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

Gyulai Gergő László

UTAZÁSTERVEZŐ ALKALMAZÁS FEJLESZTÉSE ANDROID PLATFORMON

KONZULENS

Somogyi Norbert Zsolt

BUDAPEST, 2022

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2022. 11. 30.

...…….

Gyulai Gergő László

Összefoglaló

Mai felgyorsult világunkban nem árt néha kicsit lelassítani és időt szakítani magunkra. Az utazási ágazatokat tönkretette a járvány. A korlátozások megakadályozták az utazást, a nyaralni vágyók az otthonaikban vészelték át ezt az időszakot, és még akkor is, amikor a társadalom kezd visszaállni a régi kerékvágásba, sokan óvakodnak attól, hogy visszatérjenek a világjárvány előtti utazási szokásaikhoz. Ezzel szemben viszont még többen vannak, akik végre újra útra kelnének és felfedeznék a világ egy távoli zugát, vagy csak egyszerűen lazítanának egy közeli fürdővárosban. Ezeknek az embereknek nyújtana segítséget a szakdolgozatomban elkészült mobilalkalmazás.

Utazások során már-már elengedhetetlen az előzetes tervezés: mikor, hol, mit szeretnénk csinálni. Mindegy, hogy egy sítúrára készülünk az Alpokban, meg akarjuk mászni a Magas-Tátra egyik csúcsát, vagy egy közelben lévő kastélyba akarunk ellátogatni, mindig szükség van előzetes tervezésre. Ha netán időjárási nehézségek lépnek fel vagy valami előre nem megjósolható okból meghiúsul egy úticél elérése, szükség lehet változtatásra előzetes terveinken. A TripPlanner Android alkalmazás célközönsége azok az emberek, akik szeretnék rendszerezetten megszervezni utazásaikat.

Az alkalmazásom Android alatt készült, mivel ez jelenleg a mobilpiac legnagyobb részesedésével rendelkező operációs rendszere. A Google ajánlásának megfelelően az alkalmazás Kotlin nyelven íródott, valamint helyet kapott benne Firebase alapú felhasználó- és adatkezelés. A rendszer RainbowCake architektúrát követ, valamint a felhasználói felület modern Jetpack Compose eszközkészlet felhasználásával készült, amelyekkel szerencsém volt mélyebben megismerkedni a szakmai gyakorlatom alatt.

Abstract

In today's fast-paced world, sometimes it does not hurt to slow down a little and take some time for yourself. Travel industries have been devastated by the pandemic. Restrictions have prevented travel, vacationers have been stuck at home, and even as society begins to return to normal, many are wary of returning to their pre-pandemic travel habits. On the other hand, there are many more who would finally get on the road again and explore a distant corner of the world, or simply relax in a nearby spa town. The mobile application created in my thesis would help these people.

During trips, advance planning is almost essential. Whether you are preparing for a ski trip in the Alps, want to climb one of the peaks of the High Tatras, or want to visit a nearby castle, you always need to plan everything in advance. If weather difficulties occur or if we cannot reach a destination for some unforeseeable reason, it may be necessary to change our preliminary plans. The target users of TripPlanner Android application are people who want to systematically organize their trips.

My application is built on Android, as it currently has the largest share among operating systems in the mobile market. In accordance with Google's recommendation, the application was written in Kotlin and included Firebase user and data management. The system follows RainbowCake architecture, and the user interface was created using the modern Jetpack Compose toolset, which I got to know more deeply during my summer internship.

1 Bevezetés

Az alábbi fejezetben leírom, hogy mi motivált az alkalmazás megalkotására és ezen téma kiválasztására, valamint, hogy miért is jó ez az app. Ezt követően röviden bemutatom az alkalmazást funkcióit, majd pedig leírom, hogy milyen gondolatmenet alapján építettem fel a szakdolgozatom.

1.1 Motiváció, alkalmazás bemutatása

Egy olyan alkalmazást szerettem volna a dolgozatomhoz létrehozni, amit magam is szívesen használnék, valamint mások számára is hasznos lehet. Véleményem szerint az app pont olyanra sikeredett, amilyenre megálmodtam: hasznos és egyszerűen használható. Több, már sikeres mobilos és egyéb alkalmazásból is ihletet merítettem az app létrehozásához, például az utazás kategóriáinak megvalósításához, hogy a végeredmény felhasználóközelibb lehessen. Ezeket bővebben kifejtem a 3.2 fejezetben.

A TripPlanner használatával minden felhasználó egyszerűbbé teheti nyaralásának megtervezését, hiszen bárkivel előfordulhat, hogy nem tudja eldönteni, mégis milyen helyeket mely napokon látogasson meg egy utazás folyamán. A térkép, naptár és lista nézetek segítségével pedig a felhasználó sokkal körültekintőbben, rendszerezettebben tudja megszervezni utazásának minden lépését. A felhasználónak semmi más dolga nincs, mint rögzíteni a meglátogatni kívánt helyeket, majd ezeket úgy összerendezni, ahogy az számára a legmegfelelőbb. Az egyes helyekről pedig egyéb információkat is megszerezhet a többi felhasználó bejegyzései alapján.

Szakdolgozatom témája egy olyan alkalmazás létrehozása volt, amely segít a felhasználójának az utazása minden lépését megtervezni. Lehetőség van regisztrálni, bejelentkezni az alkalmazásba. Az appal lehetőség van az egy utazás alatt bejárandó helyek felvételére, módosítására, törlésére. A felvett helyekhez tartozik lista, naptár és térkép nézet. A lista nézetben látszódik minden felvett, a hely neve, országa, látogatás tervezett ideje és kategóriája. A naptár nézetben megjelennek a helyek a megadott napokon. A térkép nézet pedig a helyek elhelyezkedéseit jelöli. A lista nézet elemeire kattintva eljuthatunk a részletező nézetbe, ahol az általunk megadott hely részleteit, jelenlegi időjárását és más felhasználók által tett megjegyzéseket tekinthetünk meg.

Az elkészült app a legmodernebb alkalmazásfejlesztési eszközöket használva jött létre. Az Android ajánlásait követve a program nyelve Kotlin lett, továbbá a UI Jetpack Compose alapú. Az ajánlott MVI és MVVM architektúrákat áttanulmányozva, valamint a szakmai gyakorlat alatt szerzett tapasztalatok hatására döntöttem el, hogy az alkalmazásom a RainbowCake architektúrára fog épülni. Ezeken felül Firebase alapú felhasználókezelés, Room, és Firestore Database alapú adatkezelés, valamint Retrofit alapú hálózati kommunikációkezelés is bekerült az appba. A térkép működéséhez Google Maps-et, valamit Geocoder-t használtam. A függőséginjektálás pedig Dagger Hilt alapokon történik.

1.2 Dolgozat felépítése

A bevezető fejezetet követően a dolgozatom további részeiben rátérek az alkalmazásban használt fontosabb, nem triviális technológiák részletezésére a 2. fejezetben. Ezután a 3. fejezetben bemutatom az app tervezésének folyamatát, majd a 4. fejezetben következik a tényleges implementáció leírása forráskódrészletekkel és az applikációról készült képernyőképekkel. Végezetül részletezem a tesztelés folyamatát az 5. fejezetben, legvégül pedig összefoglalom a szakdolgozatom lényegét a 6. fejezetben.

2 Felhasznált technológiák

Az alábbi fejezetben kifejtem, hogy milyen fontosabb technológiákat használtam az alkalmazásom megalkotásakor, valamint leírom, hogy az app milyen területein volt szükség ezekre.

2.1 Firebase Authentication

A legtöbb alkalmazásnak igazolnia és ismernie kell a felhasználó személyazonosságát. Ez lehetővé teszi az alkalmazás számára, hogy biztonságosan mentse a felhasználója adatait, és ugyanazt a személyre szabott élményt nyújtsa a felhasználó összes eszközén. A Firebase Authentication [1] lehetőséget nyújt alkalmazásra való regisztrációra, regisztrációt követő e-mailes magerősítésre, bejelentkezésre, kijelentkezésre, valamint email és jelszó módosítására. A regisztrációra és a későbbi bejelentkezésre többféle módszer áll rendelkezésre: email, telefon vagy akár Google, vagy Facebook alapú azonosításra is van lehetőség.

2.2 Firebase Firestore Database

A Cloud Firestore [2] egy rugalmas, jól méretezhető adatbázis a Firebase és a Google Cloud mobil-, web- és szerverfejlesztéséhez. Felhőalapú és valós idejű, vagyis szinkronban tartja az adatokat az adatbázis használói között a valós idejű listener-eken keresztül. Ezen kívül offline támogatást is kínál, így olyan érzékeny alkalmazásokat is létre lehet hozni, amelyek a hálózati késleltetéstől vagy az internetkapcsolattól függetlenül működnek.

2.3 Room Database

Azok az alkalmazások, amelyek nagy mennyiségű strukturált adatot kezelnek, nagy hasznot húzhatnak az adatok lokális megőrzéséből és kezeléséből. A leggyakoribb felhasználási eset a releváns adatok gyorsítótárazása, így amikor az eszköz nem tud hozzáférni a hálózathoz, a felhasználó offline állapotban is hozzáférhet az adatokhoz. A Room [3] perzisztencia könyvtár egy absztrakciós réteget biztosít az SQLite [4] felett, hogy folyékony adatbázis-hozzáférést tegyen lehetővé, miközben az SQLite teljes erejét kihasználja.

2.4 Google Maps MapFragment

Az Androidhoz készült Maps SDK segítségével lehetőség van térképeket hozzáadni Android alapú alkalmazásokhoz. A Google Maps [5] információkat is nyújthat a térkép helyeiről, és támogathatja a felhasználói interakciókat, valamint a felhasználó jelölőket és ábrákat is hozzáadhat térképéhez.

2.5 Geocoder

A Geocoder [6] felelős a geokódolásért és a fordított geokódolásért. A geokódolás egy utcanév vagy egy hely egyéb leírásának (szélességi, hosszúsági) koordinátává alakításának folyamata. A geokódolás segítségével hely és országnévből könnyedén lehet jelölőket helyezni az alkalmazásunkba épített térképre.

2.6 Retrofit

A hálózatépítés az Android alkalmazások egyik legfontosabb része. A Retrofit [7] egy típusbiztos HTTP-kliens Androidra, melyet a legtöbb alkalmazás használ. A Retrofit használata megkönnyíti a hálózatépítést az Android-alkalmazásokban, mivel számos olyan funkcióval rendelkezik, mint például az egyéni fejlécek és kéréstípusok egyszerű hozzáadása, fájlfeltöltések, vagy a mock-olás, amelyek révén könnyebben csatlakoztathatunk webszolgáltatásokat appunkhoz.

2.7 Jetpack Compose

Manapság már nem tudunk olyan alkalmazást készíteni, amely kielégítené a felhasználók szükségleteit kifinomult felhasználói felület nélkül. A Jetpack Compose [8] modern eszközkészlet natív Android felhasználói felület létrehozásához. Leegyszerűsíti és felgyorsítja a felhasználói felület fejlesztését Androidon. Az adott app készítőjének Kotlin kódban kell leírnia a UI paramétereit és a Compose motor ebből fogja generálni a felületet. A UI frissítése is könnyű, mivel az alkalmazás állapota alapján automatikusan frissül. Kevesebb kódot igényel, valamit nincs szüksége XML layout-ra. Viszonylag új technológia, úgyhogy több hiányossága és hibája is van. Az alkalmazásomban pontosan emiatt több esetben is szükséges volt visszatérni XML-re Compose helyett. Ezeket az implementálásról szóló 4. fejezetben bővebben is kifejtem.

