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

Nagy Gergely

NÖVÉNYGONDOZÁST TÁMOGATÓ ALKALMAZÁS FEJLESZTÉSE ANDROID PLATFORMRA

KONZULENS

Gazdi László

BUDAPEST, 2024

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2024. 12. 06.

...…….
Nagy Gergely

Összefoglaló

A mai modernizált, városiasodott világban egyre kevesebb figyelmet fordítunk környezetünkre, beleértve a növényeket is. A sűrű mindennapok rohanásában hajlamosak vagyunk elhanyagolni őket, holott tisztában vagyunk létfontosságú szerepükkel életünkben. Ennek ellenére sem fordítunk időt és energiát akár új, akár meglévő növényeink gondozásának elsajátítására. Pedig ma már a technológia fejlődése rengeteg lehetőséget nyújt, hogy kényelmesen, időtakarékosan szerezzük meg a szükséges információkat.

Szakdolgozatomban erre a problémára készítettem egy növénygondozást támogató alkalmazást, amely segít a felhasználónak a növényeivel és egy helyre összegyűjti a fontos tudnivalókat róluk. Lehetőséget kínál saját növények elmentésére, melyek kategorizálhatók (szoba vagy kerti növény), illetve megadható a locsolás rendszeressége. Ezt felhasználva az alkalmazás értesítést küld a növények öntözésének esedékességéről. Továbbá a felhasználó böngészhet interneten elérhető növények között, melyekről számára releváns, részletes információkat kap. Még tovább növelve a kényelmet és az információk elérhetőségét, az applikációban helyet kapott mesterséges intelligencia is, amely mind az internetes, mind a saját növényekről szolgáltat tippeket, javaslatokat.

A dolgozat bemutatja az alkalmazás tervezésének és fejlesztésének folyamatát, beleértve a felhasznált technológiákat és az alkalmazásban használt tervezési minták részletes leírását, melyek meghatározásánál a könnyű bővíthetőség volt a fő szempont.

Abstract

In today's modernized, urbanized world, we pay less and less attention to our environment, including plants. In the rush of our busy daily lives, we tend to neglect them, even though we are aware of their vital role in our lives. Nevertheless we fail to dedicate the necessary time and energy to learning how to care for new or our own plants. However, with advancements in technology, there are now numerous opportunities to acquire the necessary information conveniently and efficiently.

In my thesis, I have developed a plant care support application to address this issue, helping users manage their plants and gather essential information about them in one place. The app allows users to save their own plants, which can be categorized (indoor or outdoor), and specify the watering frequency. Based on this, the application sends notifications when it’s time to water the plants. Additionally, users can browse plants available on the internet and access detailed, relevant information about them. To further enhance convenience and accessibility, the application also includes artificial intelligence, which offers tips and suggestions for both online and personal plants.

The thesis describes the design and development process of the application, including the technologies used and a detailed description of the design patterns used in the application, focusing on ensuring ease of extensibility.

1 Bevezetés

A növények gondozása kihívás lehet főleg, ha új hobbiként vágunk bele és nem rendelkezünk a megfelelő ismeretekkel. Maga a növény tartás is, a gondozást leszámítva egyre nehezebbé válik. A nagy városokban sok ember él társasházakban udvar nélkül, de egy kertes háznak sem feltétlen van akkora udvara, ahol elférnének a virágok kényelmesen. Így sokan már ott megállnak, hogy egyáltalán vegyenek növényt. Ha pedig esetleg eljut az ember a vásárlásig, rengeteg szemponttal kell számolnia, ami egyszerre megterhelő lehet. Utána kell néznie, hogy milyen növény lenne jó számára (a lakásba vagy kertbe), annak mik a jellemzői, mire kell odafigyelni. Milyen igényei vannak: napfény, víz, táptalaj. El fog-e férni, nem lesz-e túl nagy vagy esetleg túl kicsi. Ennek mind utána kell nézni és számításba kell venni, ami nem egyszerű feladat.

Az interneten persze számos adat áll rendelkezésre, amiből tanulhatunk. Nagy hátránya ennek, hogy sok időt vesz igénybe. Meg kell keressük magunknak az oldalakat, ahol elérhetőek az ismertető anyagok és végig is kell olvasnunk a legtöbbet. Mivel a rengeteg információ nem csak előny, de hátrány is egyben, hamar ellentmondásba ütközhetünk, ha egy adott növényről az egyik ismertető mást állít, mint a másik. A források ellenőrzéséről nem is beszélve.

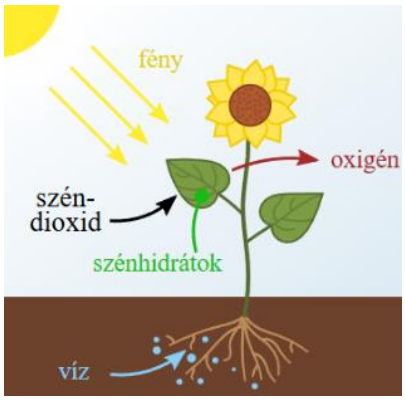
Így a növénygondozás nem egy egyszerű feladat, annak ellenére, hogy lehetne kevésbé bonyolult. A vele járó ismeretszerzés, tanulás, az elmélet elsajátítása a jelen formájában sok munkát igényel.

1.1 Motiváció

Személy szerint én nagyon szeretek a növényekkel foglalkozni, azonban mivel nekem ez csak hobbi, a tudásom meglehetősen kezdetleges. Hamar belefáradok, ha egy virághoz túl sokat kell keresgélni, böngészni az internetet, hogy kiderítsem hogyan kell vele bánni. Ennek megoldására egy telefonos alkalmazás megvalósítása jutott eszembe, ami összegyűjti egy helyre a releváns adatokat és könnyen elérhetővé teszi azokat. A mobiltelefon mára már szerves részévé vált az életünknek, szinte mindenhová magunkkal visszük. Így kézenfekvő megoldásnak tűnt ezt a platformot választani.

1.1.1 Növények fontossága

A növények jelenléte otthonainkban és a szabadban túlmutatnak az esztétikai értékeken. Szerepük létfontosságú a földi élet fennmaradásában. Ők építik fel vízből, ásványi sókból a Napból érkező fényenergia segítségével azokat a szerves anyagokat, amelyek lehetővé teszik az állati, beleértve az emberi életet is [1]. Ez a folyamat a fotoszintézis, mely során a növény megköti a levegőben lévő szén-dioxidot, javítva így annak minőségét is.



1. ábra Fotoszintézis folyamata

Nem csupán a fotoszintézissel járulnak hozzá életünkhöz. Nagy hatásuk van a klímára is. Például a fák nagyon hatékonyan hűtik a környezetet. A Yale Egyetem tanulmánya [2] szerint egy kifejlett fa nagyjából két lakossági klímaberendezésnek megfelelő hűtő hatást produkál, miközben a klímával ellentétben nem melegíti a környezetét. Mindemellett pozitív hatással vannak a közérzetünkre. A növényápolás kiváló stresszoldó, segít mentális egészségünk megőrzésében [3]. Továbbá számos élőlénynek szolgálnak táplálékul, gazdagítva a mi ételeinket is. Valamint a gyógyító hatású vegyületeket tartalmazó növények millióknak hoznak gyógyulást évről évre. Ezért nagyon fontos, hogy megőrizzük nevelésükhöz szükséges ismereteket és népszerűsítsük fontosságukat.

1.1.2 Okostelefon és mesterséges intelligencia életünkben

Az okostelefon-használat az utóbbi években egyre nagyobb teret foglal el életünkben. Nemcsak az utcákon, metrón, tömegközlekedésben, de az iskolákban és az éttermek környékén is mindenhol okostelefont böngésző gyermekeket, fiatalokat, felnőtteket látni. Mindennapi tevékenységeink sorában kiemelt fontosságot kap a telefonon elérhető, életünket megkönnyítő funkciók és alkalmazások letöltése, a

hírfolyamok ellenőrzése és rendszeres frissítése [4]. Ma már gyakorlatilag sehova nem indulunk el telefonunk nélkül, hiszen életünk jelentős részét rajta tároljuk. Minden tevékenységünket megelőzi egy „mobilozás” valamilyen formában. Ha el akarunk jutni valahova, előtte megnézzük telefonunkon merre induljunk. Ha be szeretnénk ülni egy étterembe, előtte megnézzük annak értékeléseit. Boltban a segítségével fizetünk. Rajta keresztül tartjuk a kapcsolatot ismerőseinkkel. Hosszasan lehetne még sorolni, de ezekből is jól látható, hogy mennyire életünk részévé vált ez a technológia.

A mesterséges intelligencia már eddig is jelen volt és segítette életünket. Közlekedésben önjáró járművek, valamint beszédfelismerés, hangalapú asszisztensek, pénzügyi területeken algoritmikus kereskedés stb. formájában. Azonban 2023-ban a nagy nyelvi modellek[1] berobbanásával a mindennapjainkba, a mesterséges intelligencia is kezd hasonló szintre jutni életünkben, mint a mobil. Az átlagember egyre több mindenre használja, legyen az munka vagy akár egyszerű internetes keresés és a kezdeti szkepticizmus a helyességével szemben minél inkább kezd eltűnni.

1.1.3 Konklúzió

Látva a mesterséges intelligencia okozta új trendet és figyelembe véve az okostelefonok szerepét, kézenfekvő volt számomra, hogy a kettőt ötvözve egy olyan megoldás hozható létre a problémára, amely követi a technológia fejlődését és könnyen elérhető a legtöbb ember számára.

1.2 Dolgozat felépítése

A dolgozat az alábbiak szerint épül fel. A második fejezetben kifejtem a feladat specifikálását, részletezem a megvalósítandó alkalmazást. A harmadik fejezetben felsorolom a felhasznált technológiákat, illetve röviden bemutatom őket. A negyedig fejezet a tervezésről szól, ideértve a felhasználó felület tervezését és a használandó architektúrális és tervezési minták leírását. Az ötödik fejezetben a fejlesztés során előforduló érdekesebb megoldásokat mutatok be, valamint kitérek a tesztelésre. A hatodik fejezetben összefoglalom az elvégzett munkát és kitérek a továbbfejlesztési lehetőségekre is.

[1] A nagy nyelvi modellek (LLM-ek) fejlett mesterséges intelligencia-rendszerek, amelyeket emberi szöveg feldolgozására, megértésére és előállítására terveztek [5]

2 A feladat részletes specifikálása

2.1 Feladatkiírás értelmezése

A feladatom egy Android alkalmazás megtervezése és fejlesztése. Először magát a platformot szükséges megismernem, annak működését a fejlesztéssel kapcsolatban. Jelenleg Android alkalmazásfejlesztésben két fő felhasználói felület (UI – User Interface) készítési megközelítés közül lehet választani. Az első az Extensible Markup Language – XML alapú lehetőség, a második pedig a Jetpack Compose által nyújtott modernebb megoldás. Mindkét opciót tanulmányozom és mérlegelnem kell az egyes esetekben az előnyöket és hátrányokat, majd ezalapján kiválasztanom melyikkel fogok dolgozni.

A tényleges tervezés előtt érdemes megvizsgálni a piacon elérhető hasonló alkalmazásokat, ezek felépítését, funkcióit. Továbbá a felhasználók által írt visszajelzések is hasznosnak bizonyulhatnak, hiszen ezekből látható, melyek a valóban hasznos funkciók, mire lenne még igény, illetve mit kell mindenképpen elkerülni.

Ezek után, a piackutatásból szerzett ismeretekkel kiegészülve megtervezem magát az alkalmazást. Ez magába foglalja a kliens-szerver architektúra, az alkalmazás belső struktúrája, illetve a felhasználói felület tervezését. A folyamat során érdemes a már jól bevált tervezési mintákra (design pattern) építeni. Ezek nagy mértékben növelik a karbantarthatóságot és a bővíthetőséget is.

A következő lépésben az alkalmazás fejlesztésére kerül sor, a követelményeknek megfelelően. A felhasználóknak lehetőségük kell legyen elmenteni saját növényeket és beállítani az öntözések időpontját. Az appnak értesítést kell küldenie az éppen meglocsolandó virágokról. Emellett további növények közti böngészés lehetőségét is biztosítani kell a felhasználóknak, róluk részletes adatokat szolgáltatva. A mesterséges intelligencia által kapott információkkal ki kell egészíteni a részletes adatokat, mind a saját, mind az egyéb növények esetében.

Végül különböző teszteket kell készítsek az alkalmazás helyes működésének ellenőrzésére. Unit tesztekkel az üzleti logikát, UI tesztekkel pedig a felhasználói felületet ellenőrzöm.

2.2 Funkcionális követelmények

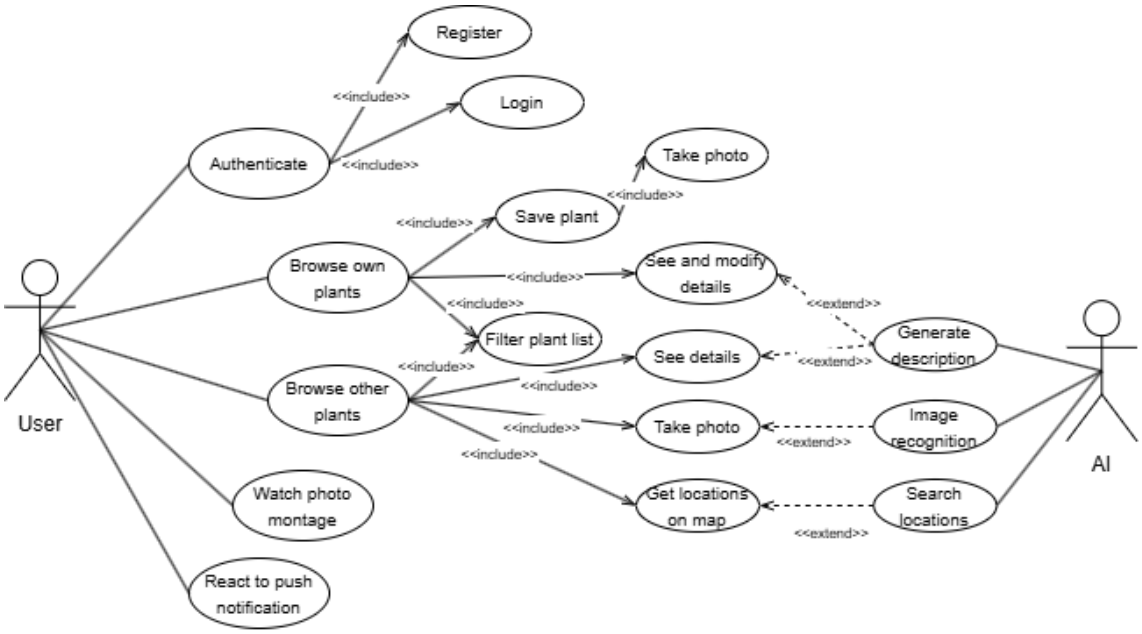
Az alkalmazásnak képesnek kell lennie valamilyen formában hitelesíteni a felhasználókat, ezáltal megkülönböztetve őket, így lehetővé téve a több felhasználó általi használatát.

Továbbá el kell tudnia mentenie a felhasználók növényeit egy listába. A mentés csak a szükséges adatok megadásával történhet meg: név, ciklus, öntözés gyakorisága, utolsó locsolás időpontja, beltéri/kültéri. A lista szűrésére különböző lehetőségeket kell biztosítson az app: legyen szöveg alapján szűrhető (például név szerinti keresés) és legyen egy kültér/beltér szűrő is. Emellett a felhasználónak tudnia kell módosítania a már elmentett növényeinek adatait, illetve a részletei közé a mesterséges intelligencia által adott leírást is fel kell venni.

A felhasználók által elmentett növényeken felül egyéb növények közötti böngészést is biztosítania kell az alkalmazásnak. Az előbbiekben leírt szűrési funkciókból csak a szöveg alapú keresésnek kell elérhetőnek lennie ennél a listánál. Ezen növények részleteinél nincs lehetőség szerkesztésre, hanem csak fontosabb adatok és a mesterséges intelligencia tippjei szerepelnek. Valamint meg kell tudnia határozni az app-nak, hogy az országban hol kapható az adott növény.

Ezenfelül az alkalmazásnak biztosítania kell a felhasználóknak a telefon kamerájával való képek készítését. Ezzel a funkcióval a saját növényeknél készíthető kép, mely mentésre kerül a növényhez, illetve az egyéb növényeknél kép készítésével lehet keresni, ami alapján az app megmondja milyen növény látható a rajta. Továbbá, ha készültek képek a felhasználó saját növényeihez, akkor összeállítást kell csináljon ezekből, melyek akárhányszor megnézhetők.

Végül de nem utolsó sorban push értesítéseket is fogadnia kell az app-nak, melyekben a felhasználók értesülhetnek a növényeik öntözésének esedékességéről.



2. ábra Use-case diagram

2.3 Piackutatás

A jelenlegi piacot átnézve azt tapasztaltam, hogy ebben a témakörben a legtöbb alkalmazás képalapú növényazonosításra szolgál. Kifejezetten arra a célra, melyre a feladatom összpontosul, viszonylag kevés app áll rendelkezésre. Ez számomra rendkívül meglepő, hiszen maga a piac és az Android felhasználók száma is meglehetősen nagy. Így kibővítettem a kutatási területet és megvizsgáltam az iOS platformra elérhető hasonló alkalmazásokat is. Ugyanaz fogadott, mint az előzőekben, annyi különbséggel, hogy a növénygondozásra is szolgáló alkalmazások száma még inkább elenyésző. A következőkben bemutatok 3 alkalmazást, amely a kevés közül is a leginkább felkapott és közel áll a feladatom céljához.

