



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

Optikai karakterfelismerésen alapuló jegyzet digitalizációs mobilalkalmazás kifejlesztése

SZAKDOLGOZAT

Készítette
Váradi Vivien

Konzulens
dr. Ekler Péter

2021. december 7.

Tartalomjegyzék

Kivonat	i
Abstract	ii
1. Bevezetés	1
2. Platform	2
2.1. Felépítés	2
2.1.1. Linux kernel	2
2.1.2. Hardver absztrakciós réteg	2
2.1.3. Android Runtime	3
2.1.4. Natív C/C++ könyvtárak	3
2.1.5. Java API keretrendszer	4
2.1.6. Rendszeralkalmazások	4
2.2. Fejlesztői környezet	4
2.2.1. Funkciók	4
2.2.2. Projektstruktúra	5
3. Architektúra	6
3.1. Felépítés	6
3.1.1. View	7
3.1.2. ViewModel	7
3.1.3. Presenter	7
3.1.4. Interactor	7
3.1.5. DataSource	7
3.2. Funkciók	7
3.2.1. Dependency Injection	7
3.2.2. View State	8
3.2.3. ViewFlipper	8
3.2.4. Események	9
3.2.5. Tesztelés	10
4. Felhasznált könyvtárak	11
4.1. Google Cloud Vision API	11
4.2. Firebase	12
4.2.1. Cloud Firestore	12
4.2.2. Authentication	13
4.2.3. Analytics	13
4.2.4. Crashlytics	14
4.3. Groupie	15
4.4. EasyPermissions	15

4.5. Material Components	16
4.6. Jetpack Navigation Component	17
4.7. JUnit 4	18
4.8. Mockito	18
5. Követelmények	20
6. Alkalmazás működése	21
6.1. Bejelentkezés	21
6.2. Kategorizált lista	21
6.3. Jegyzetlista	22
6.4. Jegyzet létrehozása	23
6.5. Jegyzet szerkesztése	24
6.6. Kategória létrehozása	25
6.7. Kategória szerkesztése	26
7. Implementáció	28
7.1. Projektfelépítés	28
7.2. Szerveroldali komponensek integrációja	28
7.3. Képfeldolgozás integrációja	31
7.4. Adatmodellek elkészítése	32
7.5. Egyedi FloatingActionButton	33
7.6. Egyedi legördülő menü	34
7.7. Képernyők felépítése	35
7.8. Kategorizált listanézet	36
7.9. Egyszerű listanézet	38
7.10. Navigáció	40
7.11. Tesztelés	41
8. Összefoglalás	44
8.1. Tapasztalatok	44
8.2. Értékelés	44
8.3. Továbbfejlesztési lehetőségek	45
Irodalomjegyzék	46

HALLGATÓI NYILATKOZAT

Alulírott *Váradi Vivien*, 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(k), 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 autentikált 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.

Budapest, 2021. december 7.

Váradi Vivien
hallgató

Kivonat

Az elmúlt évek rohamos technológiai fejlődésének köszönhetően az okostelefonok hatalmas teret hódítottak maguknak, társadalmunk jelentős hányada mára már rendelkezik legalább egy ilyen eszközzel. Ennek következményeképpen szinte mindent készülékeinken intézünk: kapcsolattartást, hivatalos ügyeket, vagy éppen a tanulást. Ezek megkönnyítésére folyamatosan jelennek meg a különböző natív alkalmazások, melyeket feltelepítve csupán pár kattintásra egyszerűsödnek az elvégezni kívánt feladatok.

Ám rengeteg alkalmazási területen még nincs igazán jól használható applikáció, ilyen például az egyetemi jegyzetelés. Akinek nem makulátlan a kézírása, és a gyors tempójú előadások, gyakorlatok során sietősen kell papírra vetnie az elhangzottakat, az gyakran szembesülhet vele, hogy a következő héten már nem tudja elolvasni az előző heti jegyzetét. Ha pedig valaki tömegközlekedésen szeretné az időt hasznosan eltölteni, vagy utazás közben tanulni, akkor mindig mindenhova vinnie kell magával a füzeteket, illetve könyveket. Ezen a kényelmetlen helyzeten szerettem volna egy kicsit segíteni egy alkalmazással.

A célom az volt, hogy egy hétköznapiakban kényelmesen használható programot hozzak létre, mely az optikai karakterfelismerés - optical character recognition, röviden OCR - segítségével a lefotózott dokumentumokat digitalizálja. A projekt megvalósítása során létrehoztam egy Android kliensalkalmazást, mely a Google Cloud Vision API használatával digitális szöveggé alakítja a fényképen megjelenő nyomtatott/írott szöveget, és azt egy tetszőleges struktúrában Firebase segítségével eltárolja és megjeleníti.

Abstract

Due to the rapid technical advancement of the past couple of years, smartphones have conquered the world, and as of today a significant portion of our society owns at least one smart device. As a consequence we do almost everything on our phones: keeping in touch, official matters or even studying. In order to make these easier native apps are constantantly being released that simplify the tasks at hand.

However, there are a lot of use cases where there are no such applications yet, and taking notes at the university is one of them. The person whose handwriting is not perfectly clean and has to quickly jot the material down during lectures and practices often has to face the fact that they cannot read their own handwriting from the week before. And if someone wants to use their time wisely on public transportation, or study a little while travelling, then they always have to take their notebooks and books with them everywhere. These are the main, uncomfortable scenarios that I wanted to help with this application.

My goal was to produce a program, that's comfortable to use in everyday life, which digitizes pictures of documents with the help of optical character recognition, or OCR for short. During the realization I created an Android client application that uses Google Cloud Vision API to produce digital text from a printed/handwritten text in a picture, and stores it in a user-defined structure with the help of Firebase.

1. fejezet

Bevezetés

A bevezető tartalmazza a diplomaterv-kiírás elemzését, történelmi előzményeit, a feladat indokoltságát (a motiváció leírását), az eddigi megoldásokat, és ennek tükrében a hallgató megoldásának összefoglalását.

A bevezető szokás szerint a diplomaterv felépítésével záródik, azaz annak rövid leírásával, hogy melyik fejezet mivel foglalkozik.

2. fejezet

Platform

Az Android egy nyílt forráskódú, Linux alapú operációs rendszer, mely először okostelefonokra készült, de mára már eszközök igen széles skáláján megtalálható a karórától kezdve a háztartási eszközökön át az autókig. A legtöbbet használt mobil operációs rendszer, piaci részesedése közel 73%.^[1]

Népszerűsége pedig nem véletlen; a legolcsóbbtól a legdrágábbig mindegyik szegmensben található Android készülék, így mindenki kiválaszthatja a számára legmegfelelőbbet. Hatalmas szabadságot ad a felhasználó kezébe, lecserélhetünk bármilyen alapértelmezett alkalmazást, sőt, az operációs rendszert is le lehet váltani más verzióra, vagy akár egy teljesen más custom ROM¹-ra.

Ez a változatosság és szabadság fejlesztői szempontból kettős. Egyrésztől gyakorlatilag lehetséges bármilyen alkalmazást írni, és a nyílt forráskód miatt bármikor bele lehet nézni a platform kódjába, hogy megértsük a viselkedését, másrésztől viszont lehetetlenné teszi, hogy ellenőrizni tudjuk az alkalmazásunk működését minden létező készüléken.

Az alábbiakban a platform felépítését, sajátosságait fogom ismertetni, majd pedig a fejlesztéshez szükséges környezetet és eszközöket mutatom be.

2.1. Felépítés

A leírás kiegészítéseként a platform felépítését a 2.1. ábra szemlélteti.

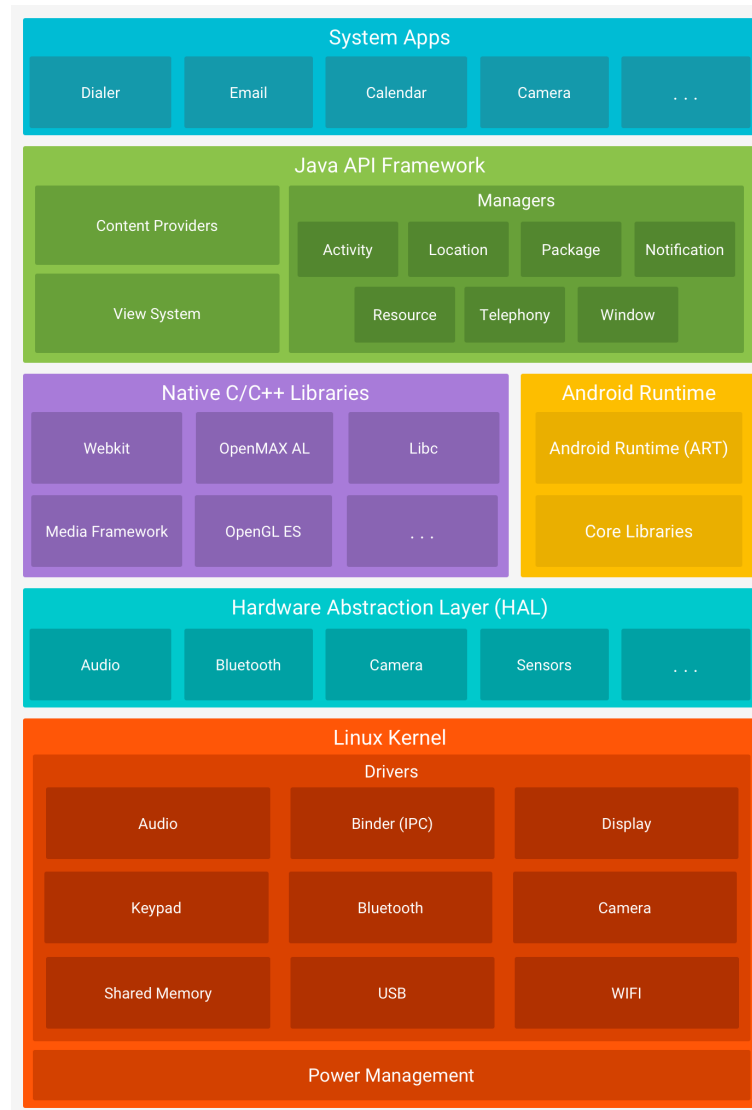
2.1.1. Linux kernel

Az operációs rendszer alapját a Linux kernel képezi, ezáltal ki tudja használni az évek során elért stabilitást és biztonságot, illetve lehetővé teszi a készülékgyártóknak, hogy egy már jól ismert technológiával dolgozzanak az illesztőprogramok fejlesztése során. Feladatai közé tartozik a memóriakezelés, a folyamatok ütemezése és a teljesítménykezelés. Ez utóbbi kiemelt fontosságú, hiszen a mobil eszközök akkumulátora véges, így a lehető legalacsonyabb fogyasztásra kell törekedni. ^[2]

2.1.2. Hardver absztrakciós réteg

Közvetlenül a kernel felett található a hardver absztrakciós réteg - hardware abstraction layer, röviden HAL -, mely a készülék hardveres adottságait (pl.: kamera, szenzorok) ajánlja ki a felette található Java API keretrendszernek. Több modulból áll, melyek egy-egy specifikus hardverkomponens interfészt valósítanak meg, és a rendszer dinamikusan tölti be ezeket, amikor az API hozzá akar férni valamelyikhez. ^[3]

¹Egy custom ROM az Android egy nem hivatalos, módosított verziója, mely általában több testreszabási lehetőséget és szabad kezet ad a felhasználójának.



2.1. ábra. Az Android operációs rendszer architektúrája.

2.1.3. Android Runtime

A következő komponens az Android Runtime, röviden ART. Ez egy virtuális gép, melyből mindegyik alkalmazás rendelkezik egy saját példánnyal, így egymástól és az operációs rendszertől izoláltan futhatnak. Mind az ART, mind az elődje, a Dalvik Virtual Machine specifikusan Androidra készültek. Az elődjéhez képest az Android Runtime rendelkezik plusz funkciókkal, ilyen többek között a telepítésidejű fordítás, az optimalizált szemétyűjtés vagy a jobb hibakeresési támogatás. [3]

2.1.4. Natív C/C++ könyvtárak

Ahogy azt a 2.1. ábra mutatja, natív C/C++ könyvtárak is helyet kaptak a rendszerben. Ezek a kernelen futnak, és sok rendszerkomponensnek szüksége van rájuk. A platform biztosít Java API-t néhányhoz, így például hozzáférhetünk az OpenGL grafikai könyvtárhoz natív kódból. [3]

2.1.5. Java API keretrendszer

A fejlesztők számára a legfontosabb elem a Java API keretrendszer. Ez tartalmazza az operációs rendszer minden funkcióját, a moduláris, könnyedén újrahasznosítható építőelemeket, melyek felhasználásával a fejlesztők alkalmazásaikat elkészíthetik. Többek között magában foglalja az alábbiakat:

- *View System*: Felhasználói felületek létrehozásában van segítségünkre, kiterjedt és még tovább bővíthető elemkészlettel (pl.: listák, szövegmezők, gombok).
- *Resource Manager*: Hozzáférést biztosít a fejlesztőnek minden erőforrásfájlhoz. Ezek tipikusan XML-fájlok, melyek leírják a projektben használatos lokalizált szövegeket, vektorgrafikus ábrákat és a fentebb is említett felhasználói felületeket.
- *Activity Manager*: Az Android alkalmazások alapkövei az Activityk, melyeknek meghatározott életciklus-szakaszai vannak a létrehozástól a megszűnésig. Ezt az életciklust kezeli az Activity Manager, illetve egy közös navigációs backstacket biztosít, mely tárolja, hogy az alkalmazásban milyen képernyőket látogattunk meg a használat folyamán. [3]

2.1.6. Rendszeralkalmazások

A rendszer beépített alkalmazásokkal érkezik a leggyakoribb feladatok elvégzésére. Ilyen például az SMS-küldés, internet böngészés, telefonálás, és a naptár. Ezek, ahogy a fenti bevezetőben is említettem, néhány kivétellel teljesen lecserélhetők felhasználó által letöltött alkalmazásokra.

Mindez nem csak a felhasználó számára lényeges - a fejlesztő is fel tudja használni. Például, ha a fejlesztő szeretne telefonhívást indítani a saját alkalmazásából, akkor nem kell ezt a funkcionalitást implementálnia, elég csak meghívnia a készüléken elérhető alapértelmezett alkalmazást, ami majd elvégzi ezt. [3]

2.2. Fejlesztői környezet

Az Android fejlesztés hivatalos eszköze az Android Studio, mely egy IntelliJ IDEA alapú integrált fejlesztési környezet, röviden IDE.[4] Rengeteg funkcióval rendelkezik, így most csak a fontosabbakat fogom érinteni.

2.2.1. Funkciók

- *Build eszközök*: Rugalmas, Gradle-alapú build rendszert használ, így a projektbeállításokat és a külső könyvtárakat elég csak az erre kijelölt *.gradle* kiterjesztésű fájlokba felvenni, a többit elvégzi helyettünk.
- *Egységes környezet*: Nem kell külön program más eszközökre való fejlesztéshez, akár telefonra, táblagépre vagy okosórára szeretnénk alkalmazást írni, mindet megtehetjük a Studioból.
- *Emulator*: Fejlesztésnél rendkívül praktikus, hogyha nem kell minden alkalommal fizikai készüléket keresni, ha futtatni szeretnénk a programunkat. Erre szolgál a beépített emulátor, amivel különböző virtuális készülékeken tesztelhetünk alkalmazásokat. Ez egy teljes operációs rendszert tár elénk, így nem csak az alkalmazásunkat tudjuk rajta megnézni, hanem a rendszeralkalmazások is mind elérhetők. Hívásindítást és -fogadást, SMS-eket, konfigurációváltozást vagy akár helyadat-változást is

tudunk vele emulálni. Egyszerre több emulátorunk is lehet, ezeket az AVD Manager-ben tudjuk kezelni. Új létrehozásánál kiválaszthatjuk, hogy milyen készülékmodellt szeretnénk, melyik Android verzióval, és egyéb hardverbeállításokat is megadhatunk, például, hogy mennyi memóriát szeretnénk a készüléknek.

- *Verziókezelés:* Az IDE több verziókezelő rendszert is beépítetten támogat, így egy kényelmes felületen intézhetjük a változtatások követését.
- *Kódelemzési funkciók:* Ezek azok, amik igazán kényelmessé teszik az Android Studio használatát. Folyamatos segítséget nyújt a kódírásban, a kódkiegészítés jelentősen meggyorsítja a haladást. Jelzi, ha bármi elavult, így nem utólag kell megváltoztatni a kódot. Az egyik leghasznosabb funkció pedig a nem használt kódrészletek jelzése. Mivel az IDE kiszűrki azokat a változókat és függvényeket, amik egyszer sincsenek meghívva, így könnyű karbantartani, hogy ne maradjon a kódbázisban felesleges sor.

2.2.2. Projektstruktúra

Minden projekt egy vagy több modulból áll, és mindent tartalmaz a forráskódtól kezdve a tesztkódon át a build konfigurációkig. A modulok segítségével különálló funkcionális egységekre bontható az alkalmazás, melyeket egymástól függetlenül lehet buildelni, tesztelni és debugolni. [5] Célszerű modulokat használni, ha például különböző eszköztípusokra szeretnénk fejleszteni, hiszen így az egymástól független részeket elkülöníthetjük, de a közös kódot meg tudjuk osztani köztük.

A fejlesztőkörnyezetet megnyitva a projektfájlok megjelenítése alapértelmezetten eltér a fájlrendszerbeli hierarchiától. Ezt Android nézetnek hívja az IDE, és úgy lett kifejlesztve, hogy a lehető legkényelmesebb legyen a fejlesztők számára. Itt alapvetően három mappára szedve találhatók meg a forrásfájlok.

Az első a *manifests*, melyben az *AndroidManifest.xml* fájl található. Ez fontos adatokat tartalmaz az alkalmazásról, melyekre szüksége van a build eszközöknek, az operációs rendszernek, és a Google Play-nek is. Fel kell sorolni benne többek között az összes komponensét, a futás során szükséges engedélyeket (pl.: fájlrendszer-hozzáférés, kamera), illetve a működéshez elengedhetetlen szoftveres és hardveres funkciókat. Utóbbit ellenőrzi is a Play Store, és csak olyan készülékekre engedi feltelepíteni az alkalmazást, melyek ennek megfelelnek. [6]

A következő mappa általában *java* névre hallgat, és ebben található a projekt összes forráskódja. A nevével ellentétben ma már legtöbbször Kotlin kódot tartalmaz, mivel 2019 óta ez az Android fejlesztés hivatalos, preferált nyelve, de az elnevezés megmaradt azokból az időkből, amikor még mindenki Java-t használt.

