



M Ű E G Y E T E M 1 7 8 2

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Kozák Ágota Boglárka

**DIAGNOSZTIKAI NAPLÓ
FELDOLGOZÁS ÉS
VIZUALIZÁCIÓ
MIKROSZOLGÁLTATÁSOK
ARCHITEKTÚRÁBAN**

KONZULENS

Dr. Dudás Ákos

BUDAPEST, 2021

Tartalomjegyzék

Összefoglaló	5
Abstract.....	6
1 Bevezetés	7
2 A feladatkiírás pontosítása és részletes elemzése	8
2.1 A naplózó rendszer ismertetése	8
2.2 Az elkészítendő rendszer ismertetése	9
3 A rendszer megtervezése	10
3.1 A tervezési szempontok bemutatása	10
3.1.1 Skálázhatóság.....	10
3.1.2 Könnyű kiterjeszthetőség.....	11
3.1.3 Megbízhatóság és hibatűrés	11
3.1.4 Modularitás	11
3.1.5 Egyéb szempontok.....	12
3.2 A választott architektúra és technológiai megoldások.....	12
3.2.1 Mikroszolgáltatások architektúra.....	12
3.2.2 A választott technológiák bemutatása.....	14
4 Implementáció	19
4.1 A fejlesztés menete	19
4.1.1 Docker és Visual Studio Code.....	19
4.1.2 Verziókezelés.....	22
4.1.3 Kódminőség ellenőrzése	23
4.2 A szolgáltatások megvalósítása	25
4.2.1 Parser mikroszolgáltatás	25
4.2.2 Postprocessor mikroszolgáltatás	35
4.2.3 ElasticUploader mikroszolgáltatás	42
4.3 Adattárolás és vizualizáció	45
4.3.1 Adattárolás.....	45
4.3.2 Vizualizáció	46
4.4 Tesztelés.....	51
4.4.1 Unit tesztek	51
4.4.2 Integrációs tesztek.....	56

4.4.3 CI automatizált teszt futtatás.....	58
5 Konklúzió és továbbfejlesztési lehetőségek.....	60
Irodalomjegyzék.....	61

HALLGATÓI NYILATKOZAT

Alulírott **Kozák Ágota Boglárka**, szigorló hallgató kijelentem, hogy ezt a diplomatervet 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 hitelesített 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.

Kelt: Budapest, 2021. 12. 09.

.....
Kozák Ágota Boglárka

Összefoglaló

Napjainkban a legtöbb szoftver – különösen egy komplex rendszer esetében – a működése közben bekövetkezett eseményeket, esetleg hibákat naplófájlokba jegyzi fel, amely fájlok és az általuk tartalmazott adatok fontos szerepet játszanak a fejlesztési, üzemeltetési, illetve hibakeresési folyamatokban, hiszen értékes információkat nyerhetünk ki belőlük az alkalmazások működésével kapcsolatban.

A projektem célja egy okos elektromos hálózatban működő mérők és a rajtuk futó szoftverek által előállított naplófájlok feldolgozására, a kinyert adatok tárolására, valamint vizualizációjára képes rendszer megtervezése és megvalósítása volt. A tervezés és megvalósítás során arra törekedtem, hogy a kész rendszer amellett, hogy segítse a hálózatban bekövetkező események megértését, képes legyen kezelni a potenciálisan nagy feldolgozandó adatmennyiséget, illetve később könnyedén kiterjeszthető legyen, hiszen a naplófájlok formátuma nem rögzített és várhatóan változni fog.

A diplomatervezés feladatom keretein belül egy mikroszolgáltatások architektúrát követő rendszert valósítottam meg, melyben három saját fejlesztésű szolgáltatás működik, melyeket *Go* nyelven implementáltam. A mikroszolgáltatások közötti kommunikációt *RabbitMQ* használatával oldottam meg, ezzel elősegítve a rendszer áteresztőképességének növelését, és biztosítva a kommunikáció során küldött üzenetek megismételhetőségét. Az adatok tárolására egy *Elasticsearch* adatbázist választottam, amely egyrészt nem igényel sémadefiniációt, így a változékony adatoknál jó választásnak bizonyult, másrészt a benne tárolt adatokhoz használható a *Kibana*, mint adatvizualizációs eszköz. A naplófájlokból kinyert adatokhoz öt különböző példa megjelenítést dolgoztam ki a bekövetkezett események elemzésének megkönnyítéséhez.

Az egyes mikroszolgáltatások *Docker* konténerekben futnak, a rendszer és a benne lévő összes szolgáltatás definiálásához pedig *Docker Compose*-t használtam. A konténer környezetben történő fejlesztést a *Visual Studio Code Remote-Containers* bővítményével oldottam meg.

Az elkészült alkalmazáshoz unit teszteket és integrációs teszteket is készítettem, amelyek automatizáltan kerülnek futtatásra a projekthez tartozó *GitHub repository*-ban beüzemelt *CI* rendszer segítségével.

Abstract

In today's world, most software – especially in case of a complex system – logs events and errors that occurred during its operation in log files. These log files and the data they contain play an important role in the development, operation and debugging processes, as we can extract valuable information about the operation of the applications from them.

The aim of my project was to design and implement a system capable of processing log files generated by software running on the smart meters and other components of a smart electrical grid, and that is also capable of storing and visualizing the extracted data. While designing and implementing this log processing system, I tried to prepare the system to be able to handle the potentially large amount of data needed to be processed, as well as to make it easy to extend in the future, as the format of the log files is not set and is expected to change.

During this project, I implemented a system following a microservices architecture, consisting of six individual services, from which I implemented three using the *Go* programming language. I solved the communication between the microservices using *RabbitMQ*, thus helping to increase the throughput of the system and ensuring the repeatability of the messages sent during this communication. I chose *Elasticsearch* to store the processed data, which does not require a schema definition, so it seemed to be a good choice for the variable data coming from the log files, and by using *Elasticsearch*, I had the option to use *Kibana* as a data visualization tool for the processed log files. For the data extracted from the log files, I developed five different example visualizations to facilitate the analysis of the events and errors that occurred in the smart meter system.

Each microservice runs in a *Docker* container, and I used *Docker Compose* to define the system and all contained services. To make the development process in this container environment easier, I used the *Visual Studio Code Remote-Containers* extension, which enables the use of a *Docker* container as a full-featured development environment.

I also created unit tests and integration tests for the application, which are run automatically using the CI system installed in the project's GitHub repository.

1 Bevezetés

A napjainkban működő alkalmazások jelentős része a működése közben bekövetkezett hibákat/eseményeket naplófájlokba rögzíti. Ezeknek a naplófájloknak nagy jelentősége van az alkalmazások, vagy bonyolultabb rendszerek karbantartása, fejlesztése során, mivel értékes információkat tartalmaznak a múltban bekövetkezett eseményekről, hálózati kommunikációról, hibákról.

A fájlok feldolgozása azonban nem mindig egyszerű feladat, hiszen a naplófájlok mérete igen nagyra nőhet, főleg komplex rendszerek esetében, így emberi szemmel átolvasva nem, vagy csak nagyon nehezen nyerhető ki belőlük hasznos információ. Ezért ilyen célokra hasznos egy napló feldolgozó rendszer, amely képes kezelni a nagy méretű fájlokat, releváns információkat kinyerni belőlük, valamint képes emberi fogyasztásra alkalmas formában megjeleníteni azokat, így segítve a fejlesztők, karbantartók számára a hibakeresést.

Egy használati eset egy ilyen napló feldolgozó rendszerre például az okos városokban lévő intelligens elektromos hálózathoz tartozó okos mérők rendszerének karbantartása és fejlesztése. Ez egy komplex rendszer, amely üzemeltetésében több hálózati eszköz és szoftver is szerepet játszik. A naplófájlok így több szoftvertől is származhatnak, emiatt a formátumuk is eltérő lehet, valamint egy konkrét hibaeset megtalálásához előfordulhat, hogy több naplófájl adatait kell összeegyeztetnünk, amely átolvasással nem megoldható feladat.

A diplomatervezés projektem keretein belül egy a fentebb leírtaknak megfelelő napló feldolgozó rendszer megtervezésével és megvalósításával foglalkoztam. A projekt célja továbbá ennek a problémakörnek és a megoldáshoz használható eszközöknek és technológiáknak a megismerése is volt.

A dolgozatomban először a feladatkiírást fejtem ki bővebben, majd a harmadik fejezetben bemutatom, hogyan terveztem meg a feladatnak megfelelő szoftvert és annak architektúráját, és a megvalósításhoz milyen technológiai megoldásokat választottam. A negyedik fejezetben a megvalósítás részleteit ismertetem, majd az ötödik fejezetben kitérek a jövőbeli továbbfejlesztési lehetőségekre is.

2 A feladatkiírás pontosítása és részletes elemzése

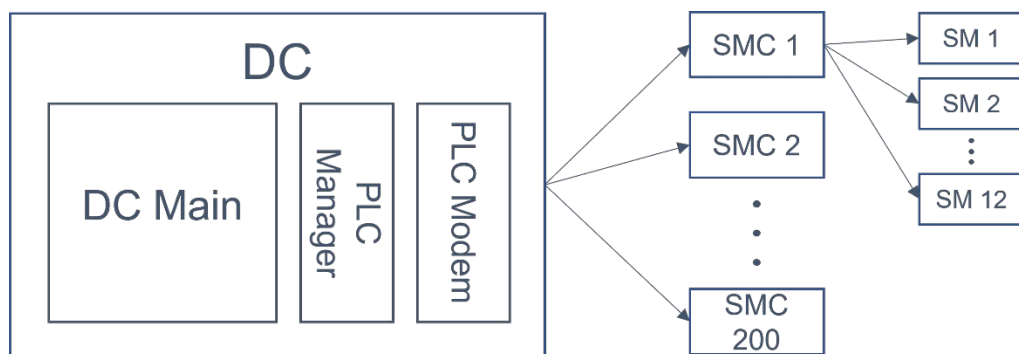
A feladat egy - a bevezetésben már ismertetett - naplófeldolgozó és megjelenítő alkalmazás megtervezése és elkészítése, konkrétan pedig okos falvakban elhelyezett okos mérők, által előállított napló fájlok és egyéb diagnosztikai adatok feldolgozása a későbbi hibakeresés megkönnyítésének céljából, illetve ezek vizualizációja.

2.1 A naplózó rendszer ismertetése

Ebben az alfejezetben a naplókat előállító rendszert és annak részeit ismertetem.

A napló fájlokat okos falvakban, ún. „*Smart Village*”-ekben elhelyezett okos mérők, „*Smart Meters*” és a hozzájuk tartozó szoftverek generálják. Minden okos faluban a rendszer a következő komponensekből épül fel: [1]

- Okos mérők (*Smart Meters - SM*)
- Okos mérő vezérlők (*Smart Meter Controllers – SMC*)
- Egy elosztó vezérlő (*Distribution Controller – DC*)
- *PLC modem* és *PLC Manager* (az *SMC*-k és a *DC*-k közötti kommunikációt biztosítja)



2.1. ábra: Az okos mérők hálózatának egyszerűsített váza

Az *SMC*-k *G3-PLC technológia* segítségével kommunikálnak a *DC*-kkel. A *G3-PLC* egy keskeny sávú hálózati kommunikációs technológia, mely lehetővé teszi a nagy sebességű, nagyon megbízható, nagy hatótávolságú kommunikációt a meglévő villamosvezeték-hálózaton. [2]

A rendszer jelenleg a *DC Main* által, és a *PLC Manager* által generált napló fájlok feldolgozására és vizualizációjára koncentrál.

2.2 Az elkészítendő rendszer ismertetése

A diplomatervezés feladatom keretein belül elkészítendő napló-feldolgozó rendszernek számos funkcionális és nem-funkcionális követelménynek kell megfelelnie, amelyeket figyelembe kellett vennem a tervezés és az implementációs folyamat közben.

A rendszernek képesnek kell lennie nagy méretű és változatos naplófájlok feldolgozására, az adatok elmentésére és vizualizációjára is. A feldolgozáshoz a rendszernek képesnek kell lennie a napló fájlok felolvasására, majd az egyes naplózott események között valamilyen kapcsolat felderítésére. A fájlok formátumának változékonysága miatt fontos, hogy a feldolgozás olyan módon legyen megvalósítva, hogy az a későbbiek során könnyen kiterjeszthető, megváltoztatható maradjon. Ezen kívül a nagy mennyiségű adat feldolgozása megköveteli, hogy a feldolgozás hatékonyan és skálázható módon történjen, amelynek biztosítására a mikroszolgáltatások architektúra nyújthat megoldást.

A naplókat előállító eszközök és szoftverek diagnosztizálásának megkönnyítéséhez a feldolgozott adatok megjelenítésére is szükség van, ehhez pedig ki kell dolgozni azt a vizualizációs módszert, amely leginkább segíti az adatok megértését.

A feladat része a feldolgozó rendszer megtervezése és azoknak a technológiai megoldásoknak a kiválasztása, amelyek leginkább segítik elérni a kitűzött célokat, illetve ezek alapos megismerése, használatuknak elsajátítása.

3 A rendszer megtervezése

A logfeldolgozó rendszer megvalósításában az első lépés a tervezés volt. A tervezési fázis mindig nagyon fontos egy szoftver fejlesztése során, hiszen egy jó terv megkönnyítheti a fejlesztést és a karbantartást a későbbiekben, míg egy rossz terv, vagy annak hiánya nagyon megnehezítheti a fejlesztő életét. Ebben a fejezetben ismertetem, hogy milyen szempontok alapján milyen tervezői döntéseket hoztam, és bemutatom az elkészült rendszer architektúrájának tervét, illetve, hogy milyen technológiai megoldásokat választottam a megvalósításhoz.

3.1 A tervezési szempontok bemutatása

Egy szoftver megtervezésekor számos szempontot kell figyelembe vennünk, hogy a terveink alapján megvalósított szoftver a lehető legjobban kielégítse a vele szemben támasztott elvárásokat, követelményeket, illetve a szoftver életciklusa során leegyszerűsítse a karbantartást és esetleges új funkciók bevezetését. Ez az alfejezet azokat a szempontokat mutatja be, amelyekre különös figyelmet fordítottam a saját feladatom keretein belül elkészítendő rendszer megtervezésekor.

3.1.1 Skálázhatóság

A skálázhatóság egy szoftvernek vagy eszköznek azon képessége, hogy növelni tudja az áteresztőképességét a felhasználói igények alapján. Ha már a fejlesztés elejétől fogva prioritásként kezeljük, az alacsonyabb karbantartási költségeket jobb felhasználói élményt és használhatóságot eredményez, ezért ezt már a rendszer megtervezése közben is igyekeztem figyelembe venni.

A feladatorként fejlesztendő rendszerben a skálázhatóság a nagy és gyorsan változó adatmennyiség miatt játszik jelentős szerepet, ugyanis a feldolgozandó naplófájlok akár ötvenezer sorból is állhatnak, ezen kívül a fájlok számossága is jelentős, valamint az adatok folyamatosan generálódnak, tehát mindig egyre több lesz belőle. A feldolgozásnak képesnek kell lennie kezelni azt is, ha ez a bemeneti adatmennyiség hirtelen megnő, például a feldolgozás párhuzamosításával, illetve mikroszolgáltatások architektúrában egy-egy újabb feldolgozó szolgáltatás-példány hozzáadásával, hogy a megnövekedett munkamennyiséget többfelé lehessen szétosztani.

3.1.2 Könnyű kiterjeszthetőség

Egy szoftver kiterjeszthetősége egy olyan tulajdonság, amely azt írja le, hogy a szoftvert milyen mértékű erőfeszítéssel tudjuk bővíteni például egy új funkcióval, vagy a meglévő funkcionalitás módosításával. Ha egy rendszert hosszúéletűre szeretnénk tervezni, akkor fontos odafigyelnünk, hogy az implementációnk bővíthető legyen, hiszen idővel egyre több új funkciót, változtatást kellhet beletennünk a termékbe.

Ez a tulajdonság azért nagyon fontos a naplófeldolgozó rendszerben, mert maguk a napló fájlok is nagyon változatosak, a formátumuk nincs kikötve, ezen kívül azt sem feltétlenül tudjuk előre, hogy milyen információkat kell kinyernünk egy adott naplófájlból, vagy annak egy sorából. Ha egy fájlban megváltozik például az időbélyeg formátuma, nem szeretnénk, ha a változás az egész rendszert érintené, jó lenne, ha csak a felolvasást végző komponensnek a dátumok olvasásáért felelős kis részében kellene módosítanunk, a többi komponens pedig változatlan maradhat.

3.1.3 Megbízhatóság és hibatűrés

Az általam elkészítendő rendszerben ideális lenne, ha egy hiba a feldolgozás közben nem akasztaná meg az egész rendszert, például, ha egy adat egységre a feldolgozás valamiért elesik, attól ne álljon le az egész rendszer és ne veszítsük el az addig a pontig feldolgozott adatokat. Erre a problémára is megoldást nyújthat a mikroszolgáltatások architektúra, ahol ezek a felelősségek külön szolgáltatásokba vannak szervezve. Ekkor lehet akár több példányunk is a feldolgozó szolgáltatásunkból, ezen kívül a feldolgozó komponens hibája miatt az adatbázisba mentés és más szolgáltatások nem fognak leállni. Továbbá a szolgáltatásaink között üzenetsor alapú kommunikáció használata garantálja az újra próbálhatóságot.

3.1.4 Modularitás

A modularitás azt jelenti, hogy a szoftver projektünk kódját egy hatalmas darab helyett több kisebb modulra bontjuk. Ez leegyszerűsíti a fejlesztést, hiszen egyszerre csak kisebb és kevésbé bonyolult kódrészleteken kell dolgoznunk.

A saját feladatomban például a fájlok felolvasása, az adatok feldolgozása majd adatbázisba mentése egymástól elválasztható folyamatok, ezeket külön mikroszolgáltatásokba szervezve növelhetem a rendszer modularitását, egyszerűsíthetem a kód bázis megértését így megkönnyítve a fejlesztést és karbantartást.

3.1.5 Egyéb szempontok

A fentiek mellett érdemes például megemlíteni a tesztelhetőséget, mint tervezési szempontot. A kódunk különálló modulokba vagy szolgáltatásokba szervezése a jobb tesztelhetőséghez is hozzájárul, mivel a felelősségek szétválasztása miatt könnyebb megállapítani a teszteseteket, illetve egy változtatás miatt nem szükséges a teljes rendszert újra tesztelni, elég lehet a megváltozott részen újra futtatni a tesztek.

Egy további szempont lehet a programozási nyelvhez vagy technológiához való kötöttség elkerülése. Például, ha bizonyos naplófájljaink vagy adatunk olyan formátumban érkezik hozzánk, amely beolvasására egy bizonyos nyelv lenne a legalkalmasabb, viszont nem szeretnénk emiatt az egész alkalmazást azon a nyelven megírni (mert például az a nyelv kevésbé alkalmas párhuzamosítás megvalósítására), akkor ezeket a feladatokat külön szolgáltatásokba szervezve külön-külön választhatjuk meg a használni kívánt programozási nyelvet. Ha például másik adatbázist vagy megjelenítési módszert szeretnénk használni, akkor ezeknek a cseréje is egyszerűbb mikroszolgáltatások architektúra használata mellett.

3.2 A választott architektúra és technológiai megoldások

Az előző alfejezetben bemutatott szempontok alapján most ismertetem a rendszer megvalósításához választott architektúrát és technológiai megoldásokat, amelyekkel igyekeztem megoldást találni az eddigiekben említett kérdésekre. Arra is kitérek majd, hogy a választott technológiák hogyan segítenek megfelelni a fentebb felvázolt szempontoknak.

