



Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Szarvady Ambrus

**KÖZÖSSÉGI
PROGRAMSZERVEZŐ
ALKALMAZÁS FEJLESZTÉSE
ANDROID PLATFORMON**

KONZULENS

Somogyi Norbert Zsolt

BUDAPEST, 2022

Tartalomjegyzék

Összefoglaló	5
Abstract.....	6
1 Bevezetés	7
1.1 Motiváció	7
1.2 Célkitűzés és funkcionalitás.....	7
1.3 Előzmények	9
2 Felhasznált technológiák	10
2.1 Android felépítése.....	10
2.2 Activity életciklus	11
2.3 Fragment életciklus.....	12
2.4 Firebase	13
2.4.1 Firebase Authentication	15
2.4.2 Cloud Firestore	16
2.4.3 Cloud Storage	17
3 Tervezés	18
3.1 Követelmény-elemzés.....	18
3.1.1 Funkcionális követelmények	18
3.1.2 Nem-funkcionális követelmények	19
3.1.3 Használati esetek.....	20
3.2 Adatmodell.....	20
3.3 Architektúra	21
3.3.1 Architekturális minták	21
3.3.2 MVP.....	22
3.3.3 Kiegészített MVP.....	23
4 Implementáció	25
4.1 Bejelentkező és regisztrációs oldal	25
4.2 Főoldal	29
4.2.1 Elérhető események	29
4.2.2 Eseményeim oldal.....	33
4.2.3 Esemény készítő oldal	35
4.2.4 Közösségi oldal.....	37

4.2.5 Saját Profil Oldal	38
4.3 Esemény adatlap	41
4.4 Felhasználói adatlap.....	42
4.5 Chat oldal.....	44
5 Tesztelés	47
5.1 Teszt lefedettség	47
5.2 Tesztek Felépítése.....	48
5.3 Nehézségek	49
5.3.1 MVP + Repository	49
5.3.2 Függőségek	50
5.3.3 This kulcsszó	50
6 Összefoglaló	51
7 Hivatkozások	52

HALLGATÓI NYILATKOZAT

Alulírott **Szarvady Ambrus**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2022. 04. 05

.....
Szarvady Ambrus

Összefoglaló

A mai világban számtalan közösségi alkalmazás létezik, melyek sok ember számára jelentősen megkönnyítik az emberi kapcsolatok teremtését és fenntartását. A különböző alkalmazások a kapcsolat teremtés különböző típusaira, fázisaira próbálnak megoldást nyújtani. Például a Tinder a párkapcsolat keresésben, a LinkedIn a munkakapcsolat teremtésben, míg a Facebook a már meglévő kapcsolatok fenntartásában segít. A felhasználóknak ezeken az alkalmazásokon online, az interneten keresztül kell kapcsolatot teremteniük. Ez sokak számára nehézséget okoz.

A szakdolgozatomhoz készített alkalmazás erre a problémára nyújt megoldást. A felhasználók eseményeket hozhatnak létre, amikre a többi felhasználó jelentkezhet és elmehet rájuk. Mivel a felhasználók előbb találkoznak élőben, és csak ezután építenek kapcsolatot, így a kapcsolat építés offline történik, ráadásul az esemény közös pont lehet, ezzel is segítve a kapcsolat teremtést.

A szakdolgozatomban kitérek az alkalmazás elkészítésének teljes folyamatára. A tervezés során meghozott mérnöki döntésekre, a felhasznált technológiákra, valamint a fejlesztés során fellépő problémák megoldására is. Bemutatom az alkalmazás felépítését, megvalósításának részleteit. Végezetül értékelem az elkészült eredményeket, összegzem a munka során szerzett tapasztalataimat.

Abstract

In today's world the applications make the relationship creation between people way easier. Different applications try to be a solution for different types and phases of the relationship. For Instance, Tinder tries to be a solution for creating love relationships, Linked In is for creating work relationships, and Facebook is for keeping up the existing connections. With these applications the users have to make relationships online, through the internet. This can be hard for a lot of people.

The application which I created for my thesis tries to solve this problem. The Users can create events, and to these events Others can subscribe to and attend. Because the Users meet up first, and only connecting afterwards, the creation of a relationship is made offline, and in bonus the event can be a middle ground for creating a discussion. These things make the relationship creations easier for a lot of people.

In my thesis I write about the full process of the application creation. About the developer choices, made during the development, about the used technologies, and about the problems occurring during development, and the solutions for them. I show the structure of the application, and the implementation of it. At last, I evaluate the results, and summarize the experiences gained during the development process.

1 Bevezetés

Ebben a fejezetben bemutatom az alkalmazás létjogosultságának motivációját és célkitűzéseit, illetve ismertetem a dolgozat felépítését.

1.1 Motiváció

A modern világban, ahol az emberek döntő többségének zsebében ott lapul egy okostelefon, számtalan alkalmazás könnyíti meg az egyes felhasználók életét. A kapcsolatteremtő alkalmazások az élet megkönnyítésén dolgoznak azáltal, hogy segítségükkel könnyebben teremthetünk, vagy épp tarthatunk kapcsolatot. A kapcsolattartásnak fontos aspektusa az offline találkozás, hiszen ez által mélyíthetjük el kapcsolatunkat, illetve szerezhetünk közös emlékeket.

Ezen találkozókhöz jelenleg a legtöbb esetben szükségszerű az ezeket megelőző kapcsolatteremtés. Ez több felhasználói csoport számára is problémaforrás lehet. Azoknak a felhasználóknak, akiknek nehézséget okoz az online térben való ismerkedés, nehéz a meglévő alkalmazások segítségével kapcsolatot teremteni. Léteznek azonban olyanok is, akik nem is szeretnék tartós kapcsolatot teremteni, csupán egy közösségi programban kívánnak részt venni. A dolgozatban bemutatott alkalmazás mindezek számára nyújt segítséget azáltal, hogy kapcsolatteremtés nélkül ad lehetőséget társasági programokban való részvételhez. Ezáltal azon felhasználóknak, akiknek nehezebb az online ismerkedés, lehetőségük nyílik offline ismerkedni. Ennek előnye, hogy a választott esemény közös pontként segíthet a későbbi esetleges kapcsolat-teremtésben. Emellett lehetőség nyílik azok számára is, akik csak egy társasági programban szeretnék részt venni, hogy gyorsan és egyszerűen szerezzenek társaságot a kívánt szociális programjukhoz.

1.2 Célkitűzés és funkcionalitás

A megvalósítandó alkalmazás célja tehát felhasználók közötti események szervezése. A felhasználók eseményeket tehetnek közzé, képesek egymás eseményeire reagálni, részvételi szándékot jelezni. Egy eseményen való részvétel alapvetően kétféle módon jöhet létre. Az első opció, hogy a felhasználó saját maga hoz létre egy eseményt, amire mások jelentkezhetnek, és ezáltal keletkezik egy társaság az adott eseményhez. Ilyen esemény bármi lehet, például egy közös piknik a fűben. Ilyenkor az esemény

hirdetésének tartalmaznia kell az esemény alapvető információit (pl. időpont, helyszín, egy rövid leírás), ezzel segítve a többi felhasználót a részvétel döntésében. A másik lehetőség, hogy a felhasználó egy már meglévő eseményre iratkozik fel.

Az események szervezésén kívül az alkalmazásnak lehetőséget kell biztosítania felhasználók közötti kommunikációra is. Ennek megfelelően a felhasználók profilokkal rendelkeznek, amelyek rövid leírással, illetve profilképpel elláthatók. Ezen felül a többi felhasználó képes értékelni mások profilját, ezzel is segítve az eseményen való részvétel döntését. Mindezek által nem csupán az esemény adatai alapján, de az eseményen résztvevő személyek alapján is eldönthető, hogy egy felhasználó szeretne-e részt venni egy eseményen.

Egy eseményre való feliratkozás után az esemény elérhetővé válik a felhasználó eseményeit gyűjtő oldalán. Ekkor megnyílik a lehetőség számára, hogy részt vegyen az eseményhez tartozó csoport beszélgetésben, ahol a találkozó részleteit beszélhetik meg a résztvevők. A csoportbeszélgetésen kívül egy másik kapcsolatteremtő része az alkalmazásnak a társalgó lista. Ide azok a személyek kerülnek be, akik vagy jelentkeztek az egyik eseményünkre, vagy szerveztek olyan eseményt, amin részt veszünk. Itt lehetőségünk van megtekinteni a profiljukat, ahol a személyes adataik mellett a felhasználó által létrehozott eseményeket is láthatjuk. Ezen kívül lehetőségünk van a listán szereplő felhasználókkal privát beszélgetésbe elegyedni. A társalgó listán egy felhasználó addig szerepel, amíg van közös eseményük, vagy az egyikük úgy dönt, hogy barát státuszba helyezi a másikat. Ilyenkor a felhasználók a barát státusz megszakításáig bent maradnak egymás társalgó listájában. Ez nem csak a szervezés lebonyolításában segít, hanem a találkozó utáni esetleges kapcsolat-tartásban is.

Az alkalmazás Android operációs rendszeren fut, a minimum Android verzió 8.0, ezáltal nagy felhasználói réteg számára érhető el. A működéshez szükséges a felhasználókat autentikálni, hogy a különböző profilokat kezelni tudjam. Az autentikációs folyamat arra szolgál, hogy beazonosítsam a felhasználót. Ezenkívül szükség van adatbázisra az események, felhasználók, üzenetek tárolására, valamint a feltöltött profilképeket is el kell tudni menteni. Ezek megoldásához a Firebase szolgáltatását vettem igénybe, ami egy BaaS azaz Backend-As-a-Service szolgáltatás. Ez röviden azt jelenti, hogy az alkalmazás backend, azaz az a felhasználó számára láthatatlan, a szerveren háttérben futó részét külső, teljesen működőképes szolgáltatásként kapjuk. Ez nagyban meggyorsítja az alkalmazás fejlesztését. Ezért is

használják sokan a Firebaseet MVP – Minimum Viable Product – fejlesztésre. Ez annyit tesz, hogy az alkalmazást minél gyorsabban elkészítik, hogy a lehető leghamarabb kerülhessen a felhasználó kezébe. Az alkalmazás letisztult, és intuitív kezelőfelületének eléréséhez több Androidos technológiát is igénybe vettem. Ilyenek például a Fragmentek, az alkalmazás letisztult, folyékony használhatóságának érdekében, vagy a RecyclerView, amely segítségével az eseményeket és a barátokat könnyen egy rugalmas listába tudtam rendezni.

A felhasznált Androidos technológiák mellett a Firebase több szolgáltatását is igénybe vettem. Az első igénybe vett szolgáltatás a Firebase Authentication felhasználó kezelő szolgáltatás volt. A következő szolgáltatás a Firestore volt, ami egy NoSQL adatbázis. Ez az adatbázis tárolja az alkalmazás működéséhez szükséges adatokat. Az utolsó szolgáltatást a profil képek tárolásához vettem igénybe. Ez a Cloud Storage volt, ami egy felhő alapú tárhely, ahova fájlokat lehet fel-és letölteni.

A dolgozat felépítése a következő. A 2. fejezetben bemutatom az alkalmazás elkészítéséhez felhasznált technológiákat. A 3. fejezetben ismertetem a tervezés fázisában meghozott döntéseket. A 4. fejezetben bemutatom az alkalmazás implementációját. Az 5. fejezetben kitérek a tesztelés fázisára. Végül a 6. fejezetben értékelem az eredményeket, összefoglalom a szakdolgozatom elkészítése során gyűjtött tapasztalatokat.

1.3 Előzmények

A szakdolgozatomhoz készített alkalmazás alapja a Mobil- és webes szoftverek (VIAUAC00) tárgyhoz készített nagy házifeladatomban. Az alkalmazás ezen kezdetleges verziójában a Bejelentkező/Regisztrációs oldal, az elérhető események oldal, a saját eseményeim oldal, valamint az esemény létrehozó oldal kezdetleges verziói készültek el. Minden más a szakdolgozat keretein belül született. A kiinduló projekt publikusan elérhető Github-on:

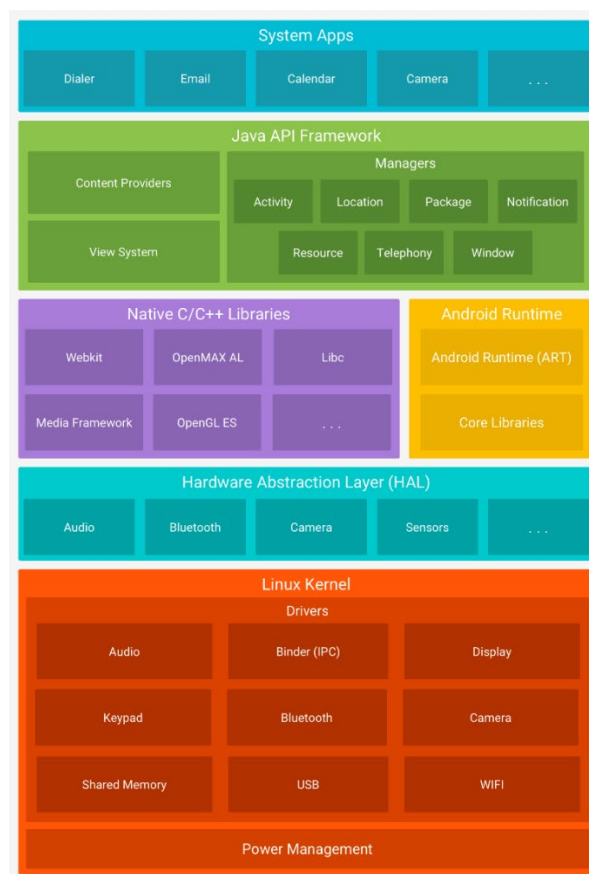
<https://github.com/kayyer/SzakdolgozatKiindulas>

2 Felhasznált technológiák

Ebben a fejezetben röviden bemutatom az alkalmazás felépítéséhez és elkészítéséhez kapcsolódó és felhasznált technológiai alapokat.

