# Szálláskiadó rendszer fejlesztése iOS platformon okosotthon integrációval

Diplomaterv 1 – Készítette: Séllyei Bence (VF90WF)

## Összefoglaló

A couchsurfing alapötlete az, hogy ha valakinek van egy felesleges szobája, ágya, kanapéja, akkor azt megoszthatja különböző utazókkal. Általában a vendéglátók nem csak a helyet biztosítják az utazóknak, hanem idegenvezetővé is válnak és megmutatják kedvenc helyeiket az adott városban, cserébe illik valamivel meghálálni a házigazda vendégszeretetét. A hála általában nem feltétlenül anyagi jellegű, ez az úgynevezett szolgáltatás olyan utazóknak lett kitalálva, akik csak egy-két napot töltetének egy adott városban vagy átutazóban vannak és épp nincs hol megszállniuk, egy éjszakáért pedig nem mennének szállodába. A házigazdák pedig általában új emberek megismerése miatt ajánlják fel felesleges fekvőhelyüket az utazóknak. Ennek az ötletnek vagy szolgáltatásnak szeretne egy platformot adni ez az alkalmazás, ahol szabadon böngészhetünk utazóként kirándulásaink során, ha éppen nincs hol megszállnunk egy éjszakára.

## Architektúra

Az alkalmazást iOS platformon valósítottam meg, amelyhez az Apple új UI megoldását, a SwiftUI-t használtam fel, az alkalmazás üzleti logikájához és adattárolásához pedig a Firebase szolgáltatásait vettem igénybe.

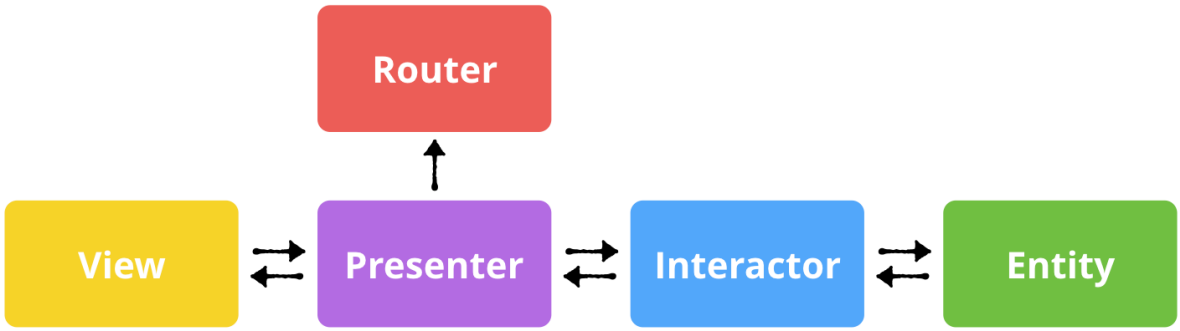
### SwiftUI

A SwiftUI egy platformfüggetlen UI keretrendszer, amellyel deklaratív módon lehet felhasználói felületet tervezni és készíteni Apple eszközökön futó alkalmazásaink számára. Ezzel az egy API-val képesek vagyunk macOS-re, iOS-re, valamint a többi Apple által fejlesztett operációs rendszerre felhasználói felületet írni mindezt az ilyen alkalmazásokhoz használt Swift nyelven. A keretrendszerhez tartozik még az Xcode-ban elhelyezett grafikus tervező, amely az általunk írt SwiftUI kód alapján képes előnézetet mutatni az alkalmazásról anélkül, hogy azt el kelljen indítanunk egy valódi eszközön vagy szimulátoron. A SwiftUI API

még nem fedi le teljesen a korábbi UIKit könyvtár funkcionalitásait, azonban lehetőségünk van a két könyvtár együttes használatára is, így némi plusz kód megírásával bármikor használhatunk UIKit-es vezérlőket SwiftUI felületünkhöz.

## VIPER architektúra

A VIPER egy architekturális minta mint például az MVC vagy az MVVM, de ezekhez képest a VIPER arra készteti a fejlesztőt, hogy az egyes komponensei között minél jobban szétválassza a felelősségeket ezzel betartva 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, fogadja és továbbítja a felhasználói interakciókat a következő komponens felé. Az I mint Interactor felelős az adatok lekéréséért 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.




1. ábra Az architektúra komponenseinek kapcsolata

A fenti 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 definiál bizonyos property wrapper-eket (más nyelvekben hasonlóak az annotációkhoz), amiket hogy ha ráteszünk egy property-re, akkor a UI megváltozásakor (pl.:

szöveget írunk egy szövegmezőbe), akkor a UI rögtön befrissíti a property értékét és fordítva. Emiatt a keretrendszer működése inkább jobban illeszkedik az MVVM mintához, mint a VIPER-hez. Ennek ellenére én mégis VIPER architektúrát használtam az alkalmazásban annyi változtatással kiegészítve, hogy a Presenter komponense a moduloknak hasonlít egy MVVM-beli ViewModelhez.

