

SZAKDOLGOZAT



MISKOLCI EGYETEM

Androidos alkalmazásfejlesztés bemutatása a horgásznapló rendszer digitalizációjával

Készítette:

Csonka Patrik

Programtervező informatikus

Témavezető:

Dr. Agárdi Anita

MISKOLC, 2024

MISKOLCI EGYETEM

Gépészmérnöki és Informatikai Kar

Alkalmazott Matematikai Intézeti Tanszék

Szám:

SZAKDOLGOZAT FELADAT

Csonka Patrik (CMU4ZN) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: Alkalmazásfejlesztés Androidos környezetben

A szakdolgozat címe: Androidos alkalmazásfejlesztés bemutatása a horgásznapló rendszer digitalizációjával

A feladat részletezése:

A Magyarországon használatos papír alapú horgásznapló rendszert fogom megvalósítani Android Studio segítségével egy alkalmazás formájában. A szakdolgozatomban bemutatom a fejlesztés folyamatát ebben a fejlesztői környezetben, illetve részletezem az alkalmazás felépítését és a használatos technikákat, API-kat.

Témavezető: Dr. Agárdi Anita

A feladat kiadásának ideje: 2024 Március 4.

.....
szakfelelős

EREDETISÉGI NYILATKOZAT

Alulírott **Csonka Patrik**; Neptun-kód: CMU4ZN a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírással igazolom, hogy *Androidos alkalmazásfejlesztés bemutatása a horgásznapló rendszer digitalizációjával* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szó szerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, év hó nap

.....
Hallgató

- | | | |
|----|-----------------------------------|--|
| 1. | A szakdolgozat feladat módosítása | szükséges (módosítás külön lapon)
nem szükséges |
|----|-----------------------------------|--|

.....

témavezető(k)

2. A feladat kidolgozását ellenőriztem:

konzulens (dátum, aláírás):

.....

.....

.....

3. A szakdolgozat beadható:

.....

témavezető(k)

4. A szakdolgozat szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....

..... egyéb mellékletet (részletezve)

.....

tartalmaz.

.....

témavezető(k)

- | | | |
|----|--------------------------|----------------|
| 5. | A szakdolgozat bírálatra | bocsátható |
| | | nem bocsátható |

A bíráló neve:

.....

szakfelelős

- ## 6. A szakdolgozat osztályzata

a témavezető javaslata:

a bíráló javaslata:

a szakdolgozat végleges eredménye:

Miskolc,

.....
a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. Bevezetés	1
2. A telefonos alkalmazás fejlesztés	3
2.1. Különböző szoftverek bemutatása	3
2.2. Az Android Studio részletezése	5
2.3. Androidos alkalmazás fejlesztés	6
3. Tervezés	7
3.1. Az alkalmazás fejlesztés bemutatása	7
3.2. Model-View-ViewModel	7
3.2.1. Model	8
3.2.2. View	8
3.2.3. ViewModel	8
3.3. A program felépítése	8
3.3.1. Bejelentkezés	9
3.3.2. A fogási napló	10
3.3.3. Fiók	10
4. Megvalósítás	11
4.1. Előkészület	11
4.2. Android Studio projektek felépítése	13
4.2.1. Könyvtárak	13
4.2.2. Gradle	13
4.2.3. Manifest	13
4.3. Az alkalmazás létrehozása	14
4.3.1. MainActivity.kt	14
4.3.2. MainApp.kt	15
4.4. SplashScreen	17
4.5. LoginScreen	17
4.5.1. AccountService	17
4.5.2. AccountServiceImpl	18
4.5.3. LoginViewModel	19
4.5.4. LoginView	20
4.6. OnlineCatchLog	22
4.6.1. Model	22
4.6.2. CatchLogViewModel	25
4.6.3. OnlineCatchLogView	27
4.6.4. AddLogDialog	29
4.7. Account	30

4.7.1. AccountViewModel	30
4.7.2. AccountView	31
5. Tesztelés	33
6. Összefoglalás	34
Források	35

1. fejezet

Bevezetés

A telefonok térnyerése az utóbbi évtizedben tagadhatatlan, életünk számos részét érintette, illetve befolyásolta. Ezek alól nem képeznek kivételt a szabadidős elfoglaltságok, ezáltal a horgászat sem. Ma már nehéz találni olyan horgászati tevékenységet végző embert, akinek a zsebében ne lenne ott egy olyan telefon, amely minimum 50 milliószor több számítást végző kapacitással rendelkezik, mint az 1962-ben létrehozott Apollo 11-es űrrakéta fedélzeti számítógépe [13]. Több horgásztársamnak a fejében is megfordult már, hogy akkor miért kell még mindig a fogásait papírra leírni, majd összegezni az év végén, és leadni azt a helyi horgász szövetségnek.

A rendszerváltás után rendszeresített fogási napló egy fontos problémát hivatott megoldani, ugyanis az egyre népszerűbb sport elfoglaltság veszélyeztette a hazai vizek halállományát, amiről ekkor még oly’ keveset tudtunk. Ez a rendszer volt hivatott arra a feladatra, hogy naplózzák, számon tartsák a kifogott halakat és összefüggéseket vonjanak le a hazai vizek élőlény állományáról.



1.1. ábra: 2023. évi állami fogási naplók.

Ezen a ponton találkozott a két kedvenc területem, a horgászat és a programozás. Az egyetemi éveim alatt szerteágazó tudásra tehettem szert, megismerhettem sokféle programozási nyelvet és technikát. Külön kiemelném az adatbázis kezelést, és az objektum orientált programozási elvet, amelyek nagy mértékben hozzásegítettek a szakdolgozati témám kiválasztásához, és megvalósításához. Sokszor szó esett arról, hogy a programozás rengetegszer hétköznapi problémákat hivatott megoldani, ezáltal könnyebbé tenni a felhasználó életét.

Ezen a vonalon haladva határoztam el, hogy a szakdolgozati témám az állami fogási napló rendszer digitalizálása lesz. Ezzel az applikációval szeretném felhívni a figyelmet arra, hogy időleges az előbbiekben említett rendszer átdolgozása, és a mai igényeket kielégítő okos telefonos integráció létrehozása.

A továbbiakban be fogom mutatni, hogy az én elképzelésem alapján hogyan valósítható meg egy Androidos alkalmazás Android Studio fejlesztői környezetben. Célja az alkalmazásnak, hogy felhasználó barát módon segítse horgásztársaimat a hétvégi kikapcsolódás során gyűjtött adatok - például a fogott halak súlya, a fogás pontos helye és időpontja - rögzítésére, és rendszerezésére. Az applikáció ezen túl segít a régebben rögzített fogások visszakeresésére, és áttekintésére. Kiemelt szempont volt az, hogy kezdő telefon felhasználók, ezáltal az idősebb, okostelefont kevésbé használni tudók is könnyen és gyorsan használatba vehessék az alkalmazást. Törekedtem arra, hogy a program kezelői felülete a lehető legegyszerűbb, és átlátható legyen.



1.2. ábra: Horgászati tevékenységet végző ember.

A dolgozat az alábbi felépítést követi: az első fejezet bevezeti az olvasót a jelenlegi piaci helyzetbe, és rámutat a hiányosságokra, majd egy lehetséges megoldást nyújt. A második fejezet a fejlesztéshez szükséges technológiai alapokat, különösen az Android Studio és a hozzá kapcsolódó eszközök használatát mutatja be. A harmadik fejezet az alkalmazás architektúráját és a funkciók részletes leírását tartalmazza, míg a negyedik fejezet az alkalmazás megvalósításának módszereit tárgyalja. Az ötödik fejezet a tesztelési fázist mutatja be, az utolsó fejezet pedig összefoglalja a projekt eredményeit, és javaslatokat fogalmaz meg a jövőbeni fejlesztésekre.

2. fejezet

A telefonos alkalmazás fejlesztés

A bevezetőben említett probléma nem újkeletű, a MOHOSZ tisztában van a horgász fogási napló digitalizálásával kapcsolatos felmerülő igényekkel, több próbálkozás is volt már, mindent átfogó megoldás azonban eddig még nem valósult meg, csak tervek vannak terítéken[18].

A telefonos alkalmazások fejlesztésére több különböző módszer is létezik, ebben a fejezetben szeretném ezt részletesebben bemutatni.

2.1. Különböző szoftverek bemutatása

A telefonos applikációk fejlesztésére többféle platformon elérhető szoftverek léteznek. Továbbiakban ismertetni szeretném a kifejezetten számítógépes platformra készített népszerűbb programokat, amelyek a telefonos alkalmazás fejlesztést hivatottak megkönnyíteni.

Az egyik legnépszerűbb választás a Flutter, ami a felhasználó számára lehetővé teszi, hogy egyszerre fejlesszenek a két legnagyobb telefonos platformra, azaz Android-ra, és iOS-re (továbbiakban Cross-platform). Ez egy DART keretrendszer alapú "widget" fejlesztést kínál, aminek az előnye a gyors User Interface készítés, és széleskörű testreszabhatóság[6]. Készítője a Google.



2.1. ábra: Flutter logó

A másik népszerű választás a React Native[19], amely szintűgy egy Cross-platform alkalmazás készítő szoftver, amelyet a Facebook fejlesztett ki. A program JavaScript, illetve TypeScript programnyelvet használ, nagy hangsúlyt fektet a gyorsaságra.



2.2. ábra: React Native logó

Említésre méltó a Unity[21], amely kifejezetten játék fejlesztésre készült szoftver. Többek között képes számítógépes, konzolos és telefonos alkalmazás fejlesztésre is. Fejlesztési nyelve a C#.



2.3. ábra: Unity logó

A következő alkalmazás az Xcode[3], amely az Apple hivatalos fejlesztői környezete. Ezzel a szoftverrel csak az Apple ökoszisztémán létező operációs rendszerekre tudunk fejleszteni, azaz iOS, iPadOS, macOS, watchOS és tvOS. Fejlesztési nyelve a Swift, és Objective C.