2.1 Android felépítése

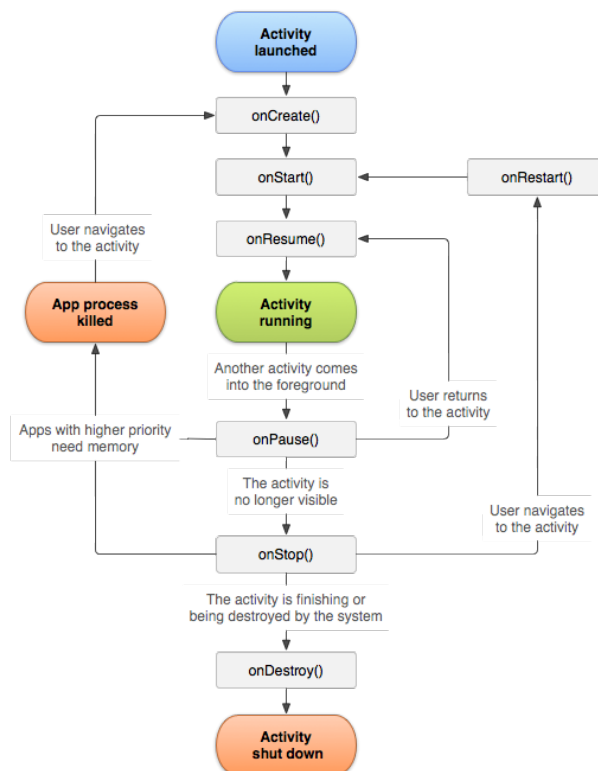
Az alkalmazást Android operációs rendszerre készítettem el, mely az egyik legelterjedtebb okostelefon (és okoseszköz) operációsrendszer. A felépítése alapja, ahogy azt az 1. ábra szemlélteti, egy Linux-alapú kernel. Itt találhatóak a készülék gyártója által készített driverek, amik olyan szoftverek, melyek az egyes hardver komponensek (mint például kamera vagy mikrofon) vezérlését végzik. A második szint a HAL (Hardware Abstraction Layer - hardver absztrakciós szint), az itt lévő elemek a kernelben található driverek használatát fedik el, ezen keresztül érhetőek el a magasabb szinteken. A következő szinten két elem található. A sárga színnel jelölt ART melyet arra terveztek, hogy minimalizált memória igényű virtuális gépeket futtasson, ennek külön példányát futtatja minden egyes Androidos alkalmazás az Android 5.0 óta. A szint másik eleme az ábrán lila színnel jelölt C/C++ könyvtárak, amik segítségével a következő szinten található Java API Framework elemei elérhetik azon Androidos szolgáltatásokat, melyek natív C/C++-os könyvtárakat igényelnek. Ezekre épül a korábban említett Java API Framework, melyek igénybevételel elérhetjük az Android által nyújtott funkciókat az alkalmazás fejlesztés során. Az alkalmazás fejlesztés során megírt alkalmazások a legfelső szinten találhatók.



1. ábra: Az Android felépítése [1]

2.2 Activity életciklus

Az alkalmazásom fejlesztése során Activity-ket (felhasználói felülettel rendelkező különálló nézet) és Fragmenteket (egy nagyobb összefüggő képernyő felületért felelős objektum) is használtam. Az Activity életciklusát a 2. ábra szemlélteti. Az első állapot az onCreate(), ez az Activity létrejöttékor hívódik meg. Ezután az onStart() következik, ami felkészíti az Activity-t, hogy a felhasználó számára is látható legyen. Ezt követi az onResume(), ami akkor, és addig fut, amíg az Activity fókuszban van, ekkor már látható az Activity a felhasználó számára. Az onPause() akkor hívódik meg, ha az Activity már nincs előtérben, az onStop() pedig, ha már nem látszik. Az utolsó állapot az onDestroy(), ami az Activity megsemmisülése előtt hívódik meg. Az imént felsorolt életciklusból én az onCreate() függvényt írom felül, ami az Activity létrehozásakor lép érvénybe. Itt a a Fragment Manager adapter segítségével beállítom, hogy melyik Fragmenteket, milyen sorrendben szeretném az Activity-hez kötni.

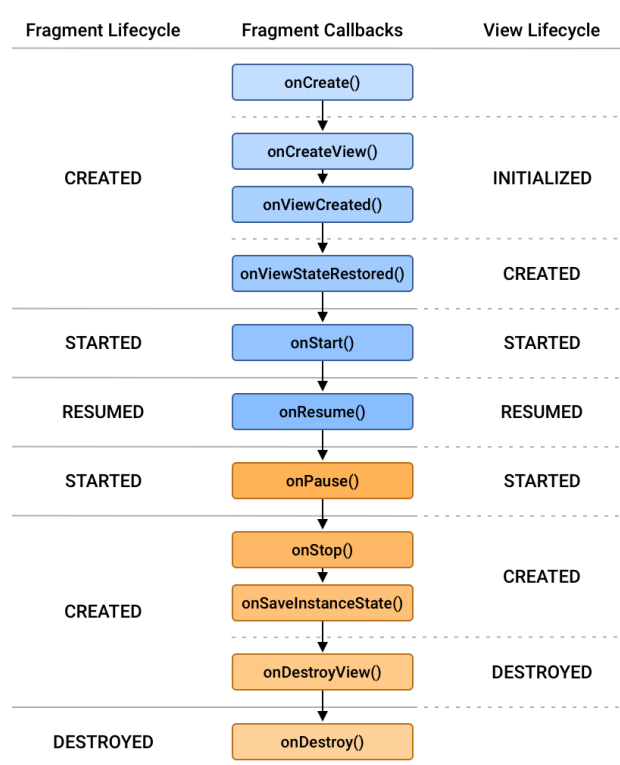


2. ábra: Az Activity életciklusa [2]

2.3 Fragment életciklus

A Fragment egy olyan újrafelhasználható eleme a felhasználói interfésznek, amely a kijelző egy nagyobb szegmenséért felelős. Saját életciklusa van, melyet a 3. ábra szemléltet. Az ábra baloldalán látszik a Fragment életciklusa, jobb oldalán a View életciklusa, középen pedig, az előző életciklusokban történő változásokkor meghívódó függvények. Az életciklus első foka, amikor a Fragment létrejön. Ilyenkor az `onCreate()` függvény hívódik meg. Ekkor a View még nem inicializálódott. Az `onCreateView()` és az `onViewStateRestored()` hívásakor a View nem csak inicializálva van, de már létre is jött. Az `onStart()` létrejöttékor már garantálva van, hogy mind a Fragment, mind View elérhető. Az `onResume()` hívásakor, az alkalmazás látható, és készen áll a felhasználói interakcióra. Ez után az életciklusok fordulóponthoz érkeztek. Eddig a pontig az állapotváltásoktól felépültek, mostantól viszont leállnak. Először az `onPause()` hívódik meg. Ekkor a Fragment még látható a felhasználó számára, ám a View-val együtt az állapota visszavált RESUMED-ról STARTED-ra. A következő lépés akkor következik be, ha a Fragment már nem látható. Ekkor a Fragment és a View állapota STARTED-ról CREATED-re változik, és meghívódik az `onStop()` függvény. Az `onStop()` után az

onSaveInstanceState() ,ami elmenti a Fragment és a View állapotát. Ezután az onDestroyView() következik, ekkor a View eléri az életciklusa végét átlép a DESTROYED állapotba, majd megsemmisül. A Fragment állapota továbbra is CREATED. Ha a Fragmentet eltávolítják, vagy a Fragment Manager megsemmisül, a Fragment életciklusa is véget ér. Átlép a DESTROYED állapotba és meghívódik az onDestroy() függvény. A felsorolt függvényhívásokból én az onCreateView()-t írom felül, ahol a Fragmentekben használt változókat inicializálom. Az onStart() függvényben hívom meg azokat a függvényeket amiknek akkor kell lefutnia valahányszor az adott Fragment az előtérbe kerül.



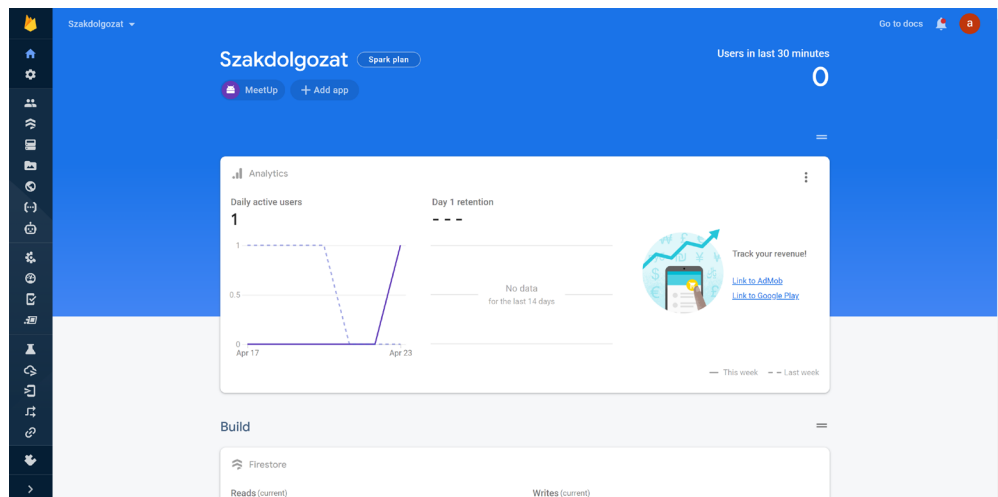
3. ábra: A Fragment életciklusa [3]

2.4 Firebase

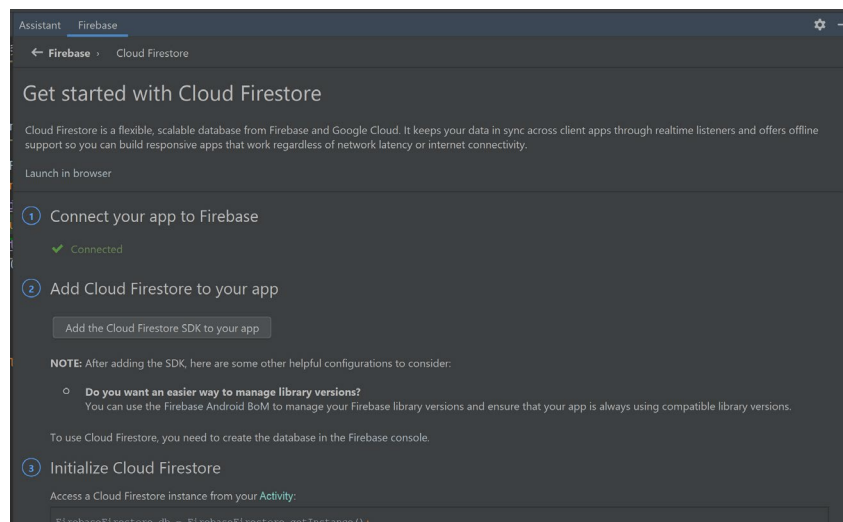
Az alkalmazás backend részét a Firebase [4] szolgáltatás igénybevételével oldottam meg. A backend az alkalmazás server oldali részét tartalmazza, ahol az adatokat, fájlokat tárolom, valamint a felhasználói fiókokat kezelem. A Firebase egy Backend-as-a-Service (BaaS). Ezek olyan szolgáltatások, melyek főként a mobil és weblap fejlesztést könnyítik meg azáltal, hogy a backend részeket felhőalapú szolgáltatásként kapjuk. Ez azt jelenti, hogy a serveroldali programrészeket nem nekünk

kell megírni és karbantartani, hanem a szolgáltatás megfelelő függvényeit használva, átadhatjuk ezt a felelősséget a szolgáltatónak. Ez nagyban meggyorsítja a fejlesztési folyamatot, hiszen így fejlesztőként a frontend részre fókuszálhatunk.

A BaaS szolgáltatások közül munkám során a Firebase-t választottam. A Firebase bár 2011-ben önálló céggént indult, 2014-ben felvásárolta a Google. Ennek köszönhetően a Firebase integrálva van az Android Studioba [5], ami a hivatalos fejlesztő környezet az Android alkalmazásokhoz. Emiatt a Firebase használatát egyszerű megvalósítani. Első lépésben létre kell hozni egy új Firebase projektet, a Firebase konzolon keresztül (ez console.firebase.google.com url-en érhető el). Ekkor a projekt főképernyője kerülünk, ahogy ezt a 4. ábra is szemlélteti. Itt statisztikát találunk az alkalmazás egyes elemeiről. Itt látható például, hogy hány szerver írás, illetve olvasás történt. Ez azért fontos, mert a Firebase árazása ettől függ. Napi 10000/50000 írás/olvasásig ingyenes, ezután a felhasználás mennyiségével arányosan növekszik. Számomra ez azért szerencsés, mert a szakdolgozatomhoz ennyi írás/olvasás bőségesen elegendő, így nem kerül semmibe ezt a szolgáltatást használnom. Ezen az oldalon kiválaszthatjuk mire szeretnénk fejleszteni (Web/Android/iOS) majd a felkonfigurálást követően letöltünk egy fájlt, amit az alkalmazás projektjébe illesztve elérhetővé válnak számunkra a Firebase szolgáltatások. Ezen felül az egyes Firebase szolgáltatások használatba vétele előtt szükség van a a Gradle [6] fájlba a függőségeket beilleszteni. A Gradle az Android Studio által használt Build eszköztár, amely automatizálja és menedzseli a buildelési folyamatot. Mivel a Firebase az Android Studio eszköztár része, ezért a függőség implementálással könnyű dolgunk van, hiszen elég, ha az eszközök fülön kiválasztjuk a Firebase-t és máris kiválaszthatjuk melyik Firebase szolgáltatást szeretnénk igénybe venni (5. ábra).