2.8 RainbowCake

A RainbowCake [9] egy Kotlin alapú architektúra Android alkalmazásoknak. Egyértelműen elválasztódnak benne a rétegek és komponensek, valamint a nézetek is folyamatosan biztonságos és konzisztens állapotban vannak. A RainbowCake egyrészt függőségek halmaza, amelyek osztályokat és egyéb konstrukciókat tartalmaznak az alkalmazásokban, másrészt útmutatást nyújt ezen alkalmazások megvalósításához. Az alkalmazásomban használt architektúrát később, a 3.4. fejezetben bővebben is tárgyalom.

2.9 Dagger Hilt

A függőség-injektálás egy szoftverfejlesztési technika, ahol az objektumok függőségeinek példányosítását szétválasztjuk a használatuktól. Ennek következményeképpen az adott osztály lazán csatolva fog függeni az adott függőségtől, ami miatt könnyen bővíthetővé válik az adott függőség vonatkozásában. A Dagger [10] egy függőség-injektáló keretrendszer, amely némi deklaratív konfigurációt követően (pl. annotációk) automatikusan szolgáltatja a megadott osztályok példányainak a megadott függőségeket. A Hilt [11] egy absztrakció a Dagger fölött, amely szabványos módot biztosít Dagger alapú függőség-injektálás beépítésére egy Android alkalmazásba és megkönnyíti a Dagger használatát. Az alkalmazásban alkalmazott függőség-injektálás témáját a 3.5. fejezetben bővebben is tárgyalom.

2.10 MockK

A mock-olás egy olyan technika, amely a tesztelési kódot olvashatóvá és karbantarthatóvá teszi és fő célja, hogy az alkalmazás egyes komponenseit elkülönítve lehessen tesztelni. A módszer lényege, hogy bizonyos komponenseket a tesztelés során egy helyettesítő vázra cserélünk ki. A váz interfésze megegyezik az eredeti komponensével és műveleteinek programozottan megadható, hogy milyen eredményt szolgáltassanak. A MockK [12] egy Kotlin alapú könyvtár, amely nagyszerűen használható Android alkalmazás teszteléséhez, és lehetővé teszi objektumok mock-olását az Android unit tesztekben.

3 Tervezés

Az alábbi fejezetben kifejtem, hogy milyen lépéseken ment keresztül az alkalmazás az ötleteléstől az implementációig. Részletezem, hogy milyen egyéb, meg nem valósult terveim voltak a szakdolgozatom témáját illetően, valamint leírom, hogy milyen már megvalósult appokból merítettem ihletet. Wireframe-ekkel szemléltetem a UI tervezési fázisait, valamint bővebben kitérek az alkalmazás architektúrájára is.

3.1 Tervezés előtti ötletek

Szakdolgozatprojektem készítésének első szakaszában még csak fejben ötleteltem, hogy milyen témája legyen a leendő alkalmazásomnak. Mindenképpen valami hasznosat és egyszerűen használhatót szerettem volna alkotni, ami megmutatja az egyetemen szerzett tudásomat és méltó lehet egy szakdolgozat alapjához. Az, hogy az alkalmazás Android operációs rendszer alatt, Kotlin nyelven fog készülni már ekkor is biztos volt. Kezdeti ötleteim között szerepelt egy fitnesz applikáció, amivel a felhasználók nyomon tudnák követni edzéstervüket, sportolási szokásaikat és esetleg megoszthatnák véleményüket és mozgási tevékenységeiket más felhasználóknak is. Egy másik ötletem egy ételek ajánlására és értékelésére specializálódott alkalmazás lett volna, amelyben a felhasználók különféle recepteket vagy éttermi ételeket oszthatnának meg egymás között. Végül egy utazástervező alkalmazás mellett döntöttem, mivel számomra ez tűnt a leginkább hasznosnak, hiszen magam is gyakran utazok.

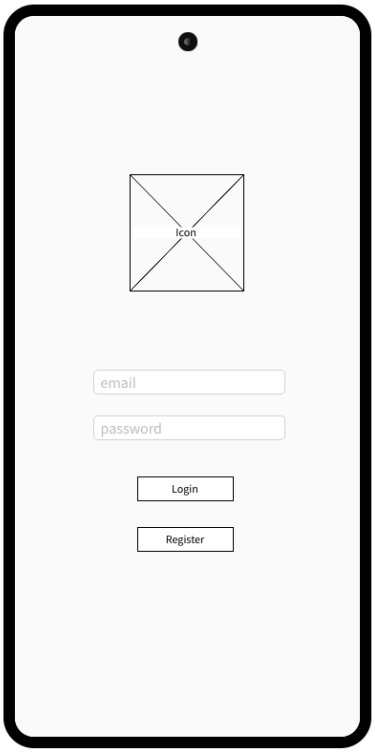
3.2 Hasonló alkalmazások

Többféle alkalmazásból és egyéb oldalakról is ötleteket, ihletet merítettem a tervezési folyamat közben. Az egyik ilyen, általam is gyakran használt népszerű app, a Tripadvisor [13] volt, ahol a felhasználók az utazásuk alkalmával szerzett ismereteiket tudják megosztani, valamint képesek hotelt vagy éttermet is foglalni maguknak. Ami a legjobban megfogott a Tripavisor-ban, az a felhasználóinak a közössége, akik segítséget nyújtanak utazótársaiknak akár a legolcsóbb autóbérlő hely, akár a legszebb tengerpart megtalálásáról legyen szó. További ötleteket, például az utazási kategóriák beosztását a Google Travel [14] oldalról kölcsönöztem, ami a Google utazástervező szolgáltatása, amely jelenleg mobilos applikáció formájában sajnos nem elérhető.

3.3 UI tervezése

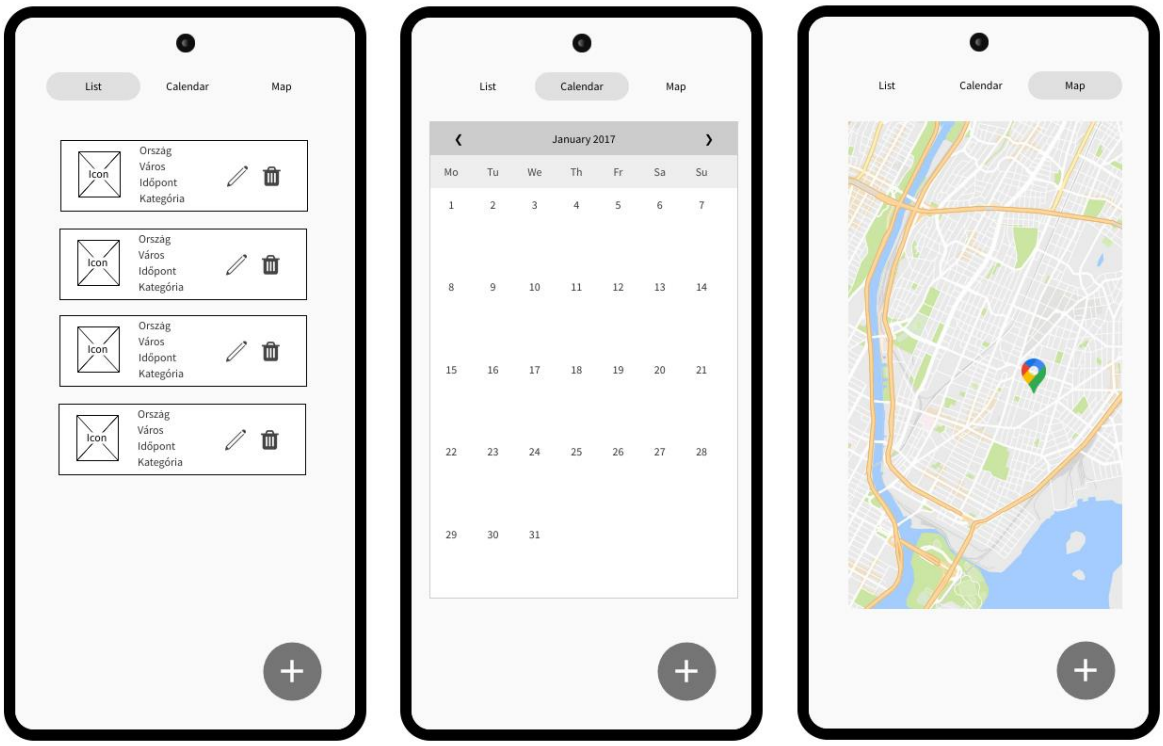
A felhasználói felület tervezése viszonylag gyors folyamat volt számomra, mivel az ötleteim az alkalmazás kinézetére már a tervezés korai szakaszában is tiszták voltak. Az applikáció kinézetéről wireframe-eket készítettem MockFlow [15] segítségével. A wireframe egy vázlat vagy tervrajz, amely hasznos segítséget nyújt a programozóknak és tervezőknek az elkészítendő szoftver vagy webhely szerkezetének átgondolásában.

A kezdőképernyőt mindenképpen egy szokásos login képernyőként szerettem volna megvalósítani. Legfelül az alkalmazás logójával, alatta a bejelentkezéshez szükséges adatok (email/telefonszám, jelszó) szövegdobozaival. Az alkalmazás logója ekkor már megvolt, viszont ezt még túlságosan képlékenynek éreztem ahhoz, hogy a wireframe-re kerüljön. A jelszó szövegdobozát természetesen úgy szerettem volna megvalósítani, hogy a jelszó láthatósága váltogatható legyen. A szövegdobozok alatt pedig a bejelentkezés gombját terveztem tenni. Ekkor még úgy képzeltem el, hogy a regisztrációnak külön oldalt is csinálok, de a felhasználók szempontjából feleslegesnek éreztem, ha több adatot is meg kellene adni regisztrációhoz, így a login oldalon lehet regisztrálni és bejelentkezni is. Emiatt pedig az oldal aljára került egy regisztráció gomb.



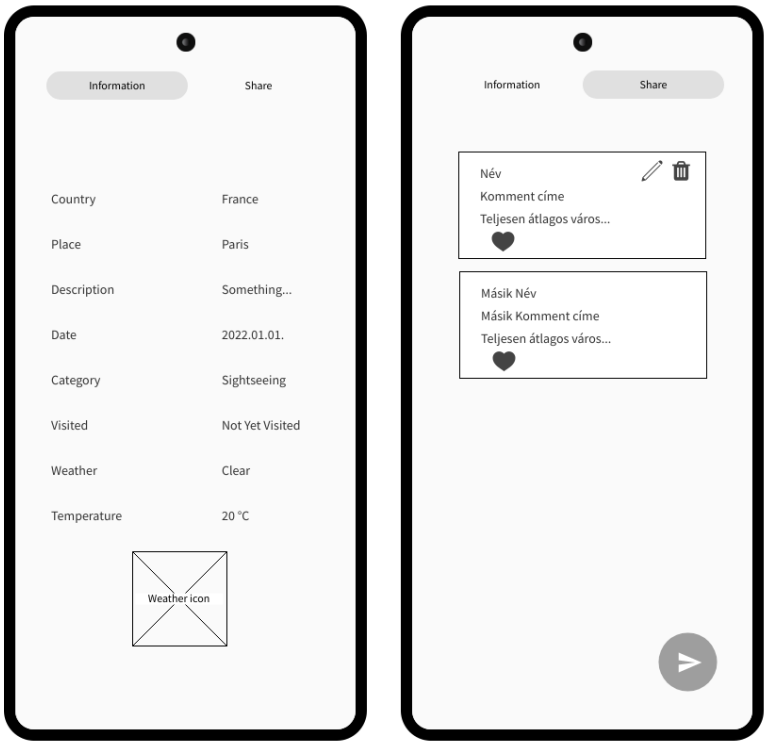
1. ábra Wireframe a login nézetről

Bejelentkezést követően az alkalmazás az utak listájára navigál. Egyes előzetes ötleteimben a listaelem felvétele még külön oldalon lett volna, de mire a wireframe-ek készítéséig eljutottam, inkább DialogFragment-es módszer mellett döntöttem. Így került az oldalakra egy FloatingActionButton, amelyre, ha rákattintunk, feljön egy dialog ablak, ahol a mezőket kitöltve adhatunk utazást a listánkhoz. Az utazásoknak lista, naptár és térkép nézete is van, melyek között a felső navigation bar segítségével közlekedhetünk. A lista nézeten megjelenik minden felvett utazás kártyákon, utazási idő rendjében. A kártyákon megjelenik az utazás kategóriájának ikonja, majd pedig az utazás országa, városa, időpontja és kategóriája. Végül minden kártyán egy-egy ikon szerepel az adott utazás módosítására és törlésére. Az ez után következő nézet az utazások naptárja. Ebben az utazás dátumától függően jelennek meg az utak a naptár adott napján. A wireframe-ek megalkotásakor azt még nem döntöttem el, hogy a naptárak milyen módon jelenítenék meg az adott helyeket, mivel ekkor még nem tudtam milyen lehetőségek állnak rendelkezésemre naptár specifikálás terén. Az utolsó nézet a sorban a térképet jeleníti meg. Itt az utazás földrajzi elhelyezkedését tekintheti meg a felhasználó. Az egyes utazásokhoz tartozik egy térkép marker, amivel a felhasználó könnyedén beazonosíthatja utazásának helyét. Úgy terveztem, hogy a felhasználókezeléshez kapcsolódó műveletek, pl. kijelentkezés, egy drawer-ben kapnak helyet.



