Matematikai és Informatikai Intézet

Okos otthon hub és irányítóközpont

Készítette
Lovász Ákos
Programtervező informatikus BSc

Témavezető
Dr. Tajti Tibor
Egyetemi adjunktus

Eger, 2021

Tartalomjegyzék

Bevezetés

Tanulmányaim folyamán számos technológiával ismerkedtem meg, melyek mindegyike rengeteg lehetőséget tárt fel előttem, viszont a szakmai gyakorlatom során kiemelkedően megragadta a fantáziámat az Andoid fejlesztés és a hardverprogramozás összekapcsolása által kialakult rendszerek lehetősége.

Az Android alkalmazások fejlesztése iránt mindig is érdeklődtem, egy-egy kisebb alkalmazást gyakorlásként már készítettem ezt megelőzően, de komolyabban itt kezdtem vele foglalkozni, megismerkedni a vele járó sajátosságokkal.

Az ilyen jellegű eszközök kapcsolata és kommunikációja már korai gondolataimban is az okos otthonok felépítésére emlékeztetett, ezért is gondoltam megfelelő téma választásnak.

A döntést követő kutatás során szembetűnő hátránya volt az okos otthon rendszereknek, hogy a legtöbb „márkás" megoldás elsősorban drága és csak felületes hozzáférést tesznek lehetővé, melyet teljes mértékben a rendszer gyártója határoz meg.

Az alternatív, olcsóbb rendszerek bár nyíltabb hozzáállással próbálnak előnyt szerezni, sokszor erősen a technikai oldalába mélyednek, inkább fejlesztők otthoni hobbyprojektjeként jelentek meg, így egy átlagos felhasználónak bonyolultnak, nehezen kezelhetőnek tűnhetnek. Ezen felül gyakran futhatunk olyan problémába, hogy az általunk választott rendszerben lévő hiányosságokat csak más gyártótól származó eszköz nyújtana megoldást, viszont különböző gyártók eszközei nagyon ritkán kompatibilisek egymással.

Ezeket az észrevételeket figyelembe véve egyértelműnek tűnt, hogy van lehetőség egy olyan rendszer kivitelezésére, ami elsősorban olcsóbb, de ugyanakkor nem túlbonyolított, felhasználóbarát marad. Fontos a nyitottság, a bővíthetőség, és a széleskörű kompatibilitás lehetősége, hogy a felhasználó biztos lehessen abban, hogy a jövőben felmerülő hiányosságok egyszerűen pótolhatók.

1. fejezet

A rendszer alapjai

A rendszer két fő komponensből áll. A kiszolgáló, mely egy Orange Pi Zero egykártyás számítógép, amin fut a Node-RED, egy olyan webes felületet biztosító szolgáltatás, mely grafikusan kezelhető komponensek összekapcsolásával teszi lehetővé a renszer működését befolyásolni, és a Mosquitto MQTT bróker, ami lehetővé teszi a Node-RED[1] és az Androidos alkamazás közötti kommunikációt.

1.1. A kiszolgáló hardver

Az eszköz egy nyílt forráskódu egykártyás számítógép, amin Armbian[9] (ARM processzor architektúrára specializált Debian) operációs renszer fut, de lehetőséget nyújt Ubuntu, vagy akár Android operációs rendszer telepítésére is. Ezen fut a Node-RED felület és a Mosquitto MQTT bróker, melyeket a helyi hálózaton bármely eszköz el tud érni, csupán az eszköz IP címét kell ismernie.

1.2. Az Android alkalmazás

Az Androidos alkalmazás célja a felhasználónak hozzáférést nyújtani az összes elérhető eszközhöz, azok állapotát megjeleníteni és felületet biztosítani azok irányítására, állapotuk megváltoztatására.

A kiszolgálóval való kommunikációt az Eclipse nyílt forráskódú Paho[6] Androidos kliens oldali MQTT implementációját használatba véve valósítja meg az alkalmazás.

A kommunikáció két irányú, azaz nem csak az alkalmazás tudja az okos eszközöket irányítani, hanem fogad üzeneteket a kiszolgálótól, így tud naprakész információt prezentálni az eszközök állapotáról a felhasználó számára.

A felület rugalmasságából adódóan az alkamazás nem kizárólag okos otthon kezelésére alkalmas, bármilyen MQTT protokoll alapú rendszeren való kommunikációra képes, viszont miven a fejlesztés során az okos otthonok kezelése volt az elsődleges

1.1. ábra. Orange Pi Zero[3], a Node-RED-hez és az MQTT brókerhez használt eszköz

szempont, így erre a célra használva a legoptimálisabb a felhasználói élmény.

1.3. Node-RED

A Node-RED egy nyílt forráskódú, „flow” alapú programozási eszköz az IBM Emerging Technologies[12] által fejlesztve az OpenJS Foundation[13] részeként. Ez egy Node.js alapú fejlesztési eszköz, aminek a felületét egy böngészőn keresztül lehet elérni ahol „node”-okat elhelyezve a felületen egy funkcióhálózatot létrehozva lehet úgymond programozni. Ez a funkcióhálózat egy „Deploy” gomb hatására bekerül a futási környezetbe, így effektíve az eddigi viselkedést felülírva, változtatásainkat elmentve.

1.4. MQTT

Az MQTT (Message Queueing Telemetry Transport)[4] egy OASIS szabványú kommunikációs protokoll, melyet IoT (Internet of Things) eszközök kommunikációjához fejlesztettek ki. A protokoll alapja a „Publish/Subscribe” alapú kommunikáció, azaz egy eszköznek lehetősége van egy adott „topic”-ra üzenetet továbbítani, vagy feliratkozni, azaz az adott „topic”-on beérkező üzeneteket megkapni. Ezek az üzenetek egy brókeren keresztül érik el céljukat, mivel a bróker tárolja hogy mely eszkör milyen témára iratkozott fel, ez alapján tudja a megfelelő klienseknek továbbítani a megfelelő üzenetet. Mivel az MQTT IoT eszközök kommunikációjához készült, így fejlesztése alatt különös figyelmet fordítottak az erőforrások megspórolásához, ezért ez a protokoll nagyon kevés erőforrást vesz igénybe, szinte bármilyen eszköz használatba tudja venni.

1.5. Okos eszközök

Okos eszköznek számít bármilyen berendezés, mely rendelkezik hálózati kommunikációra képes alkatrészekkel, ebben az esetben egy MQTT protokollt használatba vevő eszköz. Az MQTT protokoll rugalmasságából adódóan már a renszer tesztelése során is több eszköztípust vettem használatba, pédául Androidos tableteket, telefonokat és ESP D1 Mini mikrokontrollereket.

2. fejezet

Hardver

2.1. Orange Pi Zero

Egy egykártyás számítógép az Orange Pi felhozatalából a Zero model, ami beépített WiFi modullal, Ethernet porttal, 512MB RAM-al és egy 4 magos ARM processzorral ellátva tesztjeim alapján egy kisebb ház okos eszközeinek ellátására elegendő, viszont természetesen van lehetőség erősebb hardveren futtatni a kiszolgáló szolgáltatásokat. Ezek a szolgáltatások Linux és Windows operációs rendszert futtató hardverek bármelyikén képesek futni, így lehetőségünk van akár régi, már nem használt androidos telefonon, vagy ellentétsen, csak erre a célra kitűzött szervergépet kialakítani. Természetesen a végletek között rengeteg lehetőségünk van igényeink szerint választani kiszolgáló hardvert, például egy Orange Pi Zero, Raspberry Pi 4, vagy Seeed Odyssey. Az Orange Pi Zero-ra esett a választásom alacsony ára mellett nyújtott lehetőségei tárháza miatt.