3.2.1 Mikroszolgáltatások architektúra

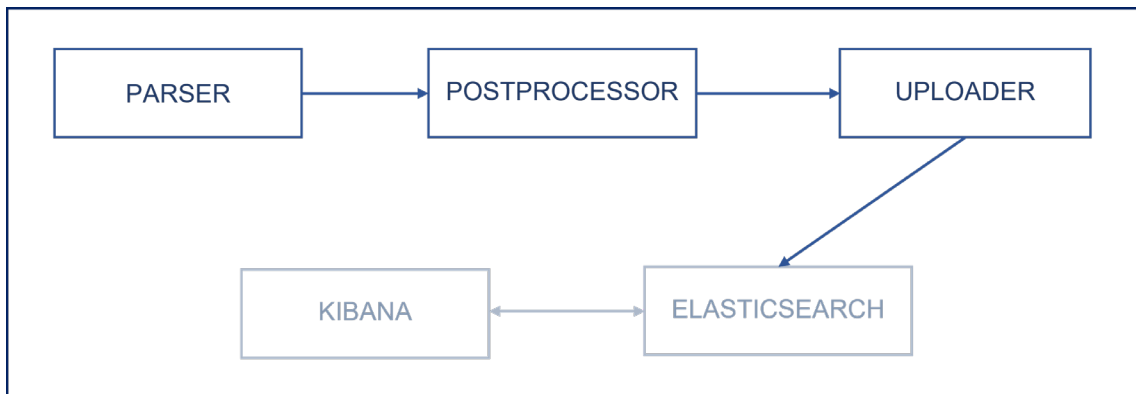
Ahogy az a dolgozatom címéből is kiolvasható, és amire már az előző alfejezetekben is utaltam, a naplófeldolgozó rendszert mikroszolgáltatások architektúrában terveztem, illetve valósítottam meg.

A mikroszolgáltatások architektúra az alkalmazásfejlesztésnek egy olyan megközelítése, amelyben egy nagy alkalmazást moduláris komponensekből vagy szolgáltatásokból építünk fel, szemben a tradicionális megközelítéssel, ahol a szoftvert egyetlen egység alkotja. Minden szolgáltatás (vagy modul) egy adott feladatot vagy üzleti célt támogat, és szinkron vagy aszinkron protokollok, például HTTP/REST vagy AMQP segítségével kommunikál a többi szolgáltatással. Ezzel a megközelítéssel az egyes

szolgáltatásokat egymástól függetlenül, különböző nyelvek és eszközök használatával lehet fejleszteni, tesztelni és telepíteni. Ezen felül a mikroszolgáltatások architektúra használatával gyors skálázhatóságot és jobb hiba izolációt is nyerhetünk.

A feladatkiírásban megfogalmazott követelmények alapján egy olyan feldolgozó csővezetéktervet terveztem, amelyben az egyes szolgáltatások a napló fájlok feldolgozásának egy-egy jól elkülöníthető részét valósítják meg, ezzel is elősegítve a változások kezelését például a fájlok formátumát illetően.

Az alábbi ábra mutatja a tervezett architektúra részeit és közöttük lévő kapcsolatokat. A nyilak a szolgáltatások közötti adatáramlás irányát mutatják, a saját fejlesztésű komponenseket sötétkék, míg az egyéb felhasznált szolgáltatásokat szürke színnel jelöltem.



3.1. ábra: Egyszerűsített architektúra terv

A *Parser* szolgáltatás végzi a napló fájlok feldolgozásának legelső lépését: a felolvasást. A felolvasott sorokat ezután továbbítja a *Postprocessor* szolgáltatásnak, amely utólagos feldolgozást végez a kapott bejegyzéseken, annak érdekében, hogy minél több információt tudjunk meg a naplófájlokból. A feldolgozás utolsó lépését, az adatbázisba mentést az *ElasticUploader* nevű komponens végzi, amelynek egyetlen felelőssége, hogy a kapott adatokat feltöltse egy *Elasticsearch* adatbázisba. Végül a feldolgozott adatok vizualizációja a *Kibana* segítségével történik.

A fájlok felolvasásának és az utófeldolgozásnak a szétválasztása azzal az előnnyel jár, hogy a napló fájlok formátumának változása nem érinti az utófeldolgozás kódját, illetve ilyen módon a felolvasó szolgáltatás relatíve gyors tud maradni, az utófeldolgozás folyamatát pedig tudjuk skálázni több *Postprocessor* példány bevezetésével. Az adatbázisba mentés különválasztásának egyik oka, hogyha le szeretnénk cserélni az

adattárolásra használt technológiát, esetleg több helyre is szeretnénk menteni az adatokat, akkor ez a változás nem fogja a feldolgozó komponens megváltoztatását eredményezni.

Az egyes komponensek működéséről részletesebben a negyedik fejezetben számolok be.

3.2.2 A választott technológiák bemutatása

Ebben az alfejezetben arról számolok be részletesen, hogy az előzőekben bemutatott architektúra egyes elemeihez milyen technológiákat választottam, a szolgáltatásokat milyen nyelven implementáltam, illetve, hogy az egyes komponensek közötti kommunikáció hogyan valósul meg.

3.2.2.1 Programozási nyelv

A logfeldolgozó rendszer szolgáltatásainak implementációjához a *Go* programozási nyelvet választottam.

A *Go* egy nyílt forráskódú programozási nyelv, amely megkönnyíti az egyszerű, megbízható és hatékony szoftverek készítését. A *Google* által lett kifejlesztve, az első verzió 2009-ben jelent meg. [3] Szintaxisa hasonlít a *C* nyelvhez, viszont ez a programozási nyelv olyan eszközöket is tartalmaz, amelyek lehetővé teszik a biztonságos memóriakezelést, az objektumok kezelését, a szemétygyűjtést és a statikus típusok használatát.

Ezen kívül a *Go* nyelv beépített megoldásokat nyújt a párhuzamosítás megvalósítására. Ilyenek például a *goroutine*-ok és a *channel*-ek. Egy *goroutine* egy *lightweight thread*, amit a *Go runtime* kezel. Minden *goroutine* az adott programban jelen lévő más *goroutine*-októl függetlenül és velük egyidejűleg kerül végrehajtásra. A *goroutine*-ok ugyanabban a címtérben futnak, ezért a megosztott memóriához való hozzáférést szinkronizálni kell, erre a célra használhatók egyéb eszközök mellett a csatornák (*channel*). [4] Egy *channel* egy típusos csatorna, amelyen keresztül értéket küldhetünk vagy fogadhatunk a '<->' operátor segítségével. Ezen nyelvi elemek által nyújtott előnyöket igyekeztem kihasználni a napló feldolgozó rendszer megvalósítása során a minél jobb teljesítmény elérése érdekében.

A *Go* nem szabad formátumú nyelv: a konvenciói sok formázási részletet határoznak meg, beleértve a behúzás és szóköz használatának módját. Ezen konvenciók betartására a *Go* számos formatter-t és linter-t biztosít a fejlesztők számára.

A *Go* nyelvet hatékonysága, és a párhuzamosítás megvalósítására használható beépített megoldásai miatt választottam ennek a nagy adatmennyiség feldolgozására szánt alkalmazásnak a megvalósításához.

Mindhárom mikroszolgáltatást – a *Parser*-t, a *Postprocessor*-t és az *ElasticUploader*-t is *Go* nyelven implementáltam.

3.2.2.2 Adattárolás és vizualizáció

Az adatok vizualizációjára a beépített keresési és vizualizációs képességei miatt a *Kibana*-t választottam, ez a választás pedig meghatározta azt is, hogy az adatok tárolása egy *Elasticsearch* adatbázisban kell, hogy történjen.

Az *Elasticsearch* egy elosztott, dokumentum alapú, nyílt forráskódú kereső és elemző motor minden típusú adat számára, beleértve a szöveges, numerikus, földrajzi, strukturált és strukturálatlan adatokat is. [5] Az *Elasticsearch* „sémaruhalmas” abban az értelemben, hogy nem szükséges egy előre definiált sémát megadnunk az adatoknak, elég a JSON dokumentumot átadnunk neki, amiből megpróbálja kitalálni az adataink típusát. [6] Ez a napló feldolgozó rendszer kapcsán előnyös, hiszen a napló formátumok és kinyerendő adatok változékonyak, így az állandó séma módosítás igen kényelmetlenné tenné a fejlesztést.

Az *Elasticsearch*-el történő interakciókhoz a ***go-elasticsearch*** nevű kliens könyvtárat használtam, amely egy *Go* nyelven implementált *ES* klienst biztosít a használóinak. [7]

A *Kibana* egy nyílt forráskódú frontend alkalmazás, amely az *Elastic Stack* tetején helyezkedik el, és keresési és adatmegjelenítési lehetőségeket kínál az *Elasticsearch*-ben indexelt adatokhoz. [8] Az *Elastic Stack* diagramkészítő eszközeként is tekinthetünk rá. Főként nagy mennyiségű adat elemzésére használják, rengeteg vizualizáció készítésére alkalmas, például vonaldiagram, oszlopdiaagram, kördiagram, hőtérkép, régiótérkép, koordinátatérkép stb. Ez a lista tovább bővíthető különböző *Kibana plugin*-ek használatával, például a ***kibana-milestones-vis*** plugin segítségével mérföldkő-jellegű vizualizációkat készíthetünk mondjuk események ábrázolásához. [9] A *Kibana* használata mellett szól továbbá a könnyű használhatóság és érthetőség.

3.2.2.3 Kommunikáció

A különböző mikroszolgáltatások közti kommunikációt *RabbitMQ* segítségével oldottam meg.

Több tízezer felhasználóval a *RabbitMQ* az egyik legnépszerűbb nyílt forráskódú üzenetközvetítő. [10] Eredetileg az *AMQP 0-9-1* (*Advanced Message Queuing Protocol*) protokollhoz lett fejlesztve, de emellett támogatja még a *STOMP*, *MQTT* és *AMQP 1.0* protokollokat is. [11] Az *AMQP 0-9-1* egy bináris üzenetküldési protokoll és szemantikai keretrendszer mikroszolgáltatásokhoz és vállalati üzenetküldéshez. A klienseknek viszonylag könnyű implementálni ezt a protokollt, hiszen rengeteg kliens könyvtár áll rendelkezésre számos különböző programozási nyelvhez és környezethez, köztük a *Go* nyelvhez is. [12]

Alternatív kommunikációs megoldásnak felmerült még a *HTTP* és *REST* is, de a választásom végül az üzenetsor alapú kommunikációra esett. Ennek fő oka a teljesítmény maximalizálása volt: a *RabbitMQ* nagy áteresztőképességgel rendelkezik és a termelőktől érkező üzeneteket várakozási sorokban (*queue*) tárolja, ameddig egy fogyasztó el nem kezdi feldolgozni azokat. A *HTTP*-vel szemben az üzenetsor alapú kommunikáció aszinkron természetű, így az üzeneteket küldő termelő szolgáltatásoknak nem kell tudniuk a fogyasztók működéséről, sem bevárni a válaszukat ahhoz, hogy folytathassák a végrehajtást. A *RabbitMQ* garanciát ad az üzenetek továbbítására: ha egy üzenetet nem sikerül kézbesíteni, akkor újra bekerül a várakozási sorba, továbbá a perzisztens kézbesítési mód használatával az üzeneteink a *RabbitMQ* szerver újraindítását is túléljük. [13] Ezek alapján az üzenetsor alapú kommunikáció nagyobb megbízhatóságot ad nagyszámú konkurens felhasználó (esetünkben konkurens goroutine-ok és szolgáltatások, amelyek az adatfeldolgozást végzik) esetén, mint amit egy *RESTful API* és *HTTP* használata biztosítana.

Egy *RabbitMQ* alapú kommunikáció során a termelők (*message producer*) üzeneteit a *RabbitMQ* egy várakozási sorba (*queue*) teszi, ahonnan a fogyasztók (*consumer*) kiveszik az üzeneteket feldolgozásra. A termelők úgynevezett **exchange**-eknek küldik az üzeneteket, és az exchange-ek felelősek azért, hogy az üzeneteket *header-attribútumok*, *binding*-ok és *routing key*-k segítségével különböző sorokba irányítsák. Minden üzenetsor egy *exchange*-hez van kötve egy úgynevezett *binding key* segítségével, az *exchange* pedig az üzenettel együtt megkap egy *routing key*-t, hogy az alapján tudja tovább irányítani az adott üzenetet.

Négy különböző típusú exchange-t használhatunk:

- **Direct:** Az üzeneteket pontosan azokba a sorokba továbbítja, amelyeknél a *binding key* értéke teljesen megegyezik az üzenettel kapott *routing key* értékével. [14]
- **Topic:** Hasonlóan a *direct* típushoz, itt is a *binding key* és a *routing key* értéke kerül összehasonlításra, viszont itt lehetőségünk van helyettesítő karakterek használatára a *binding key* értékének megadásakor, pl.: „lazy.*.#”, ahol a '#' nulla vagy egy szót, a '*' pedig pontosan egy szót helyettesít, így ez a minta megfelelhet a „lazy.rabbit” és a „lazy.red.rabbit” értékeknek. [15]
- **Headers:** Fejléceket használ az üzenetek irányításához.
- **Fanout:** Ez az *exchange* típus minden hozzá kötött sornak továbbítja a kapott üzeneteket, tekintet nélkül a *routing key* értékére. [16]

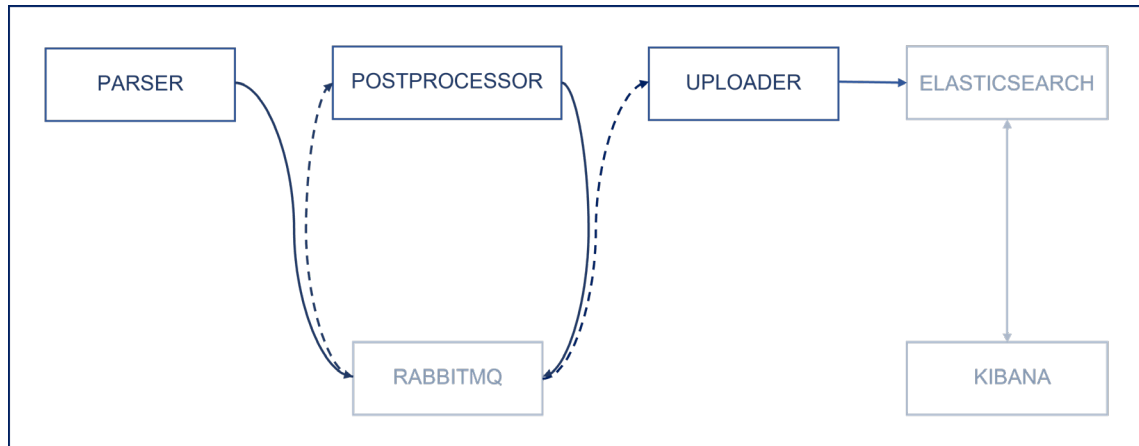
A napló feldolgozó rendszerben minden esetben **direct** típusú exchange-t használtam, mivel az én esetemben a cél, hogy minden üzenetet pontosan egy szolgáltatás kapjon meg, viszont, ha a jövőben esetleg be kell vezetni különböző jellegű feldolgozó szolgáltatásokat is, akkor a *topic* exchange-nek is jó hasznát lehetne venni.

3.2.2.4 Docker

Minden szolgáltatás *Docker* konténerben fut, amelyek összehangolására *Docker Compose*-t használtam. Ezzel a megoldással az egész feldolgozó rendszert egyetlen *docker-compose.yml* fájljal le tudtam írni, ahol minden, a rendszer számára szükséges szolgáltatást is felsoroltam (ezek az *Elasticsearch*, a *Kibana* és a *RabbitMQ*). A *Docker* konténerekben történő fejlesztés menetéről a negyedik fejezetben részletesebben is szót ejtek.

3.2.2.5 Összegzés

A következő ábrán a komponensek közötti kommunikáció részleteivel kiegészített rendszer látható. A szaggatott görbe nyilak üzenetek fogyasztását (*consume*), míg a folytonos görbe nyilak az üzenetek termelését (*publish*) jelzik.



3.2. ábra: A rendszer architektúrájának ábrája a kommunikáció részleteivel kiegészítve

4 Implementáció

A naplófeldolgozó rendszer architektúrájának megtervezése és a megfelelő technológiák kiválasztása után a rendszer megvalósítása következett. Az előző fejezetben bemutatott tervekben szereplő architektúra, az egyedi megoldások és a kiválasztott technológiák okoztak némi fejtörést a megvalósítás folyamata során, mint például a kód strukturálása, megfelelő kliens könyvtárak keresése, a tesztelés megvalósítása, illetve a *Docker* konténerekben történő fejlesztés. Ez a fejezet betekintést ad az elkészült rendszer implementációjába, és az említett nehézségek megoldásaiba.

4.1 A fejlesztés menete

Ebben az alfejezetben bemutatom, hogy milyen fejlesztői környezet használatával valósítottam meg a három mikroszolgáltatást, hogyan oldottam meg a hibakeresést konténerizált környezetben, hogyan biztosítottam a verziókezelést, illetve a kódminőség ellenőrzését fejlesztés közben.

4.1.1 Docker és Visual Studio Code

A fejlesztői környezet összerakásának egy külön alfejezetet szántam a dolgozatomban, mivel a választott technológiai és architektúrai megoldások – a mikroszolgáltatások architektúra és *Docker* konténerekben történő futtatás – extra időráfordítást igényeltek ahhoz, hogy a fejlesztés, a hibakeresés és a tesztelés is gördülékenyen menjen.

A fejlesztői környezet és a kód szerkesztő kiválasztásában a legnagyobb szerepet a *Go* programozási nyelvhez kapcsolódó szokások és a *Docker* konténerek használata játszotta. A *Go* ökoszisztéma számos szerkesztő plugin-t és IDE-t (*Integrated Development Environment*, integrált fejlesztői környezet) kínál a jó kódszerkesztési, navigációs, tesztelési és hibakeresési élmény biztosítása érdekében, például: [17]

- **Visual Studio Code:** elérhető *Go* kiterjesztés, amely támogatja a *Go* programozási nyelv használatát.
- **GoLand:** önálló IDE-ként, vagy az *IntelliJ IDEA Ultimate* bővítményeként is elérhető.
- **vim**, a *vim-go* plugin használatával.

A fenti lehetőségek közül a **Visual Studio Code**-ra esett a választásom, egyebek mellett azért, mert ennek a kód szerkesztőnek a használatával már volt tapasztalatom, és nem kellett egy teljesen új szerkesztőt telepítenem és megismernem.

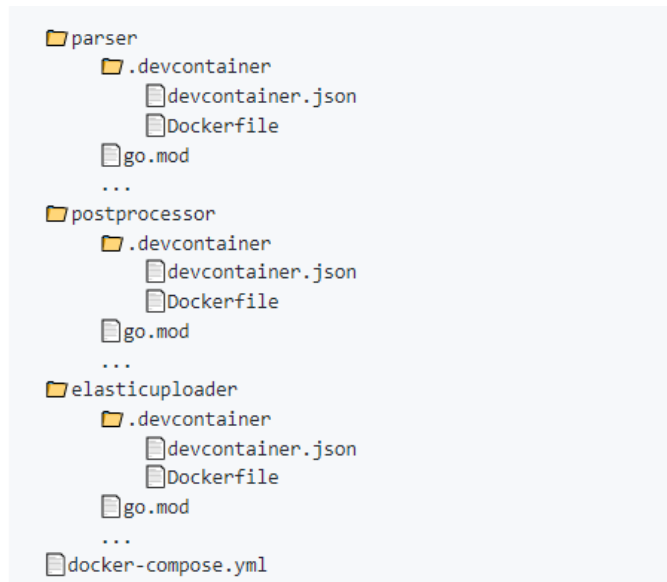
A Docker konténerekben történő fejlesztés és hibakeresés megoldására a *Visual Studio Code Remote-Containers* bővítménye adott választ. A *Remote-Containers* bővítmény lehetővé teszi egy *Docker* konténer teljes értékű fejlesztői környezetként való használatát. [18] Ez azért előnyös megoldás a lokálisan, konténerek használata nélkül történő fejlesztéshez képest, mert ezzel a megoldással elég egy *docker-compose.yml* fájlban definiálni minden függőséget, mint például a *RabbitMQ*, az *Elasticsearch* vagy a *Kibana*, és ezek automatikusan elindulnak a fejlesztői környezet indításakor. Ezáltal, ha például egy másik fejlesztőnek is dolgoznia kell a projekten, akkor nem szükséges extra időt tölteni azzal, hogy a felhasznált szolgáltatásokat telepítsük az ő számítógépére is, elég, ha a *Visual Studio Code* (a szükséges kiterjesztéssel) és a *Docker* telepítve van. Ennek segítségével az is biztosítva van, hogy minden fejlesztő a megfelelő verziókat használja a szükséges technológiákból. Ezen kívül a fejlesztéshez szükséges környezeti változók is automatikusan beállításra kerülnek.

