TANSZÉKVEZETŐ

DIPLOMATERVEZÉSI FELADAT

Séllyei Bence
Mérnökinformatikus hallgató részére

Szálláskiadó rendszer fejlesztése iOS platformon okosotthon integrációval

Manapság az emberek számos lehetőség közül választhatnak a szálláshelyek tekintetében, amikor utazásra szánják rá magukat. A lehetőségek közül viszont egyre többen szállnak meg szívesebben magánszemélyek által hírdetett szobákban vagy lakásokban, mint hotelokban vagy szállodákban külföldi tartózkodásuk során. A magánszemélyek által bérbeadott szállásoknak rengeteg előnyük van, többnyire sokkal olcsóbbak a szállodáknál, kevesebb pénzért kaphatnak nagyobb élhető területet, valamint általában otthonosabb érzést is nyújtanak vendégeik számára.

Azonban az ilyen szálláshelyekből hiányzik a személyzet, ezért ha valamilyen technikai gondja támad a vendégnek (pl. fázik, bekapcsolná a fűtést, de túl bonyolult vagy nem érti hogyan kell beüzemelni), akkor a szállásadóhoz kell fordulni, ami sok esetben kényelmetlen lehet mindkét fél számára. Ezen probléma megoldására használhatók olyan eszközök, amelyek például a különféle bonyolult fűtés- vagy klímaberendezések használatát egységesítik, egyszerűsítik és lehetőséget adnak arra, hogy az eszközöket akár távolról, telefonunk segítségével is vezérelhetjük.

A hallgató feladata egy olyan rendszer megtervezése és elkészítése iOS platformon, amellyel a felhasználók bérbeadó szállásokat böngészhetnek és vehetnek ki, valamint a kibérelt szobákban, lakásokban található okos eszközöket vezérelhetik telefonjukról egy alkalmazás segítségével.

A hallgató feladatának a következőkre kell kiterjednie:

- Mutassa be röviden az iOS platformot és a felhasznált technológiákat.
- Tervezze meg és ismertesse a rendszer felhasználói felületét és architektúráját.
- Készítse el a rendszer prototípusát.
- Valósítson meg a rendszerrel együttműködő okosotthon megoldásokat.

Tanszéki konzulens: Dr. Forstner Bertalan Ph.D. egyetemi docens

Budapest, 2022. március 06.

Dr. Charaf Hassan
egyetemi tanár
tanszékvezető

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

H-1117 Budapest, Magyar tudósok krt 2. Q.B.207
Telefon: 463-2870. Fax: 463-2871
www.aut.bme.hu

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

Séllyei Bence

SZÁLLÁSKIADÓ RENDSZER FEJLESZTÉSE IOS PLATFORMON OKOSOTTHON INTEGÁRICÓVAL

KONZULENS

Dr. Forstner Bertalan

BUDAPEST, 2022

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Séllyei Bence**, 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 hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2022. 12. 08.

...…….

Séllyei Bence

Összefoglaló

A technológia rohamos fejlődésével egyre több és több olyan eszköz jelenik meg, amely segít a mindennapjainkban. Legyen szó akár okostelefonokról vagy más okoseszközökről, ezek fő feladata, hogy minél egyszerűbben kiszolgáljanak minket funkcióikkal mind a munkában, mind pedig a szabadidőnkben. Az elmúlt években megjelenő okosotthon megoldások az egyik legjobb példa az ilyesfajta kiszolgálásra, mivel az okosotthon koncepciója egyértelműen a lakhelyünk még hatékonyabb kihasználására törekszik.

A technológia fejlődésével már az utazás is sokkal elérhetőbb mindenki számára, mint korábban. Az Airbnb térnyerésével pedig már jelentősen olcsóbban is hozzájuthatunk szállásokhoz külföldi utazásaink során. A két területet összekapcsolva született meg egy alkalmazásötlet, valamint a dolgozat témája.

A feladat egy olyan iOS alkalmazás elkészítése volt, amely teljesíti egy szállásfoglaló platform követelményeit, valamint lehetővé teszi, hogy kiadó szállásainkhoz okoseszközöket rendeljünk, melyeket vendégeink szabadon használhatnak ott-tartózkodásuk ideje alatt. A mobilalkalmazás megvalósítása mellett egy Vapor alapú webalkalmazást is készült, aminek az alkalmazáshoz tartozó adatok tárolása és szolgáltatása volt a feladata. Az authentikáció miatt a Firebase szolgáltatásaival, az okosotthon funkciók kezeléséhez pedig az openHAB platformmal is együttműködik a rendszer.

A dolgozat betekintést nyújt a rendszer tervezésébe és annak megvalósításába. Bemutatja, hogy hogyan lehet az openHAB platformon keresztül okosotthon eszközökkel kommunikálni, és hogy milyen implementációs lépések vezettek el odáig, hogy egy mobilapplikációval tudjunk kinyitni egy zárat vagy bekapcsolni egy fűtést.

Abstract

With the rapid development of technology, more and more devices are appearing to help us in our everyday lives. Whether it's smartphones or other smart devices, their main purpose is to make it as easy as possible for us to use their functions at work and at leisure. The smart home solutions that have emerged in recent years are one of the best examples of this kind of service, as the concept of the smart home is clearly aimed at making more efficient use of our living space.

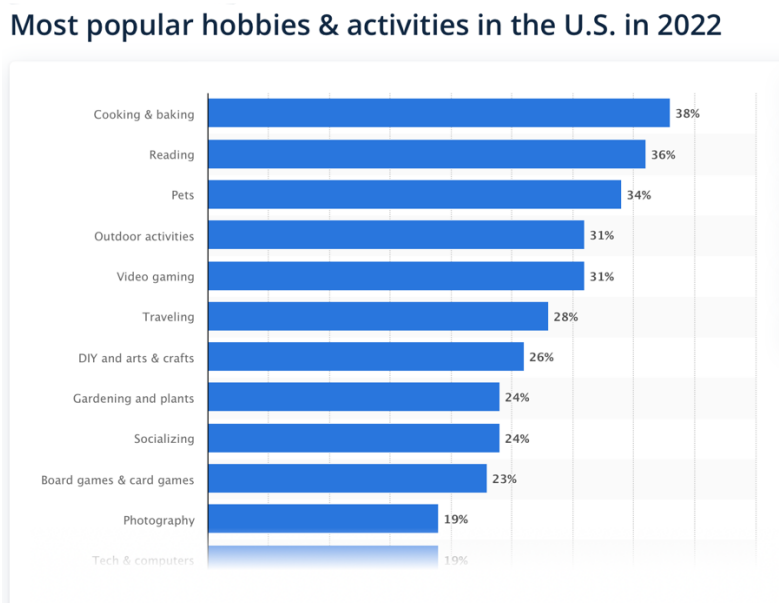
With advances in technology, travel is now more accessible than ever. And with the rise of Airbnb, we can now find accommodation for our travels abroad at significantly lower prices. By combining these two areas, an application idea and the topic of this thesis have been born.

The task was to create an iOS application that fulfils the requirements of a booking platform and allows us to assign smart devices to our rental accommodation, which our guests can use freely during their stay. In addition to the implementation of the mobile application, a Vapor-based web application was also developed to store and serve the data for the application. The system also works with Firebase for authentication and with the openHAB platform to manage the smart home functionality.

The thesis provides an insight into the design and implementation of the system. It shows how to communicate with smart home devices via the openHAB platform and the implementation steps that led to the ability to open a lock or turn on a heater with a mobile application.

1 Bevezetés

Manapság az utazás az egyik legnépszerűbb tevékenység az emberek körében. Rengetegen találják meg a saját kikapcsolódásukat egy jól eltöltött hétvégén egy másik városban, vagy akár országban, mint a saját otthonukban. Nem csoda, hogy így tudnak a legjobban kikapcsolódni, hiszen főként a fiatalok általában szeretnek új dolgokat kipróbálni, helyeket látni, emberekkel ismerkedni, illetve olyan kultúrákat megismerni, amelyekkel még sohasem találkoztak. [1] Egy 2022-es amerikai felmérés szerint a megkérdezettek 28%-a válaszolta, hogy a kedvenc szabadidős tevékenysége az utazás. [2]



1.1. ábra Amerikaiak kedvenc hobbijai 2022-ben[1]

1.1 Szálláslehetőségek utazásunk során

Egy utazás megszervezésénél számos döntést meg kell hoznunk, beleértve ezekbe a közlekedési eszközt, amivel eljutunk az úticélunkra, vagy a szállás kiválasztását, hogy hol alszunk majd az ott tartózkodásunk ideje alatt. Mindkettő esetében nagyon sok opció közül választhatunk, bár utóbbinál valamennyivel szabadabbak a választási lehetőségeink. Ilyen lehetőségek közé tartoznak a hagyományos szálláshelyek, mint például hotelek és szállodák, az elmúlt években térhódító Airbnb szállások, vagy akár az

[1] Kép forrása: https://www.statista.com/forecasts/997050/hobbies-and-activities-in-the-us

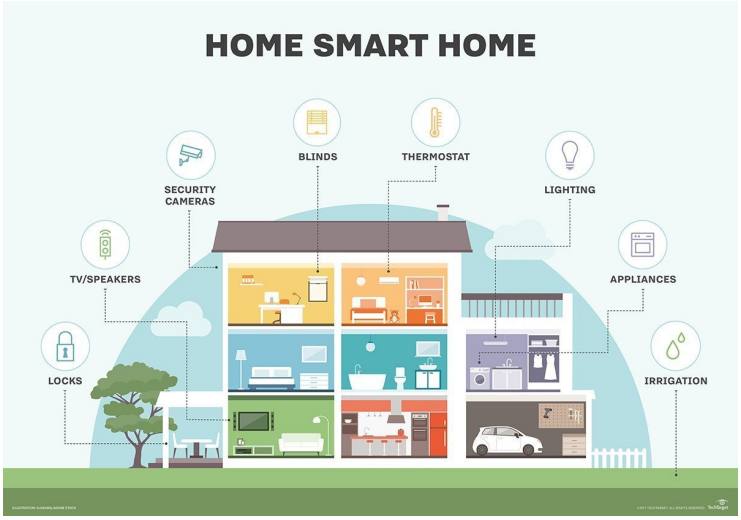
Airbnb megjelenése előtt nagyon népszerű CouchSurfing platform is. Airbnb-nek nevezzük azt az online foglalási rendszert, melyen keresztül magánszemélyek adhatnak ki szobákat vagy lakásokat rövidebb időszakokra. A CouchSurfing pedig az a platform volt, ahol szabad ágyainkat, kanapéinkat vagy szobáinkat oszthattuk meg 1-2 éjszakára kalandozni vágyókkal.

Utóbbi két lehetőség az évek során egyre népszerűbb lett a hagyományos szálláshelyekhez képest. Ennek a legfontosabb oka valószínűleg a jóval olcsóbb árazás volt. A CouchSurfing koncepciója az emberek köré épült, akik ismerkedni vagy időt eltölteni szeretettek volna idegenekkel, amíg megosztják velük szabad fekvőhelyeiket. Ebből kifolyólag a vendégfogadók nem kértek fizetséget cserébe, csupán jó társaságot. Az Airbnb esetében pedig nem is biztos, hogy maga a szállás kerül kevesebb pénzbe, viszont rengeteg más tényezőn spórolhatunk, ha ezt az opciót választjuk. Mivel privát lakhelyekről van szó, ezért szolgáltatási díjakat nem kell fizetnünk, például nincsen szobaszervíz vagy mindennapos takarítás. Cserébe a bérelt szállásunk tartalmazhat konyhát, amivel sok pénzt spórolhatunk meg, ha magunknak készítjük el az ételt, amit elfogyasztunk. [3][4]

1.2 Okosotthonok

Az okosotthon egy olyan lakóhely, ami internetkapcsolattal rendelkező eszközökkel együttműködve teszi elérhetővé az ott lakóknak az olyan rendszerek működtetését, mint például a fűtés vagy a világítás. Az okosotthon technológiát sokszor otthoni automatizálási rendszerként is emlegetik. Az okosotthonok biztonságot, kényelmet és energiatakarékosságot biztosítanak az ott tartózkodók számára. A rendszerhez csatlakoztatott eszközöket általában okostelefonokra készült alkalmazásokkal lehet vezérelni. [5]

Tapasztalataim azt mutatják, hogy a két témakört, az utazást, szállásfoglalást és az okosotthon rendszereket, remekül össze lehetne kötni. Számos alkalommal jártam már úgy, hogy utazásom során hosszú ideig kellett egyezkednem a szállásadómmal, hogy mikor tudja átadni nekem a kulcsot a szállásomhoz, mivel vagy ő nem ért rá akkor amikor én érkeztem, vagy pedig nem értem oda a neki megfelelő időben. Egy másik alkalommal, egy téli utazás során, nem volt számomra megfelelően beállítva a fűtés, ami miatt a szállásadónak vissza kellett jönnie a szállásra, hogy beállítsa a hőmérsékletet.



1.2. ábra Egy okosotthon és a benne található eszközök[2]

A fenti példák, problémák rendkívül egyszerűen kiküszöbölhetők lettek volna okoseszközök használatával. Ha egy okoszárhoz automatikusan kaptam volna hozzáférést, nem kell a vendégfogadónak megjelennie a kulcs átadása miatt. Ha egy okostermosztáttal be tudtam volna állítani a fűtést a lakásban, akkor szintén nem kellett volna visszajönnie a szállásadónak.

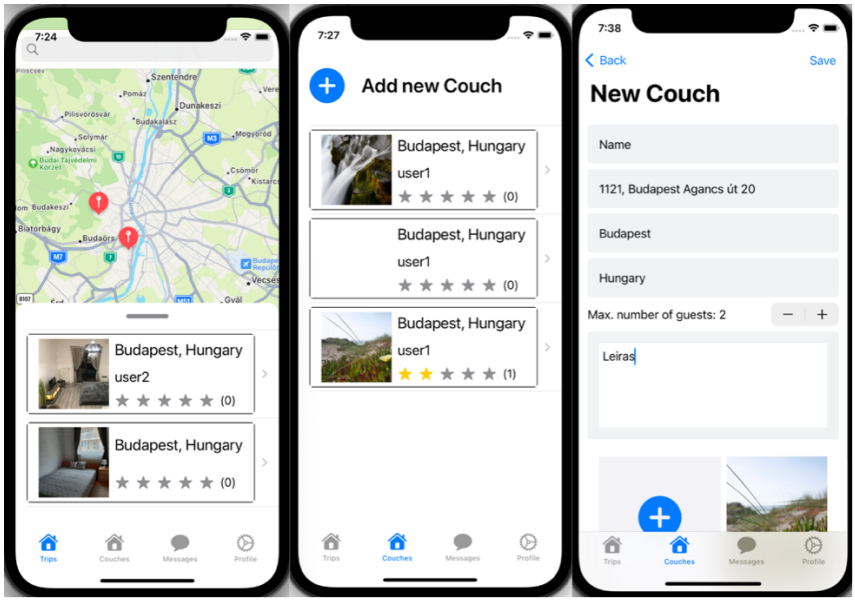
Ebben a diplomatervben bemutatom, hogy hogyan terveztem meg és készítettem el a fenti problémák megoldására egy rendszert. A dolgozat 2. fejezetében bemutatom a mobilalkalmazásban található funkciókat és nézeteket, majd betekintést adok az elkészítés során használt technológiákba. Végül pedig a 4. és 5. fejezetben azt mutatom be, hogy milyen komponensekből áll a rendszerem, hogyan terveztem meg ezeket a komponenseket, és a köztük lévő kommunikációt, valamint hogyan valósítottam meg a rendszert és a benne található okoseszközök integrációját.

[2] Kép forrása: https://www.techtarget.com/iotagenda/definition/smart-home-or-building

2 Alkalmazás funkcióinak bemutatása

Ebben a fejezetben sorra bemutatom a mobilalkalmazás elkészült felhasználói felületeit, illetve azt, hogy milyen funkciókat tartalmaz az applikáció és honnan tudja ezeket elérni a felhasználó.

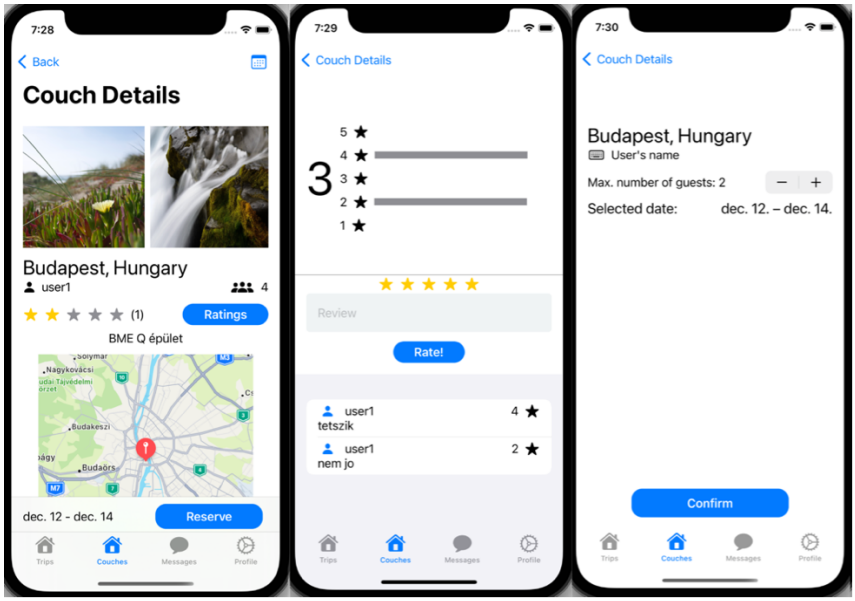
Az alkalmazás kezdőképernyője egy bejelentkezési felület. Innen elérhető maga a bejelentkezési űrlap, valamint egy regisztráció gomb. Sikeres bejelentkezést vagy regisztrációt követően az alkalmazás átugrik egy térképes nézetre.



