Matematikai és Informatikai Intézet

Moduláris Plug & Play IoT / okos otthon rendszer

Készítette
Farkas Levente
Program tervező informatikus BSc

Témavezető
Dr. Geda Gábor
egyetemi docens

Eger, 2025

Tartalomjegyzék

Bevezetés

Az emberi civilizáció létezésétől kezdve a kényelemre törekedett. Minden nap valamivel könnyebbé, egyszerűbbé akarjuk tenni a mindennapokat, hogy minél kevesebbel kelljen foglalkoznunk és több időt tölthessünk más tevékenységgel. Az okos otthon rendszerek pontosan ezt teszik lehetővé, ezek régen drágák voltak így a kevésbé tehetős emberek nem engedhették meg maguknak. Ma azonban olcsóbb ökoszisztémák is léteznek, de még így sem tudják sokan megfizetni. A projekt, amiről ezen dolgozat szól egy olcsóbb, közösség alapú, Apple szerű ökoszisztéma létrehozására törekszik, amely költséghatékonyabb megoldást kínál.

Számomra lenyűgöző belegondolni abba, hogy pár sor kóddal ki tudunk hatni a környezetünkre. Főképp azért, mert a programozás szempontjából én mindig adatkezelésként tekintettem a munkánkra. Még a szakdolgozat témájának kiválasztása előtt kapcsolatba kerültem a mikrokontrollerek világával és attól a ponttól nem tudtam elengedni ezen eszközök varázsát. Tekintettel arra, hogy az áramkörök és elektromosság foglalkoztatott hobbi szinten már régóta, ez egy nagyszerű lehetőség volt, hogy kettő nagy szenvedélyemet, az informatikát és az áramköröket ötvözzem a szakdolgozatomban. Számomra hihetetlen volt akár csak egy LED[1] villanása is.

Szakdolgozatom termék-orientált. Célja az, hogy a kevésbé tehetős emberek számára is elérhetővé tegye az okos otthonok varázsát. Általános tapasztalat szerint az okos eszközök árát nagyrészt a programozható mikrochipek teszik ki, ezeknek egy részről a hozzá tartozó áramkört kell vezérelniük, amely nem egy nehéz feladat, azonban az adatnak, amely alapján ez működni fog valahogy el kell jutnia az eszközhöz. Az elemek kapcsolata vezeték nélküli kapcsolattal valósul meg, amely minden esetben WiFi-n keresztül történik, azonban ez a fajta megközelítés jelentősen megemeli a hardver árát. Elképzelésem szerint a több különálló drága technológiát központosítani lehetne, amely az összköltség csökkenéséhez vezetne.

Az eszköz önmagában nem képes sok mindenre, azonban a hozzá kapcsolódó „buta" modulok használatával lehetőség nyílik arra, hogy egy eszköz négy modult, azaz működésében különálló okos eszközt kezeljen. A fejlesztés idejében az eszköz négy modult képes befogadni, azonban ez áramkör és program módosítással növelhető.

Számomra fontos volt még az általam nem ismert, vagy nem használt technológiák

[1] Light Emitting Diode, avagy fényt kibocsájtó dióda.

használata, így próbáltam minél több számomra új technológiát használni, ezekről a technológiákról bővebben a 1. fejezetben olvashatnak.

1. fejezet

Technológiák

1.1. Eszköz oldalon alkalmazott technológiák

A rendszer sokrétű felépítése eredményeként több technológiát és programot fel kellet kutatnom, valamint ezek használatát minél jobban el kellett sajátítanom.

1.1.1. Programozási nyelv

Ahogy később a a 2. fejezet a 2.1. szekciójában olvasható az ESP32 eszközöket C/C++ nyelveken lehet főképp programozni, ezek mellett script nyelveket is lehet használni.

A megoldásban első sorban C++ nyelvet akartam használni, azonban ez a megközelítés felvetett számos problémát a megvalósítás közben (ezekről bővebben a a 2.2.2. alszekcióban olvashat). A problémák hatására célszerűnek találtam a C++ programozási nyelv leváltását, amelyet a MicroPython váltotta fel.

A fejlesztés alapjául a programozási nyelv változásának eredményeként a ESP32_GENERIC_S3-SPIRAM_OCT-20241129-v1.24.1 firmware szolgált . Ami elérhető a MicroPython weboldalán.

1.1.2. Integrált fejlesztői környezet, IDE

A fejlesztéshez választott fejlesztői környezet eredetileg a Visual Studio Code volt. – azonban bármely extensiont próbáltam használni egyik sem tudta sikeresen feltölteni a programot az ESP-re – Ennek eredményéül a fejlesztés első szakaszában a VSCode-ban megírt programot az Arduino IDE segítségével töltöttem fel.

A programozási nyelv váltása után a VSCode továbbra sem volt működőképes, valamint az Arduino IDE nem támogatta a MicroPython nyelvet. Miután a probléma megoldására általam ismert módszerek tárháza kifogyott elkezdtem egy másik IDE után keresni. A keresésem eredményeként rátaláltam egy weboldalra[1], ahol több fej-

[1] `https://randomnerdtutorials.com/micropython-ides-esp32-esp8266/`

lesztői környezetet említenek. A keresés folyamatában számos GitHub és StackOverflow beszélgetést néztem át a válasz után kutatva. Ezekben a beszélgetésekben többször is említették a ThonyIDE-t. Ennek eredményéül kezdtem el használni az említett környezetet.