4. ábra: A Firebase konzolja



5. ábra: Az Android Studio Firebase asszisztense

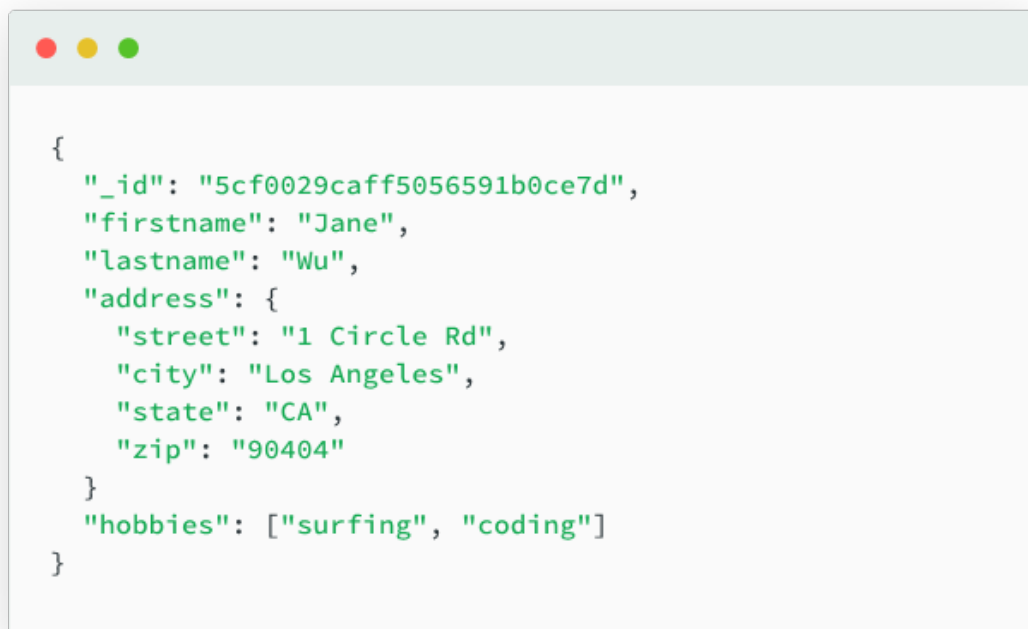
2.4.1 Firebase Authentication

A szakdolgozatomhoz három Firebase szolgáltatást vettem igénybe. Az első a Firebase autentikáció volt. Ez a szolgáltatás a felhasználók kezelésében segít. Kiválasztható, hogy milyen módszerrel léphessenek be a felhasználók. Az alkalmazásom az email/jelszó párost használja a felhasználók azonosításához. Ezzel a módszerrel a felhasználói adatok biztonságos tárolását a Firebase-re bízhatjuk, így nem kell erre külön figyelmet szentelnünk, szemben azzal, ha csak egy egyszerű adatbázisba vennénk fel őket. Egy létrehozott felhasználóról a Firebase csak alapvető adatokat tárol, mint például az email cím. Mivel a felhasználókról több server-oldali adat tárolására van szükség,

ezért a többi felhasználói adat tárolásához egy másik Firebase szolgáltatást célszerű használni, ez a Cloud Firestore [7].

2.4.2 Cloud Firestore

A Cloud Firestore egy NoSQL-alapú [8] adatbázis. A NoSQL adatbázisok sajátossága, hogy nem jól-definiált sémákban tárolják az adatokat. Az ilyen adatbázisok (ahogy a 6. ábra szemlélteti) nem táblákban tárolják az adatokat, hanem kollekciókban. Egy kollekció számos dokumentumból állhat, ezek reprezentálják a relációs rekordokat. A dokumentumok mező-érték (field-value) párokat tartalmazhatnak, amelyek egy rekord egy attribútumának felelnek meg.



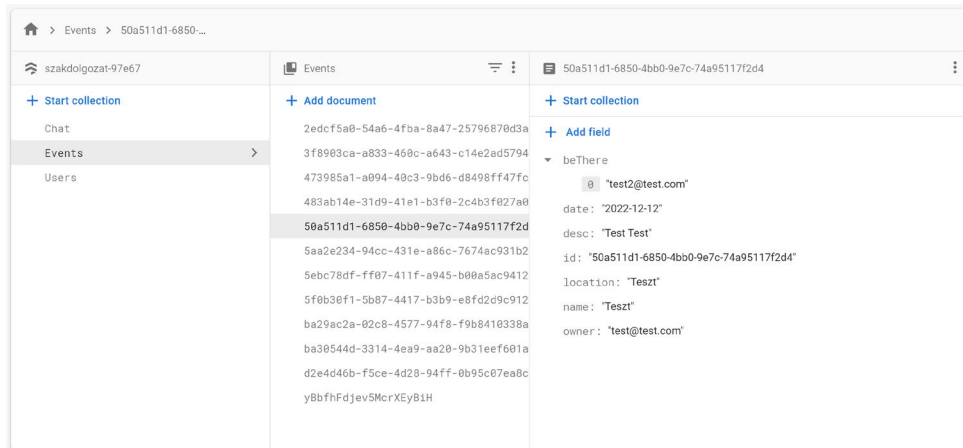
```
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  },
  "hobbies": ["surfing", "coding"]
}
```

6. ábra: NoSQL adattárolás példa [9]

Ekkor, ha az egyik dokumentum egy másik dokumentum valamely mezőjére hivatkozna, akkor az adott dokumentumnak közvetlenül tartalmaznia kell az adott mezőt. Ezáltal egy dokumentumon belül minden adat egy helyen tárolódik, viszont adat duplikáció lép fel, hiszen a kívánt mezőt mindkét dokumentumnak tartalmaznia kell. Az adat duplikáció nehézséget okoz az adatbázisba történő írásnál, mivel figyelni kell, hogy az írt adat minden objektumban konzisztens maradjon. Ezzel szemben az adatbázisból történő olvasásnál előnyt jelent mivel minden adat egy helyen van. Azoknál az applikációknál, melyek sokkal többet olvasnak, mint írnak, az olvasással járó előny

gyakran többet számít az írásnál fellépő nehézségeknél, ezért ilyen alkalmazások fejlesztéséhez sokszor jobb ötlet NoSQL adatbázist használni relációs adatbázis helyett.

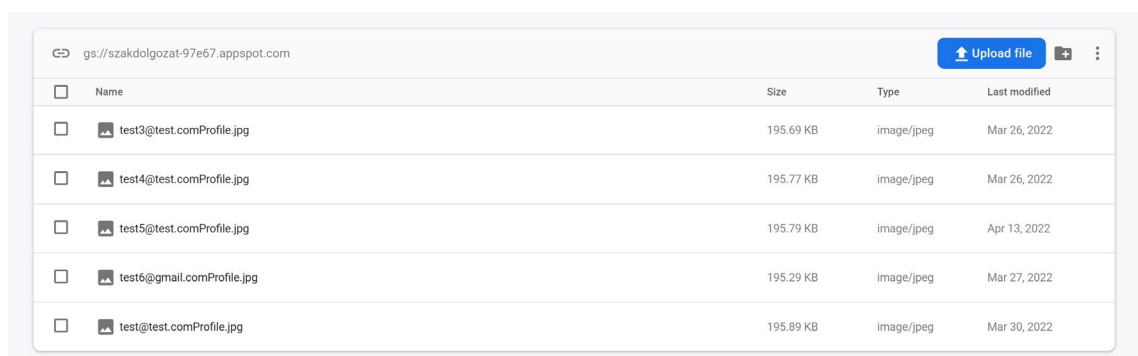
A Cloud Firestore-ban kollekciókban tárolunk dokumentumokat, amelyekben az egyes mezők szerepelnek (7. ábra). Az adatok lekérdezése lehet egyszeri, vagy akár történhet folyamatosan, ha az adatok módosulnak.



7. ábra: A Firestore adatbázis felhasználói interfésze

2.4.3 Cloud Storage

A Firestoreban csak mezőket tárolhatunk, a dokumentumok mérete maximum 1 megabájt lehet. Így a fájlok (például a felhasználók profilképei) tárolását más módon kellett megoldanom. Ehhez a Cloud Storage [10] szolgáltatást vettem igénybe. Ide egyszerű parancsokkal lehet fájlokat feltölteni, majd letölteni. A profilképek feltöltéskor a fájlnev neve az adott felhasználó email címe lesz, ezáltal amikor szükség van rájuk az email címük segítségével könnyedén megtalálható a hozzájuk tartozó fájl. A Cloud Storage-ban lévő fájlok tárolására mutat példát a 8. ábra.



8. ábra: A Cloud Storage felhasználói interfésze

3 Tervezés

Egy alkalmazás készítésekor az első lépés a tervezés. Fontos az alapos tervezés, mivel egy jól megtervezett alkalmazás fejlesztése könnyebb és kevesebb hibalehetőséget tartalmaz. A tervezés több szakaszra osztható. Az első az alkalmazástól elvárt követelmények meghatározása. Az alkalmazás tervezése során két fajta követelmény típust vehetünk figyelembe: a funkcionális és a nem-funkcionális követelményeket. A tervezés második szakasza az adatbázisban tárolt elemek modellezése, vagyis az adatmodell kialakítása. Az utolsó lépés az alkalmazás architektúráis felépítésének megtervezése volt.

3.1 Követelmény-elemzés

Az alkalmazással szemben támasztott követelmények alapvetően kétféle típusúak lehetnek: funkcionálisak vagy nem-funkcionálisak. A továbbiakban ezeket tekintjük át.

3.1.1 Funkcionális követelmények

A funkcionális követelmények olyan követelmények, melyek az alkalmazással szemben támasztott funkcionális elvárásokat írják le. Az alkalmazásom tervezésénél ezek a következő elvárások voltak:

- Az egyes felhasználóknak saját profiljuk, eseményeik vannak.
- A felhasználók fiókokat hozhatnak létre, és ezekbe bejelentkezve azonosítja őket a rendszer.
- A felhasználóknak létre kell tudniuk hozni eseményeket. Más felhasználóknak látniuk kell ezeket az eseményeket, és fel kell tudniuk iratkozni rájuk.
- A felhasználók le kell tudjanak iratkozni a feliratkozott eseményükről. Az eseményt létrehozó felhasználónak, ha meggondolja magát tudnia kell törölni a létrehozott eseményét.
- A felhasználóknak látniuk kell a létrehozott, és feliratkozott eseményeiket. A felhasználóknak szükségük van egy adott esemény adatlap oldalára, ahol az adott eseményről kapnak bővebb információt.

- A felhasználóknak kapcsolatot kell tudni tartani. Ehhez szükség van az esemény szervezőjével való privát beszélgetésre, valamint egy csoport beszélgetésre, melyben az összes eseményre feliratkozott felhasználó részt vehet.
- Azért, hogy a felhasználók, ne csak az esemény részletei, de a rajta résztvevők alapján is hozhassák meg a feliratkozásról való döntésüket, szükség van a felhasználók által személyre szabható profilokra.
- A magas személyre szabhatóság érdekében a felhasználói profilokon szükség van a felhasználói profilképek, becenevek, leírások beállíthatóságára. Továbbá a felhasználók megítélhetőségének növelése érdekében szükség van egy lehetőségre, ahol más felhasználók értékelhetik őket.
- A felhasználóknak látniuk kell egymás felhasználói adatlapját
- A felhasználóknak látniuk kell a többi felhasználót, akikkel kapcsolatban állnak
- Lehetőséget kell adni a felhasználóknak, hogy az esemény végeztével is tudjanak egymással kommunikálni, ha szeretnének.

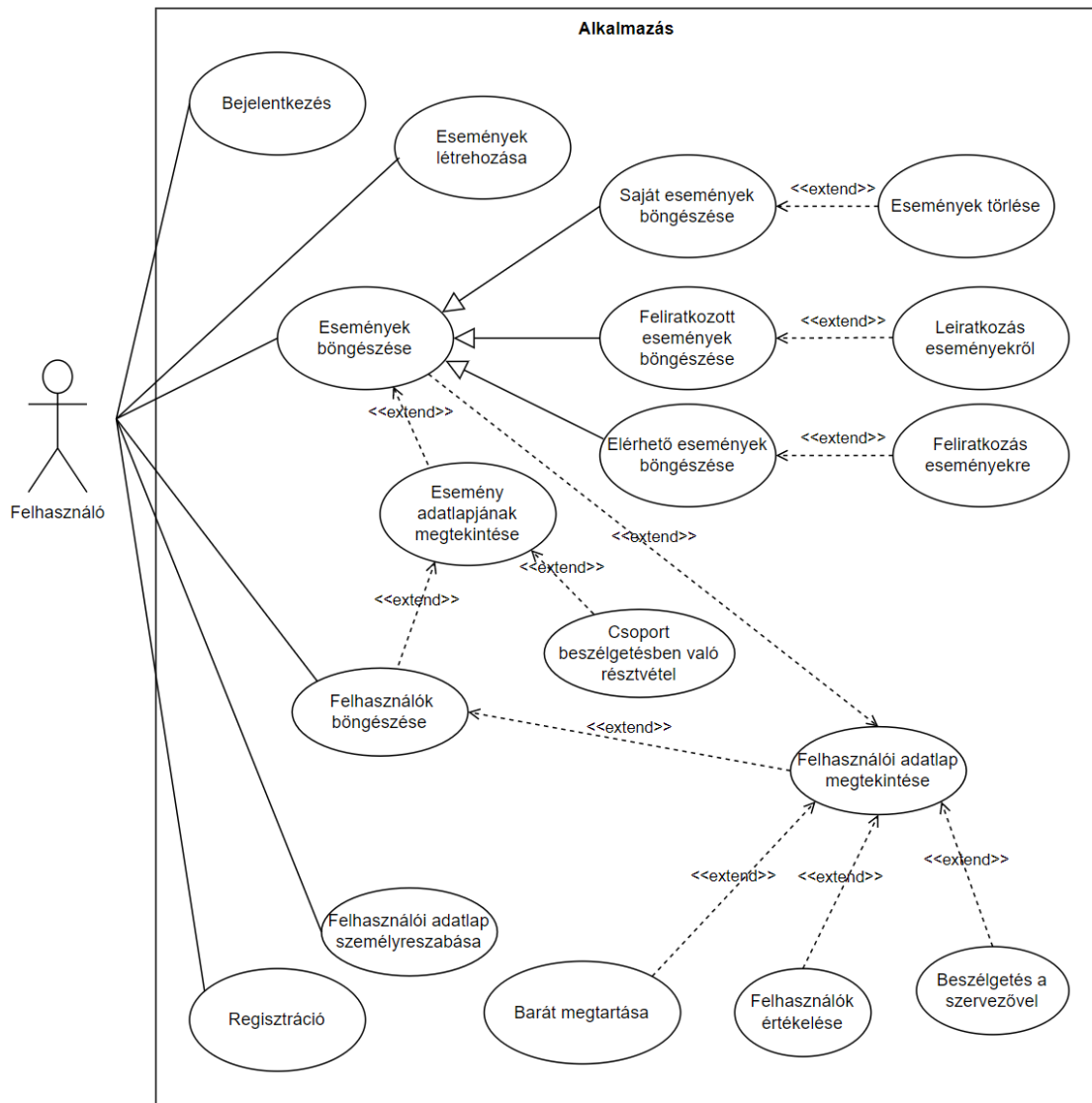
3.1.2 Nem-funkcionális követelmények

A nemfunkcionális követelmények olyan elvárások melyeket a rendszer működésével szemben elvárunk. Az alkalmazással szemben a következő nemfunkcionális követelményeim voltak:

- A felhasználók profiljuk létrehozásakor személyes adatokat adnak meg (jelszó, email cím), ezeket az adatokat biztonságosan kell tárolni.
- Amikor a felhasználók mezőt töltenek ki szükség van a beviteli mező ellenőrzésére, hogy megfelelő adatot adtak-e meg.
- Az adatbázisból való könnyű adatkinyeréshez szükséges, hogy az adatbázisban tárolt dokumentum nevek könnyen kikövetkeztethetők legyenek.

