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. november 28.

Tartalomjegyzék

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. november 28.

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 az alkalmazással.

A célom az volt, hogy egy hétköznapokban 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 karóráktól 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, akár az operációs rendszert is.

Ez a változatosság és szabadság fejlesztői szempontból kettős. Egyrészről gyakorlatilag lehetséges bármilyen alkalmazást írni, és a nyílt forráskód miatt bármikor bele lehet nézni a platform kódjába, hogy megértsük a viselkedését, másrészről 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

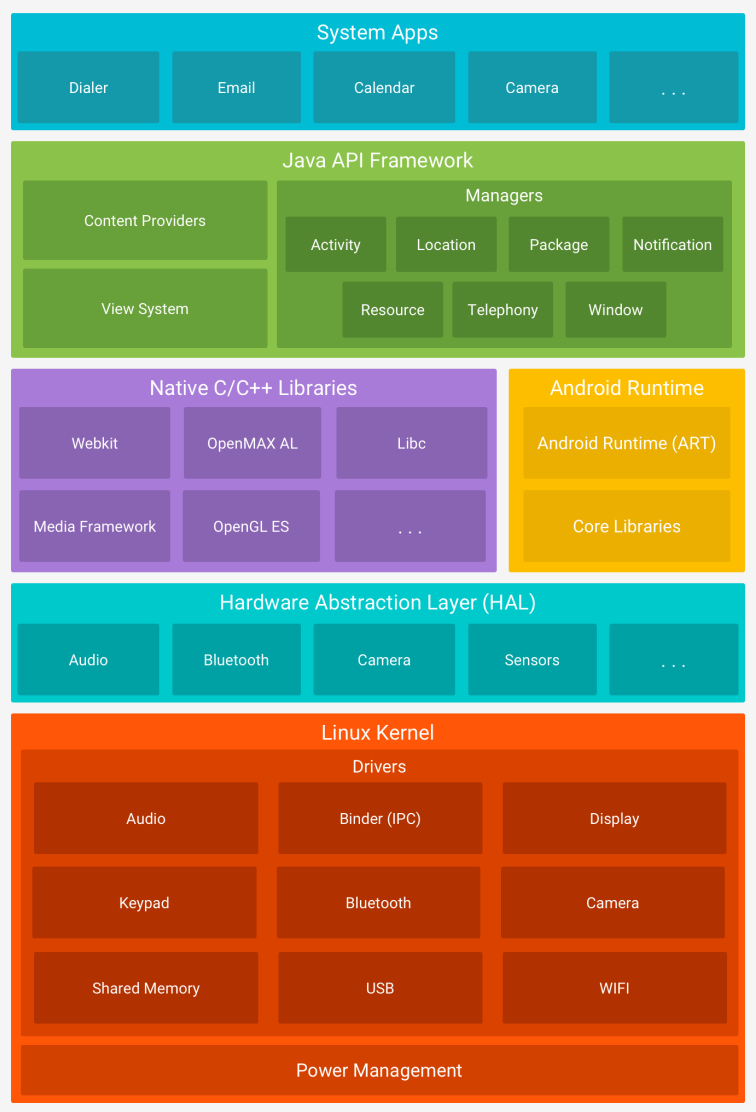
A leírás kiegészítéseként a platform felépítését a 2.1. ábra szemlélteti.

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]



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 szemétgyűjtés vagy a jobb hibakeresési támogatás.

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.

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.

2.1.6. Rendszeralkalmazások

A rendszer beépített alkalmazásokkal érkezik a leggyakoribb feladatok elvégzésére. Ilyen például az SMS-küldés, internet böngészés, telefonálás, naptár és így tovább. Ezek, ahogy a fenti bevezetőben is említettem, néhány kivételével 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.

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 az emulátor, amivel virtuális készülékeken tesztelhetünk. 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 Managerben 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 számomra 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. Ami viszont számomra a leghasznosabb funkció, az a nem használt kódrészletek jelzése. Mivel az IDE kiszürkíti 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 funkcionalitási 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 csoportra 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 csoport *java* névre hallgat, és ebben található a projekt összes forráskódja. A nevével ellentétben ma már legtöbbször Kotlin kódot tartalmaz, mivel egy-két éve már ez az Android fejlesztés hivatalos nyelve, de az elnevezés megmaradt azokból az időkből, amikor még mindenki Java-t használt.

Az utolsó csoport 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.