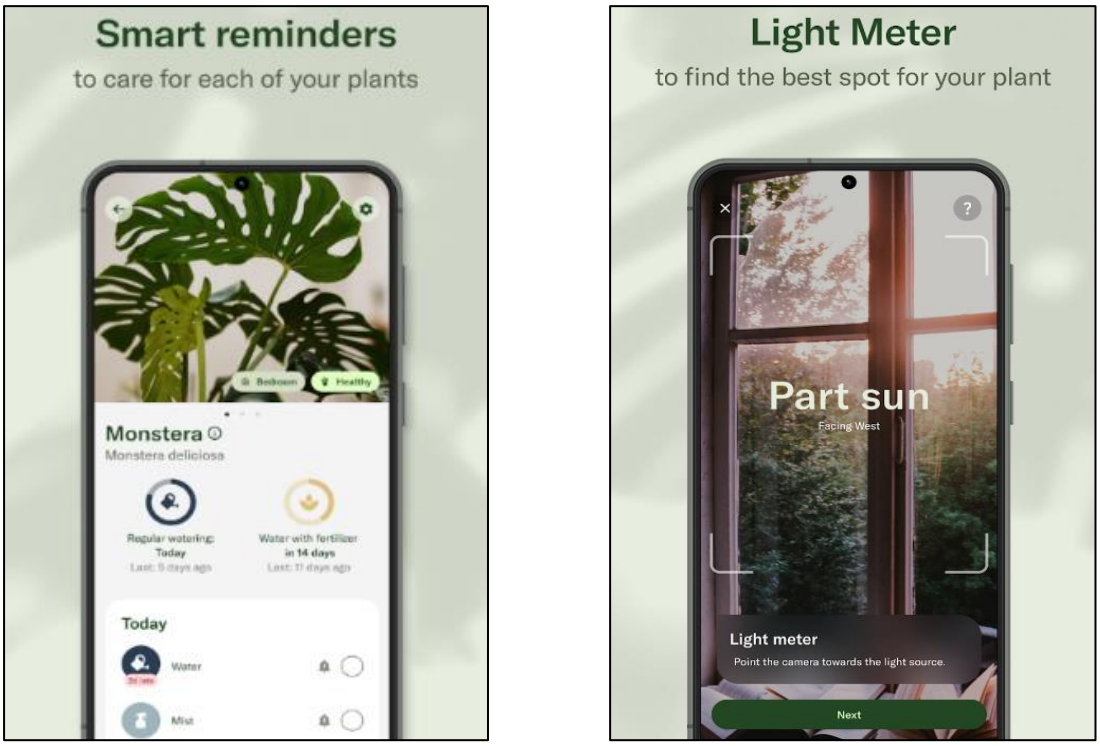
2.3.1 Planta[2]

Az alkalmazás induláskor elkér a felhasználótól információkat, mint például helyileg milyen növényei vannak (kerti, beltéri), mennyire ért a növényekhez, mennyire érdeklik a növények. Értesítést küld a növények locsolásának esedékességéről, mellé pedig figyelembe veszi az időjárást is. Az öntözés mellett a tápanyag pótlásról is figyelmeztet. Lehetőség van kamerával azonosítani növényt, így könnyen kideríteni annak nevét. Továbbá nem csak növény, de betegség keresésére is ad lehetőséget. Elég

[2] https://getplanta.com/

csak lefényképezni a növényt és az app diagnosztizálja, majd javaslatokat ad az ápolással kapcsolatban. Valamint a napfényigény egyszerű közlése mellett képes meghatározni a szobában lévő fényviszonyokat, így leellenőrizhető, hogy megfelelő-e a hely ahová helyeznénk a növényt. Azonban saját növényt felvenni csak egy előre meghatározott listából van lehetőség.

Felhasználói felület szinten jól átgondolt a felépítés, letisztult, könnyen használható. A visszajelzéseket átolvasva a teljesítmény és lokalozáció hiányossága



3. ábra Planta app

mellett, a legnagyobb negatívuma az alkalmazásnak, hogy előfizetéses modellt használ. Az ingyenes verzió funkciói minimálisra csökkentettek.

2.3.2 Blossom[3]

Funkciókat tekintve megegyezik a Planta appal, annyi különbséggel, hogy míg az előzőt lehet korlátozottan ingyenesen is használni, itt csak 7 nap próbaidő van miután fizetni kell a feliratkozást havonta. Ennek működését nem tudtam kipróbálni. A visszajelzésekből viszont látszik, hogy akárcsak az előzőnél elsősorban itt is az

[3] https://blossomplant.com/

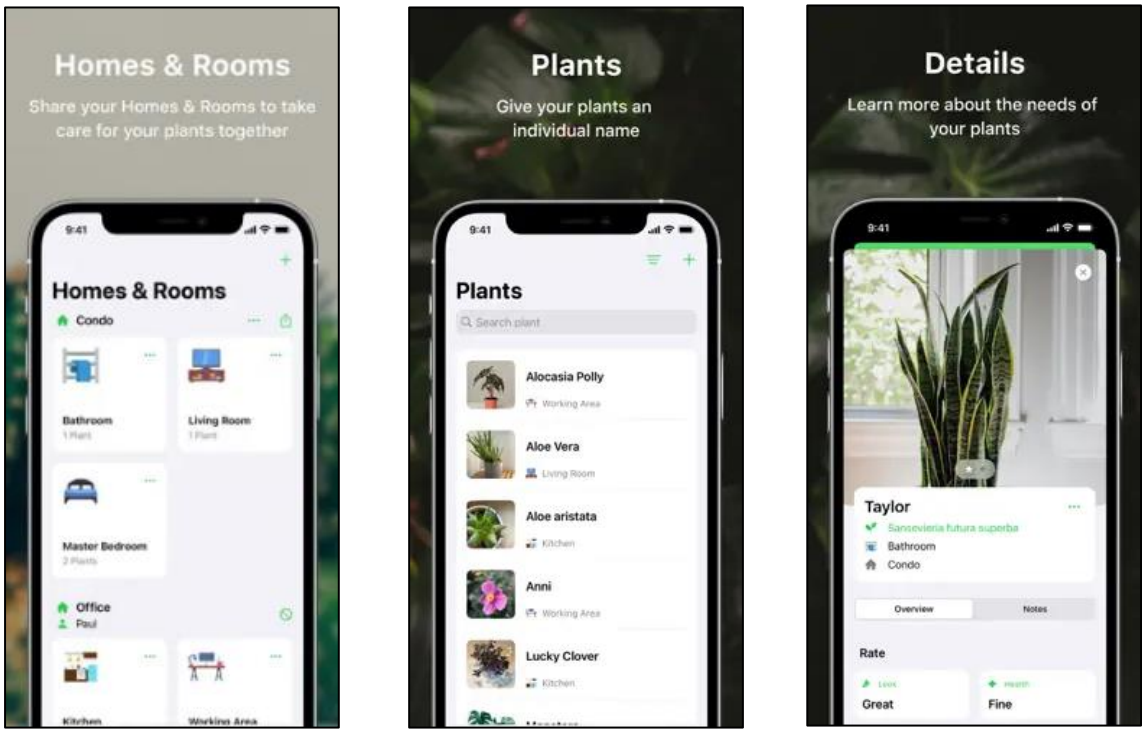
előfizetéses modell nem vonzó az emberek számára. Illetve a felhasználói felületet is kritika éri, sokan túl bonyolultnak találják, nem nyújt kellemes, letisztult felhasználói élményt.

4. ábra Blossom app

2.3.3 Plantbuddy[4]

Ezt az alkalmazást az iOS platformon lévő App Store kínálatában találtam. Így sajnos ezt sem tudtam kipróbálni. Viszont érdekes új funkciókat kínál az előzőek mellé. Itt már van lehetőség saját növényt elmenteni korlátok nélkül, így elnevezhető egyénileg a növény. Létre lehet hozni szobákat, melyek a saját otthonunkat reprezentálják és el lehet bennünk helyezni az elmentett növényeket. Ugyanúgy push értesítés formájában figyelmeztet a locsolásról, valamint fel lehet tölteni a telefon kamerájával készített képet a növényeinkhez és ezeket meg is lehet osztani más emberekkel a közösségből. Növényazonosítás és fénymeghatározás nincs az alkalmazásban, viszont az értékelése a három közül a legjobb.

[4] https://www.plantbuddy.app/index-en.php?ref=producthunt



5. ábra Plantbuddy app

2.3.4 Konklúzió

A megfigyelt alkalmazásokból látszik, hogy a feladat során meghatározott funkciók hasznosak és adott esetben igény is van rájuk. Mesterséges intelligencia nem tudom, hogy elérhető-e ezekben az appokban, viszont biztosra veszem, hogy hasznos kiegészítésük lenne. Így célom ezzel a technológiával kiegészítve egy ingyenes, mindenki számára elérhető alkalmazás készítése, mely segít az embereknek a növények gondozásával.

3 Felhasznált technológiák

3.1 Android

Az Android[5] egy Linux kernelre épülő mobil operációs rendszer, elsősorban érintőképernyős eszközökre (okostelefon, tablet) tervezve. Abból a célból született, hogy egységes nyílt forráskódú operációs rendszere legyen ezeknek a készülékeknek. Sokan csak a telefonokkal kapcsolják össze, ám jóval több rejlik mögötte. 2008-ban jelent meg első stabil változata és a kezdeti nehézségek ellenére népszerűsége azóta egyre csak növekszik. Ehhez nagyban hozzájárult széleskörű felhasználási lehetősége, hiszen minden olyan helyen kényelmes az Android, ahol alapvetően kicsi a kijelző, limitáltak az erőforrások és az adatbevitel nem egérrel és/vagy billentyűzettel történik. Az autóipar is nagy érdeklődést mutat iránta, a fedélzeti számítógépek használatában. Jelenleg piacvezető a mobil operációs rendszerek között.

6. ábra Mobil OS statisztika 2024[6]

Fejlesztés terén nagy előnye, hogy magas szintű nyelvekkel lehet alkalmazásokat készíteni rá. Először még Java, majd később 2019-ben a Kotlin vált az Android hivatalosan ajánlott fejlesztési nyelvévé, amely azóta a platform egyik legnépszerűbb és legrugalmasabb választása a modern Android-alkalmazások fejlesztésében [6]. Az elkészült alkalmazás könnyen közzé tehető a Google Play Áruházban, ahol az összes felhasználó számár elérhető.

[5] developer.android.com

[6] gs.statcounter.com/os-market-share/mobile/worldwide

3.2 Kotlin

A Kotlin[7] egy viszonylag új, nyílt forráskódú programozási nyelv, mely 2016-ban jelent meg a JetBrains[8] fejlesztésével. Fő célja egy modernebb, a Java[9] hiányosságait kiküszöbölendő biztonságos alternatíva szolgáltatása. Egyszerű és olvasható szintaxissal rendelkezik, miközben megőrzi a statikusan típusos nyelvek biztonságát. Ötvözi az objektumorientált és a funkcionális programozási paradigmákat, így lehetőséget biztosít, hogy a fejlesztők mindkét stílus eszközeit alkalmazzák a kódjukban. Habár a Java alternatívájaként készült, szorosan együttműködik vele, így Java kódot könnyedén integrálhatunk Kotlin kódba és fordítva. Ez fontos szempont az Android-fejlesztésben, mivel rengeteg meglévő Java-könyvtár és API használható zökkenőmentesen Kotlinból.

Sok nyelvben problémát okoz a null referenciák kezelése. A Kotlin egyik legnagyobb előnye a null-biztonság, amelyet a nyelv típusrendszere biztosít. Ez annyit tesz, hogy a változók értéke alapértelmezetten nem lehet null, biztonságosabbá téve így az alkalmazásokat.

3.3 Jetpack Compose

A Jetpack Compose[10] egy nyílt forráskódú Kotlin alapú deklaratív UI keretrendszer Androidhoz. Segítségével a felhasználói felületeket a kódban határozhatjuk meg, így azokat egyszerűbben testre szabhatjuk. Támogatja az állapotalapú megjelenítést és megkönnyíti az újra felhasználható komponensek létrehozását. A Compose szorosan integrálódik a Jetpack könyvtárakkal, támogatva például a LiveData és a ViewModel használatát, ami tisztább, karbantarthatóbb kódot eredményez.

3.3.1 Deklaratív és imperatív megközelítés különbsége

Az Android fejlesztés hagyományosan az imperatív megközelítésre támaszkodott, ahol a felhasználói felület elemeit XML-ben határozzuk meg, majd a Java vagy Kotlin kódban programozzuk a viselkedésüket.

---

[7] https://kotlinlang.org/docs/home.html

[8] https://www.jetbrains.com/

[9] https://docs.oracle.com/en/java/

[10] https://developer.android.com/compose

```
<TextView
    android:id="@+id/textView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Eredeti szöveg" />

<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Szöveg frissítése" />
```

A UI állapota és annak frissítése explicit módon történik. Minden változtatásnál pontosan meg kell mondani, hogyan és mikor frissítse a rendszer a felhasználói felület elemeit. Például, ha egy szöveget szeretnénk frissíteni egy gomb megnyomása után, akkor meg kell keresni a megfelelő nézetet, és manuálisan beállítani az új értéket.

```
val textView: TextView = findViewById(R.id.textView)
button.setOnClickListener {
    textView.text = "Új szöveg"
}
```

Ez a folyamat egy komplexebb alkalmazás esetében nehezen karbantarthatóvá válik.

Deklaratív programozásban a fejlesztő nem azt mondja meg, hogyan kell végrehajtani egy műveletet, hanem azt határozza meg, hogy milyen eredményt szeretne elérni. A rendszer pedig automatikusan elvégzi a szükséges lépéseket. Ez leegyszerűsíti a UI fejlesztést, mivel a felhasználói felületet közvetlenül a kívánt állapotok alapján definiáljuk, és a rendszer frissíti azt, amikor az állapot változik. Jetpack Compose-ban a felhasználói felület elemei közvetlenül a kódban vannak meghatározva. Amikor az állapot megváltozik, a Compose újrarendereli a UI-t a legfrissebb állapot alapján. Ez lehetővé teszi a tiszta és karbantartható kód írását, mivel a komponensek mindig a jelenlegi állapotot tükrözik, és a rendszer kezeli a frissítéseket.

```
var counter by remember { mutableStateOf(0) }

Column {
    Text(text = "Érték: $counter")
    Button(onClick = { counter++ {
        Text("Számláló növelése")
    }
}
```

A példában a remember és mutableStateOf kulcsszavakat használom. Az előbbi szolgál arra, hogy az állapotot megtartsuk az újrarenderelés során. Az utóbbi pedig dinamikusan tárolja az állapotot. Ez lehet szám, szöveg, bool, de akár egy osztály is. Az

értéke módosítható, és bármely változás automatikusan elindítja a felhasználói felület újbóli kirajzolását, hogy a legfrissebb állapotot tükrözze.

3.3.2 Előnyök XML-lel szemben

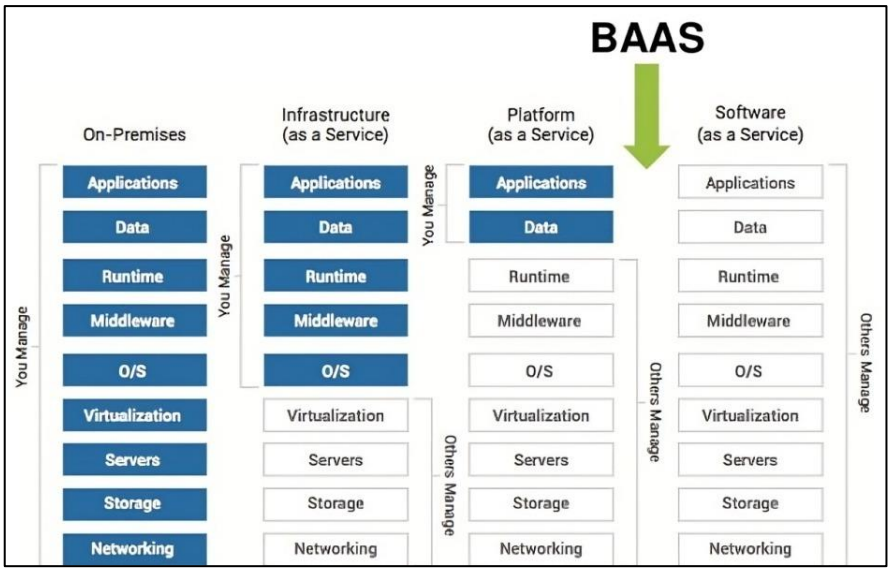
Az említett állapotkezelés mellett Jetpack Compose használatával egyszerűbb és tisztább kód írható. Jobb lesz a karbantarthatóság, mivel egy helyen van a UI és a viselkedés. Segítségével létrehozhatóak újra használható komponensek. Továbbá a Compose teljesen Kotlin alapú, így az összes Kotlin nyújtotta előny (pl. típusellenőrzés, null biztonság, lambda kifejezések) közvetlenül kihasználható.

3.4 Firebase

A Firebase[11] a Google által fejlesztett felhőalapú BaaS – Backend as a Service platform. Különféle szolgáltatásokat kínál mobil- és webalkalmazások készítéséhez. Fő célja, hogy egyszerűsítse az alkalmazások backend-részének fejlesztését, lehetővé téve az erőforrás használat kiszervezését [7].

A BaaS egy olyan felhőalapú szolgáltatás, amely lehetővé teszi az alkalmazás backend-funkcióinak egyszerű kezelését saját szerverek üzemeltetése és saját backend-infrastruktúra kialakítása nélkül. A BaaS szolgáltatók, mint a Firebase, előre kialakított API-kat és SDK-kat (Software Development Kit) biztosítanak, amelyekkel a frontendet könnyen össze lehet kapcsolni a felhő alapú háttérrel.