3.1.3 Használati esetek

Az alkalmazást minden felhasználó ugyanolyan jogkörrel és funkcionalitással használhatja. Ezeket a 9. ábra demonstrálja.

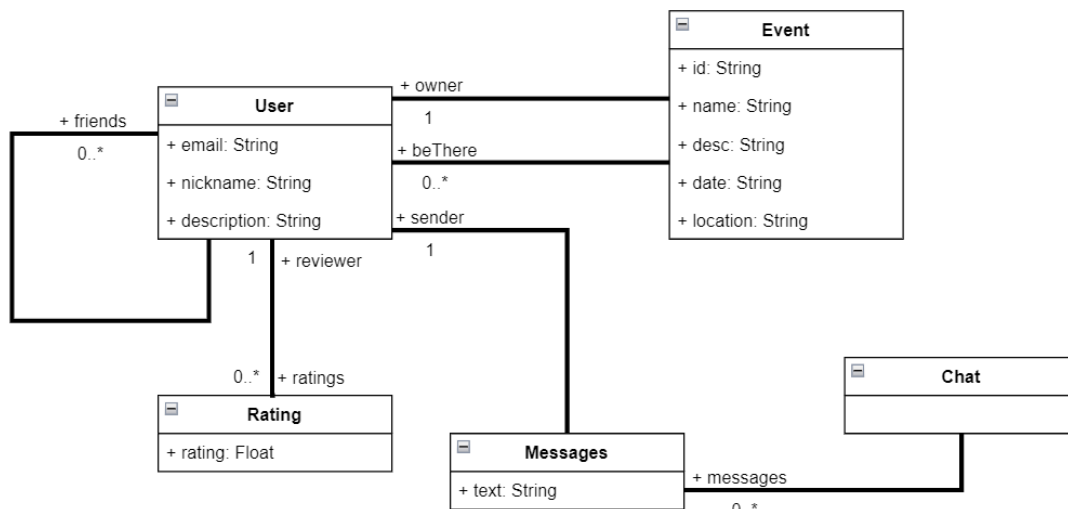


9. ábra: Az alkalmazás használati eset diagramja

3.2 Adatmodell

Mivel az alkalmazás backend része a Firestore szolgáltatás segítségével van megoldva, ami egy NoSQL adatbázis, ezért az adatbázisban lévő adatok modellezéséhez egy osztály diagramot használhatunk (10. ábra), ami kellően hasonlít a NoSQL adatbázisok által használt JSON objektum struktúrához. A felhasználók autentikációja a Firebase Authentication szolgáltatással van megoldva, ezért az ehhez kapcsolódó adatok nem

jelennek meg az adatbázisban. A felhasználóhoz kapcsolódó egyéb adatok (például becenév, vagy leírás) a Firestore adatbázisában tárolódnak. A felhasználó az autentikációhoz használt email címével azonosítható az adatbázisban. Emiatt, bár az osztály diagramon User típusú asszociáció kapcsolja össze a User osztályt a vele asszociációban lévő osztállyal, valójában az adatbázisban egy szöveges mező tárolódik, ami a beazonosításhoz szükséges email címet azonosítja.



10. ábra: Az alkalmazás adatmodellje

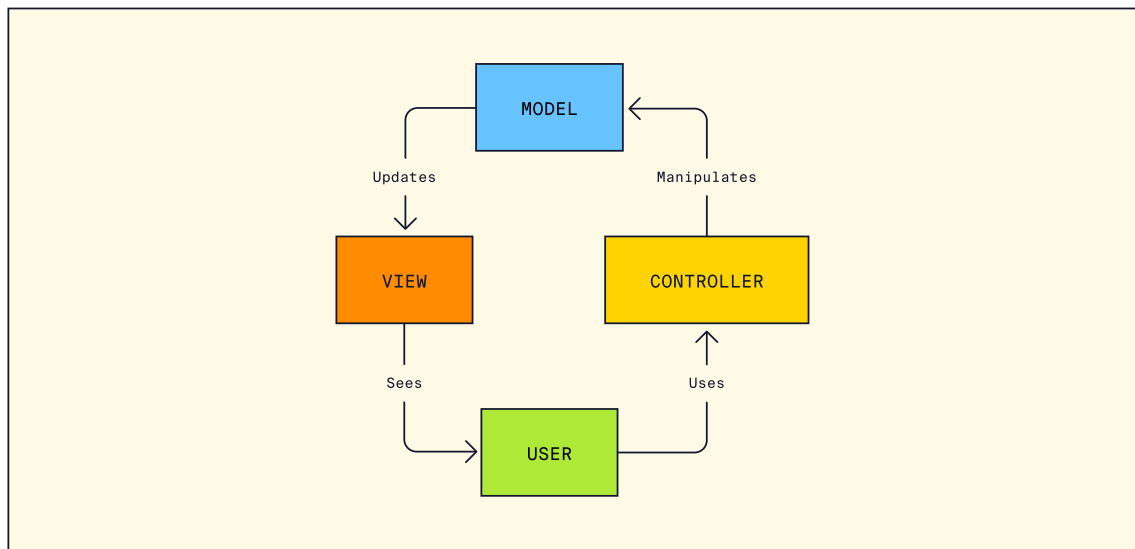
3.3 Architektúra

Nagyobb projektek fejlesztésekor kiemelten fontos az alkalmazás architektúrájának alapos megtervezése. Ez több szempontból is lényeges. A különböző elemek megfelelő szeparációja segítségével az alkalmazás könnyebben karbantartható, valamint tovább fejleszthető lesz.

3.3.1 Architekturális minták

Az architekturális minták általános, újra felhasználható megoldások, melyek gyakran előforduló problémákra nyújtanak megoldást. Az Androidos alkalmazások tervezésekor gyakran előforduló probléma, hogy egy osztály (pl. Activity vagy Fragment) túl sok dologért felelős. Ez probléma lehet, mivel így a program bővítése, karbantartása

könnyen nehézségekbe ütközhet. Ezen probléma megoldására, az Androidos alkalmazásokat egy kiválasztott architektúráis mintát követve érdemes elkészíteni. Az Androidos alkalmazásokhoz leggyakrabban használt architektúráis minták a Model-View-Controller (MVC) és a Model-View-Presenter (MVP). Az MVC felépítését szemlélteti a 11. ábra. A Model tárolja az adatokat, és az adatok változásakor jelez a View-nak. A View a megjelenítésért felelős, ezt látja a Felhasználó. A Controlleren keresztül manipulálhatja a felhasználó az adatokat, itt végez az alkalmazás minden esetleges validációs műveletet is.



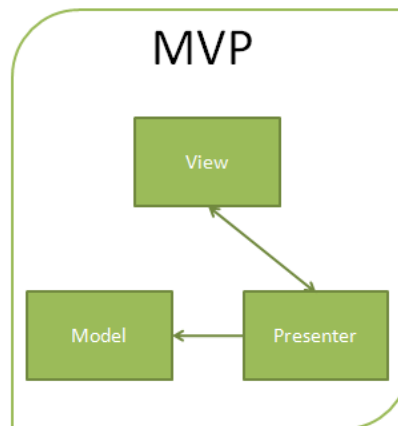
11. ábra: MVC architektúra [11]

3.3.2 MVP

Az MVP architektúra az MVC leszármazottja. Felépítését a 12. ábra szemlélteti. Bár a felépítése hasonló az MVC-hez, több helyen is különbözik tőle. Míg az MVC architektúrában a felhasználói műveletek feldolgozása alapvetően a controlleren keresztül történik, az MVP architektúrában ezért a View felelős. A View-ban csak a megjelenítésért felelős kód részlet kap helyet, ha a felhasználó interaktál vele segítségül hívja a Presentert. A Presenter felel a logikáért, ő köti össze a Modelt a View-val. Az utolsó elem a Model. MVP architektúrában a Model az adatok tárolásáért felelős.

Bár az architektúrák növelik az alkalmazás komplexitását, valamint tervezésük időigényes, mégis sok előnnyel járnak. MVP architektúra esetén mivel az egyes elemek jól elkülönülnek egymástól, a jövőben azok könnyen lecserélhetők. Például, ha az alkalmazásból webalkalmazást szeretnénk tervezni, elég lenne a View elemet lecserélni, nem kellene átírni az egész kódot, hiszen a logikai elemek változatlanul maradhatnak

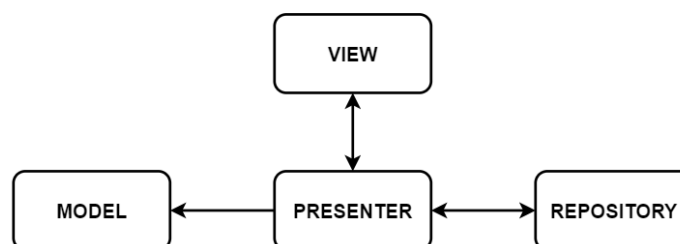
a Presenter-ben. Egy másik nagy előnye az MVP architektúra alapú tervezésnek, hogy mivel az egyes elemek kellően szeparáltak, a függőségek is szeparáltak lesznek, így tesztelésnél nem futunk bele olyan problémákba, melyeket a View-ban használt Android specifikus függvények függősége okozna. A fent említett előnyök miatt az alkalmazásom tervezése során az MVP architektúrát választottam.



12. ábra: MVP architektúra [12]

3.3.3 Kiegészített MVP

Bár MVP architektúrát alkalmaztam, a Firebase eléréséhez használt függvények miatt a Presenterben sok volt a függőség, ami tesztelésnél gondot okoz. Emiatt kiegészítettem az alap MVP architektúrát egy Repository elemmel, amely a Firebase szerver eléréseket kezeli. A Presenter tartalmazza ezt a Repository elemet, és használja, ha a szervert el akarja érni, a Repository pedig a megfelelő adatok megszerzése után értesíti a Presentert. Emiatt a Presenter rétegben semmiféle függőség nincs, így könnyen tesztelhető, valamint a függőségeket tartalmazó elemek könnyen lecserélhetővé váltak. Megemlítendő, hogy a Presenter külső függőségeinek kezelését gyakran hasonlóan, úgynevezett Interactorok segítségével szokták megoldani. Az így módosított architektúrát a 13. ábra szemlélteti.



13. ábra: Kiegészített MVP architektúra

A tervezés során egy másik fellépő problémát a RecyclerView listákhoz használt adapterek okozták. Mivel ezen adapterekben keveredtek a View és az adat, tehát Model részek, nehéz volt őket szétválasztani. Végül úgy oldottam meg ezt a problémát, hogy az adapter nem egy listát kap inicializáláskor, hanem egy Presentert, ami tartalmazza az adapterhez tartozó listát. Ezáltal teljesen szét tudtam választani a View-t a Modeltől, így minden elem pontosan a saját felelősségi körébe tartozó feladatokat látja el.

4 Implementáció

Ebben a fejezetben bemutatom az alkalmazásom implementációját az egyes oldalakra tagolva, funkcionkénti bontásban.

4.1 Bejelentkező és regisztrációs oldal

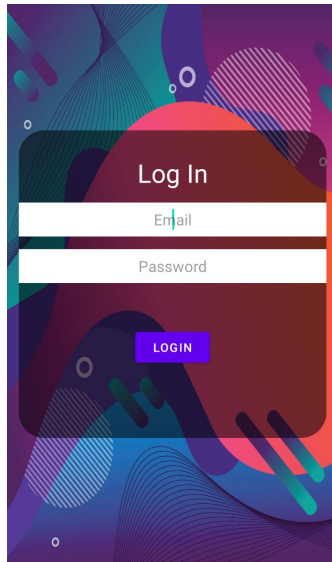
Az első oldal, amivel a felhasználó találkozni fog, a bejelentkező oldal. Mivel az alkalmazás oldalai nagyon hasonlóképpen épülnek fel, a bejelentkező oldalon keresztül mutatom be részletesen az oldalak alapvető felépítését a forráskód szintjén, és a jövőben csak az ettől történő eltéréseket fogom bemutatni. A bejelentkező oldal egy Activity-ből áll. Itt binding segítségével hozzákötöm az Activity-hez készített xml fájlt, ahol az oldal felépítését írom le deklaratívan. Ebben az xml fájlban a leglényegesebb elem a ViewPager elem, mely segítségével Fragmenteket tűzdelhetünk az Activity-kre. Mivel a dizájn nagy része a Fragmentekben történik, ezért ebben a fájlban a ViewPageren kívül nem sok más elem található.

Az Activity-ben beállítom a ViewPager adapterét a *LoginAdapterre*. Az adapter tartalmaz egy Fragment Manager-t, amelyre feltűzdeltem a bejelentkezés és regisztrációs Fragmentet. Ezt követően elkészítettem a két Fragmentet. Innentől már különbözik a többi oldaltól a bejelentkező oldal.

```
class LoginAdapter(fm: FragmentManager) : FragmentPagerAdapter(fm,
    BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT) {
    override fun getItem(position: Int): Fragment = when(position){
        0 -> LoginFragment()
        1 -> RegisterFragment()
        else -> LoginFragment()
    }

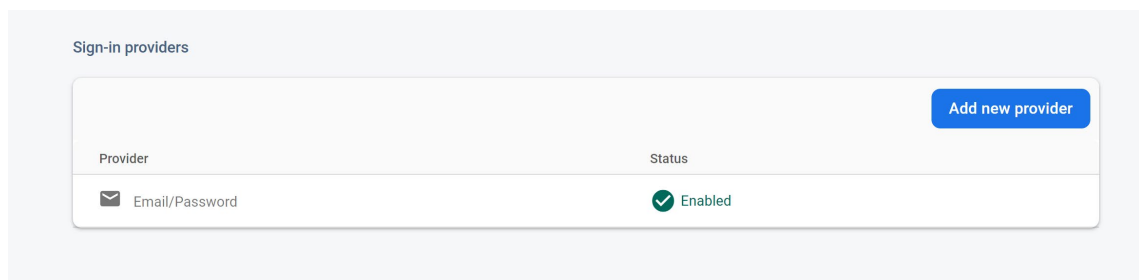
    override fun getCount() : Int = NUM_PAGES

    companion object{
        const val NUM_PAGES = 2
    }
}
```



14. ábra: Bejelentkező oldal

A bejelentkező oldal megírása előtt engedélyezni kell az email/jelszó bejelentkezési módszert a Firebase Konzolon, a FirebaseAuth szekciónál (15. ábra).



