



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Autós integrációs lehetőségek vizsgálata modern mobil platformokon egy konkrét alkalmazáson keresztül

DIPLOMATERV

Készítette
Pénzes Martin

Konzulens
Dr. Ekler Péter

2022. december 8.

Tartalomjegyzék

Kivonat	i
Abstract	ii
1. Bevezetés	1
2. Feladatspecifikáció	6
2.1. Use Case diagram	7
2.2. Drótváz terv	9
3. Irodalomkutatás és felhasznált technológiák	12
3.1. Felhasznált technológiák	15
3.1.1. Alamofire	16
3.1.2. API	16
3.1.3. Bundler	16
3.1.4. CocoaPods	16
3.1.5. Exposed	17
3.1.6. Factory	17
3.1.7. Firebase	18
3.1.8. JSON	19
3.1.9. Kotlin	19
3.1.10. Ktor	20
3.1.11. SQL	20
3.1.12. SwiftGen	20
3.1.13. SwiftLint	21
3.1.14. Swift Package Manager	21
3.1.15. URI	22
3.1.16. YAML	22
4. Részletes megvalósítás	24
4.1. Natív iOS alkalmazás	24
4.1.1. UIKit	24
4.1.2. SwiftUI	25
4.1.3. iOS Verzió	27
4.1.4. Verziókezelés	28
4.1.5. ListenNotes API integráció	30
4.1.6. Automatizálás	34
4.1.7. Üzleti logika és képernyők	37
4.1.8. Képernyőkép generálás	42
4.2. Háttérszolgáltatás	44
4.2.1. OpenAPI	44

4.2.2. Docker	45
4.3. Teljes rendszer	47
5. Tesztelés	48
5.1. Felhasználói leírás	48
5.1.1. Telepítés	48
6. Összefoglalás és továbbfejlesztési lehetőségek	50
Irodalomjegyzék	52

HALLGATÓI NYILATKOZAT

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

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

Budapest, 2022. december 8.

Pénzes Martin
hallgató

Kivonat

Manapság egyre több autóban elérhető az Android Auto illetve az Apple CarPlay, amik lehetővé teszik a felhasználóknak, hogy vezetés közben is úgymond „csatlakozva” legyenek a világhoz azáltal, hogy a mobiltelefon az autó hangszórón keresztül olvassa fel az esetleges üzeneteiket, e-mailjeiket vagy játssza le a választott médiatartalmat, az esetleges rádióhallgatás kiváltására stb. A technológia hatalmas fontossággal bír mert, ha a felhasználó nem egy kontrollált felületen, hanem a mobiltelefonján végzi el a tevékenységet, akkor nemcsak saját magát, hanem másokat is veszélybe sodor ezzel. Így a szabadidejében podcasteket hallgató személy számára is elérhetővé kell tenni, hogy az autó infotainment rendszerén keresztül választhasson és játszhassa le a médiatartalmat és ne a telefonján babráljon.

Diplomatervemben a korábban említett probléma megoldására alkalmas applikációt terveztem megvalósítani, amelynek a használata intuitív és az esetlegesen felmerülő egyéb felhasználói igényeket is kielégíti. Az elkészült alkalmazás nyelve angol, azonban a podcastek nyelvezete terén széleskörű választék elérhető. Nem mellesleg angol nyelven szinte minden témaiban található a területtel foglalkozó podcast. Ezen felül továbbá az is feladat, hogy az itt szerzett tudást felhasználva összehasonlítsam az Android platformra való autóra integrált alkalmazás fejlesztésének lehetőségeivel. A feladat hasznosnak bizonyult, mert a megvalósítása közben mélyebben megismerkedhettem és később is kamatoztatható tapasztalatokat szerezhettem az iOS platformmal és az arra való szoftverfejlesztéssel kapcsolatban.

A dolgozat kezdetén pontosítom a kitűzött feladatot, segítséget adva a diplomamunka olvasójának az elvárt követelmények megértéséről.

A feladat által megfogalmazott kihívásnak a tudatában, az ismerkedési folyamat során szerzett tapasztalatokat adom át. Majd a megtervezési folyamatot ismertetem, hogyan álltam neki a feladatnak, azokat kisebb, könnyebben elérhető célokkel részekre felbontva.

A teljes út ismeretében pedig annak a bejárásán utazunk el a célig, vagyis a kész rendszerig, amely a feladatkiírásban lefektetett problémákra megoldást nyújt.

Végezetül pedig bemutatom az alkalmazás tesztelésének módszerét és segítséget nyújtók ahhoz, hogy az olvasó is leteszthesse ezt.

Abstract

Nowadays Android Auto and Apple CarPlay is increasingly available in more and more cars, making it possible for users to stay connected to the world around them even while driving. This is established by letting the phone use the built in speakers and microphones of the vehicle to read aloud incoming messages, e-mails and compose new ones or play custom content instead of listening to the radio. This technology is of great importance because if a driver is doing these activities with their phones in their hands instead of on a controlled screen, then they not only put themselves in danger but other road users as well. Therefore it is necessary for someone who likes to listen to podcasts in their free time to make choosing from and listening to them available through the infotainment system of a vehicle. By doing this the roads can be a safer place for everyone.

In my thesis I planned to implement an application suitable to solve the previously mentioned problem, the use of which is intuitive and also satisfies any other user needs that may arise. Although the language of this application is English, the podcasts are available in a wide range of languages. Not to mention that there are little to no subjects regarding with one can't find a podcast which talks about those. The task proved to be useful, since during its implementation I was able to get a deeper knowledge and gain valuable experience in connection with the iOS platform and software development for it.

The beginning of the thesis clarifies the set of tasks that were laid down, providing help for the reader with understanding the full scope of my master's thesis.

Aware of the challenge posed by the task, I pass on the experiences gained during the research phase. After that I will describe the planning process, how I tackled the task, breaking them down into smaller, more easily achievable goals.

Knowing the entire path, we travel along it to the goal, i.e. to the finished system, which provides solutions to the problems laid out in the task description.

Finally, I present the method of testing the application and help the reader to test it as well.

1. fejezet

Bevezetés

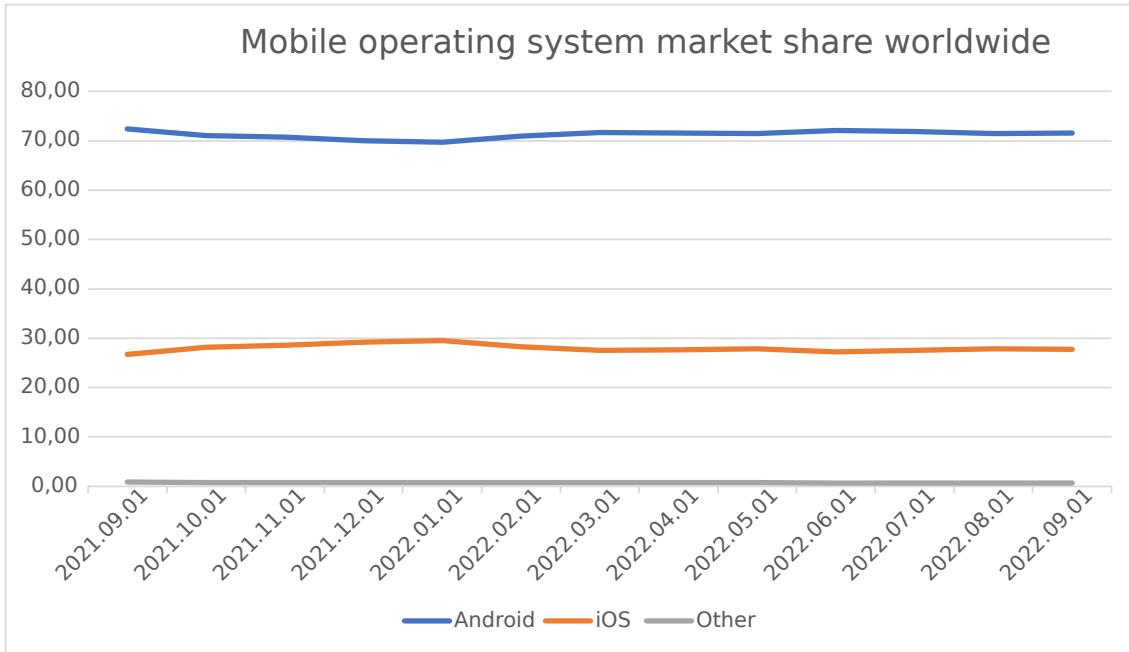
A mobiltelefonok felé irányuló érdeklődésem már tizenéves korom óta fenn áll. A legelső készülék, amit használtam a szüleimé volt. Azonban akkoriban ez nem volt képes csak a telefonálásra, üzenetek küldésére, fogadására és ami számomra a legfontosabb volt, játékra. Ilyen játék volt például a Snake, amelyben egy kígyót lehetett irányítani egy 2 dimenziós pályán és véletlenszerű módon elhelyezett almákat kellett megenni. Ezeknek az elfogyasztásával lehetett pontot szerezni, amiből értelemszerűen minél többet össze kellett szedni, amennyit csak tudtunk. Azonban az évek során egyre jobban áttért az érdeklődésem a telefonok többi funkciójára is. A legelső tárgy során, amelyen Android szoftverfejlesztéssel foglalkoztunk teljesen megfogott a platformra való alkalmazáskészítés, ezért elkezdtettem egyre jobban megismerni a lehetőségekkal. Az alapképzés végén a szakdolgozatom témajaként egy Android Auto [20] kompatibilis alkalmazás elkészítését tűztem ki célul. Azonban a mesterképzés során úgy döntöttem, hogy kipróbálom milyen az iOS platformra való fejlesztés. Az ottani tárgy során ez jobban magával ragadt így átpártoltam a másik platformra. Ezután eldöntöttem, hogy a mesterképzés végén elkészítendő diplomamunkám alatt egy iOS alkalmazást szeretnék elkészíteni.

Az Android alapú operációs rendszerek a globális piac majdnem 72%-át kiteszik (lásd 1.1. ábra), azonban a piaci eloszlás országról országra változó. Amíg Magyarországon az arány 80%-20%, az Android javára, addig az Amerikai Egyesült Államokban fordított az arány [44] 55%-45%, vagyis több az iPhone készüléket használó személy. Azonban ettől függetlenül nem szabad itthon sem csak az egyik platformot kiválasztani, mert azzal automatikusan elveszítjük az esetleges felhasználókat. Továbbá a felépített architektúra felhasználásával költséghatékonyabban lehet a másik típusú mobilra elérhetővé tenni az applikációkat, mivel már mondjuk a háttérszolgáltatást nyújtó komponens nem kerül újra megírásra csak használatra. A piaci alkalmazások átlagos felépítésére és az általam felépített rendszerre a későbbi fejezetekben fogok kitérni.

Az autós integrációk többsége lehet a mobil operációs rendszereken. Legelső lépcsőfok, amely majdnem mindegyik autóban elérhető az AUX¹ kábelrel való csatlakozás általában a középső tárolóban vagy a kesztyűtartóban található csatlakozóhoz. A következő módja, hogy a telefon képes Bluetooth² segítségével a járműhöz csatlakozni és így átjátszani a hangot. Ebben az esetben már egy, az autóban megtalálható kijelzőn keresz-

¹Telefon csatlakozó, más néven *jack* egy olyan családja az elektromos csatlakozóknak, amellyel tipikusan analóg audio jelet továbbítunk. Telefon kapcsolószekrényekhez találták fel a 19. században. Formája egy vastag tűhöz hasonlít, amely fogadására egy hasonló méretű lyuk alkalmas. Használata nagyban elősegítette a kapcsolóknál ülő dolgozók munkáját.

²Rövid hatótávolságú, adatcseréhez használt, nyílt, vezetéknélküli szabvány. Alkalmazásával számítógépek, mobiltelefonok (telefonikhangozítók), fejhallgatók és egyéb készülékek között automatikusan létesíthetünk kis hatótávolságú rádiós kapcsolatot, amihez a készülékek kis teljesítményű rádióhullámot használnak.



1.1. ábra. Operációs rendszerek globális piaci eloszlása [45]

tül tudjuk irányítani a mobiltelefont, amelyen tudunk hívásokat indítani, fogadni illetve hangfájlokat lejátszani.

A legutolsó módszer, amikor a kocsi fejegysége képes Android Auto, illetve Apple CarPlay [5] csatlakozásra. Ilyenkor nyílik legnagyobb lehetőség a mobiltelefon autón keresztüli biztonságos használatára. Az Apple CarPlay felépítéséről láthatunk képet az 1.2. ábrán. Használatukkal könnyedén elérhetjük a hívásokon és zenéken kívül a naptárunkat, hangoskönyveket, mobiltelefonra feltelepített navigációs alkalmazásokat is. Ez utóbbi hasznos lehet ha a sofőr inkább szereti használni egy harmadik fél által készített navigációs szoftvert, mint például a Waze [49] nevű alkalmazást.

Azonban az autógyártók majdnem mindenki felár ellenében telepítik az autóba az Apple Carplay és Android Auto funkcionálitást, viszont lehetőség van utólagosan is engedélyezni ezt az opciót nem szükségszerű egyből az jármű megvásárlásakor. Egyes autógyártók, mint a BMW elkezdték előfizetés ellenében engedélyezni a szolgáltatást. Ennek az ára \$80 USD lett volna évente. Azzal érveltek, hogy ezzel csökkenthető az autók kezdeti ára, illetve akinek szüksége van rá, az egy egyszerű úton hozzájuthat. Azonban ez nagy felháborodást keltett, mivel más autógyártóknál egy kezdeti felár ellenében aktiválásra kerül az autókban. Ez a kezdeti felár átlagosan kettő, maximum három évnyi előfizetés összege, amelynek tudatában érhető, hogy a jármű élettartam során többszörösen ráfizetnek. A felháborodást követően visszavonták a döntést 2019-ben és onnantól ingyenessé tették az Apple Carplay-t az autóikban [10], azonban kérdéses még, hogy ezt az ár kiesést beleépítették-e az autók indulórárába. Nem mellesleg a BMW hasonló lépést akar megtenni az ülésfűtés terén is, amely szintén előfizetés alapon működne.

Android Auto-ra és Apple CarPlay-re a csatlakozás működhet vezetékkel, ami elterjedtebb, illetve vezeték nélkül is. A két opció között nincs lényegbeli különbség csak a csatlakozás módja. Értelemszerűen a vezeték nélküli kapcsolódás nagyban elősegíti a használó kényelmét. Az újabb autókban továbbá lehetőség van vezeték nélküli töltésre, amely a középső konzolban szokott elhelyezésre kerülni. Ahhoz, hogy telefon vezeték nélkül tudjon tölteni az autóban, mind annak, mind a vezeték nélküli padnak tartalmaznia kell induktív tekercset, alapvetően rézhuzalba csomagolt vasat. Amikor a tekercsek közel kerülnek



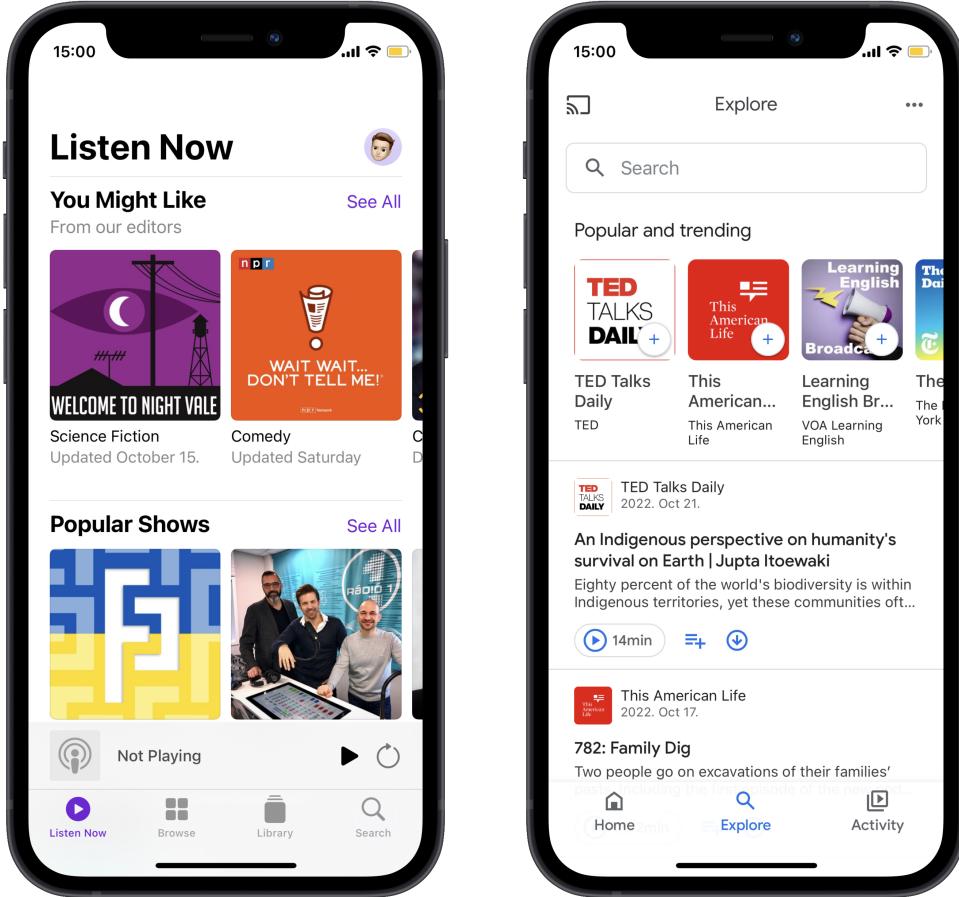
1.2. ábra. Apple CarPlay autó infotainment rendszerén

egymáshoz, azaz a telefont az autó vezeték nélküli töltőpadjára helyezzük a tekercsek közelege lehetővé teszi, hogy elektromágneses mező jöjjön létre, amelynek következtében lehetőség nyílik, hogy egy tekercsről áramot tudjon átadni (a töltőpadról) másikra (a telefonra). A telefonban lévő indukciós tekercs ezután továbbítja az elektromos áramot az akkumulátorához és ezáltal töltődik. Azonban hátránya is van ennek a fajta elektromosság átadásnak, mert nem képes ugyanakkora mennyiségű töltést továbbítani, így az eszközök lassabban töltenek. Nem mellesleg az is közre játszik, hogy a töltés során nagyobb mennyiségi energia vész el a hő formájában, ezáltal a telefon hajlamos jobban melegedni, amely pedig az akkumulátor élettartamának a gyorsabb csökkenéséhez vezethet. Egy Pixel 4-gyel végzett amatőr, 2020-as energiafelhasználási elemzés [41] megállapította, hogy a vezetékes töltés 0 és 100 százalék között 14,26 Wh-t (wattórát), míg egy vezeték nélküli töltőállvány 19,8 Wh-t, ami 39%-os növekedést jelent. Egy általános márkaúj vezeték nélküli töltőpárna és a telefon rosszul igazítása akár 25,62 Wh fogyasztást is eredményezett, ami 80%-os növekedést jelent. Az elemzés megállapította, hogy bár ez valószínűleg nem lesz észrevehető az egyének számára, negatív következményekkel jár az okos telefonok vezeték nélküli töltésének elterjedebbé tételeben, mivel több mint hatmilliárd okos telefon használnak világszerte.

Az előző bekezdésben említett töltési módszerre az Apple által készített iPhone-ok már az iPhone 8 széria óta képesek, amelyek 2017-ben kerültek kiadásra. Továbbá a Google Pixel szériája is tudja ezt a funkcionálitást a Pixel 3 óta, amely 2018-ban került ki a piacokra. Az indukciós töltés egyébként nem egy új technológia, így töltenek például az elektromos fogkefék is, amik manapság már elég elterjedtek. Az Android Auto-hoz illetve az Apple CarPlay-hez való vezeték nélküli csatlakozás során sem kell amiatt aggódni, hogy mégiscsak szükségünk lesz egy kábelre ha esetleg lemerülne a telefonunk.

A technológia egyre gyorsabb terjedésének következtében relevánssá vált, hogy az autók infotainment rendszerén is használhatóvá tegyük a releváns alkalmazásokat. Ennek a következtében döntöttem úgy, hogy egy új alkalmazást készítetek, amely a követelménynek eleget tesz. Az alkalmazás témája pedig a podcastek köré épül. A podcast egy olyan program, amely digitális formátumban letölthető az internetről. A podcast-sorozatok általában egy vagy több visszatérő házigazdát tartalmaznak, akik egy adott témáról vagy aktuális

eseményről vitáznak. A podcastokon belüli vita és tartalom a gondosan megszerkesztettől, a teljesen rögtönzöttig terjedhet. A kidolgozott és művészeti hangzást ötvözik a tudományos kutatástól az életrajzi szelet újságírásig terjedő tematikus témaikkal. Sok podcast-sorozat tartalmaz egy kapcsolódó webhelyet linkekkel és műsorjegyzetekkel, vendégéletrajzokkal, átitratokkal, további forrásokkal, kommentárokkal, és még a műsor tartalmának megvitatására szolgáló közösségi fórummal is.



1.3. ábra. Apple Podcasts (balra) és Google Podcasts (jobbra)

Az ilyen jellegű tartalmak lejátszására már több alkalmazás is megtalálható az App Store [3] illetve a Play Store [27] felületén. Ezek olyan beépített alkalmazások, amelyekkel már valószínűleg találkozhatott is az olvasó, innen könnyedén le tudjuk tölteni az eszközéinkre az alkalmazásokat. Az Apple készülékein az App Store elérhető, míg az Androidos eszközökön a Play Store. Ezekhez hasonló még az AppGallery [29], amely a Huawei készülékeken elérhető. Erre azért van szükség, mert a Huawei eszközökön már nem érhetők el a Google szolgáltatásai. A jelenleg is piacon lévő és sok felhasználóval bíró podcastek lejátszására képes alkalmazások a Spotify, csak az iPhone készülékeken elérhető és azonban alapból megtalálható Apple Podcasts applikáció és a Android készítői által fejlesztett Google Podcasts. Az előbb bemutatott alkalmazások felületéről az 1.3. ábrán találhatóak képernyőképek. A jelenleg meglévő megoldások alap funkcionálisként mind képesek arra, hogy a podcasteket le tudják játszani, nem véletlen hiszen ez a téma ami köré épülnek azonban nem ez teszi őket egyedivé. Ezen felül a felhasználó az általa kedvelt sorozatokra fel tud iratkozni, így könnyebben elérve őket. Értesítések kérésére is van lehetőség. Ilyenkor az általunk kedvelt podcastek új epizódjairól küld értesítést a felhasználóknak. Azonban

csak a Spotify képes arra, hogy ismerősöket kövessünk. Viszont kutatásom során arra jutottam, hogy ez az alkalmazás csak a zenehallgatást osztja meg a többi felhasználóval, a podcast sorozatok hallgatási szokásokról semmi információt nem továbbít. Az általam elkészített alkalmazásban ezeket a funkciókat szerettem volna megvalósítani így a munkám elején egy hasonló funkcionalitással rendelkező applikáció tervét készítettem el.

- a 2. fejezetben a feladat pontosabb leírását és az elkészítés előtt elkészült terveket mutatom be,
- a 3. fejezetben a munka megkezdése előtti irodalomkutatásról lesz szó ahol az Android Auto és Apple Carplay mellett az általános technológiáknak bemutatása a téma,
- a 4. fejezetben bemutatom a fejlesztés részletes menetét és az elkészült program felépítését,
- az 5. fejezetben a felhasználók számára ismertetem a használat módját képernyőképpel,
- a 6. fejezetben összegzem a projekt során tapasztaltakat és felsorolom az általam talált továbbfejlesztési lehetőségeket,
- végezetül pedig az irodalomjegyzék látható.

2. fejezet

Feladatspecifikáció

A feladatkiírás elkészítendő munkaként egy olyan összetett rendszer megvalósítását kéri amely több elemből épül fel. Legfőbb funkcionalitása a podcastok lejátszása és elmentése. A rendszer egyik része a felhasználó által is látott natív iOS alkalmazás. Ebben a felhasználónak képesnek kell lennie, hogy listát böngészve saját preferenciái alapján kedvencek közé elmenthesse az előzőleg bemutatott podcastokból tetszőleges számút. A böngészés elősegítése érdekében a listában csak minimális mennyiségű információ lehet elérhető, hogy az olvasó ne érezze magát elárasztva a rengeteg szövegtől. Ha azonban részletekre is kívánCSI az egyes sorozatoknál, akkor a megadott elemekre kattintás után egy részletező nézetnek kell megjelennie. Itt további hasznos információk segítik a felhasználót, amelyek felkelthetik az érdeklődését a podcast iránt, esetlegesen a kedvencek közé is teheti, hogy később meg tudja egyszerűen hallgatni. Ahhoz, hogy a már korábban lementett tartalmakat könnyen tudja kezelní, egy külön képernyőn található listán csak azokat mutatva, együtt is meg kell jeleníteni. Ebben a listában változtatásokat végezhet a kedvencek közé tett tartalommal kapcsolatban, viszont ügyelni kell arra is, hogy esetleges véletlenszerű törlés során lehetőség legyen a művelet visszavonására is. A használó által látott tartalom minél nagyobb mértékű személyre szabása érdekében a hallgatási szokások és beállítások alapján ajánlásokat is kell mutatni a kezdőképernyőn, amelyek között olyan új podcastokat találhat, amikről eddig még nem hallott, vagy csak elterelték a figyelmét. Ezen felül a felhasználónak lehetőséget kell adni ahoz, hogy szűrni tudja a számára elérhető tartalmat. Jelentheti ez azt, hogy felnőtt tartalommal rendelkező podcasteket nem szeretne látni, illetve azt is, hogy a sorozatokat csak egy bizonyos nyelven vagy csak egy specifikus régióból szeretne hallgatni. A nyelv használatos a podcast során beszélt nyelvnek a specifikálására, a régió pedig keresések során játszik szerepet, ilyen például, hogy a legfelkapottabb sorozatok keresése során megadható a régió, amelyre szeretnénk szűrni.