[11] https://firebase.google.com/



7. ábra BaaS helye a szolgáltatási modellek között

A BaaS nagy előnye, hogy gyorsabbá teszi a fejlesztést és csökkenti az üzemeltetési költségeket. A skálázhatóság, a biztonság és a folyamatos rendelkezésre állás automatikusan biztosítva van a szolgáltató által.

Azért döntöttem Saját backend helyett a Firebase használata mellet, mert a jelenlegi iparban rendkívül elterjedt a használata és szerettem volna minél jobban megismerni az általa nyújtott lehetőségeket. Rengeteg az elérhető funkció, melyek különböző kategóriákat fednek le, mint például tesztelés vagy analitika. Az általam használtakat a következőkben röviden bemutatom.

8. ábra Firebase funkciói [8]

3.4.1 Authentication

Könnyen használható SDK-val egyszerű és biztonságos hitelesítési lehetőséget biztosít a felhasználók számára. Támogatja az egyszerű email-jelszó azonosítást, illetve a népszerű Google, Facebook vagy X (régebben Twitter) külső szolgáltatókkal történő bejelentkezést.

3.4.2 Firestore

A Firestore egy NoSQL alapú adatbázis. Az adatokat dokumentumokban tárolja, melyek gyűjteményekbe (collection) rendezhetőek, így kialakítható egy egyszerű, többszintű hierarchia. Támogatja az alap típusok (szám, karakter stb.) tárolását, valamint komplex objektumok kezelését is. Valós időben értesíti a klienst a változásokról és csak a megváltozott részeket küldi el.

3.4.3 Storage

A Firebase Storage egy skálázható, biztonságos fájltárolási megoldás, amely kifejezetten nagy méretű fájlok, például képek, videók vagy dokumentumok tárolására és kezelésére szolgál. A fájlok egyszerűen feltölthetők és letölthetők a Firebase SDK-k segítségével, és a tárolt adatokhoz való hozzáférést biztonsági szabályokkal lehet korlátozni.

3.4.4 Cloud Messaging (FCM)

Megbízható és erőforráshatékony kapcsolatot biztosít a szerver és az eszközök között, mellyel lehetővé teszi üzenetek és push értesítések küldését és fogadását.

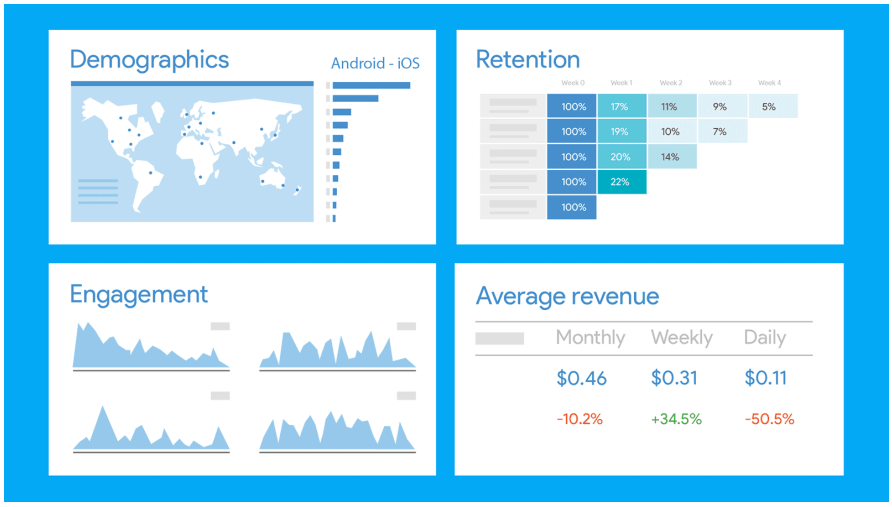
3.4.5 Cloud Functions

Lehetővé teszi backend kód automatikus futtatását események hatására, melyeket a Firebase szolgáltatásai vagy *HTTP* kérések váltanak ki. A kódot a Google Cloud tárolja és egy menedzselt környezetben fut, lehetőséget adva így az ütemezésre (például napi egyszer, 8 órakor fusson).

3.4.6 Analytics

Az Analytics különböző adatokat mér az alkalmazás használata során és jelentést készít róluk, ezzel segít a fejlesztőknek jobban megérteni alkalmazásuk használati szokásait. Például kimutatja a felhasználói aktivitást, az éppen bejelentkezett

felhasználók számát, mennyi időt töltenek el az alkalmazás használatával, milyen régiókból származnak és még sok egyéb statisztikát.



9. ábra Firebase Analytics példa

3.4.7 Crashlytics

Az Analytics-hez hasonlóan a Crashlytics is információt gyűjt, azonban ahogy azt a neve is sugallja, az alkalmazás használata során előforduló crash-ekről. Összefoglalja és rangsorolja az így felmerülő stabilitási problémákat, időt spórolva ezzel a fejlesztőknek a hiba detektálásában és kijavításában.

3.5 API

Az alkalmazásban elérhető bizonyos funkciók megvalósításához, mint például a mesterséges intelligencia integrálása, különböző API-kat kerestem és próbáltam ki. Az alkalmazásba végül bekerült, fontosabb API-kat az alábbiakban bemutatom.

3.5.1 Perenual API

Szükségem volt növények listájára, melyek megjelennek az appban a felhasználó saját növényein túl. Ehhez a Perenual[12] szolgátatását találtam, amely tulajdonképpen egy növényadatbázis API. Több mint 10,000 faj elérését teszi lehetővé egyszerű *HTTP* kérések formájában. Rengeteg adatot nyújt az egyes növényekkel kapcsolatban. Továbbá

[12] https://perenual.com/docs/api

sok egyéb lehetőség is megtalálható, mint például gondozási leírások tulajdonság szerint tagolva:

```
{
     "id": 1,
     "type": "sunlight",
     "description": "Sunlight is the most important environmental factor
controlling the growth and health of European silver fir (Abies alba). ..."
},

     "id": 3,
     "type": "watering",
     "description": "Watering European silver fir trees is essential for
them to stay healthy. It is important to provide them with regular watering,
..."
}
```

Sajnos használata ingyenes felhasználóknak korlátozott, naponta 100 kérés az elérhető keret. Azonban a feladatomhoz egyéb alternatívát, ami legalább ilyen jól illeszkedik hozzá nem találtam.

3.5.2 AI/ML API

Mesterséges intelligenciához több API-t is kipróbáltam, végül az AI/ML[13] mellett döntöttem. Rengeteg AI modell elérhető a kínálatában: nagy nyelvi modell (LLM – Large Language Model), képfeldolgozó modell, beszédfelismerő modell stb. Ezeken belül pedig szintén nagy a választék, hozzáférést nyújt például az OpenAI GPT modelljeihez vagy a Google által fejlesztett Gemini, illetve a Meta Llama modelljéhez is és mind szabadon használható. Természetesen itt is vannak ingyenes korlátok (óránként 10 kérés max 256 tokennel), viszont teljesen ingyenes, korlátlan mesterséges intelligencia szolgáltatás nem elérhető sehol. Így hát a hatalmas választék miatt esett erre a választásom.

3.5.3 Kindwise plant.id API

Az alkalmazásban elérhető kép alapú növényazonosításhoz a Kindwise plant.id[14] API-ját használom. Ez egy speciálisan növényazonosításra tervezett API, mely képfelismerési technológiák segítségével működik. Elküldhető neki egy vagy több kép, illetve az azonosításban segítendő további adat, mint például a kép készítésének ideje,

[13] https://docs.aimlapi.com/

[14] https://www.kindwise.com/plant-id

földrajzi helye (hosszúság, szélesség formájában) stb., melyek segítségével eredményesen meg tudja mondani milyen növény van az elküldött képen/képeken.

Lehetőséget nyújt még növény egészségi állapotának vizsgálatára is. Szintén kép alapján meg tudja adni mi baja lehet a növénynek. Azonban ez a funkció teljes egészében fizetős.

3.6 Retrofit

Az API-k kiválasztása után szükség van egy hatékony módszerre azok elérésére és használatára. Ebben nyújt segítséget a Retrofit[15], mely egy *HTTP* kliens könyvtár. Lehetővé teszi az API-k egyszerű és strukturált elérését az Android alkalmazásban. Az API-k metódusait Java interfészekként hozhatjuk létre, ahol a Retrofit automatikusan lekezeli a *HTTP* kéréseket és válaszokat.

További előnye, hogy támogatja az olyan adatátviteli formátumok, mint a *JSON*, automatikus konvertálását Java modell osztályokká, leegyszerűsítve ezzel az adatfeldolgozást. Ezt a folyamatot gyakran a Gson könyvtár integrálásával egészítik ki, amely hatékonyan végzi el ezt az átalakítást.

3.7 Dagger-Hilt

A Dagger-Hilt[16] az Android-fejlesztésben használt népszerű dependency injection (függőséginjektáló) keretrendszer, amely a Google által fejlesztett Dagger egyszerűsített, Androidra optimalizált változata. A függőséginjektálás egy olyan tervezési minta, amely lehetővé teszi az objektumok közötti kapcsolatok létrehozását anélkül, hogy közvetlenül kellene példányosítani őket, ezáltal lazább csatolást elérve, növelve a tesztelhetőséget és karbantarthatóságot.

A Hilt automatizálja és leegyszerűsíti a Dagger által használt folyamatokat. Ennek köszönhetően a fejlesztőknek nem kell manuálisan konfigurálniuk a Dagger különféle moduljait és komponenseit. A Hilt automatikusan létrehozza azokat a helyeket, ahol a különféle osztályokat injektálni kell.

[15] https://square.github.io/retrofit/

[16] https://dagger.dev/hilt/

3.8 Google Cloud Platform

A Google Cloud Platform[17] (GCP) a Google felhőalapú szolgáltatásainak gyűjteménye, amelyek között infrastruktúra- és platformszolgáltatások is találhatók. Részletesen nem mutatom be mindet, mivel csak a Google Maps SDK-t használom. Ez egy ingyenes szolgáltatás, mellyel megjeleníthető interaktív térkép rajta helyadatokkal és minden funkcióval, ami a Google Maps-el jár.

3.9 GitHub

A GitHub[18] az iparban széles körben elterjedt verziókezelő platform. A Git nevű elosztott verziókezelő rendszerre épül, amely lehetővé teszi a kód különböző verzióinak nyomon követését és az egyes változtatások biztonságos kezelését. Lehetőséget nyújt, hogy a fejlesztők visszatérjenek korábbi verziókhoz vagy együtt dolgozzanak ugyanazon a projekten anélkül, hogy felülírnák egymás munkáját, illetve pull requesteken keresztül javasolhatnak módosításokat.

Munkám során ezen a platformon tettem elérhetővé a projektet, így biztosítottam a kód biztonságos tárolását, valamint lehetőséget adtam annak átlátható nyomon követésére és fejlesztésére.

[17] https://cloud.google.com/?hl=en

[18] https://docs.github.com/en

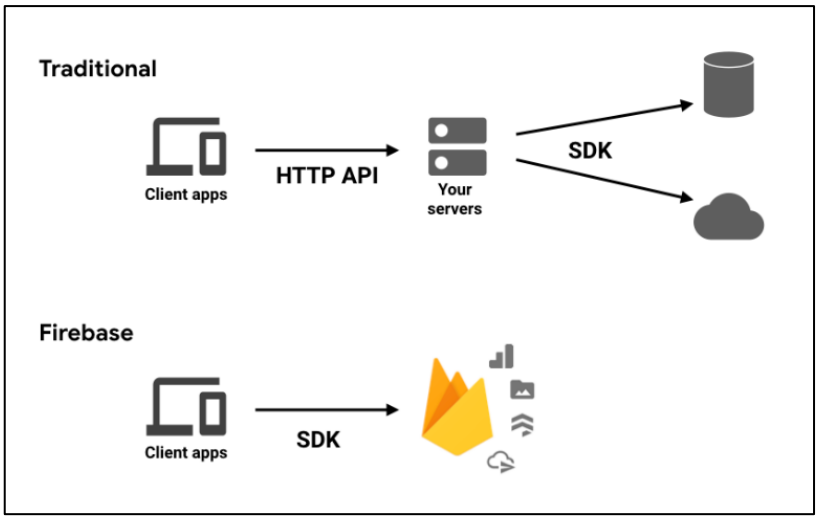
4 Tervezés

Tervezés során kiválasztottam az architektúrákat, melyeket használni szeretnék az alkalmazás elkészítése során. Android alapú fejlesztésnél több lehetőség közül lehet válogatni. Én a tanulmányaim során megismert többrétegű architektúrát és MVVM mintát választottam. Ezek a legáltalánosabban használtak a platformon. Továbbá készítettem a fontosabb oldalakhoz nagyon egyszerű látványterveket, melyek segítségével később pontosan meghatároztam milyen oldalakból álljon az alkalmazás és azok között hogyan működjön a navigáció. Gráf formájában reprezentáltam az előállt útvonalakat.

4.1 Kliens-szerver architektúra

A kliens-szerver architektúra lényege, hogy elválasztja egymástól a klienst és a szervert. A kliens olyan számítógép, amely hozzáfér egy szolgáltatáshoz, amelyet egy számítógép-hálózathoz tartozó másik gép nyújt. A kiszolgáló vagy szerver olyan számítógépet, illetve szoftvert jelent, amely más gépek számára a rajta tárolt vagy előállított adatok felhasználását, a kiszolgáló hardver erőforrásainak kihasználását, illetve más szolgáltatások elérését teszi lehetővé [9].

A Firebase ismertetésénél megindokoltam mellette való döntésem okát. Használatával nagy mértékben egyszerűsíthető és gyorsítható a fejlesztés. Az alkalmazás közvetlenük kapcsolódik a Firebase-hez, ami leegyszerűsíti az adatáramlást. Azonban ennek hátrányai is vannak, például nincs teljes kontroll a backend felett. Ezért a felhasználói felület és az üzleti logika egy helyre, a klienshez kerül. Viszont az alkalmazás méretét tekintve ezek a problémák nem okoznak jelentős gondot. Továbbá, ha később az alkalmazás mérete és összetettsége növekedne, a Firebase könnyen integrálható más rendszerekkel és szükség esetén az architektúra egyszerűen átalakítható.

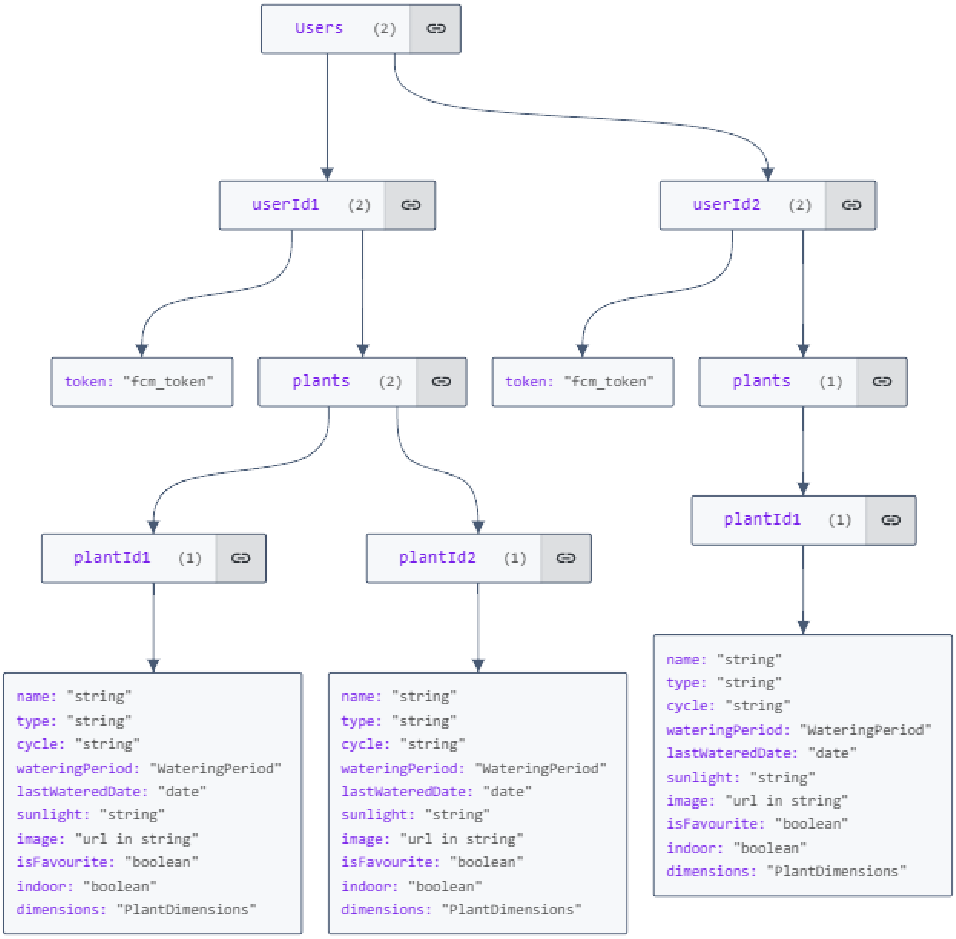


10. ábra Kliens-szerver architektúra Firebase-el

4.1.1 Firestore

Az adatbázisban a felhasználókat és növényeiket fogom nyilvántartani. Több felhasználó lehet, mindenkihez egyedi azonosító tartozik és egy felhasználóhoz több növény tartozhat, de egy növény csak egy felhasználóhoz. A növények egyedi ID alapján különböznek egymástól, így lehetnek azonos nevűek is.

Egy kollekcióba kerülnek a felhasználók, külön dokumentumokként. Egy felhasználó dokumentumon belül pedig újabb kollekcióba a növények lesznek. A növények mellé pedig szükséges a felhasználó tokenjét is elmenteni, aminek segítségével küldhető számára értesítés. Ez egy egyszerű mező lesz a növények kollekció mellett. A felhasználó nevét, email címét, profilképét stb. nem szükséges tárolni, mivel azok az Authentication SDK-val elérhetőek. A növényekhez tárolt további adatok az adatstruktúrát ábrázoló 10. ábrán megtalálhatóak.