1.2. A REST API technológia háttere

Az alkalmazás kommunikációs rétegét egy REST API biztosította, amelyet Node.js platformon fejlesztettem Express.js keretrendszer segítségével. A választott technológiák lehetővé tették a gyors prototípuskészítést és a skálázható szerveroldali logika kialakítását.

1.2.1. Kulcstechnológiák és Döntések

1. Node.js és Express.js
   - A Node.js ideális volt aszinkron, eseményvezérelt architektúrájából eredően a párhuzamos kérések kezelésére.
   - Express.js-re a gyors fejlesztési ciklus és egyszerű útvonalak kialakítása miatt esett a választás.
2. Adatcsere formátuma a JSON[2] több okból is.
   - Széles körben használt és elfogadott modern fejlesztői eszközökben.
   - Könnyen értelmezhető ember és gép számára is.
   - Alacsony adatmennyiségű, de strukturált információtovábbítást tesz lehetővé.

A bejövő HTTP kérések törzsében rejlő JSON adat feldolgozásának könnyítése érdekében a body-parser könyvtárat használtam.

1.2.2. Technológiai előnyök

- Platformfüggetlen. Az API-t bármely eszköz használhatja, legyen az beágyazott eszköz, vagy egy mobil applikáció.
- Teljesítmény. A Node.js által használt I/O műveletek nem blokkolók, így tökéletes az esetlegesen több ezer eszköz kéréseink fogadására.

[2] JavaScript Object Notation

1.3. Adatbázis

A MongoDB egy nyílt forráskódú, dokumentumorientált NoSQL adatbázis, amelyet skálázható és nagy teljesítményű alkalmazások fejlesztésére terveztek. A hagyományos relációs adatbázisokkal (pl. MySQL, PostgreSQL) ellentétben a MongoDB nem táblákban, hanem rugalmas, JSON-szerű dokumentumokban tárolja az adatokat. Ez lehetővé teszi a gyors és hatékony adatkezelést, különösen olyan esetekben, amikor az adatok szerkezete dinamikus vagy változó.

Fő jellemzői a MongoDB-nek:

1. Dokumentumorientált adatmodell:
   - Az adatokat BSON (Binary JSON) formátumban tárolja, ami egy bináris reprezentációja a JSON-nek.
   - Egy dokumentum egy kulcs-érték párokból álló struktúra, amely hasonlít a programozási nyelvekben használt objektumokhoz.
   - Példa egy dokumentumra:

```
{
    "_id": {
        "$oid": "67d6c6e5e3c0cb995623d001"
    },
    "owner": "testOwner",
    "location": "Unknown",
    "device_name": "Unnamed",
    "aditionalInfo": {
        "canBeControlledBy": [
            "testOwner2"
        ]
    }
}
```

Kódrészlet 1.1. Egy teszt dokumentum az adatbázisból.

2. Rugalmas séma:
   - A MongoDB nem követeli meg, hogy minden dokumentumnak ugyanaz a szerkezete legyen. Ez lehetővé teszi, hogy különböző típusú adatokat tároljunk ugyanabban a gyűjteményben (collection).
   - Például egy users gyűjteményben lehetnek olyan dokumentumok, amelyeknek nincs age mezője, vagy más mezők is lehetnek.

3. Skálázhatóság:
   - A MongoDB horizontálisan skálázható, ami azt jelenti, hogy az adatbázis több szerverre (shard) osztható szét, hogy kezelni tudja a nagy mennyiségű adatot és a magas terhelést.
   - Támogatja a replikációt is, ami magas rendelkezésre állást és hibaelállást biztosít.
4. Teljesítmény:
   - A MongoDB gyors adatelérést biztosít indexek használatával. Többféle indexet támogat, beleértve az egyszerű, összetett, szöveges és geospatial indexeket.
   - A memóriában történő adatkezelés (in-memory storage) tovább növeli a teljesítményt.
5. Aggregáció és lekérdezés:
   - A MongoDB hatékony lekérdezési nyelvet (Query Language) és aggregációs keretrendszert (Aggregation Framework) biztosít az adatok elemzéséhez és feldolgozásához.

A MongoDB továbbá támogatja a CRUD műveleteket (Create, Read, Update, Delete), valamint a tranzakciókat is (a 4.0 verziótól kezdve).[1] A MongoDB nem kényszeríti a sémák használatát, azonban van rá lehetőségünk, ha a feladat azt kívánja. A sémákban megadhatunk kötelező mezőket, azok típusát, valamint ezekhez köthetünk különböző ellenőrzéseket, úgynevezett validátor értékeket.

```
{
    validator: {
        $jsonSchema: {
            bsonType: "object",
            required: [
                "username",
                "email",
                "password"
            ],
            properties: {
                username: {
                    bsonType: "string",
                    description: "Only String type is
                       allowed!"
```

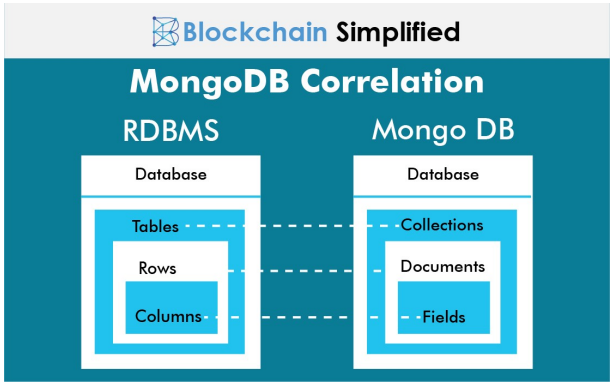
```
                    },
                    email: {
                        bsonType: "string",
                        pattern:"^[0-9a-z-]+@[0-9a-z-]+\.[a-z]+$",
                        description: "Invalid email adress or
                            data type is not string!"
                    },
                    password: {
                        bsonType: "string",
                        description: "Only String type is
                            allowed!"
                    }
                }
            }
        }
}
```

Kódrészlet 1.2. Felhasználó collection sémája az adatbázisból. (A séma mezőinek azonosítója nem felelnek meg a json nyelv szabájainak, mivel a fájl tartalma konzolban került bevitelre.)

