BKK GTFS adatok vizualizációja Grafanában

Féléves feladat - Felhő alapú IoT és Big Data platformok

Vakán Péter – FKPG5E

A projekt célja

Az alábbi dokumentációban ismertetett projekt célja egy egyszerű, szimulált IoT-rendszer létrehozása, amelyben a BKK által közzétett GTFS menetrendi adatok alapján járműmozgásokat modellezek/vizualizálok. A járművek virtuális "szenzorokként" viselkednek, amelyek bizonyos időközönként adatokat (pl. aktuális helyzet, késés) generálnak. A feladatot konténerizált környezetben oldottam meg, így lehetőségem volt egy olyan rendszert kiépíteni, amely a részkomponenseit minél optimálisabban integrálja össze.

Használt technológiák bemutatása

A projekt elkészítéséhez négy technológiát használtam, amelyek segítségével sikerült egy összehangolt, a feladatot sikeresen ellátó rendszert kiépítenem. A használt technológiák lehetővé teszik a valós idejű adatfeldolgozást, tárolást és vizualizációt.

Python programnyelv

A Python nevű interpretált programozási nyelvet használva implementáltam az IoT szimulációs projektet, amely a bemeneti adatokat (BKK GTFS adatai) olvassa be. A program lényegében IoT szenzorok valós időben való szimulációját végzi el úgy, hogy kiválaszt a bemeneti GTFS adatok közül véletlenszerűen öt darab járművet, majd a kiválasztott járművekről elküldi a szükséges adatokat adatpontonként az adatbázisnak. Az alkalmazás során törekedtem arra, hogy az adatok küldése minél valósághűbb legyen, ennek érdekében az alábbiakat építettem bele az implementációmba. Az egyes adatpontok elküldése után mesterségesen vár a program random 1-től 100 másodpercet. Emellett a szimulációban résztvevő járművek az adataikat véletlenszerű sorrendben küldik el, amit szálkezeléssel tudtam megvalósítani; minden egyes jármű szimulációját külön szálon végzem, ezért azok az információkat egymástól teljesen függetlenül, a valósághoz hasonló módon küldik el.

InfluxDB adatbázis

A járműadatok tárolására az InfluxDB nevű időalapú adatbázist választottam, amely az egyik legelterjedtebb adatbáziskezelő rendszer az IoT szenzorok által küldött adatok világában, emellett sikeresen összeköthető akár MQTT protokollt megvalósító rendszerekkel, illetve különböző adatvizualizációs programokkal.

Az InfluxDB a szenzorok, egyéb eszközök által küldött (mért) adatokat Tag-Field struktúrában fogadja a publikus API-ján, majd tárolja el őket idősorosan. A Tag-Field struktúrában az alábbi adattípusok adhatóak meg:

- Tag-ek: Statikus azonosítók (pl. vehicle_id, trip_id), amelyekkel szűrni, csoportosítani stb. lehet az adatokon végzett lekérdezésekben.
- Field-ek: Dinamikus értékek (pl. stop_name, trip_name, latitude, longitude, delay), amelyek a projektemben a jármű aktuális állapotával kapcsolatos információkat reprezentálják.

Mindemellett egyrészről elmondható az is, hogy az InfluxDB skálázhatóság jó; akár több ezer jármű több ezer adatát is képes lekezelni gyakori frissítéssekkel. Másrészről a használatával egyszerűen meg tudtam oldani a Python influxdb_client nevű könyvtárán keresztül az adatok szimulációjának elküldését, ahol az egyes eszközök az egyes pillanatokban Point típusú adatot küldek el.