2.2. Okos eszközök

2.2.1. ESP 8266

Okos eszköz fejlesztéséra egy rendkívül olcsó lehetőség az ESP D1 Mini mikrokontroller, ami Arduino nyelven programozható, WiFi moduljának és a hivatalos Arduino MQTT könyvtárnak köszönhetően viszonylag egyszerűen okos otthon eszközzé lehet alakítani.

2.2.2. Androidos eszköz

Mivel az Androidos alkamazások bizonyos könyvtárai és fejlesztői eszközei az alkalmazott Android verziótól függnek, így az alkalmazás fejlesztése során kitűzött minimum Android verziót el kell érnie a használni kívánt eszköznek. Ebben az esetben a minimum támogatott Android verzió a Marshmallow, azaz Android 6.0. Ez a legrégebbi Android

megjelenés amit bevett szokásként támogatnak a modern alkamazások, ennél korábbi verziót támogatni nehézkes, általában nem éri meg, mivel a felhasználók kevesebb mint 1%-a használ olyan eszközt, ami 6.0-nál régebbi Androidot futtat.

2.2.3. Egyéb eszközök

Minden olyan okos eszköz integrálható a rendszerbe, amely képes MQTT protokollon keresztül kommunikálni egy helyi hálózaton.

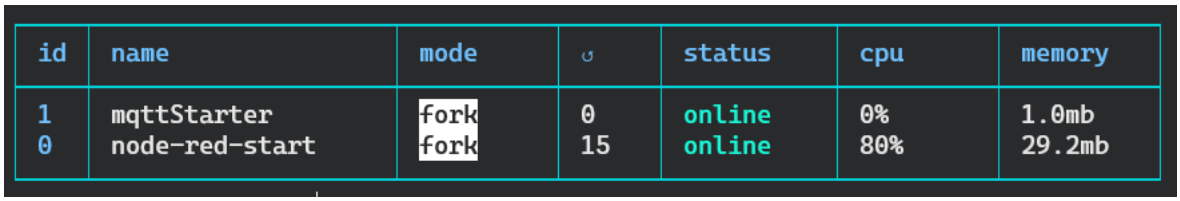
3. fejezet

Szoftver

A teljes rendszer olyan módon van felépítve, hogy a szerver oldali programoktól elvárt, hogy állandó futás mellett lehetőséget nyújtsanak bővítésre, konfigurációra és változtatásokra.

3.1. PM2

Az állandó futást a PM2[14] nevű folyamatvezető program biztosítja, akár egy váratlan újraindítás vagy áramkimaradást követően az operációs rendszer indulásával ezek a programok is indulnak, megfelelő konfigurációval.



id	name	mode	↺	status	cpu	memory
1	mqttStarter	fork	0	online	0%	1.0mb
0	node-red-start	fork	15	online	80%	29.2mb

3.1. ábra. PM2[14] a kiszolgálók automatikus indítására használt eszköz

Ez a terminálban futtatható Node.JS eszköz egyszerűen telepíthető NPM vagy Yarn csomagkezelőkkel, és azonnal használatba vehető. Lehetőséget nyújt nem csak Node.JS alkalmazások futtatására, de egyéb futtatható fájlok kezelésére is, például shell scriptek futtatására.

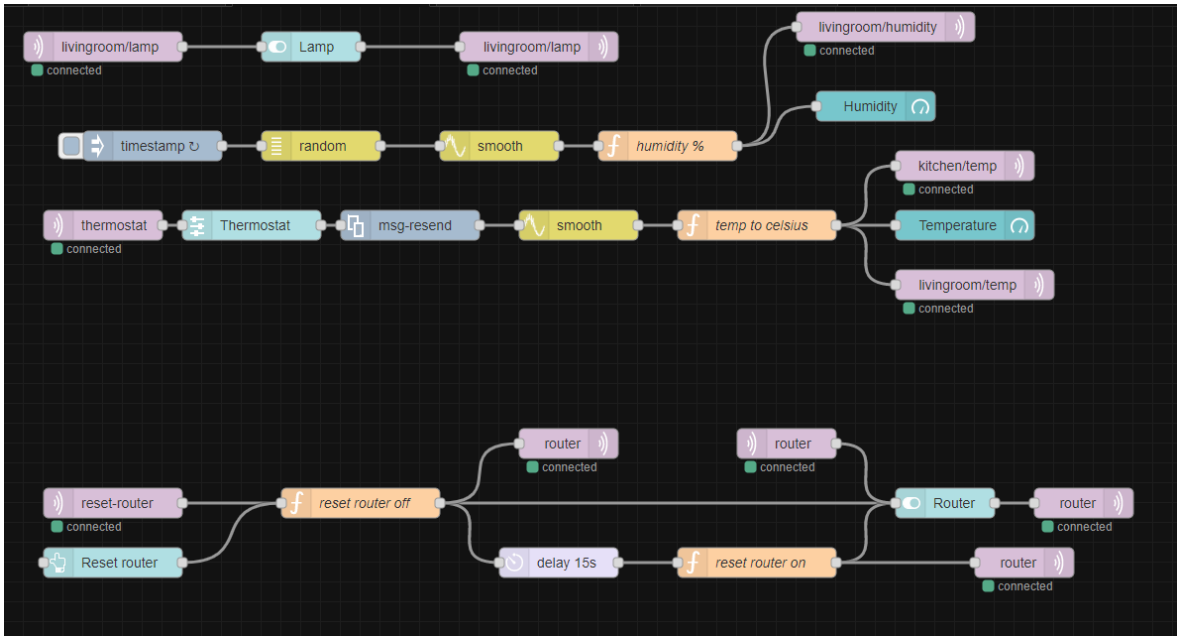
3.2. Node-RED

3.2.1. Flow

A Node-RED rendszerben csomópontok elhelyezésével és azok összekötésével lehet egy folyamatábrára hasonlító szerkezetet kiépíteni, ami a program viselkedését befolyásolja. Minden csomópont egy funkciót lát el, ehez mérten van a csomópontnak egy vagy több

bemeneti és/vagy egy vagy több kimeneti pontja, amin keresztül kommunikál a többi csomóponttal.

Egy „flow” a Node-RED felületén a benne lévő csomópontok és azok kapcsolatát foglalja magában. Minden flow reprezentálhat egy házban egy-egy szobát, nagyobb rendszereknél pédául egy-egy épületet, vagy több emeletes épületekben egy-egy emeletet. Ez felhasználható rendszerezési, kategorizálási vagy szerepköri felosztásra is, teljes mértékben a felhasználótól függ.



3.2. ábra. Minta ház nappali flow

3.3. ábra. Egy minta ház kiépítésében a nappaliban található eszközök irányítása

Az MQTT csomópontok belső konfigurációja csupán két dolgot vár el felhelyezése során: az MQTT topic, amire az adott csomópont feliratkozik, és az MQTT bróker címe. A bróker címét minden MQTT csomópont megosztja, így a kiszolgáló címének változásakor elég egy helyen megváltoztatni a címet, az összes csomópont megkapja az új címet és újra tud csatlakozni.

Implementáltam minden szobába egy okos lámpát, hőmérőt és páratartalom mérőt. Ezen felül a nappaliban és a hálószobában vagy egy-egy termosztát, melyeket függetlenül irányíthatunk. Mivel ez a minta ház csak szimuláció, így a szemléltetés érdekében a hőmérsékletváltozást egy időintervallum alatt fokozatosan változtatom a termosztát állítást követően, így reprezentálva a való világban a ház fokozatos felfűtését/lehűtését. Szintén szemléltetési céllal a páratartalom mérők véletlenszerű értékeket vesznek fel, mivel egy bemutató példa házban nem szükséges a valóságnak megfelelő értékeket szimulálni, a rendszer szemléltetése a lényeg.