## UIKit vezérlők használata SwiftUI-ban

Ahogy korábban írtam már a SwiftUI keretrendszer tudása nem azonos a UIKit könyvtár tudásával, ezért sok esetben vissza kellett nyúlnom a UIKit-es megoldásokhoz. Az egyik ilyen SwiftUI-os hiányosság a térkép nézetnél került elő, ahol nem lehetett a térképen megjelenő annotációkhoz úgynevezett Callout nézetet hozzáadni. Ez azért lett volna fontos, hogy később a felhasználó a térképen ki tudjon választani helyszíneket, amiket az alkalmazás annotációkkal jelez a térképen. Ennek megvalósításához fel kellett készítenem a MapKit-es MKMapView-t arra, hogy meg tudjon jelenni SwiftUI nézetként. Ezt úgy lehet megtenni, hogy készítünk egy SwiftUI nézetet, ami nem a View-ból származik le, hanem egy UIViewRepresentable nevű osztályból. Ennek az osztálynak három metódust kell megvalósítania. Az első a makeUIView, ebben lehet létrehozni és tetszés szerint inicializálni a megfelelő UIKit-es View-t. Az updateUIView metódusban lehet frissíteni a UI-t. Az egész megvalósításához szükségünk van még egy belső Coordinator nevű osztályra. Ez az osztály fogja megvalósítani a nézetekhez tartozó Delegate-eket. Ezt az osztályt minden esetben létre kell hozni, azért hogy a UIViewRepresentable makeCoordinator metódusát felül tudjuk definiálni.

UIKit View-kon kívül még ViewControllereket is hasonló módon tudunk használni SwiftUI-al, annyi különbséget kell csak eszközölnünk, hogy UIViewRepresentable helyett UIViewControllerRepresentable-ből származunk le az osztályunkban.

## Firebase

A Firebase egy alkalmazásfejlesztői platform, amely eszközök megosztásával segíti a szoftverfejlesztőket, hogy egyszerűbben és gyorsabban tudjanak mobil és webes alkalmazásokat fejleszteni. A platform rengeteg olyan szolgáltatást bocsát a rendelkezésünkre, amelyeket alapesetben nekünk kellene megírnunk és megterveznünk,

ahhoz hogy az alkalmazásunkban fel tudjuk használni. Ilyenek például az authentikáció, adatbázis használat, fájltárhely, notifikációk kezelése vagy ezeken kívül még sok más. Ezeket a szolgáltatásokat a Firebase a felhőben teszi elérhetővé számunkra, és nagyon kevés erőfeszítéssel tudjuk ezeket felhasználni alkalmazásainkban.

Én három Firebase alapú szolgáltatást használtam fel az alkalmazásomban. Fájlokat, azon belül is képeket a Firebase Storage-ba tölt fel és le az alkalmazás futása közben, az egyes entitások adatait a szolgáltatás adatbázisába töltöm fel, és a felhasználók azonosítását is a Firebase Authentication eszközével végzem el az alkalmazásban.

## Firebase Realtime Database, Firebase Storage

A Firebase egy realtime, felhőalapú, NoSQL adatbázist ad a fejlesztőknek, hogy strukturáltan tudják tárolni alkalmazásuk futtatásához szükséges adataikat. Az adatbázisba JSON-ként kerülnek be az adatok, ezért nincs szüksége a működéshez semmilyen előre megtervezett adatmodellre.

Használata úgy működik, hogy létre kell hoznunk egy DatabaseReference objektumot, ami egy URL-lel megadott adatbázisban, egy névvel hivatkozott csomópont alá tudunk beszúrni vagy lekérdezni adatokat.