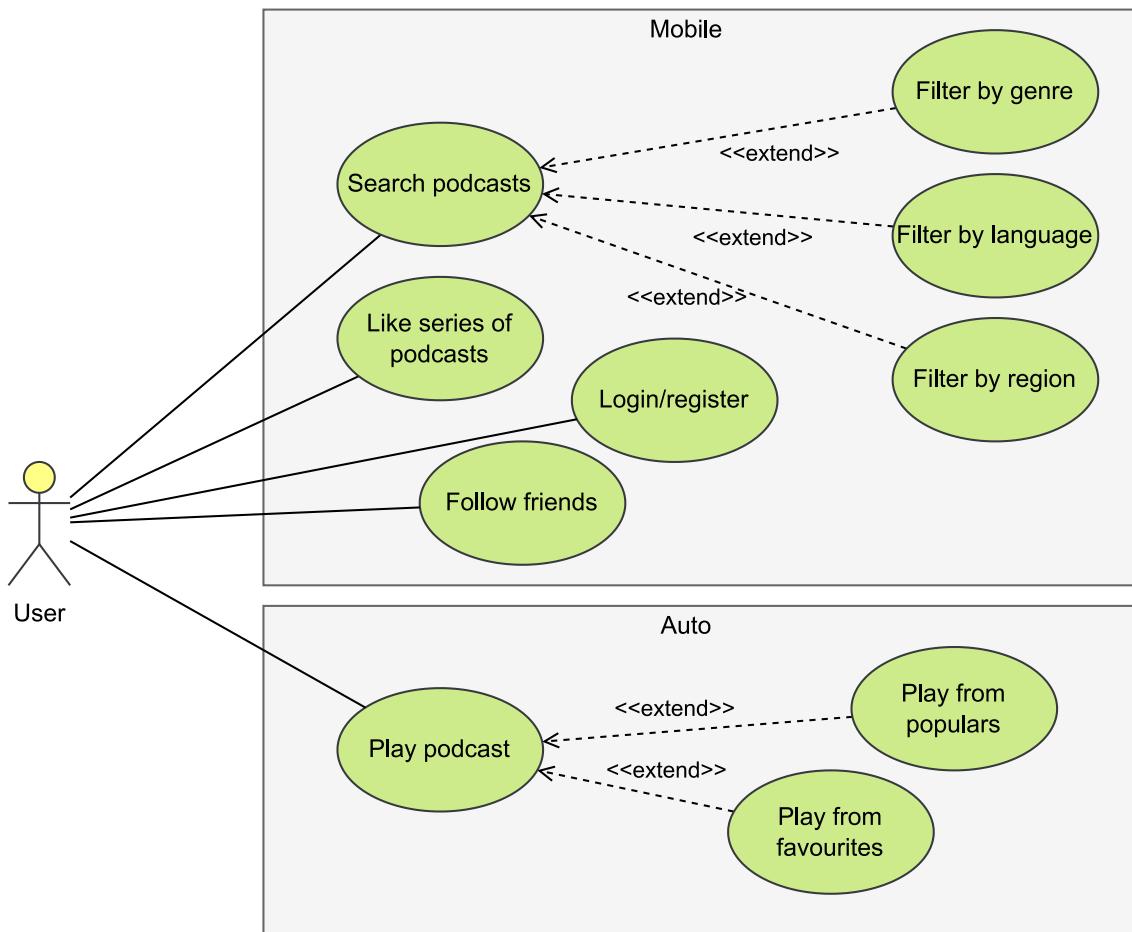
A rendszer másik része egy szolgáltatás, amelyen keresztül a felhasználókról adatokat tárolnak. Ez egy backend¹ szolgáltatást foglal magában, amely kiszolgálja az alkalmazás által indított kéréseket. A felhasználókról az adatokat egy adatbázisban szükséges eltárolni, amelyhez csak a szolgáltatás férhet hozzá. A tárolni kell a felhasználókról a preferenciákat, mint például a nyelv, régió, explicit tartalom engedélyezése. Ha extra információkat is megad, amik közül lehetőség van a névre, születési dátumra, esetlegesen ha nem szeretné megadni a teljes dátumot, akkor csak az évre és a nemre. Az így jelen lévő adatokat felhasználva a szolgáltatás képes hasonló személyeknek a megtalálásában, amelyek hallgatási előzményeit felhasználva ajánlani kell tudni a felhasználónak podcast sorozatokat, amelyek a személyi hasonlóság következtében elnyerhetik tetszését.

¹Alkalmazások adatait nyújtó, nem lokálisan futó szoftver. Internetes kommunikáció keresztül érhető el. Párja a frontend, amely jelen esetben a natív iOS applikációt foglalja magában.

A feladat elvégzése után pedig a szerzett tapasztalatokat, fejlesztési lehetőségeket és korlátokat jellemzni kell, amik az Android Auto és Apple CarPlay platformokat jellemzik. A feladatkiírás csak egy Apple CarPlay kompatibilis applikáció elkészítését kéri, azonban az alapképzés végén elkészített szakdolgozatom megalkotása során szerzett Android Auto tapasztalatokat kutatással ötvözve összehasonlítás tehető a két platformmal kapcsolatban.

2.1. Use Case diagram

A natív iOS alkalmazásnak az elkészítése azzal kezdődött, hogy a felhasználó szemszögeből átgondoltam milyen folyamatokat lesz képes az applikációban végrehajtani, majd ezeket összeszedve egy összefoglaló use case diagramot² készítettem, amely a 2.1. ábrán található.



2.1. ábra. Alkalmazáshoz készült use case diagram

Először is szükség van egy olyan folyamatra, ahol a felhasználó böngészheti a számára elérhető podcastokat. Itt ki kell ajánlani az összes API-n megtalálható sorozatot. Azonban a jelenleg elérhető különböző podcast sorozatok száma az általam használt szolgáltatáson keresztül 2 961 490 darab [39]. Ennek következtében a felhasználó számára lehetetlen, hogy egy egyszerű listában böngészve ráleljön arra, amelyet meghallgatva megragadja és beleszeret a sorozatba. Az egyik mód az előző probléma megoldására az, hogy egy

²A use case diagram (magyarul: használati eset diagram) a felhasználó lehetséges interakcióinak grafikus ábrázolása egy rendszerrel. A használati eset diagram különböző típusú felhasználókat mutat be, akikkel a rendszer rendelkezik, és gyakran más típusú diagramok is kísérik használatát. Az egyes használati eseteket jelölhetik körök vagy ellipszisek. A rendszerben szereplőket gyakran pálcikafigurákkal jelölik.

képernyőn keresési lehetőség van a podcastok között. Itt a felhasználó képes arra, hogy többféle feltétel mentén szűrje az összes elérhető sorozatot. A legalapvetőbb módja ennek, hogy egy szöveges beviteli mezőbe beírja a keresés szövegét, majd ezt továbbítva az API felé már egy minimális hosszúságú listát látva könnyedén tudja a kiválasztást végezni. Az egyetlen limitáló faktor a felhasználó képzelőereje, hogy milyen keresési szöveget ír be. Itt lehet a szűrést végezni a podcast sorozatok szöveges tartalmain, mint például a címe és leírása, vagy esetlegesen az epizódok szövegi részein, amely szintén annak a címe és leírása. A kapott találatok további szűrésére is van lehetőség. Egy sorozat során a házigazda által beszélt nyelv nem változik, ezáltal ha csak angol nyelven szeretne a felhasználó hallgatni, akkor arra is van lehetőség. A szolgáltatás által nyújtott podcastokat 73 különböző nyelvre lehet szűrni, amelyek közt megtalálható a magyar is, illetve egzotikusak is, mint például a csamorro [50] az ausztrónéz nyelvcshaládból, amit a Mariana-szigeteken beszélnek vagy a feröeri nyelv [51] az indoeurópai nyelvcshaládból, amelyet a Feröer szigeteken honos. Ezeket rendre ~58 000 és ~70 000 ember beszéli így látható, hogy az alkalmazásban sokféle nemzetiségi felhasználó hallgatási preferenciáit ki tudom elégíteni. Továbbá lehetőség van az egyes régiók alapján is szűkíteni a listát.

A képzelőerő, mint limitáló faktort eliminálva megjeleníthető egy olyan kétdimenziós lista, amelyben az első dimenzió tartalmazza a podcast sorozatok kategóriáit, vagy az ismerősöket akiket a felhasználó követ. A második dimenzió pedig tartalmazhatja a megadott kategóriában legnépszerűbb elemeket, illetve a barátok által lementett szériákat. Itt az első dimenzió lenne a telefonnak a fő görgetési irányban, azaz vertikálisan és a második pedig a másodlagos direktíban, vagyis horizontálisan.

Az imént említett listákat felhasználva az alkalmazásban a hallgató képes böngészni a számára elérhető temérdek mennyiségű sorozatokat és megtalálni azokat, amelyekbe később meghallgatva beleszeret és rendszeres hallgatója lesz. Egy probléma viszont van, mégpedig az, hogy ezeket sajnos fejben kell tartania. Ezt nem várhatjuk el a felhasználótól, ezért lehetőséget kell nyújtani ahhoz, hogy a kedvelt sorozatokat megtalálásukkor lementse, elősegítve a későbbi egyszerű megtalálásukat.

A barátok követésének megvalósítására egy új folyamatra van szükség, ahol a felhasználó kereshet a meglévő regisztrált hallgatók közül. Az adatvédelmet szem előtt tartva erre csak úgy van lehetőség, ha tudjuk a másik személynek a felhasználói azonosítóját. Ezáltal nem engedélyezett, hogy csak úgy ismeretlen embert kövessünk. A felhasználói azonosítót a hallgató választja, azonban ennek egyedinek kell lennie.

A felhasználók azonosíthatósága érdekében az előző folyamathoz szükség van arra, hogy regisztrálni tudjanak. Ez továbbá elősegíti azt is, hogy ha több eszközön is használnánk az alkalmazást, akkor minden telefonon elérjük a már kedvelt sorozatokat, illetve eszközváltáskor is megmaradnak a lementett információk. Azonban ha csak ezek a többlet funkciók érhetőek csak el a regisztrációval, akkor a felhasználók nagy része nem fogja igénybe venni, hanem csak anélkül fogják használni az applikációt. Ezáltal folyamat végettével lehetőség van a preferenciák megadására is, amelyben specifikálhatóak a hallgatási szokások, mint a korábban említett régió és nyelv vagy esetlegesen a kedvelt műfajok, például az egészség és fitnessz vagy a krimi.

Ahogyan ezt már többször is említettem korábban, az applikáció fő funkcionalitása a podcastok lejátszása így ez magában foglal egy külön folyamatot. Azonban a tervezés során azt a megkötést tettek, hogy csak az autóra csatlakozva lesz erre képes Apple CarPlay segítségével. Későbbi bővíthetőséget szem előtt tartva könnyen kiegészíthetőnek tartottam ezt a tulajdonságot a mobiltelefonra is így nem jelent hátrányt fejlesztői szempontból a kihagyása vagy elhalasztása. A felhasználó itt több kisebb folyamat közül választhat a hallgatási forrás kiválasztásának szempontjából. Ha korábban a telefont használva már mentett le sorozatokat a kedvencei közé, akkor ezeket egész nagy valószínűsséggel szeretné is hallgatni. Ennek következtében mutatni kell egy olyan listát, ahol ezek közül tud vá-

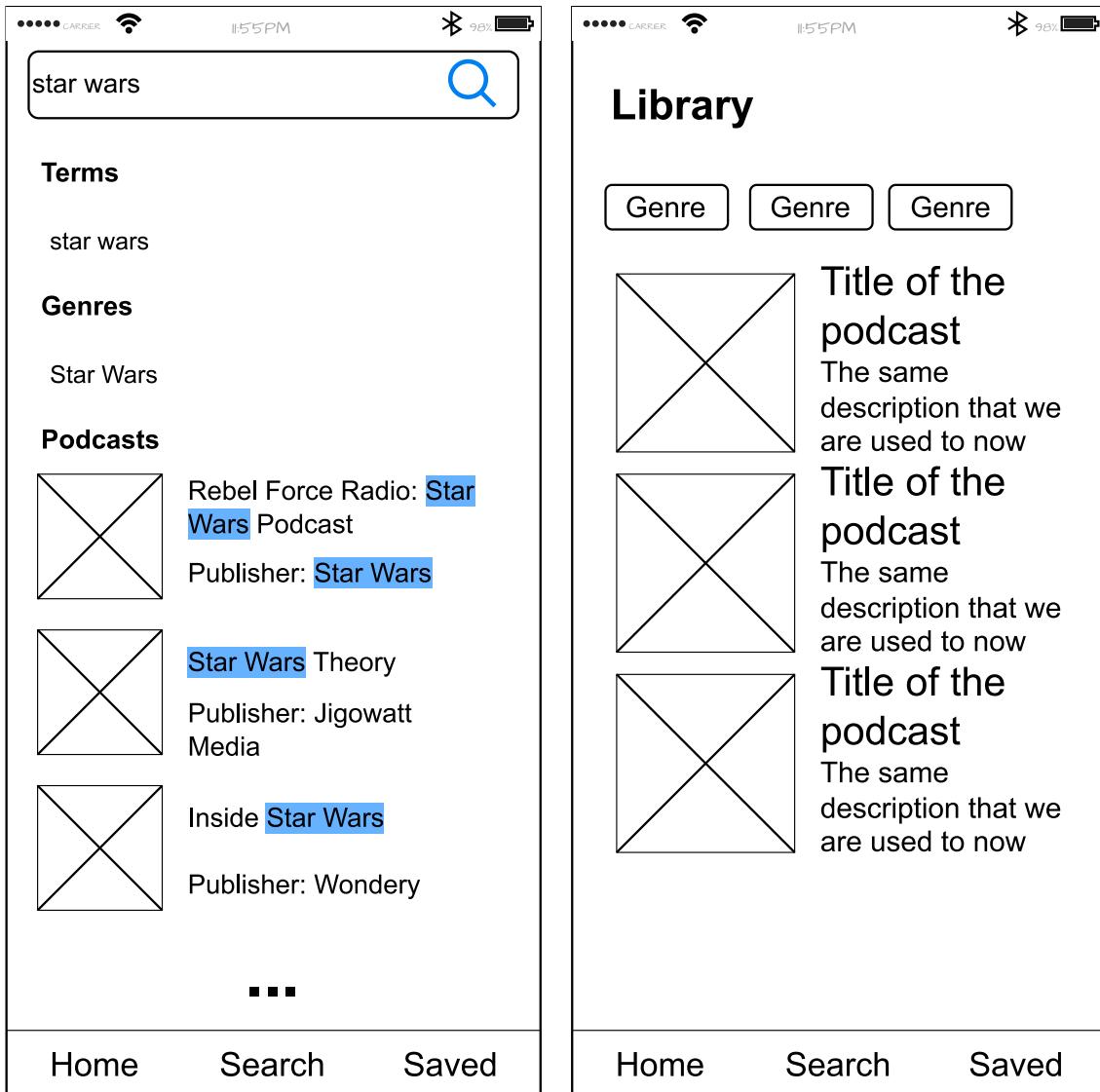
lasztani. Azonban előfordulhat az az eset, hogy nincs lementve egyetlen egy sorozat sem a kedvencek közé. Ilyenkor nem szabad hagyni, hogy a felhasználó ne tudja használni az alkalmazást. Ezáltal egy olyan listát is tartalmaznia kell a felhasználói felületnek, ahol az esetleges korábban beállított preferenciáinak a felhasználásával, vagy ha ezek hiányoznak akkor szűrés nélkül láthatóak a jelenleg népszerű podcast sorozatok. Így a sofőr vagy esetlegesen utasa minden esetben képes lesz a podcastok hallgatására és nem fog emiatt átpártolni egy konkurens applikációhoz.

2.2. Drótváz terv

Az applikáció megírása előtt szükség volt még egy lépésre, amely során az alkalmazás képernyőiről egy kezdetleges tervezetet készítetek. Itt tanulmányoztam a külső API által nyújtott információkat és azok alapján minimális képernyőket készítettem el. Ezeket a terveket úgy nevezik hogy wireframe (drótváz), amelyek során a képernyőkről egy skicc szerű design készül.



2.2. ábra. Kezdőképernyő (bal oldalon) és podcast részletező nézet (jobb oldalon) drótváz tervei



2.3. ábra. Keresőképernyő (bal oldalon) és a könyvtár (jobb oldalon) drótváz tervei

Legfőbb szempontom az elkészítés során az volt, hogy minél több adatot meg tudjak jeleníteni a felhasználó számára, azonban ezeknek a lekéréséhez ne legyen szükség több végpontot meghívni. Ez a szempont egy kitalált példával illusztrálva válik egyszerűvé. Tegyük fel, hogy egy listában meg akarok jeleníteni menhelyen lévő kisállatoknak neveit, fajtáját és életkorukat, egy részletes nézetben pedig még egyéb adatokat is, mint például a tulajdonságaikat, allergiáikat. Egy hálózaton keresztüli API hívás segítségével elkérek egy listát, amely visszaadja a nevüket és fajtájukat az egyes állatoknak, azonban az életkorukat, jellemzőiket és allergiáikat csak egy másik végponton keresztül érem el, ahonnan egy specifikus állatra lehet csak keresni. Ennek következtében a lista nézetben nem szeretném megjeleníteni az életkort, mert ez azt eredményezné, hogy a kezdetleges hálózati híváson kívül minden elemre kell még egy hívást indítani. A helyzet nem feltétlen problémás, mert amikor megjelenítem a részletes képernyőt, akkor a meglévő adatokat fel tudom használni. Viszont azokhoz az állatokhoz is lekértem, amelyeknél a felhasználó nem nyitotta volna meg a részletes nézetet, ezáltal nagy mértékű hálózati kommunikációt eredményezve feleslegesen. A probléma éles alkalmazásoknál nem merül fel, mert azok megírása során

kommunikálva van, hogy milyen adatokra lesz szükség egy képernyőn és ezáltal az életkor is visszaadásra kerül az egyes állatokhoz. Azonban jelen esetben az API egy külsős szolgáltatás, így nekem kell a alkalmazást úgy megvalósítani, hogy az ne jelentsen a legelső pillanattól hátrányt.

Az előző fejezetben bemutatott folyamatokhoz készítettem el a szükséges tervezeteket. Ezekben csak nagyjából vannak felrakva elemek, amelyek egy általános keretet adnak a később megvalósítandó designnek. A tervek egy része látható a 2.2 és a 2.3 ábrákon. Az általam használt oldal a diagramok elkészítésére a diagrams.net/draw.io. Ez egy nyílt forráskódú technológiai halom diagramalkalmazások készítésére, és az ő megállapításuk szerint a világ legszélesebb körben használt böngészőalapú végfelhasználói diagramkészítő szoftvere. Használata kényelmes, mert megígéri, hogy nem rejtiük el az adataink, és minden biztosítanak valamilyen módot az adatok ingyenes megnyitására és szerkesztésére. Az az elgondolásuk, hogy amikor a cégek pénzt fizetnek nekik, annak azért kell lennie, mert hozzáadott értéket adnak, nem pedig azért, mert elzárjuk az adataikat. Ezekből a designekből később a tervezők pixel pontos design terveket készítenek, amelyeket a fejlesztők felhasználva megvalósítják a kérő képernyőket. Azonban úgy ítétem meg, hogy nekem a pontos designek elkészítéséhez túl sok időre lett volna szükség, ennek következtében ezt a lépést kihagyva egyből a fejlesztés során alakultak ki a szükséges tervek.

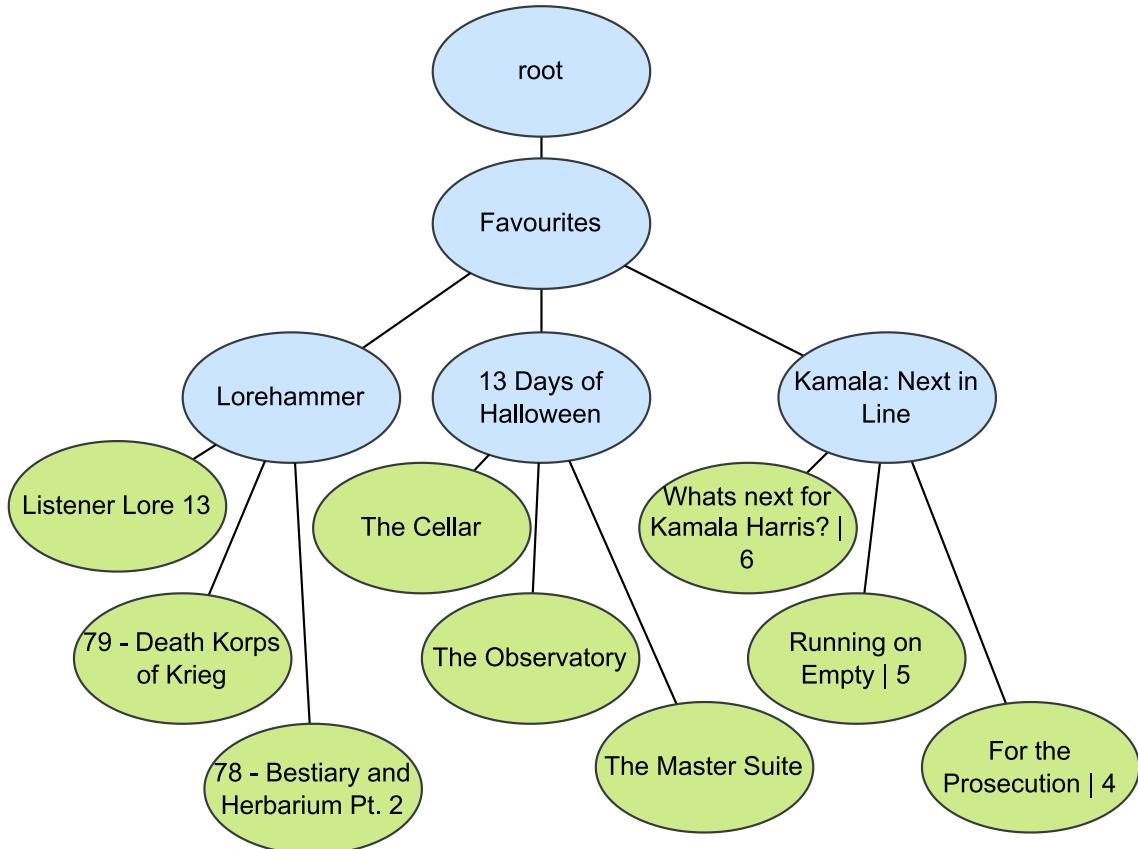
3. fejezet

Irodalomkutatás és felhasznált technológiák

Az Android Auto és a Média alkalmazás fejlesztés is már ismert volt számonra, mivel korábban már készítettem ezekben a témaévekben alkalmazásokat. Az alapképzés során az önálló laboratórium és alatt és szakdolgozat során is egy média alkalmazást készítettem el. Az elsőnél egy híreket felolvasó applikáció készült el. Itt a felhasználó kiválasztotta, hogy milyen témaévekben érdeklik a legfrissebb információk, majd azokat az autóban ülve Android Auto segítségével meghallgathatta őket. Az utóbbi során pedig a jelenlegihez nagyon hasonló alkalmazás készült el csak natív Android nem natív iOS platformra és nem Apple Carplay hanem Android Auto felhasználásával. A lehetőségek bemutatását az Android platformmal kezdem, amelynél az autós integrációs alkalmazáshoz való fejlesztésnek kétféle módja van: Android Auto és Android Automotive OS. A kettő nagyban különbözik egymástól ezért nem összetévesztendők. Az Android Auto egy vezetőre optimalizált alkalmazásélményt biztosít azoknak a felhasználóknak, akik Android telefonnal és Android Auto alkalmazással rendelkeznek, de az járműük nem képes az Android Automotive OS futtatására. Ha egy felhasználó autója vagy utángyártott sztereórendszerre támogatja az Android Auto-t, akkor telefonja csatlakoztatásával közvetlenül az autója kijelzőjén használhatja az alkalmazásokat. Engedélyezhetjük az Android Auto számára a telefonon lévő alkalmazáshoz való csatlakozást azáltal, hogy olyan szolgáltatásokat hozunk létre, amelyek segítségével az Android Auto platformra optimalizált felületet jeleníthetünk meg a felhasználók számára. Az Android Automotive OS egy Android-alapú információs és szórakoztatónrendszer, amely járművekbe van beépítve. Az autó rendszere egy önálló Android-eszköz, amely vezetésre van optimalizálva. Az Android Automotive operációs rendszerrel a felhasználók telefonjuk helyett közvetlenül az autóra telepítik az alkalmazást.

Médiaalkalmazások esetén az alkalmazásnak tartalmaznia kell egy médiaböngésző szolgáltatást [21]. Ugyanazt a médiaböngésző szolgáltatást használhatjuk az Android Automotive OS és az Android Auto rendszerrel is. Általánosságban is igaz, hogy a kettőre való együttes fejlesztés során lehetőségünk van úgy megírni az applikációkat, hogy a közös szolgáltatásokat kiszervezve minimalizálható a redundáns kód. A médiaböngésző szolgáltatás egy olyan Android rendszerben megtalálható interfész, amelynek a megvalósításával közzétehetjük az alkalmazásunk tartalmát. A médiaböngésző a médiaalkalmazások által használt API a médiaböngésző-szolgáltatások felfedezésére és azok tartalmának megjelenítésére. Az Android Auto és az Android Automotive OS médiaböngészőt használ az alkalmazás médiaböngésző szolgáltatásának megtalálásához. Az Android az egyes elemeket *MediaItem* (média elem) [22] nevezi. A korábban említett böngésző a tartalmát egy ilyen egységekből álló fába rendezzi. A fára egy szemléletes példa látható a 3.1. ábrán, ahol három podcast szerepel, mint böngészhető elemek és minden podcasthez három epizód.

Az egyes elemekre kétféle címkét tehetünk. Az egyik a *Playable* (lejátszható), ez a jelölés azt jelzi, hogy az elem egy levél a tartalmat alkotó fán. Az elem egyetlen hangfolyamot képvisel, például egy dalt egy albumon, egy fejezetet egy hangoskönyvben vagy ami az én alkalmazásomhoz illik egy podcast egy epizódját. A másik pedig a *Browsable* (böngészhető), amely pedig azt jelenti, hogy az elem egy csomópont a tartalomfán, és gyermekei vannak. Például az elem egy albumot képvisel, gyerek elemei pedig az albumon található dalok, vagy egy podcastet jelképez ahol a gyerekek az epizódok. Az a médialelem, amelyen megtalálható minden kettő címke egy lejátszási listaként működik. Kiválasztjuk magát az elemet, hogy az összes leszárma zottat lejátsszuk, vagy böngészhetjük is azokat.