Ahogy a 1.2.-es kódrészletben is látható típus és mező megszorításokat tartalmaz a séma, e mellett az email mező még reguláris kifejezéssel is vizsgálva van.

1.3.1. MongoDB struktúrája

A MongoDB az adatbázison belül táblák helyett collection-ök vannak, ezen belül rekordok, avagy sorokkal ellentétben dokumentumokat találunk, ahol oszlopok helyett mezők vannak. Az összehasonlítást egy hagyományos SQL adatbázissal a 1.1. ábrán lehet látni. [1]



1.1. ábra. SQL adatbázis és MongoDB szerkezeti hasonlóságai.

1.4. Frontend technológia

Az eszköz hatékony vezéréléséhez szükségünk van egy applikációhoz, azonban lehet, hogy egyes emberek nem mindenhez a telefonjukat akarják használni. Ez problémát vet fel, mivel több kódbázis fenntartása és javítása sok embert és erőforrást igényel.

Megoldásként egy olyan keretrendszerre, vagy programozási nyelvre van szükség, amely működőképes több platformon is.

Például:

- React Native
- Kotlin
- Flutter
- Lynx

1. *Megjegyzés.* A legújabb keretrendszer a LynxJs, amely 2025.03.05.-én jelent meg. Ahogy a React Native ez is javascript alapú, így a Lynx és React Native egyfajta versenyhelyzetben vannak. A keretrendzerek összehasonlítását olvashatja a Medium.com-on, valamint a LynxJs.org-on tudhat meg többet a keretrendszerről.

React Native-ról meglehetősen sok negatív megjegyzést hallottam, miszerint rendkívül nehéz a használata és rengeteg nehezen javítható problémát eredményez, ha nem tökéletesen használjuk. Ismeretségi körömbe tartozik egy személy, aki arra esküdött fel, hogy soha többé nem fogja használni a túlságosan komplikált fejlesztés eredményeként. A Lynx tekintve, hogy rendkívül új, nagy valószínűséggel több, még rejtett hibákat tartalmazhat, valamint a dokumentáción kívül logikám szerint rendkívül alacsony a jelenleg elérhető egyéb források száma. A Kotlin, mint a Java helyettesítője alapvetően egy JVM[3]-et használ a program futtatására, amely nem optimális teljesítményhez vezethet. A probléma megoldására létrejött a Kotlin Native[15], azonban rendkívül lassú a JRE[4] verzióhoz képest, valamint nem létezik hozzá sok alapvető csomag, vagy „könyvtár”[16]. A fent említett okok miatt, valamint a tény, hogy a Flutter natívan futó programot állít elő miatt választásom a Flutter-re esett.

1.4.1. Miért Flutter?

- Könnyű crossplatform build lehetőség
- Opensource
- Gyorsaság[5]

[3] Java Virtual Machine
[4] Java Runtime Environment
[5] „Flutter is powered by Dart, a language optimized for fast apps on any platform”[2]

– Új technológia megismerése

– Native build-ek

A Flutter ezen tulajdonságai rendkívül kedvezővé tette a frontend applikáció elkészítéséhez.

1.4.2. Flutter ismertetése

A Flutter egy olyan eszköz, amely lehetővé teszi a fejlesztők számára, hogy natív, többplatformos alkalmazásokat hozzanak létre egyetlen programozási nyelv és egyetlen kódbázis segítségével. Nem olyan alkalmazást hoz létre, amely böngészőben fut, vagy amelyet natív alkalmazásokba csomagolnak. Ehelyett natív alkalmazásokat készít iOS-re és Androidra is, amelyek később közzétehetők az app store-okban.

Az alkalmazást létrehozásához egyetlen programozási nyelvre van szükségünk, ahelyett, hogy különböző nyelveket használnánk egy iOS vagy Android alkalmazás fejlesztéséhez. Így csak egyetlen kódbázissal kell foglalkoznunk. A Flutter egy SDK[6], amely lehetővé teszi, hogy a kódbázisodat natív gépi kódra fordítsd, amely a fent említett platformokon fut.

A fordítási eszközök mellett a Flutter keretrendszerként is funkcionál, UI építőelemeket (widgeteket) biztosít, mint például lapok, legördülő menük, gombok stb., valamint néhány segédfüggvényt és csomagot, amelyeket az SDK segítségével lehet lefordítani.[3]

Flutter applikációt a fent említettek mellett még Windows-on, Linux-on és Web-en is lehet használni.

[6] Software Development Kit

2. fejezet

ESP32, az eszköz

A projekt kulcsfontosságú része, maga az eszköz, amely lehetővé teszi a modulok vezérlését. A 2.1-es ábrán látható mikrokontrollerhez hasonlót több gyártó is gyárt, azonban egy ESP kifejezetten a benne található processzortól lesz ESP. Az ESP32-ben található Xtensa LX6 processzor két maggal van ellátva és órajele 240 MHz[1].