Grafana platform

Az adatok interaktív és több nézetben való megjelenítésére a Grafana nevű nyíltforráskódú platformot használtam, amelynek főbb erősségei többek között az alábbiak:

- speciális felhasználói interfész elemek (például az általam is használt GeoMap)
- a megjelenítés valós idejű frissítése (mindig az aktuálisan becsatolt DataSource adatbázisban lévő adatok jelennek meg az adott nézeten)
- JSON alapú konfiguráció (lehetővé teszi a nézetek gyors létrehozását, módosítását, megosztását)
    - számos konfigurációs beállítás
- automatizált DataSource kapcsolat létrehozása

Úgy gondolom, hogy a Grafana egy tökéletes választás volt az adatok megjelenítésére, mert sikerül olyan felületeket találnom, amelyeken azokat minél informatívabban tudtam megjeleníteni.

Docker és Docker Compose

A Docker egy konténerizációs platform, amely lehetővé teszi alkalmazások és függőségeik elkülönített, hordozható környezetekben (konténerekben) való futtatását. Ellentétben a virtualizációs technológiával, ahol minden virtuális gép saját operációs rendszert futtat Hypervisor segítségével, a Docker-konténerek a gazdagép operációs rendszer kernelét osztják meg, így sokkal könnyebbek, gyorsabbak és erőforrás-hatékonyabbak.

A Docker használatának fő előnyei:

- Környezetfüggetlenség: A konténerek belülről minden operációs rendszeren ugyanúgy működnek, így elkerülhetők az eltérő környezet által előjövő problémák.
- Erőforrás-hatékonyság: A konténerek megosztják a gazdagép kernelét, így sokkal könnyebbek és gyorsabban indíthatóak, mint például virtuális gépek.
- Függőségek kezelése: Minden konténer tartalmazza az alkalmazás futáshoz szükséges könyvtárakat, így nincsen szükség rendszerszintű telepítésekre.
- Skálázhatóság: Konténerek pillanatok alatt indíthatók vagy leállíthatók, számok egyszerűen növelhető, vagy csökkenthető különösebb erőforrás növekedés/csökkenés nélkül

A Docker Compose pedig egyszerűsíti több konténer együttes kezelését. Központosított módon kezelhető a teljes környezet, mert egyetlen YAML fájlban (docker-compose.yaml) definiálhatók a szolgáltatások (pl. adatbázis, backend, frontend), amelyek ezután akár egy paranccsal (docker-compose up) elindíthatóak. Automatizálható az egyes komponensek integrációja. Hálózati konfiguráció (sim-net) állítható be úgy, hogy a konténerek egymással név szerint tudjanak kommunikálni. A konténerek között különböző függőségek adhatóak meg.

Git és Github

A fájlok verziókezelés az egyik legelterjedtebb verziókezelő szoftverrel, a Git-el oldottam meg, valamint a

Rendszerarchitektúra

Az alábbi fejezetben bemutatom az elkészített rendszer architektúráját, ahol első sorban az egymással összecsatlakoztatott nagyobb komponensek és az azok közötti kapcsolatok ábrázolására összpontosítok.