3.1. ábra. Böngészési fa példa részlet (kékkel a böngészhető, zölddel a lejátszható elemek)

Az Android Auto platformon megjelenő képernyők azokra optimalizáltak. Nincs is szükség arra, hogy nekünk kelljen létrehozni ezeket a képernyőket, hanem csak az adatot kell a platform felé elküldeni általa megadott formátumban és ezután a többiről maga a rendszer gondoskodik. Azonban ha az Automotive OS-t is támogatni akarjuk, akkor azokhoz tartozó tevékenységeket nekünk kell megtervezni. Ezen felül specifikus képernyőket, mint például a bejelentkezés és a beállítások, amelyeket kifejezetten az Android Automotive OS-hez kell elkészíteni. Az így elkészített képernyőknek meg kell felelnie az autóra készített irányelveknek [24]. Ez jellemzően nagyobb koppintási célokot és betűmérteket, nappali és éjszakai módok támogatását, valamint magasabb kontrasztarányokat foglal magában. A járműre optimalizált felhasználói felületek csak akkor jeleníthetők meg, ha az autós felhasználói élményre vonatkozó korlátozások (CUXR) nincsenek érvényben, mivel ezek a felületek fokozott figyelmet vagy interakciót igényelhetnek a felhasználótól. A CUXR-ek nem érvényesek, amikor az autó leáll vagy leparkolt, de mindenkor érvényben vannak, amikor az autó mozgásban van.

Az applikációnk témájaként öt lehetséges opció közül választhatunk, amik nem minden platformra egyaránt elérhetőek. Az első témakör a médiaalkalmazás, amely ahogy korábban említve volt a felhasználóknak nyújt bongészési és lejátszási tulajdonságot zenéhez, rádióhoz, hangoskönyvekhez vagy egyéb hangalapú tartalomhoz az autóban, amely minden platformon elérhető. Az üzenetküldő alkalmazások, amelyek csak Android Auto-n elérhetőek lehetővé teszik a felhasználók számára a bejövő értesítések fogadását, az üzenetek hangos felolvasását szövegfelolvasó segítségével, és az autóban lévő hangbemeneten keresztül válaszokat küldhetnek. Az Android for Cars (Android autók számára) könyvtár segítségével lehetőségünk van térképes alkalmazások megvalósítására. Az egyik ilyen a navigációs applikációk, beleértve a járművezetői/kiszállítási szolgáltatásokat nyújtó szolgáltatókat is, amelyek részletes útbaigazítással segítik a felhasználókat arra, hogy eljussanak a kívánt helyre. A másik módja a Point of Interest applikációk (POI - érdeklődési pontok). A POI-alkalmazások lehetővé teszik a felhasználók számára, hogy megtalálják a szükséges helyet, és megtegyék a megfelelő műveleteket. Ez magában foglalja, de nem kizárolagosan, a parkolási, töltési és üzemanyag-alkalmazásokat, amelyek segítenek a felhasználóknak parkolóhely megtalálásában, illetve a legközelebbi töltő- vagy üzemanyagtöltő állomás felkeresésében. Az ilyen alkalmazások lehetővé teszik a felhasználó számára, hogy navigációs szándékkal bongésszenek és fedezzenek fel érdekes helyeket. Legutolsó témakör pedig a videoalkalmazások, amelyek lehetővé teszik a felhasználók számára a videók megtekintését streaming segítségével (közvetlen lejátszás internetről), miközben az autó parkol. Ezeknek az alkalmazásoknak a fő célja a streaming videók megjelenítése.

Az Apple CarPlay is hasonló alapelvekre épül, hogy átveszi azokat a dolgokat, amelyeket iPhone eszközünkkel szeretnénk csinálni vezetés közben, és közvetlenül az autó beépített kijelzőjére helyezi őket. A felhasználók az App Store áruházból töltik le a CarPlay alkalmazásokat, és más alkalmazásokhoz hasonlóan használják őket iPhone-on. Ha egy ilyen alkalmazással rendelkező iPhone készülék CarPlay funkcionalitásra képes járműhöz csatlakozik, az alkalmazás ikonja megjelenik a CarPlay kezdőképernyőjén. Az applikációk nem különálló alkalmazások, a CarPlay támogatást egy meglévő alkalmazáshoz kell hozzáadni. Az autón megjelenő alkalmazásokat úgy terveztek, hogy úgy nézzenek ki és működjenek, mint az általunk megírt applikációk iPhone-on, de a felhasználói felület elemei hasonlóak a beépített CarPlay alkalmazásokhoz. Az alkalmazás a keretrendszer használja a felhasználói felület elemeinek bemutatására a felhasználó számára. Az iOS kezeli azok megjelenítését és menedzseli az autóval való interfést. Az alkalmazásnak nem kell foglalkoznia a felhasználói felület elemeinek elrendezésével a különböző képernyőfelbontásokhoz, és nem kell támogatnia a beviteli hardvereket, például érintőképernyőket, gombokat vagy érintőpadokat. A CarPlay alkalmazásoknak meg kell felelniük a *CarPlay Entitlement Appendix* (CarPlay Jogsultsági Mellékét) meghatározott alapvető követelményeknek, és követniük kell az alkalmazásokra vonatkozó irányelveit.

Az alkalmazásjogsultság igénylésére el az Apple weboldalán van lehetőség. Itt meg kell adnunk információkat az alkalmazásról, beleértve a kért jogsultság típusát is. Ezenkívül el kell fogadnunk korábban említett a CarPlay jogsultsági mellékletet. Ezek után az Apple felülvizsgálja kérelmünk és ha alkalmazásunk megfelel a CarPlay alkalmazás feltételeinek, az Apple jogosultságot rendel az Apple Developer fiókunkhoz, majd értesítenek minket az eredményről. minden alkalmazásnak be kell tartani bizonyos előre lefektetett követelményeket. A CarPlay alkalmazást elsősorban úgy kell megtervezni, hogy a megadott funkciót biztosítsa a felhasználó számára (például az audioalkalmazásokat elsősorban hanglejátszási szolgáltatások nyújtására, a parkolóalkalmazásokat elsősorban parkolási szolgáltatások nyújtására kell tervezni stb.). Soha nem szabad utasítani a felhasználókat, hogy vegyék fel iPhone-jukat egy feladat végrehajtásához. Ha valamelyen hiba történt, például egy kötelező bejelentkezés, csak értesíthetjük a felhasználókat a követel-

ményről, hogy biztonságosan intézkedhessenek. A felhasználói üzenetek azonban nem tartalmazhatnak olyan megfogalmazást, amely arra kéri a felhasználókat, hogy manipulálják iPhone-jukat. minden CarPlay felhasználói folyamatnak lehetővé kell tennie az iPhone-nal való interakció nélkül. minden CarPlay felhasználói folyamatnak értelmesnek kell lennie ahhoz, hogy vezetés közben használható legyen. Nem adhatunk meg olyan funkciókat a CarPlayben, amelyek nem kapcsolódnak az elsődleges feladathoz (pl. nem kapcsolódó beállítások, karbantartási funkciók stb.). Nem érhető el játék vagy közösségi hálózat az autón keresztül. Soha nem jelenítjük meg az üzenetek, szöveges üzenetek vagy e-mailek tartalmát a CarPlay képernyőjén. A sablonokat rendeltetésszerűen kell használni, és csak a megadott információtípusokkal töltetjük fel a sablonokat (pl. listasablont kell használni a lista megjelenítéséhez a kijelöléshez, az albumborítót a *now playing* (most játszott) képernyőn kell használni az albumborító megjelenítéséhez stb.). minden hangos interakciót a SiriKit segítségével kell kezelni, azonban a navigációs alkalmazások kivételt képeznek.

	Audio	Communication	Navigation	Driving task, EV charging, fueling, parking, and quick food ordering
Action Sheet		●	●	●
Alert	●	●	●	●
Grid	●	●	●	●
List	●	●	●	●
Tab bar	●	●	●	●
Information		●	●	●
Point of Interest				●
Now Playing	●			
Contact		●	●	
Map			●	
Search			●	
Voice control			●	

3.2. ábra. A CarPlay alkalmazásokhoz elérhető sablonok megjelölve engedélyezett téma szerint.

A CarPlay felületen a képernyők megjelenítésére a már korábban említett sablonokon keresztül történik. Az elérhető sablonokról és azok korlátairól a 3.2. ábrán láthatunk táblázatot. Ahogy a képen is láthatjuk, az elérhető opciók a hangot lejátszó applikáció, kommunikációs alkalmazás, navigáció, vezetési feladat, elektromos és hagyományos autó töltés és tankolás illetve ételrendelés. Az általam választott audió alkalmazásban lehet a legkevesebb féle sablont felhasználni, amelyek az Alert (értesítés), a Grid (rács), a List (lista), a Tab bar (menüsáv) és a Now Playing (jelenleg játszott) sablonok.

3.1. Felhasznált technológiák

Ebben a fejezetben az alkalmazás készítése során felhasznált külső függőségeket, könyvtárakat, szolgáltatásokat és technológiákat mutatom be.

3.1.1. Alamofire

Az *Alamofire* az egyik legnépszerűbb és legszélesebb körben használt Swift hálózati könyvtár. Az Apple *Foundation* [4] hálózati stackjére épülve elegáns API-t biztosít a hálózati kérésekhez. A GitHubon [42] több mint harmincezer csillaggal ez az egyik legjobban értékelt Swift adattár.

3.1.2. API

A mobilon meghatározott adatokat egy alkalmazásprogramozási felületen (*Application Programming Interface*) keresztül kérem le. Ez egy olyan interfész, amellyel interakciókat lehet végrehajtani. Az API leírása határozza meg a lehetséges hívásokat, hogyan hozzuk létre ezeket, az kommunikációra használt információknak a formátumát. Mivel az alkalmazások csak az API interfészétől függnek ezért nem szükséges a belső működését ismerni a programrendszernek, illetve ez a rendszer lecserélhető anélkül, hogy ez hatással lenne az interfész használó programokra. Az alkalmazás által használt API a podcastek megjelenítésére *Listen Notes* [39], amely nagy mennyiségű és fajtájú podcastek elérése ad lehetőséget, illetve az általam írt backend szolgáltatás.

3.1.3. Bundler

Bundler egy olyan függőségkezelő rendszer, amely a Ruby projekteket, hogy ugyanabban a kondícióban futhassanak különböző eszközökön konziszensen. Ezt úgy teszi, hogy listát képez a különböző Gem-ekről és azoknak a verziókról, amik szükségesek, hogy a projekt helyesen fusson. Gem-eknek nevezzük a különböző Ruby külső függőségeket. Ilyen Gem például a SwiftGen, SwiftLint vagy a CocoaPods is. Amikor a fejlesztő elkezdi a projektet megalkotni, akkor a Bundlert használva feltelepíti a projekthez szükséges függőségeket. Ezután, hogyha egy másik fejlesztő szintén a projektre kerül, akkor az ő eszközén is ugyanaz a verzió fog szerepelni, így elhárítható a különböző verziókból eredő *"Miért nem működik az én gépemen?"* kérdés és *"Fogalmam sincs az enyémen minden jó."* válasz páros a fejlesztők között. Továbbá arra is használható, hogy felfrissítsünk vele egy Gem kollekciót. A frissítés során ügyel arra, hogy az egyesfüggőségek kompatibilisek maradjanak egymással. Ezt úgy kell elkövetni, hogy van egy Gem, ami már az 1.9-es verzióval tart. Azonban egy másik Gem függ ettől, viszont maximum csak az 1.8-as verzióig kompatibilis. Egy harmadik Gem szintén függ az elsőtől, de ez maximum csak az 1.7-es verzióig működik. Ebben az egyszerű esetben akkor az elsőből az 1.7 lesz használva és így mindenki úgymond 'boldog', azonban ettől sokkal komplexebb helyzetek is előfordulhatnak, ha több Gem-ről van szó és mindegyiknek van alsó és felső határa is a verziók megkötésében.

3.1.4. CocoaPods

A CocoaPods a Swift és az Objective-C Cocoa projekteket népszerű függőségi menedzsere. A CocoaPods webhely szerint több ezer könyvtár és több millió alkalmazás használja. De mi is az a függőségi menedzser, és miért van szüksége rá? A függőség kezelő segítségével egyszerűen hozzáadhatjuk, eltávolíthatjuk, frissíthetjük és kezelhetjük az alkalmazás által használt, harmadik féltől származó függőségeket. Egy másik hasonló függőségkezelő rendszer a Swift Package Manager. Például ahelyett, hogy újra fentáblánk a saját hálózati könyvtárát, egyszerűen behúzhatjuk az Alamofire-t egy függőségi kezelő segítségével. Megadhatjuk a használandó pontos verziót vagy az elfogadható verziók tartományát. Ez azt jelenti, hogy még ha az Alamofire frissítést is kap olyan módosításokkal, amelyek nem kompatibilisek visszafelé, az alkalmazás továbbra is használhatja a régebbi verziót, amíg készen nem áll a frissítésre.

3.1.5. Exposed

A szolgáltatás, amellyel a mobiltelefonok kommunikálnak csak egy közvetítő szerepet játszik, mert az általa visszaadott adatok egy adatbázisból kerülnek továbbításra. Az adatbázissal való kommunikáció megkönnyítésére lett létrehozva az *Exposed* könyvtár [32], amellyel kényelmesen meg lehet valósítani az adatoknak a lekérdezését és manipulálását. A csomag kétféle minta szerint teszi ezt lehetővé. Az első az Adatelérési Objektum minta (*DAO – Data Access Object*) [11], amelynek a funkciója, hogy elrejti az alkalmazás elől az adatmanipulációt megvalósító specifikus műveleteket, ezáltal egy absztrakciós réteget húzva a kommunikáció fölé. A másik pedig a szakterület-specifikus nyelv (*DSL – Domain-Specific Language*) szerű, amely pedig az SQL utasításokhoz hasonló módon teszi lehetővé a kérések megfogalmazását. Az egyik fontos tulajdonsága a könyvtárnak, hogy a kommunikáció során a fejlesztőnek nem kell figyelni arra, hogy SQL injektálás¹ (*SQL injection*) lehetőségét megszüntesse, mert erről már eleve gondoskodik a csomag.

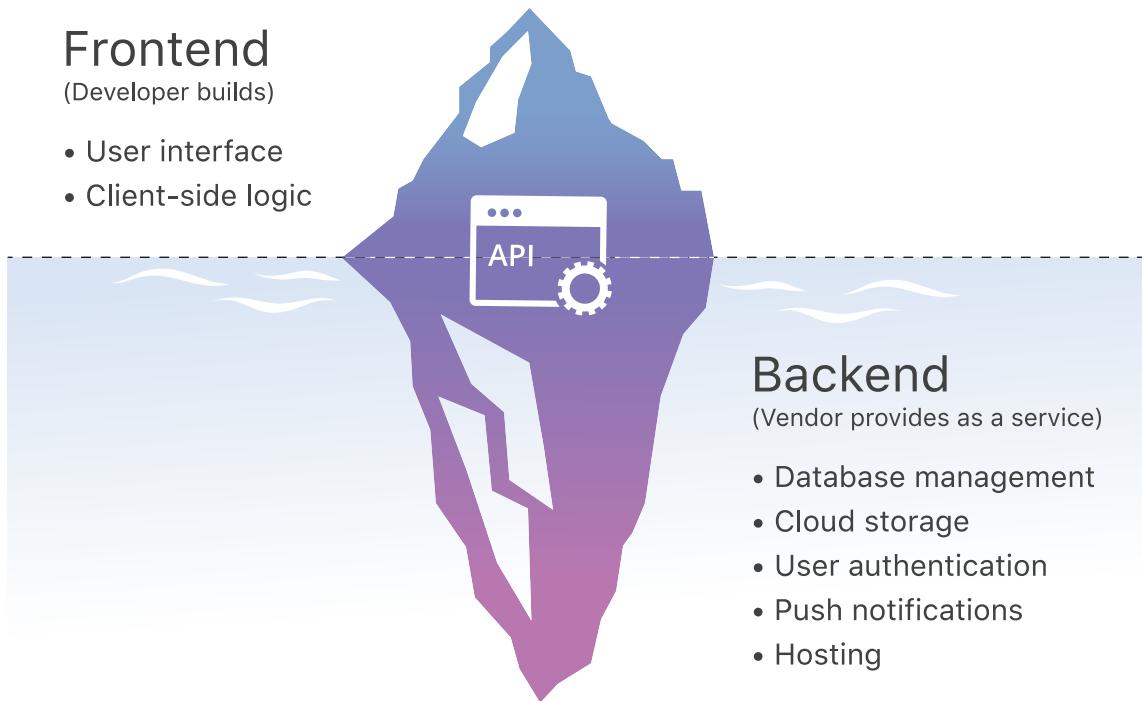
3.1.6. Factory

Programozás során gyakran előforduló eset, hogy egy objektum (kliens) függ egy másik objektumtól (szolgáltatás). Alaphelyzetben a kliens mondaná meg, hogy Ő melyik szolgáltatást használja és így Ő gondoskodik ennek az objektumnak a létrehozásáról. A *Dependency Injection* ezt fordítja meg, mivel ebben az esetben nem a kliens dönti el, hanem „megmondják” neki, hogy melyik, a specifikus szolgáltatást implementáló objektumot használja. Ezt a feladatot látja el az injektor, amely minden kliens osztályt összeköt a megfelelő szolgáltatással, illetve gondoskodik ezek létrehozásáról. A függőség befecsken-dezésnek a nagy előnye, hogy későbbi változtatás során könnyen válthatunk egy másik szolgáltatásra, illetve nem kell ismernie a szolgáltatást. iOS alkalmazások fejlesztése során az egyik ismert ilyen függőség injektálás funkciót ellátó keretrendszer a *Resolver* [38]. A használatához megadjuk előre, hogy milyen függőségeket szeretnénk beregisztrálni, majd az alkalmazás indulásakor a rendszer ezeket előregisztrálja induláskor. A feloldása az egyes függőségeknek viszont futási időben történik. Ennek az a hátránya, hogy ha valami nincs beregisztrálva, akkor az alkalmazás összeomlik, ami viszont nem egy előnyös dolog. Ennek a következtében a szokás, hogy Unit teszteket kell írni a regisztrációkra, hogy mindenkor megírtéjen a regisztrálás. Az előregisztrálás nem optimális dolog, azonban a hátránya viszonylag csekély. Míg az előzetes regisztráció teljesítménycsökkenéshez vezethet az alkalmazás indításakor, a gyakorlatban a folyamat általában gyors és nem észrevehető. Ezen felül továbbá a *Resolver* típuskövetkeztetést használ a regisztrált szolgáltatások dinamikus megkeresésére és visszaküldésére egy tárolóból. Ez a tulajdonsága, ami viszont kicsit problémásabb. Ha nem talál egyező típust, az alkalmazás összeomlásához vezethet, ha megpróbáljuk feloldani az adott típust, és nem található megfelelő regisztráció. A való életben ez nem igazán probléma, mivel az ilyen dolgokat ahogy mondtam észrevesszük és gyorsan kijavítjuk a Unit tesztek első futtatásakor vagy az alkalmazás második indításakor, hogy megnézzük, működik-e a legújabb funkcionk. Azonban ez szintén nem ideális, a legjobb ha már felépítéskor kiesne ez a hiba. Erre lett létrehozva a *Factory* [37], amely ugyanattól a személytől származik, mint a *Resolver*. A *Factory* fordítási időre biztonságos, mert egy adott típushoz léteznie kell egy regisztráció, különben a kód egyszerűen nem fordul le. Ennek következtében az alkalmazásomban az újonnan létrehozott megoldást választottam a függőségek injektálásához.

¹akkor történik, amikor az adatbázison belül olyan parancs fut le a tudtunk nélkül, amit nem szeretnénk, hogy végrehajtódjon, mert ez kárt tehet az adatbázisban

3.1.7. Firebase

Ha a felhasználónak több készüléke van, akkor szeretné, hogy minden telefonon szinkronizálva legyenek a kedvencek közé helyezett podcastek és az elmentett beállítások. Továbbá, ha valami történik a készülékkel, akkor az eddigi adatai elvesztésével a felhasználót is elveszíthetjük mert elképzelhető, hogy keresni fog egy másik alkalmazást, amely erre is képes, meggyőzően ezzel az eset újbóli bekövetkeztét. A jelen probléma megoldására használtam fel a Google Firebase [26] nevű szolgáltatását. Ez egy úgynevezett Backend-as-a-Service (BaaS, backend, mint szolgáltatás) egy felhőszolgáltatási modell, amelyben a fejlesztők kiszervezik a web- vagy mobilalkalmazások színpadai mögötti összes aspektusát, így csak a frontendet kell írniuk és karbantartniuk. A BaaS-szállítók előre megírt szoftvereket biztosítanak a szervereken végbenyomtató tevékenységekhez, például felhasználói hitelesítéshez, adatbázis-kezeléshez, távoli frissítéshez és push értesítésekhez (mobilalkalmazásokhoz), valamint felhőalapú tároláshoz és tárhelyszolgáltatáshoz.



3.3. ábra. Backend-as-a-Service struktúra bemutatása. Felül található a kliens oldal, lent pedig a szerver oldal.

Gondolunk egy alkalmazás fejlesztésére BaaS-szolgáltató használata nélkül, mint egy film rendezésére. A filmrendező feladata a forgatócsoportok, a világítás, a díszletépítés, a gardrób, a színészválogatás és a gyártási ütemterv felügyelete vagy irányítása, a filmben megjelenő jelenetek tényleges forgatása és rendezése mellett. Most képzelje el, ha létezne egy szolgálat, amely az összes kulisszák mögötti tevékenységet elvégezte volna, így a rendezőnek csak a jelenetet kellene irányítania és leforgatnia. Ez a BaaS ötlete: a szolgáltató gondoskodik a „fényekről” és a „kameráról” (vagy a szerveroldali² funkciókról), hogy a rendező (a fejlesztő) csak az „akcióra” koncentrálhasson – amit lát és tapasztal a felhasználó.

Ennek a segítségével könnyedén integrálni lehet meglévő szolgáltatásokat az alkalmazásba, amelyek közé tartozik például az analitika gyűjtése, a felhasználók hitelesítése és az

²Mindenre utal, ami az internetes kliens-szerver modellben a kliens helyett egy kiszolgálón van tárolva vagy azon történik.

adatok, fájlok tárolása a felhőben. A használatához regisztrálni kell egy Firebase projekt-hez az alkalmazást, majd az alkalmazás könyvtárába kell letölteni egy JSON kiterjesztésű fájlt, amely megadja az alkalmazásnak, hogy hol érheti el a felhőben a projektet. Ezután a Firebase könyvtárát használva az alkalmazáson belül tudok regisztrálni és beléptetni felhasználókat, akik egyedi azonosítókat kapnak. Az hitelesítésről a szolgáltatás gondoskodik, illetve ő tárolja el a felhasználóknak a belépéshoz szükséges adatait. A jelszó a folyamat során nincs eltárolva. A regisztráció során egy publikus kulcs segítségével a jelszóból generálódik egy új érték. Ez az érték egyedi és csak az eredetinek megadott jelszóból lehet újból generálni. A kulcs ismeretében sem lehet az értékből visszafejteni a jelszót, így ezt az értéket már nyugodtan el lehet tárolni mert, ha esetlegesen valaki jogosulatlanul hozzáfér ezekhez az értékekhez, a felhasználónak a jelszavát nem tudja megszerezni annak a felhasználásával. Bejelentkezés során a beírt jelszónak a kulcs segítségével generált értékére történik az egyezőségi vizsgálat az adatbázisban eltárolt értékkel.

A Firebase másik használati módja az alkalmazásban a felhasználói beállítások értékeinek, illetve a kedvencek közé rakott podcastek azonosítóinak az eltárolása lehetne a felhőben. A szakdolgozatom során ezt a megoldást használtam, azonban jelen dolgozatom részeként elkészítettem ezeknek a tárolására egy saját szolgáltatást.

3.1.8. JSON

A *JavaScript Object Notation* (JSON) egy olyan séma leíró, amely adattovábbításra lett teremtve. Előnye, hogy emberek által is könnyen olvasható. Az adatokat objektumokként reprezentáljuk, amelyek egy nyitó és záró kapcsos zárójel között helyezkednek el. Ezekben belül attribútumokat sorolunk fel és hozzá tartozó értékeket. Az attribútum nevét dupla idézőjelek közé kell tenni, majd ezt követően egy kettőspont jelzi, hogy az érték következik. Ezek lehetnek számok, szövegek, binárisok (boolean), tömbök, egyéb beágyazott objektumok és a null érték. Az elemeket egymástól vesszővel választjuk el. A tömböket szögletes zárójel jelzi. A szövegeket és a binárisokat szintén dupla idézőjellel kell körülvenni, azonban a számok köré nem kell tennünk semmit. Az alábbi példában látható egy egyszerű objektum, amely dolgozók tömbjét tartalmazza, ahol egy dolgozónak megtalálható a neve, életkora és az email címe. Az API felől az adatok JSON formátumban érkeznek, amelyeket beépített nyelvi elemekkel alakítok át osztályokká.

```
{
  "dolgozók": [
    {
      "név": "Anett",
      "életkor": 26,
      "email": "anett@example.com"
    },
    {
      "név": "Bertalan",
      "életkor": 24,
      "email": "berci@example.com"
    }
  ]
}
```

3.1. lista. Példa JSON fájl

3.1.9. Kotlin

Android alkalmazások megvalósítására két főbb nyelvet lehet használni. Az egyik a Java [40], amely már régebb óta használatos, a másik pedig a Kotlin [33], amely csak 2011-ben lett megalkotva. A fejlesztését a JetBrains [31] csapat végzi. Azonban a Google 2017-ben kiáltott a Kotlin mellett és felkapta. Ez odáig jutott, hogy manapság az új Android projektek nagy része már Kotlin nyelven készül, továbbá ez csak egy apróságnak tűnik, de az Android

Developer [23] fórumon a példakódoknál a Kotlin szerepel elsőnek és csak utána a Java jelezve a Google szerinti prioritást. Továbbá nagy segítség hogy a két nyelv kicserélhető, ezáltal könnyedén behúzható meglévő projektekbe. Az Android fejlesztés során megtetszett a Kotlin nyelv és ezáltal választottam ezt az elkészítendő háttérszolgáltatás nyelvének.