2.1. ESP32 bemutatása

Az ESP32 egy rendkívül sokoldalú és hatékony mikrokontroller, amely számos területen alkalmazható. Két magos processzora, integrált WiFi és Bluetooth képességei, valamint gazdag I/O lehetőségei miatt kiváló választás mind IoT, mind ipari vagy okos otthon alkalmazásokhoz. Az ESP32 fejlesztése egyszerű és gyors, köszönhetően a széles körben elérhető fejlesztői eszközöknek és könyvtáraknak. Ezek a tulajdonságok teszik az ESP32-t az egyik legnépszerűbb mikrokontrollerré a beágyazott rendszerek világában. Mind e mellett az általam választott változat nem szenved RAM, vagy ROM hiányában sem. Az adatlap szerint el van látva 512 KB SPRAM-al, 384 KB ROM-al és változó méretű PSRAM[2]-al. Az utóbbi a használt eszköz esetében 8 Mb, valamint még fel van szerelve 32MB Octal Flash[3] memóriával.

Az általam a 2.1 használt modell az ESP32-S3-DevKitC-1 N32R8V, amely egy ESP32-S3-WROOM-2 chipet tartalmaz, ezt az Espressif System fejlesztette ki.

Ezen eszközök számos bemeneti és kimeneti perifériákkal rendelkeznek:

- Digitális I/O pin-ek
- Analóg bemenetek (ADC[4])

[1] Megahertz

[2] Pseudo SRAM

[3] Az Octal (xSPI) Flash memória egy gyors memória típus, amely kedvelt beültetett rendszerek között, nagyrészt a memóriáról történő gyors boot-nak köszönhetően. A memória típus 400MB/s olvasási sebességre képes.[17]

[4] Analog to Digital Converter, lehetővé teszi az áramkörben érzékelt analóg jelek feldolgozását.

2.1. ábra. Általam használt ESP32 modell

– Digitális-analóg átalakítók (DAC[5])

– PWM (Pulse Width Modulation[6]) kimenetek

– SPI, I2C, UART kommunikációs interfészek

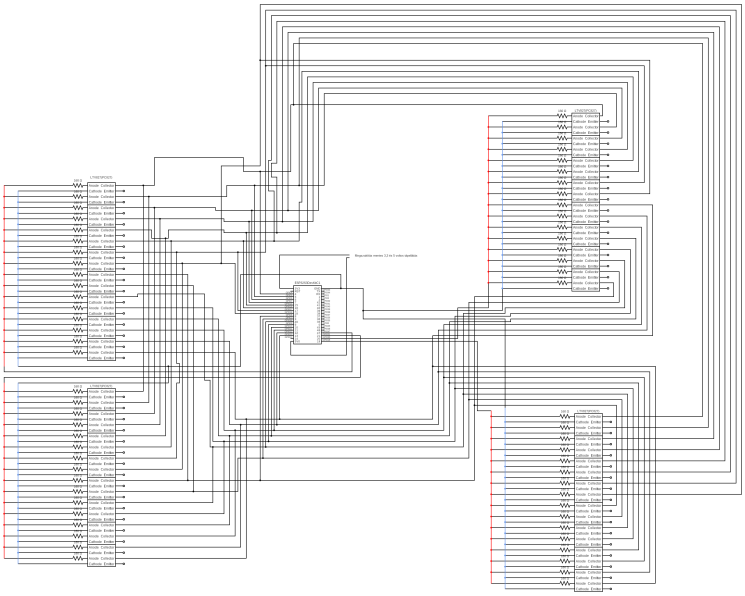
Ezek a csatlakozók lehetővé teszik különböző érzékelők, kijelzők és egyéb perifériák csatlakoztatását. Ezeket az eszközöket nagyrészt C és C++ nyelven lehet programozni, azonban van lehetőség a MicroPython és Lua script nyelvek használatára.

2.2. Fejlesztés

2.2.1. Hardver szintű fejlesztés

A projekt megvalósításához szükség volt egy egyedi áramkörre, amivel maximalizálhatom az elérhető portok számát. A 2.2-es ábrán látható az említett áramkör, amely segítségével elérem az általam elfogadható eredményt. Ebben a kulcsfontosságú alkatrészek az optocsatoló, amelyek használatával igyekeztem az esetlegesen előforduló interferenciát csökkenteni. Ez a kapcsolási rajz a Circuit Diagram segítségével készült. Az ábra nem tartalmazza az érzékelő vezetékeket, amivel az eszköz érzékeli, hogy csatlakoztatva van-e a modul.

[5] Digital to Analog Converter, lehetővé teszi az áramkörre kibocsájtott feszültség modulálását.
[6] Ezen technológia teszi lehetővé a DAC működését.



2.2. ábra. Hatékonyság növelés érdekében kifejlesztett egyedi áramkör.

2.2.2. Szoftver fejlesztése és feltárt problémák

Fejlesztés elején több probléma is felmerült.

