Általános információk

A diplomaterv szerkezete:

1. Diplomaterv feladatkiírás
2. Címoldal
3. Tartalomjegyzék
4. A diplomatervező nyilatkozata az önálló munkáról és az elektronikus adatok kezeléséről
5. Tartalmi összefoglaló magyarul és angolul
6. Bevezetés: a feladat értelmezése, a tervezés célja, a feladat indokoltsága, a diplomaterv felépítésének rövid összefoglalása
7. A feladatkiírás pontosítása és részletes elemzése
8. Előzmények (irodalomkutatás, hasonló alkotások), az ezekből levonható következtetések
9. A tervezés részletes leírása, a döntési lehetőségek értékelése és a választott megoldások indoklása
10. A megtervezett műszaki alkotás értékelése, kritikai elemzése, továbbfejlesztési lehetőségek
11. Esetleges köszönetnyilvánítások
12. Részletesés pontos irodalomjegyzék
13. Függelék(ek)

Felhasználható a következő oldaltól kezdődő Diplomaterv sablon dokumentum tartalma. Ügyeljen a tanszék, a hallgató, a konzulens nevét és a beadás évét jelölő szövegdobozokra, mert azokra külön ki kell adni a frissítést. A mezők tartalma a sablonban a dokumentum adatlapja alapján automatikusan kerül kitöltésre (Fájl/Információ/Tulajdonságok/Speciális tulajdonságok).

A diplomaterv szabványos méretű A4-es lapokra kerüljön. Az oldalak tükörmargóval készüljenek (mindenhol 2.5cm, baloldalon 1cm-es kötéssel). Az alapértelmezett betűkészlet a 12 pontos Times New Roman, másfeles sorközzel.

Minden oldalon - az első négy szerkezeti elem kivételével - szerepelnie kell az oldalszámnak.

A fejezeteket decimális beosztással kell ellátni. Az ábrákat a megfelelő helyre be kell illeszteni, fejezetenként decimális számmal és kifejező címmel kell ellátni. A fejezeteket decimális aláosztással számozzuk, maximálisan 3 aláosztás mélységben (pl. 2.3.4.1.). Az ábrákat, táblázatokat és képleteket célszerű fejezetenként külön számozni (pl. 2.4. ábra, 4.2 táblázat vagy képletnél (3.2)). A fejezetcímeket igazítsuk balra, a normál szövegnél viszont használjunk sorkiegyenlítést. Az ábrákat, táblázatokat és a hozzájuk tartozó címet igazítsuk középre. A cím a jelölt rész alatt helyezkedjen el.

A képeket lehetőleg rajzoló programmal készítsék el, az egyenleteket egyenlet-szerkesztő segítségével írják le.

Az irodalomjegyzék szövegközi hivatkozása történhet a Harvard-rendszerben (a szerző és az évszám megadásával) vagy sorszámozva. A teljes lista névsor szerinti sorrendben a szöveg végén szerepeljen (sorszámozott irodalmi hivatkozások esetén hivatkozási sorrendben). A szakirodalmi források címeit azonban mindig az eredeti nyelven kell megadni, esetleg zárójelben a fordítással. A listában szereplő valamennyi publikációra hivatkozni kell a szövegben. Minden publikáció a szerzők után a következő adatok szerepelnek: folyóirat cikkeknél a pontos cím, a folyóirat címe, évfolyam, szám, oldalszám tól-ig. A folyóirat címeket csak akkor rövidítsük, ha azok nagyon közismertek vagy nagyon hosszúak. Internet hivatkozások megadásakor fontos, hogy az elérési út előtt megadjuk az oldal tulajdonosát és tartalmát (mivel a link egy idő után akár elérhetetlenné is válhat), valamint az elérés időpontját.

Fontos:

* a szakdolgozat készítő/diplomatervező nyilatkozata (a jelen sablonban szereplő szövegtartalommal) kötelező előírás Karunkon, ennek hiányában a szakdolgozat/diplomaterv nem bírálható és nem védhető!
* mind a dolgozat, mind a melléklet maximálisan 15 MB méretű lehet!

Jó munkát, sikeres szakdolgozat készítést ill. diplomatervezést kívánunk!

