



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

# Kvíz generálási lehetőségek vizsgálata egy fullstack megoldás megvalósítása keretében

SZAKDOLGOZAT

*Készítette*  
Muzslai László

*Konzulens*  
dr. Ekler Péter

# Tartalomjegyzék

<b>Kivonat</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>1. Bevezetés</b>	<b>1</b>
1.1. Témaválasztás indoklása . . . . .	1
1.2. Felhasznált technológia jelentősége/elterjedtsége . . . . .	2
1.3. Dolgozat felépítése . . . . .	3
<b>2. Feladatspecifikáció</b>	<b>5</b>
2.1. Feladat részletes leírása . . . . .	5
2.1.1. Általános célok bemutatása . . . . .	5
2.1.2. Egy feladatsor felépítése . . . . .	6
2.1.3. Komponensek létrehozásának lépései . . . . .	6
2.1.3.1. Pont . . . . .	6
2.1.3.2. Témakör . . . . .	7
2.1.3.3. Igaz-hamis kérdés . . . . .	7
2.1.3.4. Feleletválasztós kérdés . . . . .	8
2.1.3.5. Feladatsor . . . . .	8
2.1.3.6. Válaszok ellenőrzése . . . . .	9
2.1.3.7. Feladatsorok exportálása . . . . .	9
2.2. Diagramok . . . . .	9
<b>3. Irodalomkutatás</b>	<b>11</b>
3.1. Felhasznált technológiák . . . . .	11
3.1.1. Jetpack Compose . . . . .	11
3.1.1.1. State és StateFlow . . . . .	12
3.1.1.2. ViewModel . . . . .	13
3.1.1.3. Navigation and routing . . . . .	15
3.1.2. Ktor . . . . .	16

3.1.3.	KotlinX Serilizáció . . . . .	16
3.1.4.	CameraX . . . . .	17
3.1.5.	ML-Kit . . . . .	17
3.1.6.	Acccompanist-engedélykezelés . . . . .	17
3.1.7.	PdfDocument és PDFBox . . . . .	17
3.1.8.	Kotlin- és Compose Multiplatform . . . . .	18
3.1.9.	Gradle build rendszer . . . . .	19
3.1.10.	Fejlesztőkörnyeztek . . . . .	20
3.1.11.	REST API, Postman és adatbázis . . . . .	21
3.1.12.	Kipróbált, de végül nem használt egyéb érdekes megoldások . . . . .	22
3.2.	Hasonló multiplatform megoldások összehasonlítása . . . . .	22
<b>4.</b>	<b>Felsőszintű architektúra</b>	<b>24</b>
4.1.	High level architektúra . . . . .	24
4.1.1.	A program struktúrális szervezése . . . . .	24
4.1.1.1.	A Kotlin Multiplatform alkalmazások felépítése . . . . .	24
4.1.1.2.	Követett tervezési minták . . . . .	26
4.2.	Rendszer felépítései, komponensei . . . . .	31
4.2.1.	Általános komponensek felépítése . . . . .	31
4.2.2.	Business logic komponensek felépítése . . . . .	32
4.2.3.	Common Client komponensek felépítése . . . . .	32
<b>5.</b>	<b>Részletes megvalósítás</b>	<b>34</b>
5.1.	Compose . . . . .	34
5.1.1.	Compose alapelvek, használata valós környezetben . . . . .	34
5.1.2.	Alkalmazás bemutatása képernyőképekkel . . . . .	36
5.1.2.1.	A fő képernyők . . . . .	36
5.1.2.2.	Elemek listázása . . . . .	37
5.1.2.3.	Részletes nézet . . . . .	38
5.1.2.4.	Szerekesztő nézet . . . . .	40
5.1.2.5.	Új elem létrehozása . . . . .	42
5.1.2.6.	Ellenőrző képernyő . . . . .	42
5.1.3.	Szekevencia diagram egy átlagos interakcióra . . . . .	43
5.2.	Navigáció . . . . .	44
5.3.	ViewModel . . . . .	45
5.4.	HTTP kliens . . . . .	46
5.5.	PDF . . . . .	47

5.6. Szöveg felismerés . . . . .	47
<b>6. Továbbfejlesztési lehetőségek</b>	<b>49</b>
<b>7. Összefoglalás</b>	<b>50</b>
<b>Köszönetnyilvánítás</b>	<b>51</b>
<b>Irodalomjegyzék</b>	<b>53</b>
<b>Függelék</b>	<b>54</b>
F.1. A TeXstudio felülete . . . . .	54
F.2. Válasz az „Élet, a világmindenség, meg minden” kérdésére . . . . .	55

## HALLGATÓI NYILATKOZAT

Alulírott *Muzslai 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 autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedélyel titkosított diplomatervezet esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2024. november 21.

---

*Muzslai László*  
hallgató

# Kivonat

A közoktatásban és a felsőoktatásban is gyakori probléma mind a tanárok, mind a diákok számára az időhiány a rengeteg munka miatt. Ez az alkalmazás a tanárok munkáját hivatott segíteni az előző félévben elkészített REST API-t felhasználva, és a hozzá írt Android-eszközökre készült program továbbfejlesztése által. Az alkalmazás lehetőséget biztosít egy nagyméretű kérdésbank létrehozására és tárolására. A kérdések bármikor módosíthatók, törölhetők, vagy hozhatók létre újak. Természetesen nem csak egy ember dolgozhat ugyanazon a tárgyon; a kérdésbank és a számonkérések közösen szerkeszthetők.

Az alkalmazás lehetőséget biztosít a kérdések létrehozása mellett egyéni téma-körök létrehozására, amivel a kérdéseket és feladatsorokat lehet csoportosítani. Továbbá el lehet készíteni a saját pontrendszeret, akár több félét is, amelyet dinamikusan lehet változtatni a kérdésekkel. Fontos szempont volt az automatizált javítás segítése, így csak egyszerű kérdések vannak: igaz-hamis és feleletválasztós kérdések. Sajnos az AI még nem tart ott, hogy bármilyen kézírást pontosan felismerjen, és ebből a szövegből megállapítsa annak helyességét. Ennek ellenére a szöveg-felismerő funkció így is támogatja a javítást, aminek az eredményét megkapja a javító.

Ezekből az elemekből áll össze a számonkérés. Ez a szoftver csak a kérdéssorok összeállításáért és kiértékeléséért felel. Ennek megfelelően elő kell állítani magát a feladatsort. Egy dolgozatot ki lehet exportálni PDF formátumban, erről egy előnézet is lesz, amin nagyjából látszik, hogyan fog kinézni, de a végső változat csak az exportálást követően fog látszani. Ezt követően szabadon nyomtathatóvá válik.

Egy modern szoftver esetén elvárt, hogy könnyen és kényelmesen lehessen kezelni, mindenki számára a neki tetsző módon. Ennek alapján úgy döntöttem, hogy felhasználom az Android fejlesztésben szerzett tapasztalataimat. 2021 augusztusában jelent meg a Compose Multiplatform technológia, amely kedvez az Android-fejlesztőknek, mivel a natív Android-megoldások könnyen átültethetők egy cross-platform alkalmazásba. Jelenleg stabilan működik Android-, asztali- és iOS-alkalmazások készítéséhez, eszköz hiányában az első kettőt készítettem el.

# Abstract

In both public and higher education, time constraints are a common issue for both teachers and students due to the large workload. This application is intended to assist teachers by utilizing the REST API developed in the previous semester and enhancing the program created for Android devices. The application allows for the creation and storage of a large question bank. Questions can be modified, deleted, or new ones can be created at any time. Of course, more than one person can work on the same subject; the question bank and the tests can be edited collaboratively.

In addition to creating questions, the application also allows for the creation of custom topics, which can be used to organize questions and assignments. Furthermore, a custom scoring system can be created, even multiple types, which can be dynamically adjusted for different questions. An important aspect was to assist in automated grading, so only simple questions are included: true/false and multiple-choice questions. Unfortunately, AI is not yet at the level where it can accurately recognize any handwriting and determine its correctness from the text. Nonetheless, the text recognition function still supports grading, and the results are provided to the grader.

These elements come together to form the assessment. This software is responsible solely for compiling and evaluating the question sheets. Accordingly, the task sheet itself must be generated. A test can be exported in PDF format, with a preview available that roughly shows how it will look, though the final version will only be visible after exporting. After this, it can be freely printed.

For modern software, it is expected to be easy and convenient to use, allowing everyone to handle it in their preferred way. Based on this, I decided to leverage my experience in Android development. The Compose Multiplatform technology, released in August 2021, is favorable for Android developers, as native Android solutions can easily be adapted into a cross-platform application. It currently works stably for creating Android, desktop, and iOS applications; due to a lack of devices, I have implemented the first two.

## 1. fejezet

# Bevezetés

Ebben a fejezetben kitérek arra, hogy miért ezt a témát választottam. Utána bemutatom a használt Compose Multiplatform jelentősséget és elterjedtségét. A legvégén összefoglalom a dolgozat ezt követő témáit.

### 1.1. Témaválasztás indoklása

2023/2024 őszi félévében ismerkedtem meg a mobil, azon belül is az Android fejlesztéssel. Az ezt követő félévben tovább mélyítettem a tudásomat ebben a témaban, az Önálló laboratórium tárgy keretein belül elkezdtem fejleszteni a szakdolgozatom alapjaként szolgáló alkalmazást. Szintén ebben a félévben hallgattam az Android alapú szoftverfejlesztés és a Kotlin alapú szoftverfejlesztés tárgyakat, amik segítettek jobban megérteni ezt a területet. Az így elsajátított tudás és az önálló kutatás és tanulás során előállt egy Kotlin nyelven írt REST API, ami egy PostgreSQL adatbázissal biztosít kommunikációt, és természetesen egy Android alkalmazás, amelyet a szakdolgozat során tovább bővítettem és alakítottam át egy cross-platform szoftverré.

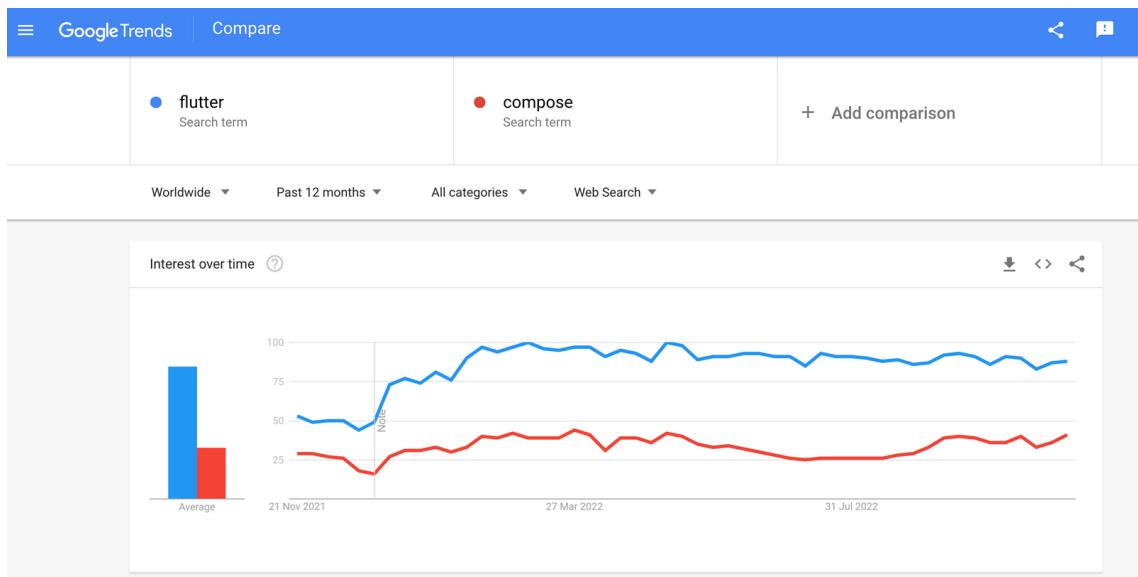
A kérdéssor összeállító alkalmazás ötletét a konzulensem vetette fel, hasznos lenne, ha létezne egy ilyen eszköz, amivel könnyen megoldható ez a feladat. Megtetszett nekem is az ötlet, mivel vannak ismerőseim és családtagjaim, akik szintén tudnának egy ilyen alkalmazást hasznosítani a munkájuk vagy egyéb elfoglaltságaik kapcsán. Egy nagyobb méretű fullstack alkalmazás előállítása túlmutat az Önálló laboratórium keretein, így rengeteg fejlesztési ötlet és lehetőség nem fért bele a félévbe. Továbbgondolva ezt a projektet, folytattam a munkát a szoftveren. Ezen kívül minden szeretek új és érdekes dolgokat kipróbálni, és ha megtetszik, alaposan tanulmányozni és megtanulni. Pont ezért választottam a Google és a JetBrains legújabb megoldásait a szakdolgozathoz.

A most elkészített szoftvert jelentősen tovább lehet még fejleszteni, felhasználók, szervezetek regisztrálásával és elkülönítésével, több fajta kérdéstípus megvalósításával. Bevezetni a szervezeteken belül az oktató és diákok csoportokat, és egy online kitöltési formát is megtervezni, létrehozni. Az Önálló laboratórium alatt is élveztem ezzel foglalkozni, és még mindig szívesen fejleszteném tovább, és tenném jobbá az alkalmazást.

## 1.2. Felhasznált technológia jelentősége/elterjedtsége

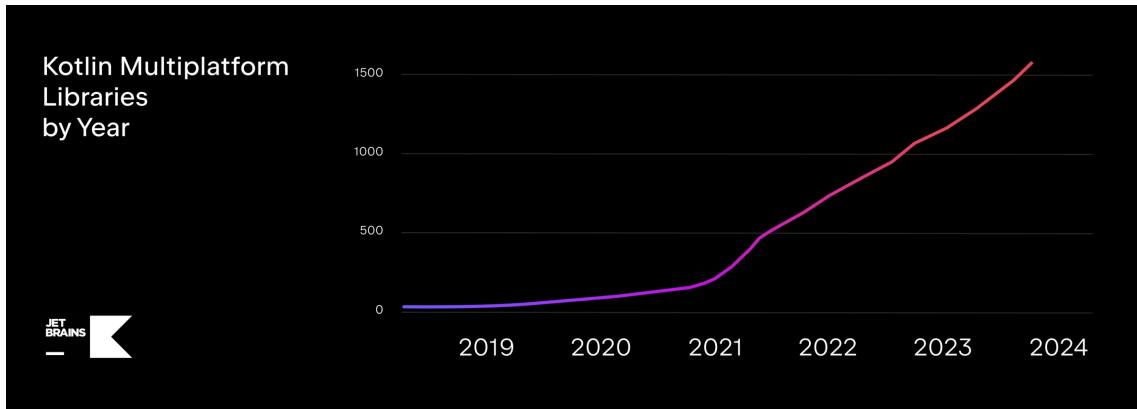
Megfigyelve a mai trendeket, láttam, hogy a multiplatform fejlesztés egyre népszerűbb, mivel gyorsabban és hatékonyabban lehet egyszerűbb alkalmazásokat elkészíteni több fajta felhasználói réteg számára. Úgy döntöttem, hogy kipróbálok egy új megoldást, ami a Compose Multiplatform; 2021 augusztusában jelent meg az 1.0 alpha verziója, és ezt a Kotlin Multiplatform egy évvel korábbi megjelenése tette lehetővé. Az Android fejlesztők körében kifejezetten népszerű lett, annak ellenére is, hogy még mindig van rengeteg funkció, amit nem támogat, de jelenleg is aktívan fejlesztik, és válik hónapról-hónapra egyre jobbá. A fejlesztések havonta / néhány havonta érkeznek minden a Kotlin nyelvhez és a Kotlin Multiplatformhoz[26], minden a Compose Multiplatformhoz[19]. Jelenleg vannak sokkal jobban elterjedt, használthatóbb cross-platform keretrendszerök, mint például a Flutter (1.1. ábra) vagy a React Native, ami iOS és Android fejlesztést tesz lehetővé JavaScript/TypeScript nyelven, így a webes fejlesztők gyakran használják natív mobilos appokhoz.

Szerencsére a fejlesztés folyamatos és gyors, amit segít az is, hogy sok nyílt forráskódú könyvtár is készül a felhasználók által.[23] A Compose Multiplatform ennek az ellenpontja lesz[10]; sajnos a webes támogatás még csak alpha verzióban van, így elég instabil, és sok olyan eszköz nem használható még, ami a többi területen már stabilan működik, így ezzel egyelőre a szakdolgozat keretein belül nem foglalkoztam részletesebben a webes megoldás megvalósításával.



1.1. ábra. A Flutter és a Compose keresési arányai [2]

A visszajelzések és a mostani szoftverfejlesztési irányokból arra lehet következtetni, hogy még hosszú jövő áll a technológiák előtt. Könnyen, gyorsan és hatékonyan lehet akár egyszerre az összes platformra alkalmazást fejleszteni, nagy méretű közös és kis méretű natív kódbazis írásával és karbantartásával, összesen két nyelv ismeretével. Webre, Androidra és asztali alkalmazáshoz elegendő lehet a Kotlin nyelv, esetleg egy kis HTML és JavaScript ismeret; iOS esetén minimális Swift és SwiftUI ismeret jól jöhet, de ez hasonlít a Kotlinra és a Compose-ra. Az a tény, hogy a Compose Multiplatform csupán Kotlin nyelven írt kódzással képes natívan megjeleníteni az elkészült alkalmazást minden platformon, nagyon erős eszközéssé teszi. Például az Androidon futó alkalmazás az Androidos



**1.2. ábra.** A könyvtárak növekedésének gyors üteme [23]

gombokat, görgetést stb. használja, míg iOS-en az ott megszokott stílust és irányítást kapja a felhasználó.

### 1.3. Dolgozat felépítése

Az alábbiakban bemutatom a dolgozat felépítését.

1. *Feladatspecifikáció: (2. fejezet)*
  - (a) *Feladat részletes leírása: (2.1. szakasz)*
  - (b) *Diagramok: (2.2. szakasz)* Bemutatók néhány diagramot és ábrát, amelyek segítik az alkalmazás felépítésének megértését.
2. *Irodalomkutatás: (3. fejezet)*
  - (a) *Felhasznált technológiák: (3.1. szakasz)* az 1.2. szakaszban leírtak és a használt könyvtárak részletes bemutatása
  - (b) *Hasonló megoldások: (3.2. szakasz)* Rövid összevetés a többi hasonló cross-platform megoldással.
3. *Felsőszintű architektúra: (4. fejezet)* Nagyvonalakban bemutattam az alakalmazás fő vázát és felépítését.
  - (a) *High level architektúra: (4.1. szakasz)*
  - (b) *Rendszer felépítései, komponensei: (4.2. szakasz)*
4. *Részletes megvalósítás: (5. fejezet)* Kitérek minden az eddigiekben bemutatott technológia pontos használatára, és arra, hogy nálam milyen formában jelentek meg.
  - (a) *Compose: (5.1. szakasz)*
  - (b) *Navigáció: (5.2. szakasz)*
  - (c) *ViewModel: (5.3. szakasz)*
  - (d) *HTTP kliens: (5.4. szakasz)*
  - (e) *PDF: (5.5. szakasz)*
  - (f) *Szövegfelismerés: (5.6. szakasz)*

5. Továbbfejlesztési lehetőségek: (6. fejezet)

6. Összefoglalás: (7. fejezet)

## 2. fejezet

# Feladatspecifikáció

Ebben a fejezetben bemutatom az alkalmazás céljait. Az első részben általános összefoglalom a felhasználói elvárásokat és legfőbb építőköveket a programzó szemszögéből. Ezt követően megmutatom a használati Usecaseeket.

Kitérek a főbb funkciókra, azok felépítésére és elvárt használati módjaira. Az utolsó fejezetben mindezt egy Usecase UML diagramban összefoglalom. Megtalálható benne az összes fent leírt használati eset, jól látható lesz, hogy a funkciók hasonlítanak egymásra így a felhasználó könnyen meg tudja tanulni az alkalmazás használatát.

### 2.1. Feladat részletes leírása

Ebben az alszakaszban részletesen kitérek az alkalmazás komponenseinek megtervezésére, és azok használatára. Bemutatom, hogy milyen egy feladatsor felépítése, és annak a részei milyen mezőket vagy értékeket tartalmaznak. Ebben a részben szövegesen végig éhet kísérni a 2.1. ábrán látható felépítést.

#### 2.1.1. Általános célok bemutatása

A program célja az, hogy egy vagy több felhasználó létre tudjon hozni igaz-hamis illetve feleletválasztós (tetszőleges számú válaszopciónal) kérdésekkel álló feladatsort. Az alkalmazás csak papír alapú kitöltést támogat, ezt olyan formában teszi, hogy az összeállító a kész kérdéssort ki tudja exportálni PDF formátumba.

A szoftver a fent leírtakon kívül rendelkezik egy automatizált javítási rendszerrel, okos-telefont használva a Google ML Kit[16] segítségével be lehet scannelni egy megfelelően formázott választ, és az belekerül egy form-ba amit a szervernek elküldve visszaküldi az eredményt. A kézirásos szöveg felismerés sok esetben nem szolgáltatott megfelelően pontos eredményt, így ez a funkció csak alpha verzióban támogatott. Hibásan felismert, vagy felismerhetetlen válaszok esetén kézzel is szerkeszthető a válasz a kiértékelés előtt.

