# מבוא

## רקע לפרויקט:

הפרויקט שלי נקרא PayLink, והוא עוסק בפיתוח אפליקציית מובייל פשוטה ונוחה לניהול העברות ובקשות כספיות בין משתמשים. האפליקציה מאפשרת לכל משתמש לשלוח בקשת תשלום או לבצע העברה, לעקוב אחרי היסטוריית הפעולות שביצע, ולאשר או לדחות בקשות שהוא קיבל ממשתמשים אחרים. בנוסף, היא כוללת תמיכה במספר מטבעות – שקל, דולר ויורו – תוך שימוש ב־API חיצוני להמרת מטבעות בזמן אמת, וכן ממשק נוח ועדכני להצגת היתרה. קהל היעד של האפליקציה הוא בעיקר משתמשים פרטיים – צעירים, סטודנטים, בעלי עסקים קטנים וכל מי שמבצע תשלומים באופן יומיומי מול גורמים פרטיים. בחרתי לפתח אפליקציה כזו כיוון שהיא משלבת בין נושאים שמעניינים אותי אישית – חוויית משתמש, אבטחת מידע, עבודה עם מסדי נתונים בזמן אמת והתחברות ל־API חיצוניים – ובמקביל נותנת מענה לצורך ממשי ורלוונטי בעולם האמיתי.

## תהליך המחקר

בתהליך המחקר המקדים לקראת פיתוח PayLink, ביצעתי סקירה מעמיקה של אפליקציות דומות הקיימות בשוק כמו Bit, PayBox ו־Venmo, במטרה להבין את המאפיינים המרכזיים שהמשתמשים מצפים להם מאפליקציה להעברות כספים. מצאתי כי רוב האפליקציות בשוק מתמקדות בפשטות ובמהירות בביצוע פעולות, אך לעיתים מתעלמות מהיבטי שקיפות וניהול היסטוריה ברמה גבוהה, במיוחד כשמדובר בבקשות שממתינות לאישור. בהתאם לכך, חקרתי את תחום ניהול הכספים הדיגיטלי ולמדתי כיצד לבנות ממשק שיאפשר למשתמש לא רק להעביר כסף, אלא גם לעקוב אחר כל הפעולות שבוצעו, לאשר או לדחות בקשות שממתינות, ולבחור את סוג המטבע הרלוונטי עבור כל פעולה.

## טכנולוגיות נוספות

במהלך הפיתוח שילבתי מספר טכנולוגיות מתקדמות שלא נלמדות במסגרת תוכנית הלימודים, והן אלו שהפכו את PayLink לאפליקציה מקצועית, חכמה ואינטואיטיבית יותר.

REST API- זוהי שיטת תקשורת בין השרת לאפליקציה, שמאפשרת שליפת נתונים ממקורות חיצוניים. בפרויקט השתמשתי ב־REST API לצורך המרת מטבעות דרך אתר exchangerate.host. כל העברה שמבוצעת לא בשקלים, נשלחת דרך קריאה ל־API שמחזיר את השער המעודכן, ולפיו מתבצע החישוב בפועל.

HttpClient- זוהי ספריית תקשורת שמאפשרת לשלוח בקשות HTTP ולקבל תגובות. השתמשתי בה לביצוע קריאות GET אל שרתי המרת המטבע, ולשליפת נתונים בפורמט JSON. HttpClient אינה נלמדת במסגרת הלימודים ולכן נדרש ממני ללמוד אותה עצמאית, כולל ניהול שגיאות וקריאות אסינכרוניות.

JSON ו־JObject- JSON הוא פורמט הנתונים שבו מתקבלים הנתונים מה־API. באמצעות הספרייה Newtonsoft.Json נעשה שימוש ב־JObject.Parse כדי לפענח את התגובה שהתקבלה מהשרת ולשלוף ממנה את הנתון הדרוש – שער ההמרה. השימוש בטכנולוגיה זו דרש ממני הבנה של מבני נתונים ופרשנות מדויקת.

בחירה בין מטבעות – Spinner- הוספתי לאפליקציה רכיב Spinner (רשימה נפתחת) אשר מאפשר למשתמש לבחור את סוג המטבע לביצוע ההעברה – שקל, דולר או אירו. הבחירה משפיעה הן על סמל המטבע המוצג לצד הסכום והן על החישוב האוטומטי של הסכום המומר, מה שמייעל את התהליך ומפשט את השימוש.

התאמות בממשק המשתמש (UI)- ביצעתי התאמות עיצוביות כדי לשפר את חוויית המשתמש, כמו הצגת סמל המטבע בהתאם לבחירה, הגבלת שדות קלט (למשל, לשם הזנת מספרים בלבד בשדות של סכום ומספר טלפון), ועדכון רשימת ההעברות במסך הראשי לאחר חזרה ממסך ההעברה.

ניהול חריגות (Exception Handling)- על מנת להבטיח יציבות ותגובה נכונה של האפליקציה, הוספתי מנגנונים לניהול חריגות באמצעות try-catch. כך למשל, האפליקציה מטפלת במצבים שבהם הקלט אינו תקין או כאשר מתרחשת שגיאה בעת חיבור לשרת, תוך הצגת הודעות מתאימות למשתמש במקום קריסת המערכת.