2.1. ábra Térképes nézet, saját szálláshelyeim és szálláshely létrehozása

A 2.1-es ábra baloldalán látható az említett térképes nézet. Ezen a városhoz tartozó szálláshelyek kerülnek megjelenítésre kis piros jelzők formájában. A képernyőre félig belógó lista egy le és felnyitható nézet. Ebben a nézetben is megjelenítésre kerültek a városban található szállások. Ha itt rányomunk egy listaelemre, akkor a szállás részletes nézetére lép tovább az alkalmazás. Minden nézeten megjelenik egy alsó tab bar, amin található gombokkal lehet az egyes lapok között váltani. A 2.1-es ábra középső képernyőjén a bejelentkezett felhasználó által hirdetett szálláshelyeket lehet megtekinteni. A felhasználó innen tud új szálláshelyet is felvenni a + gombbal. A jobboldali képernyőn az új szálláshely létrehozásához szükséges űrlap nézete látható. Itt lehet tetszőlegesen elnevezni a szálláshelyet, megadni a címét, városát, illetve az országot is. Beállíthatjuk, hogy maximum hány férőhelyes a szállás, leírást is adhatunk meg hozzá,

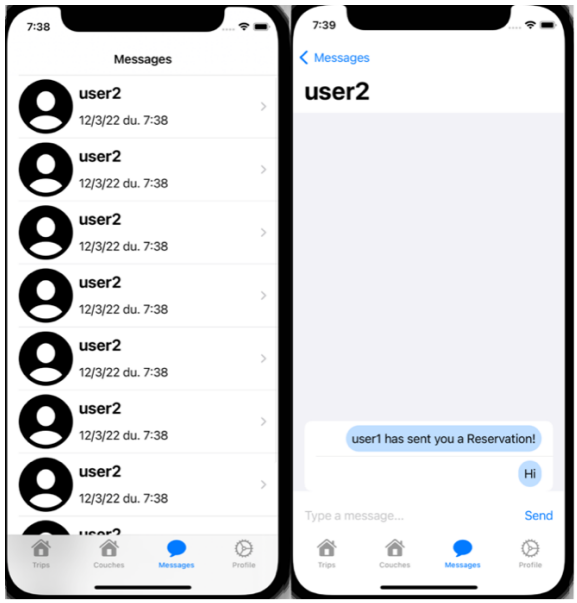
valamint a képeket is itt lehet feltölteni a szálláshoz. A már kiválasztott képeket jobbra húzva lehet megtekinteni.



2.2. ábra Szállás részletes nézete, értékelések, foglalás

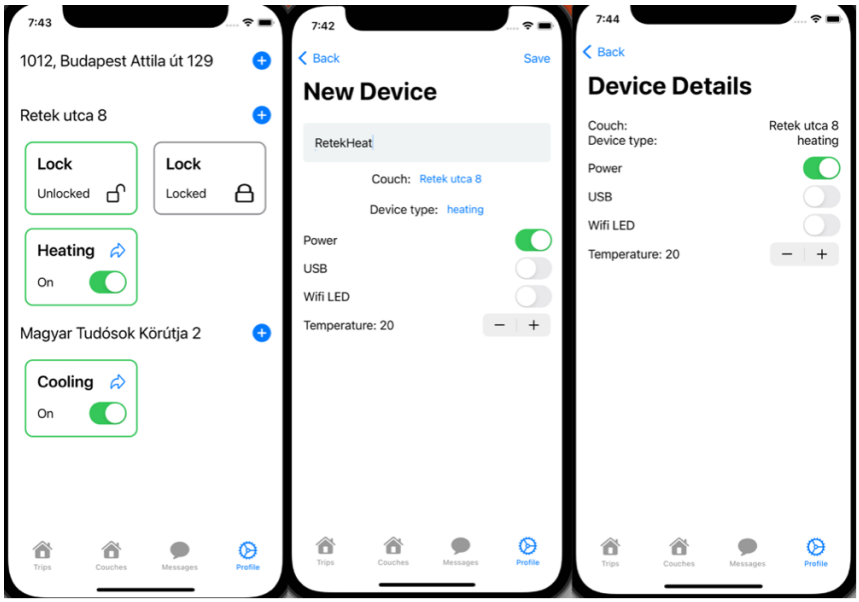
A 2.2-es ábrán a baloldalon látható a szállások részletes nézete. Ezen megjelennek a szálláshely képei, amiket szintén jobbra húzva lehet megtekinteni. A képeken kívül megjelenik még, hogy melyik városban, országban található a szálláshely, a szállás tulajdonosának neve, a maximum férőhelyek száma. Az értékeléseknek egy összegző nézete is megtalálható itt, a csillagok jelölik az összes értékelés átlagát, a zárójelben pedig az értékelések száma látható. A nézet egy gombot is tartalmaz még, amivel az összes értékelést lehet megtekinteni egy új nézeten. A szállás részletes oldalán a jobb felső sarokban található egy naptár ikon. Ennek megnyomásakor egy naptár jelenik meg a nézet felett, amin egy intervallumot lehet kiválasztani, hogy mikor szeretnénk a szálláshelyen tartózkodni. Sikeres dátumválasztás után a nézet alsó sávjában aktívvá válik a foglalás gomb. Ha ezt megnyomjuk, akkor kerülünk a fenti, 2.2-es ábrán látható jobboldali nézetre. Itt egyszerűen csak a foglalás részletei jelennek meg, valamint lehet módosítani, hogy hány fő részére szeretnénk foglalni a szállást. A 2.2-es ábra középső képernyőjén látható az értékelések részletes nézete. A fenti szekcióban grafikusan jelenik meg, hogy hogyan oszlanak meg az értékelések a szálláshelyen, valamint azt, hogy mi az értékelések átlaga. Középen új értékelést tudunk hozzáadni csillagok és komment formájában. A legalsó részen pedig egy lista található, benne az összes eddigi értékeléssel.

Az alkalmazás tartalmaz még egy üzenetküldő funkciót is. Üzenetváltást jön létre két felhasználó között, ha az egyik lefoglalta a másik szállását. Ebben a funkcióban tudják megbeszélni a foglalás további részleteit, a szálláshely elfoglalásának menetét.



2.3. ábra Összes üzenet, valamint egy üzenetváltás nézete

Végül az okoseszközök menedzselését tartalmazó nézetek következnek. Három nézet tartozik ehhez a funkciócsoporthoz. Az elsőn az összes elérhető eszköz található. A másodikon lehet hozzáadni új eszközt. A harmadikon pedig egy meglévő eszköz részleteit lehet megtekinteni, illetve annak tulajdonságait.



2.4. ábra Összes eszköz megtekintése, új hozzáadása, részletek megjelenítése

A 2.4-es ábra baloldali nézetén lehet megtekinteni az összes eszközt. Ezek szállásonként csoportosítva vannak. A szállásokkal egy vonalban található egy + gomb, ezt megnyomva lehet továbbmenni a középső nézetre. Az összegző nézeten találhatók az egyes kapcsolók, amik aktuális állapota szövegesen, illetve színekkel is ábrázolva van. Minden kapcsolón van egy nyíl ikon, aminek megnyomásával lehet a részleteket megtekinteni a jobboldali nézeten. A hozzáadási képernyőn meg lehet adni egy tetszőleges nevet, ki lehet választani még egyszer, hogy melyik szálláshoz szeretnénk hozzáadni, illetve az is megadható, hogy milyen típusú eszközt szeretnénk felvenni. Annak alapján, hogy milyen típust választottunk, az alsó listában a kapcsolók és szám beviteli mezők automatikusan frissülnek. A felsorolt kapcsolókkal kezdeti értékeket adhatunk meg az újonnan hozzáadott eszközhöz. A 2.4-es ábra jobboldali, részletes nézete nagyon hasonló a létrehozási űrlaphoz, de ezen a képernyőn már nem lehet átállítani a szálláshelyet, sem pedig az eszköz típusát.

3 Felhasznált technológiák áttekintése

A dolgozat keretein belül elkészített rendszer prototípusához a jelen fejezetben bemutatott technológiákat használtam fel. Mindegyik itt felsorolt technológia nagyon hasznos funkciókkal és megoldásokkal rendelkezik, amelyek megkönnyítik a helyes szoftverek tervezését és megvalósítását. A technológiák kiválasztásánál azokat részesítettem előnyben, amelyekkel még nem találkoztam korábban, hogy minél több új tapasztalatot szerezzek ezeknek a megismerésével.

3.1 iOS

Az iOS az Apple által fejlesztett operációs rendszer, amelyet az iPhone-ok és iPad-ek használnak. Manapság már csak az iPhone-ok operációs rendszere ez, ugyanis 2019 óta az iPad önálló operációs rendszert kapott iPadOS néven. Az iOS 2007-ben jelent meg az első iPhone-al együtt.

Az iOS a megjelenése óta rendkívül népszerű operációs rendszer lett, napjainkban a mobil operációs rendszerek piaci részesedésének 27,49%-éval rendelkezik. Ezzel a számmal az Android után ez a második legtöbb felhasználóval rendelkező operációs rendszer. [1] Az iOS alkalmazások fejlesztésnek rengeteg előnye és hátránya is van az Android fejlesztéssel szemben. Ezek mellett szerintem csak személyes preferencia kérdése, hogy ki melyik platformra szeret inkább fejleszteni.

Az iOS nagyobb előnyei viszont az Androiddal szemben, hogy sokkal kisebb az operációs rendszerek közötti fragmentáció. Az Apple alkalmazásboltjának (App Store) statisztikái szerint az iPhone-ok 82%-a használ iOS 15-ös verziót, ezek közül is az elmúlt 4 évben kiadott eszközök 89%-a használja még ezt a verziójú operációs rendszert. [7] Másik nagy előnye, hogy ugyan sok különböző kijelzőméretű iPhone létezik, ezeknek a száma sem olyan magas, mint az Android esetében. Tehát az iOS alkalmazások fejlesztése során sokkal kevesebb operációs rendszert, illetve kijelzőfelbontást kell támogatni, mint a versenytárs esetében.

Az előnyök mellett természetesen a hátrányokat is meg kell említenem. iOS-ra készült alkalmazásokat csak és kizárólag macOS operációs rendszer alatt tudunk fejleszteni. Ez a rendszer nem mindenki számára elérhető, valamint rengetegen vannak olyanok is, akik nem szeretnek ezzel a rendszerrel dolgozni. Egyéb hátrány még, hogy az

App Store-ban az alkalmazások megjelenése nagyon szigorú szabályok mellett lehetséges csak. Amikor elkészül egy alkalmazás, akkor a szoftvert be kell küldeni az Apple tesztelőinek, akik megnézik, hogy maga az alkalmazás működésben megfelel-e az App Store irányelveinek.

3.2 Swift programozási nyelv

A Swift nyelv egy Apple által fejlesztett általános célú programozási nyelv, amely modern megközelítéseket használva készült el. A nyelv 2014-ben jelent meg, ami az Apple által korábban használt Objective-C nyelvet akarta kiváltani. Az Objective-C egy C-hez hasonló objektumorientált nyelv. A Swift nyelvben található modern megközelítések a biztonságos kódolás, a megfelelő teljesítmény és az erős kifejező erővel bírás voltak. A fejlesztőknek a legfőbb céljuk a nyelvvel az volt, hogy a régi C-n alapuló nyelvekre alternatívát nyújtsanak, és minél több területen lehessen a nyelvet használni, legyen szó akár mobil és asztali alkalmazásfejlesztésről, vagy éppen webes alkalmazásokról. [8]

3.2.1 Biztonság, típusosság

Azt a megközelítést, hogy a Swift arra lett tervezve, hogy biztonságos kódot írjunk több nyelvi elem és tulajdonság is alátámasztja. Például a fordító biztosra megy azzal kapcsolatban, hogy minden változó inicializálva van használat előtt, a tömböket és integer típusokat túlcsordulás miatt folyamatosan ellenőrzi, illetve automatikus memóriakezelést is használ. Másik biztonsági tulajdonság, hogy a nyelv statikusan típusos. Ezt azt jelenti, hogy változóknak és konstansuknak létrehozástól kezdve konkrét típusuk van, amit nem lehet megváltoztatni.

```
var str = „Hello World!” // string
var double = 3.14 // double
```

A nyelv típusokra való kényelmi funkciója a Type inference, aminek a lényege, hogy a változókhoz nem kell explicit kiírnunk a típusukat, mert a fordító megpróbálja kitalálni a típust a kezdeti értékéből.

```
func add(a: Int, b: Int) -> Int { return a+b }

add(1, 2) // OK
add(1, 3.0) // Error
```

A nyelv típusrendszerére vonatkozik még, hogy az erősen típusos. Ez azt jelenti, hogy sehol nincsen automatikus konverzió, minden ilyen átalakítást explicit jelölni kell a

kódban. Például egy olyan függvénynek, ami Int típusú paramétert vár, nem adhatunk meg Double paramétert, mert akkor az fordítási időben hibát fog nekünk jelezni. [9]

3.2.2 Optionals

Ahogy korábban említettem a fordító gondoskodik arról, hogy minden változó kapjon értéket mielőtt használnánk azokat. Ez nem minden esetben igaz, ugyanis a nyelvben be lett vezetve egy új funkció Optionals (Opcionálisok) néven. Az Optional egy olyan speciális típus, ami engedélyezi egy változó számára, hogy annak ne legyen értéke. Swift esetében egy érték hiányát a nil értékkel lehet jelölni.

```
var str: String?
str = „Ertek”
str = nil
```

Optional típusú változót a ? operátorral tudunk deklarálni. Ennek, ha nem adunk meg kezdeti értéket, akkor nil lesz az értéke. Ahhoz, hogy a változóból megkapjuk az értéket valahogy ki kell csomagolnunk. Kicsomagolásra több megoldást is nyújt számunkra a nyelv, ez egyik ilyen például a Force Unwrapping:

```
var str: String? = „Ertek”
str! // OK
str = nil
str! // App will crash
```

Az érték „kierőltetését” a változóból a ! operátorral tudjuk megtenni. Ezt az operátort azonban csak abban az esetekben szabad használni, ha biztosan tudjuk, hogy van érték a változóban. Ha nincsen benne érték és használjuk az operátort, akkor le fog állni a programunk futás idejű hibával.

Egy másik megoldás az érték kinyerésére a Conditional Unwrapping:

```
var str: String? = „Ertek”
if let ertek = str {
     // here str was not nil, value is in variable ertek
} else {
     // here str was nil
}
```

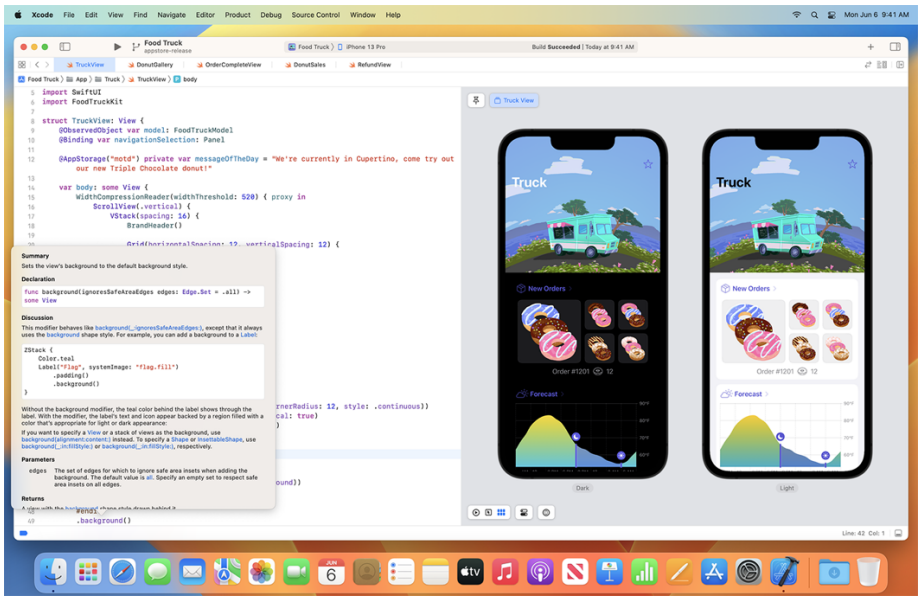
Ez egy nagyon egyszerű és kézenfekvő megoldás. Egy hagyományos if-else blokkban leellenőrizzük, hogy tartalmazott-e értéket az Optional változónk. Az if-let használatával egy új változóba ki is tudjuk venni az Optional értékét, ha volt neki. Ekkor szabadon hozzáférünk az értékhez az if-en belül, nem kell további kicsomagolást elvégeznünk rajta. [10]

Harmadik megoldás az Optional használatra pedig az Optional Chaining:

```
var str: String?
str?.capitalized
```

Ez a megoldás biztonságos és kényelmes módja az Optional használatának. Ha egy Optional-nek szeretnénk meghívni egy metódusát vagy elérni az egyik adattagját, akkor a ? operátorral kicsomagolhatjuk az Optional-t. Ebben az esetben, ha nem volt értéke, akkor a művelet egyszerűen nem hajtódik végre, nem fog futási idejű hibát produkálni a program.

3.3 SwiftUI



3.1. ábra Szerkesztőfelület és előnézet az Xcode-ban[3]

A SwiftUI az Apple új UI technológiája, ami az iOS 13-as verzióval jelent meg 2019-ben. A SwiftUI keretrendszerrel deklaratív módon tudunk felhasználói felületeket készíteni minden Apple eszközön futó alkalmazásunk számára. Ezzel az egy technológiával tehát képesek vagyunk iOS, iPadOS, macOS, tvOS és akár watchOS alkalmazásoknak is elkészíteni a felületét. A keretrendszerhez tartozik még az Xcode-ban elhelyezett grafikus tervezőnézet is, amely az általunk SwiftUI-al írt felületi kód alapján képes előnézetet generálni az alkalmazásról anélkül, hogy azt el kelljen indítanunk egy valódi eszközön vagy szimulátoron. [11]

[3] Kép forrása: https://developer.apple.com/xcode/swiftui/

Más modern UI technológiákhoz hasonlóan egy SwiftUI nézet is az általa megfigyelt állapotok alapján frissíti be önmagát. A fejlesztőknek egyáltalán nem kell direktben befrissíteniük a UI-t, elég csak ha a felülethez hozzákötött modell állapotát változtatjuk, a keretrendszer pedig majd a megváltozott állapot alapján eldönti, hogy melyik részét szükséges frissítenie a nézetnek. Ezt az állapotfigyelési funkciót többféleképpen is meg tudjuk valósítani egy SwiftUI nézetnél.

```
struct ContentView: View {
     @State private var tapCount = 0

     var body: some View {
          Button(„Tap Count: \(tapCount)”) {
               tapCount += 1
          }
     }
}
```

A fenti kódrészlet a legegyszerűbb módját mutatja be az állapotfigyelésnek. A nézet tartalmaz egy tapCount változót, amit a @State property-vel vettünk fel. A property által a változó menedzselését átadtuk a SwiftUI-nak, ami addig tartja a memóriában a változót, amíg a nézet létezik. A property-vel azt is megjelöltük még, hogy a SwiftUI-nak frissítenie kell a nézetet, ha a változónak új értéket adtunk. Tehát a fenti kódban a gomb szövege mindig frissülni fog, ha megnyomtuk a gombot. A @State-et használó megoldás akkor nagyon hasznos, hogy ha egyszerű változóink vannak, a változóinkat csak a nézet használja és soha nem kerülnek ki a nézeten kívülre.

