



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 9.

Tartalomjegyzék

Kivonat	i
Abstract	ii
1. Bevezetés	1
1.1. Probléma	1
1.2. Megoldás	1
1.3. A szakdolgozat felépítése	2
2. A vizualizálni kívánt rendszer bemutatása	3
2.1. Elméleti összefoglaló	3
2.1.1. Cloud, felhőalapú rendszerek	3
2.1.1.1. Mikroszolgáltatások	4
2.1.1.2. Konténerek	4
2.1.1.3. Kubernetes	4
2.1.2. MQTT	4
2.1.3. OpenAPI	4
2.2. Rendszerszintű architektúra	5
2.2.1. Főbb komponensek	5
2.2.1.1. IoT eszközök	5
2.2.1.2. Input Service	6
2.2.1.3. AI Service	6
2.2.1.4. Guard Service	6
2.2.1.5. Command and Control Service	6
3. Tervek és alternatívák	7
3.1. Tervezés	7
3.2. Alternatívák	8
3.2.1. Grafana	8
3.2.2. Kibana	8
3.2.3. Kubernetes Dashboard (Web UI)	8
4. Használt technológiák	10
4.1. A fejlesztési folyamat technológiái	10

4.1.1.	Git	10
4.1.2.	Trello	10
4.1.3.	Visual Studio	10
4.1.4.	Visual Studio Code	10
4.2.	Backend technológiák	11
4.2.1.	ASP.NET Core	11
4.2.2.	Entity Framework Core	11
4.2.3.	JSON Web Token	12
4.2.4.	SignalR	12
4.2.5.	MQTT.NET	12
4.2.6.	NLog	12
4.3.	Frontend technológiák	12
4.3.1.	React.js	12
4.3.2.	Material UI	13
4.3.3.	Apexcharts	13
4.3.4.	Google Maps Api	13
5.	Szerver oldal	14
5.1.	Architektúra	14
5.2.	Adatelérési réteg	14
5.2.1.	Entitások	15
5.2.2.	Seedelés	15
5.3.	Üzleti logikai réteg	15
5.3.1.	Kommunikációs Szolgáltatások	16
5.4.	Megjelenítési réteg	17
5.4.1.	Swagger	18
5.4.2.	Kontrollerek	18
6.	Kliens oldal	21
6.1.	Architektúra	21
6.2.	Kommunikáció a szerveroldallal	22
6.3.	Komponensek	23
6.3.1.	Navigáció	23
6.3.2.	Login	23
6.3.3.	Logs	24
6.3.4.	Eszközállapot- és hangüzenet-kezelő szolgáltatás	24
6.3.5.	Dashboard	25
6.3.5.1.	Külső szolgáltatások	25
6.3.5.2.	Eszközök és szenzorok állapota	26
6.3.5.3.	Hőtérkép diagramok	27
6.3.5.4.	Riasztás számláló	28
6.3.5.5.	Üzenetek gyakorisága	29

6.3.6. Devices	30
6.3.7. Heatmap	31
7. Tesztkörnyezet	32
7.1. Helyettesítő szolgáltatások	32
7.2. MQTT tesztalkalmazás	33
8. Docker image készítés	34
9. Értékelés	35
9.1. Továbbfejlesztési lehetőségek	35
Irodalomjegyzék	36

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 9.

Kunkli Richárd
hallgató

Kivonat

Adott egy tanszéken fejlesztett felhőalapú elosztott rendszer, melynek eszközei madárhangok azonosítására képesek. Ha a rendszer úgy észleli, hogy az egyik általa 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 diagramok 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 táplálékkul a megtermelt szőlőt. Erre a problémára dolgoztak ki a tanszéken diáktársaim egy felhőalapú konténerizált rendszert, a Birbnetes-t mely a természetben elhelyezett 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 felvételtől, 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. 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 naplózás vagy a hiba bejelentés, de ezek használata nehézkes, nem kézenfekvő. Szükség van egy olyan megoldásra, amivel egy helyen és egyszerűen lehet kezelni és felügyelni a rendszer egyes elemeit.

1.2. Megoldás

A jelen szakdolgozat egy olyan webes alkalmazás elkészítését dokumentálja, mellyel 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 diagramokon ábrázolok. Ilyen statisztikák például, hogy időben melyik eszköz mikor észlelt madárhangot, vagy hogy hány hang üzenet érkezik az eszközöktől másodpercenként.

1.3. A szakdolgozat felépítése

A szakdolgozatom első részében, a 2. fejezetben, bemutatom a vizualizálni kívánt rendszer felépítését, az egyes komponensek közötti kapcsolatokat, valamint a vizualizációs szempontból releváns technológiákat, amire a rendszer 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 vizualizálni kívánt rendszer bemutatása

Az alkalmazásom célja egy létező rendszer, a Birbnetes folyamatainak vizualizálása. Ebben a fejezetben ismertetem a Birbnetes mikroszolgáltatás rendszerének architektúráját és az általa használt technológiákat. Részletesen kifejtem az alkalmazásom szempontjából fontos komponensek feladatát és működését.

2.1. Elméleti összefoglaló

A bemutatásra kerülő rendszert a tanszéken egy projekt keretén belül készítették kollégáim, melyet részletesen dokumentáltak korábbi nyilvános publikációkban [20] [11]. A következőkben a rendszer által használt technológiákat és elveket csak olyan szinten részletezem, hogy annak működése érhető legyen.

2.1.1. Cloud, felhőalapú rendszerek

A cloud lényegében annyit jelent, hogy a szervert, amin az alkalmazás fut, nem a fejlesztőnek kell üzemeltetnie, hanem valamilyen másik szervezet¹által vannak karban tartva. Ez több okból is hasznos:

- **Költséghatékonyabb.** Nem szükséges berendezéseket vásárolni, azok üzemeltetési díja nem közvetlen a fejlesztőt éri. Az egyetlen költség a bérlet, 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.

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

2.1.1.1. Mikroszolgáltatások

A mikroszolgáltatások (microservices) 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 minél kevesebb feladata és függősége legyen, ezzel megnő a tesztelhetőség és könnyebb a skálázhatóság.

2.1.1.2. Konténerek

A konténerek az operációs rendszer virtualizációt megvalósító egyik alkalmazása. Ezekre különböző korlátozások rakhatók 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 [29] a 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, minimális erőforrást igényelnek a szerveren, így nem kell utánuk annyit 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 tovább küldése 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. OpenAPI

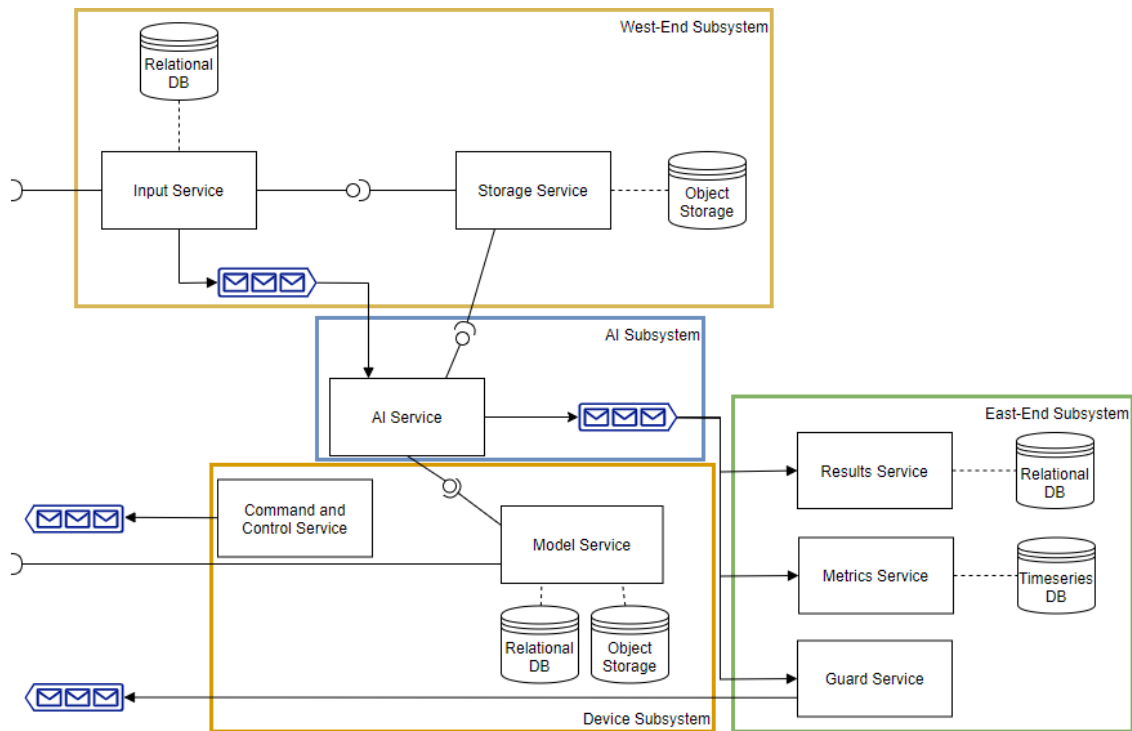
Az OpenAPI egy nyilvános alkalmazás-programozási leíró, amely a fejlesztők számára hozzáférést biztosít egy másik alkalmazáshoz. Az API-k leírják és meghatározzák, hogy egy alkalmazás hogyan kommunikálhat egy másikkal, melyet használva a fejlesztők könnyedén képesek a kommunikációra képes kódot írni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

2.2. Rendszerszintű architektúra

A Birbnetes 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 OpenAPI 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 Birbnetes rendszer architektúrája. Forrás: Madárhang azonosító és riasztó felhő-natív rendszer TDK dolgozat [20]

2.2.1.1. IoT eszközök

Szülőültetvényekben telepített eszközök, melyek adott időközönként publikálják állapotukat 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 ezeket 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 relációs adatbázisába. Ilyenek például a beküldő eszköz azonosítója, a beérkezés dátuma vagy a hangüzenet rendszerszintű egyedi azonosítója. Amint a szolgáltatás a beérkezett üzenettel kapcsolatban elvégezte az összes feladatát, publikál egy üzenetet egy másik 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 előzőkkel ellentétben egyáltalán nem vesz részt a minták fogadásában, feldolgozásában vagy kezelésében. Felelősége az eszközök és azok szenzorjai 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 a munkám elején a fontosabb vizualizációs alternatívákat értékeltem.