2.4. ábra: Xcode logó

Utoljára pedig az általam választott szoftvert mutatnám be röviden, az Android Studio-t[10]. Ez a Google által fejlesztett hivatalos fejlesztő környezet, amellyel Androidos alkalmazások készítését teszik elérhetővé. Ez egy teljeskörű eszközkészlet, rendelkezik Android SDK-val, és beépített emulátorral, ami megkönnyíti a gyors tesztelést. Fejlesztési nyelve elsősorban a Kotlin, és a Java.



2.5. ábra: Android Studio logó

Az előbbieken felsorolt fejlesztői környezeteknek nagyon sok előnye, és hátránya van. Az általam megadott szempontok a következők voltak:

- Androidon futtatható legyen az alkalmazás, hozzám közelebb áll a platform szemlélete a nyílt forráskódú applikációk terén.
- A fejlesztő környezet által használt nyelv sem elhanyagolható. Szerettem volna, ha egy magas szintű programozási nyelv lenne a használt az adott környezetben, tekintve hogy azokban szereztem a legtöbb tapasztalatom.
- Fontos volt a beépített emulátor, hogy az alkalmazásomat minél gyorsabban tudjam tesztelni.
- Nem utolsó sorban szerettem volna, ha az általam választott API-kat minél gördülékenyebben tudom integrálni.

Ezen szempontok alapján haladva úgy döntöttem, hogy az Android Studio lesz számomra a legmegfelelőbb választás.

2.2. Az Android Studio részletezése

Az Android Studio a Google által készített hivatalos fejlesztőkörnyezet, amellyel natív Android alkalmazásokat tudunk készíteni. A környezet a JetBrains IntelliJ IDEA[14] alapjaira épül, ebből lett tovább fejlesztve az Android platformra történő alkalmazások fejlesztésére. Az Androidos platformon ez a legjellemzőbben használt IDE.



2.6. ábra: Android Developers logó

Az Android Studio főbb funkciói, és jellemzői:

- **Emulator:** A beépített emulátorral[9] a fejlesztés során gyorsan ki lehet próbálni az alkalmazáson eszközölt változtatásokat. Ezenkívül az emulátorral beállíthatjuk a tesztelni kívánt Android verziót, illetve készüléket, ezáltal tesztelni tudjuk alkalmazásunkat különböző verziószámú operációs rendszereken.
- **Build System (Gradle):** Az Android Studio Gradle-t[12] használ építési rendszerként, amely megkönnyíti az alkalmazások automatizált építését és kezelheti a különböző verziókat, függőségeket. A Gradle segítségével lehetőségünk van különböző buildek létrehozására, ami kulcsfontosságú volt a fejlesztés egyes fázisaiban.
- **Integrált hibakeresés és teljesítményelemzés:** A fejlesztői környezet lehetővé teszi, hogy könnyen diagnosztizáljuk és kijavítsuk az alkalmazásunk problémáit. Különböző eszközökkel, például CPU és memóriafigyelőkkel segít abban, hogy optimalizáljuk az alkalmazás teljesítményét.

- **Firestore és egyéb felhőszolgáltatások támogatása:** Az Android Studio integrációt biztosít a Google Firestore platformmal[11], ami megkönnyíti az adatbázisok, hitelesítés, felhasználói elemzések, értesítések és más felhőszolgáltatások beépítését az alkalmazásba. Ez volt a legfontosabb szempont számomra, mivel az alkalmazásom több ponton is támaszkodik ezekre az API-kra.
- **Aktív közösség és támogatás:** A Google rendszeresen frissíti az Android Stúdiót, és nagy fejlesztői közösség[7] is támogatja, ahol gyorsan választ kaphatunk a kérdéseinkre.

2.3. Androidos alkalmazás fejlesztés

Az előbbiekben részleteztem pár szempontot a fejlesztői környezettel kapcsolatban, azonban nem szeretném kihagyni azt sem, hogy milyen előnyei vannak az erre a platformra való fejlesztésnek közvetlenül. A specifikus platform kiválasztás előnyei:

- **Felhasználók aránya:** Magyarországon az Android operációs rendszerrel rendelkező eszközök piaci részesedése eléri a 80% több független forrás szerint is.[20][15]
- **Nyílt forráskód:** Az Android egy nyitottabb ökoszisztéma[8], nagyobb szabadságot kap a fejlesztő az alkalmazás funkcionalitásának, és megjelenítésének létrehozásában.
- **Költségek:** Az alkalmazás fejlesztése, feltéve hogy mi csinálunk mindent, nem kerül pénzbe. Ezalól nem kivétel a fejlesztői környezet használata, alkalmazásunk elérhetővé tétele, és karbantartása sem.
- **Pulikálás:** Az elkészült Androidos alkalmazást sokkal könnyebben tudjuk publikálni különböző áruházakban[4], nem vagyunk rákényszerítve a telefon gyári alkalmazás áruházára.

Az Android platform legfőbb hátránya:

- **Fragmentáció, kompatibilitási problémák:** Legfőbb hátránya a platformnak a megszámlálhatatlan mennyiségű hardver, és képernyőméret kombinációk. Ezáltal alkalmazásunk fejlesztésekor nagy valószínűséggel gyártunk olyan hibákat, amelyek láthatatlanok voltak számunkra. Ezentúl különböző Android verziók eltérőek lehetnek mind funkciókban, és használható API-kban. Ezáltal a fejlesztés elhúzódhat, mivel a lehető legtöbb készüléket kell lefednünk az alkalmazásunkal, ami időigényes.

3. fejezet

Tervezés

3.1. Az alkalmazás fejlesztés bemutatása

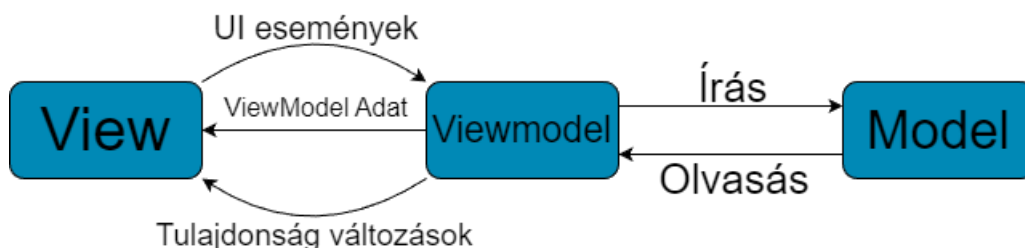
Az Androidos alkalmazás, vagyis az e-Horgásnapló egy Android operációs rendszeren futó, adat rögzítésre használatos alkalmazás. Célom az egyszerű, felhasználóbarát kezelőfelület, ahol a lehető legegyszerűbben rögzíteni tudja a horgász a kifogott hal adatait, és a fogás helyszínét. Az alkalmazást Android Studio segítségével fogom elkészíteni Kotlin segítségével a könnyebb átláthatóság, és kevesebb boilerplate[16] kód miatt. Az alkalmazásom a korábban használt Model-View-controller fejlesztési elv helyett az új Model-View-ViewModel architektúrát használja, ezáltal az alkalmazás jövő állóbb.

3.2. Model-View-ViewModel

Ahogy az előző szekcióban említettem az alkalmazásom a Model-View-ViewModel architektúrát alkalmazza. Ennek a mintának az a célja, hogy elkülönítse az alkalmazás logikai rétegeit, így egyszerűbb lesz a kód karbantarthatósága, és újra használása. Ebben különösen segítségünkre lesz a nemrég bevezetett Jetpack Compose, ami a régi XML elrendezést hivatott leváltani.

A MVVM-nek[17] három fő része van:

- Model: Az adatokat, és a logikát tartalmazó réteg.
- View: A felhasználói felület.
- ViewModel: Köztes réteg, ez köti össze a Model-t, és a View-t. Feladata az adatok feldolgozása, és kezelése.



3.1. ábra: MVVM architektúra diagram

3.2.1. Model

A Model az alkalmazás adatait és üzleti logikáját tartalmazza. Ide tartoznak az olyan elemek, mint az adatbázisok, a hálózati API-k, vagy a fájlkezelés. A Model réteg az adatforrások elérését és az adatok frissítését biztosítja, de nem tartalmaz semmilyen logikát a felhasználói felülethez.

3.2.2. View

A View az a réteg, amely a felhasználói felületet kezeli. Ez felelős az adatok megjelenítéséért és a felhasználói bevitel kezeléséért. A View réteg nem tartalmaz logikát az adatok feldolgozására; csak megjeleníti az adatokat, amelyeket a ViewModel szolgáltat számára.

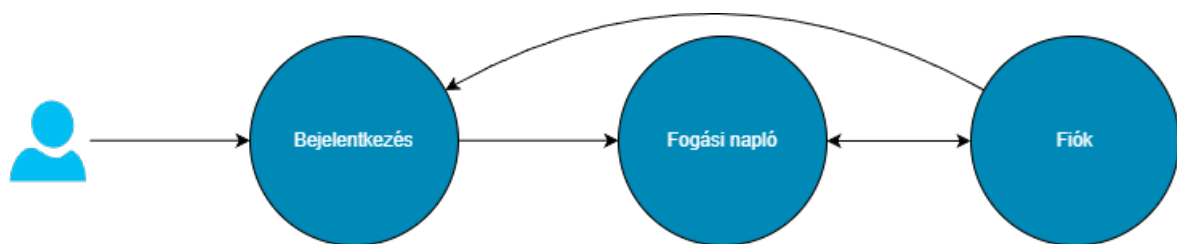
3.2.3. ViewModel

A ViewModel az MVVM minta központi eleme, amely kommunikál a Model és a View között. A ViewModel adatokat kér a Modeltől, feldolgozza azokat, és úgy adja át a View-nak, hogy azok egyből megjeleníthetők legyenek. A ViewModel általában LiveData objektumokat használ, amelyek lehetővé teszik, hogy az adatok automatikusan frissüljenek a View-ban, ha változás történik bennük.

3.3. A program felépítése

A MVVM architektúra kulcsfontosságú szerepet fog betölteni az alkalmazásunkban. Mivel ezen szemlélet alapján kezdem el az alkalmazást, fontos hogy a **mindsk 21** (minimum Operációs rendszer követelmény) legyen beállítva. Ez jelen esetben az Android 5-ös verziót jelenti.