A *Remote-Containers* bővítmény elindít egy fejlesztői konténert (vagy csatlakozik hozzá), amelybe aztán a munkaterület fájljai beilleszthetők a lokális fájlrendszerből. A további bővítmények telepítése és futtatása a konténerben történik, ahol azok teljes hozzáféréssel rendelkeznek az eszközökhöz, a platformhoz és a fájlrendszerhez.

A *Remote-Containers* beüzemeléséhez létre kell hozni egy *.devcontainer* mappát a projekt gyökeri könyvtárában, amelyben el kell helyezni egy *devcontainer.json* fájlt és egy *Dockerfile*-t vagy *docker-compose.yml* fájlt. Ezek alapján létrejön és elindul egy konténer, amely a *devcontainer.json* fájlban tartalmazzott konfigurációnak megfelelően lesz beállítva és telepítve. Ezután a lokális *Visual Studio Code* kapcsolódik a *Visual Studio Code* szerverhez, amely a létrejött fejlesztői konténerben fut. [19]

A saját esetemben három mikroszolgáltatáshoz kellett ezt a környezetet beállítanom, mivel a feldolgozó rendszerben három szolgáltatást fejlesztettem többé-kevésbé egyidejűleg. Alap esetben a *devcontainer.json* fájlban csak egyetlen service definiálására van lehetőség, így a saját rendszerem beállításához több erőfeszítést kellett tennem. Az általam adaptált megoldás erre a problémára az volt, hogy minden szolgáltatáshoz külön-külön definiáltam egy *.devcontainer* mappát, illetve a szükséges

devcontainer.json és *Docker* fájlokat, majd definiáltam egy közös *docker-compose.yml* fájlt, amire minden *devcontainer.json* fájl hivatkozik. Ez a megoldás a *VSCode* ajánlása, amelyről a dokumentációjukban olvastam. [20] Az így létrejött mappaszerkezet látható a következő ábrán.



4.1. ábra: A naplófájl feldolgozó rendszer mappa struktúrájának egyszerűsített ábrája

A *Parser* szolgáltatáshoz tartozó *devcontainer.json* fájl tartalma ekkor a következő (a másik két szolgáltatáshoz tartozó konfiguráció ehhez nagyon hasonló):

```

{
  "name": "Container Postprocessor",
  "dockerComposeFile": ["../..../docker-compose.yml"],
  "service": "container-postprocessor",
  "workspaceFolder": "/workspace",
  "extensions": ["ms-vscode.go", "golang.go"],
  "shutdownAction": "none",
  "settings": {
    "remote.extensionKind": {
      "ms-azuretools.vscode-docker": "workspace"
    },
    // Golang general settings
    "go.useLanguageServer": true,
    "go.autocompleteUnimportedPackages": true,
    "go.gotoSymbol.includeImports": true,
    "go.gotoSymbol.includeGoroot": true,
    // Options for gopls
    "gopls": { ... },
    "go.lintTool": "golangci-lint",
    // Lint flags
    "go.lintFlags": [...],
    // Golang on save
    "go.buildOnSave": "workspace",
    ...
  },
}
```

Ezzel a konfigurációval a fejlesztői környezet használata a következőképpen zajlik:

1. Egy *VS Code* ablakban az F1 billentyű lenyomása után kiválasztjuk a „*Remote-Containers: Open Folder in Container...*” opciót, majd kiválasztjuk az egyik szolgáltatás mappáját, például a *Parser* szolgáltatását. Ekkor elindul az összes konténer, ami a közös *docker-compose.yml* fájlban definiálva van.
2. Miután elindult a fejlesztői konténer, nyissunk egy új *VS Code* ablakot, és ugyanúgy, mint az előbb, válasszuk ki egy másik mikroszolgáltatás mappáját, például a *Postprocessor*-ét. Ennek hatására a *VS Code* felcsatlakozik a már futó *postprocessor* fejlesztői konténerre.
3. Ismételjük meg a folyamatot a harmadik szolgáltatásra is.
4. Ezután az egyes ablakokban az F5 billentyűt lenyomva indulnak el *debug* módban az egyes szolgáltatások. Ennek a működéséhez egy további *launch.json* fájlt kellett definiálnom (ahogy az a *VS Code* környezetben megszokott).

A fenti lépések végrehajtása után a megszokott módon tudunk fejleszteni, tesztelni és hibát keresni akár a kód léptetésével a fejlesztői konténerekben belül. Ez a konfiguráció azért is nagyon hasznos, mert így a fejlesztéshez nem szükséges az adott számítógépre *RabbitMQ*-t, *ElasticSearch*-t vagy *Go*-t telepíteni, elég, ha a *Docker Desktop* és a *Visual Studio Code* (és esetleg *Git*) telepítve van, a többitől a konténerizált környezet és a *Docker Compose* gondoskodik.

4.1.2 Verziókezelés

A verziókezelés, vagy *version control*, a szoftver kód változásainak nyomon követésének és kezelésének gyakorlata. A szoftverfejlesztés magában foglalja a program kód folyamatos módosítását, a verziókezelő rendszer pedig megkönnyíti ezt a feladatot. A verziókezelő szoftver a különböző konfigurációs fájlok és a forráskód karbantartására használható és segíti a fejlesztőket a különböző szoftververziók biztonságos és szervezett tárolásában.

A projektem verziókezelésére a *Git* verziókezelő rendszert használtam, a projekt *repository* pedig *GitHub*-on van tárolva, így egy biztonsági mentésként is szolgál arra az

esetre, ha bármi történne a lokális verzióval. A *GitHub repository* létezése azért is szükséges, mert *Go*-ban a kódot *module*-okba, azon belül *package*-ekbe szervezik, egy *package* használatához azt importálni kell, ez az importálás pedig egy *import path* segítségével történik. Az *import path* leírja, hogyan szerezhető be a *package* forráskódja egy verziókezelő rendszer, például a *Git* segítségével. A *go* ezt a tulajdonságot használja a *package*-ek automatikus lekérésére a távoli *repository*-kból. [21]

4.1.3 Kódminőség ellenőrzése

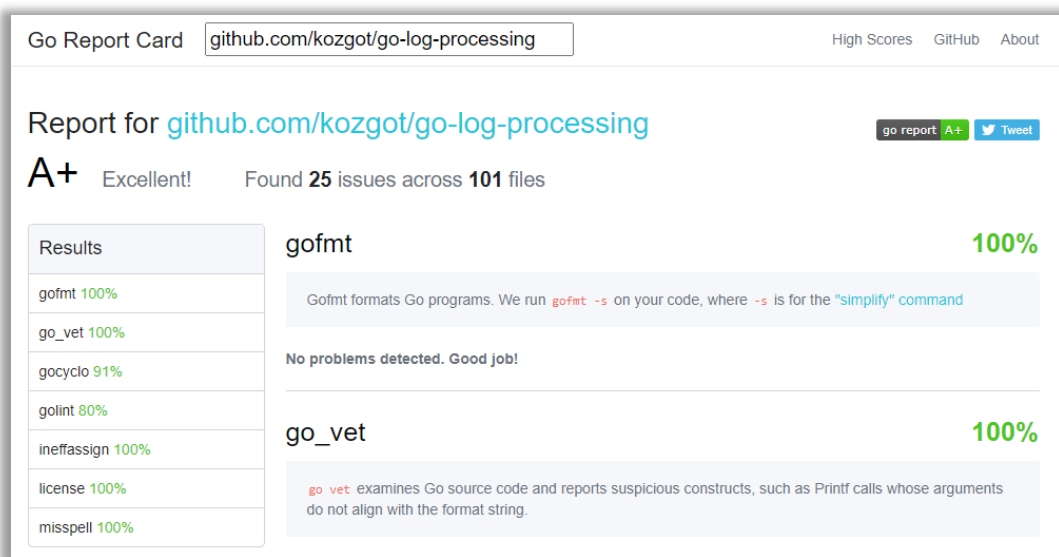
Ahogy már a harmadik fejezetben is írtam, *Go* programozási nyelv nem szabad formátumú nyelv, sok konvencióval rendelkezik a kód formátumának rögzítésére. Ennek okai közé tartozik például, hogy a fejlesztőknek így nem kell mindig más formázási stílusokhoz alkalmazkodni, és kevesebb idő telik el a formázási stílusokra történő áttállással, hogyha mindenki ugyanahhoz a stílushoz tartja magát.

A formázási konvenciók betartására a *Go* ökoszisztéma számos eszközt kínál számunkra, például lintereket (ilyen a *golangci-lint*), amelyek segítenek például az elnevezések, kommentek helyesírásának betartásában, a sorok hosszának egy adott hosszúság alatt tartásában, a nem használt változók észrevételében és még sorolhatnánk.

Ezen kívül egy hasznos eszköz a *Go Report Card*, amely egy webalkalmazás, amely jelentést készít egy nyílt forráskódú *Go* projekt kódminőségéről. Számos eszközt használ a kódminőség ellenőrzésére, például a *gocyclo* figyelmeztet, ha túl magas komplexitású függvényt talál, míg a *golint* a magyarázó komment nélküli exportált függvényekre, típusokra, interfészekre hívja fel a figyelmünket.

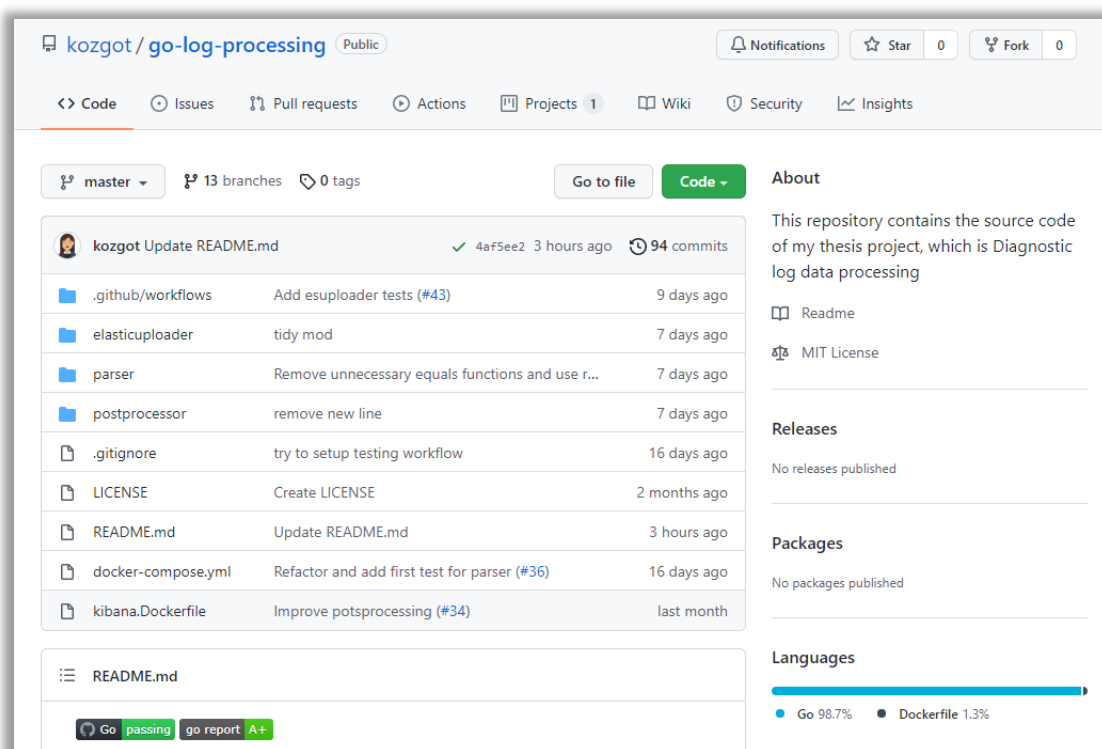
A *Go Report Card* rendelkezik egy parancssori interfésszel is (*CLI, Command Line Interface*), amelyet lokálisan futtathatunk, ha a projektünk nem elérhető *GitHub*-on, vagy privát *repository*-t használunk. [22]

Az alábbi ábrán látható a napló feldolgozó rendszer projektjéhez tartozó *Go Report Card* jelentés.



4.2. ábra: A Go Report Card jelentése a naplófeldolgozó rendszer forráskódjáról

A *Go Report Card* visszajelzéséről egy jelvény is kitéhető a projekt *GitHub* repository-jának README fájljába [23], ezt mutatja a következő ábra.



4.3. ábra: A projekt GitHub oldala

Az eddig említett eszközökön kívül még egy *GitHub workflow*-t is beállítottam, ami a *master* ág minden commit-ját ellenőrzi egy *build* és egy *teszt* fázissal. Ezekről a tesztelésről szóló alfejezetben részletesebben is lesz szó.

4.2 A szolgáltatások megvalósítása

Az előző alfejezetben bemutattam, hogy hogyan raktam össze a fejlesztői környezetet, és hogyan oldottam meg az ezzel kapcsolatos nehézségeket ezzel megkönnyítve a fejlesztés és hibakeresés folyamatát a későbbiekben. Ez az alfejezet arról fog szólni, hogy az egyes mikroszolgáltatások hogyan állnak össze, és milyen módon valósítottam meg a fő funkcióikat.

4.2.1 Parser mikroszolgáltatás

Ennek a mikroszolgáltatásnak a fő felelőssége, hogy felolvassa és memóriabeli objektumokká transzformálja a naplófájlok sorait, majd azokat publikálja egy *RabbitMQ exchange* felé. Fontos, hogy ez a folyamat jól átlátható és könnyen továbbfejleszthető módon legyen megvalósítva, és képes legyen kezelni az esetleges apró eltéréseket a sorok között, hiszen a naplófájlok formátuma nem rögzített és igen változatos. Ez az alfejezet bemutatja, hogyan épül fel a *parser* mikroszolgáltatás, mik a fő funkciói és ezek hogyan vannak megvalósítva.

4.2.1.1 A szolgáltatás felépítése

A fejlesztés során sokat gondolkodtam, hogy hogyan építsem fel a *Go* kódom struktúráját, ugyanis a *Go* nyelvben erre nincs egy előre definiált standard mód, viszont rengeteg lehetőségünk van, hogy úgy szervezzük a kódunkat, ahogy az a saját projektünk számára a legmegfelelőbb. Léteznek ajánlások a *Go* közösség által, például a ***golang-standards/project-layout*** [24] nevű projekt (amely a neve ellenére nem egy standardot ír le a projektek szervezésére, hanem a *Go* ökoszisztémában létező és kialakulóban lévő elrendezési minták egy gyűjteménye), illetve számos cikk és blog post is iránymutatást nyújthat számunkra ebben a témában. Egy valami azonban adott: a *Go* kód mindig *package*-ekbe van szervezve, az összetartozó *package*-eket pedig – a *Go* 1.13-as verziója óta – *go module*-ok tartalmazzák. Innentől a kérdés már csak az, milyen *package*-ekből álljon a kódunk?

A legalapvetőbb, legegyszerűbb kódszervezési stratégia, ha minden kódunkat egyetlen *package*-be tesszük. Ez megfelelő megoldás lehet egyszerűbb és kisebb kódbázisú alkalmazások esetében, azonban egy bonyolultabb, nagyobb alkalmazásnál nehezen karbantartható, kevésbé átlátható kódot eredményezne.

Egy másik lehetőség, hogy a felelősségek mentén szétdaraboljuk a kódot különböző *package*-ekbe, amelyek egymással kommunikálhatnak, így átláthatóbb és karbantarthatóbb kódot kapva. Ekkor arra kell figyelni, hogy minden *package* egy felelősséget valósítson meg, illetve, hogy ne legyenek körkörös függőségek, ugyanis a *Go* ezt nem engedi meg.

Egy harmadik lehetőség, hogy a második esethez hasonlóan a kódot a felelősségek mentén bontjuk *package*-ekre, viszont ezúttal teljesen független *package*-eket készítünk, azaz nem engedjük, hogy egymással kommunikáljanak – ehelyett minden *package*-nek minden függőségét helyileg kell deklarálnia interfészeken és változókon keresztül. [25]

Ezek közül én a második lehetőséget valósítottam meg, mivel az első megoldás nem biztosít elég jó átláthatóságot, a harmadikban pedig a függőségek helyi deklarációi miatt rengeteg interfész duplikálva lenne jelen a kódban, amely sok esetben inkább lassíthatja a fejlesztést, illetve egy ilyen projekt esetében nem jár elég sok előnnyel.

Annak érdekében, hogy további struktúrát adjak a forráskódomnak, a *package*-eket további könyvtárakba szerveztem a következő logika szerint. A *parser* szolgáltatás mappáján belül négy további mappa található, amelyek a következők:

- ***cmd***: Ez a könyvtár tartalmazza a projekt fő alkalmazását, azaz a *main.go* fájlt, amely az alkalmazás belépési pontját (*main* függvény) tartalmazza. Ez a ***golang-standards/project-layout*** ajánlása alapján kevés kódot tartalmaz, csak meghívja az *internal* vagy a *pkg* alatt található *package*-ek kódját.
- ***internal***: Ide kerül minden privát alkalmazáskód, amelyet nem szeretnénk, hogy külső fél használjon. A *parser* szolgáltatás esetében a *models* kivételével minden *package* *internal*.
- ***pkg***: Olyan kód, amelyet használhat külső alkalmazás. Ide a *parser* szolgáltatás esetében csak a *models package* került, amely az adatmodellek definícióit tartalmazza, mivel ezekre épít a *postprocessor* szolgáltatás.

A *parser* mikroszolgáltatásban minden más kód *internal*.

- **tests:** Itt találhatók a szolgáltatáshoz tartozó teszt fájlok, illetve egyéb teszteléshez használt segéd funkciók és erőforrás fájlok. Erről részletesebben a 4.4 fejezetben fogok írni.

4.2.1.2 A napló fájlok letöltése

A naplófájlok felolvasásának megkezdéséhez a fájlokat meg kell szerezni valahogyan, így elsőként azt kellett megterveznem, hogy ez a lépés hogyan történjen. Ez a kérdés magában foglalja azt is, hogy a termelődő napló fájlok hol kerüljenek tárolásra a feldolgozás megkezdéséig.

A tárolás módjának kiválasztásakor fontos szempont volt, hogy a tárhely tudjon nagy mennyiségű adatot tárolni, a tárolt adatok bárhol elérhetők legyenek és a letöltésük relatíve gyors legyen. Ezen felül előnyös, ha a tárhellyel történő interakció egyszerűen megvalósítható *Go* kódból. Ezek alapján döntöttem úgy, hogy a naplófájlokat **Azure Blob Storage** tárolókban fogom elhelyezni, ahonnan a *parser* szolgáltatás letölti őket a feldolgozási folyamat megkezdésekor.

Az **Azure Blob Storage** a *Microsoft Azure* egy szolgáltatása, amely lehetővé teszi a felhasználók számára, hogy nagy mennyiségű strukturálatlan adatot tároljanak a *Microsoft* adattárolási platformján. Gyors és egyszerű adatelérést, skálázhatóságot és biztonságot garantál a felhasználók számára. [26] Használatához széles körben érhető el kliens könyvtárak például *.NET*-hez, *Python*-hoz és *Go*-hoz is, amely a github.com/azure/azure-storage-blob-go címen érhető el. [27]

A fájlok letöltését a *filedownloader package*-ben valósítottam meg, amely két *go* fájlt tartalmaz. A *file_downloader.go* fájlban egy interfészt definiáltam, amely azokat a függvénydefiníciókat tartalmazza, amelyekre a *parser* szolgáltatásnak szüksége van. Ezek a függvények *ListFileNames*, amely listázza az elérhető naplófájlok neveit, és a *DownloadFile*, amely fájl név (elérési útvonal) alapján letölt egy naplófájlt. Az interfész a következőképp néz ki:

```
type FileDownloader interface {
    ListFileNames() []string
    DownloadFile(fileName string) io.ReadCloser
}
```

Ezzel a megoldással a *parser* működéséhez szükséges funkciókat egy interfész mögé rejtettem, amely két fontos előnnyel is jár: egyrészt a *parser* nem függ közvetlenül az *azure blob storage* klienskönyvtártól, így az bármikor lecserélhető egy másik

szolgáltatásra, másrészt pedig teszteléskor így egyszerűen mockolható a letöltéshez használt szolgáltatás.