3.1. Tervezés

A munkám elején egyeztettem a seregély riasztó keretrendszert fejlesztő kollégákkal, hogy ki tudjam választani a vizualizáció szempontjából legfontosabb komponenseket. A jellemző adatvizualizációs megoldások közül az alábbi hármat találtam kulcsfontosságúnak a következő célokra:

- **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.
- **Eszközállapotok.** 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.
- **Diagramok.** A hőtérképen kívül egyéb olyan diagramok is hasznosak lehetnek, ahol látható például, hogy melyik eszköz melyik percben észlelt madárhangot vagy, hogy egy eszköz összesen hány madárhangot észlelt. Minél több információ, annál jobb.

Ezeken kívül fontos követelmény volt még, hogy az alkalmazásom futtatható legyen Linux környezetben is, hogy az telepíthető legyen a Birbnetes Kubernetes [29] klaszterébe.

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

⁰Microsoft Teams: Csevegő és gyülekezés tartó alkalmazás.

3.2. Alternatívák

Az imént vázolt igények kielégítésére sok, széles körben alkalmazott megoldás létezik már, melyek jó példát mutattak a saját alkalmazásom fejlesztése során.

3.2.1. Grafana

A Grafana [27] az egy nyílt forráskódú platformfüggetlen vizualizációs web alkalmazás. Egy támogatott adatbázishoz csatlakoztatva különféle interaktív gráfokat és diagramokat generál. A testreszabhatóság maximalizálásának 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

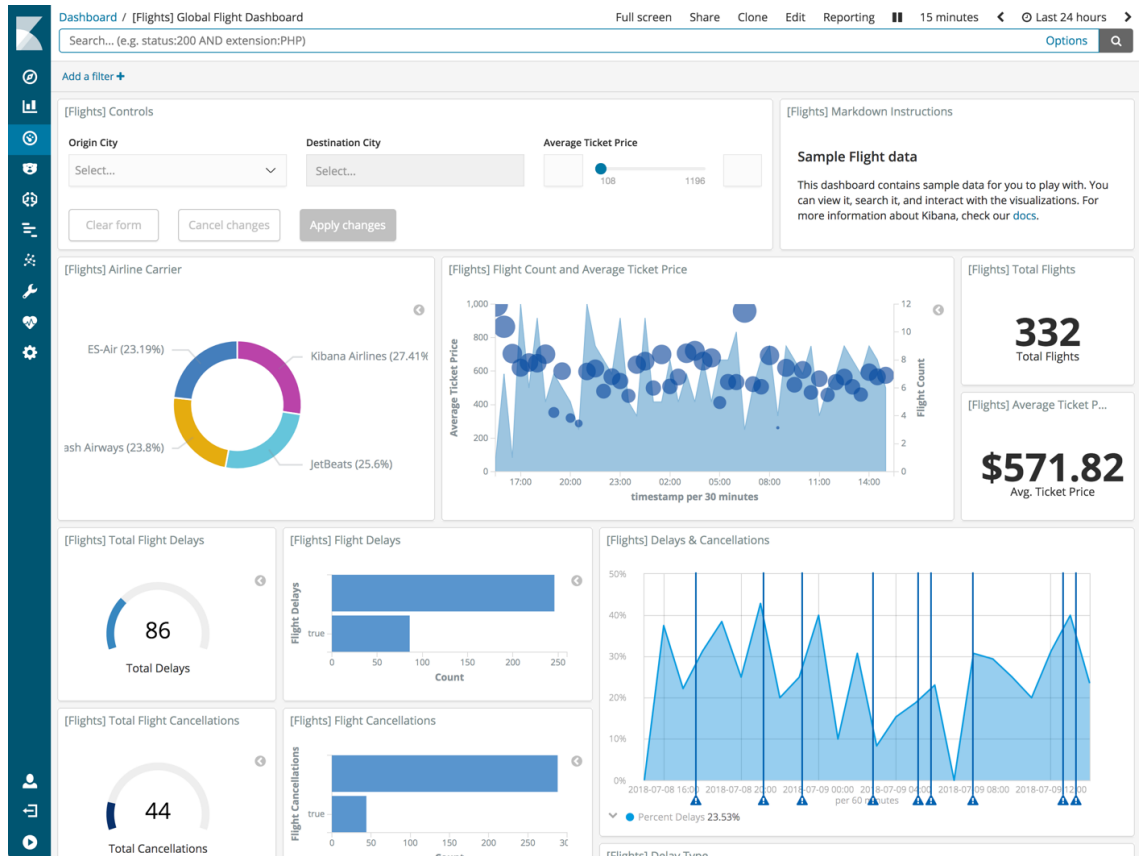
3.2.2. Kibana

A Kibana [28] 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 Elasticsearch¹ adatok, főként napló bejegyzések, analizálására használják.

3.2.3. Kubernetes Dashboard (Web UI)

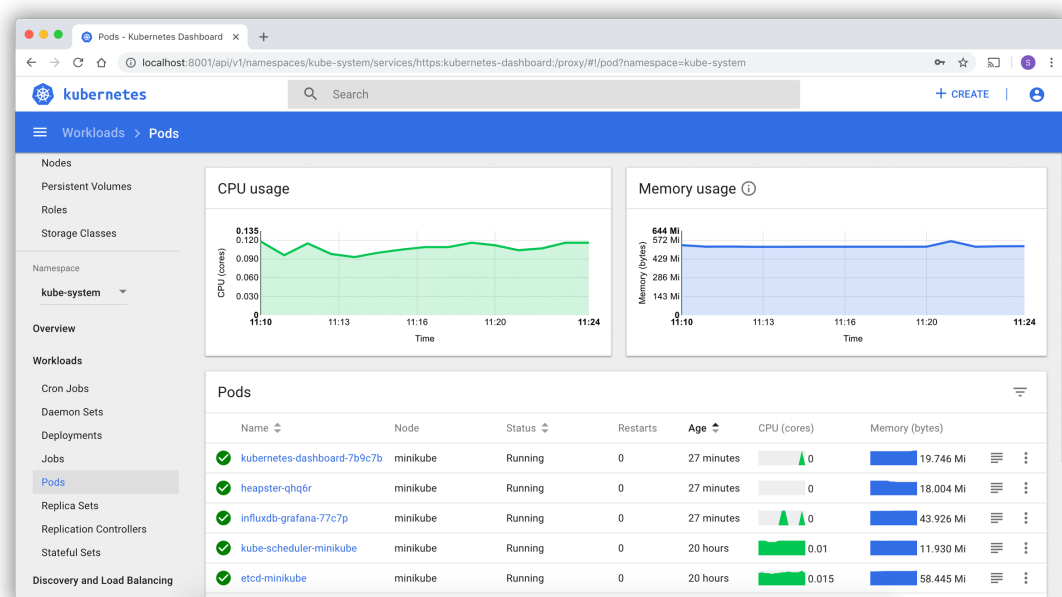
A Kubernetes Dashboard [12] elsősorban nem a különböző adatok vizualizálását szolgálja, inkább a klaszter menedzselését próbálja egyszerűbbé és jobban áttekinthetővé tenni.

¹Ingyenes 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 [5] 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

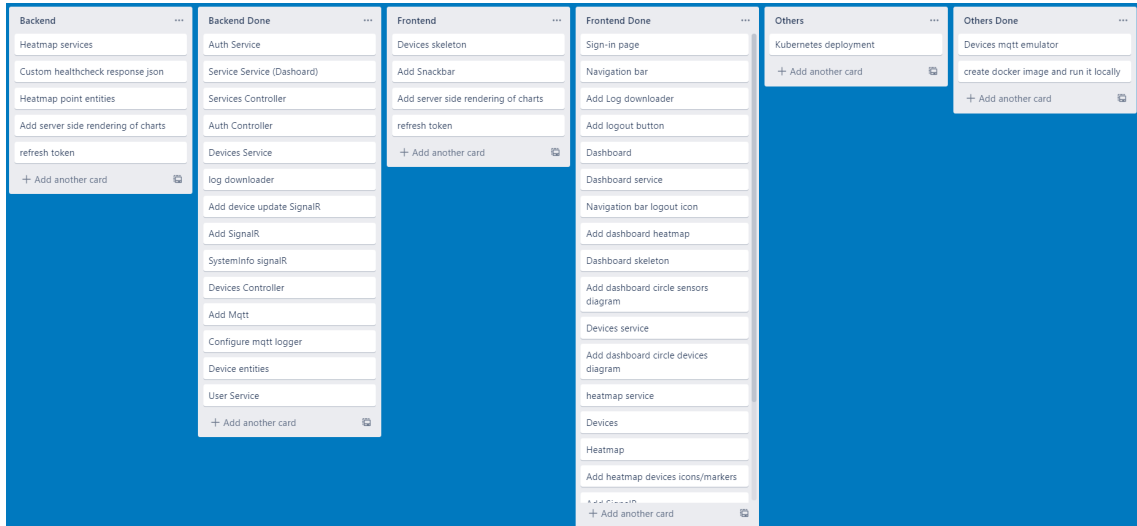
A Trello [30] egy webes projektmenedzsment alkalmazás. Azért használtam a fejlesztés során, mert szerettem volna egy helyet, ami tükrözi a fejlesztés állapotát, ahova le tudom írni az alkalmazással kapcsolatos ötleteimet. Különböző listákban tároltam a fejlesztésre váró és a kész feladatokat szerver, kliens és egyéb szerint.

4.1.3. Visual Studio

A Visual Studio [24] 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 [22] inkább szövegszerkesztő, mint fejlesztőkörnyezet. Ennek köszönhetően jelentősen gyorsabb és egyszerűbb a használata. Különböző 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 platformfüggetlen, 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 csomagkezelő rendszerének, a NuGet-nek [10] köszönhető. Használatának egyik előnye, hogy ugyan az a C# kód tud futni a szerver és 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 leképző keretrendszer a .NET-hez. Az adatbázissal való kommunikációt könnyítést szolgálja. Használatával

C#-ban lehet adatbázis lekérdezéseket írni a LINQ (Language-Integrated Query) szoftvercsomag 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) [9] 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 [8] 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 [13] is egy .NET szoftvercsomag, mely a Birbnetes 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 [23]-ot választottam, egyrészt mert egyszerű a használata, másrészt mert már korábban használtam. Konfigurációs fájljában meg lehet adni a naplózott események célját, mely lehet akár fájl vagy konzol is. Meg lehet még adni az események elrendezését, hogy azok milyen formában kerüljenek a célokba, milyen plusz információt tartalmazzanak.

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 [4] 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 [1] elsősorban egy kezelőfelület tervezési útmutató a Google által, melyet követve szép és minőségi felületeket lehet készíteni.