Az okos lámpa egy egyszerű kapcsolóként jelenik meg a bemutató házban, ez természetesen egy okos kapcsolóval ellátott lámpát reprezentál, melyet irányíthatunk az

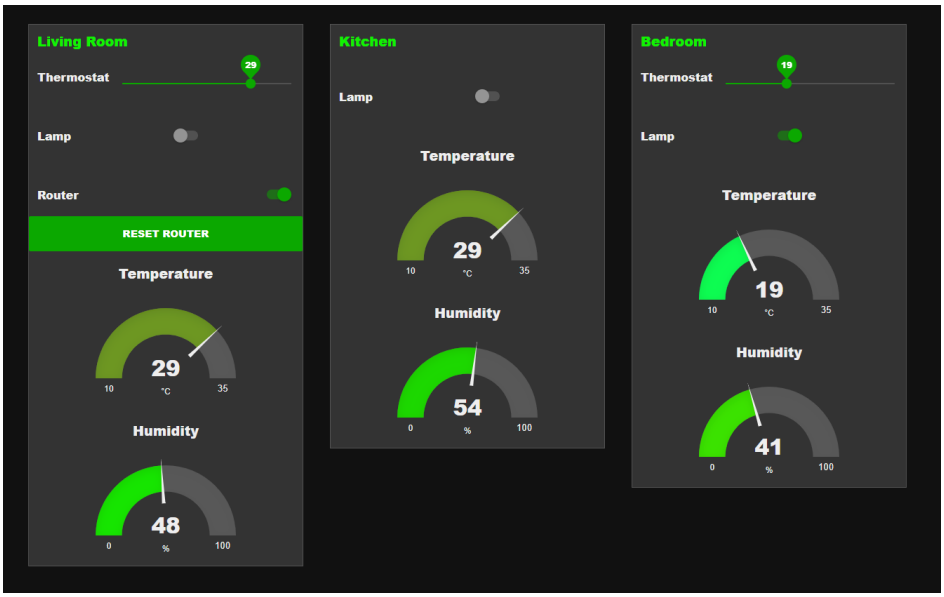
Androidos alkalmazásból, vagy a webes felületről.

Ezen felül felhelyeztem egy internet router újraindító gombot, ezzel adva példát egy egygombos funkió ellátására, mivel egyszerűbb egy újraindítás gombot megnyomni, mint kapcsolóként kezelni és a felhasználóra bízni, hogy ne kapcsolja vissza idő előtt a router-t, így megelőzve az esetleges késés által előidézett időkimaradás eltörlését, mivel az MQTT sorban küldi az üzeneteket, ha a rendszer lelassul egy pillanatra, lehet, hogy két üzenet, amit a felhasználó egymástól eltérő időpintban küldött el, mégis egyszerre érkezik a kiszolgálóhoz.

Ez a webes szerkesztőfelületet a helyi hálózaton a kiszolgáló eszköz IP címén, a konfigurációban megadott portszámon (alapbeállítás szerint 1880) lehet elértni. Az itt végzett változtatások csak akkor kerülnek a futási környezetbe, ha a „Deploy” feliratú gombot megnyomjuk, különben minden változtatás csak a felületen történik, átmenetileg van mentve, hogy közben a szolgáltatás az előző verzióval tovább tudjon futni, így állandó elérést és irányítást biztosítva a már felcsatlakozott okos eszközök számára.

3.2.2. Dashboard

A Node-RED lehetőséget nyújt egyéb csomópontok telepítésére, melyeket felhasználók vagy egyéb entitások fejlesztettek. Egy ilyen csomópont kollekció a Node-RED Dashboard[2], mely által lehet készíteni egy webes felületet, amin keresztül lehet szemléltetni, irányítani a rendszer elemeit.



3.4. ábra. A Node-RED Dashboard[2], szobákra bontva, példa eszközökkel ellátva

Ez nem olyan rugalmas, mint az Android alkalmazás, mivel minden elemet kézzel kell felvenni a felületre, majd minden konfigurációját és interakcióját a flow szerkesztőben kell kézileg megadni, minden elemet külön fel kell programozni.

3.3. MQTT

3.3.1. Broker

MQTT brókernek az Eclipse Mosquitto[5]-t választottam, mivel egy jól ismert alapítvány által fejlesztett, teljesen nyílt forráskódú multi-platform projekt, melyben különös figyelmet fordítottak az erőforrásokkal való spórolásra, így egykártyás számítógépeken való alkamazását megkönnyítve.

Szintén előnyt jelentett választásom során a jó minőségű dokumentáció, mely alapján viszonylag zökkenőmentes volt a telepítés és a konfiguálás.

3.3.2. Konfigurálás

Mivel a konfigurációs fájlban meg kell adni a brókernek, hogy mely címen fog csatlakozási kérelmeket és üzeneteket kapni, készítettem egy olyan shell scriptet, mely minden indításnál egy új, naprakész konfigurációs fájlt készít a bróker indítása előtt.

```
echo -n "listener 1883 " > mqtt.conf; ip -4 addr show eth0 |
    ↪ grep -oP '(?<=inet\s)\d+(\.\d+){3}' >> mqtt.conf echo $"
    ↪ allow_anonymous true" >> mqtt.conf
```

Ez a shell script generál egy olyan konfigurációs fájlt, ami a következőket állítja be:

- 1883-as porton várja az üzeneteket
- Lekérdezi a kiszolgáló számítógép IPv4 címét, amit egy regex szűrőn keresztül formázom megfelelő módon és a portszám után illesztem be, előírás szerint.
- Engedélyezi az anoním kapcsolatokat. Mivel ez a rendszer egy helyi hálózatra van tervezve, nem jelent biztonsági kockázatot az autentikáció nélküli kapcsolat, ezt kiváltja az MQTT ID rendszer.

3.4. Android

3.4.1. Activity

Az Android „aktivitásokra” bontja a kódot, melyek a hozzájuk tartozó „töredékek” alappilléreként szolgál. Fő célja az aktivitásokra bontásnak az erőforrások megspórolása, minden aktivitás egy specifikus feladatkört lát el, miközben a másik aktivitások a háttérben leállnak, amíg nem lépnek újra használatba. Az én esetemben egy fő aktivitás látja el az alapvető feladatait az applikációmnak, mivel ennek a fő aktivitásnak a kiszolgálóval való kapcsolattartás és a felhasználói adatok tárolása, így több aktivitásra bontani ezt csak feleslegesen bonyolítana mind a program logikáján, mind az átláthatóságán.

Az Android rendszer alapjáraton 4 fő „szálat” különböztet meg:

Main Thread

A fő szál. Ez az alkalmazás indításakor az operációs rendszer által nyitott szál, mely felelős az események kezeléséért, felhasználói felület megrajzolásáért és egyéb elemek létrehozásáért.

Ui Thread

A felhasználói felületi szál, ami felelős a felhasználóval való kapcsolat fenntartásáért és a felületi eseménykezelésért mint például egy gomblenyomás. Fontos, hogy a UI szálat sosem szabad megakasztani hosszú folyamatokkal, például hálózati csatlakozás indításával, mivel ilyenkor a felület teljes egészében leáll, a felhasználó felé nem reszponzív, nem tud semmilyen eseményt kezelni amíg a folyamat ami megakasztotta a szálat be nem fejeződik.

Worker Thread

Az egyéb többszálas folyamatokat kezelő szál. Ez a szál nyújt megoldást a UI szál megakasztására, ezt kell használni hosszabb, nem azonnal ellátható események kezelésére. Felmerül használata során viszon az a probléma, hogy a felhasználói felületet csakis a UI szálról lehet frissíteni.

Binder Thread

A kötött szolgáltatások távolról meghívható metódusokat tartalmaznak. Ha a végrehajtott metódusra történő felhívás ugyanabban a folyamatban van, amelyben a kötött szolgáltatás fut, a metódus a hívószalagban hajtódik végre. Ha azonban a hívás egy másik folyamatból származik, akkor a metódus egy olyan szálban kerül végrehajtásra, amelyet a rendszer ugyanabban a folyamatban tart, mint az meghívó.