Az utolsó mappa pedig a *res*, melyben az erőforrásfájlok találhatók. Ezek általában XML fájlok, és az alkalmazásban felhasznált layoutokat, animációkat, stringeket írják le, de ide kerülnek például a képi erőforrások is.

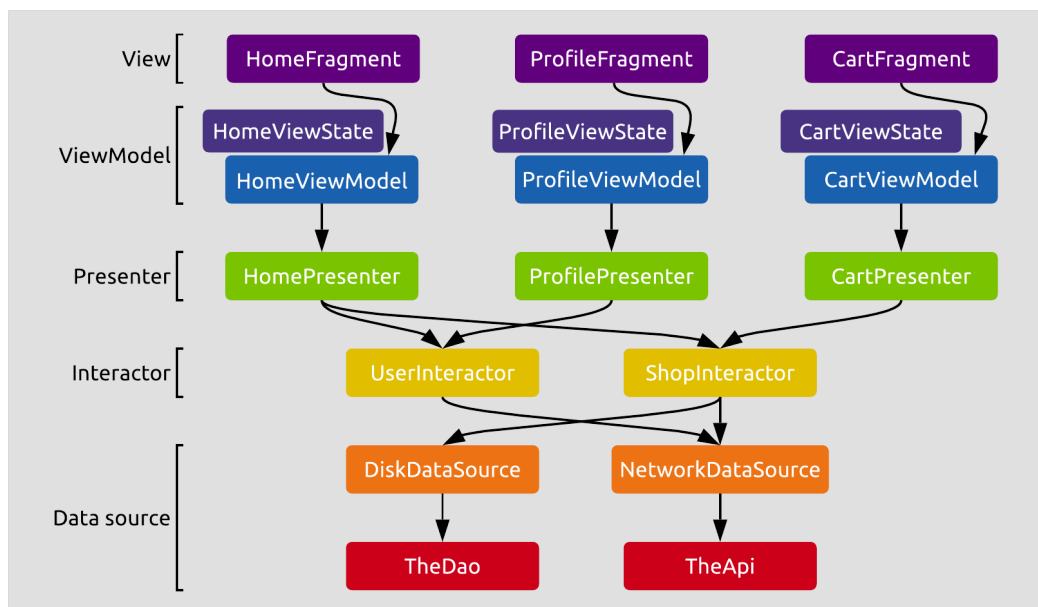
3. fejezet

Architektúra

A fejlesztés során elsődleges célom volt, hogy egy olyan architektúrára alapozzam az alkalmazást, mely megfelelően szétválasztja a felelősségeket, és ezáltal egy könnyen karbantartható, átlátható kódot alkossak. Először egyből az MVVM (Model - View - ViewModel) architektúrára gondoltam, mivel azt már korábban is használtam, és jó tapasztalataim voltak vele. A RainbowCake[7] egy MVVM-re építő architektúra, melynek fő fókusza a felelősségek szétválasztása és a képernyő konzisztens állapotban tartása. Tekintve, hogy ez pontosan megegyezik a céljaimmal, így a kipróbálása mellett döntöttem.

3.1. Felépítés

Az architektúra rétegeit a 3.1. ábra szemlélteti.



3.1. ábra. A RainbowCake architektúra rétegei.

3.1.1. View

A view-k alkotják az alkalmazás képernyőit, melyekkel a felhasználó találkozik használat közben. Ezek lehetnek Activityk vagy Fragmentek, és fő feladatuk továbbítani a felhasználói inputot a ViewModel felé, melytől cserébe új adatot, állapotot vagy eseményt kapnak.

3.1.2. ViewModel

A felhasználói felülettel kapcsolatos logikát végzik, illetve tárolják és a Presentertől kapott adatok alapján frissítik a képernyőhöz tartozó ViewState-et.

3.1.3. Presenter

Háttérszálra teszik a hívásokat, és továbbítják azokat az Interactorok felé, majd az eredményt képernyő specifikus prezentációs modellekké transzformálják, és azt adják oda a ViewModelnek. A projektből ezt a réteget teljesen kihagytam, mivel a View ugyanazt a modellt jeleníti meg, mint amit az adatbázisból visszakap, így nem volt szükség a modelltranszformációra.

3.1.4. Interactor

Az Interactorok, ahogy az ábrán is látható, nem egy-egy képernyőhöz köthetők, hanem funkcionalitásokhoz. Ők végzik a fő üzleti logikát, manipulálják az adatokat és számításokat végeznek. Az alkalmazásomban csak egyetlen darab van, ugyanis az adatok megjelenítése nem igényel sok üzleti logikát, viszont Presenterek hiányában ő kapta meg a felelősséget, hogy a hívásokat háttérszálra tegye.

3.1.5. DataSource

Egységes interfészt biztosítanak az adathozzáféréshez, feladatuk a különböző hívások (pl.: lokális adatbázis, Firebase) implementációjának elfedése, és az adatok konzisztens állapotban tartása.

3.2. Funkciók

A következőkben röviden kifejttem az architektúra főbb funkcióit, melyek segítségemre voltak az alkalmazás fejlesztése során.

3.2.1. Dependency Injection

A függőséginjektálás - dependency injection, röviden DI - egy gyakran használt technika, mely jelentősen könnyebbé teszi a kód újrafelhasználását, refaktorálását és tesztelését.[8]

Függőségnek nevezzük azt, amikor egy osztálynak szüksége van egy másik osztály működésére vagy képességeire. Ilyenkor, ha ezt saját maga példányosítja, akkor szoros csatolás jön létre a két osztály között, ennek következményeképpen pedig nem lehet a tartalmazott objektum helyett később egy másik implementációt vagy leszármazottat használni. Emellett a tesztelés is megnehezedik, hiszen az osztály egy valódi példányt használ a másiktól, azt nem tudjuk egy tesztobjektummal helyettesíteni.

Függőséginjektálásról akkor beszélünk, hogyha a fent leírt módszer helyett az osztály a függőségeit paraméterként kapja, például a konstruktorban. Ezáltal az említett problémák megszűnnek, az osztályunk újrafelhasználható lesz, és könnyedén tesztelni is tudjuk, ha valós függőség helyett egy mock objektumot adunk neki.

A DI megvalósítására két opció létezik: manuális, illetve automatizált. Az előbbi kisebb alkalmazásoknál megoldható lehet, ha csak pár osztály létezik kevés függőséggel, de nagyon hamar kezelhetlenné válik az alkalmazás növekedésével. Erre nyújtanak megoldást a különböző könyvtárak, melyek megfelelő beállítások mellett automatikusan elvégzik a függőségek létrehozását és biztosítását. Egy ilyen könyvtár a Dagger 2, mely az egyik legnépszerűbb ezen a téren. A RainbowCake beépítetten támogatja a használatát, csak fel kell venni a projekt függőségeként. Ezt követően a megfelelő működéshez még szükség van pár osztály megvalósítására.

Az első az Application, mely a projektben a ScanMyNotesApplication nevet kapta. Neki annyi a feladata, hogy leszármazzon a RainbowCakeApplication-ból, és felülírja az *injector* property-t a konkrét Dagger komponenssel, jelen esetben a DaggerAppComponent-tel. Ezzel kapcsolatosan még annyi feladat van, hogy a projekt manifest fájljában az *android:name* attribútumot átírjuk az itt megadott Application nevére.

A következő a ViewModuleModule, melyre azért van szükség, hogy az injektálási mechanizmus ismerje a ViewModeleket. Ebben minden, az alkalmazásban szereplő ViewModelt fel kell sorolni és a megfelelő annotációkkal ellátni. Tulajdonképpen ugyanez a szerepe a NetworkModule-nak is, csak ő az API-kat biztosítja, és annyi a különbség, hogy ezek a ViewModellekkel ellentétben Singletonok¹.

A fentiekén kívül még két beállítást kell elvégeznünk. Az egyik az ApplicationModule osztály létrehozása, mely egy darab Singleton kontextust biztosít az alkalmazás számára, a másik pedig az AppComponent interfész, melyben fel kell sorolni az összes modult, amire szükség lesz az applikációban.

Ez azért nem egy kétperces setup folyamat, de cserébe ezután csak annyit kell tennünk, hogy amelyik konstruktor paramétereit injektálni szeretnénk, azt megjelöljük egy *@Inject* annotációval, és a Dagger biztosítani fogja az osztályunk számára azokat a függőségeket.

3.2.2. View State

A RainbowCake állapotkezelése jelentősen megkönnyíti a UI különböző állapotainak nyomon követését, és az annak megfelelő nézet megjelenítését. Ha egy képernyőnek több egymástól független, elkülöníthető állapota van, akkor ezt nem érdemes egyetlen *data class* tagváltozóiban tárolni, mert az inkonzisztens állapothoz vezethet.[9]

Erre kínál egy kényelmes megoldást a Kotlinban elérhető *sealed class*, mely egy olyan osztály, aminek fordítási időben ismerjük az összes leszármazottját. Ha a fragment állapotait egy ilyen *sealed class*-ból származtatjuk le, akkor lévén osztályok, tartalmazhatnak saját tagváltozókat (például a megjelenítendő adatokat vagy a hibaüzenetet), és a Kotlin *when* feltételes szerkezetének segítségével ki lehet kényszeríteni, hogy minden állapot le legyen kezelve. Így már nem fordulhat elő inkonzisztencia, hiszen az egymást kölcsönösen kizáró állapotokból egyszerre csak egy lehet aktív a futás során. A képernyő render logikája pedig leegyszerűsödik: az architektúra által biztosított *render* függvényben az állapottól függően változtathatjuk a megjelenítést. Az pedig garantálva van, hogy a *render* mindig lefut, amikor megváltozik a képernyő állapota.

3.2.3. ViewFlipper

A ViewFlipper egy layout komponens, mely az állapotok olvasható elkülönítését teszi lehetővé. [10] Ugyanis, ha a képernyő minden elemét külön manipuláljuk a fent említett *render* függvényben, akkor ez egy idő után olvashatatlan kódot fog eredményezni, rengeteg hiba-lehetőséggel. Ezt orvosolja a ViewFlipper, melyben egyszerre egy tartalmazott layoutot tudunk megjeleníteni, így nem kell manuálisan állítani a különálló elemek láthatóságát.

¹ Egyetlen példány létezhet belőlük.

Használata rendkívül egyszerű, a layoutot leíró XML fájlban egy ViewFlipper komponenst kell létrehozni (3.2. ábra), az épp látható gyerek elemét pedig a *displayedChild* tagváltozón keresztül tudjuk elérni és módosítani (3.3. ábra).

```
<?xml version="1.0" encoding="utf-8"?>
<ViewFlipper xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/noteListViewFlipper"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <include layout="@layout/layout_loading"
        android:id="@+id/loadingView"/>

    <include layout="@layout/layout_note_list"
        android:id="@+id/noteListView"/>

</ViewFlipper>
```

3.2. ábra. A ViewFlipper komponens használata XML-ben.

```
override fun render(viewState: NoteListViewState) {
    when(viewState){
        is Initial -> Log.d("DEBUG", msg: "Screen state is initial")
        is Loading -> binding.noteListViewFlipper.displayedChild = LOADING
        is ListLoaded -> {
            setSearchVisibility()
            adapter.showList(viewState.noteList)
            binding.noteListViewFlipper.displayedChild = VIEWING
        }
        is Error -> {
            Log.d("ERROR", viewState.message)
        }
    }
}
```

3.3. ábra. A ViewFlipper megjelenített gyerekének változtatása.

3.2.4. Események

A RainbowCake biztosít keretet arra is, hogyha egyszeri eseményeket szeretnénk a képernyőn megjeleníteni. Ilyen lehet például, ha hibába ütközik az alkalmazás, vagy a visszakapott adatok alapján navigálni szeretnénk egy másik képernyőre. Ezt nem tárolhatjuk az állapotban - hiszen nem olyan dolog, ami minden renderelésnél történik -, ezért megalkották a *OneShotEvent* típust, mely az ilyen eseményeket reprezentálja. [11]

Használata rendkívül egyszerű, minden lehetséges event típust definiálunk a ViewModelben, majd a megfelelő helyeken a *postEvent* függvénnyel tudjuk elküldeni a fragmentnek, mely az *onEvent* függvény felüldefiniálásával tudja ezeket kezelni.

3.2.5. Tesztelés

A könyvtár tartalmaz beépített támogatást a ViewModelek és a Presenterek tesztelésére, ezek közül a projekt szempontjából a ViewModel tesztelése releváns. Ennek a fő nehézségét az adja, hogy a beérkezett adatokra állapotváltozás, illetve események formájában reagál a ViewModel, melyeket a validáláshoz meg kell figyelnünk valamilyen módon. [12]

Ehhez nyújt segítséget az architektúra *ViewModelTest* osztálya. A tesztosztályunkat ebből leszármaztatva elérhetővé válik a *observeStateAndEvents* függvény, mely biztosít egy *stateObserver* és egy *eventsObserver* objektumot. Az előbbi a teszt futása során előforduló *ViewState*-eket figyeli, az utóbbi pedig az eseményeket. Mindkét *observer* tartalmaz *assert* függvényeket az eredmények ellenőrzésére, ilyen például az *assertObserved*, amivel a futás során előforduló összes állapotot vagy eseményt hasonlíthatjuk az elvárthoz, illetve az *assertObservedLast*, mely az előzővel ellentétben csak a legutolsót figyeli.

4. fejezet

Felhasznált könyvtárak

4.1. Google Cloud Vision API

A Vision API a Google Cloud szolgáltatásainak képfelismerésre specializált része. Használata csak egy bizonyos, havonta újuló kvótáig ingyenes, mely szerencsére bőven elegendő volt a projekt elkészítése és tesztelése során. Számos előre betanított modellt tartalmaz, melyekkel detektálhatunk és osztályozhatunk tárgyakat, arcokat, szövegeket vagy akár felnőtt tartalmakat is. [13]

A jegyzetek digitalizálásához szövegfelismerésre van szükség, ami optikai karakterfelismerés alkalmazásával valósítható meg. Ez az `API DOCUMENT_TEXT_DETECTION` funkciójával lehetséges, mely optimalizálva van mind nagy sűrűségű dokumentumok, mind kézírás detektálására. Ennek során a kiválasztott képet egy base64 kódolt string formájában kell az API-nak elküldeni, a kívánt felismerés elvégzése után pedig az eredményt egy *TextAnnotation* típusú objektumban kapjuk vissza. Ez egy strukturált reprezentációja a kinyert szövegnek, amit oldalakra, paragrafusokra vagy akár szavakra is bonthatunk. A projekt esetében elegendő volt az objektumon a *text* property használata, mely az eredményt egyetlen stringben adja vissza.

A szolgáltatás számtalan nyelvet támogat, többek között a magyart is, és végül emiatt esett erre a választásom. Nincs is igazán sok elérhető API, mely képes lenne kézírásfelismerésre, de ez az egyetlen, ami ezt magyarul is támogatja. Esetleg a Microsoft Azure Computer Vision szállhatna vele versenybe, de ilyen téren az is elmarad, mert jelenleg csak nyomtatott dokumentumok detektálására képes magyarul. Emellett a Cloud Vision API mellett szól az is, hogy mivel mind ez, mind maga az Android Google termék, így sokkal egyszerűbb a kettő közti integráció, arról nem is beszélve, hogy a dokumentáció minősége is a Google-nél a legmagasabb. Természetesen kerestem más alternatívákat is, de a legtöbb kézírás-felismerő szolgáltatás a digitális jegyzetelésre koncentrál - amikor a felhasználó egy érintőképernyőre ír egy speciálisan erre készített "tollal" -, de nekem a use case-t tekintve ezek nem feleltek meg.

Az API kézírás-felismerő képességének megvannak a korlátai, amik kevésbé használhatóvá teszik az alkalmazást, mint amennyire én terveztem. Ma már az Egyesült Államokban elég ritkán írnak kézzel az emberek, és ez meglátszik a modell teljesítményén. Az én kézírásom szinte megfejthetetlen a Vision API számára, nagyon gondosan és odafigyelve kell formálnom a betűket ahhoz, hogy felismerje. A szép kézírást viszont egészen megbízhatóan teljesíti, illetve az írott nagybetűkkel is elboldogul. Így sajnos nem sikerült egy olyan szinten használható alkalmazást készíteni belőle, mint ahogy én elképzeltem, de szebben írt, rövidebb szövegek digitalizálására megfelel.

4.2. Firebase

A Firebase szintén a Google terméke fejlesztők számára, mely megkönnyíti és felgyorsítja a fejlesztési folyamatokat azáltal, hogy egy teljes backend-infrastruktúrát biztosít. Így a fejlesztőnek elég csak ezeket a szolgáltatásokat integrálnia az alkalmazásába ahelyett, hogy azt is külön írnia kellene. Lehetőséget kínál felhasználó-kezelésre, adattárolásra, teljesítményfigyelésre, biztosít analitikát, crash-elemzést és még sok más.

Alább található az azon Firebase-szolgáltatások, melyeket felhasználtam az applikáció fejlesztése során; most ezekre fogok kicsit bővebben kitérni.

4.2.1. Cloud Firestore

Egy alkalmazásban a felhasználó által bevitt adatokat valamilyen módon el kell tárolnunk, hogy aztán később is hozzá lehessen férni. Erre két lehetőségünk van: lokális adatbázis használata a készüléken (pl.: Room) vagy a felhőalapú adattárolás. Én az utóbbit választottam, mert felhasználói szempontból sokkal kényelmesebb egyszer egy fiókot csinálni és ahhoz kötni az adatokat, mint mondjuk egy esetleges készülékcserénél kutatni, hogy hogyan lehet átemelni az adatokat az újra. A Firebase két szolgáltatást is biztosít erre a célra, ezek név szerint a Realtime Database és a Cloud Firestore.

A Realtime Database régebb óta létezik, és az adatok JSON formátumú tárolására ad lehetőséget. Ez egyszerű adatszerkezet esetén ideális, de a komplex, hierarchikus adatok szervezésére nem a legjobb választás. Ezen okból döntöttem inkább a Cloud Firestore használata mellett.

A Cloud Firestore a Firebase legújabb adatbázis-szolgáltatása, mely egy új, intuitívabb adatmodellre épít gyorsabb lekérdezésekkel és jobb skálázódással. [14] Adatmodelljét tekintve egy NoSQL adatbázisról van szó, melyben táblák és sorok helyett dokumentumokba és kollekciókba rendezve tároljuk az adatot. A dokumentumok kulcs-érték párokból állnak, de tartalmazhatnak kollekciókat is, melyekben további dokumentumok találhatóak. [15] Az adatbázisnak nincs sémája, így szabadon alkothatjuk meg az adataink struktúráját - akár ugyanazon a kollekción belül található két dokumentum tartalma is eltérhet egymástól.

