });
}

// Request location permission.
void getLocationPermission() async {
    LocationPermission permission = await Geolocator.checkPermission();
    if (permission == LocationPermission.denied) {
        permission = await Geolocator.requestPermission();
        if (permission == LocationPermission.deniedForever) {
            // Permissions are denied forever, handle appropriately.
            return showDialog(
                context: context,
                builder: (BuildContext context) {
                    return AlertDialog(
                        backgroundColor: Color.fromRGBO(255, 255, 202, 123),
                        title: const Text("Error"),
                        content: const Text(

```

```

        'Location permissions are permanently denied, please enable
them for the best experience.',
        style: TextStyle(color: Colors.black),
    ),
    actions: <Widget>[
        TextButton(
            child: Text("OK"),
            onPressed: () {
                Navigator.of(context).pop();
            },
        ),
    ],
);
}
});

// Get the current device location.
void getCurrentLocation() async {
    LocationPermission permission = await Geolocator.checkPermission();

    if (permission == LocationPermission.whileInUse ||
        permission == LocationPermission.always) {
        currentPosition = await Geolocator.getCurrentPosition(
            desiredAccuracy: LocationAccuracy.high);
        currentLatLng =
            LatLng(currentPosition.latitude, currentPosition.longitude);
        checkClosePlaces();
    }
}

// Handle tab item tap event and update the selected index.
void _onItemTapped(int index) {
    setState(() {
        pageIndex = index;
    });
}

// Fetch nearby places based on the user's location.
Future<void> getNearbyPlaces(String locationType) async {
    const String apiKey = "AIzaSyAiVDx-0lq5biKrKHE7hijl5ANbtrfCbwQ";
    final String baseUrl =
        "https://maps.googleapis.com/maps/api/place/nearbysearch/json";
    final String location =
        "${currentPosition.latitude},${currentPosition.longitude}";
    final int radius = 100;
    final String type = locationType;
}

```

```

final String url =
    "$baseUrl?location=$location&radius=$radius&type=$type&key=$apiKey";
print(location);

final response = await http.get(Uri.parse(url));
if (response.statusCode == 200) {
    Map<String, dynamic> data = json.decode(response.body);
    List<dynamic> places = data['results'];
    print(data);
    if (!places.isEmpty) {
        String name = places.first["name"];
        await _userController.getToken(widget.user.uid).then((value) {
            print(value);
            _notificationController.sendCloseToLocationNotification(value, name);
        });
    }
} else {
    print('Failed to load nearby places');
}
}

// Fetch task locations from user's groups.
void getTasksLocations() async {
    List<String> myGroupUids =
        await _userController.fetchGroupUids(widget.user.uid);
    myGroupUids.forEach((groupUid) async {
        List<String> myTasksUids = await _groupController.fetchTaskUids(groupUid);
        myTasksUids.forEach((taskUid) async {
            String flagIsTask = await _taskController.isTask(groupUid, taskUid);
            if (flagIsTask == "true") {
                String flagStatus =
                    await _taskController.getStatus(groupUid, taskUid);
                if (flagStatus.contains("published")) {
                    List<dynamic> assignedToUid =
                        await _taskController.getAssignedToUid(groupUid, taskUid);
                    assignedToUid.forEach((uid) async {
                        if (uid == widget.user.uid) {
                            String location =
                                await _taskController.getTaskLocation(groupUid, taskUid);
                            taskLocations.add(location);
                        }
                    });
                }
            }
        });
    });
}
}

```

```

// Check if user is close to any task locations.
Future<void> checkClosePlaces() async {
    getTasksLocations();
    await Future.delayed(Duration(seconds: 5));
    taskLocations.forEach((taskLocation) {
        if (!taskLocation.contains("None")) {
            getNearbyPlaces(taskLocation);
        }
    });
    print(taskLocations);

    taskLocations = [];
}

// Stop the location update timer.
void stopTimer() {
    locationTimer?.cancel();
}

// Build the MainlyScreen widget.
@Override
Widget build(BuildContext context) {
    // Define the pages to be displayed in the bottom navigation bar.
    final List<Widget> _pages = [
        HomeScreen(widget.user),
        SearchScreen(widget.user),
        OptionsScreen(widget.user)
    ];

    // Create the main scaffold of the screen.
    return Scaffold(
        backgroundColor: Color.fromARGB(255, 253, 225, 183),
        body: _pages[indexPage], // Display the selected page.
        bottomNavigationBar: Container(
            color: Color.fromARGB(255, 255, 202, 123),
            child: Padding(
                padding: const EdgeInsets.symmetric(horizontal: 10.0, vertical: 12),
                child: GNav(
                    gap: 8,
                    backgroundColor: Color.fromARGB(255, 255, 202, 123),
                    tabBackgroundColor: Colors.white.withOpacity(0.2),
                    padding: EdgeInsets.all(16),
                    onTabChange: (value) {
                        _onItemTapped(value); // Handle tab changes.
                    },
                    tabs: const [
                        GButton(

```

```
        icon: Icons.home_max_outlined,
        text: "Home",
    ),
    GButton(
        icon: Icons.search_outlined,
        text: "Search",
    ),
    GButton(
        icon: Icons.settings_applications_outlined,
        text: "Settings",
    ),
),
],
),
),
),
);
}
}
```

homeScreen

```
import "dart:async";
import "package:cloud_firestore/cloud_firestore.dart";
import "package:flutter/material.dart";
import "package:task_manage_app/models/check_list_controller.dart";
import "package:task_manage_app/models/group_controller.dart";
import "package:task_manage_app/models/group_model.dart";
import "package:task_manage_app/models/notification_model.dart";
import "package:task_manage_app/models/popmenu.dart";
import "package:task_manage_app/models/task_controller.dart";
import "package:task_manage_app/models/user_controller.dart";
import "package:task_manage_app/models/user_model.dart";
import "package:task_manage_app/screens/room_screen.dart";
import "package:task_manage_app/widgets/create_group.dart";
import 'package:intl/intl.dart';

// This is the HomeScreen class that represents the main screen of the app.
class HomeScreen extends StatefulWidget {
  UserModel user;

  // Constructor for the HomeScreen class.
  HomeScreen(this.user);

  @override
  State<HomeScreen> createState() => _HomeScreenState();
}

class _HomeScreenState extends State<HomeScreen> {
  late GroupModel groupModel;
  Map<String, dynamic>? friendsListData;
  TaskController _taskController = TaskController();
  GroupController _groupController = GroupController();
  CheckListController _checkListController = CheckListController();
  UserController _userController = UserController();
  int? countTask;
  int? seenTask;

  // This function fetches a stream of notifications.
  Stream<List<NotificationItem>> fetchNotificationsStream() {
    return _checkListController
      .getCheckListStream(widget.user.uid)
      .map((snapshot) {
        List<NotificationItem> notifications = snapshot.docs.map((doc) {
          return NotificationItem(
            doc["uid"],

```

```

        doc["whoDidName"] +
            " Did " +
            doc["taskInfo"] +
            " in " +
            doc["groupName"] +
            " group ",
        "Do you accept it?",  

        doc["done"],  

        doc["date"],  

        doc["groupUid"],  

        doc["taskUid"],  

        doc["whoDid"]);
    }).toList();

    // Order and filter the notifications locally
    List<NotificationItem> orderedAndFiltered = notifications
        .where((notification) => !notification.done)
        .toList()
        ..sort((a, b) => b.date.compareTo(a.date));

    return orderedAndFiltered; // Return the ordered and filtered list
);
}

// This function shows the notifications menu.
void _showNotifications(BuildContext context) {
    // Calculate the position of the menu relative to a button.
    final RenderBox button = context.findRenderObject() as RenderBox;
    final RenderBox overlay =
        Overlay.of(context).context.findRenderObject() as RenderBox;
    final RelativeRect position = RelativeRect.fromRect(
        Rect.fromPoints(
            button.localToGlobal(
                Offset(button.size.width, 50),
                ancestor: overlay,
            ),
            button.localToGlobal(
                Offset(button.size.width, button.size.height + 50),
                ancestor: overlay,
            ),
        ),
        Offset.zero & overlay.size,
    );

    // Show the notifications menu.
    showMenu<NotificationItem>(
        context: context,
        position: position,

```

```

        items: _buildMenuItems(context),
    );
}

// This function builds the menu items for notifications.
List<PopupMenuEntry<NotificationItem>> _buildMenuItems(BuildContext context) {
    return [
        PopupMenuItemWidget(
            child: StreamBuilder<List<NotificationItem>>(
                stream: fetchNotificationsStream(),
                builder: (context, snapshot) {
                    if (snapshot.connectionState == ConnectionState.waiting) {
                        return CircularProgressIndicator();
                    }

                    if (snapshot.hasError) {
                        return Text('Error: ${snapshot.error}');
                    }
                }
            )
        )
    ];
}

List<NotificationItem> notifications = snapshot.data ?? [];
if (notifications.length < 1) {
    return Padding(
        padding: const EdgeInsets.all(10.0),
        child: Text("No Notifications!"),
    );
}
return Column(
    children: notifications.map((NotificationItem notification) {
        return PopupMenuItem<NotificationItem>(
            value: notification,
            enabled: !notification.done,
            child: ListTile(
                title: Text(notification.title),
                subtitle: Text(notification.content),
                trailing: Row(
                    mainAxisSize: MainAxisSize.min,
                    children: [
                        IconButton(
                            icon: Icon(Icons.check),
                            onPressed: () {
                                _checkListController.updateDone(
                                    true, widget.user.uid, notification.uid);
                                _taskController.updateTaskStatus(
                                    notification.groupUid,
                                    notification.taskUid,
                                    "TaskStatus.completed");
                                _groupController
                                    .getFriendsList(notification.groupUid)
                            }
                        )
                    ],
                )
            )
        );
    })
);
}

```

```

        .then((friendsList) {
            setState(() {
                friendsListData = friendsList;
            });
        }).then((value) => {
            friendsListData![notification.whoDid]++;
            _groupController.updateFriendsPoints(
                notification.groupUid,
                friendsListData!)
        });
    },
),
IconButton(
    icon: Icon(Icons.close),
    onPressed: () {
        _taskController.updateTaskStatus(
            notification.groupUid,
            notification.taskUid,
            "TaskStatus.published");
    }
),
_checkListController.deleteCheckList(
    widget.user.uid, notification.uid);
},
),
],
),
),
),
);
}).toList(),
);
},
),
),
],
);
];
}
}

// This function formats a date.
String formatDate(DateTime date) {
    // formatting date to difference from now to date
    final now = DateTime.now();
    final today = DateTime(now.year, now.month, now.day);
    final oneWeekAgo = today.subtract(Duration(days: 7));

    // Check if the date was within the last week
    if (date.isAfter(oneWeekAgo)) {
        // Get the difference in days between the date and now
        int daysAgo = today.difference(date).inDays;
    }
}

```

```

    if (daysAgo == 0) {
        return _formatDateToTime(date);
    } else if (daysAgo == 1) {
        return 'Yesterday';
    } else {
        // Format the date as the day of the week if it was less than a week ago
        return DateFormat('EEEE').format(date);
    }
} else if (date.isAfter(oneWeekAgo.subtract(Duration(days: 1)))) {
    return 'A week ago';
} else {
    // Format the date as a full date if it was more than a week ago
    return DateFormat.yMMMd().format(date);
}
}

// This function formats a date to time.
String _formatDateToTime(DateTime date) {
    final formatter = DateFormat('HH:mm');
    return formatter.format(date);
}

