Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Vizualizációs megoldás IoT adat elemző rendszerhez

Szakdolgozat

Készítette
Kunkli Richárd

Konzulens
dr. Simon Csaba

2020. december 3.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Kunkli Richárd*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot 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 autentikált 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.

Budapest, 2020. december 3.

Kunkli Richárd
hallgató

Kivonat

Adott egy tanszéken fejlesztett felhő alapú elosztott rendszer, melynek eszközei madárhangok azonosítására képesek. Ha a rendszer úgy észleli, hogy az egyik álatala vezérelt eszköz mikrofonja felvételén madárhang található, akkor riasztást kezdeményez az eszközön ezzel elijesztve a madarat ezáltal megóvva a növényzetet.

A rendszernek több kisebb komponense van, amelyek rengeteg adatot dolgoznak fel és nincs jelenleg egy olyan egységes grafikus felület ahol a rendszer teljes állapotát át lehetne tekinteni, ahol a feldolgozott adatokat vizualizálni lehetne.

A piacon létezik már több olyan szoftver csomag, amely hasonló problémákra próbál megoldást nyújtani, de ezek sem mindig tudják kielégíteni azokat a speciális igényeket, amelyek egy ilyen rendszernél felmerülnek.

Jelen szakdolgozat célja egy olyan vizualizációs megoldás bemutatása, amelynek segítségével a rendszer könnyedén áttekinthető és kezelhető. A tanszéki rendszer által kezelt eszközök a felületen is vezérelhetők és azok működéséről különböző statisztikákat felhasználva egyszerűen értelmezhető diagrammok generálódnak.

A backend megvalósítására az ASP.NET Core-t választottam, mely platformfüggetlen megoldást nyújt a web kérések kiszolgálására. A frontend-et a React.js használatával készítettem, mely segítségével egyszerűen és gyorsan lehet reszponzív felhasználói felületeket készíteni. Dolgozatomban bemutatom a tanszéken fejlesztett rendszert, a mikroszolgáltatások vizualizálásának alternatíváit, ismertetem az általam választott technológiákat és a készített alkalmazás felépítését.

Abstract

There is a department developed cloud-based distributed system whose devices are capable of identifying bird sounds. If the system detects a bird's voice on the recording of a microphone on one of the devices, it will trigger an alarm on the device scaring the bird away thereby protecting the vegetation.

The system has several smaller components that process a lot of data and currently there is no unified graphical user interface where the overall state of the system could be reviewed, where the processed data could be visualized.

There are already several software packages on the market that try to solve similar problems, however they aren't always able to meet the special needs that arise with such a system.

The purpose of this thesis is to present a visualization solution that allows the users to easily review and manage the system. The devices maintained by the department developed system can be controlled on the interface and easy-to-understand diagrams are generated using statistics about their operation.

I chose ASP.NET Core as the backend framework, which provides a platform-independent solution for serving web requests. The frontend was created using React.js, which allows for an easy and quick way to create responsive user interfaces. In my thesis I present the system developed at the department, the alternatives of visualization of microservices, I describe the technologies I have chosen and the structure of the application I have created.

1. fejezet

Bevezetés

Szőlőtulajdonosoknak éves szinten jelentős kárt okoznak a seregélyek, akik előszeretettel választják táplálékul a megtermelt szőlőt. Erre a problémára dolgoztak ki a tanszéken diáktársaim egy felhő alapú konténerizált rendszert, a Birdnetes-t mely a természetben elkelyezett eszközökkel kommunikál, azokat vezérli. Az eszközök bizonyos időközönként hangfelvételt készítenek a környezetükről, majd valamilyen formában elküldik ezeket a felvételeket a központi rendszernek, amely egy erre a célra kifejlesztett mesterséges intelligenciát használva eldönti a felvételről, hogy azon található-e seregély hang vagy sem. Ha igen akkor jelez a felvételt küldő eszköznek, hogy szólaltassa meg a riasztó berendezését, hogy elijessze a madarakat.

1.1. A probléma

A jelen rendszer használata során nincs vizuális visszacsatolás az esetleges riasztásokról azok gyakoriságáról és a rendszer állapotáról sem. Különböző diagnosztikai eszközök ugyan implementálva lettek mint például a logolás vagy a hiba bejelentés, de ezek használata nehézkes, nem kézenfekvő. Szükség van valamire amivel egy helyen és egyszerűen lehet kezelni és felügyelni a rendszer egyes elemeit.

1.2. A megoldás

A jelen szakdolgozat egy olyan webes alkalmazás elkészítését dokumentálja, melyel a felhasználók képesek a természetben elhelyezett eszközök állapotát vizsgálni, azokat akár ki és bekapcsolni igény szerint. Az egyes rendszer eseményeket vizsgálva a szoftver statisztikákat készít, melyeket különböző diagrammokon ábrázolok. Ilyen statisztikák például, hogy időben melyik eszköz mikor észlelt madár hangot, vagy hogy hány hang üzenet érkezik az eszközöktől másodpercenként.

1.3. A szakdolgozat felépítése

A szakdolgozatom első részében, a 2. fejezetben, bemutatom a Birdnetes felépítését, az egyes komponensek közötti kapcsolatokat és a technológiát, amire épült. A 3. fejezetben ismertetem a jelenleg az iparban is használt mikroszolgáltatás működését vizualizáló alternatívákat, majd a saját megoldásom tervezetét, az arra vonatkozó elvárásokat. A 4. fejezetben az alkalmazásom által használt technológiákat mutatom be, ezzel előkészítve az 5. és 6. fejezetet, ahol ismertetem a szerver- és kliensalkalmazások felépítését. A 7. és 8. fejezet az alkalmazás teszteléséről és telepítéséről szól. Az utolsó fejezetben értékelem a munkám eredményét, levonom a tapasztalatokat és bemutatok néhány továbbfejlesztési lehetőséget.

2. fejezet

A Birdnetes bemutatása

Ebben a fejezetben ismertetem a Birdnetes mikroszolgáltatás rendszerének architektúráját és az általa használt technológiákat. Részletesen kifejtem az alkalmazásom szempontjából fontos komponensek feladatát és működését.

2.1. Gyors elméleti összefoglaló

Ez a szakasz nem azt a célt szolgálja, hogy minnél részletesebb képet mutasson az itt leírt technológiákról. Arra sokkal jobb eszköz Pünkösdi Marcellnek és Torma Kristófnak, a Birdnetes alkótóinak TDK dolgozata[18]. Ez csupán egy rövid összefoglaló a Birdnetes működésének megértése szempontjából elengedhetetlen technológiákról és elvekről, hogy valamennyire érthetőbbek legyenek a fejezetben elhangzó kifejezések.