Komplexebb objektumoknál, amik mondjuk saját típusok és több adattagjuk van egyszerre, érdemesebb az @ObservedObject property-t használni a deklarálásnál. Ez a megoldás hasonlít az előzőhöz, viszont annyi különbség van benne, hogy az @ObservedObject-el létrehozott objektumoknál nekünk kell menedzselni a belsejében történő dolgokat, például létrehozni belőle egy példányt vagy megfelelően inicializálni az adattagjait. Az ilyen objektumoknak meg kell valósítaniuk még az ObservableObject protokollt ahhoz, hogy együtt tudjanak működni a SwiftUI nézetekkel. Az ilyen protokollt megvalósító osztályok esetében a @Published property-vel tudjuk ellátni azokat az adattagokat, amik változása esetén szeretnénk frissíteni az objektumot figyelő nézetet. [12]

3.4 Firebase

A Firebase egy alkalmazásfejlesztői platform, amely eszközök megosztásával segíti a fejlesztőket, hogy egyszerűbben és gyorsabban tudjanak mobil és webes alkalmazásokat fejleszteni. A platform rengeteg olyan szolgáltatást bocsát a fejlesztők rendelkezésére, amelyeket alapesetben nekünk kellene megterveznünk és megvalósítanunk ahhoz, hogy az alkalmazásainkban fel tudjuk használni. A Firebase számos szolgáltatást tartalmaz, amelyek nem csak az alkalmazások elkészülésében segítenek, hanem akár a kiadási folyamatban is. Kiadás pedig figyelhetjük is a platformon keresztül az elkészült alkalmazásunkat, ahol statisztikákat láthatunk esetlegesen előforduló hibákról. A Firebase számos útmutatóval és dokumentációval segíti a szolgáltatásainak beüzemelését, ezért a platform használata meglehetősen egyszerű.

3.4.1 Realtime Database

A Firebase Realtime Database egy felhőalapú, NoSQL adatbázis. Az adatokat JSON struktúrában tárolja, ezért adatsémára vonatkozóan nem köti semmilyen szigorú szabály a fejlesztőket. Az adatokat lokálisan is tárolja minden kliensnél, ezért abban az esetben, amikor az alkalmazás offline módba kerül, akkor is működőképes marad. A felhőben viszont továbbra is frissülnek az adatok a többi kliens által, emiatt ha újra online lesz alkalmazás, akkor fogja befrissíteni a lokális adatbázist a felhőben lévővel.

Az adatbázis a Realtime Database Security Rules szolgáltatással lehet konfigurálni. Ebben megadhatjuk, hogy milyen struktúrában szeretnénk tárolni az adatokat, validációkat is elvégezhetünk vele az adatokon, illetve innen szabályozhatjuk, hogy ki tudja írni és olvasni az adatokat. Firebase-es authentikációval is együtt tud működni. Olyankor azt is megadhatjuk, hogy melyik felhasználó milyen adatokhoz férhet hozzá. [13]

Használata úgy működik, hogy miután a projektünkbe beletettük a szükséges könyvtárakat, utána létre kell hoznunk egy DatabaseReference objektumot, aminek meg kell adnunk egy URL-t, ami az adatbázisunkra mutat. Az URL végére be kell még illesztenünk annak a csomópontnak a nevét, ami alól lekérdeznénk vagy ahová beszúrnánk az adatokat. Egy ilyen csomópont a relációs adatbázisoknál egy táblának felel meg.

```
let couchRef = self.databaseRef?.child(couch.id)
couchRef?.setValue(couch.toAnyObject())
```

A fenti példa egy új objektum adatbázisba való beszúrását mutatja meg. Az említett DatabaseReference objektumon meghivatkozunk egy új gyerek csomópontot egy ID-val, ide fog majd kerülni a beszúrandó objektum. Ennek a gyereknek pedig meg kell hívni a setValue függvényét, amivel be is szúrjuk az adott helyre az objektumot. A setValue metódus egy Dictionary-t vár paraméterként, ami kulcs-érték párokban tartalmazza az objektum tulajdonságait.

```
databaseRef?.observeSingleEvent(of: .value, with: { snapshot in ... }
```

A fenti kódsor egy adatbázis lekérdezést mutat meg. A függvényhívás lekérdezi a referenciacsomópont összes gyerekét, majd egy closure paramétereként adja ezeket vissza. Egyéb lekérdezéseket, amelyek szűkebb megoldáshalmazt adnak vissza, a queryOrdered és a queryEqual fügvényekkel lehet megadni úgy, hogy a referenciaobjektumon hívjuk meg egymás után fűzve.

3.4.2 Storage

A Storage egy Firebase által üzemeltetett felhőalapú fájltárhely. Elképzelésben, működésben és használatban nagyon hasonló a Realtime Database-hez. Képeket, videókat vagy egyéb fájlokat tudunk ide feltölteni, illetve letölteni kliensalkalmazásokból. Security Rules funkció itt is elérhető, tehát a fájlok kezelésénél is meg tudjuk szabni, hogy kik férhetnek hozzá és kik nem a fájlokhoz. Authentikációval is szintén együttműködik, tehát bejelentkezett felhasználók szerint is adhatunk jogosultságokat. [14]

```
storageRef?.child(filePath).putData(data, metadata: nil) { (_, error) in
     if error == nil {
          self.storageRef?.child(filePath).downloadURL() { url, error in
               if error == nil {
                    completion(url?.absoluteString ?? „”)
               }
          }
     }
}
```

A fenti példában egy fájlfeltöltés látható. Használatban nagyon hasonló a Realtime Database-hez. Itt is van egy referenciaobjektum, aminek meg kell hivatkozni egy csomópontját, amihez feltölthetjük az adatot. Különbség viszont, hogy például képfeltöltés után a kép URL-jét le kell töltenünk külön, mert később csak ezzel lehet majd hivatkozni a fájlra.

3.4.3 Authentikáció

A Firebase Authentication szolgáltatás egy egyszerű authentikációt valósít meg a fejlesztők számára, amit felhasználhatnak saját alkalmazásukban. A Firebase többfajta authentikációs módszert is biztosít, például email-jelszó alapút, telefonszámmal történőt, vagy akár third-party szolgáltatókon keresztüli azonosítást is. A többi Firebase-es szolgáltatással szorosan együttműködik, tehát a felhőben elmentett felhasználói adatok védelmét is biztosítja. Működését tekintve a kliensalkalmazás elküldi a Firebase SDK-nak a felhasználói adatokat vagy egy OAuth tokent, amivel a Firebase végrehajta az azonosítást és visszaadja a művelet sikerességét. Alapvető felhasználókezelést is megvalósít a szolgáltatás, tehát email cím, név vagy egyéb egyszerű felhasználóspecifikus adat tárolását is megoldhatjuk egyéb Firebase funkció használata nélkül. [15]

3.5 Vapor

A Vapor egy nyíltforráskódú webes keretrendszer, amit Swift nyelven írtak. Az egész keretrendszer aszinkron módon működik, ezt a fejlesztők úgy érték el, hogy a keretrendszert az Apple által fejlesztett SwiftNIO könyvtárra építették. A Vapor segítségével készíthetünk back-end alkalmazásokat applikációk számára, illetve önállóan működő webalkalmazásokat. A keretrendszert 2016 óta fejlesztik és azóta ez a legnépszerűbb szerver oldali Swift keretrendszer.

A keretrendszer rengeteg könyvtárat tartalmaz, amikkel számos különféle funkciót kapunk a webalkalmazásunk fejlesztéséhez. A két legnépszerűbb és legtöbbet használt ilyen könyvtár a Leaf és a Fluent. A Leaf egy sablongeneráló könyvtár, amivel webes nézeteket tudunk generálni a kliensoldali webalkalmazásunkhoz. A Fluent pedig egy ORM (Object Relational Mapping) könyvtár, ami számos natív és aszinkron adatbázis drivert tartalmaz. [16]

3.5.1 Fluent

A Fluent egy Vapor keretrendszer számára fejlesztett ORM könyvtár. Ez a könyvtár egy absztrakciós réteget képez egy Vapor alkalmazás és maga az adatbázis között. ORM rétegek használatával az adatbázisok használata sokkal egyszerűbb, hiszen egy nagyon egyszerű API-n keresztül tudunk hozzáférni az adatbázishoz. Ezen a rétegen

keresztül nem kell közvetlenül az adatbázissal kommunikálnunk és lekérdezéseket sem kell írnunk hozzá.

A Fluent számos adatbázis típust támogat, legyen szó akár relációs adatbázisokról vagy NoSQL adatbázisokról. A könyvtár elfedi a fejlesztők elől az adatbázisra vonatkozó információkat, ezért Fluent-et használva nem fogja a fejlesztő érezni, hogy milyen típusú adatbázisban tárolja az adatait. Ennek ellenére akkor tudjuk kihasználni a legjobban a könyvtárat, ha ismerjük, hogy milyen adatbázis van alatta és tudjuk, hogy hogyan kell egy olyan adatbázist használni. [17]

A Fluent rengeteg eszközt biztosít, amivel egyszerűen ki tudjuk alakítani az alkalmazásunk adatrétegét. Az adatbázisban tárolt típusok létrehozásához a könyvtár által megadott Model osztályból leszármazva tudunk. Ennek a belsejében az adattagokhoz property wrapper-eket tudunk hozzáadni, amik megadják, hogy milyen szerepe van az adattagoknak a tárolás szempontjából.

```
final class Acronym: Model {
     let schema = „acronyms”

     @ID
     var id: UUID?

     @Field(key: „short”)
     var short: String

     ...
}
```

Az @ID property-vel tudjuk jelölni a táblában a kulcsot, a @Field-el pedig az egyszerű mezőkkel. A @Field esetében adhatunk meg egy név alternatívát, hogy milyen névvel tárolja az adatbázistáblában az adott oszlopot. Létezik még @Parent, @Child, @Children property wrapper-ek is. Ezekkel az egy-egy, illetve az egy-több kapcsolatot tudjuk jelölni táblák között. A több-többes kapcsolatot a @Siblings property-vel lehet leképezni, azonban ehhez nem elég önállóan ezt a jelzőt megadnunk. A több-többes kapcsolat létrehozásához egy kapcsolótáblát is létre kell hoznunk, amit az adatbázis egy külön táblában fog tárolni. Ebben a kapcsolótáblában kell felvenni az összekapcsolni kívánt típusokat, mindkettőt @Parent property-vel ellátva.

```
struct CreateAcronym: Migration {
     func prepare(on database: Database) -> EventLoopFuture<Void> {
          database.schema(„acronyms”)
          .id()
          .field(„short”, .string, .required)
          .create()
     }

     func revert(on database: Database) -> EventLoopFuture<Void> {
          database.schema(„acronyms”).delete()
     }
}
```

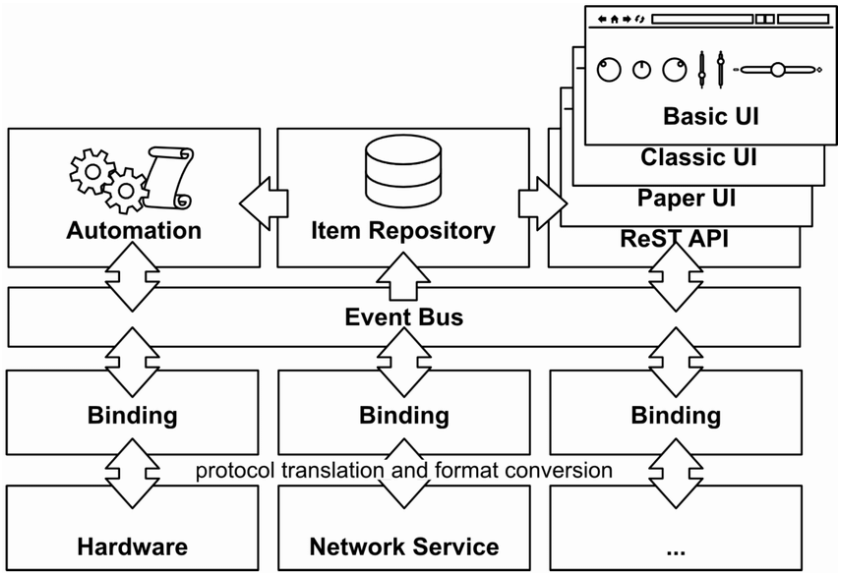
Az adatréteg létrehozásához még kell készítenünk minden típushoz egy-egy migrációt. Ezek különálló osztályok, amelyek a Migration protokollt valósítják meg. Két metódusa van a protokollnak, az egyik a prepare, a másik pedig a revert. Az előbbi akkor fut, amikor felmigráljuk az adatbázisunkat, utóbbi pedig amikor töröljük a migrációkat. A prepare függvényben fel kell sorolnunk, hogy az adott tábla milyen mezőkkel rendelkezik. A revert függvényben elég csak a táblán a delete metódust meghívni, a többit meg elintézi nekünk a Fluent.

3.6 openHAB

3.6.1 Általános bemutatás

Az egyes okoseszközök vezérlésére az openHAB nevű, nyílt forráskódú, technológia-független automatizációs platformot használtam, amely kifejezetten okosotthonban található eszközök menedzselésére lett kitalálva. A platform több gyártó több száz termékét támogatja, valamint más népszerű okosotthon rendszerekkel is együtt tud működni, pl. Amazon Alexa, Google Assistant vagy Apple HomeKit.

A platform JVM-en (Java Virtual Machine) fut, ezért a legtöbb operációs rendszerre telepíthető, amely rendelkezik internet kapcsolattal. Feltelepítése nagyon egyszerű és gyors, csupán a gyártó honlapjáról le kell tölteni az aktuális stabil verziót, kicsomagolás után pedig az operációs rendszerünknek megfelelő script fájllal elindítható az openHAB szerver. Különböző Add-On-ok (openHAB-on Binding-ként hivatkoznak rá) telepítésével bővíthetjük, hogy a saját openHAB szerverünk milyen gyártójú eszközökkel vagy webszerverekkel tudjon együttműködni. Add-On-ok bármikor telepíthetők vagy törölhetők, nem muszáj mindent az első indításkor feltelepíteni. A Binding-ok sikeres telepítése után a rendszer automatikusan feltérképezi vannak-e már okosotthon eszközök a hálózatunkon, találat után pedig hozzá is adja őket a rendszerhez.



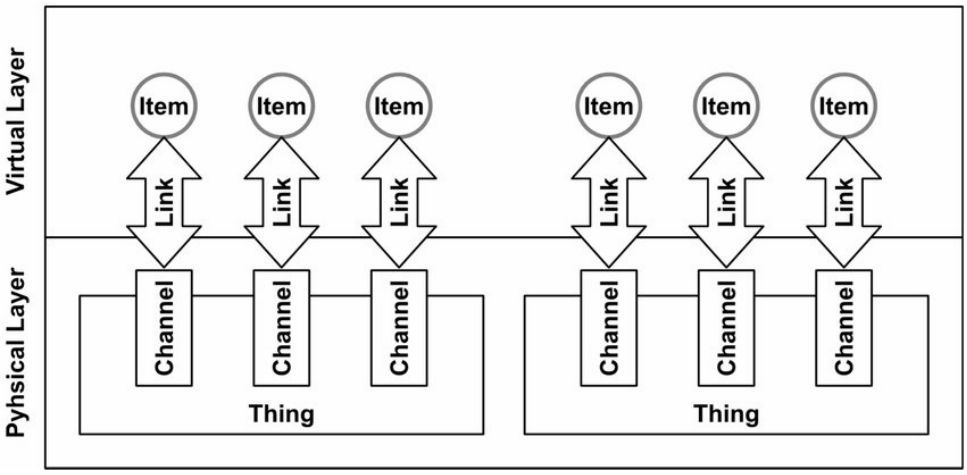
3.2. ábra Az openHAB rendszer architektúrája[4]

Az openHAB-hoz tartozik egy böngészőből elérhető UI felület, amelyen a felhasználók könnyen elérhetik eszközeik adatait és elvégezhetik a szükséges konfigurációkat. A felület mellett a platform biztosít egy REST API-t is, melyen keresztül minden funkcionalitást elérhetünk, amit a felhasználói felületen is. [18]

3.6.2 Architektúra rétegei

Az előző szakaszban említettem, hogy egy fizikai eszközhöz (Thing), akkor tudunk hozzáférni, ha feltelepítettük az eszköz gyártója által kibocsátott openHAB Binding-ot. Ennek a Binding-nak a segítségével tudja a rendszer beazonosítani a fizikai eszközöket, azok tulajdonságait és állapotait, valamint a hozzájuk tartozó parancsok listáját. Minden Thing csatornákat (Channel) definiál, amik leírják az eszköz különböző funkcióit. Például egy villanykapcsolót fel és le lehet kapcsolni, tehát ennek lesz valamilyen kapcsoló (Switch) típusú csatornája, amin keresztül le lehet kérdezni az állapotát, valamint frissíteni lehet azt. Számos csatornatípus van a platformon, ezekből a legtöbbet használt a Switch és a Number. A Number típusból több fajta van, ez előre leírja, hogy milyen kontextusban kell értelmezni az adott számot. Például egy légkondicionálónak lehet Number:Temperature csatornája, amin keresztül a beállított hőmérsékletet lehet lekérdezni. [18]

[4] Kép forrása: https://atlas.informatik.uni-tuebingen.de/~menth/papers/Menth17i.pdf



3.3. ábra A virtuális és fizikai réteg[5]

A platform fizikai rétegében tehát a Thing-ek és a Channel-ek helyezkednek el, a virtuális réteget pedig az Item-ek alkotják. Az Item-ek képviselik a fizikai eszközöknek egy kis részét, amit a felhasználó éppen látni akar belőle. Ezek akkor lehetnek nagyon hasznosak, ha több olyan eszközünk van, aminek csak pár funkcióját akarjuk használni és nem az összeset. Ilyenkor a szükséges funkciókhoz fel tudunk venni Item-eket, amiket hozzá lehet kapcsolni (Link-elni) a megfelelő csatornákhoz. Ezek a Link-ek kapcsolják össze a fizikai réteget a virtuálissal, ezeken keresztül férünk hozzá az eszközök csatornáihoz. Az Item-en történő beállításokat a Link küldi tovább az eszközöknek és az eszközön történő változásokról is a Link küld értesítést az Item-nek. [19]

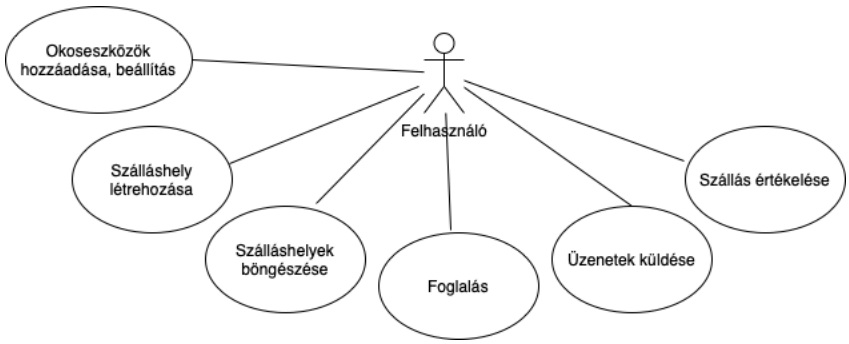
[5] Kép forrása: https://atlas.informatik.uni-tuebingen.de/~menth/papers/Menth17i.pdf

4 Tervezés

Az alkalmazás tervezése során a feladat témájának és támpontjainak megfelelően kellett megalkotnom a fő funkciók specifikációját és az applikáció felhasználói felületét. Az alkalmazás működéséhez egy adatmodellre is szükségem volt. A tervezési folyamatok során a különböző architektúrák kiválasztása, valamint a komponensek közötti kommunikáció kitalálása is nagy hangsúlyt kapott. A következő alfejezetekben az említett tervezői folyamatokat és azok eredményét fogom részletesen ismertetni.