@Override
Widget build(BuildContext context) {
    // Scaffold is the main structure of the screen.
    return Scaffold(
        // Set the background color.
        backgroundColor: Color.fromARGB(255, 253, 225, 183),

        // AppBar is the top app bar with title and actions.
        appBar: AppBar(
            title: Text("Home"), // Display the title "Home".
            centerTitle: true, // Center-align the title.
            backgroundColor:
                Color.fromARGB(255, 255, 202, 123), // Set the app bar color.
            actions: [
                // Display notifications icon and count using a StreamBuilder.
                StreamBuilder<List<NotificationItem>>(
                    stream: fetchNotificationsStream(),
                    builder: (context, snapshot) {
                        if (snapshot.connectionState == ConnectionState.waiting) {
                            return Padding(
                                padding: const EdgeInsets.all(12.0),
                                child: CircularProgressIndicator(),
                            );
                        }
                        if (snapshot.hasError) {
                            return Text('Error: ${snapshot.error}');
                        }
                    }
                )
            ]
        )
    );
}

```

```

    }
    List<NotificationItem> notifications = snapshot.data ?? [];
    bool hasNewNotification =
        notifications.any((notification) => !notification.done);

    return Stack(
        children: [
            IconButton(
                onPressed: () {
                    _showNotifications(context);
                },
                icon: Icon(Icons.notifications),
            ),
            if (hasNewNotification)
                Positioned(
                    right: 0,
                    child: Container(
                        padding: EdgeInsets.all(1),
                        decoration: BoxDecoration(
                            color: Colors.red,
                            borderRadius: BorderRadius.circular(6),
                        ),
                        constraints: BoxConstraints(
                            minWidth: 12,
                            minHeight: 12,
                        ),
                    ),
                ),
        ],
    );
},
],
),
],
),
),
body: StreamBuilder<QuerySnapshot>(
    stream: _groupController
        .getUserGroupsOrderedByCreationDate(widget.user.uid),
    builder: (context, AsyncSnapshot<QuerySnapshot> snapshot) {
        if (snapshot.connectionState == ConnectionState.waiting) {
            return Center(
                child: CircularProgressIndicator(),
            );
        }
        if (snapshot.hasError) {
            return Center(
                child: Text("Error: ${snapshot.error}"),
            );
        }
    }
}

```

```

if (snapshot.data!.docs.length < 1) {
    return Center(
        child: Text("No Groups Available"),
    );
}
return ListView.builder(
    itemCount: snapshot.data!.docs.length,
    itemBuilder: (context, index) {
        var group = snapshot.data!.docs[index];
        return StreamBuilder<DocumentSnapshot>(
            stream: group["groupRef"].snapshots(),
            builder: (BuildContext context,
                AsyncSnapshot<DocumentSnapshot> groupSnapshot) {
                if (groupSnapshot.connectionState ==
                    ConnectionState.waiting) {
                    return ListTile(
                        leading: CircleAvatar(),
                        title: Text('Loading...'),
                        subtitle: Text(''),
                    );
                }
                if (groupSnapshot.hasError) {
                    return ListTile(
                        leading: CircleAvatar(),
                        title: Text('Error: ${groupSnapshot.error}'),
                        subtitle: Text(''),
                    );
                }
                if (!groupSnapshot.hasData) {
                    return ListTile(
                        leading: CircleAvatar(),
                        title: Text('No data available'),
                        subtitle: Text(''),
                    );
                }
            }
        );
    }
}

// Extract group data from the document snapshot.
Map<String, dynamic>? groupData =
    groupSnapshot.data!.data() as Map<String, dynamic>?;

if (groupData == null) {
    return ListTile(
        leading: CircleAvatar(),
        title: Text('Group Deleted'),
        subtitle: Text(''),
    );
}
String groupUid = groupData["uid"] as String;

```

```

String groupCreatorUid = groupData["creatorUid"] as String;
Map<String, dynamic> friendsPoints =
    groupData["friendsPoints"] as Map<String, dynamic>;

String groupName = groupData["groupName"] as String;
String groupImage = groupData["image"] as String;
String taskInfo = groupData["lastTaskCreator"] as String;
Timestamp taskCreationDate =
    groupData["taskCreationDate"] as Timestamp;
Timestamp endTime = Timestamp.now();
Duration duration =
    endTime.toDate().difference(taskCreationDate.toDate());

// Extract task count and seen task count.
countTask = groupData["taskCount"] as int;
seenTask = group["countSeen"] as int;

return Dismissible(
    key: Key(
        groupUid), // this key should be unique for each ListTile
    background: Container(
        color: Colors.red,
        alignment: Alignment.centerRight,
        child: Padding(
            padding: const EdgeInsets.only(right: 10.0),
            child: Icon(
                Icons.delete,
                color: Colors.white,
            ),
        ),
    ),
    onDismissed: (direction) {
        final groupData =
            _groupController.getGroupByUid(groupUid);

        if (widget.user.uid == groupCreatorUid) {
            groupData.then((doc) {
                if (doc.exists) {
                    var groupUsers = doc.data()?[ 'friendsPoints' ].keys;
                    _groupController.deleteGroup(groupUid);
                    groupUsers.forEach((groupUser) => _groupController
                        .deleteGroupRef(groupUser, groupUid));
                } else {
                    print('Document does not exist on the database');
                }
            });
        } else {
            friendsPoints.remove(widget.user.uid);
        }
    },
);
}

```

```

        _groupController.deleteGroupRef(
            widget.user.uid, groupUid);
        _groupController.updateFriendsPoints(
            groupUid, friendsPoints);

        _taskController.setQuitGroupTask(
            groupUid, widget.user.name);
    }
},
child: Stack(children: [
    ListTile(
        leading: CircleAvatar(
            child: Image.network(groupImage),
        ),
        title: Text(groupName),
        trailing: Text(formatDate(taskCreationDate.toDate())),
        subtitle: Container(
            child: Text(
                taskInfo,
                style: TextStyle(color: Colors.grey),
                overflow: TextOverflow.ellipsis,
            ),
        ),
        onTap: () {
            GroupModel groupModel =
                GroupModel.fromJson(groupData);
            _userController.updateCountSeen(
                widget.user.uid, groupUid, groupModel.taskCount);
            Navigator.push(
                context,
                MaterialPageRoute(
                    builder: (context) => RoomScreen(
                        currentUser: widget.user,
                        group: groupModel,
                        onUpdateImage: (newImageUrl) {
                            setState(() {
                                groupImage = newImageUrl;
                            });
                        },
                        onUpdateName: (newName) {
                            setState(() {
                                groupImage = newName;
                            });
                        },
                        onUpdateTask: (newName) {
                            setState(() {
                                taskInfo = newName;
                            });
                        };
                    )
                );
        }
    );
}

```



```
showDialog(  
    context: context,  
    builder: (context) => GroupCreationDialog(  
        widget.user,  
        ),  
    );  
},  
);  
}  
}
```

roomScreen

```
import "package:cloud_firestore/cloud_firestore.dart";
import "package:flutter/material.dart";
import "package:task_manage_app/models/group_controller.dart";
import "package:task_manage_app/models/group_model.dart";
import "package:task_manage_app/models/task_controller.dart";
import "package:task_manage_app/models/task_model.dart";
import "package:task_manage_app/models/user_controller.dart";
import "package:task_manage_app/models/user_model.dart";
import "package:task_manage_app/screens/room_options_screen.dart";
import "package:task_manage_app/widgets/single_message.dart";
import "package:task_manage_app/widgets/task_creation.dart";

// Define a StatefulWidget for the RoomScreen.
class RoomScreen extends StatefulWidget {
    UserModel currentUser;
    GroupModel group;
    Function(String) onUpdateImage;
    Function(String) onUpdateName;
    Function(String) onUpdateTask;
    Function(Timestamp) onUpdateTaskTime;

    RoomScreen({
        required this.currentUser,
        required this.group,
        required this.onUpdateImage,
        required this.onUpdateName,
        required this.onUpdateTask,
        required this.onUpdateTaskTime,
    });

    @override
    _RoomScreenState createState() => _RoomScreenState();
}

// Define the state for the RoomScreen widget.
class _RoomScreenState extends State<RoomScreen> {
    // Define widget state variables.
    String? imageUrl;
    String? groupName;
    String? taskInfoCreator;
    Timestamp? taskTimeCreation;
    String? taskStatus;
    int? seenTask;

    final ScrollController scrollController = ScrollController();
```

```

GroupController _groupController = GroupController();
TaskController _taskController = TaskController();
UserController _userController = UserController();

// Initialize the widget state.
void initState() {
    super.initState();

    // Fetch and set the group name.
    _groupController.getGroupName(widget.group.uid).then((groupName) => {
        setState(() {
            _groupName = groupName;
        })
    });
}

// Fetch and set the group image URL.
_groupController.getImageUrl(widget.group.uid).then((url) {
    setState(() {
        imageUrl = url;
    });
});

// Fetch and set the group task info creator.
_groupController.getGroupTaskInfo(widget.group.uid).then((taskInfo) {
    setState(() {
        _taskInfoCreator = taskInfo;
    });
});

// Fetch and set the number of seen tasks for the current user.
_userController
    .getCountSeen(widget.currentUser.uid, widget.group.uid)
    .then((count) {
    setState(() {
        seenTask = count;
    });
});

// Fetch and set the task creation time for the group.
_groupController
    .getGroupTaskCreationTime(widget.group.uid)
    .then((taskCreationTime) {
    setState(() {
        _taskTimeCreation = taskCreationTime;
    });
});
}

```

```

@Override
Widget build(BuildContext context) {
  return Scaffold(
    backgroundColor: Color.fromARGB(255, 253, 225, 183),
    appBar: AppBar(
      // Define the app bar for the room screen.
      backgroundColor: Color.fromARGB(255, 255, 202, 123),
      title: GestureDetector(
        onTap: () {
          // Navigate to room options screen on app bar title tap.
          Navigator.push(
            context,
            MaterialPageRoute(
              builder: (context) => RoomOptionsScreen(
                widget.group,
                widget.currentUser,
                (newImageUrl) {
                  setState(() {
                    imageUrl = newImageUrl;
                    widget.onUpdateImage(newImageUrl);
                  });
                },
                (newName) {
                  setState(() {
                    groupName = newName;
                    widget.onUpdateName(newName);
                  });
                },
              )));
        },
        child: Row(children: [
          ClipRRect(
            borderRadius: BorderRadius.circular(80),
            child: imageUrl != null
              ? Image.network(
                  imageUrl!,
                  height: 35,
                )
              : CircularProgressIndicator(),
        ),
        SizedBox(
          width: 5,
        ),
        groupName != null
          ? Text(
              groupName!,
              style: TextStyle(fontSize: 20),
            )
        ],
      ),
    ),
  );
}

```

```
        ),
        : CircularProgressIndicator(),
    ],
),
),
body: Column(
    children: [
        SizedBox(
            height: 20,
        ),
        Expanded(
            child: Container(
                padding: EdgeInsets.all(10),
                decoration: BoxDecoration(
                    color: Color.fromRGBO(255, 253, 225, 183),
                    borderRadius: BorderRadius.only(
                        topLeft: Radius.circular(25),
                        topRight: Radius.circular(25))),
            ),
        ),
        StreamBuilder(
            stream: _taskController
                .getTasksOrderedByCreationDate(widget.group.uid),
            builder: (context, AsyncSnapshot snapshot) {
                if (snapshot.hasData) {
                    if (snapshot.data.docs.length < 1) {
                        return Center(
                            child: Text("Set A Task"),
                        );
                    }
                }
            },
        ),
    ],
),
return ListView.builder(
    itemCount: snapshot.data.docs.length,
    reverse: false,
    controller:
        _scrollController, // Controller is attached here
    physics: BouncingScrollPhysics(),
    itemBuilder: (context, index) {
        if (snapshot.data.docs[index]["isTask"] == "false")
            return Center(
                child: Card(
                    color: Color.fromRGBO(255, 255, 202, 123),
                    child: Text(snapshot
                        .data.docs[index]["lastTaskCreator"]
                        .toString()),
                ),
            );
        bool isMe = snapshot.data.docs[index]["creatorUid"] ==
            widget.currentUser.uid;
```