2. ábra Wireframe az utazások nézeteiről

A lista nézet egyes elemeire kattintva az alkalmazás a részletező nézetre navigálja a felhasználót. A részletező oldalnak információ és megosztás nézete is van, melyek között a felső navigation bar segítségével közlekedhetünk. Az információ nézeten láthatjuk az adott utazás részleteit: ország, hely, leírás, dátum, kategória, látogatottság. Ezeken felül úgy gondoltam érdemes lenne a hely időjárási viszonyairól is információkat közölni, mivel egy utazás során ez rendkívül hasznos lehet. A megosztás nézeten láthatjuk, hogy a többi felhasználó milyen infókat osztott meg az adott helyről. Ezeket az infókat értékelni is tudná a felhasználó, valamint a saját kommentjeit mindenki tudná módosítani és törölni is. Az új komment felvétele szintén FloatingActionButton-nal és DialogFragment-tel történne.

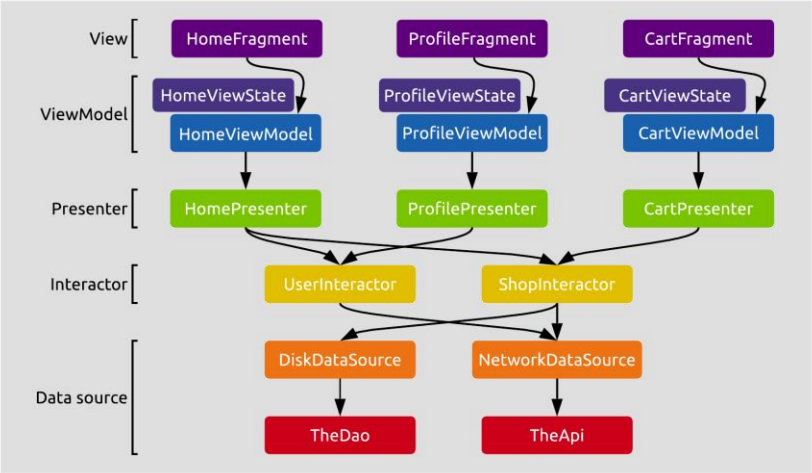


3. ábra Wireframe a részletező nézetekről

3.4 Architektúra

A RainbowCake architektúra egyrészt függőségek halmaza, amelyek osztályokat és egyéb konstrukciókat tartalmaznak az alkalmazásokban, másrészt útmutatást nyújt ezen alkalmazások megvalósításához. Az architektúra főbb céljai:

- A rétegek és komponensek egyértelmű szétválasztása.
- A nézet folyamatos biztonságos és konzisztens állapotban tartása.
- Konfiguráció változások megfelelő kezelése.
- Hosszabb ideig tartó műveletek háttérszálakon való végrehajtásának könnyítése.



4. ábra RainbowCake architektúra rétegei[1]

Az architektúra rétegeinek leírása:

- **View** (Fragment vagy Activity): Az alkalmazásképernyők, amelyek a ViewModel-ek állapotát figyelik és felelősek a megjelenítésért a felhasználói felületen. Ezenkívül eseményeket továbbítanak a ViewModel-nek és értesülnek az állapot változásokról és eseményekről.
- **ViewModel**: Tárolja a felhasználói felület aktuális állapotát, valamint kezeli a felhasználói felülettel kapcsolatos logikát. Frissíti az állapotot a Presenter-től kapott eredmények alapján. Coroutine-okat indítanak minden input eseménye hatására és a Presenter felé továbbítják a kéréseket.
- **Presenter**: A végrehajtást a háttérszálakra helyezi és Interactor-okat használva éri el az üzleti logikát. Ezután az eredményeket képernyő-specifikus megjelenítési modellekké alakítják át, amelyeket a ViewModel állapotként tárolhat.
- **Interactor**: Tartalmazza az alkalmazás alsó szintű üzleti logikáját. Aggregálják és kezelik az adatokat, valamint számításokat végeznek. Nem egyetlen képernyőhöz vannak kötve, hanem az alkalmazás főbb funkciói szerint csoportosítják a funkciókat.
- **Data source**: Ellátják az Interactor-okat adatokkal. Fő feladatuk, hogy szétválasszák az implementáció és domain rétegeket egymástól, valamint, hogy a tárolt adataikat konzisztens állapotban tartsák.

[1] https://rainbowcake.dev/

Az alkalmazás architektúra diagramja a dolgozat végén látható. Az egész app RainbowCake alatt készült, így egyértelműen elkülönülnek benne az egyes rétegek. Minden alkalmazásnézethez tartozik egy Fragment, ViewModel, ViewState és Presenter. A négy Interactor tartalmazza az alsó szintű üzleti logikát, valamint ezekhez tartozik a négy adatforrás is:

- Firebase Auth: A felhasználókezeléssel kapcsolatos adatokat kezeli Firebase-ben
- TripList Dao: Lokálisan tárolja az utazások listáját
- Weather Api: Az időjárással kapcsolatos adatokat kezeli
- Cloud Firestore: A kommentekkel kapcsolatos adatokat kezeli Firebase-ben

3.5 Függőség-injektálás

A Függőség-injektálás (Dependency injection, DI) egy olyan technika, amelyet széles körben használnak a programozásban, és jól illeszkedik az Android fejlesztéséhez, ahol a függőségeket egy osztálynak biztosítják ahelyett, hogy maguk hoznák létre őket. A DI alapelvek követése megalapozza a jó alkalmazásarchitektúrát, a kódok nagyobb újrafelhasználhatóságát és az egyszerű tesztelést.

A RainbowCake két különböző package-t tartalmaz a DI integrációhoz, de teljesen egyedi megoldás is használható a keretrendszerrel. A Hilt könyvtár szabványos módszert határoz meg a DI alkalmazásában azáltal, hogy konténereket biztosít a projekt minden osztályához, és automatikusan kezeli az életciklusukat. A Hilt a népszerű Dagger DI könyvtárra épül, így a Dagger által biztosított fordítási idő helyesség, futásidejű teljesítmény, méretezhetőség előnyeit élvezi. A Dagger automatikusan olyan kódot generál, amely utánozza azt a kódot, amelyet egyébként kézzel írt volna a fejlesztő. Mivel a kód a fordítási időben jön létre, nyomon követhető és hatékonyabb, mint a többi reflexió alapú megoldás. Mivel sok Android keretrendszer osztályt maga az operációs rendszer példányosít, a Dagger Android alkalmazásokban való használatakor van egy ehhez kapcsolódó ismétlődő kód. A Dagger-rel ellentétben a Hilt integrálva van a Jetpack könyvtárakkal és az Android keretrendszer osztályaival, és eltávolítja a legtöbb ilyen ismétlődő kódot. Csökkenti a kézi függőség-injektálás szükségességét a projektben.

Mivel a Hilt-tel volt szerencsém szakmai gyakorlatom alatt megismerkedni és mivel a RainbowCake támogatja többek között ezt a DI megoldást is, a Hilt-et választottam az alkalmazás implementálásához. Mindemellett rendkívül népszerű az alkalmazásfejlesztők körében is, habár viszonylag új technológiáról beszélünk.

4 Implementáció

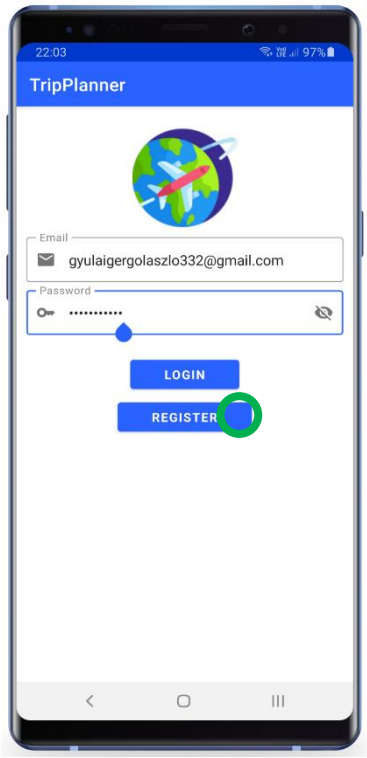
Az alábbi fejezetben részletesebben is kifejtem az elkészült alkalmazásom minden elemét. Bemutatom a már véglegesített felhasználói felületet képernyőképekkel, valamint ismertetem az app belső működését és a felhasznált technológiákat kódrészekkel. A kész applikáció összesen 6 főnézetből áll, amelyek alapján fogok sorban haladni a fejezet során.

4.1 Regisztráció

Akárcsak a legtöbb nagyobb méretű alkalmazásnak, így a TripPlanner-nek is szüksége van a felhasználó azonosságának igazolására és tárolására. Az app a Firebase Authentication eszköztárát kihasználva ad lehetőséget a felhasználónak a regisztrációra.

4.1.1 View

Az applikációban a regisztrációra és bejelentkezésre használt nézet egy és ugyanaz. Az oldal teljes egészében Jetpack Compose felhasználásával készült, valamint kihasználja a Material Design által nyújtott lehetőségeket a szövegmezők megalkotásánál. Az két mezőt kitöltve tudunk regisztrálni a Register gombra nyomva.



5. ábra Regisztráció nézet

4.1.2 Interactor

Amennyiben a felhasználó az email vagy a password mezőt üresen hagyva kattintana rá a Register gombra, abban az esetben egy toast jelenik meg, hogy emlékeztessen ezen mezők kitöltésére. Ha az email hiányzik, a „Please enter your email”, ha a jelszó hiányzik, akkor a „Please enter your password” üzenet jelenik meg. Ha mindkét mező töltve van, elkezdődhet a regisztráció folyamata az autorizációért felelős Interactor-on keresztül. A rendszer többek között vizsgálja az email cím formájának helyességét, s ha nem találja megfelelőnek, akkor a „The email address is badly formatted” üzenettel jelez. Ezen felül ellenőrzi, hogy a jelszó megfelelő biztonságot biztosít-e a felhasználónak. Abban az esetben, ha 6 karakternél rövidebb jelszót szeretnénk beállítani, az „The given password is invalid” üzenettel szembesülünk. Ha a két mezőbe írt adatok megfelelnek, megtörténik a tényleges regisztráció. Ezt a „Registration was successful. Verification email has been sent.” üzenetek jelzik. Ekkor a beállított email címre egy verifikációs email érkezik, ami közli a regisztráció sikerességét, valamint az ebben található linkre kattintva tudjuk igazolni a regisztrációt.