Az alkalmazás a főmenüből indul, innen lehet tovább navigálni az összes opcióhoz. A főmenü máshogy néz ki a használt eszköztől függően. A választásunk után egy listázó nézet tárul elénk, ahol látjuk az adott opcióhoz tartozó összes eddig felvett elemet. Itt tudunk új elemet is hozzáadni az adott kategórihoz, illetve a listán történő kattintással a részletes nézet tárul elénk. A részletes nézeten minden adatot egyszerre látunk, itt tudjuk törlni és módosítani is. Mind a létrehozás és a szerkesztés során \*-gal vannak jelölve a kötelező értékek. Törlés során van egy figyelemfelhívó ablak is, mivel a törlés az végleges és

nem vonható vissza. Vannak egyediséget megkívánó mezők, így amennyiben már létezik a megadott értékkel egy felvett elem, jelzi a za alkalmazás, hogy ezt módosítani kell mentés előtt.

A szoftver működési elve és kommunikációja röviden összefoglalva az alábbi. A felhasználó megnyitja az alkalmazást, majd interaktál akezelő felülettel. A kattintások során amik igénylik az adatbázis elérést (listázás, részletes nézet megjelenítése, új elem létrehozása, szerkesztés, törlés) az alaklamazás szabványos HTTP kéréseket intéz a REST API-hoz. Az adatok forgalma szabványos JSON formátummal zajlik minden adatküldés, minden adatok fogadása során minden irányban. A REST API és az adatbázis is egy virtuális gépen fut egy-egy Docker konténerben.

### 2.1.2. Egy feladatsor felépítése

Egy feladatsor a következő elemekből épül fel:

- *Témakör*
- *Kérdések:*
  - *Igaz-hamis:* A szokásos egyszerű igaz-hamis típusfeladat.
    - \* *Kérdés:* Meg kell adni magát az eldöntendő kérdést.
    - \* *Pontozási módszer:* Egyedi pontozási módszert lehet létrehozni. Beállítható az összpontszám és a helyes és helytelen válaszokra adott pontérték, amely lehet negatív is.
    - \* *Témakör:* Segít a kérdések kategorizálásában és szűrésében az összeállítás során.
    - \* *Helyes válasz:* Szükséges a javítás elvégzéséhez.
  - *Feleletválasztós:* A szokásos egyszerű feleletválasztós típusfeladat, testreszabható mennyiségű válaszopciónal.
    - \* *Kérdés:* Meg kell adni magát a kérdést, több helyes válasz is lehetséges.
    - \* *Pontozási módszer:* Egyedi pontozási módszert lehet létrehozni. Beállítható az összpontszám és a helyes és helytelen válaszokra adott pontérték, amely lehet negatív is.
    - \* *Témakör:* Segít a kérdések kategorizálásában és szűrésében az összeállítás során.
    - \* *Válaszok:* Meg kell adni a válaszopciónal.
    - \* *Helyes válasz(ok listája):* Szükséges a javítás elvégzéséhez.

### 2.1.3. Komponensek létrehozásának lépései

Az alábbiakban bemutatom az egyes alkotóelemek életútját az alkalmazáson belül.

#### 2.1.3.1. Pont

A pontok listáját az ennek megfelelő menüpont kiválasztása után érjük el. Itt egy floating button segítségével adhatunk új pontot az alkalmazáshoz. A létrehozáshoz, amelyet szintén egy floating button segítségével érhetünk el a részletes nézetről, minden mező kitöltése kötelező, csak így biztosítható megfelelően az elvárt működése a javítási funkció miatt.

- *Típus/Név:* A pont típusa vagy neve. Ez egyedi mező is egyben, így lehet rá hivatkozni és megtalálni az alkalmazásban.
- *Pont:* A feladatra adható maximális pontszám.
- *Helyes válasz:* A helyes válaszokra adható pont, ajánlott úgy megvalósítani a pontozást, hogy ezeknek az összege a teljes pontszám legyen.
- *Helytelen válasz:* A helytelen válaszok során levont pontmennyiségek. Amennyiben nincs levonás, az értéke 0; egyébként egy negatív szám.

A pont létrehozása után látható lesz a listás nézetben, ahol kiválasztva ellenőrizhetjük a megadott értékeket, szükség esetén módosíthatjuk is. A pontokat törölni is lehet, de csak akkor, ha egyetlen kérdéshez sem használjuk, különben inkonzisztens állapot alakulna ki és pont nélküli kérdések keletkeznének. Mindkét műveletet a részletes oldalon tehetjük meg.

#### **2.1.3.2. Témakör**

A témaik listáját az ennek megfelelő menüpont kiválasztása után érjük el. Itt egy floating button segítségével adhatunk új témakört az alkalmazáshoz. A létrehozáshoz, amelyet szintén egy floating button segítségével érhetünk el a részletes nézetről, minden mező kitöltése kötelező, csak így biztosítható megfelelően az elvárt működés.

- *Témakör neve:* A témakör megnevezése, egyedi mező.
- *Témakör leírása:* Kötelező egy rövid leírást adni az egyértelműség érdekében.
- *Szülő témakör:* Megadható egy félérendelt témakör is.

A témakör létrehozása után látható lesz a listás nézetben, ahol kiválasztva ellenőrizhetjük a megadott értékeket, szükség esetén módosíthatjuk is. A témakörököt törölni is lehet, de csak akkor, ha egyetlen kérdéshez és feladatsorhoz sem használjuk, különben inkonzisztens állapot alakulna ki, és témakör nélküli kérdések és feladatsorok keletkeznének. Mindkét műveletet a részletes oldalon tehetjük meg.

#### **2.1.3.3. Igaz-hamis kérdés**

Az igaz-hamis kérdések listáját az ennek megfelelő menüpont kiválasztása után érjük el. Itt egy floating button segítségével adhatunk új kérdést az alkalmazáshoz. A létrehozáshoz, amelyet szintén egy floating button segítségével érhetünk el a részletes nézetről, minden mező kitöltése kötelező, csak így biztosítható megfelelően az elvárt működés.

- *Témakör neve:* A kérdéshez tartozó témakör megnevezése, a meglévő elemek közül választható.
- *Pont típusa:* A kérdéshez tartozó pont megnevezése, a meglévő elemek közül választható.
- *Kérdés:* A kérdés szövege, egyedinek kell lennie.
- *Helyes válasz:* Meg kell adni a helyes válaszopciót, amely lehet igaz vagy hamis.

A kérdés létrehozása után látható lesz a listás nézetben, ahol kiválasztva ellenőrizhetjük a megadott értékeket, szükség esetén módosíthatjuk is. A kérdéseket törölni is lehet, de csak akkor, ha egyetlen feladatsorhoz sem használjuk, különben inkonzisztens állapot alakulna ki, és hiányoznának kérdések az összeállított feladatsorokból. Mindkét műveletet a részletes oldalon tehetjük meg.

#### 2.1.3.4. Feleletválasztós kérdés

A feleletválasztós kérdések listáját az ennek megfelelő menüpont kiválasztása után érjük el. Itt egy floating button segítségével adhatunk új kérdést az alkalmazáshoz. A létrehozáshoz, amelyet szintén egy floating button segítségével érhetünk el a részletes nézetről, minden mező kitöltése kötelező, csak így biztosítható megfelelően az elvárt működés.

- *Témakör neve:* A kérdéshez tartozó témakör megnevezése, a meglévő elemek közül választható.
- *Pont típusa:* A kérdéshez tartozó pont megnevezése, a meglévő elemek közül választható.
- *Kérdés:* A kérdés szövege, egyedinek kell lennie.
- *Válaszok megadása:* Meg kell adni a válaszokat, és jelölni kell a helyes válaszokat.

A kérdés létrehozása után látható lesz a listás nézetben, ahol kiválasztva ellenőrizhetjük a megadott értékeket, szükség esetén módosíthatjuk is. A kérdéseket törölni is lehet, de csak akkor, ha egyetlen feladatsorhoz sem használjuk, különben inkonzisztens állapot alakulna ki, és hiányoznának kérdések az összeállított feladatsorokból. Mindkét műveletet a részletes oldalon tehetjük meg.

#### 2.1.3.5. Feladatsor

A feladatsorok listáját az ennek megfelelő menüpont kiválasztása után érjük el. Itt egy floating button segítségével adhatunk új feladatsort az alkalmazáshoz. A létrehozáshoz, amelyet szintén egy floating button segítségével érhetünk el a részletes nézetről, minden mező kitöltése kötelező, csak így biztosítható megfelelően az elvárt működés.

- *Feladatsor neve:* Egyedi mező.
- *Témakör kiválasztása:* A kérdéshez tartozó témakör megnevezése, a meglévő elemek közül választható.

A feladatsor létrehozása után látható lesz a listás nézetben, ahol kiválasztva a részletes oldalt látjuk. Itt van lehetőségünk a kérdések hozzáadására és eltávolítására a feladatsorból. A témákra és kérdéstípusokra szűrve válogathatunk a kérdéseink között. Hozzáadás után a kérdések mozgathatóak a listában. A más típusuba tartozó kérdések más színnel vannak jelölve ezzel is segítve a felhasználót a kérdés típusának gyors felmérésében. A feladatsorok szabadon törölhetők. Innen léphetünk át a szerkesztés oldalra, ahol a nevet és a témakört tudjuk módosítani.

### **2.1.3.6. Válaszok ellenőrzése**

Amennyiben ezt a menüpontot választjuk, akkor a 2.1.3.5. alalszakasz-ban is leírt feladatsor listát látjuk, innen is létre lehet hozni új feladatsort. Az itt kiválasztott elem viszont egy form beküldő oldalra navigál, ahol egymás alatt látszanak a kérdések. A más típusuba tartozó kérdések más színnel vannak jelölve ezzel is segítve a felhasználót a kérdés típusának gyors felmérésében.

Itt lehet megadni a válaszainkat, illetve mobil eszközön a szövegfelismerés funkciója gyorsíthatja meg a kitöltést. A lehetséges formátumra, amit a szoftver elvár található segítség, így nehezebb ezért rossz megoldást megadni. Ha mégis hibás formátumban küldi be a felhasználó a válaszokat kap róla figyelmeztetést. A beküldést követően hamarosan megjelenik az eredmény a képernyőn.

### **2.1.3.7. Feladatsorok exportálása**

Amennyiben ezt a menüpontot választjuk, akkor a 2.1.3.5. alalszakasz-ban is leírt feladatsor listát látjuk, innen is létre lehet hozni új feladatsort. Az itt kiválasztott elem viszont egy előzetes feladatsor megjelenítő oldalra navigál.

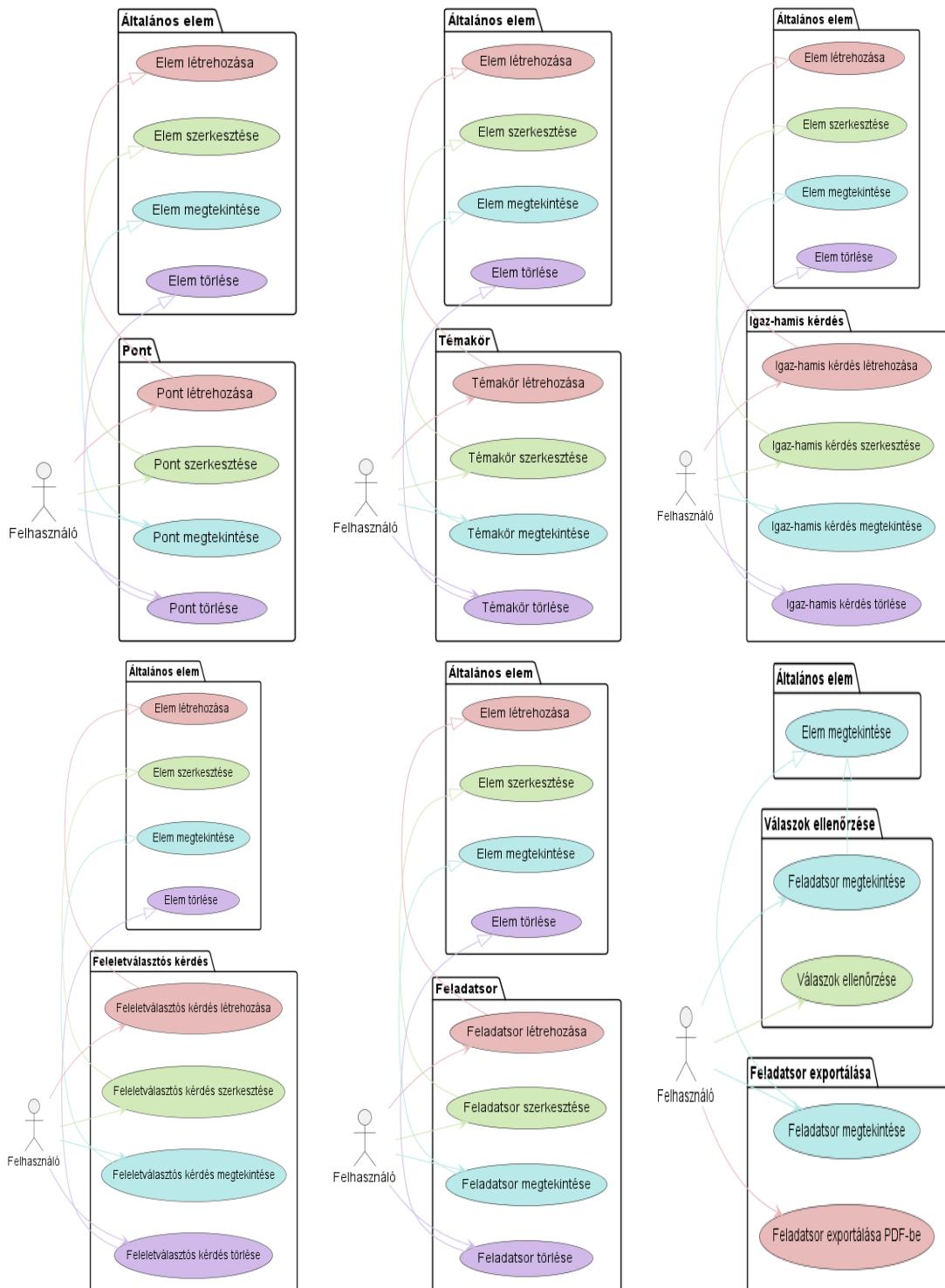
Amennyiben az itt látottakkal elégedettek vagyunk, rendben vannak a pontok és a kérdések, akkor exportálhatjuk is a munkát PDF formátumban. Az itt látottak csak egy vázlatos elrendezést adnak; az adatok ellenőrzésére szolgálnak. Előfordulhat, hogy a mobilos és az asztali verzió kis mértékben eltér egymástól a megjelenésben.

## **2.2. Diagramok**

Ebben az alfejezetben az elkészült alkalamzás használati eseteit bemutató diagram található. A UseCase diagram segítségével egy tapasztalt fejlesztő gyorsan fel tudja mérni az alkalamzás fő felépítését és megközelítő struktúráját.

A csak a magas szintű használati esetek jelennek meg a diagramon az átláthatóság miatt. A képernyők közötti navigáció az ábra átláthatóságán jelentősen rontana, így arról csak a szöveges leírásban esett szó.

A PlantUML segítségével elkészített diagramokon látható, hogy létezik öt egymáshoz hasonló működésű használati eset, így ezek származhatnak egy általános elemből vagy usecaseből. Az azonos színekkel jelölt esetek hasonló funkcionalitást jeleznek ezzel is segítve a gyors megértését a szoftvernek. A Felhasználóból induló nyilak a lehetséges cselekedeteket jelölik, míg az ezekből a usecasekből induló nyilak az általános cselekedetekből való leszármazást.



**2.1. ábra.** Usecase diagramja az alkalmazásnak.

### **3. fejezet**

## **Irodalomkutatás**

Ebben a fejezetben bemutatom az általam használt forrásokat és technológiákat. Esetenként ábrákkal és kód részletekkel ilusztrálom az adott technológiát a könnyebb érhetőség kedvéért. Az alfejezet végén bemutatok néhány alternatív megoldást a multiplatform fejlesztésre.

### **3.1. Felhasznált technológiák**

Ez az elfejezet a használt technológiák bemutatására fókuszál, a könnyebb érhetőség kedvéért ábrák, képek és forráshivatkozásokkal kiegészítve.

#### **3.1.1. Jetpack Compose**

A Jetpack Compose a korábbi Android fejlesztési módszer mellett hozott létre egy alternatív megoldást. Kezdetben nem lehetett tudni, hogy a fejlesztők hogyan fognak reagálni az új irányra. Korábban a Java nyelv mellett megjelent a Kotlin nyelv is, ami később szinte teljesen leváltotta az elődjét. Ebből arra lehetett következtetni, hogy egy új és modernebb megoldás meg tudja álni a helyét az XML View megoldást ellenében. Jelenleg mind a két megoldás támogatott, de a fejlesztések irányában egyértelműen a Compose felé húz.

"A Jetpack Compose egy új, deklaratív UI toolkit, amit a Google hozott létre kifejezetten natív Android alkalmazások fejlesztéséhez."<sup>[21]</sup> A deklaratív nyelvekhez hasonlóan, azt kellmegadnunk, hogy mit szeretnénk látni és nem azt leírni, hogy ez hogyan történjen meg. Nekünk elegendő azt leírni, hogy a gomb hogyan nézzen ki és hol legyen és megadni egy lambda paraméternek, hogy a megnyomása soránmi történjen. Mivel ez egy UI toolkit, ezért az összes vezérző és szerkezeti elem hasonlóan néz ki, hasonlóan lehet használni így a fejlesztői és a felhasználói élmény is egységes és megszokott minőségű lesz minden alkalmossal. A Google ezt a Material design segítségével hozta létre, illetve annak újabb változataival. Erről részletesebben itt találhatók információk: <https://m3.material.io/>.

Az alábbiakban egy a Google által írt rövid kód részleten (3.1. kód részlet) bemutatom a legfontosabb részeit a Compose alapjainak.<sup>[15]</sup> Az első lépése az alaklamzás elkészítésének az a Composable függvény megírása. minden a UI-t megjelenítő függvény a @Composable annotációt viseli. Innentől kezdve hagyományos Kotlin függvényként viselkedik, megadhatunk tetszőleges paramétereket (name, modifier) és alapértelmezett értékeket is. Egy Composable függvényből tetszőleges másik Composable függvény meghívható megfe-

lelőláthatóság esetén. Ilyen például a `Text()` is ami a Material könyvtárnak egyik tagja és egyszerű szöveget jelenít meg.

A UI megírása után ezt a megfelelő helyen meg is kell jelenítenünk, erre az alkalmazás belépési pontja után van lehetőségünk. Android esetén ez az activity `onCreate` függvénye. A `setContent` vár egy lambda függvényt, aminek viselnie kell a `@Composable` annotációt, használhatjuk hozzá a Kotlin trailing lambda megoldását, aholis, ha az utolsó paramétere a függvénynek egy lambda, akkor a többi paraméter megadása után között megadhatjuk a függvény törzsét. A `BasicsCodelabTheme` is egy Composable függvény amiben az alap beállítások után meghívhatjuk a saját `Greeting` függvényünket. A UI felépítse innentől kezdve már egyszerű. A Composable függvények megírása után egymásból meghívva azokat előáll az alkalmazás.

```
 @Composable
 fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            BasicsCodelabTheme {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    Greeting("Android")
                }
            }
        }
    }
}

// Jetpack Compose forráskód
public fun ComponentActivity.setContent(
    parent: CompositionContext? = null,
    content: @Composable () -> Unit
)
```

### 3.1. kódrészlet. Példa a Compose használatára.

A következőkben bemutatom a UI toolkit fontosabb általam használt részeit. Kitérek arra, hogy mire jó, miért ezt használtam és hogyan lehet őket hatékonyan felhasználni a legújabb Compose Multiplatform verziókban.

#### 3.1.1.1. State és StateFlow

A State és a StateFlow hasnló problémára ad megoldást. A State alapvetően Compose sepcifikus megoldás, ha változik az értéke akkor a lefut a recomposoition. Ezzel szemben a StateFlow a Kotlin nyelvben általánosan használt eszköz és sokkal bővebb felhasználással rendelkezik, mint az egyszerű State. Az egyszerű State-et általában egy Composable függvényen belül használják, míg a a StateFlowt ViewModelekben.

Ennek ellenére mind a két megoldás tökéletesen alkalmazható, jelenleg már Compose Multiplatform alkalmazásokban is. Mivel ViewModelben használva az adatok nem besznek el a képernyő elforgatása során ezért egyszerűbb műveletekre és adaokra nincs lényegi különbség a működésben.

"A StateFlow előnyei:"[3]

- *"Flow operátorok:* A StateFlow támogat olyan operátorokat, mint a map, filter, és combine, lehetővé téve az adatok rugalmas feldolgozását és összetett adatfolyamok létrehozását."
- *"Folyamat-megszakadás kezelése:* A SavedStateHandle-lel kombinálva biztosítja az UI állapot megőrzését, még a képernyő elforgatása vagy újraindítás esetén is."
- *"ViewModel újrafelhasználhatóság:* A StateFlow lehetővé teszi a ViewModel függetlenítését a UI-rétegtől, ami elősegíti a moduláris, tesztelhető és újrafelhasználható architektúrát."