```

        TaskModel taskModel = TaskModel.fromJson(
            snapshot.data.docs[index].data()));
        return SingleMessage(
            group: widget.group,
            taskModel: taskModel,
            user: widget.currentUser,
            message: snapshot.data.docs[index]["taskInfo"],
            isMe: isMe,
            controller: _scrollController,
        );
    },
),
),
),
),
),
Container(
    width: MediaQuery.of(context).size.width,
    height: 86,
    color: Color.fromARGB(
        255, 255, 202, 123), // Change this color as needed
)
],
),
floatingActionButton: StreamBuilder<DocumentSnapshot>(
    stream: _groupController.getGroupStream(widget.group.uid),
    builder: (context, snapshot) {
        if (snapshot.connectionState == ConnectionState.active) {
            if (snapshot.hasData && snapshot.data != null) {
                Map<String, dynamic>? groupData =
                    snapshot.data!.data() as Map<String, dynamic>?;
                String businessMode =
                    groupData!["businessMode"] as String; // default to mode2
                // If businessMode is mode1 and current user is not the creator,
                hide the button
                if (businessMode == "mode1" &&
                    widget.currentUser.uid != widget.group.creatorUid) {
                    return SizedBox.shrink(); // returns an empty widget
                }
                // If businessMode is mode2 or the current user is the creator,
                show the button
                return FloatingActionButton(
                    child: Icon(Icons.add),
                    onPressed: () {
                        showDialog(
                            context: context,

```


roomOptionsScreen

```
import "dart:io";
import 'package:firebase_storage/firebase_storage.dart' as firebase_storage;
import "package:task_manage_app/models/group_controller.dart";
import "package:cloud_firestore/cloud_firestore.dart";
import "package:flutter/material.dart";
import "package:image_cropper/image_cropper.dart";
import "package:image_picker/image_picker.dart";
import "package:task_manage_app/models/friends_selection.dart";
import "package:task_manage_app/models/group_model.dart";
import "package:task_manage_app/models/task_controller.dart";
import "package:task_manage_app/models/user_model.dart";
import "package:task_manage_app/widgets/friends_list.dart";
import "package:task_manage_app/widgets/friends_list_widget.dart";
import "package:task_manage_app/widgets/reusable_widgets.dart";

class RoomOptionsScreen extends StatefulWidget {
    GroupModel group;
    UserModel user;
    Function(String) onUpdateImage;
    Function(String) onUpdateName;

    RoomOptionsScreen(
        this.group, this.user, this.onUpdateImage, this.onUpdateName);

    @override
    State<RoomOptionsScreen> createState() => _RoomOptionsScreenState();
}

class _RoomOptionsScreenState extends State<RoomOptionsScreen> {
    String? imageUrl;
    String _groupName = "";
    TextEditingController _groupNameController = TextEditingController();
    List<String> friendsUidList = [];
    Map<String, int> friendsPoints = {};
    late ValueNotifier<String> businessMode;
    List<String> selectedFriends = [];
    List<String> selectedFriendsUid = [];

    File? _selectedImage;

    TaskController _taskController = TaskController();
    GroupController _groupController = GroupController();

    void initState() {
        super.initState();
    }
}
```

```

_groupController.getimageUrl(widget.group.uid).then((url) {
  setState(() {
    imageUrl = url;
  });
});
_groupNameController.addListener(_onChange);
fetchFriends();
businessMode = ValueNotifier<String>(widget.group.businessMode);
}

void _onChange() {
  groupName = _groupNameController.text;
}

void dispose() {
  _groupNameController.dispose();
  super.dispose();
}

Future<void> _uploadImageToFirebase() async {
  if (_selectedImage == null) return;

  try {
    final fileName = DateTime.now().millisecondsSinceEpoch.toString();
    final destination = 'assets/images/$fileName';

    final storageRef =
        firebase_storage.FirebaseStorage.instance.ref().child(destination);
    final uploadTask = storageRef.putFile(_selectedImage!);
    final snapshot = await uploadTask.whenComplete(() {});

    final downloadUrl = await snapshot.ref.getDownloadURL();

    _groupController.updateImage(widget.group.uid, downloadUrl);
    _taskController.setChangeGroupImageTask(
      widget.group.uid, widget.user.name);
  }

  setState(() {
    imageUrl =
        downloadUrl; // Update the imageUrl variable with the new download
URL
  });
  widget.onUpdateImage(downloadUrl);
} catch (error) {
  print('Upload error: $error');
}
}

```

```

Future<void> _pickImage() async {
    final imagePicker = ImagePicker();

    final pickedImage =
        await imagePicker.pickImage(source: ImageSource.gallery);

    if (pickedImage != null) {
        ImageCropper imageCropper = ImageCropper();
        final croppedImage = await imageCropper.cropImage(
            sourcePath: pickedImage.path,
            aspectRatio: CropAspectRatio(ratioX: 1, ratioY: 1),
            compressQuality: 70,
            maxWidth: 80,
            maxHeight: 80,
        );
    }

    if (croppedImage != null) {
        setState(() {
            _selectedImage = File(croppedImage.path);
        });
    }
}

Future<void> updateBusinessMode() async {
    String mode;
    if (businessMode.toString().contains("mode1"))
        mode = "mode1";
    else
        mode = "mode2";

    _groupController.updateBusinessMode(widget.group.uid, mode);
}

Future<void> fetchFriends() async {
    try {
        final documentSnapshot = _groupController.getGroupByUid(widget.group.uid);
        documentSnapshot.then((doc) {
            if (doc.exists) {
                var groupUsers = doc.data()?['friendsPoints'];
                var friendsListData = groupUsers.keys;

                setState(() {
                    groupUsers.forEach((key, value) => friendsPoints[key] = value);
                    friendsUidList = List<String>.from(friendsListData);
                });
            } else {
                print('Document does not exist on the database');
            }
        });
    }
}

```

```

        }
    });
} catch (error) {
    // Handle error
    print("Error fetching friends: $error");
}
}

void resetPoints() {
    setState(() {
        friendsPoints.forEach((key, value) {
            friendsPoints[key] = 0;
        });
    });
}

_groupController.updateFriendsPoints(widget.group.uid, friendsPoints);
}

Future<void> handleFriendSelection(FriendSelectionResult result) async {
    int taskCount = await GroupController().getTaskCount(widget.group.uid);
    setState(() {
        selectedFriends = [];
        selectedFriendsUid = [];
        for (final friendUid in result.friendUids) {
            int index = 0;
            if (!friendsUidList.contains(friendUid)) {
                _groupController.setGroupRef(friendUid, widget.group.uid, taskCount);

                friendsUidList.add(friendUid);
                friendsPoints[friendUid] = 0;

                _taskController.setAddToTheGroupTask(
                    widget.group.uid, result.friendNames[index]);
            }
        }
    });
}

_groupController.updateFriendsPoints(widget.group.uid, friendsPoints);
}

@Override
Widget build(BuildContext context) {
    Future<String> groupName = _groupController.getGroupName(widget.group.uid);

    return Scaffold(
        backgroundColor: Color.fromARGB(255, 253, 225, 183),
        appBar: AppBar(
            title: Text("Group Settings"),

```

```

centerTitle: true,
backgroundColor: Color.fromARGB(255, 255, 202, 123),
actions: [
  Padding(
    padding: const EdgeInsets.all(12.0),
  )
],
),
body: Padding(
  padding: const EdgeInsets.symmetric(vertical: 0, horizontal: 20),
  child: SingleChildScrollView(
    child: Center(
      child: Column(
        children: [
          SizedBox(
            height: 20,
          ),
          GestureDetector(
            onTap: () {
              _pickImage().then((_) {
                _uploadImageToFirebase();
              });
            },
            child: imageUrl != null
              ? Container(
                  decoration: BoxDecoration(
                    border:
                      Border.all(color: Colors.black, width: 0.4),
                  ),
                  child: Image.network(imageUrl!),
                  height: 80,
                  width: 80,
                )
              : CircularProgressIndicator(),
            ),
          SizedBox(
            height: 29,
          ),
          reusable_StringBuilder(
            groupName, Icons.group_outlined, _groupNameController),
          SizedBox(
            height: 20,
          ),
          reusable_button("Update Settings", () async {
            if (_groupName == "" || groupName == _groupName) {
              return;
            }
            _groupController.updateGroupName(

```

```

        widget.group.uid, _groupName);
    widget.onUpdateName(_groupName);
    _taskController.setChangeGroupNameTask(
        widget.group.uid, widget.user.name);

    showDialog(
        context: context,
        builder: (context) => reusable_AlertDialog(
            "Settings been changed", context));
),
SizedBox(
    height: 20,
),
widget.user.uid == widget.group.creatorUid
    ? reusable_button("Reset Points ", resetPoints)
    : SizedBox(),
SizedBox(
    height: 10,
),
ValueListenableBuilder(
    valueListenable: businessMode,
    builder: (context, value, child) {
        return SwitchListTile(
            title: Text('Business Mode'),
            value: value == 'mode1',
            onChanged: (newValue) {
                if (widget.user.uid == widget.group.creatorUid) {
                    _taskController.setChangeGroupModeTask(
                        widget.group.uid, widget.user.name);
                    businessMode.value = newValue ? 'mode1' : 'mode2';
                    updateBusinessMode();
                } else {
                    null;
                }
            },
        );
    },
),
Text(
    "Friends:",
    style: TextStyle(decoration: TextDecoration.underline),
),
FriendsListView(friendsUidList, friendsPoints, widget.user,
    widget.group),
SizedBox(
    height: 20,
),
widget.user.uid == widget.group.creatorUid

```

```
? reusable_button("Add Friends", () {
    Navigator.push(
        context,
        MaterialPageRoute(
            builder: (context) => FriendsListPage(
                currentUser: widget.user,
                onFriendSelection: handleFriendSelection,
                initialSelectedFriends: selectedFriends,
                initialSelectedFriendsUid: selectedFriendsUid,
                group: widget.group,
            ),
        ),
    );
})
: SizedBox(),
SizedBox(
    height: 20,
),
],
),
),
));
}
}
}
```

searchScreen

```
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:flutter/material.dart';
import 'package:task_manage_app/models/user_controller.dart';
import 'package:task_manage_app/models/user_model.dart';
import 'package:task_manage_app/widgets/reusable_widgets.dart';

// Import necessary libraries and packages.