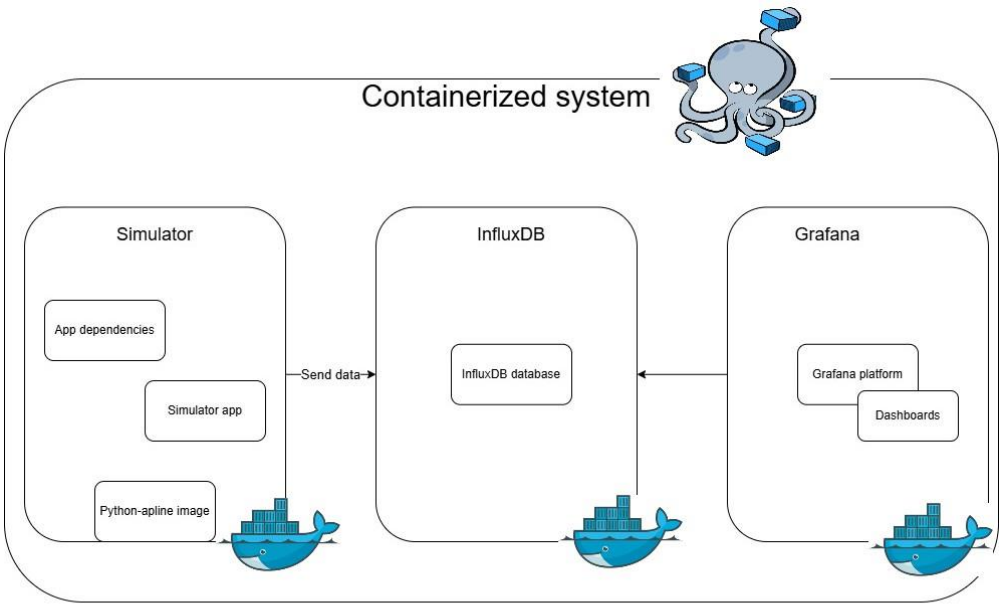
Docker architektúra komponensei

A rendszer három Docker konténerből épül fel, amelyeket orkesztrációjáért az említett docker-compose.yaml fájl felelős. A konténerek egy közös hálózaton (sim-net) belül kommunikálnak, és minden szolgáltatás elérhető a konténer neve alapján (Docker DNS segítségével).

- Simulator – az szimulációs python program és annak összes függőségét tartalmazza
- InfluxDB – az adatbáziskezelő szoftvert tartalmazza
- Grafana – a vizualizációs szoftvert tartalmazza

```
PS C:\Users\User\Documents\Projects\BKK_GTFS_DOCKER_HW> docker ps -a
CONTAINER ID   IMAGE                          COMMAND                  CREATED         STATUS                   PORTS                    NAMES
3b8cc0a04741   bkk_gtfs_docker_hw-simulator   "python run_simulati…"   9 minutes ago   Up 37 seconds            5000/tcp                 simulator
5d8ac63696e7   grafana/grafana                "/run.sh"                9 minutes ago   Up 2 minutes             0.0.0.0:3000->3000/tcp   grafana
84853b854f4c   influxdb:2.7                   "/entrypoint.sh infl…"   9 minutes ago   Up 2 minutes (healthy)   0.0.0.0:8086->8086/tcp   influxdb
PS C:\Users\User\Documents\Projects\BKK_GTFS_DOCKER_HW>
```

1. ábra A futó konténerek listája



2. ábra A rendszer leegyszerűsített architektúra diagramja

Az adatstruktúra

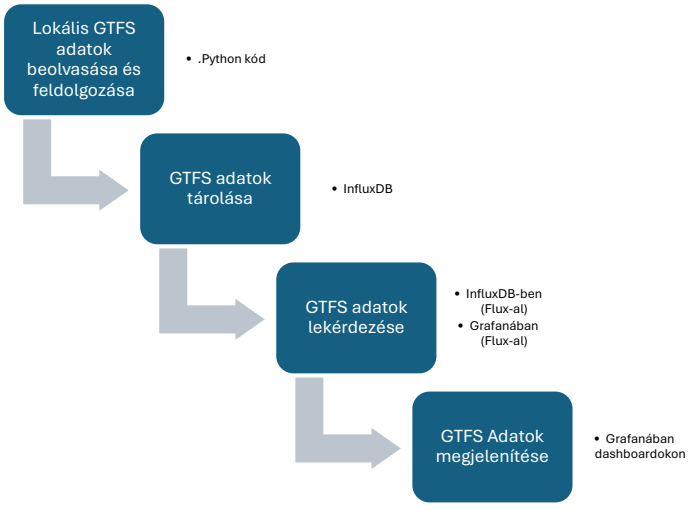
A rendszerben az egyes járművek az egyes méréseknél az alábbi információkat küldik el magukról:

- vehicle_id – az adott jármű egyedi azonosítója
- trip_id – az adott jármű adott útjának egyedi azonosítója
- stop_id – az aktuális megálló neve, ahol a jármű tartózkodik a mérés küldésekor
- trip_name – az adott út neve
- stop_name – az adott megálló neve
- latitude – földrajzi szélesség értéke
- longitude – földrajzi hosszúság értéke
- delay – az adott késés értéke
- current_datetime – az adott időpont egységes felosztásban

A mérések mindig akkor történnek, amikor az adott jármű egy adott megállóhoz eljut. Az adatstruktúra minden elemét a GTFS fájlokból olvastam ki, kivéve a késés (delay) értékét, amelynek a program futás közben generált mindig [-2;5] zárt intervallum közötti egész számot.