FELADATKIÍRÁS

A feladatkiírást a **tanszék saját előírása szerint** vagy a tanszéki adminisztrációban lehet átvenni, és a tanszéki pecséttel ellátott, a tanszékvezető által aláírt lapot kell belefűzni a leadott munkába, vagy a tanszékvezető által elektronikusan jóváhagyott feladatkiírást kell a Diplomaterv Portálról letölteni és a leadott munkába belefűzni (ezen oldal HELYETT, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell megismételni a feladatkiírást.



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Konzulens

BUDAPEST, 2022

Tartalomjegyzék

[Összefoglaló 6](#_Toc59896130)

[Abstract 7](#_Toc59896131)

[1 Bevezetés 8](#_Toc59896132)

[1.1 Formázási tudnivalók 8](#_Toc59896133)

[1.1.1 Címsorok 8](#_Toc59896134)

[1.1.2 Képek 8](#_Toc59896135)

[1.1.3 Kódrészletek 8](#_Toc59896136)

[1.1.4 Irodalomjegyzék 8](#_Toc59896137)

[2 Utolsó simítások 10](#_Toc59896138)

[Irodalomjegyzék 34](#_Toc59896139)

[Függelék 35](#_Toc59896140)

Hallgatói nyilatkozat

Alulírott **Rezeda Kázmér**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot/ diplomatervet (nem kívánt törlendő) 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. 11. 25.

...…………………………………………….

Összefoglaló

Ide jön a ½-1 oldalas magyar nyelvű összefoglaló, melynek szövege a Diplomaterv Portálra külön is feltöltésre kerül.

Abstract

Ide jön a ½-1 oldalas angol nyelvű összefoglaló, amelynek szövege a Diplomaterv Portálra külön is feltöltésre kerül.

# Bevezetés

A következő fejezet pár példán keresztül bemutatja a diplomatervekben és szakdolgozatokban szokásosan előkerülő formázások megvalósítását.

## Formázási tudnivalók

A dokumentum folyószövegéhez használjuk a **Normál** (angol Word esetén Normal) stílust.

### Címsorok

A fejezetcímek esetén a **Címsor 1-4** (Heading 1-4) stílusokat használjuk.

### Képek

A képhez használjuk a **Kép** stílust.

Képaláírást a képen jobb gombbal kattintva a Képaláírás beszúrása… opcióval adhatunk hozzá, így az automatikusan **Képaláírás** (Caption) stílusú lesz.



.. ábra: Példa képaláírásra

### Kódrészletek

Kódrészletek beillesztése esetén használjuk a **Kód** stílust.

using System;

namespace MyApp

{

class Program

{

static void Main( string[] args )

{

Console.WriteLine( "Szia Világ!" );

}

}

}

### Irodalomjegyzék

Az Irodalomjegyzékben szereplő hivatkozásokat **Irodalomjegyzék sor** stílussal formázzuk, a címüket pedig **Irodalomjegyzék forrás** stílussal emeljük ki.

A szövegbe a hivatkozásokat a Kereszthivatkozás beszúrása (Insert cross-reference) funkcióval helyezzük el (példa egy így beszúrt hivatkozásra: [1]), így azok automatikusan frissülnek a hivatkozások átrendezésekor.

# Utolsó simítások

Miután elkészültünk a dokumentációval, ne felejtsük el a következő lépéseket:

* Kereszthivatkozások frissítése: miután kijelöltük a teljes szöveget (Ctrl+A), nyomjuk meg az F9 billentyűt, és a Word frissíti az összes kereszthivatkozást. Ilyenkor ellenőrizzük, hogy nem jelent-e meg valahol a "Hiba! A könyvjelző nem létezik." szöveg.
* Dokumentum tulajdonságok megadása: a dokumentumhoz tartozó meta adatok kitöltése (szerző, cím, kulcsszavak stb.). Erre való a Dokumentum tulajdonságai panel, mely a Fájl / Információ / Tulajdonságok / Dokumentumpanel megjelenítése úton érhető el.
* Kinézet ellenőrzése PDF-ben: a legjobb teszt a végén, ha PDF-et készítünk a dokumentumból, és azt leellenőrizzük.