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

A fenti kis kódrészlet egy új objektum adatbázisba való beszúrását mutatja meg. Az említett DatabaseReference objektumon meghivatkozunk egy új gyereket, ide fog majd kerülni a beszúrandó objektumunk. Ennek a gyereknek pedig meg kell hívni a setValue függvényét, amivel be is szúrjuk az adott helyre az objektumot. A setValue paraméterként egy Dictionary típust vár, amiben kulcs-érték párokban vannak felsorolva az objektum tulajdonságai.

```
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 függvényekkel lehet megadni, úgy hogy ezeket a referenciaobjektumon hívjuk meg egymás után fűzve.

A Firebase Storage használata nagyon hasonló a Database-éhez, itt is szükségünk van egy referenciaobjektumra, amin beszúráskor létrehozunk egy új gyereket, majd feltöltjük hozzá a kívánt fájlt. Fontos, hogy a fájlok URL-jét feltöltés után le is töltsük, hiszen ezzel lehet majd a továbbiakban hivatkozni arra a fájlra.

## Navigáció SwiftUI-ban

SwiftUI-ban a navigáció eltérő módon működik, mint más UI technológiákban vagy UIKit-ben. Megszokott módszer, hogy valamilyen függvényhívás hatására tudunk újabb nézetekre navigálni az éppen aktuálisról, viszont a 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 lenyomása után, valamint egy UI elrendezést, hogy miként is rajzolódjon ki a speciális gombunk.

Egy gombnyomásra azonnal végbemenő navigáció viszont nem minden esetben az amire szüksége lehet a fejlesztőknek, sokszor szeretnénk valamilyen műveletet még elvégezni navigáció előtt (például egy Login gomb esetében be szeretnénk jelentkeztetni a felhasználót, és csak akkor elsütni a navigációt, amikor a bejelentkezés sikeres volt). SwiftUI ezekre az esetekre is kínál megoldást. A NavigationLink-nek van még egy isActive property-je, amit ha igaz értékre állítunk, akkor történik meg maga a navigáció. A speciális gomb kinézetét pedig egy beépített EmptyView-val el is tűntethetjük, így igazából nem fog kirajzolódni a gomb, és csak az isActive értékének állításával tudjuk elindítani a navigációt.

## Főbb komponensek

Az alkalmazás egy tabokból álló alkalmazás, ami azt jelenti, hogy jelen esetben négy különböző nézet (tab) mindig meg van nyitva az alkalmazásban és ezek között a képernyő alján lévő navigációs sávval lehet navigálni. A négy tab a következő nézetekből áll:

- Trips: egy térképes nézet, ahol böngészni lehet, hogy mely városokban milyen szálláshelyek elérhetőek
- Couches: innen lehet új szálláshelyet hozzáadni és sajátjainkat a leggyorsabb útvonalon elérni
- Messages: itt érhetjük el a különböző felhasználókkal váltott üzeneteinket
- Profile: innen lehet kijelentkeznie a felhasználónak

VIPER architektúrának megfelelően minden különböző nézethez készítettem egy modult, ami a VIPER komponenseit tartalmazza az adott nézethez. Ezek a modulok egyik esetben sem tartalmaznak entitás osztályokat, ezeket egy központi helyre gyűjtöttem össze, mivel nem igaz, hogy minden modulhoz különböző entitások tartoznak. A navigációt tartalmazó Router komponens sincs minden modulban, mert vannak olyan nézetek, ahonnan előre nem lehet navigálni, csak visszafelé, ehhez viszont egy PresentationMode típusú property dismiss metódusát használtam, ami a navigációs veremről mindig a legfelső elemet veszi le.

### Register és Login modulok

A Login az alkalmazás kezdőképernyője, innen lehet bejelentkezni vagy navigálni a regisztrációra. Az authentikációt Firebase-el valósítottam meg, a bejelentkezési űrlap inputjainak validációját pedig egy Github repository[1] segítségével oldottam meg. A validáció igazából a Firebase-től visszajövő hibaüzeneteket jeleníti meg a felhasználónak, ha például olyan emailt adott meg amivel nincsen regisztrált felhasználó vagy rossz jelszót gépelt be. A regisztrációnál az új felhasználó felvételén kívül van még egy adatbázis művelet, amivel beszúrja az adatbázis User csomópontja alá az új felhasználót. Erre későbbi műveleteknél szükség lesz, ezért kellett ezt itt lekezelni.

[1] https://github.com/jnewc/SwiftUI-Validation