1. A modul vezérlő fájlok tárolása. Amennyiben C++ vagy C nyelvet használunk a belső tárhely kizárólagosan az általunk feltöltött fájlokat tartalmazza és a hozzáférés nehézkes fájlrendszer nélkül. Szerencsére elérhető több csomag is ennek a megoldására, mint a LittleFS, vagy a Spiffs

2. A modul vezérlő fájlok használata, vagy futtatása. A projekt egyik fő feladata, hogy a modulok programja és a fő program egymástól függetlenül legyenek képesek működni, azaz ne a fő program hajtsa végre a szükséges lépéseket, hanem a modul vezérlő. Egy megoldás erre a fájl dinamikus betöltése és a program belépési pontjának meghatározása, majd a pont memória címén keresztüli futtatás. Ez a megközelítés a programot rendkívül bonyolultá tette volna és memória biztonsági problémákat is felvetett volna, mivel a C++ nyelvben íródó programnak C-t is használnia kellett volna.

Ez a kettő komolyabb probléma vezetett arra, hogy más nyelvet keressek. A MicroPython mind a kettő problémára megoldást nyújtott a beépített fájlrendszerével és JIT[7] interpreterével, ugyanis így nem bináris kóddal kell dolgozni, hanem egyszerű *.py* fájlokkal. Ez a változtatás nagyban megkönnyítette a program fejlesztését.

A váltás után neki tudtam kezdeni a tényleges fejlesztésnek. Itt a főbb problémákat a fájl letöltése és azoknak a futtatása jelentette. A fájl letöltését több módszerrel is megkíséreltem, azonban egyik sem hozott megfelelő eredményt, végül a Google API

[7] Just In Time

használatával sikeresen le tudtam tölteni a fájlt. Az API mindössze a Google-től igényelt API kulcsot és a fájl azonosítóját kéri (Az azonosító a megosztási linkből kinyerhető.).

Ezen a ponton a szerverrel való kommunikáció is kérdéses volt. Elsőre Websocket technológiát terveztem használni, azonban ezt drágának találtam és a szervernek egyszerre több kapcsolatot is fenn kellett volna tartania, miközben relatívan kevés és ritka az információ áramlása. A következő ötlet egy az eszközön futó API volt, ez azonban ennek a megoldásnak, hogy működőképes legyen szükséges az eszköz IP címe, valamint esetlegesen portforwarding-re van szükség amely nem egyezik az eszköz Plug&Play irányzatával. Ennek eredményéül a jól ismert RestAPI technológia mellett döntöttem. Az API-ról többet olvashat a 3. fejezetben.

API-ra még így szükség van a konfigurációhoz, ehhez a Microdot könyvtárat használtam, valamint a szerver felé irányuló kérésekhez a urequests és az abból származó adatok feldolgozásához a nyelv által alapértelmezetten tartalmazott modulokat használtam, például a *ujson* modult. A konfigurációhoz használható más technológia az eszköz esetében a Bluetooth volt, azonban ezt az egyszerűség kedvéért nem használtam.

2. *Megjegyzés.* A Bluetooth-os konfiguráció kihagyása, jó döntés volt. A Tarlogic Március 6. megjelent cikk-je alapján[18]. A cikk említést tesz egy több millió eszközt sújtó sebezhetőségi pontról. A cikk által említett eszközök Bluetooth chipjeiben rejtett funkcionalitást véltek felfedezni, amely biztonsági résként fogható fel.

A felhasználónak mindenféle képen jelezni kell az eszköz belső állapotáról, ennek a legegyszerűbb formája a színek használata, szerencsére az eszköz el van látva egy beépített NeoPixel LED-el, amely használatához a NeoPixel könyvtárra volt szükségem.

A konfigurációhoz és működéshez szükséges a MicroPython-ban alapértelmezetten megtalálható network modul. A modul segítségével tud az eszköz létrehozni saját WiFi hálózatot, valamit csatlakozni egy már meglévő hálózathoz. Szintén alapértelmezett csomag az OS csomag, amely a legtöbb Python verzióban megtalálható. Ez a könyvtár rendkívül fontos az eszköz működésének szempontjából, mivel használatával dönthető el, hogy bizonyos fájlok, például az eszköz saját azonosítóját tartalmazó fájl jelen van-e a fájlrendszerben.

A műveletek megvalósítása egy végtelen ciklusban történik, ahol minden iteráció egy a szerver felé irányuló kéréssel kezdődik. A végpont amely kezeli ezt a kérést az egyetemen tanultaktól eltér, a végpontról többet olvashat a 3.2.2. szekcióban. Innentől az esetlegesen több feladatot egy a végtelen ciklusba beágyazott **for** ciklus kezeli, amely törzsében minden egyes feladatot megvizsgálunk.

A ciklus törzsében található if-elif szerkezet döntéshozó szervként működik. A feladat, mint JSON dokumentum *aditionalInfo* mezőjében található *method* mező rejti a feladat típusát, ezek a típusok és hozzájuk társított folyamatok megtekinthetőek a 2.2.3. szekcióban található felsorolásban.

Ezen folyamatok közül a legnehezebben a „run” volt megvalósítható. Itt a modulhoz társított python fájl azonosítója alapján választjuk ki a tárolt modulok közül a feladatban megadott modult. Minden modulhoz tartozik egy metódus, például a az első számú modulhoz a „runa” metódus tartozik. Ezek a metódusok kivétel nélkül el vannak látva egy dekorátorral, amely feladata a megfelelő áramkör kiválasztása. Ezen metódusok belsejében hívódnak meg a különböző modulok „run"" metódusai, amely egy paraméterrel van ellátva. Ez a paraméter tartalmazza a feladat paramétereit.