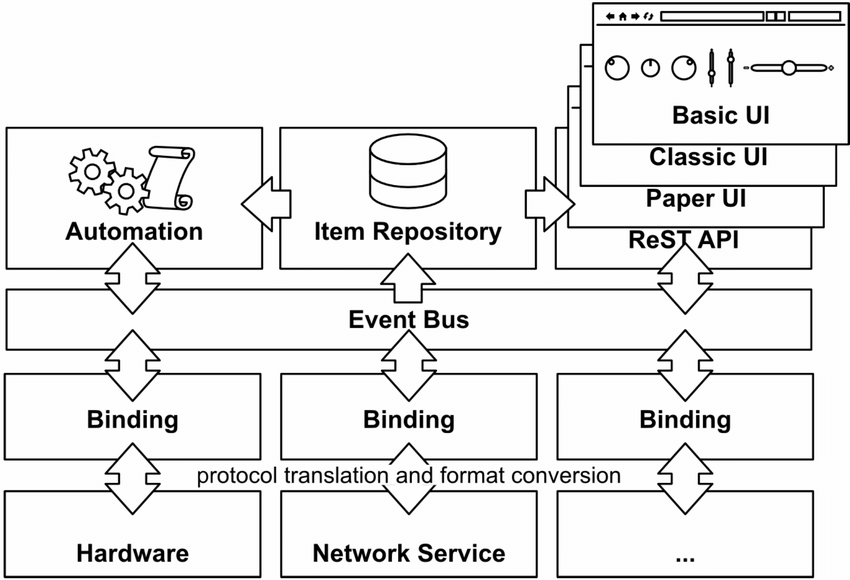
# Felhasznált technológiák áttekintése

## openHAB

### 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.

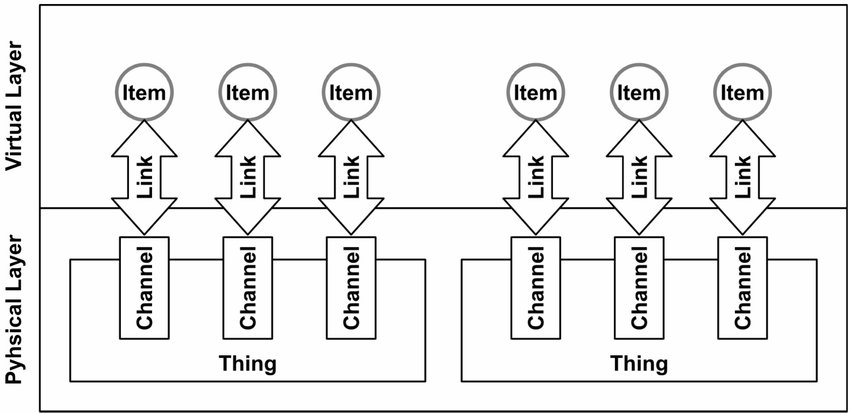


.. ábra Az openHAB rendszer architektúrája

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.

### 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.



.. ábra A virtuális és fizikai réteg

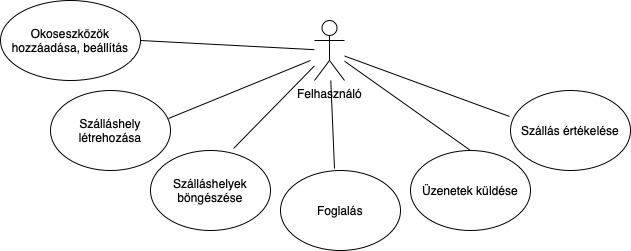
A platform fizikai rétegében tehát a Thing-ek és a Channel-ek helyezkednek el, a virtuális réteget pedig az Item-ek alkotják. Az Item-ek képviselik a fizikai eszközöknek egy kis részét, amit a felhasználó éppen látni akar belőle. Ezek akkor lehetnek nagyon hasznosak, ha több olyan eszközünk van, aminek 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.

# Tervezés

## 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.

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.