## Map modul

Ez az alkalmazásnak az eddigi legbonyolultabb felülete. Tartalmaz egy térképet, egy keresőmezőt, valamint egy egyéni SheetView-t, amit fel-le lehet húzogatni a képernyőn kitakarva ezzel a térképnek egy részét. Mind a térkép, mind a keresőmező UIKit-es elemek, amiket felkészítettem, hogy SwiftUI nézetként meg tudjanak jelenni. Erre a felkészítésre, azért volt szükség, hogy a térképen később lehessen kijelölni annotációkat és hogy az annotáció kijelölésekor lehessen CalloutView-t megjeleníteni hozzájuk. SearchBar átírása pedig azért volt elengedhetetlen, mert SwiftUI-ban nem találtam, olyan SearchBar-t, ami nem karakter gépelésenként indította volna a keresést, hanem a billentyűzet jobb alsó sarkában található Return gomb megnyomására. Gépelés közben való keresés nem lett volna megvalósítható, mert a térképes nézet felett országokra és városokra lehet rákeresni, ezek azonban nincsenek mind betöltve a memóriába alkalmazás futása közben. Helyette olyan megoldást választottam, hogy a gépelés befejezésekor a beírt szövegből Geocoding-al kiszámolja a hely koordinátáit és arra állítja át a program a kamera képét a térképen. Ország és városnevek Geocode-olására az Apple CLGeocoder megoldását használtam fel.

## Couches és NewCouch modul

A Couches nézetről tudunk átnavigálni a NewCouch nézetre, ahol új szálláshelyet tudunk felvenni. A Couches nézet még tartalmaz egy egyszerű listát, ahol saját szálláshelyeink vannak listázva, innen lehet részletes nézetükre is navigálni.

A NewCouch modul nézete egy egyszerű űrlap nézet, ahol egysoros szövegmezőket, egy többsoros szövegmezőt, egy számlálót és egy képválasztót lehet megtalálni. Ha mindent kitöltöttünk (ami amúgy nem kötelező), akkor a képernyő tetején lévő NavigationBar Save gombjával lehet elmenteni a szálláshelyet. A felhasználó a szálláshelyének a címét tudja megadni, ez azonban később nem lesz látható az alkalmazásban, erre azért van szükség, hogy az alkalmazás ebből ki tudja számítani a hely koordinátáit. A cím alapú koordinátaszámolásra a CLGeocoder nem volt megfelelő, legtöbb esetben soha nem találta meg a keresett címhez tartozó koordinátákat. Githubon viszont találtam egy könyvtárat[2],

[2] https://github.com/lminhtm/LMGeocoderSwift

ami egy olyan Geocoder volt, ami egyszerre használja a Google és az Apple ilyen szolgáltatását, ezzel biztosítva a megfelelő pontosságot. Ezt felhasználva is vannak esetek, amikor nem találja meg a keresett koordinátákat, ebben az esetben hibára futunk és egy Alert-el jelezzük a felhasználónak, hogy nem sikerült a beszúrás.

A szálláshelyekhez képeket is lehet beszúrni. ImagePicker-ből is a UIKit-eset használtam fel, ezt találtam a legegyszerűbb módszernek, hogy elérjem az eszköz beépített galériáját és onnan tudjon képet választani a felhasználó. A képeket Firebase Storage-ba tölti fel az alkalmazás a szállás adatbázisba beszúrásával párhuzamosan. Tapasztalataim alapján a fájlfeltöltés majd a fájl URL-jének letöltése egy fájl esetén is sok idő volt, ezért olyan megoldást választottam, hogy először a szálláshelyet veszi fel az adatbázisba az applikáció, majd feltölti egyesével a képeket, letölti hozzájuk az elérési útvonalakat, majd amikor az összes URL előállt, akkor befrissíti ezeket az adatbázisban. Ahhoz, hogy az URL-eket később hozzá lehessen adni az objektumhoz Firebase-ben, előre felvettem minden Couch objektumba egy URL listát egy placeholder elemmel, így történik minden esetben az adatbázisba beszúrás és az URL-ek befrissítésekor cserélődik ki ez a placeholder elem az URL-ekkel.

## CouchDetails és Reservation modul

A CouchDetails modul tartalmazza a szálláshelyek részletes megjelenítéséért felelős nézetet és logikát. A nézet felépítésben hasonlít az új szálláshely hozzáadós nézethez, viszont ez a nézet csak read-only mezőket tartalmaz. A NewCouch-hoz képest a felület kapott még egy értékeléseket tartalmazó szekciót, ami megmutatja a felhasználó számára, hogy az 5-ből hány csillagos értékelésen áll az aktuális szálláshely és hány szavazatot kapott idáig. A csillagok mellett van még egy gomb, amivel az értékeléseknek egy részletesebb nézetére tud továbblépni a felhasználó.

A részletes nézet tetején található NavigationBar-on elhelyeztem egy kis naptár ikonos gombot, ezzel lehet megnyitni a naptárt, ahonnan kiválaszthatja a felhasználó hogy mettől meddig szeretné lefoglalni a szálláshelyet. Ez a naptár egy Github-on talált MultiDatePicker[3]. Az Apple UI vezérlői között nem találtam olyan naptárt, amivel ki lehetne választani két dátumot, melyek egy tartomány adnak meg. A vezérlőhöz nem volt semmilyen package,