Adatfolyam lépései

A kihívást lényegében az jelentette, hogy a lokálisan letöltött GTFS formájú adatokból hogyan küldöm el a lényeges részeket az adatbázisnak, majd hogyan mentem el őket megfelelő formátumban annak érdekében, hogy a végén sikeresen le tudjam kérni őket a Grafanában, hogy a lekérdezés eredményét megjelenítsem különböző grafikus felületeken. Ezen folyamatot ábrázolom az alábbi folyamatábrán.



Docker-es optimalizációk

A teljes konténerizált rendszer elkészítésekor törekedtem ott, ahol csak tudtam arra, hogy minél kevesebb helyet foglaljon és lehetőleg minél gyorsabb fordítási és futtatási időt érjek el.

Ennek érdekében, részben a tantárgy során elhangzottak alapján, az alább felsorolt részletekre ügyeltem:

- A python docker image alpine verziójának használata, körülbelül tized akkora méretű (~ 85 MB) mint a „sima” python docker image.
- Minél kevesebb, csak a szükséges layerek (rétegek) használata a Dockerfile-ban.
- .dockerignore fájl használata: ezáltal nem fogja bemásolni fordításkor a felesleges környezeti fájlokat (például a pycache-t) az adott konténerben, így csökkenthető az adott konténer mérete és rövidíthető a fordítás ideje, mert nem másol fel felesleges fájlokat.

Dockerfile

```
1  FROM python:3.10-alpine
2
3  WORKDIR /app
4
5  COPY src/ /app/
6
7  RUN pip install --no-cache-dir -r requirements.txt
8
9  CMD ["python", "run_simulation.py"]
10
11 EXPOSE 5000
```

3. ábra A szimulátor Dockerfile-ja

A fentebb látható ábrán a konténer fordításakor minden sor egy-egy új régetre (layer-re) fordul, így fontos ügyelni arra, hogy ha lehetséges, akkor minél kevesebb műveletet végezzünk el a Dockerfile-ban a konténer felállításához.

Automatizációk kihasználása

Ahol volt rá lehetőség, ott alkalmaztam konfigurációs leíró fájlokat annak érdekében, hogy a teljes rendszer felállításának egyes részeit ne kelljen mindig manuálisan megoldani, ehelyett a

konfigurációs fájlok beolvasásával automatikusan felépülnek a rendszer *docker compose up* paranccsal való indításakor.