2.1.1. Cloud, felhő

A cloud lényegében annyit jelent, hogy a szervert, amin az alkalmazás fut, nem a fejlesztőnek kell üzemeltetnie, hanem valamilyen másik szervezet[1] által vannak karban tartva. Ez több okból is hasznos:

- **Olcsóbb**. Nem kell berendezéseket vásárolni, nincs üzemeltetési díj. Az egyetlen költség a bérlés, ami általában töredéke annak, amit akkor fizetnénk ha magunk csinálnánk az egészet.
- **Gyorsabb fejlesztés**. Az alkalmazás futtatására használt szervereket általában a fejlesztő nem látja, ezekkel nem kell foglalkoznia. Ha az alkalmazásnak hirtelen nagyobb erőforrás igénye lesz, a rendszer automatikusan skálázódik.
- **Nagyobb megbízhatóság**. Az ilyen szolgáltatást nyújtó szervezeteknek ez az egyik legnagyobb feladata. Az alkalmazás bárhol és bármikor elérhető.

[1] Ilyenek például a Microsoft Azure, az Amazon Web Services vagy a Google Cloud.

2.1.1.1. Mikroszolgáltatások

A mikroszolgáltatások nem sok mindenben különböznek egy általános szolgáltatástól. Ugyan úgy valamilyen kéréseket kiszolgáló egységek, legyen az web kérések kiszolgálása HTTP-n keresztül vagy akár parancssori utasítások feldolgozása. Az egyetlen fő különbség az a szolgáltatások felelősségköre. A mikroszolgáltatások fejlesztésénél a fejlesztők elsősorban arra törekednek, hogy egy komponensnek minnél kevesebb feladata és függősége legyen, ezzel megnő a tesztelhetőség és könyebb a skálázhatóság.

2.1.1.2. Konténerek

A konténer technikailag semmivel sem több mint egy Linux-on futó processz amelyre különböző korlátozásokat szabtak. Ilyen korlátozások lehetnek például, hogy a konténer nem látja a teljes fájlrendszert, annak csak egy kijelölt részét, megadható a konténer által használható processzor és memória igény vagy akár korlátozható az is, hogy a konténer hogyan használhatja a hálózatot. Léteznek eszközök, például a Docker[2], mely lehetővé teszi a fejlesztők számára az ilyen konténerek könnyed létrehozását és futtatását.

2.1.1.3. Kubernetes

A Kubernetes[10] az ilyen komplex konténerizált mikroszolgáltatás rendszerek menedzselésének könnyítését szolgálja. Kihasználja és ötvözi az imént említett technológiák előnyeit, hogy egy robosztus rendszert alkosson. Használatával felgyorsulhat és automatizált lehet az egyes konténerek telepítése, futtatása, de talán a legfőbb előnye, hogy segítségével könnyedén megoldható a rendszert ért terhelési igények szerinti dinamikus skálázódás. Azok a mikroszolgáltatások, amikre a rendszernek épp nincs szüksége, nem futnak, nem igényelnek erőforrást a szerveren, így nem kell utánnuk fizetni sem. Ezzel ellentétben, ha valamely szolgáltatás után hirtelen megnő az igény, akkor az könnyedén duplikálható.

2.1.2. MQTT

Az MQTT (Message Queue Telemetry Transport) az egy kliens-szerver publish/subscribe üzenetküldő protokoll. Könnyű implementálni és alacsony a sávszélesség igénye, mellyel tökéletes jelöltje a Machine to Machine (M2M), illetve az Internet of Things (IoT) kommunikáció megvalósítására. Működéséhez szükség van egy szerverre, amelynek feladata a beérkező üzenetek továbbküldése témák alapján. Egyes kliensek fel tudnak iratkozni bizonyos témákra, míg más kliensek publikálnak és a szerver levezényli a két fél között a kommunikációt.

2.1.3. Open API

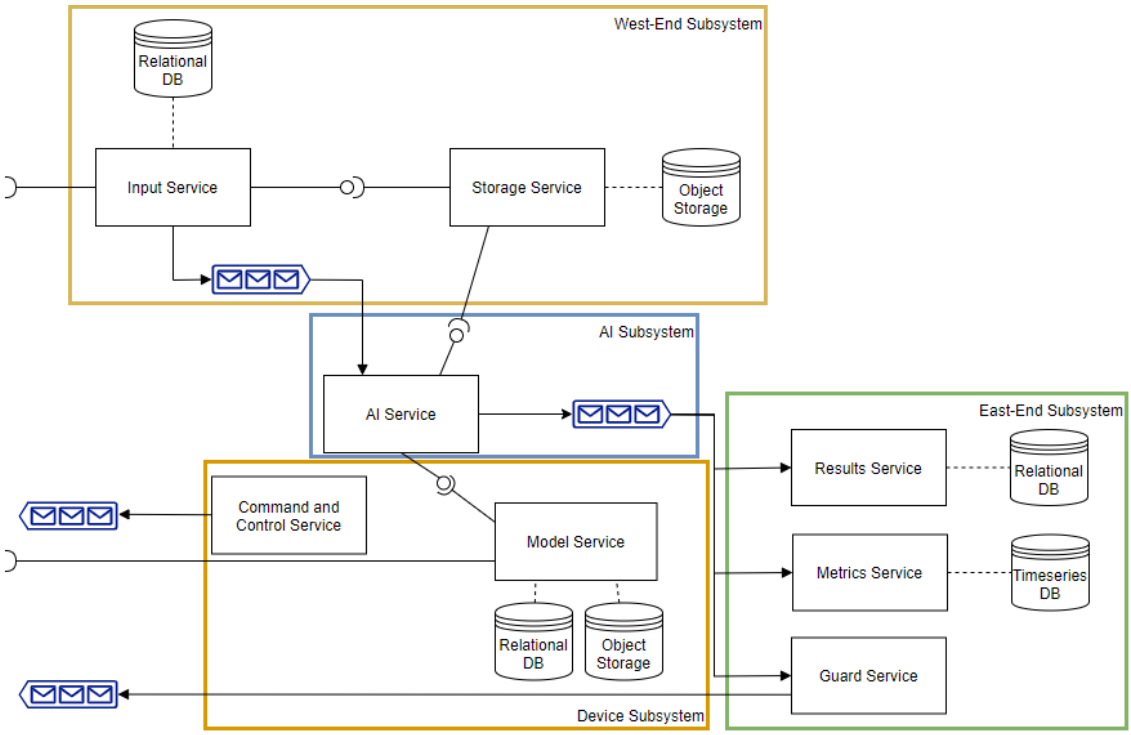
Az Open API egy nyilvános alkalmazás-programozási leíró, amely a fejlesztők számára hozzáférést biztosít egy másik alkalmazáshoz. Az API-k lírják és meghatározzák, hogy egy alkalmazás hogyan kommunikálhat egy másikkal, melyet használva a fejlesztők könnyedén képesek a kommunikációra képes kódot írni vagy generálni.

2.2. Rendszerszintű architektúra

A Birdnetes fejlesztése során kifejezetten fontos szerepe volt a mikroszolgáltatás alapú rendszerek elvei követésének. A rendszer egy Kubernetes klaszterben van telepítve és több kisebb komponensből áll, melyek egymás között a HTTP és az MQTT protokollok segítségével kommunikálnak. A rendszer összes szolgáltatásának van egy Open API leírója, melyet használva hamar volt egy olyan kódbázisom, amely képes volt a rendszerrel való kommunikációra.