2.2.3. Működése

Az eszköz használata konfigurációval kezdődik, melyet kék fénnyel jelez a felhasználónak. A konfiguráció után megpróbál csatlakozni a megadott hálózathoz eközben az eszközön található LED kék fényről pulzáló zöld fényre vált. Amennyiben a kapcsolódás sikeres az eszköz zöld fénnyel jelzi ezt, ha sikertelen, akkor újraindul a folyamat.

Sikeres kapcsolódás után önmagát regisztrálja az eszköz a hálózatra. Innentől az eszköz használatra kész és azonnal kérést küld az API-nak a hozzá kapcsolt feladatok iránt érdeklődve. A szerver válasza alapján a következő műveleteket végezheti el az eszköz:

- „add_new”: A parancs kiadásával új modult tudunk hozzáadni az eszközhöz.
- „remove”: Az „add_new” parancs ellentettje. Modul eltávolítására adja ki a parancsot.
- „disown”: Gyári visszaállítás.
- „network_reset”: Az eszköz elfelejti a hálózatot. Hasznos lehet, ha az eszközt egy másik épületbe helyezzük át.
- „module_reset”: Az eszköz belső állapotát visszaállítja, ezzel egyszerre az összes modul eltávolítható.
- „run”: Futtatja a modulhoz megadott programot.

Fontos megjegyezni, hogy minden feladat elvégzése annak törlésével és egy új kéréssel záródik. A „run” parancs esetén az eszköz a modul azonosítója alapján dönti el, hogy melyik modult kell használnia, azaz megállapítja a modul áramkörét és kiválasztja azt. Az „add_new” esetén az eszköz újraindul, hogy a program sikeresen érzékelni tudja a fájl változásait, erre a „remove” parancs esetén nincs szükség, mivel a program belső állapota változik csak.

Az eszköz esetleges áramkimaradásra fel van készítve, mivel a belső állapot változása folyamatosan mentve van a belső tárhelybe.

3. fejezet

A REST API fejlesztése

Az API az Express.js keretrendszerrel készült és minden entitás CRUD műveleteit ellátja.

3.1. Adatbázis kezelése

Az adatbázishoz egy egyszerű connect stringel történik, amelyet a MongoDB oldalán kérhetünk. A string tartalmazza a felhasználónevet, fiókhoz tartozó jelszót és más az adatbázissal kapcsolatos adatokat, mint az appName paramétert.

A kapcsolat létesítéséhez szükség van a MongoDB natív JavaScript driver-ére, ezt az NPM segítségével letölthetjük, majd az ebben található MongoClient osztályt importálhatjuk és új példányt hozhatunk létre, az osztály konstruktora a már feljebb említett connect stringet várja paraméterül. Innentől kiválaszthatjuk az adatbázist amin dolgozni fogunk.

A programkódban minden collection-re külön fájl készült. Az adatbázishoz tartozó metódusok mindig egy JSON objektumot várnak. A műveletek elvégzését MongoBD által definiált metódusok hívásával lehetségesek. Az adatbázis különbséget tesz az alapján, hogy egy vagy több dokumentumot várunk eredményül, ez az összes metódusra igaz. A collection-ben való keresés nem nehéz, csak meg kell adni a mező nevét és annak értékét, ez azonban nem annyira egyszerű, ha a „_id” mező alapján akarunk keresni, hiszen az eddigiek alapján arra készülünk, hogy a megfelelő mezőhöz kell kapcsolni az adatot, azonban ahogy a 1.1.-es dokumentumban láthatjuk a mező nem string hanem objektum típusú, ez azonban nem észrevehető, ha a klienst, vagy a weblapot használjuk. A dokumentációt[4] keresve nem találtam megoldást így a fórumhoz[5] fordultam, ahol megtaláltam amit kerestem. Ahhoz, hogy ID alapján tudjunk keresni szükségünk van az „ObjetId” nevű osztályra, ennek a konstruktorába kell elhelyezni az ID-t.

3.2. Végpontok

Ahogy az adatbázis kezelésnél itt is minden collection-nek külön fájl van dedikálva, nagy részük nem különbözik a már megszokott végpontoktól. Azonban, az nem optimális, ha az okos eszköz folyamatosan kéréseket küld az API-nak. Aprobléma megoldása az IoC[1] alkalmazása akár csak a modern GUI-val ellátott alkalmazások esetén. Nem a program vár vezérlésre, hanem majd jelzünk a programnak, hogy végezze el a feladatot. Ez azt jelenti, hogy az API-nak kell jeleznie az okos eszköznek a feladat elvégzésére. Erre egy jó megoldás lehet az RPC[2], azonban így a szervernek tudnia kell az eszköz IP címét, valamint portforward-ot kell végeznünk.

3.2.1. Mi felel meg az elvárásoknak?

Elvárásaink:

1. Az eszköznek kell kezdeményezni a kapcsolatot, ezzel elkerülve a lehetséges problémákat melyeket a hálózat, vagy a tűzfal okozhat.

2. Az eszköznek várnia kell a feladatra, nem küldhet request-et percenként a szervernek.

3. Az eszközt értesíteni kell a feladat létrejöttéről, akárcsak az Observer.

Ezen elvárásoknak megfelel például a WebSocket, azonban ezt túlságosan pazarló megoldásnak találtam, ahogy a 2.2.2.-es szekcióban is említettem. Egy másik Lehetőség az SSE[3]-volt, azonban míg könnyű implementálni a kapcsolat csak egy irányú, ezáltal az eszköz nem küldhet vissza adatot a szervernek[6]. Az SSE-ről többet olvashat a 3.2.1. szekcióban, valamint megtekintheti az SSE és WebSocket összehasonlítását a 3.1. táblázatban.