4.1 Funkciók definiálása

A tervezés első szakaszában próbáltam meghatározni, hogy milyen funkciókra lenne szükség egy szálláskiadó alkalmazásban. Nyilván a legtöbb funkciót egyértelműen meg tudtam határozni, mivel számos más ehhez hasonló szolgáltatás vagy alkalmazás létezik már, és párat már használtam is belőlük. Mindenesetre összegyűjtöttem ezeket a funkciókat, mert úgy láttam, hogy ez megfelelő irányvonalat ad majd nekem a felületi tervek, illetve az adatmodellek elkészítéséhez.



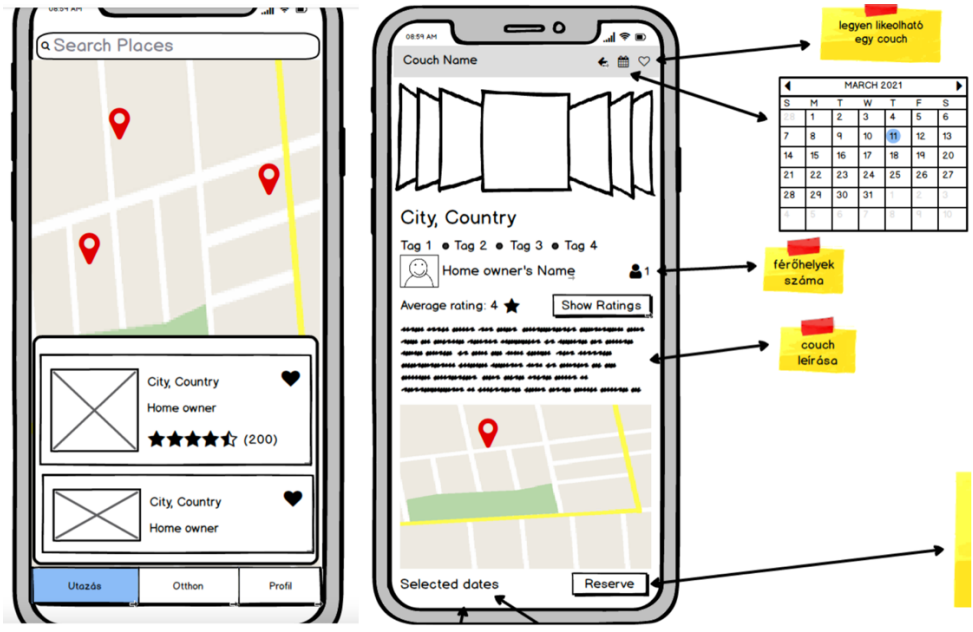
4.4.1. ábra Alkalmazás use-case diagramja

Az alkalmazás felhasználóit nem szerettem volna különböző csoportokra bontani (például szállásadók és vendégek), mivel az egyes csoportok tagjai nem biztos, hogy csak egyféleképpen fogják használni az alkalmazást. Emiatt egy felhasználói csoportot határoztam még, ami az átlag felhasználók csoportja lett. Ez a csoport hozzáfér az alkalmazás összes funkciójához, mint például bejelentkezés és regisztráció, szálláshelyek böngészése és létrehozása, szállás foglalása és értékelése, valamint üzenetváltás vendégek és szállásfogadók között.

4.2 Felületi tervek elkészítése

Az alkalmazás felületi terveihez először wireframe-eket (drótvázakat) készítettem. A wireframe-k tervrajzai a felhasználói felületnek, egyáltalán nem a végleges felületet írják le, inkább annak struktúráját. Ezekkel a kis rajzokkal jól lehet szemléltetni még milyen adatok fognak megjelenni egy képernyőn vagy milyen funkciót szolgál az adott nézet. A drótvázak általában egyszerűsített tervek, színeket vagy nagyon részletes és bonyolult vezérlőket nem tartalmaznak. Mivel ezeket könnyű szerkeszteni, ezért érdemes előre elkészíteni őket, mielőtt implementálnánk a kész felületi terveket, mert azokban sokkal nehezebb változtatásokat átvinni, mint a wireframe-kbe. [20]

Drótvázakból három féle típus létezik. A tartalom drótvázban nagyobb térbeli blokkok helyezkednek el, amelyek különböző kategóriájú tartalmak számára vannak fenntartva. Az ilyen blokkok belső felépítését nem tartalmazza ez a drótváztípus, csak azt mutatja meg, hogy nagyjából hol fognak elhelyezkedni az egyes elemek. A lo-fi drótvázak az előző típusra épülnek és megmutatják, hogy a tartalom drótvázban található blokkokon belül milyen elemek hogyan helyezkednek el pontosan. A harmadik típus az interaktív drótváz, amely tartalmazza, hogy az egyes nézetek között hogyan fog majd működni a navigáció.



4.2. ábra Drótvázak

A drótvázak elkészítéséhez a Balsamiq Mockups nevű szoftvert használtam. Ez a szoftver egyszerű felhasználói felületével kiváló ilyen wireframe-k készítésére. A

szoftver rendelkezik egy alap ábrakészlettel, ami az alapvető vezérlőket tartalmazza, de beépített könyvtárával tudunk további vezérlőábrákat hozzáadni a projektünkbe, pontosan úgy mintha külső könyvtárakat adnánk hozzá egy készülő szoftverhez. Ezt a wireframe készítő eszközt azért találtam még nagyon hasznosnak, mert ha összekötjük a gombokat az egyes nézetekkel, valamint kiexportáljuk a terveket PDF formátumba, akkor élőben is kipróbálható, hogy melyik nézetről melyik további nézetekre tudunk majd navigálni az elkészült alkalmazásban.

A tervek elkészítésekor tartalom drótvázakat nem, csak lo-fi drótvázakat csináltam, amit végül a Balsamiq Mockups képességeit kihasználva interaktív drótvázzá alakítottam. Az alapvetően definiált funkciókhoz készítettem első körben drótvázakat. A tervek készítésekor a képernyők mellé jegyzeteket készítettem bizonyos követelményekről, amiket nem lehetett jól ábrázolni a drótvázon, viszont fontosak voltak funkcionális szempontból. Ezeken kívül workflow magyarázatokat, illetve kisebb ábrák szöveges kifejtését írtam le a nézetek mellé. Mivel a tervezésnek ebben a szakaszában még az adatmodell sem volt kitalálva, ezért a felületeken megjelenő adatokból próbáltam összegyűjteni, hogy milyen táblákra, azon belül pedig, hogy milyen oszlopokra lesz majd szükségem.

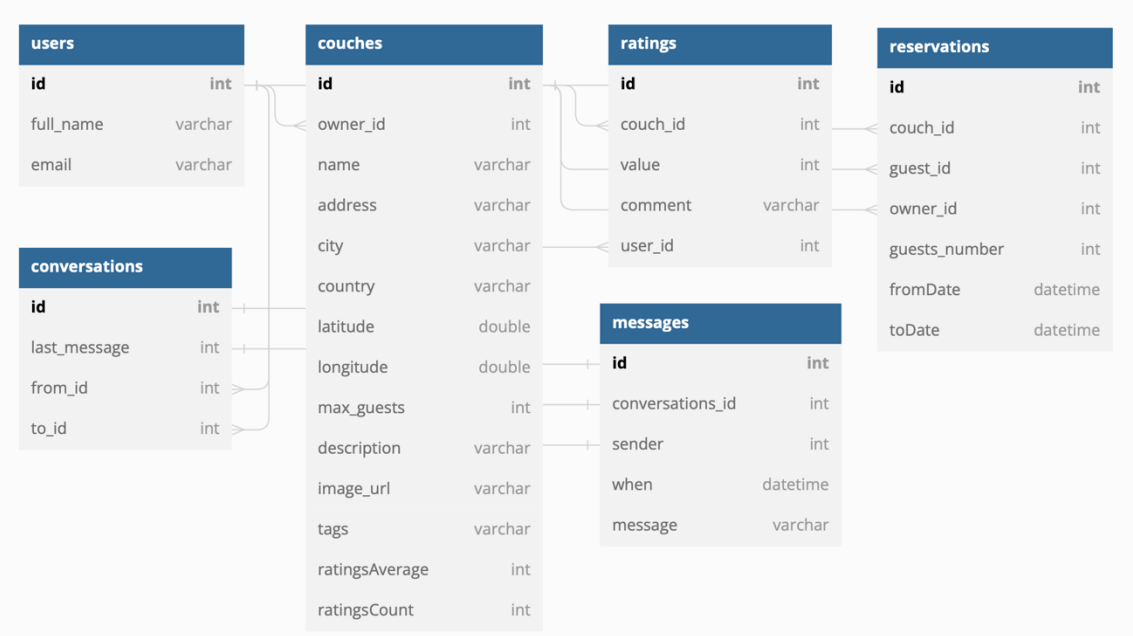
A későbbiekben az újonnan kitalált funkciók tervezésekor csak akkor készítettem drótvázakat, ha nem volt számomra sem egyértelmű, hogy pontosan hogyan jelenítsek meg bizonyos információkat a felhasználó számára.[6]

4.3. ábra Drótvázak

[6] További, itt be nem mutatott drótvázak elérhetőek a dolgozat mellékletében.

4.3 Adatmodell elkészítése

Az adatmodell elkészítésénél törekedtem arra, hogy minél kevesebb lekérdezést kelljen majd elvégeznem az alkalmazásban. Emiatt egyes táblákba bekerültek származtatott adatok, amelyeket egyéb esetben más táblák adataiból tudnék kiszámolni. Ilyen adatok például a Couches táblában az értékelésekre vonatkozó tulajdonságok vagy a Conversations tábla last_message oszlopa. Az ilyen fajta tárolás több szempontból sem szerencsés megoldása a problémának. Igaz, hogy kevesebb lekérdezésből jutok hozzá ezekhez az adatokhoz, de redundáns adattárolást viszek be a rendszerbe, ezért több helyen kell majd frissítenem a táblákat egy rekordmódosítás vagy beszúrás esetén. Egy elfelejtett hívással ez egyszerűen el is rontható, amivel pedig inkonzisztenciát is bevihetek az adatbázisba. A felsorolt problémák miatt az implementáláskor igyekeztem a redundanciát minimálisra csökkenteni, illetve teljesen megszüntetni.



4.4. ábra Adatmodell

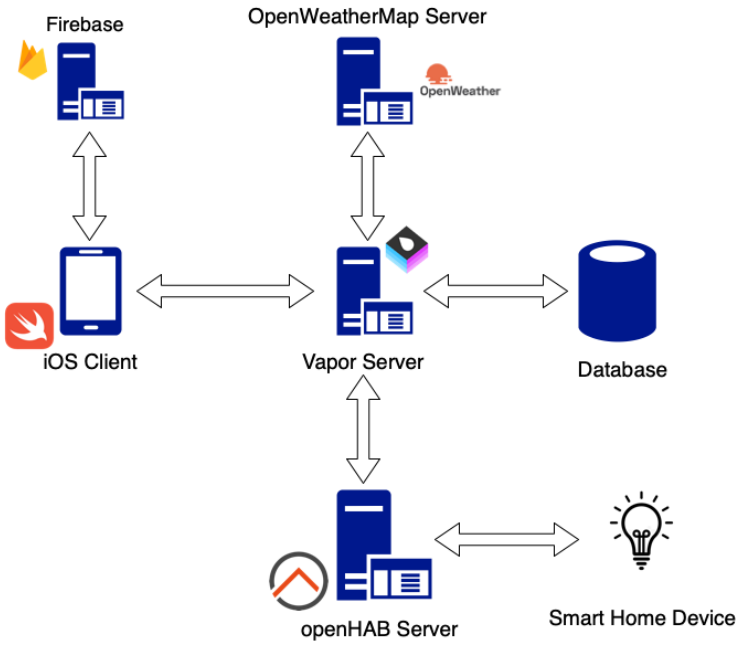
Az adatmodellt a dbdiagram.io[7] online eszközzel készítettem el. Ez az eszköz nagyon hasznos adatmodellező, szövegesen, SQL-szerű szintaxissal írhatjuk le a táblákat, amiket egy grafikus nézeten leírás után rögtön megtekinthetünk vagy mozgathatunk a jobb átláthatóság kedvéért. A kész modellből tudunk PDF vagy SQL fájlt exportálni, így bárkinek meg tudjuk mutatni, hogy hogyan néz ki grafikus formában a modellünk. Az

[7] https://dbdiagram.io/home

SQL-t vissza is lehet importálni az eszközbe, tehát ha félbehagytuk a munkánkat akkor, az előző állapottól lehet azt folytatni. Az SQL-t természetesen az adatbázis létrehozásánál is tudjuk használni, szóval ezt nem kell kétszer megírnunk.

4.4 Rendszer magas szintű architektúrája

A rendszer a felépítését tekintve egy kliens-szerver architektúra. Központi eleme egy Vapor keretrendszeren alapuló szerver, amit én készítettem. A szerver egy relációs adatbázisban tárolja az adatait, ezen kívül HTTP kéréseken keresztül kommunikál a klienssel és egyéb más webszerverekkel.



4.5. ábra A rendszer magas szintű architektúrája

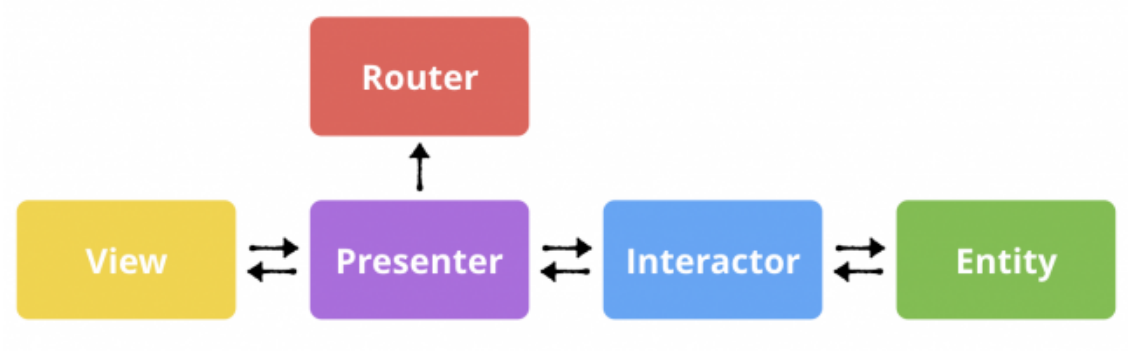
A kliens egy iOS platformon futó, natív mobilalkalmazás. Eleinte a rendszer architektúrája sokkal kevesebb elemmel rendelkezett, nem volt saját magam által készített szerver oldali komponens, ezért a mobilalkalmazás a Firebase szolgáltatásait vette igénybe. Az adattárolás, fájlok tárolása, illetve a felhasználókezelés és authentikáció volt Firebase használatával megvalósítva. A saját szerver oldali komponens megvalósításakor nem volt előírás arra, hogy minden egyes Firebase szolgáltatást lecseréljek sajátra, ezért az authentikáció, illetve a fájlok kezelése a komponensek kidolgozottsága miatt nem került át az én Vapor szerverem funkciói közé.

Az okosotthon funkciók bevezetésekor bővítettem még az architektúrát, két 3rd party webes szolgáltatással. Az egyik ilyen szolgáltatás egy openHAB szerveren futó

webalkalmazás volt, aminek a felelőssége a hozzá beregisztrált okoseszközök nyilvántartása és elérése. Az openHAB szerveren keresztül tudom elérni ezeket az okoseszközöket, adatokat lekérdezni róluk (például tulajdonságaikat, állapotukat) és parancsokat kiadni, amiket az openHAB majd továbbít az eszközök felé. A másik webszolgáltatás, amit még beépítettem az architektúrába az OpenWeatherMap volt. Ez azért lett beépítve a rendszerbe, mert az egyik általam készített automata funkcióhoz szükséges folyamatosan lekérdezni az időjárást, illetve a hőmérsékletet.

4.5 Kliens architektúrája

A kliensoldali szoftver architektúrájának a VIPER-t választottam. Azért erre a mintára esett a választásom, mert MVVM-et (Model-View-ViewModel) már használtam korábban és szerettem volna valamilyen új mobilos környezetben használatos mintával is megismerkedni.



4.6. ábra Az architektúra komponenseinek kapcsolata[8]

A VIPER egy architekturális minta, mint például az MVC vagy az MVVM. Míg az Apple-stílusú MVC minta arra motiválja a fejlesztőket, hogy minden üzleti logikát egy UI-hoz közeli osztályba helyezzenek el, addig a VIPER, az MVVM-hez hasonlóan, egy másik megközelítést alkalmaz. Ez a minta arra készteti a fejlesztőt, hogy az egyes komponensei között minél jobban szétválassza a felelősségeket, ezzel pedig betartsa a Single Responsibility Principle-t. Az architektúra nevében minden betű egy-egy komponenst jelöl. A V a nézetre utal (View), amely leírja az aktuális képernyő felhasználói felületét, adatot jelenít meg, illetve fogadja és továbbítja a felhasználói

[8] Kép forrása: https://www.kodeco.com/8440907-getting-started-with-the-viper-architecture-pattern

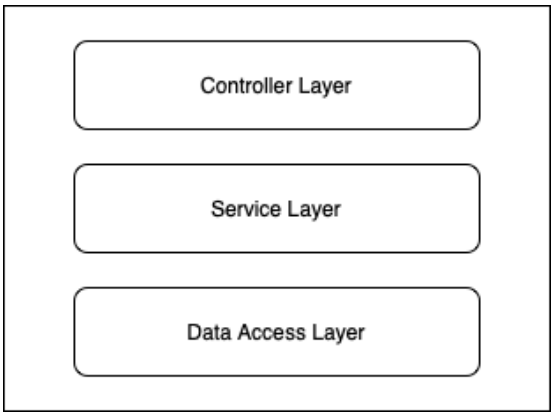
interakciókat a következő komponens felé. Az I mint Interactor felelős az adatok lekérdezéséért az adattárolási rétegtől, valamint tartalmazza az adott modulhoz tartozó üzleti logikát is. A P mint Presenter feladata, hogy az Interactor által összegyűjtött adatot a UI-nak megfelelő módon előkészítse a megjelenítésre. A Presenter feladatai közé tartozik még az is, hogy továbbítsa a kéréseket a UI-tól az Interactor, valamint a navigációs komponens felé. Az E mint entitás (Entity) jelenti a modulhoz tartozó adatelérési réteget. Az R mint Router az adott modul navigációs komponense, amely az egyes képernyők közti navigációt hajtja végre az alkalmazásban. [21]

A 4.6-os ábra jól mutatja, hogy pontosan milyen kapcsolatban is állnak egymással a komponensek, hogyan kerülnek az adatok a UI-ra, valamint a nyilakból azt is leolvashatjuk, hogy melyik komponens kivel tud kommunikálni.