Az a mezők ellenőrzéséért és a regisztráció indításáért felelős kódrészlet:

```
onClick = {
     if(emailInput == "") {
          Toast.makeText(context,     "Please     enter     your     email",
Toast.LENGTH_SHORT).show()
     } else if(passInput == "") {
          Toast.makeText(context,    "Please    enter    your    password",
Toast.LENGTH_SHORT).show()
     } else { onRegisterClick(emailInput, passInput, context) }
}
```

Az alábbi kódrészlet felelős a tényleges regisztráció kezeléséért:

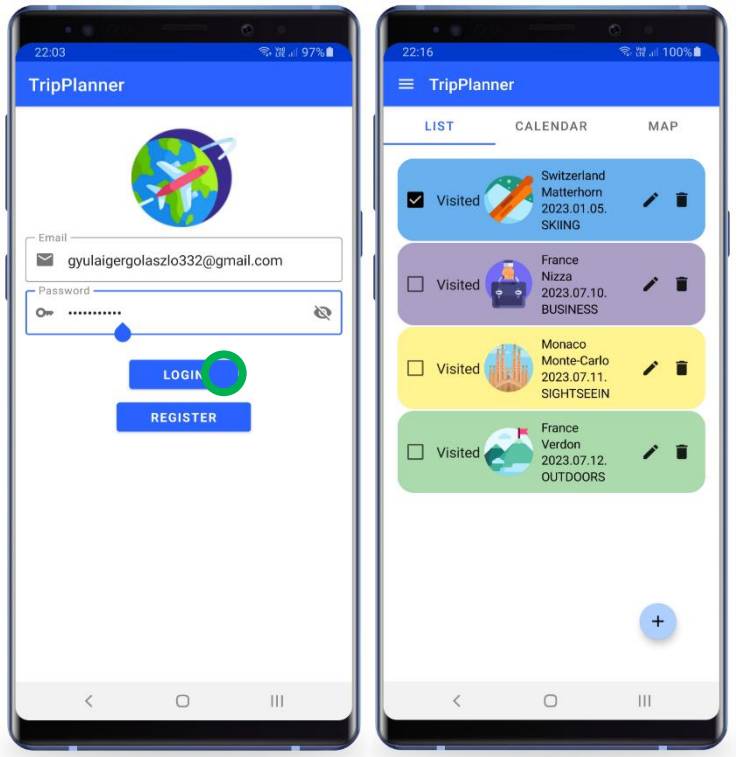
```
firebaseAuth
     .createUserWithEmailAndPassword(mail, pass)
     .addOnSuccessListener { result ->
          val firebaseUser = result.user
          val profileChangeRequest = UserProfileChangeRequest.Builder()
               .setDisplayName(firebaseUser?.email?.substringBefore('@'))
               .build()
          firebaseUser?.updateProfile(profileChangeRequest)
          firebaseUser?.sendEmailVerification()

          Toast.makeText(context,            "Registration            was
successful\nVerification email has been sent", Toast.LENGTH_SHORT).show()
     }
     .addOnFailureListener { exception ->
          Toast.makeText(context,                        exception.message,
Toast.LENGTH_SHORT).show() }
```

4.2 Bejelentkezés

A regisztrációhoz természetesen tartoznia kell bejelentkezésnek is. Az applikációba a regisztrációhoz hasonlóan a bejelentkezést is a Firebase Authentication végzi. Ez kezeli az email és a jelszó helyességét, valamint jelez, ha valami gyanús tevékenységet érzékel a bejelentkezni kívánó felhasználótól.

4.2.1 View

A bejelentkezés nézete teljesen megegyezik a regisztráció nézetével, valamint az előre elkészített login wireframe-mel. Az oldal tetején megjelenik az alkalmazás ikonja. Alatta látható a Material Design segítségével létrejött email, valamint password szövegdobozok. A jelszó beírásánál a jobb oldali szem ikonnal változtathatjuk, hogy olvasható vagy védett láthatóságúvá szeretnénk-e tenni a jelszót. Ezalatt van a Login és a Register gomb. Az előbbire kattintva pedig már el is lehet indítani a bejelentkezés folyamatát. Bejelentkezni ugyanúgy nem lehet üres mezőkkel, mint regisztrálni. Ha üresen hagynánk a password vagy az email mezőt, akkor itt is egy toast üzenet jönne fel, ami emlékezteti a felhasználót ezen mezők kitöltésére.



6. ábra Bejelentkezés nézet

4.2.2 Interactor

Amennyiben úgy kattintunk a login gombra, hogy a login és a password mezők is töltöttek, akkor a bejelentkezés folyamata el is kezdődhet az autorizációért felelős Interactor-on keresztül. A bejelentkezésnél is vizsgálja és kezeli a rendszer az email cím formai helyességét, s ha nem találja megfelelőnek, akkor a „The email address is badly formatted" üzenettel jelzi. Amennyiben valamelyik mezőt helytelenül írnánk be, vagy ha egy nemlétező fiókba próbálnánk belépni, a „There is no user record corresponding to this identifier. The user may have been deleted." üzenetet fogjuk látni. Ha egy olyan fiókba próbálnánk belépni, amelynek e-mailjére érkezett üzenetben még nem igazolták vissza a regisztrációt, akkor a „Please verify your email" üzenet jelenne meg. Ha a két mezőbe írt adatok megfelelnek egy létező, visszaigazolt felhasználói fiók adatainak, akkor el is indulhat a tényleges bejelentkezés. Ekkor feljön az alkalmazás legfontosabb kezdőoldala, az utazási lista nézettel.

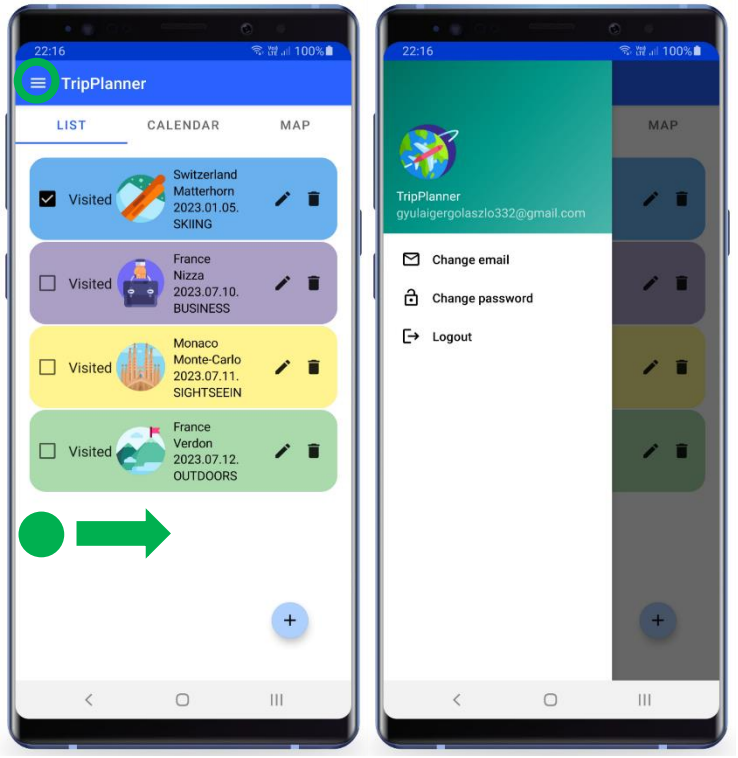
A különböző oldalak közötti navigálásra, váltakozásra a RainbowCake által biztosított navigator-t használom. A Fragment-ek közötti navigálás az egyszem Activity kevés feladatainak egyike. A megnyitott nézeteket az alkalmazás stack-ben tárolja, ahova új ablak nyitásakor új elem érkezik, visszalépéskor pedig eltávolítódik. A navigator többek között lehetőséget ad az oldalak közötti váltakozáskor az animációk használatára. Az összes animációt kipróbálva viszont arra a döntésre jutottam, hogy a legjobban a tiszta, animációmentes átmenet néz ki, emiatt pedig mind a belépő, mind a kilépő animációkat nullára állítottam.

Az alábbi kódrészlet felelős a tényleges bejelentkezés kezeléséért:

```
firebaseAuth
     .signInWithEmailAndPassword(mail, pass)
     .addOnSuccessListener {
          if(firebaseAuth.currentUser!!.isEmailVerified) {
               navigator?.add(TripListFragment(),
                    enterAnim = 0,
                    exitAnim = 0,
                    popEnterAnim = 0,
                    popExitAnim = 0
               )
          }
          else
               Toast.makeText(context,   "Please   verify   your   email",
Toast.LENGTH_SHORT).show()
     }
     .addOnFailureListener { exception ->
          Toast.makeText(context,              exception.localizedMessage,
Toast.LENGTH_SHORT).show() }
```

4.3 Egyéb felhasználókezelés

Sikeres bejelentkezést követően az utazáslista nézet tárul a felhasználó szeme elé. Ha viszont a bal oldalon levő drawer-t lenyitjuk, megjelennek további felhasználókezeléssel kapcsolatos lehetőségek. Ez a nézet már nem Compose-ban, hanem XML layout-ban készült. Ez azért volt szükséges, mivel a ViewPager-t jelenleg nem lehetséges olyan módon megvalósítani Compose-on belül, ahogy szerettem volna. A ViewPager felelős azért, hogy a lista, naptár és térkép nézeteket a felhasználó tudja váltogatni, azonban erre a Compose nem képes olyan módon, hogy az megfeleljen a RainbowCake architektúrának.



7. ábra Felhasználókezelés drawer

4.3.1 Kijelentkezés

A kijelentkezés elindításához a drawer-ben található legalsó Logout gombra kell kattintani. Ekkor először a rendszer kijelentkezteti a felhasználó Firebase-ből, majd a RainbowCake navigator segítségével visszajuthatunk a bejelentkezés oldalra.

A kijelentkezéshez használt kódrészletek:

```
firebaseAuth.signOut()

navigator?.pop()
```

4.3.2 Jelszó változtatás

A második, „Change password” feliratú gombbal a felhasználó a jelszavát tudja megváltoztatni. Ekkor egy üzenet érkezik a felhasználó email címére, amiben a linkre kattintva van lehetősége új jelszót beállítani.

A jelszóváltoztatáshoz használt kódrészlet:

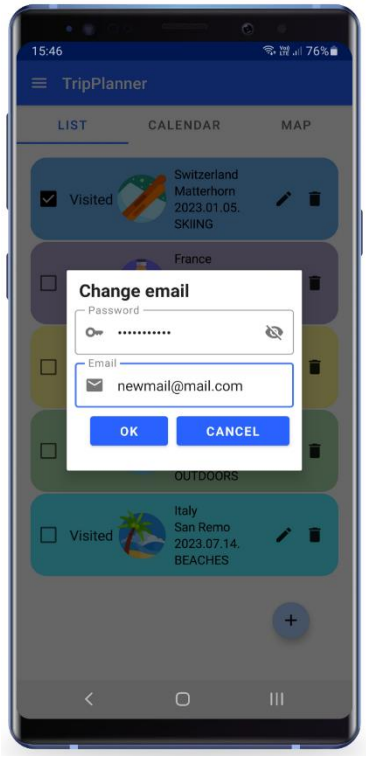
```
firebaseAuth.sendPasswordResetEmail(mail)
```

4.3.3 Email változtatás

Mindezek mellett a felhasználó az email címét is meg tudja változtatni a legfelső „Change email” feliratú gombbal. A felugró dialógus ablakba be kell írni a jelenlegi jelszót és az új emailt, majd az ok gombra kattintva el is kezdődik a folyamat. Ezt a változtatást persze vissza is kell igazolni emailben.