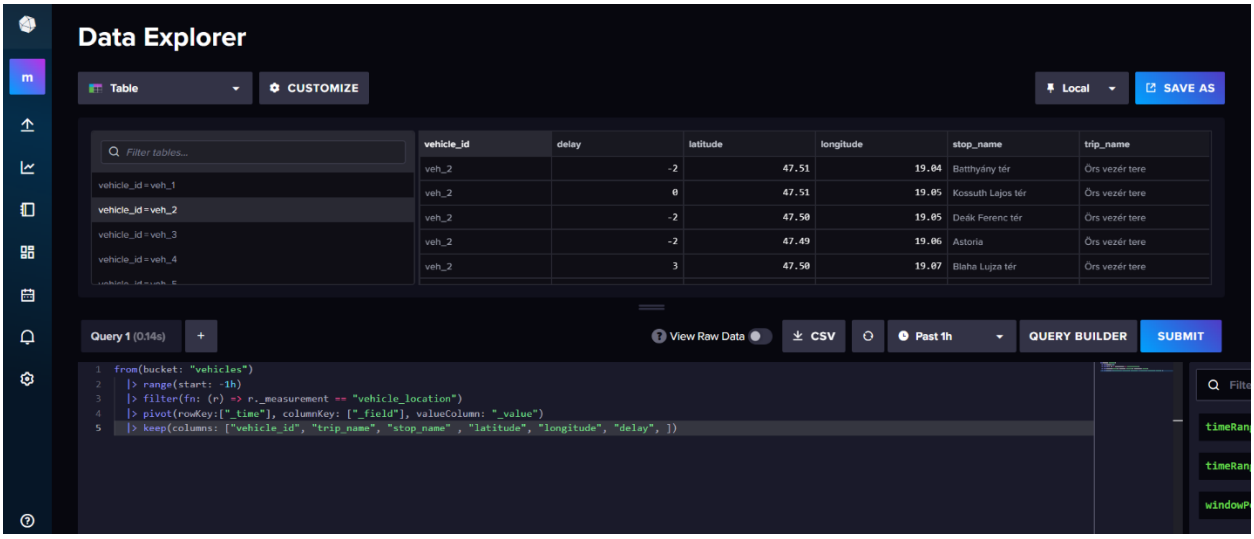
Az influxdb-datasource.yaml leíró fájl Grafana konténer volume-ába (saját fájlrendszer, amiben a lokális fájlokat használni tudja az adott konténer) való feltöltéssel megvalósítható az InfluxDB adatbáziskapcsolat automatikus beállítása a Grafanában. A Grafana innentől ezt fogja alapértelmezett DataSource-ként (adatforrásként) értelmezni.

A Grafanában megjelenített Dashboard-okat a Grafana konténer volum-ába előre feltöltött .JSON leíró fájlok segítségével automatikusan, a rendszer elindulásakor állítom össze.

A fentebb leírtak alapján mire elindul a teljes rendszer, sikeresen feláll az adatbázis kapcsolat, illetve elkészülnek a kért Dashboardok Grafanában, így nem kell minden egyes futtatáskor külön beállítani őket, amivel perceket spórolhatunk meg az életünkből.

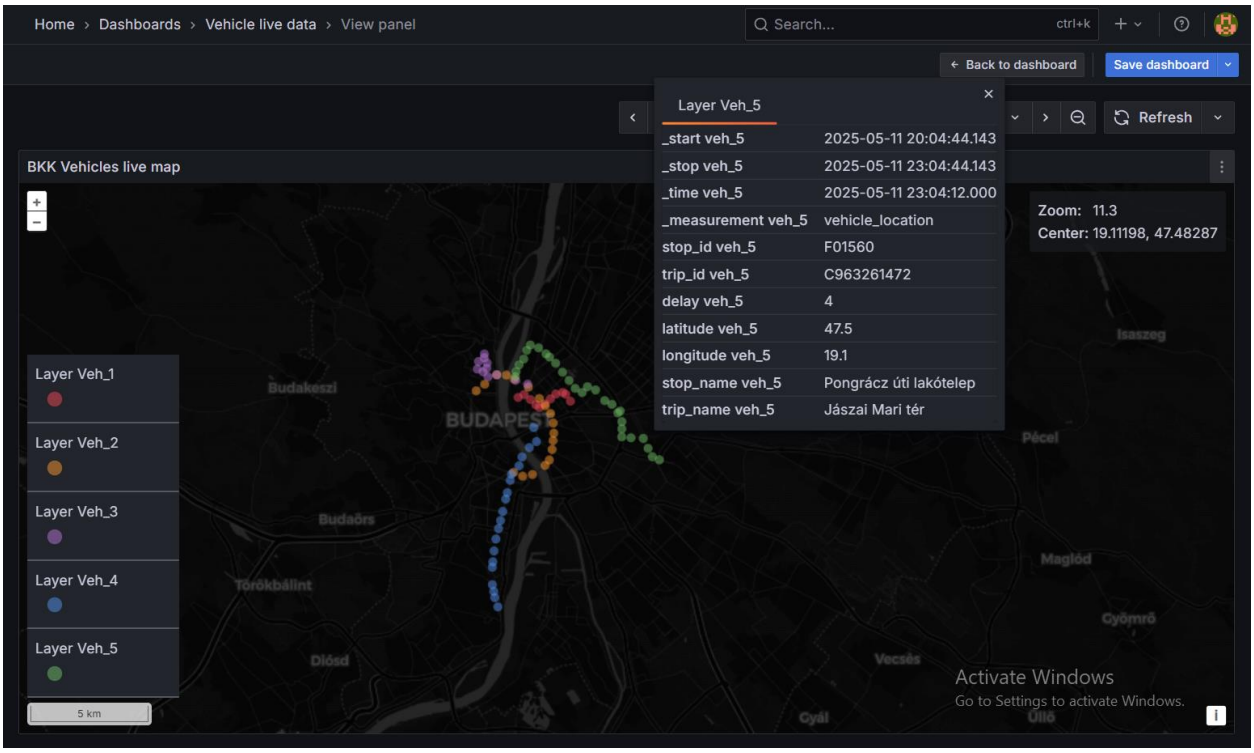
Főbb funkciók, elkészített adatvizualizációk