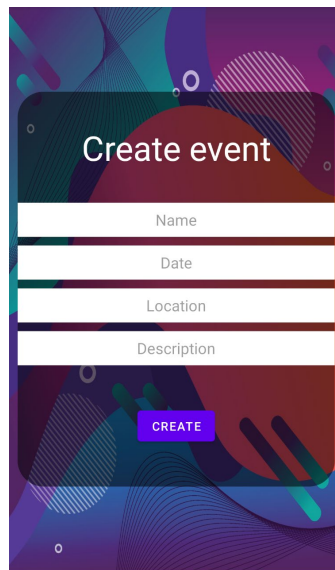
15. ábra: FirebaseAuth autentikációk

A *LoginFragment*-ben valósítom meg a bejelentkezést. A *LoginFragment* csak a nézetért felel. A logikai részt (bejelentkezés) a *LoginPresenter*-re bízom. Mivel a bejelentkezés egy adatbázis elérés, ezért a *LoginPresenter* ezt a feladatot rábízom a *LoginRepository*-ra aki meghívja a FirebaseAuth *signInWithEmailAndPassword* függvényét. Ezután az eredménytől függően (sikerrel járt-e) meghívja a *LoginPresenter* megfelelő függvényét.

```
fun login(email: String, password: String) {
    if (email.isEmpty() || password.isEmpty())
        return
    auth.signInWithEmailAndPassword(email, password)
        .addOnCompleteListener { task ->
            if (task.isSuccessful) {
                pres.loginSuccess()
            } else {
                pres.loginFailure()
            }
        }
}
```

A *LoginPresenter*-nek ekkor már csak annyi dolga van, hogy attól függően, hogy melyik függvényét hívta meg a *LoginRepository* meghívja a *LoginFragment startMainActivity*, vagy *showToast* függvényét. Előbbi a sikeres bejelentkezéskor hívódik, és elindítja a *MainActivityt* ami az alkalmazás fő oldala. Utóbbi rövid hibaüzenettel tájékoztatja a felhasználót a sikertelen bejelentkezési kísérletről. A felhasználóknak szánt üzeneteket Toast üzenetek formájában valósítom meg.

A *LoginActivity*-hez csatolt másik Fragment a *RegisterFragment*. Ez a regisztrációs oldal.



16. ábra: FirebaseAuth autentikációk

Működése nagyon hasonló a *LoginFragment/LoginPresenter/LoginRepository* hármashoz, de néhány dologban eltér. A *RegisterFragment* felel azért, hogy a felhasználó által megadott adatok helyesek legyenek. Ezt a *checkInputs* függvénnyel teszi meg, melyben ellenőrzi, és hiba esetén jelzi az adott beviteli mezőn, hogy a felhasználó:

- Megfelelő formátumú email címet használt-e
- A jelszó legalább 6 karakter (ez a minimum karakterszám a Firebase autentikációnál)
- A jelszó és a megerősítőjelszó egyezik-e
- Az email ki van-e töltve
- A jelszó ki van-e töltve

A *checkInputs* függvényt a *RegisterPresenter* hívja meg mielőtt meghívna a *RegisterRepository* *createUser* függvényét. A *RegisterRepository* osztály *createUser* függvénye amellet, hogy létrehozza a felhasználót a FirebaseAuth-ban a *FirebaseAuth.getInstance().createUserWithEmailAndPassword* függvény segítségével, sikeres létrehozás esetén a Presenter függvény hívása előtt még meghívja az *addUserToDB* függvényt.

```
private fun addUserToDB(){
    val user = auth.currentUser
    val newUser = User(email = user?.email)
    db.collection("Users").document(newUser.email!!).set(newUser)
}
```

Erre azért van szükség mert a felhasználóról több információt szeretnénk tárolni, mint amennyi a FirebaseAuth-ban lehetséges, ezért az adatbázisban készítünk számukra egy külön dokumentumot. Az *addUserToDB* függvény létrehoz a Firebase Firestore adatbázis **Users** nevű kollekciójában egy új dokumentumot, melynek az azonosítója a létrehozott felhasználó email címe lesz. Emellett a dokumentum email mezőjében eltárolja a felhasználó email címét. A **User** dokumentumok módosításához, illetve lekérdezéséhez tartozik egy *User* adatosztály, ami tartalmazza az összes **User** dokumentumban található mezőt.

```
data class User(var friends: ArrayList<Friend>? = null,
    var nickname: String? = null, var email: String? = null,
    var description: String? = null,
    var ratings: ArrayList<FriendRating>? = null)
```

Mivel a dokumentum azonosítója a felhasználó email címe, ezért a jövőben az email cím bírtokában könnyen meg lehet találni. A felsoroltak mellett a regisztrációs oldal hasonlóan működik a bejelentkező oldalhoz: sikeres regisztráció esetén az alkalmazás a *MainActivity*-re vált, sikertelen regisztráció esetén pedig rövid üzenetben értesítést kap erről a felhasználó.

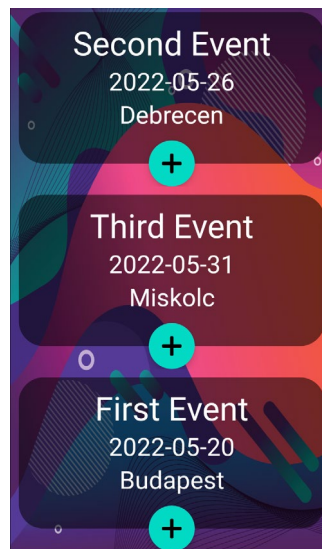
Miután bejelentkezett a felhasználó, a *FirebaseAuth.getInstance().currentUser* mezővel lehet lekérni az adatait. Ezen adatokból a legfontosabb a currentUser.email mező, amely a felhasználó email címét tartalmazza. Ezzel az email címmel azonosítom a felhasználót mind a Firestore adatbázisban, mind a Cloud Storage tárhelyben.

4.2 Főoldal

A felhasználó a bejelentkezést követően a főoldalra navigálódik. A főoldal a *MainActivity*-n keresztül van megvalósítva. Erre csak úgy, mint a *LoginActivity*-re, Fragmentekeket helyezek el. A *MainActivity*-n öt darab Fragment található.

4.2.1 Elérhető események

Az első Fragment az *AllEventFragment*. Itt a felhasználó azon eseményeket látja, melyeket nem ő készített, és feliratkozva sincs.



17. ábra: Elérhető események oldal

Az események az adatbázis **Events** kollekciójában tárolódnak. A dokumentumok módosításához és megjelenítéséhez létrehoztam egy *Event* adatosztályt, amely tartalmazza a dokumentum összes mezőjét, valamint az egyenlőség vizsgálat műveletét is felülírja, mivel minden eseményt a saját egyedileg létrehozott id mezője azonosít.

```
data class Event(var id: String? = null,
    var owner: String? = null,
    var name: String? = null,
    var date: String? = null,
    var desc: String? = null,
    var location: String? = null,
    var beThere: ArrayList<String>? = null){
    override fun equals(other: Any?): Boolean {
        if (other is Event)
            return other.id == this.id
        return false
    }
}
```

Az alkalmazásban több, különböző szűrőknek megfelelő eseményeket felsoroló lista található. Az összes ilyen lista az *EventListPresenter*-t és az *EventListRepository*-t hívja segítségül különböző paraméterekkel. Mivel ezek a listák nagyon hasonlóak az *AllEventListFragment*-nél, ezért csak egyszer tekintjük át a működésüket, és a jövőben csak az ettől történő eltérésekre fogunk kitérni.

Az esemény listák RecyclerView formájában vannak megvalósítva. A RecyclerView egy eleme egy esemény. Ezen az elemen megtalálható az esemény neve, dátuma, és helyszíne. Ezen kívül minden eseményen található egy gomb, amelynek háromféle kinézete és funkciója van. A gomb mivolta a jelenlegi felhasználó adataitól függ. Ha a felhasználó se nem az esemény szervezője, se nem résztvevője, a gomb ikonja egy pluszjel lesz, és rákattintva a felhasználó feliratkozik az eseményre. Ha a felhasználó az esemény rendezője, a gomb ikonja egy x jel lesz, és rákattintva törlődik az esemény. Az utolsó lehetőség, ha a felhasználó az esemény résztvevője, ekkor a gomb ikonja egy mínuszjel lesz, és rákattintva leiratkozik az eseményről. A típus választást az *EventPresenter.selector* függvénye oldja meg, amely egy *State* enumerációval tér vissza annak függvényében, hogy a felhasználó melyik feltételnek tesz eleget.

```
fun selector(event: Event): State {
    if (event.owner.equals(repo.currentUser))
        return State.DELETE
    if (event.beThere != null &&
        event.beThere!!.contains(repo.currentUser))
        return State.UNSUBSCRIBE
    return State.SUBSCRIBE
}
```

Emellett minden esemény teljes egésze kattintható, amely az esemény adatlapját nyitja meg az eseményhez tartozó adatok átadásával együtt. Az adatátadást az *EventListPresenter.frameClick* függvénye oldja meg, paraméterezett Activity hívással.

```
override fun frameClick(position: Int){
    var event = events.getEvent(position)
    val eventArray = ArrayList<String>()
    eventArray.add(event.owner ?: "missing")
    eventArray.add(event.name ?: "missing")
    eventArray.add(event.date ?: "missing")
    eventArray.add(event.desc ?: "missing")
    eventArray.add(event.location ?: "missing")
    holder?.openEventActivity(eventArray,event.beThere,true,event.id,
        event.owner == repo.currentUser ||
        event.beThere?.contains(repo.currentUser) == true
    )
}
```

A RecyclerView adapterét (*MainListAdapter*) speciálisan kellett megoldanom, az MVP szemlélet mód betartásáért. Az adapter egy *EventListPresenter*-t kap paraméterül. Az *EventListPresenter* tartalmaz egy *EventListModel*-t, ami egy *Event* típusú listát tesz elérhetővé. Az adapter *onBindViewHolder* meghívásakor meghívja az *EventListPresenter* *bindViewHolder* függvényét, ami az *EventListModel*-en keresztül eléri az events listát. Ezután eszerint meghívja az Adapter *MyViewHolder* belső osztályának megfelelő függvényét, hogy az beállítsa az adott értékét a nézet egy elemére.

```

override fun onBindViewHolder(holder: MyViewHolder, position: Int) {
    eventListPres.bindViewHolder(holder, position)
    holder.subButton.setOnClickListener {
        eventListPres.subBtnClick(position)
    }
    holder.frame.setOnClickListener {
        eventListPres.frameClick(position)
    }
}

override fun bindViewHolder(holder: IEventListContract.IEventAdapter,
position: Int) {
    var event = events.getEvent(position)
    this.holder = holder
    holder.setName(event.name)
    holder.setDate(event.date)
    holder.setLocation(event.location)
    holder.setSubBtnImg(selector(event))
}

```

Azért, hogy az *EventListPresenter* könnyedén elérhesse a *MyViewHolder* osztályt, ez megvalósítja az *IEventListContract.IEventAdapter* interfészt. A kattintás figyelőket az Adapterben állítom rá az adott elemre, de a logikát az *EventPresenter* megfelelő függvényére bízom.

```

inner class MyViewHolder(itemView: View) :
    RecyclerView.ViewHolder(itemView), IEventListContract.IEventAdapter { ... }

```

A kattintásnál, ha nézetet kell állítani, akkor az *EventPresenter* ugyancsak az adapter *MyViewHolder* megfelelő függvényét fogja hívni a megfelelő paraméterrel. Ezzel a módszerrel teljesen szétválasztom a nézetet a logikától, így eleget teszek az MVP architektúra követelményeinek.

Az esemény listát az adatbázisból töltöm fel. Ehhez az *EventListPresenter* segítségével hívja az *EventListRepository* megfelelő függvényét. Az adott függvény

oldalanként különbözik, hiszen egyes oldalak más-más szűrési feltételnek eleget tevő eseményekből képeznek listát.

```
fun allEventChange(){
    eventDB.whereNotEqualTo("owner",
    auth.currentUser?.email).addSnapshotListener { value, _ ->
    pres.onDataChange(value) }
}
```

Az *AllEventFragment* oldal e tekintetben speciális megoldást kíván, ugyanis a Firestore adatbázisból történő lekérdezésnél nem lehet olyan szűrési feltételt megadni, hogy nem tartalmazza a résztvevők listája a jelenlegi felhasználót. Ezért ennél az oldalnál az *EventListRepository*-ban csak arra szűrök, hogy a jelenlegi felhasználó az esemény rendezője-e, a másik feltételre (hogy a jelenlegi felhasználó a résztvevők között van-e) csak később az *EventListPresenter*-ben. Az *addSnapshotListener* függvény meghívódik valahányszor az adatbázisban az adott feltételnek eleget tevő dokumentumban változás történik. Ekkor az *EventListRepository* meghívja az *EventListPresenter* megfelelő függvényét, ami ellenőrzi, hogy az adatbázisban történt változás hozzáadás, törlés, vagy módosítás volt-e, és ehhez mérten módosítja az esemény listát.

Az *AllEventFragment*-en kilistázott eseményeken található gombok feliratkoztatják a felhasználót az eseményre. Ekkor az adott esemény résztvevői közé bekerül a jelenlegi felhasználó, valamint meghívódik az *EventListRepository* által megvalósított *IFriend* interfész *addFriend* függvénye, egyszer a jelenlegi felhasználóval és egyszer az esemény rendezőjével. Az *addFriend* függvény az adatbázisban frissíti az egyik felhasználó barátlistáját a másik felhasználóval. Ha még nem volt rajta a friends barátlistán akkor felkerül rá, és a kapcsolati szintjük egy lesz, ha már rajta van a listán, akkor eggyel nő a kapcsolati szintjük. A kapcsolati szintről részletesebben a Barátok oldal leírásánál olvashatunk. Emellett, ha a createChat paramétere igaz értéket kap, akkor létrehoz egy új dokumentumot a **Chat** kollekcióban. A dokumentum neve a megadott felhasználók email címe egy X karakterrel összekapcsolva (user1Xuser2). Így a jövőben a két felhasználó email címével könnyedén beazonosíthatóvá válik a hozzájuk tartozó **Chat** dokumentum.