## אתגרים מרכזיים:

במהלך הפיתוח, נדרשתי להתמודד עם מספר אתגרים טכנולוגיים, ובראשם העבודה עם API להמרת מטבעות. אחד הקשיים המרכזיים היה ההתממשקות עם שירות חיצוני מתוך סביבת Xamarin – סביבה שמבוססת על .NET אך בעלת מגבלות מסוימות, בעיקר בכל הקשור לשימוש ב־HttpClient ובתעודות אבטחה (SSL/TLS). הפתרון שנבחר כלל הגדרה מפורשת של פרוטוקול האבטחה (Tls12) והסתמכות על ספריית System.Net.Http תוך שימוש בגישה אסינכרונית לעיבוד התקשורת, כדי לשמור על תגובתיות האפליקציה.

בנוסף, תהליך ההמרה כלל עיבוד של תוצאות בפורמט JSON. לצורך כך נעשה שימוש ב־JObject מתוך ספריית Newtonsoft.Json – טכנולוגיה שאינה נלמדת כחלק מתכנית הלימודים אך מאפשרת גישה נוחה ונקודתית לפרטי מידע ממבנה JSON מורכב.

אתגר משמעותי נוסף היה ביצירת סנכרון חלק בין פעולות המשתמש לבין מסד הנתונים המרוחק – Firebase Realtime Database. מאחר והעברות כספים משנות את יתרת המשתמשים, נדרשתי לבנות מנגנון שמבצע עדכון מיידי ואמין במסד הנתונים, כך שהתוצאה תשתקף באופן ישיר בממשק המשתמש. לשם כך נעשה שימוש בגישה אסינכרונית ובקריאות חוזרות ל־Firebase לאחר כל פעולה, תוך הקפדה על ניהול נכון של זרימת הנתונים כדי למנוע מצבים של תצוגה לא עדכנית או טעויות בסכומים.

חידוש נוסף שבחרתי להכניס הוא העברת פונקציית עדכון ייעודית אל רכיב TransferAdapter, כך שכל פעולה שמתרחשת בתוך רשימת ההעברות – כמו אישור או דחייה של העברה – תוכל לגרום לעדכון מידי של המסך הראשי (MainPage) ובפרט של יתרת המשתמש. פתרון זה אפשר שמירה על ממשק ריאקטיבי ומעודכן, בלי לרענן ידנית את המסך או להחזיר נתונים מה־Adapter ישירות אל הפעילות הראשית.

# תיאור תחום הידע

## אובייקטים נחוצים

באפליקציית ניהול החשבון והעברות הכספים שבניתי, נעשה שימוש באובייקטים מרכזיים המאפיינים אפליקציות ניהול פיננסיות.

האובייקטים המרכזיים כוללים:

Account (חשבון): מייצג משתמש יחיד במערכת, וכולל מזהה משתמש (ID), שם, יתרה כספית (remainder), ורשימה של העברות שביצע או קיבל.

Transfer (העברה): מייצג פעולה פיננסית בין שני חשבונות. כולל מזהים של השולח והמקבל, סכום ההעברה, סוג המטבע, תאריך ביצוע וסטטוס (ממתינה/אושרה/נדחתה).

FirebaseManager: רכיב שמרכז את כל הפעולות מול מסד הנתונים Firebase. מאפשר קריאה, כתיבה ועדכון של נתונים בצורה אסינכרונית.

CurrencyConverter: מחלקה חיצונית שאחראית על המרת מטבעות באמצעות שימוש ב־REST API, המתחבר לשירות exchangerate.host.

TransferAdapter: מתאם שמציג את רשימת ההעברות ברשימה (ListView), ומאפשר גם עדכון של סטטוסי ההעברות במסך הראשי באמצעות קריאה חוזרת לפונקציית עדכון יתרה.

## סוגי נתונים

בהתאם לאובייקטים, נעשה שימוש בסוגי נתונים מגוונים:

Strings – לשמות, מזהים, תיאורים ומטבעות.

Doubles – לייצוג סכומים ויתרות.

Integers – עבור סכום קלט מהמשתמש לפני ההמרה.

Booleans – לייצוג סטטוס של העברה (למשל, אם אושרה או לא).

Dates / timestamps – עבור תאריכי העברות.

Dictioneries ((Dictionery<string,Transfer>– לאחסון העברות שבוצעו או ממתינות, עבור כל משתמש.

## ייצוג מידע

כדי לייעל את ניהול הנתונים ושמירתם, נעשה שימוש ב-רשימות (Lists) וב־מילונים (Dictionaries) לייצוג מבני נתונים גמישים, שניתן לעדכן בזמן אמת.

החשבון (Account) כולל רשימת העברות המיוצגת כמילון (Dictionary) כך שניתן לגשת לכל העברה באמצעות ה-id שלה (שהוא ה- key במילון). כך, מתאפשר בקלות להוסיף, לעדכן או להסיר העברות.

Firebase פועל במבנה דמוי JSON ומייצג את כל הנתונים במבנה היררכי של key-value. לכן היה נוח למפות את הנתונים לאובייקטים באמצעות מילונים, מחרוזות ואוספים.

המרת מטבעות מתבצעת באמצעות פרמטרים בכתובת URL, ולאחר מכן המידע שמתקבל (בפורמט JSON) מנותח באמצעות שימוש ב־JObject, מה שמאפשר גישה מהירה לערכים הנחוצים (כמו שער ההמרה).

## תיאור פעולות על המידע

המחלקה FirebaseManager אחראית על כל הפעולות הלוגיות על המידע מול מסד הנתונים המרוחק Firebase, כולל קריאה, כתיבה, עדכון ומחיקה של נתונים. להלן הפירוט:

1. יצירה והוספה (Create/Insert):

SaveAccountAsync(Account account) – פעולה זו שומרת חשבון חדש או מעדכנת קיים במסד הנתונים תחת הענף "accounts", תוך שימוש במזהה הייחודי של החשבון (Id).