2.2.1. Főbb komponensek

A 2.1-es ábrán láthatóak a rendszer komponensei, melyek mindegyike egy-egy mikroszolgáltatás. Az egymás mellett lévő kék levélborítékok az MQTT kommunikációt jelölik, amellyel például a természetben elhelyezett eszközök felé irányuló kommunikáció is történik. A következő alszakaszokban bemutatom az alkalmazásom szempontjából fontosabb komponenseket.



2.1. ábra. A Birdnetes rendszer architektúrája

2.2.1.1. IoT eszközök

Szőlőültetvényekben telepített eszközök, melyek adott időközönként publikálják állapotaikat egyéb metaadatokkal egy üzenetsoron. Emellett folyamatosan hangfelvételt készítenek a beépített mikrofonjaikkal, mely hangfelvételekről egy másik belső szenzor eldönti, hogy érdemes-e felküldeni a rendszerbe, ha igen, akkor egy másik üzenetsoron publikálják eze-

ket a hangfelvételeket. Tartalmaznak még egy hangszórót is, mely a madarak elijesztését szolgálja.

2.2.1.2. Input Service

A kihelyezett IoT eszközök által felvett hangfájlok ezen a komponensen keresztül érkeznek be a rendszerbe. Itt történik a hanganyaghoz tartozó metaadatok lementése az Input Service saját adatbázisába. Ilyenek például a beküldő eszköz azonosítója, a beérkezés dátuma vagy a hangüzenet rendszerszintű egyedi azonosítója. Amint a szolgáltatás a berékezett üzenettel kapcsolatban elvégezte az összes feladatát, publikál egy üzenetet az MQTT üzenetsorra a többi kliensnek feldolgozásra.

2.2.1.3. AI Service

Az AI Service példányai fogadják az Input Service-től érkező üzeneteket és elkezdik klasszifikálni az abban található hanganyagot. Meghatározzák, hogy a hanganyag mekkora valószínűséggel volt seregély hang vagy sem. Ennek eredményét a hangminta egyedi azonosítójával együtt publikálják egy másik üzenetsoron.

2.2.1.4. Guard Service

A Guard Service feliratkozik az AI Service által publikált üzenetek témájára és valamilyen valószínűségi kritérium alapján eldönti, hogy a hangminta tartalmaz-e seregély hangot. Ha igen, akkor az üzenetsoron küld egy riasztás parancsot a hanganyagot küldő eszköznek.

2.2.1.5. Command and Control Service

A Command and Control Service az előzőekkel ellentétben egyáltalán nem vesz részt a minták fogadásában, feldolgozásában vagy kezelésében. Felelősége az eszközök és azok szenzorai állapotának menedzselése és követése. Ezen keresztül lehet az egyes eszközöket ki- és bekapcsolni.

3. fejezet

Tervek és alternatívák

Ebben a fejezetben bemutatom a fejlesztés előtti állapotot, amikor még csak tervezgettük, hogy milyen is legyen az alkalmazás. Illetve bemutatok, néhány vizualizációs alternatívát, melyek jó iránymutatásként szólgálltak a fejlesztés során.

3.1. Tervezés

Az első dolgom az volt, hogy Kristóffal és Marcellel beültünk egy Teams[1]-en tartott gyűlésre, ahol elmagyarázták nagyvonalakban, hogy hogyan is működik a rendszer, mik az egyes kompnensek feladatai. Ezek után az előttem álló fejlesztésre váró alkalmazás részleteit beszéltük meg, az elvárt igényeket azzal kapcsolatban. Itt rögtön több ötlet is felmerült, melyek közül a legkiemelkedőbbek:

- **Hőtérkép**. Hasznos lenne egy olyan felület, ahol az eszközök GPS koordinátái és a seregély detektálást jelző üzenetek alapján, meg lehetne jeleníteni a seregélyek hozzávetőleges előfordulásának helyeit és gyakoriságát egy térképen, hőtérképes formában.
- **Eszköz állapotok**. Jelenleg a Command and Control mikroszolgáltatás felé indított kéréseken kívül, nincs lehetőség a kihelyezett eszközök állapotának vizsgálatára. Szükség lenne egy olyan felületre, ahol ezek állapotai láthatóak, esetleg dinamikusan is frissülnek.
- **Diagrammok**. A hőtérképen kívül egyéb olyan diagrammok is hasznosak lehetnek, ahol látható például, hogy melyik eszköz melyik percben észlelt madárhangot vagy, hogy egy eszköz összesen hány madárhangot észelt. Minnél több információ, annál jobb.

Ezeken kívül fontos követelmény volt még, hogy az alkalmazásom futtatható legyen Linux környezetben is, hogy az telepíthető legyen a Birdnetes Kubernetes[10] klaszterébe.

Az alkalmazásom kapott egy nevet is, mely a Birdnetes-t és az említett hőtérképes ötletet ötvözve Birdmap lett.

[1]Microsoft Teams: Csevegő és gyülekezés tartó alkalmazás.

3.2. Alternatívák

Az imént vázolt igények kielégítésére rengeteg kiforrott megoldás létezik már, melyek jó példát mutattak a saját alkalmazásom fejlesztése során.

3.2.1. Grafana

A Grafana[5] az egy nyílt forráskódú platformfüggetlen vizualizációs web alkalmazás. Egy támogatott adatbázishoz csatlakoztatva különféle interaktív gráfokat és diagrammokat generál. A testreszabhatóság maximalizásának érdekében különböző, akár harmadik fél által készített, bővítmények használatát is támogatja, melyekkel új adatforrások és panel típusok integrálhatók. A 3.1-es ábra egy jó példa arra, hogy hogyan néz ki egy általános Grafana felület.