A Material UI [19] egy szoftvercsomag, mely ezeket az útmutatásokat követő egyszerű React komponenseket tartalmaz. Alkalmazásával könnyű esztétikus felhasználói felületeket készíteni, minimalizált a CSS használatával.

4.3.3. Apexcharts

Az Apexcharts [25] egy nyílt forráskódú JavaScript szoftvercsomag, amellyel könnyen konfigurálható, modern kinézetű diagramokat 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-et [6] 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 három rétegű architektúrát alkalmaztam, melynek lényege, hogy az alkalmazást logikailag három elkülönülő részre bontjuk:

- **Adatelérési 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 közvetlen 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. Adatelérési réteg

Az adatelérést az Entity Framework Core segítségével oldottam meg. Telepítettem egy MSSQL adatbázis szerveret 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.

```
1     public record User
2     {
3         public int Id { get; set; }
4         public string Name { get; set; }
5         public byte[] PasswordHash { get; set; }
6         public byte[] PasswordSalt { get; set; }
7
8         public Roles Role { get; set; }
9
10        public bool IsFromConfig { get; set; }
11    }
12
13    public record Service
14    {
15        public int Id { get; set; }
16        public string Name { get; set; }
17        public Uri Url { get; set; }
18
19        public bool IsFromConfig { get; set; }
20    }
```

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 megkülönböztetésnek 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 Birbnetes Command and Control és Input komponensekkel kommunikáló szolgáltatások is, melyeket azok OpenAPI leírói alapján az NSwag Studio [17] 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) algoritmust, pontosabban annak a HMACSHA512 [7] 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.

Azért hasznos ily módon, és nem mondjuk egyszerű szöveges formában tárolni a felhasználók jelszavát, mert így a felhasználón kívül senki sem tudja, hogy mi volt az eredeti jelszava, az algoritmus egyirányú volta miatt¹. Ha véletlenül rossz kezekbe kerülne az adatbázis tartalma, akkor sem fognak tudni bejelentkezni a felhasználók adataival.

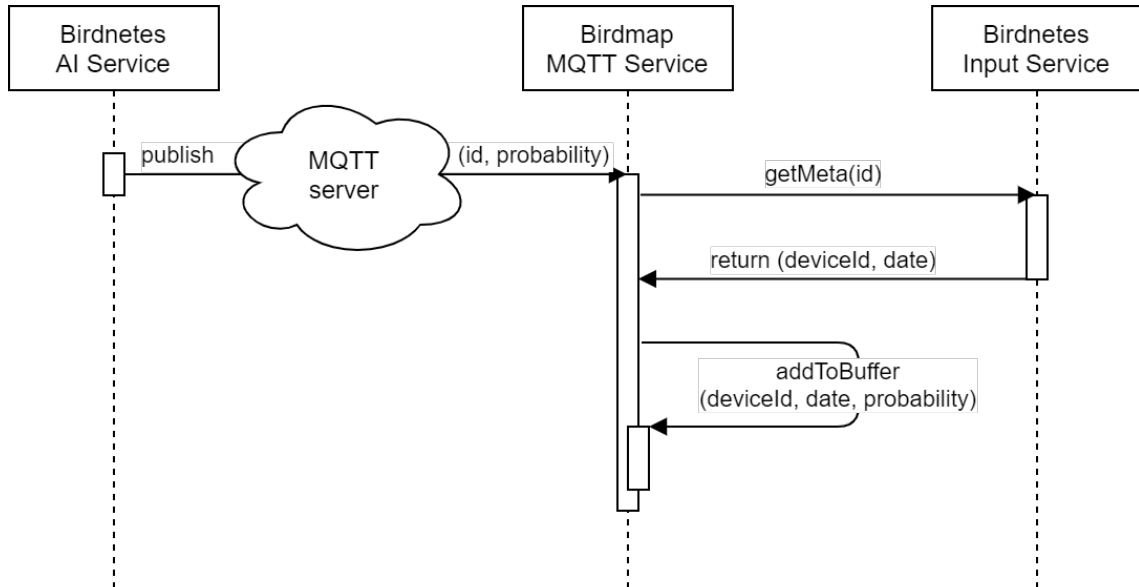
5.3.1. Kommunikációs Szolgáltatások

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 Birbnetes 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 5.1-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 milliszekundum 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.

¹Generálni egyszerű és gyors. Visszafejteni közel lehetetlen.



5.1. ábra. A Birdmap MQTT szolgáltatásának szekvenciája

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 szolgáltatás napló bejegyzéseit a 5.2 lista alapján szűrtem. Minden Debug szinttől nagyobb és Error szinttől kisebb bejegyzés, mely tartalmazza az Mqtt kulcsszót az mqttFile azonosítójú fájlba kerül.

```

<targets>
  ...
  <target xsi:type="File" name="mqttFile" fileName="..." layout="..." />
  ...
</targets>

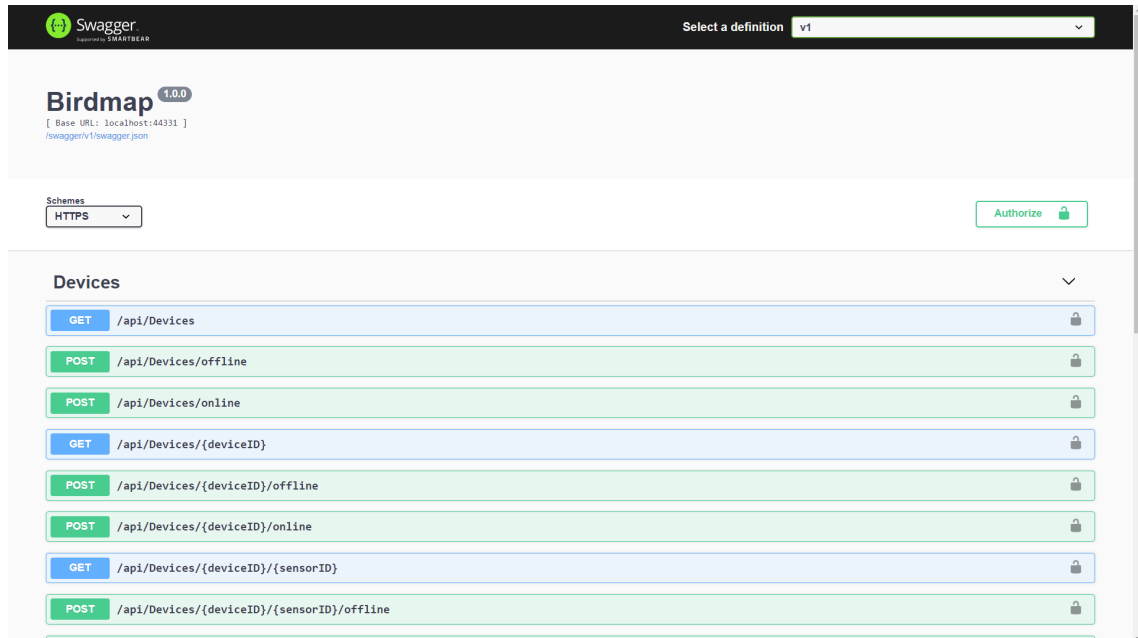
<rules>
  ...
  <logger name="*.Mqtt*" minlevel="Trace" maxlevel="Warning" writeTo="mqttFile"
    final="true"/>
  ...
</rules>
  
```

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.

5.4.1. Swagger

Az NSwag [16] szoftvercsomag segítségével regisztrálok egy szolgáltatást, mely a szervertől található kontrollereket felhasználva generál egy OpenAPI specifikációt és annak egy Swagger UI [21] felületet, ahol a végpontok kipróbálhatóak, tesztelhetőek kliensoldal nélkül is.



5.2. ábra. Az alkalmazásom Swagger felülete

5.4.2. Kontrollerek

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

```
1 [Authorize(Roles = "User, Admin")]
2 [ApiController]
3 [Route("api/[controller]")]
4 public class DevicesController : ControllerBase
5 {
6     [Authorize(Roles = "Admin")]
7     [HttpPost, Route("online")]
8     public async Task<IActionResult> Onlineall()
9     {
10         ...
11     }
12     ...
13 }
```

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

A jogosultságok kezelését a JSON Web Token-ekkel oldottam meg. A felhasználó 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 vég-

pontjait alapértelmezetten User és Admin jogosultságú felhasználó hívhatja, az "api/devices/online" végpontot azonban csak Admin jogosultságú. Hasonlóképpen oldottam meg ezt a többi kontrollernél is. A User felhasználók csak olyan végpontokat hívhat, mely kizárólag az állapotok olvasásával jár. Az Admin felhasználók hívhatnak bármilyen végpontot.

A szerveroldalon négy különböző kontroller található, melyek mindegyikének alapvető feladata az üzleti logikát megvalósító szolgáltatások használata, a működés naplózás, illetve az imént említett végpontok autorizálása és kiszolgálása. Ezekon kívül a kontrollerek speciális feladata a következő:

- Az **AuthController** felel a felhasználók bejelentkezésének lebonyolításáért, a JSON Web Token elkészítéséért. Az [Authorize] helyett itt az [AllowAnonymous] attribútum van használva, mellyel azt lehet jelezni, hogy a végpont bejelentkezés nélkül is hívható.
- A **ServiceController** felel az alkalmazás által használt külső szolgáltatások állapotának lekérdezhetőségéért. Ilyenek például a Birbnetes rendszer vagy az MQTT szolgáltatás állapota.
- A **DevicesController** felel a Command and Control mikroszolgáltatással való kommunikáció megvalósításáért, illetve a SignalR használatáért. Ha egy felhasználó valamelyik végpontot használva változtat valamelyik eszköz állapotán, akkor a kontroller jelez erről a klienseknek.
- A **LogController** felel azért, hogy az Admin jogosultságú felhasználók letölthessék a szerveroldalon készült naplófájlokat.

Az adatbázisból érkező adatok gyakran túl sok vagy túl kevés információt tartalmaznak ahhoz, hogy kiolvasás után rögtön elküldjem a kliensoldalnak. Például amikor a felhasználó bejelentkezik a kiolvasott User objektum tartalmazza annak jelszavát (hashtelt formában), viszont nem tartalmazza az autorizációhoz használt token adatait. Ennek a megoldására adatátviteli objektumokat hoztam létre, melyek csak azokat a mezőket tartalmazták amelyekre a felhasználónak szüksége van. Az adatbázisból kiolvasott objektum hasznos részeit és egyéb használni kívánt információt átmásolom az átviteli objektumba. Majd ezt küldöm el a kliensoldal felé.