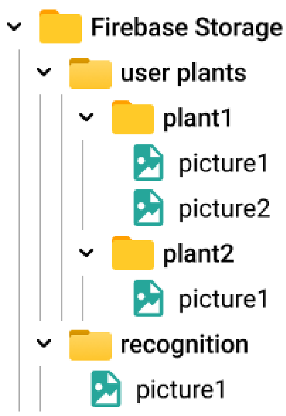


11. ábra Firestore adatstruktúra

4.1.2 Storage

A Storage-ot képek tárolására használom. Ide töltöm fel a felhasználók által a növényeikhez készített képeket, illetve a növényazonosításhoz készült képeket. A fájlok tárolásához egyszerű mappastruktúrát biztosít a Storage.

Külön mappába helyezem a felhasználók növényeit és az azonosításhoz használtakat. Egy növény képe lecserélhető, viszont minden elmentett képet megőrzök az összeállítások, montázsok megvalósításához. Mivel így egy növényhez több kép is tartozhat, ezek külön mappákban lesznek. A 11. ábrán szemléltetem az elképzelt mappastruktúrát.



12. ábra Storage mappastruktúra

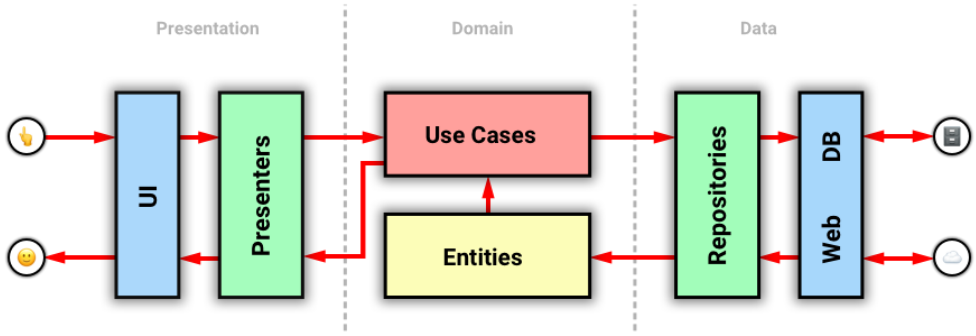
4.2 Magas szintű architektúra

Hasonló szoftverek fejlesztéséhez leginkább a többrétegű (v. háromrétegű) architektúrát alkalmazzák. Ez az architektúra megkülönbözteti az alkalmazás három fő komponenscsaládját, avagy rétegét: a megjelenítési réteget, az üzleti logikai réteget, és az adatelérési réteget. Minden egyes réteg más-más funkcionalitásért felel. Ez a logikai szervezés megkönnyíti a szoftverfejlesztők munkáját azáltal, hogy egyértelmű felelősségi köröket és határokat jelöl ki a rétegekben. Az elnevezés arra utal, hogy mindegyik réteg maga is tovább bontható további rétegekre az alkalmazás komplexitásának függvényében.

Mivel a megvalósítandó alkalmazás egyszerű működéssel bír és az összetett, több rétegű logikai szétválasztás növelheti a komplexitást anélkül, hogy valódi előnyt nyújtana, a három fő réteget nem bontom tovább. Ezek a következő szerepekkel bírnak:

- **Megjelenítési réteg:** A felhasználói felület különböző elemekből áll (gombok, szövegek, szövegbeviteli mezők stb.), melyekkel a felhasználók interaktálhatnak. Jetpack Compose-ban ezek a *Composable*-ök. Ez a réteg az adott képernyőn lévő elemek megjelenítéséért és a velük történő interakciók kezeléséért felelős. Legfontosabb feladata, hogy a kapott adatokat felhasználóbarát módon jelenítse meg (például dátum formázása). Továbbá amikor a felhasználó adatot visz be a rendszerbe, ez a réteg felel az adatok validációjáért is.
- **Üzleti logikai réteg:** Itt található az alkalmazás működését és a különböző használati eseteket leíró logika. Az üzleti logikai réteg célja a felhasználói interakciók és az adatok kezelése közötti közvetítés, amely biztosítja az adatok helyes feldolgozását és integritását.
- **Adatelérési réteg:** Az adatelérési réteg feladata az adatforrások kényelmes elérésének biztosítása. Fő funkciója az elemi adatszolgáltatási műveletek biztosítása, mint egy új rekord tárolása, meglevő módosítása vagy törlése. Célja, hogy kényelmes, a felsőbb réteg számára egyszerűen használható szolgáltatásként nyújtsák az adatokat. Ide tartozik például az

SQL parancsok kezelése, valamint az adatbázis tárolási modelljének leképzése az üzleti logika számára is kényelmesen használható módon.



13. ábra 3 rétegű architektúra

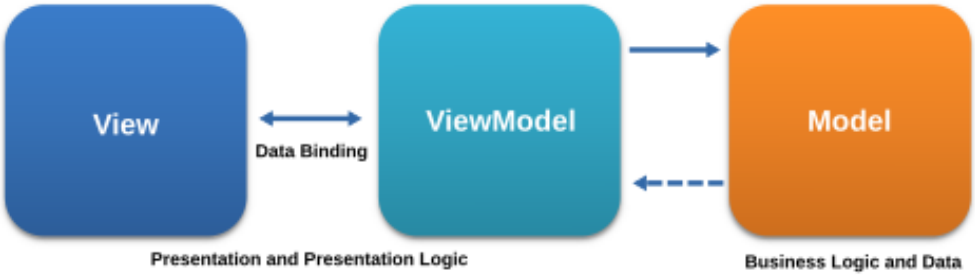
Az előző fejezetben kitértem arra, hogy a felhasználói felület és az üzleti logika egy helyen lesz, az architektúrán viszont az látszik, hogy ezek elkülönülnek egymástól. Azonban így is, hogy egy helyen vannak, szét lehet választani őket, így kialakítható a felvázolt 3 réteg a szerepüknek megfelelően.

4.3 Prezentációs architektúra

Az MVVM minta célja, hogy a felhasználói felületet és a mögötte lévő logikát szétválassza, és ezzel egy lazább csatolású alkalmazást hozzon létre, ami növeli a tesztelhetőséget, a karbantarthatóságot és az újrafelhasználhatóságot. Három (+1) fő részből áll:

- **Model**: Az alkalmazás üzleti modelljét tartalmazza, amelyet a ViewModel-ek használhatnak az adatok tárolására. Függetlenül működik a többi résztől, úgymond „nem tud róluk”.
- **View**: A felhasználói felület leírását tartalmazza, és a tisztán a nézetekhez kapcsolódó logikát (pl.: animációk kezelését). Jetpack Compose keretrendszerben a Composable függvényekből áll. Figyeli a ViewModel-ben a változásokat és azalapján változtatja a megjelenést.
- **ViewModel**: A nézet absztrakciója, mely tartalmazza a nézet állapotát és a nézeten végrehajtható műveleteket. Független a nézettől, a laza csatolást a ViewModel és a nézet között az adatkötés biztosítja. Ez a réteg elfedi a Model réteg komplexitását, így könnyebbé teszi az adatok elérését.

- **Services**: Az alkalmazás üzleti logikáját tartalmazó osztályok, amelyeket a ViewModel-ek használnak. Ha minden üzleti logika a ViewModel-ekben lenne, azok túl bonyolultak és átláthatatlanok lennének. Ez nem az MVVM minta része, de megemlítem, mivel rendkívül hasznos és alkalmazom a megvalósítás során.



14. ábra MVVM minta

A mintát úgy alkalmazom, hogy minden képernyőhöz külön View és ViewModel tartozik. Ennek további előnye a logika kiszervezésén túl, hogy a ViewModel osztályok élettartartama hosszabb a nézetet tartalmazó Activity osztályokénál. Így a ViewModel-ben tárolt értékek nem vesznek el az Activity megsemmisülésekor, például a képernyő elforgatásakor. Ezzel biztosítható a felhasználói felület konzisztensen tartása.

4.4 Tervezési minták

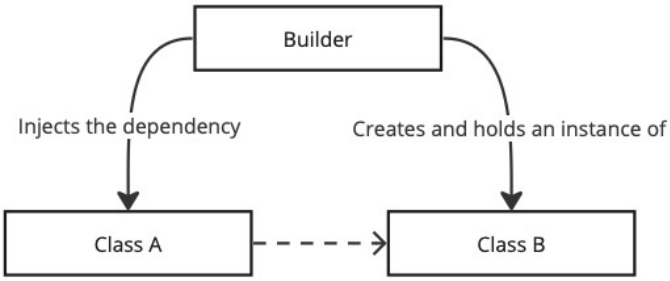
A tervezés során három főbb mintát határoztam meg, melyek alapján felépítettem az alkalmazást. Ezek a DI (Dependency Injection – Függőséginjektálás), Repository és Interactor tervezési minták. Használatuk jól illeszkedik az alkalmazott architektúrába, lehetővé teszik a különböző felelősségi körök egyértelmű elválasztását.

4.4.1 Függőséginjektálás

A függőséginjektálás egy olyan technika az objektumorientált programozásban, amely csökkenti az objektumok közötti szoros kapcsolatokat. Itt a „függőség” egy olyan kódrészletet jelent, amely egy másik erőforrástól, gyakran egy másik objektumtól függ a működéséhez. Például egy alkalmazásban két osztály van: A és B osztály. Ha az A osztály létrehoz egy példányt a B osztályból, akkor az erősen függ tőle, ami szoros csatolást hoz létre, megnehezítve így a kód tesztelését, módosítását vagy újrafelhasználását [10].

A függőséginjektálás lehetővé teszi, hogy a B osztály példányát külsőleg adjuk át az A osztálynak, például egy konstruktoron vagy publikus tulajdonságon keresztül, így A

osztály működhet B osztály teljes definíciója nélkül is. Ezáltal a kód lazán csatolt és karbantarthatóbb lesz, mivel a függőségek nincsenek beágyazva a kódba, hanem kívülről kerülnek átadásra.

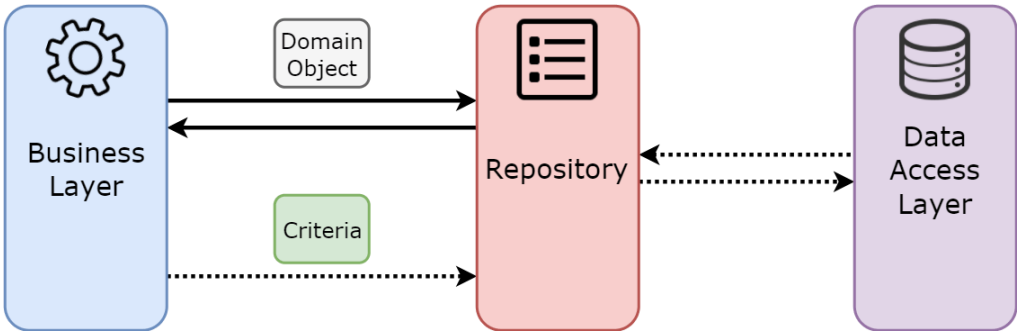


15. ábra Függőséginjektálás példa

A **3.7 Dagger-Hilt** fejezetben bemutatott keretrendszer látja el a függőségek kezelését az alkalmazásban. Segítségével elég egy objektumot, például osztályt megjelölni *@Inject* annotációval és konstruktorába paraméterként felvenni a szükséges komponenseket, a Dagger-Hilt pedig automatikusan elvégzi az injektálást. Ilyen módon kapják meg a ViewModel-ek az Interactorokat, Service-eket, tovább haladva a láncon azok pedig a Repository-kat, melyek pedig a működésükhöz szükséges, például Firebase függőségeket.

4.4.2 Repository

A Repository minta az adatelérési rétegben kap szerepet. Ez a minta felelős az adatok lekéréséért és kezeléséért, legyen szó távoli forrásokról (például egy API-ról) vagy helyi adatbázisról. A Repository elrejti az adatok forrásának részleteit, és egységes interfészt biztosít az adatokhoz való hozzáféréshez az üzleti logikai réteg számára. Ezáltal a kódban az adatforrásokhoz való hozzáférés egységes felületen keresztül történik, így az alkalmazás logikája nem függ az adatkezelés részleteitől.



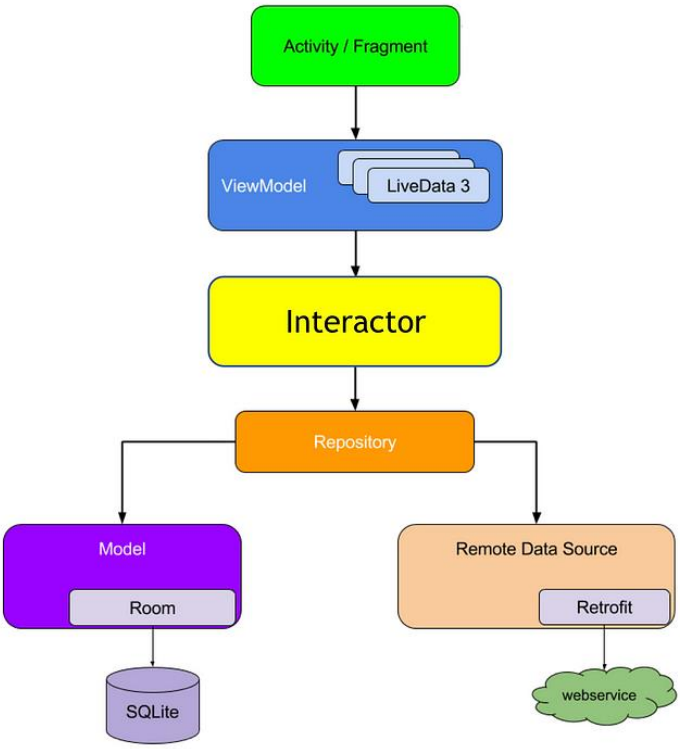
16. ábra Repository minta

A mintát gyakran együtt használják a DTO-kal (Data Transfer Object). Ez lehetővé teszi, hogy az adatokat a repository különböző forrásokból egységes formátumban szolgáltassa a felsőbb rétegek számára, így azok nem függenek az adatforrás konkrét struktúrájától. Az alkalmazásban ilyen objektumok segítségével biztosítom az üzleti logika számára az API-ból és a Firestore-ból származó adatokat.

Továbbá azáltal, hogy a repository egy egységes felületet biztosít az adatokhoz, a tesztelés során könnyen létrehozhatunk általa „mock” vagy „stub” osztályokat, amelyek valós adatforrások nélkül is biztosítják a szükséges adatokat a logikai réteg teszteléséhez. Ez a megközelítés lehetővé teszi, hogy az üzleti logikát függetlenül teszteljük az adatforrásoktól, így az egységtesztek gyorsabbak és könnyebben karbantarthatók.

4.4.3 Interactor

Az Interactor minta a felhasználói műveletek logikáját és az adatelérés folyamatát foglalja magában, gyakran az alkalmazás üzleti rétegében kap szerepet. Az Interactorok felelősek az alkalmazás üzleti szabályainak végrehajtásáért, és kapcsolatot teremtenek az adatelérési réteg (például Repository) és a felhasználói felület között. Az Interactorok használatával a logikai rétegek elkülöníthetők, és kezelhetőek az egyes műveletek összetett folyamatai úgy, hogy az adatkezelést elvégző Repository-któl függetlenek maradnak. Ez lehetővé teszi az üzleti logika egyszerűbb bővítését és újrafelhasználását különböző részekben.



17. ábra Interactor minta

Alkalmazásomban a különböző Repository-k eléréséhez alkalmazok Interactor-okat. Az előzőekben említett előnyökön túl az Interactor megkönnyíti még az egységtesztek írását is, mivel különálló logikai egységként kezelhető, és az üzleti folyamatok tesztelése során egyszerűen „mock” vagy „stub” osztályokkal helyettesíthető. Ezáltal az alkalmazás logikája anélkül tesztelhető, hogy az adatelérési réteg vagy a felhasználói felület implementációjára támaszkodna.

4.5 Felhasználó felület fontosabb oldalai

A megvalósítandó architektúrák és alkalmazandó tervezési minták kiválasztása után hozzáláttam a felhasználói felület látványterveinek elkészítéséhez. Azért kezdtem ezzel, mert számomra így könnyebb volt az alkalmazás tényleges felépítésének kialakítása. A következőkben bemutatom a tervet. Fontos azonban, hogy ez nem a végleges állapotot tükrözi, hanem a tervezésnek a legelejét, amiből aztán a megvalósítás során kinőtte magát az alkalmazás.

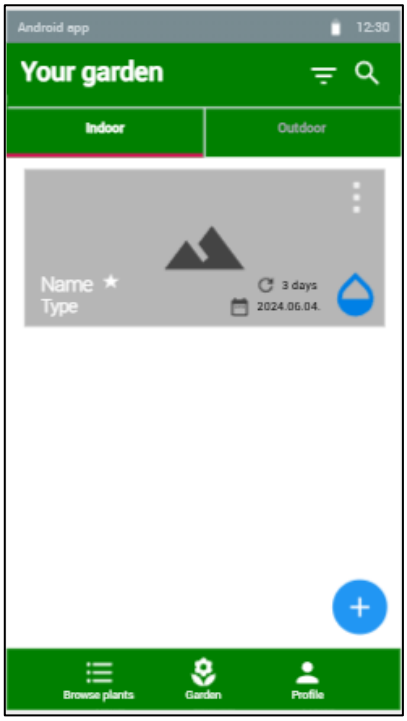
Rengeteg eszköz érhető el online formában is UI látványtervek készítéséhez, viszont ezek legtöbbje esetén saját kézzel kell megrajzolni minden komponenst. Tekintve, hogy az én képszerkesztő és rajzoló tudásom kimerül a Microsoft Paint-ben, olyan oldalt kerestem, ahol előre adottak a különböző elemek (például gombok, ikonok,

szövegbeviteli mezők stb.). Végül a Fluid UI[19] webes szerkesztőt találtam erre a célra, amely végtelenül egyszerű megjelenítést biztosít.