[3] https://github.com/peterent/MultiDatePicker

amivel könnyen hozzá tudtam volna adni a forráskódot az alkalmazáshoz, ezért lemásoltam a szükséges forrásfájlokat a projektembe. Ez utólag belegondolva nem volt nagy probléma, mert így lehetőségem volt módosítani a naptár forrásfájljait, ezzel tudtam hozzá plusz funkciókat hozzáadni. Saját funkciómmal kiegészülve, naptár példányosításakor megadhatunk neki egy dátumtartományokból álló listát, amit majd kiszürkítve jelenít meg a naptár, ezeket nem lehet majd kiválasztani. Így nagyon egyszerűen lehet a felhasználó számára jelölni, hogy mely dátumok foglaltak a szálláshelyen.

A nézet az alján tartalmaz még egy Toolbar-t, ami egy gombot és egy kis szöveget tartalmaz. Itt jeleníti meg a nézet, hogy mely dátumokat lettek kiválasztva a felhasználó által, és ha van kiválasztva dátum, akkor a gomb megnyomásával lehet navigálni a foglalás részleteit tartalmazó nézethez. Toolbar-hoz hozzáadni olyan gombot, aminek megnyomására navigáció megy végbe nem működött az eddigi módszer szerint. Az új megoldás az lett, hogy a gomb csak egy Bool-t állítgat, a navigációs link pedig a nézet gyökérnézetében van, elrejtve a felhasználó elől.

A foglalás részleteit tartalmazó nézet a Reservation modulban található. Ez egy nagyon egyszerű nézet, pár alapvető adatot tartalmaz és itt lehet beállítani, hogy hány vendég fog megszállni a megadott idő alatt. Sikeres foglalás után az alkalmazás a foglalt dátum első napjára ütemez egy notifikációt.

## Conversation és Message modul

Az alkalmazásba egy chatszerű funkció is bekerült, mellyel üzeneteket tud váltani egymás között két felhasználó. A Conversation modul tartalmazza egy listában, hogy a bejelentkezett felhasználónak kivel vannak beszélgetései, ha pedig egy beszélgetést megnyit a felhasználó, akkor egy iMessage-hez hasonló chat nézetben láthatja a küldött és kapott üzeneteket.

Egy beszélgetést csak foglalással lehet létrehozni. Sikeres foglaláskor azonnal létrejön a beszélgetés a foglaló fél és a szállásadó között, valamint egy automatikus üzenet is kiküldésre kerül a foglalótól. Az üzenetek érkezéséről a felhasználó nem kap notifikációkat az alkalmazástól.

## Ratings modul

Az alkalmazásban lehetősége van a felhasználónak értékelnie az egyes szálláshelyeket. Az értékelésnek két része van, egy számos és egy szöveges. Szöveges értékelés megadása opcionális. Értékeléseket a Ratings modul nézetén keresztül adhat a felhasználó, melyet egy szálláshely részletes oldaláról érhet el. Ezen a nézeten a korábbi értékelések és egy ezekből összeállított összesítő jelenik meg. Az összesítő mutatja az adott értékelések átlagát, valamint azt, hogy az értékelések közötti arányok hogyan oszlanak meg. Alatta egy listában lehet megnézni, hogy milyen értékeléseket adtak a felhasználók idáig.

## Service komponensek

VIPER-ben az üzleti logikát az Interactor osztályokban kellene elhelyezni, azonban én az entitás osztályokhoz hasonlóan ezeket a logikákat külön osztályokba szerveztem ki, mert ezeket nem csak egy modul használja fel és a későbbiekben, ha a backendet le szeretném cserélni egy másik megoldásra, akkor nem kell az összes Interactort újraírni, elég csak a Service komponenseket.

Ennek megfelelően az adatbázis „táblákhoz” és a Firebase Storage kezeléshez tartozó műveleteket egy-egy Service osztályba szerveztem ki, melyeket Dependency Injectionnel szúrtam be az összes Interactor osztályba. Az Interactorok így protocolokon keresztül láthatják a Service-ek interfészeit, az hogy ezeknek az implementációi Firebase-re épülnek, azt nem látják.

A CouchService új szálláshely felvételére, frissítésére, a kurrens felhasználóhoz tartozó szálláshelyek és az összes szálláshely (kivéve a kurrens felhasználóét) lekérdezésére képes. A UserService új felhasználó felvételére és annak lekérdezésére képes, az ImageManager pedig csak képek feltöltésére alkalmas. Külön képek letöltését végző műveletet ide nem implementáltam, mert találtam egy olyan SwiftUI ImageView-t[4], ami képes URL-lel hivatkozott képek letöltésére és megjelenítésére is.