A lenti kódban (3.2. kód részlet) láthatunk példát minden az egyszerű State használatára, ebben az esetben a Screen állapotát tárolom el Statekben. A megjelnített adatok itt a StateFlow logikáját követik. Létezik egy privát MutableStateFlow amin történnek a változások például, ha adat érkezik. Van egy másik azonos nevű érték is ami ugyan annak a StateFlownak egy immutabilis változata, ehhez fér hozzá a UI, így onnan nem érkezhet változás közvetlenül a StateFlowba. Amennyiben erre szükség van, a viewmodel biztosíthat erre vonatkozóan függvényeket és beállíthatja a privát MutableStateFlow értékét.

```
class TopicListViewModel: ViewModel() {  
    var topicListScreenState: TopicListScreenState by mutableStateOf(TopicListScreenState.  
        Loading)  
    private val _topicListUiState = MutableStateFlow(TopicListUiState())  
    val topicListUiState: StateFlow<TopicListUiState> = _topicListUiState  
    ...  
    fun getAllTopicList(){  
        topicListScreenState = TopicListScreenState.Loading // State Változás  
        viewModelScope.launch {  
            topicListScreenState = try{ // State Változás  
                val result = ApiService.getAllTopicNames()  
                _topicListUiState.value = TopicListUiState( //StateFlow Változás  
                    topicList = result.map { nameDto ->  
                        TopicRowUiState(  
                            topic = nameDto.name,  
                            id = nameDto.uuid  
                        )  
                )  
            }  
            TopicListScreenState.Success(result) // State Változás  
        } catch (e: IOException) {  
            TopicListScreenState.Error.errorMessage = e.toString()// "Network error"  
            TopicListScreenState.Error // State Változás  
        }  
    }  
}
```

**3.2. kód részlet.** Példa a State és StateFlow használatára.

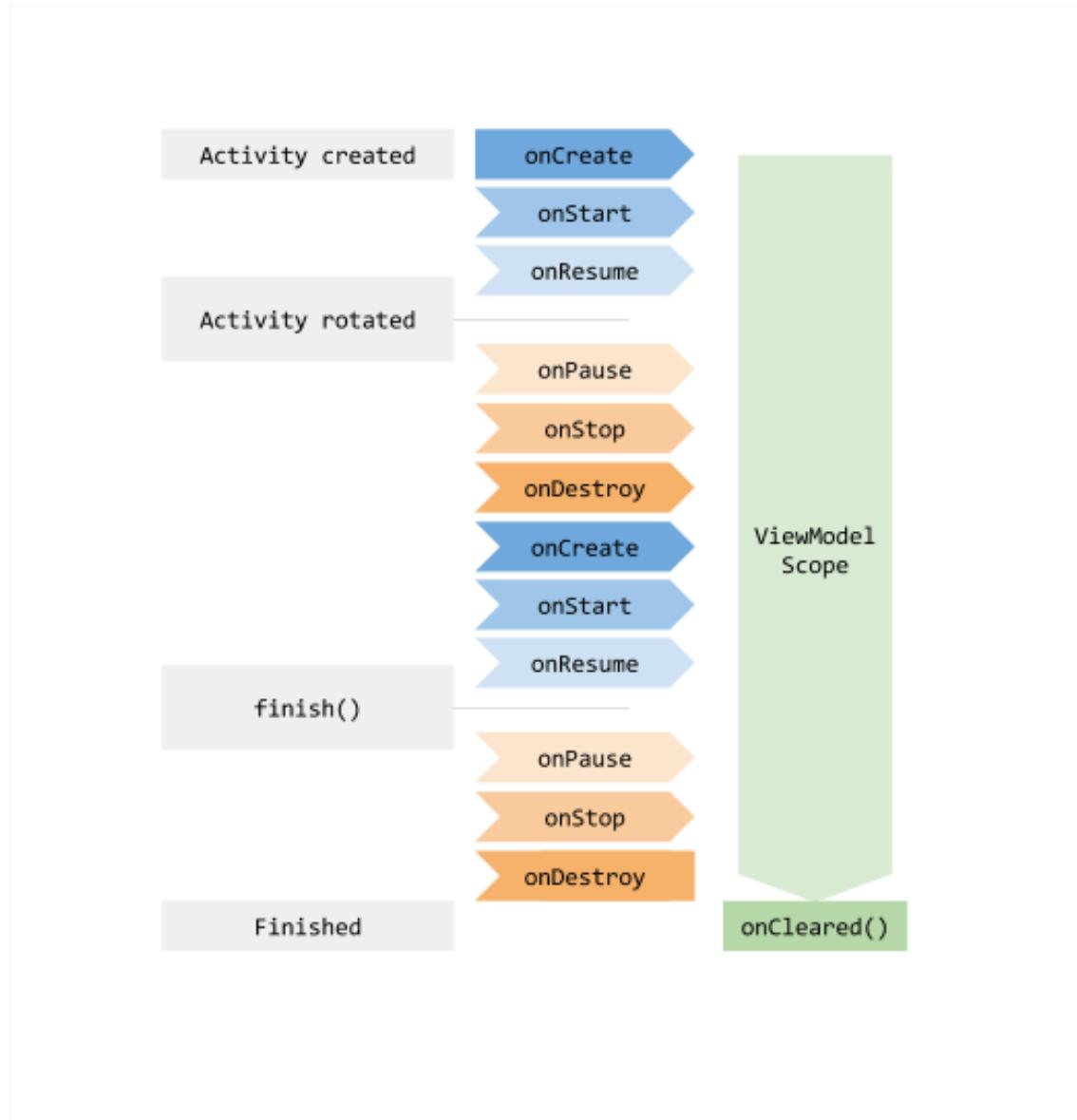
### 3.1.1.2. ViewModel

Az Android ViewModel már használható Kotlin- és Compose Multiplatform környezetekben is[6], így természetesen ezt a jól bevált megoldást választottam. Egy rövid összefoglalót szeretnék adni a hivatalos dokumentáció alapján a képességeiről:[17]

A ViewModel az Android Jetpack része, és az UI állapotának megőrzésére szolgál konfigurációs változások során, például képernyőforgatáskor. 3.1. ábra Fő előnyei közé tartozik az állapot tartásának és az üzleti logika elérése a UI-rétegen. A ViewModel segít

elválasztani az adatkezelést a UI-rétegtől, ehhez a ViewModelben felhasználható a Kotlin coroutine ami egyfajta aszinkron működést tesz lehetővé, és kompatibilis olyan Jetpack könyvtárakkal, mint a Hilt, Compose, és a Navigation. Best practice szerint kerülni kell az életciklushoz kötött objektumok tárolását benne, hogy elkerüljük a memóriásivárgást.

Mint minden MVM és MVVM architektúrában az adatok elérése, átalakítása a UI számára, tartós tárolása a ViewModel feladata. Több megközelítés is lehetséges: minden képernyő kapjon egy saját ViewModelt; egy ViewModel tudjon többet és legyen újra felhasznáva több képernyőn. Én az első megoldást választottam, így könnyebb kezeleni a különböző képernyők állapotait és egyszerűbb átlátni egy-egy egységet.



**3.1. ábra.** A ViewModel álltalánosságban véve hosszabb életű, mint egy View. Ez különösen igaz Android környezetben, ahol számolni kell a képernyő elforgatással és az alaklazás háttérbe kerülésével. A tartósan tárolnikívánt adatokat ezért mindig ViewModel kell tárolni. [17]

### 3.1.1.3. Navigation and routing

Az Androidban már jól működő navigációt felelős könyvtárak már használhatóak a Kotlin Multiplatform fejlesztéséhez [9]. Ennek minden összesen pár egyszerű lépése van, így könnyű használni és rugalamsan működik minden környezettel. Mindezt kódvab is bemutatom. (3.3. kódrészlet) A következők a szükséges lépések: [9].

1. "Sorold fel a navigációs gráfban szereplő útvonalakat, mindegyik egyedi string határozza meg az útvonalat."
2. "Hozz létre egy NavController példányt a navigáció kezeléséhez"
3. "Adj hozzá egy NavHost komponenst az alkalmazásodhoz."
  - "Válaszd ki a kezdő útvonalat a korábban definiált útvonalak közül."
  - "Hozd létre a navigációs gráfot közvetlenül a NavHost részeként vagy programozottan a NavController.createGraph() függvényel."

```
//Első lépés: útvonalak létrehozása
//Érdemes sealed classt használni és data objectként létrehozni a routeokat
sealed class ExamDestination(val route: String) {
    //Lehet egyszerű
    data object LoginScreenDestination : ExamDestination("LoginScreen")

    //Vagy paraméterekkel és adatokkal ellátott
    data object TopicDetailsDestination : ExamDestination("TopicDetails") {
        const val topicIdArg = "0"
        val routeWithArgs = "$route/{$topicIdArg}"
    }
}

//Második lépés: NavController létrehozása
fun NavigationComponent() {
    MaterialTheme {
        val navController = rememberNavController() //Itt történik
        Scaffold() { innerPadding ->
            ExamNavHost(      //Paraméterek egy Composable függvényt vár, ezen belül is egy NavHost fü
                ggvényt
                    navController = navController,
                    modifier = Modifier.padding(innerPadding)
            )
        }
    }
}

//Harmadik lépés: NavHost komponens hozzáadása
actual fun ExamNavHost(
    navController: NavController,
    modifier: Modifier
) {
    NavHost(
        navController = navController, //NavController hozzárendelése
        startDestination = ExamDestination.MainScreenDestination.route, //Kezdő útvonal beállítása
        modifier = modifier
    ) {
        //Navigációs gráf egy elemének létrehozása
        composable(
            route = ExamDestination.TopicListDestination.route,
        ) {
            TopicListScreen(
                addNewTopic = { navController.navigate(ExamDestination.NewTopicDestination.route) },
                navigateToTopicDetails = { topicId ->
                    navController.navigate("${ExamDestination.TopicDetailsDestination.route}/{$
topicId}")
                },
            )
        }
    }
}
```

```
        navigateBack = { navController.popBackStack() }  
    )  
}  
}  
}
```

**3.3. kódrészlet.** Példa a Navigation használatára.

### 3.1.2. Ktor

A Ktor egy Kotlin HTTP kommunikációt megvalósító könyvtár[7]. Alkalmas minden szerver oladni kód írására, a REST API-om is ezt használja és kliens oladni kód megvalósítására is. A használata nagyon egyszerű, és testreszabható. Én egy Kotlin objectet használtam, ami magában foglalja az ApiService-et. Létre kellett hozni egy http klienst és beállítani a base url-t, illetve a content typenak a JSON üzenet formátumot. Ezt követően a végpont hívásokat kellett már csak létrehozni.

```
object ApiService {
    private var authToken: String? = null // Mutable token that can be updated at runtime

    private val httpClient = HttpClient() {
        install(ContentNegotiation) { // content type beállítása
            json(Json {
                ignoreUnknownKeys = true
                prettyPrint = true
            })
        }

        defaultRequest { // Base url beállítása
            url("http://mlaci.sch.bme.hu:46258") // Set the base URL 152.66.182.70:46258
            192.168.1.17:46258
            authToken?.let { token ->
                header(HttpHeaders.Authorization, "Bearer $token") // Add the Bearer token if it's
                not null
            }
        }
    }

    suspend fun getAllPoints(): List<PointDto> = httpClient.get("/point").body() //végpontok
}
```

### 3.4. kódrészlet. Példa a Ktor használatára.

### 3.1.3. KotlinX Serilizáció

A serializáció a JSON formátum konvertálása miatt van szükség. Az alábbiakban a hivatalos dokumentációból olvasható egy részlet ami jól összefoglalja a használatát. Ez a technológia is használható Kotlin Multiplatform területen.

"A szerializáció során az alkalmazások adatait egy olyan formátumba alakítjuk, amely hálózaton átvihető vagy tárolható adatbázisban vagy fájlban. Az ellenkező folyamat, a deszerializáció, az adatokat külső forrásból olvassa be és konvertálja futásidéjű objektum-má. A Kotlinban a szerializációhoz elérhető a `kotlinx.serialization` eszköz, amely Grad-le bővítményt, futásidéjű könyvtárakat és fordítói bővítményeket tartalmaz, így segítve a különböző nyelvű rendszerek közötti adatcserét, mint a JSON és a protocol buffers formátumokkal."<sup>[11]</sup>

### **3.1.4. CameraX**

A CameraX technológia kizárolag Android platformon használható. Itt azonban egy nagyon széles és gazdag APIt biztosít a fejlesztéshez. A legfontosabb felhasználható funkciói az Preview azaz előnézet, amikor kép készítése nélkül megkelenik a képernyőn a kamera képe. Az Image analysis, azaz képfeldolgozó funkcionálitás. Hozzáférhetünk a buffer tartalmához így fel tudjuk azt használni egyéb célokra különböző algoritmusok futtatásához és összekapcsolható a Google ML-Kit technológiákkal. 3.1.5. alszakasz A képeket menetben is tudjuk, hasonlóan a beépített kamera alkalmazáshoz és ugyan úgy videót is tudunk vele rögzíteni. Ezek a leírás a Google Android Developers dokumentációja alapján készült. Részletesebb információk itt találhatóak: [14]

### **3.1.5. ML-Kit**

Az ML-Kit a Google által fejlesztett mesterséges intelligencia alapú API. Számtalan felhasználási területtel rendelkezik, ezekeből néhányat felsorolok: szöveg- és arcfelismerés, dokumentum szkennelés, kép feliraotozás, fordítás, nyelv detekció és még számos más lehetőség. Én ezek közül az Androidos alkalmazásban a képen történő szövegfelismerést próbáltam ki.[16] Sajnos ez a funkció is Android specifikus így egy iOS alkalmazáson ez a probléma más megközelítést igényelne. Messze nem tökéletes még ez a technológia, de kipróbálásra mindenféleképpen érdekes és hasznos lehet.

### **3.1.6. Accompanist-engedélykezelés**

Az engedélykezelés nem egyszerű feladat az Android rendszerekben így célszerű erre kifejlesztett könyvtárakat használni. Egy ilyen könyvtár az Accompanist, amelyet a Google fejleszt. A használata sokkal egyszerűbbé teszi ezt a bonyolult folyamatot. Néhány egyszerű lépéssel egy kész megoldás kapunk.

Elsőként az szükséges engedélyeket be kell jegyezni a manifest fájlba. Következő lépésként ellenőrizni kell, hogy az alkalmazásunk rendelkezik a szükséges engedélyekkel vagy sem. Amennyiben nem akkor a használat előtt ezt kérünk kell, de ezt csak úgy tehetjük meh, hogy a többi funkció elérhető legyen. Az én esetben, csak a szövegfelismerő funkcióra kattintás után kérem el az engedélyt, de maga a válaszokat elküldő képernyő használható az engedélyek nélkül is. Az elkerített engedélyeket egy permissionStateben tároljuk el, így innen lehet ellenőrizni, hogy korábban már megadta-e a felhasználó, így legközelebb nem kell elérni tőle. [20]

Egyedül a veszélyes engedélyeket kell ilyen módon elérni, mint például a kamera használata, tehát amik a felhasználót veszélyeztetik. Vannak nem veszélyes engedélyek is, ilyen az internet használat, ezt csak a manifest fájlban kell rögzíteni.

### **3.1.7. PdfDocument és PDFBox**

A PdfDocument a Google tulajdonában lévő PDF szerkesztő eszköz. Ennek a felhasználával egy PDF fájlba írhatunk tetszőlegesen elhelyezett szöveget és képeket. Létrehozhatók különböző Paint objektumok amik segítségével egyszerűen rajzolható táblázat és formázható a szöveg. Ez a megoldás csak Android eszközökkel kompatibilis.

<sup>16</sup>Az Apache PDFBox® könyvtár egy nyílt forráskódú Java eszköz PDF dokumentumok kezelésére. Lehetővé teszi új PDF dokumentumok létrehozását, meglévő dokumen-

tumok módosítását és tartalom kinyerését a PDF fájlokból. Az Apache PDFBox több parancssori eszközt is tartalmaz, és az Apache License v2.0 alatt került kiadásra." [1] Hasonlóan használható, mint a PdfDocument, de valamelyest eltér a két könyvtát API készlete. Elsőként Androidra készült el az exportálás funkció, de mivel nem sikerül teljesen ugyan azt az eredményt létrehozni minden a két esetben, ezért egy valós projektben a multiplatform rendszerekben is használható PDFBox megoldást használnám minden platformon.

### 3.1.8. Kotlin- és Compose Multiplatform

Többször beszéltem már a Kotlin- és Compose Multiplatform fogalmakról. Leg könnyebben úgy lehet leírni a kapcsolatukat, mint a Compose Multiplatform részhalmaza a Kotlin Multiplatformnak. Rengeteg nagy cég használja a KMP technológiát közöttük a Netflix, 9GAG, McDonalds' és Philips. [23] Az általam korábban felsorolt technológiák közül, ami leginkább ebbe a kategóriába esik az a Ktor (3.1.2. alszakasz) és a KotlinX szerelződés (3.1.3. alszakasz). Mivel 2024 őszén elérhetővé vált az Android ViewModel (3.1.1.2. alalszakasz) és a navigáció (3.1.1.3. alalszakasz) is így ezeket is ide sorolhatjuk már a statekkal együtt (3.1.1.1. alalszakasz).

Az egyetlen fontosabb rész amit kihagytam, az maga a Compose deklaratív UI toolkit (3.1.1. alszakasz), ami a Compose Multiplatformot alkotja. 2021-ben vált lehetővé a Compose használata nem csak Android alapú rendszerekhez, a KMP viszont 2017-ben kezdte meg az úját. Nagy jelentősége van a CMP-nek mivel így kizárolólag Kotlin nyelven Android fejlesztők tudnak iOS és asztali alkalmazást fejleszteni minimális natív kóddal, de mégis natív élményt nyújtva. Jelenleg a webes irány még alpha verzióban van, de jelenleg is folyik a fejlesztés a Kotlin WASM-re (Web Assembly) valóhatékony fordításán. A Kotlin JavaScript kódával is le lehet fordítani a Java mellett.

Három féle módon lehet Kotlin Multiplatform kódázist fejleszteni (3.2. ábra). Az előző, bal oldali ábra értelmezése, hogy a kódázis egy kis része írózik KMP-ben, például csak az adatbázis vagy REST API elérés. A következő ábra azt mutatja, hogy a logika teljes egészben KMP-ben írózik, így például használják a ViewModeleket (3.1.1.2. alalszakasz), de a UI natív módon készül, Androidra Composeban iOS-ben SwiftUIban. Az utolsó ábra már a Compose Multiplatform megjelne, amikor minden platformra Composeban készül el a felhasználói felület. Balról jobbra haladva egyre nő a kód újrafelhasználhatósága, így kevesebb munka szükséges és könnyebb is a kód karbantartása, mivel előre láthatólag egyre kevesebb helyen kell megváltoztatni azt.



**3.2. ábra.** A KMP fejlesztés változatai. [23]

Semmi sem teljesen tökéletes így előfordultaht, hogy az egyik platformna máshogyan, nem lehet vagy nem érdemes valamit megvalósítani, mint például egy asztali alaklmazá-

son egy fotó elkészítését az én példámban. Ilyenkor jöhetnek szóba az expect és actual függvények. A közös kódban ilyenkor egy függvénytörzset definiálunk, csak itt lehet ebben az esetben alapértelmezett paramétereket beállítani, például egy Modified Composable függvény esetén. A megvalósítás ilyenkor az alaklazás specifikus kódban történik (androidMain, desktopMain, iOSTMain). (3.5. kódrészlet) Ehhez az actual functiont kell megvalósítani, és a platformtól függően ezek fognak meghívódni automatikusan, mivel a build során ezek lesznek behelyesítve az expect függvény helyére. Már létezik actual és expect osztály is, én alkalmaztam is, de ez még kisérleti verzióban van.

```
//commonMain-ben lévő expect függvénytörzs, alapértelmezett paraméterrel.
@Composable
expect fun MainCameraScreen(examId: String = "0", navigateBack: () -> Unit)

//androidMain-ben lévő valós megvalósítás
@Composable
actual fun MainCameraScreen(examId: String, navigateBack: () -> Unit) {

    val cameraPermissionState: PermissionState = rememberPermissionState(android.Manifest.permission.CAMERA)

    MainCameraContent(
        hasPermission = cameraPermissionState.status.isGranted,
        examId = examId,
        onRequestPermission = cameraPermissionState::launchPermissionRequest,
        navigateBack = navigateBack
    )
}

//desktopMain-ben lévő placeholder megvalósítás, értesíti a felhasználót, hogy ez a funkció az eszközén nem támogatott
@Composable
actual fun MainCameraScreen(examId: String, navigateBack: () -> Unit) {
    Scaffold(
        topBar = {
            TopAppBarContent(stringResource(Res.string.camera), navigateBack)
        },
        innerPadding ->
            UnsupportedFeatureScreen(modifier = Modifier.padding(innerPadding))
    )
}
```

### 3.5. kódrészlet. Expect és actual használata

#### 3.1.9. Gradle build rendszer

A Compose Multiplatform projektek is a Gradle build rendszert használják, első sorban a függőségek megszerzésere és az alkalmazás létrehozására. Egy átlagos fejlesztőnek minden összesen annyi dolga van, hogy kigyűjtje a használt függőségeket és a fejlesztőkörnyezet általában segít a megfelelő verziók megtalálásában. Újban a Kotlin DSL használata terjedt el.