Emellett a Firestore biztosít még egy rugalmas szabályrendszert, melynek segítségével kontrollálhatjuk a hozzáférést az adatbázisunk különböző részeihez. Itt úgynevezett *match* kifejezésekkel illesztjük rá a szabályokat a megadott elérési útvonalakra, és ha a szabályok nem teljesülnek akkor a lekérdezés meghiúsul. Különböző szabályokat adhatunk meg olvasásra és írásra, de akár lebontva a *get*, *list*, *create*, *update*, *delete* műveletekre is.

Az applikáció biztonsági szabályai a következőképpen néznek ki (4.1. ábra).

```
1 rules_version = '2';
2 service cloud.firestore {
3   match /databases/{database}/documents {
4     match /{document=**} {
5       allow read, write: if
6         request.time < timestamp.date(2022, 1, 31);
7     }
8   }
9 }
10 }
```

4.1. ábra. A projekt biztonsági szabályai.

A fenti szabály minden dokumentumra illeszkedik, és megenged minden írást és minden olvasást abban az esetben, hogyha a lekérdezés 2022. január 31. előtt érkezik be. Ez egy

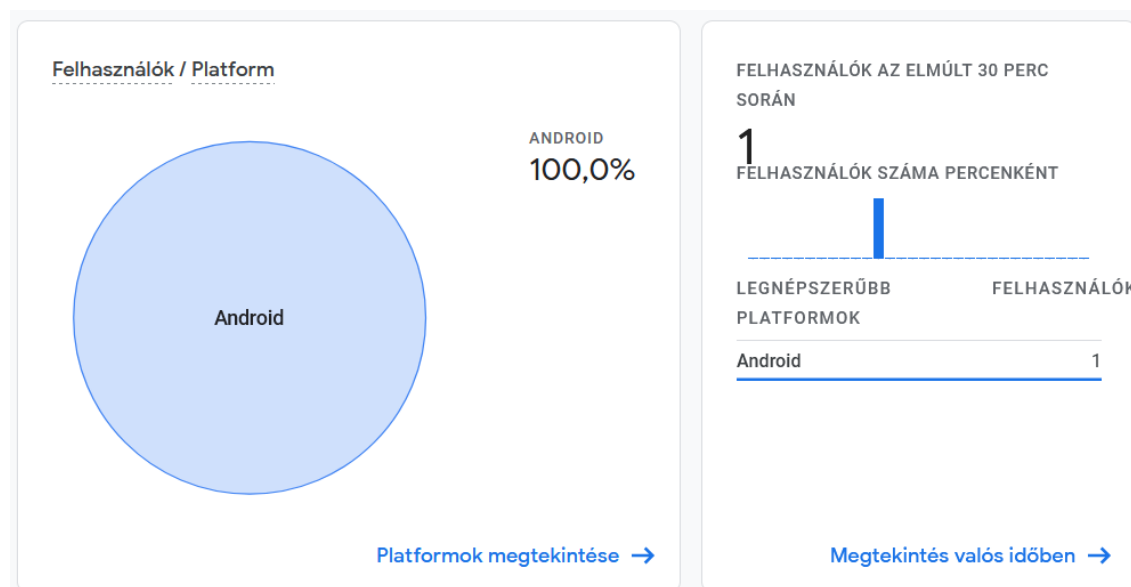
tesztszabály, melyet a Firestore generál a projekt létrehozásakor a fejlesztés megkönnyítése céljából. Természetesen ezt éles alkalmazásban mindenképpen le kell cserélni, hiszen így bárki hozzáférhet az adatbázishoz, aki tudja a projektazonosítót, ám a tesztelési fázisban még bőven elegendő.

4.2.2. Authentication

Ahhoz, hogy a felhasználók csak és kizárólag a saját adataikhoz férjenek hozzá, valamilyen módon tárolni és azonosítani kell őket. Erre kínál megoldást a Firebase Authentication, mely backend-szolgáltatásokat, könnyen használható SDK-kat és UI könyvtárakat biztosít a felhasználók autentikációjára. Lehetővé tesz többek között jelszavas, telefonszámos, illetve harmadik fél általi (pl.: Google, Facebook, Twitter) bejelentkezést is. Automatikusan integrálódik más Firebase szolgáltatásokkal, de saját fejlesztésű háttérrendszerekkel is könnyedén használható. [16]

4.2.3. Analytics

A Firebase Analytics egy ingyenes analitikai megoldás, mely szintén integrálódik más Firebase szolgáltatásokkal. Automatikusan elkap előre definiált eseményeket, de lehetővé teszi a fejlesztő számára saját események létrehozását. Az aktivitást, és a megfigyelt események alapján alkotott statisztikákat pedig bejelentkezés után el lehet érni a Firebase console¹-ban az alkalmazás adatai alatt. [17] Számos hasznos metrikát és diagramot készít az applikáció stabilitásáról, használatáról vagy éppen bevételéről. A fejlesztő figyelemmel kísérheti, hogy hányan használják az alkalmazást (4.2. ábra), mely országokból (4.3. ábra), illetve információt kaphat a tevékenységek időtartamáról, platformokról és elkapott eseményekről.



4.2. ábra. Platform megoszlása a felhasználók között, közelmúlt aktivitása.

Ezen statisztikák nyilvánvalóan nem annyira látványosak ilyen kis mértékű felhasználással, de egy nagyobb felhasználóbázisnál már jelentősen segíthetik az alkalmazás fejlő-

¹<https://console.firebase.google.com>

Felhasználók ▾ / Ország



4.3. ábra. Felhasználók megoszlása az országok között.

dését. Mindez nem csak a fejlesztőknek lényeges, hanem az üzleti résztvevőknek is, hiszen az Analytics számos statisztikát készít a bevételekről és az ügyélszerzésről is.

4.2.4. Crashlytics

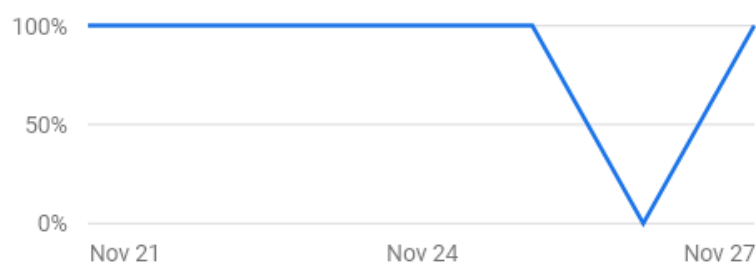
A Firebase Crashlytics jelentéseket készít a fejlesztőknek a felhasználókat érő crashekről. Segít nyomon követni és priorizálni a hibákat, csoportosítja őket, és kiemeli a hozzájuk vezető körülményeket. Valósídejű figyelmeztetést küld az újonnan felbukkanó vagy éppen növekvő problémákról, melyek azonnali figyelmet igényelhetnek. [18]

Szintén a Firebase console-ban érhető el, ahol láthatjuk a crash-free statisztikát (4.4. ábra), a trendeket és az aktuális problémákat. Utóbbira kattintva még bővebb információt kaphatunk az előfordulási gyakoriságról, az érintett felhasználókról és verziókról, és a crash teljes stack trace-ét megtekinthetjük, kiemelve a keletkezés helyét és okát.

Crash-free statistics

Crash-free users ?

50%



4.4. ábra. Crash-free felhasználók aránya napokra lebontva.

4.3. Groupie

Android alkalmazásokban szinte mindig található legalább egy darab listanézet, mely ugyanolyan típusú elemeket sorol fel a képernyőn. Ezt legegyszerűbben a *RecyclerView* komponens segítségével lehet megoldani, mely a megvalósítás mélységeit elfedi előlünk. Két dolgot kell megadnunk a működéséhez: egy sor kinézetét, illetve a megjelenítendő adatokat. A *RecyclerView* - ahogy a neve is sejteti - újrahasznosítja a lista elemeit, ami annyit jelent, hogy a képernyőről eltűnő sorokat nem megszünteti, hanem azokat használja fel az éppen megjelenő sorok létrehozásakor. Mindez jelentősen javítja az alkalmazás teljesítményét és csökkenti az energiafelhasználást. [19]

Azonban a projekt során szükségem volt egy funkcióra, amit a *RecyclerView* alapértelmezetten nem tud, ez pedig a lista elemeinek kinyitható/bezárható csoportokba szervezése. Így találtam rá a *Groupie* névre hallgató könyvtárra, mely kiküszöböli ezt a hiányosságot.

A *Groupie* nagyban megkönnyíti a listák kezelését, ugyanis a tartalmat logikai csoportokként kezeli. [20] Használata rendkívül egyszerű: mindenhol, ahol eddig *RecyclerView.Adapter*-t kellett használni, ezután *GroupieAdapter* szerepel majd. Minden típusú megjeleníteni kívánt elemnek készíteni kell egy saját osztályt, mely a könyvtár által biztosított *Item* osztályból származik le, és felparaméterezi a neki szánt layout-ot a kapott adatokkal. A speciális viselkedések támogatására léteznek különféle interfészek, melyeket pluszban megvalósíthatunk a kívánt működés elérése érdekében. Ilyen például az *ExpandableItem*, mely pontosan azt a funkcionalitást biztosítja, amire szükségem volt: rendelhetőek alá egyéb elemek, melyek felhasználói interakció hatására elrejtethők illetve megjeleníthetők.

A listák építőelemei itt a csoportok, ahol egy item egyelemű csoportnak számít. Így leegyszerűsödik a lista feltöltése is, ugyanis akár felváltva adhatunk az adapterhez több-elemű csoportokat és egyes elemeket. Automatikusan kezeli a lista tartalmának változását, így nem kell manuálisan ezzel foglalkozni és csökken a hibalehetőségek száma. Emellett nagy hangsúlyt fektettek a bővíthetőségre, a könyvtár tudatosan úgy lett felépítve, hogy a lehető legegyszerűbb legyen a meglévő osztályok felhasználásával saját viselkedést definiálni. Ez pedig egy hatalmas előny, ami az enyémnél sokkal komplexebb use case-ekre is alkalmassá teszi.

4.4. EasyPermissions

Android platformon a felhasználó adatait és készülékét egy komplex engedélyrendszer védi. Az operációs rendszer *Marshmallow* névre hallgató verziója (6.0) előtt minden engedélyt telepítési időben kellett jóváhagyni, ám idővel kiderült, hogy ez nem nyújt megfelelő védelmet, hiszen a felhasználók nem olvassák el telepítés előtt, hogy mibe egyeznek bele. Ezért született meg a ma is használatos rendszer, melyben az engedélyek három csoportba sorolhatók:

- *Telepítésidejű engedélyek:* Ezek korlátozott hozzáférést biztosítanak olyan adatokhoz és tevékenységekhez, melyek minimálisan befolyásolják a rendszert, és az alkalmazás telepítésével megadásra kerülnek. Ilyen például a hozzáférés a hálózathoz, internethez vagy épp a rezgőmotorhoz. Két típusba sorolhatók: lehetnek normál engedélyek, amik csak nagyon kis kockázatot jelentenek a felhasználó adataira és más alkalmazások működésére, illetve *signature* engedélyek, melyek akkor kerülnek telepítéskor elfogadásra, ha egy már korábban a készülékre telepített alkalmazás elkérte az adott engedélyt, és ugyanaz a tanúsítvány írja alá ezt és a telepítendő applikációt.

- *Futásidejű engedélyek:* További hozzáférést biztosítanak korlátozott adatokhoz, illetve olyan tevékenységekhez, melyek komolyabban befolyásolják a rendszert vagy más alkalmazásokat. Sok ezek közül potenciálisan érzékeny információt tartalmazhat, mint például a tárolási hely, kamera vagy mikrofon, ezért különösen fontos, hogy a felhasználó tisztában legyen az általa használt alkalmazások engedélyeivel. Így az ilyen, más néven "veszélyes" engedélyeket az applikáció futása közben kell elkérni, közvetlenül az engedélyt igénylő művelet elvégzése előtt. Ilyenkor lehetőséget kap a fejlesztő egy felugró ablakban megmagyarázni, hogy pontosan mit akar végrehajtani, amihez elengedhetetlen az adott hozzáférés, majd a felhasználó ezen információ birtokában döntheti el, hogy megengedi-e. Ezzel sokkal könnyebben kiszűrhetővé váltak a rosszindulatú alkalmazások, hiszen ha induláskor olyan engedélykérés jön, mely az applikáció funkciói alapján teljesen indokolatlan, akkor meg lehet ezt tőle tagadni, és még azelőtt eltávolítani, mielőtt komolyabb kárt tudna okozni.
- *Speciális engedélyek:* Csak a platform és a készülékgyártók definiálhatják ezeket, általában akkor teszik meg, amikor egy különösen nagy befolyású funkció hozzáférését akarják korlátozni, mint például a megjelenítés a többi alkalmazás fölött. [21]

Mivel a projekt természetéből adódóan szüksége van hozzáférésre a kamerához, így nekem is implementálnom kellett futásidejű engedélykérést. Erre az EasyPermissions-ktx könyvtárat választottam, mely a Google Java nyelven írt EasyPermissions könyvtárának Kotlin változata. Egységes, könnyen tanulható és átlátható interfészt biztosít az engedélyek kezelésére, megmentve a fejlesztőt a hosszú és átláthatatlan *if - else* szerkezetek írásától a különböző engedélyek állapotának ellenőrzésére.

Segítségével könnyedén lekezelhetjük a felhasználó döntéseit, mivel minden eshetőségre biztosít callback-függvényeket. Ellenőrizni tudjuk az egyes engedélyek állapotát - és ezt minden ehhez kapcsolódó tevékenység előtt meg is kell tenni, hiszen a felhasználó azt bármikor visszavonhatja -, reagálhatunk az elutasításra, de még arra is, hogyha a *Never ask again* opció került kiválasztásra. Ez utóbbi esetben már nem dobhatjuk fel többször az engedélykérést az alkalmazásban, de ha mindenképp elengedhetetlen a működéshez, akkor elnavigálhatjuk a felhasználót a beállítások panelre, ahol manuálisan megadhatja az engedélyt.

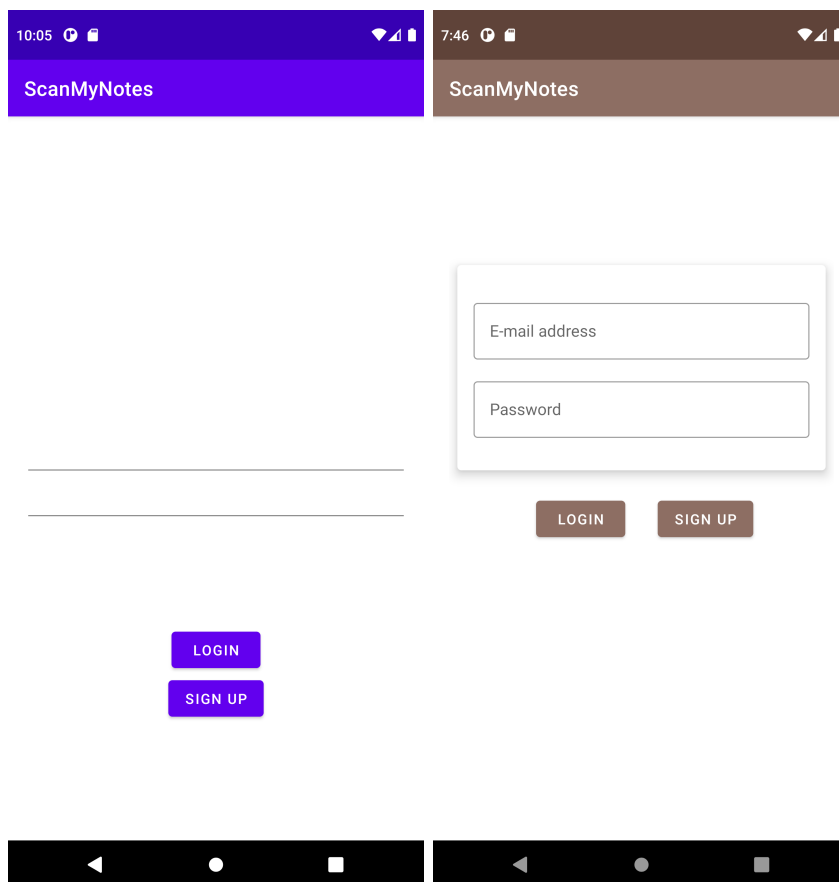
4.5. Material Components

A Material² egy kicsit eltér a fentebb felsoroltaktól, mert ez jóval több, mint egy könyvtár. Egy teljes design rendszerről van szó, mely lehetővé teszi a fejlesztők számára a letisztult, intuitív felhasználói felületek alkotását. Kiterjedt dokumentációban ismerteti az alkotás során követendő alapelveket és példákkal illusztrálja, hogy mit mikor és mikor nem célszerű alkalmazni. Több eszközt is biztosít a fejlesztés könnyítésére, a dokumentációban összegyűjtve találunk kompatibilis ikonokat, betűtípusokat, de még egy színpaletta generálót is.

Ennek csupán egy része a Material Components, mely használatra kész elemeket tartalmaz, amiket könnyedén integrálhatunk a különböző platformok projektjeibe. Androidon szinte minden elérhető UI elemnek létezik Material megfelelője, melyekkel egységesíthető a felület és minőségibb érzetet nyújt a felhasználó számára. Az elemek működését a valós világ ihlette: árnyékokat vetnek és interakcióba lépnek a körülöttük található objektumokkal. Tipográfiát, méretezést és térközöket alkalmaznak a hierarchia megalkotására és a legfontosabb részek kihangsúlyozására.

²<https://material.io/>

Az alábbi két képen látható a minőségbeli különbség a Material használatával és anélkül (4.5. ábra).



4.5. ábra. Bejelentkezési képernyő a Material komponensek és alapelvek alkalmazása előtt és után.

4.6. Jetpack Navigation Component

A Navigation Component a Google megoldása a képernyők közti navigáció megvalósítására. Jetpack fantázianévvel látták el azokat a komponenseket és megoldásokat, melyekkel hosszú távon terveznek dolgozni, így mindenképp szerettem volna megismerkedni a használatával, mert később is hasznos lehet ez a tudás.