3.1. ábra. A Grafana demo oldalának, a `https://play.grafana.org`-nak a felülete

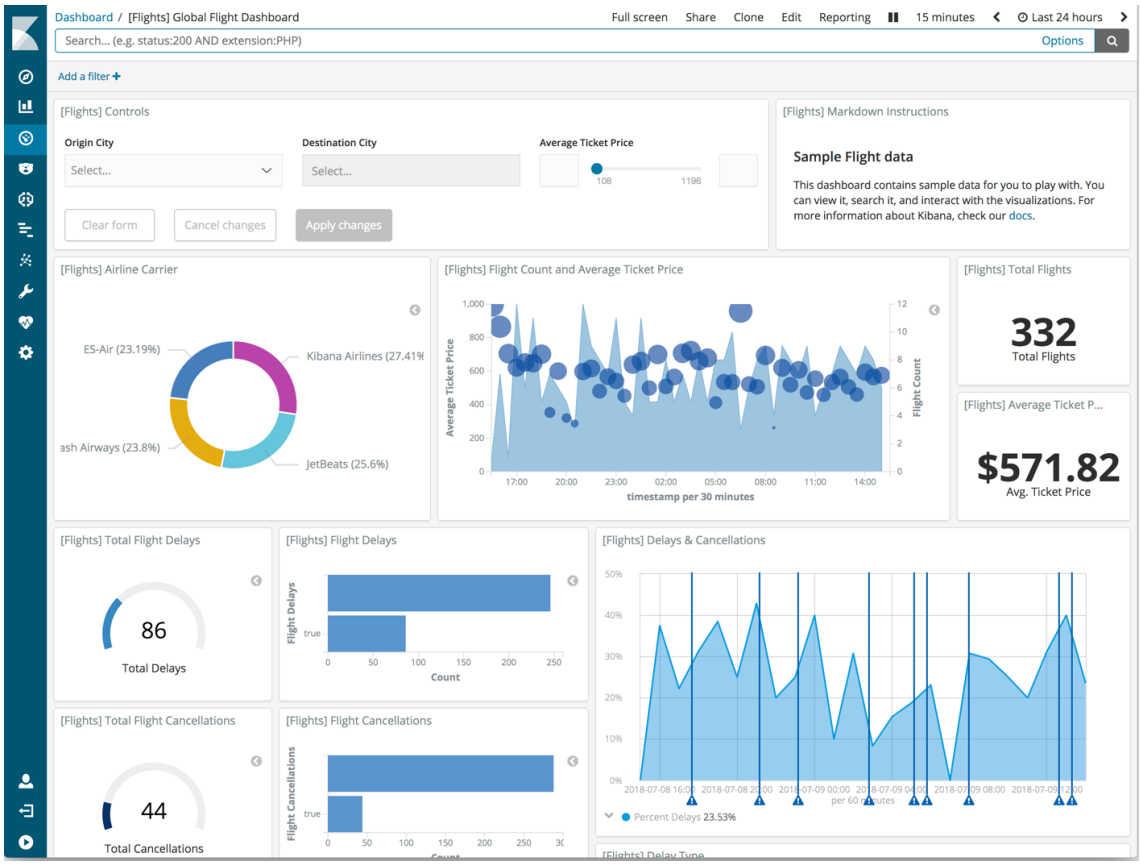
3.2.2. Kibana

A Kibana[8] jelentősen hasonlít a Grafanához, azonban amíg a utóbbit inkább az időben változó metrikák vizualizálására használják például processzor leterheltség vagy memória használat, addig az előbbit elsődlegesen az Elasticsearchadatok, főként napló bejegyzések, analizálására használják.

3.2.3. Kubernetes Dashboard (Web UI)

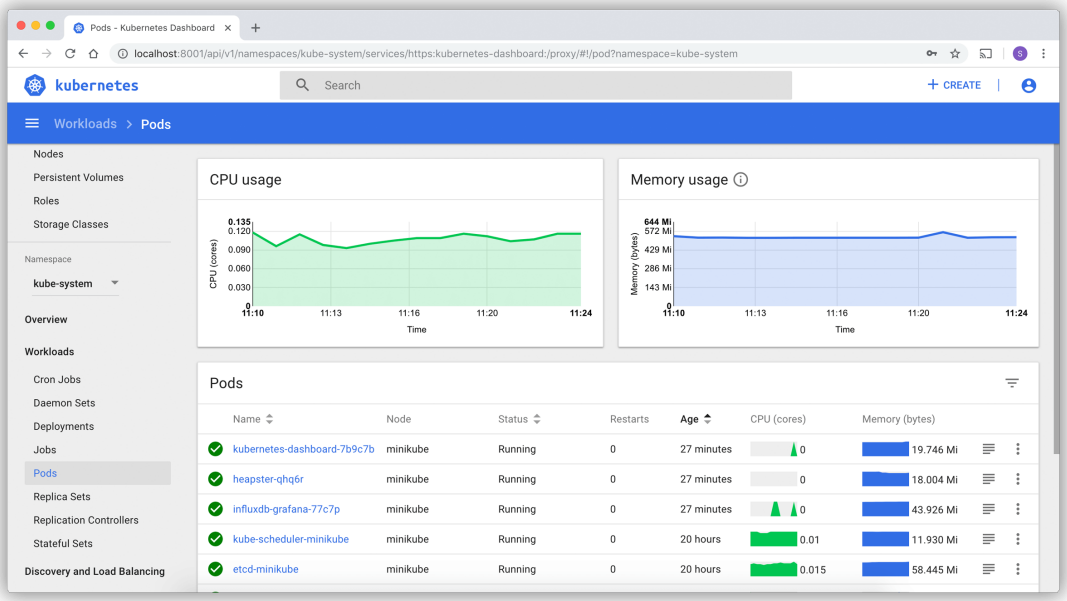
A Kubernetes Dashboard[9] elsősorban nem a különböző adatok vizualizálását szolgálja, inkább a klaszter menedzselését próbálja egyszerűbbé és jobban áttekinthetővé tenni.

[1]Ingyenes és nyílt forráskódú index alapú keresőmotor



3.2. ábra. Egy példa a Kibana kezelőfelületére

Azonban egy jó példa arra, hogy egy rendszer webes kezelőfelületének, milyennek is kell lennie.



3.3. ábra. A Kubernetes Dashboard felülete

4. fejezet

Használt technológiák

Ezzel a fejezettel az a célom, hogy ismertessem a fejlesztés során, illetve az alkalmazásom által használt technológiákat, hogy a következő fejezetekben alapozni tudjak ezeknek az ismeretére.

4.1. A fejlesztési folyamat technológiái

Ebben a szakaszban azokat az eszközöket, alkalmazásokat és fejlesztőkörnyezeteket mutatom be, melyeket a fejlesztés során, a fejlesztéshez használtam.

4.1.1. Git

A Git[3] egy verziókezelő rendszer. Használatával a felhasználó le tudja menteni egy adott fájlrendszerben található fájlok állapotát. Megkönnyíti az egy projekten dolgozó programozók közötti kooperációt. Manapság lassan elképzelhetetlen a fejlesztés valamilyen verziókezelő használata nélkül.

4.1.2. Trello

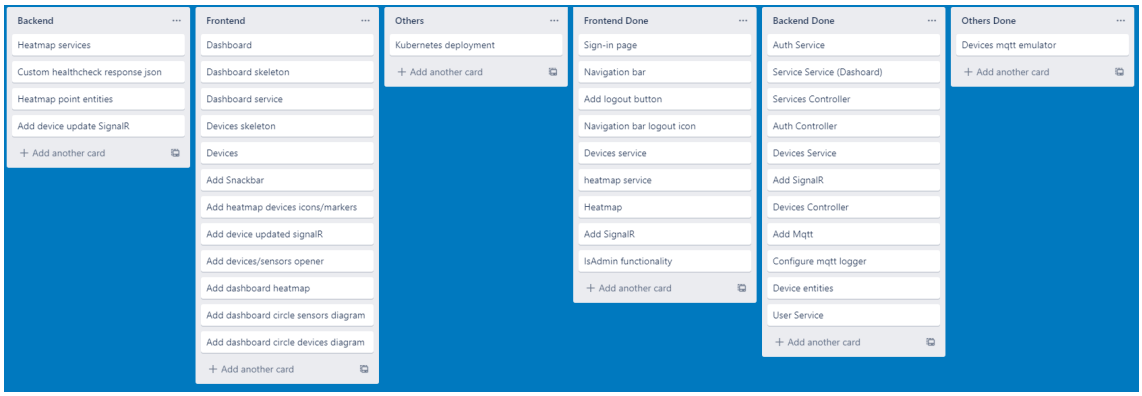
A Trello[20] egy webes lista készítő és kezelő alkalmazás. Azért használtam a fejlesztés során, mert szerettem volna egy helyet, ami tükrözi a fejlesztés állapotát, ahova le tudom írni az alkalmazással kapcsolatos ötleteimet. Különböző listákban tároltam a fejlesztésre váró és a kész feladatokat szerver, kliens és egyéb szerint.