"A DSL (Domain-Specific Language) egy programozási nyelv, amely egy meghatározott problémakör megoldására összpontosít. Az általános célú nyelvektől eltérően a DSL-ek, például az SQL és a regexek, csak egy szűk területre fókusznak, ami lehetővé teszi a problémák deklaratív módon való megoldását."<sup>[22]</sup> Ennek a segítségével egyszerűbben tudjuk megadni a Gradle függőségeket.(3.6. kódrészlet)

```
kotlin { //Részlet a build.gradle fájlból
    androidTarget {
        compilerOptions {
            jvmTarget.set(JvmTarget.JVM_11)
        }
    }
    sourceSets {
```

```

        androidMain.dependencies {
            implementation(libs.androidx.activity.compose)
        }
    }
}

```

### 3.6. kódrészlet. Kotlin DSL

Ezen kívül a verziók egyszerűbb karbantartására használhatunk version catalog fájlt, ez a libs.version.toml. A toml a "Tom's Obvious, Minimal Language" rövidítése, ezt előző sorban egyszerűbb konfigurációs fájlok esetében használják, ha úgy tetszik egy "butább" yaml formátum. Az szükséges részei a [versions] és a [libraries], illeve szükség lehet [plugins]-re is. (3.7. kódrészlet) Az itt felvett értékekre lehet hivatkozni a build.gradle fájl(ok)ban.

```

#Részlet a libs.version.toml fájlból

[versions]
agp = "8.2.2"
android-compileSdk = "34"
android-minSdk = "24"
android-targetSdk = "34"
androidx-activityCompose = "1.9.2"
androidx-appcompat = "1.7.0"
androidx-constraintlayout = "2.1.4"
androidx-core-ktx = "1.13.1"

[libraries]
androidx-core = { module = "androidx.core:core", version.ref = "androidx-core-ktx" }
androidx-core-ktx-v1120 = { module = "androidx.core:core-ktx", version.ref = "coreKtx" }

[plugins]
androidApplication = { id = "com.android.application", version.ref = "agp" }
androidLibrary = { id = "com.android.library", version.ref = "agp" }

```

### 3.7. kódrészlet. Version catalog

#### 3.1.10. Fejlesztőkörnyeztek

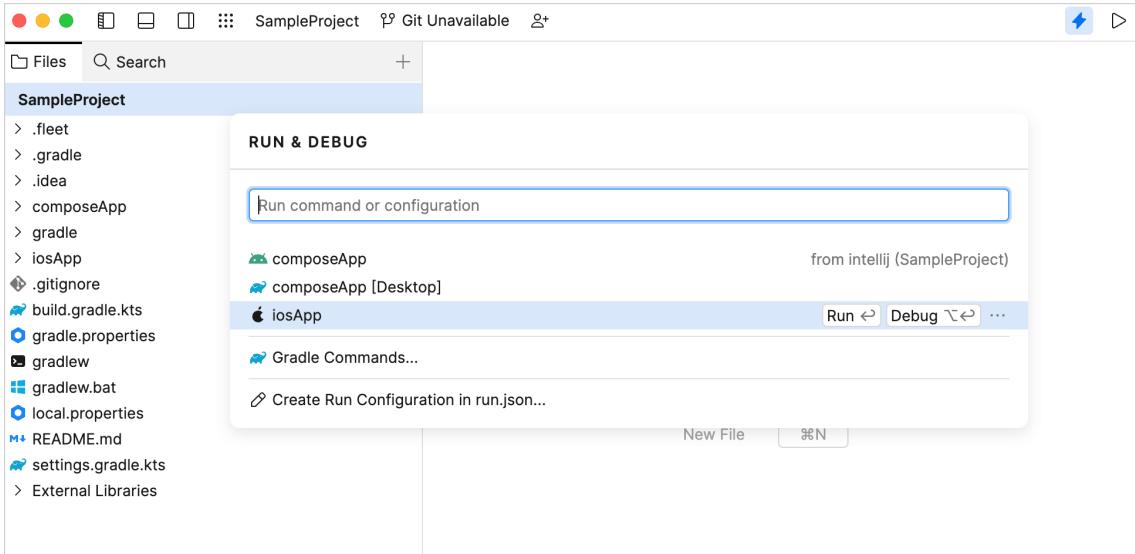
Fejlesztőkörnyezetnek a JetBrains egyik eszközét választottam, a Fleet-et. Kifejezetten Multiplatform fejlesztéshez készítettek, szinte az összes nyelvet támogata ami ebben a témaiban szóba jöhét, és minden funkcióval rendelkezik amivel a natív fejlesztésre készült IDE-ik is. Többek között kódkiegészítéssel és kód kiemeléssel is, így nem kell IDE-t váltani, ha más nyelven kell éppen dolgozni.

"Amikor az Smart Mode engedélyezve van, a Fleet nyelvspecifikus funkciókat kínál, mint például a kódkiegészítés, navigáció, hibakeresés és refaktorálás. Ha ez a mód le van tiltva, a Fleet egyszerű szövegszerkesztőként működik; gyorsan meg lehet nyitni fájlokat és módosításokat végezni, de a fejlettebb funkciók nélkül. A háttérben a Fleet kiválasztja a kód feldolgozásához szükséges háttérmotort. Intelligens módban például a Kotlin feldolgozási motorja az IntelliJ IDEA-hoz használt motor, így ismerős funkciók maradnak elérhetők." [12]

Egy másik hasznos funkció, hogy egy helyről lehet minden platformra buildelni az alkalmazást (3.3. ábra) így nem kell egy külön Android Studiot és egy Xcodeot is megnyitni, ha szeretnéd tesztelni az alkalmazást az adott eszközön.

A fő fejlesztőkörnyezeten kívül amíg csak az Android applikációt fejlesztem az Android Studiot használtam. A REST API megírásához és karbantarásához szintén a JetBrains termékét az IntelliJ IDEA Ultimate-et használtam, ami kifejezetten Java és Kotlin projektekre készült. Kisebb mértékben a fejlesztéshez és nagyobb mértékben a dokumenta-

tációhoz és a szakdolgozat megírásához A Visual Studio Code-ot használtam, mivel sok hasznos bővítménnyel rendelkezik például L<sup>A</sup>T<sub>E</sub>Xés PlantUML használatához.



**3.3. ábra.** A különböző eszközökre egy helyen lehet buildelni és futtani az alkalmazást. [12]

### 3.1.11. REST API, Postman és adatbázis

A kliens oldali alkalmazásokat egy saját REST API szolgálja ki. Ezt az előző félévben készítettem el. Teljesen egészében Kotlin nyelv felhasználásával. A HTTP kommunikáció megvalósításához a már korábban 3.1.2. alszakaszban bemutatott Ktor-t használtam. Itt a szerver oldali funkcionalitása került előtérbe. A szerilizáció itt is a 3.1.3. alszakaszban leírtak szerint történt. Az adatbázis az úgynevezett code-first felfogás alapján készült. Ez azt jelenti, hogy kódban leírtam az adatázis felépítését és a kigenerálását rábíztam a Exposed-ra ami a JetBrains Ital fejlesztett adatbázis elérést lehetővé tevő könyvtár. A használt adatbázis javaslatok alapján a PostgreSQL lett. Ennek a folyamatnak egy részletesebb leírása a [8] forrásban olvasható, ez alapján készíttem el a saját szerveremet.

Mind az adatbázis, mind a REST API egy-egy Docker-konténerben fut egy viruális gépen, így biztosítva a folyamatos elérhetőséget. "Mi az a Docker? A Docker egy nyílt platform alkalmazások fejlesztésére, szállítására és futtatására. Lehetővé teszi, hogy az alkalmazásokat elválasszuk az infrastruktúrától, így gyorsabban tudunk szoftvereket szállítani. A Docker segítségével az infrastruktúrát ugyanúgy kezelhetjük, mint az alkalmazásokat. A Docker szállítási, tesztelési és kódtelepítési módszertanait kihasználva jelentősen csökkenthetjük az időt a kód megríása és a termelési környezetben való futtatása között." [24]

"A Docker platform: A Docker lehetőséget biztosít arra, hogy az alkalmazást egy lazán izolált környezetben, úgynevezett konténerben csomagoljuk és futtassuk. Az izoláció és a biztonság lehetővé teszi, hogy egy adott gépen egyszerre több konténert futtassunk. A konténerek könnyűsűlyűak, és minden tartalmaznak, ami szükséges az alkalmazás futtatásához, így nem kell a gazdagépre telepített környezetre támaszkodni. A konténereket megoszthatjuk munka közben, biztosítva, hogy mindenki ugyanazt a konténert kapja, amely ugyanúgy működik." [24]

Rendelkezik saját DNS címmel is, így könnyen megjegyezhető és elérhető fejlesztés és tesztelés közben is. A teszteléshez a Postmant használtam. Ez egy olyan alaklamzás amiben HTTP kéréseket lehet létrehozni, és a mezőket tetszőlegesen testreszabni. Gyakran volt rá szükség ebben a félévben is, mert bizonyos adatformákat/követelmények változtak és ez sokkal hatékonyabb hibakezelést tett lehetővé, mint egy böngészős HTTP kérés vagy az alaklamzásból történő debugolás.

### 3.1.12. Kipróbált, de végül nem használt egyéb érdekes megoldások

Az önálló laboratórium során elkészítettem alkalmazások technológiáit felhasznált, első sorban kísérlelésezés miatt. Ezek egy részére találtam jól használható Multiplatform alternatívát, más részére nem, vagy csak nagyon sok munkával lehetet volna megvalósítani. Rendelkezett lokális Room adatbázissal is, erre egy alternatív megoldás KMP területen az SQLDelight technológia, de 2024 őszétől kezdve a Room is támogatott. Ezt userek tárolására használtam Firebase integrációval. A Firebase sajnos sokkal nehezebben használható ebben a környezetben és mivel az alkalmazás jelenlegi állapotában a usereknek nincs jelentősség így ezek nem kerültek be a végső alkalmazásba.

Egy korábbi KMP ViewModel alternatíva a moko.viewmodel (Mobile Kotlin Model-ViewModel architecture) volt, de ennél kényelmesebb az Androidos. Mivel nem volt szükséges így a user- és login servicekre és lokális adatbázisra így elvettem a dependency injection használatát, mert így már nem okozott akorra problémát a viewmodelök létrehozása. Az Androidban használt Hilt itt nem használható még, és a Koin, ami egy hasonló KMP implementáció, nem könnyítené meg annyival ezt a folyamatot, mint a Hilt. Ezeknél a függőségeknél már a verziók összehangolása se volt egyértelmű feladat, minél több függőségre van szükség annál nagyobb a valószínűsége, hogy valami összeütközök vagy eltörök egy új frissítés miatt, főleg, ha az nem egy hivatalos függőség a Google vagy JetBrains által. Külön-külön rendesen működtek, de az előbb felsorolt prónlémák miatt úgy döntöttem, hogy egy egyszerűbb és letisztultabb megoldást használjak a függőségek terén.

## 3.2. Hasonló multiplatform megoldások összehasonlítása

Négy különböző technológiát vizsgáltam meg és a korábbi tapasztalataim és információm alapján próbáltam választani közülök. Az első a Compose Multiplatform, a második a MAUI, a harmadik a React Native, a negyedik a Flutter. Az alábbiakban található egy rövid összefoglalás ezekről a technológiákról.

**Kotlin- és Compose Multiplatform:** A JetBrains és a Google által fejlesztett technológiák, amelyek lehetővé teszik a kód megosztását platformok között anélkül, hogy új nyelvet kellene bevezetni. Nemrégiben stabilizálták, és támogatja a felhasználói felület megosztását a Compose Multiplatform segítségével. [10]

**.NET MAUI:** A Microsoft C# és XAML alapú keretrendszer, amely cross-platform API-kat, hot reload-ot biztosít, és asztali, illetve mobil platformokra céloz. [10]

**React Native:** A Meta JavaScript alapú keretrendszer, amely a natív UI-t helyezi előtérbe, erős közösségi támogatással, a Fast Refresh-t használja, és a Flipper-t integrálja hibakereséshez. [10]

**Flutter:** A Google Dart alapú keretrendszer, amely a hot reload és az egyéni rendelés révén ismert, támogatja a Material Design-t, és széles körben használják cross-platform alkalmazásokhoz (pl. eBay, Alibaba). [10]

A kiválásztás során számos szempontot figyelembe vettetem. Fontos volt a tanulási lehetőség a témából, lehetőleg olyan módon, hogy azt később is tudjam kamatoztatni. Egy másik szempont volt, hogy ne legyen teljesen ismeretlen a hasznát eszköz és környezet, ennek célja az volt, hogy az időm nagy részét ne tutorial videók nézésvel töltsem az alapoktól kezdve, hanem a dokumentációk alapján is el tudjak igazodni hatékonyan. Nem utolsó szempont volt, hogy szerettem volna folytatni az előző félévben elkezdett alkalmazásomat. Az összes opción megvizsgálása után alkalmasnak találtam a Compose Multiplatformot a szakdolgozat témajaként.

Ezek közül az utolsót zártam ki a leghamrabb, a fenti négy közül ez az egyetlen ami nem natív UI eléményt nyújt a felhasználóknak és egy számomra teljesen ismeretlen nyelven a Google által létrehozott nyelvben a Dartsban kell programozni. Első sorban mobilos fejlesztésre használjak, a többi része nem a legoptimálisabb és én ennél egy általánosabb megoldást kerestem. [25]

A következő technológia amit kizártam az a React Native volt, ez a React egyfajta változata ami natív élményt tud nyújtani weben, Androidon, iOS-en és asztali alkalmazáson is. Ez egy JavaScript/TypeScript és HTML alapú megoldás és a korábbi tapasztalataim alapján ezek a nyelvek nem alkalmasak egy nagy mérteű projekt fejlesztésre és karabantartására. JavaScriptben gyorsan lehet fejleszteni, de nagyon nehéz utána a kód továbbfejlesztése és karbantartása. [27]

A MAUI egy .NET alapú megoldás, így C# nyelven lehet programozni. A UI XAML alapokon nyugszik, hasonlóan, mint a WinUI. Így ez a megoldás az Windowsos asztali alkalmazásokhoz hasonlít a legjobban felfogásban és programozásban. Volt több elődje is, de jelenleg ez a legújabb változata a Windows alapokon nyugvó multiplatform fejlesztésnek. A legfőbb indok az volt, hogy én elsősorban Android fejlesztő vagyok és, ha van lehetőség akkor abból az irányból szerettem volna kiindulni. Egy másik érv ellene, hogy webes irányban nincs elmozdulás, míg az előző kettőnél igen, illetve a Compose Multiplatform is nyit ebeb az irányba. [18]

Mindent összevetve a Compose Multiplatform volt számora a legalkalmasabb és legérdekesebb technológia. Ebben volt a legtöbb tapasztalam és a későbbiekbén is szeretnék ezen a területen dolgozni, akár natív Android fejlesztés vagy a Compose és Kotlin Multiplatform világában. Továbbá mindenkor érdekes kihívás egy éppen aktívan fejlődő technológiát megismerni és dolgozni benne. Így azt is ki tudtam próbálni, hogy milyen nehézségek járnak egy natív alkalmazás multiplatformba való átültetésével.

## 4. fejezet

# Felsőszintű architektúra

Ebben a fejezetben bemutatom a szoftver felépítését. Ez a rész első sorban az architektúrális- és programszervezési megoldásokra fókuszál. Hasonlóan a 2. fejezet fejezethez ez is csak egy átfogó képet fog adni a szoftverről, a részletekbe nem merül el.

### 4.1. High level architektúra

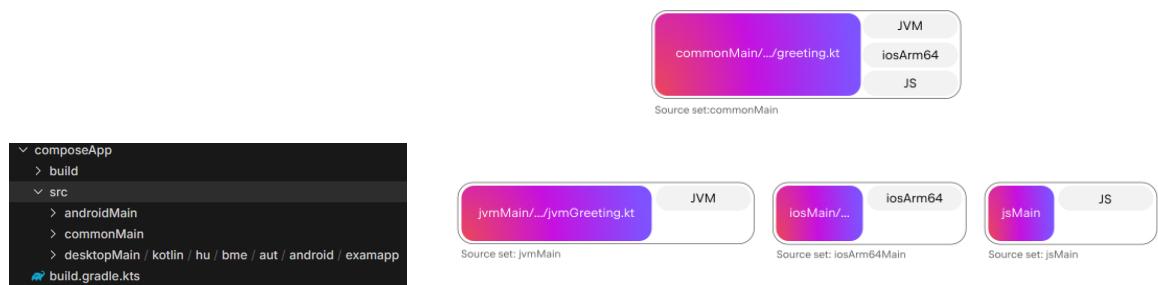
Ez az alfejezet bemutatja a forráskód szerevzését és a legfontosabb követet tervezési mintákat.

#### 4.1.1. A program struktúrális szervezése

Ez a rész a kód forrásfájlokba szervezését mutatja be. A hatékony munkához elengedhetetlen egy jól felépített és követett mintát konzisztensen használni a fejlesztés során.

##### 4.1.1.1. A Kotlin Multiplatform alkalmazások felépítése

Minden Kotlin Multiplatform projekt tartalmaz legalább 3 Main könyvtárat az src mapán belül. Szukséges egy commonMain mappa, ami a közös kódrészleteket tartalmazza. Minél több tartalom található ebben megvalóstíva és nem külön kiszervezve annál jobb a kód újrafelhasználhatóságunk. A többi Main mappa a platform specifikus kódrészleteket tartalmazza. Az én alaklamazásomban van egy androidMain és egy desktopMain, ez a jvmMain-nek felelhető meg. Sajnos eszközhiány miatt iOS-re nem tudtam elkészíteni az alkalamzást, mivel annak lebuildeléséhez szükség van egy Mac számítógépre.



4.1. ábra. Fájl struktúrája az alkalmazásnak. Második kép: [5]

A build megfelelő működéséért a Run Configurations fájlok és a build.gradle.kts fájlok felelnek. A gradel fájlokban kell a függőségeket megadni, a verziók támogatására használható a libs.version.toml fájl.

A 4.1. ábrán is látható a build.gradle.kts fájlt (4.1. kód részlet) is. Az ebben lévő Kotlin DSL-lel létrehozott Gradlenek is tükrözni kell ezt a felépítést.

```
import org.jetbrains.compose.desktop.application.dsl.TargetFormat //Importok
...
//Szükséges pluginok
plugins { alias(libs.plugins.kotlinMultiplatform)
    ...
}

kotlin {    // Projekt struktúrájának létrehozása
    androidTarget {    // Android taget beállítása
        ...
    }
    jvm("desktop") //Asztali alkalmazás beállítása

    sourceSets {    Itt kerülnek hozzáadásra a függőségek a különböző platformokhoz
        val desktopMain by getting

        //Android specifikus függőségek
        androidMain.dependencies { implementation(libs.androidx.activity.compose)
            ...
        }
        //Minden támogatott alkalmazás által használt függőségek
        commonMain.dependencies { implementation(compose.foundation)
            ...
        }
        //Asztali alkalmazás specifikus függőségek
        desktopMain.dependencies { implementation(compose.desktop.currentOs)
            ...
        }
    }
}

//Egyéb Android beállítások
android { namespace = "hu.bme.aut.android.examapp"
    ...
}
// Szükséges függőségek
dependencies { implementation(libs.androidx.ui.android)
    ...
}
//Egyéb asztali beállítások
compose.desktop { application { mainClass = "hu.bme.aut.android.examapp.MainKt"
    ...
}
    ...
}
```

#### 4.1. kód részlet. build.gradle.kts fájl struktúrája

Az alkalmazást az összes platformra le kell fordítani, így szükség van egy belépési pontra minden eszköznél. Ez értelem szerűen nem lehet a közös kódbázisban, mivel minden platform más módon tud inicializálni. Ellenben az a kód amit itt meg kell hívni, az már lehet egy közös Composable függvény. (4.5. kód részlet) Egy egyszerűbb alkalmazásnál, ahol megfelelő közös nézetet tudunk létrehozni ennyi platform specifikus kód elég is. Az optimális a fenti lenne, de ez általban nem lehetséges, így szükség van a közös kóból való "kilépésre" és a platform specifikus kód meghívására. Ez egyszerűen megvalósítható az expect és actual függvények és osztályok segítségével (3.1.8. alszakasz).

A fő mappastruktúrán belül is fontos a tervezés. 2 fő részből tevődik össze a UI és a Service mappákból. A Service tartalmaz minden közvetlenül nem a megjelenített képernyőkkel foglalkozó részeket. Itt található a navigáció, a PDF rendrelése, és kivételesen

itt van az ahhoz tartozó képernyő is. Továbbá a szövegfelismerés és a HTTP kommunikáció is itt található. Az utóbb az api mappában és ezen belül találhatók a dto-k (data transfer object) amik az alkalmazás modell rétegét adják.

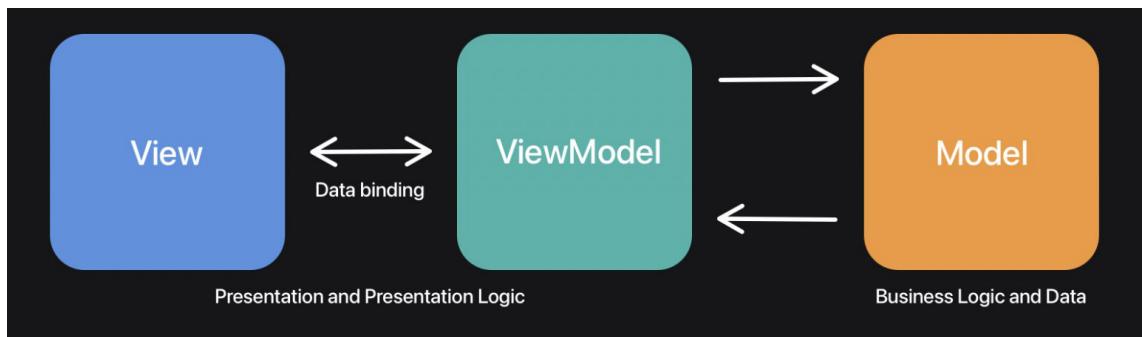
A UI mappában minden használati esetre van egy saját mappa ami a képernyőket tartalmazza, ezen kívül itt van a.viewmodel mappa is ahol hasonló szervezési struktúrban találhatóak a képernyőkhöz tartozó ViewModelek. Erre is több lehetséges megoldás lenne, például egy közös ViewModel a közös tematikájú képernyőkhöz. Ebben a mappában van még a components menü is ami azokat a Composable függvényeket tartalmazza amit több képernyő is fel tud használni, például egy dropdown lista vagy a navigációs TopAppBar.