Az email változtatásához használt kódrészlet:

```
firebaseAuth.currentUser?.reauthenticate(credential)
     ?.addOnSuccessListener{
          firebaseAuth.currentUser?.verifyBeforeUpdateEmail(newEmail)
          Toast.makeText(context, "Verification email has been sent",
Toast.LENGTH_SHORT).show() }
     ?.addOnFailureListener { exception ->
          Toast.makeText(context,              exception.localizedMessage,
Toast.LENGTH_SHORT).show() }
```



8. ábra Email változtatás dialógusnézet

4.4 Utazási lista

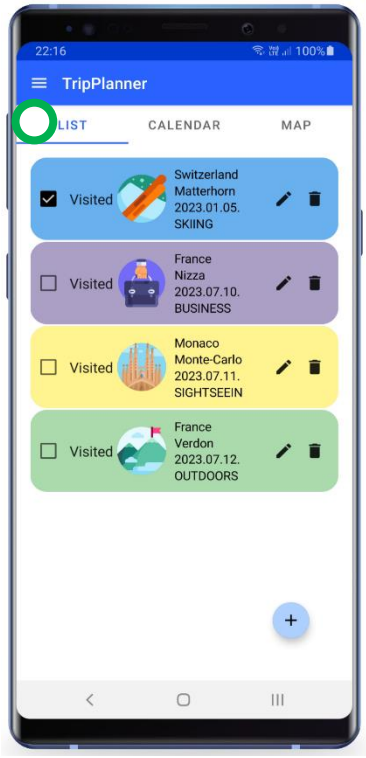
Az utazások lista nézete az alkalmazás legfontosabb része. Itt láthatjuk az összes eddigi és még meg nem tett útjainkat sorban, valamint ide sorakoznak majd az elkövetkezendő kiruccanásaink, túráink. A lista elemeire kattintva pedig átnavigálhatunk az adott út részletező nézetére.

4.4.1 View

A lista nézet teljes egészében Jetpack Compose felhasználásával készült. A lista elemei jelölik az utazás során meglátogatni kívánt helyeket, melyeket előzőleg rögzített a felhasználó. Az elemek elején egy checkbox van, amivel jelölni tudjuk, hogy az adott helyet már meglátogattuk-e vagy sem. Ezután látható az utazás kategóriájának ikonja, majd részletezve láthatjuk az úthoz tartozó országot, helyet, időpontot és a kategória nevét. Végül a sorban két ikon is van: ezek az elem módosító és a törlő gombok.

A listában színjelölést alkalmaztam, melyek az egyes kategóriákat jelölik:

- zöld – szabadban (outdoors)
- ciánkék – tengerpart (beaches)
- sárga – városnézés (sightseeing)
- élénk kék – síelés (skiing)
- lila – üzleti (business)



9. ábra Utazások lista nézete

4.4.2 Interactor

Az utazáslistát a felhasználó szabadon tudja megalkotni, módosítgatni vagy törölni, ahogyan a legtöbb listát is képes a felhasználója. Az utazásokért felelős Interactor-ban működnek a lista legfontosabb metódusai. Az adatbázisból innen lehet elérni az utazási lista adatait, amelyeket betölthetünk a képernyőre. A betöltés előtt viszont még időrendbe helyezi az elemeket, hogy a képernyőre már helyes sorrendben legyenek betöltve. Ezen kívül itt adódik hozzá a listához új utazás, itt módosul már meglevő utazás, valamint itt törlődik utazás az adatbázisból.

A lista működéséhez használt kódrészletek:

```
suspend fun load(): List<TripListItem> {
     val data  = mutableListOf<TripListItem>()
     data.addAll(database.getAll())
     data.sortBy { it.date }
     return data
}

suspend fun add(newItem: TripListItem): List<TripListItem> {
     database.insert(newItem)
     return load()
}

suspend fun edit(editedItem: TripListItem): List<TripListItem> {
     database.update(editedItem)
     return load()
}

suspend fun remove(removedItem: TripListItem): List<TripListItem> {
     database.delete(removedItem)
     return load()
}
```

4.4.3 Data source

Az adatbázissémák megalkotásába már a tervezési fázisban belekezdtem, de csak az implementáció megvalósításánál lett véglegesítve minden, az adatbázis használathoz kapcsolódó elem. Ennek fő oka az volt, hogy struktúráját eleinte eléggé képlékenynek éreztem, amit könnyedén változtathatónak szerettem volna hagyni. Mivel az alkalmazásomat úgy szerettem volna megalkotni, hogy nagy mennyiségű strukturált adatot is tudjon kezelni, a Room adatbázis mellett döntöttem, mellyel az egyetemi éveim alatt közelebbről is megismerkedtem. Eleinte úgy gondoltam, hogy szükség lehet Firebase adatbázist is alkalmazni az utak tárolásához, viszont végül úgy döntöttem, hogy elegendő lokálisan tárolni mindent az utazásokról, és nem éreztem szükségét egy valós idejű adatbázisnak ebben az esetben.

Az adatbázisban tárolt elemek oszlopai:

- id: az elem azonosítója
- place: az utazás helyét tárolja, amit a felhasználó meg szeretne látogatni, esetleg már meg is látogatott
- country: a helyhez tartozó országot tárolja
- description: az utazás részletesebb leírását tárolja, ahova a felhasználó konkrétabb terveket és egyéb információkat jegyezhet fel
- date: az adott utazáshoz tartozó pontos dátumot tárolja YYYY.MM.DD. formátumban
- category: az utazás kategóriáját tárolja, amit a rendszer egy enum osztályból vesz ki
  - outdoors
  - beaches
  - sightseeing
  - skiing
  - business
- visited: az utazás látogatottsági besorolása (meglátogatott/egyelőre még meg nem látogatott)

Az utazáslista elemeinek adat osztálya:

```
data class TripListItem(
     @ColumnInfo(name = "id") @PrimaryKey(autoGenerate = true) var id: Long?
= null,
     @ColumnInfo(name = "Place") var place: String,
     @ColumnInfo(name = "Country") var country: String,
     @ColumnInfo(name = "Description") var description: String,
     @ColumnInfo(name = "Date") var date: String,
     @ColumnInfo(name = "Category") var category: Category,
     @ColumnInfo(name = "Visited") var visited: Boolean
) : Parcelable
```

A lista adatbázisműveleteinek kódrészletei:

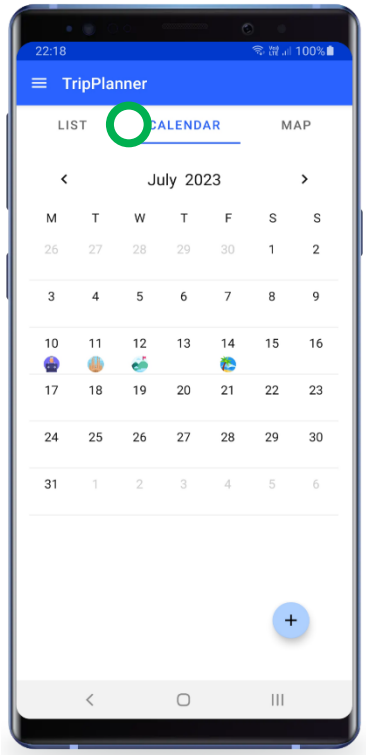
```
interface TripListItemDao {
    @Query("SELECT * FROM triplistitem")
    suspend fun getAll(): List<TripListItem>

    @Insert
    suspend fun insert(tripListItems: TripListItem): Long

    @Update
    suspend fun update(tripListItems: TripListItem)

    @Delete
    suspend fun delete(tripListItems: TripListItem)
}
```

4.5 Naptár

A naptár a lista mellett elhelyezkedő második elem. A nézet megalkotásánál nem Compose-t használtam. Mivel a technológia igen új és még egyelőre hiányos, így sajnos nem találtam az elképzeléseimnek megfelelő naptárat. Emiatt maradtam egy XML layout alapú megoldásnál. Az alap naptárak azonban itt sem voltak elegendőek, így keresnem kellett egy jobb megoldást. Végül ráleltem az Applandeo naptárra [16].

Az Applandeo Material-Calendar-View egy Material Design alapú, egyszerű és testreszabható naptár widget Androidra. A widget-nek két funkciója van: egy dátumválasztó a dátumok kiválasztásához (XML widget-ként és párbeszédpanelként is elérhető), valamint egy klasszikus naptár. A dátumválasztó működhet egynapos, több napos vagy tartományválasztóként. Én egyedül a szimpla naptár funkcióját használtam, viszont teljesen kielégítette minden kívánalmaimat.

A naptár nézetben sorakoznak az utazások az utak idejétől függően. Minden utat egy-egy ikon jelöl, amely megegyezik az út kategóriájának ikonjával. Ha egy napon több utazás is van, akkor a listában előrébb helyezkedő kategória ikonja kerül a naptárba. Ha az ikonokra kattint a felhasználó, akkor egy toast üzenetben jön fel az utazás helye. Ha több utazás is van egy napon, akkor értelemszerűen mind megjelenítődik üzenetként.



10. ábra Utazások naptár nézete

4.6 Térkép

Minden utazás egyik leginkább nélkülözhetetlen eleme a térkép. Enélkül az utazók eltévedhetnek és igencsak megkeserülne emiatt minden kalandor nyaralása. Régen papír alapú térképeket használtak az emberek, de ma már mindenki a zsebében hordozhatja a föld átméretezett mását minden részletével. Az alkalmazásomba épített térképen megjelenítődik minden, a listába felvett utazás.

4.6.1 View

A térkép a naptár mellett elhelyezkedő harmadik elem. A nézet nem Compose alapú, mivel a technológiát sajnos a térkép megalkotásának idejében eléggé hiányosnak véltem. Habár lehetséges Compose térkép nézetet létrehozni, de a jelölők folyamatos hozzáadása, elvétele, módosítása, valamint a térkép kamerájának módosítására jelenleg nem találtam hatékony módot. Így maradtam a jól bevált XML layout alapú megoldásnál. A nézeten megjelenítődnek az utazás során meglátogatni kívánt helyek markerekként. Ezek a jelölők mind az út kategóriájától függően más-más színűek, valamint a markerekre kattintva megjelenik a felhasználó által megadott hely neve is.

11. ábra Utazások térkép nézete

4.6.2 Interactor

Az utazásokért felelős Interactor-ban működnek a lista legfontosabb metódusai. Új térkép létrehozása előtt először törölni kell az előzőt. Ahhoz, hogy a térkép megalkotásának fontosabb részei működjenek, meg kell bizonyosodni róla, hogy a készülék csatlakozik-e az internethez. Kapcsolat nélkül a felhasználó csak egy üres térképet látna az alkalmazásban. A térkép legfontosabb részei a jelölők, amik Geocoder segítségével állítódnak be. A Geocoder a kapott utazás helyének és országának nevéhez legenerálja a szélességi és hosszúsági koordinátákat. A koordinátára ezután kerülhet egy marker, aminek a színe az út kategóriájától függ. Végül a térkép kamerája olyan pozícióba kerül, hogy minden jelölő jól látszódjon.