Hogy az adatok másolását ne kézzel kelljen csinálnom, az AutoMapper [26] szoftvercsomagot alkalmaztam, melynek használata rendkívül egyszerű. Meg lehet adni profilokat, ahol két objektum közötti leképezéseket lehet felvenni. A szoftvercsomag automatikusan átmásolja az azonos nevű mezőket az egyik objektumból a másikba, de meg lehet adni egyedi leképezéseket is.

```
1 // Creating maps.
2 CreateMap<User, AuthenticateResponse>()
3     .ForMember(m => m.Username, opt => opt.MapFrom(m => m.Name))
4     .ForMember(m => m.UserRole, opt => opt.MapFrom(m => m.Role))
5     .ReverseMap();
6
7 CreateMap<Service, ServiceRequest>()
8     .ReverseMap();
9
10 // Using maps.
11 IMapper mapper = GetMapper();
12 User user = GetUserFromDb();
13 AuthenticateResponse response = mapper.Map<AuthenticateResponse>(user);
```

5.4. lista. Egy példa az AutoMapper használatára.

6. fejezet

Kliens oldal

Ebben a fejezetben bemutatom a kliensoldal architektúráját. Ismertetem a különböző komponensek felépítését.

6.1. Architektúra

Az alkalmazásnak minden oldala egy külön React komponens, mely mindegyikének saját mappája van a főkönyvtár alatt, ahol az egyes oldalak által használt szolgáltatások és egyéb komponensek találhatóak. A közösen használt szolgáltatások és komponensek a common mappába kerültek.

A kliensoldal belépési pontja az App.js fájlban található App komponens. Itt egy React Switch-ben fel van sorolva az összes oldal komponense azok elérési útvonalai szerint. Ezt szemlélteti a 6.1-es lista. Az a komponens jelenik meg a felületen, amelyiknek path mező értéke megegyezik az URL-ben találhatóval.

```
1 <Switch>
2   <PublicRoute exact path="/login" component={AuthComponent} />
3   <AdminRoute exact path="/logs" component={LogsComponent} />
4   <DevicesContextProvider>
5     <PrivateRoute exact path="/" component={DashboardComponent} />
6     <PrivateRoute exact path="/devices/:id?" component={DevicesComponent} />
7     <PrivateRoute exact path="/heatmap" component={HeatmapComponent} />
8   </DevicesContextProvider>
9 </Switch>
```

6.1. lista. Az App.js Switch tartalma.

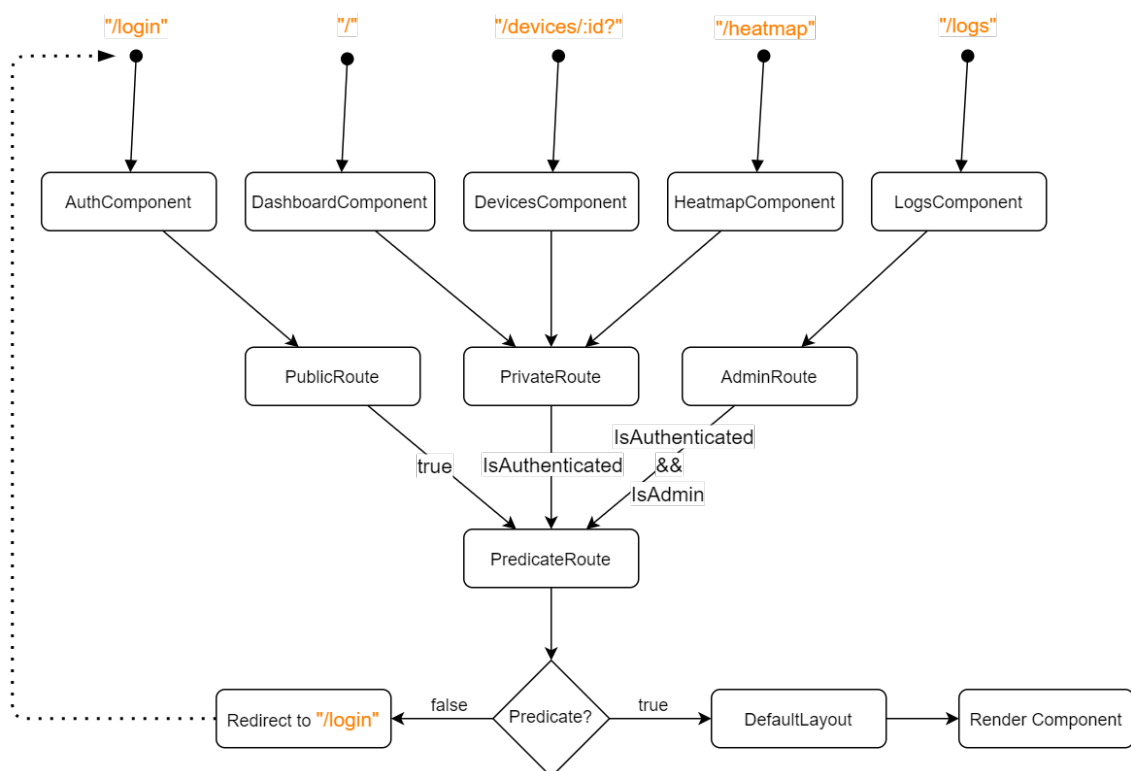
Hozzáférés szempontjából háromfajta oldalt különböztetünk meg:

- **Publikus oldal.** Az oldal bejelentkezés nélkül is látogatható.
- **Privát oldal.** Az oldal csak bejelentkezés után látogatható.
- **Admin oldal.** Az oldalt csak bejelentkezett admin felhasználók látogathatják.

Ezek alapján készítettem két generikus komponens. Az egyik a DefaultLayout komponens, mely az oldal alapértelmezett elrendezéséért felel. Paraméterében át lehet adni

egy másik megjeleníteni kívánt komponenst, melyet a fejléc alatt jelenít meg. Mivel minden komponens ebbe az bázis komponensbe van csomagolva, így akárhova navigálunk az oldalon a felület mindig egységes marad.

A másik komponens a PredicateRoute, melynek paraméterében meg lehet adni egy feltételt, illetve egy másik komponenst. Ha a feltétel hamis akkor átirányítja a felhasználót a bejelentkező oldalra, ha igaz akkor megjeleníti a DefaultLayout-ba csomagolt komponenst. Publikus oldalnál a feltétel mindig igaz. Privátnál a feltétel a bejelentkezéshez van kötve. Az admin oldal feltétele egyrészt szintén a bejelentkezés, másrészt a felhasználó Admin jogosultsága. Ezt a folyamatot próbálja szemléltetni a 6.1-es ábra. Legfelül sárgával vannak feltüntetve a hívható végpontok, alattuk a hozzájuk kapcsolt megjelenítendő komponensek, azok alatt pedig a hozzáférést szabályozó komponensek.



6.1. ábra. A Birdmap kliensoldalának architektúrája

6.2. Kommunikáció a szerveroldallal

A szerveroldallal való kommunikációt rendkívül egyszerűen tudtam implementálni köszönhetően a 5.4.1-as fejezetben bemutatott Swagger oldalnak és annak, hogy az NSwag Studio-val [17] a C#-on kívül lehet TypeScript¹klienseket is generálni a leíró fájlból. Így készültek el a komponensek kommunikációért felelős szolgáltatásai.

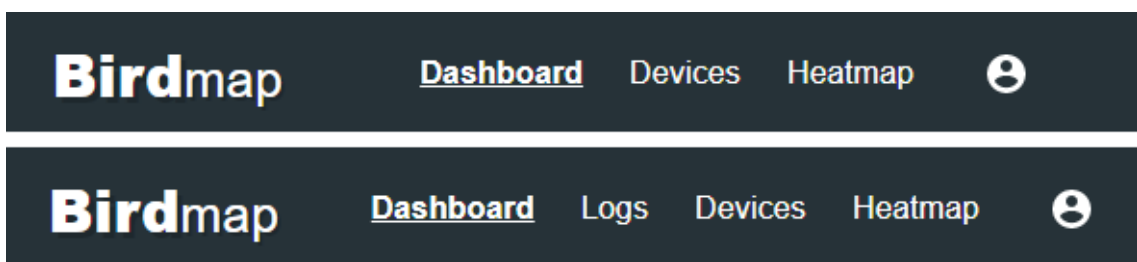
¹JavaScript-re épített statikus típusdefiniókat tartalmazó nyelv. JavaScript és TypeScript együtt is használható.

6.3. Komponensek

Ebben a szakaszban ismertetem az egyes oldalak komponenseit és azok alkomponenseit, illetve a navigációért felelős fejléceket.

6.3.1. Navigáció

A fejléc két komponensből áll. Az egyik az oldal címe a másik az oldalak linkjeit tartalmazó komponens. Utóbbit a React NavLink komponenseivel készítettem, melyeknek meg lehet adni, hogy kattintásra hova irányítsa a felhasználót. Ha a jelenlegi webcím tartalmazza a linknek megadott címet, akkor az aktív státuszba kerül, melyre külön stílus osztályok vonatkoznak. Ezt használva, az aktív linkeket egy aláhúzással jelölöm.

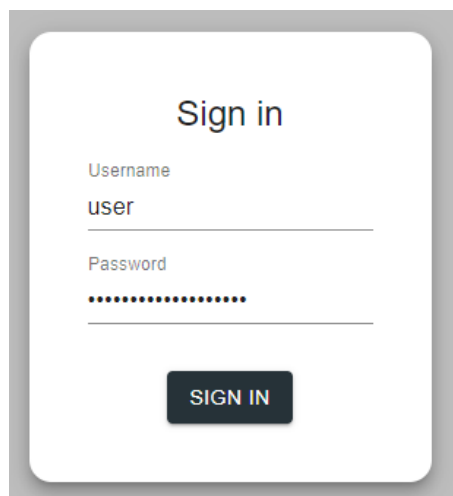


6.2. ábra. A Birdmap fejléce. Felül a User, alul az Admin felhasználóké

A fejléc alapértelmezetten része a DefaultLayout komponensnek, így minden oldalon megjelenítésre kerül.

6.3.2. Login

A bejelentkező oldal viszonylag egyszerű. Két szövegdobozt és egy bejelentkező gombot tartalmaz, ahogy az a 6.3-as ábrán is látszik.

The image shows a screenshot of the Birdmap login form. It features a 'Sign in' title, a 'Username' field with the value 'user', a 'Password' field with masked characters, and a 'SIGN IN' button.