SendTransferAsync(…) – פעולה זו יוצרת אובייקט העברה חדש מסוג Transfer, מקצה לו מזהה ייחודי, ומוסיפה את ההעברה לרשימת ההעברות של השולח וגם לזו של המקבל. בנוסף, הפעולה ממיינת את רשימות ההעברות לפי סטטוס ותאריך לשם ארגון ברור יותר.

2. קריאה (Read):

GetAccountAsync(string title) – פעולה שמחזירה אובייקט Account לפי מזהה משתמש, ומשמשת לטעינת פרטי משתמש ופעולותיו. כך מתבצע עדכון המידע באפליקציה/

CheckPassword(string id, string password) – פעולה לקריאת נתוני חשבון והשוואת סיסמה, לצורך אימות משתמש.

3. עדכון (Update):

UpdateAccountAsync(…) – פעולה לעדכון פרטי חשבון שלם, כולל העברות והתנהגות עסקית נוספת.

UpdateTransferAsync(…) – פעולה לעדכון פרטני של העברה ספציפית, למשל שינוי סטטוס או ערכים אחרים.

ApproveTransfer(…)– פעולה לוגית מורכבת שמבצעת מספר עדכונים: משנה את סטטוס ההעברה ל־approved, משנה את יתרות החשבון של השולח והמקבל בהתאם, שומרת את השינויים במסד הנתונים עבור שני המשתמשים.

RejectTransferAsync(…)– פעולה מקבילה לזו של אישור, אך משנה את הסטטוס ל־rejected, ללא שינוי יתרות.

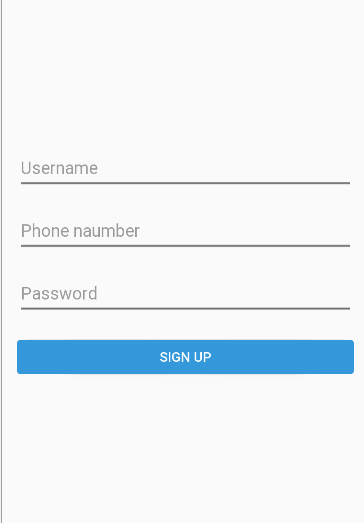
4. מחיקה (Delete):

DeleteAccountAsync(accountId) – מוחקת חשבון קיים ממסד הנתונים לפי מזהה. יש לשים לב שזו מחיקה מוחלטת, ולא רק הסתרה מהצגה.

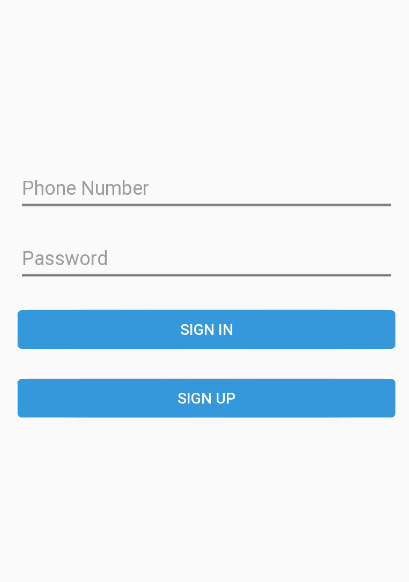
בנוסף קיימת הפעולה של המרת מטבעות GetCurrencyToShekelRateAsync אשר מקבלת כפרמטר את המטבע להמרה וכמות ממנו.

# ארכיטקטורת הפרויקט

## מסך הרשמה (Sign Up)



## מסך התחברות

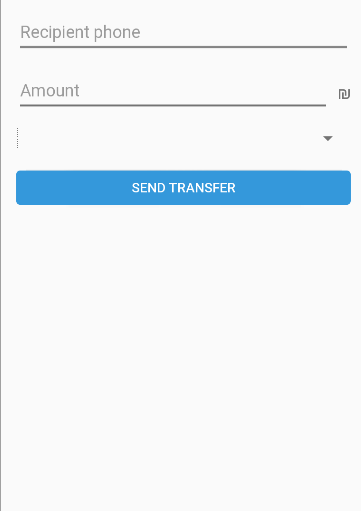


## מסך ראשי

תמונה שמכילה טקסט, צילום מסך, גופן

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

## מסך ביצוע העברה / בקשה להעברה



# מימוש הפרויקט

## ניהול הפרויקט

### Class FirebaseManager

**שם מחלקה: FirebaseManager**

**תפקיד המחלקה:**

ניהול כל התקשורת מול מסד הנתונים Firebase - שמירה, שליפה, עדכון ומחיקה של חשבונות והעברות. בנוסף, אחראית על לוגיקת האישור והדחייה של העברות.

**תכונות המחלקה:**

1. private readonly FirebaseClient \_firebaseClient

* **תחום הכרה:** פרטי.
* **תפקיד ושימוש:** אובייקט שמייצג את החיבור למסד הנתונים ב-Firebase. כל פעולה במסד מתבצעת דרכו.