4.1.3. Visual Studio

A Visual Studio[14] a Microsoft fejlesztőkörnyezete. Jól alkalmazható a .NET keretrendszer technológiáival, ezért ezt használtam a szerveroldal fejlesztése során.

4.1.4. Visual Studio Code

Egy másik Microsoft termék, viszont a fentivel ellentétben a Visual Studio Code[13] inkább szövegszerkeztő, mint fejlesztőkörnyezet. Ennek köszönhetően jelentősen gyorsabb és egyszerűbb a használata. Különféle bővítmények használatával nagyon jó program nyelv



4.1. ábra. Egy példa állapot a Trello felületére a fejlesztés során

támogatottságot lehet elérni. Többek között ezen okok miatt preferáltam a kliensoldal fejlesztésére.

4.2. Backend technológiák

Ebben a szakaszban a szerveroldal megvalósítására használt .NET technológiákat mutatom be. A választásom több ok miatt esett a .NET keretrendszer használatára.

Egyrészt úgy gondoltam, hogy az alkalmazásom fajsúlyosabb részét inkább a kliensoldal fogja képezni ezért, hogy arra több energiát tudjak fordítani, valami olyat választottam, amivel már foglalkoztam korábban, amivel gyorsabban és rutinosabban megy a fejlesztés.

Másrészt nemrég jelent meg a .NET új 5-ös verziója, melynek használatával jelentős teljesítmény javulást ígértek több területen is, és úgy gondoltam, hogy ez a projekt tökéletes lenne ennek próbatételére.

Mindemellett a .NET teljesen platformüggetlen, mely az egyik legfontosabb követelmény volt az alkalmazással szemben.

4.2.1. ASP.NET Core

Az ASP.NET Core a .NET család ingyenes, nyílt forráskódú webes keretrendszere. Gyors és moduláris fejlesztést tesz lehetővé, mely főként a NuGet csomagoknak köszönhető. Használatána egyik előnye, hogy ugyan az a C# kód tud futni a szerver éa a kliens oldalon, de támogat más kliens oldali keretrendszereket is, mint például az Angular-t, a Vue.js-t vagy a React.js-t.

4.2.2. Entity Framework Core

Az Entity Framework Core (röviden EF Core) egy objektum-relációs leképező keretrendszer a .NET-hez. Az adatbázissal való kommunikációt könnyítését szolgálja. Használatával C#-ban lehet adatbázis lekérdezéseket írni a LINQ (Language-Integrated Query) segítségével.

4.2.3. JSON Web Token

Az autorizációt többféleképpen meg lehet oldani egy alkalmazás szempontjából. Az egyik ilyen megoldás a JSON Web Token-ek (röviden JWT) [7] használata, ami nem más mint egy szabvány, mely módszert ad a felek közötti információ biztonságos továbbítására JSON objektumokkal. Ezen objektumok adatokat tárolhatnak a felhasználóról például a neve vagy a szerepe, melyek segítségével a szerver eldöntheti, hogy van-e jogosultsága hívni az adott végpontot.

A Microsoft-nak van egy beépített szoftvercsomagja, mellyel ilyen tokeneket lehet készíteni és validálni. A szerveroldal jogosultság kezelését ezzel a csomaggal oldottam meg.

4.2.4. SignalR

A SignalR egy .NET szoftvercsomag, mely lehetővé teszi a szerveroldal számára a kliensekkel való aszinkron kommunikációt. A szerver valós időben tud értesítéseket küldeni a kliensek számára, amelyek feliratkoztak az ilyen eseményekre.

4.2.5. MQTT.NET

Az MQTT.NET is egy .NET szoftvercsomag, mely a Birdnetes által is használt, a 2.1.2-es alfejezetben bemutatott MQTT kommunikáció C# nyelvű megvalósítását szolgálja.

4.2.6. NLog

A szerveroldali naplózás megvalósítására több szoftvercsomag is létezik. Az NLog[15]-ot választottam, egyrészt mert egyszerű a használata, másrészt mert már használtam korábban.

4.3. Frontend technológiák

Ebben a szakaszban a kliensoldalon használt technológiákat mutatom be. Választásomnál fő motiváció az volt, hogy szerettem volna valami újat kipróbálni, aminek nincs köze a .NET keretrendszerhez.

4.3.1. React.js

A React.js[17] egy JavaScript szoftvercsomag, melyet webes felületek fejlesztésére használnak. Fő építő elemei a komponensek, melyek elszeparált újrafelhasználható felület egységek. Használatának egyik előnye, hogy automatizált az állapot kezelés, tehát ha változik egy komponens állapota, akkor a React újra-rendereli azt.

4.3.2. Material UI

A Material[11] elsősorban egy kezelőfelület tervezési útmutató a Google által, melyet követve szép és minőségi felületeket lehet készíteni.

A Material UI[12] egy szoftvercsomag, mely ezeket az útmutatásokat követő egyszerű React komponenseket tartalmaz. Alkalmazásával könnyő esztétikus felhasználói felületeket készíteni, minimalizált a CSS használattal.

4.3.3. Apexcharts

Az Apexcharts[1] egy nyílt forráskódú JavaScript szoftvercsomag, amellyel könnyen konfigurálható, modern kinézetű diagrammokat lehet készíteni. Sokféle kliensoldali (és szerveroldali) technológiát támogat, köztük a React-et is. A kezelőfelületen található vizualizációk szinte összes elemét ennek használatával csináltam.

4.3.4. Google Maps Api

A Google szinte összes termékének van API-ja, ami lehetővé teszi a programozók számára, hogy integrálják ezeket saját alkalmazásaikban. A Google Maps sincs másképp és mivel ennek interfésze külön támogatja a hőtérképes réteg használatát is, nem gondoltam, hogy ettől jobb eszközt tudnék találni a feladat megvalósítására.

A Google Maps API-t, ami alapvetően csak egy JavaScript csomag, rengetegen újracsomagolják, hogy különböző részét, különböző keretrendszerekben is lehessen használni. Ezek közül én a Google Map React[4]-et választottam, egyrészt mert támogatja a hőtérképes réteg használatát, másrészt mert lehetővé teszi a térképen való React komponensek renderelését az alapértelmezett markerek helyett.