4.5.1 Növényeket listázó oldal

A növények megjelenítését a listákban kis kártyákként képzeltem el, melyek háttere a növény képe. Az API-ból származó növények esetében csak néhány adat szerepel a kártyán, míg a felhasználó virágainál gombok is elérhetőek.

A 17. ábrán a felhasználó növényei listájának megjelenítése látható. A Top App Bar alatt szerepel a beltér/kültér szűrő. A Jobb felső sarokban a nagyító utal arra, hogy lehetőség van szövegesen keresni, a jobb alsó sarokban lévő gombbal pedig új növény vehető fel a listába. A kártyán lévő gombok egy vízcsepp ikon és egy csillag ikon formájában jelennek meg. Az előbbi szolgál a locsolás dátumának beállítására az aktuális napra, az utóbbival pedig kedvencnek jelölhető a növény. A kártyára kattintva érhető el az adott elem részletes nézete. Továbbá az alsó navigációs sávon látható gombok segítségével navigálhat navigálhat a felhasználó a megfelelő oldalra.



18. ábra Felhasználó növényei oldal

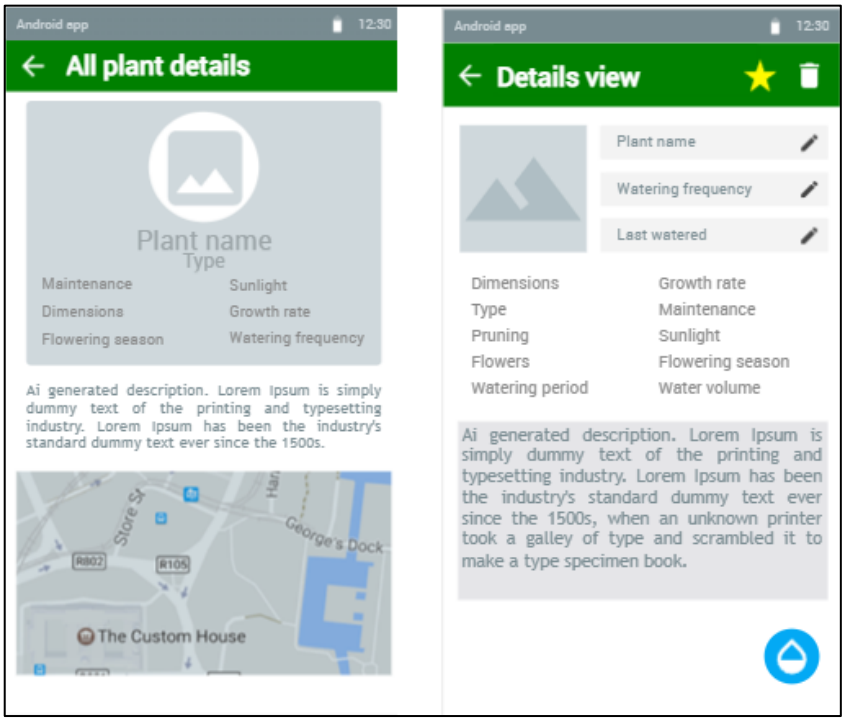
[19] https://www.fluidui.com/

Az API-ból vett növények oldalához nem készült terv, mivel néhány apróságot leszámítva megegyezik kinézete a felhasználó növényeinek oldalával. Ezek kimerülnek az előbbiekben említett kártyák különbségében, illetve a beltér/kültér szűrő elhagyásában. Továbbá a jobb alsó gomb megmarad, de funkcionalitása már nem új növény felvétele lesz, hanem a kamera megnyitása kép alapú azonosításhoz.

4.5.2 Növények részletes nézete

A növények részletei is hasonlítanak egymáshoz, azonban más-más elemek jelennek meg a két oldalon, így mindkettőhöz készítettem tervet.

Az API növények részleteinél (18. ábra bal oldali kép) megjelenik a képe, alatta a neve és néhány egyéb adata. Ezen szekció alatt kap helyet a mesterséges intelligencia által generált szöveg. Az oldal alján pedig a térkép helyezkedik el.



19. ábra Növények részletei (API növények balra, felhasználói növények jobbra)

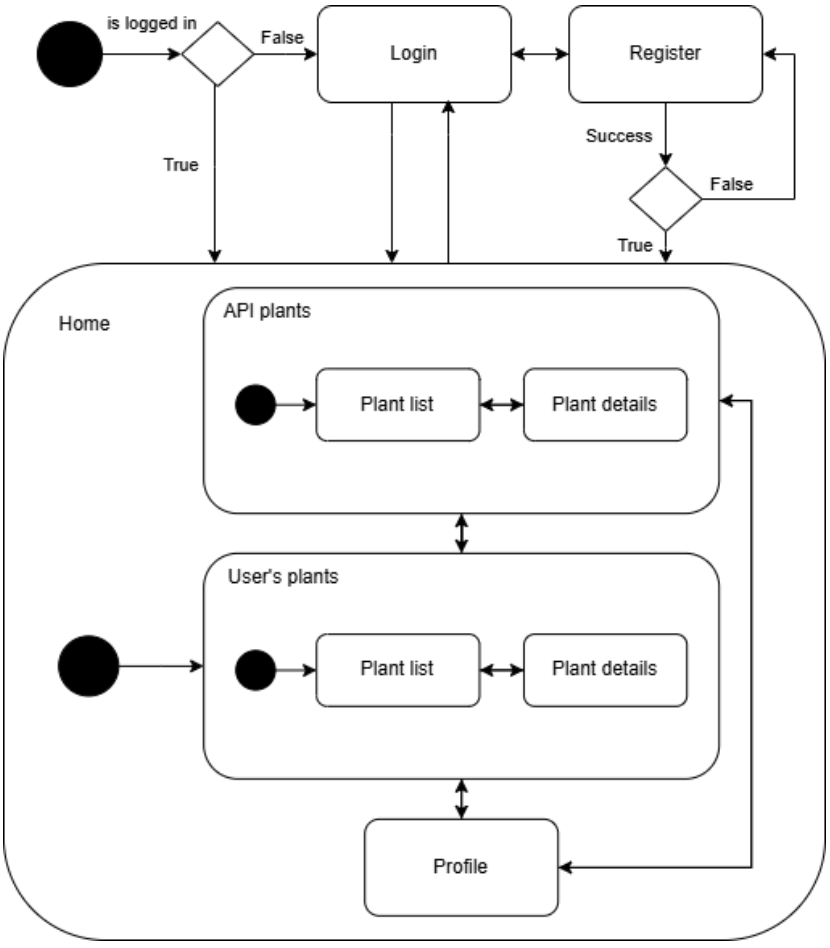
A felhasználó növényeinek részletes nézetén (18. ábra jobb oldali kép) ugyanúgy megjelenik a kép és az adatok, melyek itt már szerkeszthető mezőben szerepelnek. A felső sávban érhető el a növény törlését végző gomb, illetve a csillaggal itt is beállítható a növény kedvenc státusza. Az adatok alatt pedig a mesterséges intelligencia szövege szerepel. Továbbá megjelenik egy gomb a bal alsó sarokban, mellyel a kártyához hasonlóan az öntözés dátuma állítható az aktuális dátumra. Az ábrán néhány adatnál nem

látszik azok szerkeszthetősége, viszont maga a koncepció már megvolt a tervezés ezen fázisában is.

A bejelentkező, regisztrációs, profil oldalakhoz nem készítettem látványtervet, hiszen ezek nagyon egyszerű nézetek és funkcióikat tekintve nem, vagy csak kis mértékben térnek el a más alkalmazásokból megszokott hasonló oldalaktól.

4.6 Navigációs gráf

A felhasználói felület tervezése után könnyebben meghatározhatóvá vált számomra a navigációs útvonalak definiálása a képernyők között. Állapotdiagram formájában rajzoltam meg a navigációs gráfot, annak okán, hogy átláthatóbbak legyenek az elágazások. A 20. ábrán látható az elkészült gráf.



20. ábra Navigációs gráf

Mint az minden alkalmazásban megszokott, a hitelesítéshez én is két nézetet alkalmazok: bejelentkezés és regisztráció. A két rész között az átjárás szabadon megtehető. Az alkalmazást elindítva automatikus bejelentkeztetés történik, aminek

sikertelensége esetén a bejelentkező oldal az alkalmazás kiinduló pontja, ellenkező esetben pedig a felhasználó növényeinek listája (az ábrán User's plants). Az alkalmazás többi része sikeres bejelentkezés vagy regisztráció után válik elérhetővé. Ezt a diagramon csak a regisztrációnál jeleztem, sikeres regisztráció után nem kell még külön bejelentkezni is, a bejelentkezésnél pedig egyértelmű, hogy csak helyes azonosítás után lehet navigálni.

Az API-ból származó növények, a felhasználó növényei és a profil képernyőt egy részbe tettem, mivel ezek együttesen képezik az alkalmazás főoldalát. Bármelyik nézetből eljuthatunk a másik kettőre (például a Navigation Bar segítségével). Kilépést követően pedig a bejelentkező képernyő jelenik meg.

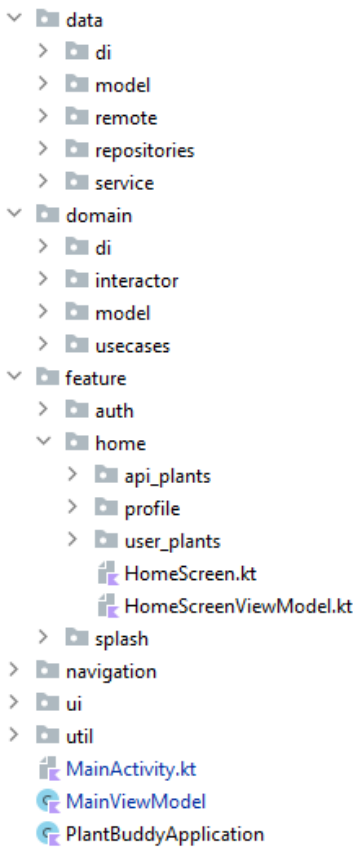
A felhasználói növények és az API növények oldalt is egy-egy külön részbe helyeztem, mert egy ilyen oldalhoz két nézet is tartozik: a lista és a részletes nézet. Egy ilyen komponensre való navigáláskor először a lista nézet válik elérhetővé, ahonnan egy növényre nyomva navigál az alkalmazás annak részletes nézetére. A részletes nézetből pedig csak a lista nézetre van lehetőség visszatérni.

5 Megvalósítás érdekesebb részletei

Ebben a fejezetben bemutatom az elkészült alkalmazást és a megvalósítás érdekesebb részleteit.

5.1 Az elkészült projekt struktúrája

A 20. ábra mutatja be a kialakult mappastruktúrát. Jól kivehető a tervezésnél leírt háromrétegű architektúra. Az adatelérési réteget a data mappa, az üzleti logikai réteget a domain mappa reprezentálja, a feature mappa pedig a magjelenítési réteget foglalja magába. Továbbá megfigyelhetőek az alkalmazott tervezési minták: DI, Repository, Interactor, mindegyik a megfelelő rétegekben. A függőséginjektálás két helyen is látható. Ennek oka, hogy az Interactor-ok az üzleti logikai rétegben vannak és ezeket is injektálhatóvá kell tenni. Így próbáltam minél jobban elkülöníteni azokat a függőségeket, melyek különböző részekben szerepelnek.



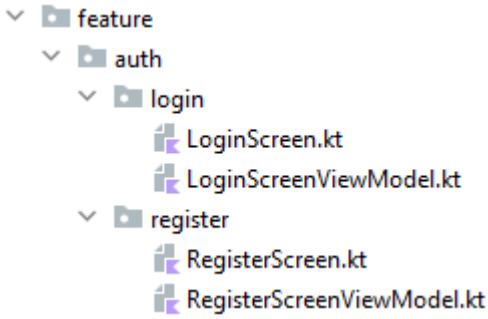
21. ábra Az alkalmazás mappastruktúrája

A model mappákban találhatóak azok az adatosztályok (modellek), melyek az alkalmazás különböző rétegei számára az adastruktúrákat reprezentálják. Az adatelérési rétegben lévő az adatforrásokkal, mint a Firestore adatbázis és a külső API-kkal való kommunikáció során használt adatszerkezeteket tartalmazza. Itt szerepelnek a tervezés során említett DTO-k. Az üzleti logikai rétegben lévő pedig a különböző adatforrásokból származó adatok egyesített, üzleti logika szempontjából releváns reprezentációt tartalmazza.

A ui mappában az olyan újrafelhasználható Composable elemek vannak, melyek megjelennek több képernyőn is.

Az MVVM minta jobban látható a következő ábrán, ahol az auth mappát lenyitva megjelenik a bejelentkező és regisztrációs felület. Mindegyikhez külön View és ViewModel tartozik. Ez az összes többi nézetre is igaz. Illetve a minta bemutatásánál

említettem a Service osztályok használatát. Ezek az adatelérési rétegben helyezkednek el a data mappában.



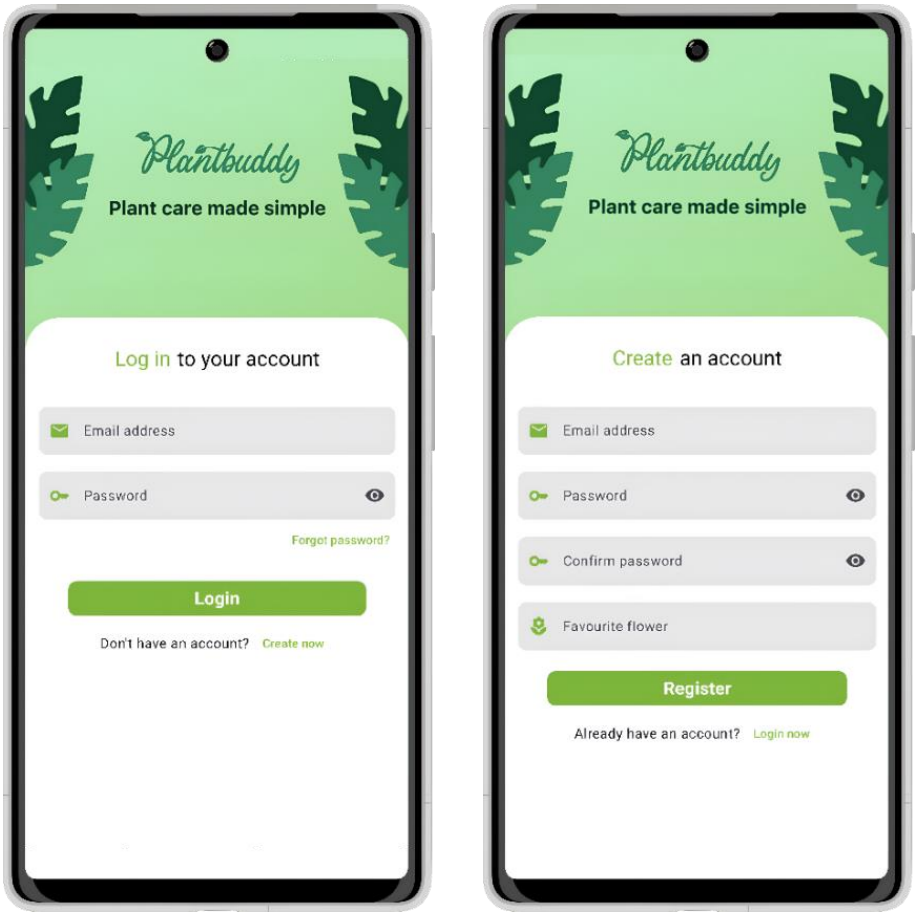
22. ábra MVVM minta egy megjelenése

5.2 Felhasználói felület

Az alkalmazást elindítva először egy töltőképernyő (22.ábra) jelenik meg. Ez csak addig látható, ameddig az automatikus bejelentkeztetés történik és az eredmény alapján lép tovább a megfelelő oldalra.

23. ábra Splash Screen

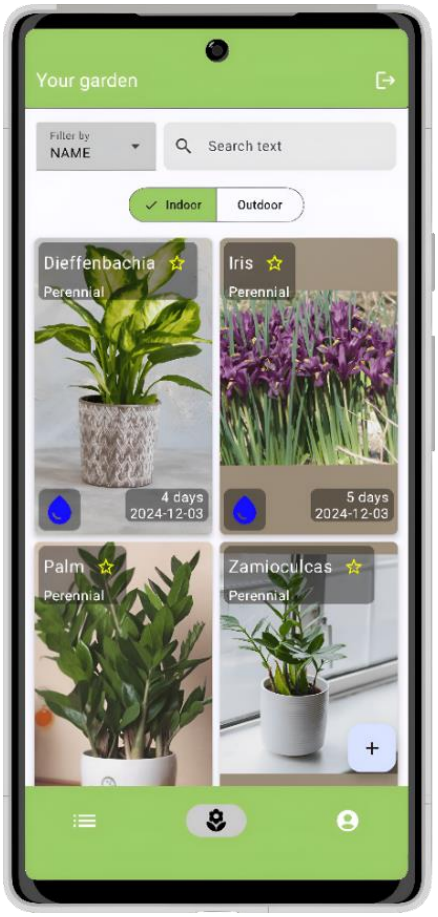
Amennyiben a felhasználó munkamenete lejárt vagy újonnan érkezik az alkalmazásba és nincs még fiókja, úgy a töltőképernyő után a bejelentkeztető felület fogadja. Innen a „Login” gomb alatt lévő „Create now” szövegre nyomva érhető el a regisztráló felület, ahol hasonlóan a gomb alatt található zöld szöveg segítségével lehet visszajutni a bejelentkezéshez. (23. ábra)



24. ábra Bejelentkező felület (balra) és regisztrációs oldal (jobbra)

Sikeres hitelesítést követően a saját növények oldal (24. ábra) jelenik meg. Itt láthatóak a felhasználó növényei egy lista formájában, az utolsó locsolás dátuma szerint növekvő sorrendbe rendezve (a legrégebbi van legelöl). Az oldal tetején lévő szövegbeviteli mező és lenyitható gomb segítségével kereshet a felhasználó. A Kiválasztott opció (név, típus, kedvencek) szerint, a mezőbe gépelve szűri a megjelenő növényeket. Alatta kapott helyet a beltér/kültér szűrő egy gomb formájában, zöld háttérrel jelezve éppen melyik aktív.