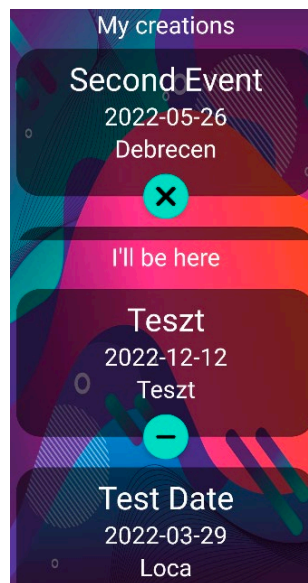

```

fun subToEvent(event: Event){
    if (event.beThere == null) {
        event.beThere = ArrayList()
    }
    event.beThere?.add(auth.currentUser?.email!!)
    val updEvent = db.collection("Events").document(event.id ?: "")
    updEvent.update("beThere", event.beThere)
    AddFriend(auth.currentUser?.email!!, event.owner!!, true)
    AddFriend(event.owner!!, auth.currentUser?.email!!)
}

```

4.2.2 Eseményeim oldal

A második *MainActivity*-re csatolt oldal a *MyEventFragment*. Ez a Fragment valósítja meg az aktuális felhasználó saját eseményeit tartalmazó oldalt. Az oldalon két lista található, az egyik a felhasználó által létrehozott eseményeket tartalmazza, a másik pedig azon eseményeket, amelyeken a felhasználó részt vesz. Mindkét lista hasonlóan épül fel az *AllEventFragmenten* található listához.



18. ábra: Eseményeim oldal

A fenti lista a felhasználó által létrehozott eseményeket tartalmazza. Azért, hogy a lista a megfelelő eseményekből álljon, paraméteresen hívom meg az *EventListPresenter* *eventChangeListener*-jét, ami így az *EventListRepository* *myEventChange* függvényét fogja meghívni.

Az *EventListRepository* *myEventChange* függvénye az Esemény kollekció azon eseményeire állít fel figyelőt, melyeknél az esemény létrehozója megegyezik a jelenlegi felhasználóval.

```

fun myEventChange(){
    eventDB.whereEqualTo("owner",
    auth.currentUser?.email).addSnapshotListener { value, _ ->
    pres.onDataChange(value,1) }
}

```

Ezen eseményekkel történő változás után ugyanaz az *onDataChange* függvénye fog lefutni az *EventListPresenter*nek mint a többi esemény listánál, de ez is paraméteresen, hogy a legvégén a megfelelő adaptert értesítse a *MyEventFragment*.

```

override fun updateAdapter(pickAdapter: Int?) {
    if(pickAdapter == 1)
        adapterCreated.notifyDataSetChanged()
    else if(pickAdapter == 2)
        adapterLiked.notifyDataSetChanged()
}

```

A felhasználó által létrehozott eseményeken található gomb egy X ikonnal van ellátva. Ha erre rákattint a felhasználó, az eseménye törlődik az adatbázisból. Mivel a többi lista *SnapshotListener* figyelővel figyeli az adatbázisban történt változásokat, az esemény törlődni fog a megjelenített listákból is. A gombra kattintva az *EventListPresenter* az *EventListRepository deleteMyEvent* függvényét fogja meghívni, amely az adatbázis **Events** kollekcijából törli azt az eseményt, amelynek az azonosítója megegyezik a megadott eseményével.

```

fun deleteMyEvent(event: Event){
    db.collection("Events").document(event.id ?: "").delete()
    if(event.beThere != null) {
        for (wasThere in event.beThere!!) {
            RemoveFriend(auth.currentUser?.email!!, wasThere, true)
            RemoveFriend(wasThere, auth.currentUser?.email!!, true )
        }
    }
}

```

Emellett ciklikusan meghívja az *IFriend* interfész *removeFriend* függvényét az összes résztvevővel, egyszer a felhasználóval és az esemény résztvevőivel, egyszer pedig az esemény résztvevőivel és a felhasználóval. A *removeFriend* függvény eggyel csökkenti a paraméterként kapott felhasználók kapcsolat-pontját, ha pedig ez eléri a nullát, törli az egyik felhasználót a másik barátlistájáról. Emellett, ha a chatet is törölni kell, akkor törli a Chat kollekciónál a felhasználóhoz tartozó dokumentumot. Mivel nem tudhatjuk, hogy milyen sorrendben szerepel a két felhasználó neve az adatbázisban, mind a két sorrendben

meghívom a dokumentum törlő függvényt, így biztosan eltávolítódik a **Chat** dokumentum.

Az oldalon a másik lista azokat az eseményeket tartalmazza, amelyekre a felhasználó feliratkozott. Az *EventListRepository* *beThereEventChange* függvénye felel a megfelelő események adatbázisból való betöltésért.

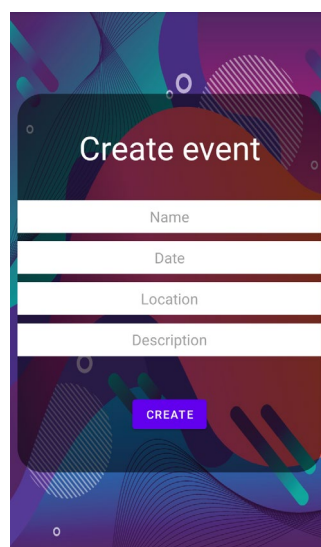
```
fun beThereEventChange(){
    eventDB.whereArrayContains("beThere",auth.currentUser?.email.toString()).addSnapshotListener { value, _ -> pres.onDataChange(value,2) }
}
```

A listában található eseményeken lévő gomb mínuszjel ikonnal van ellátva, és kattintásra törli a felhasználót az esemény résztvevőket tároló listájáról, az *EventListRepository* *unsubFromEvent* függvényének segítségével.

```
fun unsubFromEvent(event: Event){
    event.beThere?.remove(auth.currentUser?.email)
    val updEvent = db.collection("Events").document(event.id ?: "")
    updEvent.update("beThere", event.beThere)
    RemoveFriend(event.owner!!, auth.currentUser?.email!!, true)
    RemoveFriend(auth.currentUser?.email!!, event.owner!!, true)
}
```

4.2.3 Esemény készítő oldal

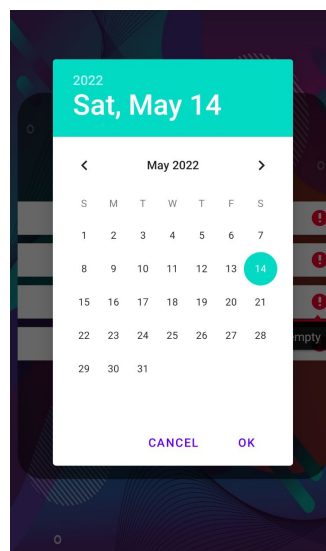
A *MainActivity*-n található harmadik Fragment a *CreateEventFragment*, melyet a 19. ábra szemléltet.



19. ábra: Esemény készítő oldal

Az oldalon a felhasználónak lehetősége van új esemény létrehozására. A *CreateEventFragment* a megjelenítésen felül az egyes beviteli mezők ellenőrzéséért is felelős. Az egyes beviteli mezők nem lehetnek üresek, valamint a dátum nem lehet múltbéli. Utóbbi elérésében segítséget nyújt a felhasználónak a dátum választó, amely akkor nyílik meg, ha a felhasználó a dátum beviteli mezőre kattint (20. ábra). Ezt a *CreateEventPresenterben* inicializálom és a *CreateEventFragmentben* egy *DatePickerDialog*-gal oldok meg.

```
override fun initDatePicker() {  
    val calendar = Calendar.getInstance()  
    createEventView.openDatepicker(calendar[Calendar.YEAR],calendar[Calendar.MONTH],calendar[Calendar.DAY_OF_MONTH])  
}
```



20. ábra: Dátum választó

A create gombra kattintva, a beviteli mezők megfelelő kitöltésének ellenőrzése után, az alkalmazás létrehozza az új eseményt az adatbázisban, ami az adatbázist figyelő függvényeknek köszönhetően frissíti az eseményeket tároló listákat is. A létrehozás előtt az *EventListPresenter* készít egy új **Event** típusú változót, amely egy olyan adatosztály, amely az adatbázis **Events** kollekciójában található dokumentumok mezőit tartalmazza.

Az új **Event** változó mezőértékeit beállítom a beviteli mezőben megadott adatokra, valamint egy új egyedi azonosítót is létrehozok, és ezt beállítom az **Event** id mezőjébe. Ezután meghívódik az *EventCreatorRepository* *createEvent* függvénye, amely először az esemény tulajdonos mezőjébe beállítja a jelenlegi felhasználót, majd létrehoz egy új dokumentumot az **Events** adatbázis kollekcióban, és egyet a **Chat** kollekcióban. Mindkét

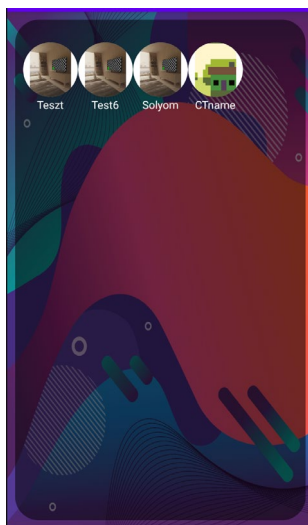
dokumentum azonosítója az esemény azonosítója lesz, így a jövőben könnyen elérhetők. A **Chat** kollekcióban létrehozott dokumentum, az eseményhez tartozó csoport beszélgetés üzeneteit hivatott tárolni. Végül meghívódik a *CreateEventFragment* *finishCreation* függvénye amely egy rövid üzenettel jelzi a felhasználónak az esemény létrehozásának befejeztét.

```
override fun create(name: String, date: String, desc: String, location:
String) {
    if (createEventView.checkInput()) {
        var uniqueID = UUID.randomUUID().toString()
        var newEvent = Event(id = uniqueID, name = name, date = date, desc
= desc, location = location)
        repo.createEvent(newEvent)
        createEventView.finishCreation()
    }
}

fun createEvent(event: Event){
    event.owner = auth.currentUser?.email
    var db = FirebaseFirestore.getInstance()
    db.collection("Events").document(event.id!!).set(event)
    db.collection("Chat").document(event.id!!).set(Chat(ArrayList()))
}
```

4.2.4 Közösségi oldal

A *MainActivity*-re csatolt negyedik Fragment a *PeopleFragment*. Itt egy lista található a jelenlegi felhasználó barátairól. Barát kategóriába sorolandó egy felhasználó, ha a másik felhasználó eseményén részt vesz, vagy ő szervezi az eseményt amelyiken a másik felhasználó részt vesz.



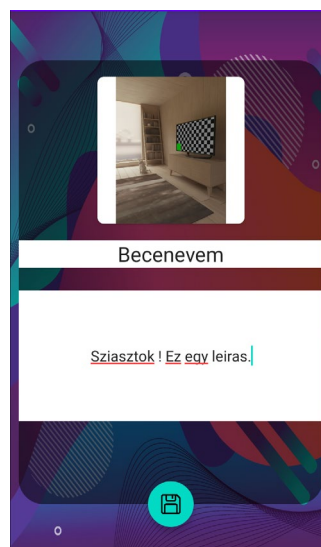
21. ábra: Közösségi oldal

A lista egyes elemei az adott barát profilképét tartalmazzák, vagy ennek hiányában egy profilkép-helyettesítő alapértelmezett képet. Emellett az adott barát beceneve is megjelenik, ha van neki. A képre kattintva a barát profil adatlapja nyílik meg.

A lista nagyon hasonlóan épül fel az eseményeket tartalmazó listához, viszont az **Events** kollekció helyett a felhasználó friends tömbjében található felhasználókból épül fel. A friends tömb **Friend** adatosztály típusú változókat tárol, melynek két mezője van: egy name és egy connectCnt. Előbbi a barát nevét tárolja, amely segítségével a **Users** kollekcióból kinyerhető a barát többi adata. A connectCnt változó a kapcsolat szintjét hivatott tárolni, minden közös eseménynél eggyel nő, valamint a felhasználók profilján a „maradjunk barátok” kapcsoló igazra billentésével is növelhető. A barát csak akkor törlődik a barátlistáról, ha a kapcsolat-szintjük eléri a nullát. Így hiába törlődik olyan esemény, amelyen együtt részt vesznek, ha van még más közös eseményük, vagy az egyik fél tovább szeretne beszélgetni a másikkal, a két felhasználó egymás barát listáján maradhat.

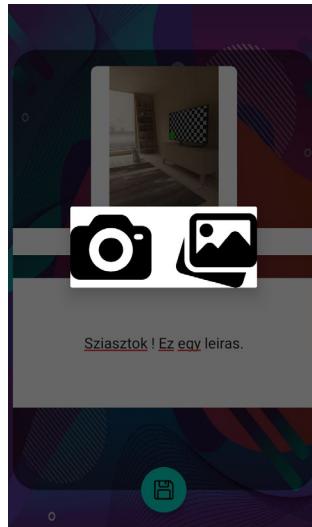
4.2.5 Saját Profil Oldal

A *MainActivity* ötödik Fragmentjén, a *MyProfileFragmenten*, a felhasználó személyre szabhatja a profilját. Profilképét, Becenevet és leírást állíthat be.



22. ábra: Saját profil oldal

A profilkép beállítása a képre kattintva érhető el. Ekkor egy felugró ablak jelenik meg, ahol a felhasználó eldöntheti, a kamera alkalmazással szeretne képet készíteni, vagy a gallériából választana a már meglévő képei közül.