5. fejezet

Szerver oldal

Ebben a fejezetben bemutatom a szerveroldal architektúráját, felépítését. Ismertetem a különböző szoftver komponensek feladatát.

5.1. Architektúra

A szerveroldal fejlesztésénél a háromrétegű architektúrát alkalmaztam, melynek lényege, hogy az alkalmazást logikailag három elkülönülő részre bontjuk:

- **Adat elérési réteg**. Ez a rész felel a tárolt entitások modell definícióiért, illetve azoknak a kiolvasásáért, tárolásáért egy adatbázisból vagy fájlrendszerből.
- **Megjelenítési réteg**. Ezen réteg feladata a kliensoldal közvetlek kiszolgálása. Bármilyen irányú kommunikáció a kliensek felé ezen a rétegen keresztül történik.
- **Üzleti logikai réteg**. Minden ami nem a közvetlen kommunikációért, megjelenítésért vagy adat elérésért, tárolásért felel, az ide kerül. A fenti két réteg között helyezkedik el és feladata a különböző folyamatok értékelése és futtatása, valamint az adatok feldolgozása.

Az ASP.NET Core beépítetten támogatja a dependency injection-t, mely a `Startup` osztály `ConfigureServices` metódusával konfigurálható. Én minden rétegbe tettem egy ilyen `Startup` osztályt, hogy azok feleljenek a saját szolgáltatásaik konfigurálásáért és regisztrálásáért.

5.2. Adat elérési réteg

Az adatelérést az Entity Framework Core segítségével oldottam meg. Telepítettem egy MSSQL adatbázis szervert a számítógépemre, melynek csatlakozási paramétereivel a `Startup` osztályban felkonfigurálom az EF Core által nyújtott `DbContext` saját leszármazott változatát. Így csak az entitások elkészítése és azok alapértelmezett értékeinek az adatbázisba való feltöltése marad hátra.

5.2.1. Entitások

Mivel az adatok nagy részét külső szolgáltatások fogják nyújtani, így lokálisan összesen két entitás létrehozására volt szükség. Az egyik a `User`, mely az alkalmazás felhasználóinak adatait tárolja. A másik a `Service`, mely a külső szolgáltatások adatainak tárolását szolgálja, amelyeket azért tárolok az adatbázisban és nem mondjuk a konfigurációs fájlban, mert szerettem volna, hogyha a kezelőfelületen lehetne őket szerkeszteni, törölni.

```
public record User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public byte[] PasswordHash { get; set; }
    public byte[] PasswordSalt { get; set; }

    public Roles Role { get; set; }

    public bool IsFromConfig { get; set; }
}

public record Service
{
    public int Id { get; set; }
    public string Name { get; set; }
    public Uri Uri { get; set; }

    public bool IsFromConfig { get; set; }
}
```

5.1. lista. A User és a Service modell

Az alkalmazás használata szempontjából a felhasználók két csoportba oszlanak. Vannak adminisztrátor és sima felhasználók, utóbbi csak az adatok olvasására, míg előbb azok módosítására is jogosult. A `Role` mező ennek a megkülönböztetsnek a jelzője.

5.2.2. Seedelés

Az alkalmazás konfigurációs fájljából meg lehet adni alapértelmezett felhasználókat és szolgáltatásokat. Ezeknek megkülönböztetésére szolgál az entitások `IsFromConfig` mezője. A szerver indítása legelején, megvizsgálja, hogy létezik-e az adatbázis és ha igen kitöröl minden olyan entitást ahol az `IsFromConfig` mező igaz. Majd hozzáadja az újonnan beolvasott értékeket.

5.3. Üzleti logikai réteg

Ebben a rétegben található meg a szerver legtöbb szolgáltatása. It vannak implementálva a Birdnetes Command and Control és Input komponensekkel kommunikáló szolgáltatások is, melyeket azok OpenAPI leírói alapján az NSwag[16] alkalmazással generáltam. Az OpenAPI a klienseken kívül definiálja még az azok által használt modelleket is. A Command and Control által használt `Device` modell tartalmazza annak egyedi azonosítóját, státuszát, koordinátáit és a használt szenzorok listáját, melyeknek szintén van egy modellje `Sensor` néven. Ennek szintén van azonosítója és státusza. Az Input szolgáltatásnak

is van saját modellje, amely a hangüzenetek metaadatait reprezentálja. Többek között tartalmazza a kihelyezett eszköz egyedi azonosítóját és a hangüzenet keltének dátumát.

Ugyan itt található meg a `User` és `Service` entitások létrehozásáért, olvasásáért, szerkesztéséért és törléséért felelős szolgáltatások is. Valamint itt található még az autentikációért felelős szolgáltatás is. A felhasználók jelszavainak tárolására a HMAC (Hash-based Message Authentication Code) algorithmust, pontosabban annak a `HMACSHA512`[6] C# implementációját használtam.

Minden jelszóhoz generálok egy egyedi kulcsot és azzal egy hash-t, majd ezeket tárolom a `User` modell `PasswordSalt` és `PasswordHash` mezőiben. Amikor egy felhasználó be akar jelentkezni először megvizsgálom, hogy egyáltalán létezik-e az adatbázisban az adott nevű felhasználó, ha igen, akkor a megadott jelszóból az imént említett folyamattal generált kulcsot és hash-t összehasonlítom az adatbázisban tárolttal.

5.4. Megjelenítési réteg

A fejezet elején említett `Startup` osztály ebben a rétegben található, itt kerülnek az egyes szolgáltatások regisztrálásra. Itt történik a 5.2.2 fejezetben leírt adatbázis seedelése is.

Többek között a naplózás is itt kerül inicializálásra, mely az NLog saját konfigurációs fájljával történik. Meg lehet adni különböző szűrőket és kimeneteket, amellyel szelektálni lehet, hogy az egyes naplózott események hova kerüljenek. Például az MQTT szolgáltalás napló bejegyzéseit a 5.2 lista alapján szűrtem. Minden `Debug` szintől nagyobb és `Error` szinttől kisebb bejegyzés, mely tartalmazza az `Mqtt` kulcsszót az `mqttFile` azonosítójú fájlba kerül.