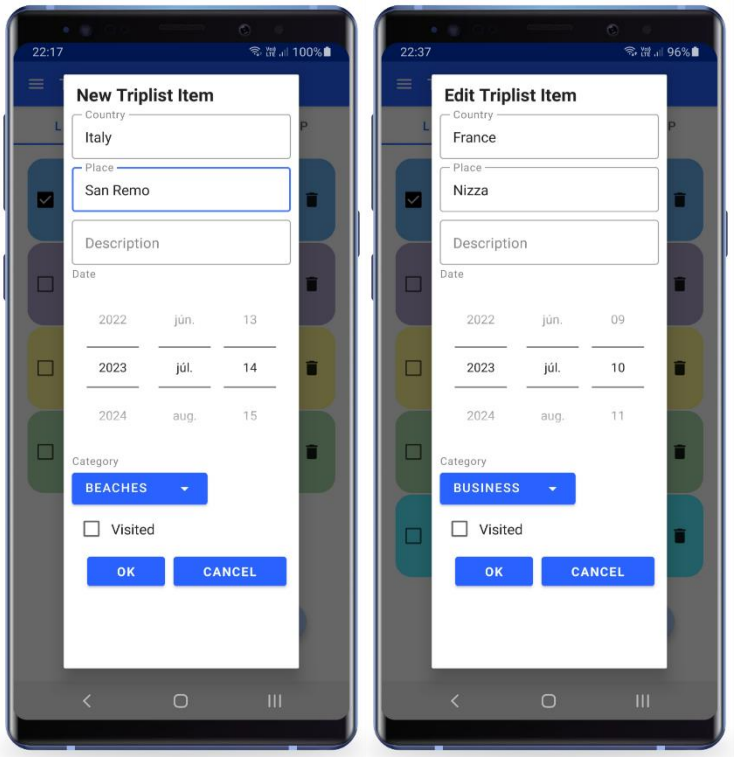
Térképhez tartozó kódrészletek:

```
googleMap.clear()

val connected = ConnectivityChecker.isConnected(context)
val geocoder = Geocoder(context, Locale.getDefault())
val bld = LatLngBounds.Builder()
var matches: MutableList<Address>? = null

geocoder.getFromLocationName(item.place + " " + item.country, 1)

coordinates = LatLng(matches[0].latitude, matches[0].longitude)
when(item.category) {
     TripListItem.Category.OUTDOORS         ->         googleMap.addMarker(
MarkerOptions().position(coordinates).title(item.place).icon(
BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_GREEN)))
     TripListItem.Category.BEACHES          ->         googleMap.addMarker(
MarkerOptions().position(coordinates).title(item.place).icon(
BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_CYAN)))
     TripListItem.Category.SIGHTSEEING       ->       googleMap.addMarker(
MarkerOptions().position(coordinates).title(item.place).icon(
BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_YELLOW))
     TripListItem.Category.SKIING          ->          googleMap.addMarker(
MarkerOptions().position(coordinates).title(item.place).icon(
BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_AZURE)))
     TripListItem.Category.BUSINESS         ->         googleMap.addMarker(
MarkerOptions().position(coordinates).title(item.place).icon(
BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_VIOLET))
)}
bld.include(coordinates)

val bounds = bld.build()
googleMap.moveCamera(CameraUpdateFactory.newLatLngBounds(bounds, 1000,1000,
70))

googleMap.setMapStyle(MapStyleOptions(context.getString(
R.string.style_json)))
```

4.7 Út hozzáadása és módosítása

Az út hozzáadó és módosító nézetek is Compose-ban készültek. Felül három szövegdobozba lehet beírni az utazáshoz tartozó országot, helyet és leírást. Alatta lehet beállítani az utazás időpontját, majd kiválasztani a kategóriáját. Végül ki lehet pipálni, hogy az adott helyet már meglátogatta-e a felhasználó. Mivel spinner dátumválasztó jelenleg nincs Compose-ban, így ez XML alapú, ami AndroidView-vel alakul át Compose-ra.

A lista, naptár és a térkép nézetekben, jobb alul található lebegőgombra kattintva lehet új helyet felvenni. Ekkor az elemlétrehozó dialógus ablak jön fel, ahol az adatok kitöltését követően az ok gombra kattintva hozhatjuk létre az új utazást. Az utazási listában a kívánt helyhez tartozó ceruza jelölésű gombra kattintva módosíthatunk az adott hely adatai. Ekkor az elemmódosító dialógus ablak jön fel, ami annyiban különbözik az előzőtől, hogy megjelennek benne az adott hely jelenlegi adatai. A módosítások elvégzése után az ok gombra kattintva érvényesíthetjük a módosításokat. Mind létrehozásnál, mind módosításnál kötelező megadni hely nevét, máskülönben egy toast üzenet emlékezteti erre a felhasználót.



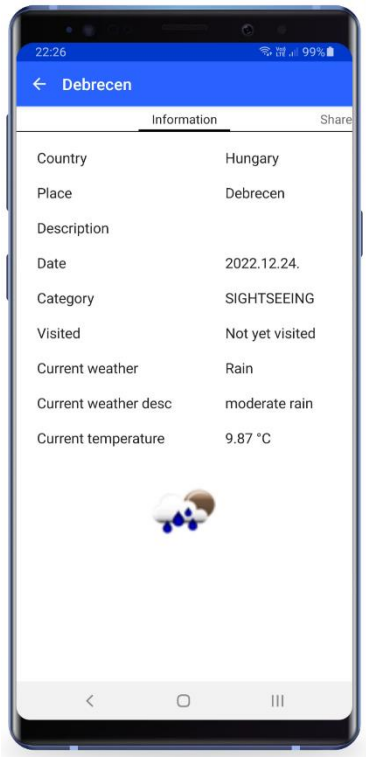
12. ábra Út hozzáadás és módosítás dialógus nézetek

4.8 Információk

A részletező információs nézetre az utazáslista nézetből tudunk átnavigálni a lista egyes elemeire rákattintva. Ebben a nézetben tekintheti meg a felhasználó az adott úthoz tartozó minden megadott információt, valamint az út helyének jelenlegi időjárását.

4.8.1 View

Az információs nézet teljes egészében Jetpack Compose használatával készült. Az oldalon megjelennek az adott úthoz tartozó adatok, amelyeket a felhasználó adott meg előzőleg: ország, hely, leírás, dátum, kategória és látogatottság. Ezek mellet pedig megjelennek egyéb adatok is, amik a helyhez tartozó pillanatnyi időjárást írják le: időjárás neve, időjárás leírása, hőmérséklet. Legalul pedig látható az időjáráshoz tartozó ikon is. Az ikon megjelenítéséért a GlideImage [17] felelős. A GlideImage képek betöltéséért felelős Compose osztály Glide [18] felhasználásával. A Glide egy gyors és hatékony, nyílt forráskódú médiakezelési és képbetöltési keretrendszer az Android számára, amely a média dekódolását, a memória és a lemez gyorsítótárazását, valamint az erőforrásösszevonást egy egyszerű és könnyen használható felületbe foglalja.



13. ábra Út információ nézete

4.8.2 Interactor

Az információs nézet időjárással kapcsolatos adatait az alkalmazás a részletekért felelős Interactor-ban található metódussal tölti be. Innen lehet elérni a hálózati kapcsolatért felelős repository-t, ahonnan az időjárási adatok érkeznek. Mielőtt azonban a hálózati kérést elindítaná a rendszer, az Interactor-ban leellenőrzi az internet kapcsolatot. Amennyiben nincs internet kapcsolat, nem indul el az időjárás lekérdezése és az információs nézetbe se töltődik be semmi.

Az időjárás információk lekérdezésének kódrészletei:

```
suspend fun getWeather(place: String, context: Context): WeatherData? {
     if(isConnected(context))
          return network.getWeather(place)
     else
          return null
}
```

4.8.3 Data source

Mindenféleképpen szerettem volna időjárás megjelenítést az egyes helyekhez az alkalmazás megtervezésekor, viszont ekkor még nem választottam ki, hogy az időjárási adatok milyen forrásból érkezzenek. Végül az OpenWeather [19] oldalt választottam, hiszen ennek a legtöbb funkciója ingyenes és van lehetőség ikonokat is lekérni, amik valamivel szebbé tehetik az információs nézetet. A hálózati kapcsolat kialakításáért és az kommunikáció megvalósításáért OkHttp-t és Retrofit-et használ az alkalmazás. Az OkHttp [20] egy hatékony HTTP-kliens, amely tartja magát, ha a hálózat problémákkal küzd, és csendben helyreáll az általános csatlakozási problémákból. A Retrofit egy magas szintű REST absztrakció, amely az OkHttp-re épül. A rendszer az OpenWeather API-jának segítségével létrehozza a kapcsolatot az anyaoldallal, ahonnan az utazás helynevének ismeretében lekéri az időjárási adatokat. Ha az adott helynévhez nem tartozik semmilyen időjárási adat, akkor a rendszer visszatér ezzel az információval és nem töltődnek be az részletező nézetben az időjárással kapcsolatos mezők és a kép.

Az adat osztályokban tárolt elemek, amelyeket később felhasznál az app:

- Weather main: a jelenlegi időjárás egy vagy két szavas leírása
- Weather description: a jelenlegi időjárás hosszabb leírása, kifejtése
- Weather icon: a jelenlegi időjárás ikonjának neve
- MainWeatherData temp: a jelenlegi hőmérséklet

Az időjárás adat osztályai:

```
data class WeatherData (
     var weather: List<Weather>? = null,
     var main: MainWeatherData? = null,
)

data class Weather (
     val id: Long = 0,
     val main: String? = null,
     val description: String? = null,
     val icon: String? = null
)

data class MainWeatherData (
     val temp: Float = 0f
)
```

A hálózati kapcsolat létrehozásáért és a kommunikációért felelős kódrészletek:

```
private const val SERVICE_URL = "https://api.openweathermap.org"

@Provides
@Singleton
fun provideOkHttpClient(): OkHttpClient = OkHttpClient.Builder()
     .addInterceptor {
          val original = it.request()
          val httpUrl = original.url()
          val newHttpUrl = httpUrl.newBuilder().addQueryParameter("appid",
BuildConfig.APP_ID).build()
          val request = original.newBuilder().url(newHttpUrl).build()
          it.proceed(request)
     }
     .build()

@Provides
@Singleton
fun provideRetrofit(okHttpClient: OkHttpClient):
Retrofit = Retrofit.Builder()
     .baseUrl(SERVICE_URL)
     .client(okHttpClient)
     .addConverterFactory(GsonConverterFactory.create())
     .build()

@Provides
@Singleton
fun provideApi(retrofit: Retrofit):
WeatherApi = retrofit.create(WeatherApi::class.java)

interface WeatherApi {
     @GET("/data/2.5/weather")
     suspend fun getWeather(
          @Query("q") cityName: String?,
          @Query("units") units: String?
     ): WeatherData?
}

suspend fun getWeather(city: String?): WeatherData? {
     return api.getWeather(city, "metric")
}
```

4.9 Kommentek

A megosztás nézetre az információs nézetről lehet átnavigálni csúsztatással. Ebben a nézetben tudja a felhasználó megtekinteni más felhasználók tanácsait, információit és élményeit az adott hellyel kapcsolatban. Ezen kívül a felhasználó is itt tudja saját gondolatait megosztani.

4.9.1 View

A kommentelő nézet is teljesen Jetpack Compose-ban készült. Az oldalon lista formájában jelennek meg a felhasználók által rögzített kommentek, melyeket egymás között, egymásnak osztottak meg az alkalmazás használói. Az egyes kommentek tetején látható a becenév, amelyet a kommentáló talál ki magának, ezzel úgymond névtelenül, de mégis névvel lehet tartalmakat megosztani. A becenév alatt látható a komment címe, mely félkövér betűstílust követ. Ezalatt pedig a tényleges komment helyezkedik el. A megosztott tartalmakat értékelni is tudják a felhasználók egymás között. Ha valaki hasznosnak talál egy megosztott kommentet, akkor ezt az alsó like jelre kattintva tudja jelezni a többi felhasználónak. A kommentek jobb felső sarkában levő ceruza és kuka ikonokra kattintva tudja az adatokat módosítani vagy törölni a kommentet létrehozó felhasználó. A lista elemei a kedvelések számának csökkenő sorrendjét követik.