A második, *azure_file_downloader.go* nevű fájlban található ennek az interfésznek az *azure blob storage*-t használó implementációja. A *Go* nyelvben az interfészek implicit módon vannak implementálva, nincs konkrét kulcsszó az interfészek megvalósítására. Ezen kívül a *Go* nyelvben osztályok sincsenek, helyettük vannak viszont *struct*-ok, amelyekhez metódusok adhatók, így összefogva az adatokat a metódusokkal, amelyek az adatokon dolgoznak – egy osztályhoz hasonlóan. [28] Ezek alapján az implementáció a következőképp néz ki:

```
// AzureDownloader contains data needed to list or dowload blobs from azure.
type AzureDownloader struct {
    Credential      *azblob.SharedKeyCredential
    StorageAccountURL *url.URL
    ContainerURL     azblob.ContainerURL
}

// ListFileNames lists the blobs in the azure container.
func (downloader *AzureDownloader) ListFileNames() []string {...}

// DownloadFile downloads the blob with the given name from azure.
func (downloader *AzureDownloader) DownloadFile(fileName string)
io.ReadCloser {...}
```

A *ListFileNames* az *azblob.ContainerURL.ListBlobsFlatSegment* függvényét használva lekérdezi a megadott konténerben található fájlok neveit, és visszaadja azokat egy tömbben.

A *DownloadFile* a megadott fájlnevet használva az *azblob.BlockBlobURL.Download* függvény segítségével tölti le a *blob* tartalmát, majd visszaad egy *io.ReadCloser*-t, amelyen keresztül olvasható az adat.

Az *azure storage account* címe, a tároló konténer neve, valamint a hozzáférési kulcs környezeti változókbán vannak definiálva.

4.2.1.3 A sorok olvasása és transzformálása

A naplósorok felolvasása és transzformálása reguláris kifejezések segítségével három lépésben történik. Először a naplózási szintek (*log level*) kerülnek felolvasásra, amelyek alapján egyből egy szűrés is történik, hogy a továbbiakban a számunkra nem érdekes sorokkal (például *DEBUG* szintűekkel) ne foglalkozzunk. Ezt a logikát a *loglevelparser package* tartalmazza. Ezt a lépést azért érdemes először elvégezni, mert egyfelől így rögtön kiszűrhetjük a nem fontos sorokat, másrészt a továbbiakban a

naplózási szintekre támaszkodhatunk a transzformációk elvégzése során (mivel máshogy nézhet ki egy INFO szintű sor tartalma, mint mondjuk egy ERROR szintűé).

Ezután az időbélyegeket olvassuk fel és transzformáljuk *time.Time* objektumokká, ezért a *timestampparser package* felelős. Ennek különválasztása azzal az előnnyel jár, hogyha az időbélyegek formátuma változik, akkor csak ehhez az egy *package*-hez kell hozzányúlni. Ez most két fajta formátumot tud kezelni, mivel a két feldolgozott napló formátum két különböző időbélyeg formátumot tartalmaz.

Végül a sorok egyedi tartalmát dolgozzuk fel, ez a legváltozatosabb formátumú rész minden napló sorban, így az ezt kezelő logika is szerteágazó. Legelőször a naplózási szinteknek megfelelően ágazik szét a logika: külön logika felel az INFO, a WARNING vagy WARN és az ERROR szintű sorok felolvasásáért, amelyekért sorban az *InforParser*, *WarningParser* és *ErrorParser* típusok a felelősek. Az ehhez kapcsolódó összes kód a *contentparser package*-ben található meg.

A naplófájlok felolvasásának belépési pontja a *logparser package*-ben található *LogParser* típus *ParseLogfiles* függvénye, amelyet egy *http handler* függvény hív meg a *localhost:8080/process* (a *Docker* konténereket helyben futtatva) végpontra küldött *http* kérések beérkezésekor. A *LogParser* két függőséggel rendelkezik, amelyek közül az egyik a már fentebb részletezett fájl letöltő, a másik pedig egy *RabbitMQ* üzenet publikáló szolgáltatás, amely a *rabbitmq package*-ben található (erről a 4.2.1.5. alfejezetben írok részletesen). A *ParseLogfiles* függvény lekérdezi az elérhető napló fájlokat a letöltő szolgáltatás segítségével, majd minden fájlra meghívja a *fileparser package*-ben található *ParseSingleFile* függvényt, amely soronként olvassa a megadott fájlt, és sorban meghívja rájuk a felolvasás három szakaszát végrehajtó függvényeket, végül a sorok tartalmának felolvasása után közzéteszi a felolvasott sort az üzenetpublikáló komponens használatával.

Ezekén kívül van még három *package* az *internal* könyvtárban, amelyek semleges, a többi *package* által használt funkciókat tartalmaznak. A *common* közös logikát tartalmaz a sorok felolvasásával kapcsolatosan, amelyek több helyen is használva vannak a kódban. A *formats package* pedig a különböző sorok formátumait leíró reguláris kifejezéseket tartalmazza. A *utils package*-be került egy általános hibakezelő funkció, amely hiba esetén kiírja a kapott üzenetet a konzolra, és hibával leállítja az alkalmazás futását.

4.2.1.4 A feldolgozott napló sorok és az adatmodellek

Ebben az alfejezetben bemutatom, hogyan épülnek fel az adatmodellek, amelyeket a *parser* szolgáltatás tölt fel a felolvasott adatokkal, és ismertetem a feldolgozott napló sorok közül a leglényegesebbeket.

Egy feldolgozott sort a *ParsedLogEntry* típus reprezentálja, amely öt mezővel rendelkezik:

```
type ParsedLogEntry struct {
    Timestamp    time.Time
    Level        string
    ErrorParams   *ErrorParams
    WarningParams *WarningParams
    InfoParams    *InfoParams
}
```

Mivel a *Go* nyelvben nincsenek osztályok, ezért az öröklés koncepció sem létezik ebben a nyelvben. Így ősosztályok és öröklés helyett a különböző típusú sorok definiálását kompozícióval kellett megoldanom, ezért a *ParsedLogEntry* struktúra a minden sorra értelmezett *Timestamp* (időbélyeg) és *Level* (naplózási szint) mezőkön kívül három opcionálisan kitöltendő mezőt is tartalmaz. Ez a három mező az *ErrorParams*, amely akkor van kitöltve, ha a *Level* mező értéke „ERROR”, a *WarningParams*, amely akkor van kitöltve, ha a *Level* értéke „WARN” vagy „WARNING”, végül az *InfoParams*, amely INFO szintű sorok esetében tartalmaz értéket. A *ParsedLogEntry* struktúrához két függvényt adtam, amelyek a struktúra JSON formátummá alakításáért és a visszaalakításért felelősek.

Az *ErrorParams* struktúra az ERROR szintű sorok tartalmát tárolja, öt mezővel rendelkezik:

```
type ErrorParams struct {
    ErrorCode  int
    Message    string
    Severity   int
    Description string
    Source     string
}
```

Egy példa egy ilyen formátumú sorra:

```
Wed Jun 10 10:26:37 2020 ERROR    : error_code[241] message[DLMS error]
severity[3] description[n/a] source[dc18-smc32]
```

A *WarningParams* struktúra a WARN, illetve a WARNING szintű sorok tartalmát tárolja, és a következőképpen áll össze:

```
type WarningParams struct {
    WarningType WarningEntryType
    TaskFailedWarningParams *TaskFailedWarningParams
    JoinMessageParams      *SmcJoinMessageParams
    TimeoutParams          *TimeOutParams
    LostConnectionParams    *LostConnectionParams
}
```

A többféle warning sorok definiálását szintén kompozícióval valósítottam meg. A *WarningType* mező azt írja le, hogy milyen típusú warning az adott naplózott esemény, ennek értékei jelenleg a következők lehetnek:

- *TaskFailedWarning*: a „*Task failed...*” kezdetű sorokat jelenti, ilyen típusú soroknál a *TaskFailedWarningParams* mező van kitöltve, amely a következő adatokat tartalmazza:

```
type TaskFailedWarningParams struct {
    Name          string
    SmcUID         string
    UID           int
    Priority       int
    Retry         int
    FileName      string
    Creation      time.Time
    MinLaunchTime time.Time
    Details       *ErrorParams // details of the inner error
}
```

- *JoinRejectedWarning*: a *plc_manager.log* fájlokban az elutasított SMC csatlakozási próbálkozásokat leíró sorok ilyen típusúak. Ezeknél a *JoinMessageParams* mező tartalmaz értéket, amely a következő mezőkkel rendelkezik:

```
type SmcJoinMessageParams struct {
    Ok          bool
    Response    string
    JoinType    string
    SmcAddress  SmcAddressParams
}
```

Ezt a típust egy fajta INFO szintű sorhoz is felhasználtam, mivel a sikeres SMC csatlakozásokat jelentő sorok a naplózási szinttől eltekintve azonos formátumúak.

- *TimeoutWarning*: a „*Timeout protocol...*” kezdetű sorokat jelenti, ilyen típusnál a *TimeoutParams* mező értéke van kitöltve, amely a következőképp néz ki:

```
type TimeOutParams struct {
    Protocol string
    URL      string
}
```

- *ConnectionLostWarning*: a „*Connection of type...*” kezdetű sorokat reprezentálja, ilyen soroknál a *LostConnectionParams* mező tartalmaz értéket. Ez a következő mezőkkel rendelkezik:

```
type LostConnectionParams struct {
    Type      int
    Reason    string
    ClientID  string
    URL       string
    Topic     string
    Timeout   int
    Connected bool
}
```

Az *InfoParams* struktúra az INFO naplózási szintű sorokat reprezentálja, ilyen szintűekből van a legtöbb fajta bejegyzés a naplófájlokban. A struktúra a következő kódrészleten látható:

```
type InfoParams struct {
    EntryType           InfoEntryType
    RoutingMessage      *RoutingTableParams
    JoinMessage         *SmcJoinMessageParams
    StatusMessage       *StatusMessageParams
    DCMMessage          *DCMessageParams
    ConnectionAttempt   *ConnectionAttemptParams
    SmcConfigUpdate     *SmcConfigUpdateParams
    ConnectionReleased  *ConnectionReleasedParams
    InitConnection      *InitConnectionParams
    InternalDiagnosticsData *InternalDiagnosticsData
}
```

Mivel nagyon sok formátumú INFO bejegyzést képes feldolgozni a rendszer, így nagyon sok modell struktúrát definiáltam hozzájuk, amelyek közül ebben a dolgozatban csak a lényegesebbeket fogom kiemelni. A többi típus és struktúra megértéséhez segítséget adhatnak a *models package*-ben található *InfoEntryType* és *DCMessageType* típusok, és a hozzájuk fűzött magyarázó kommentek.

Az *EntryType* mező az INFO szintű bejegyzés típusát határozza meg. A különböző bejegyzés-típusokat ezúttal is kompozíció használatával definiáltam.

Az egyik fontos INFO szintű bejegyzés típus a *DCMessage*, amely azokat a bejegyzéseket foglalja magában, amelyek eleje illeszkedik a következő két formátumra:

```
Wed Jun 10 11:29:57 2020 INFO      : <--[read index profiles]--(SMC) ...
Wed Jun 10 11:58:41 2020 INFO      : --[index]-->(SVI) ...
```

A fentiek közül az első a bejövő, a második a kimenő „üzenetekre” ad egy-egy példát. Ehhez hasonló formátumú sorokból is sok féle található a *dc_main.log* fájlokban, ezeket a típusokat írja le a *DCMessageType*. Ezek közül néhányat felsorolok:

- *Consumption*: A fogyasztási adatokat tartalmazó üzenetek típusa.
- *SmcConfig*: Az ilyen típusú bejegyzések az adatbázisból érkező *SMC* konfigurációs adatokat tartalmazzák.
- *PodConfig*: Az ilyen típusú bejegyzések az adatbázisból érkező *pod*¹ konfigurációs adatokat tartalmazzák.
- *SmcAddress*: Az ilyen típusú bejegyzések az adatbázisból érkező *SMC* cím adatokat tartalmazzák.

Egy másik lényeges bejegyzés típus például a *ConnectionAttempt*, amely egy *SMC*-hez történő csatlakozási kísérletet ír le. Az ehhez tartozó adatstruktúra a következő:

```
type ConnectionAttemptParams struct {
    URL      string
    SmcUID    string
    At       string
}
```

4.2.1.5 A felolvasott sorok közzététele

A *parser* szolgáltatás az általa felolvasott sorokat egy *RabbitMQ exchange* felé továbbítja. Ezt a funkciót a *rabbitmq* package-ben helyeztem el, amely a *filedownloader*-hez hasonlóan két go fájlt tartalmaz. A *message_producer.go* fájl az üzenettovábbító szolgáltatás interfészdefinícióját tartalmazza, amelyben azokat a függvényeket definiáltam, amikre a *parser* szolgáltatásnak szüksége van, az *amqp_producer.go* fájlban pedig ennek az interfésznek az *AMQP* protokollt használó implementációja található,

¹ Egy pod a következtetésem szerint egy elektromos mérőt jelent, viszont ezzel kapcsolatban nem áll rendelkezésemre dokumentáció vagy magyarázat, mindössze a naplófájlokban lévő adatokból következtettem erre.

amely megvalósításához a github.com/streadway/amqp klienskönyvtárt használtam fel. Az interfész a következőképp néz ki:

```
type MessageProducer interface {
    PublishStringMessage(indexName string)
    PublishEntry(line models.ParsedLogEntry)
    OpenChannelAndConnection()
    CloseChannelAndConnection()
}
```

A *PublishStringMessage* függvény egy szöveges üzenetet továbbít a *RabbitMQ* felé. Ezt arra használja a **parser** szolgáltatás, hogy értesítse a felolvasott sorokat fogyasztó szolgáltatásokat, amikor a felolvasandó fájlok végére ért.

A *PublishEntry* függvény egy felolvasott napló sort tesz közzé. A *PublishStringMessage* és *PublishEntry* függvények a *publish* függvényt használják a konkrét publikálás elvégzésére. Ez a függvény az **amqp package Channel** típusának *Publish* függvényével küldi el a sorosított adatokat a megfelelő *exchange name* és *routing key* megadásával, amelyek környezeti változóknak vannak definiálva. A közzétételhez perzisztens szállítási módot használ, annak érdekében, hogy a küldött üzenetek túléljék az esetleges *RabbitMQ* szerver újraindításokat. A *publish* függvény megvalósítása az alábbi kódrészleten látható.

```
func (producer *AmqpProducer) publish(data []byte) {
    body := data

    err := producer.channel.Publish(
        producer.exchangeName, // exchange
        producer.routingKey,    // routing key
        false,                  // mandatory
        false,                  // immediate
        amqp.Publishing{
            DeliveryMode: amqp.Persistent,
            ContentType:  "application/json",
            Body:         body,
        })
    utils.FailOnError(err, "Failed to publish a message")
}
```

Az *OpenChannelAndConnection* függvény kapcsolódik a *RabbitMQ* szerverhez a megadott URL-en (amely szintén környezeti változóban van definiálva), majd nyit egy csatornát és létrehoz egy *exchange*-t a szerveren. A *CloseChannelAndConnection* bontja a létrehozott kapcsolatot és bezárja a csatornát.

4.2.2 Postprocessor mikroszolgáltatás

A *postprocessor* szolgáltatás fő felelőssége, hogy egy *RabbitMQ* sorból kivegye a *parser* által küldött adatokat, végrehajtson rajtuk egy feldolgozó logikát, majd a feldolgozás eredményeit továbbítsa egy *RabbitMQ exchange* segítségével az *Elasticsearch*-be mentést végző szolgáltatásnak. A feldolgozás lényege, hogy olyan kapcsolatot találjunk az adatok között, amelyeket egyszerű átolvasással, vagy *Kibana*-ban történő szűrők beállításával nem látnánk meg, és amelyek segíthetnek megérteni, mi történt a hálózatban, ami az esetleges hibákat okozhatta. Ebben az alfejezetben bemutatom, hogyan épül fel a *postprocessor* mikroszolgáltatás, melyek a fő funkciói és ezeket hogyan valósítottam meg.

4.2.2.1 A szolgáltatás felépítése

Hasonlóan a *parser* szolgáltatáshoz, a *postprocessor* könyvtár gyökerében most is négy mappa található: *cmd*, *internal*, *pkg* és *tests*. A *cmd* mappában most is egyetlen *main.go* fájl található, amely a mikroszolgáltatás belépési pontja. A *pkg* mappa ezúttal is csak a *models package*-t tartalmazza, amelyben az adatmodellek típusai vannak definiálva. A *tests* mappa ezesetben is a mikroszolgáltatás tesztjeit és a teszteléshez használt segédfunkciókat tartalmazza.

Az *internal* mappába három *package* került:

- ***processing***: Itt található a feldolgozási logikát tartalmazó fájlok.
- ***rabbitmq***: Ez a *package* tartalmazza a *RabbitMQ*-val történő interakció logikáját.
- ***utils***: Ugyanúgy, mint a *parser* szolgáltatás esetében, egy egyszerű hibakezelést megvalósító függvényt tartalmaz, amelyet a kód többi része használ.

4.2.2.2 Interakció a RabbitMQ-val

A *RabbitMQ*-val kapcsolatos logikát a *rabbitmq package*-ben valósítottam meg. Ebben a *package*-ben négy fájl található, ebből kettő a *consumer* (üzenet fogyasztó) viselkedés, kettő pedig a *producer* (üzenet termelő) viselkedés megvalósításával kapcsolatos.

A *consumer* logika felelős azért, hogy üzeneteket fogyasszon egy *RabbitMQ* várakozási sorból, amely üzenetek a felolvasott naplófájl sorokat reprezentálják, majd ezeket az üzeneteket átadja a feldolgozó komponensnek. Az ehhez a funkcióhoz kapcsolódó függvényeket a *message_consumer.go* fájlban lévő *MessageConsumer* interfész foglalja magába, a feldolgozó komponens az itt definiált függvényeket használja, ezekre van szüksége.

```
type MessageConsumer interface {  
    ConsumeMessages() <-chan amqp.Delivery  
    CloseConnectionAndChannel()  
    Connect()  
}
```

A *MessageConsumer* interfészt az *amqp_consumer.go* fájlban lévő *AmqpConsumer* típus valósítja meg. Az implementációhoz a github.com/streadway/amqp klienskönyvtárt használtam fel. Ezzel a megoldással elrejtettem a mikroszolgáltatásom által használt fő funkciókat egy interfész mögé, így könnyítve a tesztelést, illetve az esetleges implementáció-cserét.

A *MessageConsumer* interfész *ConsumeMessages* nevű függvényének meghívásával fogyaszthatunk üzeneteket a megfelelő *RabbitMQ* várakozási sorból. Ezt úgy valósítottam meg, hogy elfedje az *AMQP* protokoll használatának részleteit, azáltal, hogy egy lépésben elvégzi a várakozási sor létrehozását és *exchange*-hez kötését, majd pedig a *RabbitMQ consumer* regisztrálását és visszaad egy *amqp.Delivery* típusú *channel*-t, ahová aszinkron módon érkeznek majd az üzenetek a *RabbitMQ* felől. A *Connect* és *CloseConnectionAndChannel* függvények a *RabbitMQ* kapcsolat felépítéséért és bontásáért felelnek.