Itt három dologra van szükség a navigáció megvalósításához: egy navigációs gráfra, egy NavHost fragmentre és egy NavController-re. A gráf tartalmaz minden ehhez kapcsolódó információt, ami magába foglalja a lehetséges célképernyőket és ezek közti útvonalakat. A NavHost egy üres konténer, mely megjeleníti az éppen aktuális képernyőt, amire a felhasználó navigált. Ennek alapértelmezett implementációja a NavHostFragment, melyben a fragmenteket tudjuk megjeleníteni. A NavController pedig összeköti a kettőt, irányítja a NavHost-on belüli navigációt és ő felelős a tartalom kicseréléséért amikor a felhasználó új oldalra lép.

A könyvtár sok kényelmi funkciót tartalmaz, többek között a SafeArgs plugin segítségével típusbiztossá tehetjük a navigációkor átadott argumentumokat. [22]

4.7. JUnit 4

Egy applikáció tesztelésének számos aspektusa van, és egyedül ezen aspektusok összessége tudja maximálisan biztosítani a helyes működést. A következő pár mondatban összefoglalom a tesztek legfőbb típusait, majd kitérek arra, amit én is alkalmaztam a projektben.

A unit tesztek célja, hogy a program különböző részeit izoláljuk egymástól, és más osztályoktól, moduloktól függetlenül validáljuk a működésüket. Az integrációs tesztek során a szoftver különböző moduljait együtt egy csoportként teszteljük, hogy a közös működés helyességét ellenőrizzük.[23] End-to-end tesztelésnél teljes folyamatokat ellenőrzünk az elejétől a végéig, mint például egy bejelentkezés vagy egy online vásárlás.[24] A UI tesztek során pedig a felhasználói input helyes kezelésén és a felület megfelelő elrendezésén kívül azt is ellenőrizni kell, hogy a felhasználó számára kellőképpen intuitív és könnyen kezelhető-e az alkalmazás.[25]

A fejlesztés során elsősorban a unit tesztekre koncentráltam, mert egy alkalmazás tesztelésének ez a legelemibb lépése. Ebben könnyíti meg a dolgunkat a JUnit, mely egy Java nyelvhez készült, nyílt forráskódú tesztkeretrendszer. Mivel a Java és a Kotlin teljesen átjárható, így ez tökéletes Android teszteléshez is. Segítségével tesztosztályokat, azon belül pedig teszteseteket definiálhatunk, melyeket aztán lefuttathatunk bármikor, így például refaktorálás vagy új funkció bevezetése során megbizonyosodhatunk róla, hogy a változtatások nem törtek el semmit a kódbázisban. Általában minden projektbeli osztályhoz egy külön tesztosztályt hozunk létre, a különböző funkciókat egy-egy tesztfüggvénybe szétválasztva, hogy ne függjenek egymástól.

A keretrendszer annotáció alapú, ami annyit jelent, hogy a megfelelő kódrészletek megjelölésével jelezhetjük, hogy milyen funkciót tölt be. Az elérhető legfontosabb annotációk a következők:

- *@Test(timeout)*: Az ezzel megjelölt függvények a tesztesetek, külön-külön végrehajthatók lesznek. A zárójelben található opcionális *timeout* tulajdonság megadásával lekorlátozhatjuk a futás idejét, így ha a teszteset túllépi a megadott időt, akkor a teszt sikertelen lesz.
- *@Before*: Megjelöli a függvényt, mely a tesztosztályban minden teszteset előtt le fog futni, egyfajta előkészítésként.
- *@BeforeClass*: Hasonló, mint a *Before*, de csak egyszer fut le, az összes teszt végrehajtása előtt.
- *@After*: A tesztosztály jelölt függvényét a keretrendszer minden egyes teszteset lefutása után meg hívja.
- *@AfterClass*: A *BeforeClass*-hoz hasonlóan működik, csak az összes teszt lefutása után hívódik meg egyszer.

Ezekon kívül a keretrendszer még biztosít egy *Assert* nevű osztályt, mely számos függvényt tartalmaz az eredmények ellenőrzésére. Ezek közül a legtöbbet használt az *assertEquals*, melynek paraméterként kell megadni az elvárt és a kapott eredményt, ő pedig összehasonlítja, és eltérés esetén meghíúsítja a tesztesetet. [26]

4.8. Mockito

A unit tesztelés során követelmény, hogy az alkalmazás többi részétől teljesen izoláltan validáljuk a működést. Ez felvet néhány problémát, mivel egy program működése során elég ritka az, hogy egy osztály teljesen független legyen a többitől. Ilyen esetekben *mockolni*

szoktuk a tesztelendő rendszer határain kívül eső objektumokat. Ez annyit tesz, hogy az osztály függőségeit helyettesítjük nem valódi objektumokkal, és ezeknek specifikáljuk az elvárt működését.

A Mockito ezt a folyamatot hivatott könnyebbé tenni. A *@Mock* annotációval tudjuk megjelölni azokat a változókat, melyeket helyettesíteni szeretnénk, majd a *MockitoAnnotations.openMocks* függvényt kell meghívni, hogy a keretrendszer ezeket legenerálja. Ezután a teszteseteinkben ellenőrizhetjük az interakciókat illetve a mockolt objektumok hívásait úgynevezett *stub* metódushívásokkal helyettesíthetjük. Erre szolgál a *when().thenReturn()* függvénypáros, ahol a *when* paramétere a metódushívás, a *thenReturn* paramétere pedig az eredmény, amit a mock objektum visszaad az adott hívás során az eredeti futás eredménye helyett.

Ezáltal a tesztjeink izolálttá válnak, mivel nem fognak függni egy létező objektum állapotától, és ténylegesen egy egységként tudjuk őket kezelni.

5. fejezet

Követelmények

A fejlesztés során alapvető célom volt egy, a modern paradigmáknak és a felhasználók elvárásainak megfelelő applikáció megalkotása. Törekedtem az objektumorientált szemlélet alkalmazására, a felelőségek szétválasztására és a maximális felhasználói élmény nyújtására. Fontos volt, hogy az alkalmazás folyamatai ne térjenek el nagymértékben attól, mint amit a felhasználók más applikációkban megszokhattak, és minden művelet meg legyen valósítva az adatokon, amire szükségük lehet.

Ezek alapján az alábbi követelmények fogalmazódtak meg az alkalmazással szemben:

1. A felhasználónak legyen lehetősége fiókot létrehozni, e-mail cím és jelszó megadásával bejelentkezni, illetve fiókjából ki is lépni.
2. Az adatok legyenek perzisztensen tárolva, az alkalmazás bezárásával, abból való kilépéssel vagy egy esetleges készülékcseré esetén se vesszenek el.
3. Az adatok csak bejelentkezés után váljanak láthatóvá, minden felhasználó csak a saját adataihoz férhessen hozzá.
4. A felhasználó képes legyen kategóriákat létrehozni a jegyzetek számára, ezeket akár tetszőleges mélységben további kategóriákba ágyazni.
5. Legyen lehetőség jegyzetek létrehozására dokumentumok lefényképezése által. A szöveg digitalizálása után a jegyzet legyen címmel ellátható, kategóriába sorolható és a tartalma szerkeszthető.
6. Inkonzisztens adatok létrehozására ne adjon lehetőséget, a felhasználói input mindig legyen validálva.
7. A jegyzeteken és kategóriákon lehessen minden fő műveletet - létrehozás, olvasás, módosítás, törlés - elvégezni, ezek során a felhasználói felület és az adatbázis maradjanak konzisztensek egymással.
8. Módosítás során lehessen a jegyzetet újabb fényképek készítésével kiegészíteni, az ily módon digitalizált szöveg kerüljön hozzáfűzésre az eredeti tartalomhoz.
9. A UI esztétikai és felhasználói élmény szempontjából legyen megfelelő, a hosszú ideig tartó folyamatokat jelezze a felhasználónak töltőképernyő segítségével.
10. A jegyzetek listája lehessen rendezhető és kereshető.

6. fejezet

Alkalmazás működése

A következőkben az alkalmazás funkcióit fogom bemutatni, a leírásokat képernyőképekkel kiegészítve. A képek néhol eltérnek egymástól, melynek oka hogy az applikáció funkcionálisából fakadóan nem volt lehetséges mindet az emulátoron elkészíteni, hanem a jegyzetek készítéséhez fizikai készülékre is szükség volt.

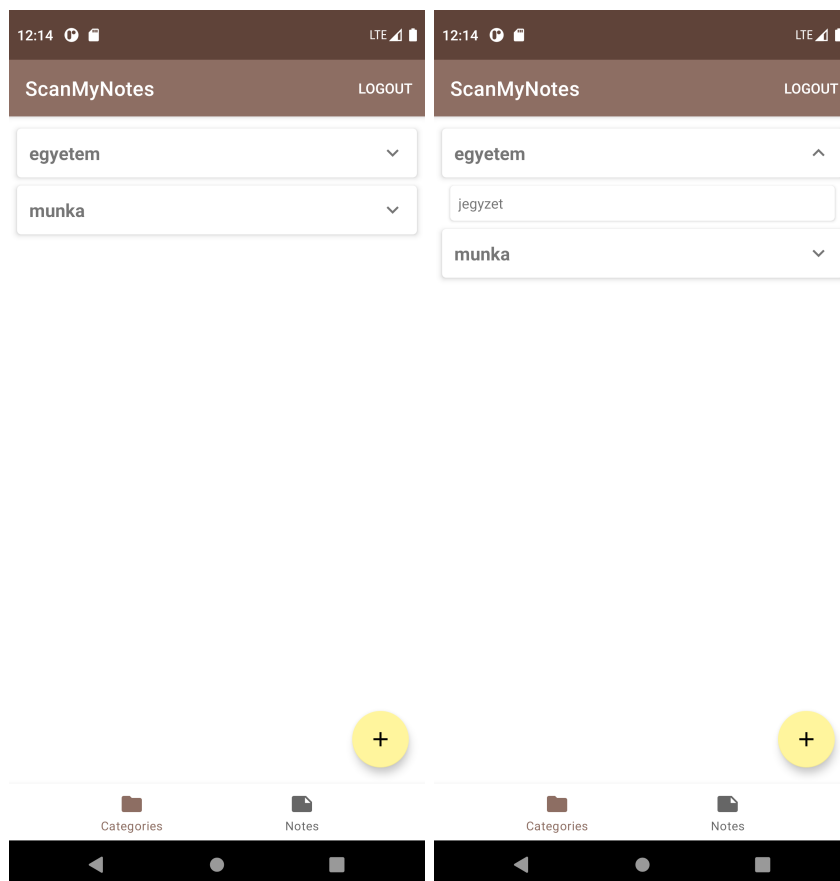
6.1. Bejelentkezés

A telepítést követően a bejelentkezési képernyő az első, amivel a felhasználó találkozik. Ez már korábban megjelent a dolgozatban, a 4.5. ábra bemutatásában. Itt a beviteli mezők segédszövegei egyértelműsítik, hogy milyen adatok elvártak a bejelentkezéshez illetve regisztrációhoz. Ezek gombnyomás hatására validálásra kerülnek, és amennyiben az e-mail formátuma nem érvényes vagy a jelszó hossza nem éri el a 6 karaktert, akkor a folyamat meghiúsul, és a felhasználó értesül róla, hogy mely mező(k) tartalmát kell javítania.

6.2. Kategorizált lista

Sikeres bejelentkezést követően az alkalmazás a fő képernyőjére navigál, ahol az eddig elmentett jegyzeteket láthatjuk kategóriákba rendezve. A kategóriák alapértelmezetten össze vannak csukva, csak a legfelső szinten található elemeket látjuk. A lista sorainak jobb szélén elhelyezkedő nyilakra kattintva tudjuk kibontani az adott kategóriát, ezzel megtekinteni a tartalmát (6.1. ábra).

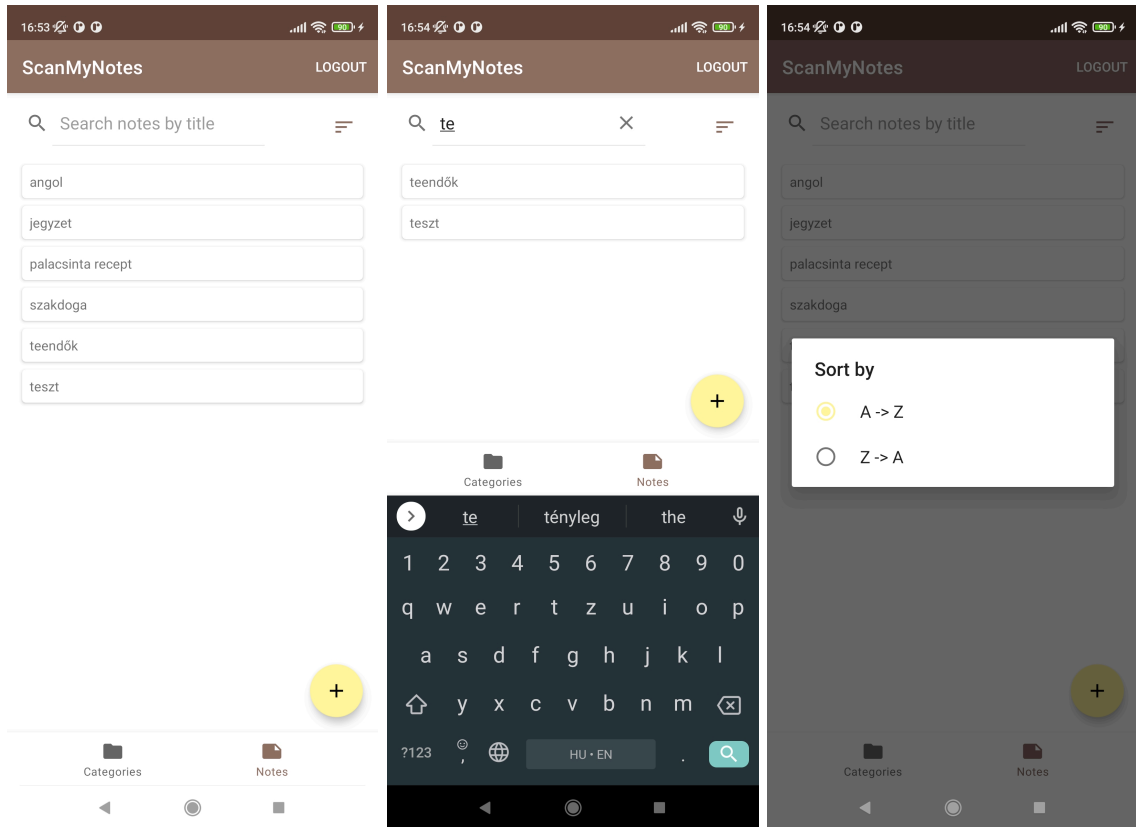
Innen számos lehetőségünk adódik navigációra az applikáción belül, kezdve a jobb felső sarokban található, meglehetősen magától értetődő kijelentkezés gombbal. Ezt megnyomva a fent leírt bejelentkezési képernyőre navigálunk, ahol adataink megadásával újra bejelentkezhetünk.



6.1. ábra. Az alkalmazás kezdőlapja alapértelmezett állapotban, illetve egy kategória kibontva.

6.3. Jegyzetlista

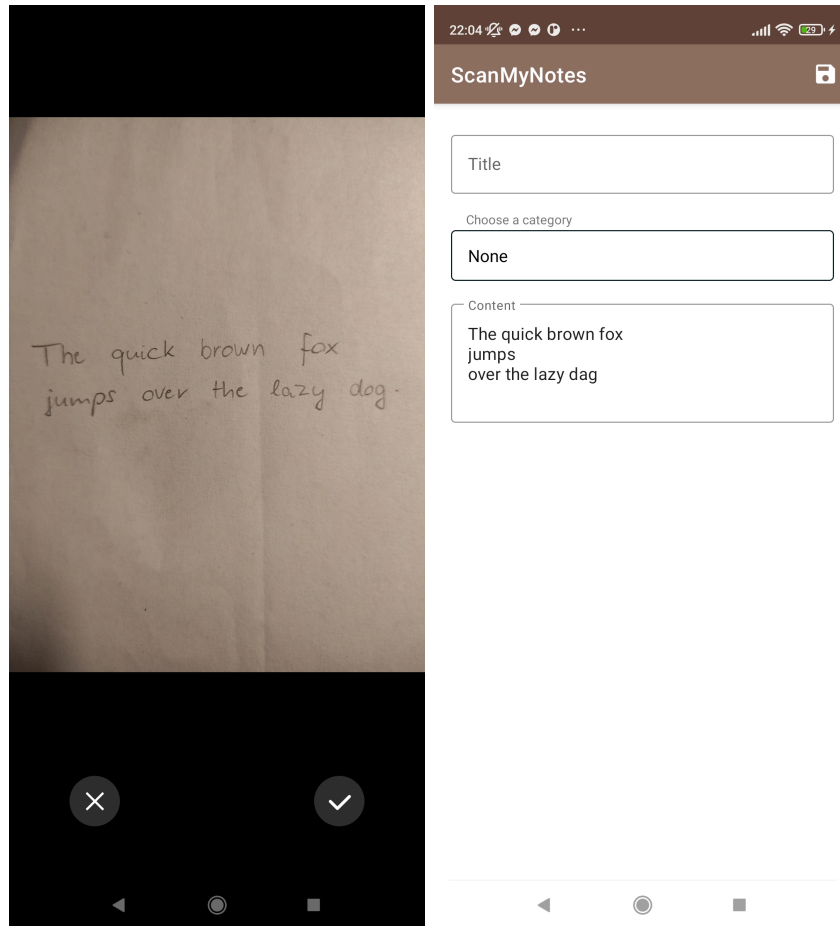
A főképernyőn alul egy navigációs sávot találunk, mellyel a két különböző listamegjele-
nítés között tudunk váltani. Míg a *Categories* opció alatt egy hierarchikusan egymás alá
rendezett listát láthatunk, a *Notes* opció csak a jegyzeteket tárja elénk, kategóriától füg-
getlenül. Itt több lehetőség tárul elénk: a képernyő tetején található egy keresősáv és egy
rendezés gomb. Keresni a jegyzetek címe alapján tudunk, itt gépelés közben azonnal szű-
kül az eredményhalmaz. A rendezés szintén a cím alapján működik, jelenleg növekvő és
csökkenő betűrendet támogat az alkalmazás (6.2. ábra).



6.2. ábra. A jegyzetek listája, és a rajta elvégezhető műveletek.