// Define a StatefulWidget for the SearchScreen.
class SearchScreen extends StatefulWidget {
    UserModel user;

    SearchScreen(this.user);

    @override
    State<SearchScreen> createState() => _SearchScreenState();
}

// Define the state for the SearchScreen widget.
class _SearchScreenState extends State<SearchScreen> {
    TextEditingController searchController = TextEditingController();
    List<Map> searchResult = [];
    bool isLoading = false;
    int pageIndex = 0;
    String _search = '';

    UserController _userController = UserController();

    // Initialize the widget state.
    void initState() {
        super.initState();
        searchController.addListener(_onChange);
    }

    @override
    void dispose() {
        searchController.dispose();
        super.dispose();
    }

    // Update the _search variable when text in the search field changes.
    void _onChange() {
        setState(() {
            _search = searchController.text;
        });
    }
}
```

```

}

// Perform a user search operation.
void onSearch() async {
  setState(() {
    searchResult = [];
    isLoading = true;
  });

  String searchText = searchController.text;
  _userController.searchUsersByEmail(searchText).then((value) {
    if (value.docs.length < 1) {
      ScaffoldMessenger.of(context).showSnackBar(SnackBar(
        content: Text("No User Found"),
      ));
      setState(() {
        isLoading = false;
      });
      return;
    }
    value.docs.forEach((user) {
      if (user.data()["email"] != widget.user.email) {
        searchResult.add(user.data());
      }
    });
    setState(() {
      isLoading = false;
    });
  });
}

@Override
Widget build(BuildContext context) {
  return Scaffold(
    backgroundColor: Color.fromARGB(255, 253, 225, 183),
    appBar: AppBar(
      title: Text("Search People"),
      centerTitle: true,
      backgroundColor: Color.fromARGB(255, 255, 202, 123),
    ),
    body: Column(children: [
      // Search input field and search button.
      Row(
        children: [
          Expanded(
            child: Padding(
              padding: const EdgeInsets.all(15.0),
              child: reusableTextField(

```

```

        "type email..",
        Icons.person_outlined,
        false,
        searchController,
    ),
),
),
),
IconButton(
    onPressed: () {
        onSearch();
    },
    icon: Icon(Icons.search),
)
],
),
),

// Display search results if available.
if (searchResult.length > 0)
    Expanded(
        child: ListView.builder(
            itemCount: searchResult.length,
            shrinkWrap: true,
            itemBuilder: (context, index) {
                return ListTile(
                    leading: CircleAvatar(
                        child: Image.network(searchResult[index]["image"]),
                    ),
                    title: Text(searchResult[index]["name"]),
                    subtitle: Text(searchResult[index]["email"]),
                    trailing: IconButton(
                        onPressed: () async {
                            DocumentSnapshot userExist =
                                await _userController.getDocumentSnapshot(
                                    widget.user.uid,
                                    searchResult[index]["uid"],
                                );
                            if (!userExist.exists) {
                                _userController.setFriend(
                                    widget.user.uid,
                                    searchResult[index]["uid"],
                                );
                            }

                            setState(() {
                                searchController.text = "";
                            });
                            showDialog(
                                context: context,
                                builder: (context) =>

```

```
        reusable_AlertDialog("Friend Added", context),
    );
} else {
    showDialog(
        context: context,
        builder: (context) => reusable_AlertDialog(
            "Friend Already Exists",
            context,
        ),
    );
},
icon: Icon(Icons.add),
),
),
),
),
),
),
)
}

// Show loading indicator while searching.
else if (isLoading == true)
Center(
    child: CircularProgressIndicator(),
),
],
);
}
}
```

optionsScreen

```
// Import necessary libraries and packages.  
import "dart:io";  
import 'package:firebase_storage/firebase_storage.dart' as firebase_storage;  
import 'package:firebase_auth/firebase_auth.dart';  
import 'package:google_sign_in/google_sign_in.dart';  
import 'package:image_picker/image_picker.dart';  
import 'package:image_cropper/image_cropper.dart';  
import "package:flutter/material.dart";  
import "package:task_manage_app/models/user_controller.dart";  
import "package:task_manage_app/models/user_model.dart";  
import "package:task_manage_app/screens/auth_screen.dart";  
import "package:task_manage_app/screens/friends_screen.dart";  
import "package:task_manage_app/screens/mainly_screen.dart";  
import "package:task_manage_app/widgets/reusable_widgets.dart";  
  
// Define a StatefulWidget for the OptionsScreen.  
class OptionsScreen extends StatefulWidget {  
    UserModel user;  
    OptionsScreen(this.user);  
  
    @override  
    State<OptionsScreen> createState() => _OptionsScreenState();  
}  
  
// Define the state for the OptionsScreen widget.  
class _OptionsScreenState extends State<OptionsScreen> {  
    // Declare controllers for user input fields.  
    TextEditingController _userNameController = TextEditingController();  
    TextEditingController _emailController = TextEditingController();  
    TextEditingController _passController = TextEditingController();  
    TextEditingController _changeEmailController = TextEditingController();  
  
    // Get the current Firebase user.  
    User? user = FirebaseAuth.instance.currentUser;  
  
    // Initialize variables for user information.  
    String _userName = "", _email = "", _pass = "", _changeEmail = "";  
  
    // Declare a variable for the selected image.  
    File? _selectedImage;  
  
    // Declare a variable for the user's profile image URL.  
    String? imageUrl;
```

```

// Create a UserController instance to manage user data.
UserController _userController = UserController();

// Function to upload the selected image to Firebase Storage.
Future<void> _uploadImageToFirebase() async {
    if (_selectedImage == null) return;

    try {
        // Generate a unique filename for the image.
        final fileName = DateTime.now().millisecondsSinceEpoch.toString();
        final destination = 'assets/images/$fileName';

        // Create a reference to Firebase Storage.
        final storageRef =
            firebase_storage.FirebaseStorage.instance.ref().child(destination);

        // Upload the image file.
        final uploadTask = storageRef.putFile(_selectedImage!);

        // Wait for the upload to complete.
        final snapshot = await uploadTask.whenComplete(() {});

        // Get the download URL of the uploaded image.
        final downloadUrl = await snapshot.ref.getDownloadURL();

        // Update the user's profile image URL.
        _userController.updateImage(user!.uid, downloadUrl);

        // Update the imageUrl variable with the new download URL.
        setState(() {
            imageUrl = downloadUrl;
        });
    } catch (error) {
        print('Upload error: $error');
    }
}

// Function to pick an image from the gallery and perform cropping.
Future<void> _pickImage() async {
    final imagePicker = ImagePicker();

    // Pick an image from the gallery.
    final pickedImage =
        await imagePicker.pickImage(source: ImageSource.gallery);
    if (pickedImage != null) {
        ImageCropper imageCropper = ImageCropper();
        // Crop the picked image.
        final croppedImage = await imageCropper.cropImage(

```

```

        sourcePath: pickedImage.path,
        aspectRatio: CropAspectRatio(ratioX: 1, ratioY: 1),
        compressQuality: 70,
        maxWidth: 80,
        maxHeight: 80,
    );
    if (croppedImage != null) {
        setState(() {
            _selectedImage = File(croppedImage.path);
        });
    }
}
// Initialize the widget state.
void initState() {
    super.initState();

    // Add listeners to user input controllers.
    _userNameController.addListener(_onChange);
    _emailController.addListener(_onChange);
    _passController.addListener(_onChange);
    _changeEmailController.addListener(_onChange);

    // Load the user's profile image URL.
    _userController.getImageUrl(widget.user.uid).then((url) {
        setState(() {
            imageUrl = url;
        });
    });
}
// Dispose of controllers when the widget is disposed.
void disposeControllers() {
    super.dispose();
}
// Update state variables when the controller values change.
void _onChange() {
    setState(() {
        _userName = _userNameController.text;
        _email = _emailController.text;
        _pass = _passController.text;
        _changeEmail = _changeEmailController.text;
    });
}

@Override
Widget build(BuildContext context) {
    // Fetch the user's name and email.
    Future<String> userName = _userController.getUserName(widget.user.uid);

```

```

Future<String> email = _userController.getEmail(widget.user.uid);

// Build the OptionsScreen scaffold.
return Scaffold(
  backgroundColor: Color.fromARGB(255, 253, 225, 183),
  appBar: AppBar(
    title: Text("Settings"),
    centerTitle: true,
    backgroundColor: Color.fromARGB(255, 255, 202, 123),
    actions: [
      Padding(
        padding: const EdgeInsets.all(12.0),
        child: ElevatedButton(
          onPressed: () async {
            mainScreenKey.currentState?.stopTimer();
            await GoogleSignIn().signOut();
            await FirebaseAuth.instance.signOut();
            Navigator.pushAndRemoveUntil(
              context,
              MaterialPageRoute(builder: (context) => AuthScreen()),
              (route) => false);
          },
          child: Text(
            "Logout",
            style: TextStyle(color: Colors.white),
          ),
          style: ButtonStyle(
            backgroundColor: MaterialStateProperty.all(Colors.red),
          ),
        ),
      ),
    ],
  ),
  body: Padding(
    padding: const EdgeInsets.symmetric(vertical: 0, horizontal: 20),
    child: SingleChildScrollView(
      child: Column(
        children: [
          SizedBox(
            height: 20,
          ),
          GestureDetector(
            onTap: () {
              _pickImage().then((_) {
                _uploadImageToFirebase();
              });
            },
            child: imageUrl != null

```

```

    ? Container(
        decoration: BoxDecoration(
            border: Border.all(color: Colors.black, width: 0.4),
        ),
        child: Image.network(imageUrl!),
        height: 80,
        width: 80,
    )
    : CircularProgressIndicator(),
),
SizedBox(
    height: 20,
),
reusable_StringBuilder(
    userName, Icons.person_outlined, _userNameController),
SizedBox(
    height: 10,
),
reusable_StringBuilder(
    email, Icons.email_outlined, _emailController),
SizedBox(
    height: 40,
),
reusable_button("Change Settings", () async {
    if (_email != "" && email != user?.email) {
        if (RegExp(
            r"^[a-zA-Z0-9.a-zA-Z0-9.!#$%&'*+-/=?^`{|}~]+@[a-zA-Z0-
9]+\.[a-zA-Z]+")
            .hasMatch(_email)) {
            User? user = FirebaseAuth.instance.currentUser;
            if (user?.providerData[0].providerId == 'google.com') {
                try {
                    GoogleSignIn googleSignIn = GoogleSignIn();
                    GoogleSignInAccount? googleUser =
                        await googleSignIn.signIn();
                    if (googleUser != null) {
                        GoogleSignInAuthentication googleAuth =
                            await googleUser.authentication;
                        AuthCredential credential =
                            GoogleAuthProvider.credential(
                                accessToken: googleAuth.accessToken,
                                idToken: googleAuth.idToken,
                            );
                        await user
                            ?.reauthenticateWithCredential(credential)
                            .then((value) async =>
                                await user.updateEmail(_emailController.text))
                            .then((value) => _userController.updateEmail(

```

```

        widget.user.uid, _email))
    .then((value) async =>
        await user.sendEmailVerification())
    .then((value) => showDialog(
        context: context,
        builder: (context) => reusable_AlertDialog(
            "Email Been Changed please verify it",
            context)));
}
} catch (ex) {
    print(ex);
}
} else {
try {
    showDialog(
        context: context,
        builder: (context) => AlertDialog(
            backgroundColor:
                Color.fromARGB(255, 255, 202, 123),
            title: Text("Change Email.."),
            actions: [
                reusableTextField(
                    "Enter Password",
                    Icons.lock_outlined,
                    true,
                    _changeEmailController),
                Row(
                    children: [
                        TextButton(
                            child: Text('Cancel'),
                            onPressed: () {
                                Navigator.of(context).pop();
                            },
                        ),
                        TextButton(
                            child: Text('Change'),
                            onPressed: () async {
                                try {
                                    User? user = FirebaseAuth
                                        .instance.currentUser;
                                    AuthCredential credentials =
                                        EmailAuthProvider.credential(
                                            email: widget.user.email,
                                            password: _changeEmail);

                                    await user
                                        ?.reauthenticateWithCredential(
                                            credentials);
                                }
                            }
                        )
                    ],
                )
            ],
        )
    );
}
}
}

```