A *producer* logika feladata, hogy a feldolgozott adatokat *RabbitMQ* felé továbbítsa. Hasonlóan a *consumer* logikához, ezt is úgy igyekeztem megvalósítani, hogy a belső működéssel és az *AMQP* protokoll használatával kapcsolatos részletek a *message_producer.go* fájlban található *MessageProducer* interfész mögött rejtve maradjanak. Az interfész definíciót mutatja a következő kódrészlet.

```

type MessageProducer interface {
    PublishEvent(event models.SmcEvent)
    PublishConsumption(cons models.ConsumptionValue)
    Connect()
    CloseChannelAndConnection()
}

```

A *MessageConsumer* interfészhez hasonlóan a *Connect* és *CloseChannelAndConnection* függvények a kapcsolat felépítéséért és lebontásáért felelősek. A *PublishEvent* és *PublishConsumption* függvények a feldolgozási folyamat által előállított két különböző típusú adat továbbítását végzik (a feldolgozási folyamatról a következő kettő alfejezetben írok részletesebben).

Az interfész megvalósítása az *amqp_producer.go* fájlban található. Ugyanúgy, mint a *parser* szolgáltatás esetében, itt is perzisztens kézbesítési módot használok az adatok továbbítására, és a *routing key* értékét, az *exchange* nevét és a várakozási sor nevét most is környezeti változókból tárolom.

4.2.2.3 Az események feldolgozása

Az előző alfejezetben már utaltam rá, hogy az adatok feldolgozásának folyamata két részből tevődik össze, ezek közül az első az események feldolgozása.

Ez egy olyan feldolgozást valósít meg, amely minden naplózott eseményt és adatot megpróbál egy adott *SMC*-hez (*Smart Meter Controller*) kötni, és ez alapján minden egyes *SMC*-hez előállít egy esemény listát, amely segítségével végigkövethető, hogy az idő folyamán milyen események következtek be, amelyek érintették az adott *SMC*-t.

A feldolgozás folyamatát a ***processing*** *package*-ben lévő *EntryProcessor* végzi, amely két függőséggel rendelkezik: az egyik egy *MessageConsumer*, a másik pedig egy *MessageProducer*, amelyeket az előző alfejezetben mutattam be. Az *EntryProcessor* a sorok feldolgozása közben hat adatstruktúrát épít fel, amelyek a következők:

- ***eventsBySmcUID***: Ez egy *map* adatstruktúra, amely kulcsként az *SMC*-k *string* típusú egyedi azonosítóit használja, az értékek pedig *SmcEvent* tömbök, amelyekbe az adott *SMC*-khez tartozó események kerülnek. Ez lesz a feldolgozás első fázisának eredménye.
- ***smcDataBySmcUID***: Szintén egy *map* adatstruktúra, ahol a kulcsok az *SMC* azonosítók, viszont itt az értékek *SmcData* objektumok, amelyek egy

SMC adatait (például fizikai cím, logikai cím) tárolják. Ezt a struktúrát a sorok feldolgozása közben építi fel az *EntryProcessor*. A különböző formátumú és különböző eseményeket jelentő sorok más-más információdarabkát tartalmaznak az *SMC*-kel kapcsolatban, például *DCMessage* típusú, azon belül is az *SmcAddress* típusú sorokból megtudhatjuk egy adott *SMC* fizikai, logikai és rövid címét, míg egy *ConnectionAttempt* típusú sorból a hozzá tartozó *URL* olvasható ki.

- ***smcUIDsByURL***: Ez egy *map* adatstruktúra, ahol a kulcsok az *SMC URL*-ek, az értékek pedig az *URL*-ekhez tartozó *SMC* egyedi azonosítók. A *ConnectionAttempt* típusú bejegyzésekből építem fel ezt a struktúrát, amelyet később arra használok fel, hogy az olyan naplózott eseményeket, amelyekhez csak az *URL* érhető el, hozzá tudjam kötni egy *SMC* azonosítóhoz. Ilyen esemény lehet például egy *connection lost warning*.
- ***podUIDToSmcUID***: Egy *map*, amiben a kulcsok a *pod*-ok egyedi azonosítói, az értékek pedig az *SMC*-k azonosítói, amikhez a *pod*-ok tartoznak. Ezt a struktúrát a *PodConfig* típusú *DCMessage* sorok feldolgozásával építettem fel, amelyekből kiderül, hogy egy adott *pod* azonosító melyik *SMC* azonosítóhoz tartozik. Ezeket az adatokat az *IndexReceived* típusú sorok feldolgozásához használom, amelyekben csak a *pod* azonosítók és sorszámok szerepelnek, így ezeket kell az *SMC* azonosítókhoz kötni.
- ***consumptionValues***: Egy lista, amely az összes fogyasztási adatot tartalmazza az adott időszakban (abban az időszakban, amit az éppen feldolgozott naplófájl leír). Ennek a listának a feltöltését az *EntryProcessor* végzi, viszont ez még egy hiányos eredményt ad, mivel ezen a ponton még nem tudjuk az egyes fogyasztási adatokat *SMC* azonosítókhoz kötni. Erről több információ a következő alfejezetben olvasható.
- ***indexValues***: Egy lista, amely az *SMC*-k felől érkező *index*² adatokat tartalmazza egy adott időintervallumban. Ezekből kiolvasható egy *pod*

² Egy index a *DC* által az *SMC*-ktől periodikusan gyűjtött adatokat jelenti.

azonosító és sorszám, illetve egy szolgáltatási szint azonosító szám, amelyek segítségével ezeket az adatokat össze tudjuk kapcsolni az egyes *SMC*-kkel.

A feldolgozási logika a naplózási szintek (WARN, WARNING, ERROR és INFO) szerint négy fő irányba ágazik szét, melyek közül az INFO szintű sorok feldolgozása a legbonyolultabb, mivel a legtöbb fajta naplóbejegyzés ilyen szintű.

A WARNING és WARN szintű naplóbejegyzéseket a ***WarningProcessor*** kezeli, amely két függvénnyel rendelkezik: az egyik a *ProcessWarn*, a másik pedig a *ProcessWarning*. Mindkettő egy darab bejegyzést vár paraméterül, amelyből előállítanak és visszaadnak egy *SmcEvent*-et, amely egy eseményt ír le, és egy *SmcData*-t, amelyben az adott bejegyzésből kiolvasható *SMC*-vel kapcsolatos adatok vannak. A *ProcessWarn* a WARN szintű sorokat kezeli, amelyek közül csak a *TimeoutWarning* típusúakra koncentrál, ugyanis ez egy megszakadt kapcsolatra utalhat a *DC* és egy *SMC* között, amely a leginkább érdekes a hibák felderítése szempontjából. A *ProcessWarning* a WARNING bejegyzésekkel foglalkozik, amelyek mindegyike egy visszautasított *SMC* csatlakozási kísérletet ír le, így ezek között a feldolgozás nem tesz különbséget.

Az ERROR szintű bejegyzéseket az ***ErrorProcessor*** dolgozza fel, amely egyetlen függvénnyel rendelkezik, amely a paraméterül kapott bejegyzésből egy *SmcEvent* és egy *SmcData* objektumot állít elő, hasonlóan a *WarningProcessor* függvényeihez. Az *ErrorProcessor* csak azokkal a bejegyzésekkel foglalkozik, amelyeknél a *Source* mező (azaz a forrás) tartalmaz értéket, ugyanis ennek hiányában nem tudtam őket *SMC* azonosítókhoz hozzárendelni. Azonban, mivel az ERROR típusú sorok minden esetben csoportokban helyezkednek el a naplófájlokban (egymás után közvetlenül több ERROR szintű bejegyzés, ugyanazzal az időbélyeggel), így feltételezhető, hogy az ilyen egymásutáni sorok összetartoznak és ugyanazt a hibát írják le, amelyek közül a legtöbb esetben legalább az egyiknél találhatunk *Source* értéket, ezáltal a hiba események többsége ilyen módon feldolgozásra kerül.

Ahogy már említettem, az INFO szintű bejegyzések feldolgozása a legszerteágazóbb és legösszetettebb, mivel ilyen bejegyzésből nagyon sok különböző féle fordul elő a naplófájlokban. Ezekkel a bejegyzésekkel az *InfoProcessor* komponens foglalkozik, amelynek *ProcessInfoEntry* függvénye egy bejegyzésből négy eredményt állít elő:

- Egy *SmcData* típusút, amely az adott bejegyzésből kiolvasható *SMC*-vel kapcsolatos adatokat tartalmazza, például azonosító, URL, fizikai cím, logikai cím stb.
- Egy *SmcEvent* típusút, amely a bejegyzés által reprezentált esemény adatait tartalmazza.
- Egy *ConsumptionValue* típusút, amely a *Consumption* típusú *DCMessage* bejegyzésekben található adatokat tartalmazza. Ilyen adatok például az aktuális és előző fogyasztás nagyságát jellemző szám, az időintervallum kezdete és vége, amelyre az adott értékek értelmezve vannak és a szolgáltatási szint azonosító száma.
- Egy *IndexValue* típusút, amely *IndexReceived* típusú *DCMessage* sorokból kiolvasható adatokat tartalmazza, amelyek: egy *pod* azonosító és sorszám, amelyek segítségével egy *SMC*-hez tudjuk kapcsolni az adatokat, az aktuális és előző értéket leíró számok, az intervallum kezdetét és végét jelölő időpontok, amelyekre az adott értékek vonatkoznak és egy szolgáltatási szint azonosító szám.

Az *EntryProcessor* az *InfoProcessor*, *WarningProcessor* és *ErrorProcessor* által előállított eredmények alapján frissíti a megfelelő adatstruktúrákat (*eventsBySmcUID*, *smcDataBySmcUID* stb.), és ahol szükséges, kikeresi az *smcUIDsByURL* struktúrából az adatokhoz tartozó *SMC* azonosítót az URL értéke alapján. Ezután az eredményül kapott *SmcEvent* típusú adatokat továbbítja a *RabbitMQ* szerver felé.

4.2.2.4 A fogyasztási adatok feldolgozása

A feldolgozás második lépése a fogyasztási adatokat köti az egyes *SMC*-khez, és összeállít belőlük egy adatsort, amely megmutatja, hogy az időben hogyan változik az egyes *SMC*-k által mért energiafogyasztás nagysága.

Ennek a lépésnek az elvégzéséhez meg kell várni, hogy a feldolgozás elérje a feldolgozandó naplóbejegyzések végét, amelyet a *RabbitMQ* várakozási sorból kiolvasott „END” üzenet jelöl. Ennek az az oka, hogy a felolvasó szolgáltatást úgy valósítottam meg, hogy az a naplófájlokat párhuzamosan, naplófájlanként egy-egy *goroutine* indításával olvassa fel, így a bejegyzések konkurens módon továbbítódnak a *RabbitMQ* szerver felé. Továbbá, ha egy üzenet kézbesítését valamilyen oknál fogva meg kell

ismételni, mondjuk mert a feldolgozás során hiba keletkezett, akkor az üzenetek sorrendje már nem az eredeti sorrendet tükrözi. Ez azt eredményezi, hogy a naplóbejegyzések nem feltétlenül a létrehozásuk sorrendjében kerülnek kiolvasásra a várakozási sorból. A feldolgozás azonban épít a bejegyzések sorrendjére, a következő módon. A fogyasztási adatokat a *Consumption* típusú *DCMessage* bejegyzések tartalmazzák, amelyek nem tartalmaznak olyan mezőt, amely segítségével ezeket az adatokat *SMC*-khez tudnánk kötni. Ezeket a bejegyzéseket minden esetben *IndexReceived* típusú *DCMessage* bejegyzések előzik meg, amelyeknek az időbélyege, az időintervallumot kijelölő időpontjai, valamint a szolgáltatási szint azonosító száma megegyezik az utánuk következő *Consumption* bejegyzésekével, amely arra enged következtetni, hogy ezek a bejegyzések összetartoznak. Ez alapján a fogyasztási adatokat az őket megelőző index adatok felhasználásával kapcsolom hozzá az egyes *SMC* azonosítókhoz. Ehhez viszont az kell, hogy egy adott *Consumption* bejegyzés feldolgozásakor már fel legyen dolgozva a neki megfelelő *IndexReceived* bejegyzés, ami a fentebb részletezett okok miatt nem garantált, ezért a feldolgozás bevárja az „END” üzenetet, majd ezután elindítja a fogyasztási adatok utófeldolgozását.

A fogyasztási adatokat az *EntryProcessor* gyűjti össze még a bejegyzések végét jelző üzenet beérkezése előtt, azonban ezek még nincsenek *SMC*-khez kötve, ezt a lépést a *ConsumptionProcessor* komponens végzi el. A *ConsumptionProcessor* ehhez felhasználja a feldolgozás első lépése során összegyűjtött index értékeket (*indexValues* tömb), és minden egyes fogyasztási adathoz megkeresi azt az index értéket, amelynek az időintervalluma és szolgáltatási szint azonosítója megegyezik az adott fogyasztási adatéval, majd a talált *IndexValue SmcUID* mezőjének értékével kitölti az éppen feldolgozott *ConsumptionValue SmcUID* mezőjét. Ezután a feldolgozott fogyasztási értéket továbbítja a *RabbitMQ* felé.

4.2.3 ElasticUploader mikroszolgáltatás

Ennek a szolgáltatásnak a felelőssége a *RabbitMQ* várakozási sorokból kiolvasott üzenetek feltöltése *Elasticsearch* indexekbe. Ebben az alfejezetben ismertetem, hogyan épül fel az *elasticuploader* mikroszolgáltatás, és hogyan valósítottam meg a lényeges funkcióit.

4.2.3.1 A szolgáltatás felépítése

Hasonlóan az előző két mikroszolgáltatáshoz, a funkciókat megvalósító *package*-eket ezúttal is négy fő mappába szerveztem: *cmd*, *internal*, *pkg* és *tests*. A *cmd* mappában az alkalmazás belépési pontját tartalmazó *main.go* fájl található, a *tests* mappa a teszt fájlokat és a velük kapcsolatos segédfunkciókat és erőforrásokat tartalmazza, a *pkg* mappába pedig ez esetben is csak a *models package* került.

Az *internal* mappába az *elastic*, *rabbitmq*, *uploader* és *utils package*-eket helyeztem el. Az *elastic package* foglalja magában az *Elasticsearch* szerverrel történő interakciót, míg a *rabbitmq package* a *RabbitMQ* szerverrel való kommunikációt valósítja meg. Az *uploader package*-be a feltöltési folyamatot vezérlő logikát helyeztem el, a *utils package* pedig ezúttal is egy általános hibakezelő funkciót tartalmaz.

4.2.3.2 Interakció RabbitMQ-val

A feltöltendő adatokat az *elasticuploader* mikroszolgáltatás *RabbitMQ* várakozási sorokból szerzi meg, ezt a lépést valósítja meg a *rabbitmq package*. A *parser* és *postprocessor* mikroszolgáltatásokhoz hasonlóan itt is egy interfész mögé rejtettem a szükséges funkcionalitást. Az interfész a *message_consumer.go* fájlban, a megvalósítása pedig az *amqp_consumer.go* fájlban található. Mind az interfész, mind a megvalósítás szinte azonos, mint a *postprocessor* szolgáltatásban lévő *MessageConsumer* esetében, így ezt nem részletezem még egyszer.

4.2.3.3 Az adatok feltöltése

Az *Elasticsearch*-el történő interakcióért az *esutil package* felelős, melyben két fájl található. Az *es_client.go* fájlban lévő *EsClient* interfészben definiáltam azokat a funkciókat, amelyeket a szolgáltatás használ az adatok indexeléséhez, az *es_client_wrapper.go* fájlban pedig ennek az interfésznek a megvalósítása található. Az interfész implementációjához a github.com/elastic/go-elasticsearch kliens könyvtárat használtam fel. Az interfész két funkcióval rendelkezik:

- `CreateEsIndex(index string)`: Egy *ES* indexet hoz létre, ha az már létezik, akkor először kitörli a meglévőt, majd újra létrehozza a paraméterben kapott névvel.
- `BulkUpload(dataUnits []models.ESDocument, indexName string)`: A paraméterben kapott adatok kötegelt indexelését végzi. Miután végrehajtotta a kérést az adatoknak a megadott indexbe történő feltöltésére, kiírja a konzolra a feltöltés eredményét – hány darab sort töltött fel mennyi idő alatt és ezekből mennyinél keletkezett hiba a folyamat közben. A feltöltéshez a *go-elasticsearch* könyvtár *esutil package*-ének *BulkIndexer* komponensét használtam.

Az adatok feltöltésének vezérlését az ***uploader*** *package*-ben található *UploaderService* végzi. Ez a komponens használja az előzőekben bemutatott *EsClient* és *MessageProducer* komponenseket, és koordinálja az adatok feltöltését. A *RabbitMQ* várakozási sorokból kivett adatokat egy *UploadBuffer* típusú ideiglenes tárolóba helyezi, amely gondoskodik az adatok feltöltéséről, és az adatokhoz történő hozzáféréskor a kölcsönös kizárás megvalósításáról. Az ideiglenes tárolóból az adatok két különböző feltétel valamelyikének teljesülése esetén kerülnek feltöltésre, amely feltételek a következők:

- Az ideiglenes tárolóban lévő adatok száma eléri a megadott küszöbértéket (ez most ezer darabra van beállítva).
- Az utolsó feltöltés óta eltelt legalább öt másodperc, és az ideiglenes tároló nem üres.

Ezzel a módszerrel igyekeztem elkerülni azt, hogy rövid időn belül túl sok kérés legyen küldve az *Elasticsearch* szolgáltatásnak, ezzel javítva az adatok indexelésének sebességét. Az időzített feltöltés abban segít, hogy amikor elérjük a feldolgozás végét, de a tárolóban lévő adatok száma nem éri el az ezret, akkor is fel legyen töltve minden adat.

Ezen felül az *UploadBuffer* két feltöltés között a *BackupBuffer* komponens használatával bizonyos mennyiségként elmenti a még nem feltöltött adatokat egy JSON fájlba a lokális fájlrendszerbe. Ez azt a célt szolgálja, hogy ha az *elasticuploader* mikroszolgáltatás valamilyen oknál fogva újraindul, akkor ne veszítsük el a buffer-ben lévő adatokat. A *BackupBuffer* komponenst úgy valósítottam meg, hogy az minden 10 adategység után elmentse a kapott adatokat egy JSON fájlba, amelybe az adatokon kívül

az éppen használatos index neveket is beleírja, hogy a szolgáltatás legközelebbi elindulásakor tudja, hogy milyen indexbe kell feltöltenie ezeket az adatokat.

Az *UploadBuffer* komponens a buffer-ben lévő adatok feltöltésekor (küszöbérték vagy megadott idő elérése után) a *BackupBuffer* megfelelő függvényeinek meghívásával törli annak tartalmát, valamint az elmentett JSON fájl tartalmát is. A szolgáltatás minden elindulásakor megnézi, hogy talál-e valamilyen adatot a biztonsági mentés fájlban, és ha igen, akkor feltölti azokat a fájlban rögzített nevű indexbe. Ezzel a megoldással egy újraindítás esetén maximum kilenc adategységet veszíthetünk el, az enélkül bekövetkező 999 helyett. Ha minden egyes adatot kiírnánk JSON fájlba, akkor az adatvesztést teljesen elkerülhetnénk, viszont szerettem volna spórolni a fájlműveletekkel a teljesítmény javítása érdekében. Az *UploadBuffer* megvalósítása az *upload_buffer.go* fájlban, a *BackupBuffer* pedig a *backup_buffer.go* fájlban található.

4.2.3.4 Időzített index létrehozás

Az adatok *ES* indexekbe mentéséhez az *elasticuploader* szolgáltatásnak gondoskodnia kell a megfelelő indexek létezéséről is. Az adatokat két különböző féle indexbe szerveztem, az egyikbe az események, a másikba a fogyasztási adatok kerülnek.