6.4. Jegyzet létrehozása

Az alkalmazás fő funkcionálisága a jegyzetek tárolása, így elég fontos, hogy legyen lehetőség újak létrehozására. Ez a képernyő jobb alsó sarkában található plusz gombra kattintva tehető meg. Az ott megjelenő két újabb gomb közül az alsó megnyomására felugrik egy kameraablak, ahol egy fénykép készítése után megtörténik a digitalizáció, és a szerkesztési oldalra ugrunk. Itt egy cím megadásával fejezhetjük be a létrehozási folyamatot, de opcionálisan hozzárendelhetjük egy kategóriához is (6.3. ábra). Amíg a cím vagy a tartalom üres, addig az alkalmazás nem fogja engedni elmenteni a jegyzetet, figyelmeztetést rak az üresen hagyott mezőre.

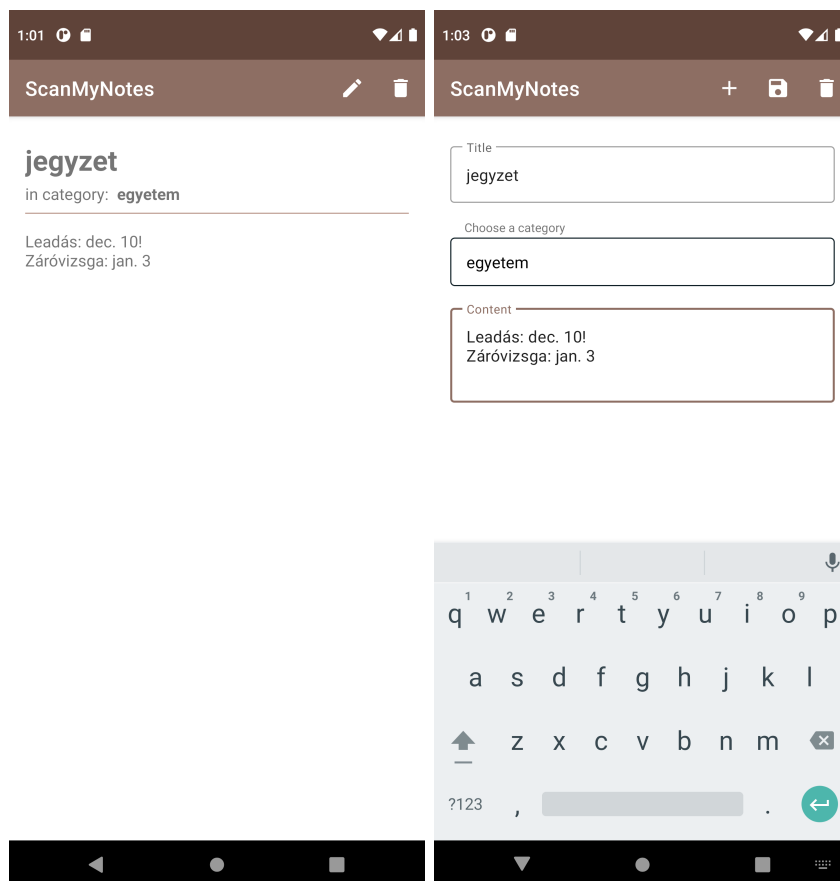


6.3. ábra. A létrehozás során készített kép, illetve az abból kialakuló jegyzet szerkesztése.

6.5. Jegyzet szerkesztése

Amennyiben a listában egy jegyzetre kattintunk, illetve újat hozunk létre, akkor annak részletes oldalára navigálunk. Itt megtekinthetjük a tartalmát, a jobb felső sarokban található ceruza ikonra nyomva pedig szerkeszthetjük is (6.4. ábra). Megváltoztathatjuk a címét, tartalmát, kategóriáját, a fent megjelenő pluszjel segítségével pedig készíthetünk újabb fotót, melynek szövege hozzáfűzésre kerül az eddigihez. A fentebb említett megkötések itt is érvényesek, azaz a cím és a tartalom nem lehet üres az elmentés pillanatában.

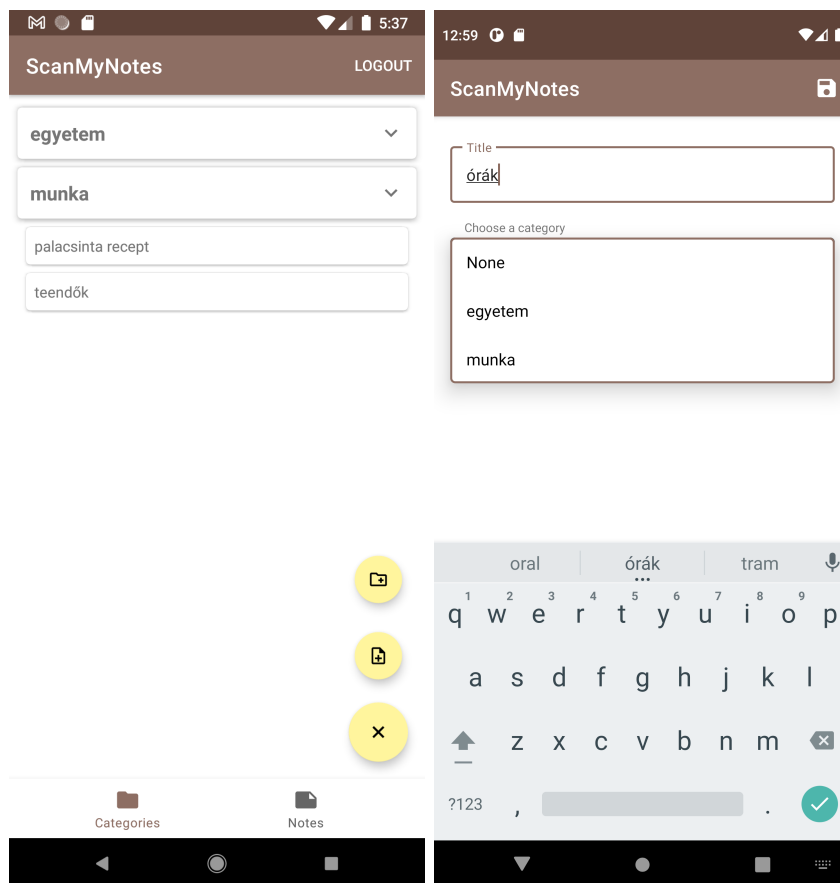
Mind megtekintési, mind szerkesztési módban elérhető a fenti sávban a szemetes ikon, mellyel törölhetjük a jegyzetet a listánkból. Vigyázat, ez a művelet nem visszafordítható!



6.4. ábra. A jegyzet megtekintési, illetve szerkesztési képernyője.

6.6. Kategória létrehozása

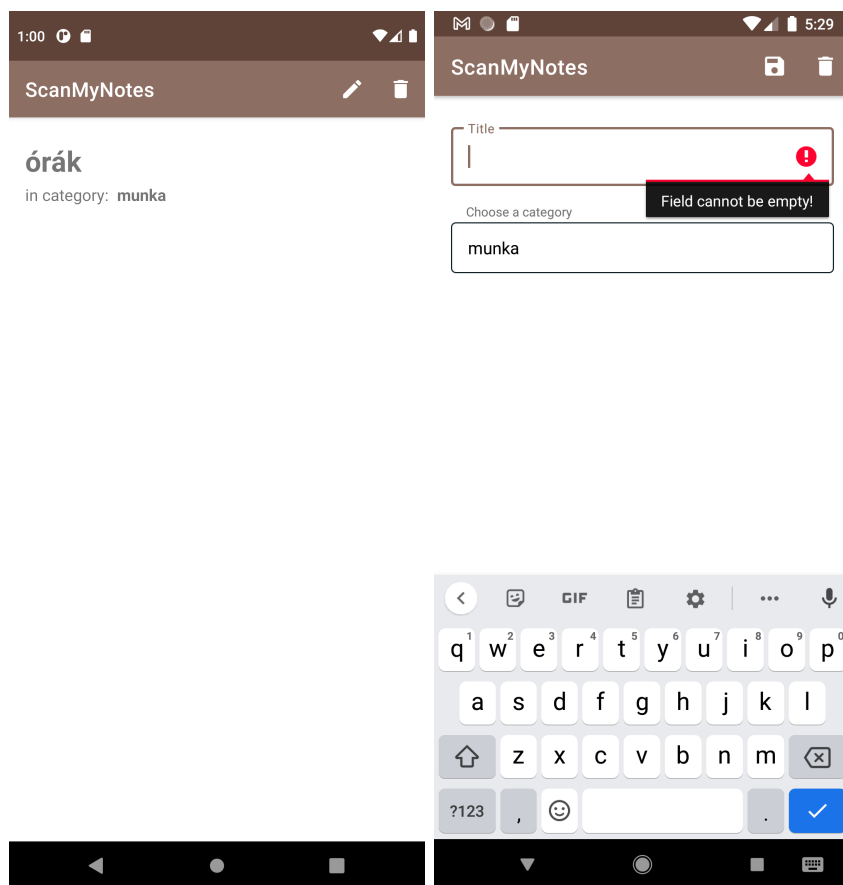
Az alkalmazásban elérhető másik adattípus a kategória, mely rendszerezési célt szolgál. Képes magába foglalni jegyzeteket és más kategóriákat is, tetszőleges mélységben. Szintén a jobb alsó sarokban található gomb biztosítja a létrehozás lehetőségét, ám ebben az esetben a felugró két kisebb gomb közül a felsőt kell választani. Itt egy, a jegyzetkészítéshez nagyon hasonló oldalon tudunk címet és opcionálisan szülőkategóriát megadni, és amennyiben nem üres a cím mezője, el is menthetjük (6.5. ábra).



6.5. ábra. A létrehozás gomb kinyitott állapotban, új kategória felvétele.

6.7. Kategória szerkesztése

Új kategória létrehozása után, illetve a listában egy kategóriára nyomva annak részleteit tekinthetjük meg. Itt megjelenik a címe és esetleges szülője, és jobb fent szintén található egy ceruza ikon, mely lehetővé teszi a szerkesztést (6.6. ábra). Hasonlóan a jegyzet megtekintés és szerkesztés során is törölhetjük az adott kategóriát, ilyenkor egy felugró ablak figyelmeztet rá, hogy a törlés során az összes tartalmazott objektum is törlődni fog.



6.6. ábra. A kategória részletes képernyője, illetve a szerkesztési képernyő által feldobott hiba, ha üresen hagyjuk a címet.

7. fejezet

Implementáció

Egy alkalmazás megvalósítása mindig komplex folyamat, mely sok kisebb részfolyamatot foglal magába. Az alábbiakban a fejlesztés részleteiről fogok írni, először a megvalósítást mutatom be a projektfelépítéstől kezdve a logikán át a felhasználói felületig, majd egy kicsit kitérek a tesztelésre is.

A következőkben az alkalmazás implementációjába fogok részletesebben belemenni a projektfelépítéstől kezdve a logikán át a felhasználói felületig.

7.1. Projektfelépítés

A projekt elrendezése úgy lett kialakítva, hogy megfeleljen az Android fejlesztéshez ajánlott konvencióknak, emellett a package-ek nevei egyértelműek legyenek és könnyedén meg lehessen találni bármit, amit keresünk. A felépítés a következő:

- *data*: Az adatelérési réteg osztályait tartalmazza. Két package található benne: egy *models* és egy *network*. Előbbi a hálózati modell(ek)e)t tartalmazza, utóbbi pedig az API-kat és a DataSource-okat.
- *di*: Az architektúra ajánlása alapján ebbe a package-be kell tenni a függőséginjektálás működéséhez szükséges fájlokat.
- *domain*: A domain réteg tartalmazza az üzleti logikát. Ez általában interactorokból és üzleti modellekből áll. Jelen applikációban az egy darab interactort és a modelleket jelenti.
- *ui*: Ez tartalmazza a felhasználói felülethez kapcsolódó összes kódot, azaz itt találhatók a Fragmentek, ViewModel-ek, ViewState-ek, illetve egyéb, megjelenítéshez kapcsolódó szükséges osztályok.
- *util*: Ide kerültek azok a kódok, amiket máshova nem lehetett beilleszteni. Egyetlen fájl található benne, olyan kiegészítésekkel, amiket több osztály is használ, ezért nem lehetett egy funkcióhoz kötni.

7.2. Szerveroldali komponensek integrációja

Az alkalmazás backendjét a Firebase szolgáltatja, melynek kiterjedt és alapos dokumentációja nagyban megkönnyíti az integrációs folyamatot.



Első lépésként be kell jelentkezni a Firebase console¹-ba, és létrehozni az alkalmazás számára egy Firebase projektet. Amikor kész, akkor ehhez hozzá kell adni a már létező

¹<https://console.firebase.google.com/>

applikációt azáltal, hogy megadjuk az alkalmazás package nevét². Ezután generálódik egy *google-services.json* nevű fájl, melyet le kell tölteni, és a lokális projekt *app* mappájába beilleszteni. A projekt szintű *build.gradle* fájlba fel kell venni a Google Services Gradle plugint, mint függőséget, és ezt a modul szintű *build.gradle* fájlban egy *apply plugin* parancs felvételével alkalmazni kell. Végül pedig a használni kívánt Firebase szolgáltatásokat is fel kell venni függőségként az utóbbi fájlba, és egy gradle szinkronizálás után elérhetővé válnak az alkalmazásban.

A projektben négy Firebase szolgáltatás került felhasználásra, melyek az Authentication, Firestore, Analytics és Crashlytics. Az alkalmazásban használt kódon kívül a menedzselésük a fentebb említett console-ban lehetséges, mely rendkívül kényelmes. Minden modul külön-külön be- és kikapcsolható a projekt igényeinek megfelelően, így az alkalmazás növekedésével később akár plusz funkciókat is bevezethetünk, mindössze egy kattintással.

Az első szolgáltatás az Authentication, mely elég fontos aspektusa a működésnek. Bekapcsolás után számos lehetőség tárul elénk, de teljes mértékben miénk az irányítás: alapértelmezetten minden ki van kapcsolva, csak azt fogjuk tudni használni, amit mi magunk kapcsolunk be. A *Sign-in method* fülre kattintva tudunk egy vagy több bejelentkezési módot választani, melyet az alkalmazásban használni szeretnénk. Támogatja az autentikációt e-mail és jelszó, telefonszám, Facebook, Google, Apple és GitHub használatával, és ezeken kívül még néhány módszerrel. A projektben az e-mail és jelszó kombinációját választottam, ugyanis ahhoz vannak a legjobban hozzászokva a felhasználók, az e-mail cím az, ami tényleg mindenkinek van, ellentétben egy Apple vagy egy Google fiókkal. Legalább egy bejelentkezési módot bekapcsolva pedig a *Users* fülön láthatjuk a regisztrált felhasználók listáját (7.1. ábra)

Identifier	Providers	Created ↓	Signed In
test2@test.hu		Nov 27, 2021	Nov 27, 2021
test@test.hu		Nov 27, 2021	Dec 4, 2021

7.1. ábra. A regisztrált felhasználók listája.

A bejelentkeztetést a *ui.login* package-ben található *LoginFragment* végzi. Ő a többi képernyővel ellentétben csak egy sima Fragment, nem a RainbowCake architektúra része, ugyanis tulajdonképpen két függvényből áll az egész, így nem tartottam célszerűnek végigvezetni a rétegeken. Ahogy azt a 4.5. ábra korábban bemutatta, egyetlen képernyő áll rendelkezésre mind a bejelentkezéshez, mind a regisztrációhoz. Ez a döntés azért született meg, mert az alkalmazás jelenlegi funkcionalitásában nincs szerepe a felhasználói profilnak, nincs szükség semmilyen plusz információra a felhasználótól, ami később bármilyen formában felhasználásra kerülne, így felesleges két teljesen ugyanolyan UI-t készíteni csak a megkülönböztetés kedvéért.

Az autentikáció végrehajtásához a Firebase biztosít egy *FirebaseAuth* típusú objektumot, melyen keresztül elérhető az összes ezzel kapcsolatos művelet. Meg lehet nézni, hogy van-e bejelentkezett felhasználó, és ezzel átugorható a bejelentkezési folyamat, ha valaki korábban már használta az alkalmazást. Ha nem, akkor pedig (e-mail és jelszavas bejelentkezés esetén) a *signInWithEmailAndPassword* és a *createUserWithEmailAndPassword* függvények állnak rendelkezésre, melyeknek az e-mail címet és a jelszót paraméterként megadva elvégzik a megfelelő műveletet. Az eredményt aszinkron módon küldi, amire egy

²A package név egy egyedi azonosító, mely alapján egyértelműen meghatározható az alkalmazás. Két ugyanolyan package névvel rendelkező applikáció nem telepíthető fel egy eszközre, de még a Google Play Store-ba sem kerülhet.

onCompleteListener segítségével lehet feliratkozni. Ezen belül ellenőrizhető a hívás kimenetele, és sikeres művelet esetén át tudunk navigálni az alkalmazás megfelelő képernyőjére, hiba esetén pedig kezelni tudjuk azt.

A következő szolgáltatás a Firestore, mely adatok felhőalapú tárolását teszi lehetővé egy hierarchikus struktúrában. Bekapcsolása után elérhetővé válik a kezelőfelülete, ahol szabályokat adhatunk meg a hozzáférésre, illetve láthatunk minden tárolt adatot egy könnyen átlátható vizuális reprezentációban. Az alkalmazás adatai az (7.2. ábra) által mutatott módon kerülnek tárolásra.

Az összes adat egy *users* nevű kollekcióban található, amiben minden felhasználóhoz tartozik egy dokumentum, melynek neve a regisztrációkor generált UID. Mivel ez garantáltan egyéni, így mindenki csak a saját adatait fogja elérni, és nem fordulhat elő, hogy egy felhasználó tudtán kívül felülírja egy másik adatait. A felhasználók dokumentumai két kollekciót tárolnak: egyet a kategóriáknak, és egyet a jegyzeteknek. Ezekben belül egy-egy dokumentum tárol egy-egy jegyzetet illetve kategóriát, melyeknek szintén egyéni azonosítója van.

Az alkalmazásban a *data.network* package-ben található *FirebaseApi* nevű osztály végzi az adatok lekérését és objektumokká való konvertálását. Az adatbázist egy *FirebaseFirestore* típusú objektumon keresztül érem el, a 7.3. ábra által mutatott kóddal.

scanmynotesfree	users	5I41jZ42yGhcBqkKc7FPiubFKgK2
+ Start collection	+ Add document	+ Start collection
users >	5I41jZ42yGhcBqkKc7FPiubFKgK2 >	categories
	I7R9MC4BDnV18RVFXhFPaZANmZX2	notes
	Tb6f0juZqnVXBuHhJJLOUBofWk03	

7.2. ábra. A felhasználók adatainak struktúrája.

```
val categoriesRef = db.collection(usersCollection).document(userId).collection(
    categoriesCollection)
val categoriesSnapshot = try {
    categoriesRef.orderBy(titleString).get().await()
} catch (error: FirebaseFirestoreException) {
    return Result.failure(error.message.toString())
}

val categories = categoriesSnapshot.map { document ->
    val category = document.toObject<Category>()
    category.id = document.id
    category ^map
}

return Result.success(categories)
```

7.3. ábra. A kategóriák lekérésének kódja az adatbázisból.