6.3. ábra. A Birdmap bejelentkező felülete

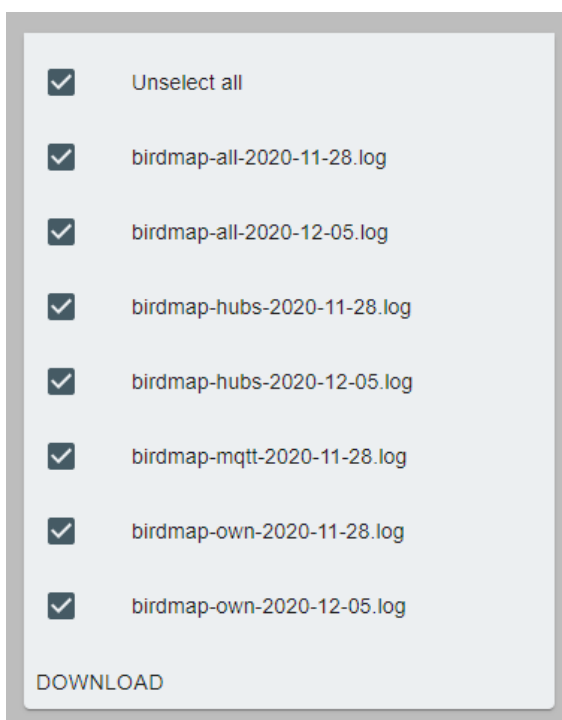
A generált szerverrel kommunikáló szolgáltatás be van csomagolva egy közösen használt másik szolgáltatásba. Ennek célja, hogy a bejelentkezés eredményét több komponens is olvashassa, hiszen az alkalmazás felületét alapvetően megkülönbözteti, egyrészt a bejelentkezés sikeressége, másrészt a bejelentkezett felhasználó jogosultsági köre.

Sikeres bejelentkezés után a szerver elküldi a felhasználó szerepét, illetve a hozzáférési token-t, amelyre a kliens többi szolgáltatásának is szüksége lesz a kommunikációhoz. Ezeket az oldal `sessionStorage`-ában²tárolom és a becsomagolt szolgáltatáson keresztül elérhetőek.

Kijelentkezni a navigációs fejlécben található profil ikonra való kattintással lehet.

6.3.3. Logs

Ez az oldal az Admin felhasználó számára lehetővé teszi a szerveren található naplófájlok letöltését zip fájlformátumú archív fájlokban. Komponense a 6.4-es ábrán látható.



6.4. ábra. A Birdmap naplófájlok letöltésének felülete

6.3.4. Eszközállapot- és hangüzenet-kezelő szolgáltatás

A szakasz további komponenseinek van egy közös ismertetője. Mégpedig, hogy mindegyiknek szüksége van a kihelyezett eszközök adataira és az azok által publikált hangüzenetekből képzett valószínűsége. A Reactnek van egy beépített komponense `Context` [31] néven, mellyel különböző komponensek között lehet adatokat megosztani. Ezt használva készítettem egy `DevicesContextProvider` osztályt, melynek feladata a szerver eszköz kontrollereivel való kommunikáció a megfelelő szolgáltatáson keresztül, illetve a `SignalR`

²Webtárhely objektum. Lehetővé teszi a kulcs-érték párok tárolását a böngészőben.

csatornákra való feliratkozás. Ezekből az adatokból egy `DevicesContext` készül, mely a `Provider` által átadásra kerül annak minden gyerekének. A 6.1-es listában látható, hogy a `DevicesContextProvider` szülője a `Dashboard`, `Devices` és `Heatmap` komponenseknek.

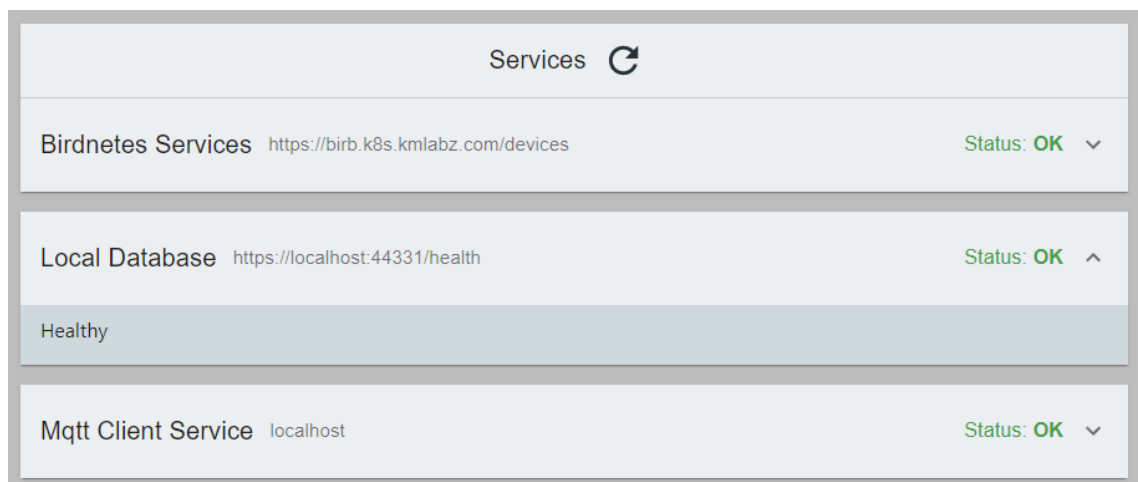
6.3.5. Dashboard

A `Dashboard` az alkalmazás kezdő oldala. Itt található meg a külső szolgáltatások állapotát vizsgáló komponens, illetve a kihelyezett eszközök működési folyamatában áttekintést nyújtó diagramok mindegyike.

Az oldal megjelenítésekor elindul egy másodpercenként ismétlődő folyamat, mely a `DevicesContext`-ből kiolvasott értékekből legenerálja a diagramokon megjelenítendő összes adatot. Ez azonban az adat mennyiségétől függően akár egy-két másodpercig is eltarthat, ami rendkívül lassúvá és használhatatlanná tenné a felületet. Ennek elkerülése érdekében az adatfeldolgozó folyamat egyszerre csak egy pár elemet dolgoz fel, mely folyamatok között 20 milliszekundum szüneteket iktattam be. Továbbá hogy a különböző diagramok animációi is zökkenőmentesek legyenek, azok adatai cserélése között is van 300 milliszekundum szünet. Így valamivel lassabb az adatfeldolgozás, de a felület használható marad.

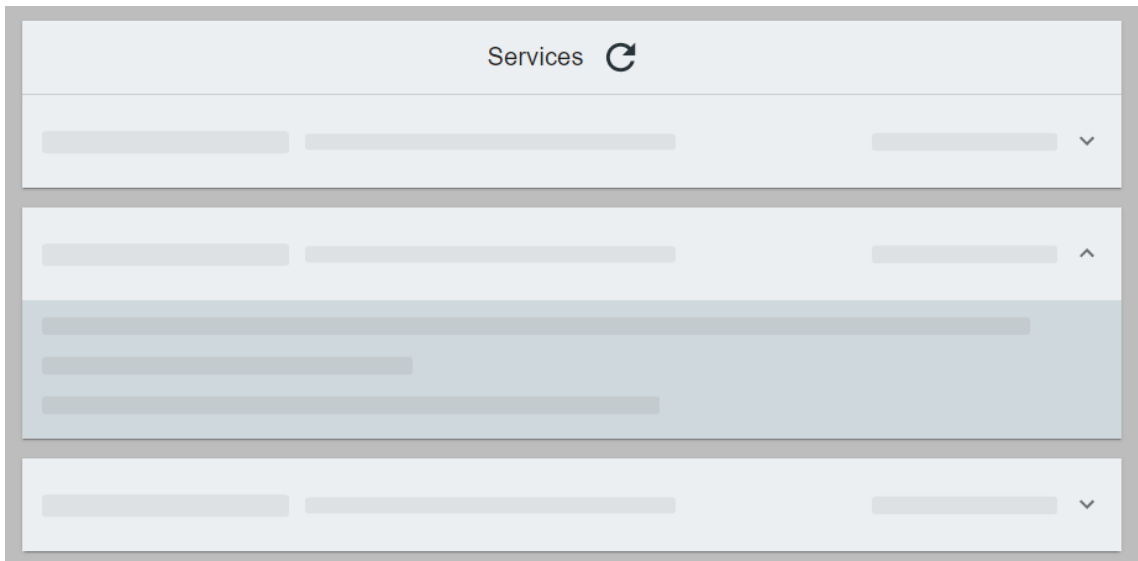
6.3.5.1. Külső szolgáltatások

Az alkalmazás használatának szempontjából van néhány olyan külső szolgáltatás, melyek elérhetősége hiányában a rendszer működésképtelen. Ilyen például a `Birbnetes` klasztere vagy a szerver `MQTT` szolgáltatása. Ezért készítettem el az 6.5-ös ábrán látható információs panelt, ahol a szolgáltatások állapotát lehet látni, hogy a felhasználó tudja miért nem működik esetleg az alkalmazás. A felület megvalósításhoz a `Material UI` `Accordion` elemét használtam, ami lényegében egy lenyíló lista. Ennek fejlécében a szolgáltatás neve, elérési útvonala és státusza látható. A lenyíló elemben a szolgáltatástól érkezett válasz van megjelenítve.



6.5. ábra. Az alkalmazás által használt külső komponensek állapotának megjelenítéséért felelős komponens

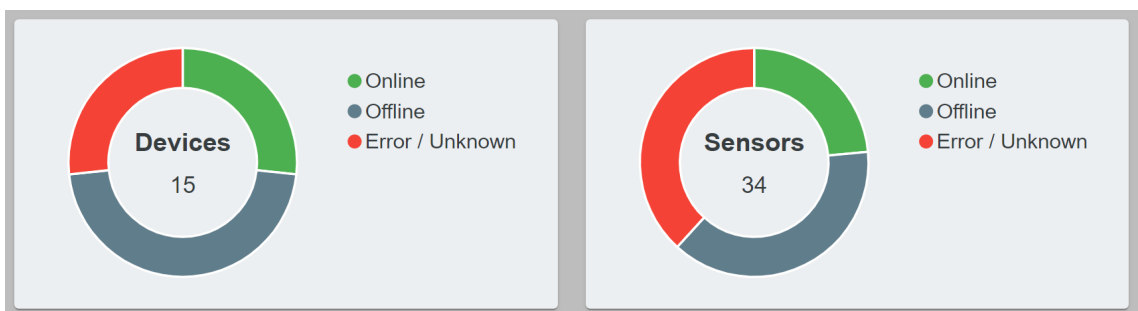
Az oldal betöltése vagy a frissítés gomb megnyomása esetén az adatok lekérésre kerülnek a szervertől. Ez a folyamat akár öt-hat másodpercig is eltarthat, mely közben a felhasználó egy üres listát látna. Ennek elkerülésére használom a Material UI Skeleton komponensét, mely egy megadható méretű töltő csíkkal helyettesíti az Accordion-ban található elemeket a 6.6-os ábrán látható módon. Azért célszerű ennek a használata, mert így a felhasználónak több információja van arról, hogy a felületen milyen adatok és hol fognak megjelenni. A felhasználói élmény maximalizálása érdekében a frissítés előtt lekérdezem a szervertől, hogy hány darab szolgáltatás található az adatbázisban és annyi darab töltőcsíkos Accordion-t jelenítek meg.