Ahhoz, hogy az összes feldolgozás eredményeként keletkezett adatok ne mind két hatalmas indexbe kerüljenek, úgy valósítottam meg a feltöltő szolgáltatást, hogy minden nap éjfélkor új indexeket hozzon létre, és a későbbiekben már ezekbe az új indexekbe mentse az adatokat. Ezen felül szolgáltatás elindításakor is létrejön egy új index, hogy ez a speciális eset is kezelve legyen. Minden index nevének a végére kerül a létrehozás pillanatának megfelelő időbélyeg, így biztosítva a nevek egyediségét, és segítve a későbbiek során annak beazonosítását, hogy az indexben található adatok melyik napon keletkeztek.

Az időzítéshez a *github.com/robfig/cron* könyvtárat használtam fel. [29] Ennek egy *string* formátumú konfigurációt kell megadni ahhoz, hogy időzítse az általunk definiált függvény végrehajtását, például az „*@midnight*” azt jelenti, hogy minden éjfélkor végre szeretnénk hajtani az adott műveletet. Erről több információ a könyvtár dokumentációjában [30] érhető el. Az általam időzített logika először lezárja az *UploadBuffer mutex* mezőjét, hogy más *goroutine* közben ne nyúlhasson a buffer tartalmához, majd a buffert kiüríti és a tartalmát feltölti *Elasticsearch*-be. Ezután létrehozza az új indexeket, majd feloldja a zárolást.

4.3 Adattárolás és vizualizáció

Ahogy már az előző fejezetekben is utaltam rá, az adatok tárolását egy *Elasticsearch* adatbázisban valósítottam meg, amelynek egyik oka, hogy a vizualizációk elkészítéséhez a *Kibana*-t szerettem volna használni annak széleskörű vizualizációs képességei miatt. Ebben az alfejezetben először bemutatom, hogy milyen *ES* indexekben kerülnek tárolásra az adatok, majd néhány példán keresztül szemléltetem az adatok megjelenítését *Kibana*-ban.

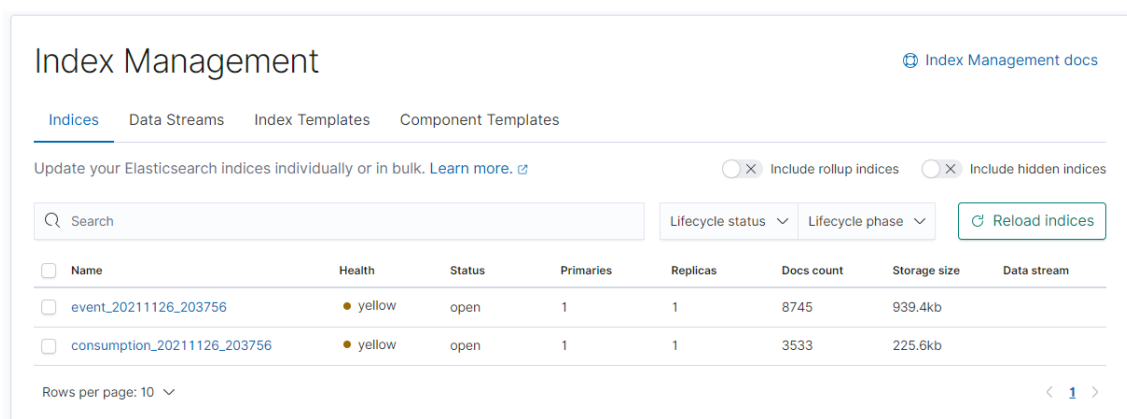
4.3.1 Adattárolás

A naplófájlokból kinyert adatokat két féle indexbe mentem el, az egyikbe az *SMC*-kkel kapcsolatos események, a másikba pedig a fogyasztási adatok kerülnek.

Az eseményeket tároló indexek neve a következő mintára illeszkedik: az indexnév az „*event*” szóval kezdődik, majd a „_” elválasztó karakter után a létrehozás pillanatának megfelelő időbélyeg látható „*yyyyMMdd_HHmmss*” formátumban. Például az „*event_20211126_203756*” indexnév a 2021.11.26. napon 20 óra 37 perc 56 másodperc időpillanatban létrejött eseményeket tároló indexet jelenti.

A fogyasztási adatok indexének neve hasonlóan néz ki, mint az eseményeknél, csak itt „*event*” helyett a nevek a „*consumption*” szóval kezdődnek.

Az alábbi ábrán látható a *Kibana Index Management* felülete, ahol az *Elasticsearch cluster* által jelenleg tárolt indexek láthatók.



The screenshot shows the Kibana Index Management interface. At the top, there's a header with 'Index Management' and a link to 'Index Management docs'. Below the header, there are tabs for 'Indices', 'Data Streams', 'Index Templates', and 'Component Templates'. A sub-header says 'Update your Elasticsearch indices individually or in bulk. Learn more.' with a link. There are two toggle switches: 'Include rollout indices' and 'Include hidden indices'. Below these, there's a search bar and two dropdown menus for 'Lifecycle status' and 'Lifecycle phase'. A 'Reload indices' button is also present. The main part of the interface is a table with the following columns: Name, Health, Status, Primaries, Replicas, Docs count, Storage size, and Data stream. Two indices are listed: 'event_20211126_203756' and 'consumption_20211126_203756'. Both have a 'yellow' health status and 'open' status. The table also shows 1 primary and 1 replica for each index, along with document counts and storage sizes. At the bottom, there's a 'Rows per page: 10' dropdown and a pagination control showing '1'.

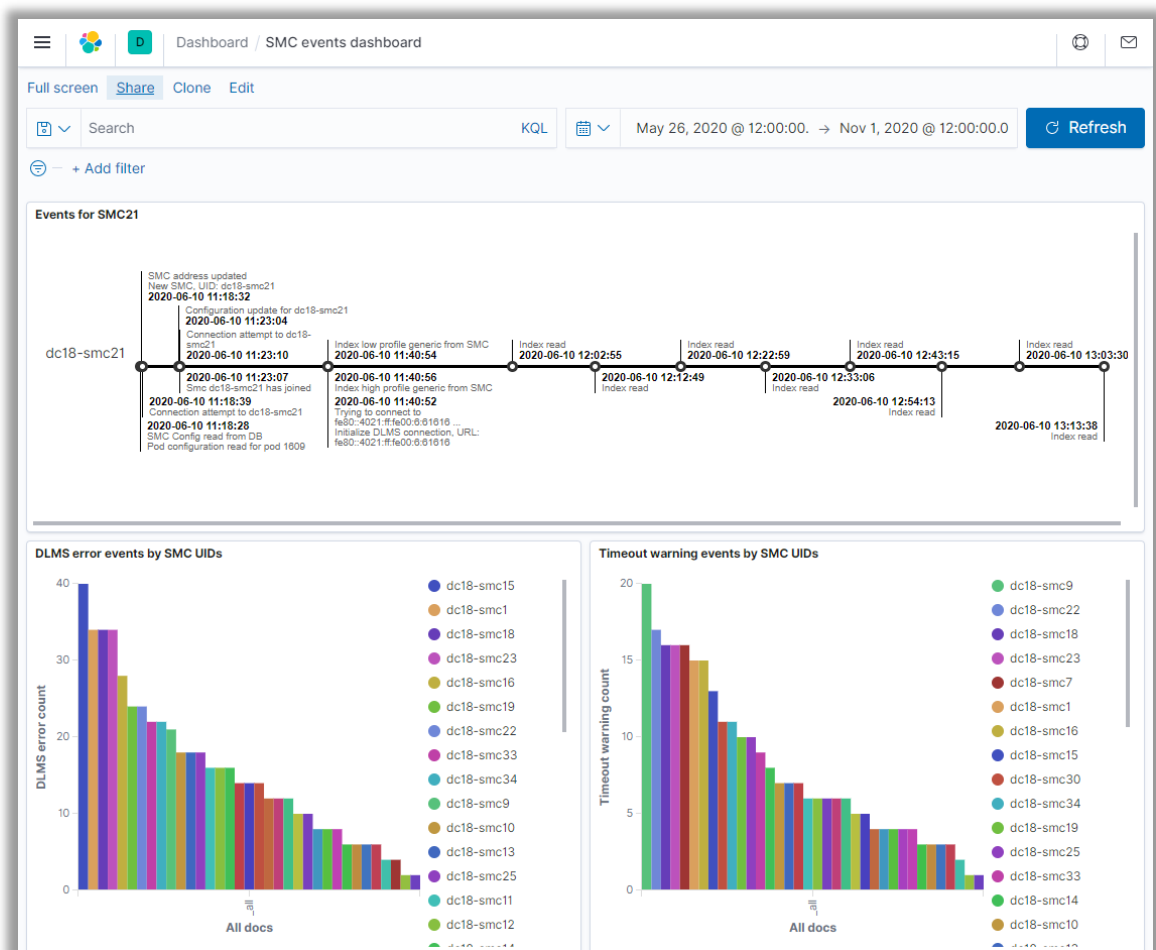
Name	Health	Status	Primaries	Replicas	Docs count	Storage size	Data stream
<input type="checkbox"/> event_20211126_203756	● yellow	open	1	1	8745	939.4kb	
<input type="checkbox"/> consumption_20211126_203756	● yellow	open	1	1	3533	225.6kb	

4.3.1. ábra: A Kibana Index Management felülete

4.3.2 Vizualizáció

A feldolgozás eredményeként kapott adatokhoz két *Kibana Dashboard*-ot készítettem, egyet az események, egyet pedig a fogyasztás változásának ábrázolásához.

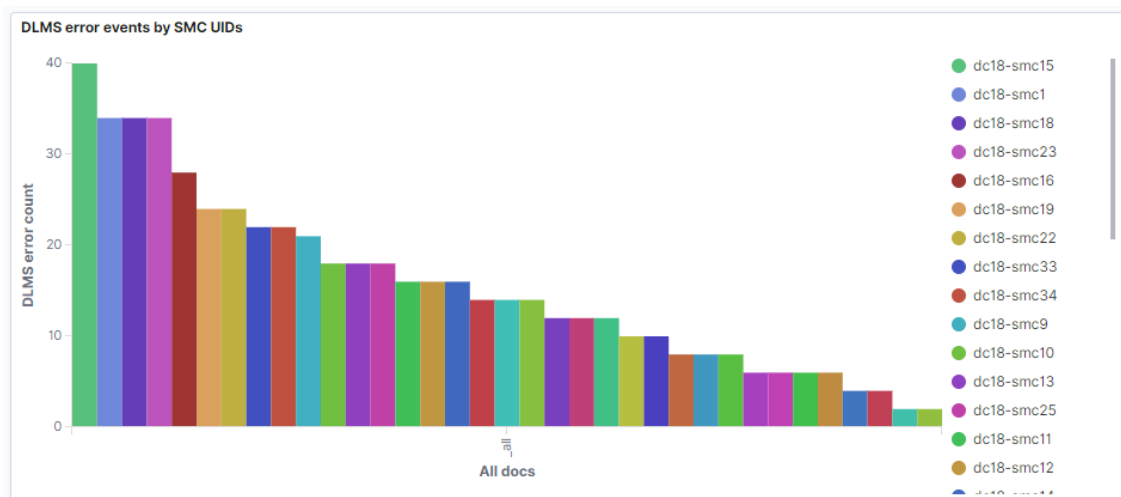
Az eseményekhez tartozó *Dashboard* látható a következő ábrán.



4.3.2.1. ábra: SMC events dashboard

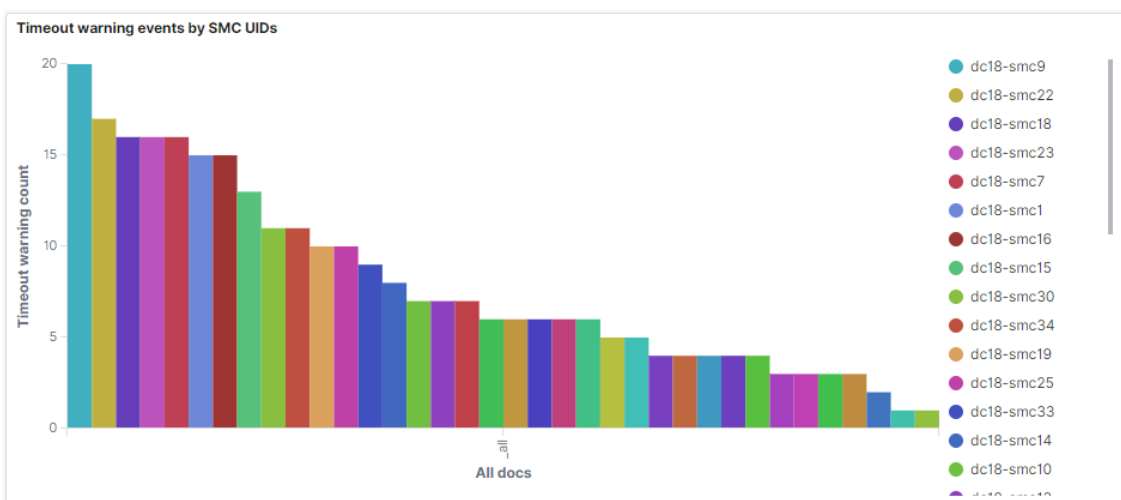
Az alsó két diagramon a hibák (*error*), illetve figyelmeztetések (*warning*) száma látható *SMC* azonosítónként csoportosítva, a felső diagramon pedig azt láthatjuk, hogy milyen események történtek a *dc18-smc21* azonosítójú *SMC*-vel 2020.06.10-én.

A bal alsó ábráról leolvasható, hogy a legtöbb hibaesemény a *dc18-smc15* azonosítójú, míg a legkevesebb a *dc18-smc5* azonosítójú (ez már nem fért bele a képbe) *Smart Meter Controller*-rel volt kapcsolatos. Ez az oszlopdiagram látható felnagyítva a következő ábrán.



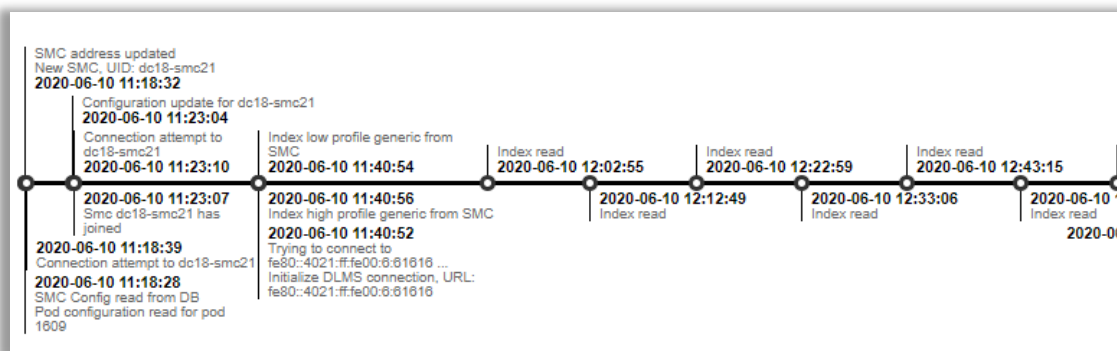
4.3.2.2. ábra: A hiba jellegű események számának ábrázolása oszlopdiagramon az egyes SMC azonosítók szerint csoportosítva

A dashboard jobb alsó diagramján látható, hogy a legtöbb figyelmeztetés szintű esemény a *dc18-smc9*, míg a legkevesebb a *dc18-smc5* azonosítójú SMC esetében fordult elő. Ennek a felnagyított verzióját mutatja a következő kép.



4.3.2.3. ábra: A figyelmeztetés jellegű események számának ábrázolása oszlopdiagramon az egyes SMC azonosítók szerint csoportosítva

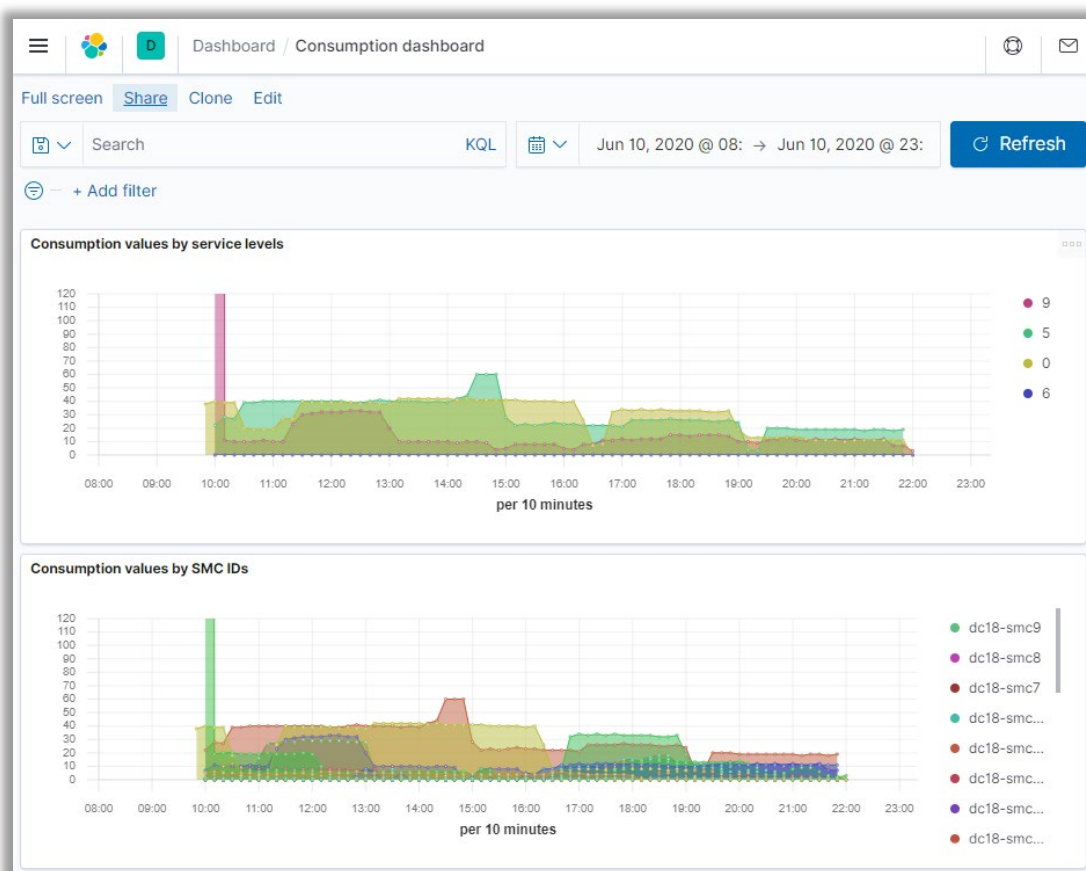
A legfelső diagram kissé felnagyítva az alábbi ábrán látható. Ez egy ún. *Milestones* (mérőföldkő) vizualizáció, amely segítségével különböző események egymásutániségát tudjuk ábrázolni. Ezt a fajta diagramot a *Kibana* alapvetően nem tartalmazza, a létrehozásához a *kibana-milestones-vis* plugint használtam fel.



4.3.2.4. ábra: Mérföldkő-vizualizáció a dc18-smc21 azonosítóval rendelkező SMC eseményeiről

Ezen az ábrán láthatjuk, hogy a *dc18-smc21* azonosítóval rendelkező *SMC*-hez kétszer is megpróbált csatlakozni a *DC*: először 2020.06.10. 11:18:39-kor, majd 11:23:10-kor újra. Az is látható az ábrán, hogy az első próbálkozás valószínűleg azért nem sikerült, mert az *SMC* csak 11:23:07-kor csatlakozott a hálózathoz. Az idővonal második felén pedig megfigyelhetjük a *DC* által végzett periodikus adatgyűjtést az „*Index read*” nevű események formájában.