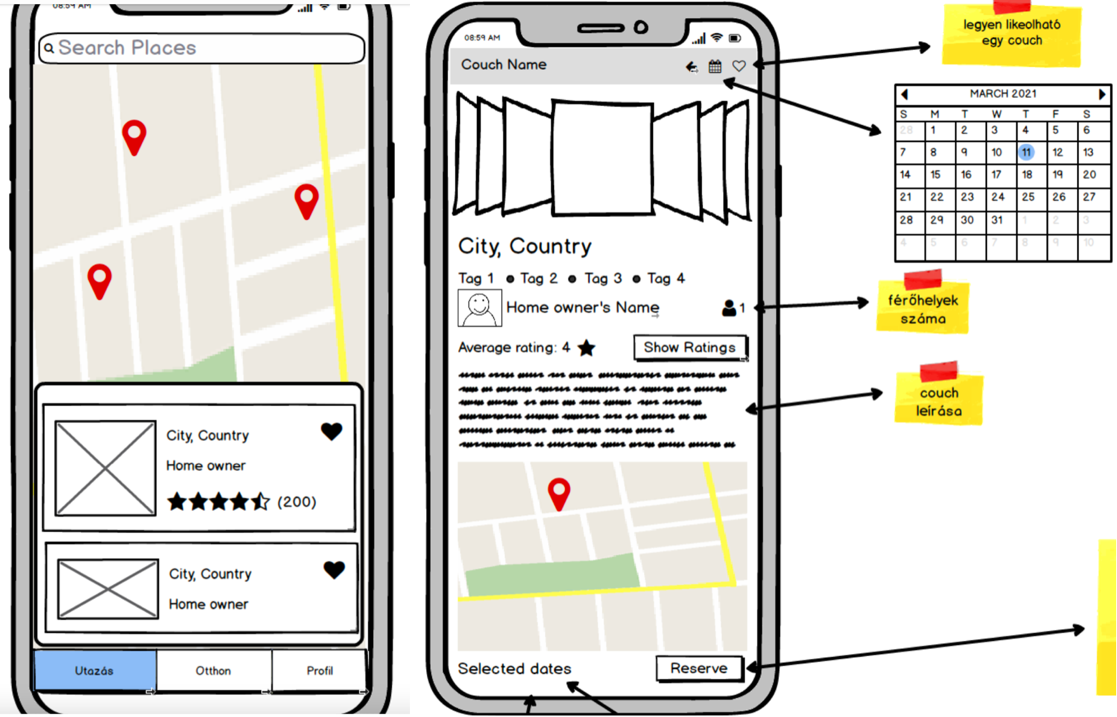
4.. ábra Alkalmazás use-case diagramja

## 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. [1]

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.

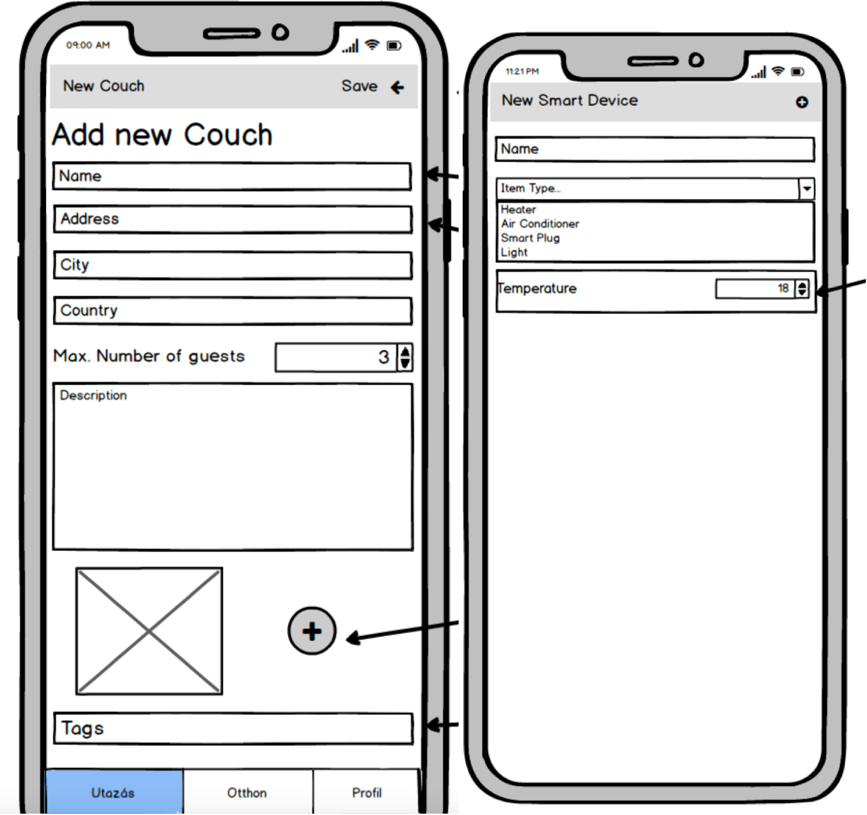
A drótvázak elkészítéséhez a Balsamiq Mockups nevű szoftvert használtam. Ez a szoftver egyszerű felhasználói felületével kiváló ilyen wireframe-k készítésére. A szoftver rendelkezik egy alap ábrakészlettel, ami az alapvető vezérlőket tartalmazza, de beépített könyvtárával tudunk további vezérlőábrákat 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.