Egy másik lehetőség az MQTT[4] volt, amely IoT felhasználásra tökéletes, azonban, hogy ez működjön szükségünk van egy broker-re pl.: Mosquitto, AWS IoT. Ez egy plusz költséget jelent a rendszerben, amely drágábbá tenné a rendszert, ez nem lehetőség[7]. Az MQTT-ről többet olvashat a 3.2.1. szekcióban.

Egy pénzügyileg kedvezőbb megoldás a Polling. Amely HTTP protokollal tökéletesen működik és eleget tesz az 1. feltételnek és a 3. kritériumnak is, azonban ebben az esetben az eszköz bizonyos intervallumonként újra kérést fog küldeni a szervernek, ez a 2. pontnak nem felel meg.

A Polling egy változata a Long-Polling azonban megfelel az elvárásainknak. A módszer implementálásával, az eszköz addig várakozik a válaszra amíg azt meg nem kapja,

[1] Inversion of Controll
[2] Remote Procedure Call
[3] Server Sent Event
[4] Message Queuing Telemetry Transport

majd annak érkezésével végrehajtja az abban kapott feladatot és az esetleges eredményeket vissza is tudja küldeni.

SSE

Az SSE (Server-Sent Events) egy olyan HTTP-alapú technológia, amely lehetővé teszi, hogy a szerver valós időben küldjön adatokat a kliensnek egy egyirányú kommunikációs csatornán keresztül. Az SSE-t gyakran használják valós idejű értesítések, frissítések és adatstreamelés céljaira, ahol a kliensnek folyamatosan frissülő információra van szüksége anélkül, hogy újra és újra lekérdezné a szervert[11, 6].

Működési elv

Az SSE a standard HTTP/1.1 vagy HTTP/2 protokollt használja, és egy tartós TCP-kapcsolaton keresztül működik. A kliens (általában egy böngésző) létrehoz egy kapcsolatot a szerverrel egy speciális EventSource API segítségével, majd a szerver folyamatosan küldhet eseményeket a kliens felé text/event-stream MIME típusban.

SSE jellemzöi

1. Egyirányú kommunikáció — Csak a szerver küldhet adatot a kliens felé.
2. Egyszerű implementáció — Nincs szükség komplex protokollokra (mint a WebSocket).
3. HTTP-alapú — Nem igényel külön portot, működik a meglévő HTTP(S) infrastruktúrán.
4. Automatikus újrakapcsolódás — Ha a kapcsolat megszakad, a kliens automatikusan újrapróbálkozik.
5. Támogatja az eseménytípusokat -– Lehetőség van különböző típusú események küldésére (pl. message, update).

MQTT

Az MQTT egy nyílt, könnyűsúlyú üzenetküldő protokoll, amelyet eredetileg a publiskerelőfizet (pub/sub) modellre terveztek az alacsony sávszélességű, magas késleltetésű hálózatok (pl. IoT eszközök) számára. Az MQTT-t ma széles körben használják az Ipari Internet of Things (IIoT), otthoni automatizálás, és valós idejű adatátviteli rendszerek területén. A protokollt az OASIS szabványosította, és a ISO/IEC 20922 szabvány részét képezi[13, 12].

Történet és fejlődés

– 1999: Az IBM és az Arcom (ma Eurotech) fejlesztette ki Andy Stanford-Clark és Arlen Nipper vezetésével, olajvezeték-figyelő rendszerekhez[14].

– 2014: Az MQTT v3.1.1 vált hivatalos OASIS szabvánnyá.

– 2019: Megjelent az MQTT v5, új funkciókkal (pl. üzenetlejárat, okokódok, megosztott előfizetések).

Működési elv (Publish-Subscribe modell)
Az MQTT központi broker segítségével működik, amely üzeneteket továbbít a kliensek között. A kommunikáció témakörök (topics) alapján történik. Kulcsfogalmak:

1. Közzétevő (Publisher): Üzeneteket küld egy témakörre (*publish*).
2. Előfizető (Subscriber): Feliratkozik egy témakörre, és fogadja az üzeneteket (*subscribe*).
3. Broker: Közvetíti az üzeneteket a megfelelő előfizetőknek.
4. Témakör (Topic): Hierarchikus útvonal (pl. *iot/sensor1/temperature*).

3.2.2. Long-Polling megvalósítása

Tekintve, hogy Long-Polling-ot még nem használtam meg kellett ismernem[8, 10]. Ezek mellett praktikusabb példákat is keresnem kellet, hogy jobban megértsem[9].

Implementálás után teszteltem a funkciót, eleinte jól működött, azonban amikor több feladatot is feltöltöttem az API összeomlott és az eszköz kommunikációs hibát dobott. A hiba gyökere az volt, hogy, míg a bejövő request elindította a folyamatot, addig egy másik ismét megtette ezt, így a kapcsolat a response elküldésével lezárult és nem tudta az API elküldeni a másikat. Ennek megoldása ként egy „jelenleg futó” másodlagos lista létrehozása volt szükséges.