23. ábra: Kép felvétele opciók

A kamera indításához a *PopUpPresenter* címként beállítja a felhasználó email címét .jpg kiterjesztéssel, majd meghívja a *PopUpFragment openCamera* függvényét.

```
override fun initCamera(){
    val fileName = repo.auth.currentUser?.email + ".jpg"
    values.put(MediaStore.Images.Media.TITLE, fileName)
    values.put(MediaStore.Images.Media.DESRIPTION, "Image capture by
camera")
    popUpView.openCamera(values)
}
```

A *PopUpFragment openCamera* függvénye egy Uri-t gyárt a Presentertől kapott adatokból, és azt extraként átadja a kamera oldalnak, majd elindítja azt.

```
override fun openCamera(values: ContentValues){
    imageUri = requireActivity().contentResolver.insert(
        MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
        values
    )
    val intent = Intent(MediaStore.ACTION_IMAGE_CAPTURE)
    intent.putExtra(MediaStore.EXTRA_OUTPUT, imageUri)
    startActivityForResult(intent, 1231)
}
```

Ha a felhasználó a galériából választana képet, a galéria ikonra kattintva lefut a klikkelés figyelő, ami megnyitja a galériát.

```
binding.galleryBtn.setOnClickListener{
    var openGalleryIntent =
    Intent(Intent.ACTION_PICK,MediaStore.Images.Media.EXTERNAL_CONTENT_URI)
    startActivityForResult(openGalleryIntent,1000)
}
```

Mindkét mód után lefut az *onActivityResult* figyelő, ami ha a galéria után hívódott, még beállítja az Uri-t a választott képre, majd mindenképp lefut a *PopUpPresenter* Uri paraméterrel felvértezett *uploadPicture* függvénye.

```
override fun onActivityResult(  
    requestCode: Int,  
    resultCode: Int,  
    data: Intent?  
) {  
    super.onActivityResult(requestCode, resultCode, data)  
    if (requestCode == 1000 && resultCode == Activity.RESULT_OK) {  
        imageUri = data?.data  
    }  
    (parentFragment as MyProfileFragment).setProfPic(imageUri)  
    popUpPres.uploadPicture(imageUri)  
}
```

Ez a függvény meghívja a *PopUpRepository* *uploadImage* függvényét, amely a Cloud Storage-ba elmenti a képet a felhasználó email címe + Profile.jpg címmel. Ezáltal amikor szükség van rá a felhasználó email címével könnyedén elérhető.

```
fun uploadImage(imageUri: Uri?){  
    if(imageUri != null)  
        storage.reference.child(auth.currentUser?.email +  
            "Profile.jpg").putFile(imageUri)  
}
```

A felhasználó profilja beállítása végeztével rákattint a mentés gombra, amely frissíti a változtatásokat az adatbázis **Users** kollekciójának a felhasználó nevével ellátott dokumentumában.

```
fun update(desc: String, name: String){  
    profile?.update("description", desc)  
    profile?.update("nickname", name)  
}
```

Azért, hogy az oldalt megnyitva mindig a legfrissebb felhasználói adatok szerepeljenek, a *MyProfileRepository* *getUserData* függvényében beállított, a **Users** kollekció felhasználóhoz tartozó dokumentumára beállított figyelő felelős.

```
fun getUserData(){  
    profile = db.collection("Users").document(auth.currentUser?.email!!)  
    profile?.addSnapshotListener(EventListener<DocumentSnapshot> { value,  
        - ->  
            val myUser = value?.toObject(User::class.java)
```



```

        pres.setUserData(myUser)
    })
}

```

4.3 Esemény adatlap

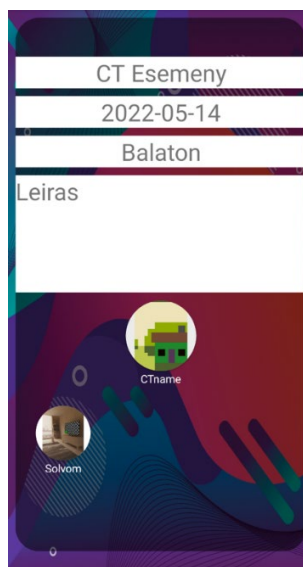
Az esemény oldal az *EventActivity*-n található, melyre kétféle adapter csatlakoztatható. Egy, az Activity hívásakor átadott logikai változó dönti el, melyik Adapter kötődik az Activity-hez. A két adapter között a különbség, hogy az *EventAdapter* két Fragmentet tartalmaz, az *EventFragment*-et és a *ChatFragment*-et. Ezzel szemben az *EventWOChatAdapter* csak az *EventFragment*-et. Ezáltal azon eseményeknél, amelynél a felhasználó jogosult látni a Chat oldalt, az *EventAdapter* csatlakozik az *EventActivity*-re. Azon eseményeknél pedig, amiknél nem jogosult, az *EventWOChatAdapter* csatlakozik.

```

val extras = this.intent.extras
var chatAvailable = extras?.getBoolean("chatAvailable")
if(chatAvailable == true)
    binding.vpMain.adapter = EventAdapter(supportFragmentManager)
else
    binding.vpMain.adapter = EventWOChatAdapter(supportFragmentManager)

```

Az esemény adatlap előhívható bármely eseményre kattintva, és az adott eseményről tartalmaz bővebb infomációt.



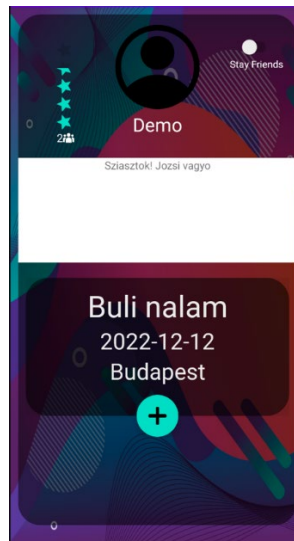
24. ábra: Esemény adatlap oldal

Az esemény adatai az *EventActivity* hívásakor adódik át. A leírás alatt található nagyobb profilkép az esemény létrehozójához tartozik, rákattintva a felhasználó profilja nyílik

meg. A létrehozó alatt egy PeopleList van, pont mint amilyen a *PeopleFragmenten* is, csak itt nem a felhasználó barátlistájából tevődik össze hanem az esemény résztvevőiből. Ehhez az *EventPresenter* tartalmaz egy *PeoplePresenter* változót, és a RecyclerView-nak ezt a változót adja át.

4.4 Felhasználói adatlap

A felhasználói adatlap a *ProfileActivity*-n található, és sokban hasonlít az esemény adatlaphoz. Erre az Activity-re is kétféle adapter kapcsolható. A *ProfileAdapter* a *ProfileFragment* mellett a *ChatFragmentet* is kezeli, a *ProfileWOChatAdapter* csak a *ProfileFragment*-et teszi elérhetővé. Ha az adatlapot a közösségi oldalról nyitják meg, akkor a Chat oldal és a “maradjunk barátok” kapcsoló is elérhető, tehát ilyenkor a *ProfileAdapter* kapcsolódik a *ProfileActivity*hez.



25. ábra: Felhasználói adatlap

A felhasználói adatlap (25. ábra) megjeleníti a felhasználó nyilvános adatait. Látható a profilképe, melyet a korábban bemutatott módon a Cloud Storageból tölt le az alkalmazás, az adott felhasználó email címe segítségével. A kép alatt található a felhasználó beceneve, ezalatt a leírása, majd alatta egy lista az általa létrehozott eseményekről. Az esemény lista ugyanúgy épül fel, mint az alkalmazásban található többi esemény lista, viszont ez nem függőleges, hanem vízszintes irányban járható be. Mivel az eseményeken található gombok dinamikusan változnak a jelenlegi felhasználó adott eseménnyel való kapcsolatának függvényében, ezért ebben a listában felváltva szerepelhetnek olyan események, amelyekre a felhasználó feliratkozhat, és olyan

események, amelyekről leiratkozhat. A listához a *ProfilePresenter* egy *EventListPresenter* változót tárol, amit átad a RecyclerViewnak.

Emellett a felhasználói adatlapon található még egy csillag alapú értékelő rendszer, és egy “maradjunk barátok” kapcsoló. Előbbi fél csillagos pontossággal történő értékelést tesz lehetővé. Az értékelés alatt található egy számláló, mely megmutatja a felhasználót értékelők számát. Mindkét elem az android Shared Preferences módszerével tölti be az adatokat a Fragment indulásakor. Ezzel a módszerrel perzisztensen lehet adatokat tárolni kulcs-érték párokban az eszközön. Erre különböző okok miatt van szükség. Az értékelések elemnél két kulcs érték párt használok. Az egyik egy Float típusú, a felhasználó email címe + Rating kulccsal, melyből az értékeléseket nyerem ki. A másik egy String típusú a felhasználó email címe + Reviewers kulccsal, mellyel az értékelők számát mentem el. Erre azért van szükség, hogy ne kelljen mindig újraszámolni az értékelést, és ezzel felesleges adatbázisforgalmat generálni. Mivel a felhasználó értékeléseire egy SnapshotListener figyel, ezért ha itt változás történik a *ProfilePresenter* *setRating* függvényében újra számolódik az értékelés (a összes meglévő értékelés átlaga számolódik ki), majd elmentődik a megosztott preferenciába. Az értékelés csak akkor módosítható, ha az adatlapot a közösségi oldalról nyitottuk meg, ezzel garantálva, hogy csakis olyan személy értékelhet aki valamilyen kapcsolatban áll a felhasználóval. Ha a jelenlegi felhasználó már korábban értékelte a felhasználót, nem új értékelés keletkezik, hanem a már korábbi értékelését cseréli le az újra.

```
profilePres.autoChangeRating(sharedPref.getFloat(friendData[0] + "Rating",  
0.0f))  
binding.reviewers.text = sharedPref.getString(friendData[0] +  
"Reviewers", "0")
```

A “maradjunk barátok” kapcsoló igaz értékre billentve eggyel növeli az adott baráthoz tartozó kapcsolati szintet az adatbázisban a *ProfilePresenter* *bffSwitchChecked* függvény segítségével, mely meghívja az *ProfileRepository* *addFriend* függvényét, amely az adatbázis módosításáért felel. Mivel mindkét fél be tudja billenteni a kapcsolót, összesen kettővel növelhető a kapcsolati szint. Ezzel el tudtam érni, hogy ha az egyik fél szeretné a közösségi oldalán látni a másik felet, azt mindenképpen megtehesse. A kapcsoló hamisra billentésével eggyel csökken a kapcsolati szint.

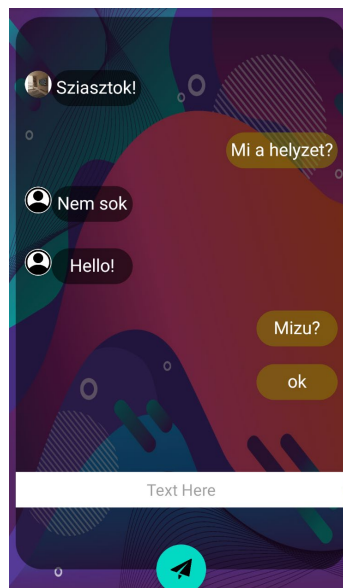
A lényegi részt a *IFriend* interfész *removeFriend* függvénye végzi, amely eggyel csökkenti a kapcsolati szintet, és ha ez eléri a nullát, akkor törli a barátot a barátlistáról és a baráthoz

tartozó Chat dokumentumot az adatbázisból. Mivel a kapcsoló mást nem csinál, csak a kapcsolati szintet változtatja, nem lehet eldönteni, hogy korábban igazra volt-e állítva vagy sem. Ezért mentem el az állapotát perzisztensen egy Boolean változóba, melyet felhasználók email címével azonosítok. Így már a Fragment létrehozásakor, könnyedén beállíthatom a kapcsolót a megfelelő állapotára.

```
val editor = sharedPref.edit()
editor.putBoolean(name + "X" + friendData[0], isBff)
editor.apply()
```

4.5 Chat oldal

A Chat oldalon a felhasználóknak lehetősége van egymással kommunikálni. Az oldal kétféle verzióban érhető el. Az események adatlapról, ha a felhasználó részt vesz az eseményen, jobbra húzva elérheti a Chat oldalt, ahol egy csoportbeszélgetés található az esemény résztvevőivel. Másrészt a profil adatlapról ugyancsak elérhető egy Chat oldal, ha az adatlapot a közösségi oldalról nyitották meg. Itt privát beszélgetést lehet folytatni az adott felhasználóval. Amikor egy adatlapnak van Chat oldala, megnyitáskor egy rövid üzenettel jelzi az alkalmazás a felhasználónak, hogy jobbra húzással elérheti azt.



26. ábra: Chat oldal

A Chat betöltése hasonló a két verziónál, azonban van egy kis eltérés is közöttük. A *ChatPresenter* *initMessages* függvénye adja át a *ChatRepository*-nak a megfelelő dokumentum-nevet.

```

override fun initMessages(category: Int, id: String) {
    if (category == 1) {
        repo.getMessages(id + "X" + repo.auth.currentUser?.email!!)
        repo.getMessages(repo.auth.currentUser?.email!! + "X" + id)
    } else if (category == 2) {
        repo.getMessages(id)
    }
}
}

```

A category paraméterrel dönti el, hogy csoportos, vagy privát beszélgetést kell-e betölteni. A privát beszélgetésnél mindkét sorrendben meghívom, mivel nem tudható, hogy a Chat dokumentum azonosítója a felhasználók nevének melyik sorrendjéből tevődik össze. A függvény második paramétere az id, ez privát beszélgetésnél a felhasználó neve lesz, csoportos beszélgetésnél pedig az esemény azonosítója.

A *ChatRepository* *getMessages* függvénye letölti az adatbázisból a **Chat** kollekció azon dokumentumát amely azonosítója megegyezik a paraméterként kapott szöveggel. Az adatbázisból szerzett adatot egy **Chat** típusú változóba mentem el, amely egy **Message** típusú listát tartalmaz. A **Message** változónak két mezője van: a text, ami az üzenetet tartalmazza, és a sender, ami pedig a küldő email címét. Ezután meghívódik a *ChatPresenter* *showMessages* függvénye, amely frissíti az üzeneteket tároló listát, és értesíti a *ChatFragment*-et, hogy görgessen a lista aljára, így mindig a legfrissebb üzenetek látszódnak.

```

fun showMessages(msgs: Chat){
    messages.clearMessages()
    for (msg in msgs.Messages!!) {
        messages.addMessage(msg)
        chatView.updateAdapter()
    }
    chatView.scrollToBottom(messages.getMsgCnt() - 1)
}

```

A lista egyes elemeinek betöltődésekor meghívódik a *ChatPresenter* *bindViewModel* függvénye, amely beállítja az üzenet szövegét, valamint a küldő becenevét, illetve profilképét. Utóbbit a Cloud Storage-ból tölti le a korábban bemutatott módon. Ezek után attól függően, hogy a jelenlegi felhasználó-e az üzenet küldője meghívódik a *ChatAdapter* *MyViewHolder* belső osztályának *leftSide* vagy *rightSide* függvénye.