```

        await user
            ?.updateEmail(
                _emailController.text)
            .then((value) async => await
user
            .sendEmailVerification())
            .then((value) async =>
            _userController
            .updateEmail(
                user.uid, _email))
            .then((value) =>
            Navigator.of(context)
            .pop())
            .then((value) => showDialog(
            context: context,
            builder: (context) =>
            reusable_AlertDialog(
                "Email Been Changed
please verify it",
                context)));
        } catch (error) {
            final errorr =
            error.toString().split("]");
            showDialog(
            context: context,
            builder: (context) =>
            reusable_AlertDialog(
                errorr[1], context));
        }
    },
),
],
),
],
),
]);
} catch (error) {
    print('Error updating email: $error');
}
}
}
} else {
showDialog(
    context: context,
    builder: (context) => reusable_AlertDialog(
        "Email Format Is Wrong", context));
return;
}
}
if (_userName != "" && _userName != widget.user.name) {

```

```
        _userController.updateUserName(user!.uid, _userName).then(
            (value) => showDialog(
                context: context,
                builder: (context) => reusable_AlertDialog(
                    "Settings Been Changed", context)));
    }
}),
SizedBox(
    height: 10,
),
reusable_button("Friends List", () {
    Navigator.push(
        context,
        MaterialPageRoute(
            builder: (context) =>
                FreindsScreen(currentUser: widget.user)));
})
],
),
),
),
);
}
}
```

friendsScreen

```
import "package:cloud_firestore/cloud_firestore.dart";
import "package:flutter/material.dart";
import "package:task_manage_app/models/user_controller.dart";
import "package:task_manage_app/models/user_model.dart";

// This class provides the UI for displaying the list of friends for the current
user.
class FreindsScreen extends StatefulWidget {
  UserModel currentUser;

  FreindsScreen({
    required this.currentUser,
  });

  @override
  State<FreindsScreen> createState() => _FreindsScreenState();
}

class _FreindsScreenState extends State<FreindsScreen> {
  List<Map<String, dynamic>> friends = [];
  final _userController = UserController();

  @override
  void initState() {
    super.initState();
    fetchFriends();
  }

  Future<void> deleteFriend(String friendId) async {
    // Delete a friend from the friends list in Firestore.
    try {
      await _userController.deleteFriendDocReference(
        widget.currentUser.uid, friendId);
      print("Friend deleted successfully");
    } catch (e) {
      print("Failed to delete friend: $e");
    }
  }

  Future<void> fetchFriends() async {
    // Fetch the friends data from Firestore.
    try {
      QuerySnapshot querySnapshot =
        await _userController.getFriendQuerySnapshot(widget.currentUser.uid);
```

```

if (querySnapshot.docs.isNotEmpty) {
    setState(() {
        friends = querySnapshot.docs
            .map((doc) => doc.data() as Map<String, dynamic>)
            .toList();
    });
} else {
    ScaffoldMessenger.of(context)
        .showSnackBar(SnackBar(content: Text("No User Found")));
}
} catch (error) {
    print("Error fetching friends: $error");
}
}

@Override
Widget build(BuildContext context) {
    return Scaffold(
        // Background color for the entire screen.
        backgroundColor: Color.fromARGB(255, 253, 225, 183),

        // App bar at the top of the screen.
        appBar: AppBar(
            title: Text("Friends List"),
            centerTitle: true,
        ),

        // Main content of the screen.
        body: Padding(
            // Padding around the content.
            padding: const EdgeInsets.symmetric(vertical: 0, horizontal: 20),

            // Allows the content to be scrollable.
            child: SingleChildScrollView(
                child: Container(
                    child: Column(
                        children: [
                            // Looping through each friend and rendering them.
                            for (final friend in friends)

                                // Fetching friend's data asynchronously.
                                FutureBuilder<DocumentSnapshot>(
                                    future: friend["friendRef"].get(),

                                    // Building the widget based on the fetched data's state.
                                    builder: (BuildContext context,
                                        AsyncSnapshot<DocumentSnapshot> snapshot) {
                                        if (snapshot.connectionState ==

```

```

        ConnectionState.waiting) {
    // Display a loading indicator until the data is
fetched.

    return CircularProgressIndicator();
}
if (snapshot.hasError) {
    // Display an error if one occurs.
    return Text('Error: ${snapshot.error}');
}
if (!snapshot.hasData) {
    // Display a message if no data is available.
    return Text('No data available');
}

// Extracting friend's data from the fetched snapshot.
Map<String, dynamic>? friendData =
    snapshot.data!.data() as Map<String, dynamic>?;

if (friendData == null) {
    // Display an error if the data format is invalid.
    return Text('Invalid data format');
}

// Extracting friend's details.
String friendImg = friendData["image"] as String;
String friendName = friendData["name"] as String;
String friendEmail = friendData["email"] as String;

return ListTile(
    // Displaying friend's image.
    leading: CircleAvatar(
        backgroundImage: NetworkImage(friendImg),
    ),
    // Displaying friend's name.
    title: Text(friendName),
    // Displaying friend's email.
    subtitle: Text(friendEmail),

    // Button to delete the friend.
    trailing: IconButton(
        icon: Icon(Icons.delete),
        onPressed: () {
            // Show a confirmation dialog before deleting.
            showDialog(
                context: context,
                builder: (context) => AlertDialog(
                    backgroundColor:
                        Color.fromRGBO(255, 255, 202, 123),

```

```

        title: Text("Are You Sure"),
        actions: [
            // Cancel button to dismiss the dialog.
            TextButton(
                child: Text('Cancel'),
                onPressed: () {
                    Navigator.of(context).pop();
                },
            ),
            // OK button to proceed with deletion.
            TextButton(
                child: Text('OK'),
                onPressed: () async {
                    await deleteFriend(friendData["uid"]);
                    setState(() {
                        friends.remove(friend);
                    });
                    Navigator.of(context).pop();
                },
            )
        ],
    ),
);
),
);
);
);
),
],
),
),
),
),
),
);
}
}

```

Important Widgets

createGroupDialog

```
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:flutter/material.dart';
import 'package:task_manage_app/models/group_controller.dart';
import 'package:task_manage_app/models/group_model.dart';
import 'package:task_manage_app/models/task_controller.dart';
import 'package:task_manage_app/models/user_controller.dart';
import 'package:task_manage_app/models/user_model.dart';
import 'package:task_manage_app/screens/friends_screen.dart';
import 'package:task_manage_app/widgets/friends_list.dart';
import 'package:task_manage_app/widgets/reusable_widgets.dart';

// A widget for creating a new group.
// It allows users to input group details like group name, friends in it and
decide the group mode.

// Define a StatefulWidget for the GroupCreationDialog.
class GroupCreationDialog extends StatefulWidget {
    UserModel user;
    GroupCreationDialog(this.user);
    @override
    _GroupCreationDialogState createState() => _GroupCreationDialogState();
}
// Define the state for the GroupCreationDialog widget.
class _GroupCreationDialogState extends State<GroupCreationDialog> {
    String? groupName;
    List<String> selectedFriends = [];
    List<String> selectedFriendsUid = [];
    Map<String, dynamic> friendsPoints = {};
    String? businessMode = "mode2";
    UserController _userController = UserController();
    GroupController _groupController = GroupController();
    TaskController _taskController = TaskController();
    // Function to handle friend selection
    void handleFriendSelection(List<String> friendNames, List<String> friendUid) {
        setState(() {
            selectedFriends = friendNames;
            selectedFriendsUid = friendUid;
            friendsPoints[widget.user.uid] = 0;
            for (final friendUid in selectedFriendsUid) {
                friendsPoints[friendUid] = 0;
            }
        })
    }
}
```

```

    });
}

// Function to handle form submission
Future<void> submitForm() async {
    if (selectedFriends.isEmpty) {
        setState(() {
            friendsPoints[widget.user.uid] = 0;
        });
    }
    if (groupName != null && groupName != "") {
        final docRef = FirebaseFirestore.instance.collection("groups").doc();
        _groupController
            .setGroup(
                docRef,
                groupName!,
                widget.user.uid,
                friendsPoints,
                "https://www.freepnglogos.com/uploads/crowd-png/crowd-people-png-
result-cliparts-for-22.png",
                businessMode!,
                widget.user.name)
            .then((value) async =>
                _userController.setGroupRef(widget.user.uid, docRef.id))
            .then((value) async =>
                {_taskController.setCreateGroupTask(docRef.id, widget.user.name)})
            .then((value) async => {
                for (final friendId in selectedFriendsUid)
                    {
                        _userController.setGroupRef(friendId, docRef.id),
                    }
            });
    // Close the dialog
    Navigator.of(context).pop();
} else {
    showDialog(
        context: context,
        builder: (context) =>
            reusable_AlertDialog('Group Name is not set', context));
}
}

@Override
Widget build(BuildContext context) {
    return Dialog(
        backgroundColor: Color.fromARGB(255, 255, 202, 123),
        child: Padding(
            padding: EdgeInsets.all(20.0),
            child: SizedBox(
                width: MediaQuery.of(context).size.width * 0.8,

```

```

child: SingleChildScrollView(
    child: Column(
        mainAxisSize: MainAxisSize.min,
        children: [
            TextFormField(
                decoration: InputDecoration(labelText: 'Group Name'),
                onChanged: (value) {
                    setState(() {
                        groupName = value;
                    });
                },
            ),
            SizedBox(height: 16),
            ElevatedButton(
                child: Text('Select Friends'),
                onPressed: () {
                    // Navigate to the friends screen page
                    Navigator.push(
                        context,
                        MaterialPageRoute(
                            builder: (context) => FriendsListPage(
                                currentUser: widget
                                    .user, // you need to pass currentUser object here
                                onFriendSelection: (result) {
                                    handleFriendSelection(
                                        result.friendNames, result.friendUids);
                                },
                                initialSelectedFriends: selectedFriends,
                                initialSelectedFriendsUid: selectedFriendsUid,
                            ),
                        ),
                    );
                },
            ),
            SizedBox(height: 16),
            if (selectedFriends.isNotEmpty)
                Container(
                    height: 100, // restrict the container height
                    child: ListView.builder(
                        itemCount: selectedFriends.length,
                        itemBuilder: (context, index) {
                            return Text(selectedFriends[index]);
                        },
                    ),
                ),
            SizedBox(height: 16),
            Text('Business Mode'),
            SwitchListTile(

```

```

        title: Text('Business Mode'),
        value: businessMode != null && businessMode == 'mode1',
        onChanged: (value) {
            setState(() {
                businessMode = value ? 'mode1' : "mode2";
            });
        },
    ),
    Row(
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
        children: [
            TextButton(
                child: Text(
                    'Cancel',
                    style: TextStyle(color: Colors.black),
                ),
                onPressed: () {
                    // Close the dialog without saving
                    Navigator.of(context).pop();
                },
            ),
            TextButton(
                child:
                    Text('Create', style: TextStyle(color: Colors.black)),
                onPressed: submitForm,
            ),
        ],
    ),
),
),
),
),
),
),
);
}
}

```

createTaskDialog

```
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import 'package:task_manage_app/models/group_controller.dart';
import 'package:task_manage_app/models/group_model.dart';
import 'package:task_manage_app/models/notification_controller.dart';
import 'package:task_manage_app/models/task_controller.dart';
import 'package:task_manage_app/models/user_controller.dart';
import 'package:task_manage_app/models/user_model.dart';
import 'package:task_manage_app/models/task_status_eum.dart';
import 'package:task_manage_app/widgets/reusable_widgets.dart';
import 'package:cloud_functions/cloud_functions.dart';

// A widget for creating a new task within a group.
// It allows users to input task details like task info, date, time, location,
// and assignees.

class TaskCreation extends StatefulWidget {
    UserModel user; // The current user.
    GroupModel group; // The group in which the task will be created.
    Function(String) onUpdateTask; // Callback function to update the task.
    Function(Timestamp)
        onUpdateTaskTime; // Callback function to update task time.
    int seenTask; // Number of seen tasks.