A SwiftUI az egyes funkciói miatt működésben jobban illeszkedik az MVVM-hez, mint a VIPER-hez. Ennek ellenére én mégis utóbbit választottam a klienshez, annyi változtatással kiegészítve, hogy a modulok Presenter komponense hasonlítson egy MVVM-beli ViewModelhez, ami már megfelelően illeszkedik a SwiftUI-ban használatos megoldásokhoz.

4.6 Szerver architektúrája

A szerver oldali komponens architektúrája egy egyszerű rétegelt struktúrát követ. A webalkalmazást három rétegre bontottam szét, ahol az egyes részek mindig csak az alattuk lévő egység szolgáltatásait látják.



4.7. ábra Szerver oldali architektúra

A legfelső réteg a kontroller réteg. Ennek a felelőssége a REST konvenciók betartása. Itt találhatóak az API-ban megjelenő szolgáltatások végpontjai, ez a Web API

interfésze is. Ennek a rétegnek a feladata, hogy publikálja a felhasználók felé az alkalmazás elérhető műveleteit és kiszolgálja a bejövő kéréseket. A kérések kiszolgálásakor a kérés paramétereit a következő réteg számára értelmezhető formára hozza. A középső rétegben találhatóak a Service komponensek, amiben az alkalmazás üzleti logikái találhatók. A Service réteg alatt az adatelérési réteg található. Ebben a legalsó rétegben vannak az alkalmazás entitásainak leírásai, illetve az adatbázis tábláinak létrehozásához szükséges leírások, más néven migrációk.

4.7 Kommunikáció kliens és szerver között

Ebben a szakaszban az előzőekben bemutatott kontroller réteg egy részét fogom jobban kifejteni. Ebben a rétegben találhatóak az API-ban elérhető szolgáltatások végpontjai. Ezek a végpontok együtt egy interfészt alkotnak, amin keresztül a felhasználó elérheti a szerver oldali komponens szolgáltatásait és műveleteit. Az egyes műveletek meghívása HTTP kéréssel működik, amely megfelelő végpont címmel és HTTP igével könnyen összeállítható.

A UserController-ben találhatók a felhasználókezeléssel kapcsolatos műveletek végpontjai. A felhasználókezelés részben megosztott a Firebase hasonló funkcióival, ezért itt nem foglalkoztam a regisztráció, bejelentkezés műveletekkel. A felhasználókkal kapcsolatos CRUD[9] műveletek (kivéve módosítás), illetve ezek végpontjainak felsorolása:

Név	Végpont	Paraméter(ek)	HTTP ige	Rövid leírás
getAllUsers	/users	-	GET	Összes felhasználó lekérdezése.
getUser	/users/{id}	felhasználó ID-ja	GET	Egy adott felhasználó lekérdezése ID alapján.
getUserByExternalId	/users/external/{externalId}	felhasználó külső ID-ja	GET	Egy adott felhasználó lekérdezése külső ID alapján.
createUser	/users	-	POST	Felhasználó létrehozása.

4.1. táblázat A UserController végpontjai

[9] CRUD = CREATE-READ-UPDATE-DELETE

A CouchController szolgáltatásainak és műveleteinek, illetve ezek végpontjainak felsorolása:

Név	Végpont	Paraméter(ek)	HTTP ige	Rövid leírás
getAllCouchesForUser	/couches/user/{userId}	felhasználó ID-ja	GET	Egy felhasználóhoz tartozó összes szálláshely lekérdezése.
getAllCouchesForCityExceptUserId	/couches/city/{city}/{userId}	város neve, felhasználó ID-ja	GET	Egy városban található összes szálláshely lekérdezése, kivéve a megadott felhasználóhoz tartozók.
getCouch	/couches/{id}	szálláshely ID-ja	GET	Egy adott szálláshely lekérdezése.
createCouch	/couches	-	POST	Szálláshely létrehozása.

4.2. táblázat A CouchController végpontjai

A MessageController szolgáltatásainak és műveleteinek, illetve ezek végpontjainak felsorolása:

Név	Végpont	Paraméter(ek)	HTTP ige	Rövid leírás
getMessages	/messages/{conversationId }	beszélgetés ID-ja	GET	Egy beszélgetéshez tartozó összes üzenet lekérdezése.
createRating	/messages	-	POST	Üzenet létrehozása.

4.3. táblázat A MessageController végpontjai

A ConversationController szolgáltatásainak és műveleteinek, illetve ezek végpontjainak felsorolása:

Név	Végpont	Paraméter(ek)	HTTP ige	Rövid leírás
getConversations	/conversations/{userId }	felhasználó ID-ja	GET	Egy felhasználóhoz tartozó összes beszélgetés lekérdezése.
createConversation	/conversations	-	POST	Beszélgetés létrehozása.

4.4. táblázat A ConversationController végpontjai

A ReservationController szolgáltatásainak és műveleteinek, illetve ezek végpontjainak felsorolása:

Név	Végpont	Paraméter(ek)	HTTP ige	Rövid leírás
getReservationsForUser	/reservations/user/{userId}	felhasználó ID-ja	GET	Egy felhasználóhoz tartozó összes foglalás lekérdezése.
getReservationsForCouch	/reservations/{couchId}	szálláshely ID-ja	GET	Egy szálláshelyhez tartozó összes foglalás lekérdezése.
createReservation	/reservations	-	POST	Foglalás létrehozása.

4.5. táblázat A ReservationController végpontjai

A RatingController szolgáltatásainak és műveleteinek, illetve ezek végpontjainak felsorolása:

Név	Végpont	Paraméter(ek)	HTTP ige	Rövid leírás
getRatings	/ratings/{couchId}	szálláshely ID-ja	GET	Egy szálláshelyhez tartozó összes értékelés lekérdezése.
createRating	/ratings	-	POST	Értékelés létrehozása.

4.6. táblázat A RatingController végpontjai

A HomeController szolgáltatásainak és műveleteinek, illetve ezek végpontjainak felsorolása:

Név	Végpont	Paraméter(ek)	HTTP ige	Rövid leírás
getAllItems	/home/items/{couchId}	szállás ID-ja	GET	Egy szálláshelyhez tartozó összes okoseszköz lekérdezése.
getItem	/home/item/{configurationId}	eszköz ID-ja	GET	Egy okoseszköz lekérdezése.
switchItem	/home/switch/{itemId}	kapcsoló ID-ja	POST	Egy kapcsoló fel- vagy lekapcsolása.
getConfigurationTypes	/home/types	-	GET	Támogatott okoseszköz típusok lekérdezése.
getConfigurationTypeProperties	/home/types/properties/{configurationType}	okoseszköz típusa	GET	Adott típushoz tartozó beállítások lekérdezése.
getConfigurationProperties	/home/properties/{configurationId}	eszköz ID-ja	GET	Egy okoseszközhöz tartozó beállítások lekérdezése.
setPropertyState	/home/properties	-	POST	
createHomeConfiguration	/home/items	-	POST	

4.7. táblázat A HomeController végpontjai

A fenti táblázatokban csak a legfontosabb műveleteket tűntettem fel, ezeken kívül készültek még más végpontok is, azok azonban nem lettek felhasználva a kliensalkalmazásban, vagy nagyon kézenfekvő a funkciójuk. Azoknál a POST típusú műveleteknél, ahol nem tűntettem fel paramétert, ott a HTTP kérés body részében lehet további adatokat küldeni JSON formátumban a szerver felé.

4.8 Swagger használata

Ahhoz, hogy a korábban ismertetett végpontokat majd egyszerűen fel tudjam használni a kliensalkalmazásban, szükségem volt egy API leíróra. Ezek a leírók nagyon hasznosak, mivel egyaránt szolgálnak szerver oldali dokumentációként, valamint különböző eszközökkel kliens vagy szerver oldali kódot is tudunk generálni belőlük.

A dokumentáció és API leíró elkészítéséhez a Swagger nevű eszközt használtam. A Swagger számos problémára nyújt megoldást, tartalmaz például kódgenerátorokat, valamint egy webes kódszerkesztőt, amivel böngészőből tudunk leírókat kézzel összeállítani. A szerkesztő használata esetén a leírt végpontok rögtön meg is jelennek vizuális formában és helyben ki is lehet próbálni őket. Leírót készíteni nem csak kézi megoldással lehet, például automatikusan is lehet ilyet generálni, ha kész projektben felhasználunk külső könyvtárakat. Ilyen könyvtárat sajnos nem találtam, ami a Vapor webalkalmazásokkal együtt tudna működni, szóval a webes szerkesztőt használtam a leíró elkészítéséhez.

A következőkben bemutatok egy egyszerű példát, hogy hogyan állítottam össze a Swagger webes szerkesztőjében egy végpontot és egy hozzá tartozó modellosztály leírást.

A /couches/user/{userId} végpont API leírása Swagger-ben:

```
/couches/user/{userId}:
     get:
          tags:
          -  „couch”
          summary: „Find all couch for user”
          operationId: „findAllCouchForUser”
          produces:
          -  „application/json”
          parameters:
          -  name: „userId”
             in: „path”
             description: „ID of user whose couches to return”
             required: true
             type: „string”
          responses:
               „200”:
                    description: „successful operation”
                    schema:
                         type: „array”
                         items:
                              $ref: „#/definitions/APICouch”
```

Mindegyik végpontot az elérési útjával kell kezdeni, utána jöhet a HTTP ige és az egyéb tulajdonságai. A tags alatt megadtam, hogy melyik végpont csoportba sorolja ezt a műveletet, valamint leírással is bővítettem, hogy a dokumentáció is szókimondóbb

legyen. Az operationId egy kötelező mező, kihagyása esetén automatikusan hibát jelez a szerkesztői felület. A consumes és produces tulajdonságokkal, azt lehet megadni, hogy milyen formátumú objektumot vár a kérés a body-ban vagy ad vissza eredményként. Én minden esetben a JSON formátumot adtam meg, azonban a POST kéréseken kívül a consumes részt nem töltöttem ki. A paraméteres műveletekhez felvettem az adott paramétert a megfelelő tulajdonsághoz. Itt megadtam, hogy hol keresse, milyen formátumú az adott paraméter, illetve, hogy milyen névvel találja meg az elérési útvonalban. Dokumentációs szempontokat figyelembe véve itt is megadtam leírást, hogy mire kell használni a paramétert.

A responses részben pedig azt lehet megadni, hogy milyen válaszokat tud adni a kérés, amikor befejeződött. Az egyes válaszokat HTTP státusz kódok szerint csoportosítottam, sikeres művelet esetén pedig megadtam, hogy milyen formátumban adja majd vissza a kérés az eredményt. Ennél a résznél látható, hogy a $ref kulcsszóval lehet hivatkozni olyan DTO (Data Transfer Object) részeket a leíróban, amik már meg lettek írva, jelen esetben például az APICouch adatmodell sémáját.

A végponthoz tartozó modellosztály leírása:

```
definitions:
     APICouch:
          type: „object”
          properties:
               id:
                    type: „string”
               latitude:
                    type: „number”
                    format: „double”
               ....
               user:
                    $ref: „#/definitions/APIUser”
```

A modellek sémája többször is előfordulhat egy leíróban, ezért a Swagger-ben lehetőségünk van ezeket külön szervezni a végpontoktól. Én a leírófájlom végére tettem egy ilyen szakaszt, ami csak ezeket a sémákat tartalmazza. Programozási nyelvekhez hasonlóan lehet megadni egy ilyen modellt, először jön a modell neve, utána a típusa, majd a tulajdonságai, szintén név-típus párossal. Típusokat néhány esetben lehet formátum jelöléssel bővíteni vagy helyettesíteni korábban definiált részekre hivatkozva.

4.9 Kommunikáció az openHAB-al

Egy korábbi szakaszban már bemutattam az openHAB-ot, mint platformot és meg is említettem, hogy a fejlesztők rendelkezésére bocsátottak egy gazdag REST API interfészt, amely segítségével használhatjuk az openHAB funkcióit saját alkalmazásainkban. A saját szerver oldali komponensem szintén ezt az interfészt használja a platformmal való együttműködéshez. Ebben a szakaszban bemutatom, hogy melyik végpontokat használtam fel az interfészről, illetve azt is bemutatom, hogy miért éppen ezekre volt szükségem.

A fő célom az volt, hogy az iOS alkalmazásban megjelenjenek okoseszközök, amiknek meg lehet tekinteni a tulajdonságait, illetve lehet állítgatni különböző állapotait. A tervezésnél egy olyan feltételezéssel éltem az openHAB-al kapcsolatban, hogy minden okoseszköz hozzá lett adva a platformhoz és fel is lett konfigurálva a fizikai rétegben, a virtuálisban viszont nem. Ennek megfelelően fel kellett készülnöm arra, hogy csak akkor fogom tudni állítgatni az eszközöket és lekérdezni az állapotukat, ha mindent beállítok a virtuális rétegben is. Emiatt felvettem még egy workflow-t, amiben a felhasználó a kliensen keresztül új okoseszközt tud felvenni a saját szálláshelyéhez, ezzel pedig az openHAB-on kialakítva a virtuális réteget az eszközökhöz.

Az Item-ek létrehozásához használt végpontok:

Végpont	Paraméter(ek)	HTTP ige	Rövid leírás
/things	-	GET	Összes Thing lekérdezése.
/items/{itemName}	Item neve	PUT	Paraméterként megadott névvel Item hozzáadása.
/links/{itemName}/{channelUID}	Item neve, Channel UID azonosítója	PUT	Link létrehozása Item és Channel között.

4.8. táblázat Item létrehozáshoz használt végpontok

Az Item-eket saját magunknak kell létrehozni és összecsatlakoztatni a megfelelő csatornákhoz. A létrehozáshoz és csatlakozáshoz egyaránt szükségem volt az adott Thing csatornáira. Az Item-eket annak megfelelően kell létrehozni, hogy melyik csatornához szeretnénk kapcsolni azt. Mindkettő rendelkezik egy típus tulajdonsággal, aminek meg

kell egyezniük, ugyanis egy Switch típusú Item nem kompatibilis egy Number típusú csatornával.

A REST API-n keresztül nem lehet lekérdezni önálló végponttal a csatornákat, mivel a Thing-ek adatszerkezetileg tartalmazzák az összes hozzájuk tartozót. Emiatt a csatornákhoz való hozzáférést úgy oldottam meg, hogy lekérdeztem az összes Thing-et az egyik fent látható végponttal. Ezután már az Item-eket létre tudtam hozni és csatlakoztatni is tudtam őket a megfelelő helyre, mert meg volt hozzájuk minden információ. Item-et a fenti végpontok közül a /items/{itemName}-el tudtam létrehozni, amihez paraméterként egy nevet, a kérés body részében pedig a következő JSON objektumot kellett megadnom:

```
{
     „type”: „string”,
     „name”: „string”,
     „label”: „string
}
```

A type mezőben a csatorna megfelelő típusát kellett megadnom, a name mezőben található névnek meg kellett egyeznie a paraméterben megadott névvel, valamint opcionálisan megadhattam még egy címkét is, amit a UI felületen fog majd megjeleníteni az openHAB. Az elnevezések kicsit megtévesztők lehetnek, mivel itt a név az egy azonosítóként szolgál és egyedinek kell lennie a többi Item között. Emiatt hasznos a label mező használata, itt lehet ténylegesen nevet adni az Item-nek, amivel leírhatjuk, hogy funkcionálisan mi a szerepe. A létrehozásnál természetesen más mezőket is ki lehet még tölteni, azonban én a minimálisra törekedtem és kihagytam a feleslegesnek vélt mezőket.

A csatornáknak az Item-ekhez hasonlóan számos tulajdonságuk lehet, a két legfontosabb viszont a *típus* és a *UID* (Unique Identifier) mezők. A *UID* mező azért fontos, mert ez azonosítja az Item-ek számára a csatornákat, ezt kell megadni a csatlakoztatásukkor a /links/{itemName}/{channelUID} végpontnak paraméterként. A kérés sikerességéhez a body-ban szintén el kell küldeni a megfelelő paramétereket a következő formában:

```
{
     „itemName”: „string”,
     „channelUID”: „string”
}
```

A csatornák esetében is megfigyelhető egy félrevezető elnevezés, mivel van egy sima *ID* tulajdonságuk is, azonban ez nem azonosításra szolgál, inkább egy funkcionalitás megnevezéseként.

Az Item-ek hozzáféréséhez használt végpontok:

Végpont	Paraméter(ek)	HTTP ige	Rövid leírás
/items/{itemName}	Item neve	GET	Item lekérdezése.
/items/{itemName}	Item neve	POST	Paraméterként megadott nevű Item-nek parancs kiadása.

4.9. táblázat Item-ek hozzáféréséhez használt végpontok

Miután az openHAB-on a virtuális réteg is teljesen összeállt, nagyon egyszerű volt az eszközök tulajdonságait lekérdezni, illetve beállítani. A lekérdezésnél a /items/{itemName} végpontot kellett meghívnom, ami egy a létrehozásnál is látott JSON objektumot adott vissza. Itt ez a válaszobjektum kiegészült még egy state mezővel, ami szövegesen vagy számokkal leírva tartalmazta az aktuális állapotot.

Az értékek beállításánál a /items/{itemName} végpontot kellett használnom. Ez a végpont érdekesebb a többinél, mivel itt is kellett a body-ban üzenetet küldenem a platformnak, azonban nem egy JSON objektum formájában. Egy egyszerű szövegként kell ennek megadnom azt a parancsot, amit az aktuális Item-nek szerettem volna kiadni. Ez egy kapcsolónál lehetett ON vagy OFF, Number típusú csatornára pedig magát a számot kellett elküldeni. Mint a többi végpontnál, itt is nagyon kellett ügyelni a megfelelő formátumra, ami a szöveges parancsoknál a nagybetűs formát jelentette.

5 Megvalósítás

Ebben a fejezetben a rendszer implementációjának fontosabb és érdekesebb részleteit fogom bemutatni. Az alfejezetekben kitérek mind a kliensalkalmazásban, mind pedig a szerver oldalon történt kihívásokra és azok megoldásaira, valamint az okos otthon funkciók megvalósításának fontosabb pontjaira és a szerver oldal tesztelésére.