Itt a *db* változó az a bizonyos objektum, melyen keresztül elérhető az adatbázis. A hívástól függően egy *CollectionReference* vagy egy *DocumentReference* lehet az eredmény, ami az adott kollekció/dokumentum helyére mutat. Ezek felhasználhatók adatok írására és olvasására, ha pedig nem létezik adat azon a helyen, akkor létrehozza azt. Alapértelmezetten a referenciákon végzett minden hívás aszinkron módon hajtódik végre, de ez a működés

nem volt ideális az alkalmazásom számára, ezért a `kotlinx.coroutines.tasks.await()` függvény segítségével bevárom a lekérdezés eredményét. Ezután már csak annyi van hátra, hogy a visszakapott Snapshot objektumból kialakítsam azokat a típusokat, amikre szüksége van az alkalmazásnak. Szerencsére ez rendkívül egyszerű, mivel a Firestore biztosít egy `toObject` függvényt, melynek a kívánt típust template-paraméterként megadva elvégzi a konverziót. Az id-t pedig azért kell még utána beállítani a kapott objektumon, mert az nem egy tárolt mezője a dokumentumnak, hanem az azonosítója, így arra nem terjed ki a `toObject`.

A harmadik szolgáltatás az Analytics, mely rengeteg hasznos információt szolgáltat a felhasználókról és szokásaikról. Az előzőekhez hasonlóan itt is egy objektum biztosítja a kívánt funkcionalitást, melynek típusa ez esetben `FirebaseAnalytics`. Ez figyel pár darab beépített eseményt, mint például a képernyők látogatását vagy a munkamenet-kezdést, de sajátot is lehet definiálni (7.4. ábra). Itt egy új felhasználó regisztrációját rögzíti a rendszer annak metódusával együtt, ami jelen esetben e-mail és jelszó.

```
val bundle = Bundle()
val method = "emailpassword"
bundle.putString(FirebaseAnalytics.Param.METHOD, method)
firebaseAnalytics.logEvent(FirebaseAnalytics.Event.SIGN_UP, bundle)
```

7.4. ábra. Egyéni esemény létrehozása és logolása Analytics segítségével.

Az utolsó Firebase szolgáltatás, amit az alkalmazásban felhasználtam, a Crashlytics. Az előzőektől ellentétben az első bekezdésben leírtakon kívül ennél még szükség van további integrációs lépésekre is a működéshez. Van ugyanis egy saját pluginja, amit a fentiekkel megegyező módon fel kell venni mindkét gradle fájlba. Ezt követően szükség van egy teszt crash előidézésére, majd az alkalmazás újraindítására, hogy az elküldhesse a hibajelentést a Crashlyticsnek. Ezt követően viszont nincs már szükség semmi másra, a kódba sem szükséges semmit felvenni, mert automatikusan figyeli az alkalmazást. Természetesen számos lehetőségünk van finomhangolni a működését egyéni kulcsok hozzáadásával, de akár a nem-végzetes hibák figyelésével is.

7.3. Képfeldolgozás integrációja

Az alkalmazás fő funkciója a kézírás-digitalizálás, amit a Google Cloud Vision API végez. Ennek az ökoszisztémája nagyon hasonló a Firebase-hez, annyi különbséggel, hogy itt a Google Cloud Console³-ban kell létrehozni a projektet. Ezt a kettőt akár össze is lehetne kötni, lévén Google rendszer mindkettő, de akkor a Firebase szolgáltatások is átkerülnek ingyenesről egy úgynevezett *pay as you go*⁴ plan-re. Én pedig szerettem volna minimalizálni az esetleges költségeket, így elvettem ezt a lehetőséget.

Miután ez kész, célszerű a számlázás bekapcsolásával és egy *billing account* létrehozásával kezdeni, mert bár mindent be lehet állítani enélkül is, addig nem fog működni az alkalmazásban az API, ameddig ez nincs rendben.

A következő lépés a kívánt API bekapcsolása, ugyanis itt is minden alapértelmezetten ki van kapcsolva. Ehhez az APIk közül kikerestem a Cloud Vision-t, majd egyetlen gombot kellett megnyomnom. A működéséhez azonban ennyi még nem elég, mert a használatához autentikálni is kell a kliensalkalmazást. Ehhez én egy API kulcsot hoztam

³<https://console.cloud.google.com/>

⁴A pay as you go olyan fizetési opció, mely során egy bizonyos kvóta alatt ingyenesen használhatók a szolgáltatások, afölött pedig annyit kell fizetni, amennyit használjuk.

létre, és létrehoztam egy `apikey.properties` nevű fájlt, melybe felvettem. Ezt úgy lehet felhasználni a projektben, hogy a 7.5. ábra és a 7.6. ábra kódrészleteit fel kell venni a modul szintű `build.gradle` fájlba. A második ábrán látható, ahogy egy `string` típusú és `VISION_API_KEY` nevű változót csinálunk a fájlból kiolvasott kulcsból. Mindez azért szükséges, mert az API kulcs egy érzékeny adat, annak segítségével bárki felhasználhatja a saját alkalmazásában a szolgáltatást az én nevem alatt. Tekintve pedig, hogy fizetős szolgáltatásról van szó, ezt még komolyabban védeni kell, hogy ne szenvedjek anyagi kárt. Ezzel a módszerrel pedig rendkívül egyszerű ezt megtenni, hiszen a verziókezelő rendszerbe való feltöltés előtt csak annyit kell tenni, hogy az `apikey.properties` fájlt kivesszük a követés alól.

```
def apikeyPropertiesFile = rootProject.file("apikey.properties")
def apikeyProperties = new Properties()
apikeyProperties.load(new FileInputStream(apikeyPropertiesFile))
```

7.5. ábra. A létrehozott `apikey.properties` fájl használata a projektben.

```
buildConfigField 'String', 'VISION_API_KEY', apikeyProperties['google_vision_api_key']
```

7.6. ábra. A megfelelő API kulcs kiolvasása a fájlból.

A képfeldolgozás megvalósítása a következő: a fotózás befejezésével a kép egy `Bitmap` objektumként érkezik meg az interactorhoz, ahol a hívás háttérszálla kerül. Ezáltal már biztonságosan lehet hosszabb műveleteket végezni, így itt végzem el kép base64 kódolását, mely során egy Vision API által biztosított `Image` típusú objektumot állítok elő. A további hívások már ezt az objektumot kapják paraméterként, így a `DataSource`-on keresztül a `VisionApi` osztályhoz jut. Ez végzi a további előkészítési folyamatot, küldi el detektálásra a képet, és az eredményt a megfelelő formában adja vissza. Mindez meglehetősen egyszerűen megoldható az API saját objektumainak használatával. Itt először megadom az API kulcsot, majd specifikálom a kívánt detektálási módot. Ez azért fontos, mert az API a kézírás-felismerésen kívül számtalan másra is képes, így nekem kell kiválasztanom, hogy pontosan mit szeretnék elvégeztetni. Ezután a képből és a választott funkcióból egy `BatchAnnotateImageRequest` objektumot készítek (ami jelenleg egyetlen képet tartalmaz, de van lehetőség egyszerre több kép feldolgozására is), és az `annotate` függvény segítségével megkapom ennek az annotációját. A teljes szöveget az eredmény `fullTextAnnotation` tagváltozója tartalmazza.

7.4. Adatmodellek elkészítése

Az alkalmazásban szükség volt olyan osztályokra, melyek az adatbázisból visszakapott adatokat fogják össze, és szállítják a rétegek között. Ezeket hívják adatmodellnek, és a megvalósítás során négy darabot készítettem. Ebből három a `domain.models` package-ben található, ezek egyben a megjelenítési modellek is, egy pedig a `data.models` package-be tartozik, ugyanis hálózati hívások eredményét reprezentálja.

Ez utóbbi a `Result` nevű osztály, és tulajdonképpen a visszakapott adatokat vagy hibaüzenetet csomagolja be a könnyebb hibakezelés érdekében. Egy sablonosztályról⁵ van szó, mely két belső `data class`-t tartalmaz: egy `Success`, illetve egy `Failure` nevűt. A `Success` a sikeres hívások eredményét tárolja egyetlen `data` nevű paraméterében, mely tetszőleges

⁵Olyan osztály, mely template paramétereket használ, ezáltal tetszőleges adattípussal használható.

adattípust felvehet, a Failure pedig a sikertelen hívások hibaüzenetét továbbítja. Emellett *companion object*⁶-ben tartalmaz egy-egy függvényt, melyekkel a két állapot valamelyikét lehet inicializálni. A FirebaseApi összes függvényét úgy implementáltam, hogy valamilyen Result típust adjon vissza, így a fentebbi rétegekben minden esetet könnyedén le lehet kezelni.

A domain modellek közül az első és legalapvetőbb a ListItem osztály. Ő egy *open class*, mivel Kotlinban csak akkor származtathatunk le egy osztályból egy másikat, hogyha a szülőt megjelöljük az *open* kulcsszóval. A ListItemnek pedig pont ez a funkciója, hogy közös őst biztosítson a kategóriák és a jegyzetek számára, ami főleg a kategorizált lista építésénél fontos. Tartalmazza a két osztály közös adattagjait, melyek az *id*, *title* és *parentId* névre hallgatnak. Ezek közül az utolsónak lehet null az értéke, amennyiben az adott elemnek nincs szülője a listában.

Ennek leszármazottai a Note illetve a Category nevet kapták. Ők az alkalmazásban tárolt és megjelenített jegyzeteket, illetve kategóriákat reprezentálják. Ősükhöz képest egy-egy adattaggal bővültek, a Note *content* adattagja a jegyzetek szövegét tárolja, míg a Category *listItems* listájában a tartalmazott objektumai találhatóak. Ez utóbbi nincs eltávolva az adatbázisban, hiszen könnyedén előállítható az adatok lekérdezése után a *parentId* adattagok alapján.

7.5. Egyedi FloatingActionButton

Az új adatok fevételehez szükség volt valamilyen gombra, aminek a megnyomásával a felhasználó a megfelelő oldalra navigálhat. Mivel ez egy elég fontos funkció, így célszerű kitüntetett helyre tenni az alkalmazásban, ahol a felhasználók azonnal láthatják. Az ilyen típusú gombok megvalósítására készült el a Material Components könyvtár FloatingActionButton névre keresztelt komponense. Ez egy általában a jobb alsó sarokban elhelyezkedő, tartalom felett megjelenő gomb, mely görgetéstől függetlenül a képernyőn marad, így biztosítva a folyamatos elérhetőséget.

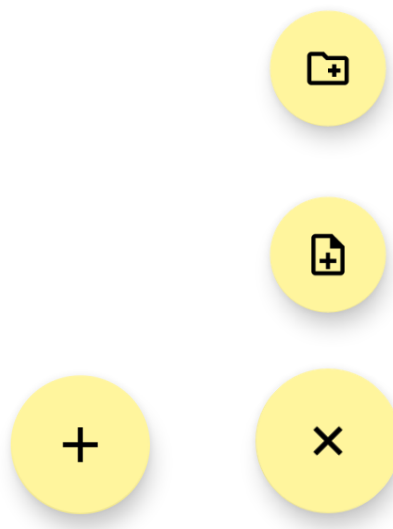
Azonban funkcionalitás szempontjából a kategóriák és a jegyzetek létrehozásához nekem két ilyen gombra lett volna szükségem, ami viszont szembemegy a Material Design alapelveivel. Ezért döntöttem úgy, hogy ezt felhasználva megalkotom a gombot amire szükségem van (7.7. ábra).

Működése a következő: alapértelmezett állapotban a bal oldali ábrán megjelenő egyetlen gomb látszik. Erre kattintva egy animációval elfordul 45 fokkal, és közben a másik kettő, kicsit kisebb gomb alulról beúszik és fokozatosan megjelenik. A két újonnan megjelent gomb biztosítja a jegyzetek és kategóriák létrehozásának lehetőségét, az eredeti gombot megnyomva pedig a kezdeti állapot áll vissza.

Ehhez először is a listanézeten elhelyeztem három FloatingActionButtonot a jobb alsó sarokba, egymás fölé. Létrehoztam három erőforrásfájlt az ikonoknak, a legalsó egy pluszjelet kapott, a felette lévő egy új jegyzet ikont, a felső pedig egy új mappa ikont. A felső kettő alapértelmezett láthatóságát *gone*-ra állítottam, ami a helyes működéshez elengedhetetlen, ugyanis ha csak *invisible* állapotban van egy UI elem, akkor ott marad a képernyőn, csak látszani nem fog, ami ezen gombok esetében azt eredményezné, hogy a mögötte lévő terület nem lenne kattintható.

Ezután következtek az animációk, melyeket Androidon szintén erőforrásfájlokkal lehet megoldani XML-ben. A projektben ezeket a *res/anim* mappába kell elhelyezni, és a kívánt hatás eléréséhez négy darabra volt szükségem. Egy ilyen XML fájl gyökér eleme egy *set*,

⁶Kotlinban a companion object tartalmazza tulajdonképpen azokat az adattagokat és függvényeket, amelyeket Javában statikusnak hívtunk. Ezek osztályszintűek, ami annyit jelent, hogy nincs szükség az osztály példányosítására ahhoz, hogy elérjük őket.



7.7. ábra. Az elkészített egyedi gomb alapértelmezett és kinyitott állapotban.

melyben egy UI elem pozícióját, méretét, elforgatását és átlátszóságát animálhatjuk. Az alábbi képen a gomb kinyitási animációjának kódja látható (7.8. ábra).

```
<set xmlns:android="http://schemas.android.com/apk/res/android"
    android:fillAfter="true">

    <rotate
        android:pivotX="50%"
        android:pivotY="50%"
        android:fromDegrees="0"
        android:toDegrees="45"
        android:duration="300"/>
</set>
```

7.8. ábra. A kezdeti gomb elforgatásának animációja XML-ben.

Ami a legérdekesebb, hogy a pivotX és pivotY tulajdonságokkal meghatározhatjuk, hogy az adott elem forgatásakor a középpont hol legyen. Így egészen komplex animációk is létrehozhatók, habár nekem most csak egy egyszerű középpont körüli forgatás kellett.

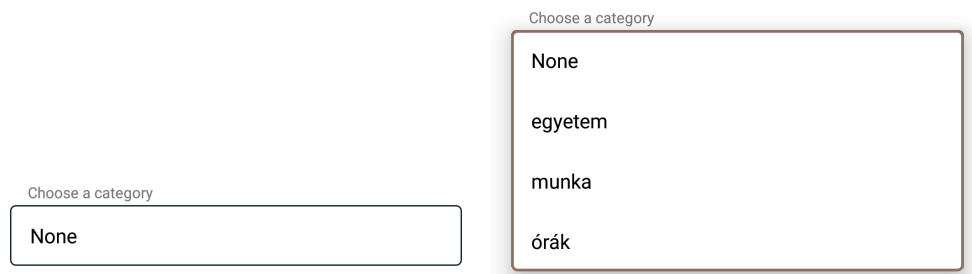
7.6. Egyedi legördülő menü

Létrehozás és módosítás során lehetőséget ad az alkalmazás kiválasztani egy kategóriát, melybe az objektumot tenni szeretnénk. Ennek megvalósítására a legördülő menüt találtam legalkalmasabbnak, melyet Androidon a *Spinner* nevet kapta. Ezt a komponenst egy *ArrayAdapter* segítségével töltöttem fel, mely a neki odaadott kollekció minden elemére visszaad egy, a létrehozásakor megadott layout-ot, ami itt most egy alapértelmezett *simple_spinner_dropdown_item*. Ez remekül biztosítja a számomra szükséges funkciót, csak épp nem túl esztétikus, és semmiképp sem illik a Material elemek közé. Ennek a megoldására két lehetőségem volt: a Material könyvtár részeként érkező *TextField* használata, vagy saját magam próbálok egy hasonló kinézetet elérni a *Spinner* testreszabásával. Az előbbi megoldás nagyon jól passzolt volna az alkalmazásba, de az egész logikát a *Spinner*-

re építve írtam meg, és sajnos a `TextField` használata ettől teljesen eltér, úgyhogy mindent át kellett volna írnom. Ezért az utóbbi opciónál döntöttem.

Ehhez kettő XML fájlt hoztam létre a `res/drawable` mappában, egyet az inaktív, csukott állapot, és egyet a kinyitott állapot hátterének. Itt a `shape` XML-attribútum segítségével tudunk különféle alakzatokat kirajzolni, alapértelmezettként pedig téglalapot rajzol. Nekem pont erre volt szükségem, így csak egy kicsit testre kellett szabni. Az inaktív háttér keretének adtam lekerekített sarkokat, paddinget és vonalvastagságot, emellett a színét beállítottam a Material `TextField` inaktív színére, hogy a lehető legjobban egyezzen a többi elem megjelenésével. A lenyitott háttér ettől annyiban különbözik, hogy a keret színe az alkalmazás fő színét kapta meg, dupla olyan vastag lett, és az egész kapott egy fehér kitöltést, hogy eltakarja a mögötte lévő elemeket.

Ezekkel pedig úgy tudtam felülírni a `Spinner` eredeti kinézetét, hogy az XML-ben ráraktam az `android:background` és `android:popupBackground` attribútumokat, melyeknek értékül adtam a fent leírt két layout-ot. Így már egészen esztétikusra sikerült, és majdnem észrevehetetlen a különbség a többi elemhez képest. Az eredményt a 7.9. ábra szemlélteti.



7.9. ábra. A `Spinner` kinézete becsukva, illetve kinyitva.

7.7. Képernyők felépítése

Habár a legtöbb felület elrendezése és funkciója eltérő, a felszín alatt hasonlóképpen működnek. Az alkalmazás a *single activity*⁷ struktúrát követi, így minden képernyő egy `Fragment`, melyhez a `RainbowCake` követelményei szerint tartozik saját `ViewState` és `ViewModel`.