    TaskCreation({
        required this.user,
        required this.group,
        required this.onUpdateTask,
        required this.onUpdateTaskTime,
        required this.seenTask,
    });

    @override
    State<TaskCreation> createState() => _TaskCreationState();
}

class _TaskCreationState extends State<TaskCreation> {
    TextEditingController _taskInfoController = TextEditingController();
    String _dropDownLocationValue = "None";
    String _dropDownFriendValue = "EveryOne";
    DateTime selectedDate = DateTime.now();
    TimeOfDay selectedTime = TimeOfDay(hour: 00, minute: 00);
    late String selectedTimeString;
```

```

TaskStatus taskStatus = TaskStatus.published;
Set<String> selectedFriends = {};
List<dynamic> selectedFriendsUid = [];
int? taskCount;
String _taskInfo = "";
List<String> nameArr = [];
Map<String, String> nameToUid = new Map<String, String>();
NotificationController _notificationController = NotificationController();

GroupController _groupController = GroupController();
UserController _userController = UserController();
TaskController _taskController = TaskController();

List<DropdownMenuItem<String>> itemsList = [
    DropdownMenuItem(
        child: Text("None"),
        value: "None",
    ),
    // Add more location options here...
];

void initState() {
    super.initState();
    makeNamesArray();
    _taskInfoController.addListener(_onChange);
    setState(() {
        selectedTimeString = selectedTime.toString().split("(")[1].split(")")[0];
    });
    _taskController.getTaskCount(widget.group.uid).then((count) {
        setState(() {
            taskCount = count;
        });
    });
}
}

void disposeControllers() {
    super.dispose();
}

void _onChange() {
    setState(() {
        _taskInfo = _taskInfoController.text;
    });
}

// Show a dialog to select multiple friends for the task.
Future<void> _showMultipleSelectionDialog() async {
    await showDialog(

```

```

context: context,
builder: (BuildContext context) {
    return StatefulBuilder(
        builder: (BuildContext context, StateSetter setStateDialog) {
            return AlertDialog(
                title: Text("Select Friends"),
                content: Container(
                    width: double.maxFinite,
                    child: ListView(
                        children: nameArr.map((name) {
                            return CheckboxListTile(
                                value: selectedFriends.contains(name),
                                onChanged: (bool? value) {
                                    setStateDialog(() {
                                        if (value == true) {
                                            selectedFriends.add(name);
                                            selectedFriendsUid.add(nameToUid[name]!);
                                        } else {
                                            selectedFriends.remove(name);
                                            selectedFriendsUid.remove(nameToUid[name]);
                                        }
                                    });
                                    setState(() {});
                                },
                                title: Text(name),
                            );
                        }).toList(),
                    ),
                ),
                actions: [
                    TextButton(
                        onPressed: () => Navigator.of(context).pop(),
                        child: Text("Done"),
                    )
                ],
            );
        },
    );
}
}

// Fetch user names and UIDs to populate the friend selection dialog.
Future<void> makeNamesArray() async {
    Future<Map<String, dynamic>> friendsPoints =
        _groupController.getFriendsList(widget.group.uid);
    Map<String, dynamic>? data = await friendsPoints;
    if (data != null) {

```

```

        await Future.forEach(data.keys, (friendUid) async {
            DocumentSnapshot<Map<String, dynamic>> userSnapshot =
                await _userController.getUserDocumentSnapshot(friendUid);
            String? friendName = userSnapshot.data()?[ "name"];
            if (friendName != null) {
                setState(() {
                    nameArr.add(friendName);
                    nameToUid[friendName] = friendUid;
                });
            }
        });
    }

    String _formatDate(DateTime date) {
        final formatter = DateFormat('dd.MM.yyyy');
        return formatter.format(date);
    }

    String _formatDateToTime(DateTime date) {
        final formatter = DateFormat('HH:mm');
        return formatter.format(date);
    }

    // Show a date picker to select the task's date.
    Future<void> _selectDate(BuildContext context) async {
        final DateTime? picked = await showDatePicker(
            context: context,
            initialDate: selectedDate,
            firstDate: DateTime(2023),
            lastDate: DateTime(2030),
        );
        if (picked != null && picked != selectedDate) {
            setState(() {
                selectedDate = picked;
            });
        }
    }

    // Show a time picker to select the task's time.
    Future<void> _selectTime(BuildContext context) async {
        final TimeOfDay? picked = await showTimePicker(
            context: context,
            initialTime: selectedTime,
        );
        if (picked != null && picked != selectedTime) {
            setState(() {
                selectedTime = picked;
                selectedTimeString =
            });
        }
    }
}

```

```

        selectedTime.toString().split("(")[1].split(")")[0];
    });
}
}

// Handle the location dropdown selection.
void DropdownLocationSelection(String? selectedValue) {
    if (selectedValue is String) {
        setState(() {
            _dropDownLocationValue = selectedValue;
        });
    }
}

@Override
Widget build(BuildContext context) {
    return Dialog(
        backgroundColor: Color.fromARGB(255, 255, 202, 123),
        child: Padding(
            padding: EdgeInsets.all(20.0),
            child: SizedBox(
                width: MediaQuery.of(context).size.width * 0.8,
                child: SingleChildScrollView(
                    child: Column(
                        mainAxisSize: MainAxisSize.min,
                        children: [
                            Text(
                                "Task Creation",
                                style: TextStyle(fontSize: 20),
                            ),
                            SizedBox(
                                height: 20,
                            ),
                            Text("Enter Task"),
                            TextField(
                                decoration: InputDecoration(
                                    hintText: "Task Info..",
                                    border: OutlineInputBorder(
                                        borderRadius: BorderRadius.circular(10))),
                                controller: _taskInfoController,
                            ),
                            SizedBox(
                                height: 25,
                            ),
                            Text("Enter Date"),
                            TextButton(
                                onPressed: () => _selectDate(context),
                                child: Text(

```

```

        _formatDate(selectedDate),
        style: TextStyle(fontSize: 17, color: Colors.black),
    ),
),
SizedBox(
    height: 25,
),
Text("Enter Time"),
TextButton(
    onPressed: () => _selectTime(context),
    child: Text(
        selectedTimeString,
        style: TextStyle(fontSize: 17, color: Colors.black),
    )),
SizedBox(
    height: 25,
),
Text("Enter Location Type"),
DropdownButton(
    items: itemsList,
    onChanged: DropdownLocationSelection,
    value: _dropDownLocationValue,
),
SizedBox(
    height: 25,
),
Text("Select who needs to do"),
TextButton(
    onPressed: _showMultipleSelectionDialog,
    child: Text(
        selectedFriends.isEmpty
            ? "Select Friends"
            : selectedFriends.join(", "),
        style: TextStyle(fontSize: 17, color: Colors.black),
    )),
),
SizedBox(
    height: 25,
),
Row(
    mainAxisAlignment: MainAxisAlignment.spaceBetween,
    children: [
        TextButton(
            child: Text(
                'Cancel',
                style:
                    TextStyle(fontSize: 17, color: Colors.black),
            ),
),

```

```

onPressed: () {
    Navigator.of(context).pop();
},
),
TextButton(
    child: Text(
        'Create',
        style:
            TextStyle(fontSize: 17, color: Colors.black),
),
onPressed: () async {
    if (!_taskInfo.isEmpty) {
        if (selectedFriends.isEmpty) {
            selectedFriends = {"all"};
            selectedFriendsUid =
                widget.group.friendsPoints.keys.toList();
        }
        try {
            _taskController
                .setTask(
                    widget.group.uid,
                    widget.user.uid,
                    _taskInfo,
                    selectedFriends,
                    _dropDownLocationValue,
                    selectedDate,
                    selectedTimeString,
                    taskStatus.toString(),
                    taskCount!,
                    selectedFriendsUid)
                .then((value) async {
                    _groupController.taskCreationUpdate(
                        widget.group.uid,
                        taskCount!,
                        widget.user.name);
                })
                .then((value) async {
                    widget.group.friendsPoints
                        .forEach((key, value) {
                            _groupController.updateCreationDate(
                                key, widget.group.uid);
                        });
                })
                .then((value) => {
                    _userController.updateCountSeen(
                        widget.user.uid,
                        widget.group.uid,
                        widget.seenTask + 1)
                });
        }
    }
}

```

```
        })
        .then((value) async => {
            await _userController
                .getToken(widget.user.uid)
                .then((value) {
                    String groupName =
                        widget.group.groupName;
                    widget.group.friendsPoints.keys
                        .forEach(
                            (friendsUid) async => {
                                if (friendsUid !=
                                    widget.user.uid)
                                {
                                    await
                            _userController
                                .getToken(
                                    friendsUi
d)
                                .then(
                                    (value) =>
{
    _no
tificationController.sendCreatedTaskNotification(value, widget.user.name,
_taskInfo, groupName)
})
}
})
})
});
widget.onUpdateTask(
    widget.user.name,
);
widget.onUpdateTaskTime(
    Timestamp.now(),
);
Navigator.of(context).pop();
} catch (ex) {
    print(ex);
}
} else {
    showDialog(
        context: context,
        builder: (context) => reusableAlertDialog(
            "Set Task Info", context));
}
},
),
```

```
        ],  
        ),  
        ],  
        ),  
        )));  
    }  
}
```

singleTask

```
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:flutter/material.dart';
import 'package:task_manage_app/models/check_list_controller.dart';
import 'package:task_manage_app/models/group_controller.dart';
import 'package:task_manage_app/models/group_model.dart';
import 'package:task_manage_app/models/task_controller.dart';
import 'package:task_manage_app/models/task_model.dart';
import 'package:task_manage_app/models/user_controller.dart';
import 'package:task_manage_app/models/user_model.dart';
import 'package:task_manage_app/models/task_status_eum.dart';

// A single message/task in the chat or task list within a group.
// It displays information about the task, such as the creator's name, deadline,
and task status.

class SingleMessage extends StatefulWidget {
    final GroupModel group; // The group to which the task belongs.
    final UserModel user; // The current user.
    final String message; // The task message or description.
    final bool isMe; // Indicates if the task was created by the current user.
    final ScrollController controller; // Controller for scrolling to this task.
    final TaskModel taskModel; // The task model containing task information.