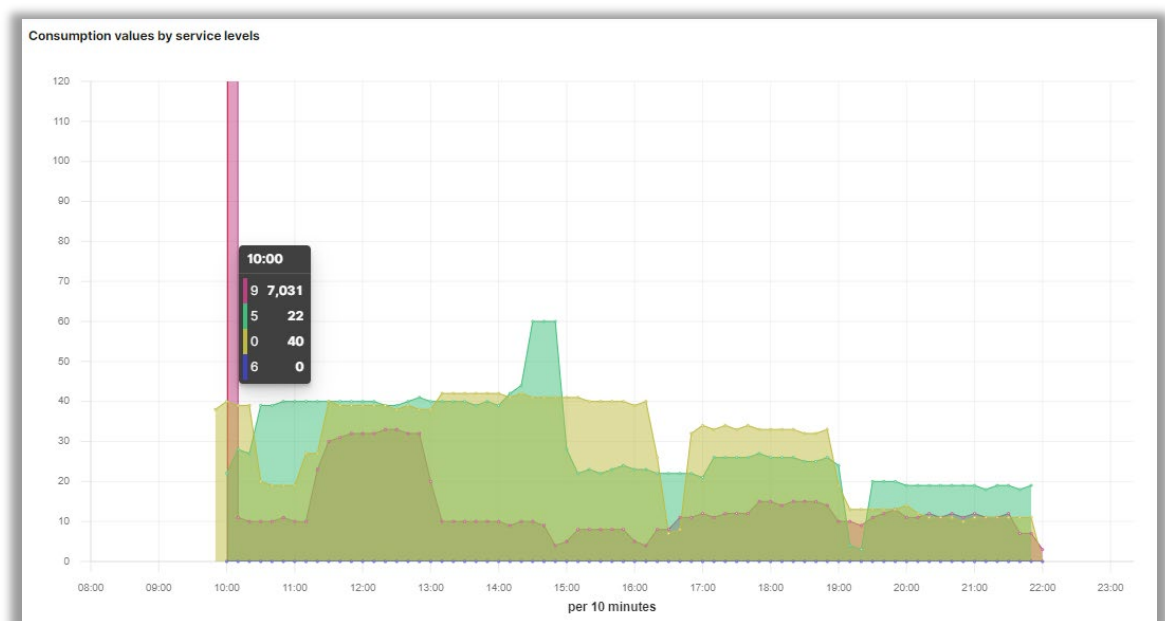
A fogyasztási adatokat megjelenítő *Dashboard*-ot mutatja a következő kép.



4.3.2.5. ábra: A fogyasztási adatokat ábrázoló dashboard

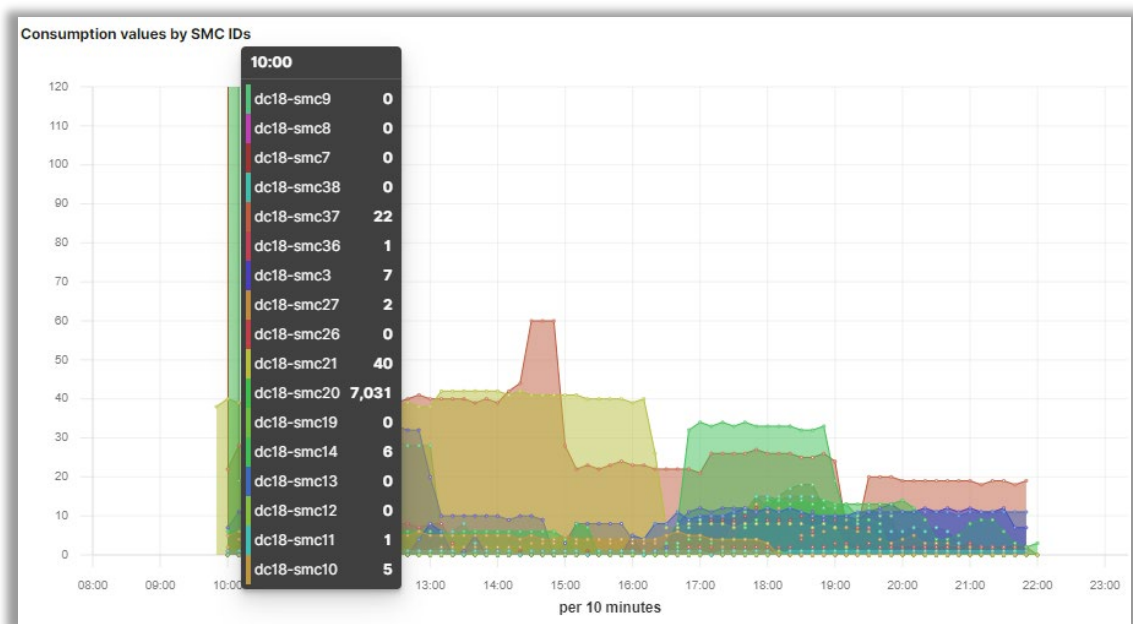
Ezen a dashboard-on két *Time Series* (idősor) típusú vizualizációt helyeztem el, a fogyasztási adatok két különböző szempontból történő ábrázolására.

A felső diagram a különböző szolgáltatási szinteken mérhető fogyasztás változását mutatja az idő folyamán. Erről leolvashatjuk, hogy a 9-es azonosítójú szolgáltatási szint esetében délelőtt 10 óra környékén van egy nagyon kiugró érték a többi mért adathoz képest. Ez valamilyen anomália jelenlétére enged következtetni, hiszen 10 óra után erről a szolgáltatási szintről is csak maximum ~35 nagyságú mérések érkeznek, amihez képest a 10:00-kor mért 7031-es érték irreálisan magas. A diagram kissé felnagyítva látható a következő ábrán.



4.3.2.6. ábra: Az energiafogyasztás időbeli változásának ábrázolása szolgáltatási szint azonosítónként csoportosítva

A dashboard alsó diagramján szintén az energiafogyasztás időbeli változását követhetjük, de ezúttal *SMC* azonosítónként csoportosítva. Ezt megvizsgálva azt is megállapíthatjuk, hogy a kiugróan magas 7031 érték a dc18-smc20 azonosítóval rendelkező *SMC* felől érkezett. Ezt a diagramot a következő ábrán láthatjuk felnagyítva.



**4.3.2.7. ábra: Az energiafogyasztás időbeli változásának ábrázolása szolgáltatási szint
azonosítónként csoportosítva**

4.4 Tesztelés

Egy szoftver fejlesztése során a tesztelés egy nagyon fontos lépés a hibák feltárása és a helyes működés ellenőrzése céljából. Ebben az alfejezetben azt ismertetem, hogy milyen módon valósítottam meg az általam fejlesztett rendszer tesztelését, milyen teszteseteket fedtem le az elkészült tesztekkel, illetve hogyan biztosítom, hogy a fejlesztés során a meglévő funkciók működőképeseek maradjanak.

A teszt kódot és a tesztekkel kapcsolatos erőforrásokat minden szolgáltatás esetében a *tests* mappában helyeztem el. A *Go* ökoszisztémában a teszt kódot általában nem egy külön helyre, hanem a tesztelt viselkedés mellé, ugyanabba a *package*-be teszik. [31] Ezt a konvenciót én azért nem követtem, mert számomra logikusabb és átláthatóbb az összes teszteléshez kapcsolatos fájlt egy helyen tartani, elkülönítve a forráskód többi részétől.

4.4.1 Unit tesztek

Az egyes mikroszolgáltatások tesztelésének első szintje az egyes komponensek és függvények izolált tesztelése unit tesztek segítségével. Ennek során mindhárom mikroszolgáltatás esetében a főbb funkciókhoz készítettem unit tesztek, ezeket fogom ebben az alfejezetben bemutatni.

4.4.1.1 A parser mikroszolgáltatás unit tesztjei

A *parser* mikroszolgáltatás főbb funkcióihoz tartozó unit tesztek négy *package*-ben találhatók a *tests* mappán belül, amelyek: *loglevelparser_unit_tests*, *timestampparser_unit_tests*, *contentparser_unit_tests* és *logparser_unit_tests*.

A *loglevelparser_unit_tests* *package*-ben a *loglevelparser* *package* működését ellenőrző unit teszt található. Ebben a tesztben egy-egy INFO, WARN, WARNING, ERROR, DEBUG és VERBOSE naplózási szintű naplóbejegyzésre ellenőrzöm, hogy a *loglevelparser* a megfelelő naplózási szinteket olvassa-e ki belőlük, illetve kiszűri-e az irreleváns bejegyzéseket (VERBOSE és DEBUG), és így a megfelelő objektumok állnak-e elő a folyamat eredményeként.

A kódismétlés elkerülése végett az egyes teszteseteket (bemeneti naplóbejegyzés és elvárt kimenet párok) egy listába rendeztem, és a konkrét ellenőrzést végző kódot a lista egyes elemeire futtatom egy *for* ciklussal (ezt a megoldást alkalmaztam a többi unit teszt megírásakor is).

A *timestampparser_unit_tests* a *timestampparser package* működését ellenőrzi. Ebben a tesztben a *loglevelparser* tesztjeihez hasonló módon definiáltam teszt eseteket, különböző formátumú sorokat adva bemenetként, és ellenőrzöm, hogy a megfelelő idő kerül-e kiolvasásra ezekből a *timestampparser* által.

A *contentparser_unit_tests* a *contentparser package* funkcionalitásának ellenőrzésére tartalmaz tesztek. Az előző tesztekhez hasonlóan itt is egy teszt eset-listát használtam a bemenetek és az elvárt kimenetek megadására, amelyeken végig haladva ellenőrzöm, hogy a *contentparser* által visszaadott *ParsedLogEntry* objektum tartalma megfelel-e az elvártaknak.

A *logparser_unit_tests package* a *logparser package* működésének helyességét ellenőrzi. Ebben a *package*-ben két teszt függvény található, melyek közül az első egy *dc_main.log* fájlban megfelelő formátumú teszt naplófájlból, a második pedig egy *plc_manager.log* formátumára illeszkedő teszt naplófájlból előálló eredmények helyességét ellenőrzi. Az ellenőrizhetőség leegyszerűsítése végett ezek a naplófájlok sokkal kisebb terjedelműek, mint a valódi naplófájlok, de a tartalmukat igyekeztem változatosan összeállítani a valódi fájlokban megtalálható sorokból. Ezek a unit tesztek inicializálnak egy *LogParser*-t, majd a *ParseLogfiles* függvényét meghívva indítják el a naplófájlok felolvasását. A *LogParser* komponensnek két függősége van: egy *MessageProducer*, mely segítségével a felolvasott bejegyzéseket küldi a *RabbitMQ* szervernek, és egy *Filedownloader*, amelyet a feldolgozni kívánt naplófájlok letöltésére használ. Ezekhez *mock* implementációkat készítettem, amelyeket felhasználtam a unit tesztek megírásához. A *mock* implementációkat a *tests* mappán belül található *mocks package*-ben helyeztem el.

A *Filedownloader* interfész *mock* implementációja a *MockFileDownloader*, amely egy *FileNameToDownload* mezővel rendelkezik, amely a letöltendő fájl nevét tartalmazza. Ezt a fájl nevet a *mock* komponens létrehozásakor adjuk meg, és a *Filedownloader* interfészből implementált *ListFileNames* függvény ezt a nevet fogja visszaadni a hívó félnek. A *DownloadFile* függvényt pedig úgy valósítottam meg, hogy a lokális fájlrendszerben keresse a megadott elérési útvonallal a letöltendő fájlt, és azt adja vissza a hívónak.

A *MessageProducer* interfészhez a *MessageProducerMock* implementációt készítettem el, amely egy *Entries* mezőt tartalmaz, ahova a *PublishEntry* függvény elhelyezi a paraméterben kapott *ParsedLogEntry* objektumokat. Ezáltal a teszt kódból

egyszerűen le tudom kérdezni azt, hogy milyen naplóbejegyzéseket küldött el a *LogParser* komponens. Az interfész összes többi függvényéhez üres implementációt készítettem, mivel ezeknek nincs jelentősége az elkészült unit tesztek esetében.

A tesztekben a létrehozott *MockFileDownloader* komponensnek átadásra kerül a *logparser_unit_tests/resources* mappában lévő valamelyik teszt log fájl, majd a *MessageProducerMock Entries* mezőjének tartalmát hasonlítom össze a *resources* mappában JSON formátumba sorosítva tárolt elvárt kimenettel.

4.4.1.2 A *postprocessor* mikroszolgáltatás unit tesztjei

A *postprocessor* szolgáltatáshoz tartozó unit tesztek a *processor_unit_tests* *package*-ben találhatók. Ez a *package* négy teszt fájlt tartalmaz: *info_processor_test.go*, *warning_processor_test.go*, *error_processor_test.go* és *processor_test.go*, melyek sorban az *InfoProcessor*, *WarningProcessor*, *ErrorProcessor* és *EntryProcessor* működését tesztelik.

Az *InfoProcessor*, *WarningProcessor* és *ErrorProcessor* tesztjei egyszerűek: adott bemenetekre ellenőrzik, hogy megfelelő eredmények állnak-e elő a feldolgozásuk során.

Az *EntryProcessor* komponens unit tesztje annyival bonyolultabb, hogy itt szükség volt a komponensek függőségeinek mock implementációkkal való helyettesítésére. Ebben az esetben három mock implementációt készítettem: egyet a *MessageConsumer* interfészhez, egyet a *MessageProducer* interfészhez és egyet az *amqp package Acknowledger* interfészéhez. [32] Ez utóbbira azért volt szükség, mert a *MessageConsumer* interfészt implementáló *MockMessageConsumer ConsumeMessages* függvénye *amqp.Delivery* típusú objektumokat kell, hogy tegyen egy csatornára, amelyet visszaad a hívónak, ezek létrehozásához pedig szükség van egy *Acknowledger* megadására. Ennek az *Acknowledger*-nek az *Ack* függvénye kerül meghívásra az *EntryProcessor* által. Egy *amqp.Delivery* létrehozása látható az alábbi kódrészleten:

```
func NewMockDelivery(data []byte, tag uint64) amqp.Delivery {
    acknowledger := MockAcknowledger{}
    delivery := amqp.Delivery{
        Acknowledger: &acknowledger,
        DeliveryTag:   tag,
        Body:         data,
    }
    return delivery
}
```

A *MockAcknowledger* megvalósításakor az *Acknowledger* interfész minden függvényéhez üres implementációt készítettem, mivel ezek nem játszanak fontos szerepet a unit tesztben, mindössze annyi a lényeg, hogy az *amqp.Delivery* objektumok *Ack* függvényének meghívása ne okozzon futási idejű hibát (ez történne, ha nem adnánk meg *Acknowledger*-t a létrehozásukkor).

A *MockMessageConsumer* a parser mikroszolgáltatás felől *RabbitMQ*-n keresztül érkező üzeneteket hivatott helyettesíteni, azáltal, hogy a *RabbitMQ* szerverrel történő kommunikáció helyett összerak egy `<-chan amqp.Delivery`-csatornát, amelyet visszaad a hívónak. Ezeket a bemeneti üzeneteket teszt adatokból állítja elő, amelyet a *mock* objektum létrehozásakor adunk meg.

A *MessageProducer* interfész *mock* implementációját a *MockMessageProducer* tartalmazza. Ez helyettesíti a feldolgozott adatok *RabbitMQ* felé történő továbbítását. Mivel az adatok feldolgozása aszinkron történik (a *RabbitMQ* sorból való üzenetfogyasztás miatt), a teszt végrehajtásakor valahogyan meg kell várnunk, amíg a feldolgozás az adatok végére ér. Ezt úgy valósítottam meg, hogy a *MockMessageProducer*-nek átadtam egy csatornát (`done chan string`), amire az jelez, mikor az összes várt adat közzé lett téve (ezt az elvárt adatok és közzétett adatok száma alapján állapítja meg). A *PublishEvent* és *PublishConsumption* függvények a kapott adatokat a *MockMessageProducer Data* nevű mezőjébe teszik bele, ahonnan a teszt ellenőrző fázisában lekérdezem, hogy mi lett az *EntryProcessor* által végzett feldolgozás eredménye.

A teszthez tartozó bemeneti adatok és elvárt kimenetek a *resources* mappában található JSON formátumban.

4.4.1.3 Az elasticuploader mikroszolgáltatás unit tesztjei

Az *elasticuploader* mikroszolgáltatás unit tesztjei az *uploader_unit_tests* *package*-ben található. Ebben a *package*-ben két teszt fájlt láthatunk, amelyek a következők: *uploader_test.go* és *uploader_timed_test.go*.

Az *uploader_test.go* fájl egy olyan unit tesztet tartalmaz, amely azt ellenőrzi, hogy egy adott teszt adathalmazból minden elemet elküld-e feltöltésre az *UploaderService* komponens. Ennek megvalósításához az *UploaderService* két függőségének helyettesítésére *mock* implementációkat készítettem az *EsClient* és *MessageConsumer* interfészekhez, amely implementációk a *mocks* *package*-ben található.

Az *EsClientMock* valósítja meg az *EsClient* interfészt. Az *EsClientMock* egy *Indexes* nevű *map* típusú mezővel rendelkezik, amely esetében a kulcsok az indexek nevei, az értékek pedig *ESDocument* tömbök (az *UploaderService* ilyen *ESDocument* típusú adatokat ad át az *EsClient BulkUpload* függvényének ES adatbázisba történő feltöltésre). Ebbe a *map* struktúrába kerülnek az *EsClientMock BulkUpload* függvényének átadott adatok, így a teszt ellenőrző fázisában lekérdezhető, hogy adott index névhez milyen adatok lettek elküldve az *UploaderService* által.

A *MessageConsumer* interfészt a *MessageConsumerMock* implementálja. Ez helyettesíti a valódi *RabbitMQ* szervertől való adatbeszerzést a feltöltés elvégzéséhez. Hasonlóan a *postprocessor* mikroszolgáltatáshoz, a *Consume* függvény itt is egy *amqp.Delivery*-csatornát készít a teszt adatokból, amelyhez ezúttal is készítettem egy *mock* implementációt az *Acknowledger* interfészhez. Ez a megvalósítás azonban annyiban különbözik az előzőtől, hogy itt a *MockAcknowledger*-nek átadásra kerül egy csatorna, amin jelezhet, ha minden *Delivery* megerősítésre került (az *Ack* függvény meghívásával), és a várt adatok száma, amely segítségével megállapíthatja, hogy minden bemeneti adatra meghívódott-e az *Ack* függvény. Erre azért van szükség, hogy a teszt kód be tudja várni az adatok aszinkron módon történő kezelését.

A *MessageConsumerMock* létrehozásakor továbbá lehetőségünk van megadni egy *deliveryDelaySeconds* paramétert, amelyet arra használok, hogy egy mesterséges késleltetést helyezzek a bemeneti adatok előállításába. Ezt a funkciót az *uploader_timed_test.go* fájlban található tesztben használok ki, amely azt hivatott ellenőrizni, hogy az időzített index létrehozás megfelelően működik-e. Ehhez a tesztben használt *UploaderService* példányt úgy állítottam be, hogy az új indexek létrehozásának időintervalluma 10 másodperc legyen, a bemeneti adatok előállításába pedig szintén egy 10 másodperces késleltetést helyeztem el, így végeredményként a bemeneti adatoknak két külön indexbe kellett kerülnie, ezt ellenőrzi ez a unit teszt.

A tesztekhez tartozó bemeneti adatok az *uploader_unit_tests/resources* mappában találhatóak JSON formátumban.

4.4.2 Integrációs tesztek

A mikroszolgáltatások tesztelésének második szintje az integrációs tesztelés volt, amely segítségével azt ellenőriztem, hogy az egyes mikroszolgáltatások külső szolgáltatásokkal történő interakciójakor minden az elvárt módon működik-e. Ilyen külső szolgáltatásokkal történő interakció az *Elasticsearch* adatbázisba való adatfeltöltés, és az üzenetek termelése/fogyasztása a *RabbitMQ* várakozási sorokból.

4.4.2.1 A parser mikroszolgáltatás integrációs tesztjei

A *parser* mikroszolgáltatáshoz készült integrációs tesztek a *parser tests* mappáján belül a *logparser_integartion_tests package*-ben találhatók. Ebben egy darab teszt fájlt helyeztem el, amely két teszt esetet tartalmaz: az első egy *dc_main.log* formátumának megfelelő teszt napló fájlt, a második pedig egy *plc_manager.log* formátumú teszt naplófájlt használ bemenetként.