Az alábbi képeken látható információk a projekt sikerességét bizonyítják, hiszen ezen adatokat a Docker architektúra komponensei című fejezetben leírt komponensek összehangolt felépítésének segítségével kérdezhetőek le Flux vagy SQL lekérdezésekkel, illetve vizualizálhatóak Grafanában.



4. ábra Adatok lekérdezése InfluxDB Data Explorer fülén járművenként csoportosítva

A fenti ábrán látható, hogy sikeresen le tudtam kérdezni az eltárolt adatokat az egyes járművekre (vehicle 1… vehicle 5) csoportosítva az adatbázisból.

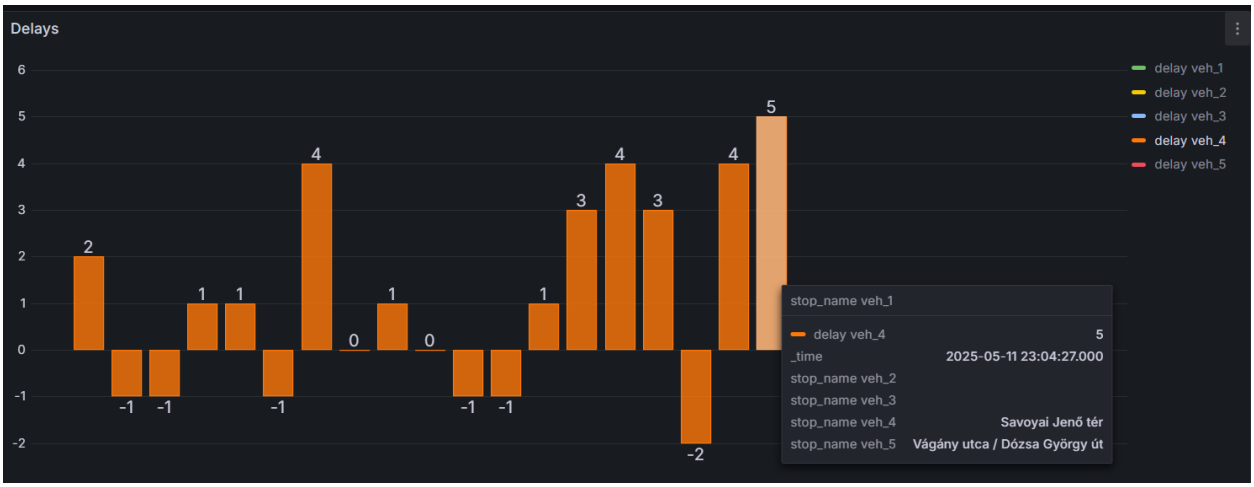


5. ábra A járművek útvonalának megjelenítése valós időben

A fenti ábrán látható a teljes rendszer egyik legfontosabb funkciójának végeredménye. Az öt különböző járműv útvonalának egyes állomásai láthatóak. Minden járművet külön layer-ben jelenítettem meg, úgy, hogy az egyes layer-eknél külön, a jármű azonosítójára szűrve, kértem le az egyes járművekhez tartozó információkat. Minden egyes, a térképen felvett pont egy megállót reprezentál, amelyet valós időben, az adatbázisba bekerülés pillanatakor lekérhető és megjeleníthető. A megálló pontjának GeoMap térképen el kellett mentenem annak latitude (földrajzi szélesség) és longitude (földrajzi hosszúság) értékei.

6. ábra Késések ábrázolása

A fenti ábrán az egyes megállókra csoportosítva láthatóak ez egyes járművek (külön színnel jelölve) késéseinek értékei. A lenti ábra az egyes járművekre rászűrt késési adatokat mutatja.



7. ábra A késési értékek a negyedik (veh_4) járműre szűrve.

Kihívások számomra és az ezekre adott megoldásaim

A projekt során több technikai kihívással is szembesültem, amelyek leküzdése elengedhetetlen volt a rendszer működőképessé tétele érdekében

Az egyik első probléma az volt, hogy a Grafana és a szimulátor konténer nem tudta elérni az InfluxDB-t localhost címen, mivel a Docker konténerek saját izolált hálózatban futnak. A megoldás az volt, hogy a docker-compose.yaml fájlban definiáltam egy közös sim-net nevű hálózatot, amit konténerhez hozzárendeltem. Ezután az InfluxDB elérhetővé vált a többi konténer