3.4.2. Fragmentek

A fragment az része az alkalmazás felhasználói felületének, ami saját felosztását definiálja és kezeli. Saját életciklussal rendelkezik, viszont egyedülálló elemként nem használhatóak, mindig kell lennie egy tulajdonosának, ami lehet egy másik fragment vagy activity. Ehez a tulajdonoshoz kötve jelenhet meg egy fragment felülete hozzá csatolva vagy részévé válva.

A fragmentek célja hogy egy activityn belül külön választott felhasználói felületet prezentálhassunk, egymástól független elemekre bontva, így megkönnyítve a felhasználói felület felépítésének procedúráját és az erőforrások megtakarítását.

A fragmentek ideális használati köre egy activityn belüli navigációs elemek által elérni kívánt felületek prezentációja. Mivel egy fragment életciklusa akkor ér véget, amikor a felhasználó egy másik fragmentre váltással felülírja azt, hosszútávú adattárolásra nem alkalmas, bár erre is vannak beépített megoldások, például a fragmentet tartalmazó activiy szintjén tárolni az adatokat, vagy a fragmentek belső „instance bundle" változójával kezelni, bár az utóbbi típusmegközései miatt igencsak korlátozó módszer, de több módszer kombinálása sem kizárt, így mindent olyan módon lehet kezelni, ahogyan az adott esetben optimális.

3.4.3. Felhasználói felület

Az alkalmazásom felhasználói felülete 3 fő fragmentből áll: Home, Settings és Help. A Home fragment a fő interakciós felülete az alkalmazásnak, itt tölti a felhasználó az ideje nagy részét, itt éri el és irányíthatja a hálózaton található okos eszközöket.

A Settings fragment a kiszolgáló címének megadására, felhasználói profil kiválasztására és a kapcsolat létesítésére szolgál.

A Help fragment felhasználási segítséget nyújt a felhasználó számára, ha nem biztos az alkalmazás működésének rendjében, ezen leírás alapján tud tájékozódni a rendeltetésszerű felhasználásról.

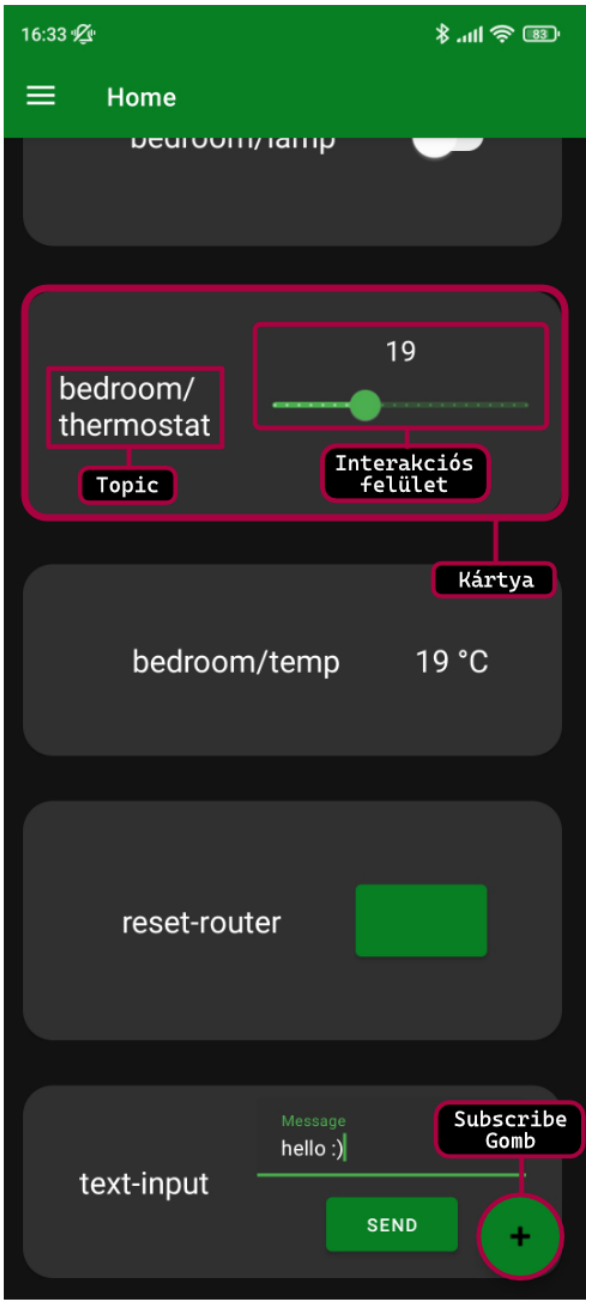
Az alkamazásban található minden felhasznált grafikai elem a Material Design[7] előírásait követve készültek, ami a Google által kifejlesztett design nyelv. Eredetileg a Google által használt kártya alapú design továbbfejlesztése reszponzív animációk, átmenetek és mélységi effektek (megvilágítás, árnyékok) implemetálásával. Ezeket az előírásokat és kész elemeket felhasználva biztosítottam, hogy a felhasználói felület rögtön ismerős hatást keltsen, könnyen lehessen rajta eligazodni és hogy a jövőben kevésbé tűnjön elavultnak, régimódinak.

Home

A Home fragment szolgál az eszközök kezelésére, irányítására, adatok prezentálására. A felület jobb alsó sarkában található „Floating Action Button", vagy FAB megnyomására a felhasználó feliratkozgat egy MQTT topicra, majd kiválaszthatja a létrehozandó kártya interakciós típusát.

Ha sikeresen feliratkozik egy topicra, a felületen létrejön egy „kártya" a kiválasztott interakciós típussal, például egy csúszkával, gombbal vagy kapcsolóval. Ezt követően a felhasználó a kártyán keresztül tudja kezelni az adott topicot használó okos eszközöket.

Ha a felhasnáló el szeretné távolítani a kártyát a felületről, egy hosszú érintéssel a kártya bármely részére előidéz egy „Unsubscribe" kontextusi gombot, melyel törölheti a kártyát a felületről.



3.5. ábra. A Home fragment elrendezése és elemei

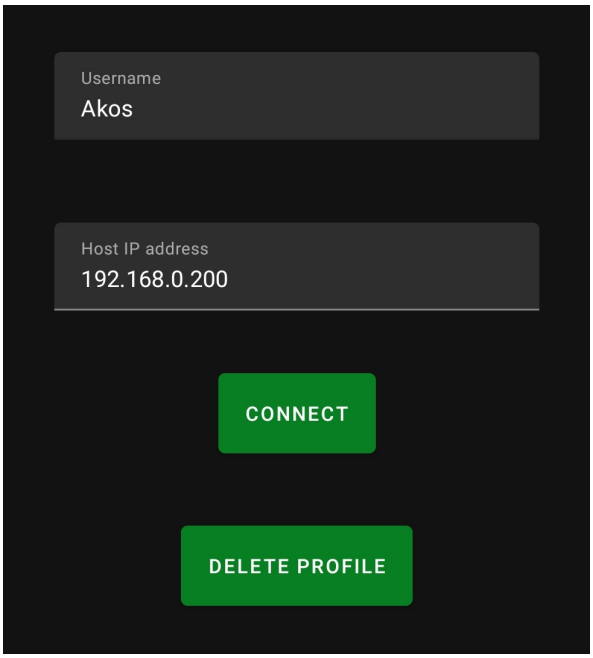
Settings

A Settings fragment a kiszzolgálóhoz való csatlakozásra és a felhasználói profil beállítására szolgál.