5.1 SwiftUI és UIKit együttes használata

A SwiftUI-t már korábban bemutattam és ott említettem is, hogy még aktívan fejlődő technológiaként nem ér fel a UIKit tudásához. Természetesen az Apple kínál megoldást a fejlesztőknek, ha olyan funkciókat szeretnének felhasználni a SwiftUI alapú alkalmazásukban, ami még nem elérhető az új technológiájukban. Én is belefutottam ebbe a problémába, ezért meg kellett ismernem, hogy hogyan tudom a két UI keretrendszert együttesen használni.

Az alkalmazásomban a szálláshely hozzáadás funkciónál szerettem volna egy olyan kiegészítést, hogy a felhasználó tudjon képeket hozzáadni a feltöltendő szálláshelyéhez. Ehhez készítettem egy olyan nézetet, amelyben vízszintesen egymás mellett képek jelennek meg, sok kép esetén pedig scroll-ozni is lehet közöttük. Ebben a képsorozatban a legbaloldalibb elemként egy plusz gombot is elhelyeztem, aminek a megnyomására egy képválasztó felületi elem jelenik meg, ahol a felhasználó a hozzáadni kívánt képeket tudja kiválasztani egyesével. Amikor ezt a funkciót implementáltam, a legfrissebb iOS a 14-es verziójú volt, ami még nem tartalmazott SwiftUI-os Image Picker-t. A UIKit természetesen tartalmazta ezt a funkciót, ezért fel kellett készítenem a UIKit-es megoldást, hogy az SwiftUI nézetként meg tudjon jelenni.

Ahhoz, hogy SwiftUI-ban használni tudjam a UIKit-es vezérlőket, létre kellett hoznom egy SwiftUI nézetet, aminek az osztálya nem a megszokott View osztályból származik le, hanem egy UIViewControllerRepresentable vagy UIViewRepresentable osztályból. Értelemszerűen az ViewController újrafelhasználásakor az előzőt, UIKit View esetén pedig az utóbbit kell használni. Ennek az osztálynak három metódust kell megvalósítania a működéshez. Az első a makeUIView(Controller), amiben létre lehet hozni és tetszés szerint lehet inicializálni a megfelelő nézetet vagy vezérlőt. Az updateUIView(Controller) metódusban lehet frissíteni a UI-t. Az egész megvalósításához

szükségem volt még egy belső Coordinator nevű osztályra is. Ez az osztály fogja megvalósítani a nézetekhez tartozó Delegate-eket. Ez azt osztályt minden esetben létre kellett hoznom, hogy a UIView(Controller)Representable makeCoordinator metódusát meg tudjam valósítani.

```
struct ImagePickerKit: UIViewControllerRepresentable {
     func makeUIViewController(context: Context) -> UIImagePickerController
     {
          let imagePicker = UIImagePickerController()
          imagePicker.delegate = context.coordinator
          ....

          return imagePicker
     }

     func updateUIViewController(_: UIViewControllerType, context: Context)
     { }

     func makeCoordinator() -> Coordinator
     {
          Coordinator(self)
     }

     class Coordinator: ....
     ....
}
```

A fenti kódrészletben látható, hogy nekem a ViewController típusú ősből kellett leszármaznom, mivel a UIImagePickerController is egy ViewController osztály. A három metódus megvalósítása is jól látszik, a makeUI... kezdetűben a legfontosabb, hogy itt kézzel hozom létre a megfelelő nézetet, amit majd visszatérési értékként ad tovább a függvény. Visszatérés előtt viszont fontos beállítani, hogy hol találhatók a delegate implementációk, amik a Coordinator osztályban helyezkednek el.

```
class     Coordinator:     NSObject,     UIImagePickerControllerDelegate,
UINavigationControllerDelegate {
     var parent: ImagePickerKit

     init(_ parent: ImagePickerKit) { self.parent = parent }

     func imagePickerController(_: ..., didFinishPickingMediaWithInfo info:
...) {
          ...
     }
}
```

Fent a Coordinator osztály megvalósítása található. Két delegate interfészt valósít meg ez az osztály, az egyik a kép választó funkciót valósítja meg, a másik pedig a navigációt. Az imagePickerController függvényben az info paramétertől kérem el, hogy

választott-e a felhasználó képet és ha igen, akkor beállítom azt a képet a kiválasztott képnek. Kép kiválasztása után pedig eltűntetem az Image Pickert.

5.2 Navigáció SwiftUI-ban

SwiftUI-ban a navigáció eltérő módon működik, mint más UI technológiákban vagy UIKit-ben. A megszokott módszer, hogy valamilyen függvényhívás hatására tudunk újabb nézetekre navigálni az éppen aktuálisról, viszont deklaratív keretrendszerben ezt a függvényhívást felváltja egy speciális UI elem. Ez az elem a NavigationLink, ami valójában egy speciális gombként működik, aminek lenyomásakor megtörténik a navigáció. Paraméterként kell neki megadni, hogy mi a célképernyő, amire meg szeretnénk érkezni a gomb lenyomása után, valamint egy UI elrendezést, hogy miként is rajzolódjon ki a speciális gomb.

VIPER architektúrában a navigációt úgy valósítottam meg, hogy az megfeleljen az architektúra felépítésének. A Router komponens az a szereplő, amiben összeállítom a következő nézethez szükséges osztályokat. Ez az osztály jó leírása annak, hogy az egyes nézetekről melyik további nézetekre lehet továbbnavigálni az alkalmazásban. Egy ilyen nézetösszeállítás a következőképpen néz ki:

```
func makeNewCouchView() -> View {
     let interactor = NewCouchInteractor(...)
     let presenter = NewCouchPresenter(interactor)
     return NewCouchView(presenter)
}
```

Mivel a Router-hez csak a Presenter-ből férek hozzá, ezért itt helyeztem el egy függvényt, ami létrehozza a NavigationLink-et, majd továbbadja a Link-nek a Router függvényében létrehozott nézetet.

```
func linkBuilderForNewCouch<Content: View>(@ViewBuilder content: () ->
Content) -> some View
{
     NavigationLink(destination: router.makeNewCouchView()) {
          content()
     }
}
```

A fenti függvénynek paraméterül meg lehet adni egy ViewBuilder függvényt, ami egy View-val tér vissza. A függvényben emiatt tetszőleges UI elrendezést adhattam meg, hogy hogyan nézzen ki a navigációt elindító gomb.

A fenti kódrészlet azonban nem minden navigációs esetre ad megfelelő megoldást. Az alkalmazásban több helyen is szerettem volna, olyan logikát beépíteni, ahol a navigációt csak egy azt megelőző művelet után indítom el, annak függvényében, hogy milyen eredményt kaptam a művelettől. Ennek az esetnek talán a legjobb példája a bejelentkezés megvalósítása. Itt szerettem volna először bejelentkeztetni a felhasználót, aztán amikor ez sikerült, továbbnavigálni őt egy másik nézetre. A SwiftUI ezekre az esetekre is kínál megoldást. A NavigationLink-nek van egy isActive tulajdonsága, amit, ha igaz értékre állítunk, akkor automatikusan megtörténik a navigáció. Az ilyen NavigationLink-eket én eltűntettem a felhasználó elől úgy, hogy a speciális gomb kinézetének egy EmptyView-t adtam meg. Így természetesen létre kellett hoznom egy másik gombot, hogy tudjon mit nyomni a felhasználó. Ez az új gomb már egy hagyományosan működő gomb volt, aminek a logikájához meg tudtam adni a műveletet, amit a navigáció előtt el akartam végezni, majd a navigációt el tudtam indítani azzal, hogy az isActive-et igaz értékre állítottam be.

5.3 Okos otthon funkciók implementálása

A rendszerben található elkészült okos otthon funkciókat három csoportba tudnám rendezni. Az első csoportba a fűtési rendszerek és légkondicionálók vezérlését, a másodikba a zárak kezelését, a harmadikba pedig az előző két csoportban található eszközökkel kapcsolatos automatizmusokat tudnám sorolni. Külön csoportba nem sorolnám, de ezeknek a támogatott eszköztípusoknak a kliensalkalmazásból való felkonfigurálását is megvalósítottam.

5.3.1 Új eszköz felvétele

A tervezés részben már összegyűjtöttem, hogy milyen végpontokra volt szükségem egy új eszköz felvételéhez és az openHAB-on található virtuális réteg kiépítéséhez. Az egyes HTTP kéréseket már csak össze kellett raknom valahogy a szerver oldali komponensben. Az openHAB-al való kommunikáció megírásához nem használtam Swagger-t. A Vapor keretrendszerben van egy Client nevű API, amivel külső erőforrások eléréséhez tudtam könnyen HTTP kéréseket indítani. Csináltam is egy interfész – Service osztály párost, amelyek tartalmazták a szükséges metódusokat. A Service osztálynak egy Client típusú objektumot kellett megadnom paraméterként, ezen keresztül lehetett indítani a HTTP kéréseket.

```
func getAllThings() async throws -> [Thing] {
     let header = HTTPHeaders([(„Authorization”, „...”)])
     let uri = URI(string: „...”)
     let response = try await httpClient.get(uri, headers: header)
     return try decodeClientResponse(response: response)
}
```

A Client osztálynak van több metódusa, amik HTTP igékről lettek elnevezve, ezért nem okozott különösebb gondot ennek az objektumnak a használata. Mindegyik ilyen metódusnak kötelező volt megadnom egy elérési utat, valamint opcionálisan megadhattam még neki HTTP fejlécet is. A fejlécet végül az openHAB-os eléréshez kötelező volt megadnom, mert ebbe tettem bele az authorizációhoz szükséges tokent. Ha paramétert is szükséges volt megadnom a kéréshez, akkor vagy az URI-ban küldtem el az adatot, vagy pedig a kérés body részében. A body rész kitöltését a Client megfelelő metódusának meghívásakor egy closure-ben tudtam megtenni:

```
let response = try await httpClient.put(uri, headers: header) { req in
     try req.content.encode(body)
}
```

A válaszok egyszerű kezeléséhez és kicsomagolásához készítettem még egy generikus segédfüggvényt, ami ellenőrzi a válaszul kapott HTTP státuszkódot, majd sikeres válasz esetén kicsomagolja a függvény sablon paraméterének megfelelő típusba a választ.

```
func decodeClientResponse<T: Decodable>(response: ClientResponse) -> [T] {
     guard response.status == .ok else { throw Abort(.badRequest) }
     let data = String(buffer: buffer).data(using: .utf8)
     let decoder = JSONDecoder()

     do {
          return try decoder.decode([T].self, from: data)
     } catch let error {
          ...
     }
}
```

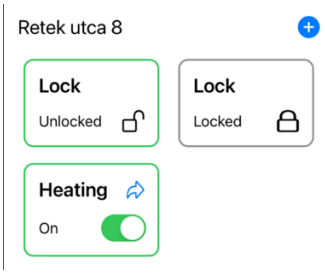
A fenti függvények elkészültével már tudtam kommunikálni az openHAB-al a szerver oldali komponensből. Ezután az alkalmazás adatmodelljébe felvettem egy új táblát HomeConfiguration néven. Erre azért volt szükségem, mert a felvett eszközökről szerettem volna olyan adatokat eltárolni, amelyek az openHAB-on nincsenek meg. Ilyen adat volt például az, hogy az eszköz melyik szálláshelyhez tartozik. Mivel az openHAB virtuális rétegének legtetején az Item-ek állnak, amelyek csak egy konkrét eszköznek egy tulajdonságát reprezentálják, az új tábla rekordjaiban el tudtam tárolni azt is, hogy az egyes eszközökhöz melyik Item-ek tartoznak pontosan. Az új tábla mellett még felvettem

egy másik új típust is, amiben a támogatott típusokat soroltam fel. Ebbe került bele először a fűtés (heating) és hűtés (cooling), majd később pedig az automatizációs funkció (weatherWatcher), valamint a zárak (lock). Az openHAB-al való kommunikációért felelős Service-t és a HomeConfiguration tábla műveleteit tartalmazó Service osztályt egy új HomeService nevű egységben hoztam össze. A fő üzleti logikát ez a komponens tartalmazza és ez lett beszúrva a Controller rétegbe is.

A kliensoldalon a hozzáadáshoz készítettem egy új VIPER modult, benne egy nézettel, Presenterrel és Interactorral. A nézet felépítése azért volt érdekes számomra, mert a kiválasztott eszköztípushoz minden tulajdonságot meg akartam jeleníteni, amit az openHAB biztosított. Emiatt nem ismertem előre, hogy hány kapcsolót vagy hány szám beviteli mezőt kell megjelenítenem. Végül viszont nem volt annyira bonyolult felépítenem egy dinamikus nézetet. Az eszköz csatornáit könnyen le tudtam kérdezni a szervertől és utána rendelkezésemre állt, hogy hány darab Switch típusú és Number:Temperature típusú csatornája van. Ezeket a csatornákat külön listákba csoportosítottam típus szerint, majd utána a nézetben listánként haladva jelenítettem meg a megfelelő UI elemeket. A nézetre ezen kívül került még egy szövegmező, amiben tetszőlegesen el lehet nevezni az új eszközt. Ez a név félig azonosítóként is szolgál, mivel az openHAB-on létrejövő Item-ek részben az itt megadott névvel jönnek létre. A nézeten található még két darab Picker is, ezekből lehet kiválasztani, hogy melyik szálláshelyhez szeretnénk hozzáadni az eszközt, illetve milyen eszköztípust szeretnénk.

5.3.2 Hűtés és fűtés kezelése

Szerintem az alkalmazás funkciói közé a fűtés és légkondicionálónak a kezelése illett bele a legjobban. Ez a két eszköz tartalmaz egy picit bővebb logikát a többihez képest, ugyanis ezek nem csak egy-egy kapcsolóból állnak, hanem számszerű értékeket is be lehet állítani rajtuk. A tulajdonságok beállításához a tervezésben ismertetett végpontokat tudtam használni.



5.1. ábra Kapcsolók az összefoglaló nézeten

A kliensben a fő kapcsolók állításához készítettem egy összefoglaló nézetet. Ezen szálláshelyenként csoportosítva jelenítettem meg az elérhető eszközöket. Erről a nézetről lehet továbbnavigálni az új eszköz hozzáadására, illetve egyes eszközök bővebb tulajdonságait tartalmazó nézetre is. A kapcsolók megjelenítéséhez készítettem egy CardView-t, ami az eszköz típusát, magát a kapcsolót és annak állapotát tartalmazta szöveges formában. A kapcsoló aktuális állapotát kiemeltem egy körvonallal, ami állapotnak megfelelően változtatta a színét, hogy az be van-e kapcsolva vagy sem. A CardView-t külön osztályba szerveztem ki az újrahasznosítás miatt és ennek az osztálynak kis closure-ket adtam meg paraméterként, amikkel megadtam, hogy mi történjen a kapcsoló megnyomásakor vagy navigáció kezdeményezésekor.

Ehhez az összefoglaló nézethez készítettem egy saját végpontot is a szerver oldalon. Ez azért volt jó, mert így a kulcsfontosságú tulajdonságokat nagyon egyszerűen és költséghatékonyan tudtam beállítani. A végpontnak csak elég volt egy ID-t megadnom, amivel a szerver oldal be tudta azonosítani a megfelelő eszközt, majd az át is kapcsolta magát az új állapotba.

5.3.3 Automatizált funkció bemutatása

A rendszerben megvalósításra került egy automatikusan működő okosfunkció is. Ennek a lényege, hogy ha egy szálláshelyhez van felvéve fűtés vagy klíma, akkor energiatakarékossági szempontból ezeket automatikusan lehet ki és bekapcsolni. A rendszer figyeli az érintett városok külső hőmérsékletét és ha az egy megadott érték alá vagy fölé ment, akkor ki- vagy bekapcsolja az érintett eszközt.

A funkciót a Vapor Queues használatával valósítottam meg a szerver oldalon. A Queues egy teljesen Swift nyelven írt üzenetsor-kezelő rendszer, amellyel feladatokat lehet ütemezni. Ez akkor nagyon hasznos funkció, ha ütemezetten szeretnénk valamilyen feladatot elvégezni bejövő kérésektől függetlenül, vagy ha valamilyen komplex műveletet akarunk futtatni az adatbázison, ami sokáig tart. [22]

```
struct EmailJob: AsyncJob {
     func dequeue(_ context: QueueContext, _ payload: Email) async throws {
          // some heavy task
     }
}
```

A Vapor biztosít a fejlesztők számára egy Job protokollt, aminek megvalósításával tudtam saját Job-okat készíteni. Lehetőség van aszinkron feladatok implementálására is, ezekhez az AsyncJob protokollt valósítottam meg. [22]

```
struct CleanupJob: AsyncScheduledJob {
     func run(context: QueueContext) async throws {
          // some scheduled heavy task
     }
}
```

A Job és AsyncJob protokollok azokhoz a feladattípusokhoz megfelelőek, amiket kézzel szeretne a fejlesztő elindítani. Bizonyos időre ütemezett feladatok létrehozásához a ScheduledJob és AsyncScheduledJob protokollokat kell megvalósítani. [22]

```
struct Email: Codable {
     let to: String
     let message: String
}

struct EmailJob: AsyncJob {
     typealias Payload = Email

     func dequeue(_ context: QueueContext, _ payload: Email) async throws {
          // some heavy task
     }
}
```

A Job-oknak van még lehetőségünk paramétereket is átadni. Például egy emailküldő feladatnak meg kell adnunk, hogy kinek és milyen üzenetet szeretnénk küldeni. Ezt a paramétert Payload-nak nevezzük. A paramétereket egy új típusban foglalhatjuk össze, amit aztán Payload objektumként megadhatunk egy Job-nak. [22]

Az általam kitalált funkcióhoz két Job-ot készítettem. Az első feladat egy ütemezett feladat volt, amit arra használtam, hogy az feladatban érintett városok külső hőmérsékletét lekérdeztem egy webszolgáltatástól. A hőmérséklet lekérdezéséhez készítettem még egy különálló Service osztályt. Ez a Service az OpenWeatherMap szolgáltatását hívta meg a korábban bemutatott Vapor Client API-val. A feladat egy Dictionary-ben gyűjtötte össze az érintett városokat és a hozzájuk tartozó hőmérséklet értékeket.

```
class WeatherJob: AsyncScheduledJob {
     ...
     func run(context: ContextQueue) throws async {
          ...
          try     await     context.queue.dispatch(WeatherWatcherJob.self,
          .init(dictionary))
     }
}
```

Az ütemezett Job végén automatikusan elindítottam egy másik Job-ot, aminek a feladata az volt, hogy a hőmérséklet értékek alapján elvégezze a szükséges ki- és bekapcsolásokat az eszközökön. A város-hőmérséklet összerendelésekhez készítettem egy saját Payload típust, amibe csak egy Dictionary-t raktam. A kézi feladatindításnál csak a feladat típusát, illetve a Payload objektumot kellett megadnom.

```
app.queues.add(WeatherWatcherJob(...))
app.queues.schedule(WeatherJob(...)).hourly().at(5)
```

A futtatni kívánt Job-okat minden esetben be kell regisztrálni az alkalmazás indításakor függetlenül attól, hogy kézzel vagy ütemezve indítjuk majd őket. A Job-okat a configure függvényben regisztrálta be az alkalmazásba, itt meg tudtam adni nekik könnyedén a Service osztály függőségeiket is. A sima feladatokat az add nevű függvénnyel tudtam hozzáadni, míg az ütemezetteket a schedule függvénnyel. A schedule függvény meghívása után meg tudtam adni további függvényekkel, hogy mikor és milyen időközönként szeretném futtatni a feladatot. A kódrészletben látszik, hogy úgy hoztam létre a feladatot, hogy az minden egész óra öt perckor fusson. Ezen kívül akár évre, hónapra, napra pontosan is meg lehet adni a futás időpontját.