**פעולות המחלקה:**

1. **FirebaseManager(string firebaseUrl)**

**תיאור:** פעולה בונה שיוצרת חיבור ל-Firebase בעזרת כתובת URL שנשלחת מהמשתמש.

1. **Task SaveAccountAsync(Account account)**

**תיאור:** שומר את החשבון שנשלח למסד תחת מזהה ייחודי.

1. **Task<Account> GetAccountAsync(string title)**

**תיאור:** מחפש ומחזיר חשבון לפי מזהה. אם לא נמצא, נזרקת שגיאה מסוגAccountNotFoundException

1. **Task<List<Account>> GetAccountsAsync()**

**תיאור:** מחזיר את כל החשבונות הקיימים במסד הנתונים.

1. **Task UpdateAccountAsync(string accountId, Account updatedAccount)**

**תיאור:** מעדכן את פרטי החשבון שזוהה לפי המזהה.

1. **Task DeleteAccountAsync(string accountId)  
   תיאור:** מוחק את החשבון שזוהה לפי המזהה ממסד הנתונים.
2. **Task SendTransferAsync(string fromId, string toId, double amount, bool isARequest)**

public async Task SendTransferAsync(string fromId, string toId, double amount, bool isARequest)

{

if (amount <= 0)

throw new InvalidTransferAmountException(amount);

var fromAcc = await GetAccountAsync(fromId);

var toAcc = await GetAccountAsync(toId);

if (fromAcc.Transfers == null)

throw new MissingTransfersException(fromId);

if (toAcc.Transfers == null)

throw new MissingTransfersException(toId);

string id = Guid.NewGuid().ToString();

var newTransfer = new Transfer(DateTime.Now.ToString("dd/MM/yyyy"), fromId, toId, amount, isARequest, Transfer.RequestStatus.waiting);

toAcc.Transfers[id] = newTransfer;

fromAcc.Transfers[id] = newTransfer;

toAcc.Transfers = toAcc.Transfers

.OrderBy(kv => kv.Value.Status)

.ThenBy(kv => DateTime.TryParse(kv.Value.Date, out var dt) ? dt : DateTime.MaxValue)

.ToDictionary(kv => kv.Key, kv => kv.Value);

fromAcc.Transfers = fromAcc.Transfers

.OrderBy(kv => kv.Value.Status)

.ThenBy(kv => DateTime.TryParse(kv.Value.Date, out var dt) ? dt : DateTime.MaxValue)

.ToDictionary(kv => kv.Key, kv => kv.Value);

await SaveAccountAsync(toAcc);

await SaveAccountAsync(fromAcc);

}

**תיאור:** יוצרת העברה חדשה בין שני חשבונות.

**פירוט לוגי:**

* + בודקת אם הסכום חיובי.
  + מוודאת שלשני החשבונות קיימת רשימת העברות.
  + יוצרת אובייקט Transfer עם מזהה חדש.
  + מוסיפה את ההעברה לשני החשבונות.
  + ממיינת את רשימות ההעברות לפי סטטוס ותאריך.
  + שומרת את החשבונות המעודכנים.

1. **Task<Transfer?> GetTransferAsync(string accountId, string transferId)**

**תיאור:** מחזיר את ההעברה לפי מזהה החשבון ומזהה ההעברה.

1. **Task UpdateTransferAsync(string accountId, string transferId, Transfer updatedTransfer)**

**תיאור:** מעדכן העברה במסד הנתונים.

1. **Task ApproveTransfer(string transferId, string destId)**

public async Task ApproveTransfer(string transferId, string destId)

{

var transfer = await GetTransferAsync(destId, transferId);

if (transfer.Status != Transfer.RequestStatus.waiting)

throw new InvalidTransferStatusException(transferId, transfer.Status);

Account dest = await GetAccountAsync(destId);

Account source = await GetAccountAsync(transfer.Source);

if (dest.Transfers == null)

throw new MissingTransfersException(destId);

if (source.Transfers == null)

throw new MissingTransfersException(transfer.Source);

int transferCoefficient = transfer.IsARequest ? -1 : 1;

dest.addAmmount(transfer.Amount \* transferCoefficient);

source.addAmmount(transfer.Amount \* transferCoefficient \* -1);

transfer.Status = Transfer.RequestStatus.approved;

dest.Transfers[transferId] = transfer;

source.Transfers[transferId] = transfer;

await UpdateAccountAsync(dest.Id, dest);

await UpdateAccountAsync(source.Id, source);

{

**תיאור:** מאשרת העברה. מעדכנת את הסטטוס ל־ approved ומבצעת שינויי יתרה בהתאם.

**לוגיקה:**

* + בדיקת סטטוס העברה.
  + חישוב מקדם לפי האם זו בקשה או העברה רגילה.
  + עדכון יתרות החשבונות.
  + שמירת המצב החדש במסד הנתונים.

1. **Task RejectTransferAsync(string transferId, string destId)**

public async Task RejectTransferAsync(string transferId, string destId)

}

var transfer = await GetTransferAsync(destId, transferId);

if (transfer.Status != Transfer.RequestStatus.waiting)

throw new InvalidTransferStatusException(transferId, transfer.Status);

Account dest = await GetAccountAsync(destId);

Account source = await GetAccountAsync(transfer.Source);

if (dest.Transfers == null)

throw new MissingTransfersException(destId);

if (source.Transfers == null)

throw new MissingTransfersException(transfer.Source);

transfer.Status = Transfer.RequestStatus.rejected;

dest.Transfers[transferId] = transfer;

source.Transfers[transferId] = transfer;

await UpdateAccountAsync(dest.Id, dest);

await UpdateAccountAsync(source.Id, source);

{

**תיאור:** דוחה העברה - מעדכן את הסטטוס ל־rejected ומעדכן את שני החשבונות.

**לוגיקה:**

* + בדיקת סטטוס העברה.
  + שמירת המצב החדש במסד הנתונים.

1. **Task<bool> CheckPassword(string id, string password)**

**תיאור:**  בודק האם הסיסמה שהוזנה תואמת לסיסמה השמורה בחשבון.

### Class TransferAdapter

**שם המחלקה:** TransferAdapter

**תפקיד המחלקה:** מתאם (Adapter) להצגת רשימת העברות/בקשות (Transfer) ברכיב ListView באנדרואיד. תומכת באישור או דחיית בקשות לפי מצב הטרנספר ומעדכנת את הנתונים בהתאם ב-Firebase.