Egy másfajta tervezés lehet az is, hogy a képernyőt leíró fájlok mellett közvetlenül vannak a viewmodelek és ezek vannak közös mappákba szervezve. Ezt elvettem, mivel így is túl sok mappa volt és nehezebben láttam át ezzel a második megoldással a program szerkezetét.

A legtöbb fájl a commonMainben van, így a platform specifikus mappákban nem található meg minden mappa a fentiek közül, de a szükség actual függvények ennek megfelelő szerkezetben találhatóak meg itt is.

#### 4.1.1.2. Követett tervezési minták

Az első és legfontosabb tervezési minta az **MVVM architektúra**. Ebben a mintában három fő komponenst lehet elkölníteni egymástól: View, ViewModel és Model. Mind-egyiknek megvan a szerepe és egy jól működő és karabantartható szoftver esetén mindegyik létfontosságú. Ezek a rétegek egymást között tudnak kommunikálni, a leggyakoribb irány a View -> ViewModel -> Model -> ViewModel -> View. Tehát a View kér adatot valamilyen felhasználói esemény hatására, és a ViewModel ennek teljesítése érdekében a Modeltől kéri el a megjelenítendő adatokat. Ennek a láncnak azonban szinte bármely része megtörténhet akár tetszőleges irányban is, attól függően, hogy milyen funkcionálitással rendelkezik.



4.2. ábra. MVVM minta. [29]

Ezek közül a legegyszerűbb a **Model réteg**. (4.2. kód részlet) Ennek egyetlen célja az adatok tárolása a memóriában. Célszerű ezt olyan formában megtenni ami már nem igényel sok átalakítást a köztes, ViewModel rétegben. Az adatok gyakran valamelyen REST APIból érkeznek, tehát egy másik szoftvertől. Ehhez úgynevezett DTO-kat (data transfer object) használunk, az én programomban ezek lényegében megegyeznek a modellekkel, mivel kifejzetten erre készült a REST API. Természetesen így is van szükség egyéb data classokra vagyis valós modellekre. A UInak szüksége lehet egyéb értékekre, például, hogy megfelelő módon van-e kitöltve létrehozás során az objektum vagy sem. Továbbá az összetett elemek esetén a hivatkozásokat a uuidkon keresztül fel kell oldani megjelenítés

előtt. Ezt a szerepet általában már a ViewModel végzi el. Amennyiben nem saját adatforrást használunk, vagy komplexebb adatokat kapunk érdemes külön modell osztályokat létrehozni csak a szükséges adatokkal tovább dolgozni. Ezt még a HTTP kommunikáciáról felelős komponensben érdemes megtenni.

```
import kotlinx.serialization.Serializable // Szerilizáció

@Serializable // Szerilizációért felelős kód rész
data class TopicDto(      // Kotlin data class
    val uuid: String = "",
    val topic: String,
    val description: String,
    val parentTopic: String = ""
)
```

**4.2. kód részlet.** Egy dto struktúrája. Szerilizáció: 3.1.3. alaszakasz

A leggyakoribb mód erre a HTTP alapú kommunikáció, ahol az adatok JSON formátumban érkeznek. Elképezelehtő erre más alternatíve, például az adatok XML formátumban érkeznek. Más módon működő rendszerek is lehetnek, például WebSocketek, ahol az első csatlakozás során kialakul egy kétirányú csatorna ahol, mind a szerver mind a kliens tud kommunikációt kezdeményezni. IoT környezetben az MQTT protocolt szokás alkalmazni. "Ez egy pub/sub protokoll, amely kis csomagmárettel és alacsony sávszélességgel rendelkezik, így ideális korlátozott hálózatokhoz és alacsony feldolgozási teljesítményű eszközökhöz. Az MQTT képes kezelni a szakaszos hálózati kapcsolódást, és támogatja a szolgáltatásminőség (QoS) szinteket a megbízható üzenettovábbítás biztosítása érdekében." [13] Egy további megoldás a gRPC ami valós alternatíva lehet a REST API kommunikációra ebben a környezetben is. "A gRPC egy nyílt forráskódú keretrendszer, amelyet a Google fejlesztett RPC API-k építésére. Lehetővé teszi a fejlesztők számára, hogy szolgáltatás-interfészeket definiáljanak, valamint kliens- és szerveroldali kódot generáljanak több programozási nyelven. A gRPC protokoll buffereket és nyelvfüggetlen adat-szerilizációs formátumot használ a hatékony adatátvitel érdekében, ami ideálissá teszi magas teljesítményt igénylő alkalmazások számára. A gRPC nem feltétlenül a legjobb választás nagy mennyiséggű adatmanipulációhoz vagy olyan alkalmazásokhoz, amelyek széleskörű kliens támogatást igényelnek. Ugyanakkor a gRPC magas teljesítményéről és alacsony erőforrásigényéről ismert, így jó választás azokhoz az alkalmazásokhoz, amelyek gyors és hatékony kommunikációt igényelnek a szolgáltatások között." [13]

A köztes szinten a **ViewModel** található. Megoldáshoz az Android ViewModel Kotlin Multiplatform implementációját használtam fel (3.1.1.2. alaszakasz), de bármi hasonló megoldás megfelel. Ennek a rétegnek számos feladata van. Az adatok lekérdezése itt valósul meg. Úgynevezett coroutine scopekban kezdeményezhetünk hosszabb ideig tartó függvényhívásokat. "A Kotlin standard könyvtára csak minimális alacsony szintű API-kat biztosít, hogy más könyvtárak használhassák a koroutinekat. Ellentétben sok más hasonló képességű nyelvvel, az async és await nem kulcsszavak a Kotlinban, és még a standard könyvtár részei sem. Ezenkívül a Kotlin felfüggeszthető függvény (suspending function) koncepciója biztonságosabb és kevésbé hibára hajlamos absztrakciót kínál az aszinkron műveletekhez, mint a jövők (futures) és ígéretek (promises). A kotlinx.coroutines egy Jet-Brains által fejlesztett gazdag könyvtár koroutinekhoz. Számos magas szintű, koroutine-kompatibilis primitívet tartalmaz, amelyeket ez az útmutató is tárgyal, beleértve a launch, async és más függvényeket." [4] Röviden összefoglalva aszinkron módon működnek így és nem UI szalon hajtónak végre ezáltal nem blokkolják azt és reszponzív marad a teljes idő alatt. Mindebből következik, hogy a REST API service hívásai innen indulnak, majd kerülnek bele Statekbe vagy StateFlowkba (3.1.1.1. alaszakasz). Egy érdekesebb megközelítés a LiveData. Ez az eszköz az Observer mintát valósítja meg. Amennyiben például

egy WebSocket alapú megoldást választottam volna, akkor a szerver autmoatikusan tudna üzenetet üldeni amiben leküldi az új adatot a kliensnek és egyéb frissítés nélkül megjelne az új adat. Ez egy példa a Model -> ViewModel -> View irányra. A legtöbb chat applikáció (például Messenger) is hasonéó megközelítést alkalmazhat.

A ViewModelek további feladatai közé tartozik az adatok transzformálása. Szükség lehet a hivatkozások feloldására például egy kérdés esetén nem egy uuid-t szeretne a felhasználó látni, a pont és a téma mezőkben, hanem azok neveit. Ezen kívül biztosíthat egyéb függvényeket az adatok módosítására, tovább navigálásra vagy bármilyen általános célokra. Ezzel elkerülethető, hogy logikát leíró kódot a Viewn kelljen definálni, elég egy összefogó részben levő függvényt meghívni. (4.3. kódrészlet)

```
data class TopicDetails(
    val id: String = "",
    val topic: String = "",
    val parent: String = "",           //Ez itt még egy uuid
    val description: String = "",     //Itt már fel lett oldva, ez fog a UI-on megjelenni
)

fun TopicDetails.toTopic(): TopicDto = //Ez egy extension function ami visszaalakítja a UIon megjelenő adatot a REST API által kezelhrtő formára
    TopicDto(
        uid = id,
        topic = topic,
        parentTopic = if (parent == "null") "" else parent,
        description = description,
    )

// A névfeloldott változat kiégszül egy valid mezővel, mivel ez egy szrekeszthető nézethez tartozó modell lesz
fun TopicDto.toTopicUiState(isEntryValid: Boolean = false, parentName: String): TopicUiState =
    TopicUiState(
        topicDetails = this.toTopicDetails(parentName),
        isEntryValid = isEntryValid
    )

//Biztosított függvény az adatok módosítására
fun updateUiState(topicDetails: TopicDetails) {
    topicUiState =
        TopicUiState(topicDetails = topicDetails, isEntryValid = validateInput(topicDetails))
}
```

**4.3. kódrészlet.** Példa az dto átalakítására a valós UIon használt modellé és egy példa logikát leíró függvényre. Az eredeti dto: 4.2. kódrészlet

A **View** a megjelenítési réteg, a felhasználó ezzel a résszel tud közvetlenül interakcióba lépni. Általános működési elve a következő: amikor a felhasználó az adott képernyőre navigál az ViewModel betölti az adatokat. Ezt követően a felhasználó szabadon végezhet műveleteket rajta. Ahogy a 2.1. ábrán is látható megtekintheti az adott oldalt, esetenként törölheti vagy szerkesztheti az adatokat vagy felvehet újat. Ezeket a funkciókat a megfelelő ViewModel beli függvény meghívásával érhetjük el és valamely felhasználói interakció váltja ki például egy gomb megnyomása.

```
@Composable
private fun NewTopicScreenState(
    viewModel: TopicEntryViewModel, ...           // A használt ViewModel
) {
    ...
    Scaffold(...) { innerPadding ->
        TopicEntryBody(
            topicUiState = viewModel.topicUiState,          // A megjelenítendő UI adat
            onTopicValueChange = viewModel::updateUiState,   // ViewModelben definált függvény paraméterként való átadása
            onSaveClick = {

```

```

        coroutineScope.launch {
            funkció. Szintén a ViewModel függvényét hívja meg
                if (viewModel.saveTopic()) {
                    navigateBack()
                } else {
                    showNotify = true
                    notifyMessage = "Topic with this name already exists"
                }
            },
            modifier = Modifier
                .padding(innerPadding)
                .verticalScroll(rememberScrollState())
                .fillMaxWidth()
        )
    }
}

```

#### 4.4. kódrészlet.

Egy a View rétegbe tartozó felület leírásának részlete

Összefoglalva az MVVM tervezési mintát azt lehet mondani, hogy nagyon hatékony használható és önmagában ezzel jelentősen növelhető a kód újrafelhasználhatósága. A mai lehetőségeket kihasználva a Compose Multiplatform területén ViewModelet, dto-kat és Modeleket elegendő egyszer, a commonMain alatt létrehozni és megvalósítani. Az ezekhez tartozó esetleges függvényeket, amennyiben szükség van rá elegendő egy actual/expect párral lecserélni, nálam a kamera és a PDF exportális is így működik. (4.6. kódrészlet) Az **expect/actual** tervezési elv működik minden rétegen, de általában csak a UI és néhány specifikus funkció megvalósítására van szükség a logika megvalósításához. Ilyenkor is elég egy közös ViewModel és a build rendszer behelyesíti a megfelelő függényt a platformnak megfelelően.

```

// Tipikus Android compose alkalmazás. (androidMain)
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            App()
        }
    }
}

//Kotlin swing alkalmazás. (desktopMain)
fun main() = application {
    Window(
        onCloseRequest = ::exitApplication,
        title = "Exam App",
    ) {
        App()
    }
}

//A hívott közös kód. (commonMain)
@Composable
fun App() {
    MaterialTheme {
        NavigationComponent()
    }
}

```

#### 4.5. kódrészlet.

Alkalmazás elindítása

```

//Közös függvény fejléc, lehet alapértelmezett paramétere is.
@Composable
internal expect fun Notify(message: String)

//Android megvalósítás.
@Composable

```

```

internal actual fun Notify(message: String) {
    Toast.makeText(
        LocalContext.current, message, Toast.LENGTH_SHORT
    ).show()
}

//Asztali alkalmazás megvalósítása.
@Composable
internal actual fun Notify(message: String) {
    if (SystemTray.isSupported()) {
        val tray = SystemTray.getSystemTray()
        val image = Toolkit.getDefaultToolkit().createImage("logo.webp")
        val trayIcon = TrayIcon(image, "Desktop Notification")
        tray.add(trayIcon)
        trayIcon.displayMessage("Desktop Notification", message, TrayIcon.MessageType.INFO)
    } else {
        ...
    }
}

```

**4.6. kódrészlet.** Egyszerűbb példa az expect és actual függvények használatára. Debugolás során használt kódrészlet.

A másik legfontosabb architekturális elem a UiState használata. (4.7. kódrészlet) Ezzel a mintával egyszerűen kezelhető több fajta képernyő ugyan azon az oldalon. A megvalósítása sealed interfések segítségével történik, így a Kotlinban a switch-case szerkezetnek megfeleltethető when használatával egyszerűen válthatunk közöttük. Ebben a mintában gyakran három állapta van a képernyőnek: loading, succes és error.

A hosszú ideig tartó HTTP kérések ideje alatt így egy töltőképernyőt mutathatunk egy látszólag lefagyott vagy hiányos oldal helyett. Az adatok megérkezése után átadhatjuk azokat a Succes képernyőnek, hiba esetén pedig mutathatjuk a hiba képernyőt ahol található valamilyen magyarázat a felhasználó számára, példál "Nem sikerült betölteni a kérőt oladlt mert a hálózat állapota nem megfelelő".

```

sealed interface TopicDetailsScreenState {
    data class Success(val point: TopicDto) : TopicDetailsScreenState
    data object Error : TopicDetailsScreenState{var errorMessage: String = ""}
    data object Loading : TopicDetailsScreenState
}

```

**4.7. kódrészlet.** UiState megvalóstása

A képernyő állapotát a ViewModelből lehet szabályozni, ahol felveszünk erre egy State-et. A képernyőre navigálás során az alapértelmezett érték a töltés. Az adatok megérkezése után a staten beállítjuk a megfelelő értéket és a View erről kap egy értesítést és lefut a recomposition. Ennek hatására megjelenik az új nézet. (4.8. kódrészlet)

```

var topicDetailsScreenState: TopicDetailsScreenState by mutableStateOf(TopicDetailsScreenState.
    Loading) // State alap loading értékkel
fun getTopic(topicId: String){
    topicDetailsScreenState = TopicDetailsScreenState.Loading // Ha nem lenne beállítva beállí
    tuj a loadingot
    viewModelScope.launch { // Aszinkron hívás környezete, így a loading animációt nem blokkoljuk
        topicDetailsScreenState = try{ // Az eredményen megérkezése után vagy mutatjuk az adatot
            ...
            val result = ApiService.getTopic(topicId)
            ...
            TopicDetailsScreenState.Success(result)
        } catch (e: ApiException) { // ...vagy egy hiba képernyőt
            TopicDetailsScreenState.Error.errorMessage = e.message ?: "Unknown error"
            TopicDetailsScreenState.Error
        }
    }
}

```

**4.8. kódrészlet.** UiState beállítása

Összefoglalva ezt a mintát: a felhasználót aktívan tudjuk tájékoztatni az program állapotáról. A töltő képernyő miatt tudni fogja, hogy a háttérben tart az adatok betöltése és nem lesz olyan érzése mintha lefagyott volna az alkalmazás. Hiba esetén is tudunk tájékoztatást adni így lehetőséget adunk a felhasználónak, hogyha nála van a hiba (nincs internet) megoldja, vagy esetlegesen tudja jelezni a megadott elérhetőségen, hogy problémát tapasztal a működésben és a megjelenített információ miatt ezt vszonylag pontosan meg tudja tenni. Ezek hatására a felhasználói élmény jelentősen javulni tud.

## 4.2. Rendszer felépítései, komponensei

Ebben az alfejezetben komponens diagramokon keresztül mutatom be az alaklamzást. Elsőnek adok egy általános nézetet, majd a fontosabb, több tartalommal rendelkező egységeket a kisebb alegységein is bemutatom.

"Az UML komponensdiagramokat az objektumorientált rendszerek fizikai aspektusainak modellezésére használják, amelyek célja a komponensalapú rendszerek vizualizálása, specifikálása és dokumentálása, valamint végrehajtható rendszerek létrehozása előre- és visszafelé történő tervezéssel. A komponensdiagramok lényegében osztálydiagramok, amelyek a rendszer komponenseire összpontosítanak, és gyakran a rendszer statikus implementációs nézetének modellezésére használják." [28]

Ebből az idézetből kiderül, hogy ebben a diagram típusban a fizikai egységek jelennek meg. Ezt érthetjük szó szerint is, például más gépen futó backend és frontend, de akár önálló fordítási egységet is érthetünk alatta. Ilyen önálló egység lehet a bussines logic réteg, .NET környezetben ez egy teljesen általános megoldás, ahol valódi fordítási egységet alkot és önálló DLL jön belőle létre. Hasonló módon foghatjuk fel a Compose Multiplatformban szereplő közös és specifikus kódokat is.

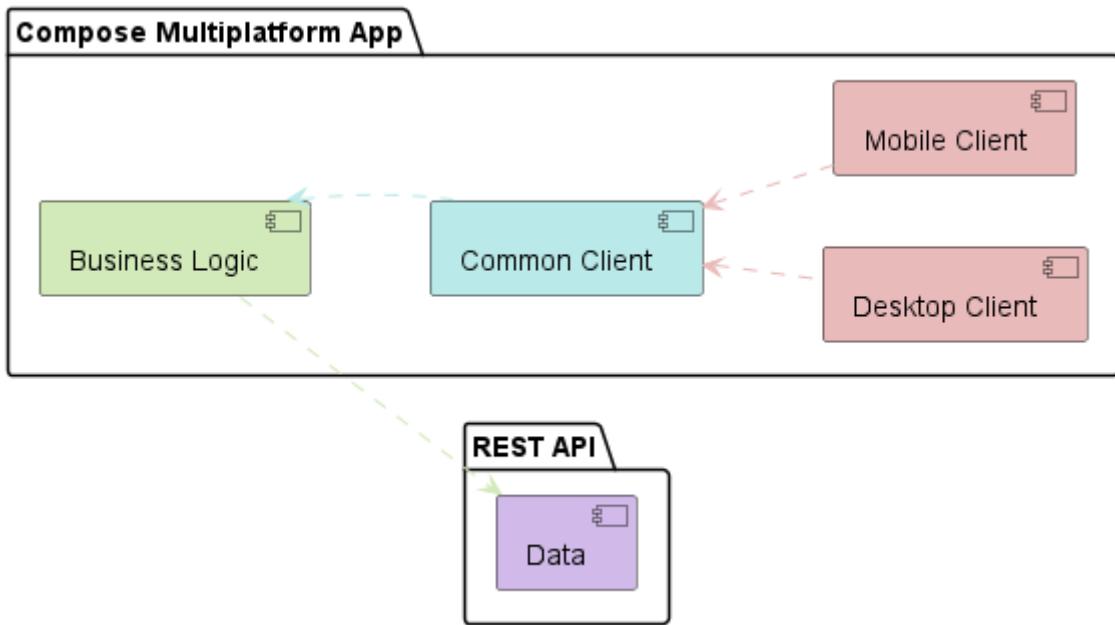
### 4.2.1. Általános komponensek felépítése

Két fő komponensből áll az alklamazás, a Compose Multiplatform applikációból és az adatokat kiszolgáló REST APIból. (4.3. ábra) Ebben a félévben a klines oldalon volt a hangsúly, ezért a backend részt nem bontom fel további részekre, kezelhetjük egyfajta "black box"-ként.

A kliens oldalon a bussines logic komponens kommunikál a backenddel, így függ tőle. Az abban történő változás vagy hiba kihat ennek a komponensnek a működésére. Többek között ebben az egységen találhatóak a ViewModelek és a Servicek közé sorolt funkciók.

A Common Client rész tartalmazza a View réteget. Itt találhatóak a Composable függvényekkel leírt képernyők. Mivel ezeken az elemeken a ViewModelből származó adatok jelennek meg a helyes működés szerint, ezért függ a Business Logictól. Ezáltal tranzitívan függ a backend helyes működésétől is. Ennek a felépítéséről és komponenseiről lesz egy részletesebb ábra.

Megjelnnek még az ábrán a Mobile Client és Desktop Client részek is. Ez a két komponens valósítja meg a platform specifikus kód részletek megvalósításáért és a belépési pontok megvalósításáért felel. Itt is érdemes észrevenni, hogy ezek a komponensek függnak a közös kódktól. Ha megváltozik valami benne az minden platform specifikus kódra hatással lesz, ha létrehozunk egy expect függvényt azt minden más komponensben meg kell valósítani. Ezkről nem fogok ebben a fejezetben részletesebben foglalkozni.



**4.3. ábra.** Átfogó komponens diagram.

#### 4.2.2. Business logic komponensek felépítése

A **bussines logic komponens** összetett és több alkotóelemből épül fel. (4.4. ábra) A ViewModelek önmagukban lehetnek komponensek, akár külön-külön is, de a diagramon egybe tüntetem fel. Ezek a viewModelek függnek a Servicektől.