Mindkét integrációs teszt a *parser* mikroszolgáltatás és a *RabbitMQ* szerver közötti interakció tesztelésére szolgál, a *LogParser* komponens inicializálásakor a fájlok letöltéséért felelős *Filedownloader* típusú függőséget a unit tesztekhez hasonlóan itt is egy *mock* implementációval helyettesítettem.

A tesztekben először inicializálom a *LogParser* függőségeit: az előbb említett *mock Filedownloader*-t, és egy *AmqpProducer*-t, amely a *MessageProducer* interfész github.com/streadway/amqp könyvtárt felhasználó, valódi *RabbitMQ* szerverrel kommunikáló implementációja és amelyet a valós kódban is használ a *LogParser* komponens. A valódi környezethez képest itt annyi az eltérés, hogy a *routing key* és *exchange name* értékeket teszt értékekkel helyettesítettem. A fentiekén kívül létrehoztam még egy *RabbitMQ* üzenetek fogyasztására alkalmas komponenst (*testConsumer*), amelyet csak a tesztekben használok, hogy a tesztek végén le tudjam ellenőrizni az eredményeket, hogy milyen üzeneteket küldött el a *LogParser* a *RabbitMQ exchange* felé. Fontos, hogy a *testConsumer ConsumeMessages* függvényét még a *LogParser ParseLogfiles* függvényének meghívása előtt kell meghívni, hiszen a *RabbitMQ* várakozási sor létrehozása a fogyasztó oldal felelőssége, a termelő (azaz a *parser*) erről nem tud, csak az általa használt *exchange*-ről, ha pedig a sor nem létezik és nincs az *exchange*-hez kötve egy üzenet küldésének pillanatában, akkor azt az üzenetet a *RabbitMQ unroutable*-nek jelöli, és nem tudja kézbesíteni.

A feldolgozott naplóbejegyzések megszerzését a *getSentParsedEntries* segédfüggvény végzi, amely blokkol, ameddig nem kapja meg a bejegyzések végét jelentő „END” üzenetet. Ezután az eredményeket a *resources* mappában JSON formátumban tárolt elvárt kimenethez hasonlítva dönthető el a teszt sikeressége.

4.4.2.2 A *postprocessor* mikroszolgáltatás integrációs tesztjei

A *postprocessor* mikroszolgáltatás integrációs tesztjeit a *tests* mappán belül található *processing_integration_tests* package-ben helyeztem el. Itt egy fájl található, benne két tesztesettel, amelyek az előző szolgáltatás integrációs tesztjeihez hasonlóan egy *plc_manager.log* formátumú és egy *dc_main.log* formátumú teszt fájlból származó adatok feldolgozását tesztelik egy valódi *RabbitMQ* szerver használatával.

Az *EntryProcessor* komponenst ezúttal, a unit tesztekkel ellentétben egy-egy valós *RabbitMQ* szerverrel való kommunikációt megvalósító *MessageProducer* és *MessageConsumer* megadásával inicializáltam. Ezeken kívül még létrehoztam egy *testOutputConsumer* nevű komponenst, amely a *parser* integrációs tesztjeihez hasonlóan a feldolgozás eredményét veszi ki a megfelelő *RabbitMQ* várakozási sorból az ellenőrzés elvégzéséhez, illetve egy *testInputProducer*-t, amely a teszt adatokat küldi abba a várakozási sorba, ahonnan az *EntryProcessor* várja a bemeneti adatokat.

A feldolgozás végrehajtása után a *testOutputConsumer* segítségével kiolvassuk a *RabbitMQ* várakozási sorból az eredményeket, majd ezt az elvárt kimenethez hasonlítva eldönthetjük a teszt kimenetelét.

A bemeneti adatok és az elvárt kimenetek a *resources* mappában vannak tárolva, JSON formátumban.

4.4.2.3 Az *elasticuploader* mikroszolgáltatás integrációs tesztjei

Az *elasticuploader* mikroszolgáltatáshoz két integrációs tesztet készítettem el, amelyeket a *tests* mappán belül található *uploader_integration_tests* package-ben helyeztem el. Az első *TestServiceIntegrationWithElasticsearch* névre hallgató teszt a feltöltést végző szolgáltatás *Elasticsearch*-el történő integrációját teszteli, míg a második, *TestServiceIntegrationWithRabbitMQ* a *RabbitMQ*-val történő kommunikáció helyes működését ellenőrzi.

Az *ES* integrációt ellenőrző tesztben az *UploaderService* létrehozásakor a *MessageConsumer* típusú függőséget mock implementációval helyettesítettem, az

EsClient típusúhoz pedig egy valódi környezetben is használt *EsClientWrapper*-t hoztam létre. A *mock* implementációról a unit tesztekéről szóló alfejezetben már írtam, így azt itt nem részletezem újra. Az eredmények ellenőrzéséhez ennél a tesztnél szükség volt még egy *ES* kliensre, amely segítségével lekérdezhető az egyes indexek tartalma, illetve a teszt után a teszt-indexek kitörölhetőek. Ennek megvalósítása a *testutils package*-ben lévő *tes_es_client.go* fájlban található.

A *RabbitMQ*-val történő kommunikáció teszteléséhez a valódi környezetben is használt *AmqpConsumer* komponenst használtam, és az *UploaderService* *EsClient* típusú függőségét helyettesítettem *mock* implementációval. Ezt a *mock* implementációt már szintén bemutattam a unit tesztek alfejezetében. Ebben a tesztben, a *postprocessor* szolgáltatás integrációs tesztjeihez hasonlóan, szükség volt a teszt adatok *RabbitMQ* felé történő eljuttatására, hogy az *UploaderService* innen olvashassa ki a feltöltendő adatokat. Ezt a lépést a *testutils package* *TestRabbitMqProducer* típusa valósítja meg. Az eredmények ellenőrzése ennél a tesztnél egyszerűbb, hiszen csak a *mock elasticsearch* kliens *Indexes* mezőjének tartalmát kell ellenőriznünk.

A teszteléshez használt bemeneti adatok az *uploader_integration_tests/resources* mappában érhetők el.

4.4.3 CI automatizált teszt futtatás

Annak biztosítása érdekében, hogy az új funkciók fejlesztése, hibák javítása, esetleg a kód refaktorálása közben a meglévő működés ne romoljon el, a fentebb bemutatott tesztek automatizált futtatását is beállítottam a projekthez tartozó *GitHub repository*-ban.

A kód ellenőrzésére és tesztek automatizált futtatására a *GitHub Actions*³-t használtam. A *workflow* definiálásakor a *GitHub Actions* által nyújtott *Go* kiinduló *workflow*-ból indultam ki, amely a fordítás (*build*) folyamatot már alapból tartalmazza.

A *workflow* teszt fázissal történő kiegészítéséhez a *GitHub Actions* service konténereit [33] használtam, amelyek segítségével olyan külső szolgáltatásokat tudtam

³ A *GitHub Actions* egy continuous integration és continuous delivery (CI/CD) platform, amely lehetővé teszi a build, teszt és telepítés folyamatok automatizálását. Segítségével workflow-kat készíthetünk, amelyek végrehajtják a build és teszt folyamatokat minden commit-ra vagy pull request-re.

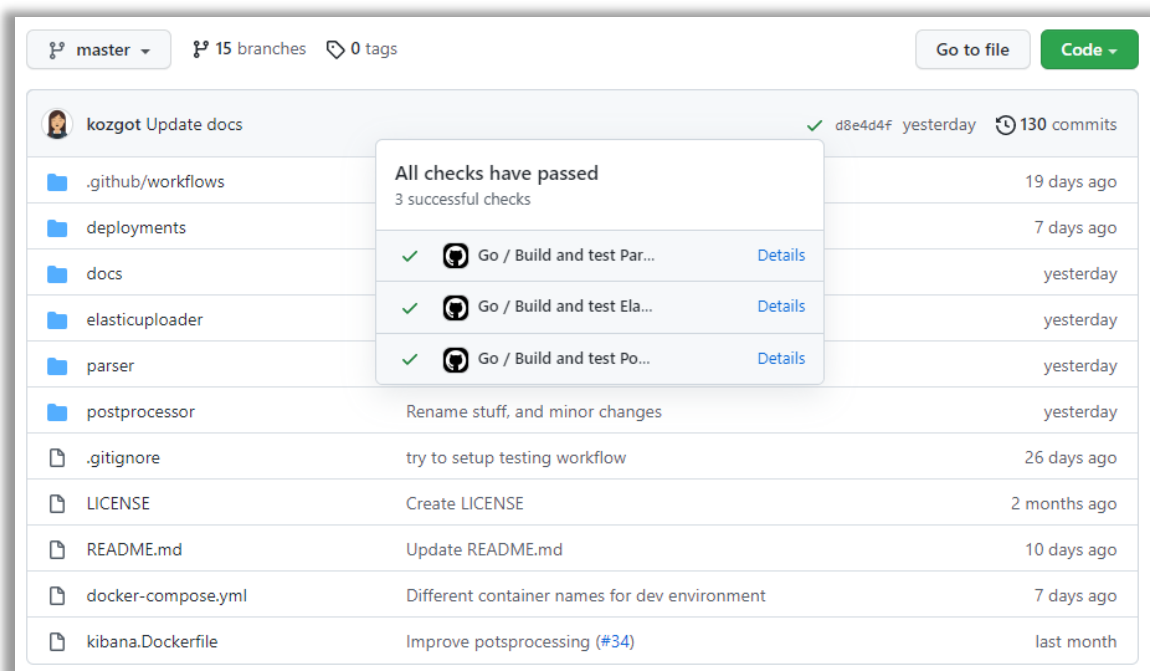
definiálni, amelyekre a tesztek futtatásához szükség van. Ezek az én esetemben a *RabbitMQ* és az *Elasticsearch*.

Minden mikroszolgáltatáshoz külön *job*-okat definiáltam, mivel a service konténereket *job*-onként tudjuk megadni, és így minden mikroszolgáltatás esetében csak a feltétlen szükséges service konténereket kell elindítani, például az *Elasticsearch*-re csak az *elasticuploader* tesztjeihez van szükség.

A *workflow*-t úgy állítottam be, hogy a *master* ágon minden *push* műveletre, és minden *master* ágra mutató *pull request*-re is automatikusan végrehajtsódjon.

Az így előálló *workflow* konfigurációja a projekt gyökeréből kiindulva a *.github/workflows/go.yml* fájlban tekinthető meg.

Az alábbi képen a három *job* sikeres lefutása látható a *master* ágon:



4.4.3.1. ábra: GitHub workflow

5 Konklúzió és továbbfejlesztési lehetőségek

A diplomatervezés projektem elkészítése során számos új tapasztalattal gazdagodtam, és több területen is mélyíthettem ismereteimet új technológiák és tervezési elvek megismerésével. A kezdetben specifikált rendszer funkciókat sikerült elkészítenem és a fejlesztés során felmerülő problémák megoldása által is új tudást szereztem.

A munkám során nagyobb tapasztalatra tehettem szert a mikroszolgáltatások architektúrában történő fejlesztéssel és az üzenetalapú kommunikációval kapcsolatban, emellett megismerhettem egy számomra teljesen új programozási nyelvet, a *Go*-t, és megtanulhattam használni annak kényelmes és hasznos elemeit, például a *goroutine*-okat és a *channel*-eket.

A diplomatervezés projektem megkezdése előtt volt már tapasztalatom a *Docker* és a *Docker Compose* használatával, viszont mélyebb tudást ezelőtt még nem sajátítottam el velük kapcsolatban. A projekt fejlesztése során azonban sok új ismeretet szereztem a konténerizációról és a *VSCoDe Remote-Containers* használatáról, illetve ezeknek a lokális fejlesztéssel szembeni előnyeit is elsőkézből tapasztalhattam meg. A fejlesztői környezet kialakítása először különösen nagy kihívást jelentett számomra, hiszen nem csak a konténerben történő fejlesztés volt számomra újdonság, hanem egy teljesen új programozási nyelvet és az azzal kapcsolatos szokásokat és ökoszisztémát is meg kellett ismernem a probléma megoldásához.

Az eddig említetteken kívül rengeteg új tudást szereztem még a reguláris kifejezések használatáról (a *parser* mikroszolgáltatás implementációjakor), a tesztelésről, azon belül is az integrációs tesztek megírásáról és azok automatizált futtatásáról *service* konténerek használatával, valamint az *Azure Blob storage* beüzemeléséről és használatáról.

Annak ellenére, hogy a rendszert sikerült a tervezett funkciókkal megvalósítanom, a projekt még számos továbbfejlesztési lehetőséget rejt magában. Először megemlítendő továbbfejlesztési lehetőség a *parser* mikroszolgáltatás kiterjesztése más formátumú naplófájlok felolvasására, illetve ezután az utófeldolgozást végző *postprocessor* mikroszolgáltatás felokosítása, hogy a lehető legtöbb hasznos információt tudjuk kinyerni a rendelkezésre álló adatokból. Emellett a rendszer tesztelésén is lehetne még fejleszteni, mivel idő hiányában csak a leglényegesebb teszteseteket készítettem el.

Irodalomjegyzék

- [1] Sagemcom, „Need specification for DC-SMC communications analysis tool - V1 - 27012020,” Sagemcom, 2020.
- [2] G.-P. Allinace, „G3-PLC User Guidelines, Introduction of G3-PLC for non-experts (version 1.1, 05/2020),” G3-PLC Allinace, 2020.
- [3] A Go hivatalos oldala, „Why Go?,” Google, 27 08 2020. [Online]. Available: <https://go.dev/solutions/google/>. [Hozzáférés dátuma: 06 12 2021].
- [4] Geeksforgeeks, „Concurrency in golang,” Geeksforgeeks, 20 11 2019. [Online]. Available: <https://www.geeksforgeeks.org/goroutines-concurrency-in-golang/>. [Hozzáférés dátuma: 06 12 2021].
- [5] Elasticsearch hivatalos oldala, „What is Elasticsearch?,” Elasticsearch, [Online]. Available: <https://www.elastic.co/what-is/elasticsearch>. [Hozzáférés dátuma: 06 12 2021].
- [6] Alex Brasetvik, „Elasticsearch as a NoSQL Database,” Elasticsearch, 15 09 2013. [Online]. Available: <https://www.elastic.co/blog/found-elasticsearch-as-nosql>. [Hozzáférés dátuma: 06 12 2021].
- [7] Elasticsearch, „The official Go client for Elasticsearch,” Elasticsearch, [Online]. Available: <https://github.com/elastic/go-elasticsearch>. [Hozzáférés dátuma: 06 12 2021].
- [8] Elasticsearch, „What is Kibana?,” Elasticsearch, [Online]. Available: <https://www.elastic.co/what-is/kibana>. [Hozzáférés dátuma: 06 12 2021].
- [9] Walter Rafelsberger (walterra), „kibana-milestones-vis GitHub repository,” [Online]. Available: <https://github.com/walterra/kibana-milestones-vis>. [Hozzáférés dátuma: 06 12 2021].

- [10] RabbitMQ, „A RabbitMQ hivatalos oldala,” [Online]. Available: <https://www.rabbitmq.com/>. [Hozzáférés dátuma: 06 12 2021].
- [11] RabbitMQ, „Which protocols does RabbitMQ support?,” [Online]. Available: <https://www.rabbitmq.com/protocols.html>. [Hozzáférés dátuma: 06 12 2021].
- [12] RabbitMQ, „Clients Libraries and Developer Tools,” [Online]. Available: <https://www.rabbitmq.com/devtools.html>. [Hozzáférés dátuma: 06 12 2021].
- [13] Akshay Kamath B és Chaitra B H, „A Comparison between RabbitMQ and REST ful API for Communication,” *International Research Journal of Engineering and Technology (IRJET)*, %1. kötet7, %1. szám5, p. 4665, 2020.
- [14] RabbitMQ, „RabbitMQ tutorial 4: Routing,” [Online]. Available: <https://www.rabbitmq.com/tutorials/tutorial-four-go.html>. [Hozzáférés dátuma: 06 12 2021].
- [15] RabbitMQ, „RabbitMQ tutorial 5: Topics,” [Online]. Available: <https://www.rabbitmq.com/tutorials/tutorial-five-go.html>. [Hozzáférés dátuma: 06 12 2021].
- [16] RabbitMQ, „RabbitMQ tutorial 3: Publish/Subscribe,” [Online]. Available: <https://www.rabbitmq.com/tutorials/tutorial-three-go.html>. [Hozzáférés dátuma: 06 12 2021].
- [17] Go hivatalos weboldal, „Editor plugins and IDEs,” [Online]. Available: <https://go.dev/doc/editors>. [Hozzáférés dátuma: 07 12 2021].
- [18] Microsoft, „Visual Studio Code dokumentáció,” [Online]. Available: <https://code.visualstudio.com/docs/remote/containers>. [Hozzáférés dátuma: 07 12 2021].
- [19] VSCode, „Remote-Containers tutorial - How it works?,” Microsoft, [Online]. Available: https://code.visualstudio.com/docs/remote/containers-tutorial#_how-it-works. [Hozzáférés dátuma: 07 12 2021].

- [20] VSCode, „Connect to multiple containers,” Microsoft, [Online]. Available: <https://code.visualstudio.com/remote/advancedcontainers/connect-multiple-containers>. [Hozzáférés dátuma: 07 12 2021].
- [21] Go dokumentáció, „Remote import paths,” Google, [Online]. Available: https://pkg.go.dev/cmd/go#hdr-Remote_import_paths. [Hozzáférés dátuma: 07 12 2021].
- [22] GoJP, „A Go Report Card GitHub oldala,” GoJP, [Online]. Available: <https://github.com/gojp/goreportcard>. [Hozzáférés dátuma: 07 12 2021].
- [23] „A naplófeldolgozó rendszer GitHub oldala,” [Online]. Available: <https://github.com/kozgot/go-log-processing>. [Hozzáférés dátuma: 07 12 2021].
- [24] „A golang-standards/project-layout GitHub oldala,” golang-standards, [Online]. Available: <https://github.com/golang-standards/project-layout>. [Hozzáférés dátuma: 07 12 2021].
- [25] Aviv Carmi, „OK Let’s Go: Three Approaches to Structuring Go Code,” PerimeterX, 15 05 2019. [Online]. Available: <https://www.perimeterx.com/tech-blog/2019/ok-lets-go/>. [Hozzáférés dátuma: 07 12 2021].
- [26] „Introduction to Azure Blob storage,” Microsoft, 30 03 2021. [Online]. Available: <https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction>. [Hozzáférés dátuma: 07 12 2021].
- [27] M. Azure, „Azure Storage Blob SDK for Go,” Microsoft, [Online]. Available: <https://github.com/azure/azure-storage-blob-go/>. [Hozzáférés dátuma: 07 12 2021].

- [28] Naveen Ramanathan, „Structs Instead of Classes - OOP in Go,” 03 05 2020. [Online]. Available: <https://golangbot.com/structs-instead-of-classes/>. [Hozzáférés dátuma: 07 12 2021].
- [29] Rob Figueiredo (robfig), „A Cron package GitHub oldala,” [Online]. Available: <https://github.com/robfig/cron>. [Hozzáférés dátuma: 07 12 2021].
- [30] „A Cron package dokumentációja,” [Online]. Available: <https://pkg.go.dev/github.com/robfig/cron>. [Hozzáférés dátuma: 07 12 2021].
- [31] Tobi Balogun, „How To Write Unit Tests in Go,” DigitalOcean, 26 11 2020. [Online]. Available: <https://www.digitalocean.com/community/tutorials/how-to-write-unit-tests-in-go-using-go-test-and-the-testing-package>. [Hozzáférés dátuma: 07 12 2021].
- [32] „Az amqp package dokumentációja,” [Online]. Available: <https://pkg.go.dev/github.com/streadway/amqp#Acknowledger>. [Hozzáférés dátuma: 07 12 2021].
- [33] „About service containers,” GitHub Docs, [Online]. Available: <https://docs.github.com/en/actions/using-containerized-services/about-service-containers>. [Hozzáférés dátuma: 07 12 2021].