**תכונות המחלקה:**

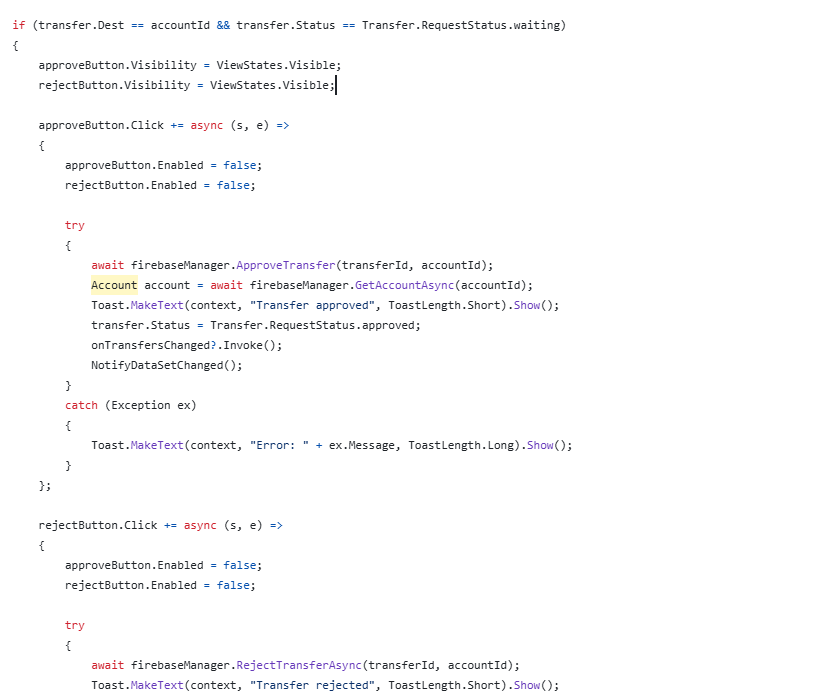
| **טיפוס** | **תחום הכרה** | **תפקיד ושימוש** | **שם השדה** |
| --- | --- | --- | --- |
| Activity | פרטי | פעילות הקונטקסט של האפליקציה. דרוש לבניית View. | context |
| List<KeyValuePair<string, Transfer>> | פרטי | רשימת ההעברות ממוינת לפי תאריך יורד. | transfers |
| string | פרטי | מזהה החשבון הנוכחי. משמש לאימות פעולות. | accountId |
| FirebaseManager | פרטי | אחראי על תקשורת עם מסד הנתונים ב-Firebase. | firebaseManager |
| Action | פרטי | פעולה שקורית כאשר רשימת ההעברות משתנה, כדי לעדכן את הממשק הגרפי. | onTransfersChanged |

**הסבר על פעולות המחלקה:**

תמונה שמכילה טקסט, צילום מסך, גופן

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

תמונה שמכילה טקסט, צילום מסך, גופן

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.



**תיאור:** מחזירה View מותאם לפריט ברשימת ההעברות. מוסיפה כפתורי אישור/דחייה אם הרלוונטי, ומבצעת את הפעולות המתאימות בפועל (כולל קריאות ל-firebase)

**פירוט לוגי:**

* **טעינת תצוגה מתאימה**  
  אם כבר קיימת תצוגה ממוחזרת (convertView), משתמשים בה.אחרת, מנפחים תצוגה חדשה מקובץ ה- XML transfer\_list\_item.
* **איתור רכיבי UI מתוך התצוגה:**  
  מאתרים את הטקסטים (כותרת + טקסט משני) ואת כפתורי האישור והדחייה בתוך ה- View.
* **שליפת נתוני ההעברה לפי מיקום:** לוקחים את הזוג (KeyValuePair) של ההעברה לפי המיקום המבוקש ברשימה. לאחר מכן, מזהים מזהה העברה (transferId) ואובייקט העברה (transfer) .
* **עדכון ה טקסטים:** בונים את הטקסט הראשי עם מקור, יעד ותאריך. בונים טקסט נוסף עם סוג ההעברה (בקשה או רגילה), כמות, סטטוס.
* **בדיקת תנאים להצגת כפתורי פעולה:** אם החשבון הנוכחי הוא יעד ההעברה **והסטטוס עדיין ממתין** –מאפשרים הצגה של כפתורי אישור ודחייה. אחרת – מחביאים את הכפתורים (כדי לא לאפשר פעולה על טרנספרים לא רלוונטיים).
* **הוספת התנהגות לכפתורים:** אישור בלחיצה, הכפתור מבצע קריאה לפעולת אישור העברה ב-Firebase. מעדכן את הסטטוס, מציג Toast ומרענן את הרשימה. דחייה- כנ"ל, אבל מבצע דחיית העברה.
* **החזרת ה-View המוכן לשורה זו ברשימה.**

### Class Account

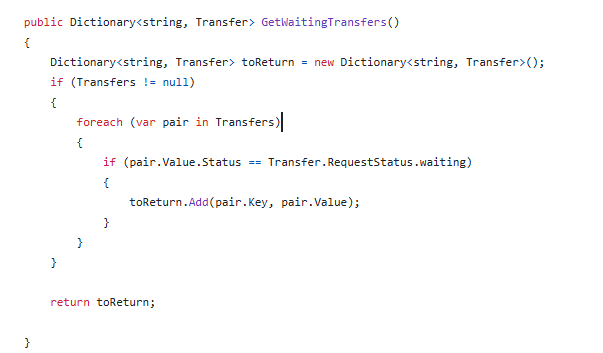
**תכונות המחלקה:**

|  |  |  |  |
| --- | --- | --- | --- |
| שם השדה | תפקיד ושימוש | תחום הכרה | טיפוס |
| id | מזהה של המשתמש. משמש לקישור בין העברות ולאיתור החשבון במסד נתונים. | פרטי | string |
| name | שם משתמש שבעזרתו האפליקציה פונה אליו. | פרטי | string |
| hashedPassword | סיסמה להגנת הפרטיות והגנה על המידע | פרטי | string |
| reminder | היתרה בחשבון | פרטי | string |

**תפקיד המחלקה:**ניהול ייצוג של חשבון משתמש במערכת הבנק. היא משמשת כיחידת מידע עיקרית בהתנהלות הלוגית של המערכת מול המשתמשים.