A ConversationService új beszélgetés létrehozására, frissítésére és a bejelentkezett felhasználó beszélgetéseinek lekérdezésére képes. A MessageService ugyanazokra a

[4] https://github.com/SDWebImage/SDWebImage

műveletekre képes, mint a ConversationService csak neki az üzenetek „táblához” van hozzáférése. A Rating és Reservation Service osztályok csak új létrehozására és összes lekérdezésére képesek.

Az alkalmazás authentikációkezeléséhez tartozó műveletei egy AuthenticationManager osztályba kerültek. A Service-ekhez képest ezt az osztályt nem Dependency Injectionnel szúrtam be a szükséges helyekre, hanem a Singleton mintát használtam. Ebbe a Manager osztályba kerültek a regisztráláshoz, bejelentkezéshez és kijelentkezéshez szükséges függvények.

# Backend alkalmazás

## Technológia

Ebben a félévben elkezdtem lefejleszteni egy saját backend alkalmazást is a couchsurfing iOS applikációm mellé. A backend szolgáltatás fejlesztéséhez a szintén Swift programozási nyelvet használó Vapor keretrendszert használtam. A Vapor egy nyílt forráskódú webes keretrendszer, amivel könnyedén tudunk RESTful API-kat, webes vagy websocket-et használó real-time alkalmazásokat fejleszteni. A keretrendszer számos funkcióval és csomagokkal rendelkezik, amikre szükségünk lehet egy webalkalmazás fejlesztése során. Ilyen például a Fluent, ami egy Object Relational Mapping (ORM) keretrendszer, a Leaf, amivel dinamikus HTML oldalakat lehet generálni, de ezeken kívül sok más funkció és middleware található a frameworkben.

## Architektúra

Az alkalmazásnak két nagyobb része van. Az egyik az entitások és a hozzájuk tartozó migrációk, amik leírják az adatbázistáblák sémáját és a kapcsolatokat a táblák között. Az alkalmazás másik része pedig a kontroller osztályok, amikben az egyes műveletek végpontjai és az entitásokhoz tartozó üzleti logikák találhatók.

### Adatbázis és Fluent

Az alkalmazás adatbázisának a Vapor által ajánlott PostgresSQL-t használom, az adatbázisrekordok és objektumok közti különbségek elfedésére pedig a Fluent ORM keretrendszert használom. A Fluent-nek köszönhetően az adatbázissal keveset kellett foglalkoznom, csak telepítés előtt el kell indítani a Docker konténert, amiben az adatbázis működik. Két ilyen konténert hoztam létre az alkalmazáshoz, egyet a teszteléshez, egyet pedig a rendes futáshoz.

Minden entitáshoz külön készítettem egy entitásosztályt, egy migrációt és egy kontrollert. Az entitásoknál mindig meg kell adnunk egy id-t, amit ID annotációval kell ellátnunk, a többi mező esetében a Field annotációt kell használni. Táblák között tudunk kapcsolatokat is definiálni, ilyenkor a Parent, Child, Children, Siblings annotációkat kell használnunk. Egy-

egyes kapcsolathoz Parent-Child, egy-többeshez Parent Children, több-többeshez pedig Siblings annotációt kell használni. Több-többes esetben létre kell még hozni egy kapcsolótáblát, aminek saját sémája van, ezért ennek is kell migrációs osztály. A migrációk esetében dönthetünk, hogy kézzel szeretnénk kezelni a futtatásukat, vagy minden alkalmazásindításnál fusson le az összes migráció.

## Kontrollerek

Az alkalmazás végpontjai és az üzleti logikák találhatók a kontroller osztályokban. Egy boot nevű függvényben tudunk route csoportokat definiálni, amik tartalmazni fogják az egy csoportba tartozó végpontokat. Végpontot HTTP igével és opcionális paraméterekkel vagy kulcsszavakkal tudunk létrehozni. Végpont felvételekor meg kell adnunk még egy kontroller osztályban lévő függvényt, ami majd futni fog, ha elérte az alkalmazás azt a végpontot.

## Unit tesztek

Az alkalmazást TDD módszernek megfelelően fejlesztettem, mivel korábban az iOS alkalmazáshoz nem készítettem teszteket egyáltalán. TDD alapján először magukat a teszteket írtam meg, majd utána készítettem el a kontrollerek implementációit. Az alkalmazást tesztelni a test metódussal lehet, aminek a HTTP igét, végpont URL-jét, valamint egy closure-t kell megadnunk, amiben a választ tudjuk ellenőrizni, hogy minden megfelelően működik-e.