Egy `ViewState` itt nézettől függően három vagy négy állapotot tartalmaz. Mindegyik rendelkezik egy *Loading* és egy *Error* állapottal, melyek nevükhöz hűen a töltést, illetve valamilyen hibát jeleznek. Ezen kívül általában egy harmadik állapot jelzi az adatok helyes megérkezését, amikor a töltés befejeződött és megjeleníthető a UI, illetve a részletező nézetek logikája tér el ettől egy kicsit, mert ott a *Viewing* és az *Editing* állapotokkal lehet váltani az olvasási és az írási nézet között.

A `ViewModel`-eknek minden esetben meg kell adni egy kezdő állapotot, ez mindegyik képernyőnél a *Loading*. Emellett rendelkeznek egy *load* függvénnyel, mely a kezdeti, megjelenítéshez szükséges adatokat kéri le, majd a hívás eredménye alapján állítja megfelelő állapotba a UI-t. Itt van nagyon nagy szerepe a létrehozott `Result` osztálynak, mert a visszakapott eredményt típus alapján szét lehet bontani, nem kell belenézni az adatba, hogy érvényes-e. Ahogy az már korábban említésre került, a `ViewModel` végez minden logikát, ami a képernyőhöz kötődik, hogy a `Fragment`-nek tényleg csak a UI megjelenítésével

⁷Évekkel ezelőtt Android fejlesztés során az `Activity` szinonimája volt a képernyőnek, azaz minden megjelenő felülethez egy-egy `Activity` tartozott. Ám később a `Fragment`ek megjelenésével ez megváltozott, és most már a legelterjedtebb módszer a *single activity* alkalmazások fejlesztése, ahol egyetlen `Activity`-ből áll a program, és abban cserélődnek a `Fragment`ek.

kelljen foglalkoznia. Ez szűri a listát kereséskor, elvégzi a rendezést, és függvényeket tartalmaz minden módosítási és létrehozási műveletre, amikkel továbbhív az interactor felé. Általában a kategória vagy jegyzet létrehozásának és törlésének eredményét egy eseményben továbbítja a Fragment-nek, ugyanis ilyenkor navigálni kell másik képernyőre.

A Fragment kezel mindent, ami a látható felhasználói felülettel kapcsolatos, hiszen csak neki van referenciája az elrendezésre és a benne található elemekre. Ez a referencia egy úgynevezett *ViewBinding* funkció segítségével jön létre, melyet a *build.gradle*-ben bekapcsolva minden layout-ot tartalmazó XML fájlhoz generálódik egy Binding osztály, melynek egy példányán keresztül közvetlenül elérhetjük az elemeket. Ezek a layout-ok mind ViewFlipper-t használnak, hogy a lehető legkönnyebb legyen cserélni a megjelenített nézetet (3.2. ábra). A már korábban is szereplő ábrán látott layout gyerekeire a megjelenésük sorrendjének indexével lehet hivatkozni, azaz ebben az esetben a *loadingView* a nullás, a *noteListView* az egyes indexet kapja. Ám az ilyen számkonstansok hosszú távon nem eredményezik a legolvashatóbb kódot, ezért készítettem hozzá egy megfeleltetést, hogy a kódban a nevükkel hivatkozhassem őket (7.10. ábra).

```
object Flipper {  
    val LOADING = 0  
    val VIEWING = 1  
}
```

7.10. ábra. A kezdeti gomb elforgatásának animációja XML-ben.

A ViewModel fentebb említett *load* függvényét a Fragment az *onStart* életciklus-függvényben hívja meg, mely minden alkalommal lefut, amikor a nézet láthatóvá válik. Azért ide tettem a hívást, mert az adatok frissítésére nem csak az első alkalommal van szükség, amikor megjelenik a Fragment, hanem minden módosítás után is, és erre ezt a megoldást láttam a legegyszerűbbnek. Az adatok frissítése után változni fog a képernyő állapota, melyet a *render* függvényben tudunk lekezelni. Erre egy példa már szerepelt korábban az architektúra leírásánál, a 3.3. ábra bemutatásában.

7.8. Kategorizált listanézet

Az alkalmazás elindításakor megjelenő egymásba ágyazott lista a legbonyolultabb képernyő a projektben. Szükség volt hozzá egy RecyclerView-ra, mellyel egy listát jeleníthetünk meg a képernyőn, illetve a Groupie könyvtárra, amit már korábban bemutattam, de most az implementációs részleteibe fogok belemenni.

Ahogy az a működésnél a képernyőképeken látszott (6.1. ábra), a kétféle listanézet között egy BottomNavigationView segítségével válthatunk. Az aktuálisan kiválasztott elem követésére létrehoztam egy SelectedNavItem nevű enumerációt, mely egy *NOTES* és egy *CATEGORIES* értéket tartalmaz. Ezáltal olvashatóan lehet kezelni a váltásokat, mindig egyértelmű lesz, hogy épp melyik képernyő van megjelenítve.

A folyamat a képernyők felépítésénél is leírt *load* függvénnyel kezdődik, melynek itt egyetlen, az előző bekezdésben bemutatott SelectedNavItem típusú paramétere van. Szüksége van egy komplex, egymásba ágyazott listára, melyet az interactor *getComplexList()* függvényének meghívásával kérhet le. Ez a hívás a NetworkDataSource-ban válik ketté, hiszen a FirebaseApi külön adja vissza a kategóriák, illetve a jegyzetek listáját. Miután a DataSource mindkét listát megkapja, összefésüli őket egy listába, mégpedig úgy, hogy abban csak azok az elemek szerepeljenek, amelyeknek nincs szülőkategóriája. Amelyik objektum *parentId*-ja viszont tartalmaz értéket, annak megkeresi ezen id alapján a ka-

tegóriáját, és annak *listItems* nevű adattagjához fűzi. Így pontosan egy olyan struktúrát kapok, mint amelyet meg szeretnék jeleníteni.

Ezt a listát először megkapja a *ViewModel*, majd a *ListLoaded* állapotba csomagolja, és megváltoztatja erre a *viewState*-et. Ezen változás hatására lefut a *Fragment* *render* függvénye, melyben lecserélem a töltőképernyőt a listanézetre, a megkapott adatokat pedig átadom a lista adapterének.

A *Groupie* működéséhez két dolog kell: a listában tárolt elemek megvalósítása, illetve egy *GroupieAdapter*. Az elemek, melyek a *notelist.items* package-ben találhatók, azért szükségesek, hogy a listában össze lehessen kötni az adatot az őt megjelenítő sorral. Két osztály készült ebből a célból, a *CategoryItem* és a *NoteItem*.

A kettő közül a *NoteItem* az egyszerűbb, ez csak a *Groupie* által biztosított *BindableItem* osztályból származik le. Erre a *ViewBinding* használata miatt van szükség, és csak a három alap függvényt kell felüldefiniálni a működéshez: a *getLayout*-ot, melyben specifikáljuk az egy sorhoz használni kívánt elrendezést, az *initializeViewBinding*-ot, mellyel nevének megfelelően a *ViewBinding*-ot állítjuk be, hogy hozzáférjünk az elrendezés elemeihez, illetve a *bind*-ot, melyben összekötjük a kettőt, és beállítjuk a megjelenítendő tartalmat.

A *CategoryItem* valamivel bonyolultabb, hiszen ennél volt szükségem a kibontható és visszacsukható viselkedésre. Ezért ez a *BindableItem*-ből való leszármazáson kívül egy *ExpandableItem* nevű interfészt is megvalósít. Ez annyiban különbözik az alap item-től, hogy kell neki egy *ExpandableGroup* típusú tagváltozó, melyet a *setExpandableGroup* függvényben a paraméterként kapott *onToggleListener*-rel inicializálok. Ez biztosítja a kibontás/összecsukás működését és tárolja az aktuális állapotot, amit az *isExpanded* tagváltozó segítségével lehet lekérni.

A kategóriák sorának elrendezése annyiban különbözik a jegyzetektől, hogy a szövegen kívül egy nyíl ikont is tartalmaz, mely mindig jelzi, hogy az adott csoport éppen ki van-e bontva. Így a *bind* függvény is változik annyiban, hogy a nyíl ikont is be kell állítani, illetve erre az ikonra tettem az *onClickListener*-t, melynek hatására becsukódik/kinyílik a csoport és változik a nyíl iránya.

Ezek után már csak az adapter maradt, mely a *NoteListAdapter* nevet kapta, és a *GroupieAdapter*-ből származik le. Ez tulajdonképpen egy wrapper osztály néhány funkcióval kiegészítve, hogy ne a *Fragment*-nek kelljen ezt a kódot tartalmaznia. Egy publikus függvénye van *showList* néven, mely becsomagolja a lista kiürítését és feltöltését egy műveletbe. Először ugyanis hív az adapteren egy *clear* függvényt - ami fontos, hiszen ha mindig csak hozzáadnánk, akkor folyamatosan sokszorozódna a megjelenített lista -, amit azért tettem bele ide, mert így nem kell külön a *Fragment*-ben mindig meghívni, ezáltal kiküszöböl egy hibalehetőséget. Ezután a lista adapterhez adása előtt még meghívja az adatokon a privát *populateList* függvényt, mely tovább hív egy rekurzív függvénybe, melynek kódja a képen látható (7.11. ábra).

Ahogy látható, az adatok hierarchikus listáján megy végig, és a hierarchiát megtartva létrehoz egy immár *NoteItem*-ekből és *CategoryItem*-ekből álló listát. Erre a feladatra meglepően jól működik a rekurzió, hiszen sokkal könnyebb vele mélységi bejárást csinálni, a kívánt struktúrához ez pedig elengedhetetlen.

Így a listakezelés a *Fragment*-ben már rendkívül egyszerűvé válik, mert a *NoteListAdapter*-t a *RecyclerView* adaptereként beállítva elég csak egy *showList*-et hívni bárhol, ahol meg kell változtatni a megjelenített lista tartalmát.

```

private fun populateList(itemList: List<ListItem>): Section {
    val mainSection = Section()
    for(item in itemList) {
        when (item) {
            is Category -> {
                val group = ExpandableGroup(CategoryItem(item))
                populateCategory(item.listItems, group)
                mainSection.add(group)
            }
            is Note -> mainSection.add(NoteItem(item))
        }
    }
    return mainSection
}

private fun populateCategory(itemList: List<ListItem>, section: ExpandableGroup) {
    for (item in itemList) {
        when (item) {
            is Category -> {
                val group = ExpandableGroup(CategoryItem(item))
                populateCategory(item.listItems, group)
                section.add(group)
            }
            is Note -> section.add(NoteItem(item))
        }
    }
}

```

7.11. ábra. Az adatok átalakítása megjeleníthető formátumba.

7.9. Egyszerű listanézet

Az egyszerű lista azt a nézetet jelenti, ami a *BottomNavigationView* *Notes* elemére kattintva jelenik meg. Ez gyakorlatilag ugyanaz a képernyő, mint a komplex listáé, csupán a *RecyclerView*-ban lecserélődnek az elemek szimpla jegyzetekre. Emellett két funkció válik elérhetővé a listán, ami eddig el volt rejtve: a keresés és a rendezés.

A keresésre kényelmes megoldást ad az *androidx* könyvtár *SearchView* komponense, mely egy előre elkészített, kattintható és gépellhető keresési nézet. Ám a működéséhez először még meg kell valósítani a *SearchView.OnQueryTextListener* interfészt. Ez két felüldefiniálható függvényt tartalmaz, melyekkel a keresés viselkedését lehet finomhangolni.

Az első az *onQueryTextSubmit*, ami akkor hívódik meg, amikor a felhasználó beírta amit szeretett volna, és a billentyűzeten megnyomja a kereséssé alakult enter gombot.

A második pedig az *onQueryTextChange*, mely minden alkalommal lefut, amikor változik a keresőmező tartalma. Én ez utóbbi használata mellett döntöttem, mert manapság a felhasználók nagy része elvárja az azonnali eredményt, amint leütik a billentyűt, és tagadhatatlanul kényelmes, hogyha már gépelés közben visszajelzést kapunk. Itt, amennyiben az új szöveg üres, azaz a felhasználó kitörölte a keresési mező tartalmát, akkor a teljes listát adja az adapternek, egyébként pedig a *ViewModel* végez egy szűrést a begépelte karakter-sor alapján a jegyzetek címére, és csak azokat az elemeket teszi az adapterbe, melyekre illeszkedett a keresés.

A rendezés a UI-on a keresés mellett található, és az ikonjáról első pillantásra felismerhető. Ehhez szintén készítettem egy enumerációt a rendezési módok tárolására *SortOptions* néven, mely egyelőre csak a cím alapján vett rendezést támogatja növekvő és csökkenő betűrendben, de lévén enumeráció, könnyedén bővíthető.

A gomb megnyomására egy dialógusablak ugrik fel, melyben ki lehet választani a kívánt rendezést (6.2. ábra - harmadik kép). Alapértelmezetten növekvő betűrendben található a lista, így ez az opció lesz kiválasztva felhasználói input előtt.

Egy ilyen testre szabott dialógust a beépített *DialogFragment* osztály segítségével lehet létrehozni. A projektben kettő is található, az egyik a kategóriatörlés előtti figyelmeztető *DeleteWarningDialog*, illetve a rendezéskor megjelenő *SortDialog*. Most ez utóbbit fogom bemutatni, de a másik is nagyon hasonló megvalósítás szempontjából.

A dialógus megvalósítását azzal kezdtem, hogy leszármaztattam a *DialogFragment* osztályból, és felüldefiniáltam az *onCreateDialog* függvényét. Itt egy *AlertDialog.Builder* objektum segítségével tudtam felépíteni az általam elképzelt dialógust. Ennek számtalan függvénye van a különböző tulajdonságok beállítására, de ebben az esetben kettőre volt csupán szükségem. Az egyik a *setTitle*, mellyel nevéhez híven a dialógus címét lehet megadni, a másik pedig a *setSingleChoiceItems*. Ez utóbbi az érdekesebb, ugyanis itt lehet a logika legnagyobb részét implementálni.

Három paramétert vár, amiből az első egy stringek tömbje, ami az elemek megjelenítendő nevét tartalmazza. Ehhez létrehoztam egy *sorting.xml* nevű erőforrásfájlt a *res/values* mappában, melyben felvettem egy *sortingOptions* nevű tömböt. Ezt az erőforrást a beépített *getStringArray* függvénnyel lehet átemelni kódba, amivel teljes értékű tömbbé válik, és ugyanúgy lehet használni, mint bármely másik változót. Ennek a módszernek az előnye, hogy nincs beégetett string a forráskódban, ami növeli a karbantarthatóságot.

A második paraméter egy egész szám, mely azt mondja meg, hogy a megjelenéskor hanyadik opció legyen kiválasztva a sorban.

A harmadik pedig egy *DialogInterface.OnClickListener* objektum, melyet esetünkben egy lambda függvénnyel meg tudunk adni (7.12. ábra). Ahhoz, hogy a kiválasztott objektumot kinyerjük, a *selected* indexet kell felhasználni az enumeráció *values*⁸ függvény által visszaadott értékein. Utána eltároljuk a kiválasztott indexet, hogy a következő megnyitáskor is megfelelő elem legyen kijelölve, majd a kinyert objektumot a *parentFragmentManager*⁹-en keresztül beállítjuk a dialógus eredményét. Ezen kívül a végén még kell hívni egy *dismiss*-t, ugyanis alapértelmezett esetben a single choice dialógusok csak gombnyomásra tűnnek el, itt pedig a kívánt működés az, hogy a rendezés kiválasztását ne kelljen gombnyomással megerősíteni.

```
builder.setTitle("Sort by")
    .setSingleChoiceItems(list, selectedItem) { _, selected ->
        val sorting = SortOptions.values()[selected]
        val data = Pair("chosenOption", sorting)
        selectedItem = selected
        parentFragmentManager.setFragmentResult("SortOption", bundleOf(data))
        dismiss()
    }
```

7.12. ábra. A dialógus beállításának kódja.

Ezután már csak annyi a teendő, hogy a *Fragment*ben, amiben kíváncsiak vagyunk a dialógus eredményére, fel kell iratkozni rá. Ezt a *parentFragmentManager*

⁸A *values* függvény az enumeráció konstans értékeinek tömbjét adja vissza a deklaráció sorrendjében.

⁹Ez egy olyan *FragmentManager*, melynek segítségével a jelenlegi *fragment* kommunikálhat a többivel, amennyiben ugyanahhoz az *Activity*-hez tartoznak.

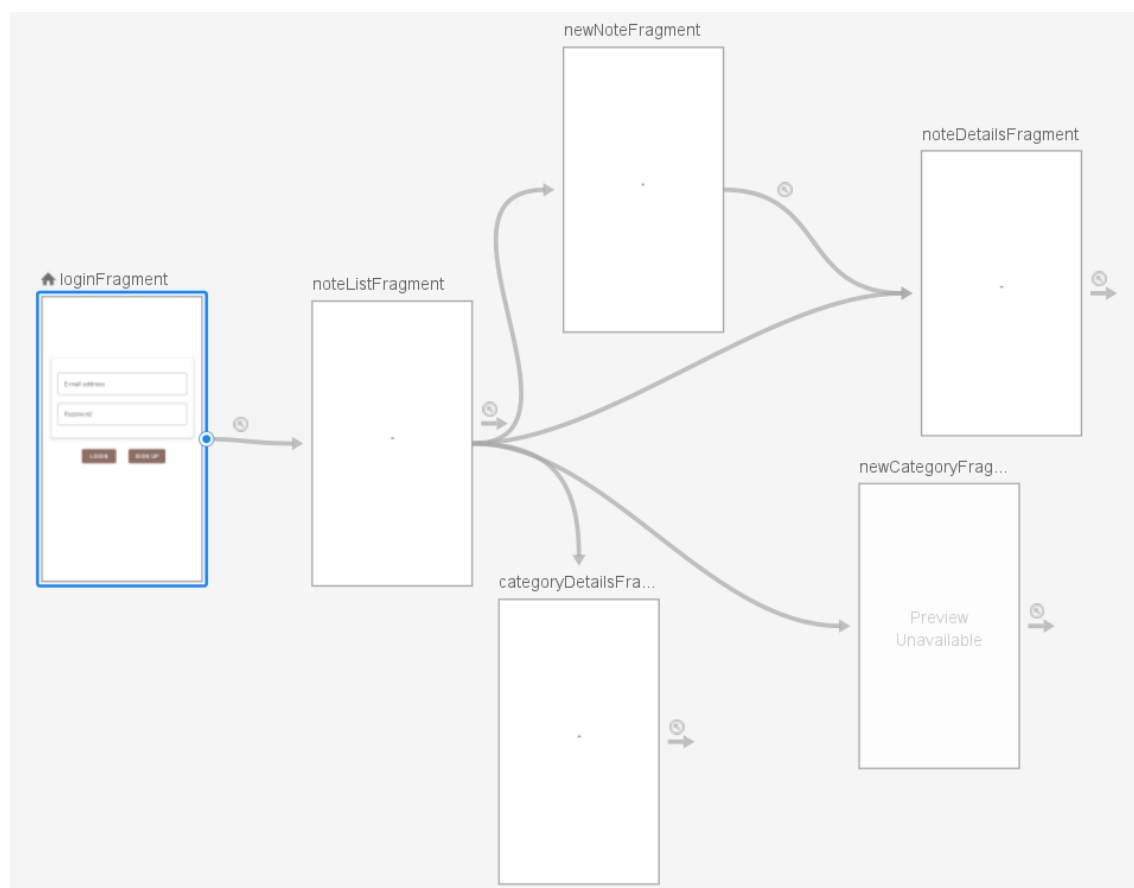
`ger.setFragmentManager` függvénnyel tudjuk megtenni, aminek paraméterként meg kell adni azt a kulcsot, amilyen adatra kíváncsiak vagyunk. Ez a fenti képen a `setFragmentManager` paramétere, melynek a `SortOption` nevet adtam. A kiválasztott rendezési módszer kinyerése után pedig a `ViewModel` által rendezett listát odaadjuk az adapternek megjelenítésre.