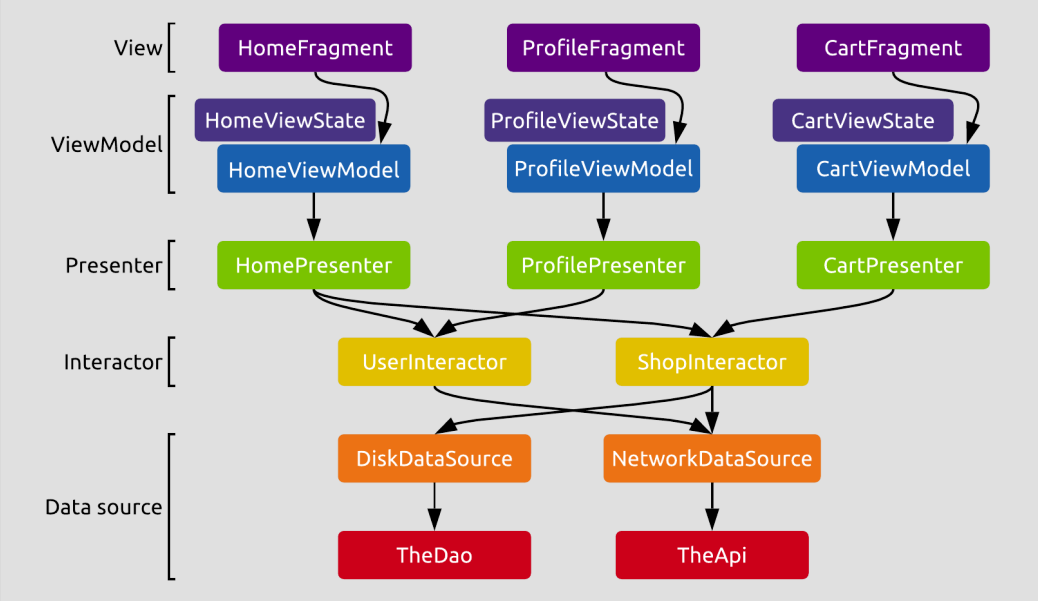
3. fejezet

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, aztán amikor erről beszélgettem a konzulensemmel, akkor ő a RainbowCake-et[7] ajánlotta. Egy MVVM-re építő architektúráról van szó, 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 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 kifejtem 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 referenciájára. 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 másikból, 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 kezelhetetlenné 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 a megfelelő függőségeket kell felvenni a projektbe. Ezt követően még pár beállítást el kell végeznünk a projekten, majd az injektálni kívánt konstruktorokat *@Inject* annotációval megjelölni, a többit pedig elvégzi helyettünk.

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 hibalehetőséggel. Ezt orvosolja a ViewFlipper, melyben egyszerre egy tartalmazott layoutot tudunk megjeleníteni, így nem kell manuálisan állítani a külön elemek láthatóságát. 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).