5.3.4 Zárak kezelése

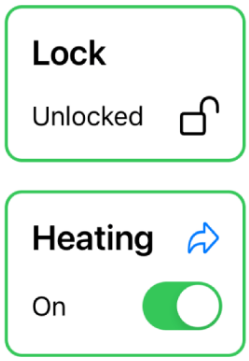
A legutoljára elkészített okosfunkció a rendszerben a zárak kezelése volt. A fejezetből ki fog derülni, hogy ez a funkció felépítésben egyszerűbb, mint a fűtés vagy a légkondicionáló kezelése. Ebben a szakaszban inkább azt szeretném bemutatni, hogy milyen lépésekre volt szükségem ahhoz, hogy a szerver oldalon az okoseszközök kezeléséhez megvalósított kódrészeket bővítsem egy új eszközt támogatásával.

A szerver oldalon először is fel kellett vennem egy új értéket a ConfigurationType típusba „lock” néven. Mivel ez a típus is le van képezve az adatbázisban, ezért fontos volt, hogy abból is irányból is frissítsem a modellt. Ezt úgy tudtam megtenni, hogy a HomeConfigurationMigration osztályban a megfelelő helyre szintén felvettem az új értéket. Ez a két változtatás kellett ahhoz, hogy tudjak Lock típusú eszközöket beszúrni az adatbázisba.

Nekem az openHAB-on csak egy eszközöm volt felkonfigurálva, aminek a Channel-jei a korábbi funkciókhoz voltak csak megfelelőek. Nem szerettem volna, hogy ha LED vagy hőmérséklet tulajdonságok jelentek volna meg egy zárnál, ezért más megoldást kerestem a zár tulajdonságaihoz.

```
func getConfigurationTypeProperties(configType: ConfigurationType) ->
[Channel] {
     if configType == .lock {
          var channel = Channel()
          channel.id = „Lock”
          channel.itemType = „Switch”
          ...
     } else {
          // kérés az openHAB felé
     }
}
```

Végül amellett döntöttem, hogy kézzel veszek fel ehhez a típushoz egy tulajdonságot, mintha az igazi Channel lenne. A HomeService azon függvényébe, ami a tulajdonságokat kérdezi le, felvettem egy if-else blokkot és a paraméterként megadott ConfigurationType szerint adtam vissza a zár Channel-jét vagy továbbítottam a kérést az openHAB felé. Ezen változtatás miatt a zár csupán egy sima kapcsolóként jelenik meg a rendszerben.



5.2. ábra A kétfajta kapcsoló UI felülete

A kliensoldalon az eszköz hozzáadásnál a támogatott eszköztípusokat, illetve a hozzájuk tartozó tulajdonságokat egyenesen a szervertől kérdezem le, emiatt ebben a komponensben nem kellett semmilyen változtatást sem végrehajtanom. A megjelenítésben viszont meg szerettem volna különböztetni a zárakat a többi kapcsolótól. Emiatt csináltam egy új CardView-t kizárólag a zárak számára. Ez abban különbözött az eddigi CardView-tól, hogy nem tartalmazott olyan gombot, ami átnavigálná a felhasználót az eszköz részletes nézetére. Kapcsoló UI vezérlőt sem tartalmazott, ehelyett az egész nézetből csináltam egy kapcsolható felületet. Ezek mellett még más szöveget jelenített meg, illetve kapott egy ikont is, ami a nyitott és csukott állapotokat mutatta.

Ezt a funkciót kicsit kevésnek éreztem ebben a formájában. Az is furcsa volt számomra, hogy ha valaki ellopja a felhasználó telefonját, akkor nagyon egyszerűen be

tud jutni az alkalmazás segítségével a szálláshelyre, mert abban benne van a szállás címe, illetve „kulcsa” is van oda. A fenti okok miatt kitaláltam, hogy bővítem a funkciót biometrikus azonosítással. Az elképzelés az volt, hogy amikor a felhasználó rányom a zár kapcsolójára, akkor előtte feljön egy FaceID vagy TouchID a telefonon és azzal azonosítani tudja magát, hogy jogosult a zár nyitására.

Manapság már a legtöbb Apple eszköz biometrikus azonosítást használ az érzékeny adatok védelmére. Ezt a funkciót az Apple a fejlesztők körében is elérhetővé tette a LocalAuthentication API-val. Ez egy Objective-C API, amit egy kicsit kényelmetlenebb használni a Swift-ben íródott alkalmazásokban. Mielőtt használni kezdtem volna a LocalAuthentication API-t, fel kellett vennem egy projekt szintű beállítást, amiben megadtam az indokot, amiért a biometrikus azonosításra szükségem van az alkalmazásban. Érdekes dolog, hogy ez a beállítás csak a FaceID használatához szükséges, valamilyen okból kifolyólag a TouchID használatához kódból meg megadni az indokot az azonosítás indításakor. Tehát az FaceID használatához a következő kulcs-érték párt kellett felvennem az Info.plist fájlba: [23]

```
<key>NSFaceIDUsageDescription</key>
<string>App needs Face ID to unlock your smart locks.</string>
```

Ezután pedig a megfelelő modul Presenter osztályába írtam meg a következő metódust:

```
private func authenticate(onCompletion: @escaping (Bool) -> Void) {
     let context = LAContext()
     var error: NSError?
     if context.canEvaluatePolicy(..., error: &error) {
          let reason = „We need to unlock your data.”
          context.evaluatePolicy(..., localizedReason: reason) { success in
               onCompletion(success)
          }
     }
}
```

A függvény működését tekintve nagyon egyszerű. A canEvaluatePolicy hívással leellenőriztem, hogy az eszköz képes-e biometrikus azonosításra. Ha az ellenőrzés sikeres, akkor végrehajtottam az azonosítást az evaluatePolicy hívással. Látszik, hogy még egy indokot meg kellett adnom a második metódushívásnak, ez volt a TouchID számára szükséges változó, amit korábban említettem. Ennek az API-nak az Objective-C része abból tűnik ki, hogy nem a Swift-ben általában megjelenő Error típust használtam a hibakezeléshez, hanem az NSError-t. Szintén szokatlan megoldás, ahogy az NSError-t a függvénynek C, illetve C++-ban előforduló módon adtam át paraméterként. [23]

A fenti függvény akkor hívtam meg, amikor a felhasználó rányomott egy zár típusú kapcsolóra. Magát a kapcsolót az azonosítás sikerességének megfelelően kapcsoltam át. A többi kapcsoló logikáját ez a változtatás nem érintette, azok azonosítás nélkül működtek tovább.

5.3.5 Funkciók kipróbálása

Sajnos nekem nem voltak pontosan olyan eszközeim, amelyekkel életszerű helyzetet teremtve ki tudtam volna próbálni a korábbi alfejezetekben bemutatott funkciókat. Azonban azért, hogy a kommunikációt megfelelően ki tudjam próbálni beszereztem egy okoskonnektort. Ezt miután az openHAB-on felkonfiguráltam, láttam, hogy egy egyszerű kapcsoló mellett hőmérséklet Channel-t is tartalmaz. Így ez az eszköz megfelelőnek bizonyult arra, hogy leutánozzam egy fűtés vagy légkondicionáló működését. Ezt az eszközt használtam minden funkció kipróbálásánál és csak ez az egy volt az openHAB-hoz is hozzáadva. Emiatt a limitáció miatt lehetett azt látni az 5.3.1-es fejezetben, hogy az elérhető eszközök Channel-jeit az összes eszköz lekérdezésével értem el.

5.4 Unit tesztek

5.4.1 Test-Driven Development alkalmazása

A rendszerhez unit teszteket csak a szerver oldali komponenshez készítettem. Ezeket a teszteket Test-Driven Development (TDD) módszernek megfelelően írtam meg. A módszer egy iteratív fejlesztési folyamat, amely arra helyezi a hangsúlyt, hogy azelőtt kell megírni az egyes funkciókhoz a teszteket, mielőtt még azt a funkciót vagy függvényt leimplementálnánk. A TDD emiatt rövid, egymás után többször ismétlődő fejlesztési ciklusból áll. A ciklus általában a „Red – Green - Refactor” lépésből áll. Első lépésként új tesztet adunk hozzá a teszt készletünkhöz. Ezután az összes tesztet futtatjuk, hogy megbizonyosodjunk arról, hogy az új teszt hibásan futott le. A hibás lefutás jele általában a piros szín szokott lenni, ezért „Red” az első lépés neve. A következő lépésben éppen annyi kódot kell hozzáadnunk a rendszerhez, hogy az új tesztünk a következő futtatáskor már megfelelően fusson le. A helyesen működő tesztek jele a zöld szín, ezért lett ennek a lépésnek „Green” a neve. Az utolsó „Refactor” nevű lépésben az újonnan írt kódunkat át kell szerveznünk azért, hogy minél tisztább kódot kapjunk, ezzel biztosítva, hogy a kódot később jól tudjuk majd olvasni, megérteni és szükség esetén bővíteni azt. Természetesen

a kód átszervezést érintő lépést úgy kell végrehajtanunk, hogy az érintett teszt közben végig „zöld maradjon”. [24]

Azért döntöttem úgy, hogy ezt a módszert alkalmazom a fejlesztés során, mivel így az esetleges módosítások közben azonnal észlelni tudtam a hibákat a rendszerben. A módszer használatával szintén ösztönözni tudtam magamat arra, hogy megfelelő kódlefedettséget érjek el a rendszernek egy kulcsfontosságú komponensében. Később amikor a kliens oldali alkalmazást kézzel teszteltem, akkor könnyebb volt azonosítani, hogy hol rontottam el valamit, mivel a tesztjeim a szerver oldalon biztosították, hogy a problémát valahol a kliensben vagy a két komponens összekötésében kell keresnem.

5.4.2 Vapor-specifikus részek bemutatása

Vapor keretrendszerrel készített webalkalmazásomban egészen egyszerűen tudtam unit teszteket implementálni. A technológiának jó tesztelési támogatottsága van, valamint több cikket is találtam a témában, amelyek segítettek az elindulásban. Az egyik cikkben[10], amit használtam, integrációs teszteket készítettek egy hasonló Web API-hoz, mint az enyém. Az integrációs tesztekben az API végpontjaihoz írtak teszteket, amik aztán a különböző műveleteket egy éles adatbázishoz hasonlóan futó teszt példányon hajtottak végre.

Amikor létrehoztam a szerver oldali alkalmazást már alapból volt benne egy minta teszt osztály, amely egy buta (dummy) végpontot tesztelt. Ez a minta osztály egy speciális teszt célállományban (Test Target) helyezkedett el. Ehhez a speciális célállományhoz a fő alkalmazástól függetlenül lehetett csomagfüggőségeket hozzáadni, például a Vapor keretrendszerhez írt tesztelést segítő csomagot (XCTVapor) vagy mock könyvtárakat. Az XCTVapor csomag egy olyan modul, amely a Swift-ből ismert XCTest könyvtárra épülve tartalmaz Vapor specifikus tesztelői osztályokat, metódusokat, illetve egyéb eszközöket.

Magát az alkalmazást leíró Application osztályt ebben az állományban is inicializálni kell, mielőtt meghívnánk azt a konkrét tesztekben. Ahhoz, hogy tesztspecifikus beállításokkal tudjam inicializálni ezt az osztályt, létrehoztam egy Application bővítményt, amely egy testable nevű függvényt tartalmaz. Ebbe a függvénybe tudtam később beletenni a kizárólag tesztekre alkalmazott beállításokat. Ilyen

[10] https://www.kodeco.com/16909142-testing-in-vapor-4

beállítás volt például a teszt adatbázis adatainak megadása, illetve a mock osztályok inicializálása.

```
final class CouchTests: XCTestCase {
     override func setUpWithError() throws {
          app = try Application.testable()
     }

     override func tearDownWithError() throws {
          app.shutdown()
     }

     func testGetAllCouches() throws { ... }
}
```

Az egyes funkciócsoportokhoz külön osztályokat hoztam létre, amiknek az XCTestCase-ből kellett leszármazniuk. A setUpWithError és tearDownWithError függvényekben tudtam inicializálni az alkalmazást, illetve még elvégezni olyan konfigurációkat, amiket nem kellett minden funkció teszteléséhez elvégeznem. Az XCTestCase leszármazott osztályoknak csak azok a függvényei fognak tesztként viselkedni, amelyek el vannak látva test előtaggal. Az ilyen függvényeket lefuttathatjuk egyesével kézzel is vagy ha az egész tesztkészletet akarjuk futtatni, akkor automatikusan is lefutnak maguktól.

```
final class CouchTests: XCTestCase {
     func testGetAllCouches() throws {
          try app.test(.GET, „/couches”, afterResponse: { res in
               let couches = try res.content.decode([Couch].self)
               XCTAssertEquals(couches.count, 2)
          })
     }
}
```

Miután inicializáltam az Application osztályt a hozzáadott testable függvénnyel, utána a beépített test függvénnyel tudtam meghívni az alkalmazásnak az egyes végpontjait. A test függvénynek egy HTTP igét, a végpont elérési útját, illetve egy afterResponse closure-t kellett megadnom, amelyet akkor futtat majd le a teszt, amikor valamilyen választ kapott az alkalmazáshoz küldött kéréshez. A fenti példában az összes szálláshelyet lekérdező végpontot hívom meg, az afterResponse closure-ben pedig a válaszból kicsomagolom a kapott Couch listát, majd leellenőrzöm, hogy hány elemet tartalmaz. A res változó egy válasz objektum, amely HTTP státuszkódot, illetve a body-ban tetszőleges JSON objektumot tartalmazhat. Ezt a válasz JSON-t a content adattagból lehet lekérdezni, illetve kicsomagolni a decode metódussal. A metódusnak elég csak egy típust megadni, ami utána automatikusan tudja konvertálni az objektumot a megadott

típusra. A válaszobjektum helyességét az XCTAssert... metódusokkal tudtam ellenőrizni. A legtöbbször az AssertEqual, AssertNotNil, illetve AssertTrue metódusokat használtam ezekből.

```
final class CouchTests: XCTestCase {
     func testCreateCouch() throws {
          try app.test(.POST, „/couches”, beforeRequest: { req in
               try req.content.encode(couch1)
          }, afterResponse: { res in
               ...
          })
     }
}
```

A test metóduson elérhető még egy beforeRequest paraméter is, amelynek szintén egy closure-t lehet megadni, amelyet a kérés indítása előtt fog futtatni a teszt. Én ezt a paramétert arra használtam fel, hogy ha POST típusú kérést akartam tesztelni, akkor ebben a kódblokkban tudtam JSON objektummá alakítani azt a Swift objektumot, ami a kéréshez szükséges további adatokat tartalmazta. Ezt a request objektumon található encode metódussal tudtam megtenni, hasonlóan a kicsomagolásnál használt decode metódushoz. Az encode-al készített JSON objektumok a HTTP kérés body részében jutnak el a webalkalmazáshoz.

A tesztek és a speciális alkalmazás létrehozás mellett készítettem még az egyes modellosztályokhoz is bővítményeket. Ezek a bővítmények egy create nevű statikus metódust tartalmaztak, amelyben egy egyszerű adatbázisba való beszúrást implementáltam. Ezek azért voltak hasznosak, mert ha teszt adatokat akartam hozzáadni az adatbázishoz egy teszt elején, akkor nem kellett külön meghívnom az érintett végpontokat a hozzáadáshoz. A create függvény alapértelmezett értékeket is tartalmazott, emiatt paraméter nélkül is meg lehetett hívni, amely nagyon egyszerű függvényhívásokat eredményezett a tesztek elején. A bővítményben található függvény szükség esetén a kapcsolódó entitásokat is létrehozta adatbázisba való beszúrás előtt, ezzel még több sort lehetett megspórolni a tesztekben.

5.5 Mock könyvtárak használata a tesztekben

A szerver oldali komponens kialakításakor számos külső eszközt vagy szolgáltatást is felhasználtam, hogy az egyes funkciók meg tudjanak valósulni. Ilyenek voltak például az adatbázis használata vagy maga az openHAB platform. Általában az ilyen rendszerek tesztelésénél, a külső szolgáltatásokat helyettesíteni szokás egy mock

(hamisítvány) objektummal. Ennek a mock objektumnak a szerepe, hogy helyettesíteni tudja a külső szolgáltatásokat. Mock objektumok használata esetén pedig a tesztek sokkal gyorsabban és megbízhatóbban tudnak lefutni, mivel nem függenek semmilyen külső tényezőtől, amely befolyásolhatná a futást.

Eleinte amikor még nem voltak tervben okosotthon funkciók a rendszerben, akkor próbáltam az integrációs tesztektől egy szinttel lejjebb vinni az üzleti logikának a tesztelését. Service tesztekben gondolkodtam, amikkel a Service osztályaim működését akartam ellenőrizni. Az elképzelés egyáltalán nem volt rossz, viszont megvalósítás nem az elképzeléseim szerint sikerült. A tesztekben megjelenő komponensek közül rosszat mock-oltam ki, emiatt a megírt Service tesztek értelmetlenek voltak. Mock objektumokat a Service osztályokból készítettem, amiket aztán a tesztekben közvetlenül meghívtam.

```
func testGetAllUsersEmpty() throws {
     let userServiceMock = IUserServiceMock()
     Given(userServiceMock, .getAllUsers(willReturn: []))
     XCTAssertEqual(try userServiceMock.getAllUsers().count, 0)
}
```

Jól látszik a fenti kódrészletből, hogy az ellenőrzést megelőző sorban pontosan megmondom, hogy a getAllUsers függvény mivel fog visszatérni, emiatt pedig szerintem felesleges a következő sor, amiben csak ellenőrzöm, hogy tényleg azt adta-e vissza, amit mondtam neki. A másik hiba, hogy magát a mock osztályt példányosítom, ezáltal pedig igazából a mock osztály implementációját sikerült letesztelnem, nem a saját UserService osztályomat. Pár ehhez hasonló tesztet megírtam, mire beláttam, hogy itt valami nem jó és más módszerhez kell folyamodnom.