3.1.10. Ktor

A Ktor [34] egy keretrendszer internethoz csatlakoztatott alkalmazások - webalkalmazások, HTTP-szolgáltatások, mobil- és böngészőalkalmazások - egyszerű felépítéséhez. A modern csatlakoztatott alkalmazásoknak aszinkronnak kell lenniük, hogy a lehető legjobb élményt nyújthassák a felhasználók számára, a Kotlin-féle coroutine pedig fantasztikus lehetőségeket kínál ennek egyszerű és egyszerű megvalósításához. Könnyűsúlyú, mert lehetőséget nyújt csak azoknak a funkcionalitásoknak a konfigurálására, amelyekre az alkalmazásnak szüksége van. A backend megvalósításához használt keretrendszer választása során a legfőbb érv az volt a Ktor mellett, hogy egy egészen új technológiáról van szó amit ki-próbálhattam. Nem mellesleg az is előny, hogy Kotlin-ban tudom írni és ezen felül épít a nyelv elemeire, amikhez már hozzászoktam és megszerettem a korábbi Androidfejlesztési projektjeim során.

3.1.11. SQL

A strukturált lekérdezőnyelv (*Structured Query Language*) egy olyan szakterület- specifikus nyelv (*domain-specific language*), ami a relációs adatbázisoknak az adatmanipulációjára és relációs adatfolyam-kezelő rendszerek adatfolyam feldolgozására lett tervezve. Jellegzetessége, hogy a lekérdezéseket olvasva olvasmányosan értelmezhetőek.

3.1.12. SwiftGen

Mobilfejlesztőként úgy érezhetjük magunk, hogy naponta találkozunk varázslattal. Az általunk írt kódok olyan alkalmazásokká alakulnak, amelyeket az emberek a világ minden tájáról használhatnak. Az Apple által biztosított eszközök segítenek életre kelteni ezt a varázslatot, és megkönnyítik az életét. Ahogy tovább haladunk a szoftverfejlesztés felé, rájöhetünk, hogy van egy varázslat, amelyet nem szeret: a mágikus string (füzér).

A számítógépes programozásban a karakterlánc hagyományosan egy karaktersorozat, akár szó szerinti állandóként, akár valamilyen változóként. Ez utóbbi lehetővé teheti elemeinek mutációját és hosszának megváltoztatását, vagy rögzíthető (a létrehozás után). A karakterláncot általában adattípusnak tekintik, és gyakran bájtoból (vagy szavakból) álló tömb-adatstruktúraként valósítják meg, amely elemek sorozatát, jellemzően karaktereket tárol, valamilyen karakterkódolás használatával. A karakterlánc általánosabb tömböket vagy más sorozat- (vagy lista-) adattípusokat és -struktúrákat is jelölhet.

```
let color = UIColor(named: "green-apple")
self.title = "Welcome!"
```

3.2. lista. String példák

A típusbiztonság, az a koncepció, hogy a változók csak meghatározott típusúak lehetnek, védőkorlátokat biztosít a fejlesztőknek, amelyek biztonságban tartják programjaikat. A mágikus karakterláncok azonban nem biztonságos kódot vezetnek be ezekbe az alkalmazásokba. Mi az a mágikus string? Az iOS-fejlesztés során sokszor találkozhatunk ilyenekkel. Erre egy példa látható a 3.2

Mutatja a "green-apple" és a "Welcome!" szövegeket karakterláncokként írva közvetlenül a kódban. Nem könnyű azt állítani, hogy néha minden fejlesztő bűnösnek találta

magát ebben a gyakorlatban. Valójában az iOS fejlesztésben nincs sok választása. Alapból az Xcode nem ad módot ennek a gyakorlatnak az elkerülésére. Azok, akik már dolgoztak az Androidon, azon kaphatják magukat, hogy összerezzenek egy ilyen kóddal. Az Android fejlesztői környezetek olyan mechanizmussal rendelkeznek, amely az alkalmazás-erőforrásokat, például karakterláncokat, színeket, képeket és betűtípusokat típusbiztos változókká alakítja. Ennek számos előnye van, mint például: csökkenti az elírások kockázatát, megakadályozza az erőforrások szükségtelen megkettőzését, erőforrás-ellenőrzést biztosít a fordítási időben, segít a régi források megtisztításában. Szerencsére létezik a SwiftGen, egy kódgenerátor, amellyel megszabadulhatunk a mágikus karakterláncoktól az alkalmazásban. Nyílt forráskódú könyvtárként érhető el a GitHubon [30], és hozzáadhatjuk iOS- és macOS-projektjeinkhez, hogy típusbiztonságot és fordítási időbeli ellenőrzést biztosítson minden eszköznek. A meglévő karakterlánc forrásokból statikusan elérhető változókat generál, amelyeket a kód ból elérhetünk. Mivel a lokalizálható szövegforrásokból generálja ezeket, ezért automatikusan működni fog a többnyelvűség támogatása is az alkalmazásban, feltéve, hogyha a meglévő erőforrásoknak specifikáljuk a szövegét minden támogatott nyelven.

3.1.13. SwiftLint

Tegyük fel azt a példát, hogy egy számokból álló listában ki akarjuk szűrni a páratlanokat, hogy csak a párosok maradjanak. A 3.3. listán látható kódrészletben minden kettő verziót ugyanazt csinálja. A különbség a kettő között az, hogy az egyik closure³ blokkjában használva van a gyorsírást elősegítő változónév, vagyis a \$0, addig a másikban nincs és specifikálva van, hogy egy szám érkezik.

```
var evenNumbers
evenNumbers = [0, 1, 2, 3, 4, 5, 6].filter { $0 % 2 == 0 }
evenNumbers = [0, 1, 2, 3, 4, 5, 6].filter { number in
    number % 2 == 0
}
```

3.3. lista. Különböző megvalósítások ugyanarra a szűrésre

Az ehhez hasonló megbeszélések gyakoriak az iOS-fejlesztők körében, akik közös projekten dolgoznak. A különböző kódírási stílusok zavart okozhatnak, és megnehezíthetik a kód olvasását. Gyakori, hogy a fejlesztők követik és betartják a meghatározott kódstílusirányelveket a következetes kódstílus fenntartása érdekében. A Raywenderlich Swift kód stílus útmutatója [35] az egyik legnépszerűbb.

Azonban sok szabály lehet. Egymás kódjának áttekintése során nehéz lehet megjegyezni ezeket a szabályokat. Tehát van-e jobb módja annak ellenőrzésére és figyelmeztetésére, ha nem tartják be az irányelvezeteket? Igen, természetesen van! A SwiftLint egy Swift kódstílus-elemző eszköz, amely segít a kód stílushibáinak megjelölésében. Segíthet érvényesíteni ezeket a stílusokat azáltal, hogy a fordítás során hibásnak jelöli, vagy figyelmeztetéseket ad, ha a stílust nem követi a fejlesztő.

3.1.14. Swift Package Manager

A csomag fogalma nem újdonság. A csomagok segítségével könnyedén használhat harmadik féltől származó, nyílt forráskódú kódot. Ezenkívül megkönnyítik a kód felosztását újrafelhasználható, logikai darabokra, amelyeket könnyedén megoszthat projektjei között, vagy akár az egész világgal. A *Swift Package Manager* vagy a *SwiftPM* a *Swift 3.0* óta

³Önálló funkcionálisblokkok, amelyek átadhatók és felhasználhatók a kódban. A Swift zárasai hasonlóak a C és az Objective-C blokkjaihoz, valamint a többi programozási nyelv lambdáihoz.

létezik. Kezdetben csak szerveroldali vagy parancssori Swift projektekhez volt elérhető. A *Swift 5* és az Xcode 11 megjelenése óta a *SwiftPM* kompatibilis az iOS, macOS és tvOS build rendszerekkel az alkalmazások létrehozásához. Ez nagyszerű hír az alkalmazásfejlesztők számára, mert most már a kódbazisukon is szabadjára engedhetik erejét! A csomag egyszerűen Swift-forráskódfájlok és egy *Package.swift* nevű metaadat- vagy jegyzékfájl gyűjteménye, amely különféle tulajdonságokat határoz meg a csomaggal kapcsolatban, például a nevét és az esetleges kódfüggőségeket.

3.1.15. URI

Az *Uniform Resource Identifier* (egységes erőforrás azonosító) egy olyan karakterlánc, amely egy webes erőforrást azonosít. Ilyen például egy webcím (URL – *Uniform Resource Locator*, egységes erőforrás helymeghatározó).

3.1.16. YAML

A YAML egy emberi olvasásra alkalmas adatserializációs nyelv. Gyakran ebben a formátumban vannak megadva a konfigurációs fájlok és adat tárolásra és továbbításra is alkalmas. A YAML sok olyan kommunikációs alkalmazást céloz meg, mint az Extensible Markup Language (XML) [53], de minimális szintaxisa szándékosan eltér a szabványos általános jelölőnyelvtől (SMGL - Standard Generalized Markup Language) [52]. Egyaránt Python-stílusú behúzást használ a beágyazás jelzésére, és egy kompaktabb formátumot, amely a listákhoz [...], a map-eléshez pedig a ...-ot használja, így a JSON-fájlok érvényesek a YAML 1.2-es verziójában.[46] Az egyéni adattípusok megengedettek, de a YAML natív módon kódolja a skalárokat (például karakterláncokat, egész számokat és lebegőpontos értékeket), listákat és asszociatív tömböket (más néven térképeket, szótárakat vagy kivonatokat).

```
---
őzike: "szarvas, őnstény szarvas"
napsugár: 'egy csepp arany a napból'
pi: 3.14159
karácsony: igaz
francia tyúkok: 3
kacsák:
  - Donald
  - Tiki
  - Niki
  - Viki
karácsony ötödik napja:
  hívó madarak: négy
  francia tyúkok: 3
  üüaranygyrk: 5
fogoly:
  szám: 1
  helyszín: "körtefa"
őtekns galambok: ökett
```

3.4. lista. YAML példa

A YAML objektumra és azokban elkódolható adatokra az imént látott fájlban lehet látni példát. A fájl három kötőjellel kezdődik. Ezek a kötőjelek egy új YAML-dokumentum kezdetét jelzik. A YAML több dokumentumot is támogat, és a megfelelő fájlelemzők minden kötőjelhármast egy új objektum kezdeteként ismernek fel. Ezután láthatjuk azt a konstrukciót, amely egy tipikus YAML-dokumentum nagy részét alkotja: egy kulcs-érték párost. Az őzike egy kulcs, amely egy karakterláncra mutat: *szarvas, egy nőstény szarvas*. A YAML nem csak karakterlánc-értékeket támogat. A fájl hat kulcs-érték párral kezdődik. Négy különböző adattípussal rendelkeznek. Az őzike és a napsugár füzérek. A pi egy

lebegőpontos szám. A karácsony logikai érték. Francia tyúk egy egész szám. A karakterláncokat szimpla vagy dupla idézőjelbe zárhatjuk, vagy idézőjelek nélkül is megadhatjuk őket. A YAML az idézőjel nélküli számokat egész számként vagy lebegőpontosként ismeri fel. A hetedik elem egy tömb. A kacsák négy elemből állnak, mindegyiket nyitó kötőjel jelöli. A kacsák elemei két szóközzel lettek beljebb húzva, amivel jelölhetjük a YAML-ben a beágyazást. A szóközök száma fájlról fájlra változhat, de a tabulátor használata nem megengedett. Az alábbiakban megnézzük, hogyan működik a behúzás. Végül látjuk a karácsony ötödik napját, amely egy angol mondóból származik. Ebben további öt elem található, mindegyik behúzva. A karácsony ötödik napját egy szótárként tekinthetjük, amely két karakterláncot, két egész számot és egy másik szótárat tartalmaz. A YAML támogatja a kulcsértékek egymásba ágyazását és típusok keverését is.

4. fejezet

Részletes megvalósítás

A projekt elkészítésének kezdetén legelső lépésként el kellett döntenи, hogy milyen technológiai vermet szeretnék használni. A legfőbb célom az volt, hogy új technológiákat próbálhassak ki, ezáltal bővítve a meglévő repertoárom. A jelen fejezetben először bemutatom a választásaimat, majd jellemzem a projekt elkészítése során megcsinált munkám.

4.1. Natív iOS alkalmazás

Egy natív iOS alkalmazást kétféleképp lehet elkészíteni. A régi fajta módszer, hogy UIKit keretrendszeret használunk. Azonban a fejlesztés során a fiatalabb SwiftUI keretrendszerhez fordultam segítségért.

4.1.1. UIKit

A UIKit [9], más néven User Interface Kit (felhasználói felület csomag) elérhetővé tesz számunkra olyan alapvető objektumokat, amelyek szükségesek, hogy iOS és tvOS alkalmazásokat készítsünk. Az objektumokat felhasználva megjeleníthetjük a saját tartalmainkat a képernyőn, interakcióba léphetünk velük és a rendszerrel is kapcsolatba léphetünk. Az alkalmazások az alapvető viselkedésükben a UIKit-re támaszkodnak, amely számos lehetőséget kínál a viselkedés testreszabására.

A fejlesztés során a folyamatokat imperatív módon adhatjuk meg. A számítástechnikában ez egy olyan szoftverfejlesztési paradigmát¹ jelöl, amely során utasításokat használunk a program állapotának megváltoztatására. Ugyanúgy, ahogyan a természetes nyelvekben a felszólító mód parancsokat fejez ki, az imperatív program a számítógép által végrehajtandó parancsokból áll. Az imperatív programozás a program működésének lépésről lépésre történő leírására összpontosít, nem pedig a várt eredmények magas szintű jellemzésére.

Az alkalmazásokat az Xcode nevű fejlesztő környezetben lehet elkészíteni, amely sablon projekteket tartalmaz az applikációhoz. Ilyen alkalmazásnak a felépítése látható a 4.1. ábrán, amely az egynézetű sablon segítségével lett készítve. A sablon projektek minimális felhasználói felületet nyújtanak. A projektet becsomagolva és lefuttatva azonnal látható az eredmény iOS eszközön vagy szimulátoron. Becsomagolás során az Xcode összeállítja a forrásfájlokat és egy applikáció csomagot készít a projektről. Ez egy olyan rendezett könyvtár, amely tartalmazza a forráskódot és az alkalmazáshoz tartozó erőforrásokat. Ezek alatt értjük a képeket, storyboard² fájlokat, szöveges erőforrásokat és meta-információkat, amik a forráskódot kiegészítik.

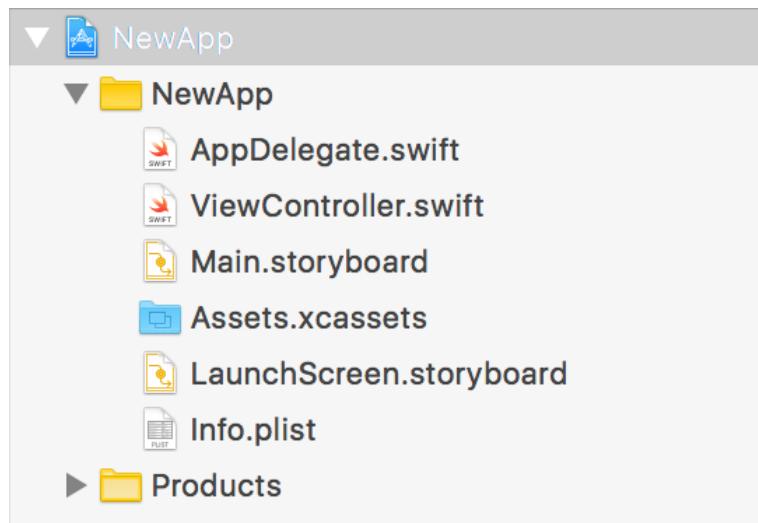
¹A számítógépes programok szerkezetének és elemeinek felépítésének stílusa

²Olyan grafikai rendszerező fájl, amelyben a képernyőket lehet definiálni

Kötelező erőforrások az alkalmazás ikonok és a kezdőképernyő storyboard fájlja. A rendszer megjeleníti az alkalmazás ikonját a kezdőképernyőn, a beállításokban és egyéb olyan helyeken, ahol a többi alkalmazástól megkülönböztetni szükséges az általunk készítettet. Mivel több különböző helyen van használva, ezért több eltérő méretben szükséges megadni az alkalmazás ikonját. Jellegzetesnek kell lennie, hogy a felhasználó könnyedén felismerje az alkalmazásunkat. Az Xcode legújabb fő verziójában, ami az Xcode 14, már nincs szükség több méretben megadni, hanem egy nagy felbontású képből automatikusan generál több megfelelő méretű és részletű képet. Továbbá itt nemcsak annyi történik, hogy lentebb állítja a felbontását a képnek. Mesterséges intelligencia segítségével megváltoztatja a képeket úgy, hogy azoknak a nagy méretű verzióban meglévő jellegzetességei megmaradjanak a kisebb pixelszámmal rendelkező ikonok során is. Így pillantásra ugyanúgy könnyen felismerhető marad az alkalmazásunk ikonja.

A kezdőképernyőt pedig az alkalmazás indításakor jeleníti meg a rendszer, amely egy úgynévezett *splash screen* (indítóképernyő). Amíg az alkalmazás elindul, addig a felhasználónak ez jelzi, hogy a valami töltődik. Amint viszont indulásra készen áll az applikációnk, akkor a rendszer kicseréli ezt a képernyőt az alkalmazásunk valódi felhasználói felületére.

A rendszer az alkalmazás konfigurációjáról és lehetőségeiről az information property list (Info.plist - információs tulajdonságlista) fájlból nyeri ki. A projekt sablonokkal egy előkonfigurált verzió jön létre ebből a fájlból, amit egész valószínűleg módosítanunk kell a projekt elkészítése során. Ilyen lehet például, hogyha az alkalmazásunknak szüksége van specifikus hardverre vagy rendszerszolgáltatásra, akkor azokat az információkat itt lehet definiálni. Ezekkel a beállításokkal kommunikálja az alkalmazásunk a rendszer felé, hogy milyen követelményekre van szüksége, ahhoz hogy az alkalmazásunk futni tudjon. Vegyük például egy navigációs applikációt, amelynek szüksége lehet GPS hardverre, hogy mutathasson részletes navigációt. Az App Store megakadályozza, hogy olyan eszközre töltsek le az applikációt, amely nem felel meg az itt megszabott követelményeknek.



4.1. ábra. A felépítése egy egyképernyős alkalmazásnak.

4.1.2. SwiftUI

A másik lehetséges módja a natív alkalmazásfejlesztésnek pedig a SwiftUI [8], amely a nevét onnan kapta, hogy a Swift programozási nyelv segítségével írjuk le a felhasználói felületeket. Feloldva Swift User Interface, vagyis Swift Felhasználói Felület. A SwiftUI nézeteket, vezérlőket és elrendezési struktúrákat biztosít az alkalmazás felhasználói felületének

deklarálásához. A keretrendszer eseménykezelőket nyújt továbbá az érintések, kézmozdulatok és más típusú beviterek alkalmazáshoz való eljuttatásához, valamint eszközöket ad az applikáció modelljeitől egészen a nézetekig és vezérlőkig, amelyekkel a felhasználók találkoznak és interakcióba lépnek.

Az alkalmazás felépítését az *App* protokoll segítségével lehet jellemzni, ahol jeleneket lehet megadni, amelyek tartalmazzák az alkalmazás felhasználói felület elemeit. Saját nézeteket hozhatunk létre, ha megfeleltetjük őket a *View* protokollnak, amelyeket összeállíthatunk beépített SwiftUI nézetekkel a szövegek, képek és egyéb formák megjelenítésével. Az elemek rendezésére rakások (stack), listák, illetve további eszközök állnak a rendelkezésünkre. Létrehozhatunk újakat és használhatunk beépített view modifier (nézet megváltoztató) függvényeket, amelyek segítségével a nézeteknek személyre szabható a megjelenése és interaktivitása. Hasznos szempont továbbá, hogy egyszerre tudjuk elkészíteni akár a macOS alkalmazást is és nem kell külön az AppKit keretrendszerhez nyúlnunk, mint UIKit esetében. A SwiftUI és UIKit keretrendszer nem mellesleg cserélgethető, amely azt jelenti, hogy lehetőségünk van akár ötvözni is a kettőt. Ez lehetőséget nyújt arra, hogy egy esetlegesen még nem megoldható dolog miatt a régi könyvtárhoz fordulunk, azonban ennek a megvalósítása nem mindenkor értetődő. Ahogy újabb és újabb verziókat adnak ki a keretrendszerből egyre kevésbé szükséges, hogy visszanyúljunk az UIKit könyvtárhoz és ami a régen megoldható azok lassan az újban is azok lesznek.

A legelső verziót az Apple 2019-ben adta ki, az iOS 13 verzióhoz. Azóta a negyedik fő frissítésnél jár, amely 2022. nyarán lett bemutatva, amely az iOS 16-os [6] operációs rendszereket futtató telefonokon érhetők csak el. Az újdonságok nem kompatibilisek visszafelé, ezért ha a legfrissebb lehetőségeket szeretnénk használni, akkor minimum verziót szükséges emelni vagy specifikálni kell a kód részlet körül, hogy az csak bizonyos verzión elérhető. Ilyenkor egy tartalék verziót szükséges létrehozni, hogy a funkcionális korábbi operációs rendszert futtató telefonokon is elérhető legyen.

A SwiftUI alkalmazásával a felhasználói felületet deklaratív módon írjuk le. A számítástechnikában a deklaratív programozás egy programozási paradigma, amely kifejezi a számítás logikáját anélkül, hogy leírná a vezérlési folyamatát. Sok nyelv, amely ezt a stílust alkalmazza, megpróbálja minimalizálni vagy kiküszöbölni a mellékhatásokat azáltal, hogy leírja, hogy a programnak mit kell elérnie a probléma tartományában, ahelyett, hogy a programozási nyelv primitíveinek sorozataként írja le, hogyan kell ezt megvalósítani (a hogyan kell nyelvi megvalósítás). Ez ellentétben áll az imperatív programozással, amely ahogyan korábban is említve volt az algoritmusokat explicit lépésekben valósítja meg. A deklaratív programozás gyakran tekinti a programokat egy formális logika elméleteinek, a számításokat pedig a logikai tér levezetéseinek. A deklaratív programozás nagymértékben leegyszerűsítheti a párhuzamos programok írását. A gyakori deklaratív nyelvek közé tartoznak az adatbázis-lekérdezési nyelvek (például SQL, XQuery), a reguláris kifejezések, a logikai programozás, a funkcionális programozás és a konfigurációkezelő rendszerek.

Az ilyen fajta leírás könnyen szemléltethető egy egyszerű példával. A szituáció az, hogy albumnak a számai szeretném megjeleníteni egymás alatt. Egy számról pedig a hozzá tartozó képet akarom bal oldalra helyezni, majd közvetlenül mellé a szám nevét és a szerzőt. A 4.1. listán látható kód részlettel írhatom le ezt könnyedén a SwiftUI segítségével. A body (törzs) attribútum felépítését értelmezve rájöhetsünk, hogy mi fog kirajzolásra kerülni a képernyőre. Legkülső elemként felrakunk egy listát, amelynek megadjuk hogy az albumnak a számain szeretnénk végigiterálni. Majd a kapcsos zárójelek között definiáljuk hogyan nézzen ki egy eleme a listánknak. Itt felveszünk egy *stack* (kazal) elemet, amely jelen esetben vízszintes (*horizontal*) a neve elején lévő H betű miatt. Ennek jellegzetessége, hogy balról jobbra pakolja fel egymás után a benne definiált nézeteket. Alapértelmezésben rak közéjük térközt, amely az éppen futó készülék alapján kerül kiszámításra vagyis kisebb képernyőjű telefonokon kevesebb, táblagépeken pedig nagyobb lesz például az értéke.