.. ábra Drótvázak

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.

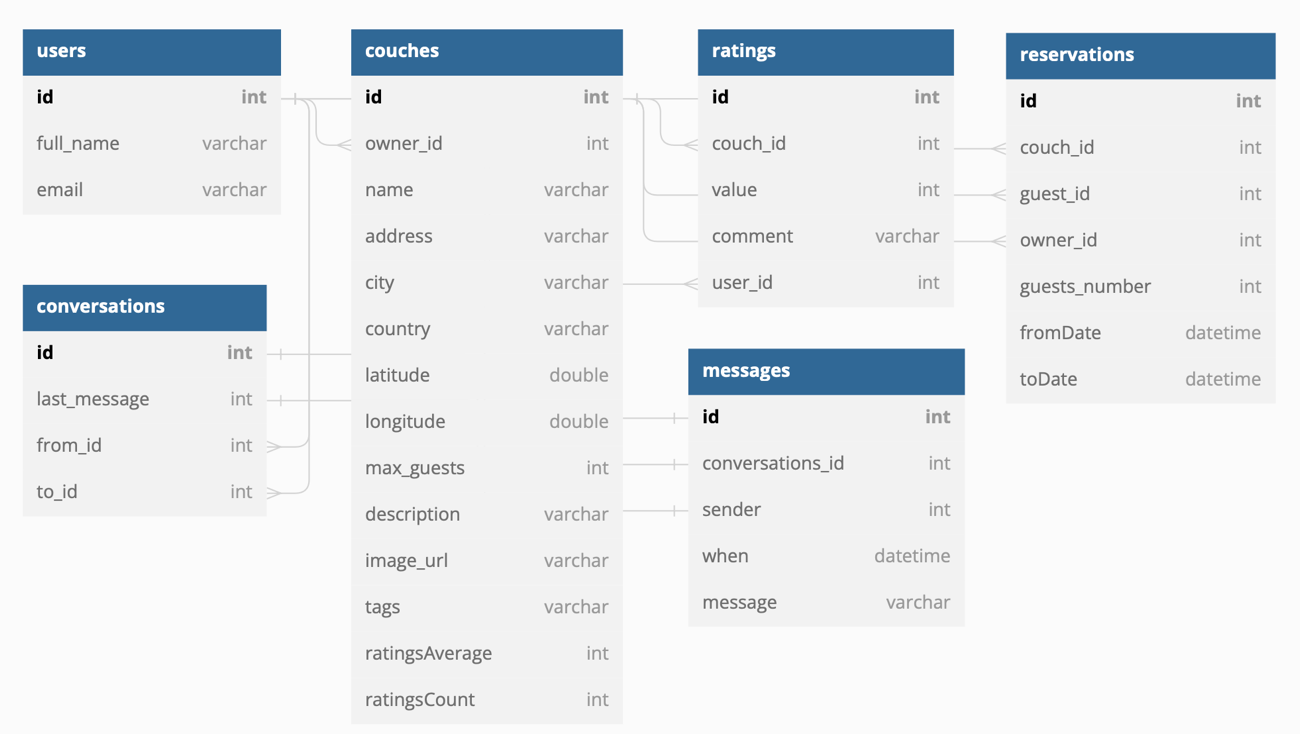
A későbbiekben az újonnan kitalált funkciók tervezésekor csak akkor készítettem drótvázakat, ha nem volt számomra sem egyértelmű, hogy pontosan hogyan jelenítsek meg bizonyos információkat a felhasználó számára.