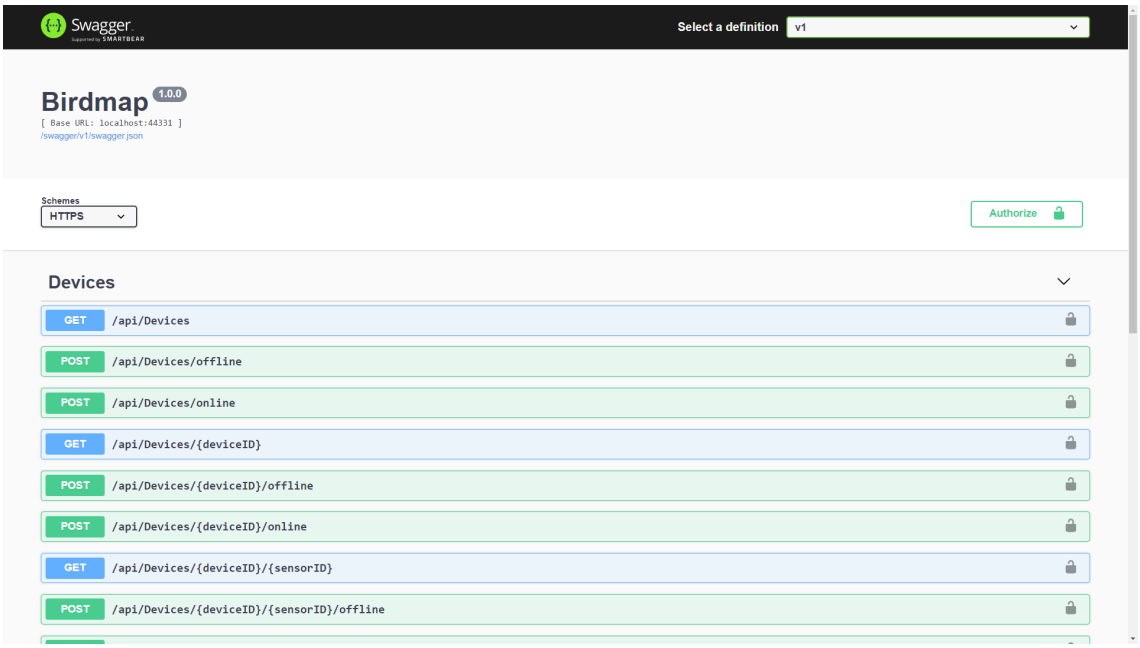
```
<targets>
    ...
    <target xsi:type="File" name="mqttFile" fileName="${basedir}Logs/birdmap-mqtt-${shortdate}.log"
        layout="..." />
    ...
</targets>

<rules>
    ...
    <logger name="*.*Mqtt*.*" minlevel="Trace" maxlevel="Warning" writeTo="mqttFile" final="true"/>
    ...
</rules>
```

5.2. lista. Az NLog.config fájl egy részlete

A `Startup` osztály másik metódusa a `Configure`, mellyel a HTTP kérések csővezetéke konfigurálható. Azaz, hogy egy kérés-t milyen sorrendben dolgozzák fel a regisztrált szolgáltatások. A szerveroldali kivételkezelésre szánt szolgáltatás, az `ExceptionHandlerMiddleware` is itt van használva, amely elkap minden kivételt, amit a csővezeték további részei dobtak és JSON formátumban visszaadja azokat a kliensnek.

Továbbá az NSwag[16] szoftvercsomag segítségével regisztrálok egy szolgáltatást, mely a szerveroldalon található kontrollereket felhasználva generál egy OpenAPI specifikációt és annak egy Swagger UI[19] felületet, ahol a végpontok kipróbálhatóak, tesztelhetőek kliensoldal nélkül is.



5.1. ábra. Az alkalmazásom Swagger felülete

5.4.1. Kommunikációs Szolgáltatások

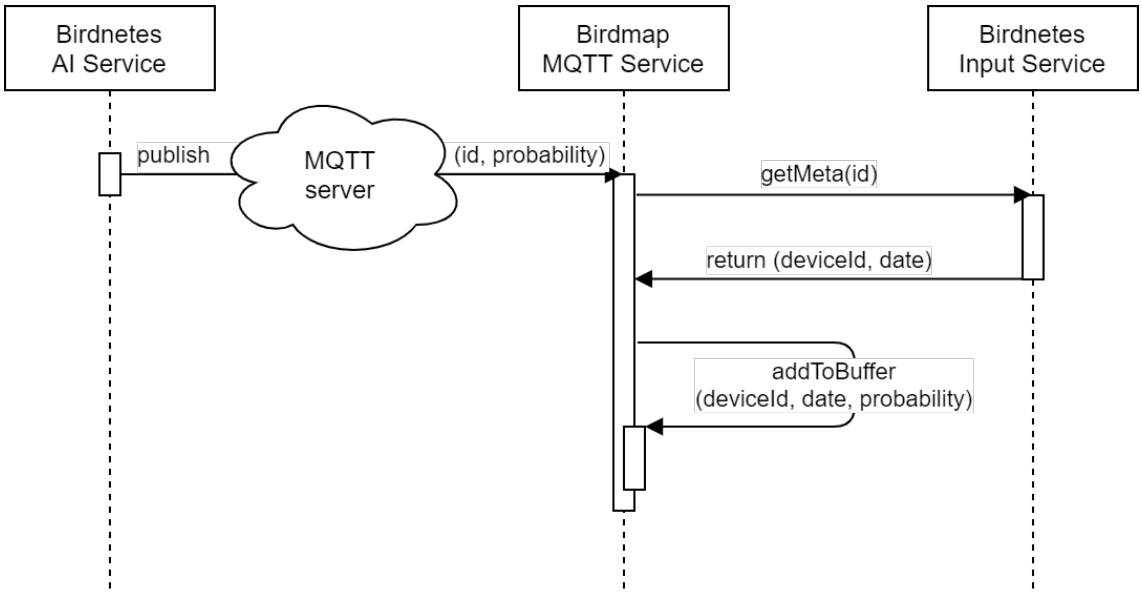
A kliensoldal frissítésére több megoldás is létezik. Például bizonyos időközönként lehetne kéréseket indítani a szerver felé a friss adatok megszerzéséért. Egy másik megoldás a SignalR használata, amellyel a klienseket eseményvezérelten lehet értesíteni, megvalósítja a kétoldalú kommunikációt. Így a kliensek csak akkor indítanak kéréseket amikor az adat tényleg változott. Ezzel a technológiával oldottam meg például, hogy az eszközök állapotainak változására frissüljön a felület.

Egy másik szerveroldalon használt szolgáltatás a Birdnetes MQTT kommunikációért felelős szolgáltatás, mely felregisztrál a 2.2.1.3-as alfejezetben bemutatott AI Service által publikált üzenetekre. Ezekben az üzenetekben található a hanganyagok egyedi azonosítója, illetve azok seregélytől való származásának valószínűsége. Ha a szolgáltatás kap egy ilyen üzenetet akkor lekérdezi a 2.2.1.2-es alfejezetben bemutatott Input Service-től a hanganyag azonosítójához tartozó metaadatokat. Ezekből felhasználva a kihelyezett eszköz azonosítóját, a hanganyag beérkezésének dátumát és az említett valószínűséget új üzenetek készülnek, melyeket egy pufferben tárolódnak. Ezt a folyamatot a 5.2-es ábra szemlélteti.

A puffer tartalmát másodperces gyakorisággal elküldöm a klienseknek a SignalR segítségével. Azért van szükség a puffer használatára, mert az MQTT-n érkezett üzenetek gyakorisága akár miliszekundum nagyságrendű is lehet. Míg a szerver képes is az üzeneteket feldolgozni, ha ezeket rögtön tovább küldeném a kliensek felé, azok nem biztos, hogy képesek lennének rá.