A felületen található két beviteli mező. Az elsőbe a felhasználói profil elnevezését, a másodikba a kiszolgáló címét kell megadni, majd a „Connect” gomb megnyomására az alkamazás kapcsolatot létesít a kiszolgálóval és ellenőrzi, hogy létezik-e már a megadott néven felhasználói profil. Ha már létezik, az alkamazáson belüli tárba betölti az elmentett adatokat, viszont ha nincs, létrehozza azt.

A „Delete Profile” gomb egy új ablakban, legördülő menüből kiválasztható profilok

listájából törli azt, amelyket a felhasználó kiválaszt.



3.6. ábra. A beállításokat tartalmazó fragment

Az itt megadott értékeket az alkalmazás eltárolja, így a következő indításkor ezek a mezők már az előző használatkor alkalmazott értékekkel kitöltve jelennek meg, így megkönnyítve felgyorsítva az alkalmazás használatát.

Help

A help fragment tartalmazza a felhasználói útmutatót. Itt lehet tájékozódni az alkalmazás rendeltetésszerű használatáról, a funkciók céljáról, használatáról és az esetleges problémák megelőzéséről, kijavításáról.

A help fragment szintén kártyákra osztja a felületet a könnyű és gyors átláthatóság érdekében és a konzisztencia fenntartásáért.

3.4.4. Kapcsolat

Az MQTT kapcsolatot Android oldalon szintén az Eclipse Foundation megoldását használom, a paho[6] nevű, nyílt forráskódú MQTT kliens Android implementációját, a Geradle rendszeren keresztül összeállított, Maven repositoryból leöltött forrás csomagját importálva.

A kapcsolat 60 másodperces időtúllépést elérve felbontódik, ha nem kap választ a kiszolgáló az életben tartó jelre. Ezért a kapcsolat létrehozása pillanatában a kliens entitását eltárolom egy activity szintű változóban, így a kapcsolat addig él, amíg a felhasználó használja az alkamazást, vagy nem zárja be több mint 1 percre.

```
MqttAndroidClient client = new MqttAndroidClient(this.
    ↪ getApplicationContext(),
"tcp://" + mqttAddress + ":1883", clientId);

client.connect(null, new IMqttActionListener() {
        @Override
        public void onSuccess(IMqttToken asyncActionToken) {
                mqttClient[0] = client;
        }

        @Override
        public void onFailure(IMqttToken asyncActionToken,
            ↪ Throwable exception) {
                Toast toast = Toast.makeText(
                    ↪ getApplicationContext(),
                                "Failed to connect to " +
                                    ↪ mqttAddress, Toast.
                                    ↪ LENGTH_SHORT);
                toast.show();
        }
});
```

Ez a kapcsolat aszinkron módon jön létre, így nem kell megállítani a felhasználót, amíg nem jön létre a kapcsolat, vagy hibát eredményez a csatlakozás, például hibás címet adott meg a felhasználó vagy a kiszolgáló nem fut.

3.4.5. Kártyák

A jelenlegi verzióban 6 típusú kártya használatára van lehetőség, de ez a jövőben egyszerűen bővíthető:

- Text: Szimpla szöveges feliratot prezentál, felhasználható pédául hőmérő értékének kijelzésére.
- Button: Egy gombot tartalmaz, ami az adott topic-ra egy "1"-es üzenetet küld, amit a kiszolgáló dolgoz fel.
- Checkbox: Egy jelölőnégyzet, mely "on" és "off" üzenetet küld a kiszolgálónak állapota szerint. Lehet használni sokoldalú állapotjelzőnek vagy kapcsolónak.
- Swithc: Egy kapcsoló, mely szintén "on" és "off" üzenetet küld a kiszolgálónak állapota szerint, így logikusan alkalmazható például okos lámpa kapcsolójaként.
- Input: Egy beviteli mezőt tartalmaz, amibe tetszőleges szöveget vagy számot írhat a felhasználó, majd az alatta lévő gomb érintésével elküldi a tartalmát az adott topic-ra. Használható például LED-es tábla feliratának módosítására.

– Slider: Egy csúszka, melynek minimum és maximum értékeit a felhasználó határozza meg, az ez által nyújtott sokoldalúságából eredendően lehet használni például termosztát beállítására, vagy okos lámpa fényerejének módosítására.

Egy kártya felépítése a korábban látott ábra szerint (3.5) két félre bontódik. Bal oldalán található a topic felirata, amelyre az adott kártya feliratkozott.

A jobb oldalán található a felhasználó által választott interakciós felület, melyen keresztül tud kommunikálni az alkalmazás a kiszolgálóval az adott topic-on.

Minden kártya egymástól teljesen független, így például több kártya ugyan arra a topicra képes feliratkozni, nem okoz problémát, minden kártya feldolgozza a rá vonatkozó, azaz a topicra érkező üzeneteket.

Egy törönni kívánt kártyát a hosszú érintésével előidézhető „Unsubscribe” gomb megnyomásával tudunk eltávolítani a felületről. Ez nem csak vizuálisan szűnteti meg az adott kártyát, de a topicról is leiratkozik, amit eddig használt, így azt a minimális erőforrást is felszabadítva amit elfoglalt.

Mielőtt egy kártya sikeres feliratkozás eredményeként létrejön, több adatfeldolgozási lépést kell megtenni. Vegyünk példának egy kapcsoló típusú kártyát:

```
private void createCard(LinearLayout layout, List<String>
    ↪ savedCardData, int type) {
     ViewGroup mqttCard = (ViewGroup) this.getLayoutInflater().
        ↪ inflate(type, null);
     TextView topicDisplay = (TextView) mqttCard.findViewById(R.
        ↪ id.text_topicDisplay);
     registerForContextMenu(mqttCard);
     topicDisplay.setText(savedCardData.get(0));
```

A kártya létrehozó metódusa megkapja paraméterként a Home fragment alapelrendezését, amiben majd el kell helyeznie az új kártyát, egy listát, ami tartalmazza a profilban tárolt kártyák adatát, így ha már volt a profilban mentett adat, azt visszaállítja a program, nem kell újból azokat megadni, és végül megkapja a kártya típusát, mely alapján kiválasztja a program, hogy milyen interakciós felületű kártyát hozzon létre.

Ezt követően inicializálja a kártya struktúra alapját, a topic kijelző szövegét, és elkezdi a kártyatípus ellenőrzését.

```
switch (type) {
        case R.layout.mqtt_card_switch:
                SwitchMaterial switch_data = (SwitchMaterial)
                    ↪ mqttCard.findViewById(R.id.switch_data);

                if (!savedCardData.get(2).equals("null")) {
                        switch_data.setChecked(savedCardData.get
                            ↪ (2).equals("on"));
```

```
                }
                switch_data.setOnClickListener(view -> {
                        String message = switch_data.isChecked()
                            ↪ ? "on" : "off";
                        publishMessage(((MainActivity)
                            ↪ getActivity()).getClient(),
                            ↪ savedCardData.get(0), message);
                });
                layout.addView(mqttCard);
                break;
```

Miután a kártyatípus eldöntésre került, inicializálja a kártya létrehozásához szükséges változókat, majd ellenőrzi, hogy volt-e már az adott topicon ilyen típusú kártya mentve. Ha volt mentett adat az adott kártyáról, azt állítja be kezdőértéknek (ebben az esetben hogy milyen állapotban volt utoljára a kapcsoló).

Ezt követően létrejön az interakciós felület „figyelője”, mely megadja, hogy az adott felület használata milyen módon reagáljon, ebben az esetben a kapcsoló állapotváltozása küld egy "on" vagy "off" üzenetet a kiszolgáló számára, állásától függően.

Végül a kártya felkerül a felhasználói felületre.

