Couchsurfing alkalmazás fejlesztése iOS platformon

Önálló laboratórium 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.

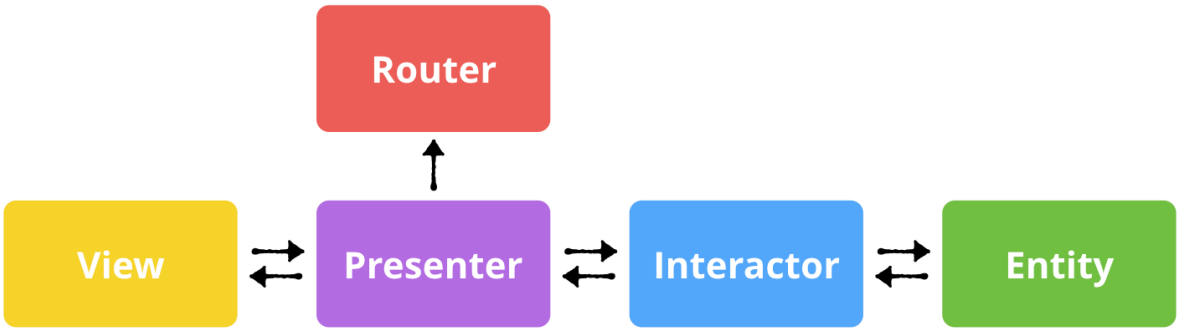
SwiftUI

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 (még csak) két különböző nézet (tab) mindig meg van nyitva az alkalmazásban és ezek között a képernyő alján lévő navigációs sávval lehet navigálni. A két 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

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 egy 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], ami egy olyan Geocoder volt, ami egyszerre használja a Google és az Apple ilyen

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

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.

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 a két adatbázis „táblához” 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 három Service a CouchService, UserService és ImageManager osztályok lettek. A CouchService jelenleg új szálláshely felvételére, frissítésére, a kurrens felhasználóhoz tartozó szálláshelyek és az összes szálláshely lekérdezésére képes. A UserService új felhasználó

felvételére és annak lekérdezésére képes most, az ImageManager pedig csak képek feltöltésére alkalmas. Külön képek letöltését végző műveletet ide nem implementáltam, mert találtam egy olyan SwiftUI ImageView-t[3], ami képes URL-lel hivatkozott képek letöltésére és megjelenítésére is.

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