Új növényt a jobb alul lévő „+” gomb megnyomásával lehet elmenteni. Navigálni pedig az alsó sávban lévő gombokkal vagy a képernyőt megfelelő irányba húzva lehet az itt elérhető három oldal között.

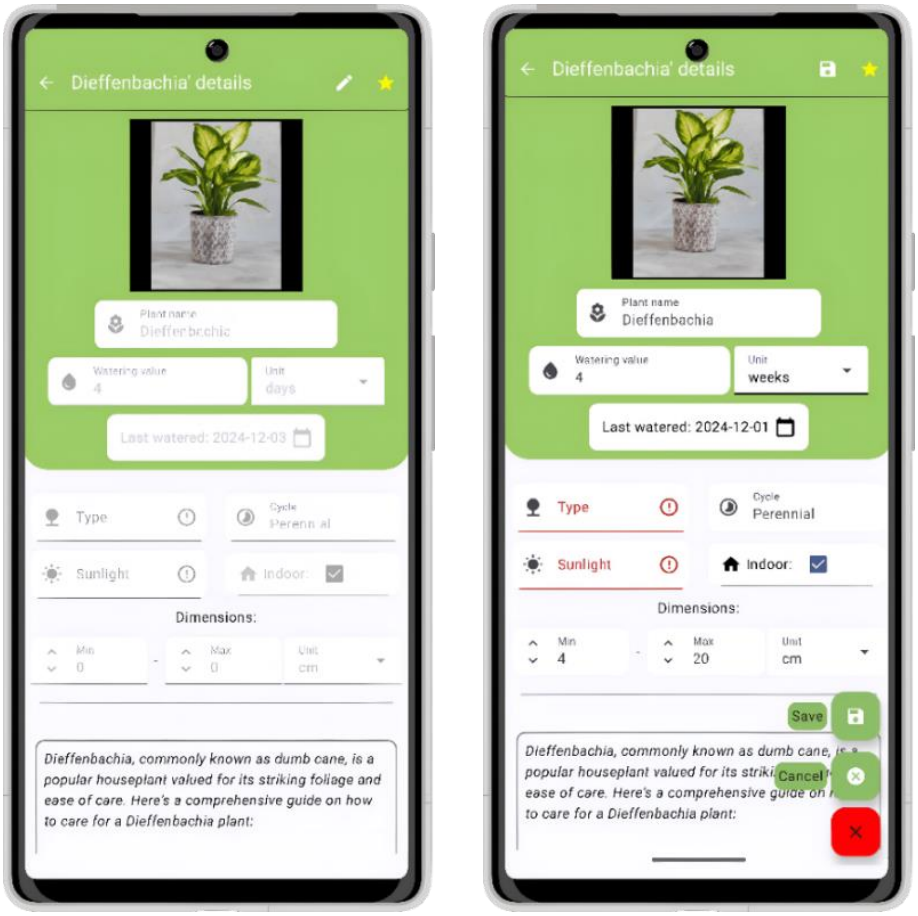


25. ábra Saját növények oldal

A saját növényeknél a listában lévő elemre kattintva lehet annak részletes nézetére navigálni. Itt megjelenik az összes adat a növényről, alatta pedig a mesterséges intelligencia által írt leírás (25. ábra). Először halványabban láthatóak a részletek, azt indikálva, hogy a szerkesztéshez plusz műveletre van szükség. Ez a művelet a felső sávban lévő ceruza ikon megnyomása. Ezután már módosíthatóak az adatok: a képre rányomva új kép készíthető, a locsolás dátumára nyomva egy DatePicker-rel új dátum választható és a lenyitható elemeknél, például az öntözési gyakoriságnál kiválaszthatóak előre meghatározott értékek, mint napok vagy hetek.

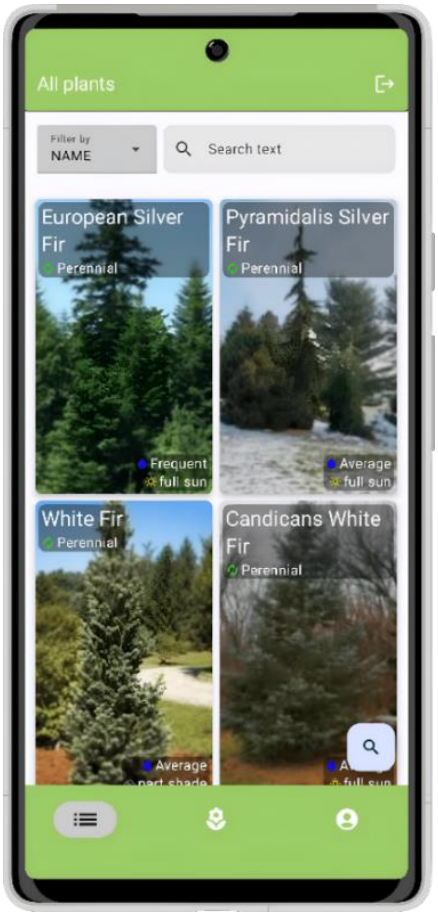
Szerkesztésnél jobb alul megjelenik egy plusz gomb is, amire rákattintva kinyílik felfelé, mint egy menü és elérhetővé válik két opció: mentés és módosítások elvetése (25. ábra, jobb oldali kép). A mentés gomb ilyenkor felül, a ceruza helyén is megjelenik, hatására pedig a módosítás érvénybe lép és kilépünk a szerkesztés módból. Az elvetés

gombbal pedig visszaállítható az aktuális módosítás előtti állapot és úgy lép vissza a program a szerkesztésből.



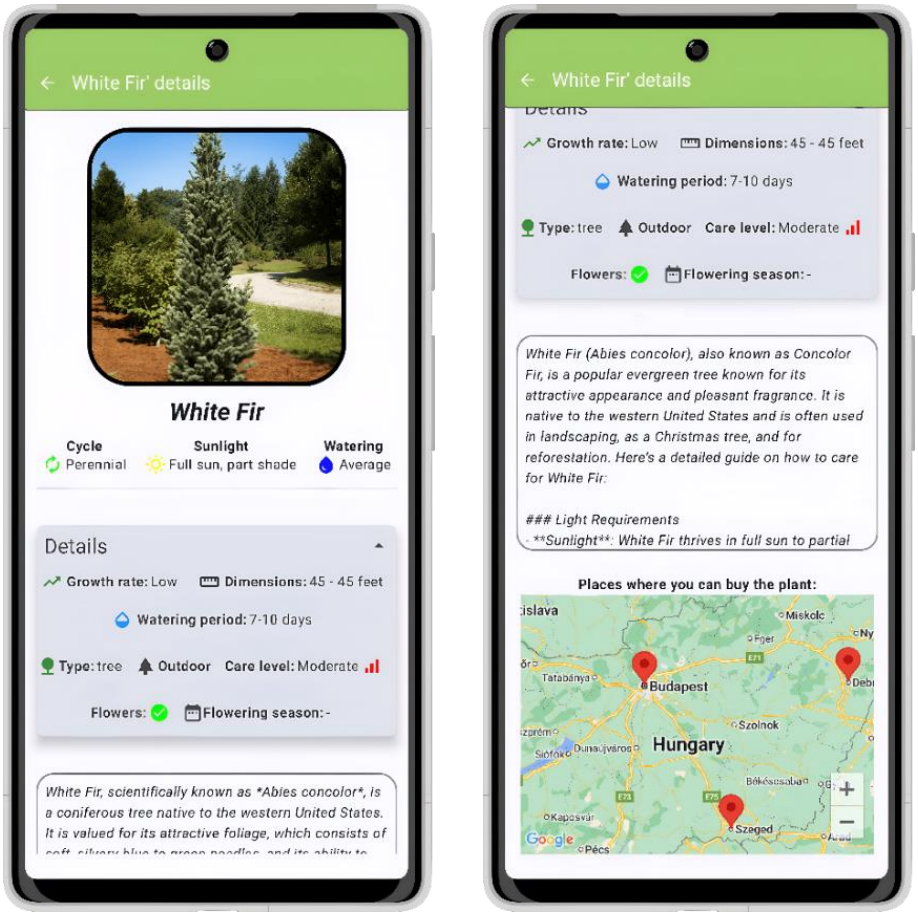
26. ábra Saját növény részletei (szerkesztés mód jobbra)

A saját növények oldalról jobbra húzva (tehát balra haladva) az API-ból kapott növények oldala (26. ábra) válik elérhetővé. Itt hasonló lista formában jelennek meg a növények. A keresés is ugyanúgy működik, mint a saját növényeknél. Továbbá a + gomb helyett itt egy nagyító ikon van jobb alul, aminek hatására megnyílik a kamera és kép készíthető egy növényről, melyet aztán azonosít az alkalmazás.



27. ábra Internetes növények oldal

Szintén a listában egy növény kártyájára való kattintással érhető el annak részletes nézete (27. ábra). Itt megjelenítek pár fontosabb adatot, amit az API-ból kigyűjtöttem. A növény neve alatt általános, a kártyán is látható információk vannak, alatta pedig egy lenyitható elem formájában további részletek láthatóak. Lejjebb görgetve az oldalon elérhető a mesterséges intelligencia által adott leírás, külön görgethető mezőben, alatta pedig a térkép. A térképen lévő piros jelölőkre nyomva megjelennek a hely adatai és egy gombbal elindítható a Google Térkép alkalmazás útvonalkeresője, célállomásként az adott hellyel.

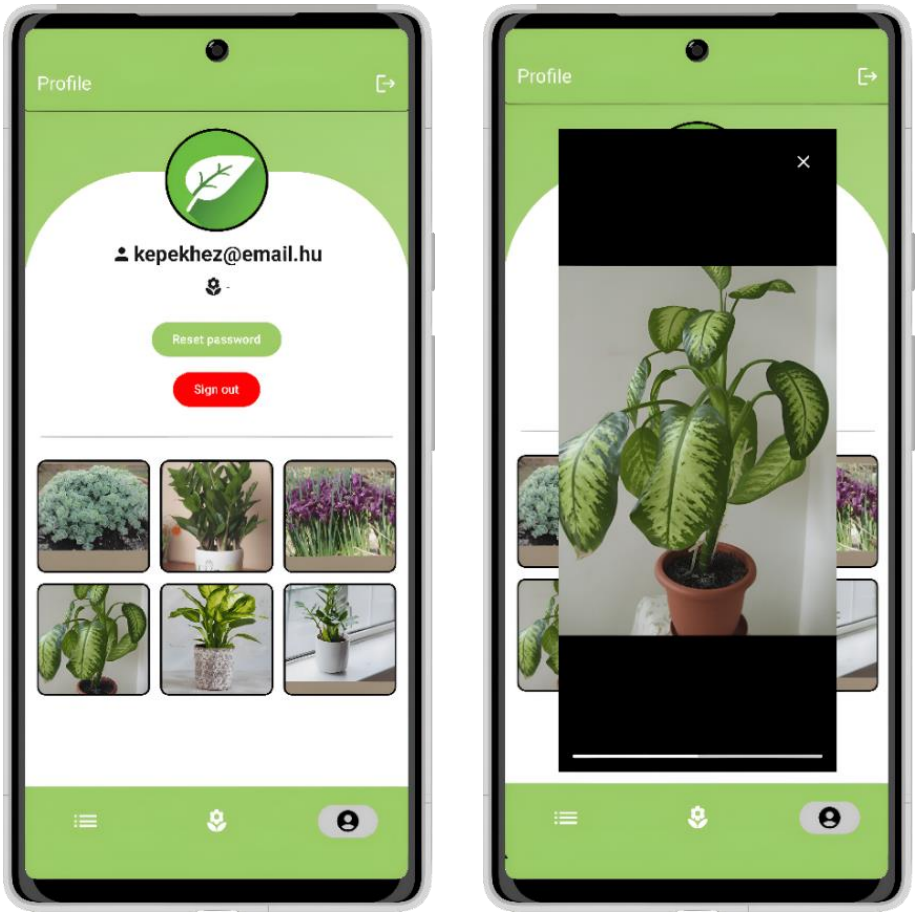


28. ábra Internetes növények részletes nézete

A képalapú azonosítás eredménye egy felugró ablakban jelenik meg (28. ábra). Látható a készített kép és a beazonosított növény neve, mellette a helyes eredmény valószínűségével. Továbbá itt is helyet kapott a mesterséges intelligencia, egy rövidebb leírást szolgáltatva a keresett növényről.

29. ábra Képalapú keresés eredménye

A Profil oldalon (29. ábra) helyet kapott egy jelszóváltoztató gomb (ez a Firebase szolgáltatása, nem az app-ban történik) és egy kijelentkezés gomb. Alatta láthatóak a felhasználó növényeiről készült képek. Az előnézetben a növényhez tartozó legelőször készített kép jelenik meg. Rányomva egy képre elindul a vetítés egy felugró ablakban. Egymás után jelenik meg minden kép a növényről, alul pedig egy töltési csík jelzi, hogy mikor jön a következő kép.



30. ábra Profil oldal (balra) és képösszeállítás megtekintése (jobbra)

5.3 Érdekességek

5.3.1 Műveletek eredménye a felhasználói felületen

Vannak bizonyos műveletek, melyek sikerességét vagy éppen sikertelenségét célszerű közölni a felhasználóval, a felhasználói élmény biztosításához. Ilyen például, ha nem sikerült a bejelentkezés, mert hibás volt a jelszó, nem sikerült betölteni az API-ból a növényeket valamilyen hiba miatt, vagy éppen sikeresen elmentésre került egy új növény, esetleg egy növény adatainak szerkesztése helyesen megtörtént.

Az ilyen események megjelenítésére Snackbar-t vagy Toast-ot használok az app-ban. Ezek jellemzően a képernyő alján megjelenő kis üzenetek. Ilyen üzenetek létrehozására és kezelésére készítettem egy UiEvent nevű osztályt. Ez egy sealed class, mely nagyon hasonlít az enum class-ra, mivel ugyanúgy lehet vele állapotot modellezni, viszont nagyobb testreszabhatóságot biztosít. Két eseményt definiál: sikert és sikertelenséget.

```
sealed class UiEvent {
    object Success: UiEvent()

    data class Failure(val message: UiText): UiEvent()
}
```

A sikertelenség esemény egy UiText típusú paramétert vár. Ez az osztály lehetőséget biztosít dinamikus és statikus szövegek megjelenítésére. A DynamicString egy dinamikusan megadott szöveg, például egy kivétel által adott üzenet. A StringResource egy erőforrás azonosítót tartalmaz, amely egy előre meghatározott üzenetet reprezentál.

```
sealed class UiText {
    data class DynamicString(val value: String) : UiText()
    data class StringResource(@StringRes val id: Int) : UiText()

    fun asString(context: Context): String {
        return when (this) {
            is DynamicString -> this.value
            is StringResource -> context.getString(this.id)
        }
    }
}
fun Throwable.toUiText(): UiText {
    val message = this.message.orEmpty()
    return if (message.isBlank()) {
        UiText.StringResource(StringResources.error_message)
    } else {
        UiText.DynamicString(message)
    }
}
```

A Throwable.toUiText() függvénnyel a kivételeket UiText formátumú üzenetté alakítom. Ha a kivétel üzenete üres, akkor egy alapértelmezett hibaüzenetet használok.

Ezen osztályok segítségével a ViewModel-ekben a különböző műveletek (bejelentkezés, új növény elmentése, növények betöltése) eredményeinek továbbítása egy Channel és Flow kombinációjával történik.

```
private val _uiEvent = Channel<UiEvent>()
val uiEvent = _uiEvent.receiveAsFlow()
```

A Channel egy olyan adatstruktúra, amely lehetőséget ad aszinkron események küldésére. Hasonlít egy csőre vagy üzenetsorra, amelyen keresztül egy vagy több producer (küldő) és egy vagy több consumer (fogadó) kommunikálhat egymással. A Flow egy aszinkron adatfolyam, amely adatokat küldhet egy producer-től a consumer-nek, miközben reagál az adatok érkezésére. Mivel lazy, vagyis lusta módon működik, a Flow csak akkor kezd adatokat kibocsátani, amikor valaki feliratkozik rá, vagyis elkezdi

begyűjteni az adatokat. Segítségével a Channel-ből jövő adatokat aszinkron módon, reaktívan gyűjthetjük be a nézetben.