3.4.6. Kommunikáció

A kártyák és a kiszolgáló közötti kommunikációt egy „publishMessage” nevű metóduson keresztül továbbítom az alkalmazásból a bróker felé. Ennek a metódusnak szüksége van egy kliens példányra, melyet az activty tárol csatlakozást követően, a topicra, amire az üzenetet kívánjuk közzétenni és végül természetesen maga az üzenet, amit el kívánunk küldeni.

Az üzenetek minden kártyatípus interakciós pontjának saját kódja határozza meg az üzenet tartalmát, például a kapcsoló típusú kártyának "on" és "off" állapotával megegyező szöveges üzenetet küld.

```
switch_data.setOnClickListener(view -> {
                    String message = switch_data.isChecked() ? "
                        ↪ on" : "off";
                    publishMessage(((MainActivity) getActivity()
                        ↪ ).getClient(), savedCardData.get(0),
                        ↪ message);
                });,
```

Az üzenetek fogadását egy „messageRecievedHandler” nevű metódus kezeli, amit az MQTT kliens callback metódusa hív meg a messageArrived belső metódusából, átadva a beérkező üzenet topicját és tartalmát.

A beérkező üzenet topicját először ellenőrzi, hogy egyezik-e az aktuálisan vizsgált kártya topicjával, majd összehasonlítja a kártya típusát az elvárt típussal. Ha minden

egyezik, akkor a dekódolt üzenet tartalmát kártyatípusnak megfelelően jeleníti meg, vagy állítja be a kártyán található interakciós elem állapotát.

```
if (topicDisplay.equals(topic)) {
        if (activeElement instanceof SwitchMaterial) {
                SwitchMaterial switchView = cardList.getChildAt(
                    ↪ i).findViewById(R.id.switch_data);
                switchView.setChecked(decodeMQTT(message).equals
                    ↪ ("on"));
        }
}
```

Az MQTT üzenetek dekódolása egy egyszerű String-é alakítás, mivel a beérkezett üzenet egy saját típusú, MttMessage objektum amiben egyéb adatok mellett az üzenet egy byte tömbben tárolódik, amit kevésbé egyszerű és gyors miden feldolgozásnál kezelni, ezért egy String-et építek belőle.

```
private String decodeMQTT(MqttMessage msg) {
        return new String(msg.getPayload(), StandardCharsets.
            ↪ UTF_8);
}
```

A kapcsolat a kiszolgálóval addig él, amíg az alkalmazás tud válaszolni a kiszolgáló által közvetített jelre, ami azonosítja, hogy mely eszközök vannak aktív állapotban. Ha erre az üzenetre az alkalmazás nem tud válaszolni, azaz a felhasználó bezárta azt, 60 másodpercen belül lezáródik a kapcsolat. Ezt a kapcsolatbontást az alkalmazás közli a felhasználóval, majd felajánlja az újracsatlakozás lehetőségét. Ha a felhasználó nem él a lehetőséggel, a kapcsolatot újból a Settings fragmentről kell létesíteni.

3.4.7. Adattárolás, Profilok

A kártyaadatok profilonkénti tárolása egy szöveges fájba való kiírással oldottam meg, olyan formátumban, hogy a fájl neve a felhasználói profil megadott neve, a tartalma pedig minden kártya amit a felhasználó létrehozott, a következő formában felbontva: Topic:Kártytípus:Tárolt adat.

Minden kiírás esetén ellenőrzöm, hogy már szerepel-e a fájlban a jelenleg mentésre kijelölt kártya, így elkerülve felesleges kiíratási folyamatokat, ha már létezik, mivel ebben az esetben felesleges lenne kitörölni és újra kiírni.

Ez az ellenőrzés viszont csak a kártya topicra és típusra vonatkozik, mivel a tárolt adat folyamatosan változik. Az ellenőrzés során megkeresem az összes olyan elmentett kártyát, ami ugyanazt a topicot és kártyatípust tartalmazza, majd csak azt a sort írom felül a friss adatokkal, így nem kell a teljes fájlt újraírni csak egy sor módosításáért.

Friss profiloknál, mivel egy üres fájl jön létre, a kereső algoritmus hibát dobna, mivel nem tud üres sorokban keresni, ezért erre az esetre egy külön elágazást implementáltam,

ami szimplán kiírja a menteni kívánt kártya adatait, mivel ebben az esetben biztosak lehetünk abban, hogy még nem tartalmazza a fájl ezt a kártyát, ezért átugorhatjuk az ellenőrzést.

```
public void addCardDataToPersistentStorage(String topic, String
    ↪ cardType, String cardData) {
        boolean found = false;
        int i = 0;
        if (cardDataStore.size() == 0) {
                this.cardDataStore.add(topic + ":" + cardType +
                    ↪ ":" + cardData);
        }
        do {
                String[] part = this.cardDataStore.get(i).split(
                    ↪ ":", 0);
                if ((part[0] + ":" + part[1]).equals(topic + ":"
                    ↪ + cardType)) {
                        this.cardDataStore.set(i, topic + ":" +
                            ↪ cardType + ":" + cardData);
                        found = true;
                }
                i++;
        }
        while (!found && i < cardDataStore.size());

        if (!found) {
                this.cardDataStore.add(topic + ":" + cardType +
                    ↪ ":" + cardData);
        }

        writeToFile(this.username + ".txt", this.cardDataStore);
}
```

A kártyák adatain kívül az alkalmazás tárolja az utoljára használt felhasználónevet és a kiszolgáló címét. Ezt az Android sajátos „SharedPrefrence” API eszközeivel tárolom, ami kulcs-érték párokat tárol, melyeket csak az alkalmazáson belül lehet elérni. Mivel ez a rendszer kis mennyiségű adatok tárolására célzott eszköz, így a kártyák adatainak tárolására nem lenne alkalmas, viszont a felhasználónév és kiszolgáló címének tárolását beolvasztani a kártyaadat tárolására szolgáló fájlba rendezetlen és bonyolultnak tűnt, ezért választottam szét a két tárolási módot.

A profilok törlésére a Settings fragmenten található „Delete Profile” gomb megnyomásával van lehetőség, ami kilistázza az összes profil fájl címét egy legördülő menüben.

Egy létező profilnév megadása a csatlakozás során azt eredményezi, hogy a profil fájlból a program beolvassa a felhasználó nevén tárolt kártyaadatokat, amit elhelyez egy memóriában tárolt listában, majd azt továbbítja a Home fragment számára, ami

adatrészekre tördelve feldolgozza azt és létrehozza az összes kártyát, amit a felhasználó az utolsó használata során elmentett.

A memóriában tárolt kártyaadatok listájának minden eleme egy teljes kártyát tartalmaz, aminek feldolgozásához részelemekre kell tördelni, majd ezeket a részelemeket felhasználva építem fel a kártyákat. Vegyük példának a csúszka típusú kártyát:

```
List<String> sliderSubData = Arrays.asList(savedCardData.get(2).
    ↪ split("\\."));
TextView sliderDataDisplay = mqttCard.findViewById(R.id.
    ↪ slider_data_display);

String rangeMin = sliderSubData.get(0);
String rangeMax = sliderSubData.get(1);
String currentVal = sliderSubData.get(2);

slider_data.setValueTo(Float.parseFloat(rangeMax));
slider_data.setValue(Float.parseFloat(currentVal));
slider_data.setValueFrom(Float.parseFloat(rangeMin));

sliderDataDisplay.setText(currentVal);
layout.addView(mqttCard);
```

A kártyák létrehozásakor a metódus megkapja az adott kártyára vonatkozó adatokat egy „savedCardData” listában, melynek 3 eleme van: topic, kártya típus és a kártyáról tárolt adat. A csúszka esetében a kártyáról tárolt adatot tovább kell tördelni, mivel nem csak az utolsó állapotát kell tárolni, hanem a felhasználó által megadott minimum és maximum felvehető értékeit is. Ezeket az extra kártyaadatokat kinyerve beállítom a csúszka minden tulajdonságát, majd létrehozom a kártyát.