14. ábra Út megosztás nézete

4.9.2 Interactor

A megosztásokért felelős Interactor-ban működnek a kommentelés legfontosabb metódusai: olvasás, írás, módosítás, törlés. Az adatbázisból innen lehet elérni az megosztások adatait, amelyeket a képernyőre tölt a rendszer.

A kommentek kezelésével kapcsolatos kódrészletek:

```
suspend fun getItems(place: String): Flow<List<SharedData>> {
     return firebaseDataSource.getItems(place)
}
suspend fun uploadPost(place: String, nick: String, title: String, comment:
String) {
     firebaseDataSource.onUploadPost(place, nick, title, comment)
}
suspend fun editPost(place: String, item: SharedData) {
     firebaseDataSource.onEditPost(place, item)
}
suspend fun deletePost(place: String, item: SharedData) {
     firebaseDataSource.onDeletePost(place, item)
}
```

4.9.3 Data source

Az adatbázis séma megalkotása ebben az esetben is az implementációs fázisban esett meg. Abban egészen biztos voltam, hogy a posztok adatait egy valós idejű, rugalmas, gyors és biztonságos adatbázisban szeretném tárolni, így esett a választás a Firestore adatbázisra. A Cloud Firestore listener-eken keresztül lehetőséget ad arra, hogy a felhasználók valós időben kapják meg az egymás között megosztott tartalmakat. Ezen kívül a megosztás offline állapotban is működőképes marad, ha esetleg a felhasználónak gondja akadna az internet eléréssel. Azért, hogy az adatok képesek legyenek valós időben eljutni az adatbázisból az Interactor-on keresztül a Fragment-be, a Kotlin Flow [21] felel. A korutinokra épülő Flow több értéket is képes kibocsátani szekvenciálisan. A feliratkozott listener-eket értesíti, ha értékváltozás történt. A feliratkozást képes automatikusan bontani, ezzel a megelőzve a memóriaszivárgást.

A SharedData adatosztály változói:

- id: A poszt azonosítója, amit a Firestore generál
- uid: A posztot készítő felhasználó azonosítója
- author: A posztot készítő felhasználó neve
- nickname: A posztot készítő felhasználó által beállított becenév
- title: A poszt címe
- body: A poszt komment része
- liked: A posztot kedvelő felhasználók azonosítóinak listája

Az elemek betöltéséért és valós idejű frissítésért felelős kódrészlet:

```
suspend fun getItems(trip: String): Flow<List<SharedData>> = callbackFlow {
     val listenerRegistration = database.collection(trip)
          .addSnapshotListener { querySnapshot: QuerySnapshot?,
firebaseFirestoreException: FirebaseFirestoreException? ->
               if (firebaseFirestoreException != null) {
                    cancel(message = "Error fetching items", cause =
firebaseFirestoreException)
                    return@addSnapshotListener
               }
               val items = mutableListOf<SharedData>()
               if (querySnapshot != null) {
                    for(document in querySnapshot) {
                         items.add(document.toObject())
                    }
               }
               this.trySend(items).isSuccess
          }
     awaitClose {
          Log.d("failure", "Cancelling items listener")
          listenerRegistration.remove()
     }
}
```

Az elemek egyszeri betöltéséért felelős kódrészlet:

```
suspend fun getItemsOnce(trip: String): List<SharedData> {
     val items = mutableListOf<SharedData>()

     database.collection(trip).get()
          .addOnSuccessListener { documents ->
               for(document in documents)
                    items.add(document.toObject())
          }
          .addOnFailureListener { exception ->
               Log.d("failure", "Error getting documents: ", exception)
          }
          .await()

     return items
}
```

Egy új elem adatbázisba töltéséért felelős kódrészlet:

```
suspend fun onUploadPost(trip: String, nick: String, title: String, comment:
String) {
     val newPost = SharedData(null, FirebaseAuth.getInstance()
.currentUser?.uid, FirebaseAuth.getInstance().currentUser?.displayName,
nick, title, comment)

     database.collection(trip).add(newPost)
          .addOnSuccessListener { documentReference ->
               Log.d("success", "DocumentSnapshot written with ID:
$documentReference.")
          }
          .addOnFailureListener { exception ->
               Log.d("failure", "Error getting documents: ", exception)
          }.await()
}
```

Egy létező elem módosításáért felelős kódrészlet:

```
suspend fun onEditPost(trip: String, item: SharedData) {
     if(item.uid == FirebaseAuth.getInstance().currentUser?.uid)
          database.collection(trip).document(item.id!!).set(item)
               .addOnSuccessListener { documentReference ->
               Log.d("success",   "DocumentSnapshot   written   with   ID:
$documentReference.")
               }
               .addOnFailureListener { exception ->
               Log.d("failure", "Error getting documents: ", exception)
          }.await()
}
```

Egy elem törléséért felelős kódrészlet:

```
suspend fun onDeletePost(place: String, item: SharedData) {
     if(item.uid == FirebaseAuth.getInstance().currentUser?.uid)
          database.collection(place).document(item.id!!).delete()
               .addOnSuccessListener { documentReference ->
                    Log.d("success",  "DocumentSnapshot  written  with  ID:
$documentReference.")
               }
               .addOnFailureListener { exception ->
               Log.d("failure", "Error getting documents: ", exception)
               }.await()
}
```

Egy új elem kedvelésének változtatásáért felelős kódrészlet:

```
suspend fun onLikePost(place: String, item: SharedData) {
     val userId = FirebaseAuth.getInstance().currentUser?.uid
     val userFound = item.liked.find { it == userId }
     if(!userFound.isNullOrEmpty()) {
          val poz = item.liked.indexOf(userFound)
          item.liked.removeAt(poz)
     }
     else {
          item.liked.add(userId!!)
     }
     database.collection(place).document(item.id!!)
          .update(mapOf(
               "liked" to item.liked
          ))
          .addOnSuccessListener { documentReference ->
          Log.d("success",    "DocumentSnapshot    written    with    ID:
$documentReference.")
          }
          .addOnFailureListener { exception ->
               Log.d("failure", "Error getting documents: ", exception)
          }.await()
}
```

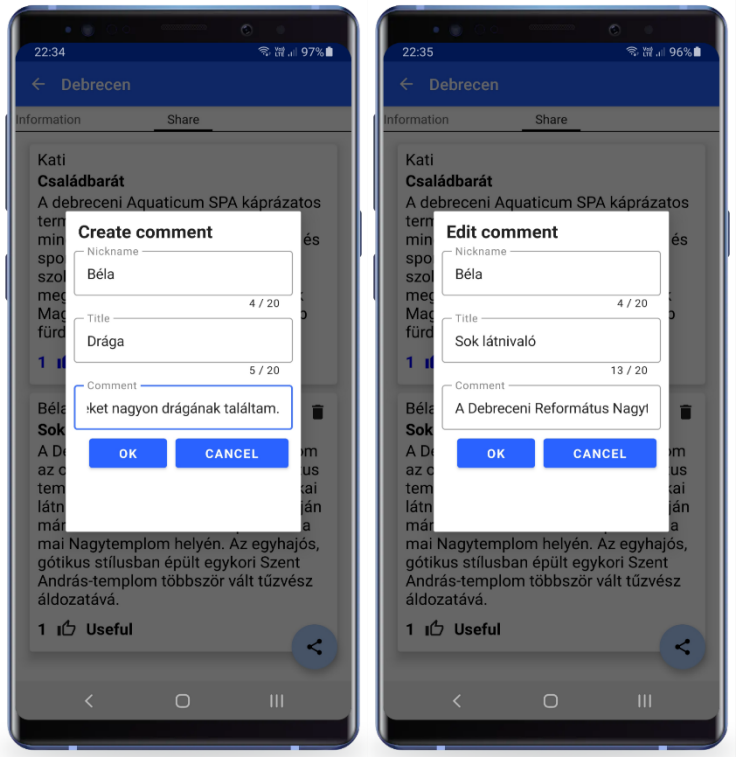
A kommentek adat osztálya:

```
data class SharedData (
     @DocumentId
     var id: String? = null,
     val uid: String? = null,
     val author: String? = null,
     var nickname: String? = null,
     var title: String? = null,
     var body: String? = null,
     var liked: MutableList<String> = mutableListOf())
```

4.10 Komment megosztása és módosítása

A komment megosztó és módosító nézetek is Compose-ban készültek. Három szövegdobozba kell beírni a szükséges adatokat. A felső szövegdobozba kerül a felhasználó beceneve, amit a többi felhasználó is láthat. A középső szövegdobozba jön a komment címe. Végül maga a komment szövege következik. Becenévre és komment címre 20-20 betűhatárt állítottam, amit a felhasználó is láthat a szövegdobozok alatt.

A megosztás nézetben, jobb alul található lebegőgombra kattintva lehet új kommentet létrehozni. Ekkor a kommentlétrehozó dialógus ablak jön fel, ahol az adatok kitöltését követően az ok gombra kattintva hozhatjuk létre az új kommentet. A kommentek listájában a kívánt kommenthez tartozó ceruza jelölésű gombra kattintva módosíthatjuk a kommentjeinket. Ekkor a kommentmódosító dialógus ablak jön fel, ami annyiban különbözik az előzőtől, hogy megjelennek benne az adott komment jelenlegi adatai. A módosítások elvégzése után az ok gombra kattintva érvényesíthetjük a módosításokat. Mind létrehozásnál, mind módosításnál kötelező megadni minden adatot, máskülönben egy toast üzenet emlékezteti erre a felhasználót.



15. ábra Komment megalkotás és módosítás dialógus nézetek

5 Tesztelés

Az alkalmazásfejlesztés egyik legfontosabb lépése a tesztelés, mely szerves részét képezi a folyamatnak. A folyamatos tesztelés megkönnyíti a design és a kódolási fázisokat, lerövidítve ezzel az alkalmazás elkészítésének idejét. A legnépszerűbb tesztelési mód Androidon a manuális tesztelés emulátoron, esetleg fizikai eszközön. Ez azonban nem a leghatékonyabb módja a hibák felderítésének. Könnyen figyelmen kívül lehet hagyni az alkalmazás viselkedésében jelentkező apróbb bug-okat, amik a fejlesztés előrehaladtával megnehezíthetik a fejlesztő munkáját. Az automatizált tesztelés megkönnyítheti a munkát, ugyanis könnyen meg lehet ismételni, gyorsabb és rendszerint hasznosabb munkát is végez, mint a manuális társa. Az alábbi fejezetben bemutatom az alkalmazás ViewModel és Presenter részeinek legfontosabb elemeihez írt automatikus teszteket.

5.1 ViewModel tesztek

A RainbowCake többféle lehetőséget is biztosít az automatikus tesztelés megalkotására az architektúra keretein belül. A fontosabb ViewModel-ek tesztelése közben nagyon sok lehetőséggel találkoztam a RainbowCake által javasoltak közül. Az egyik ilyen a ViewModelTest alaposztály, amely lecseréli az execute metódus által használt Main diszpécsert a teszt diszpécserre, valamint lecseréli a belső LiveData végrehajtót egy álvégrehajtóra, amely mindent egyetlen szálon hajt végre. Emellett a tesztek készítésénél gyakran használtam az observeStateAndEvents függvényt, ami figyeli a bemenetre adott reakciókat, mind a ViewModel állapotainak változását, mind az események változását. A függvény ezeket az állapot és esemény változásokat adja vissza, amivel már lehetőségünk is van tesztelni.