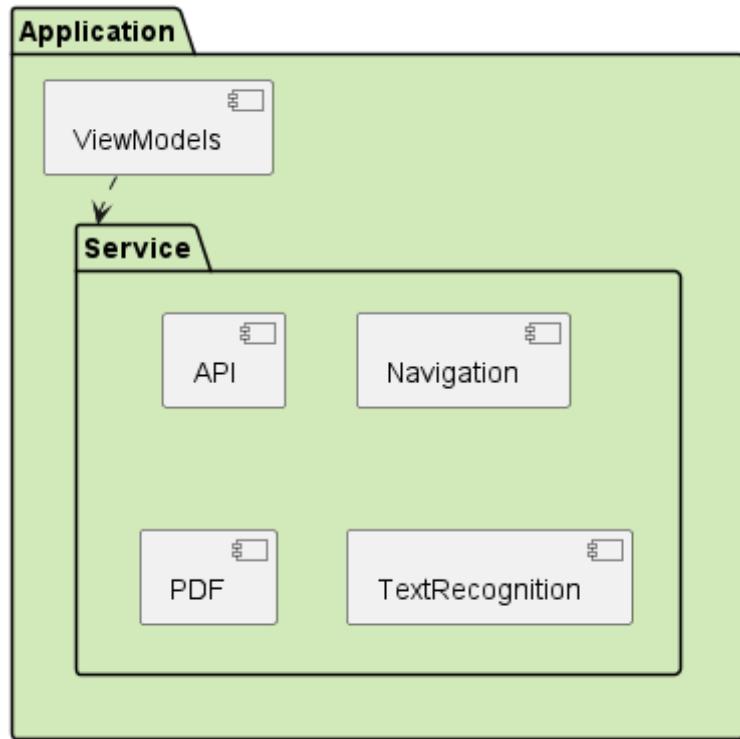
A Serviceek közé tartozik az API kommunikáció, a navigáciért felelős komponens is. A PDF elkészítéséért felelős kódrészek is egy komponensként kezelhető, hasonló módon, mint a szövegfelismerésért felelős funkció is. Az ebben az egységebn levő komponensek nagyrészt nem függnek szorosan más komponensektől, ez alól kivétel a HTTP kommunikáció ami a backendtől függ.

#### 4.2.3. Common Client komponensek felépítése

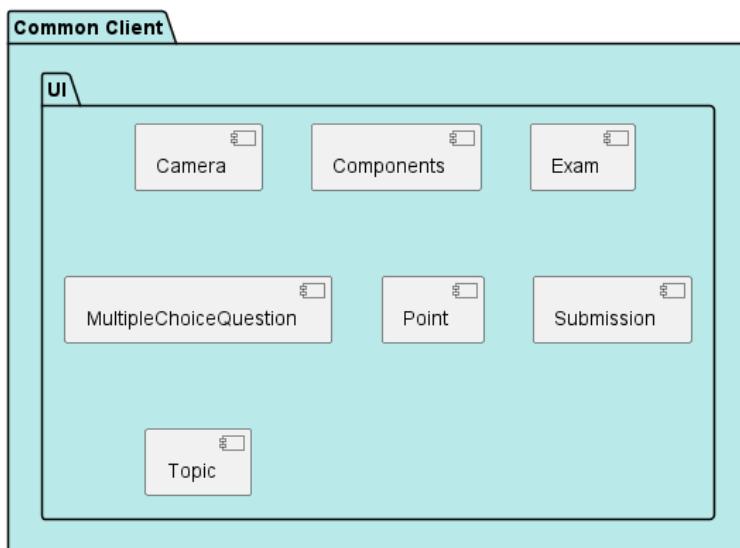
A **common Client komponens** is számos részből épül fel. (4.5. ábra) Ebben a részben találhatóak a képernyőn megjelenő elemek leírásáért felelős részek.

A kisebb komponenseket tartalmazó résztől több másik is függ, ez az ábrán az átláthatóság miatt nem szerepel. Egy-egy ilyen UI komponens egy vagy több viewModelltől is függhet, de egymás között nincs egyéb függőség, így szabadan eltávolíthatóak vagy hozzáadhatóak új nézetek.

Ezektől a komponensektől a tényleges implementációk függnek, így a Mobil és asztali komponens is. Az itt történt bármilyen változás kihatással van a valós implementációra. A közös elemek módosítása megváltoztatja minden platformon a képernyő kinézetét. Ha új expect függvényt veszünk fel, a többi komponenst is külön el kell készítenünk.



**4.4. ábra.** Business logic komponens diagram.



**4.5. ábra.** Közös UI komponens diagram.

## 5. fejezet

# Részletes megvalósítás

Ebben a fejezetben a korábban röviden bemutatott építőelemekről adok egy részletsebb leírást. Bemutatom, hogy az én projektemben milyen megoldásokat valósítottam meg a használatukkal. Ahol van értelme diagramokon keresztül mutatom be a működését, ami látványos azt képekkel is illusztrálom. Több kódrészlet is szerepelni fog a fejezetben amiknek célja a kód működésének jobb mégétése.

### 5.1. Compose

A Composerról korábban már adtam egy átfogó leírást a 3.1.1. alszakaszban, most ebben a részben részlesebben bemutatom az alapaeleveket, használtatát és a keletekező képernyőket.

#### 5.1.1. Compose alapelvek, használata valós környezetben

Mivel a Compose egy deklratív UI kit ezért összefoglalnám a legfontosabb részeit. A UI elmeket nem lehet lehet direktben példányosítani, és kódból később elérni és ezen a referencián eresztő módosítani a tulajdonságait. minden egyes elem egy függvényhívást jelent, az állapotát és tulajdonságait pedig ezen belül lehet szabályozni és bállítani egyéb objektumokon és stateken keresztül. Ezeket az állapotokat általában nem a Composable függvénye belül tároljuk hanem egy ViewModelben, mivel az tartósabb. (3.1. ábra) Egy figyelt állapot hatására meghívódik a recomposition, azaz a függvény újra meghívódik és frissíti a UI állapotát. Az általános elv az, hogy egy Composable függvény nagy betűvel kezdődik, és attól lesz egy függvény Composable, hogy elé helyezzük a @Composable annotációt.

Mi is az a recomposition? Röviden valamilyen állapotváltozást követő újra kirajzolódás. Gyakran valamilyen felhasználói interakció váltja ki (gomb megnyomása, görgetés), de a ViewModelből is jöhét ez a változás (megérkezik a kezdeti adat, folyamatosan frissülő információk jönnek), de okozhatja egy animáció is.

A Compose hasznátának van 5 fontos alapszáblya amelyek betartásával nem csak helyes, de gyors és dinamikus frissülő képernyőket tudunk egyszerűen létrehozni.

1. A Composable függvények bármilyen sorrendben végrehajthatók.
2. A Composable függvények párhuzamosan hajthatók végre.

3. A Recomposition kihagyja a lehető legtöbb Composable függvényt és lambdát.
4. A Recomposition optimista, és leállítható menet közben.
5. Egy Composable függvény nagyon gyakran is futhat/ismétlődhet, olyan gyakran is, mint egy animáció minden képkockája.

Pontosan mit is jelentenek ezek a szabályok? Az 1. és 2. pont alapján következik, hogy minden Composable függvény önálló legyen, nem függhet egy másik Composable függvénytől sem. Továbbá fontos, hogy ha ugyan azon stateet változtatják akkor az ne fusson le minden recomposoition során, ha ezt nem tartjuk be akkor folyamoatosn újra kirajzolódik a képernyő és lefagy az eszköz. A 3. pont értelmezése, hogy csak a szükséges és részek renderelődnek újra, ezzel gyorsítva a folyamatot. Így amennyiben több függvényt is meghívunk egy másikon belül csak azok fognak újra rajzolódni amik függnek a változott állapottól. Előfordulhat, hogy minden szeretnénk újra firissíteni, vagy több függvényt ami nem függ tőle, ehez használhatunk az állapottól függő Launched effectet vagy átadhatjuk az állapotot ezeken a függvényeknek is, és ott sinálhatunk vele egy egyszerű és gyors műveletet. A 4. és 5. pont összefügg, ha új állapot érkezik és nem fejeződött be a recompositon akkor előrről kedzi a folyamatot. Ne itt hajtsunk végre egy hosszú műveletet (HTTP kérések indítása), ez felesleges overheadet jelent mert senki nem állítja le az aszinkron hívást, és nem is tud hova visszaérkezni az adat így feleslegesen használtuk az erőforrásokat mind a két oldalon. Ne állítsunk be egy olyan értéket sem itt amire szükségünk lehet később, mivel nem biztos, hogy eljut a futás addig vagy, lehet, hogy a következő másodpercben már felül lett írva mással.

Az alábbi kód részletben bemutatom, hogy milyen részekből tevődik össze egy Composable függvény.

```
@Composable // A kötelező annotáció
fun ExamEditResultScreen() // Függvényparaméterek
    navigateBack: () -> Unit, // Egy lambda függvény fejléce
    viewModel: ExamEditViewModel, // ViewModel átadása paraméterként
    modifier: Modifier = Modifier // Modifier amivel a kinézetet lehet testre szabni.
) {
    val coroutineScope = rememberCoroutineScope() // Coroutine scope lekérése, így lambda függvényben használható, mivel annak a törzse nem Composable függvény.
    var showNotify by remember { mutableStateOf(false) } // Statek amit el lehet tárolni a Composable függvényben mivel ezen kívül nincs használva.
    var notifyMessage by remember { mutableStateOf("") } // Statek amit el lehet tárolni a Composable függvényben mivel ezen kívül nincs használva.
    if (showNotify) { Notify(notifyMessage); showNotify = false} // Állapottól függő megjelenítés
    Scaffold( // Beépített Composable elem amivel egy általános Android nézetet könnyen létre lehet hozni.
        topBar = { TopAppBarContent(stringResource(Res.string.exam_edit), navigateBack) }, // Itt egy topBart lehet könnyen megadni.
    ){innerPadding -> // Ez a belső része a képernyőnek, a fő része az alkalmazásnak.
        ExamEntryBody( // Egy másik Composable elem meghívása, ami a megjelnésért felel.
            examUiState = viewModel.examUiState, // Szüksége van viewModelben lévő statere
            onExamValueChange = viewModel::updateUiState, // Nevesített függvényt így lehet átadni egy lambda függvénynek
            onSaveClick = { // Lambda függvényt helyben is megadhatunk
                coroutineScope.launch { // Itt látható egy példa a trailing lambda használatára
                    if(viewModel.updateExam()){ navigateBack() } // If-else szerkezet is nyugodtan használhatunk. Siker esetén visszanavigál
                } else{ // Egyékként a sikertelenséget közli a felhasználóval
                    showNotify = true
                    notifyMessage = "Exam with this name already exists"
                }
            },
            modifier = modifier.padding(innerPadding))}}
}
```

## 5.1. kód részlet. Composable függvény részei.

### 5.1.2. Alkalmazás bemutatása képernyőképekkel

Ebben az alfejezetben bemutatom az elkészült képernyőket és a hozzájuk tartozó érdekes részeket, megoldásokat.

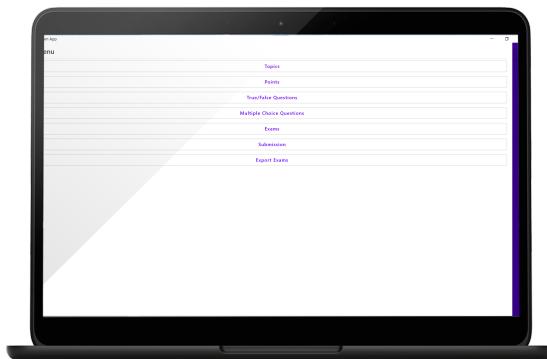
#### 5.1.2.1. A fő képernyők

A két alkalmazás típusra más-más a felhasználói igény. Androidon, ahol általában egy álló képernyő van (5.2. ábra jobb oldal) és a telefont a jobb alsó sarokban fogjuk meg nem esik kézre a fent bekapcsolható menü (bár oldalra húzással is előhozható). Asztali alkalmazás estén viszont jobban mutat ez a megolads, és egy nagyobb képernyő ahol a kiválasztott funkció képernyője meg tud jelenni. (5.2. ábra bal oldal és 5.1. ábra) Későbbiekben a navigálásban is segít, így könnyebb képernyőt váltani, a fenti vissza gomb használata kevésbé kényelmes, mint Androidon egy gyos mozdulat a telefon alján, ahol a kenzünkkel tartjuk azt.

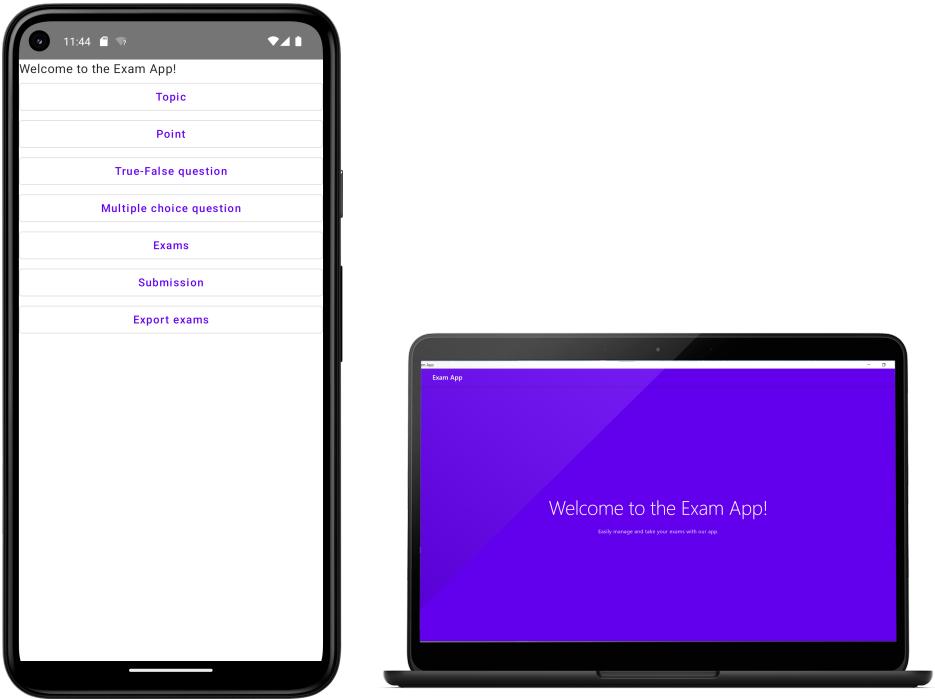
Ennek megfelelőn a MainScreen függvény a közös kódban egy expect függvény és így minden platformra az oda illő actual implementációt tudjuk megírni és alkalmazni. Végül más megoldást választottam a navigációhoz, így az asztali alkalmazásnál a szintén expect/actual NavHost létrehozáskor nem kapnak értéket a függvények.

```
@Composable
expect fun MainScreen(
    navigateToTopicList: () -> Unit = {},
    navigateToPointList: () -> Unit = {},
    navigateToTrueFalseQuestionList: () -> Unit = {},
    navigateToMultipleChoiceQuestionList: () -> Unit = {},
    navigateToExamList: () -> Unit = {},
    navigateToExportExamList: () -> Unit = {},
    navigateToSubmission: () -> Unit = {},
    onSignOut: () -> Unit = {},
)
```

**5.2. kódrészlet.** Expect Főképernyő.



**5.1. ábra.** Kinyitott menüsáv kinézete az asztali alkalmazáson



**5.2. ábra.** A fő képernyő Android és asztali alkalmazáson

### 5.1.2.2. Elemek listázása

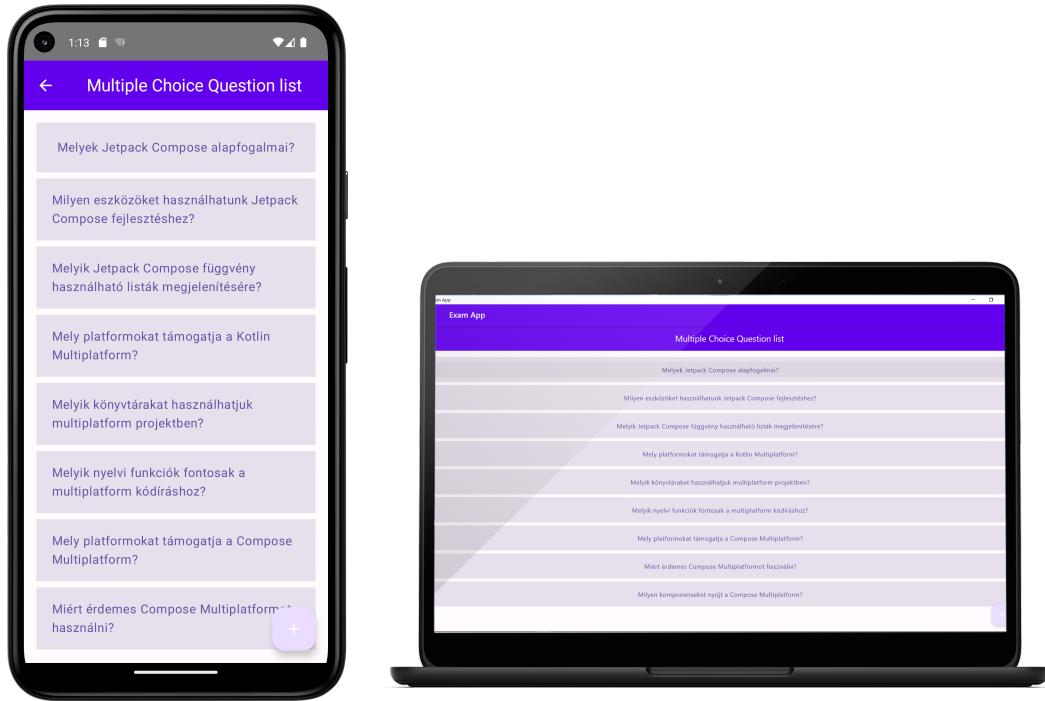
Az elemek felsorolásáért egy LazyColumn felel. (5.3. kódrészlet) Ez egy előre létrehozott, beépített Composable függvény. Több hasznos tulajdonsággal rendelkezik. Alapértelmezett módon görgethetővé válik, ha több elem van benne, mint amennyi kifér egy képernyőre. (5.3. ábra) Alkalmas nagyon sok, akár végtelen elem megjelenítésére, bár ehhez szükség van az adtaok ügyes betöltésére is. Mindig csak annyi elemet renderel ki amennyi éppen szükséges, látszódik. Előtte és utána van egy kis tartalék puffer, így görgetés esetén nem kell várni az új adtaok betöltésére. Az elemek testre is szabhatók benne, tökéletes alaklmaási területe lehet egy chat alkalmazás, ahol az elemeket az üzenet küldője alapján lehet színezni.

Ezen a képernyőn is a Scaffold megoldást választottam így kényelemsen elfér egy navigációs sáv és egy Floating Action Button az új elemek felvételéhez. Ezekben túl több korábban is bemutatott elem megjelenik ebben a kódrészletben is. Ahogy látszik az alábbi kódrészletben is, attól függően lehet elemeket beállítani, hogy milyen feltételhet kötjük az elemek megjelnítését.

```
@Composable
fun MultipleChoiceQuestionListResultScreen(
    ...
) {
    Scaffold(
        topBar = { TopAppBarContent(stringResource(Res.string.multiple_choice_question_list),
            navigateBack) },
        floatingActionButton = {
            FloatingActionButton(
                onClick = { addNewQuestion() }, ...
            ) {Icon(Icons.Filled.Add, contentDescription = "Add") }
        }
    ) { padding ->
        LazyColumn(
            contentPadding = padding,
            ...
        )
    }
}
```

```
) {  
    if (questions.isEmpty()) {  
        item { Box(modifier = Modifier ...) { Text(...) } }  
    } else {  
        items(questions) { question ->  
            TextButton(  
                onClick = { navigateToMultipleChoiceQuestionDetails(question.uuid) },  
                modifier = Modifier ...  
            ) {  
                Text(text = question.name, ...)  
            }  
        }  
    }  
}
```

### 5.3. kódrészlet. Listázó képernyő.



**5.3. ábra.** A fő képernyő Android és asztali alkalmazáson

### 5.1.2.3. Részletes nézet

A részletes oldalak általában csak az összes adatot felsorolják és mutatják meg a felhasználónak. Ezen kívül tartalmaznak egy törlés opciót ami egy megerősítő ablakot dob fel, mivel a törlés az végeleges. Továbbá innen mehetünk át a szerkesztő oldalra is.

A 5.4. ábra egy érdekesebb megoldást mutat be. Az oldal tetején a szokásos navigációs savon kívül elhelyezkedik egy a kérdések közötti keresést segítő rész. Szűrhetünk kérdéstípusokra, ezt a csúszka segítségével tehetjük meg, ez egyúttal át is színezi azt a könnyebb megkülönböztethetőség kedvéért. Ezen kívül szűrhetünk témaakra is. Az utolsó dropdown menü segítségével választhatunk ki egy kérdést amit a gombbal hozzá adhatunk a listához.

A lista része szintén LazyColumn alapú, itt a kérdések típusa alapján vannak színezve az elemek. A kérdésekre kattintva lenyílnak és megnézhetjük a legfontosabb adataikat, és innen is lehet törölni a kérdéseket a listából. Ezzel csak a számonkérésből törlődnek, nem a teljes adatbázisból. Hosszan lenyomva az elemet elhúzhatjuk felfelé és lefelé ezzel szabadon átrendezhetjük az kérdések sorrendjét.