3.5. Tesztelés

3.5.1. Reszponzivitás

A fejlesztés során az alkalmazást több típusú tesztelésnek tettem ki. A felület reszponzivitása már az első pillanattól fontos pontja volt a fejlesztésnek, így különös figyelmet fordítottam mindenféle eszköztípuson való tesztelésre, ellenőrizve az elemek helyes méretezését, viselkedését, elhelyezkedését, elérhetőségét. Ebben nagy segítséget nyújtottak az Android Studio fejlesztői környezet beépített design eszközei és a tetszőleges formátumú eszközök emulálása.

A reszponzivitás implementálásában sokat segített a már korábban említett Material Design[7] előírások betartása, de így is felbukkantak problémák a fejlesztés során, mint például a képernyő elforgatása során a felület nem hívta meg a renszer szintű felület elrendezés frissítéséért felelős metódust, így minden felhasználó által létrehozott

kártya törlésre került. Ez egy jól ismert probléma az Android rednszerben, így egyszerűen találtam megoldást, egy a Fragment osztály által örökölt metódus felülírásával.

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
        super.onConfigurationChanged(newConfig);
}
```

3.5.2. Hibakezelés

Mivel a rendszer erősen támaszkodik hálózati kapcsolatra, így sok hibakezelést kellett implementálni az esetleges lecsatlakozások, rosszul címzett üzenetek, újracsatlakozások és egyéb hálózati műveletek pontjaira. Ezeken a pontokon exception kezeléssel és naplózással íratom ki a hibát, és ahol szükségesnek éreztem, a felhasználó felé is továbbítottam a hibaüzenetet, természetesen olvashatóra formázva, mivel egy átlagos felhasználónak egy exception hibaüzenete nem biztos hogy érthető.

A felhasználóval való kommunikációt ilyen esetekben úgynevezett „Snackbar”-al, vagy „Toas”-al valósítottam meg, melyek a képernyő alján megjelenő rövid üzenetekben közlik a keletkezett hiba forrását, legyen az kapcsolati vagy akár felhasználói hiba, például egy üresen hagyott beviteli mező.

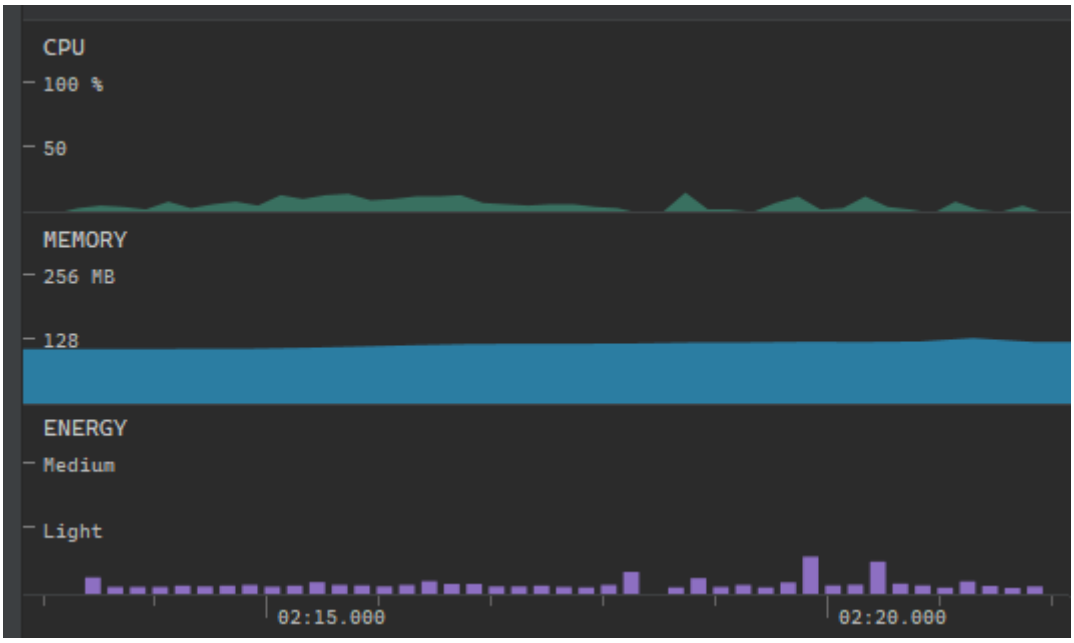
3.5.3. Erőforrás felhasználás

A fejlesztés során az erőforrások felhasználása is figyelembe lett véve, mivel az MQTT egyik legnagyobb előnye az alacsony erőforrás igénye. Az alkamazás futása közben valós idejű méréseket analizálva több olyan erőforrás igényes folyamatot is felismertem, melyek újraírása és optimalizálása után észrevehetően növekedett az applikáció reszponzivitása.

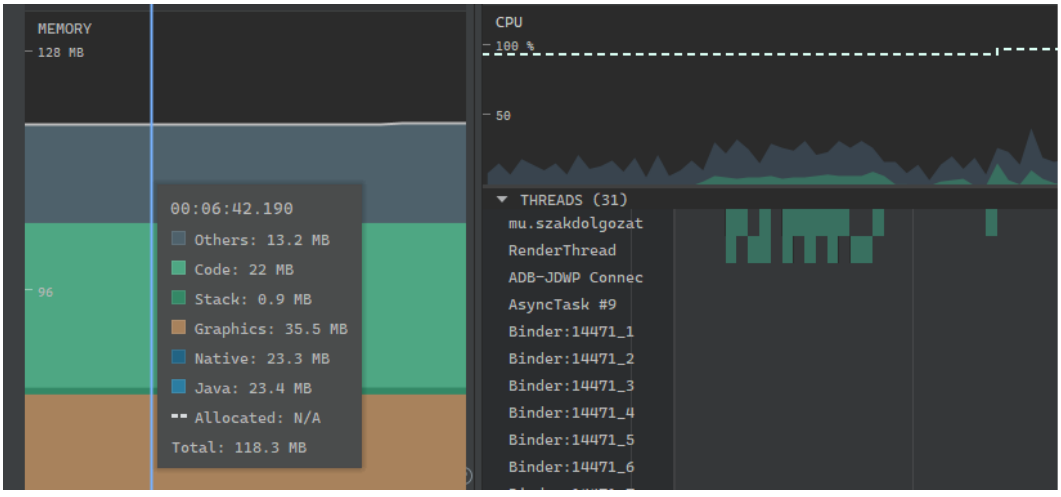
Az alkalmazás legfrissebb verziója stressz-teszt alatti maximum memóriaigénye 130MB, ami alacsonyak számít, azt figyelembe véve, hogy a mai okostelefonok memória kapacitása 8GB alá ritkán esik.

A 3.8-as ábrán látható, hogy az alkalmazás memóriaigényének majdnem egyharmadát a grafikai csomagok teszik ki. A processzorhasználat tesztjeim során maximum 12%-ot ért el, de tipikus használat alatt 6-7% volt ez az érték.

Az MQTT magához hűen rendkívül alacsony mennyiségű processzorhasználatot igényelt, nem volt szükség a kommunikáció optimalizálására.



3.7. ábra. Az app futása közben felhasznált erőforrások



3.8. ábra. Az app által felhasznált RAM és CPU, részegységekre bontva

4. fejezet

A renszer működése

4.1. Első indítás

A Node-RED, a Mosquitto bróker és az Android alkamazás is bizonyos fájlokra támaszkodik, melyek az első indításnál valószínűleg nincsenek jelen. Ez az alkalmazás felől nem jelent problémát, hiszen csak annyit jelent, hogy még nincsenek elmentett felhasználói profilok. A kiszolgáló részéről már kicsit körülményesebb az első indításra való felkészülés, de ez is inkább időigényes mint bonyolult.

4.1.1. Szerver

A kiszolgáló hardveren természetesen telepíteni kell a Node-RED szolgáltatást az általunk preferált Node.js csomagkezelővel, például npm vagy yarn, vagy használhatjuk a Node-RED által biztosított bash scriptet is Linux rendszeren. A szolgáltatás elindítását követően be kell importálnunk, vagy kézileg felépítenünk a az összes okos eszközre vonatkozó logikát, kezelést és irányítást. Ezek a flow-k exportálhatóak és importálhatóak egy JSON fájból, így könnyen hordozható és megosztható előre megépített flow-k vagy akár teljes házak felépítése.

Az Eclipse Mosquitto MQTT bróker telepítése operációs rendszertől függetlenül egyszerű telepíteni, a mosquitto honlapjánal letöltési oldalán található instrukciók alapján. A telepítést követően a 3.3.2-es beszúrt kódban látható módon lehet egyszerűen generálni egy konfigurációs fájlt aminek az alkalmazásával zökkenőmentes a hardver hordozhatóvá tétele, mivel biztosítja a legfrissebb helyi cím átadását a brókernek.

A PM2 renszer ezeket a lépéseket követően az összes indításkor vagy újraindításkor garantálja a kiszolgálók indítását, ezért tekinthető egy egyszeri konfigurációs lépésnek.

4.1.2. Android

Az alkalmazás telepítése egyenlőre a proket GitHub oldalán[15] található „Releases” menüpontban elérhető verziókban letölthető .apk telepítőfájlok által lehetséges, de a jövőben a Google Play Store-on is elérhetővé válik.

Az első indítás nem különbözik más indításoktól, csak a felhasználói profilok hiányában jelentkezhet, de ezek természetesen a felhasználó igénye szerint generálódnak vagy akár törlődnek.

4.2. Telefon csatlakoztatása kiszolgálóhoz