A programban egyszerűsége törekszek, hogy minél szélesebb felhasználó közösség igénybe tudja venni a programot. Fontos, hogy intuitív, és letisztult legyen a kezelői felület, a lehető legkevesebb hibalehetőség merüljön fel a User részéről. Ezért a programom felépítését az alábbiak szerint képzelem el:



3.2. ábra: felhasználói diagram

A felhasználó az applikáció megnyitása után a bejelentkezés kijelzőre lesz irányítva, ahol be tud jelentkezni ideális esetben a MOHOSZ-nál regisztrált adataival. Miután ez megtörtént át lesz irányítva a fogási napló oldalra, ahol látni fogja az összes fogását időrendi sorrendben.

Ezen a ponton tudja feltölteni a legfrissebb fogását a naplóba, ahol a következőket kell megadnia:

- Tó neve
- Tó víztérkódja
- Hal neve
- Hal súlya

Ezután az alkalmazás automatikusan hozzárendeli a **SystemDate** változót, így elkerülve a különböző csalásokat, későbbi fogás hozzáadásokat a rendszerben. Tehát a felhasználó nem adhatja meg a fogásának az időpontját.

A fogás rögzítése után a rendszer hozzárendeli az adatbázishoz az éppen létrejött adatokat, és sorrend szerint legfelülre rakja. Itt megtekinthetjük a frissen hozzá adott fogást a régebben hozzáadottakkal együtt.

Ezen a ponton felmerülhet egy kérdés, hogy mi a helyzet akkor, hogyha úgy döntött egyik horgásztársunk, hogy a vízpartra mindenféle okos eszköz érkezik meg? Itt jön szóba a fiók oldal, ahol ezt a problémát lenne érdemes orvosolni.

A fogási napló tetején egy **TopBar** változóval hozzárendelünk egy gombot, ami átvezet minket a fiók földre, ahol megtekinthetjük az éppen bejelentkezett egyént, és felkínáljuk a lehetőséget a kijelentkezésre. Ezután az alkalmazás vissza vezet minket a bejelentkezés földre, ahol sport társunk bejelentkezhet saját fiókjával, és nyomon követheti, illetve naplózhatja fogásait.

3.3.1. Bejelentkezés

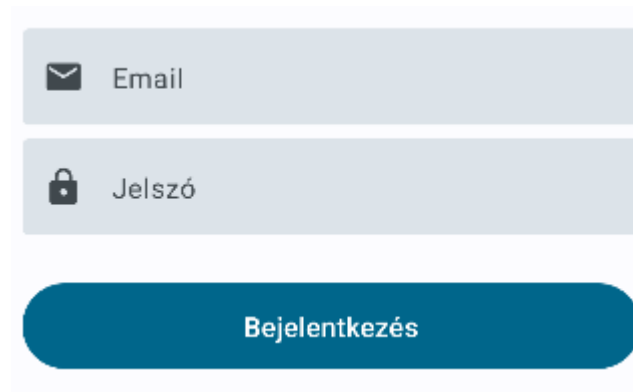
Előszöri belépésnél a felhasználónak megoldást kell kínálni a bejelentkezéshez, amit nem lehet fél vállról venni. Minden alkalmazásban megadott adat szenzitív, alapos körültekintéssel kell ezeket kezelni. Mivel fejlesztő környezetnek az Android Studio-t választottam a különböző Google által fejlesztett API-k könnyen integrálhatóak az alkalmazásba. Ezt kihasználva hosszas tanakodás után a Firebase tűnik a legmegfelelőbb választásnak.

A Firebase felhőszolgáltatást nyújt nekünk különböző előre létrehozott API-kal, amik gördülékenyebbé teszik az alkalmazás fejlesztés ezen szakaszát. Természetesen a bejelentkezésre is nyújt egy külön megoldást, aminek a neve **Firebase Authentication**[5]. Ezzel az API-val dolgozva csak meg kell adnunk a bejelentkezés típusát, és máris neki állhatunk az alkalmazásba való beépítésnek. Itt fontos megemlíteni, hogy ez a szolgáltatás nem teljesen ingyenes, csak egy bizonyos erőforrás kapacitás használatáig, azonban a mi esetünkben az alkalmazás méretét tekintve nem befolyásoló tényező.

A bejelentkezés a Firebase integrációja után így nézne ki:

- E-mail beviteli mező
- Jelszó beviteli mező
- Bejelentkezés gomb

A bejelentkezés gomb megnyomása után ellenőrizni tudjuk, hogy a felhasználó által, vagy általunk által megadott bejelentkezési információk megegyeznek-e az adatbázisban szereplő értékekkel.



3.3. ábra: A bejelentkezés panel egy elképzelt változata.

3.3.2. A fogási napló

Mitűán bejelentkezett a User szeretném, ha egyből a fogási napló oldalon találná magát. Az alkalmazás elemei része továbbra is az egyszerűség, így a könnyű felhasználást szeretném ezzel elősegíteni. Mivel itt adatokat fogunk tárolni szükségünk van valamilyen adatbázisra. Erre a célra az Android studio-ban integrált **Room**[2] adatbázist fogom használni, ami egyszerűen lokálisan tárolja a felhasználó által feltölteni kívánt adatokat. Ezen a ponton szeretném, ha sima felsorolás nézetben megtudja tekinteni a felhasználó az eddigi fogásait az alábbi szempontok szerint:

- Fogás időpontja
- Fogás helye
- A tó víztérkódja
- Hal típusa
- Hal súlya

A képernyő jobb alsó sarkában ésszerű lenne egy **FloatingActionButton**, ami az adatfelvitel **Dialog**-ra vezetne át minket, ahol feltölthetjük az adatainkat.

A fogás időpontja a korábban említett **Systemdate** változó lenne, amit nem tudunk megváltoztatni, mindig a fogás rögzítésének ideje szerepel. A fogás helye természetesen a tó neve lenne, ami egy **String** változó, itt begépelhetjük az éppen kikapcsolódás céljával használatba vett vizet. A tó víztérkódja nem elfelejthető adat, ez **Int** változó lenne. A hal típusa, pontosítva a hal neve egy **String** változó lenne, ahogy megadhatjuk a fogott halunk nevét. Végül a hal súlya értelemszerűen a mérlegelés után megállapított szám avagy **Int**, kilogramm-ban.

3.3.3. Fiók

Az utolsó szekció a fiók oldal, ami azt szolgálná, hogy a felhasználó ki tudjon jelentkezni az alkalmazásból, lehetőséget biztosítva egy másik fiók bejelentkezésére, vagy ugyanazzal az adatokkal való vissza-jelentkezésre.

Itt biztosítani kell a felhasználónak egy kijelentkezés gombot, amivel kijelentkezik a fiókjából, és visszatér a bejelentkezés oldalra.

Továbbá egy másik lehetőséget is érdemes felkínálni a felhasználónak, ahol visszatud térni a fogási napló oldalra.

4. fejezet

Megvalósítás

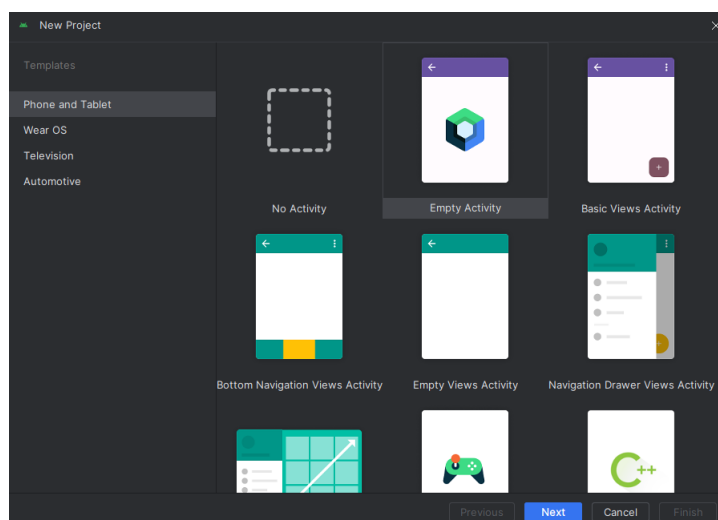
4.1. Előkészület

Mivelhogy Androidos alkalmazásom kezdeti része nagy mértékben hagyatkozik a Firebase által nyújtott szolgáltatásokra, fontos ezeket megvalósítani a program létrehozásának megkezdése előtt.

- Először is szükségünk lesz egy Firebase regisztrációra.
- Sikeres regisztráció után létre kell hoznunk egy Firebase projektet, ami az alkalmazásunk felhő alapú szolgáltatásait hivatott megvalósítani.

Ezen a ponton érdemes létrehozni az Android Studio projectet, ugyanis a Firebase-nak szüksége van a project adataira a sikeren háttérszolgáltatások ellátásához.

Szükségünk lesz az Android Studio programra, ahol a munka nagy része zajlani fog. Ezt a hivatalos weboldáról[10] tudjuk ingyenesen beszerezni. Ezután a programot fel kell telepíteni, majd használatba is vehetjük. Miután beléptünk a programba, az első dolgunk egy új project létrehozása, amit a GUI-n keresztül a bal felső sarokban lévő **File>New project** útvonalon találunk meg. Továbbiakban a projekt létrehozási képernyőre 4.1 leszünk irányítva, ahol kiválaszthatjuk programunk alapvető felépítését, avagy a vázát.




4.1. ábra: Projekt létrehozási ablak.

Itt az IDE több lehetőséget is felkínál, én az **Empty Activity** sémát használom. Ebben a módban az Android Studio elkészíti nekünk az alap felépítést, függőségeit a programnak, ezáltal nekünk nem kell bajlódni a beállításokkal. Ha ez megvan, akkor sikeresen létrehoztuk a projektünket az Android Studio keretein belül, ami biztosítja nekünk a futtathatóságot.