Nekem a tesztek környékén sok problémám volt az entitásokhoz kapcsolódó objektumok betöltésével. Minden lekérdezésnél a Fluent automatikusan betölti a kapcsolódó entitásokat, viszont ezeket hivatkozni nem mindig egyszerű. Postmannel való tesztelésnél egyszerű dolgom volt, mert ott JSON választ kaptam, ahol láthattam, hogy minden adat megérkezett, amit vártam. Unit tesztek esetén viszont az API által visszaküldött JSON választ a tesztalkalmazás rögtön átalakította entitás osztálybeli objektummá. Ebben az esetben a kapcsolódó objektumokat a $ operátorral lehet hivatkozni (eager loading). Sibling esetén ez az operátor nem működik (vagy csak én nem jöttem rá, hogyan kell helyesen használni), valamint nem teljesen egyértelmű az sem, hogy mikor kell használni ezt az operátort. Eleinte azt hittem, hogy csak az adatbázisból lekérdezett objektumok esetén kell használni az operátort, de ha kézzel kapcsoltam össze objektumokat, amiket utána beszúrtam az

adatbázisba és beszúrás után hivatkoztam az összekapcsolt elemeket, akkor is használni kellett az operátort.

## Mockolás

A fent említett tesztelésre Unit tesztként hivatkoztam, viszont ezek a tesztek valódi HTTP kéréseket indítanak a szerver felé, amik egy valódi adatbázison végeznek módosításokat vagy lekérdezéseket. Adatbázisból ugyan volt teszt és éles adatbázis is, hogy a tesztek egy teljesen új állapotból fussanak és ne rontsák el az éles adatbázis állapotát, viszont a tesztek egyik problémája így is fennáll, mégpedig, hogy a futtatásuk nagyon erőforrásigényes.

Az erőforrásigény kiküszöbölésének egy megoldása, ha alacsonyabb szintre visszük le a teszteseteket és minden külső függést megpróbálunk eltávolítani a tesztkörnyezetből. Mivel a szerver oldalon az üzleti logika Service osztályokba lett kiszervezve, amiket interfészeken keresztül érhetünk el, ezért ezeket a Service-eket kell kicserélni, hogy a tesztkörnyezetben ne használjanak adatbázisokat. Mock osztályokat készítettem a Service-ek kicserélésére, amelyek megvalósítják a szükséges interfészeket, viszont az eredeti implementációhoz képest egy sokkal egyszerűbb funkcionalitást tartalmaznak.

Mock-ok készítésére két könyvtárat próbáltam ki, a Mockingbird[5]-t és a SwiftyMocky[6]-t. Először a Mockingbird-el próbálkoztam. Ennek a könyvtárnak a használata rendkívül egyszerű, mock objektumot a mock függvénnyel lehet létrehozni és paraméterként a mockolni kívánt interfészt kell megadnunk. Mockingbird-el viszont felparaméterezni a mock-ot, hogy milyen függvényre milyen értékeket adjon vissza már nagyon körülményes volt. SwiftyMocky könyvtár viszont az előzőnek az ellentéte, a könyvtár telepítése és mock-ok generálása nehezebb, viszont a használata és felparaméterezése már jóval könnyebb. Mock generálás nem futási időben történik, mint a Mockingbird esetében, hanem előre le kell generálni a szükséges mock-okat. A generáláshoz három dologra van szükség. Az első, hogy minden interfészhez amihez szeretnénk mockot generáltatni, kell tenni egy „`//sourcery: AutoMockable`” kommentet. A második, hogy készíteni kell egy Mockfile-t, ami a generáláshoz tartozó konfigurációt adja meg. Ide kell megadni, hogy melyik mappákban keresse az AutoMockable interfészeket, milyen célmappába, célprojektbe generálja a mock-okat és

[5] https://mockingbirdswift.com
[6] https://rawcdn.githack.com/MakeAWishFoundation/SwiftyMocky/4.1.0/docs/index.html

milyen importokat tegyen a generált osztályok elejére. Harmadik lépésként pedig egy parancsot kell terminalból futtatni, amely elvégzi a tényleges generálást.

# Okosotthon funkciók

## Bevezetés

Az emberek utazásai során rengeteg probléma felléphet a szálláshelyekkel kapcsolatban, főleg ha szállásadója felkészületlenül vagy rugalmatlanul várja vendégeit. Velem előfordult már több alkalommal is, hogy a hideg időjárás ellenére sem üzemelték be a fűtőrendszert a szálláshelyen, bekapcsolása pedig további kellemetlenségeket és egyeztetéseket igényelt mindkét fél részéről. Az ilyen problémák elkerülésére készült el ennek a rendszernek a prototípusa, amely megkönnyíti majd az okoseszközök használatát a szálláshelyeken és terhet vesz le a szállásadók válláról.

## Technológia

## openHAB

Az egyes okoseszközök vezérlésére az openHAB nevű, open-source, 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, csupán a gyártó honlapjáról kell letölteni 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 esetén pedig hozzá is adja őket a rendszerhez.

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