    SingleMessage({
        required this.group,
        required this.user,
        required this.message,
        required this.isMe,
        required this.controller,
        required this.taskModel,
    });

    @override
    _SingleMessageState createState() => _SingleMessageState();
}

class _SingleMessageState extends State<SingleMessage> {
    GlobalKey key = GlobalKey();
    String? imageUrl;
    String creatorName = "You";
    GroupController _groupController = GroupController();
    TaskController _taskController = TaskController();
    CheckListController _checkListController = CheckListController();
    UserController _userController = UserController();
```

```

bool loading = false;
ValueNotifier<TaskStatus>? _taskStatusNotifier =
    ValueNotifier<TaskStatus>(TaskStatus.published);
Stream<DocumentSnapshot>? taskStream;
ValueNotifier<Key> _expansionTileKeyNotifier =
    ValueNotifier<Key>(UniqueKey());

// Get the task's current status.
void getTaskStatus() async {
    DocumentSnapshot doc = await _taskController.fetchTaskDocument(
        widget.group.uid, widget.taskModel.uid);
    if (doc.exists) {
        Map<String, dynamic> data = doc.data() as Map<String, dynamic>;
        String status = data["status"];

        if (status.contains("publish")) {
            _taskStatusNotifier = ValueNotifier<TaskStatus>(TaskStatus.published);
        }
        if (status.contains("pending")) {
            _taskStatusNotifier = ValueNotifier<TaskStatus>(TaskStatus.pending);
        }
        if (status.contains("completed")) {
            _taskStatusNotifier = ValueNotifier<TaskStatus>(TaskStatus.completed);
        }
    }
}

// Mark the task as done and trigger related actions.
Future<void> markAsDone() async {
    _checkListController.setCheckListNotification(
        widget.taskModel.creatorUid,
        widget.user.uid,
        widget.user.name,
        widget.taskModel.taskInfo,
        widget.group.uid,
        widget.group.groupName,
        widget.taskModel.uid,
    );
    setState(() {
        loading = true;
    });

    _taskController.updateToPending(
        widget.group.uid, widget.taskModel.uid, widget.user.uid);

    getTaskStatus();
}

```

```

// Simulate a network request
await Future.delayed(const Duration(seconds: 2));
if (mounted) {
  _taskStatusNotifier = ValueNotifier<TaskStatus>(TaskStatus.pending);
  setState(() {
    loading = false;
  });
}
}

// Format the task's deadline timestamp.
String makeTimeStamp() {
  return widget.taskModel.deadline
    .toDate()
    .toString()
    .split(" ")[0]
    .toString();
}

// Scroll to this task in the chat/task list.
void scrollToKey() {
  final keyContext = key.currentContext;
  if (keyContext == null) {
    print("keyContext is null");
    return;
  }
  if (keyContext != null) {
    // Get the render box of the current widget
    final box = keyContext.findRenderObject() as RenderBox;

    // Get the position of the widget relative to the top of the viewport
    final position = box.localToGlobal(Offset.zero).dy;
    // Consider the height of the app bar
    final double appBarHeight = AppBar().preferredSize.height;
    // Add some padding at the top
    widget.controller.animateTo(
      position - appBarHeight,
      duration: Duration(milliseconds: 300),
      curve: Curves.easeInOut,
    );
  }
}
}

// Initialize state and data when the widget is created.
@Override
void initState() {
  super.initState();
  taskStream = FirebaseFirestore.instance
    .collection("groups")

```

```

.doc(widget.group.uid)
.collection("tasks")
.doc(widget.taskModel.uid)
.snapshots();
getImageUrl();
getTaskStatus();
getCreatorName();
}
// Get the image URL of the task creator.
void getImageUrl() async {
String url = await _userController.getImageUrl(widget.taskModel.creatorUid);
if (mounted) {
setState(() {
imageUrl = url;
});
}
}
// Get the name of the task creator.
void getCreatorName() async {
String name =
await _userController.getUserName(widget.taskModel.creatorUid);

if (mounted) {
setState(() {
creatorName = name;
});
}
}

@Override
Widget build(BuildContext context) {
getImageUrl();
getTaskStatus();
getCreatorName();
return StreamBuilder<DocumentSnapshot>(
stream: taskStream,
builder: (context, snapshot) {
if (!snapshot.hasData) return SizedBox();

if (_taskStatusNotifier.toString().contains("completed")) {
return Card(
color: Color.fromARGB(255, 255, 202,
123), // You can customize the card's appearance
child: ListTile(
leading: imageUrl != null ? Image.network(imageUrl!) : null,
title: Text(widget.message),
subtitle: widget.group.creatorUid == widget.user.uid
? Text("You")

```

```

        : Text(creatorName),
trailing: Icon(
    Icons.check,
    size: 30,
    color: Colors.teal,
),
));
} else {
    return Card(
        color: Color.fromARGB(255, 255, 202, 123),
        child: ValueListenableBuilder<TaskStatus>(
            valueListenable: _taskStatusNotifier!,
            builder: (context, taskStatusValue, child) {
                return ValueListenableBuilder<Key>(
                    valueListenable: _expansionTileKeyNotifier,
                    builder: (context, key, _) {
                        return ExpansionTile(
                            key: key,
                            onExpansionChanged: (flag) {
                                if (flag) {
                                    Future.delayed(Duration(seconds: 1), scrollToKey);
                                }
                            },
                            leading:
                                imageUrl != null ? Image.network(imageUrl!) : null,
                            title: Text(widget.message,
                                style: TextStyle(color: Colors.black)),
                            subtitle: creatorName != null
                                ? creatorName == widget.user.name
                                    ? Text(
                                        "You",
                                        style: TextStyle(color: Colors.black),
                                    )
                                    : Text(creatorName,
                                        style: TextStyle(color: Colors.black))
                                : null,
                            children: <Widget>[
                                ListTile(
                                    title: Text('Description:'),
                                    subtitle: Text(widget.taskModel.taskInfo),
                                ),
                                ListTile(
                                    title: Text('Deadline:'),
                                    subtitle: Text(makeTimeStamp() +
                                        " at " +
                                        widget.taskModel.deadlineTime),
                                ),
                                ListTile(

```

```

        title: Text('Assigned To:'),
        subtitle: Text(widget.taskModel.assignedTo),
    ),
    ListTile(
        title: Text('Location:'),
        subtitle: Text(widget.taskModel.location),
    ),
    Container(
        padding: EdgeInsets.symmetric(horizontal: 10),
        child: Row(
            mainAxisAlignment: MainAxisAlignment
                .spaceBetween, // For space between the buttons
            children: [
                widget.group.creatorUid == widget.user.uid
                    ? IconButton(
                        icon: Icon(
                            Icons.delete,
                        ),
                        onPressed: () async {
                            _expansionTileKeyNotifier.value =
                                UniqueKey();
                            await Future.delayed(
                                Duration(milliseconds: 250));
                            _taskController.deleteTask(
                                widget.group.uid,
                                widget.taskModel.uid);
                        },
                    )
                : SizedBox(
                    width: 10,
                ),
            ],
        ),
        // Mark as done button
        ElevatedButton(
            style: ElevatedButton.styleFrom(
                backgroundColor: Colors.teal,
            ),
            child: _taskStatusNotifier!.value
                .toString()
                .contains("published")
                    ? (loading
                        ? SizedBox(
                            height: 20,
                            width: 20,
                            child: CircularProgressIndicator(
                                color: Colors.white,
                                strokeWidth: 2,
                            ),
                        ),

```

```
        )
        : Text('Mark as done'))
    : Text('Pending..'),
onPressed: _taskStatusNotifier!.value
        .toString()
        .contains("published")
    ? widget.taskModel.assignedToUid
        .toString()
        .contains(widget.user.uid)
    ? markAsDone
    : null
: null,
),
],
),
),
),
],
);
})
},
),
);
}
},
);
}
}
}
```

friendsListPage

```
import "package:cloud_firestore/cloud_firestore.dart";
import "package:flutter/material.dart";
import "package:task_manage_app/models/group_controller.dart";
import "package:task_manage_app/models/group_model.dart";
import "package:task_manage_app/models/user_controller.dart";
import "package:task_manage_app/models/user_model.dart";
import 'package:task_manage_app/models/friends_selection.dart';
// This widget displays a list of friends for selection for group creation and
group options.
class FriendsListPage extends StatefulWidget {
  UserModel currentUser; // The current user.
  final ValueChanged<FriendSelectionResult>
      onFriendSelection; // Callback to handle friend selection.
  final List<String>
      initialSelectedFriends; // Initial list of selected friend names.
  final List<String>
      initialSelectedFriendsUid; // Initial list of selected friend UIDs.
  GroupModel? group; // The group to which friends belong.

  FriendsListPage({
    required this.currentUser,
    required this.onFriendSelection,
    this.group,
    this.initialSelectedFriends = const [],
    this.initialSelectedFriendsUid = const [],
  });

  @override
  State<FriendsListPage> createState() => _FriendsListPageState();
}

class _FriendsListPageState extends State<FriendsListPage> {
  List<Map<String, dynamic>> friends = [];
  bool isLoading = true;
  List<String> selectedFriendsNames = [];
  List<String> selectedFriendsUid = [];

  UserController _userController = UserController();
  GroupController _groupController = GroupController();

  @override
  void initState() {
    super.initState();
    selectedFriendsNames = widget.initialSelectedFriends;
    selectedFriendsUid = widget.initialSelectedFriendsUid;
```

```

    fetchFriends();
}

// Toggle friend selection based on friend name and UID.
void toggleFriendSelection(String friendName, String friendUid) {
    setState(() {
        if (selectedFriendsNames.contains(friendName)) {
            selectedFriendsNames.remove(friendName);
        } else {
            selectedFriendsNames.add(friendName);
        }
        if (selectedFriendsUid.contains(friendUid)) {
            selectedFriendsUid.remove(friendUid);
        } else {
            selectedFriendsUid.add(friendUid);
        }
    });
}

// Fetch the list of friends based on the user and group.
Future<void> fetchFriends() async {
    if (widget.group != null) {
        try {
            QuerySnapshot querySnapshot = await _userController
                .getFriendQuerySnapshot(widget.currentUser.uid);

            DocumentSnapshot groupSnapshot =
                await _groupController.getGroupByUid(widget.group!.uid);

            List<String> groupFriendsUid = List<String>.from(
                (groupSnapshot.data() as Map<String, dynamic>)["friendsPoints"]
                    .keys);

            if (querySnapshot.docs.isNotEmpty) {
                setState(() {
                    friends = querySnapshot.docs
                        .map((doc) => doc.data() as Map<String, dynamic>)
                        .toList();
                    for (var i = 0; i < groupFriendsUid.length; i++) {
                        friends = friends
                            .where((friend) => !friend["friendRef"])
                            .toString()
                            .contains(groupFriendsUid[i]))
                            .toList();
                    }
                    isLoading = false;
                });
            } else {

```

```

        setState(() {
            isLoading = false;
        });
    }
} catch (error) {
    // Handle error
    print("Error fetching friends: $error");
    setState(() {
        isLoading = false;
    });
}
} else {
try {
    QuerySnapshot querySnapshot = await _userController
        .getFriendQuerySnapshot(widget.currentUser.uid);

    if (querySnapshot.docs.isNotEmpty) {
        setState(() {
            friends = querySnapshot.docs
                .map((doc) => doc.data() as Map<String, dynamic>)
                .toList();
            isLoading = false;
        });
    } else {
        setState(() {
            isLoading = false;
        });
    }
} catch (error) {
    // Handle error
    print("Error fetching friends: $error");
    setState(() {
        isLoading = false;
    });
}
}
}

@Override
Widget build(BuildContext context) {
return Scaffold(
    backgroundColor: Color.fromRGBO(255, 253, 225, 183),
    appBar: AppBar(
        title: Text("Friends List"),
        centerTitle: true,
    ),
    body: (friends.isEmpty)
        ? Center(child: Text("No User Found"))

```

```

: Padding(
    padding: const EdgeInsets.symmetric(vertical: 0, horizontal: 20),
    child: ListView.builder(
        itemCount: friends.length,
        itemBuilder: (context, index) {
            return FutureBuilder<DocumentSnapshot>(
                future: friends[index]["friendRef"].get(),
                builder: (BuildContext context,
                    AsyncSnapshot<DocumentSnapshot> snapshot) {
                    if (snapshot.connectionState ==
                        ConnectionState.waiting) {
                        return ListTile(
                            leading: CircleAvatar(),
                            title: Text('Loading...'),
                            subtitle: Text(''),
                        );
                    }
                    if (snapshot.hasError) {
                        return ListTile(
                            leading: CircleAvatar(),
                            title: Text('Error: ${snapshot.error}'),
                            subtitle: Text(''),
                        );
                    }
                    if (!snapshot.hasData) {
                        return ListTile(
                            leading: CircleAvatar(),
                            title: Text('No data available'),
                            subtitle: Text(''),
                        );
                    }
                }
            }