**פעולות במחלקה:**

* Account(string id, string name, string hashed\_password, double remainder = 5000)  
  פעולה בונה. מאתחלת את כל המאפיינים של החשבון עם ערכים שניתנו או ערך ברירת מחדל ליתרה.
* Account()  
  פעולה בונה ריקה המאפשרת יצירת מופע ללא ערכים התחלתיים (לשימוש בסריאליזציה למשל).
* ToString()  
  מחזירה מחרוזת הכוללת את שם החשבון, היתרה, ורשימת העברות בפורמט קריא.
* addAmmount(double amount)  
  מוסיפה סכום נתון ליתרת החשבון (Remainder).
* GetWaitingTransfers()  
  הפעולה מחזירה מילון חדש של כל ההעברות שהסטטוס שלהן waiting. (עוברת על כל ההעברות ובוחרת רק את אלה שממתינות לאישור)



## Class Account

**1. Class Name**

**Account**

**2. Role of the Class**

The Account class represents a bank account or financial account system in an Android application. It manages user account information, balance tracking, and money transfer operations with status management. This class serves as a core component for financial transaction management within the mobile application framework.

**3. Explanation of All Class Attributes**

**3.1 Scope and 3.2 Role and Use**

**Private Attributes:**

* string id - **Scope:** Private field - **Role:** Unique identifier for the account, ensuring each account can be distinctly identified within the system.
* string name - **Scope:** Private field - **Role:** Account holder's name, storing the personal identification of the account owner.
* string hashedPassword - **Scope:** Private field - **Role:** Stores the hashed password for security purposes, ensuring password protection without storing plain text credentials.
* double reminder - **Scope:** Private field - **Role:** Account balance/remaining amount (note: likely meant to be "remainder"), tracking the current financial balance of the account.

**Public Properties:**

* string Id - **Scope:** Public property - **Role:** Provides controlled access to the account ID with JSON serialization support for data persistence and API communication.
* string Name - **Scope:** Public property - **Role:** Provides external access to the account holder's name with appropriate encapsulation.
* string HashedPassword - **Scope:** Public property - **Role:** Provides controlled access to the hashed password with JSON serialization capabilities for secure data transmission.
* double Remainder - **Scope:** Public property - **Role:** Provides access to the account balance with JSON serialization support for financial data management.
* Dictionary<string, Transfer> Transfers - **Scope:** Public property - **Role:** Stores all transfer transactions associated with this account, keyed by transfer ID for efficient lookup and management.

**Static Attributes:**

No static attributes are present in this class.

**4. Brief Explanation of Actions in the Class**

**Main Logical Actions:**

**Constructor (Parameterized):**

public Account(string id, string name, string hashed\_password, double remainder = 5000)

{

Id = id;

Name = name;

HashedPassword = hashed\_password;

Remainder = remainder;

Transfers = new Dictionary<string, Transfer>();

}

**Purpose:** Initializes a new account with provided details and sets a default balance of 5000 if not specified. This constructor ensures proper initialization of all essential account properties and prepares the transfers collection for future use.

**addAmmount Method:**

public void addAmmount(double ammunnt)

{

Remainder += ammunnt;

}

**Purpose:** Adds a specified amount to the account balance. This method handles deposits or credits to the account, allowing for balance increases through various transaction types such as deposits, transfers, or interest payments.

**GetWaitingTransfers Method:**

public Dictionary<string, Transfer> GetWaitingTransfers()

{

Dictionary<string, Transfer> toReturn = new Dictionary<string, Transfer>();

if (Transfers != null)

{

foreach (var pair in Transfers)

{

if (pair.Value.Status == Transfer.RequestStatus.waiting)

{

toReturn.Add(pair.Key, pair.Value);

}

}

}

return toReturn;

}

**Purpose:** Filters and returns only the transfers that have a "waiting" status. This method is useful for displaying pending transactions that require user attention or administrative approval, enabling effective transaction status management.

**4.1 Flowchart for GetWaitingTransfers Method:**

**GetWaitingTransfers Method Flow:**

START → Create empty Dictionary → Check if Transfers ≠ null → [If No: Return empty Dictionary] → [If Yes: Loop through each transfer] → Check if Status == waiting → [If Yes: Add to return Dictionary] → [If No: Continue] → More transfers? → [If Yes: Continue loop] → [If No: Return populated Dictionary] → END

**ToString Method:**

public override string ToString()

{

string result = $"Name: {Name}\nRemainder: {Remainder}\n";

foreach (var kv in Transfers)

{

result += $"ID: {kv.Key}\n{kv.Value}\n";

}

return result;

}

**Purpose:** Provides a formatted string representation of the account, including name, balance, and all transfers. This method is essential for debugging purposes, logging, and providing human-readable account information for display or reporting purposes.