A *leftSide*, *rightSide* függvényekben a képernyő bal, illetve jobb oldalára pozicionálom az üzenetet, valamint szürke, illetve sárga hátteret állítok be neki. Ezáltal a felhasználó könnyebben megkülönböztetheti saját üzeneteit másokétól.

```

override fun rightSide(){
    params.endToEnd = PARENT_ID
    params.startToStart = ConstraintLayout.LayoutParams.UNSET
    frame.setBackgroundResource(R.drawable.messageframe)
    senderPic.isVisible = false
    senderName.isVisible = false
}

override fun leftSide(){
    params.endToEnd = ConstraintLayout.LayoutParams.UNSET
    params.startToStart = picAndName.id
    senderPic.isVisible = true
    senderName.isVisible = true
    frame.setBackgroundResource(R.drawable.roundedtextview)
}

```

A Chat oldal alján található beviteli mezőbe a felhasználó beírhatja az üzenetét és a küldés gombra kattintva a *ChatRepository uploadMessage* függvénye feltölti az adatbázisba az új üzenetet. Mivel a dokumentumra figyelő van beállítva, a feltöltés után automatikusan frissülni fog az üzeneteket tartalmazó lista.

5 Tesztelés

Az alkalmazás fejlesztés során fontos az alkalmazás tesztelése, ezáltal lehet minimalizálni a hibás program kiadásának esélyét. Az alkalmazást egységtesztekkel fedtem le, amely olyan automatikusan kiértékelhető teszt-típus, ami egyszerre csak az alkalmazás egy kis, egységnyi részére (egy komponensre) fókuszál. Ennek előnye, hogy fejlesztésnél könnyen kiderülhet, ha egy módosítás okozott hibát a programban.

5.1 Teszt lefedettség

Az egységteszteknel fontos metrika a kód lefedettség. Ezzel mérhető, hogy a kód mekkora részét teszteljük a tesztekkel. Az általam írt tesztek kód lefedettségét a 27. ábra és a 28. ábra mutatja be. Mivel az általam írt logika a Presenterekben található, így a tesztek célzottan a Presenterekre fókuszáltak. Látható, hogy az osztályok 100%-át lefedik a tesztek, valamint a sorok 86%-át, tehát ezen osztályokra magas a teszt lefedettség.

A Model-ben található osztályok függvényei nagyon egyszerűek, így azokat nem teszteltem külön, de mivel a Presenterek használják őket itt is automatikusan magas lett a lefedettség (a sorok 87% tesztelve van). A Repository-kon és az Interfaces IFriend interfészen keresztül használtam a Firebase szolgáltatásait, ezeket a függőségmentes tesztek érdekében nem teszteltem. A View-ban találhatóak a megjelenítésért felelős osztályok, amelyek az Android beépített függvényeit használják, ezért hasonló okokból fakadóan ezeket sem teszteltem. Azon osztályok függvényeit, amelyeket nem tesztelem, de mégis használnom kellett a tesztek során a Mockito [13] keretrendszer segítségével Mockoltam. Ez azt jelenti, hogy a valós osztály helyett egy imitációt hoztam létre, melynek függvény-hívásainál manuálisan beállítható, hogy mi történjen. Ezzel elkerülhettem, hogy a tesztek Függőségben álljanak a Firebase-től, vagy az Android függvényeitől, hiszen az egységtesztek csak a komponensek önálló működését hivatottak ellenőrizni. A teszteléshez a Mockito és a JUnit4 [14] keretrendszereket használtam.

Element	Class, %	Method, %	Line, %
Interfaces	33% (1/3)	25% (1/4)	9% (1/11)
Model	90% (10/11)	86% (20/23)	87% (35/40)
Presenter	100% (9/9)	75% (42/56)	86% (189/2...
Repository	0% (0/9)	0% (0/41)	0% (0/119)
View	0% (0/33)	0% (0/150)	0% (0/517)
BuildConfig	0% (0/1)	0% (0/1)	0% (0/1)

27. ábra: Teszt lefedettség

Element	Class, %	Method, %	Line, %
ChatPresenter	100% (1/1)	75% (6/8)	87% (29/33)
CreateEventPresenter	100% (1/1)	50% (2/4)	66% (10/15)
EventListPresenter	100% (1/1)	80% (8/10)	94% (64/68)
EventPresenter	100% (1/1)	85% (6/7)	91% (21/23)
LoginPresenter	100% (1/1)	100% (4/4)	100% (10/10)
MyProfilePresenter	100% (1/1)	83% (5/6)	86% (13/15)
PeoplePresenter	100% (1/1)	50% (5/10)	72% (26/36)
PopUpPresenter	100% (1/1)	66% (2/3)	75% (6/8)
RegisterPresenter	100% (1/1)	100% (4/4)	100% (10/10)

28. ábra: Teszt lefedettség az egyes Presentereknekél

5.2 Tesztek Felépítése

A legtöbb teszt hasonlóan épül fel. A teszt osztály tartalmazza a tesztelendő Presentert, valamint a Mockolt View-t, és Repository-t.

```
@RunWith(MockitoJUnitRunner::class)
class RegisterPresenterTest{
    private lateinit var presenter: RegisterPresenter
    @Mock
    private lateinit var view: IRegisterContract.IRegisterView
    @Mock
    private lateinit var repo: RegisterRepository
```

A teszt Before fázisában létrehozom a Presentert, a konstruktorába átadom a két Mockolt elemet.

```
@Before
fun setUp(){
    MockitoAnnotations.initMocks(this)
    presenter = RegisterPresenter(view,repo)
}
```

A teszteknél azt figyelem, hogy a Presenter osztály egyes függvényhívásai a megfelelő változtatásokat léptetik-e életbe. Ezért az ellenőrzésnél azt ellenőrzöm, hogy végül

meghívódott-e vagy a View vagy a Repository egy adott függvénye a megfelelő paraméterekkel.

```
@Test
fun testUserCreationSuccess(){
    doAnswer { presenter.successfulCreation()
}.`when`(repo).createUser(email, password)
    `when`(view.checkInputs()).thenReturn(true)
    presenter.create(email, password)
    verify(view).startMainActivity()
}
```

Természetesen a fentieket minden teszt személyre szabja, és kiegészíti újabb elemekkel, azonban mindegyik ezekre az alapokra épül.

5.3 Nehézségek

A tesztelés során több nehézségbe is ütköztem, melyeket nem volt triviális megoldani. Ebben az alfejezetben ezeket a problémákat mutatom be a választott megoldásokkal együtt.

5.3.1 MVP + Repository

Az alkalmazás kezdeti architektúrája egyszerű (Repository-mentes) MVP volt, és a Presenterben voltak megvalósítva a Firebase függvények. Sajnos ez a tesztelésnél rengeteg fejtörést okozott, mivel nagyon sok Firebase függőséggel rendelkeztek a tesztelendő osztályok. Emiatt vezettem be az architektúrába a Repository-kat, amelyekben a Presentertől elkülönítve érhetőek el a Firebase szolgáltatásai. A legtöbb Firebase-es függvény azonban aszinkron működésű, így nem volt lehetséges egyszerűen csak visszatérni a függvény végén az eredménnyel. Emiatt a Presenterek callback függvényeket tartalmaznak, amelyeket meghívhatják a Repository-k az aszinkron Firebase függvények lefutása után. Ezekkel a változtatásokkal már csak annyi a dolgom a tesztelésnél, hogy Mock-olom a Repository-t, és beállítom, hogy a függvényhívásakor egyszerűen csak hívja meg a Presenter azon függvényét egy megadott paraméterrel, amit a Firebase függvény hívás végeztével hívott volna callback függvényként.

```
doAnswer{presenter.setOwnerInfo(testUser)}.`when`(repo).getOwnerInfo("test
Email")
```

5.3.2 Függőségek

Az előző fejezetben bemutatott megoldás azonban nem bizonyult tökéletesnek, mivel a Presenter Repository változójában továbbra is függőségek voltak adott Presenter objektum tesztelésre való létrehozásakor. Ezt az objektumot nem Mockolhattam, hiszen éppen őt akartam tesztelni. Ezen probléma megoldására minden Presenter konstruktorban kapja meg a Repository változóját, így ezt a függőséget egy Mockolt Repository átadásával eliminálni tudtam. Azért, hogy ez a változtatás ne befolyásolja a program működését, egy alapértéket is beállítok a Repositoryknak, ahol létrehozok egy új Repository objektumot. Így a teszten kívül a program működése nem változik.

```
class CreateEventPresenter(  
    val createEventView: ICreateEventContract.IView,  
    val repo: CreateEventRepository = CreateEventRepository())
```

5.3.3 This kulcsszó

Ahhoz, hogy a Repository a Presenter-t értesíteni tudja, el kell tárolnia a Presenter-t egy változóban. Sajnos mivel a függőség-elkerülés érdekében a Repository-t a Presenter konstruktora hozza létre, ezért nem adhatja át this-ként az aktuális Presentert, hiszen ez a kulcsszó ekkor még nem létezik. Emiatt a Repository-kban létrehoztam egy *setPresenter()* függvényt, amellyel beállítható a Presenter objektuma a Repository-nak. Ezt a függvényt minden Presenter init függvényében hívom meg, mivel ekkor már lehet használni a this kulcsszót, így át lehet adni a jelenlegi Presentert.

```
init{  
    repo.setPresenter(this)  
}
```

6 Összefoglaló

Dolgozatomban bemutattam egy közösségi programszervező alkalmazás elkészítését a követelmény-elemzés fázisától kezdve a teszteléssel bezáróan. A célul kitűzött szoftver teljes egészében elkészült és működőképes. Az alkalmazás fejlesztése során megtanultam, hogyan kell egy nagyobb Androidos projektet kivitelezni. Megtapasztaltam és bemutattam, hogy miért fontos a jól megtervezett architektúrális felépítés, és hogyan lehet megvalósítani az MVP architektúrát egy Android projekten. Ezenkívül a Firebase szolgáltatásokban is elmélyültem és bemutattam. Itt sokat tanultam a felhasznált szolgáltatások működéséről, valamint implementációjukról. A fejlesztési folyamat végén az Android alkalmazások tesztelésének alapjaiba is betekintést nyertem.

A munkám eredményét sikeresnek tekintem, de mint mindenben, ebben is sok tovább-fejlesztési potenciál rejlik. Az alkalmazás MVP architektúrája tovább finomítható a Repository és a Presenter elemek közé egy köztes Interactor elem bevezetésével annak érdekében, hogy az egyes komponensek felelősségei mégjobban elkülönüljenek egymástól. Emellett, a tesztelés során sok fejtörést okozott az egyes függőségek eliminálása, amelynek a nehézségei elsősorban a manuális függőség-átadásból származott. Ezt javítandó a Dependency Injection (DI) [15] módszert lehetne alkalmazni azért, hogy a komponensek függőségeinek használata, kezelése és létrehozása megfelelően elkülönüljenek egymástól. Android platformon az egyik legelterjedtebb DI keretrendszer a Dagger [16] és annak egyszerűbb használatát biztosító változata, a Hilt [17]. Ezeken kívül a jövőben újabb Firebase szolgáltatások felhasználásával újabb funkciókat lehetne az alkalmazáshoz írni, például Cloud Functions szolgáltatással megvalósított értesítéseket.

Összességében nagyon hasznosnak tartom a projekt során gyűjtött tapasztalataimat. Mind a Firebase-be mind az Androidos szoftverfejlesztésbe beleszerettem, mert a gyors implementációnak köszönhetően, az egyes részek fejlesztésekor gyorsan láthatom a munkám gyümölcsét, így ez további motivációt ad a tovább haladásra. A jövőben mindenképp szeretnék még mindkettővel foglalkozni, de az Android platform további fejlesztési módszereit is szívesen elsajátítanám.

7 Hivatkozások

- [1] „Android Developers,” [Online]. Available:
<https://developer.android.com/guide/platform>. [Hozzáférés dátuma: 05 05 2022].
- [2] „Android Developers,” [Online]. Available:
<https://developer.android.com/guide/components/activities/activity-lifecycle>. [Hozzáférés dátuma: 05 05 2022].
- [3] „Android Developers,” [Online]. Available:
<https://developer.android.com/guide/fragments/lifecycle>. [Hozzáférés dátuma: 05 05 2022].
- [4] „Firebase,” [Online]. Available: <https://firebase.google.com/>. [Hozzáférés dátuma: 18 05 2022].
- [5] „Android Developers,” [Online]. Available:
<https://developer.android.com/studio>. [Hozzáférés dátuma: 18 05 2022].
- [6] „Gradle,” [Online]. Available:
https://docs.gradle.org/current/userguide/what_is_gradle.html. [Hozzáférés dátuma: 18 05 2022].
- [7] „Firebase,” [Online]. Available:
https://firebase.google.com/products/firestore?gclid=CjwKCAjwj42UBhAAEiwACIhADiNVezysbgUPvRT9w2jQ37CnBRaJw5dijko1S1g9s6J-wf3lI2COfRoCrqMQAvD_BwE&gclsrc=aw.ds. [Hozzáférés dátuma: 18 05 2022].
- [8] „Azure,” [Online]. Available: <https://azure.microsoft.com/hu-hu/overview/nosql-database/>. [Hozzáférés dátuma: 18 05 2022].
- [9] „MongoDB,” [Online]. Available: <https://www.mongodb.com/document-databases>. [Hozzáférés dátuma: 18 05 2022].

- [10] „Firebase,” [Online]. Available:
https://firebase.google.com/products/storage?gclid=CjwKCAjwj42UBhAAEiwACIhADuPURSpvbtIJ55H6zLEqMU3C50qoaGbVL_IYsy5_c3ppaeDhURgsGBoC1YwQAvD_BwE&gclidsrc=aw.ds. [Hozzáférés dátuma: 18 05 2022].
- [11] „Codecademy,” [Online]. Available: <https://www.codecademy.com/article/mvc>.
[Hozzáférés dátuma: 08 05 2022].
- [12] „Javatpoint,” [Online]. Available: <https://www.javatpoint.com/gwt-mvp>.
[Hozzáférés dátuma: 08 05 2022].
- [13] „Mockito,” [Online]. Available: <https://site.mockito.org/>. [Hozzáférés dátuma:
18 05 2022].
- [14] „JUnit,” [Online]. Available: <https://junit.org/junit4/>. [Hozzáférés dátuma: 18
05 2022].
- [15] „Android Developers,” [Online]. Available:
<https://developer.android.com/training/dependency-injection>. [Hozzáférés
dátuma: 18 05 2022]
- [16] „Dagger,” [Online]. Available: <https://dagger.dev/>. [Hozzáférés dátuma: 18 05
2022].
- [17] „Dagger,” [Online]. Available: <https://dagger.dev/hilt/>. [Hozzáférés dátuma: 18
05 2022].