Például ha hiba történik egy művelet elvégzésekor, az alábbi módon tudom értesíteni a View-t:

```
catch (e: Exception) {
    _uiEvent.send(UiEvent.Failure(e.toUiText()))
}
```

A View-ban ezeket az eseményeket a LaunchedEffect segítségével figyelem és a collect függvénnyel gyűjtöm be őket.

```
LaunchedEffect(key1 = true) {
    viewModel.uiEvent.collect { event ->
        when(event) {
            is UiEvent.Success -> {
                onSuccess()
            }
            is UiEvent.Failure -> {
                scope.launch {
                    snackBarHostState.showSnackbar(
                        message = event.message.asString(context)
                    )
                }
            }
        }
    }
}
```

A LaunchedEffect lefut, amikor a Composabe először jelenik meg. Továbbá a kulcsként kapott érték minden változásánál újból fut. Ezzel a megoldással elérem, hogy a felhasználó valós idejű visszajelzést kapjon az alkalmazásban történő műveletek sikerességéről vagy sikertelenségéről.

5.3.2 Kommunikáció View és ViewModel között

A View felelőssége, hogy reagáljon a felhasználó interakcióira (pl. szövegbeviteli mező tartalmának megváltoztatására), míg a ViewModel felel az állapot kezeléséért és a szükséges logikáért. Ennek megvalósítására létrehoztam a ViewModel-ekhez olyan adatmodell osztályokat, amelyek az alkalmazás egy adott állapotát reprezentálják. Bemutatásához egy rövidebb példát veszek az elkészült alkalmazásból.

A bejelentkezésnél a következő adatmodellt definiáltam a felhasználó által megadott e-mail cím és jelszó, valamint a jelszó láthatóságának kezelésére:

```
data class LoginUserState(
    val email: String = "",
    val password: String = "",
    val passwordVisibility: Boolean = false
)
```

A ViewModel-ben az állapotot MutableStateFlow segítségével tárolom. Ez egy olyan típus a Kotlin Flow könyvtárban, amely lehetővé teszi egy állapot értékének folyamatos követését és frissítését, különösen mikor gyorsan és dinamikusan szükséges reagálni a változásra a felhasználói felületen. Ezt a mutable objektumot StateFlow segítségével elérhetővé teszem a View számára.

```
private val _state = MutableStateFlow(LoginUserState())
val state = _state.asStateFlow()
```

A View pedig a collectAsStateWithLifecycle segítségével követi az aktuális állapotot és leképezi a felhasználó felületre. A by kulcsszó segítségével közvetlenül lehet hivatkozni az állapot mezőire (pl. state.email), és a megfigyelő automatikusan frissül, amikor változás történik.

```
val state by viewModel.state.collectAsStateWithLifecycle()
```

Szükség van még a felhasználó interakcióira való reagálásra, például ha egy szövegbeviteli mező értéke változik. A View értesül az interakcióról, viszont az elvégzendő logika a ViewModel feladata. Ezt az előző részben írt eseményekhez nagyon hasonló módon oldom meg. Egy sealed class formájában definiálom az adott View-hoz tartozó eseményeket. A példában ez a LoginUserEvent:

```
sealed class LoginUserEvent {
    data class EmailChanged(val email: String): LoginUserEvent()
    data class PasswordChanged(val password: String): LoginUserEvent()
    object PasswordVisibilityChanged: LoginUserEvent()
    object SignIn: LoginUserEvent()
}
```

Ennek segítségével a View az interakcióhoz megfelelő eseményt elküldi a ViewModel-nek, az pedig frissíti a state-et a MutableStateFlow segítségével, amely biztosítja, hogy az új állapotot a View azonnal megkapja.

```
fun onEvent(event: LoginUserEvent) {
    when(event) {
        is LoginUserEvent.EmailChanged -> {
            val newEmail = event.email.trim()
            _state.update { it.copy(email = newEmail) }
        }
     }
}
```

5.3.3 Navigáció

Az alkalmazásban a navigációhoz a Compose Navigation könyvtárát használom. Egy sealed class formájában definiáltam az útvonalakat, így azok könnyen kezelhetők. Az egyes képernyők egy-egy objektumként szerepelnek.

```
sealed class Screen(val route: String) {
    object Home: Screen("home")
    object Profile: Screen("profile")
    object Login: Screen("login")
    object Register: Screen("register")
    object Splash: Screen("splash")
    object UserPlantDetails: Screen("userPlantDetails/{plantId}")
    object ApiPlantDetails: Screen("apiPlantDetails/{plantId}")
}
```

Ezután létrehoztam a navigációs gráfot, mely egy Composable függvény. Feladata az útvonalakat karbantartása. Segítségével dolgozom fel a navigációs eseményeket.

Ennek a megközelítésnek az egyik előnye, hogy kiszervezhető a komponensekből a navigáció logikája, így az egy helyre kerül és nincs plusz függőségre szükség a nézetekben. Például a bejelentkező képernyőről két helyre lehet navigálni, a regisztrációs és saját növények oldalra. Azért, hogy ne legyen szüksége NavHostController-re a View-nak, amivel a navigációt lehet végezni, két callback formájában adom át a megfelelő logikát a függvény paraméterében. A bejelentkezés gombra kattintva, ha sikeres a hitelesítés, akkor navigál az alkalmazás a saját növények oldalra. Ez az onSuccess paraméter. A View-ban a gomb onClick paraméterének egyszerűen átadom ezt az onSuccess függvényt, így maga a nézet közvetlen nem végez semmilyen logikát.

```
composable(Screen.Login.route) {
    LoginScreen(
        onSuccess = {
            navController.navigate(Screen.Home.route) {
                popUpTo(Screen.Login.route) { inclusive = true }
            }
        },
        onRegisterClick = {
            navController.navigate(Screen.Register.route)
        }
    )
```

A növények részleteire vezető útvonalaknál megfigyelhető, hogy az útvonal neve mellett egy argumentumra is szüksége van, ami az adott növény azonosítója. Ezzel a megközelítéssel nincs szükség konkrét Plant objektumot átadni a nézetek között, mivel a részletes nézet az útvonalból kapott ID alapján le tudja azt kérdezni az API-ból vagy Firestore-ból. Ez egyszerűbb navigációt eredményez, könnyebben karbantarthatóvá válik

a kód. Tesztelés szempontjából is előnyt jelent, hogy nem kell külön üres objektumokat létrehozni csak a navigációhoz.

5.3.4 API elérés

A felhasznált API-k eléréséhez interfészeket hoztam létre, melyek tartalmazzák a végpontokhoz szükséges függvényhívásokat. Ezeket az interfészeket a Retrofit könyvtár segítségével implementáltam. Ez automatikusan kezeli a *HTTP* kérések és válaszok feldolgozását. Például a Perenual API-hoz a következő interfész társul:

```
interface PerenualApi {
    @GET("species-list")
    suspend fun getAllPlants(
        @Query("page") page: Int,
        @Query("per-page") perPager: Int,
        @Query("key") apiKey: String = ApiAccessKey
    ): Response<PlantResponse>

    @GET("species/details/{id}")
    suspend fun getPlantDetailById(
        @Path("id") id: Int,
        @Query("key") apiKey: String = ApiAccessKey
    ): Response<PlantWithDetailsDto>
}
```

Emellett minden API-hoz létrehozok egy külön Retrofit instance-t is, melyek tartalmazzák az adott API eléréséhez szükséges beállításokat. Ezek az elérést biztosító alap URL, a konverter a *JSON* formátumú adatok feldolgozására és persze a hozzá tartozó interfész. A példánál maradva ez a következőképpen valósul meg:

```
object PerenualRetrofitInstance {
    private const val BASE_URL = "https://perenual.com/api/"

    val api: PerenualApi by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(MoshiConverterFactory.create())
            .build()
            .create(PerenualApi::class.java)
    }
}
```

A lazy kulcsó használatával érem el, hogy az objektum csak akkor jöjjön létre, amikor először meghívják.

Ilyen interfészek és Retrofit instance-ok alkalmazásával magát az API-t használó osztályok (Service-ek, Repository-k) számára könnyen kezelhetővé válnak az egyes kérések. Egy egyszerű változó segítségével elindíthatóak azok.

```
private val api: PerenualApi
…
val response = api.getAllPlants(page, perPage)
if (response.isSuccessful) {
    Result.success(response.body()?.data ?: emptyList())
} else {
    Result.failure(ApiError.HttpError(response.code(),
response.message()))
}
```

5.3.4.1 Kulcsok tárolása

Az API-k eléréséhez szükség van kulcsra, melyet a szolgáltatóktól lehet igényelni. Ennek a kulcsnak fontos a biztonságos tárolása, mivel használatával bárki indíthatna kérést az én nevemben. A projektet elérhetővé tettem a GitHub felületén, ahol publikálás után bárki láthatja a forráskódot, ezért a kulcsokat közvetlen a kódban tárolni veszélyes lenne. Így a projekt local.properties fájljába helyeztem el az összes szükséges kulcsot, mert ezt a verziókezelő figyelmen kívül hagyja.

Ehhez a modul szintű build.gradle fájlban létrehoztam buildConfigField-eket, melyekbe az app eltárolja a local.properties-ből kiolvasott kulcsokat.

```
val properties = Properties()
properties.load(project.rootProject.file("local.properties").inputStream()
)

buildConfigField("String", "AIML_API_KEY",
"\"${properties.getProperty("Aiml_ApiKey")}\"")
buildConfigField("String", "KINDWISE_API_KEY",
"\"${properties.getProperty("Kindwise_ApiKey")}\"")
buildConfigField("String", "PERENUAL_API_KEY",
"\"${properties.getProperty("Perenual_ApiKey")}\"")
```

Ezeket a mezőket pedig a kódban egyszerűen lehet használni, a megadott nevek szerint:

```
val ApiAccessKey = BuildConfig.PERENUAL_API_KEY
```

5.3.4.2 Hibák kezelése

Az API hívások során felmerülő hibák kezelésére saját osztályt használok, melyben a legáltalánosabb lehetőségeket vettem fel:

- NetworkError: Hálózati probléma, például nincs internetkapcsolat.
- HttpError: *HTTP* válaszkódok kezelésé, mint a 404-Not Found vagy az 500-Internal Server Error.
- UnknownError: Bármilyen egyéb, nem várt hiba.

```
sealed class ApiError: Exception() {
    data class NetworkError(override val message: String): ApiError()
    data class HttpError(val code: Int, override val message: String):
ApiError()
    data class UnknownError(override val message: String): ApiError()
}
```

Ennek segítségével könnyebben kezelhető az adott hiba a magasabb rétegekben, illetve a felhasználót is egyszerűbben lehet tájékoztatni annak okáról.

5.3.5 A szükséges engedélyek kezelése

Android alkalmazások során bizonyos funkciók eléréséhez engedélyekre van szükség, melyek különböző típusokba sorolhatók. Közülük a futásidejű engedélyekre térek ki. Ezek az engedélyek ahogy a neve is sugallja, futásidőben kérhetőek el a felhasználótól. Az ilyen jogosultságokra való rákérdezéssel körültekintőnek kell lenni az alkalmazásban, mivel jelentősen ronthatják a felhasználói élményt. Ezalatt arra gondolok, ha mondjuk egy appnak sok ilyen engedélyre van szüksége és ezeket mind egyből a felhasználó arcába nyomja belépés után, magyarázat nélkül, az hamar elveszi a kedvet a használatától. Az ilyen esetekre több best practice is elérhető

Az alkalmazásom két ilyet használ: képek készítéséhez a telefon kamerájához való hozzáférésre van szükség, illetve az értesítések fogadását is külön jóvá kell hagyni. Én azt a megoldást választottam, hogy csak akkor kérem el az engedélyt, amikor az adott funkciót először használná a felhasználó. Ez a kamera esetében úgy néz ki, hogy amikor rányom a felhasználó az adott gombra (növény azonosítás vagy kép készítése növény elmentésénél), akkor a rendszer megnézi, hogy el lett-e már kérve az engedély és ha nem, akkor teszi fel a kérdést. Ebbe beletartozik az is, ha a felhasználó nemmel válaszol, többször nem kérdezi meg az alkalmazás.

Ezt a rememberLauncherForActivityResult segítségével érem el. Ez létrehoz egy launcher-t, amely egy konkrét tevékenységet indít el és képes megjegyezni az eredményét. A RequestPermission-el adom meg számára, hogy engedély elkéréséhez szeretném használni.

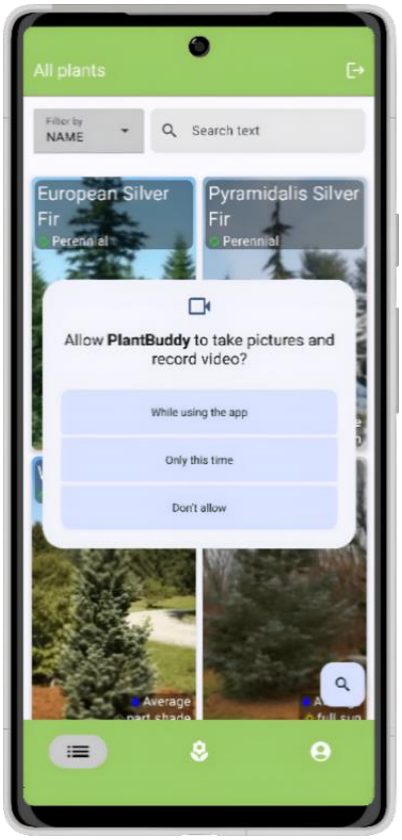
```
val permissionLauncher = rememberLauncherForActivityResult(
    contract = ActivityResultContracts.RequestPermission()
) { isGranted ->
    if (isGranted) {
        …
    } else {
        Toast.makeText(context, "Camera permission denied",
Toast.LENGTH_SHORT).show()
```

```
    }
}

if(launchCamera) {
    val cameraPermission = Manifest.permission.CAMERA
    when (PackageManager.PERMISSION_GRANTED) {
        ContextCompat.checkSelfPermission(context, cameraPermission) -> {
            …
        }

        else -> {
            permissionLauncher.launch(cameraPermission)
        }
    }
}
```

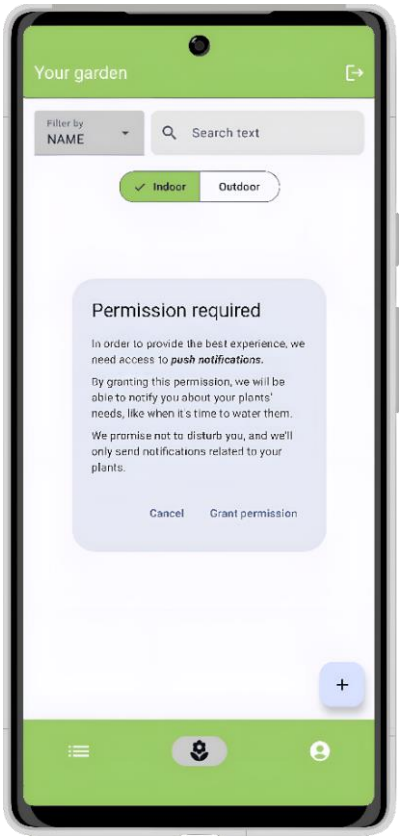
A ContextCompat.checkSelfPermission segítségével lehet leellenőrizni, hogy rendelkezésre áll-e az engedély. Ha nem akkor el kell indítani a launcher-t.



31. ábra Kamera engedély elkérése

Az értesítések engedélyének elkérésével viszont bajban voltam, mivel ez nem köthető konkrét funkcióhoz. Mondván, hogy ez egy fontos funkciója az alkalmazásnak amellett döntöttem, hogy belépés után teszem ezt meg. Az előzőekben említettem, hogy ez nem egy jó ötlet, viszont mivel csak egyetlen engedélyről van szó, mégis ezt találtam a legjobbnak ennél a kis alkalmazásnál.

Amikor a felhasználó megtagad egy ilyen hozzáférést, megjelenítek egy magyarázó szöveget, hogy miért van szükség az engedélyre, ahonnan vissza lehet térni a megadáshoz. Ezt csak egyszer teszem meg, ha a magyarázó szövegre is nem a válasz akkor az nem jelenik meg többször.



32. ábra Engedély tiltásakor megjelenő indoklás

5.3.6 Képes összeállítások

A Firebase Storage struktúrájának ismertetésénél a **4.2.1 fejezetben** megfigyelhető, hogy a növények nincsenek külön felhasználókhoz rendelve, hanem egyszerűen össze vannak ömlesztve. Azért döntöttem emellett a tárolási mód mellett, mert amikor a Firestore-ba elmentésre kerül a növény, kap egy egyedi azonosítót, mellyel később aztán megkülönböztethető a többitől. Az adatbázisban már eleve felhasználókhoz van rendelve az összes növény és az éppen aktuális felhasználó növényeinek lekérdezésére egyébként is szükség van a felhasználó növényei oldalhoz. Ennek köszönhetően már rendelkezésre állnak az azonosítók, melyek segítségével aztán könnyen meg lehet keresni a Storage-ben a képeiket és nincs szükség kétszer is felhasználókhoz rendelni őket. Ezt úgy oldottam meg, hogy feltöltéskor a növény azonosítóját adom meg a létrejövő mappa nevének.

Egy sok felhasználóval rendelkező, nagy alkalmazásban ez nem feltétlen egy jó megközelítés, mert például ha adatbázis visszaállítására van szükség, akkor a képeket egyesével kell átnézni, hogy kihez tartoznak. Viszont az én alkalmazásom méretéhez képest ezzel a megoldással nincs túlbonyolítva feleslegesen a struktúra és gyorsabb az adatelérés.

A logika megvalósításához egy növény elmentésekor az azt végző függvény siker esetén visszaadja a létrejött azonosítót, egyébként pedig nullt. A képet ezután töltöm fel a Storage-ba, mivel csak akkor áll rendelkezésre az ID, majd frissítem a növény képének URL-jét az adatbázisban a Storage által kapott linkre. Ezzel lekezelem azt a hibalehetőséget, hogy ha esetleg nem sikerülne a mentés, akkor nem kerül feltöltésre kép.