6.6. ábra. A Skeletonok alkalmazása a külső szolgáltatások állapotának betöltése közben.

6.3.5.2. Eszközök és szenzorok állapota

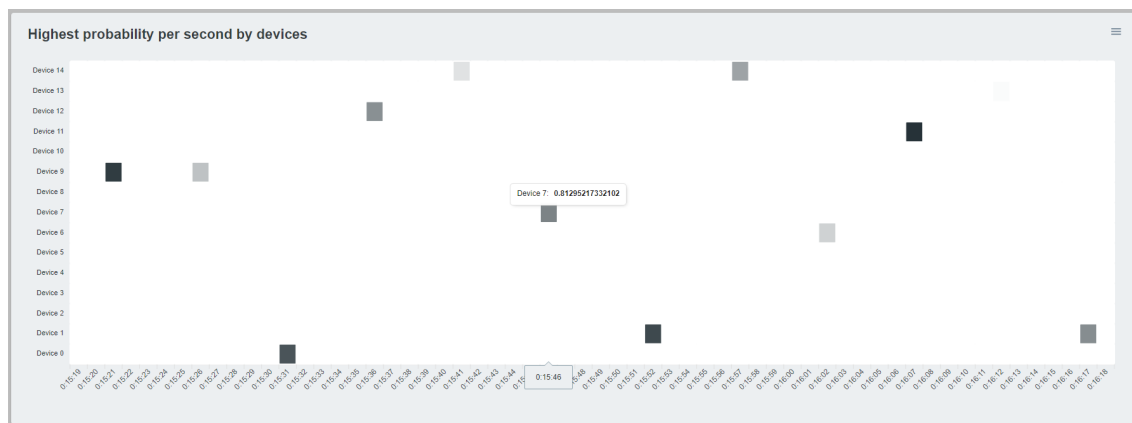
Ennek a komponensnek a szerepe, hogy áttekintést nyújtson az eszközök és szenzorok állapotáról. Úgy gondoltam, hogy erre a legcélravezetőbb eszköz a 6.7-es ábrán is látható Apexcharts fánk diagramja. Látható, hogy hány darab eszköz és szenzor van bekapcsolt, kikapcsolt, illetve hibás állapotban. Az állapotok változása esetén a DevicesContextProvider-nek köszönhetően az adatok automatikusan frissülnek.



6.7. ábra. A Dashboard eszköz- és szenzor állapotok diagramja

6.3.5.3. Hőterkép diagramok

Ezekkel a diagramokkal az a célom, hogy az eszközök által küldött észleléseket időrendben vizualizáljam. Megvalósításukhoz az Apexcharts Heatmap típusú diagramját használtam. A 6.8-as ábrán látható diagram az elmúlt egy percben küldött, másodpercenként a legnagyobb, hangüzenetekből képzett valószínűségeket ábrázolja. A 6.9-es ábrán látható diagram pedig az elmúlt egy órában percenként a legnagyobbakat.



6.8. ábra. Másodperc alapú hőterképes diagram

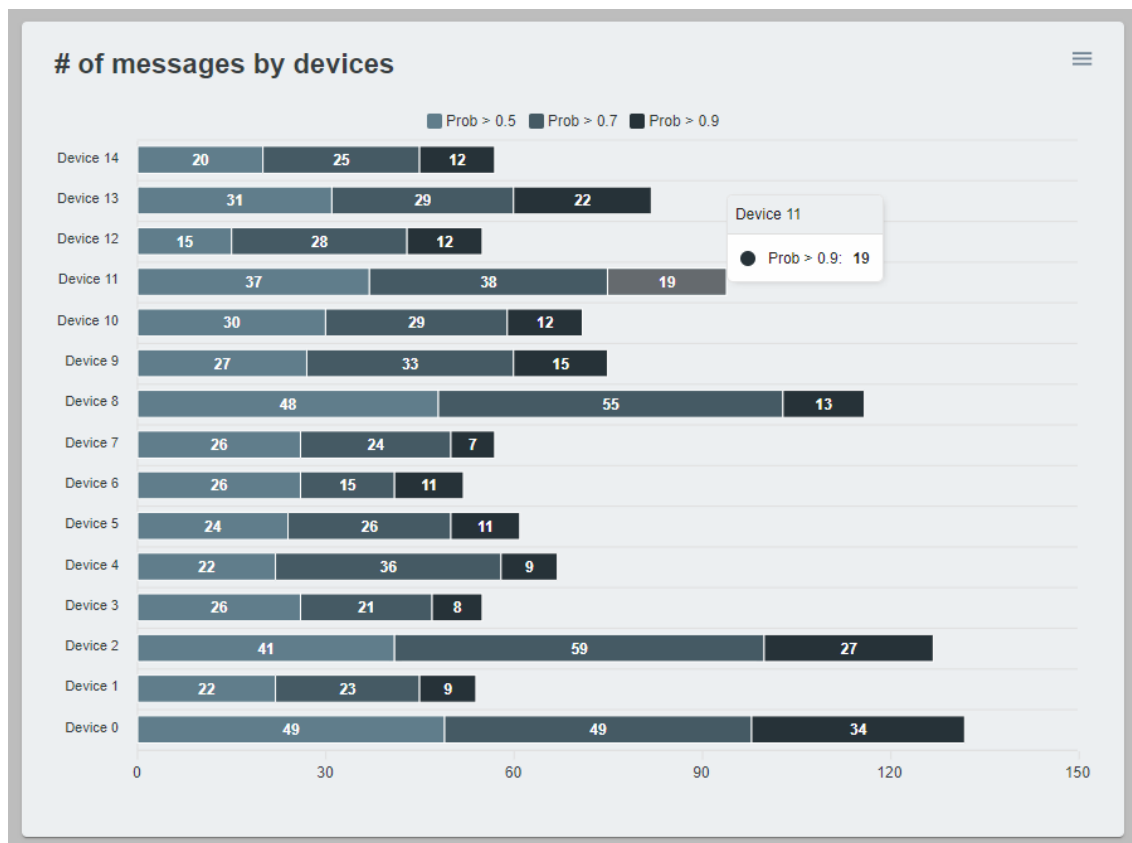


6.9. ábra. Perc alapú hőterképes diagram

A függőleges tengelyen a rendszer eszközei vannak dinamikusan megjelenítve. A vízszintes tengelyen pedig az említett időtartományok. A diagramokon látható négyzetek a valószínűség nagyságától függően sötétebbek vagy világosabbak.

6.3.5.4. Riasztás számláló

Ez egy egyszerű oszlopdiagram, mely aggregálja az egyes eszközök által küldött hangüzeneteket 0.5 valószínűség felett a 6.10-es ábrán látható módon. Segítségével megvizsgálható, hogy mely eszközök riasztanak a legtöbbet a legnagyobb valószínűséggel.

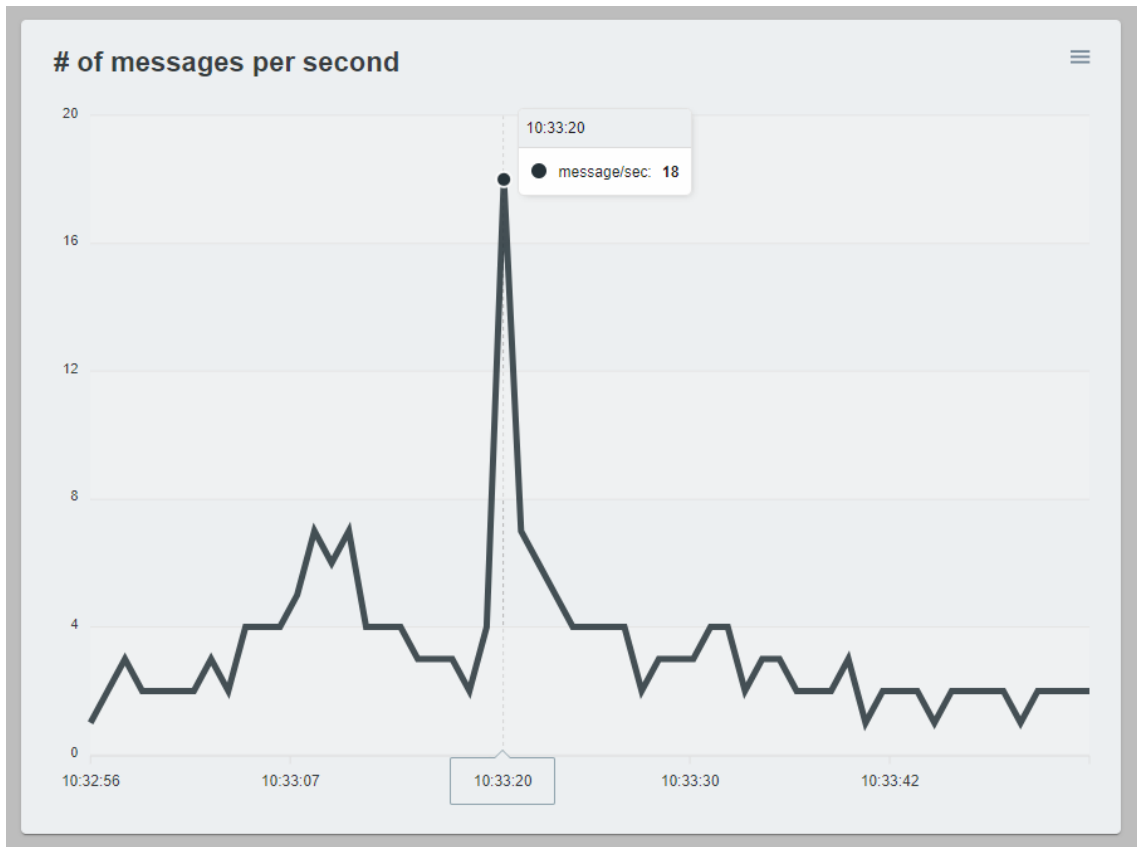


6.10. ábra. Eszközönkénti riasztásokat számláló diagram

Az egyes oszlopok három részre vannak bontva az üzenetek öt tized, hét tized és kilenc tized fölötti valószínűsége szerint.