Az okosotthon funkciók bevezetésekor tértem vissza újból a mock-olás témaköréhez. A funkciók megvalósításához szükségem volt az openHAB platformra és az OpenWeatherMap szolgáltatásaira. Az OpenWeatherMap használata eléggé felborítaná a tesztek futását, ugyanis a szolgáltatással egy város külső hőmérsékletét akartam lekérdezni. Ez elég erősen változik annak megfelelően, hogy az év melyik időszakában kérdezzük azt le, szóval a tesztjeim, amik ezt a rendszert hívják, folyamatosan buktak volna, mert mindig más hőmérsékletet kaptak volna eredményül. Emiatt a tényező miatt egyértelmű volt számomra, hogy ezt a szolgáltatást és az openHAB-ot valamivel helyettesítenem kell a tesztekben.

Mock-ok készítésére két könyvtárat próbáltam ki a projektben, az egyik a Mockingbird[11], a másik pedig a SwiftyMocky[12] volt. Először a Mockingbird-el próbálkoztam.

```
// Mocking
let bird = mock(Bird.self)
// Stubbing
given(bird.canFly).willReturn(true)
// Verification
verify(bird.fly()).wasCalled()
```

A csomag fordítási idejű kódgenerálást használ a mock-ok létrehozására. Ezt úgy tudtam használni, hogy a tesztekben meg kellett hívnom a mock függvényt, aminek a mock-olni kívánt interfész típusát kellett megadnom paraméterként. Utána a given függvénnyel tudtam megadni, hogy hogyan viselkedjen a mock objektumom. A verify függvénnyel pedig a mock megfelelő viselkedését tudtam ellenőrizni.

A SwiftyMocky egy másfajta megközelítést használ a mock objektumok generálására. Ez a csomag egy Sourcery nevű Swift kódgenerátor használatával még a fordítás előtt legenerálja a szükséges osztályokat. A Sourcery onnan ismeri fel, hogy mit kell generálni, hogy egy Mockfile nevű konfigurációs fájlban meg kell adni neki, hogy hol keresse a generálásra jelölt interfészeket. Interfészeket kétféleképpen lehet generálásra jelölni. Az egyik megoldás, hogy az interfész elé elhelyezünk egy „sourcery: AutoMockable” kommentet vagy leszármazunk a speciális AutoMockable interfészből. A Mockfile-ban még olyan információkat is megadhatunk a generált kódra vonatkozóan, hogy milyen importokat tegyen bele vagy milyen célállományokhoz adja hozzá az új fájlokat.

```
mint run swiftymocky generate
```

A kódgenerálást a fenti Terminalban kiadott paranccsal lehet elindítani.

```
let externalHomeService = IExternalHomeServiceMock()
Given(externalHomeService, .setItem(name: „Heating_On”, newState: „ON”,
willReturn: true))
```

A SwiftyMocky esetében azért fontos, hogy már a fordítás előtt meglegyenek a szükséges mock osztályok, mert a fordító a futása során számít ezekre a típusokra, ha nem

[11] https://mockingbirdswift.com

[12] https://rawcdn.githack.com/MakeAWishFoundation/SwiftyMocky/4.1.0/docs/overview.html

ismeri őket, akkor a fordítás elbukik. Miután megvannak a szükséges mock osztályok, a létrehozásuk hagyományos osztályokhoz hasonlóan történik. A viselkedésüket a Given függvényben lehet specifikálni. Ezekben tetszőlegesen megadható, hogy milyen paraméterek esetén milyen visszatérési értéke legyen a mock-olt függvényeknek.

Végül a két könyvtár közül a SwiftyMocky-t használtam a tesztek megírása során. Igaz, hogy a konfigurációja sokkal körülményesebb, mint a Mockingbird-nek, viszont nekem a SwiftyMocky-nál jobban tetszett a függvények viselkedésének megadása.

A mock objektumokat használó teszteknél az alkalmazás inicializálásán kellett némileg változtatnom. Mivel a Service-ek, Controller-ek létrehozásánál és a tesztek törzsében egyaránt szükségem volt a mock objektumokra, ezért az érintett teszteknél pár Service-t és Controller-t nem a többivel egy időben hoztam létre, hanem a TestCase osztályokban. A teszteknél a TestCase osztály setUp függvényében már amúgy is kézzel létrehoztam egy Application példányt, tehát nem volt akadálya annak, hogy itt hozzam létre még mellette a szükséges osztályokat. A Controller-ek esetében azokat be is kellett regisztrálnom a többi végpont közé, ezt szintén meg tudtam tenni a setUp függvényben. Azért, hogy ne legyen duplán beregisztrálva pár végpont, az Application teszt bővítményéből kihagytam ezeket a Controller-eket. A Job-ok tesztelésekor hasonlóan jártam el, a szükséges Job-ot is egy ilyen setUp függvényben állítottam össze és adtam át az alkalmazás várólistájának.

5.6 Kliens és szerver összekötése

Ahogy a tervezés részben bemutattam, a szerver oldali dokumentációt a Swagger nevű eszközzel készítettem el. A két komponens összekötése során ebből a dokumentációból generáltam kliensoldali kódot a Swagger Editor kódgenerátor funkciójával. A generálás egyszerű, csak ki kellett választanom, hogy milyen nyelvű kódot szeretnék és milyen verziójút. Miután kiválasztottam egy tömörített fájlba letöltötte nekem a generált fájlokat az oldal. A kódgenerálás részben hasznos volt, mivel rengeteg kódot nem nekem kellett megírnom, illetve egyéb könyvtárak használatát sem kellett így megtanulnom, mert ezeket már tartalmazta a generált kód. Viszont számos nehézség is adódott a generált kóddal kapcsolatban, amiket kézzel kellett megoldanom a helyes működéshez.

A leíróból generált kódnak két fő része volt, az API végpontok és a modellosztályok. A modellosztályok esetében az okozott nehézséget, hogy a kliensoldali

és szerveroldali modellek nem voltak teljesen azonosak. Voltak típusbeli különbségek, de pár helyen mezők tekintetében is eltértek. Ezt a Swagger-es leíró bonyolította egy fokkal, mivel a leíró által támogatott típusok nem teljesen fedik le a Swift nyelvben is elérhető típusokat, ezért gyakran más típusokat kellett használnom. Az egyik probléma megoldására a generált modellosztályokat elláttam egy API előtaggal, amivel a névütközést sikerült elkerülnöm. A másik problémára pedig bevezettem konverziós osztályokat, amikben kézzel át tudtam konvertálni a kliensoldali modelleket szerveroldalira és fordítva.

A másik probléma az volt, hogy a Swift 5-ös funkcióval generált API osztályokban az Alamofire 4.9-es verzióját használta a kódgeneráló a generálás során. Ez nem egy olyan régi verzió a könyvtárból, viszont vannak már újabbak, sőt szignifikáns változások vannak a 4.9-es és az akkori legfrissebb verzió (5.6) között. A szükséges további könyvtárakat kézzel kellett hozzáadnom a projekthez, és nem tudtam a verzióbeli problémáról, ezért automatikusan a legfrissebb verziót adtam hozzá a projekthez az összes könyvtárból. Ez utólag nem okozott olyan nagy problémát, csak érdemes lett volna odafigyelnem, hogy milyen függőségekre van szükségem a generált kód lefordításához.

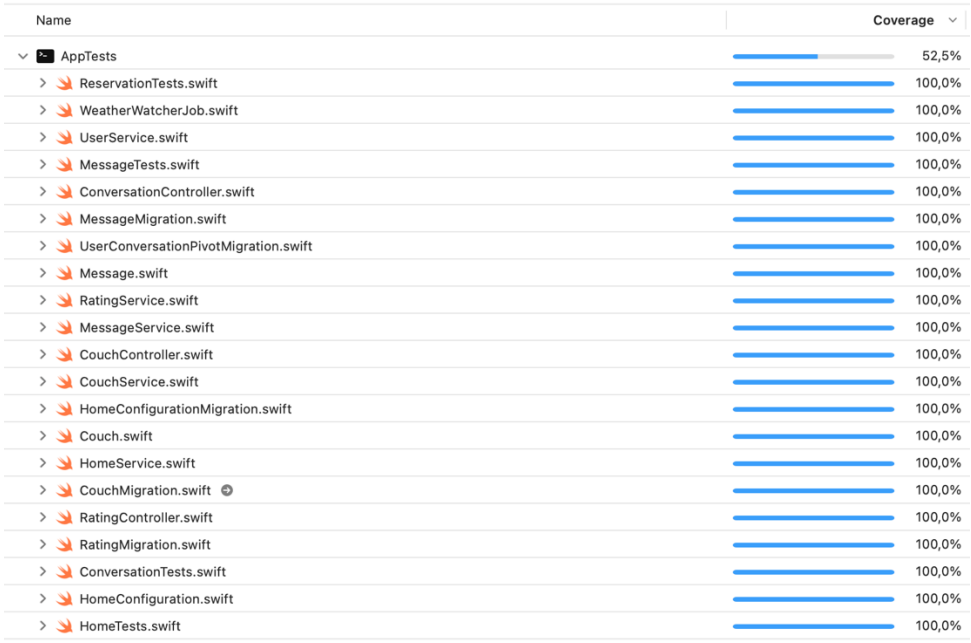
6 Összefoglalás

A diplomaterv elkészítése során nagyon sokat tanultam. Ez volt az első alkalom, hogy saját natív alkalmazást készítettem iOS platformra és szintén az első alkalom volt, hogy Swift nyelvet használtam egy projektben. Ezeken kívül pedig remek alkalom volt ez a projekt, hogy kipróbálhassak olyan új technológiákat, mint például a SwiftUI, amely egy teljesen új megközelítést adott számomra a UI fejlesztésben. Egy új architekturális mintát is megismerhettem a VIPER használatával, amely szerintem egy nagyon jó minta, amely segíti a fejlesztőt, hogy megfelelően szétválassza az alkalmazás felelősségeit. Az implementációja során kicsit sokat kell hozzá kódolni, de lehet, hogy csak első alkalommal tűnt ez ennyire hosszúnak, és a jövőben már gyorsabban fog menni a minta használata.

A szerveroldali megvalósításban is voltak számomra újdonságok. A projekt kezdetekor fogalmam se volt róla, hogy a Swift nyelvet lehet másra is használni, mint iOS vagy macOS alkalmazások fejlesztése. Megismertem a Vapor keretrendszert, mint szerver oldali Swift könyvtárat, és el is mélyültem annak rengeteg funkciójában. Firebase használatában is minimális tapasztalatom csak volt a diplomaterv kezdetéig, pedig egy nagyon hasznos platform, amely sok fontos funkciót tartalmaz, amely megkönnyíti a kliensalkalmazások fejlesztését.

A rendszer tervezésében és megvalósításában még szintén jó tapasztalat volt számomra, hogy nem volt minden funkció előre megtervezve. Emiatt pedig sokkal jobban kellett törekednem arra, hogy minden komponens megfelelően legyen szétválasztva egymástól, hogy a későbbiekben jobban tudjam majd új funkciókkal ellátni a rendszert.

A feladatkiírásban megfogalmazott feladatot az ott leírtaknak megfelelően teljesítettem. A megfogalmazott pontok szerint a dolgozatban bemutattam az iOS platformot, az elkészült rendszer kliens- és szerveroldali architektúráját, valamint felhasználói felületét. A rendszer megvalósításának keretein belül pedig számos okosotthon megoldást is elkészítettem. Külön ki szeretném még emelni, hogy a szerver oldali komponens fejlesztését folyamatosan a TDD módszernek megfelelően fejlesztettem, ezáltal pedig a kulcsfontosságú részek mind le lettek alaposan tesztelve.



6.1. ábra Kódlefedettség Xcode-ban

A lefedettségi jelentésből jól látszik, hogy a legfontosabb, üzleti logikát tartalmazó osztályok mind 100%-os lefedettséget értek el.

6.1 Továbbfejlesztési lehetőségek

A rendszerem elég sok komponenssel rendelkezik, emiatt pedig továbbfejlesztéssel kapcsolatos ötletekből rengeteg van. A kliensoldalt például úgy lehetne még bővíteni, hogy a UIKit-el megvalósított nézetekhez esetleg az Apple fejlesztett-e már SwiftUI-os megoldást, és ha igen, akkor azokat ki lehetne cserélni, hogy egységesen SwiftUI alapon legyen az egész felület megvalósítva.

Az authentikáció és a szálláshelyek képeinek tárolása a Firebase megoldásaival lett megvalósítva. Ezeket a részeket külön át lehetne emelni a Vapor-os szerver oldalba, mivel ezek még nem lettek ott megvalósítva. A képek feltöltését például el lehetne készíteni egy Job-os megoldással, hogy ha egyszerre sok képet töltene fel a felhasználó, akkor azok ne az aktuális HTTP kérésben legyenek feltöltve, hanem a háttérben. Az authentikáció Vapor-os megvalósításával pedig nagyobb hangsúlyt lehetne fektetni a jogosultságkezeléssel, illetve, hogy ne lehessen az összes kérést bárkinek használnia.

Természetesen az alkalmazásban elférne még rengeteg okosfunkció is. A rendszert ki lehetne egészíteni lámpák és egyéb világítási eszközök kezelésével is. Ehhez lehetne olyan automatizmust is készíteni, amely akkor lehet nagyon hasznos, hogy ha sokáig üresen áll a szállás és el szeretnénk rettenteni a betörőket az illetéktelen

behatolástól. Jelenlétszimulációval bizonyos időközönként fel és le lehetne kapcsolni a lámpákat, azt a hatást keltve mintha aktívan lakna ott valaki. Zárak tekintetében a jogosultságkezelés, illetve egy naplózó funkció remek hozzáadott érték lenne. Jogosultságkezeléssel szabályozni lehetne, hogy ki és mikor férhet hozzá a szálláshoz, illetve a tulajdonos adhatna hozzáférést esetleg egy takarítónak is. A naplózással pedig ellenőrizni lehetne, hogy a takarító csak azután ment el a szállásra, miután már elhagyta azt a vendég.

Irodalomjegyzék

[1] ecobnb, „10 Powerful Reasons Why People Love to Travel”, [Online]. Available: https://ecobnb.com/blog/2020/05/reasons-people-love-travel/. [Hozzáférés dátuma: 3 december 2022].

[2] statista, „Most popular hobbies & activities in the U.S. in 2022”, [Online]. Available: https://www.statista.com/forecasts/997050/hobbies-and-activities-in-the-us. [Hozzáférés dátuma: 3 december 2022].

[3] travellina.hu, „Mi az az airbnb?”, [Online]. Available: https://travellina.hu/mi-az-airbnb/#Mi_az_az_airbnb_Az_airbnb_jelentese. [Hozzáférés dátuma: 3 december 2022].

[4] Sarah Mitroff, „Before Airbnb, There Was CouchSurfing”, [Online]. Available: https://www.wired.com/2012/08/before-airbnb-there-was-couchsurfing/. [Hozzáférés dátuma: 3 december 2022].

[5] Sharon Shea, „Smart home or building”, [Online]. Available: https://www.techtarget.com/iotagenda/definition/smart-home-or-building. [Hozzáférés dátuma: 3 december 2022].

[6] gs.statcounter.com, „Mobile Operating System Market Share Worldwide”, [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide. [Hozzáférés dátuma: 2 december 2022].

[7] App Store, „iOS and iPadOS usage”, [Online]. Available: https://developer.apple.com/support/app-store. [Hozzáférés dátuma: 2 december 2022].

[8] Swift.org, „About Swift”, [Online]. Available: https://www.swift.org/about/. [Hozzáférés dátuma: 2 december 2022].

[9] aidanf.net, „Static Typing and Type Inference”, [Online]. Available: https://www.aidanf.net/learn-swift/types_and_type_inference. [Hozzáférés dátuma: 2 december 2022].

[10] aidanf.net, „Optionals”, [Online]. Available: https://www.aidanf.net/learn-swift/optionals. [Hozzáférés dátuma: 2 december 2022].

[11] Apple Developer, „SwiftUI Overview”, [Online]. Available: https://developer.apple.com/xcode/swiftui/. [Hozzáférés dátuma: 2 december 2022].

[12] Paul Hudson, „What’s the difference between @ObservedObject, @State, and @EnvironmentObject?”, [Online]. Available: https://www.hackingwithswift.com/quick-start/swiftui/whats-the-difference-between-observedobject-state-and-environmentobject. [Hozzáférés dátuma: 2 december 2022].

[13] Firebase, „Firebase Realtime Database”, [Online]. Available: https://firebase.google.com/docs/database. [Hozzáférés dátuma: 2 december 2022].

[14] Firebase, „Cloud Storage for Firebase”, [Online]. Available: https://firebase.google.com/docs/storage. [Hozzáférés dátuma: 2 december 2022].

[15] Firebase, „Firebase Authentication”, [Online]. Available: https://firebase.google.com/docs/auth. [Hozzáférés dátuma: 2 december 2022].

[16] Logan Wright, Tim Condon, Tanner Nelson, „Server-Side Swift with Vapor”, [Online]. Available: https://www.kodeco.com/books/server-side-swift-with-vapor/v3.0/chapters/1-introduction. [Hozzáférés dátuma: 2 december 2022].

[17] Logan Wright, Tim Condon, Tanner Nelson, „Server-Side Swift with Vapor”, [Online]. Available: https://www.kodeco.com/books/server-side-swift-with-vapor/v3.0/chapters/5-fluent-persisting-models. [Hozzáférés dátuma: 2 december 2022].

[18] openHAB, „Introduction”, [Online]. Available: https://www.openhab.org/docs/. [Hozzáférés dátuma: 3 december 2022].

[19] openHAB, „Concepts”, [Online]. Available: https://www.openhab.org/docs/concepts/. [Hozzáférés dátuma: 3 december 2022].

[20] Balsamiq, „What Are Wireframes?”, [Online]. Available: https://balsamiq.com/learn/articles/what-are-wireframes/. [Hozzáférés dátuma: 13 november 2022].

[21] Michael Katz, „Getting Started with the VIPER Architecture Pattern”, [Online]. Available: https://www.kodeco.com/8440907-getting-started-with-the-viper-architecture-pattern. [Hozzáférés dátuma: 14 november 2022].

[22] Vapor Docs, „Vapor: Advanced -> Queues”, [Online]. Available: https://docs.vapor.codes/advanced/queues/. [Hozzáférés dátuma: 30 november 2022].

[23] Paul Hudson, „Using Touch ID and Face ID with SwiftUI”, [Online]. Available: https://www.hackingwithswift.com/books/ios-swiftui/using-touch-id-and-face-id-with-swiftui. [Hozzáférés dátuma: 2 december 2022].

[24] testdrive.io, „What is Test-Driven Development?”, [Online]. Available: https://testdriven.io/test-driven-development/. [Hozzáférés dátuma: 26 november 2022].