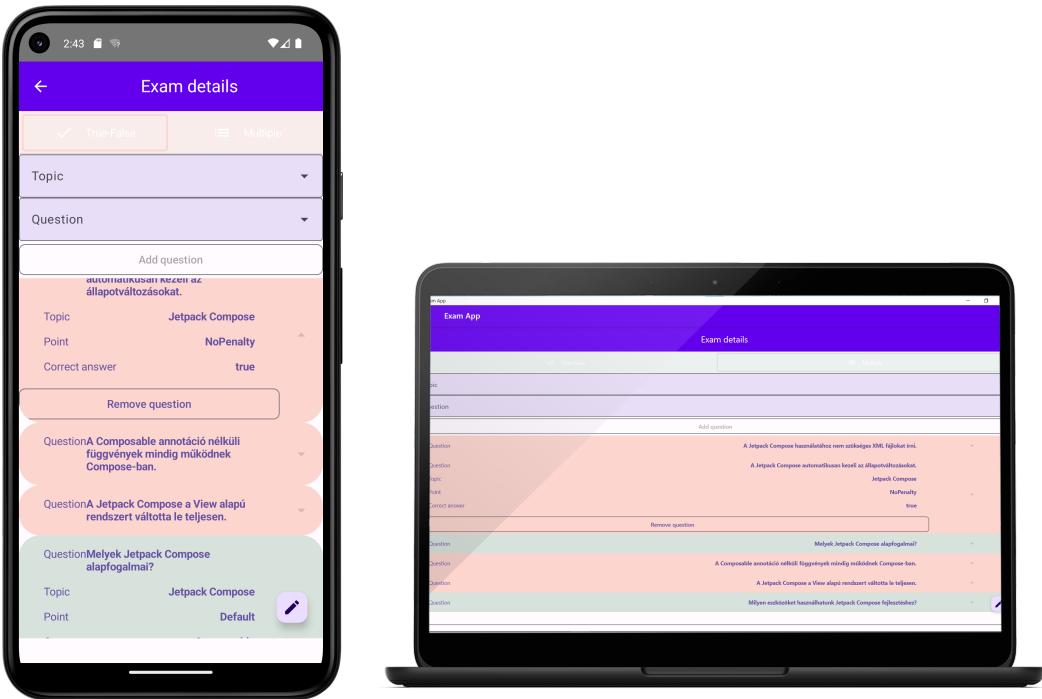
```

@Composable
fun ExpandableQuestionItem(...){
    var expandedState by remember { mutableStateOf(false) }
    val rotationState by animateFloatAsState(
        targetValue = if (expandedState) 180f else 0f, label = ""
    )
    val coroutineScope = rememberCoroutineScope()
    Card(
        modifier = Modifier.fillMaxWidth()
            .animateContentSize(
                animationSpec = tween(
                    durationMillis = 300,
                    easing = LinearOutSlowInEasing)),
        onClick = { expandedState = !expandedState}
    ) {
        Row(modifier = Modifier.background(if(question.typeOrdinal == Type.trueFalseQuestion.ordinal)
            PaleDogwood else Green)) {
            Column(...) {
                when (question.typeOrdinal) {
                    Type.trueFalseQuestion.ordinal -> {
                        val trueFalseQuestion = question as TrueFalseQuestionDto
                        if (expandedState) {
                            TrueFalseQuestionDetails(
                                trueFalseQuestion = trueFalseQuestion.toTrueFalseQuestionDetails(...),
                                modifier = Modifier.fillMaxWidth(),
                                colors = CardDefaults.cardColors(
                                    containerColor = PaleDogwood,
                                    contentColor = Purple40
                                )
                            )
                            RemoveButton(coroutineScope, examViewModel, question)
                        } else {
                            CollapsedQuestion(
                                question = trueFalseQuestion.question,
                                containerColor = PaleDogwood, contentColor = Purple40
                            )}}
                    Type.multipleChoiceQuestion.ordinal -> {...}
                }
            IconButton(modifier = Modifier.weight(1f).alpha(0.2f).rotate(rotationState),
                onClick = {expandedState = !expandedState}) {
                Icon(
                    imageVector = Icons.Default.ArrowDropDown,
                    contentDescription = "Drop-Down Arrow"
                )
            }
        }
    }
}

```

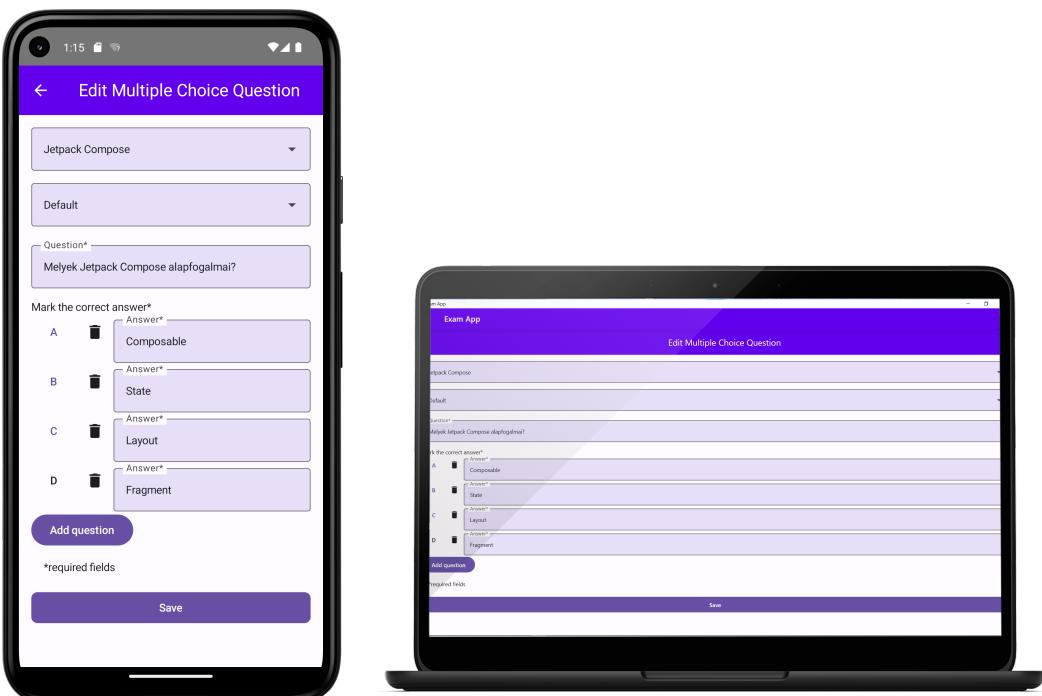
#### 5.4. kódrészlet. Kinyitható kérdés megvalósítása.

A fenti kódrészletben sok érdekes megoldás látható. Az elején felveszünk több Statet, de ezek közül a második a legizgalomsabb. Ez a State lesz az egyik fontos eleme az animációnak. Felhasználjuk benne az első Statet aminek az állapota a kérdésre kattintáskot változik, és ennek az értéktől függően (nyitott vagy csukott) változtatjuk az animációhoz tartozó értéket. A Card Composable elemen használjuk az animációt, ezt az animateContentSize segítségével tehetjük meg. Vannak más fajta animációk is, mindegyik testreszabható és sajátok is létrehozhatóak. Ez esetben egy tween animációt használunk ami 0.3 másodperc alatt az összecsukott állapotról egy nyitott állapotra áll át. Attól változik meg az elem és játszódik le az animáció, hogy rákattintunk a kérdésre. Mivel a Cardon az animateContentSizeet használtuk ezért amikor az expandedState megváltozik akkor a CollapsedQuestion helyett a recomposition után a TrueFalseQuestionDetails kell megjeleníteni ez méretváltozással jár, ezt követi le az animáció ezáltal nem hirtelen ugrik egyet a képrenyő, hanem lekövethető módon változik meg. A rotationState pedig az ArrowDropDown elem animálásban játszik szerepet, így az ikont el tudjuk forgatni 180 fokkal, ehhez a Modifierhez tartozó rotate(Int) extension functiont tudjuk használni.



**5.4. ábra.** Egy érdekesebb részletes képernyő, a vizsgák összeállításához tartozik.

#### 5.1.2.4. Szerekesztő nézet



**5.5. ábra.** Egy érdekesebb szerkesztő képernyő, a feleltválasztós kérdések létrehozásához.

A szerkesztő nézetben lehet módosítani minden adott (kivéve a feladatsorhoz tartozó kérdéseket, amit az előző alfejezetben mutattam be). Itt sok féle elem megtalálható;

szöveges bevuteli mezők, dropdown menü, a pontok esetében pedig numerikus beviteli mezők.

A feleletválasztós kérdésekknél (5.5. ábra) van egy sepcialis elem, ami a a kérdések megadásához készült. Tartalmaz egy TextButont, ami a válasz sorszáma, egy ImageButton a válsz törléséhez és egy szöveges beviteli mező a válasz megadásához. Tetszőleges opció megadható, itt viszont egy egyszerű Column mellet döntöttem és egy for ciklus segítségével rajzolom ki az elemeket.

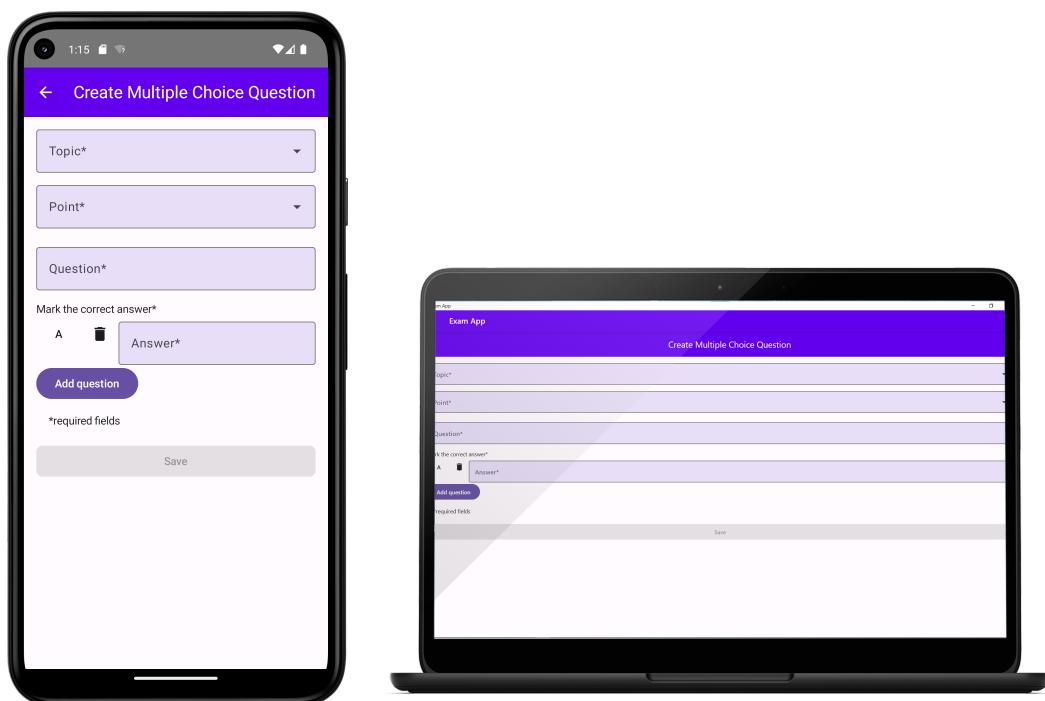
```
@Composable
fun ExamInputForm(
    enabled: Boolean = true,
    entryViewModel: ExamEntryViewModel = viewModel { ExamEntryViewModel() },
    topicListViewModel: TopicListViewModel = viewModel { TopicListViewModel() },
) {
    val coroutineScope = rememberCoroutineScope()
    Column(...) {
        OutlinedTextField(
            value = examDetails.name,
            onValueChange = { onValueChange(examDetails.copy(name = it)) },
            enabled = enabled,
            ...
        )
        DropDownList(
            name = stringResource(Res.string.exam),
            items = topicListViewModel.topicListUiState.topicList.map{it.topic}.filterNot{ it == examDetails.topicName },
            onChoose = {topic ->
                coroutineScope.launch{
                    onValueChange(examDetails.copy(topicId = entryViewModel.getTopicIdByTopic(topic)))
                }
            },
            default = examDetails.topicName,
            modifier = Modifier.fillMaxWidth(),
        )
        if (enabled) {
            Text(
                text = stringResource(Res.string.required_fields),
                modifier = Modifier.padding(start = 16.dp)
            )
        }
    }
}
```

**5.5. kódrészlet.** Szerkesztő nézet kódja.

A fenti kódrészletben megfigyelhetjük a ViewModelek átadását a Composable függvények. Esetenként többre is szükseg lehet, itt például az examEntry és a topicList ViewModelekre is szükség van, mivel a dropdown listában a témakört ki kell választani a feladatsorhoz. A dropdown menü kódjában kihasználtam a kotlin funkcionális programozást segítő kódját, ezzel kigyűjtöttem a témakörök listájából azokat amik nem egyeznek meg a jelenleg kiválasztottal. A hosszabb ideig tartó műveleteket egy coroutineScopeban valósítáttam meg egy másik lambda függvényben, így a UI nem blokkolódik le közben. A mentés gomb le van tiltva, ha hiányzik egy kötelező mező vagy ha ez egyedi mezőkben ütközés van a felhasználó kap egy értesítést és a gomb is letiltódik amíg nem változtat az értéken.

### 5.1.2.5. Új elem létrehozása

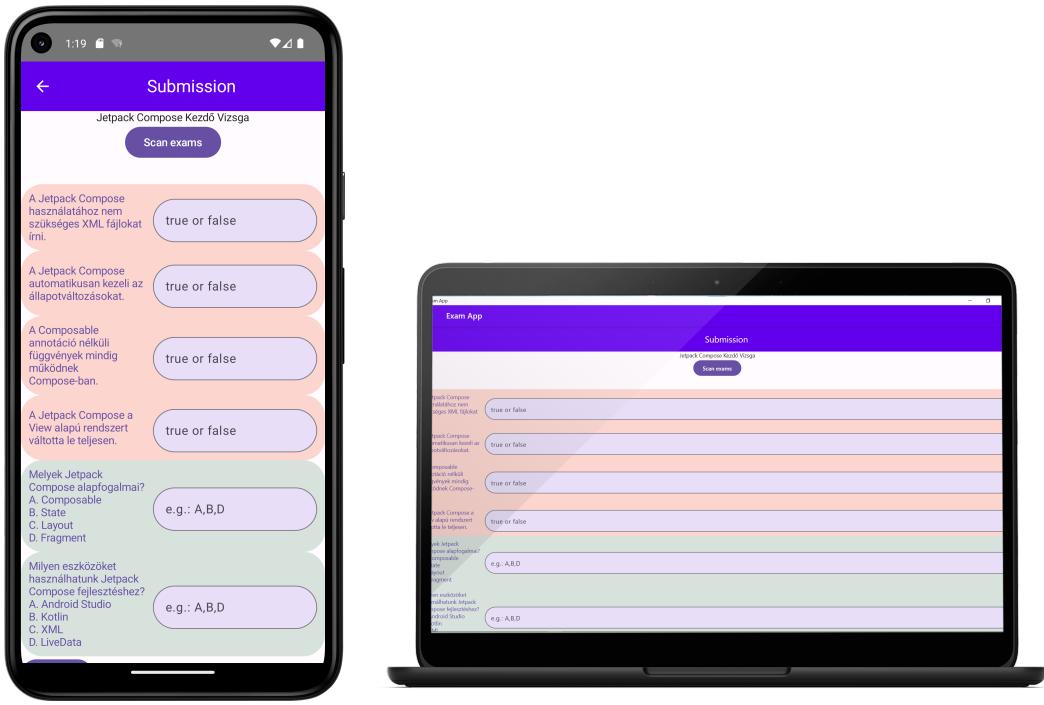
Ez a nézet lényegében megegyezik a szerkesztő nézettel. Egy jó szoftver arról ismerszik meg, hogy a lehető legtöbb helyen újra felhasználja a már meglévő kódot anélkül, hogy annak a kódját lényegesen meg kelljen változtatnia. Ebben az esetben már terevezés során megszületett a dönés. Egész pontosan van egy Body része az új elem felvételének ami vár paramétereket egy UiStateet, amiben az adott elem adatai kerülnek. Az alapértéki úgy vannak megválasztva, hogy az üres hatást keltsen, de ha már léteznek akkor a szerkesztő nézetet kapjuk, az új létrehozása helyett. Ehhez szükség van külön ViewModelre és kevés Composable függvény megírására is, de ez jelentősen kevesebb, mint az egész kód újraírása. Ezzel a kód karbantarhatósága is növekszik.



5.6. ábra. Új feleletválasztós kérdés létrehozása.

### 5.1.2.6. Ellenőrző képernyő

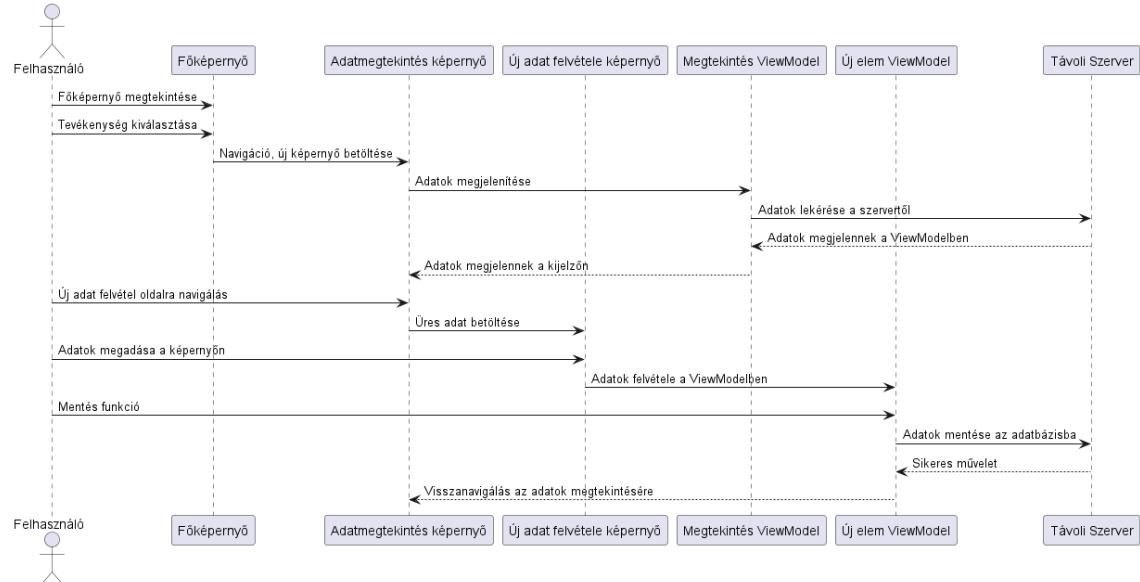
Az ellenőrző képernyőn a kérdések vannak felsorolva, a szokásos színkódolással. Olvasható a teljes szöveg és a válaszok is a feleletválasztós kérdésknél. A válasz formátuma látható kitöltés előtt, így a felhasználó tudja, hogy milyen formátumban van elvárva a válasz. A fenti Scan Exam gomb Android eszközön megnyit egy kamera nézetet ami segítségével pár gyors mozdulattal be tudjuk olvasni a válaszokat így nem kell kézzel begépelni azt. Ha valamit nem vagy rosszul detektált a mesterséges intelligencia akkor természetesen az oldalon ezek javíthatóak. A lap alján található egy gomb ami segítségével ki lehet értékelni a válaszkat, ez a backend oldalon történik meg, majd az eredményt visszaküldi és a felhasználó megtekintheti az elérte pontszámot és a százalékos értéket.



**5.7. ábra.** Új feleletválasztós kérdés létrehozása.

### 5.1.3. Szekevencia diagram egy átlagos interakcióra

Végezetül szeretnénk bemutatni egy szekevencia diagramot amin keresztül mégerthető, hogy a felhasználó, az alkalmazás és a háttérben futó szerver milyen kapcsolatbanállnak egymással.



**5.8. ábra.** Szekvencia diagram az end-to-end kommunikációról

## 5.2. Navigáció

A szoftverben az Androidból jól ismert navigációs könyvtárat használtam. Szerencsére már jól integrálható multiplatform környezetben. A 3.1.1.3. alalszakaszban erről már felületesen bemutattam. Az ebben a fejezetben bemutatott implementációs lépésekkel most részletesebben bemutatom.

Az első lépés kifejezetten egyszerű. Létre kell hozni egy a végpontokat összefogó sealed class-t, érdemes ennek egy route String típusú változót. Ezt leginkább a böngészőben is megjelenő routinghoz hasonlít. A sealed classból leszármaztathunk data objekteket. Ez egy viszonylag egyedi felhasználás. Itt az object egy singleton minta, mivel ebből az osztályból egy keletekezik összesen a program futása során. A data keyword pedig létrehoz hasznos metódusokat az adateléréshez. Tudunk megadni a leszármaztatott osztályokban egyéb paramétereket így például tudunk egyes elemekre kulcsot adni amit ennek felhasználásával tudjuk a konkrét példányt lekérdeni az adatbázisból.

```
sealed class ExamDestination(val route: String) {
    data object LoginScreenDestination : ExamDestination("LoginScreen")

    data object TopicDetailsDestination : ExamDestination("TopicDetails") {
        const val topicIdArg = "0"
        val routeWithArgs = "$route/{$topicIdArg}"
    }
}
```

**5.6. kódrészlet.** Példa a végpontok felvételére.

A másik lépésben a NavHostControllert kell létrehozni. Ezt a rememberNavController() meghívásával tudjuk megtenni egy Composable függvény törzsén belül. Ezt követően szükség lesz NavHost létrehozására ami egy Composablefüggvény. Ennek lehet átadni az előbb létrehozott NavHostControllert, és be kell állítani egy kezeti végpontot ami az első composition során fog megjelenni.