Miután létrehoztuk az Android Studio Projektet vissza kell mennünk a Firebase-en létrehozott projektünk oldalára, és hozzá kell rendelnünk a számítógépünkön létrehozott projekthez. Ezt egész egyszerűen megtehetjük, a hozzáadásnál csak a gyökérkönyvtár nevét (esetünkben `eu.thesis.onlinecatchlog`) kell megadni, ezzel elkezdjük a háttérszolgáltatás kiépítését. Következő lépésként az Authentication fület lenyitva hozzá kell adnunk a projektünkhöz, majd bekapcsolni az E-mail, és jelszó alapú bejelentkezést. Ezután már csak annyi van hátra a Firebase Console-on belül, hogy létrehozunk egy teszt fiókot, amivel az alkalmazás fejlesztése közben tesztelni tudjuk majd a funkciók működését. Ezután érdemes letölteni a Firebase által kínált `google-services.json` nevezetű fájlt, amit el kell helyeznünk a projekt mappába. Ezáltal a későbbiekben hozzáadott funkciók össze tudnak kapcsolódni a felhőben létrehozott adatokkal.

Ennek az elvégzése után térjünk vissza az Android Studio felületére, ugyanis a projektünkben még nem inicializáltuk a Firebase-t. Ezt érdemes még a fejlesztési szakasz elején megtenni, ugyanis a külső könyvtár segítségével fogunk hivatkozni például a bejelentkezési képernyőn bevitt adatokra.

Identifler	Providers	Created ↓	Signed in
test@occl.hu		29 Oct 2024	9 Nov 2024

4.2. ábra: A létrehozott tesztfiók.

A Firebase Console-ban további érdekes funkciókat is láthatunk, például használatra vonatkozó statisztikákat mutat ki, naplózza az alkalmazásunkhoz tartozó használati adatokat, amik nagyon fontosak a publikálás után. Így teljesebb képet kaphatunk a valós alkalmazás használati adatokról.

Mindezek után nyissunk egy terminált a projekt mappájában. Erre lehetőséget kínál közvetlen az Adnroid Studio is, így azt érdemes használni.

Projektünk még nem áll közvetlen összeköttetésben a szerverrel, ezért most inicializálni kell a Firebase-t a projekten belül.

A terminálba írjuk be a következőt:

```
npm install -g firebase-tools
```

Ezzel feltelepítjük a szükséges fájlokat a lokális szerver üzemeltetéséhez, amin belül tesztelni fogjuk az alkalmazásunkat.

Következőnk fontos, hogy az alkalmazás gyökérmappájába legyünk, használjuk az alábbi parancsot:

```
firebase init
```

Ezzel létrehoztunk a Firebase-nek szükséges fájlokat a projektünkben.

```
firebase login
```

Majd be kell jelentkeznünk a weboldalon megadott információkkal, hogy hitelesítsük magunkat.

4.2. Android Studio projektek felépítése

4.2.1. Könyvtárak

Java/Kotlin könyvtár: Itt található minden olyan fájl amely az alkalmazás fő logikáját hivatott megvalósítani. Ezt hívjuk forráskódnak. A három fő csomag sorrendben `com.example.yourappname`. Ez a gyökércsomag, ami tartalmaz minden olyan Kotlin fájlt, amely elengedhetetlen alkalmazásunk futtatásához. Többet között itt található meg a korábban említett **MVVM** fájlok is. Fontos kiemelni, hogy a régebbi architektúrát alkalmazó, legacy kódok másmilyen elrendezésűek.

res könyvtár: Az alkalmazáshoz szükséges forrásokat itt találjuk, amik az olvashatóságot hivatottak megkönnyíteni, illetve a kódismétlést szeretnék minimalizálni. Minden olyan UI elemen megjelenített **String** vagy **kép**, vagy **ikon**-t itt kell deklarálni. Ennek a könyvtárnak az előbbiek alapján három fő egysége van:

- **drawable**, ahol megadhatjuk a az alkalmazásunkban használatos képeket, vektorgrafikus elemeket.
- **values**, ahol deklarálhatjuk a különböző sűrűn előforduló stringeket, színeket, és stílusokat.
- **mipmap**, ahol pedig az alkalmazás ikonjait tudjuk beállítani.

Az itt megtalálható deklarációkra a fejlesztés következő szakaszaiban csak hivatkozni kell. Ezekhez a forrás fájlokhoz mi is hozzáadhatunk változókat, sőt fontos is hogy minimalizáljuk a kód ismétlést.

4.2.2. Gradle

Itt kettő fő komponenst kell megkülönböztetni: a **projekt** szintű, és az **app** szintű gradle fájlokat. Az előbbi fájl tartalmazza a projektünk moduljainak beállításait, itt állíthatjuk be a különböző Gradle és Android Plugin verziókat. Az utóbbi az érdekesebb számunkra, ugyanis itt állíthatjuk be a specifikus függőségeket, megadhatjuk a minimum Android verziót, illetve SDK verziót, nem utolsósorban különböző külső könyvtárakat importálhatunk, amivel szélesíthetjük alkalmazásunk funkcionalitását, és megkönnyíthetjük a fejlesztés folyamatát.

4.2.3. Manifest

Ennek a fájlnek régebben sokkal nagyobb jelentősége volt, ugyanis minden egyes oldalt amit létrehoztunk innen tudtunk inicializálni, alapvető funkciókat beállítani. Az új MVVM architektúra használatával ennek a fájlnek lényegében annyi szerepe maradt számunkra, hogy innen indítjuk el az alkalmazásunkat megnyitáskor.

Programkód 4.1. AndroidManifest.xml fájl egy részlete

```

<activity
    android:name=". MainActivity"
    android:exported="true"
    android:label="@string/app_name"
    android:theme="@style/Theme.ThesisExamples">
    <intent-filter>
    <action android:name="android.intent.action.MAIN"/>

    <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>

```

Ahogy az alábbi programkód részleten láthatjuk, a `MainActivity.kt` nevű fájlt inicializálni kell az `AndroidManifest.xml` nevezetű fájlban, ugyanis az alkalmazás indításakor itt kerül inicializálásra az első képernyő, ami jelen esetünkben a `MainActivity`. Ezt a beállítást az `<intent-filter>` változóval tudjuk megtenni, ahol a `<category android:name="android.intent.category.LAUNCHER"/>` beállítással megmondjuk a programnak, hogy ez a kezdőpont.

4.3. Az alkalmazás létrehozása

A projekt létrehozása után egy üres keretet kaptunk, ami futtatható, azonban sok minden nincs rajta. Miután az előkészületekben megteremtettünk minden előleges feltételt, hogy alkalmazásunkat el kezdjük fejleszteni, nincs más hátra mint a programozás.

4.3.1. MainActivity.kt

Programkód 4.2. MainActivity.kt fájl egy részlete

```

@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        configureFirebaseServices()

        setContent { MainApp() }
    }

    private fun configureFirebaseServices() {
        if (BuildConfig.DEBUG) {
            Firebase.auth.useEmulator(LOCALHOST, AUTH_PORT)
            Firebase.firestore.useEmulator(
                LOCALHOST, FIRESTORE_PORT)
        }
    }
}

```

Ennél a fájlnál részletesebben kisseretnék térni a szerkezetre, illetve a leírt kódra, mivel ez adja egy nagyon kritikus részét az alkalmazásunknak. Miután az `AndroidManifest.xml`-nél beállította az IDE ezt a kezdőpontot, itt kell inicializálnunk az alkalmazást, és a Firebase-t.

- Az `@AndroidEntryPoint` annotáció jelzi a Hilt számára, hogy ez az activity egy belépési pont, azaz injektálni kell a függőségeket, amik később kellenek az alkalmazás számára.
- A `MainActivity` osztály a `ComponentActivity`-t örökli, amely az Android fejlesztői környezet egy alap Activity-je. Szerepe különösen fontos a későbbi `@Compose` annotációkban, ugyanis ez lesz felelős a képernyő frissítésének.
- `onCreate` metódus: ez az Activity életciklusának kezdő pontja. Itt inicializálódik az alkalmazás. Ebben a részben hívjuk meg a `configureFirebaseServices()` függvényt, amely konfigurálja a lokális tesztelést.
- A `setContent` meghívásával beállítjuk a felhasználói felületet, aminek a motorja a `MainApp()`.

4.3.2. MainApp.kt

A `MainApp()` függvény az alkalmazás központi függvénye, fő komponense. Ebben a függvényben állítjuk be a témát amit előre meghatároztunk, illetve a navigációs keretet is itt inicializáljuk. A témát (`OnlineCatchLogTheme`) amit ebben a blokkban kiválasztunk az összes navigációs komponensre alkalmazni fogjuk. Ez magában foglalja a színeket, és betűtípusokat, és az éjszakai módot is.

Programkód 4.3. MainApp.kt fő függvény leírása.

```
@Composable
fun MainApp() {
    OnlineCatchLogTheme{

    }
}
```

A `NavHost` függvénnyel meghívjuk a `notesGraph` funkciót, azonban előtte inicializálnunk kell. Előbb létrehozunk egy `AppState` objektumot, amely tartalmazza az alkalmazás használata során létrejött utolsó állapotot, avagy képernyőt. Ezután a `NavHost`-on belül beállítjuk a kezdőképernyőt, amely az esetben a `SplashScreen`. Ez a képernyő egy betöltési indikátor, ami az alkalmazás első oldala. Ez a képernyő arra jó, hogy a felhasználó gyorsabbnak érezze az alkalmazás működését, ugyanis nem csak egy sima fekete kijelzőt lát, hanem animációt.

Programkód 4.4. Navhost meghívása, beállítása.

```
Surface(color = MaterialTheme.colorScheme.background) {
    val appState = rememberAppState()

    Scaffold { innerPaddingModifier ->
        NavHost(
            navController = appState.navController,
            startDestination = SPLASH_SCREEN,
            modifier = Modifier.padding(
                innerPaddingModifier)
        ) {
            notesGraph(appState)
        }
    }
}
```

A **notesGraph** függvény egy navigációs grafikus struktúráját definiálja. Itt határozzuk meg az elérhető képernyőket, és azok nevét.