## Okosfunkciók megvalósítása

Bevezetésként három okos funkciót valósítottam meg a rendszerben. A háromból kettő sima kapcsoló irányítása volt, amellyel fűtést vagy légkondicionálót kapcsolgathatunk. A harmadik egy komplexebb, az előző kettőre épülő funkció, amely bekapcsolt állapotnál figyeli a szálláshely lokációjának külső hőmérsékletét és annak megfelelően kapcsolja ki vagy be a hűtő-, fűtőrendszereket.

A kapcsolók megvalósításához tehát állapotot lekérdező és azt módosító műveleteket használtam fel. Ezeket könnyen el lehetett érni a már említett openHAB REST API-n keresztül egy GET és egy POST kérés indításával. Mivel a rendszeremben minden szálláshelyhez tartozhatnak ilyen kapcsolók, ezért a saját szerver oldali alkalmazásomba fel kellett vegyek az adatbázisba egy új táblát, ami a kapcsolókat tárolja. Minden kapcsolóhoz tartozik egy ID, szálláshely ID, kapcsolótípus és állapot mező. A fent említett lekérdező és módosító műveletek pedig a saját szerverembe is bekerültek ugyanúgy GET és POST végpontok formájában.

A harmadik, komplexebb funkcióhoz (WeatherWatcher) a hőmérsékletfigyelés megvalósítását nem az openHAB-on végeztem, hanem a saját szerver oldali komponensben. Hőmérsékletet az OpenWeather API-jától egy városnév megadásával kérdeztem le. Vapor keretrendszerben a fejlesztők rendelkezésére állnak úgynevezett Job-ok, amikkel automatikusan lehet folyamatokat indítani a szerver futása során. Job-ot kétféleképpen lehet indítani, ütemezetten vagy kézzel, mondjuk HTTP kérés formájában. A funkcióhoz mindkét típust felhasználtam: egy ütemezett folyamat minden órában lekérdezi az érintett városokhoz tartozó hőmérsékletet, majd egy város-hőmérséklet összerendelést továbbad egy kézzel indított folyamatnak, amely szükség esetén le-fel kapcsolja az érintett eszközöket.

# Kliens- és szerveroldal összekötése

## Megvalósítás

A kliensalkalmazás a saját szervertől idáig különállóan működött, a kliens az adatokat a Firebase-ről kérdezte le, és az új rekordokat, módosításokat is oda mentette el. Funkciók bővülésével és további külső szolgáltatások felhasználásával szükségessé vált a két réteg összekötése.

Az összekötéshez szükség volt egy szerver oldali dokumentációként is funkcionáló API leíróra. Dokumentációnak a Swagger megoldását választottam, számos eszköz áll rendelkezésünkre, melyekkel API leírókat lehet készíteni, pl. kódgenerátor amiből szerveroldali és kliensoldali kódot egyaránt lehet generálni, webes kódszerkesztő amellyel böngészőből tudunk leírókat készíteni kézzel, amik rögtön meg is jelennek vizuális formában és ki is próbálhatjuk őket. Ilyen leírókat szerver oldali forráskódból generálni a legegyszerűbb, de ha szükséges akkor kézzel is el lehet készíteni ezeket. Sajnos Vapor keretrendszerrel készült webalkalmazások kódjából ilyen leírót még nem lehet generálni Swaggerrel, ezért Swagger Editorral állítottam össze a leírót kézzel. Szerencsére az Editorban már volt kliensoldali kódgenerátor, ezért a kliensoldali részeket nem kellett kézzel elkészíteni.

## Problémák az összekötés során

A leíróból generált kódnak két fő része volt, az API végpontok és a modellosztályok. A modellosztályok esetében már az okozott egy nehézséget, hogy a kliensoldali és a szerveroldali modellek nem voltak teljesen azonosak. Voltak típusbeli különbséget, de pár helyen mezők tekintetében is eltértek. Ezt a Swagger-es leíró még bonyolította egy fokkal, mivel a leíróban található adattípusok nem teljesen fedik le a Swift nyelvben is elérhető típusokat, ezért gyakran más típust kellett használni, amely egy harmadik verziójú modellosztályt eredményezett a generálás során. A problémák feloldására konverziós osztályokat vezettem be, amelyekkel át lehetett váltani a kliens- és szerveroldali modelleket.

A Swagger másik problémája volt, hogy a Swift 5-ös funkcióval generált kód az API osztályok megvalósításához Alamofire 4.9-es verziót használt. Ez nem olyan régi verzió a könyvtárból,

viszont vannak már újabbak, sőt szignifikáns változások vannak a 4.9-es és a legfrissebb (5.6-os) verzió között. Érdemes odafigyelni és utánajárni, hogy a generált kód mégis milyen verziójú könyvtárakat használ, mert én hajlamos voltam a szükséges könyvtárakból a legfrissebbet hozzáadni a projekthez, ami egy fordítás után több száz fordítási hibát eredményezett.