Az alkalmazásban a kiszolgálóhoz való csatlakozáshoz meg kell adni annak az IP címét, amit optimális esetben (a kiszolgáló statikus IP címet kap) nem kell megjegyezni, mivel az alkamazás elmenti az utoljára használt címet és kitölti a felhasználó helyett ezt a mezőt. Ez a tulajdonság igaz a felhasználó nevére is, ami a profil mentésére szolgál.

Ha a kiszolgáló bontja a kapcsolatot az alkalmazással időtúllépés miatt, azaz nem kapott üzenetet az alkalmazástól a kiszolgáló egy előre meghatározott időkereten belül, az alkalmazás értesíti a felhasználót és felkínája az újracsatlakozás lehetőségét, ami megpróbál újbóli kapcsolatot létesíteni a kiszolgálóval, vagy hiba esetén újból értesíti a felhasználót.

4.3. Okos eszközök kezelése az alkalmazásban

Egy új kártya létrehozásához a felhasználónak fel kell iratkoznia egy MQTT topic-ra, majd kiválasztani az interakció típusát. Ezt a folyamatot a Home fragmenten található jobb alsó sarokban elhelyezkedő gomb érintésével tudja kezdeményezni a felhasználó. Először meg kell adni a topic nevét, ezt egy egyszerű szöveges beviteli mezőn teheti meg, majd az interakció típust kell kiválasztani egy előre meghatározott listáblól, amit egy legördülő menüben lehet kiválasztani. A csúszka típusú kártyáknak van egy extra létrehozási követelménye, mégpedig a csúszka határainak beállítása, melyet a típus kiválasztása után két számértéket elváró beviteli mezőben adhat meg a felhasználó.

A sikeres feliratkozást követően megjelenik a létrehozott kártya a felületen és már készen is áll az üzenetek fogadására vagy továbbítására, azaz a kiszolgálóval való kommunikációra. Ebből adódóan a kártyák létrehozásának előkövetelménye, hogy a felhasználó a feliratkozást megelőzően kapcsolatot létesített a kiszolgálóval a Settings fragmenten belül.

Egy létrehozott kártya utólag nem módosítható, így ha a topicon vagy a kártya típusán szeretnénk módosítani, a módosítani kívánt kártyát törölni kell, majd egy új kártyát létrehozni.

A létrehozott kártyák korábban említetten készen állnak a kommunikációra, így a felhasználónak nincsen további teendője, máris kezébe veheti az okos otthonának irányítását. Minden kártya interakciós felülete reszponzívan kommunikál a kiszolgálóval, így nem kell extra lépéseket tennie a felhasználónak az üzenetek továbbításáért, például a kapcsoló típusú kártyán lévő kapcsoló megérintését követően azonnal továbbítja állapotát a kiszolgáló felé.

4.4. Okos eszközök kezelése a webes felületen

node-red ui, több vele a szarakodás mint androidon so nem előnyös, része az üzemeltetésnek A Node-RED Dashboard lehetőséget nyújt egy webes felület létrehozására, amin keresztül lehet irányítani a beépített rendszereket. Mivel ennek a felületnek az elemei nem specifikusan okos otthon vezérkésére készültek és az egyes elemeket a rendszer programozható felületén kell implementálni, újabb elemek felhelyezése és funkcionalitásának implementálása nem egy felhasználóbarát folyamat.

Ettől függetlenül a felület nem haszontalan, mivel egy olyan irányítőközpontot lehet rajta prezentálni, amit nem csak androidos telefonon lehet elérni, így szélesebb körűvé teszi az okos otthon vezérlési lehetőségeit. Egy olyan otthonban, ahol az okos eszközök előre láthatólag rövid időn belül nem lesznek bővítve, érdemes ezt a felületet létrehozni az általa nyújtott rugalmas elérés érdekében. Ennek a felületnek a kialakítása a rendszer telepítésekor, vagy későbbi bővítések során a rendszer karbantartásáért felelős rendszergazda szerepkörébe tartozik.

5. fejezet

Továbbfejlesztési lehetőségek

user auth: bár mqtt-nél nincs sok értlme, maybe parentala controls cloud service for out of home control andoid auto-looks for the broker, this might be expensive and hard ui breaks home ui into multi column layout if it fits, such as on tablets n stuff grouping/folders

Köszönetnyilvánítás

Köszönöm a vscode-nak hogy van,

Köszönöm magyarországnak hogy jobban teljesít

Irodalomjegyzék

[1] Node Red forrás,
https://nodered.org

[2] Node Red Dashboard forrás,
https://flows.nodered.org/node/node-red-dashboard

[3] Orange Pi forrás,
http://www.orangepi.org/

[4] MQTT protokol forrás,
https://mqtt.org

[5] Mosquitto bróker forrás,
https://mosquitto.org

[6] Paho Android MQTT implementáció,
https://www.eclipse.org/paho/index.php

[7] Material design forrás
https://material.io/components/

[8] Android fejlesztői dokumentáció
https://developer.android.com/guide

[9] Armbian operációs rendszer
https://www.armbian.com/

[10] IoT based Smart Environment Using Node-RED and MQTT
Deepthi, B. & Kolluru, Venkata Ratnam & Varghese, George & Narne, Rajendraparasad & Srimannarayana, Nerella. (2020). IoT based Smart Environment Using Node-RED and MQTT. Journal of Advanced Research in Dynamical and Control Systems. 12. 10.5373/JARDCS/V12I5/20201684. https://www.researchgate.net/publication/342327250_IoT_based_Smart_Environment_Using_Node-RED_and_MQTT

[11] Android™Notes for Professionals book
https://books.goalkicker.com/AndroidBook/

[12] IMB Emerging Technologies
https://emerging-technology.co.uk/

[13] OpenJS Foundation
https://openjsf.org/

[14] PM2 Process Manager
https://pm2.keymetrics.io/

[15] A Projekt GitHub oldala
https://github.com/Lovasz-Akos/Szakdolgozat-FMNUMU