Ezt egyébként felülírhatjuk egy saját értékkel akár. Az alapértelmezett térköz mérete ismeretlen a számunkra, erről nem nyújtott semmiféle dokumentációt az Apple, hanem csak bízzuk ezt rájuk. Az az információ, hogy a nézeteket mindenképp balról jobbra helyezi el a képernyőn nem lesz mindig igaz. Az elhelyezés során figyelembe veszi, hogy a készüléknek milyen nyelv beállítása van. Azért fontos a megjelenítésnél, mert léteznek olyan nyelvek, ahol jobbról balra történik a szöveg megjelenítése és ezáltal az olvasás is. [1] Emiatt nem szabad úgy gondolkodni, hogy bal vagy jobb oldal a telefon képernyőjén, hanem *leading* (vezető) és *trailing* éleket különböztetünk meg. Jelen esetben azt mondjuk meg, hogy legyen az első elem egy kép, majd a második egy komplexebb nézet. A hátul lévő elem szintén egy stack objektum, amely azonban már függőleges (*vertical*), amit a V jelöl a szó kezdetén. A vertikális stack során az elemek fentről lefelé helyezkednek el mindenféléképp, nyelvtől függetlenül. Itt is automatikusan számítódik ki az elemek közti térköz, azonban itt a vezető élhez igazítva lesznek elhelyezve a szövegek az *alignment* (igazodás) beállítása miatt. Alapértelmezés szerint a SwiftUI minden közentre helyez, ha nem írunk felül semmit. A stack második paramétereként megadva az elemeket láthatjuk, ahogy fent egy szöveg fog megjelenni a szám címével, majd alatta pedig egy másik a szám előadójával. A második szövegen egyébként még az is specifikálva van egy nézet módosító függvényel, hogy a stílus a másodlagos legyen, amely alapján a színe nem fekete, hanem szürke lesz. Sötét üzemmódban szintén nem fehér, hanem a másodlagosságot jelző világosszürke.

```
import SwiftUI

struct AlbumDetail: View {
    var album: Album

    var body: some View {
        List(album.songs) { song in
            HStack {
                Image(album.cover)
                VStack(alignment: .leading) {
                    Text(song.title)
                    Text(song.artist.name)
                        .foregroundStyle(.secondary)
                }
            }
        }
    }
}
```

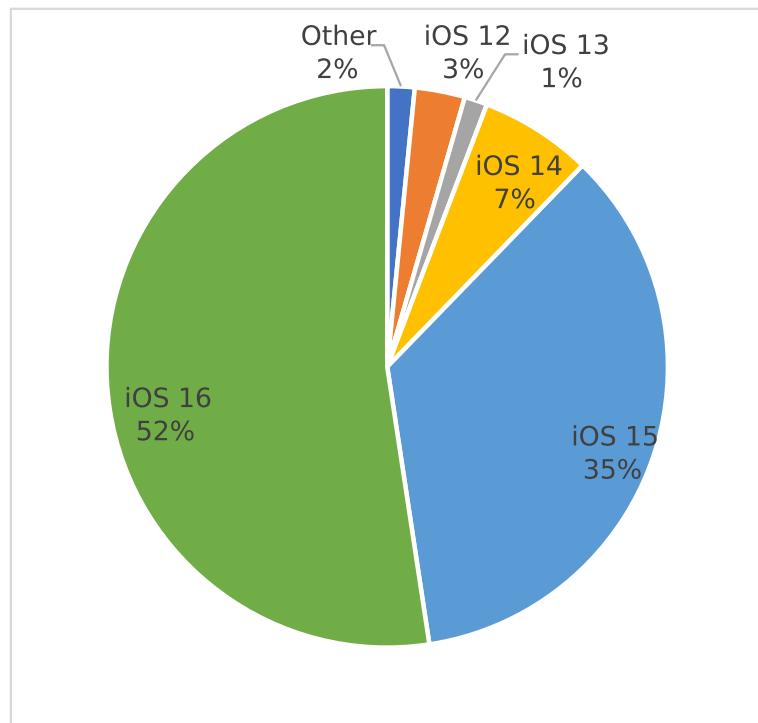
4.1. lista. Egyszerű SwiftUI nézet

4.1.3. iOS Verzió

Az alkalmazás megírásának kezdetekor a jelenleg elérhető maximális fő iOS verzió az iOS 15 [2] volt. Ennek következtében, hogy megismerkedhessék a legújabb innovációkkal a SwiftUI terén az alkalmazás minimum verziójának ezt választottam. Ahogy korábban is említettem sok múlik azon, hogy mire állítjuk be a legrégebbi operációs rendszer verziót, mert eleshetünk újdonságoktól, amik egyébként megkönyíthetik az fejlesztés során az életünk. A minimum operációs rendszer verzióinak a beállítása egyébként egyfajta ellentétet eredményez a fejlesztők és a menedzserek között, mert amíg az előbbi szeretné használni a legfrissebb és legújabb eszközöket, addig az utóbbi lent akarja tartani a verziószámot, hogy minél régebbi telefonokat használó felhasználók is tudják az alkalmazás szolgáltatásait igénybe venni.

Az alkalmazás fejlesztése során az Apple a szokásosan nyáron megtartott WWDC (WorldWide Developer Conference - világméretű fejlesztői konferencia) alatt bemutatta az

iOS 16-os verziótól elérhető SwiftUI 4 újdonságokat, majd a szokásos szeptemberi iPhone bemutatón pedig elérhetővé vált az iOS 16 is. Ennek a fő újdonságaként megváltozott a már régóta ugyanabban a stílusban megjelenített lezárt képernyő felülete. Valószínűsíthető, hogy ez okozta a nagy számú operációs rendszer váltást a felhasználók között. Egy 2022. november 25.-ei adat alapján az iPhone készülékeket használók 52%-a a legújabb iOS 16-os verziót használja, majd másodikként pedig 35%-al az iOS 15 szerepel. Az eloszlásról egy kördiagram található meg a 4.2.ábrán. Azt tekintve alapnak, hogy az alkalmazásom minimum az iOS 15-ös verziót futtató készüléken elérhetők azt jelenti, hogy esetleges App Store publikálás során az összes felhasználók 87%-át el tudom érni az applikációval, amely számomra egy elfogadható aránynak hangszik. A fejlesztés során mérlegeltem azt is, hogy megérné-e megemelni a legújabbra a követelményt, azonban nem tudtam volna kihasználni a frissen bemutatott újdonságokat, emiatt nem volt szükséges a váltás.



4.2. ábra. iOS verziók megoszlása. [43]

4.1.4. Verziókezelés

A projekt létrehozása során egy lokális git könyvtár is automatikusan létrehozásra került az Xcode által. Git a legelterjedtebb verziókezelő rendszer a világon, amely azt a feladatot látja el, hogy a fájlokon végbement változtatásokat követi. Lehetővé teszi, hogy több fejlesztő egyszerre dolgozzon egymással párhuzamosan. A legfőbb szempont, amely alapján nyer a többi verziókezelővel szemben a teljesítménye. Nem mellesleg a git-et használtam ezen a dokumentumon végbevitt változtatások követésére is.

Mindenek a gyökere egy könyvtár, amely a projektet tárolja. Ezt a könyvtárat tárolhatjuk lokálisan, vagy akár egy weboldalon is. Az általam használt tárolási hely a GitLab [19] nevű szolgáltatás, amely projektek tárolására specializálódott. Egy alkalmazás fejlesztése alatt a könyvtáron változtatásokat hozunk létre. Ezek lehetnek fájlok létrehozása, módosítása és törlése. A git ezeket a változásokat követi vagyis delta rejtjelezés segítségével történik a projekt állapotának az elmentése [12]. A fejlesztés során a projekten több mentési pont keletkezik, amiknek a neve commit (elkötelez). Ebben eltárolódhatjuk

a könyvtáron végbevitt változtatásokat. A commit történet tartalmazza az összes mentési pontot, vagyis a változtatásokat a projekt élettartama során. Ezek a pontok lehetővé teszik, hogy visszaállítsuk vagy előre mozgassuk a kódmező állapotát bármelyik commit pontra az előzményekből.

A git SHA-1³ (Secure Hash Algorithm - Biztonságos Hash Algoritmus) hasheket használ a mentési pontokra való hivatkozáshoz. Mindegyik érték egyedi és egy specifikus commit pontra mutat. Ezeket az értékeket felhasználva a git egy fa szerű struktúrát készít, amely során eltárolja, hogy az egyes pontoknak mely másik, vagy egyesítéskor mely másik kettő mentési pont volt a szülő.

A fájlok egy projektben több állapotban mennek keresztül. Az alapértelmezett a munkaállomás, amely a módosított fájlokat tartalmazza, azonban ezek még nem állnak készen a mentésre. A második az összpontosítási terület (staging), amelyhez hozzáadva változtatásokat előkészítjük őket a commit-hoz. Legvégül pedig a mentett fájlok, amelyek az összes olyan tartalmat magukba foglalják, amely már elmentésre került.

A branching (elágazás) a Git olyan funkciója, amely lehetővé teszi a fejlesztők számára, hogy az eredeti kód másolatán dolgozzanak a hibák kijavítása vagy új funkciók fejlesztése érdekében. Ha egy ágon dolgoznak, a fejlesztők addig nem változtatják a fő ágat, amíg nem akarják alkalmazni a változtatásokat. A fő ág általában a kód stabil verzióját jelenti, amelyet kiadtak vagy közzétesznek. Ezért el kell kerülni az új szolgáltatások és új kódok hozzáadását a fő ághoz, ha azok instabilok. Az elágazás izolált környezetet hoz létre az új funkciók kipróbálásához, és ha tetszenek, egyesíthetjük őket a fő ágba. Ha valami elromlik, törölhetjük az ágat, és a fő ág érintetlen marad. Az elágazás megkönnyíti az együttműködésen alapuló programozást, és lehetővé teszi, hogy mindenki egyidejűleg dolgozzon a kód saját részén.

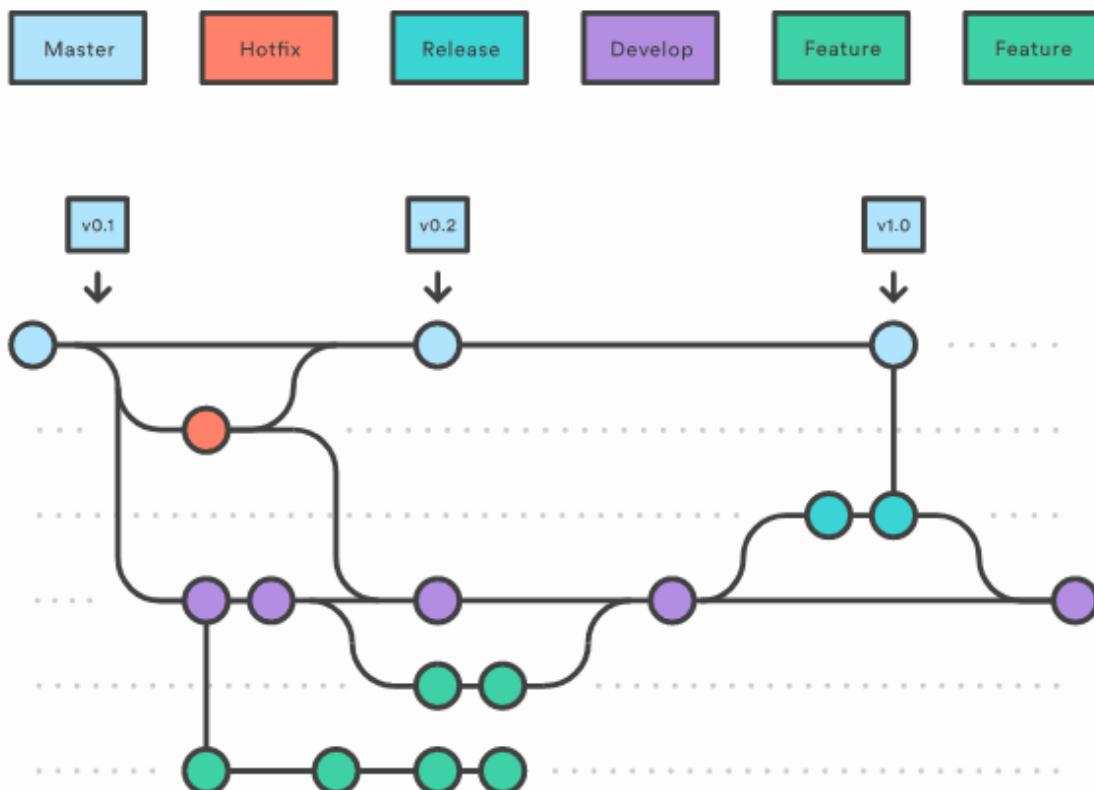
A *git merge* parancs lehetővé teszi az új funkciót vagy egy külön ágon lévő hibajavításon dolgozó fejlesztők számára, hogy a módosításokat a fő ággal egyesítsék, miután azok befejeződtek. A változtatások összevonása azt jelenti, hogy alkalmazzuk azokat a fő ágon is. Elég gyakran azonban konfliktusokkal lehet találkozni az egyesítés során. Például ütközés lép fel, ha valaki úgy dönt, hogy szerkeszti a fő ágat, miközben mi egy másik ágon dolgozunk. Ez a fajta ütközés azért fordul elő, mert egyesíteni szeretnénk a változtatásokat a fő ággal, amely most különbözik a kódmásolattól. A git gyakran fel tudja oldani automatikusan ezeket, azonban ha ugyanazt a kódrészét változtattuk mindenkiten, akkor manuálisan kell ezt megtenni, mert nem tudja melyik a helyes.

A *git fetch* és a *git pull* parancsok egyaránt a távoli adattáról történő változások lekérésére szolgálnak. A különbség az, hogy a *git fetch* csak a metaadatokat kéri le a távoli adattáról, de nem visz át semmit a helyi tárhelyre. Csak azt jelzi, hogy az utolsó lehívás óta történt-e bármilyen változás. Másrészt a *git pull* azt is ellenőrzi, hogy a távoli könyvtárban vannak-e új változások, és ezeket a változtatásokat a helyi tárolóba hozza. Tehát a *git pull* két dolgot hajt végre egyetlen parancssal: egy *git fetch*-et és egy *git merge*-t. A távoli könyvtárba a *git push* parancssal tudjuk a mi változtatásainkat feltölteni.

Amikor létrehozunk egy ágat, a git létrehozza a meglévő kód egy másolatát, amelyet továbbfejleszthetünk. Néha előfordulhat, hogy új változtatásokat kell beépítenie a fő ágból, hogy lépést tudjunk tartani az általános fejlődéssel. A rebase (újra alapozás) magában foglalja az új változtatások végrehajtását a fő ágból a fejlesztési ágba. Ez azt jelenti, hogy a git újrajátssza az új változtatásokat a fő ágból, és mentési pontokat hoz létre a szolgáltatási ág csúcsán. Ilyenkor újraírjuk a mentések történetét. A rebase funkcionalitásra nem feltétlen van szükség, ugyanezt el tudjuk érni a *merge* parancssal, azonban fejlesztők azért szokták ezt preferálni, mert tisztább történetet eredményez.

³A kriptográfiaban ez egy kriptográfiailag feltört, de még mindig széles körben használt hash függvény, amely bemenetet vesz, és egy 160 bites (20 bájtos) hash értéket állít elő, amelyet üzenetkivonatként ismerünk – általában 40 hexadecimális számjegyből.

Az általam használt git munkafolyam a *Feature Branch Workflow* (újdonság ág munkafolyam) volt. Egy szemléletes folyamatábra a 4.3. ábrán látható, amely alapján bemutatom a munkafolyam főbb elemeit. A kékkel jelölt elemek a fő ágat jelentik, ez a mindenkor kiadott verziókat tartalmazzák, amelyek megtalálhatóak az App Store-ban. Emelett fut lila színnel a fejlesztési ág. Itt található a legfrissebb verziója az alkalmazásnak. A fejlesztés során egy új funkció lefejlesztése egy *feature* ágon történik, ami zölddel van jelölve. A fejlesztési ágról indul, majd amikor a funkció elkészül, akkor a *develop* ágra visszavezetjük az új funkcionalitást. Az ábrán két új funkció is fejlesztésre kerül, amelyek közül az egyik már el is készült. A *develop* és a *main* ágak úgynevezett 'védett' ágak, ami azt jelenti, hogy oda közvetlenül nem lehet változtatást menteni. Ezeket csak más ágakkal lehet egyesíteni, hogy kiképyszerítsük a munkafolyamatot és megelőzzük az esetleges véletlen commit-okat ezekre az ágakra. Ha az épp kint lévő alkalmazásban valami hibát észlelünk, akkor erre egy javítási ágat hozunk létre a fő ágból. Itt létrehozzuk a szükséges változtatásokat a hiba elhárítására, majd ezt a fő és a fejlesztési ágba is behúzzuk, mert mindenkor helyen rosszul működik az egyes funkció. A fő ágra új verzió kiadása előtt egy *release* (kiengedés) ágat hozhatunk létre, amelyben előkészíthetjük az frissítést. Az általam fejlesztés során használt branch-ek a *master* volt, ahova nem került új verzió, mert nem lett publikálva az alkalmazás az App Store-ban, a *develop*, amelyben végigkövettem a fejlesztést és *feature* branchek voltak, amikben pedig a funkciókat fejlesztettem le.



4.3. ábra. Feature Branch Workflow bemutatása

4.1.5. ListenNotes API integráció

Ahogy azt korábban már említettem, az applikációban a podcastek forrásaként a ListenNotes nevű szolgáltatás van felhasználva. Itt regisztrálva magunk is tudunk hallgatni

podcasteket, azonban általam a másik lehetőség volt használva, ami pedig, hogy a saját könyvtárunkat elérhetővé teszik a fejlesztők felé is. Ennek a megvalósítására csak annyira volt szükség, hogy jelentkezni kellett a weboldalunkon ahol specifikáltam, hogy mire szeretném használni az Ő szolgáltatásaikat. Kikötésként megszabták, hogy nem használhatom arra, hogy ellopjam az adatokat, csak felhasználni lehet őket, vagyis nem kérhetek le minden szisztematikusan, hogy aztán azt elmentsem magamnak. Engedélyezés után pedig már tudtam is kéréseket intézni a végpontjaik felé. Alapértelmezetten az ingyenes hozzáférést kaptam meg, amit nem is változtattam meg a munkám során, mert nem volt a diplomateredményhez rá szükségem. Az előnyök, amiktől így elestem, hogy véges számú kérést intézhettek havonta csak a szolgáltatásuk felé, amely időintervallum alapján is korlátozva van. Az egy hónapban megengedett hívások száma 2500. Az intervallumos korlátozás azt jelenti, hogy nem áraszthatom el sok kérésekkel a rendszert vagy különben rövid időre letilt, amely körülbelül pár másodpercet jelent. Viszont ez az időtartam egy szoftver élettartamában nagyon hosszúnak számít. Továbbá nem kaptam meg bizonyos szűrési opciókat, mint például keresés során helyesírás ellenőrzés, amely olyan szavakat ajánl fel, amik kijavítva tartalmazzák az esetleges hibákat a keresési kulcsszavakban. Illetve egy másik végpontot sem érek el, amely a jelenlegi lekérdezés alapján hasonlókat ad vissza, ezzel segítve a felhasználót. Egy másik előny nagyon hasonló végpontot azonban elértek az ingyenes verzióval is, viszont a fizetős verzió sokkal szélesebb körben ad vissza eredményeket. Másik előny még, hogy kereséskor több eredményt ad vissza lekérdezéskor, kereshetünk RSS⁴, Apple Podcasts vagy Spotify url, azonosító alapján is.

Ahhoz, hogy kéréseket tudjak intézni a szerver felé az applikációban szükség volt a szolgáltatási rétegbe felvenni a megfelelő osztályokat, amelyeknek a feladata a kommunikáció megvalósítása. Ezeknek az elkészítése nem egy nehéz feladat, mert csak a meglévő dokumentációt kell értelmezni és az alapján elkészíteni a megfelelő függvényeket az egyes végpontokhoz és az adatokat reprezentáló osztályokat. Azonban ennek a manuális elkészítése nem valami izgalmas, és rendkívül favágó feladat. Nem beszélve arról, hogy esetlegesen hibázhatunk is az elkészítéskor, ami csak később derülhet ki. A szolgáltatáshoz elérhető egy Swift nyelvű könyvtár, amely a projektbe integrálható Swift Package Manager és CocoaPods segítségével is. Azonban ezek tartalmazzák az összes végpontot, ami elérhető a szerveren nekem viszont nincs szükségem mindegyikre, ahogy ezt korábban említettem a 2.2. szakaszban. Ennek következtében nem szerettem volna integrálni a teljes könyvtárat az alkalmazásomhoz. Szerencsére a teljes szolgáltatáshoz tartozik egy OpenAPI leíró a weboldalunkon, amelyhez léteznek generátorok.

Az OpenAPI-specifikáció, eredeti nevén Swagger-specifikáció egy olyan formátum, amely a RESTful webszolgáltatások leírására, előállítására, fogyasztására és megjelenítésére használható. Ez egy REST API specifikációs szabvány, amely meghatározza a szerkezetet és a szintaxist. Legfőbb jellemzője, hogy programozási nyelv független. Ez lehetővé teszi mind a számítógépek, mind az emberi felhasználók számára a szolgáltatási képességek azonosítását és megértését anélkül, hogy további dokumentációra, forráskódhoz való hozzáférésre vagy hálózati forgalom ellenőrzésére lenne szükség. Az OpenAPI eltávolítja a találhatósokat a szolgáltatások hívásakor, hasonlóan ahhoz, ahogyan az interfészleírások leegyszerűsítik az alacsonyabb szintű programozást. Az API specifikációt általában YAML (további információ 3.1.16. alszakasz) vagy JSON (lásd 3.1.8. alszakasz) nyelven

⁴Az RSS (*RDF Site Summary* vagy *Really Simple Syndication*) egy webes hírfolyam, amely lehetővé teszi a felhasználóknak és alkalmazásoknak, hogy szabványos, számítógéppel olvasható formátumban hozzáérjenek a webhelyek frissítéseihez. Az RSS-hírcsatornára való feliratkozás lehetővé teszi a felhasználó számára, hogy számos különböző webhelyet nyomon kövessen egyetlen hírgyűjtőben, amely folyamatosan figyeli a webhelyeket az új tartalom után, így a felhasználónak nem kell manuálisan ellenőriznie azokat. A hírgyűjtők (vagy "RSS-olvasók") beépíthetők egy böngészőbe, telepíthetők asztali számítógépre vagy mobileszközre.

írják. Jelen esetben az előbbi használtam fel, azonban bármelyiket választhattam volna. Egy ilyen fájl leírja az API teljes egészét mégpedig a létező végpontokat és azokon elérhető metódusokat, kérésekkel használt autentikációs módokat, a paramétereket, amelyek a kérés elküldéséhez szükségesek és a visszatérő objektumokat is, elérhetőséget a készítőhöz, használati feltételeket, licenceket és egyéb információkat.

Az OpenAPI legfontosabb tulajdonságai közé tartozik, hogy ez tekinthető az igazság egyetlen forrásának. Ennek következtében a háttérszolgáltatást és a mobilalkalmazást készítő fejlesztő is ez alapján készítse el a kommunikációhoz szükséges objektumokat, ezáltal elhárítva a hibás emberi kommunikáció miatt keletkezett hibákat, amelyek a termék hibás működését okoznák. Ezen felül továbbá lehetőség van, ahogy két bekezdéssel korábban említettem interfések és modellek generálására is. Ez azt teszi lehetővé, hogy a fejlesztő által ejthető hibák egy része már a szolgáltatás, vagy az applikáció felépítésekor elkapásra kerül. A generálás során az úgynevezett boilerplate⁵ kódok megírása automatikusan megtörténik, javítva ezzel a fejlesztői produktivitást, mert csökkenti a feleslegesen eltöltött időt, amit így a valódi problémák megoldására használhat fel.

Az alkalmazás készítésekor én is a generálás mellett döntöttem, amelyhez azonban szerkeszteni kellett a weboldalon elérhető OpenAPI leírót. Az így elkészült, csak a lényegi információt tartalmazó YAML fájl már használható volt bemenetként a generátorhoz. A szükséges fájlok elkészítéséhez az *OpenAPI Generator* [15] nevű szolgáltatást. Ez a parancssori program lehetővé teszi, hogy különböző konfigurációs beállításokkal létrehozzak forrásfájlokat, amelyeket a projektben felhasználhatok.

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

4.2. lista. Homebrew telepítése

A parancssori alkalmazást a Homebrew segítségével töltöttem le. Ez egy ingyenes és nyílt forráskódú szoftvercsomag-kezelő rendszer, amely leegyszerűsíti a szoftverek telepítését az Apple operációs rendszerére, a macOS-re és a Linuxra. A név azt az ötletet kívánja sugallni, hogy a felhasználó ízlésétől függően szoftvert kell készíteni a Mac-re. A Homebrew-t a könnyű használhatósága [28], valamint a parancssori felületbe való integrálása miatt javasolják az emberek [47]. A Homebrew a Software Freedom Conservancy non-profit projekt tagja, és teljes egészében nem fizetett önkéntesek működtetik [13]. A letöltésre és telepítésre a 4.3. listán található parancs futtatásával van lehetőség, feltéve ha a Homebrew már megtalálható a számítógépen. Ha a feltétel nem teljesül, akkor a 4.2. listán lévő parancs segítségével letöltésre és a megfelelő helyre másolásra kerül.

```
brew install openapi-generator
```

4.3. lista. OpenAPI generátor telepítése

A generátor megléte után létrehoztam a konfigurációs YAML fájlt, amelyben specifikálom, hogy milyen beállításokkal hozza létre a CocoaPods könyvtárat, amelyet később a többi Pod-hoz hasonló módon behúzok a projektbe. Az elérhető beállítások teljes listája a Swift nyelvhez megtalálható a weboldalukon [14]. A 4.4. listán látható konfigurációs fájlból specifikálom, hogy a hálózati hívások megvalósítására az Alamofire könyvtár (lásd 3.1.1. alszakasz) kerüljön használatra, az alapértelmezett Apple által írt URLSession helyett. A válaszok során az adatok Apple által írt Combine keretrendszer típusai segítségével kerüljenek visszaadásra. Egy másik lehetséges mód az újabban kiadott Concurrency (párhuzamosság) könyvtár használata, amely hasonló megvalósítás a többi nyelvben elérhető await megoldásokra (neve onnan származik, hogy aszinkron

⁵olyan kódrészletek, amelyek projektek során nem igényelnek kreativitást, csak egy séma alapján történő megírása történik

függvények létezhetnek, amiknek a válaszát be tudjuk várni, megállítva ezzel a függvény futását). Megadtam továbbá, hogy a projektstruktúrában hova kerüljenek a forrásfájlok. Végezetül pedig a Pod adatait specifikáltam, amelybe beletertőzik a szerző, a projekt neve, amely alapján később behavatkozható és a hely, ahonnan el lehet érni a könyvtárat. A konfigurációs fájlt egyébként nem szükségszerű külön fájlba írni, hanem megadható a parancs opcionálisan is, azonban a könnyebb karbantarthatóságot figyelembe véve ezt az opciót tartottam kézenfekvőbbnek.

```
library: "alamofire"
responseAs: "Combine"
swiftPackagePath: "ListenNotesClient/Classes/OpenAPIs"
podAuthors: "Martin Penzes"
podSource: "{ :git => 'git@github.com:Fncyy/ListenNotes-iOS-API-Client.git', :tag => s.version.
    to_s }"
projectName: "ListenNotesClient"
```

4.4. lista. Konfiguráció a swift5 generátorhoz

A konfiguráció megírását követően pedig egy parancs (lásd 4.5. lista) lefuttatásával elkészíti a megfelelő fájlokat és könyvtárstruktúrát, amely egy Pod elkészítéséhez szükségesek. A parancs során sorban megadom az API leíró fájlt, a használandó generátort, a kimenet helyét és az előbb bemutatott konfigurációs fájlt. Ezt követően az elkészült könyvtárat a konfigurációs fájlban specifikált GitHub adattárba feltöltöm, aminek a helyét a CocoaPods által használt Pod fájlban specifikálni kell. Innentől kezdve használhatóak a hálózati hívásokért felelős osztályok, amelyek tartalmazzák a különböző címkekkel ellátott végpontokat különálló API osztályokban a könnyebb kezelhetőség szempontjából. Ez azt jelenti, hogy ha a végpontok csoportosítva vannak címkevel ellátva, amely a ListenNotes API esetében Search (keresés) és Directory (könyvtár) neveket viselik, akkor az egyes végpontok szintén szeparálva lesznek SearchAPI és DirectoryAPI osztályokba.

```
openapi-generator generate \
-i ./listennotes-openapi.yaml \
-g swift5 \
-o ./listennotes-api-client-ios/ \
-c ./ios-config.yml
```