3.1. táblázat. SSE és WebSocket összehasonlítás

Jellemző	**SSE**	**WebSocket**
Kommunikáció iránya	Egyirányú (szerver → kliens)	Kétirányú (duplex)
Protokoll	HTTP/HTTPS	WS (WebSocket), WSS (secure)
Adatformátum	Csak szöveg (text/event-stream)	Szöveg és bináris is
Komplexitás	Egyszerű (beépített böngésző API)	Bonyolultabb (külön könyvtár kell)
Újrakapcsolódás	Automatikus	Manuális kezelés szükséges
Alkalmazási terület	Valós idejű értesítések, frissítések	Chat, játékok, valós idejű interakció
Böngészőtámogatás	Modern böngészők (IE nem)	Szélesebb támogatás

```
var responses = []
var currentExecution = []
taskRouter.post('/addTask', async (req, res) => {
    let { dev_id, module_id, user_id, params, aditionalInfo }
       = req.body;
    let result = await addTask(dev_id, module_id, user_id,
       params, aditionalInfo);
    executeByIndex(checkForData(dev_id))
    res.send(result);
})

taskRouter.get('/getTasksByDeviceId', async (req, res) => {
    let { dev_id } = req.headers;
    var i = checkForData(dev_id)
    if (i > -1) {
        executeByIndex(i)
    }
    else {
        let result = await getTasksByDeviceId(dev_id);

        if (JSON.stringify(result) === '[]' ||
           getExec(dev_id) > -1) {
            responses.push({
                "dev_id": dev_id,
                "res": res
            })
        }
        else {
            res.send(result);
        }
    }
})

//...

async function executeByIndex(i) {
    if (i > -1) {
        var element = responses[i]
```

```
36          currentExecution.push(element['dev_id'])
37          var resp = element['res']
38          let result = await
               getTasksByDeviceId(element['dev_id']);
39          resp.send(result)
40          remove(currentExecution, element['dev_id'])
41          remove(responses, element['dev_id'], true, 'dev_id')
42      }
43  }
44
45  function checkForData(dev_id) {
46      return responses.findIndex((el) => el['dev_id'] == dev_id)
47  }
48
49  function remove(arr, data, isJSON = false, JSONIdentifier) {
50      if (isJSON) {
51          arr.splice(arr.findIndex((el) => el[JSONIdentifier]
               == data), 1)
52      }
53      else {
54          arr.splice(arr.findIndex((el) => el == data), 1)
55      }
56  }
57
58  function getExec(JSONIdentifier) {
59      return currentExecution.findIndex((el) => el ==
           JSONIdentifier)
60  }
```

Kódrészlet 3.1. Long-Polling és a hozzá tartozó metódusok.

4. fejezet

Frontend

Irodalomjegyzék

[1] A szekció a Blockchain Simplified és a DeepSeek alapján és felhasználásával készült. [Letöltve/látogatva:2025.03.21]

[2] `https://flutter.dev/` [Letöltve/látogatva:2025.03.23]

[3] Introduction to Flutter: `https://www.fullstack.com/labs/resources/blog/an-introduction-to-flutters-world` fordítva DeepSeek által. [Letöltve/látogatva:2025.03.23]

[4] MongoDB dokumentáció: `https://www.mongodb.com/docs/manual/` [Letöltve/látogatva:2025.03.06]

[5] MongoDB felhasználói fórum: `https://www.mongodb.com/community/forums/t/mongoose-querying-on-object-id-type/265185` [Letöltve/látogatva:2025.03.06]

[6] Server Sent Event ismertetők: `https://web.dev/articles/eventsource-basics`, `https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events` [Letöltve/látogatva:2025.03.07]

[7] MQTT honlapja: `https://mqtt.org/` [Letöltve/látogatva:2025.03.07]

[8] Long-Polling-ot ismertető oldal: `https://javascript.info/long-polling` [Letöltve/látogatva:2025.03.08]

[9] Long-Polling használata Expressel: `https://medium.com/@ignatovich.dm/implementing-long-polling-with-express-and-react-2cb965203128` [Letöltve/látogatva:2025.03.08]

[10] Long-Polling használata: `https://stackoverflow.com/questions/45853418/how-to-use-long-polling-in-native-javascript-and-node-js` [Letöltve/látogatva:2025.03.08]

[11] SSE implementálása és leírása: `https://html.spec.whatwg.org/multipage/server-sent-events.html` [Letöltve/látogatva:2025.03.08]

[12] OASIS MQTT v5 specifikáció: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html [Letöltve/látogatva:2025.03.08]

[13] ISO/IEC 20922: https://www.iso.org/standard/69466.html [Letöltve/látogatva:2025.03.08]

[14] IBM MQTT fehér könyv: https://www.ibm.com/docs/en/ibm-mq/8.0.0 [Letöltve/látogatva:2025.03.08]

[15] Kotlin Native oldala: https://kotlinlang.org/docs/native-overview.html [Letöltve/látogatva:2025.03.16]

[16] Kotlin Native kérdések oldal: https://discuss.kotlinlang.org/t/questions-about-kotlin-native/26042 [Letöltve/látogatva:2025.03.16]

[17] www.ISSI.com-on található PDF:https://www.issi.com/WW/pdf/Octal-Memory.pdf [Letöltve/látogatva:2025.03.18]

[18] Tarlogic cikk az ESP32 mikrokontrollerek Bluetooth sebezhetőségéről: https://www.tarlogic.com/news/hidden-feature-esp32-chip-infect-ot-devices/ [Letöltve/látogatva:2025.03.20]