.. ábra Drótvázak

## 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. Ez 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.

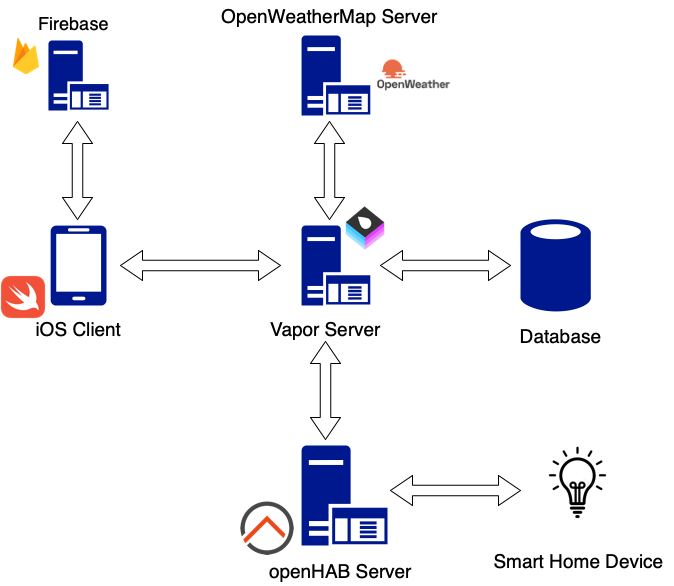


.. ábra Adatmodell

Az adatmodellt a dbdiagram.io[[1]](#footnote-1) 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 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.

## 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.



.. ábra A rendszer magas szintű architektúrája

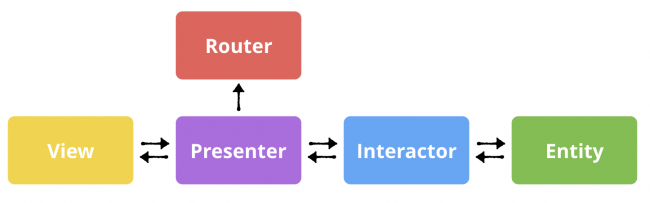
A kliens egy iOS platformon futó, natív mobilalkalmazás. Eleinte a rendszer architektúrája sokkal kevesebb elemmel rendelkezett, nem volt saját magam által készített szerver oldali komponens, ezért a mobilalkalmazás a Firebase szolgáltatásait vette igénybe. Az adattárolás, fájlok tárolása, illetve a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

## 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.

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 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. [2]



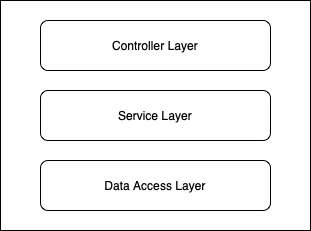
4.6. ábra Az architektúra komponenseinek kapcsolata

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 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.

## 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.



.. 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.

## 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[[2]](#footnote-2) 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. |

.1. táblázat A UserController végpontjai

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. |

.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. |

.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. 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. |

.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 |  |

.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. 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.

## 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 megoldásokat, 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 paramétert, 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 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.

## 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. |

.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 legfontosabbak 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. |

.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.

# Megvalósítás

## 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.

## 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 melyikre 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.

## 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.

### 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.

### 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.

A képen szöveg, elektronika, képernyőkép látható

Automatikusan generált leírás

.. 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.

### 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. [3]

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. [3]

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. [3]

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. [3]

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.

### Zárak kezelése

// rövid bemutatás

// hogyan kellett bővíteni a szerver oldalt

// UI vonzat

// biometrikus azonosítás implementációja

## Unit tesztek

### 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”. [4]

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.

### 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[[3]](#footnote-3), 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 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.

## 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[[4]](#footnote-4), a másik pedig a SwiftyMocky[[5]](#footnote-5) 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. A kódgenerálást egy 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 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.

## Kliens és szerver összekötése

Irodalomjegyzék

1. Balsamiq, „What Are Wireframes?”, [Online]. Available: <https://balsamiq.com/learn/articles/what-are-wireframes/>. [Hozzáférés dátuma: 13 november 2022].
2. 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].
3. Vapor Docs, „Vapor: Advanced -> Queues”, [Online]. Available: <https://docs.vapor.codes/advanced/queues/>. [Hozzáférés dátuma: 30 november 2022].
4. testdrive.io, „What is Test-Driven Development?”, [Online]. Available: <https://testdriven.io/test-driven-development/>. [Hozzáférés dátuma: 26 november 2022].

Függelék

1. <https://dbdiagram.io/home> [↑](#footnote-ref-1)
2. CRUD = CREATE-READ-UPDATE-DELETE [↑](#footnote-ref-2)
3. <https://www.kodeco.com/16909142-testing-in-vapor-4> [↑](#footnote-ref-3)
4. [https://mockingbirdswift.com](https://mockingbirdswift.com/) [↑](#footnote-ref-4)
5. <https://rawcdn.githack.com/MakeAWishFoundation/SwiftyMocky/4.1.0/docs/overview.html> [↑](#footnote-ref-5)