4.5. lista. Terminál parancs generálásra

Az kommunikációs réteg elkészülte után a projektbe behúzás következett. A *CocoaPods* telepítését a *Bundler* csomagkezelőre (lásd 3.1.3. alszakasz) szerettem volna bízni, hogy a legjobb gyakorlatokat kövessem. Ennek a haszna, ha abban nem is fog megnyilvánulni, hogy egy másik fejlesztővel kollaborálva a projekten nem fog összeakadni a különböző verziójú CocoaPods installáció, abban segítségemre lesz, hogy a hálózati réteget könnyedén integrálhatom az alkalmazásomba. A Bundler használatához létre kell hozni egy Gemfile nevezetű fájlt (lásd 4.6. lista), amely tartalmazza az egyes Gem függőségeket. Itt specifikálom, hogy honnan szedje le a dependenciákat, melyik Ruby verziót használja, ami jelen esetben legalább 3.0.0 és pár könyvtárat, amelyre szükségem van: a cocoapods, fastlane és az xcodeproj. A CocoaPods van használva ugyebár a Swift nyelvű függőségek integrálására. A fastlane segítségével az automatizálást oldottam meg. Az xcodeproj eszköz [16] pedig arra van használva, hogy az Xcode által készített fájlokat könnyen módosíthassuk. A megadott függőségeket a *bundle install* parancs kiadásával lehet letölteni az eszközre. Ilyenkor létrejön egy lock (zár) fájl *Gemfile.lock* néven. Ebben eltárolónak a letöltött függőségek, amelyet későbbi telepítési parancsok során felhasznál, hogy ugyanazok a verziók szerepeljenek minden időben és minden eszközön. A dependenciák felfrissítéséhez a *bundle update* parancs kiadásával van lehetőség. Mind a *Gemfile* és a *Gemfile.lock* fájlokat szükséges hozzáadni a verziókezelőhöz. Ahhoz, hogy a

Bundler által letöltött eszközöket használjuk a terminálban úgy kell kiadni a parancsokat, hogy az elejére rakjuk a *bundle exec* kiegészítést. Ez annyit takar hogy a függőségkezelőt megkérjük, hogy az általa menedzselt dependenciákat felhasználva futtassa le a kívánt parancsot.

```
source "https://rubygems.org"  
  
ruby '~> 3.0.0'  
  
gem "cocoapods"  
gem "fastlane"  
gem "xcodeproj"
```

4.6. lista. Gemfile tartalma

Az Xcode nem úgy működik, mint a többi fejlesztőkörnyezet, hogy egy kiválasztott mappán belül minden fájlt a projekt részének tekint. Itt külön fájlok tartalmazzák az egyes projektekre hozzáadott forrásfájlok és mappákat, amely egy *.xcodeproj* kiterjesztésű fájlban találhatóak meg. Itt mindegyiknek egy külön azonosítót ad, majd az azonosítók alapján felépíti hogy melyik mappa csoporthoz mely azonosítójú fájlokat vagy mappákat tartalmazzák. A felsorolás egyben sorrendezés is, vagyis ilyen sorrendben jelenik meg az Xcode fejlesztőkörnyezet felületén nem pedig automatikusan alfabetikus sorrend szerint rendezve. Egy workspace (munkaterület) több ilyen projektet tartalmazhat. A munkaterületet egy *.xcworkspace* kiterjesztésű fájl adja meg, amelyben a felsorolás sorrendje szabja meg az Xcode-ban a megjeleníti sorrendet. A probléma ott jön elő, hogyha több fejlesztő dolgozik ugyanazon a projekten és változtatásokat hajtanak végre az *.xcodeproj* fájlokon. Ilyenkor munkák egyesítésekor a változtatások között konfliktusok keletkezhetnek, amelyeket nehéz feloldani. Léteznek olyan eszközök, amelyek bizonyos beállítások alapján automatikusan generálnak ilyen fájlokat, mint például az *XcodeGen* [36] vagy a *Tuist* [48], azonban én nem használtam őket, mert egy újabb függőséget jelentenének, amely eltörhet a projekt életciklusa során. Azonban a konfliktusok minimalizálását elősegítendő, használtam az *xcodeproj* eszközt, amelynek segítségével az azonosítók alapján alfabetikus sorrendbe rendezem a fájlban felsorolt forrásfájlokat. Ennek a megvalósítására egy *git hook*-ot (horog) hoztam létre, amely azt csinálja, hogy a commit során, mielőtt végbenenne a commit, lefuttatja a projektfájlon a parancsot, majd az így keletkezett változtatásokat a fájlon szintén hozzáadja a commit-hoz.

4.1.6. Automatizálás

A korábban említett fastlane [25] integrálása volt a következő lépés a fejlesztés során. Ez egyszerűen leírható, mint a legegyszerűbb módja az iOS- és Android-alkalmazások felépítésének és kiadásának automatizálásának egy olyan eszközcsomag segítségével, amely önállóan, vagy akár párhuzamosan működik. Használatával eseményekre kiváltható az alkalmazáshoz készült tesztek futtatása, új verziók kiadása TestFlight⁶ platformon, kezelése és megújítása az aláírási tanúsítványoknak, kiépítési profiloknak, képernyőképek készítése az alkalmazásról különböző nyelveken, méretű eszközökön és képernyőtípusokon.

Az előny abban rejlik a fastlane műveletek (action) kihasználásakor, hogy órákat, vagy akár napokat takaríthatunk meg, hiszen minden megtörténik bizonyos események hatására. Ezeknek a favágó feladatoknak az elvégzése helyett törődhetünk a hasznos funkciók fejlesztésével inkább. A Continuous Deployment (folyamatos kiadás) mantrája pontosan ezt takarja, iteratív és gyors kódolás és kiadás, minimális akadályok mellett.

⁶Online szolgáltatás mobilalkalmazások vezeték nélküli telepítéséhez és teszteléséhez, amely jelenleg az Apple Inc. tulajdonában áll, és csak az iOS fejlesztői programon belüli fejlesztőknek kínálják.

A fastlane egy Ruby-alapú konfigurációs fájl, úgynevezett Fastfile, amely sávok szerint csoportosítva különböző célokat és igényeket szolgál ki. Például van egy sáv az App Store-ban való üzembe helyezéshez, ahol konkrét feladatok, úgynevezett műveletek vannak, amelyeket el szeretnénk végezni. Az általam kitűzött feladatok a projekt felépítése (build), a tesztek futtatása és az új verzió feltöltése az App Store Connect portára volt. Az ezekhez szükséges sávokat külön-külön ruby fájlokban hoztam létre. Azonban ezt megelőzően inicializálni kellett a fastlane-t a 4.7. listán látható paranccsal. A futása végeztével létrejön egy *fastlane* nevű mappa, amely két fájlt tartalmaz. Az első az *Appfile*, amelyben teljes applikációra vonatkozó információkat lehet felvenni. Ebben szerepel az App Store Connect team ID (App Store Connect csapat azonosító) és a Developer Portal team ID (fejlesztői portál csapat azonosító). Ezekre az értékekre az App Store felületével való kommunikáció során lesz szükség. A másik fájl a *Fastfile*, amelyben pedig a saját sávjainkat tudjuk megadni.

```
bundle exec fastlane init
```

4.7. lista. Fastlane inicializálása

Ahogy az előbb is említettem három fájl tartalmazza az egyes folyamatoknak a sáv-jait, amelyek az *import* paranccsal vannak bevéve a fő Fastfile fájlba. Ebben a központi fájlban specifikáltam azt a részt, amelynek minden sáv előtt le kell futnia. Itt felvettem a kiszervezett változókat, amelyeket a sávokban használunk majd, így ezek egy központi helyen szerepelnek és leellenőriztem, hogy megtalálhatóak e az API kulcschoz tartozó értékek környezeti változóban. Ez a kulcs van használva, hogy az App Store Connect felületével kommunikáljak nem pedig felhasználónév és jelszó. Ha az adatok megtalálhatóak, akkor létrehozom belőlük a kulcsot, amit később a sávok használhatnak. Az előnye ennek a megoldásnak az, hogy a kulcs független az emberek től így nem törhet el ha valaki esetleg elmegy a projektről és az esetleges kétfaktoros autentikáció⁷ miatt sem kell aggódni.

Az első a felépítési folyamat, amely a *build.rb* fájlban kapott helyet. A sáv lefuttatja a build folyamatot, amely során keletkezett eredményt eltárol egy paraméterként megadott mappában. Ezt származtatott adatnak⁸ hívjuk (DerivedData), amely fájlokat későbbi sávok során felhasználók optimalizálási célból. Az alkalmazás felépítése időigényes folyamat, amely perceket vesz igénybe, ezért hasznos, ha ugyanazt az eredményt fel tudjuk használni az egymás után futó sávokban. Ha a hely ahova a származtatott eredmény mentésre kerül már tartalmaz egy korábbi felépített alkalmazást, akkor az új build futtatása előtt az itt lévő adatokat megtisztítom egy beépített DerivedData törlésére használható paranccsal. A következő fájl a *test.rb*, amely az automatikus tesztelést végzi. Itt visszaállítom a szimulátorokat, amelyeken a tesztelés zajlik az eredeti állapotba, hogy véletlenül se legyen nyoma az előző futtatásnak így ne befolyásolja semmi a jelenlegi tesztek kimenetelét. Ennek a sávnak opcióként megadható, hogy építse-e fel az alkalmazást újra vagy használja fel a már meglévő felépített applikációt a paraméterként megadott helyen. Ezek után lefuttatom a projekthez írt teszteket. A harmadik pedig a *testflight.rb* fájl, amelyben elkészítem

⁷A többtényezős hitelesítés (MFA; magában foglalja a kéttényezős hitelesítést vagy a 2FA-t hasonló kifejezésekkel együtt) egy olyan elektronikus hitelesítési módszer, amelyben a felhasználó csak két vagy több bizonyíték sikeres bemutatása után kap hozzáférést egy webhelyhez vagy alkalmazáshoz (vagy tényezők) egy hitelesítési mechanizmushoz: tudás (valami, amit csak a felhasználó tud), birtoklás (valamivel csak a felhasználó rendelkezik) és hozzátartozás (valami tulajdonság, ami csak a felhasználóra igaz). Az MFA megvédi a felhasználói adatokat – amelyek magukban foglalhatják a személyes azonosítást vagy a pénzügyi eszközököt – attól, hogy illetéktelen harmadik fél hozzáérjen hozzájuk, aki képes volt például egyetlen jelszót felfedezni.

⁸A DerivedData egy mappa, amely alapértelmezés szerint a /Library/Developer/Xcode/DerivedData mappában található. Ez az a hely, ahol az Xcode mindenféle közbenső összeállítási eredményt, generált indexeket stb. tárol.

a feltöltéshez szükséges fájlt, amit pedig utána felküldök az App Store Connect portálra, ahol TestFlight segítségével kiküldésre kerül az esetleges tesztelőknek.

Az egyes sávok elkészítése után következhetett az automatikus futtatás beállítása a GitLabCI [17] segítségét felhasználva. Itt megfelelő eseményekhez lehet hozzácsatolni feladatokat, amiket le szeretnénk hogy fussanak. Legelső lépésként szükség van egy YAML konfigurációs fájlnak a létrehozására, amely a gyökér mappában helyezkedik el és *.gitlab-ci.yml* nevet viseli. Ebben adjuk meg, hogy mikor és mik fussanak le. A parancsok futtatásához szükség van egy futtatási eszköz beállítására is, amely lehet akár a saját laptopunk is, amihez van elérhető leírás is hogyan kell megcsinálni. [18] Azonban én a munkahelyi GitLab példányt használtam, ahol már van beállítva több Mac mini is így az általam futtatott csővezetékek (pipeline) ezeken futottak. A projekthez választott szabály többrétegű volt amelynek a leírása látható a 4.8. listán. A szabályok egymás után értékelődnek ki és ha talál egy olyat, ami igazra értékelődik ki, akkor lefutnak az adott parancsok. Első előírás az volt, hogy ha a pipeline egy indított merge kérés⁹ vagy arra érkező új mentési pont hatására futna akkor engedélyezve van. A következő szabály, hogyha egy új mentési pont érkezik, amihez van nyitott egyesítési kérés, akkor ne fusson le sohasem. Ez azért szükséges, mert ilyenkor lefutna egy a branch-re érkező commit miatt és az egyesítési kérés hatására is. Azonban nem szeretném, hogy ugyanarra a mentési pontra kettőször fusson le ugyanaz a csővezeték, emiatt ilyenkor az előbbi letiltom ezzel a szabállyal és csak az utóbbi engedem tovább futni. Továbbá ha nincs nyitott merge request, akkor az első kettő sor hamisra fog kiértékelődni, azonban a harmadik igazra, ezáltal ilyenkor is futhat pipeline. Legutolsó sorban megadom azt, hogy ha egy mentési pontra címke érkezik, akkor szintén futhasson csővezeték.

```
workflow:
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    - if: $CI_COMMIT_BRANCH && $CI_OPEN_MERGE_REQUESTS
  when: never
    - if: $CI_COMMIT_BRANCH
    - if: $CI_COMMIT_TAG
  ...
  
```

4.8. lista. GitlabCI konfiguráció munkamenet szabály

A szabályok megadásával választ adtam arra, hogy mikor fussanak le az események hatására parancsok, azonban az még hiányzik, hogy mi az aminek futnia kell. Erre a fájl további részében ezt fektetem le. A következőkben elmondottak a 4.9. lista tartalmát magyarázzák el, amely a *.gitlab-ci.yml* fájlból kiragadott részlet. A futást megelőzően a korábban elkészített Bundler segítségét hívom. Ugyanis a futató környezet egy teljesen elkülönített hely, ezért nem lesz megtalálható semelyik szükséges függőség a további parancsok elvégzésére. Ennek következtében kiadásra kerül a *bundle install*, amellyel feltelepülnek a *Gemfile.lock* alapján a szükséges dependenciák, mint a fastlane és a CocoaPods. Majd a *bundle exec pod install --repo-update*, amellyel a megfelelő CocoaPods függőségek is telepítésre kerülnek. A korábban említett *bundle exec* azért kell, hogy a Bundler segítségével leszedett installációt használja. A *pod install* parancs az, amivel megadom a CocoaPods könyvtárnak, hogy fel szeretném telepíteni a *Podfile* alapján a függőségeket, amely szintén a könyvtáramban megtalálható *Podfile.lock* fájlban megadott verziók alapján kerül telepítésre. A megfelelő Pod függőségek verziózva vannak. Ezekről az elérhető

⁹Merge Request vagy más néven Pull Request egy olyan esemény, amely akkor keletkezik amikor merge előtt nyitunk egy kérést. Itt a többi fejlesztő javaslatokat és változtatási kéréseket tehet vagy jóváhagyhatják a kérést. Ha a többi fejlesztő jóváhagyta, akkor egy sima *git merge* fut le alkalmazva a végbevitt változtatásokat. Ennek a segítségével is lehet a kódminőséget fenntartani. Alapból a git erre nem képes, csak az arra épülő szolgáltatások, mint a GitHub, GitLab, Bitbucket

verziókról megtalálható egy lista, amely lokálisan is lementésre kerül az egyes telepítésekkor. A `--repo-update` (tároló frissítés) opció következtében a lokális tárhelyen megtalálható lista is frissítésre kerül, ami azért szükséges, mert ha a lokális listában lévő verziók között nincs benne a letölteni kívánt verziójú Pod, akkor a parancsnak sikertelen kimenetele lesz, ami miatt a teljes csővezeték is el fog hasalni. Ezt természetesen nem szeretném, mert akkor futtathatjuk újra, ezért ennek a megelőzésére a jelzéssel (flag) együtt futtatom a legelső alkalommal.

A redundancia csökkentésére létrehoztam sablonokat az egyes futtatási szakaszokhoz. Ezekben megadtam, hogy a termékek közé miket vegyen fel, amely jelen esetben a származtatott adatok könyvtára volt. Ezen felül még azt szabtam meg, hogy az eltárolt fájlokat csak egy bizonyos ideig tárolja el, utána törlésre kerüljenek. Az általam megadott időtartam egy óra volt. Majd a sablonok felhasználásával létrehoztam a build folyamatot, amely mindenkorán lefut és utasításként lefuttatja a hozzá tartozó fastlane felépítést elvégző sávot. A következő a unit tesztek futtatásával foglalkozó munkamenet, amely a fastlane tesztelést végző sávját futtatja a korábban említett opcióval, hogy kihagyásra kerüljön a build fázis. A teszteket csak branch-eken futtatom, azonban a develop és a main ágak kihagyásra kerülnek, mert az azok egyébként is védett ágak és az egyesítés során a kérésnél mindenkorán lefutnak a szükséges munkafolyamatok. A harmadik pedig a kiadási folyamat, aminél szintén lefut a megfelelő release fastlane sáv, amit korábban elkészítettem. Itt a megkötés, hogy egy bizonyos sémájúnak kell lennie annak, ami miatt kiváltódott a futás. Ilyen sémára csak a címkék fognak illeszkedni. A sávokat kipróbálva mindegyik sikeresen működött, a mentési pontok megjelölésével pedig a TestFlight portálra feltöltődött a legelső verzió. A jelen fázis szerint elkészült az automatizálás az alkalmazáshoz, amit mostmár fel tudtam használni a fejlesztés során.

```
...
before_script:
  - "bundle install"
  - "bundle exec pod install --repo-update"
build:
  extends: .build
  script:
    - "bundle exec fastlane build_for_testing"

unit-tests:
  extends: .test
  script:
    - "bundle exec fastlane unit_tests skip_build:true"
only:
  - branches
except:
  - develop
  - main

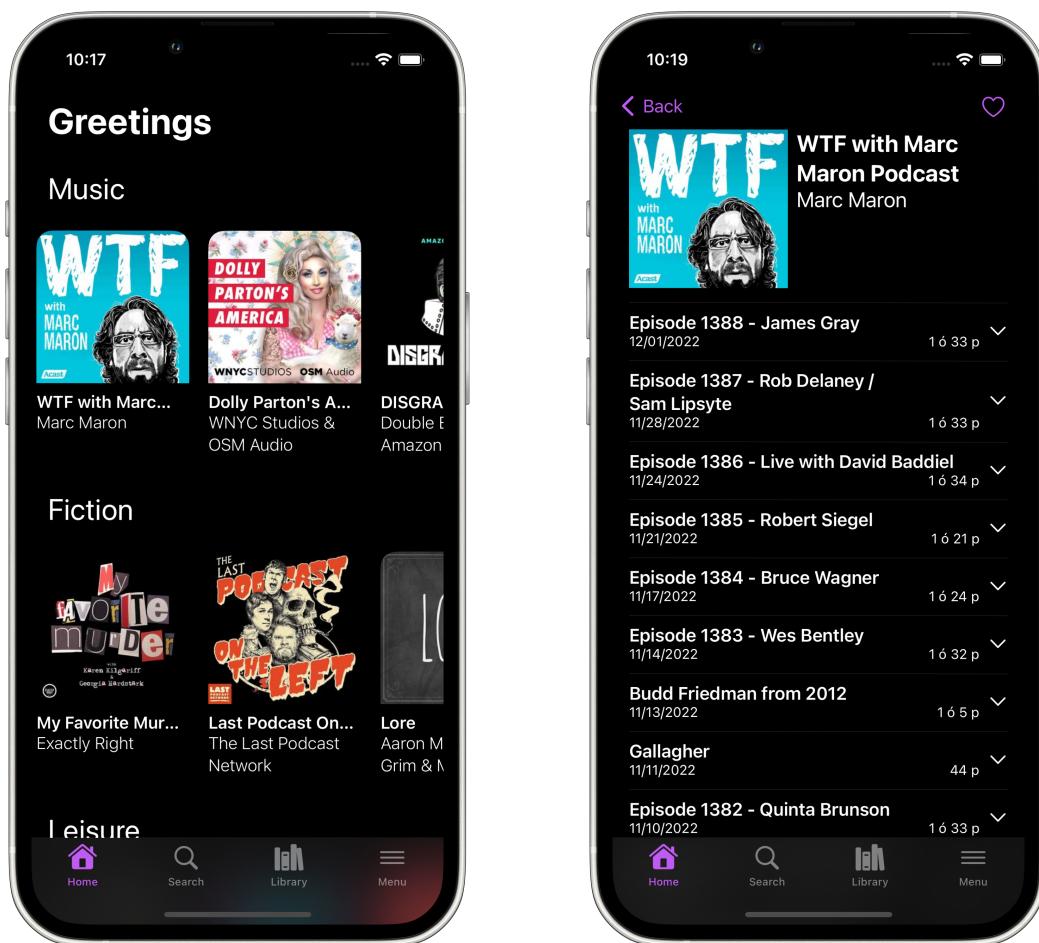
test-release:
  extends: .release
  script:
    - "bundle exec fastlane release_testflight"
only:
  - /^TEST-.*$/
```

4.9. lista. GitlabCI konfiguráció munkamenetek

4.1.7. Üzleti logika és képernyők

A ListenNotes szolgáltatással hálózati kommunikációt elősegítő korábbi fázisban generált osztályok azonban függnek egy külső dependenciától, mégpedig a ListenNotes szolgáltatásból. Ezen felül az visszaérkező válaszok szintén tartalmaznak olyan paramétereket, amely

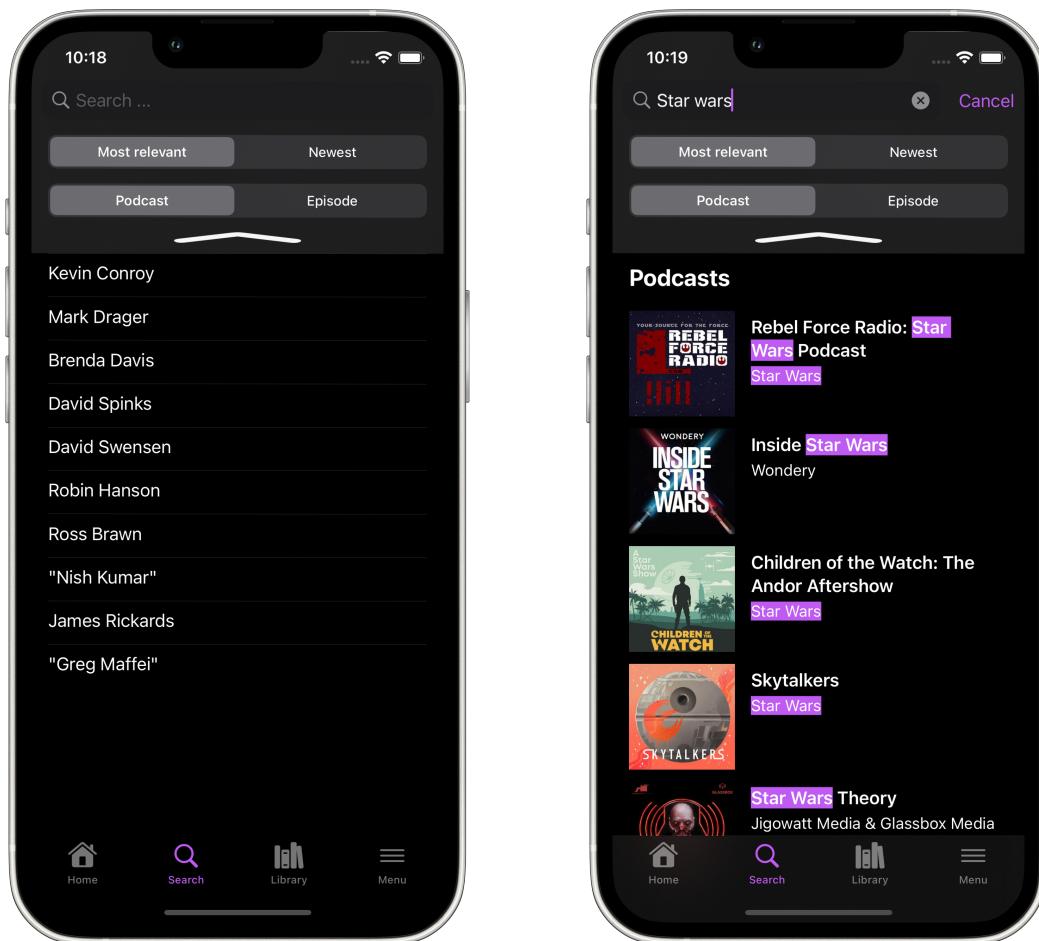
az alkalmazásom számára nem releváns vagy az ingyenes feliratkozás következtében elérhető és csak egy szöveget ad vissza, amelynek az értéke egy angol füzér: "Csak PRO és ENTERPRISE feliratkozóknak elérhető" mondattal. A számonra megfelelő modellekre való átalakítás szempontjából létrehoztam API-t elfedő kommunikációs osztályokat, amelyeket a feladatuk alapján csoportosítottam. Ezeknek a további szerepe, hogy ha esetlegesen a későbbiekben úgy döntök, hogy mégsem ezt a szolgáltatást szeretném használni, akkor csak itt történik a kódban változtatás, ezzel csökkentve a csatolást az alkalmazásom és a ListenNotes API között. A funkcionális alapján három csoportot hoztam létre: Search a kereséshez, Podcast értelemszerűen a podcast információk lekérésére és Options a lekérésekhez és a szűrésekhez használt paraméterek elérésére.