Az elérhető képernyők listája:

- **SignInScreen**: Ez a bejelentkező kijelzőnk, a **SplashScreen** után az első, amit lát a User.
- **SplashScreen**: Az alkalmazás elindítása után ez a kijelző hivatott kiküszöbölni a különböző inicializálások miatti lassabb futást. Egy animációt tartalmaz.
- **AccountScreen**: A képernyő, amellyel kijelentkezhetünk az alkalmazásból, és visszatérhetünk a bejelentkezés képernyőre.
- **CatchlogScreen**: A fő képernyője az alkalmazásnak. Itt tudjuk a korábban meghatározott módon feltölteni fogásainkat, és megtekinteni a korábban felvitt adatokat.

Programkód 4.5. Az egyik elérhető képernyő.

```
composable(SIGN_IN_SCREEN) {
    LoginView(openAndPopUp = {
        route, popUp -> appState.navigateAndPopUp(route, popUp) })
}
```

Az **openAndPopUp** függvény a navigáció kezelésére szolgál az alkalmazásban. Ez a kód minden lehetséges **View**-ban szerepel, ugyanis ezzel navigáljuk át a felhasználót a másik kijelzőre, miközben a stack-ből töröljük az előző képernyőt. Ebben az esetben a **route** adja meg annak a képernyőnek a nevét, amelyre navigálni szeretnénk. Fontos, hogy az **appState**-hez kapcsolódik, amelyben a függvényt az **appState.navigateAndPopUp** hívja meg. Ez biztosítja, hogy az **appState** megfelelően kezelje a navigációs műveleteket az alkalmazás különböző képernyői között.

4.4. SplashScreen

Ebben a szekcióban láthatjuk először, hogy gyakorlatilag mi is az az MVVM. A **Splash-ViewModel** adja a logikai függvényeket, a **SplashView** pedig a megjelenítésért felelős. Nézzük meg először a **SplashViewModel**-t.

Programkód 4.6. A SplashViewModel annotációja, és osztály definíciója

```
@HiltViewModel
class SplashViewModel @Inject constructor(
    private val accountService: AccountService
) : MainAppViewModel() {
}
```

- @HiltViewModel: Ez az annotáció jelzi a Dagger Hilt-nek, hogy ez az osztály egy ViewModel. Ezután a Hilt keretrendszer automatikusan gondoskodik a különböző függőségek injektálásáról.
- A SplashViewModel öröklí a MainAppViewModel osztályt, ahol egy alap hibaelhárító függvény van jelen, a launchCatching. Ez minden ViewModel-nek a része.
- Az accountService egy privát függőség, amelyet a Hilt injektál a SplashViewModelbe. Ez a függőség az, amely kezeli az aktuális felhasználói fiók állapotát. Tehát ha a felhasználó be van jelentkezve, akkor nem szabad még egyszer felkínálni neki erre a lehetőséget, hanem a főoldalra kell irányítani. Ellenkező esetben pedig a bejelentkező képernyőt kell neki felajánlani.

Programkód 4.7. onAppStart függvény deklarációja.

```
fun onAppStart(openAndPopUp: (String, String) -> Unit) {
    if (accountService.hasUser()) openAndPopUp(
        CATCHLOG_SCREEN, SPLASH_SCREEN)
    else openAndPopUp(
        SIGN_IN_SCREEN, SPLASH_SCREEN)
}
```

4.5. LoginScreen

Ez a screen teljes mértékben kihasználja a Model-View-ViewModel adta tulajdonságokat. Ennek az oldalnak a bemutatását a Model felől közelíteném meg.

4.5.1. AccountService

Egy interfész, amely a felhasználói fiókkezelési műveleteket definiálja. Eme interfész segítségével a felhasználók ki, és be jelentkezhetnek.

- `currentUser: Flow<User?>`: Egy Flow, amely nem összehangolt módon figyeli az aktuális felhasználói fiókot, és folyamatosan frissíti az adatokat, ha a felhasználó állapota megváltozik. A Flow hasonlóan működik mint egy lista, azonban itt az idő függvényében változik a lekérdezésre adott válasz.
- `currentUserId: String`: A jelenlegi bejelentkezett felhasználó azonosítója.
- `hasUser(): Boolean`: Ellenőrzi, hogy van-e bejelentkezett felhasználó.
- `signIn(email: String, password: String)`: Egy függvény, amely bejelentkezést hajt végre a megadott email és jelszó alapján.
- `signOut()`: Kijelentkezteti a jelenlegi felhasználót

Programkód 4.8. AccountService.kt

```
interface AccountService {
    val currentUser: Flow<User?>
    val currentUserId: String
    fun hasUser(): Boolean
    suspend fun signIn(email: String, password: String)
    suspend fun signUp(email: String, password: String)
    suspend fun signOut()
}
```

4.5.2. AccountServiceImpl

Az AccountServiceImpl az AccountService interfész implementációja. A Firebase hitelesítési szolgáltatásait használja a felhasználói bejelentkeztetés megvalósítására.

A fontosabb funkciók, kiegészítve az előző szekcióban leírtakat:

- `currentUser`: A callbackFlow-t használja, hogy folyamatosan figyelje a Firebase hitelesítési állapotát. Az AuthStateListener figyeli, ha a felhasználó állapota változik (pl. bejelentkezés vagy kijelentkezés), és frissíti a currentUser értékét.

```
override val currentUser: Flow<User?>
get() = callbackFlow {
    val listener =
        FirebaseAuth.AuthStateListener { auth ->
            this.trySend(auth.currentUser?.let {
                User(it.uid) })
        }
    FirebaseAuth.addAuthStateListener(listener)
    awaitClose {
        FirebaseAuth.removeAuthStateListener(listener) }
}
```


- `currentUserId`: Ez kérdezi le a jelenlegi felhasználó azonosítóját.

```
override val currentUserId: String
get() = Firebase.auth.currentUser?.uid?.orEmpty()
```

- `hasUser()`: Megvizsgálja, hogy a `Firebase.auth.currentUser` értéke nem null, ami azt jelenti, hogy van bejelentkezett felhasználó. Ha null, akkor értelemszerűen jelenleg nincsen bejelentkezett user.

```
override fun hasUser(): Boolean {
    return Firebase.auth.currentUser != null
}
```

- `signIn(email, password)`: Aszinkron módon bejelentkezteti a felhasználót az email és password paraméterek alapján.

```
override suspend fun signIn(
    email: String,
    password: String) {
    Firebase.auth.signInWithEmailAndPassword(
        email, password).await()
}
```

- `signOut()`: Kijelentkezteti a felhasználót a Firebase hitelesítési rendszerből.

```
override suspend fun signOut() {
    Firebase.auth.signOut()
}
```

4.5.3. LoginViewModel

Ez előbbieken felsorolt funkciók elemi részét képezik ennek a program blokknak. A `LoginViewModel` lesz az az elem, ami összeköti a Model-eket a View-al, azaz információt szolgáltat a felhasználónak. Kezdjük a függőségekkel:

- `AccountService`: Ez felelős a felhasználói bejelentkeztetésért. A `LoginViewModel` ezt használja a `signIn` meghívásával a bejelentkezési folyamat elindításához.
- `@Inject constructor`: Ennek a módszernek a segítségével állítjuk elő az `AccountService` példányt a `LoginViewModel` számára.

Található továbbá kettő darab `MutableStateFlow`, amellyel az állapotfolyamatok, és az értékek frissítését teszi lehetővé. A Compose automatikusan újra rajzolja a felhasználói felületet, ha ezek a `StateFlow`-k frissülnek.

Ezen felül a legfőbb függvény ebben a blokkban az `onSignInClick`. Itt lesz kulcsfontosságú a korábban említett `openAndPopUp`, ugyanis itt csatolunk vissza a **MainApp**-ban lévő navigálási komponensre. Először a `route`-al megnevezzük a célképernyőt, amelyre át szeretnénk irányítani a User-t sikeres bejelentkezés után. Másdszor pedig a `popUp` változóval eltávolítjuk a navigációs stack-ből a jelenleg használt képernyőt. Ez azért fontos, mert a felhasználó így nem tud a telefonján a visszalépéssel visszatérni a bejelentkező panel-ra, hanem az alkalmazásból fog kilépni.

Programkód 4.9. onSignInClick függvény.

```
fun onSignInClick(openAndPopUp: (String, String) -> Unit) {
    launchCatching {
        accountService.signIn(email.value, password.value)
        openAndPopUp(CATCHLOG_SCREEN, SIGN_IN_SCREEN)
    }
}
```

4.5.4. LoginView

Ezzel elérkeztünk az oldal megjelenítő részéhez. Az itt deklarált függvények, metódusok, változók felelősek azért, hogy a megjelenített tartalom pontosan ott legyen ahol szeretnénk.

Kezdjük itt is a függvényparaméterekkel:

- **modifier: Modifier:** Ez egy alapértelmezett paraméter, amellyel lehetővé tesszük a komponensek vizuális, és elhelyezési tulajdonságainak módosítását. Legtöbbet a méretezésre, és az elhelyezésre használtam.
- **viewModel: LoginViewModel:** Ezzel a származtatással férünk hozzá a LoginViewModel-ben deklarált bejelentkezési logikához és állapotokhoz. A `hiltViewModel()` miatt a Hilt automatikusan injektálja, így megkapjuk az állapotokat és műveleteket.

Következő fontos blokkunk az állapotgyűjtés. Amikor a felhasználó használatba veszi a e-mail és password mezőt, akkor a képernyőt frissíteni kell minden alkalommal. Tehát ezek a **State** objektumok felelősek a komponensünk újra rajzolásáért.

Programkód 4.10. E-mail, és jelszó State objektumok.

```
val email = viewModel.email.collectAsState()
val password = viewModel.password.collectAsState()
```

Számomra az egyik legnagyobb nehézséget a Jetpack Compose használata jelentette, ugyanis ez a fajta UI szemlélet nagy mértékben eltér a korábban használt Activity/Fragment felépítéstől.

Ennek a fajta UI felépítésnek ezek a főbb egységei:

- **Column:** Ez fedi le a teljes bejelentkezési kijelzőt. Ezzel azt deklaráljuk, hogy az elemek függőleges elrendezkedésben fognak szerepelni a kijelzőn.