számára a http://influxdb:8086 címen, ahol az influxdb a szolgáltatás neve. Ezt a címet használva sikeresen elértem az adatbázist a Grafana konténeréből a DataSource bekonfigurálása, illetve a későbbi lekérdezések kivitelezésének céljából.

Az InfluxDB nem engedélyezte az egyszerű felhasználónév-jelszó alapú kapcsolódást, hanem kötelező volt a tokennel való hitelesítés az InfluxDB és a Grafana között vagy egyéb fajtájú autentikáció. Ennek megfelelően a konténer inicializálásakor be kellett állítanom egy admin tokent, amelyet később a Grafana adatforrás konfigurációjához is felhasználtam. Eleinte nehézséget okozott, hogy a Grafana alapértelmezés szerint nem tokenes hitelesítéssel próbál kapcsolódni, de végül sikerült megadni a token fejlécet, amelyet az adatforrás konfiguráció fájlban is használtam.

A Grafana GeoMap nevű vizualizációs nézetén több jármű adatainak megjelenítését először egy közös Flux lekérdezéssel próbáltam megoldani, de nem sikerült, ezért azt találtam ki, hogy készítek minden járműhöz egy-egy layer-t a nézeten, majd minden layer-t az adott járműhöz tartozó adatok Flux lekérdezésével töltök fel. Ez egy jól működő megoldást eredményezett, de nem a legoptimálisabb, hiszen nagy számú adatnál, nagy számú layer-t és lekérdezést jelent.

Összegzés

A projekt célja nem csupán egy szimuláció létrehozása volt, hanem egy moduláris, könnyen bővíthető rendszer kialakítása volt, amely a modern szoftverfejlesztés gyakorlatait követi (konténerizáció, verziókövetés, automatizáció). Ez a struktúra lehetőséget teremt a jövőbeni fejlesztésekre, például valós adatok integrálására vagy gépi tanulás alapú elemzések hozzáadására.

Mindemellett a projekt elkészítése során számos olyan problémába ütköztem, amelyek megoldásával olyan hasznos tudást tudtam elsajátítani, amit remélek, hogy a közeljövőben nagyobb vagy más jellegű projektek elkészítésekor is sikeresen tudok majd alkalmazni.

Az elkészült porjekt verziókövetett fájljai az alábbi linken található publikus Github Repository-ban találhatóak: https://github.com/Peter849/BKK_GTFS_DOCKER_HW