4.4. ábra. Kezdőképernyő (bal oldalon) és podcast részletező nézet (jobb oldalon)

A podcasteket elérő hálózati réteget felhasználva ezután megírhattam az üzleti logikát az alkalmazáshoz és a képernyőket. Mivel az általam megírt szolgáltatás hagytam a legutolsó feladatnak ezért menet közben létrehoztam egy másik OpenAPI leírót, amelyben megalkottam a saját eszközeim közti kommunikációra használt lehetséges végpontokat és az azokon küldött illetve fogadott modellekét. Az alkalmazás fő eleme egy tab bar (fülsáv), amelyen négy képernyő jelenik meg. A fülsáv jellegzetessége, hogy az alkalmazás futásakor minden a négy felhasználói felület megtalálható a memóriában, hogy könnyen válthasson közöttük a felhasználó. Az alkalmazás fő dizájnnyelvezet a Spotify applikáció-

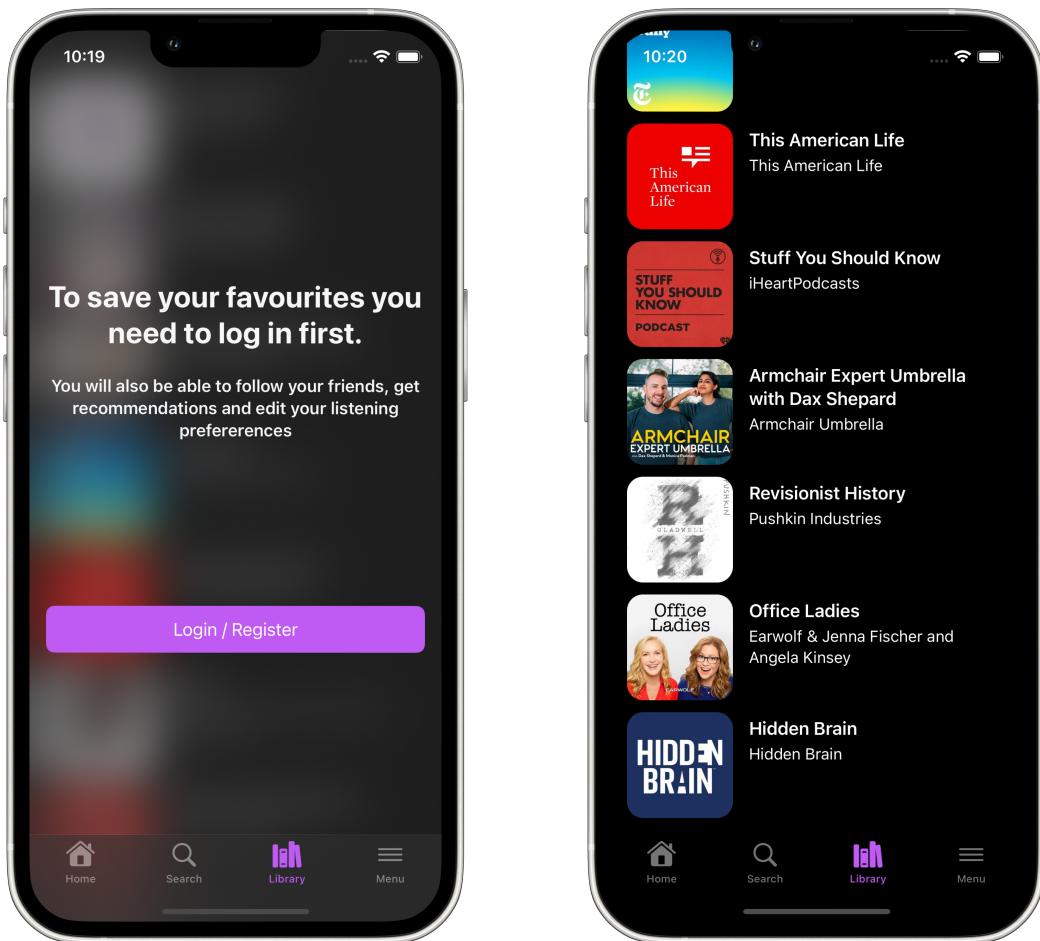
óját választottam. Ennek következtében szintén itt is alapértelmezetten sötét módban fut az alkalmazás. Az applikáció elindítása után a kezdőképernyőn találhatjuk magunk (lásd 4.4. ábra bal oldala). Itt egy fogadó üzenet található, hogy melegebb fogadtatás érje a felhasználót. Ezen felül egy többdimenziós listát böngészhetünk. A lista első dimenziója az egyes podcast típusok, amiket a felhasználó szeret illetve a barátok listája. A második dimenzió pedig olyan podcasteket tartalmaz, amelyek az adott műfajban felkapottak illetve az ismerőseik által mostanában hallgatott sorozatok. Egy elem a podcast címét, kiadóját és a hozzá tartozó képet tartalmazza.



4.5. ábra. Keresési képernyő alapállapota (bal oldalon) és kulcs-szóra keresve (jobb oldalon)

A listában egy elemet kijelölve a részletes nézetét tekinthetjük meg az adott podcastnek (lásd 4.4. ábra jobb oldala). A navigációhoz a felhasználók által már megszokhatott alapértelmezett iOS navigációt használtam. Itt megtaláljuk a hozzájuk tartozó epizódokat, azokról meta információkat, mint például a kiadás dátumát és a hosszát. Az egyes epizódokat lenyitva elolvashatjuk az epizódhoz tartozó leírást. Az elemek lenyitása során ügyeltem arra, hogy ne hirtelen történjen az elemek mozgása, hanem animáció is fussen le. Ennek a megvalósításához a SwiftUI egy *withAnimation* nevű függvényt ad, amely paramétereként egy closure-t vár. Ebben megadhatjuk a változtatásaink, amelyeknek köszönhetően ha újra kell rajzolni a képernyőt, akkor a SwiftUI best effort (legjobb erőfeszítés) jelleggel megpróbálja animálni az állapotváltozást. A képernyő jobb felső sarkában lévő szív gombbal

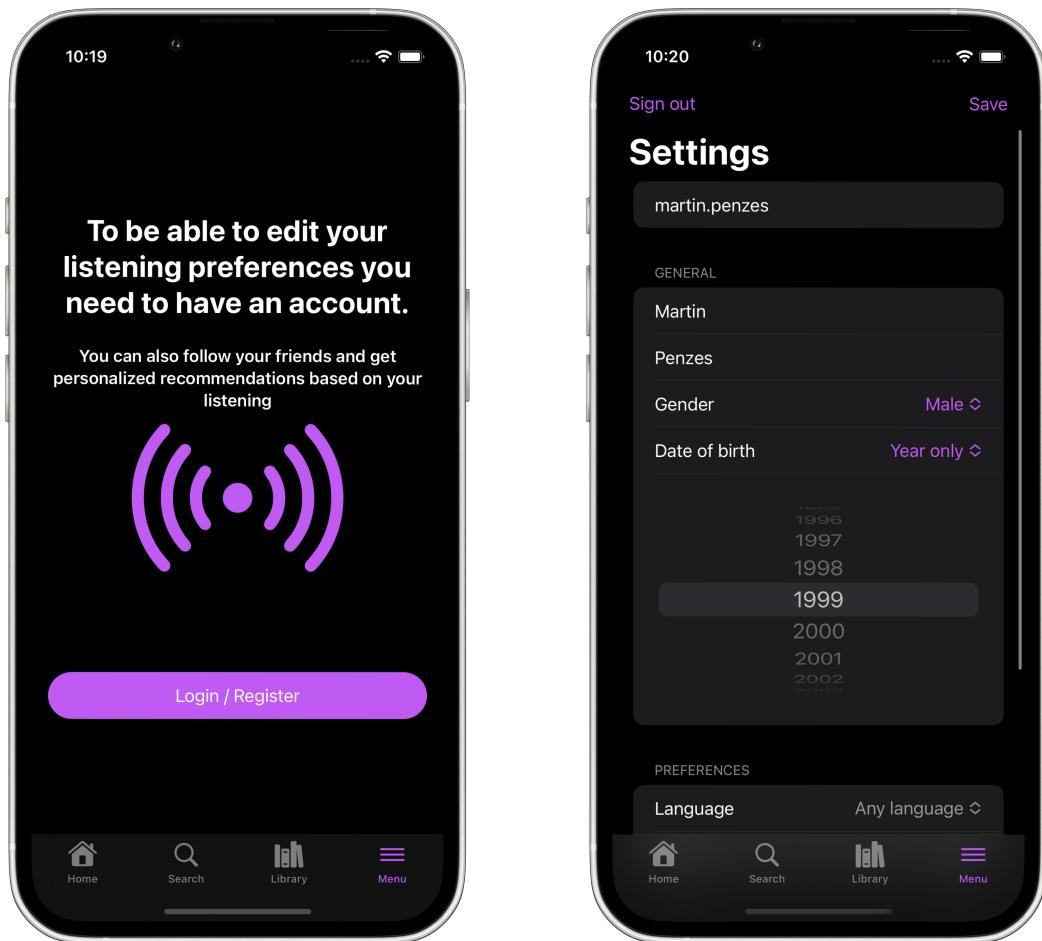
lehet a kiválasztott podcastet a kedvencek közé helyezni vagy esetleg onnan kivenni. Ehhez csak annyira volt szükség, hogy az eszköztárra felhelyeztem egy gombot, amit bekötöttem, hogy adja hozzá vagy törölje ha már benne van a mentett podcastek közt az elem. A már kedvencek közt létét egy elemnek az jelzi, hogy a szív teli lila és nem pedig üres. A SwiftUI segítségével ez is könnyen megoldható volt, hiszen csak egy logikai érték alapján kellett eldönteni, hogy a kép melyik erőforrást használja, a kitöltött vagy az üres szív ikont. Az ikont egyébként a rendszer ikonkönyvtárából vettet amelynek a neve SF Symbols. [7]



4.6. ábra. Könyvtár képernyő kijelentkezett (bal oldalon) és aktív belépett felhasználóval (jobb oldalon)

A navigációs sávval az alkalmazás fő képernyői között lehet váltani a megfelelő elemre kattintva. A háttér alapértelmezetten áttetsző, amelyen nem változtattam. Ennek az előnye, hogy nagyobb helyérzetet eredményez, illetve a felhasználó jobban követni tudja a képernyőn történő eseményeket, mert a már nem lényegi részen tartózkodó elemekről is kap egy csekély információt. A fő képernyők sorban a kezdőképernyő (Home), keresés (Search), könyvtár (Library) és a menü (Menu). A keresési képernyőn legalább a keresendő szöveget megadva lehet keresni. Alapállapotban jelenleg a többi felhasználó által gyakran keresett, trendi kulcsszavakat láthatunk (lásd 4.5. ábra bal oldala). A keresés során többféle paramétereket lehet megadni. Az egyik, hogy milyen szempont alapján történjen a szűrés. Itt az opciók, hogy a legrelevánsabbakat szeretné legfelül látni vagy a legújabbakat. A másik paraméter pedig, hogy a felhasználó podcastekre szeretné csak alkalmazni a keresést vagy

azoknak az epizódjaira. A preferált régió és a nyelv az appban globálisan beállított értékek alapján kerül kiolvasásra. Ezek meglétének hiányában az értéke *US – United States* (Amerikai Egyesült Államok) lesz. A keresőmezőbe szöveget beírva az applikáció automatikusan elküldi a kérést a szolgáltatás felé, nem kell a felhasználónak gombot megnyomnia. Azonban annak érdekében, hogy ne terheljem le a szolgáltatást figyelem a beviteli mező változásait és csak azután kerül kérés kiküldésre, ha a felhasználó utolsónak beütött karaktere óta már eltelt egy másodpercnyi időtartam. A keresési eredményeket egy listában jelenítem meg, ahol a keresett kulcsszóra az egyezési részeket a ListenNotes megjelöli. Ezt azonban át kellett alakítanom egy általam létrehozott kódolási stratégiával, mert az onnan visszajött HTML címkkel ellátott szavakat a SwiftUI nem tudja feldolgozni. A megoldás az volt, hogy létrehoztam egy saját markdown változót, amellyel körülvéve a szavakat át tudtam színezni a hátteret az alkalmazás fő színére, a lilára. A találatok közül egy elemre kattintva szintén a korábban már bemutatott részletező nézetre jutunk. A keresési eredményeket mutató képernyőről látható képernyőkép a 4.5. ábra jobb oldalán.



4.7. ábra. Menü képernyő, amikor a felhasználó kijelentkezett állapotban van (bal oldalon) és amikor be van lépve (jobb oldalon)

A harmadik fülre navigálva érhető el a könyvtár. Ebben találhatóak a korábban lementett podcastek a könnyű elérhetőség érdekében. Azonban ez a képernyő csak belépett felhasználóknak érhető el. A fül lényegi elemei egy ismertető szöveg, amelyben elmagya-

rázom, hogy miért nem érhető el ez a képernyő a felhasználónak és egy gomb, amellyel be tud lépni helyben vagy regisztrálni az alkalmazásba. Azonban ha csak ennyit tartalmazna a telefon, akkor az nem lenne valami barátságos az alkalmazásunkat használó személynek, illetve nem biztos, hogy ennyiból sikerül meggyőzni a regisztrációra. Ennek következtében kiegészítettem a szöveget, hogy információt nyújtsak arról is, mi más lehetőségek várnak még rá az applikációban ha belép vagy regisztrál. Illetve egy áttetsző panelen keresztül elhomályosítva megjelenítek egy kamu képernyőt, amelyen egy olyan listát láthat, amellyel szembetalálkozna ezen a képernyőn, feltéve ha vannak lementve podcastek és be van jelenítkezve. Ezzel sugallva még tovább, hogy mi vár rá még az alkalmazásban. A képernyőn elérhető listában az elemek ugyanúgy kattinthatóak és szintén a már többször bemutatott részletező nézetre juttatnak minket.

A legutolsó menüvel pedig a beállításokat lehet elérni, illetve a barátok listáját szerkeszteni. Alapértelmezésként ide is egy áttetsző hátteret akartam rakni a kijelentkezett felületre, azonban túlságosan nagy lett volna a képi zavar. Emiatt az egyszerű fekete háttér mellett döntöttem (lásd a 4.7. ábra bal oldala). Azonban, hogy ne legyen túlságosan üres a képernyő megjelenítettem az alkalmazás logóját. Itt is, ahogy az előző képernyőn is jellemztem apróbb betűvel, hogy mi az amit még a felhasználó felold a regisztrációval növelte a késztetést, hogy azt megtegye. A bejelentkezés után a beállítások menü látható, amely megtalálható a 4.7. ábra jobb oldalán. A barátlista mellett általános adatokat kérek el a, mint a neme, felhasználónak saját neve és felhasználóneve. Ezen felül lehetőséget adok arra is, hogy ne adja meg csak a születési évét, ha nem szeretné. Úgy ítétem meg, hogy nagyobb arányban kapnék extra születési információt azoktól a felhasználóktól, akik egyébként nem adták volna meg ezeket az adatokat, minthogy veszítsek el olyat aki teljesen kitöltötte volna ha nincs csak a két véglet opcióként. Továbbá megadhatja a preferenciáit, amely szerint régió és nyelvi szűrést szeretne alkalmazni keresés során, illetve az általa preferált podcast típusokat. A hallgatási szokások alapján ezt a listát a háttérben bővítem. A szokásokat úgy határozzom meg, hogy a legutóbbi harminc meghallgatott epizód vagy utóbbi egy hónap során hallgatott podcasteket arányosan leosztva felveszem mint kategória. A podcast hallgatáskor azonban nem csak az epizódot veszem figyelembe hanem a meghallgatott perceket is hogy arányosítani tudjam hosszabb epizódokra.

4.1.8. Képernyőkép generálás

Az alkalmazás megírása után további segítségrért fordultam a fastlane felé, mégpedig automatikus képernyőképek generálása céljából. Ennek a megvalósítására UI tesztek írására volt szükség. A UI tesztekkel szimulálhatjuk az alkalmazás használatát, amelyek során lépésről lépésre megadhatjuk, hogy mikor hova szeretnénk kattintani, vagy esetleg más gesztust bemenetként megadni a képernyő felé.

```
bundle exec fastlane snapshot init
```

4.10. lista. fastlane snapshot inicializálása

Ahhoz, hogy automatikusan képernyőképeket tudunk készíteni szükség van a snapshot felépítésére. A 4.10. listán látható parancssal elkészülnek a szükséges fájlok, ahol létrejön egy *SnapshotHelper.swift* fájl is. Ez tartalmazza a szükséges függvényeket, amiket a tesztek során meg kell hívni. A tesztesetek elején inicializálni kell a snapshot komponenst a létrejövő applikációval a 4.11. ábrán látható módon. A képernyőkép készítéséhez csak meg kell hívni a listán az ötödik sorban található függvényt, amely bemenetként várja a képernyőkép nevét. Végezetül a *bundle exec fastlane snapshot* parancsot kiadva futtatható a szolgáltatás, amely elkészíti a szükséges képernyőképeket.

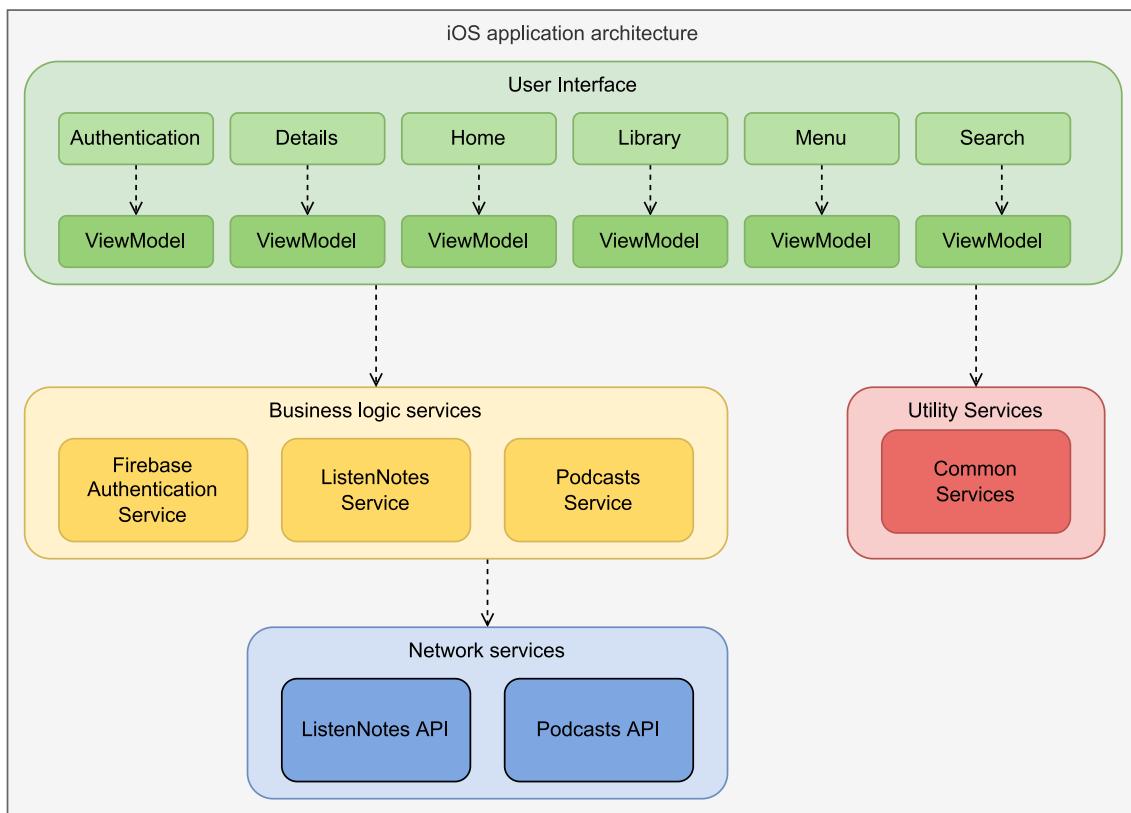
Az alkalmazás megvalósítása során háromrétegű architektúrát használtam. Ennek az a fő jellegzetessége, hogy az applikáció megvalósítása során külön csoportosítom a megjelenítéssel foglalkozó logikát, az adatok megszerzésével törődő hálózati komponenseket és az ezek közti adatátalakítási szerepet játszó üzleti logikai réteget. A 4.8. ábrán tisztán elkülönülnek az egyes rétegek egymástól. Az ábrán zöld színnel van jelölve a megjelenítésért felelős része az alkalmazásnak, sárga és piros színnel az üzleti logikát tartalmazó része és kékkel pedig a hálózati kommunikációért felelős komponensek. Ahogyan korábban már említve volt, az utóbbi része az alkalmazásomnak generálásra került.

```
let app = XCUIApplication()
setupSnapshot(app)
app.launch()

snapshot("ScreenshotName")
```

4.11. lista. fastlane snapshot felépítése

A megjelenítési réteget tovább lehet bontani, amely pedig egy modell-nézetnézetmodell (*MVVM: Model-View-View Model*) alapú architektúrára épül amelynek a lényege, hogy elhatárolja a tisztán nézetleírásért felelős részt a képernyőnek megfelelő adatok előállításával foglalkozó struktúráról. Az előnye, hogy a rétegek könnyen cserélhetők, így az alkalmazásban gyenge csatolás (*low coupling*) érhető el.



4.8. ábra. Az iOS alkalmazás általános architektúrája

4.2. Háttérszolgáltatás

A mobilalkalmazás megírása után következhetett a háttérszolgáltatás elkészítése. A szolgáltatáshoz a már korábban említett Ktor keretrendszer használtam. Alapjául egy Netty-servlet motort választottam. Ez egy olyan nem blokkoló kliens-szerver váz, amely Java hálózati alkalmazások elkészítésére szolgál. Az aszinkron eseményvezérelt hálózati alkalmazási keretrendszer és eszközök a programozás egyszerűsítésére szolgálnak, mint például a TCP és az UDP socket szerverek. A kommunikáció során küldött tartalmak formátuma az alkalmazásomban JSON, azonban egy átalakítót specifikálva bármilyen formátumot megadhatunk. A fejlesztése során legfőképpen *extension function*-öket használunk, amikkel új funkcionálitást tehetünk hozzá a szolgáltatásunkhoz. Ezeknek a lényege, hogy már egy meglévő osztályt könnyen kiegészíthetünk egy általunk megírt függvényel, amit meg-hívva egyszerűen kibővíthetjük a keretrendszer által nyújtott objektumokat. Az új funkció hozzáadása magasabb rendű függvények¹⁰ meghívásán keresztül történik. Ilyen az alábbi példában a route függvény. Ezeknek a segítségével fa struktúra szerint lehet definiálni az egyes végpontokat az alkalmazásban. Amíg ezek csak a specifikálásban játszanak szerepet, addig a HTTP kéréseket specifikáló függvényhívásokban (pl: get) már azt a kódot adjuk meg, amely a végpont meghívásakor fog lefutni. Ezek suspend (megállít) funkciók, amely annyit jelent, hogy a végrehajtásuk bármikor félbeszakítható és folytatható, automatikusan lehetővé téve ezzel az aszinkron működést.

4.2.1. OpenAPI

A szolgáltatás elkészítésekor a már korábban elkészített OpenAPI megbízást használtam fel, hogy az alapján hozjam létre a megfelelő végpontokat és a kommunikációs modellt. A generáláshoz ugyanúgy a már korábban említett OpenAPI generátort használtam fel. A letöltése a már 4.3. listán említett módon történik, amelyhez előfeltétel a Homebrew megléte, ami viszont a 4.2. listán bemutatott parancssal tehető meg. A generáláshoz szintén létrehoztam egy konfigurációs fájlt (lásd 4.12. lista), amelyben megadtam a szükséges paramétereket. Itt kikapcsolásra történtek olyan paraméterek, amelyekre számomra nem volt szükség, illetve megadtam a szükséges csomagneveket.

```
featureMetrics: false
featureAutoHead: false
featureCompression: false
featureHSTS: false
groupId: "com.dipterv.martin"
packageName: "com.dipter.martin"
artifactId: "podcasts-server"
```

4.12. lista. Konfiguráció a ktor generátorhoz

A konfigurációs fájl elkészülte után a korábbihoz hasonló módon elkészítettem a generátor segítségével a könyvtárat. A generáláshoz a 4.13. listán található parancsot használtam. A parancs során ugyanúgy sorban megadom az API leíró fájlt, a használandó generátort, a kimenet helyét és az előbb bemutatott konfigurációs fájlt.

```
openapi-generator generate \
-i ./openapi.yaml \
-g kotlin-server \
-o ./podcast-api/ \
-c ./server-config.yml
```

4.13. lista. Terminál parancs backend generálásra

¹⁰Lambdát visszaadó vagy paraméterként kapó függvényeket magasabb rendű függvényeknek nevezzük (higher order function). A lambdákat csak úgy is létrehozhatunk, azonban a függvények törzse is annak számít, így könnyedén tudunk akár funkciókat is megadni paraméterként, amely egyszerűen meghívható.

A generálás során azonban sajnos nem sikerült az így létrehozott szolgáltatást egyből futtatni. Ennek következtében amellett döntöttem, hogy a létrehozott modell és útvonal leíró fájlokat csak egyszerűen behúzom a könyvtárba. A modellekkel nem kell foglalkoznom, azonban a végpontokon meg kell hívnom a megfelelő általam megírt üzleti logikai rétegbeli függvényeket. Ennek következtében igyekeztem minimálisra csökkenteni a generált fájlokba írt kódmenyiséget. Ideális esetben a legenerált fájlokban nem szabad írni, mert azok újból elkészüléskor felülíródnak. Az általam elkészített rendszer tehát az útvonalleíróból csak továbbhív a megfelelő saját implementációs szolgáltatásomba, ahol a belső működés alapján összeszedi vagy manipulálja a hozzá kapcsolódó adatokat az adatbázisban. Adatbázisnak egy PostgreSQL adatbázist választottam, amely döntés mellett szintén az szolt, hogy korábban még ilyennel nem foglalkoztam, így remek alkalom volt a kipróbálásra.

4.2.2. Docker

Tegyük fel, hogy három különböző Python-alapú alkalmazással rendelkezünk, amelyeket egyetlen kiszolgálón kíván üzemeltetni (amely lehet fizikai vagy virtuális gép). Ezen alkalmazások mindegyike a Python más-más verzióját használja, valamint a kapcsolódó könyvtárak és függőségek alkalmazásonként eltérőek. Mivel ugyanarra a gépre nem telepíthetjük a Python különböző verzióit, ez megakadályozza, hogy minden alkalmazást ugyanazon a számítógépen tároljuk. Nézzük meg, hogyan oldhatnánk meg ezt a problémát a Docker használata nélkül. Egy ilyen forgatókönyvben megoldást találhatunk három fizikai géppel, vagy egyetlen számítógéppel, amely elég erős ahhoz, hogy három virtuális gépet üzemeltethessen és futasson rajta. Mindkét lehetőség lehetővé tenné, hogy a Python különböző verziót telepítsük ezekre a gépekre a hozzájuk tartozó függőségekkel együtt. Függetlenül attól, hogy melyik megoldást választjuk, a hardver beszerzésével és karbantartásával kapcsolatos költségek meglehetősen drágák. Azonban ezt meg lehetne oldani a Docker használatával, amely költséghatékony megoldás erre a problémára.