A lenti példákban látható az, hogy milyen módon lehet a NavHoston belül feleveníni és használni a végpontokat. Számos paraméter megadható, de ezek közül csak a route kötelező. Megadahatók egyéb argumentumok, amiket az egyik képernyőről a másiknak szereznénk átadani navArgumentek formájában. Ezzel lehet például az egyedi azaonsírokat átadni a lista nézetről a részletes nézetnek. Ezt követően az utolsó kötelező elem megadása a content, aminek a típusa: @Composable AnimatedContentScope.(NavBackStackEntry). Ebből két dolog következik, ez egy Composable scope, így hívhatunk Composable függvényeket innen és, hogy hozzáférünk a BacstackEntryre tett adtaokat így például az argumentumokat. Szintén ebben a lambda scopeban hozzáférünk a navControllerhez amivel a navigációt megvalósító függvényeket tudjuk átadni a Viewnak.

```
composable(
    route = ExamDestination.TopicDetailsDestination.routeWithArgs,
    arguments = listOf(navArgument(ExamDestination.TopicDetailsDestination.topicIdArg) {
        type = NavType.StringType
    })
) {
    val id = it.arguments?.getString(ExamDestination.TopicDetailsDestination.topicIdArg) ?: "0"
    TopicDetailsScreen(
        navigateToEditTopic = { navController.navigate("${ExamDestination.TopicEditDestination.route}/{$id}") },
        topicId = id,
        navigateBack = { navController.popBackStack() }
    )
}
```

**5.7. kódrészlet.** Példa egy végpont felvételére a navigációs gráfban.

A viewban, azaz a Composable függvényben a 5.7. kódrészletben megírt függvényt tudjuk meghívni.

```
fun TopicDetailsScreenState(
    navigateToEditPoint: (String) -> Unit,
    navigateBack: () -> Unit,
    ...
){
    Scaffold(
        topBar = { TopAppBarContent(stringResource(Res.string.topic_details), navigateBack) },
        floatingActionButton = {
            FloatingActionButton(
                onClick = { navigateToEditPoint(topic.uuid) },
                shape = MaterialTheme.shapes.medium,
                modifier = Modifier.padding(20.dp)
            )
        }
    ) { ... }
}
```

**5.8. kódrészlet.** A navigáció viewban való megjelenése.

### 5.3. ViewModel

A ViewModelmel korábban a 3.1.1.2. alalszakasz és a 4.1. szakasz részekben már foglalkoztam. Ebben a részben egy megvalósításával fogok foglalkozni.

```
class TopicDetailsViewModel(
    var topicId: String,
) : ViewModel() {

    var topicDetailsScreenState: TopicDetailsScreenState by mutableStateOf(
        TopicDetailsScreenState.Loading)
    var uiState by mutableStateOf(TopicDetailsUiState())

    init { getTopic(topicId) }

    fun setId(id: String){ topicId = id; getTopic(id) }

    fun getTopic(topicId: String){
        topicDetailsScreenState = TopicDetailsScreenState.Loading
        viewModelScope.launch {
            topicDetailsScreenState = try{
                val result = ApiService.getTopic(topicId)
                uiState = TopicDetailsUiState(result.toTopicDetails(
                    parentId =
                    if (result.parentTopic == "null") ""
                    else ApiService.getTopic(result.parentTopic).topic
                ))
                TopicDetailsScreenState.Success(result)
            } catch (e: ApiException) {
                TopicDetailsScreenState.Error.errorMessage = e.message?: "Unknown error"
                TopicDetailsScreenState.Error
            }
        }
    }

    suspend fun deleteTopic() {
        try { ApiService.deleteTopic(topicId)}
        catch (e: ApiException) {
            TopicDetailsScreenState.Error.errorMessage = e.message?: "Unknown error"
            topicDetailsScreenState = TopicDetailsScreenState.Error
        }
    }
}
```

**5.9. kódrészlet.** ViewModel implementációja.

A 5.9. kódrészletben keresztül bemutatom a legfontosabb elemeket. Az viewmodel osztály az általános Android ViewModel osztályból származik le. Ez teszi lehetővé többek között a ViewModelScope használatát. A szokásos módon adhatunk paramétereket, ebben az esetben így adjuk át az Id-t amivel a REST API-n keresztül tudjuk elérni a kívánt téma kört.

Érdemes az osztály elején felvenni az állapotokat. Ilyen Statekben tárolom a képernyő állapotát és a képernyőn megjelenő adatot is. Ezeket a megfelelő kezdeti értékkel létrehozva használhatjuk őket. Lehetne MutableStateFlow is immutabilis Stateet használni egy másik megoldásként.

Az init blokk egyszer hívódik meg, az objektum létrehozása során a konstruktor után, ebből lehet több blokk is amik a leírt sorrendben hívódnak meg, későbbi init blokkban használhatjuk a korábbi értékeket. Most itt egy aszinkron művelet eredményeként kapjuk meg az adatot. Ezen kívül egy másik inicializációs elem a setId függvény, erre akkor van szükség amikor egy specifikus elemet akarunk megjeleníteni.

Az init blokkban nem tudunk tényleges aszinkron függvényeket hívni, ezért szükséges a töltőképernyő amíg az adatok megérkeznek. A getTopic függvény kezeli le ezt a részt. Első lépés a töltőképernyő beállítása. Ezt követően kihasználjuk a viewModelScope-t, ahol végrehajthatunk hosszabb műveleteket anélkül, hogy a UI szál lefagyjon addig. Itt megvárjuk amíg a REST API visszaküldi az adatokat, siker esetén a Success képernyőt állítjuk be a stateben, hiba esetén erről adunk tájékoztatást.

Egyéb függvények megadhatók aszinkron módon, mint a törlés például, ezt nem kell megvárunk.

## 5.4. HTTP kliens

Ebben a részben a HTTP kliens működését mutatom be.

```
object ApiService {
    private val httpClient = HttpClient() {
        install(ContentNegotiation) {
            json(Json {
                ignoreUnknownKeys = true
                prettyPrint = true
            })
        }
        defaultRequest { url("http://mlaci.sch.bme.hu:46258") /* Set the base URL */ }
    }

    suspend fun getPoint(id: String): PointDto {
        val response = httpClient.get("/point/$id")
        if (response.status == HttpStatusCode.OK) return response.body()
        throw ApiException(handleHttpException(response.status))
    }

    suspend fun postPoint(point: PointDto): PointDto? {
        val response = httpClient.post("/point") {
            contentType(ContentType.Application.Json)
            setBody(point)
        }
        if (response.status == HttpStatusCode.Created) return response.body()
        throw ApiException(handleHttpException(response.status))
    }
}
```

### 5.10. kódrészlet. HTTP kliens.

A 5.10. kódrészletben egy kis rész látható a komminikáció működéséről. Egy objectben valósítom meg, így csak példány készül belőle. Létrehozok benne egy HttpClientet aminek beállítom a tartalom típusát JSONre, és az alapértelmezett URL is beállításra kerül. Az endpointokban megadott részek ehhez lesznek hozzáfűzve így kevsebb kódot kell írnis kevesebb hibázási lehetőség van.

Ezek alatt aszinkron suspend függvények találhatóak. Mind a GET, POST, PUT és DELETE műveletkre vannak, de a GETen és a POSTon keresztül a lényeges részeket be lehet mutatni. A függvények kaphatnak paramétereket, a GET esetén egy stringet ami a UUID értékét tartalmazza. A visszatérési értéke pedig egy adott típusú dto lesz. A kérést a baseURL + megadott útvonalra továbbítja a Ktor. Ezt a stinget lehet paraméterezni is, például az idval. A válaszban megkapjuk a stászkódot és a válasz törzsében siker esetén a kért objektumot, amit a KotlinX szerilizáció állít elő a JSON adatból amit ténylegesen kapunk. Hiba esetén a felhasználónak adunk erről visszajelzést egy saját kivételest által segítségével, 200 OK válasz eseténpedig átadjuk az elkészült adatot. A listázás és a törlés ehhez hasonlómódon történik.

Az alsó függvény az új adat létrehozását mutatja be, lényegében a módosítás is szinte pont így történik, csak ott a PUT metódost kell hívni és a Created stászkódot várjuk helyes visszatérés esetén. Itt paraméterek az objektumot kapjuk, amit a post üzenet törzsében állítunk be, itt történik meg a JSON szerilizáció, de most se kellezzel foglalkozni külön elég, ha az adat típusát megadjuk. Innentől minden ugyan úgy történik, mint a többi esetben.

## 5.5. PDF

A feladatsor PDFbe exportálása más módon zajlik a két platformon. Alapvetően hasznlú-anműködnek. Be kell állítani a megfelelő betűtípusokat, oldalméreteket, szövegméretet és hasonló egyéb változókat.

Ezt követően float értékek segítségével lehet pozícionálni a kiírandó szövegeket. Táblázatokat nem lehet rajzolni, de egyéb geometriai elemeket, így vonalat igen, ezekből el-készíthető egy táblázat is. Csak alacsony szintű támogatása van az APIknak, így ugyan minden elkészíthető, de nagyon időigényes. Ahhoz, hogy egy jól kinéző dokumentum képzüljön nagyon sok időt kell beletenni a megvalósításba. PDF szerkesztésére és létrehozására így tehát nem ezek a legtökéletesebb megoldások.

## 5.6. Szöveg felismerés

A Szövegfelismeréshez a Google ML-Kit megoldását választottam. Sok előre elkészített egyszerű AI modellt tartalmaz amit könynen lehet a kódban hasznosítani. A használatát a 5.11. kódrészlet mutatja be. A teljes kód látható egyben, mivel tömören és hatékonyan megoldható a probléma. Ez a funkció Android eszözökön támogatott, asztali eszözökön nagyon furcsa lenne a használata.

Az osztály leszármazik a ImageAnalysis.Analyzer, így lehet képeket feldolgozó kódot létrehozni. Be kell állítani az alapértékeket amikkel majd tud dolgozni az AI. Ezt követően meg kell írni az egyetlen függvényt amit overrideolni kell, itt tölténik a kép feldolgozása, paramétere egy ImageProxy. A szükséges hibakezelés után egy háttérszalon végrejhjódik a szöveg megkeresése és felismerése a képen. A szöveget ezt követően átadja a kód egy másik függvények, amit paraméterben adtunk meg, így tudjuk kinyerni az értékes adato-

kat. Ez a kód amíg az adott nézeten vagyunk végtelen ideig fut, ezért szükség van egy kis késleltetésre, ezzel elkerülve a felesleges hardver használatot.

```
actual class TextRecognitionAnalyzer actual constructor(
    private val onDetectedTextUpdated: (String) -> Unit
) : ImageAnalysis.Analyzer {

    actual companion object {
        const val THROTTLE_TIMEOUT_MS = 1_000L
    }

    private val scope: CoroutineScope = CoroutineScope(Dispatchers.IO + SupervisorJob())
    private val textRecognizer: TextRecognizer =
        TextRecognition.getClient(TextRecognizerOptions.DEFAULT_OPTIONS)

    @OptIn(ExperimentalGetImage::class)
    override fun analyze(imageProxy: ImageProxy) {
        scope.launch {
            val mediaImage: Image = imageProxy.image ?: run { imageProxy.close(); return@launch }
            val inputImage: InputImage =
                InputImage.fromMediaImage(mediaImage, imageProxy.imageInfo.rotationDegrees)

            suspendCoroutine { continuation ->
                textRecognizer.process(inputImage)
                    .addOnSuccessListener { visionText: Text ->
                        val detectedText: String = visionText.text
                        if (detectedText.isNotBlank()) {
                            onDetectedTextUpdated(detectedText)
                        }
                    }
                    .addOnCompleteListener {
                        continuation.resume(Unit)
                    }
            }
            delay(THROTTLE_TIMEOUT_MS)
        }.invokeOnCompletion { exception ->
            exception?.printStackTrace()
            imageProxy.close()
        }
    }
}
```

### 5.11. kódrészlet. Szövegfelismerés.

## **6. fejezet**

# **Továbbfejlesztési lehetőségek**

**7. fejezet**

## **Összefoglalás**

# Köszönetnyilvánítás

Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.

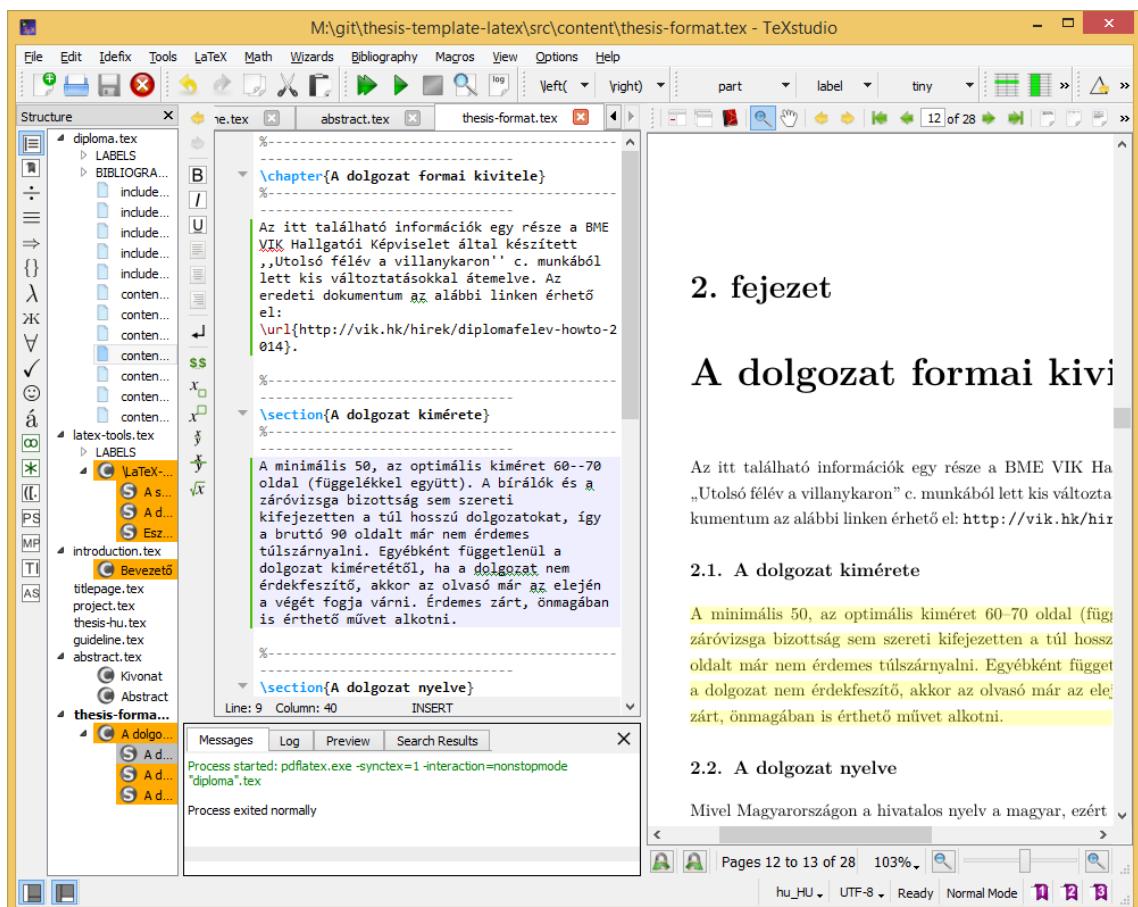
# Irodalomjegyzék

- [1] Apache: Apache pdfbox® - a java pdf library. *Apache PDFBox®*, 2024. 07. <https://pdfbox.apache.org/>.
- [2] Droid Chef: Flutter vs jetpack compose: The battle of the century. *Ishan Khanna*, 2022. 11. <https://blog.droidchef.dev/flutter-vs-jetpack-compose-the-battle-of-the-century/>.
- [3] Husayn Fakher: Compose state vs stateflow: State management in jetpack compose. *Medium*, 2024. 04. <https://medium.com/@husayn.fakher/compose-state-vs-stateflow-state-management-in-jetpack-compose-c99740732023>.
- [4] JetBrains: Coroutines guide. *Official libraries*, 2022. 02. <https://kotlinlang.org/docs/coroutines-guide.html>.
- [5] JetBrains: The basics of kotlin multiplatform project structure. *Multiplatform Development*, 2024. 09. <https://kotlinlang.org/docs/multiplatform-discover-project.html#compilation-to-a-specific-target>.
- [6] JetBrains: Common viewmodel. *Kotlin Multiplatform Development*, 2024. 10. <https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-viewmodel.html>.
- [7] JetBrains: Creating a cross-platform mobile application. *Ktor Documentation*, 2024. 04. <https://ktor.io/docs/client-create-multiplatform-application.html>.
- [8] JetBrains: Integrate a database with kotlin, ktor, and exposed. *Ktor Documentation*, 2024. 09. <https://ktor.io/docs/server-integrate-database.html>.
- [9] JetBrains: Navigation and routing. *Kotlin Multiplatform Development*, 2024. 10. <https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-navigation-routing.html>.
- [10] JetBrains: Popular cross-platform app development frameworks. *Kotlin Multiplatform Documentation*, 2024. 09. <https://www.jetbrains.com/help/kotlin-multiplatform-dev/cross-platform-frameworks.html#popular-cross-platform-app-development-frameworks>.
- [11] JetBrains: Serialization. *Official libraries*, 2024. 09. <https://kotlinlang.org/docs/serialization.html#0>.
- [12] JetBrains: Use fleet for multiplatform development — tutorial. *Kotlin Multiplatform Development*, 2024. 10. <https://www.jetbrains.com/help/kotlin-multiplatform-dev/fleet.html#prepare-your-development-environment>.

- [13] Markus Kohler: 7 alternatives to rest apis. *PubNub*, 2024. 01. <https://www.pubnub.com/blog/7-alternatives-to-rest-apis/>.
- [14] Google LLC: Camerax overview. *Android Developers Guide*, 2024. 01. <https://developer.android.com/media/camera/camerax>.
- [15] Google LLC: Jetpack compose basics. *Android codelabs*, 2024. 01. <https://developer.android.com/codelabs/jetpack-compose-basics#0>.
- [16] Google LLC: Recognize text in images with ml kit on android. *ML-Kit Guides*, 2024. 10. <https://developers.google.com/ml-kit/vision/text-recognition/v2/android>.
- [17] Google LLC: Viewmodel overview. *Android Developers Guide*, 2024. 07. <https://developer.android.com/topic/libraries/architecture/viewmodel>.
- [18] Microsoft: .net multi-platform app ui. *.NET MAUI*, 2024. 11. <https://dotnet.microsoft.com/en-us/apps/maui>.
- [19] Elvira Mustafina: Compose multiplatform 1.7.0 released. *JetBrains Blog*, 2024. 10. <https://blog.jetbrains.com/kotlin/2024/10/compose-multiplatform-1-7-0-released/>.
- [20] Lukoh Nam: Jetpack compose permissions using accompanist library, modalbottomsheet. *Medium*, 2023. 05. <https://medium.com/@lukohnam/jetpack-compose-permissions-using-accompanist-library-b1c0fbbe8831>.
- [21] Lazar Nikolov: Getting started with jetpack compose. *Sentry Blog*, 2023. 02. <https://blog.sentry.io/getting-started-with-jetpack-compose/>.
- [22] Ivan Osipov: Kotlin dsl: from theory to practice. *JMIX*, 2017. 10. <https://www.jmix.io/cuba-blog/kotlin-dsl-from-theory-to-practice>.
- [23] Ekaterina Petrova: Kotlin multiplatform goes stable. *Kotlin Blog*, 2023. 11. <https://blog.jetbrains.com/kotlin/2023/11/kotlin-multiplatform-stable/>.
- [24] Docker Team: What is docker? *Dockerdocs*, 2024. 11. <https://docs.docker.com/get-started/docker-overview/>.
- [25] Flutter Team: Flutter. *Flutter*, 2024. 11. <https://flutter.dev/>.
- [26] Kotlin Documentation Team: Update to the new release. *Kotlin Multiplatform Plugin Releases*, 2024. 09. <https://kotlinlang.org/docs/multiplatform-plugin-releases.html#update-to-the-new-release>.
- [27] React Native Team: React native. *React Native*, 2024. 11. <https://reactnative.dev/>.
- [28] Unknown: What is component diagram? *Visual Paradigm*, 2024. 11. <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-component-diagram/>.
- [29] Antoine van der Lee: Mvvm: An architectural coding pattern to structure swiftui views. *SwiftLee*, 2024. 05. <https://www.avanderlee.com/swiftui/mvvm-architectural-coding-pattern-to-structure-views/>.

# Függelék

## F.1. A TeXstudio felülete



F.1.1. ábra. A TeXstudio L<sup>A</sup>T<sub>E</sub>X-szerkesztő.

## F.2. Válasz az „Élet, a világmindenség, meg minden” kérdésére

A Pitagorasz-tételből levezetve

$$c^2 = a^2 + b^2 = 42. \quad (\text{F.2.1})$$

A Faraday-indukciós törvényből levezetve

$$\operatorname{rot} E = -\frac{dB}{dt} \quad \rightarrow \quad U_i = \oint_{\mathbf{L}} \mathbf{E} d\mathbf{l} = -\frac{d}{dt} \int_A \mathbf{B} d\mathbf{a} = 42. \quad (\text{F.2.2})$$