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.

```xml
<?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.

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. [12]

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. 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 tökéletesen 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 mást.

Alább találhatók 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. [13] 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 találhatók. [14] 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.

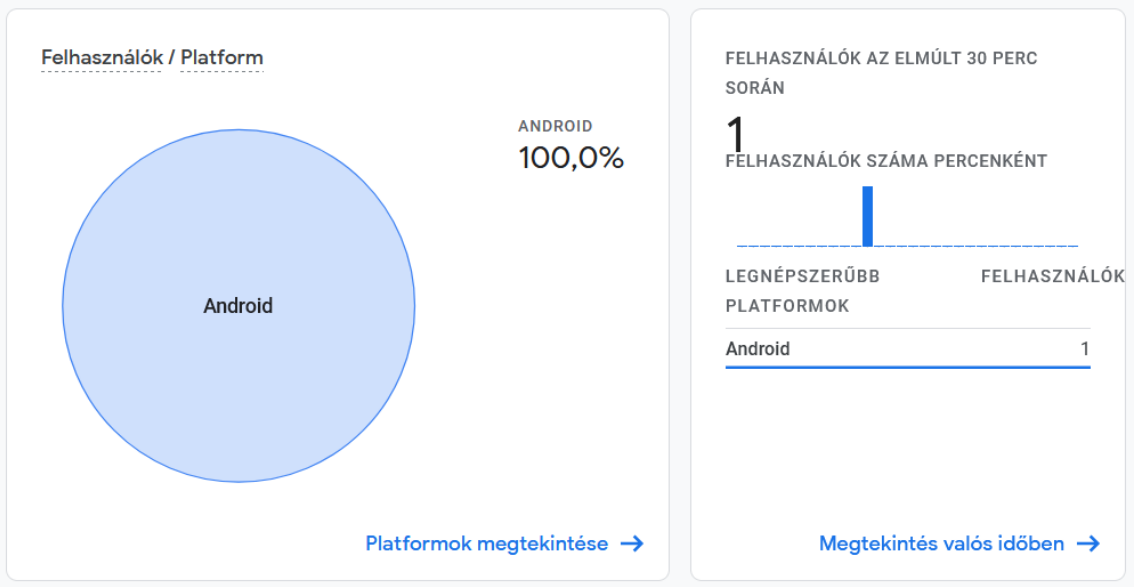
A fenti szabály minden dokumentumra illeszkedik, és megenged minden írást és minden olvasást abban az esetben, hogyha a lekérdezés 2022. január 31. előtt érkezik be. Ez egy tesztszabály, melyet a Firestore generál a projekt létrehozásakor a fejlesztés megkönnyítése céljából. Természetesen ezt éles alkalmazásban mindenképpen le kell cserélni, hiszen így bárki hozzáférhet az adatbázishoz, aki tudja a projektazonosítót, ám a tesztelési fázisban még bőven elegendő.

4.2.2. Autentikáció

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. [15]

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 console[1]-ban az alkalmazás adatai alatt. [16] 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.



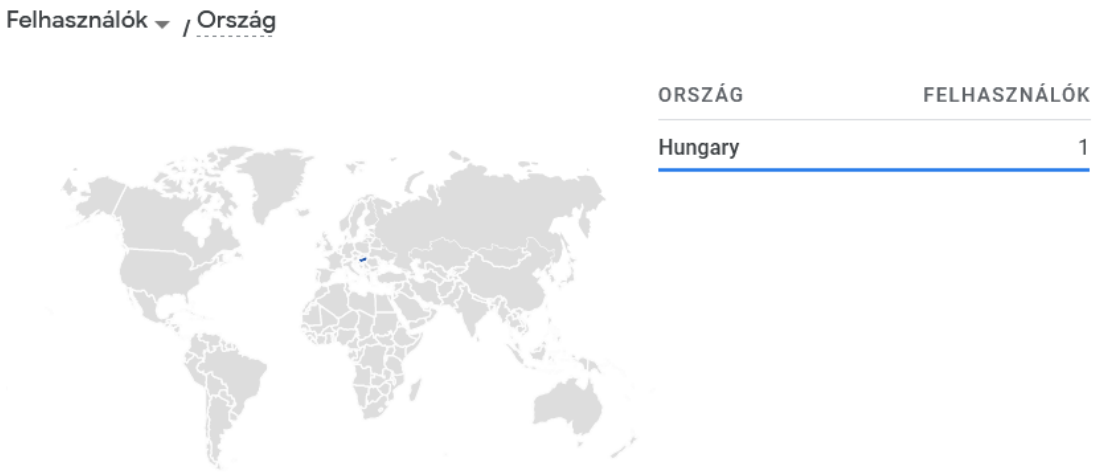
4.2. ábra. Platform megoszlása a felhasználók között, közelmúlt aktivitása.

Ezen statisztikák nyilvánvalóan nem annyira látványosak ilyen kis mértékű felhasználással, de egy nagyobb felhasználóbázisnál már jelentősen segíthetik az alkalmazás fejlődését. Mindez nem csak a fejlesztőknek lényeges, hanem az üzleti résztvevőknek is, hiszen az Analytics számos statisztikát készít a bevételekről és az ügyfélszerzésről 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

[1] `https://console.firebase.google.com`



4.3. ábra. Felhasználók megoszlása az országok között.

vezető körülményeket. Valósidejű figyelmeztetést küld az újonnan felbukkanó vagy éppen növekvő problémákról, melyek azonnali figyelmet igényelhetnek. [17]

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.

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. [18]

Azonban a projekt során szükségem volt egy funkcióra, amit a RecyclerView alapértelmezetten nem tud, ez pedig a lista elemeinek kinyitható/becsukható 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. Használata rendkívül egyszerű: mindenhol, ahol eddig *RecyclerView.Adapter*-t kellett használni ezután *GroupieAdapter* szerepel majd.

4.4. EasyPermissions

4.5. Material Components

5. fejezet

Implementáció

6. fejezet

Összefoglalás

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] Google. *Vision AI.* https://cloud.google.com/vision, [2021-11-26].

[13] Google. *Firebase documentation.* https://firebase.google.com/docs/firestore/rtdb-vs-firestore, [2021-11-27].

[14] Google. *Cloud Firestore - Data model.* https://firebase.google.com/docs/firestore/data-model, [2021-11-27].

[15] Google. *Firebase Authentication.* https://firebase.google.com/docs/auth, [2021-11-27].

[16] Google. *Google Analytics*. https://firebase.google.com/docs/analytics, [2021-11-27].

[17] Google. *Firebase Crashlytics*. https://firebase.google.com/docs/crashlytics, [2021-11-28].

[18] Android Developers. *Create dynamic lists with RecyclerView*. https://developer.android.com/guide/topics/ui/layout/recyclerview, [2021-11-28].