Az alsó példákban bemutatom a ViewModel-ek tesztelésének folyamatát. Első lépésként inicializálni kell a ViewModel-t és mockolni a Presenter-t, hogy ne zavarjon bele a ViewModel tesztelésébe. A mockolt Presenter használt metódusainak visszatérési értéket a coEvery függvény ad. Ezt követően jöhet az állapotok változásának vizsgálata. A kezdőállapot minden ViewModel esetében Loading, amiben nem történik lényeges esemény. Ezt követően jöhet a lényegi teszt, amiben a ViewModel egy metódusát kell meghívni és azt vizsgálni, hogy hogyan változik az állapota.

A kommentek betöltéséért felelős metódus tesztelésének kódrészlete:

```
@Before
fun initEach() {
     sharePresenter = mockk()
     viewModel = ShareViewModel(sharePresenter)
}

companion object {
     private val MOCK_POST = SharedData( nickname  = "Béla", title = "Szép",
body = "Nagyon szép volt",
     )
}

@Test
fun shareLoadedTest() = runTest {
     coEvery     {     sharePresenter.getItems("Paris")     }     returns
flowOf(listOf(MOCK_POST))
     coEvery { sharePresenter.getCurrentUser() } returns ("0")

     viewModel.observeStateAndEvents { stateObserver, _ ->
          stateObserver.assertObserved(Loading)
     }

     viewModel.observeStateAndEvents { stateObserver, _ ->
          viewModel.setShare("Paris")
          advanceUntilIdle()
          stateObserver.assertObserved(Loading,            ShareContent(),
ShareContent(listOf(MOCK_POST), "0", false))
     }
}
```

Az utazáslista betöltéséért felelős metódus tesztelésének kódrészlete:

```
@Before
fun initEach() {
     tripsPresenter = mockk()
     viewModel = TripsViewModel(tripsPresenter)
}

companion object {
     private  val  MOCK_TRIP  =  TripListItem(place  =  "Paris",  country  =
"France",   description   =   "",   date   =   "2022.01.01.",   category   =
TripListItem.Category.SIGHTSEEING, visited = true )
}

@Test
fun tripsLoadedTest() = runTest {
     coEvery { tripsPresenter.load() } returns listOf(MOCK_TRIP)

     viewModel.observeStateAndEvents { stateObserver, _ ->
          stateObserver.assertObserved(Loading)
     }

     viewModel.load()
     coVerify(exactly = 1) { tripsPresenter.load() }
     viewModel.observeStateAndEvents { stateObserver, _ ->
          stateObserver.assertObserved(TripsContent(trips                =
listOf(MOCK_TRIP), loading = false))
     }
}
```

5.2 Presenter tesztek

A Prestenter-ek tesztelésénél szintén a RainbowCake által ajánlott alaposztályt, a PresenterTest-et alkalmaztam. Ez lecseréli a Presenter-ekben alkalmazott IO diszpécsert a teszt diszpécserére, hogy a végrehajtás azonnal végbe mehessen a gyorsabb és hatékonyabb tesztelés érdekében.

Az alsó példában bemutatom a Presenter-ek tesztelésének folyamatát. Első lépésben a Presenter-t kell inicializálni, valamint mockolni minden Interactor-t, ami kapcsolatban áll az adott Presenter-rel. A mockolt Interactor-ok használt metódusainak visszatérési értéket a coEvery függvény ad. Végül a meg kell hívni Presenter tesztelni kívánt metódusát, aminek a visszatérési értékét az assertThat metódus teszteli.

Az kommentek betöltéséért felelős metódus tesztelésének kódrészlete:

```
@Before
fun initEach() {
     shareInteractor = mockk()
     authInteractor = mockk()
     presenter = SharePresenter(shareInteractor, authInteractor)
}

companion object {
     private val MOCK_POST = SharedData(
          nickname = "Béla",
          title = "Szép",
          body = "Nagyon szép volt",
     )
}

@Test
fun shareGetItemsTest() = runBlocking {
     val flowItems = flowOf(listOf(MOCK_POST))
     coEvery { shareInteractor.getItems("Paris") } returns flowItems

     val items = presenter.getItems("Paris")

     assertThat(items).isEqualTo(flowItems)
}
```

6 Összefoglalás

A szakdolgozatom keretein belül teljesítettem az előre meghatározott elvárásokat, amelyeket a fenti fejezetekben bővebben is kifejtettem. A 3. fejezetben bemutattam, hogy hogyan terveztem meg az alkalmazás felépítését és architektúráját, valamint megindokoltam a fontosabb tervezői döntéseket. A 2. fejezetben bemutattam az összes fontosabb, nem triviális technológiát, amelyeket felhasználtam az alkalmazás elkészítésekor. A 4. fejezetben bővebben is kifejtettem a megvalósított alkalmazást. Végül az 5. fejezetben szót ejtettem az alkalmazás teszteléséről is. Végső soron elégedett vagyok az elkészült munkámmal.

Az alkalmazást több téren is tovább lehetne fejleszteni. Szerintem a legfontosabb lenne a térkép nézet használatát továbbfejleszteni. Be lehetne vezetni, hogy a térkép csak egy megadott felhasználói interakció hatására frissüljön, mondjuk Swipe Refresh beépítésével. Emellett feldobná a térképet egy útvonaltervező alkalmazása is. Helyekhez tartozó információk, látnivalók, étkezési lehetőségek megjelenítését is be lehetne építeni az alkalmazásba. Mindemellett a kommenteléshez is hozzá lehetne adni még új funkciókat, pl. negatív értékelésre vagy média tartalmak megosztására is lehetne lehetőséget adni a felhasználók részére.

Irodalomjegyzék

[1] Firebase Authentication https://firebase.google.com/docs/auth
Hozzáférés dátuma: 2022.11.26.

[2] Firebase Firestore Database https://firebase.google.com/docs/firestore
Hozzáférés dátuma: 2022.11.26.

[3] Room Database https://developer.android.com/training/data-storage/room
Hozzáférés dátuma: 2022.11.26.

[4] SQLite https://www.sqlite.org/ Hozzáférés dátuma: 2022.11.26.

[5] Google Maps https://developers.google.com/maps/documentation/android-sdk
Hozzáférés dátuma: 2022.11.26.

[6] Geocoder https://developer.android.com/reference/android/location/Geocoder
Hozzáférés dátuma: 2022.11.26.

[7] Retrofit https://square.github.io/retrofit/ Hozzáférés dátuma: 2022.11.26.

[8] Jetpack Compose https://developer.android.com/jetpack/compose
Hozzáférés dátuma: 2022.11.26.

[9] RainbowCake https://rainbowcake.dev/ Hozzáférés dátuma: 2022.11.26.

[10] Dagger https://developer.android.com/training/dependency-injection/dagger-android Hozzáférés dátuma: 2022.11.26.

[11] Hilt https://developer.android.com/training/dependency-injection/hilt-android
Hozzáférés dátuma: 2022.11.26.

[12] MockK https://mockk.io/ Hozzáférés dátuma: 2022.11.26.

[13] Tripadvisor https://www.tripadvisor.com/ Hozzáférés dátuma: 2022.11.26.

[14] Google Travel https://www.google.com/travel/ Hozzáférés dátuma: 2022.11.26.

[15] MockFlow https://www.mockflow.com/ Hozzáférés dátuma: 2022.11.26.

[16] Applandeo Calendar https://github.com/Applandeo/Material-Calendar-View
Hozzáférés dátuma: 2022.11.26.

[17] GlideImage https://github.com/skydoves/landscapist
Hozzáférés dátuma: 2022.11.26.

[18] Glide https://github.com/bumptech/glide Hozzáférés dátuma: 2022.11.26.

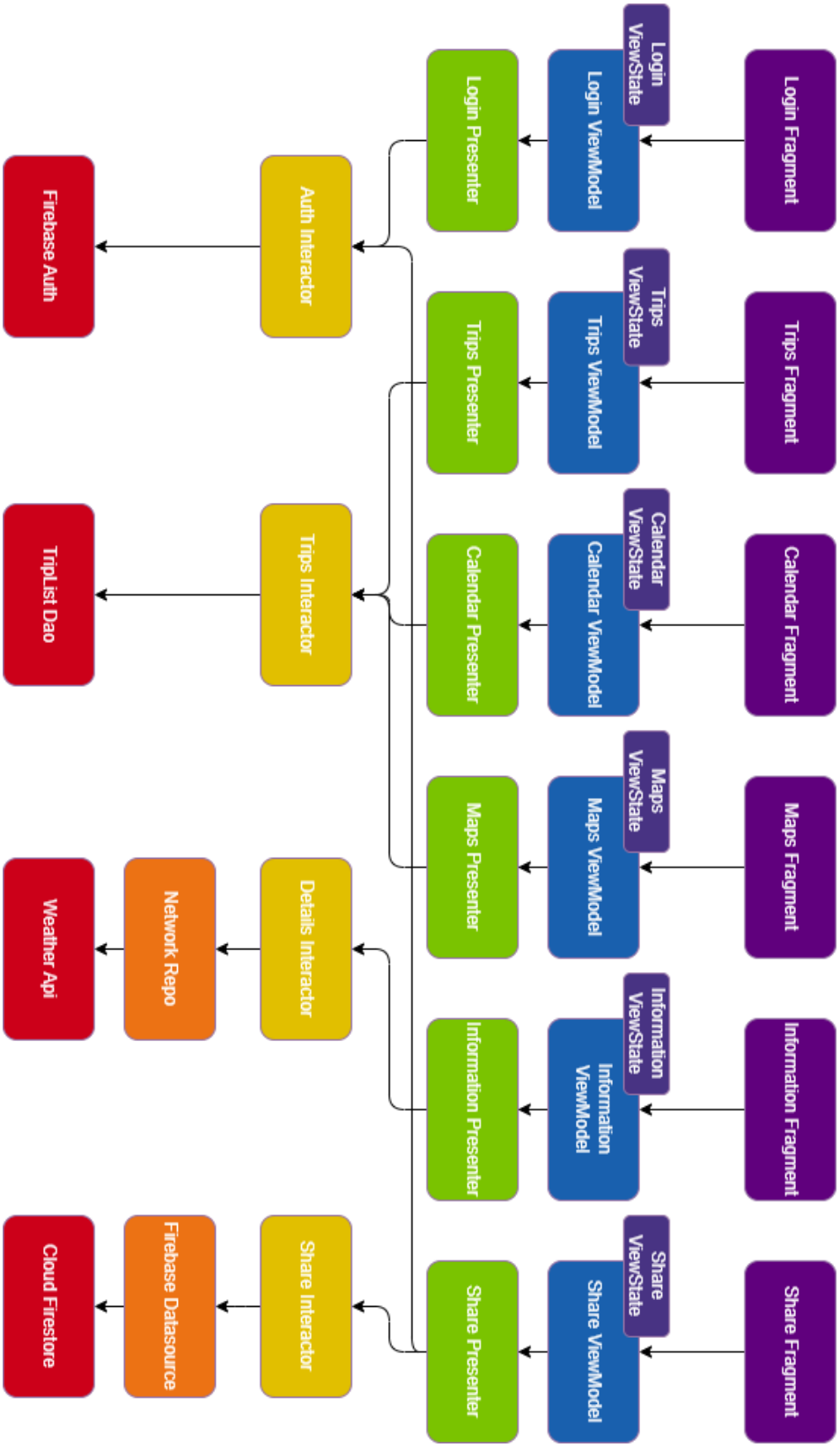
[19] OpenWeather https://openweathermap.org/ Hozzáférés dátuma: 2022.11.26.

[20] OkHttp https://square.github.io/okhttp/ Hozzáférés dátuma: 2022.11.26.

[21] Flow https://developer.android.com/kotlin/flow Hozzáférés dátuma: 2022.11.26.

Képjegyzék

Architektúra diagram



16. ábra Architektúra diagram az alkalmazásról