A `fillMaxWidth()` és `fillMaxHeight()` komponensekkel megmondjuk a Column-nak, hogy a képernyő teljes egészét ki kell tölteni. Ezáltal az ebben a blokkban szereplő elemek a képernyő közepére kerülnek.

A `verticalScroll(rememberScrollState())` -el görgethetővé tesszük a tartalmat, ezáltal ha az Androidos készülék amin használni tervezzük az alkalmazást nem rendelkezik elég nagy képernyőmérettel, vagy felbontással ugyanúgy megtudja jeleníteni a tartalmunkat.

- **Spacer:** Komponens elválasztóként működik, két elem közé beszúrva távolságot tudunk teremteni.

- **OutlinedTextField**: A bejelentkezési mezőket ezzel valósítottam meg. Az elején deklarálni tudunk pár fontos jellemzőt, amivel testreszabhatjuk ezt a blokkot.
 - `singleLine = true` lényege, hogy a mező egyetlen soros lesz.
 - `modifier.padding()` változóval margókat adunk a mezőhöz.
 - `value = email.value` esetében az e-mail aktuális értékét jelenítjük meg.
 - `onValueChange = viewModel.updateEmail(it)`
a `viewModel.updateEmail()` meghívásával frissíti az e-mailt, amikor a felhasználó begépel az adatait.
 - `placeholder` egy alap szöveget ír ki amíg nem történik gépelés, ezzel is segítve a felhasználó tájékozódását a képernyőn.
 - `leadingIcon` pedig hasonló módon megjelenít egy ikont, ami intuitívabbá teszi a kijelzőt.

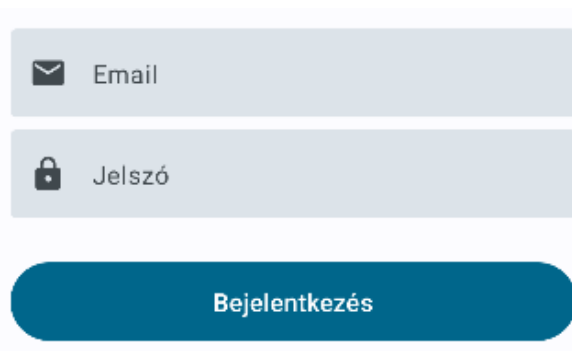
Programkód 4.11. Az `OutlinedTextField` szemléltetése.

```
OutlinedTextField(  
    singleLine = true ,  
    modifier = modifier  
        .fillMaxWidth()  
        .padding(16.dp, 4.dp) ,  
    colors = TextFieldDefaults.colors(  
        focusedContainerColor = Color.Transparent ,  
        focusedIndicatorColor = Color.Transparent ,  
        unfocusedIndicatorColor = Color.Transparent ,  
    ) ,  
    value = email.value ,  
    onValueChange = { viewModel.updateEmail(it) } ,  
    placeholder = { Text(stringResource(R.string.email)) } ,  
    leadingIcon = { Icon(  
        imageVector = Icons.Default.Email ,  
        contentDescription = "Email" ) }  
)
```

- **Button:** Értelmszerűen a bejelentkezési gomb, egyben az utolsó elem a LoginView-ban. Ennek a legfontosabb funkciója az `onClick()`. Amikor a felhasználó rákattint a gombra a ViewModel-ben meghívjuk az `onSignInClick()`-et. Ez a függvény megpróbálja bejelentkeztetni a felhasználót a korábban megadott adatok alapján. Ha a bejelentkezés sikeres a ViewModel segítségével meghívjuk az `openAndPopUp`-ot, amely átnavigál minket a CatchLogScreen-re.

Programkód 4.12. A Button szemléltetése.

```
Button(
    onClick = { viewModel.onSignInClick(openAndPopUp) },
    modifier = modifier
        .fillMaxWidth()
        .padding(16.dp, 0.dp)
) {
    Text(
        text = stringResource(R.string.sign_in),
        fontSize = 16.sp,
        modifier = modifier.padding(0.dp, 6.dp)
    )
}
```



4.3. ábra: Az elkészült Login panel.

4.6. OnlineCatchLog

Ebben a szekcióban is a Model részletezésével kezdeném. Itt részletezni fogom a háttérfolyamatokat, amik lehetővé teszik az adatbázisom működését.

4.6.1. Model

- **CatchLogDao:** A `@Dao` annotációval ellátott `CatchLogDao` egy Data Acces Object[1], amely az adatbázisom műveleteit kezeli. Itt látjuk a metódusokat, amelyen keresztül kommunikálunk a Room adatbázissal.

Programkód 4.13. CatchLogDao.kt részlet.

```
@Dao
interface CatchLogDao{

}
```

- **@Upsert:** Ez a metódus felelős azért, hogy egy új fogást rögzítsünk az adatbázisban. Másodlagosan ellenőrzi a meglévő rekordokat is.

```
@Upsert
fun upsertCatchLog(catchLog: CatchLog)
```

- **@Query:** Az alábbi lekérdezéssel elérjük az összes eddig felvitt fogást az adatbázisban. Fontos, hogy ez a lekérdezés időrendi sorrendben valósul meg, azaz mindig a legfrissebb fogás lesz felül.

```
@Query("SELECT * FROM catchlog ORDER BY catchTime ASC")
fun getLogsOrderedByTime(): Flow<List<CatchLog>>
```

- **CatchLogEvent:** Ez a zárt interfész tartalmazza az eseményeket, amelyet az alkalmazás különböző részei felhasználhatnak az események kezelésére. Minden esemény külön adat osztályként, vagy objektumként van. Ezekkel tudunk az eseményekre reagálni.
 - A **SaveCatchLog** esemény akkor hasznos, amikor frissen el szeretnénk menteni egy új kitöltött fogást.
 - A **setLakeName**, **setLakeCode**, **setFishName**, **setFishWeight** pedig az adatmezőket frissítik.
 - A **ShowDialog** illetve **HideDialog** a fogás felvitelét elindító folyamat részei. Amikor a képernyőn rá kattintunk az új fogás felvitelére ezen eseményekkel dolgozunk.
 - A **SortLogs** eseménnyel pedig beállíthatjuk az adatbázis lekérdezésének típusát, sorrendjét.

Programkód 4.14. A CatchLogEvent eseményei.

```
sealed interface CatchLogEvent {
    object SaveCatchLog
    data class setLakeName(val lakeName: String)
    data class setLakeCode(val lakeCode: Int)
    data class setFishName(val fishName: String)
    data class setFishWeight(val fishWeight: Int)
    object ShowDialog
    object HideDialog
    data class SortLogs(val sortType: SortType)
}
```

- **CatchLogDataBase:** Ezen a ponton viztosítjuk a Room adatbázis konfigurációját. Először a **@Database** annotációval létrehozuk az adatbázist. Ennek a feladata, hogy kezelje a CatchLog entitásokat, azaz a felvinni kívánt fogásokat. Ez az entitás magában foglalja a korábban említett adatokat, azaz a tó nevét, víztérkódját, hal nevét, súlyát, és a fogás időpontját. Ebben az osztályban továbbá megtalálható egy metódus, az **abstract val dao: CatchLogDao**. Ez biztosítja a hozzáférést a korábban említett CatchLogDao interfészhez, amellyel az alkalmazás a műveleteket fogja végrehajtani az adatbázison.

Programkód 4.15. CatchLogDataBase osztálya.

```
@Database(
    entities = [ CatchLog::class ],
    version = 1
)
abstract class CatchLogDataBase: RoomDatabase() {
    abstract val dao: CatchLogDao
}
```

- **DataBaseModule:** Ez egy Dagger Hilt modul, amely biztosítja az adatbázist az egész alkalmazás számára. Itt részletezni szeretném a különböző komponenseket:
 - **@Module** és **@InstallIn(SingletonComponent::class)** annotációk biztosítják azt, hogy a DataBaseModule globális függőségekkel segítse az alkalmazás többi elemét, és hogy hozzáférhető legyen az alkalmazás futása alatt.

Programkód 4.16. DataBaseModul-ban szereplő annotációk.

```
@Module
@InstallIn(SingletonComponent::class)
```

- **@Provides:** Ez a függőség segít létrehozni az adatbázist, amelyet később az alkalmazás az futása alatt használhat. Itt sajnos használnom kellett a **allowMainThreadQueries** metódust, ugyanis másképpen hibát produkált az alkalmazás. Ezt a későbbiekben fontos lenne kiküszöbölni, ugyanis az adatbázis így a fő processzor szálakon fut, ami a teljesítményünket negatív irányba eltolja. Ideális esetben ez a folyamat a háttérszálakon futna.

Programkód 4.17. provideCatchLogDataBase függőség.

```
@Provides
fun provideCatchLogDatabase(
    @ApplicationContext context: Context):
    CatchLogDataBase {
    return Room.databaseBuilder(
        context,
        CatchLogDataBase::class.java,
        "catch_log_database"
    ).allowMainThreadQueries().build()
}
```

- A `provideCatchLogDao` pedig elérhetővé teszi az alkalmazás többi részének az `CatchLogDao`-hoz való szükséges hozzáféréseket.

```
@Provides
fun provideCatchLogDao(
    database: CatchLogDataBase ): CatchLogDao {
    return database.dao
}
```

- Végezetül pedig van egy egyszerű enum típusunk, a `SortType`. Ebben az Enum-ban deklarálhatunk új rendezési típusokat, ez jelen esetben csak az idő alapú Sort, azonban ez később könnyen bővíthető köszönhetően a felépítésnek.

4.6.2. CatchLogViewModel

Itt a programkód elején 2 fontos annotáció található. Az egyik a `@OptIn`, amellyel lehetővé tesszük az alkalmazás számára, hogy kísérleti fázisban lévő funkciókat használjunk. Ez ebben az esetben a `flatMapLatest`, és a `stateIn`. Ezek a funkciók az adatáramlásra, és az állapotkezelésre vonatkoznak.