A Repository Flow formájában biztosítja az aktuális felhasználó növényeit:

```
val plants: Flow<List<UserPlant>> = authService.currentUser.flatMapLatest
{ user ->
    if (user == null) flow { emit(emptyList()) }
    else currentCollection(user.id)
        .snapshots()
        .map { snapshot ->
            snapshot
                .toObjects<FirebasePlant>()
                .map {
                     it.asUserPlant()
                }
        }
}

companion object {
    private const val USER_COLLECTION = "users"
    private const val PLANT_COLLECTION = "plants"
}

private fun currentCollection(userId: String) =
firestore.collection(USER_COLLECTION).document(userId).collection(PLANT_CO
LLECTION)
```

A Profil ViewModel-jében pedig az Interactor-on keresztül elérve a Repository-t, kigyűjthetőek a növények képei a Storage-ból:

```
firestoreInteractor.getPlants()
    .collect { plants ->
        val imageMap = mutableMapOf<String, List<String>>()
        plants.forEach { plant ->
            val imageResult =
storageInteractor.getImagesForPlant(plant.id)
            if (imageResult.isSuccess) {
                imageMap[plant.id] = imageResult.getOrDefault(emptyList())
            }
        }
    }
```

Ahol egy Map-ben tárolom el az egyes növényeket és a hozzájuk tartozó képeket. A ViewModel-ben a collect operátort használva, azonnal értesül a Repository-ban lévő Flow változásairól és frissíteni tudja annak megfelelően a Map-ben tárolt értékeket. Az ebben a formában való tárolással pedig a View-ban egyszerűen meg tudom jeleníteni az elemeket egy LazyVerticalGrid-ben.

```
LazyVerticalGrid(
    …
) {
    profileState.images.forEach { (id, imageList) ->
        if (imageList.isNotEmpty()) {
            item {
                PlantImageCell(
                    imageUrl = imageList.first(),
                    onClick = {
                        selectedImages = imageList
                        isSlideshowVisible = true
                    }
                )
            }
        }
    }
}
```

5.3.7 Értesítések

Az értesítések megalkotásakor több mindenről is döntenem kellett. A cél az volt, ha egy növény öntözési ideje eljött, akkor arról legyen push notification, amire a felhasználó reagálhat és egy gomb megnyomásával egyszerűen beállíthatja a locsolás dátumát. A locsolásnak akkor van itt az ideje, ha a növény utolsó locsolásának dátumához hozzáadva a frekventáltságot kisebb időpontot kapunk a mai dátumnál. Az értesítések kiküldésére a backend szolgál, ami az én esetemben a Firebase. Az automatizálásra a Functions funkció ad lehetőséget, mely egy job-ot futtat Google Cloud Scheduler segítségével. Erre a feladatra én egy script-et készítettem JavaScript nyelven, amelyet aztán feltöltöttem a Firebase-re.

Először azt kellett meggondolnom, hogy mikor futtassam a script-et. Túl sokszor nem lehet megtenni, mivel a nagymennyiségű erőforráshasználatért már fizetni kell. Ez nagyjából napi 3-4 futást jelent. Az ember átlagosan a hétből 5 napot dolgozik és reggel 8-tól délután 5-ig nem tartózkodik otthon. Ebből kiindulva vagy korán reggel van értelme az értesítésnek, vagy pedig este. Így napi egyszeri értesítésküldés mellett döntöttem az este folyamán, a Google Scheduler-t ennek megfelelően állítottam be.

Ezután az értesítések küldésének módját kellett megfontolnom. Ha szeretném az előbbiekben említett funkciót megtartani (locsolás beállítása gombnyomásra), akkor minden növényhez külön értesítést kell küldeni. Viszont ez sok növény esetén zavaró lesz és átláthatatlanná teszi az üzeneteket. Így arra az elhatározásra jutottam, hogy ha egynél több növény van, amit az adott napon meg kéne locsolni, akkor az összes belekerül egyetlen értesítésbe és nem lesz rajta gomb. Egyetlen növény esetén pedig megjelenik a locsolás gomb. Ezt a logikát a script fájlban valósítottam meg és egy kapcsoló beállításával jelzem az alkalmazásnak, hogy éppen milyen értesítést jelenítsen meg.

Amennyiben kerül gomb az értesítésre, egy Intent-et küldök az Activity számára. Az Intent egy olyan osztály, melynek segítségével különböző komponensek közötti kommunikáció vagy konkrét művelet kezdeményezés valósítható meg.

```
val notificationBuilder = NotificationCompat.Builder(this,
"watering_channel")
    .setSmallIcon(R.drawable.ic_notification)
    .setContentTitle(title)
    .setContentText(messageBody)
    .setPriority(NotificationCompat.PRIORITY_HIGH)
    .setAutoCancel(true)

if (isSinglePlant && plantId != null) {
    val wateringIntent = Intent(this, MainActivity::class.java).apply {
        action = "WATER_NOW"
        putExtra("PLANT_ID", plantId)
    }

    val wateringPendingIntent: PendingIntent = PendingIntent.getActivity(
        this, 0, wateringIntent, PendingIntent.FLAG_UPDATE_CURRENT or
PendingIntent.FLAG_IMMUTABLE
    )

    notificationBuilder
        .addAction(
            R.drawable.ic_watering_notification,
            "Just watered",
            wateringPendingIntent
        )
}
```

Az Intent-ben átadom a növény azonosítóját is. Az Activity-ben pedig figyelek a beérkező Intent-ekre és frissítem az azonosító segítségével a növény utolsó locsolási dátumát.

```
override fun onNewIntent(intent: Intent) {
    super.onNewIntent(intent)
    handleIntentAction(intent)
}

private fun handleIntentAction(intent: Intent) {
```

```
    val plantId = intent.getStringExtra("PLANT_ID")
    when (intent.action) {
        "WATER_NOW" -> {
            plantId?.let { viewModel.waterNow(it) }
        }
    }
}
```

Ezzel a megoldással félig-meddig megőriztem eredeti szándékomat az értesítésekkel kapcsolatban és figyelek a felhasználóbarát viselkedésre is.

5.4 Tesztelés

A tesztelés során fontosnak tartottam, hogy külön figyelmet fordítsak a felhasználói felület és az üzleti logika elkülönített tesztelésére. A felhasználói felület tesztelését manuálisan végeztem, hogy ellenőrizhessem a megfelelő interakciók hatására az alkalmazás helyes megjelenítését és működését. Az üzleti logika tesztelésére pedig egységteszteket írtam, amelyek lehetővé tették az egyes komponensek izolált tesztelését.

5.4.1 Manuális tesztelés

A fejlesztés során egy-egy olyan funkció megvalósítása után, amely hatással volt a felhasználói felületre, folyamatosan ellenőriztem a működést kézzel. Ehhez elsősorban az Android Studio beépített emulátorát használtam. Így könnyen megbizonyosodhattam a kialakított felhasználói élményről. Figyeltem, hogy az interaktív elemek hogyan működnek, nem túl bonyolult-e a használatuk, megfelelően jelenik-e meg minden funkciójuk. Ellenőriztem a többi elem megjelenését is, például az ikonokra beállított méret nem túl nagy vagy esetleg túl kicsi, a szövegek olvashatóságát illetve, hogy nem takarják-e el egymást a különböző elemek. Így hamar ki tudtam szűrni a felmerülő hibákat, illetve olyan problémák is fényre derültek, melyekre fejlesztés során nem gondoltam, csak használattal jöttek elő. Ilyen volt például egy gomb rossz elhelyezése, mellyel kényelmetlen volt a használata, vagy a görgethető tartalomba helyezett, külön görgethető tartalom együttműködésének kezelése.

Nagyobb előrehaladások alkalmával kipróbáltam az alkalmazást saját eszközön is, hogy ne csak monitoron felnagyítva lássam és ne egérrel interaktáljak. Ezzel többek között az olyan eseteket tudtam kiküszöbölni, mikor túl közel volt egymáshoz két elem és ujjal nehéz volt pontosan eltalálni az egyiket. Ez segített finomítani a felhasználói felületet és a használati élményt.

Az alkalmazás méretéből adódóan én a manuális megközelítés mellett döntöttem, azonban erre a célra készíthetőek úgynevezett instrumentális tesztek is, melyek az alkalmazás valós környezetben történő működésének ellenőrzésére szolgálnak. Ezek az alkalmazást fizikai eszközön vagy emulátoron futtatva működnek, így képesek a felhasználói felületet is tesztelni. Segítségükkel szimulálhatóak felhasználói interakciók és automatizálhatóak a UI tesztek.

5.4.2 Egységtesztelés

Az egységtesztelés olyan tesztelési eljárás, amelynek célja a program különböző egységeinek (pl. függvények, metódusok) izolált tesztelése annak biztosítására, hogy azok helyesen működjenek. A tesztek elkészítését a JUnit[20] keretrendszer segítségével végeztem. Az alábbiakban egy példán keresztül bemutatom használatát.

Minden teszt futása előtt lefut a *@Before* annotációval ellátott függvény. Itt el lehet végezni a szükséges objektumok inicializálását, így nem kell ezt minden metódusban külön megtenni. A függőségeket mock objektumokként lehet átadni, melyek valódi objektumok helyett szimulált viselkedést biztosítanak. Így elkerülhető valós adatbázis vagy külső rendszerek használata.

```
private lateinit var mockFirebaseAuth: FirebaseAuth
private lateinit var authService: FirebaseAuthService

@Before
fun setUp() {
    mockFirebaseAuth = mockk(relaxed = true)
    authService = FirebaseAuthService(mockFirebaseAuth)
}
```

A tesztek felépítése egységes:

- Az első az ***Arrange*** fázis, ahol a tesztelendő környezet előkészítése zajlik. Itt történik az adott teszthez szükséges változók létrehozása, valamint a mock osztályok funkcionalitásainak beállítása.
- A következő fázis az ***Act***, ahol futtatásra kerül a tesztelendő funckió a létrehozott környezetben.

[20] https://junit.org/junit4/

- Végül az ***Assert*** fázis jön, ahol a teszt eredményének értékelése valósul meg. Ez az elvárt és kapott eredmény összehasonlításával történik, például helyes-e a működés a várt eredmény alapján vagy a vártaknak megfelelően kivétel keletkezik.

A példában szereplő teszt célja annak ellenőrzése, hogy az AuthService-ben lévő currentUser Flow megfelelően bocsát ki egy User objektumot amikor változik a bejelentkezés állapota.

```
@Test
fun `currentUser Flow should emit User when state changes`() = runTest {
    val authStateListenerSlot = slot<FirebaseAuth.AuthStateListener>()

    val user = mockk<FirebaseUser> {
        every { uid } returns "123"
        every { email } returns "test@example.com"
    }

    every {
mockFirebaseAuth.addAuthStateListener(capture(authStateListenerSlot)) }
answers {

authStateListenerSlot.captured.onAuthStateChanged(mockFirebaseAuth)
    }
    every { mockFirebaseAuth.currentUser } returns user

    val flow = authService.currentUser
    val emissions = flow.take(1).toList()

    assertEquals(1, emissions.size)
    assertEquals(User("123", "test@example.com"), emissions.first())
}
```

Az Assert során létrehozok egy FirebaseAuth állapotfigyelőt és a tesztelt metódus által később visszaadandó User objektumot, ahol beállítottam, hogy a uid és email mezők milyen adatot adjanak vissza. Továbbá beállítom, hogy az addAuthStateListener metódus a capture(authStateListenerSlot) válaszban az előzőekben létrehozott állapotfigyelőt tárolja el. Ez biztosítja, hogy amikor a bejelentkezési állapot változik, a listener aktiválódik, és elindul a megfelelő tesztelési logika.

Az Act fázisban lekérdezem az authService Flow típusú currentUser változóját és a take(1) operátorral megszerzem az első kibocsátott értéket.

Kiértékelés során először ellenőrzöm, hogy csak egyetlen érték került-e kibocsátásra, majd pedig összevetem a kapott felhasználó adatait az általam várt értékekkel.

6 Összefoglalás

6.1 Az elkészült munka értékelése

Szakdolgozatomban egy olyan alkalmazást készítettem, mely segít felhasználóinak a növényeik gondozásával. Fő célom az volt, hogy értesítést küldjön az öntözés idejéről és amolyan kisokosként, mesterséges intelligenciával kiegészülve információval szolgáljon a növényekről.

Sok alapvető, de fontos dolgot tanultam meg az Android fejlesztéssel kapcsolatban a projekt során, az architektúrális felépítésen át az állapotkezelés és hálózati kérések megvalósításán keresztül egészen a tesztelésig. Ilyen volt a Jetpack Compose is, mely számomra újdonság volt. Nem dolgoztam vele ezelőtt és szerettem volna megismerni, ami jó döntésnek bizonyult. Már látom, miért lehet ez a jövő technológiája az Android alkalmazásfejlesztés terén. Továbbá az eddigiekhez képest mélyrehatóbban tudtam foglalkozni a Firebase-el mint backend szolgáltatással. Kipróbáltam sok funkcióját és használtam is ezeket a fejlesztés alatt.

Megtapasztaltam, hogy milyen fontos a megfelelő tervezés egy nagyobb projektnél. A tervezési fázisban megtanultam hogyan kell strukturálni az alkalmazást, hogyan kell megtervezni az architektúrát az átláthatóság és a tovább fejleszthetőség érdekében, valamint hogyan kell prioritásokat felállítani az egyes funkciók megvalósítása során.

Munkám során létrehoznom egy, a specifikációnak megfelelő alkalmazást. Sikeresen megvalósítottam az alapvető funkciókat. Lehetőség van növényeket elmenteni és szerkeszteni őket, illetve képeket készíteni hozzájuk., melyekből az app montázsokat készít. Egyéb növények között is lehet keresni, részletes információkat gyűjtve róluk és térképen megnézni hol kaphatóak. A mesterséges intelligencia minden növényhez ad tanácsot egy leírás formájában. Végül a képalapú keresés is megvalósult és persze képes értesítéseket küldeni az alkalmazás a locsolás esedékességéről.

Összességében élveztem a dolgozat és a szoftver elkészítését. Bízom benne, hogy hasznosítani tudom a továbbiakban a megszerzett tudást és tapasztalatot.

6.2 Továbbfejlesztési lehetőségek

Elsősorban a saját backend készítését emelném ki továbbfejlesztésként. Ezzel megnyílnának olyan lehetőségek, mint érzékelők használata a növényekhez, mellyel megfigyelhető lenne a talaj nedvességtartalma és tápanyagtartalma. Így a jelenlegi egyszerű, időhöz kötött értesítéseket le lehetne cserélni, hogy pontosabb adatok alapján kerüljenek kiküldésre. Illetve nem csak vízigényről lehetne figyelmeztetni a felhasználókat, hanem az említett tápanyagtartalomról és egyéb tulajdonságokról is.

Emellett lehetne közösségi funkciókat is beépíteni az alkalmazásba, mint növények megosztása egymással. Az egy háztartásban élők így könnyedén tudnák kezelni a közös növényeiket. Illetve amikor a tulaj elutazik és megkér valakit, hogy gondozza a növényeit amíg távol van, akkor egy ilyen funkcióval egyszerűen csak megoszthatná azokat és a másik egyszerűen megnézheti mikor voltak utoljára locsolva és milyen igényeik vannak.

Jelenleg az alkalmazás csak Android platformra érhető el. Ezt ki lehetne bővíteni további operációs rendszerekre is. Ma már elérhetőek multiplatform fejlesztési lehetőségek, mint a Flutter, melyekkel egyszerre lehet több platformra fejleszteni. Újdonság ebben a témakörben a Kotlin Multiplatform[21] (KMM), amely egyre nagyobb figyelmet és népszerűséget kapott az elmúlt években. A 2024-es Google I/O eseményen a Google hivatalosan bejelentette a Kotlin Multiplatform támogatását az Android fejlesztésében, mely lehetővé teszi az üzleti logika megosztását mobil, webes, szerver és asztali platformok között. Illetve egyre több eszköz és könyvtár (például a Room, Lifecycle és ViewModel) válik elérhetővé Kotlin Multiplatform támogatással.

[21] https://kotlinlang.org/docs/multiplatform.html

7 Irodalomjegyzék

[1] M. V. Attila, „A növények jelentősége,” 26 január 2014. [Online]. Available: https://molnar-v-attila.blogspot.com/2014/01/a-novenyek-jelentosege.html.

[2] F. Pearce, „Rivers in the Sky: How Deforestation Is Affecting Global Water Cycles,” in *Yale School of the Environment*, New Haven, Connecticut 06520, 2018.

[3] F. Andrea, „Zöldebben az élet: miért jó, ha vannak növényeink?,” [Online]. Available: https://www.lakaskultura.hu/kert/zoldebben-az-elet-miert-jo-ha-vannak-novenyeink/.

[4] D. C. Sándor és D. S. Attila, „Jóbarát vagy technoördög?,” [Online]. Available: https://mipszi.hu/cikk/161015-jobarat-vagy-technoordog.

[5] „Nagy nyelvi modellek (LLM): Teljes útmutató 2024-ban,” [Online]. Available: https://hu.shaip.com/blog/a-guide-large-language-model-llm/.

[6] „Kotlin for Android,” [Online]. Available: https://kotlinlang.org/docs/android-overview.html.

[7] D. Stevenson, „What is Firebase? The complete story, abridged,” 25 szeptember 2018. [Online]. Available: https://medium.com/firebase-developers/what-is-firebase-the-complete-story-abridged-bcc730c5f2c0.

[8] G. Manak, „Introduction to Firebase,” 21 augusztus 2023. [Online]. Available: https://medium.com/@gautammanak1/introduction-to-firebase-649e6b7c62bc.

[9] „Client-Server Model,” [Online]. Available: https://www.geeksforgeeks.org/client-server-model/.

[10] R. Sheldon, „What is dependency injection in object-oriented programming (OOP)?,” [Online]. Available: https://www.techtarget.com/searchapparchitecture/definition/dependency-injection.