A gépet, amelyen a Docker telepítve van, és amin fut, általában Docker-gazdaként vagy gazdagépként említik. Így amikor azt tervezzük, hogy telepítünk egy alkalmazást a gazdagépen, az létrehoz egy logikai entitást az alkalmazás hosztolására. A Docker terminológiában ezt Container-nek (tárolónak) vagy pontosabban Docker Container-nek nevezzük. A konténerben nincs semmilyen operációs rendszer telepítve, azonban van egy virtuális másolata a folyamatlábról, a hálózati interfészkről és a fájlrendszer felcsatolási pontjairól. Ezeket annak a gazdagépnak az operációs rendszerétől örököltek, amelyen a tároló található és fut. Míg a gazdagép operációs rendszerének kernelje meg van osztva az összes rajta futó konténer között. Ez lehetővé teszi, hogy minden egyes tárolót elkülönítsünk az ugyanazon a gazdagépen lévő másiktól. Így több, eltérő alkalmazáskövetelményekkel és függőséggel rendelkező konténert lehet ugyanazon a gazdagépen futtatni, mindaddig, amíg ugyanazok az operációs rendszer követelményei. Röviden, a Docker virtualizálja annak a gazdagépnak az operációs rendszerét, amelyen telepítve van és fut, a hardverösszetevők virtualizálása helyett. Ennek következtében azonban látható a hátránya is, amely pedig az, hogy ha különbözik az operációs rendszer követelménye, akkor nem lehet ugyanazon a gazdagépen futtatni őket.

Az szolgáltatásomban a könnyű felépítés eléréséhez azt választottam, hogy a megírt háttérszolgáltatást és az adatbázist egy-egy konténerben fogom futtatni. Az előnye, hogy így lokálisan nemek sincs szükségem arra, hogy a PostgreSQL adatbázishoz előkészítsem a környezetet, hanem csak egy konténer elindításával bármikor futtatni tudom. Azonban a tárolókban található fájlok a konténer törlése után megszűnnék. Ennek a kiküszöbölésére létre lehet hozni köteteimet, amelyek felcsatolhatóak a konténerekre. Ezek lehetnek a tárolók által létrehozott kötetei vagy megadhatjuk őket külső függőségeként is. Én a

másodikat választottam, így mindenképp megmarad a konténer törlésekor. Továbbá az adatbázisnak megmondtam, hogy ezt a felcsatolt kötetet használja az adatbázisok eltárolására.

```
FROM openjdk:11-jdk-slim

WORKDIR .
COPY /build/libs/PodcastsBackend-0.1-all.jar .

EXPOSE 8080

CMD java -jar ./PodcastsBackend-0.1-all.jar
```

4.14. lista. Dockerfile tartalma

Az általam megírt szolgáltatást is becsomagoltam egy konténerbe. Ennek a megvalósításához legelőször is szükség van egy *Dockerfile* fájlra (4.14. lista). A fájl lényege, hogy megadjuk hogyan lehet létrehozni egy Docker Image-et (kép). Egyszerűen fogalmazva ez egy recept, hogyan kell elkészíteni egy képet. Később azt a képet futtatva fog létrejönni a konténer. Ahhoz, hogy a szolgáltatásomat futtathassam lehessen létrehoztam egy *.jar* fájlt belőle. A *Dockerfile*-ban pedig leírom, hogy egy olyan képet használjon alapul, amelyen megtalálható a java fejlesztői környezet, majd az elkészített *.jar* fájlt másolja át a gyökérkönyvtárba. Ezen felül továbbá kiajánlom a megfelelő portot, a 8080-ast, hogy a kívülről el lehessen érni. A szolgáltatás saját maga is ezen a porton várja a bejövő kéréseket. Végezetül pedig elindítatom az alkalmazást. Ezekkel a lépésekkel lehet létrehozni egy konténert a háttérszolgáltatásomból.

```
version: '2'
services:
  postgres:
    image: "postgres:9.6.0"
    restart: "always"
    volumes:
      - "podcasts-data:/var/lib/postgresql/data"
    ports:
      - "54321:5432"
    environment:
      POSTGRES_USER: "REDACTED"
      POSTGRES_PASSWORD: "REDACTED"
      POSTGRES_DB: "podcasts"
      mem_limit: "8g"
  podcasts-backend:
    build: .
    ports:
      - 8080:8080
    depends_on:
      - postgres
    volumes:
      podcasts-data:
        external: true
```

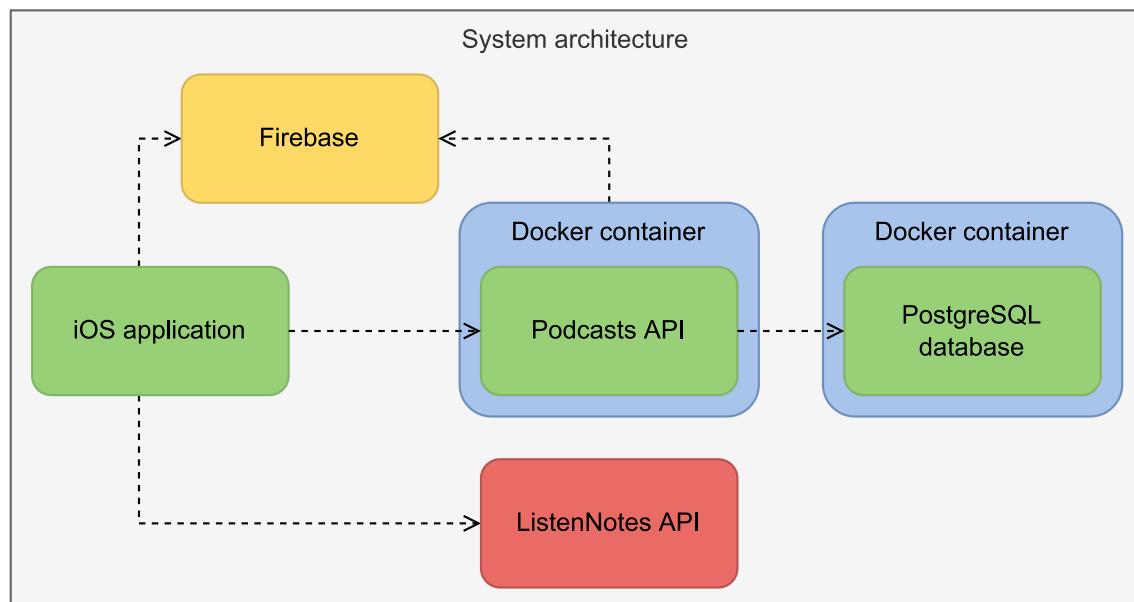
4.15. lista. Docker compose fájl

A Docker Compose segítségével meg lehet adni többkonténeres szolgáltatásokat. Használatával egyszerre lehet kezelni őket. Az általam felépített rendszer szintén több konténerből épül fel, amelyek között függőségi kapcsolat is fennáll, így hasznos ha a háttérszolgáltatás elindításakor automatikusan elindul az adatbázis is. A használatához szükség van egy *docker-compose.yml* konfigurációs fájl létrehozására, amely tartalma látható a 4.15. listán. Legelső lépésként specifikálom, hogy a Compose második verzióját szeretném használni, majd ezután megadom a szükséges szolgáltatásokat. Legelső az adatbázis, amely a *postgres* névre hallgat. Itt specifikálom a futtatáshoz szükséges adatokat, mint a használandó kép, amely alapból tartalmaz Postgres installációt, a felcsatolt kötetet és a portok

átalakítását. Következőkben megadom a saját szolgáltatásom *podcasts-backend* néven. A legelső sorral megmondom, hogy a jelenlegi könyvtárban talál információt arról, hogyan kell elkészíteni, amely a korábban említett *Dockerfile* lesz. Összekötöm a bent megtalálható portot a gazdagép portjával és függőségként specifikálom az adatbázist, hogy annak előbb kell elkészülnie, mint a jelenleginek. Végezetül megadom, hogy lesz egy kötet, amely kúlsőleg lesz létrehozva *podcasts-data* néven. Ezek után a *docker compose up* parancssal létrejön az összes konténer és megtörténik a hálózati összekötés is. A leállításhoz a *docker compose down* parancs használható.

4.3. Teljes rendszer

Az elkészült rendszer egy iOS alkalmazás, amely a podcast adatokat a Listen Notes által nyújtott szolgáltatásból szedi, a felhasználói adatokat az általam írt és konténerben futó Podcasts háttérszolgáltatás kezeli a beléptetés kivételével, amelyre a Firebase-t használom. Az adatok perzisztens tárolásáért egy konténerben futó PostgreSQL adatbázis felel. A rendszerről a 4.9. ábrán látható egy általam készített diagram.



4.9. ábra. Az rendszer általános architektúrája a zöld részek jelenlik az általam készített komponenseket.

5. fejezet

Tesztelés

Az alkalmazás tesztelésére Unit tesztek készültek, amelyek főként az üzleti logikát tesztelik. Ebbe bele tartozik az adatok átalakítása a számonra szükséges modellekre és a komponensek helyes működése a megfelelő eseményekre. Itt jelenthet sikeres események során az adatok képernyőn való megjelenítését, vagy sikerteleneknél hiba képernyő mutatása a felhasználó felé. Maguk a nézetek nem lettek tesztelve, csak a View Model résszel bezáróságosan. Az applikációt továbbá manuálisan is teszteltem szimulátoron. Valós eszközön nem volt rá lehetőség, mert ahhoz Apple által adott engedélyre lett volna szükség, amit fejlesztői fiókhoz kötnek.

5.1. Felhasználói leírás

A következő fejezetben bemutatom, hogyan lehet a szakdolgozat során elkészített alkalmazást a mobiltelefonra telepíteni. Majd ezután részletesen szemléltetem az applikáció használati módját.

5.1.1. Telepítés

Az alkalmazás tesztelésére csak szimulátoron van lehetőség, ezáltal a futtatásához Xcode szükséges. A legkisebb Xcode verzió, amellyel felépíthető az applikáció az Xcode 13, ugyanis ebben a verzióban jött be az iOS 15-ös verzió fejlesztéséhez szükséges frissítés. A projekt letöltése után azonban még szükség van további lépésekre ahhoz, hogy az futtatható legyen.

```
brew install rbenv
rbenv install 3.0.0
rbenv global 3.0.0
```

5.1. lista. Ruby environment telepítése

Legelső lépésként szükség van legalább 3.0.0-ás verziójú Ruby környezetre. Ehhez én egy Homebrew könyvtárat ajánlok, amelyhez azonban szükséges a Homebrew környezet megléte a számítógépen, amely hiányában telepíthető a 4.2. ábrán látható parancssal.

```
eval "$(rbenv init -)"
```

5.2. lista. Bash profil fájl tartalma

A Homebrew segítségével ezután letölthetünk egy Ruby környezetet menedzselő eszközt, az rbenv szolgáltatást. A telepítés után az szolgáltatásnak megadhatjuk, hogy mi a 3.0.0 verziójú Ruby környezet szeretnénk használni, amely ezzel letöltésre kerül. Végezetül

megadjuk neki, hogy bármilyen Ruby parancs során ezt a verziójú környezetet használja a rendszer. Az elvégzéshez szükséges parancsok láthatóak az 5.1. listán.

A környezet megléte után frissíteni kell a bash profilunk egy új parancssal. Ez azért szükséges, hogyha terminál ablakot nyitunk, akkor a rbenv szolgáltatás inicializálja magát, alkalmazva ezzel az általunk előzőleg beállított dolgokat. Az ehhez szükséges parancs megtalálható az 5.2. listán.

```
curl -fsSL https://github.com/rbenv/rbenv-installer/raw/HEAD/bin/rbenv-doctor | bash
```

5.3. lista. Ellenőrző script a helyes telepítéshez

Ha az előző lépésekkel végeztünk akkor ellenőrizni tudjuk az rbenv-doctor (rbenv orvos) eszköz segítségével a sikeres telepítést. Az 5.3. listán látható parancsot kiadva az 5.4. listán láthatóhoz hasonló kimenetet kell látnunk. Ha minden zöld és OK akkor helyben vagyunk.

```
penzes.martin@'REDACTED'~ \% curl -fsSL https://github.com/rbenv/rbenv-installer/raw/HEAD/bin/rbenv-doctor | bash
Checking for `rbenv` in PATH: /usr/local/bin/rbenv
Checking for rbenv shims in PATH: OK
Checking `rbenv install` support: /usr/local/bin/rbenv-install (ruby-build 20221124)
Counting installed Ruby versions: 1 versions
Auditing installed plugins: OK
```

5.4. lista. Ellenőrző script eredménye az én eszközömön

Következő lépésként le kell tölteni a projekt fejlesztése során felhasznált dependenciákat. Legelső lépésként szükség van a CocoaPods eszközre, amely Bundler segítségével lett hozzáadva az alkalmazáshoz. Ezt ugyanúgy lehet leszedni, ahogy én ezt eredetileg hozzáadtam a projekthez. Ezután pedig a CocoaPods segítségével a megfelelő függőségeket le kell tölteni. Az ehhez szükséges két parancs az 5.5. listán található.

```
bundle install
bundle exec pod install --repo-update
```

5.5. lista. A Bundler által felvett függőségek letöltése, majd a CocoaPods segítségével megadott dependenciák telepítése

Mindezek megléte után a munkaterületet Xcode alkalmazásban megnyitva, majd egy tetszőleges szimuláltort kiválasztva felépíthető és futtatható és kipróbálható az diploma-munkára elkészített podcast alkalmazásom.

6. fejezet

Összefoglalás és továbbfejlesztési lehetőségek

Ennek a fejezetnek a célja röviden összefoglalni, hogy miről volt szó a diplomatervben.

A dolgozatom elején bemutattam az Android Auto és az Apple Carplay platformokat és jellemeztem őket.

Felépítettem egy tervet az alkalmazáshoz. Ezek után bemutattam azokat a technológiákat, amelyeknek a segítségével elkészíthető egy, a felhasználók számára kényelmesen és intuitívan használható applikáció.

A következő fejezetben ismertettem az általam megvalósított alkalmazást és szolgáltatásokat, annak fejlesztése során felmerülő problémákat és az ezekre hozott döntéseket.

Végezetül az applikáció tesztelhetőségének bemutatásával zárom a dolgozatot.

Az elkészült rendszert úgy ítélem meg, hogy lefedi a kitűzött feladatokat.

A projekt elkészítése során nagy előnyt jelentett, hogy korábban már foglalkoztam iOS fejlesztéssel. Azonban a technológiaválasztással megnehezítettem a dolgomat, mert a SwiftUI számomra ismeretlen terület volt. A munkahelyen is ismerkedtünk az új technológiával így szintén hasznosnak bizonyult, hogy az itt szerzett tapasztalatokat megosztottam a munkatársaimmal.

Az alkalmazás elkészítésén kívül továbbá gyakorolhattam és megismerkedhettem olyan folyamatokkal, amelyek egy projekt életciklusa során csak egyszer kerülnek megalkotásra. Ilyen volt például az automatizálás elkészítése, amely következtében a fejlesztésnél már kevesebb dologgal kellett törődnöm, megkönnyítve ezzel az minden nap életem. Továbbá először használtam személyesen a hálózati réteg generálását végző eszközt, amelynek a segítségével hasznos időt megspórolhattam. Nem volt szükség a favágó módon megírandó osztályoknak a manuális elkészítésére.

Ezzel ellentétben a háttérszolgáltatások megírásával kapcsolatban csekély korábbi tapasztalatom volt. A mobilalkalmazások készítésekor mindig csak használnom kellett őket. A projekt remek szolgálatot tett abban, hogy betekintést nyerhettem ebbe a világba. Foglalkozhattam szerkezeti felépítés meghatározásával és adatbázis tervezéssel is. A döögös kezdés után úgy érzem egészen belejöttem, azonban érzem, hogy csak a felszínét érintettem a másik oldalnak. Továbbá megismerkedhettem a Docker környezettel, amelyet korábban csak minimálisan kipróbáltam céljából használtam. Láthattam, hogy milyen lehetőségek és esetleges hátrányok bíjnak meg mögötte.

Összességeiben azonban úgy döntöttem, hogy maradok az iOS fejlesztésnél. Viszont a diplomamunkám elkészítése alatt temérdek új kihívással találkoztam, amelyekre a megoldáskeresés majd megvalósítás érdekes volt számomra, így örömmel ültem le bármikor a projekt fejlesztéséhez. Nem mellesleg tetszik mind a Swift, mind a Kotlin nyelv így felüdülléssel használtam őket.

Az elkészült terméket is helytállónak ítélem meg a podcastek lejátszásában. Azonban magam is belátom, hogy közel sincs a többi, komoly csapatokkal rendelkező fejlesztőcégek hasonló alkalmazásaihoz képest.

Az alkalmazásban további fejlesztési lehetőségek találhatóak, amelyek megvalósítása a felhasználóknak a kényelmét és a használati esetek mennyiségett növelné. Az autóban a kormányon megtalálható egy mikrofon gomb, amivel a felhasználó hangvezérléssel is kezelheti az infotainment rendszert. Lehetőséget lehetne adni a felhasználónak, hogy ezzel vagy a felhasználói felületen lévő gombbal hangalapú keresést indítson, majd azok közül játsszon le egy epizódot. Jelenleg a podcastek lejátszására csak az Apple Carplay alkalmazáson keresztül van lehetőség. Ezt azonban a telefonon is elérhetővé lehetne tenni, így bárhol hallgathatná a kedvenc podcastjeit a felhasználó, nagyobb piaci célközönséget elérve ezáltal.

Irodalomjegyzék

- [1] Andiamo!: Right to left languages | andiamo! the language professionals.
URL <https://www.andiamo.co.uk/resources/right-to-left-languages/>. (last visited: 2022. november 27.).
- [2] Apple: About ios 15 updates. URL <https://support.apple.com/en-us/HT212788>. (last visited: 2022. november 27.).
- [3] Apple: App store - apple. URL <https://www.apple.com/app-store/>. (last visited: 2022. october 23.).
- [4] Apple: Foundation | apple developer documentation.
URL <https://developer.apple.com/documentation/foundation>. (last visited: 2022. november 27.).
- [5] Apple: ios - carplay - apple. URL <https://www.apple.com/ios/carplay/>. (last visited: 2022. october 23.).
- [6] Apple: ios 16.
URL <https://www.apple.com/ios/ios-16/>. (last visited: 2022. november 27.).
- [7] Apple: Sf symbols - apple developer.
URL <https://developer.apple.com/sf-symbols/>. (last visited: 2022. december 5.).
- [8] Apple: SwiftUI - overview - xcode - apple developer.
URL <https://developer.apple.com/xcode/swiftui/>. (last visited: 2022. november 24.).
- [9] Apple: Uikit | apple developer documentation.
URL <https://developer.apple.com/documentation/uikit>. (last visited: 2022. november 29.).
- [10] Clifford Atiyeh: Apple carplay is free for bmw owners, starting now. URL <https://www.caranddriver.com/news/a30139034/bmw-apple-carplay-free/>. (last visited: 2022. october 23.).
- [11] Baeldung: The dao pattern in java.
URL <https://www.baeldung.com/java-dao-pattern>. (last visited: 2022. november 28.).
- [12] Git community: Pak format documentation.
URL <https://git-scm.com/docs/pack-format>. (last visited: 2022. november 30.).
- [13] Homebrew contributors: Homebrew readme, 2022. 11. URL <https://github.com/Homebrew/brew/blob/master/README.md#who-are-you>. (last visited: 2022. november 29.).

- [14] OpenAPI-Generator Contributors: Documentation for the swift5 generator.
URL <https://openapi-generator.tech/docs/generators/swift5>. (last visited: 2022. november 29.).
- [15] OpenAPI-Generator Contributors: Hello from openapi generator.
URL <https://openapi-generator.tech>. (last visited: 2022. november 29.).
- [16] Eloy Durán: Xcodeproj - create and modify xcode projects from ruby. URL <https://github.com/CocoaPods/Xcodeproj>. (last visited: 2022. november 29.).
- [17] GitLab: Ci/cd.
URL <https://docs.gitlab.com/ee/ci/>. (last visited: 2022. december 4.).
- [18] GitLab: Install gitlab runner on macos.
URL <https://docs.gitlab.com/runner/install/osx.html>. (last visited: 2022. december 4.).
- [19] GitLab: The one devops platform. URL <https://about.gitlab.com>. (last visited: 2022. november 30.).
- [20] Google: Android auto | android.
URL <https://www.android.com/auto/>. (last visited: 2022. october 23.).
- [21] Google: Android: Mediabrowsercompatservice dokumentáció.
URL <https://developer.android.com/reference/androidx/media/MediaBrowserServiceCompat>. (last visited: 2022. december 4.).
- [22] Google: Android: Mediaitem dokumentáció.
URL <https://developer.android.com/reference/android/support/v4/media/MediaBrowserCompat.MediaItem>. (last visited: 2022. december 4.).
- [23] Google: Android mobile app developer tools - android developers.
URL <https://developer.android.com>. (last visited: 2022. november 23.).
- [24] Google: Design for driving | google developers. URL <https://developers.google.com/cars/design/automotive-os>. (last visited: 2022. december 4.).
- [25] Google: Fastlane - app automation done right. URL <https://fastlane.tools>. (last visited: 2022. december 3.).
- [26] Google: Firebase.
URL <https://firebase.google.com/>. (last visited: 2022. november 17.).
- [27] Google: Google play store. URL <https://play.google.com/>. (last visited: 2022. october 23.).
- [28] Chris Hoffman: How to install packages with homebrew for os x, 2020. 03. URL <https://www.howtogeek.com/211541/homebrew-for-os-x-easily-installs-desktop-apps-and-terminal-utilities/>. (last visited: 2022. november 29.).
- [29] Huawei: Appgallery.
URL <https://appgallery.huawei.com/Featured>. (last visited: 2022. october 23.).
- [30] David Jennes: Swiftgen - the swift code generator for your assets, storyboards, localizable.strings, ... — get rid of all string-based apis!
URL <https://github.com/SwiftGen/SwiftGen>. (last visited: 2022. november 27.).

- [31] JetBrains: Company - jetbrains.
URL <https://www.jetbrains.com/company/>. (last visited: 2022. november 23.).
- [32] JetBrains: Exposed - a kotlin sql framework.
URL <https://github.com/JetBrains/Exposed>. (last visited: 2022. november 28.).
- [33] JetBrains: Kotlin programming language.
URL <https://kotlinlang.org/>. (last visited: 2022. november 23.).
- [34] JetBrains: Ktor: Build asynchronous servers and clients in kotlin.
URL <https://ktor.io>. (last visited: 2022. december 7.).
- [35] Kodeco: swift-style-guide | the official swift style guide for raywenderlich.com.
URL <https://github.com/kodecocodes/swift-style-guide>. (last visited: 2022. november 27.).
- [36] Yonas Kolb: Xcodegen - a swift command line tool for generating your xcode project.
URL <https://github.com/yonaskolb/XcodeGen>. (last visited: 2022. november 29.).
- [37] Michael Long: Factory | a new approach to container-based dependency injection for swift and swiftui. URL <https://github.com/hmlongco/Factory>. (last visited: 2022. december 7.).
- [38] Michael Long: Resolver | swift ultralight dependency injection / service locator framework. URL <https://github.com/hmlongco/Resolver>. (last visited: 2022. december 7.).
- [39] Listen Notes: Listen notes homepage. URL <https://www.listennotes.com>. (last visited: 2022. october 30.).
- [40] Oracle: Java | oracle.
URL <https://www.java.com/en/>. (last visited: 2022. november 23.).
- [41] Eric Ravenscraft: Wireless charging is a disaster waiting to happen. URL <https://debugger.medium.com/wireless-charging-is-a-disaster-waiting-to-happen-48afdde70ed9>. (last visited: 2022. october 23.).
- [42] Jon Shier: Alamofire: Elegant http networking in swift. URL <https://github.com/Alamofire/Alamofire>. (last visited: 2022. november 27.).
- [43] Statcounter: ios market share worldwide, 2022. <https://gs.statcounter.com/ios-version-market-share/mobile-tablet/worldwide/#daily-20221125-20221125-bar> (last visited: 2022. november 27.).
- [44] Statcounter: Mobile operating system market share united states of america, 2022. <https://gs.statcounter.com/os-market-share/mobile/united-states-of-america> (last visited: 2022. october 23.).
- [45] Statcounter: Mobile operating system market share worldwide, 2022. <https://gs.statcounter.com/os-market-share/mobile/worldwide> (last visited: 2022. october 23.).
- [46] YAML Language Development Team: Yaml ain't markup language (yamlTM) version 1.2. URL <https://yaml.org/spec/1.2.2/>. (last visited: 2022. november 29.).

- [47] Brett Terpstra: Homebrew, the perfect gift for command line lovers, 2009. 12. URL <https://www.engadget.com/2009-12-25-homebrew-the-perfect-gift-for-command-line-lovers.html>. (last visited: 2022. november 29.).
- [48] Tuist: Xcode on steriods | tuist.
URL <https://tuist.io>. (last visited: 2022. november 279.).
- [49] Waze: Waze about us. URL <https://www.waze.com/company>. (last visited: 2022. october 23.).
- [50] Wikipedia: Chamorro language.
URL https://en.wikipedia.org/wiki/Chamorro_language. (last visited: 2022. october 30.).
- [51] Wikipedia: Faroese language.
URL https://en.wikipedia.org/wiki/Faroese_language. (last visited: 2022. october 30.).
- [52] Wikipedia: Standard generalized markup language. URL https://en.wikipedia.org/wiki/Standard_Generalized_Markup_Language. (last visited: 2022. november 29.).
- [53] Wikipedia: Xml (standard.
URL <https://en.wikipedia.org/wiki/XML>. (last visited: 2022. november 29.).