A másik a `@HiltViewModel` és `@Inject`. Ezekről szó esett már korábban is, itt azonban más függőségeket injektálunk. Ezen a ponton a `Dao`-ot, ami egy adathozzáférési objektum a `CatchLogDao`-ban. Ezzel a művelettel közvetlen hozzá tud férni az adatbázis az előbbi programkódban leírt műveletekhez.

Programkód 4.18. Annotációk szemléltetése.

```
@OptIn(ExperimentalCoroutinesApi::class)
@HiltViewModel
```

Hasonlóan a korábbi `ViewModel`-ekhez itt is van egy `onAccountClick` metódus, amely egy másik kijelzőre hivatott átirányítani minket az `openAndPopUp` segítségével.

Ezután a `ViewModel`-ben kettő állapotáramlást kell létrehoznunk, a `_sortType` és `_catchLogs`-ot. Ezekkel fogjuk kezelni a fogások listáját, valamint a korábban létrehozott rendezési sorrendet.

- `_sortType`: Ez egy `MutableStateFlow`, amely az aktuális rendezési típust tárolja. Ez estünkben a `TIME`, azaz idő szerinti sorolás.
- `_catchLogs`: Ez pedig egy `StateFlow`, ami a `_sortType` flow változásaira reagál. Ha hozzáadunk többféle rendezési típust az alkalmazásunkhoz, akkor a `flatMapLatest` segítségével egy újabb lekérdezést valósíthatunk meg.

Programkód 4.19. A különböző Flow típusok szemléltetése.

```
private val _sortType = MutableStateFlow(SortType.TIME)
private val _catchLogs = _sortType
    .flatMapLatest { sortType ->
        when(sortType) {
            SortType.TIME -> dao.getLogsOrderedByTime()
        }
    }
    .stateIn(viewModelScope,
        SharingStarted.WhileSubscribed(), emptyList())
```

A state kombinálja a `_state`, `_sortType`, `_catchLogs` flow-kat.

- **combine**: Ez a flow figyeli mindhárom áramlás aktuális állapotát. Ha bármelyikük megváltozik a state frissül.
- **stateIn**: Ez a metódus biztosítja azt, hogy a state állapotáramlása a ViewModel használatához legyen kötve, ezáltal mindig elérhető marad. Tehát amíg használjuk ezt a kijelzőt, addig fog működni. Miután nem használja az alkalmazás 5 másodpercig, a stateIn megszűnik.

Programkód 4.20. A combine szemléltetése.

```
private val _state = MutableStateFlow(CatchLogState())
val state = combine(_state, _sortType, _catchLogs)
    { state, sortType, catchLogs ->
        state.copy(
            catchLogs = catchLogs,
            sortType = sortType
        )
    }.stateIn(viewModelScope,
        SharingStarted.WhileSubscribed(5000),
        CatchLogState())
```

Ezen a ponton jutunk el, az **onEvent()**-re. Ez a metódus kezeli a fogási napló eseményeket, amelyek különböző felhasználói tevékenységeket jelznek. Itt egy **when** ággal dolgozunk, és minden különböző eseménytípusnál különböző műveletet hajtunk végre.

- **Hide-ShowCatalog**: Ez a korábban említett függvény végzi a dialógus előhozatalt, és elrakását. Ezt abban az esetben hívjuk meg ha a felhasználó új adatot akar felvinni a táblázatba, azaz naplózni szeretné a fogását. Abban az esetben pedig elrejtjük, ha feltöltött egy fogást, vagy félbe hagyja a fogás feltöltését.
- A **set** előváltozóval ellátott események felelősek azért, hogy az állapotot frissítsék az új mező értékével. Azaz amit leírnak, azt lássuk is.

```
is CatchLogEvent.setFishWeight -> {
    _state.update {
        it.copy(fishWeight = event.fishWeight)
    }
}
```


- **SaveCatchLog:** Ez az az esemény, amely a formában lévő adatokat sikeresen kitöltés esetén lementi. Ebből adódóan először ellenőrzi, hogy a `lakeName`, `fishName` helyesek-e, illetve hogy meg legyenek adva.

```
CatchLogEvent.SaveCatchLog -> {
    val lakeName = state.value.lakeName
    val lakeCode = state.value.lakeCode
    val fishName = state.value.fishName
    val fishWeight = state.value.fishWeight

    if (lakeName.isBlank() || fishName.isBlank()) {
        return
    }
}
```

Ha az előfeltételek teljesültek, azaz helyesek az adatok és nem hagytunk üresen mezőt, akkor létrehoz egy **CatchLog** objektumot, és aszinkron módba lementi a korábban létrehozott `dao.upsertCatchLog` segítségével.

```
viewModelScope.launch {
    dao.upsertCatchLog(catchLog)
}
```

Ezek után kiüríti az előbb használt mezőket, majd eltünteti a dialógust.

4.6.3. OnlineCatchLogView

Ezen a ponton elérkeztünk a `CatchLog` szakasz `View` komponenséhez. A korábban tárgyalt metódusokra csak említés szintjén szeretnék kitérni, az újakat pedig részletezem.

- **Scaffold:** Ez a `Column`-tól eltérően egy általános elrendezési eljárás. Itt csak egy alapvető struktúrát biztosítunk az alkalmazásnak, amin keresztül meg tud jelelni. Ezen belül találunk egy `topBar`-t, ami a kijelző felső részén egy sávot hoz létre, amit szabadon testreszabhatunk. A következő deklaráció a `floatingActionButton`, ami a kijelző alján egy "lebegő" gombot hoz létre, amivel interaktálhatunk.



4.4. ábra: A "lebegő" gomb.

Programkód 4.21. A topBar szemléltetése.

```

topBar = {
    TopAppBar(
        title = { Text(stringResource(R.string.app_name)) },
        actions = {
            IconButton(onClick = {
                viewModel.onAccountClick(openAndPopUp) }) {
                Icon(Icons.Filled.AccountCircle,
                    "Account")
            }
        }
    )
}

```

Ezen belül láthatjuk, hogyan is épül fel a **topBar**. Tudunk neki nevet adni a **title** változó segítségével, ami a korábban említett **Stringek** közül kerül kiválasztásra, amit már előre megírtunk. Egyel lejjebb az **actions**-nél itt is észrevehetjük, hogy az **openAndPopUp** metódust hívjuk meg. Ez azért van, mert egy gomb található ebben a szekcióban, amivel átirányítjuk a felhasználót az **Account** oldalra. végezetül pedig a kattintható gombnak adunk egy ikont.

Programkód 4.22. A lebegő gomb programbeli megvalósítása.

```

floatingActionButton = {
    FloatingActionButton(onClick = {
        onEvent(CatchLogEvent.ShowDialog)
    }) {
        Icon(
            imageVector = Icons.Default.Add,
            contentDescription = "Fogas"
        )
    }
}

```

A **floatingActionButton**-nál hasonló a helyzet. Itt azonban nem irányítjuk át a felhasználót egy másik oldalra, hanem felkinálunk neki egy dialógust, ugyanis ezzel a gombbal lehet megnyitni az adatfeltöltést. Itt beállítjuk a gomb ikonját, ami a mi esetünkben egy + jel lesz. Ezzel is törekszünk alkalmazásunk intuitívításának fenntartására.

- **LazyColumn:** Ez a sima **Column**-tól eltérően csak akkor tölti be az elemeket, ha megjelennek a képernyőn. Ez abból a szempontból fontos, hogy valószínű a felhasználónak nem 2-3 feltöltött fogása lesz, hanem valószínű több száz. Így erőforrásokat spórolhatunk, mivel nem töltjük be az összes fogást egyszerre.

Láthatjuk, hogy a betöltés csak egyszer szerepel. Ez annyiszor fut le, ahány fogást be kell töltenie az alkalmazásnak. A fontosabb információk nagyobb betűmérettel rendelkeznek, még a kevésbé fontosak, (például: víztérkód) kevésbé látszanak. Ezzel azt próbáljuk elősegíteni, hogy jobban átláthatóbb legyen a kezelői felület.

4.6.4. AddLogDialog

Az AddLogDialog majdnem egy külön komponenst képez. Ide leszünk át irányítva a lebegő gomb megnyomása után. Közvetlenül nem egy új ablak, hanem egy dialógus. Nézzük meg részletesebben.

AlertDialog: Ez biztosítja a felugró ablakot. Három fő része van. Az első az `onDismissRequest`, ahol meghatározzuk, hogy mi történjen, amikor a User megkísérel bezárni a dialógust. Ezután a `CatchLogEvent.HideDialog` eseménnyel tájékoztatjuk a `viewModel`-t a dialógus bezárásáról. A második elem a `title`, amivel értelemszerűen címet adhatunk a dialógusunknak. Ez nálam a "Fogás hozzáadása". A harmadik pedig a `text`, amellyel Column elrendezésben, azaz egymás alatt megnevezhetjük a beviteli mezők feladatát.

Programkód 4.23. Az AlertDialog felépítési váza.

```
AlertDialog(
    modifier = modifier ,
    onDismissRequest = {
        onEvent(CatchLogEvent.HideDialog)
    },
    title = { },
    text = { },
    buttons = { }
)
```

Textfield: Hasonlóan a bejelentkezési kijelzőhöz, itt is beviteli módként ezeket használok. A `value` a beviteli mező tartalma, ez a `state.lakeName` alapján frissül. Az `onValueChange` lényege, hogy ha a felhasználó bevisz egy adatot, akkor a `CatchLogEvent.setLakeName` hívjuk meg, ami frissíti a bevitt adattal az állapotot. A `placeholder` pedig hasonlóan a korábbi példával iránymutatást ad a `textField` használatával kapcsolatban.

Programkód 4.24. A lakeName beviteli mező.

```
TextField(
    value = state.lakeName ,
    onValueChange = {
        onEvent(CatchLogEvent.setLakeName(it))
    },
    placeholder = {
        Text(text = "To megnevezese")
    }
)
```

Ettől egy részen eltér a víztérkód, és a hal súlya mező. Ugyanis ezeknek a típusa `Int`, ahol nem lenne ésszerű felkínálni a teljes billentyűzetet a felhasználónak. Ezért a könnyebb bevitelt hivatott elősegíteni a `KeyboardType.Number`. Ennek a lényege az, hogy a User csak egy numerikus billentyűzetet fog kapni, amivel kifejezetten csak számokat tud bevinni a mezőbe.