**5. Database/File Storage Description**

The Account class is designed for JSON serialization, as indicated by the JsonProperty attributes throughout the class. The data structure when saved would follow this format:

**JSON Structure:**

{

"Id": "string - unique account identifier",

"Name": "string - account holder name",

"HashedPassword": "string - encrypted password",

"Remainder": "double - current account balance",

"Transfers": {

"transferId1": {

// Transfer object properties

},

"transferId2": {

// Transfer object properties

}

}

}

**Storage Characteristics:**

* **Serialization Format:** All properties are configured with JsonProperty attributes to include null values in JSON serialization, ensuring complete data preservation.
* **Data Structure:** Transfers are stored as a nested dictionary structure, allowing for efficient key-based access and management of transaction records.
* **Constructor Support:** The class supports both parameterized and parameterless constructors for flexible instantiation, accommodating various object creation scenarios including deserialization.
* **Persistence Method:** Data persistence likely occurs through JSON serialization to local files within the Android application's storage or through remote API calls to backend services.
* **Data Integrity:** The use of hashed passwords and unique identifiers ensures data security and integrity within the storage system.

## Class FirebaseManager

**1. Class Name**

**FirebaseManager**

**2. Role of the Class**

The FirebaseManager class serves as a data access layer and service manager for interacting with Firebase Realtime Database in an Android banking/financial application. It provides comprehensive CRUD (Create, Read, Update, Delete) operations for Account entities and manages complex financial transactions including money transfers, transfer approvals, and rejections. This class abstracts all Firebase database operations and implements business logic for financial transaction processing.

**3. Explanation of All Class Attributes**

**3.1 Scope and 3.2 Role and Use**

**Private Attributes:**

* readonly FirebaseClient \_firebaseClient - **Scope:** Private readonly field - **Role:** Maintains the connection to Firebase Realtime Database. This field is marked as readonly to ensure immutability after initialization, providing a stable connection throughout the object's lifetime. It serves as the primary interface for all database operations.

**Static Attributes:**

No static attributes are present in this class.

**4. Brief Explanation of Actions in the Class**

**Constructor:**

public FirebaseManager(string firebaseUrl)

{

\_firebaseClient = new FirebaseClient(firebaseUrl);

}

**Purpose:** Initializes the FirebaseManager with a connection to the specified Firebase database URL, establishing the foundation for all subsequent database operations.

**Main Logical Actions:**

**SaveAccountAsync Method:**

public async Task SaveAccountAsync(Account account)

{

await \_firebaseClient

.Child("accounts")

.Child(account.Id)

.PutAsync(account);

}

**Purpose:** Saves or updates an account in the Firebase database. Uses the account's ID as the key in the "accounts" collection.

**GetAccountAsync Method:**

public async Task<Account> GetAccountAsync(string title)

{

var account = await \_firebaseClient.Child("accounts").Child(title).OnceSingleAsync<Account>();

if (account == null)

throw new AccountNotFoundException(title);

return account;

}

**Purpose:** Retrieves a specific account by ID from Firebase. Throws a custom exception if the account is not found, ensuring proper error handling.

**GetAccountsAsync Method:**

public async Task<List<Account>> GetAccountsAsync()

{

var accounts = await \_firebaseClient

.Child("accounts")

.OnceAsync<Account>();

return accounts.Select(a => a.Object).ToList();

}

**Purpose:** Retrieves all accounts from the Firebase database and returns them as a List of Account objects.

**SendTransferAsync Method (Complex Business Logic):**

public async Task SendTransferAsync(string fromId, string toId, double amount, bool isARequest)

{

if (amount <= 0)

throw new InvalidTransferAmountException(amount);

var fromAcc = await GetAccountAsync(fromId);

var toAcc = await GetAccountAsync(toId);

if (fromAcc.Transfers == null)

throw new MissingTransfersException(fromId);

if (toAcc.Transfers == null)

throw new MissingTransfersException(toId);

string id = Guid.NewGuid().ToString();

var newTransfer = new Transfer(DateTime.Now.ToString("dd/MM/yyyy"), fromId, toId, amount, isARequest, Transfer.RequestStatus.waiting);

toAcc.Transfers[id] = newTransfer;

fromAcc.Transfers[id] = newTransfer;

// Sort transfers by status and date

toAcc.Transfers = toAcc.Transfers

.OrderBy(kv => kv.Value.Status)

.ThenBy(kv => DateTime.TryParse(kv.Value.Date, out var dt) ? dt : DateTime.MaxValue)

.ToDictionary(kv => kv.Key, kv => kv.Value);

fromAcc.Transfers = fromAcc.Transfers

.OrderBy(kv => kv.Value.Status)

.ThenBy(kv => DateTime.TryParse(kv.Value.Date, out var dt) ? dt : DateTime.MaxValue)

.ToDictionary(kv => kv.Key, kv => kv.Value);

await SaveAccountAsync(toAcc);

await SaveAccountAsync(fromAcc);

}

**Purpose:** Creates and initiates a money transfer between two accounts. This method handles both regular transfers and transfer requests. It validates the amount, retrieves both accounts, creates a new transfer with waiting status, adds it to both accounts' transfer collections, sorts the transfers, and saves both accounts to the database.