5.4.2. Kontrollerek

A kontrollerek határozzák meg, hogy a szerveroldalon milyen végpontokat, milyen paraméterekkel lehet meghívni, ahhoz milyen jogosultságok kellenek. A jogosultságok kezelését



5.2. ábra. A Birdmap MQTT szolgáltatásának szekvenciája

a JSON Web Token-ekkel oldottam meg. A fejlasználó bejelentkezéskor kap egy ilyen token-t, amelyben tárolom a hozzá tartozó szerepet. A 5.3-as listában látszik, hogy hogyan használom ezeket a szerepeket. A `DevicesController` végpontjait alapértelmezetten `User` és `Admin` jogosultságú felhasználó hívhatja, az "online" végpontot azonban csak `Admin` jogosultságú. Hasonló képpen oldottam meg ezt a többi kontrollernél is. A `User` felhasználók csak olyan végpontokat hívhat, mely kizárolag az állapotok olvasásával jár. Az `Admin` felhasználók hívhatnak bármilyen végpontot.

```
[Authorize(Roles = "User, Admin")]
[ApiController]
[Route("api/[controller]")]
public class DevicesController : ControllerBase
{
    [Authorize(Roles = "Admin")]
    [HttpPost, Route("online")]
    public async Task<IActionResult> Onlineall()
    {
        ...
    }

    ...
}
```

5.3. lista. Az eszköz kontroller és annak "online" végpontja

Controllersw Dtos mapper

Irodalomjegyzék

[1] Az apexcharts hivatalos oldala. URL https://apexcharts.com/.

[2] A docker hivatalos oldala. URL https://www.docker.com.

[3] A git hivatalos oldala. URL https://git-scm.com/.

[4] A google map react hivatalos oldala.
URL https://www.npmjs.com/package/google-map-react.

[5] A grafana hivatalos oldala. URL https://grafana.com/.

[6] Az hmacsha512 dokumentációja. URL https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.hmacsha512.

[7] Az json web token hivatalos oldala. URL https://jwt.io/introduction/.

[8] A kibana hivatalos oldala. URL https://www.elastic.co/kibana.

[9] A kubernetes dashboard hivatalos oldala. URL https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/.

[10] A kubernetes hivatalos oldala. URL https://kubernetes.io.

[11] A material hivatalos oldala. URL https://material.io/.

[12] A material ui hivatalos oldala. URL https://material-ui.com/.

[13] A microsoft visual studio code hivatalos oldala.
URL https://code.visualstudio.com/.

[14] A microsoft visual studio hivatalos oldala.
URL https://visualstudio.microsoft.com/.

[15] Az nlog hivatalos oldala. URL https://nlog-project.org/.

[16] Az nswag github oldala. URL https://github.com/RicoSuter/NSwag.

[17] A react.js hivatalos oldala. URL https://reactjs.org/.

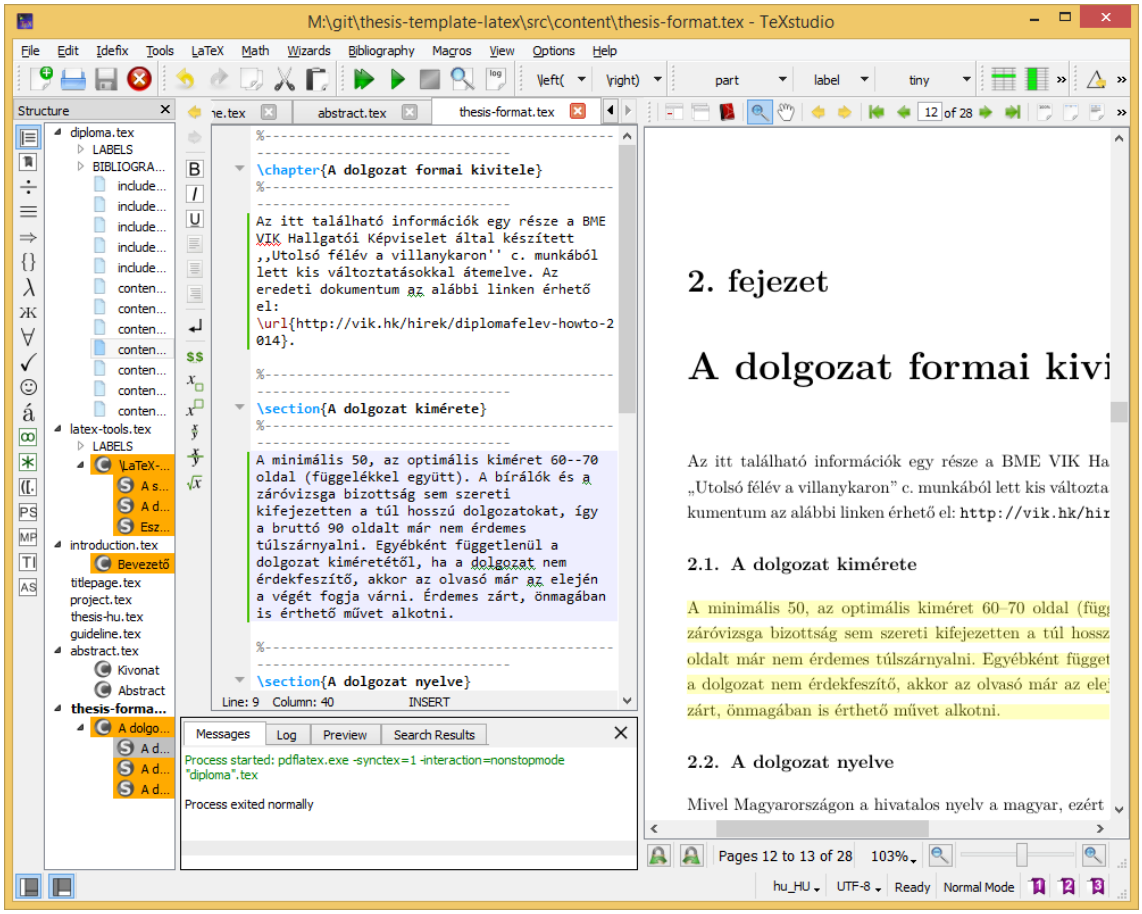
[18] Torma Kristóf és Pünkösdi Marcell: Madárhang azonosító és riasztó felhő-natív rendszer, 2020.

[19] A swagger ui hivatalos oldala. URL https://swagger.io/tools/swagger-ui/.

[20] A trello hivatalos oldala. URL https://trello.com.

Függelék

F.1. A TeXstudio felülete



F.1.1. ábra. A TeXstudio LaTeX-szerkesztő.

F.2. Válasz az „Élet, a világmindenség, meg minden” kérdésére

A Pitagorasz-tételből levezetve

$$c^2 = a^2 + b^2 = 42. \tag{F.2.1}$$

A Faraday-indukciós törvényből levezetve

$$\text{rot}\, E = -\frac{dB}{dt} \qquad \longrightarrow \qquad U_i = \oint\limits_{\mathbf{L}} \mathbf{Edl} = -\frac{d}{dt} \int\limits_{A} \mathbf{Bda} = 42. \tag{F.2.2}$$