Programkód 4.25. A víztérkód programrészlet szemléltetése.

```
TextField(
    value = state.lakeCode.toString(),
    onChange = {
        val lakeCodeInt = it.toIntOrNull()
        if (lakeCodeInt != null) {
            onEvent(CatchLogEvent.setLakeCode(lakeCodeInt))
        } else { }
    },
    placeholder = {
        Text(text = "Vizterkod")
    },
    keyboardOptions = KeyboardOptions(
        keyboardType = KeyboardType.Number
    )
)
```

4.7. Account

Itt elérkeztünk ennek a fejezet utolsó szekciójához, a fiókhoz. Itt kettő dolgot kellett megvalósítanom. Először is, fontos hogy a felhasználó itt ki tudjon jelentkezni az alkalmazásból. Másodszor pedig fel kellett kínálni egy lehetőséget arra, hogy vissza tudjunk menni a fogási napló részre.

Itt nem részletezném a Model részét a fióknak, ugyanis az a része megegyezik a bejelentkező szekcióval.

4.7.1. AccountViewModel

Az `initialize` felelős az alkalmazás újraindításáért miután kijelentkeztetjük a felhasználót. Ezen belül található a `restartApp` függvény, amely vissza küldi a felhasználót a Splash kijelzőre. Ebben a blokkban a `launchCatching`-en belül az `accountService.currentUser` folyamatot figyeli. Amint kijelentkeztetjük a felhasználót ez `null` állapotú lesz. Ezután aktiválódik a `restartApp`.

```
fun initialize(restartApp: (String) -> Unit) {
    launchCatching {
        accountService.currentUser.collect { user ->
            if (user == null) restartApp(SPLASH_SCREEN)
        }
    }
}
```

Ezután láthatjuk ezen szekció kettő fő függvényét. Az első az `onSignOutClick`. Ez jelentkezteti ki a felhasználót az `accountService.signOut()` meghívásával.

```
fun onSignOutClick() {
    launchCatching {
        accountService.signOut()
    }
}
```

A második az `onCatchLogClick`, ami vissza irányítja a felhasználót a fogási napló oldalára, és törli a stack-ből a jelenlegi kijelzőt.

```
fun onCatchLogClick(openAndPopUp: (String, String) -> Unit){
    launchCatching {
        openAndPopUp(CATCHLOG_SCREEN, ACCOUNT_SCREEN)
    }
}
```

4.7.2. AccountView

Itt a programkód eleje a `LaunchedEffect`-el kezdődik. Ezen a blokken belül hívjuk meg a korábban létrehozott `initialize`-t. Ez biztosítja, hogy ha nincs bejelentkezett felhasználó, akkor vissza kerülünk a bejelentkezés oldalra.

```
LaunchedEffect(Unit) { viewModel.initialize(restartApp) }
```

Továbbá létrehozunk a `showExitAppDialog`-ot, ami `mutableStateOf(false)` kezdeti értékkel létrehoz egy állapotot. Ez azt fogja jelezni az alkalmazásnak, hogy a kijelentkezési dialógus meg jelenjen-e.

```
var showExitAppDialog by remember { mutableStateOf(false) }
```

A `TopAppBar` változó itt hasonlóan az előző szekcióban említettekkel az `openAndPopUp` metódussal vissza vezeti a felhasználót az előző oldalra.

```
TopAppBar(
    title = { Text(stringResource(R.string.app_name)) },
    actions = {
        IconButton(onClick = {
            viewModel.onCatchLogClick(openAndPopUp) }) {
            Icon(Icons.Filled.ArrowBack, "Catch log")
        }
    }
)
```

A képernyő közepén egy szimpla gomb található, aminek az a feladata, hogy kijelentkeztesse a felhasználót. Amint erre rákattintunk a `showExitAppDialog` értéke igaz lesz, így megjelenik a dialógusunk.

A dialógus megjelenítésért az `AlertDialog` függvény gondoskodik. Itt a felhasználónak kettő opciót ajánlunk fel. Az első opcióban megszakíthatja a kijelentkezés folyamatát, így bent marad az alkalmazásban, és nem kell újra bejelentkeznie. A második opció a kijelentkezés megerősítése. Ebben az esetben a felhasználó vissza lesz irányítva a bejelentkezés oldalra, ahol ismét meg kell adnia az adatait, vagy át adhatja a telefonját és bejelentkezhet a horgásztársa.

```
AlertDialog(  
    title={Text(stringResource(R.string.sign_out_title))},  
    text={Text(stringResource(R.string.sign_out_description))},  
    dismissButton = {  
        Button(onClick = { showExitAppDialog = false }) {  
            Text(text = stringResource(R.string.cancel))  
        }  
    },  
    confirmButton = {  
        Button(onClick = {  
            viewModel.onSignOutClick()  
            showExitAppDialog = false  
        }) {  
            Text(text = stringResource(R.string.sign_out))  
        }  
    },  
    onDismissRequest = { showExitAppDialog = false }  
)
```

5. fejezet

Tesztelés

A fejezetben be kell mutatni, hogy az elkészült alkalmazás hogyan használható. (Az, hogy hogyan kell, hogy működjön, és hogy hogy lett elkészítve, az előző fejezetekben már megtörtént.)

Jellemzően az alábbi dolgok kerülhetnek ide.

- Tesztfuttatások. Le lehet írni a futási időket, memória és tárigényt.
- Felhasználói kézikönyv jellegű leírás. Kifejezetten a végfelhasználó szempontjából lehet azt bemutatni, hogy mit hogy lehet majd használni.
- Kutatás kapcsán ide főként táblázatok, görbék és egyéb részletes összesítések kerülhetnek.

6. fejezet

Összefoglalás

Hasonló szerepe van, mint a bevezetésnek. Itt már múltidőben lehet beszélni. A szerző saját meglátása szerint kell összegezni és értékelni a dolgozat fontosabb eredményeit. Meg lehet benne említeni, hogy mi az ami jobban, mi az ami kevésbé jobban sikerült a tervezettnél. El lehet benne mondani, hogy milyen további tervek, fejlesztési lehetőségek vannak még a témával kapcsolatban.

Források

- [1] Android. *Data Acces Object*. <https://developer.android.com/reference/androidx/room/Dao>.
- [2] Android Developers. *Save data in a local database using Room*. <https://developer.android.com/training/data-storage/room>.
- [3] Apple. *Xcode weboldala*. <https://developer.apple.com/xcode/>.
- [4] F-Droid project. *F-Droid Free and Open Source Android app store*. <https://f-droid.org/>.
- [5] Firebase. *Firebase Authentication*. <https://firebase.google.com/docs/auth>.
- [6] Flutter. *Flutter weboldala*. <https://flutter.dev/>.
- [7] Google. *Android Developer Community*. <https://developer.android.com/community>.
- [8] Google. *Android Open Source Project*. <https://source.android.com/>.
- [9] Google. *Android Studio emulator*. <https://developer.android.com/studio>.
- [10] Google. *Android Studio weboldala*. <https://developer.android.com/studio>.
- [11] Google. *Firebase weboldala*. <https://firebase.google.com/>.
- [12] Gradle Inc. *Gradle weboldala*. <https://gradle.org/>.
- [13] Graham Kendall. *Your Mobile Phone vs. Apollo 11's Guidance Computer*. https://www.realclearscience.com/articles/2019/07/02/your_mobile_phone_vs_apollo_11s_guidance_computer_111026.html. 2021.
- [14] JetBrains. *JetBrains intelliJ IDEA*. <https://www.jetbrains.com/idea/>.
- [15] Medve Flóra. *Leading mobile operating systems in Hungary in March 2024, by market share*. <https://www.statista.com/statistics/1120778/hungary-mobile-operating-systems-market-share/>. 2024.
- [16] Meet Zaveri. *What is boilerplate and why do we use it? Necessity of coding style guide*. <https://www.freecodecamp.org/news/whats-boilerplate-and-why-do-we-use-it-let-s-check-out-the-coding-style-guide-ac2b6c814ee7/>.
- [17] Microsoft. *Model-View-ViewModel (MVVM)*. <https://learn.microsoft.com/hu-hu/dotnet/architecture/maui/mvvm/>.
- [18] MOHOSZ. *Elindult a Horgász applikáció: újabb jelentős lépés a horgászbarát ügyintézés és a teljes digitalizáció irányába*. <https://nyito.mohosz.hu/index.php/szovetseg/19-kozlemenyek/593-elindult-a-horgasz-applikacio-ujabb-jelentos-lepes-a-horgaszbarat-ugyintezes-es-a-teljes-digitalizacio-iranyaba>. 2024.

- [19] React. *React Native weboldala*. <https://reactnative.dev/>.
- [20] statcounter. *Mobile Operating System Market Share Hungary*. <https://gs.statcounter.com/os-market-share/mobile/hungary>. 2024.
- [21] Unity. *Unity weboldala*. <https://unity.com/>.

CD Használati útmutató

Ennek a címe lehet például *A mellékelt CD tartalma* vagy *Adathordozó használati útmutató* is.

Ez jellemzően csak egy fél-egy oldalas leírás. Arra szolgál, hogy ha valaki kézhez kapja a szakdolgozathoz tartozó CD-t, akkor tudja, hogy mi hol van rajta. Jellemzően elég csak felsorolni, hogy milyen jegyzékek vannak, és azokban mi található. Az elkészített programok telepítéséhez, futtatásához tartozó instrukciók kerülhetnek ide.

A CD lemezre mindenképpen rá kell tenni

- a dolgozatot egy `dolgozat.pdf` fájl formájában,
- a LaTeX forráskódját a dolgozatnak,
- az elkészített programot, fontosabb futási eredményeket (például ha kép a kimenet),
- egy útmutatót a CD használatához (ami lehet ez a fejezet külön PDF-be vagy Markdown fájlként kimentve).