6.3.5.5. Üzenetek gyakorisága

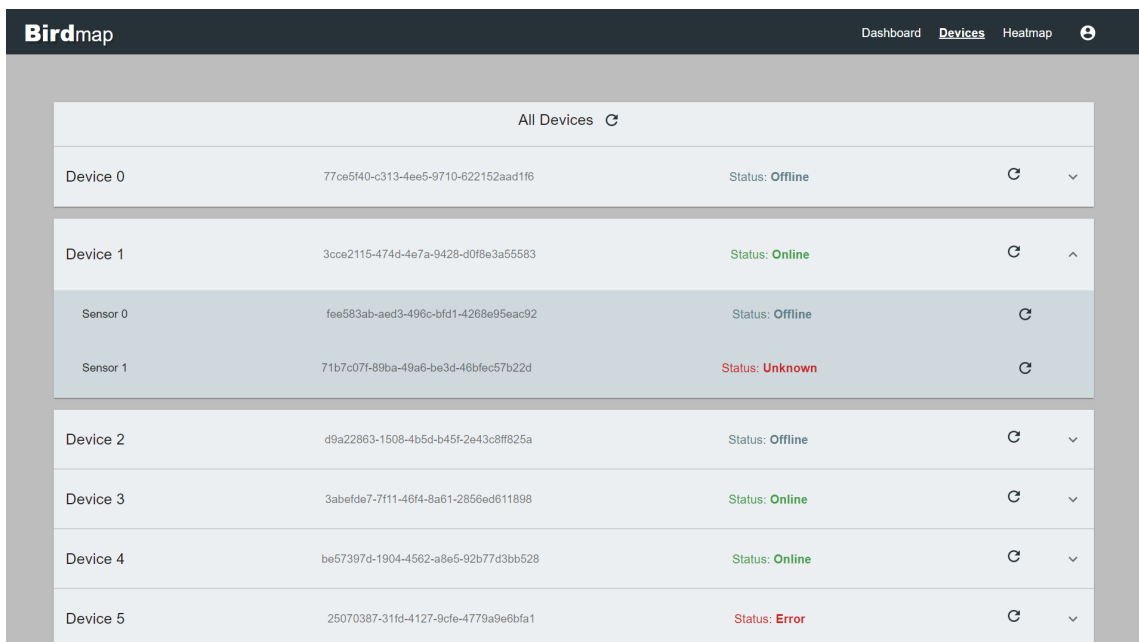
Az oldalon található utolsó diagram egy vonal diagram, melynek célja, hogy ábrázolja a rendszer által küldött üzenetek számát másodpercenként. A 6.11-es ábrán látható a komponens. A vízszintes tengelyen a legelső érték az alkalmazás által először észlelt üzenet időpontja. Az utolsó érték a legutoljára észlelt időpontja. A függőleges tengelyen az adott másodpercben érkező üzenetek száma van ábrázolva. Az előzőkkel ellentétben itt az adatok nincsennek szűrve a hangüzenet valószínűsége alapján, tehát a rendszer által küldött összes üzenet látható.



6.11. ábra. A másodpercenként érkező üzenetek számát ábrázoló diagram.

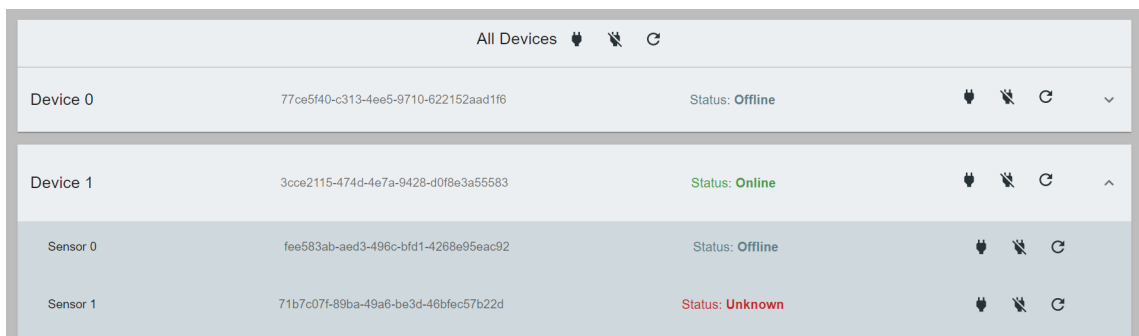
6.3.6. Devices

Ez az oldal lehetővé teszi a felhasználók számára az eszközök állapotának áttekintését, Admin felhasználók számára azok menedzselését is. Az eszközök dinamikusan jelennek meg a DevicesContextProvider adatai alapján, melyek megjelenítésére a Material UI Accordion komponensét használom. Ennek fejlécében az eszköz neve, egyedi azonosítója és státusza található. A lenyíló részben pedig az eszköz által használt szenzorok neve, azonosítója és státusza. Admin felhasználók számára a felület két fajta gombbal bővül, melyekkel be és ki lehet kapcsolni az egyes eszközöket, szenzorokat. Az Accordion-ok felett található egy külön panel, mellyel egyszerre lehet kezelni az összes eszközt és azok szenzorjait. A Devices oldal felülete a 6.12-es ábrán, az Admin felhasználók számára nyújtott plusz funkciók a 6.13-as ábrán láthatók.



All Devices			
Device 0	77ce5f40-c313-4ee5-9710-622152aad1f6	Status: Offline	🔄
Device 1	3cce2115-474d-4e7a-9428-d0f8e3a55583	Status: Online	🔄
Sensor 0	fee583ab-aed3-496c-bfd1-4268e95eac92	Status: Offline	🔄
Sensor 1	71b7c07f-89ba-49a6-be3d-46bfec57b22d	Status: Unknown	🔄
Device 2	d9a22863-1508-4b5d-b45f-2e43c8f825a	Status: Offline	🔄
Device 3	3abefde7-7f11-46f4-8a61-2856ed611898	Status: Online	🔄
Device 4	be57397d-1904-4562-a9e5-92b77d3bb528	Status: Online	🔄
Device 5	25070387-31fd-4127-9cfe-4779a9e6bfa1	Status: Error	🔄

6.12. ábra. A Devices oldal felülete.



All Devices			
Device 0	77ce5f40-c313-4ee5-9710-622152aad1f6	Status: Offline	🔌 🗑️ 🔄
Device 1	3cce2115-474d-4e7a-9428-d0f8e3a55583	Status: Online	🔌 🗑️ 🔄
Sensor 0	fee583ab-aed3-496c-bfd1-4268e95eac92	Status: Offline	🔌 🗑️ 🔄
Sensor 1	71b7c07f-89ba-49a6-be3d-46bfec57b22d	Status: Unknown	🔌 🗑️ 🔄

6.13. ábra. Az Admin felhasználók számára elérhető plusz funkciók.

6.3.7. Heatmap

Az alkalmazással szemben az egyik legfontosabb követelmény a hőtérképes vizualizáció volt, mely ezen az oldalon található. A Google Maps API segítségével megjeleníték egy térképet a felületen, majd erre kerül a hőtérképes réteg. A térképre szélességi és hosszúsági körök alapján lehet rajzolni. Ezt használva megjeleníttem a rendszer összes eszközét azok koordinátái szerint. A kék színű ikonok jelölik a bekapcsolt állapotban lévő, a sárga a kikapcsolt állapotban lévő, a piros pedig a hibás állapotban lévő eszközöket. Ha a felhasználó az egerét az ikonok fölé helyezi, megjelenik egy szövegdox, melyben az eszköz azonosítója és státusza látható. Az ikonra kattintva a felhasználó a Devices oldalra kerül, ahol megnyílik a kattintott eszköz Accordion-ja.

A `DevicesContext` tartalmazza az eszközök által küldött üzenetek adatait, melyeknek a 0.5 valószínűségtől nagyobb részhalmazát a hőtérkép által kezelhető adatokká konvertállok. Egyrészt szükség van az előbb is említett földrajzi koordinátákra, melyeket az üzenetek eszköz azonosítója alapján határozok meg. Másrészt szükség van egy súly értékre, mely a pont színezésének pirosságát határozza meg. Ezt az értéket az üzenetek valószínűség értékével tettem egyenlővé. Minél több magasabb valószínűségű riasztás érkezik egy adott eszköztől, a körülötte lévő terület annál pirosabb lesz.

A 6.14-ös ábra mutatja a térkép működését miközben 4 eszköz is seregélyeket észlelt.



6.14. ábra. A Heatmap oldal felülete.

7. fejezet

Tesztkörnyezet

Az alkalmazásom fejlesztésének megkönnyítése érdekében nagy hangsúlyt fektettem a tesztelhetőségre. Helyettesíteni akartam az éles rendszer komponenseivel való kommunikációt, hogy abban az esetben is folyni tudjon a fejlesztés, ha a rendszer épp nem elérhető. Ezen kívül hasznos, ha az alkalmazás által feldolgozott adatok személyre szabhatóak, hiszen sokszor olyan problémákra lehet így fényt deríteni, amelyek nem vagy csak jóval később jönnének elő az éles rendszer használata során.

A tesztelhetőség megvalósításához három szoftver komponenst kell helyettesítenem, melyeket az alábbi szekciókban ismertetek.

7.1. Helyettesítő szolgáltatások

Az alkalmazásom szerver oldali szolgáltatásai a Birbnetes Command and Control (a kódban Device) és Input Service-ekkel azok OpenAPI leíróiból generált interfészein keresztül kommunikál. Ezen interfészek mögé bármilyen implementáció regisztrálható, mely helyettesíti az éles rendszer működését.

Készítettem egy osztályt `DummyDeviceAndInputService` néven, mely a szerver indításakor mű eszköz adatokat generál egy lokális változóval állítható darabszámban, majd ezeket egy belső listában tárolja. Az eszközök státuszát és koordinátáit egy véletlenszám generátor segítségével határozom meg. Az osztály implementálja a Device Service interfészét, melynek metódusai az imént említett mű eszközlista elemeivel dolgoznak, azok státuszát olvassák és módosítják. Illetve implementálja az Input Service interfészét, melynek metódusa bármilyen paraméterből kapott egyedi azonosító esetén visszaad egy véletlenszerűen kiválasztott bekapcsolt státuszú eszközt a listából.