7.10. Navigáció

A felhasznált könyvtáraknál már említésre került a `Navigation Component`, de itt most részletesen kifejtem a projektben betöltött szerepét.

A megvalósítást azzal kezdtem, hogy felvettem a `build.gradle` fájlba a könyvtárat függőségként. Ezután a `MainActivity` layout erőforrásába felvettem egy `FragmentManager`-t, és beállítottam, hogy ez töltsse be az alapértelmezett `NavHost` szerepét.

Ezt követően létre kellett hoznom a `res/navigation` mappát, és abba felvenni egy navigációs gráfot. Ehhez tartozik egy kényelmes és intuitív grafikus felület (7.13. ábra), melyen könnyedén össze lehet rakni az appon belüli navigációt, de minden más erőforráshoz hasonlóan itt is fennáll a lehetőség, hogy kézzel XML-t írjunk.



7.13. ábra. Az alkalmazás navigációjának vizuális reprezentációja.

A felület oldalsó sávjaiból lehetőségünk van a kívánt fragmenteket behúzni a felületre, és különböző navigációs útvonalakat definiálni köztük. Ezeket az IDE akcióknak hívja, és sok információt tudnak tartalmazni. Azon kívül, hogy id-t és célt adhatunk egy akciónak, definiálhatunk navigáció közben megjelenítendő animációkat. Befolyásolni tudjuk a `backstack` állapotát, ami például bejelentkezés után rendkívül hasznos, hiszen szeretnénk

elkerülni, hogy a felhasználó bejelentkezve a vissza gomb megnyomásával újra a bejelentkező oldalra kerüljön. Ezt egyszerű megoldani a Navigation Component segítségével, mert szimplán a felület `popBehavior` menüjében beállítjuk, hogy az adott akció elvégzésekor a kívánt fragmenteket levegye a backstackról, így visszafelé navigációkor nem fogjuk elérni.

A másik nagyon hasznos funkció, ami megkönnyíti a fragmentek közti adatátadást, az az `argumentum`. Ezt szintén az akciókhoz tudjuk kötni, és a kiindulási fragmentben paraméterként megadni, így a célfragmentben elérhetővé válik az adat. Ennek kiegészítése a `SafeArgs` plugin, mely típusbiztos navigációt tesz lehetővé. Ez például a részletező nézetknél nagyon hasznos volt, mert a listanézetben a kiválasztott elem `id`-ját át tudtam adni navigáció során a `Details` képernyőnek, annak pedig ezt elég volt átvennie, és a megfelelő `id`-val rendelkező objektumot lekérni az adatbázisból.

Emellett még tartalmaz olyan könnyítést, hogy létre lehet hozni egy "visszatérő" akciót, mely automatikusan leszedi a backstackról saját magát, és úgy navigál vissza az öt megelőző fragmentre. Ez arra is nagyon jó, hogy ne legyen átláthatatlan a gráf, ugyanis ahogy az ábrán is látható, ezeket az akciókat úgy jelöli, hogy a fragmentből kifelé mutatnak, tulajdonképpen a "semmibe".

Ha a gráfon már szerepel minden, amire szükségünk van, akkor az egyetlen hátralévő dolog a kódból való használata. Ehhez először szükségünk lesz a `NavController`-re, amit a fragmentekből a `findNavController` függvényhívással kaphatunk meg. Ezután kelleni fog az elvégezni kívánt akció. A `SafeArgs`-nak köszönhetően minden fragmenthez generálódik egy `Directions` utótaggal ellátott osztály, mely tartalmazza a belőle kiinduló navigációs útvonalakat, olyan néven, amilyen `id`-t adtunk neki a gráfban. Így például a `NoteListFragment` kijelentkezni a `NoteListFragmentDirections.logoutAction` felhasználásával tudunk. Így már nincs más hátra, mint a `NavController` `navigate` függvényének paraméterként odaadni az akciót.

7.11. Tesztelés

Az alkalmazás fejlesztése során kétféle módon is ellenőriztem a helyes működést. Az első a manuális tesztelés, mely során mind emulátorban, mind fizikai készüléken végigmentem a funkciókon.

Fejlesztés közben számomra elengedhetetlen volt a manuális tesztelés, hiszen meglehetősen gyors visszajelzést ad a kódról, hogy megjelenik-e az új funkció, az elvárt módon jelenik-e meg és megfelelően működik-e.

Az egész alkalmazás folyamatán végigmentem, az elejétől kezdve. Csak helyes formátumú e-mail címmel és legalább 6 karakter hosszúságú jelszóval lehet regisztrálni, ha ez nem teljesül, akkor a nem megfelelő tartalmú beviteli mező hibát dob. Nem létező felhasználóval vagy hibás adatokkal szintén nem lehet belépni. A kijelentkezés gomb is az elvárt módon működik, megnyomásával a felhasználó visszakerül a bejelentkezési oldalra.

A kezdőoldalon kezdetben nincs tartalom, először létre kell hozni egy-két jegyzetet és kategóriát. Ehhez a jobb alsó sarok sárga gombja megfelelően működik, kattintásra az elvárt módon kinyílik és becsukódik, az animáció is működik. A felső gombra nyomva a kategória létrehozási képernyője jelenik meg, a szükséges két mezővel. A cím üresen hagyása hibát eredményez, a validáció helyesen működik. A `Spinner`-rel sincs probléma, a kiválasztott elem kerül elmentésre az új kategória szülőjeként. Létrehozás után a kezdeti listanézetre térünk vissza, az az elvártnak megfelelően frissül, és megjelenik benne az új elem.

Kinyitott állapotban a középső gombra kattintva amennyiben nincs az alkalmazásnak engedélye a kamera használatára, akkor feljön egy rövid magyarázat, hogy miért van szükség az engedélyre. Itt az `OK` gombra nyomva feljön a rendszer engedélykérése, melyet

lehetősége van a felhasználónak elfogadni vagy elutasítani. Amennyiben elutasítja, akkor egy Toastot dob az alkalmazás, melyben leírja, hogy el kell fogadni a funkció használatához az engedélyt. Elfogadás után pedig megnyílik egy kamerafelület, ahol fényképet lehet készíteni. Ennek befejeztével rövid várakozás után az új jegyzet létrehozása képernyőre navigálunk, ahol a detektált szöveg megjelenik a tartalom mezőben. Itt lehetőségünk van módosítani rajta, illetve címet adni és szülőkategóriát választani. Itt a cím mellett a tartalomra is érvényes a megkötés, hogy nem lehet üres, addig nem fogjuk tudni elmenteni.

Sikeres mentés után a jegyzet részletező nézetére kerülünk, ahol megfelelően megjelennek az adatai. A jobb felső sarokban a ceruza ikonra kattintva átvált szerkesztő nézetre, eltűnik a ceruza, és megjelenik egy pluszjel és egy mentés ikon. A pluszjelre nyomva újabb fényképet készíthetünk, és az abból visszaérkező szöveg hozzáfűződik az eddigi tartalomhoz. Itt a mentés ugyanúgy működik mint a létrehozásnál. Egy harmadik opció mind megtekintéskor, mind szerkesztéskor elérhető, és ez a törlés. Ennek hatására törlődik a jegyzet és visszavigázunk a listanézetre, melyen frissülés után látható, hogy a kitörölt jegyzet eltűnt.

A törlés a kategóriára is ugyanígy működik, annyi különbséggel, hogy előtte felugrik egy ablak, mely figyelmeztet, hogy ezzel minden gyereket is elveszítjük. Ha itt meggondoljuk magunkat akkor nem történik semmi, de ha megerősítjük, akkor kitörölődik vele együtt az összes benne található jegyzet és kategória. Ilyenkor szintén a listanézetre navigálunk, és láthatjuk, hogy eltűnt.

Még amit itt célszerű leellenőrizni, az az elemek áthelyezése egyik kategóriából a másikba. Ha belemegyünk a szerkesztésbe, és kiválasztunk egy másik szülőkategóriát az objektumnak, akkor mentés után a listában láthatjuk, hogy átkerült az újonnan beállított kategória alá.

Miután a hierarchikus listanézet lehetőségeit kimerítettük, már csak a jegyzetek listájának ellenőrzése maradt. A `BottomNavigationView`-n a `Notes` elemet kiválasztva megerősíthetjük, hogy a lista megváltozik, és csak a jegyzetek kerülnek bele. A nézet tetején pedig megjelenik egy keresősáv és egy rendezés gomb.

A keresősáv aktiválódik ha belekattintunk, és gépelés közben folyamatosan szűri az eredményeket, törlés közben pedig egyre bővül az illeszkedő jegyzetek halmaza. Ha az utolsó betűt is kitöröljük a keresésből, akkor visszatér az eredeti, teljes lista.

Ami a rendezést illeti, a gomb megnyomására az elvártnak megfelelően felugrik az ablak, alapértelmezetten az első opcióval kiválasztva. Ha kiválasztjuk a fordított betűrendet, akkor eltűnik a dialógus és újratöltődik a lista, megfelelően rendezve. Ha olyan opciót választunk, ami már aktív, akkor csak a dialógus tűnik el, de semmi nem történik, és ez így kell legyen, mert felesleges ugyanazt a listát újratölteni.

Ezután pedig következnek a unit tesztek. Ezekhez segítségül hívtam a `JUnit 4` és `Mockito` könyvtárakat, melyek hatalmas segítségnek bizonyultak. Mindenképp szerettem volna legalább unit teszteket írni, mert előtte még sosem írtam, és szerettem volna megtanulni. Első körben a `ViewModelek` tesztelésére koncentráltam, mert az alkalmazásban azok végzik a legtöbb feladatot, főleg az ő munkájuk a lényeges. Nem készült még mindegyikhez teszt, és nem is teljes a lefedettség, de a célokat elértem: írtam unit teszteket és értem őket.

A `test/ui` mappába raktam őket, hiszen UI közeli logikáról van szó. A `ViewModelek`-hez csináltam külön-külön tesztosztályokat, melyeken belül függvények ellenőrzik egy-egy funkció helyes működését. Ezek az osztályok a `RainbowCake` által biztosított `ViewModelTest` osztályból származnak le, melynek segítségével könnyedén meg lehet figyelni a teszt során előforduló állapotokat és eseményeket.

Mivel a `ViewModelek` az `Interactort` hívják az adatok megszerzéséhez, ezért ezt minden ilyen osztályban mockolni kellett. Ebben segített a `Mockito`, mellyel a `@Mock` annotációval megjelölt objektumoknak meg lehet szabni a viselkedését. Ehhez minden osztályban

csináltam egy *companion object*-et, mely tartalmazza az elvárt adatokat. Így a következőképp néz ki egy teszteset (7.14. ábra).

```
@Test
fun loadViewModel_withComplexList() = runBlocking { this: CoroutineScope
    `when` (mockInteractor.getComplexList()).thenReturn(Result.success(expectedComplexList))
    `when` (mockInteractor.getNotes()).thenReturn(Result.success(expectedNoteList))

    noteListViewModel.observeStateAndEvents { stateObserver, eventsObserver ->
        noteListViewModel.load(SelectedNavItem.CATEGORIES)
        stateObserver.assertObservedLast(ListLoaded(expectedComplexList))
    }
}
```

7.14. ábra. A komplex lista unit tesztje.

Ebből a lényeg szépen megfigyelhető: a *when - thenReturn* függvénypárossal lehet megszabni a mock objektumok viselkedését. Ebben a tesztesetben sikeres esetet szeretnék tesztelni, úgyhogy mindkét interactor hívás sikerrel tér vissza. Ezután elkezdem a megfigyelést, majd elvégzem a tesztelni kívánt függvényhívást. Az adott observerek segítségével pedig validálom az elvárt állapotot. Ehhez azért használom az *assertObservedLast* függvényt, mert a sima *assertObserved* a futás során összes előforduló állapotot rögzíti, és nekem itt az a lényeg, hogy a hívás végére az elvárt eredményt kapjam, amihez elég a legutolsó megfigyelt állapot.

8. fejezet

Összefoglalás

8.1. Tapasztalatok

8.2. Értékelés

Az alkalmazás követelményei résznél 10 pontba szedtem az applikációval szemben támasztott elvárásokat, így most arra szeretnék kitérni pontonként, hogy ezeket mennyire sikerült megvalósítani.

1. A felhasználói fiók létrehozására és bejelentkezésre létrejött egy képernyő, ahol egy e-mail címet és egy jelszót kell megadni. Magát az autentikációt a Firebase Authentication végzi. Belépés után a NoteList képernyő jobb felső sarkában található Logout gomb nyújt lehetőséget a kijelentkezésre.
2. Az adatok tárolását a Firebase Firestore végzi, és mivel ez egy felhőalapú szolgáltatás, így a felhasználó tetszőleges készüléken bejelentkezve eléri a saját adatait.
3. A bejelentkezési oldalról addig nem lehet tovább navigálni, amíg nem lett sikeres az autentikáció. A saját adatok elérése pedig az adattárolás struktúrájából következik, ugyanis az adatbázisban minden felhasználó adata a saját azonosítójával elnevezett dokumentumban található, futás során pedig az auth modul csak a jelenleg bejelentkezett felhasználó id-ját ismeri, másét nem.
- 4.
- 5.
- 6.
- 7.
- 8.
- 9.
- 10.

Mindezt figyelembe véve én elégedett vagyok az elkészült alkalmazással, mert bár még elég alap funkcionalitással rendelkezik, de úgy gondolom, hogy a lehető legjobban figyeltem arra, hogy könnyen bővíthetővé tegyem. Van is nem kevés ötletem a potenciális fejlesztésekre, melyből néhányat a következő szekcióban ismertetek.

8.3. Továbbfejlesztési lehetőségek

Természetesen az alkalmazás abszolút nem tökéletes, van még rengeteg opció a fejlődésre. Ezért úgy gondoltam, hogy lezárásképp megemlítek pár dolgot, amin lehetne javítani mind kódminőség, mind felhasználás szempontjából.

Irodalomjegyzék

- [1] S. O’Dea. *Mobile operating systems’ market share worldwide from January 2012 to June 2021*. <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/>, 2021, [2021-11-21].
- [2] Péter Ekler. *Mobil- és webes szoftverek előadásanyag*, 2020, [2021-11-20].
- [3] Android Developers. *Platform Architecture*. <https://developer.android.com/guide/platform/index.html>, 2021, [2021-11-21].
- [4] Android Developers. *Meet Android Studio*. <https://developer.android.com/studio/intro>.
- [5] Android Developers. *Projects Overview*. <https://developer.android.com/studio/projects>.
- [6] Android Developers. *App Manifest Overview*. <https://developer.android.com/guide/topics/manifest/manifest-intro>.
- [7] Márton Braun. *RainbowCake - A modern Android architecture framework*. <https://rainbowcake.dev/>.
- [8] Android Developers. *Dependency injection in Android*. <https://developer.android.com/training/dependency-injection>.
- [9] Márton Braun. *Designing and Working with Single View States on Android*. <https://zsmb.co/designing-and-working-with-single-view-states-on-android/>, 2020, [2021-11-22].
- [10] Márton Braun. *ViewFlipper documentation*. <https://rainbowcake.dev/best-practices/view-flippers/>, [2021-11-23].
- [11] Márton Braun. *Events documentation*. <https://rainbowcake.dev/features/events/>, [2021-11-23].
- [12] Márton Braun. *Testing documentation*. <https://rainbowcake.dev/features/testing/>, [2021-12-02].
- [13] Google. *Vision AI*. <https://cloud.google.com/vision>, [2021-11-26].
- [14] Google. *Firebase documentation*. <https://firebase.google.com/docs/firestore/rtdb-vs-firestore>, [2021-11-27].
- [15] Google. *Cloud Firestore - Data model*. <https://firebase.google.com/docs/firestore/data-model>, [2021-11-27].

- [16] Google. *Firebase Authentication*. <https://firebase.google.com/docs/auth>, [2021-11-27].
- [17] Google. *Google Analytics*. <https://firebase.google.com/docs/analytics>, [2021-11-27].
- [18] Google. *Firebase Crashlytics*. <https://firebase.google.com/docs/crashlytics>, [2021-11-28].
- [19] Android Developers. *Create dynamic lists with RecyclerView*. <https://developer.android.com/guide/topics/ui/layout/recyclerview>, [2021-11-28].
- [20] Lisa Wray. *Groupie*. <https://github.com/lisawray/groupie>, 2017, [2021-11-29].
- [21] Android Developers. *Permissions on Android*. <https://developer.android.com/guide/topics/permissions/overview>, [2021-11-28].
- [22] Android Developers. *Navigation*. <https://developer.android.com/guide/navigation/>, [2021-12-07].
- [23] Adam Carpenter. *What is Integration Testing?* <https://www.codecademy.com/resources/blog/what-is-integration-testing/>, 2021, [2021-12-02].
- [24] Stan Georgian. *What is End-to-End Testing and When Should You Use It?* <https://www.freecodecamp.org/news/end-to-end-testing-tutorial/>, 2021, [2021-12-02].
- [25] Carlos Schults. *UI Testing: A Beginner's Guide With Checklist and Examples*. <https://www.testim.io/blog/ui-testing-beginners-guide/>, 2020, [2021-12-03].
- [26] JavaTPoint. *JUnit Tutorial | Testing Framework for Java*. <https://www.javatpoint.com/junit-tutorial>, [2021-12-03].