            Map<String, dynamic>? friendData =
                snapshot.data!.data() as Map<String, dynamic>?;

            if (friendData == null) {
                return ListTile(
                    leading: CircleAvatar(),
                    title: Text('Invalid data format'),
                    subtitle: Text(''),
                );
            }

            String friendImg = friendData["image"] as String;
            String friendName = friendData["name"] as String;
            String friendEmail = friendData["email"] as String;
            String friendUid = friendData["uid"] as String;

```

```

        return ListTile(
            leading: CircleAvatar(
                backgroundImage: NetworkImage(friendImg),
            ),
            title: Text(friendName),
            subtitle: Text(friendEmail),
            trailing: Checkbox(
                value: selectedFriendsNames.contains(friendName),
                onChanged: (value) {
                    toggleFriendSelection(friendName, friendUid);
                },
            ),
        );
    });
},
floatingActionButton: FloatingActionButton(
    child: Icon(Icons.done),
    onPressed: () {
        // Callback to handle friend selection
        widget.onFriendSelection(
            FriendSelectionResult(selectedFriendsNames, selectedFriendsUid));
        Navigator.of(context).pop();
    },
),
);
}
}
}

```

friendsListView

```

import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:flutter/material.dart';
import 'package:task_manage_app/models/group_controller.dart';
import 'package:task_manage_app/models/group_model.dart';
import 'package:task_manage_app/models/task_controller.dart';
import 'package:task_manage_app/models/user_controller.dart';
import 'package:task_manage_app/models/user_model.dart';
import 'package:task_manage_app/widgets/reusable_widgets.dart';

// This widget displays a list of friends within a group, including their
// information and options for adding or removing friends.
class FriendsListView extends StatefulWidget {

```

```

final List<String> friends; // List of friend UIDs in the group.
final Map<String, int>
    friendsPoints; // A map of friend UIDs and their corresponding points.
final UserModel user; // Current user.
final GroupModel group; // The group to which friends belong.

FriendsListView(this.friends, this.friendsPoints, this.user, this.group);

@Override
_FriendsListViewState createState() => _FriendsListViewState();
}

class _FriendsListViewState extends State<FriendsListView> {
UserController _userController = UserController();
GroupController _groupController = GroupController();
TaskController _taskController = TaskController();

@Override
Widget build(BuildContext context) {
    return ListView.builder(
        shrinkWrap: true,
        physics: NeverScrollableScrollPhysics(),
        itemCount: widget.friends.length,
        itemBuilder: (context, index) {
            return StreamBuilder<DocumentSnapshot>(
                stream: _userController.getUserStream(widget.friends[index]),
                builder:
                    (BuildContext context, AsyncSnapshot<DocumentSnapshot> snapshot) {
                if (snapshot.connectionState == ConnectionState.waiting) {
                    return CircularProgressIndicator();
                } else if (snapshot.hasError) {
                    return Text("Error: ${snapshot.error}");
                } else {
                    Map<String, dynamic>? data =
                        snapshot.data!.data() as Map<String, dynamic>?;
                    // Extracting user data from the snapshot
                    String? uid = data?["uid"];
                    return ListTile(
                        leading: CircleAvatar(
                            backgroundImage: NetworkImage(data?["image"] ?? ''),
                        ),
                        title: Column(
                            mainAxisAlignment: MainAxisAlignment.start,
                            crossAxisAlignment: CrossAxisAlignment.start,
                            children: [
                                Visibility(

```

```

        visible: uid == widget.group.creatorUid,
        child: Text(
            "Group Owner",
            style: TextStyle(color: Colors.red),
        )),  

        Text(data?["name"] ?? ''),
    ],
),  

subtitle: Text(data?["email"] ?? ''),
trailing: Row(
    mainAxisAlignment: MainAxisAlignment.end,
    children: [
        Text(
            widget.friendsPoints[uid ?? ""].toString(),
            style: TextStyle(fontSize: 20),
        ),
        Visibility(
            visible: !widget.friends.contains(uid),
            child: IconButton(
                icon: const Icon(Icons.add),
                onPressed: () {
                    _userController.setFriend(widget.user.uid, uid!);
                    showDialog(
                        context: context,
                        builder: (context) => reusable_AlertDialog(
                            "Friend Added", context));
                    widget.friends.add(uid);
                },
            ),
        ),
        Visibility(
            visible: widget.user.uid == widget.group.creatorUid &&
                uid != widget.group.creatorUid,
            child: IconButton(
                icon: const Icon(Icons.delete),
                onPressed: () async {
                    setState(() {
                        widget.friends.remove(uid);
                        widget.friendsPoints.remove(uid);
                    });
                    _groupController.updateFriendsPoints(
                        widget.group.uid, widget.friendsPoints);
                    _taskController.setRemovedFromGroupTask(
                        widget.group.uid, data?["name"]);
                    _userController.deleteGroup(uid!, widget.group.uid);
                },
            ),
        ),
    ],
),

```

```
)  
],  
)  
};  
},  
);  
}  
};  
}  
}
```

reusableWidgets

```
import 'package:flutter/material.dart';

// This file contains several reusable widgets and utility functions for creating
various UI elements in the app.

// Reusable text field widget with customizable parameters.
TextField reusableTextField(String text, IconData icon, bool isPasswordType,
    TextEditingController controller) {
    return TextField(
        controller: controller,
        obscureText: isPasswordType,
        enableSuggestions: !isPasswordType,
        autocorrect: !isPasswordType,
        cursorColor: Colors.black,
        style: TextStyle(color: Colors.black.withOpacity(0.9)),
        decoration: InputDecoration(
            prefixIcon: Icon(
                icon,
                color: Colors.black,
            ),
            labelText: text,
            labelStyle: TextStyle(color: Colors.black.withOpacity(0.9)),
            filled: true,
            floatingLabelBehavior: FloatingLabelBehavior.never,
            fillColor: Colors.white70.withOpacity(0.3),
            border: OutlineInputBorder(
                borderRadius: BorderRadius.circular(30.0),
                borderSide: const BorderSide(width: 0, style: BorderStyle.none),
            ),
        ),
        keyboardType: isPasswordType
            ? TextInputType.visiblePassword
            : TextInputType.emailAddress,
    );
}

// Reusable elevated button widget with customizable parameters.
ElevatedButton reusable_button(String text, Function f) {
    return ElevatedButton(
        onPressed: () {
            f();
        },
        child: Row(
```

```

        mainAxisAlignment: MainAxisAlignment.center,
        children: [
            Text(
                text,
                style: TextStyle(fontSize: 20, color: Colors.white),
            )
        ],
    ),
    style: ButtonStyle(
        backgroundColor: MaterialStateProperty.all(Colors.black),
        padding: MaterialStateProperty.all(EdgeInsets.symmetric(vertical: 20)),
    ),
);
}

// Reusable dialog for resetting password.
AlertDialog resetPasswordDialog(Future<void> Function(String email) f,
    BuildContext context, TextEditingController controller) {
    return AlertDialog(
        backgroundColor: Color.fromARGB(255, 255, 202, 123),
        title: Text("Forgot password.."),
        actions: [
            reusableTextField("Enter Email", Icons.email_outlined, false, controller),
            Row(
                children: [
                    TextButton(
                        child: Text('Cancel'),
                        onPressed: () {
                            Navigator.of(context).pop();
                        },
                    ),
                    TextButton(
                        child: Text('Reset'),
                        onPressed: () async {
                            await f(controller.text).then((value) {
                                Navigator.of(context).pop();
                            });
                        },
                    ),
                ],
            ),
        ],
    );
}

// Reusable alert dialog with a single "OK" button.
AlertDialog reusable_AlertDialog(String str, BuildContext context) {
    return AlertDialog(

```

```

backgroundColor: Color.fromARGB(255, 255, 202, 123),
title: Text(str),
actions: [
    TextButton(
        child: Text('OK'),
        onPressed: () {
            Navigator.of(context).pop();
        },
    ),
],
);
}

// Reusable future builder for building a text field based on a future value.
FutureBuilder<String> reusable_StringBuilder(
    Future<String> value, IconData icon, TextEditingController controller) {
return FutureBuilder<String>(
    future: value,
    builder: (context, snapshot) {
        if (snapshot.hasData) {
            return reusableTextField(
                snapshot.data.toString(), icon, false, controller);
        } else if (snapshot.hasError) {
            return Text(snapshot.error.toString());
        } else {
            return Text("Loading...");
        }
    },
);
}

// Reusable future builder for loading an image based on a future URL.
FutureBuilder<String> reusableImageBuilder(Future<String> getImageUrl) {
return FutureBuilder<String>(
    future: getImageUrl,
    builder: (BuildContext context, AsyncSnapshot<String> snapshot) {
        if (snapshot.connectionState == ConnectionState.waiting) {
            // While the future is loading, display a loading indicator.
            return CircularProgressIndicator();
        } else if (snapshot.hasError) {
            return Text('Error loading image');
        } else {
            final imageUrl = snapshot.data.toString();
            return Container(
                decoration: BoxDecoration(
                    border: Border.all(color: Colors.black, width: 0.4),
                ),
                child: Image.network(imageUrl),
            );
        }
    },
);
}

```

```
    height: 80,  
    width: 80,  
    );  
  },  
  );  
}
```

Additional Functions

functions

```
// This Firebase Cloud Functions script sends a Firebase Cloud Messaging (FCM)
notification
// to a specified device token using the Firebase Admin SDK.

const functions = require("firebase-functions");
const request = require("request");

// Firebase Server Key
const SERVER_KEY =
  "AAAATD1N2Hs:APA91bE_FWaOEPvCF_SZeglbeNSLbWmNyFnEfTkr_QBpqZTSRQjFWGP6iJ-
tNh4n1D297k2QHm5dITAIfaImTLrQXVuH3iNBi9P57INrHcgweqhXUBrA8o6rjgysbWFnPk8ZEfcWldJ
";

// Define the Firebase Cloud Function that sends FCM notifications
exports.sendFCMNotification = functions.https.onCall((data, context) => {
  // Extract data from the request
  const toToken = data.toToken; // Device token to send the notification to
  const title = data.title; // Notification title
  const body = data.body; // Notification body

  // Define the HTTP request options
  const options = {
    uri: "https://fcm.googleapis.com/fcm/send",
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      Authorization: "key=" + SERVER_KEY, // Include your server key here
    },
    json: {
      to: toToken, // Target device token
      notification: {
        title: title, // Notification title
        body: body, // Notification body
      },
      data: {
        // You can add custom data here if needed
      },
    },
  },
};

// Send the FCM notification
```

```
return new Promise((resolve, reject) => {
  request(options, (error, response, body) => {
    if (error) {
      console.error("Error sending notification:", error);
      reject(error);
    } else {
      console.log("Notification sent:", body);
      resolve(body);
    }
  });
});
});
```

סיכום ומסקנות

- לסיכום הייתי אומר שלמדתי המון מהפרויקט הנ"ל ואף עברתי את כל הцеיפיות שלי מעצמי, למדתי שפה חדשה, נאלצתי לבצע הכל בעצמי ולמדתי עוד המון דברים מבחינית יישום העבודה שלמדנו לאורך תקופה הלימודים בפועל ולבסוף לייצר פרויקט רציני.