**4.1 Flowchart for SendTransferAsync Method:**

**SendTransferAsync Method Flow:**

START → Validate amount > 0 → [If No: Throw InvalidTransferAmountException] → [If Yes: Get fromAccount] → Get toAccount → Check fromAcc.Transfers ≠ null → [If No: Throw MissingTransfersException] → [If Yes: Check toAcc.Transfers ≠ null] → [If No: Throw MissingTransfersException] → [If Yes: Generate new GUID] → Create new Transfer with waiting status → Add transfer to toAcc.Transfers → Add transfer to fromAcc.Transfers → Sort toAcc.Transfers by status and date → Sort fromAcc.Transfers by status and date → Save toAccount to Firebase → Save fromAccount to Firebase → END

**ApproveTransfer Method (Complex Business Logic):**

public async Task ApproveTransfer(string transferId, string destId)

{

var transfer = await GetTransferAsync(destId, transferId);

if (transfer.Status != Transfer.RequestStatus.waiting)

throw new InvalidTransferStatusException(transferId, transfer.Status);

Account dest = await GetAccountAsync(destId);

Account source = await GetAccountAsync(transfer.Source);

if (dest.Transfers == null)

throw new MissingTransfersException(destId);

if (source.Transfers == null)

throw new MissingTransfersException(transfer.Source);

int transferCoefficient = transfer.IsARequest ? -1 : 1;

dest.addAmmount(transfer.Amount \* transferCoefficient);

source.addAmmount(transfer.Amount \* transferCoefficient \* -1);

transfer.Status = Transfer.RequestStatus.approved;

dest.Transfers[transferId] = transfer;

source.Transfers[transferId] = transfer;

await UpdateAccountAsync(dest.Id, dest);

await UpdateAccountAsync(source.Id, source);

}

**Purpose:** Approves a pending transfer and executes the actual money movement. This method handles both regular transfers and transfer requests with different directional logic based on the transfer type. It validates the transfer status, retrieves both accounts, calculates the appropriate amounts using a coefficient system, updates account balances, changes transfer status to approved, and saves both accounts.

**4.1 Flowchart for ApproveTransfer Method:**

**ApproveTransfer Method Flow:**

START → Get Transfer by transferId and destId → Check if transfer.Status == waiting → [If No: Throw InvalidTransferStatusException] → [If Yes: Get destination account] → Get source account → Check dest.Transfers ≠ null → [If No: Throw MissingTransfersException] → [If Yes: Check source.Transfers ≠ null] → [If No: Throw MissingTransfersException] → [If Yes: Determine transferCoefficient (IsARequest ? -1 : 1)] → Update dest balance (amount \* coefficient) → Update source balance (amount \* coefficient \* -1) → Set transfer.Status = approved → Update dest.Transfers[transferId] → Update source.Transfers[transferId] → Save destination account → Save source account → END

**RejectTransferAsync Method:**

public async Task RejectTransferAsync(string transferId, string destId)

{

var transfer = await GetTransferAsync(destId, transferId);

if (transfer.Status != Transfer.RequestStatus.waiting)

throw new InvalidTransferStatusException(transferId, transfer.Status);

Account dest = await GetAccountAsync(destId);

Account source = await GetAccountAsync(transfer.Source);

if (dest.Transfers == null)

throw new MissingTransfersException(destId);

if (source.Transfers == null)

throw new MissingTransfersException(transfer.Source);

transfer.Status = Transfer.RequestStatus.rejected;

dest.Transfers[transferId] = transfer;

source.Transfers[transferId] = transfer;

await UpdateAccountAsync(dest.Id, dest);

await UpdateAccountAsync(source.Id, source);

}

**Purpose:** Rejects a pending transfer without executing any money movement. Similar to approve but only changes the status to rejected and updates both accounts without modifying balances.

**CheckPassword Method:**

public async Task<bool> CheckPassword(string id, string password)

{

Account account = await GetAccountAsync(id);

return (account.HashedPassword) == password;

}

**Purpose:** Validates user credentials by comparing the provided password with the stored hashed password for authentication purposes.

**5. Database/File Storage Description**

**Firebase Realtime Database Structure:**

The FirebaseManager interacts with Firebase Realtime Database using the following hierarchical JSON structure:

**Root Database Structure:**

{

"accounts": {

"accountId1": {

"Id": "string - unique account identifier",

"Name": "string - account holder name",

"HashedPassword": "string - encrypted password",

"Remainder": "double - current account balance",

"Transfers": {

"transferId1": {

"Date": "string - date in dd/MM/yyyy format",

"Source": "string - source account ID",

"Destination": "string - destination account ID",

"Amount": "double - transfer amount",

"IsARequest": "boolean - true if transfer request, false if direct transfer",

"Status": "enum - waiting/approved/rejected"

},

"transferId2": { ... },

...

}

},

"accountId2": { ... },

...

}

}

**Database Operations Mapping:**

* **Account Storage Path:** /accounts/{accountId}
* **Transfer Storage Path:** /accounts/{accountId}/Transfers/{transferId}
* **Data Format:** JSON with automatic serialization/deserialization
* **Key Generation:** Account IDs are provided by the application, Transfer IDs are generated using Guid.NewGuid()

**Data Persistence Characteristics:**

* **Real-time Synchronization:** Firebase provides real-time updates across all connected clients
* **Atomic Operations:** Each account update is atomic, ensuring data consistency
* **Duplicate Storage:** Transfers are stored in both source and destination accounts for efficient querying
* **Sorting Strategy:** Transfers are automatically sorted by status (waiting first) then by date
* **Error Handling:** Custom exceptions for various failure scenarios (AccountNotFound, InvalidTransferAmount, etc.)

**Business Logic in Database Operations:**

* **Transfer Coefficient System:** Uses +1/-1 multipliers to handle different transfer directions
* **Status Management:** Three-state transfer lifecycle (waiting → approved/rejected)
* **Dual Account Updates:** Ensures both accounts are updated atomically for transfer operations
* **Validation Layers:** Multiple validation checks before executing financial operations

**הודעות המשתמש באפליקציה**