Az alkalmazás által regisztrált és ezáltal használt interfész implementációi a konfigurációs fájl egy logikai értéke alapján cserélhető az éles és a helyettesítő között, a 7.1-es listában látható módon.

```

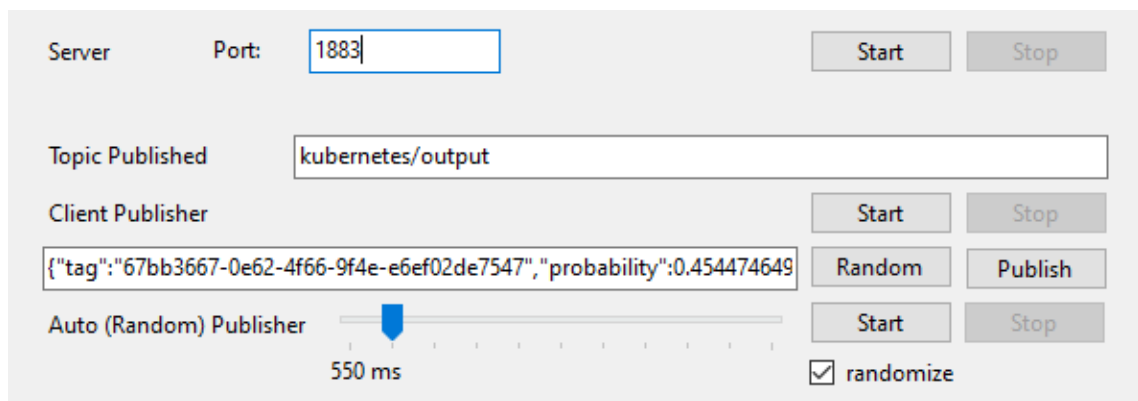
1   if (configuration.GetValue<bool>("UseDummyServices"))
2   {
3       services.AddTransient<IInputService, DummyDeviceAndInputService>();
4       services.AddTransient<IDeviceService, DummyDeviceAndInputService>();
5   }
6   else
7   {
8       services.AddTransient<IInputService, LiveInputService>();
9       services.AddTransient<IDeviceService, LiveDeviceService>();
10  }

```

7.1. lista. A helyettesítő és az éles szolgáltatások regisztrálásának logikája

7.2. MQTT tesztalkalmazás

Az MQTT.NET szoftvercsomag github oldalán található néhány példa a csomag használatára [14]. Ezek között találtam Sepp Penner MQTTnet.TestApp.WinForms [15] projektjét, mely egy Windows Forms applikáció az említett szoftvercsomag által nyújtott funkciók tesztelésére. Indítható vele MQTT szerver, feliratkozó kliens és publikáló kliens is. Ezek meglétével az alkalmazás képes az üzenetek manuális publikálására egy a felületen beállítható témában. Én azonban szerettem volna az üzeneteket automatikusan bizonyos időközönként küldeni, ezért átalakítottam az alkalmazást az igényeimnek megfelelően a 7.1-es ábrán látható módon. Elhelyeztem a felületen egy csúszkát, mellyel az üzenet küldés intervalluma állítható, illetve két új gombot, melyekkel az üzenet küldő időzítő indítható és megállítható. Az alkalmazás képes üzenetek adatainak generálására, mellyel az AI Service által publikált üzenetek modelljeivel azonos adatokat generálok.



7.1. ábra. Az MQTT kommunikációt tesztelő alkalmazás felületének egy része

8. fejezet

Docker image készítés

Az éles rendszerrel való kommunikáció megvalósításához készítenem kell egy Docker image-et, melyet telepíteni lehet a Kubernetes klaszterébe. Ehhez először készítettem egy Dockerfile-t [3], mely az image-ek automatikus elkészítését teszi lehetővé. Utasításokat lehet benne felsorolni, melyekkel a konténer környezetét kell felépíteni. Meg lehet adni kiindulópontokat, mely az image alapjául szolgál. Erre a célra én az ASP.NET futtatókörnyeztet használtam, mely tartalmazza az alkalmazás futtatásához szükséges parancsokat. Ezek után a Dockerfile utasításait használva bemásolom a Release konfigurációval fordított alkalmazásomat a konténer egy mappájába, majd a belépési pont utasítással megadom az alkalmazás indításához szükséges parancsot. Ezt futtatva sikeresen elkészül a Docker image.

Azonban az alkalmazás teljes értékű működéséhez annak szüksége van egy adatbázis konténerre is. Az ilyen jellegű többkonténeres rendszer problémákra nyújt megoldást a Docker Compose [18]. Egy YAML fájlban meg lehet adni az alkalmazás futtatásához szükséges szolgáltatásokat, illetve hogy ezek között milyen függőségi viszony van. Ennek használatával először készítek egy adatbázis konténert, mely inicializálása után indul csak el az alkalmazásom docker image-ének készítése. A két konténer közötti kommunikációhoz az alkalmazásomnak szüksége van még a kapcsolati karakterláncra, mely meghatározza az adatbázis elérésének paramétereit. A lokális futtatásnál ez az alkalmazás konfigurációs fájljában található, azonban ez a fájl már a konténer fájlrendszerében van, nehézkes hozzáférni. Szerencsére az ASP.NET támogatja a konfigurációk felülírását környezeti változókkal. Ehhez fel kell sorolnom a YAML fájl környezeti változói részében a felülírni kívánt konfigurációkat és értékeiket. Szintén ebben a fájlban megadtam az alkalmazás eléréséhez használni kívánt portokat. Ezek után az alkalmazásom készen áll a klaszterbe való telepítésre.

9. fejezet

Értékelés

Úgy gondolom, hogy az alkalmazásom elérte a célját. Egy használható felületet nyújt a Birbnetes mikroszolgáltatás rendszere működésének vizualizálására. A fejlesztés közben jelentős figyelmet fordítottam arra, hogy az alkalmazás felületi és kód komponensei között is minimalizáltak legyenek a függőségek, így a rendszerben történő változások esetén azok könnyen cserélhetőek, bővíthetőek.

9.1. Továbbfejlesztési lehetőségek

Az kliens oldalon történő diagramok adatainak generálása hamar túl nagy falatnak bizonyult. A bevetett optimalizációk ellenére sem lett hatványozottan gyorsabb a felület. Így az első és legfontosabb továbbfejlesztési teendő az adatok szerveroldalon történő generálása lenne.

A Logs oldal jelenleg csak a szerveroldalon készült napló fájlokat tartalmazza. Hasznos lenne, ha az egyes mikroszolgáltatások naplófájljai is letölthetőek lennének.

Ezen kívül előnyös lenne a rendszer belső működését vizualizáló komponensek alkalmazása is, ahol lehetne látni az egyes mikroszolgáltatásokra vonatkozó különböző metrikákat például az adatfeldolgozási időt vagy a beérkezett kérések számát.

Irodalomjegyzék

- [1] Design guidelines. URL <https://material.io/resources/get-started#design>. Megtekintve: 2020.11.31.
- [2] Docker overview. URL <https://docs.docker.com/get-started/overview/>. Megtekintve: 2020.11.28.
- [3] Dockerfile reference.
URL <https://docs.docker.com/engine/reference/builder/>. Megtekintve: 2020.12.08.
- [4] Getting started with react.js. URL <https://reactjs.org/docs/getting-started.html>. Megtekintve: 2020.11.31.
- [5] A git hivatalos oldalának about szekciója. URL <https://git-scm.com/about/branching-and-merging>. Megtekintve: 2020.11.30.
- [6] Google map react. URL <https://www.npmjs.com/package/google-map-react>. Megtekintve: 2020.11.31.
- [7] Az hmacsha512 dokumentációja.
URL <https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.hmacsha512>. Megtekintve: 2020.12.02.
- [8] Introduction to asp.net core signalr. URL <https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-5.0>. Megtekintve: 2020.12.04.
- [9] Introduction to json web tokens.
URL <https://jwt.io/introduction/>. Megtekintve: 2020.12.02.
- [10] An introduction to nuget.
URL <https://docs.microsoft.com/en-us/nuget/what-is-nuget>. Megtekintve: 2020.12.04.
- [11] Nagy Kristóf: Tömeges gép-gép kommunikáció mezőgazdasági alkalmazása, 2020.
- [12] Kubernetes web ui (dashboard). URL <https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>. Megtekintve: 2020.11.28.

- [13] Mqttnet.
URL <https://github.com/chkr1011/MQTTnet#mqttnet>. Megtekintve: 2020.12.04.
- [14] Az mqtt.net github oldalán található példák. URL <https://github.com/chkr1011/MQTTnet/wiki/Examples>. Megtekintve: 2020.12.07.
- [15] Mqttnet.testapp.winform. URL <https://github.com/SeppPenner/MQTTnet.TestApp.WinForms#mqttnettestappwinform>. Megtekintve: 2020.12.07.
- [16] Nswag: The swagger/openapi toolchain for .net, asp.net core and typescript. URL <https://github.com/RicoSuter/NSwag#nswag-the-swaggeropenapi-toolchain-for-net-aspnet-core-and-typescript>. Megtekintve: 2020.12.01.
- [17] Nswagstudio.
URL <https://github.com/RicoSuter/NSwag/wiki/NSwagStudio>. Megtekintve: 2020.12.01.
- [18] Overview of docker compose.
URL <https://docs.docker.com/compose/>. Megtekintve: 2020.12.08.
- [19] Quick start.
URL <https://material-ui.com/getting-started/usage/#quick-start>. Megtekintve: 2020.11.31.
- [20] Torma Kristóf és Pütkösdi Marcell: Madárhang azonosító és riasztó felhő-natív rendszer, 2020.
- [21] Swagger ui.
URL <https://swagger.io/tools/swagger-ui/>. Megtekintve: 2020.12.01.
- [22] User interface.
URL <https://code.visualstudio.com/docs/getstarted/userinterface>. Megtekintve: 2020.11.30.
- [23] Welcome to nlog! URL <https://nlog-project.org/>. Megtekintve: 2020.11.30.
- [24] Welcome to the visual studio ide. URL <https://docs.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2019>. Megtekintve: 2020.11.30.
- [25] What is apexcharts? URL <https://apexcharts.com/>. Megtekintve: 2020.11.31.
- [26] What is automapper? URL <https://automapper.org/>. Megtekintve: 2020.11.30.
- [27] What is grafana?
URL <https://grafana.com/docs/grafana/latest/getting-started/>. Megtekintve: 2020.11.29.

- [28] What is kibana?
URL <https://www.elastic.co/what-is/kibana>. Megtekintve: 2020.11.29.
- [29] What is kubernetes?
URL <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>.
Megtekintve: 2020.11.28.
- [30] What is trello? URL <https://trello.com/en/about>. Megtekintve: 2020.11.30.
- [31] When to use context.
URL <https://reactjs.org/docs/context.html#when-to-use-context>. Megtekintve: 2020.11.31.