FELADATKIÍRÁS

Az elektronikusan beadott változatban ez az oldal törlendő. A nyomtatott változatban ennek az oldalnak a helyére a diplomaterv portálról letöltött, jóváhagyott feladatkiírást kell befűzni.



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 6](#_Toc89088206)

[Abstract 7](#_Toc89088207)

[1 Bevezetés 8](#_Toc89088208)

[2 A feladatkiírás pontosítása és részletes elemzése 9](#_Toc89088209)

[2.1 A naplózó rendszer ismertetése 9](#_Toc89088210)

[2.2 Az elkészítendő rendszer ismertetése 10](#_Toc89088211)

[3 A rendszer megtervezése 11](#_Toc89088212)

[3.1 A tervezési szempontok bemutatása 11](#_Toc89088213)

[3.1.1 Skálázhatóság 11](#_Toc89088214)

[3.1.2 Könnyű kiterjeszthetőség 12](#_Toc89088215)

[3.1.3 Megbízhatóság és hibatűrés 12](#_Toc89088216)

[3.1.4 Modularitás 12](#_Toc89088217)

[3.1.5 Egyéb szempontok 13](#_Toc89088218)

[3.2 A választott architektúra és technológiai megoldások 13](#_Toc89088219)

[3.2.1 Mikroszolgáltatások architektúra 13](#_Toc89088220)

[3.2.2 A választott technológiák bemutatása 15](#_Toc89088221)

[4 Implementáció 20](#_Toc89088222)

[4.1 A fejlesztés menete 20](#_Toc89088223)

[4.1.1 Docker és Visual Studio Code 20](#_Toc89088224)

[4.1.2 Verziókezelés 23](#_Toc89088225)

[4.1.3 Kódminőség ellenőrzése 24](#_Toc89088226)

[4.2 A szolgáltatások megvalósítása 26](#_Toc89088227)

[4.2.1 Parser mikroszolgáltatás 26](#_Toc89088228)

[4.2.2 Postprocessor mikroszolgáltatás 36](#_Toc89088229)

[4.2.3 ElasticUploader mikroszolgáltatás 43](#_Toc89088230)

[4.3 Adattárolás és vizualizáció 44](#_Toc89088231)

[4.3.1 Adattárolás 44](#_Toc89088232)

[4.3.2 Vizualizáció 44](#_Toc89088233)

[4.4 Tesztelés 45](#_Toc89088234)

[4.4.1 Integrációs tesztek 45](#_Toc89088235)

[4.4.2 Unit tesztek 46](#_Toc89088236)

[4.4.3 CI automatizált teszt futtatás 47](#_Toc89088237)

[5 Konklúzió és továbbfejlesztési lehetőségek 48](#_Toc89088238)

[6 Köszönetnyilvánítás 49](#_Toc89088239)

[Irodalomjegyzék 50](#_Toc89088240)

[Függelék 51](#_Toc89088241)

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. 11. 29.

...…………………………………………….

Kozák Ágota Boglárka

Összefoglaló

Ide jön a ½-1 oldalas magyar nyelvű összefoglaló, melynek szövege a Diplomaterv Portálra külön is feltöltésre kerül.

Abstract

Ide jön a ½-1 oldalas angol nyelvű összefoglaló, amelynek szövege a Diplomaterv Portálra külön is feltöltésre kerül.

# 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.

Az 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.

# 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.

## 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:

* 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)

A képen szöveg látható

Automatikusan generált leírás

2.1. ábra: Az okos mérők hálózatának egyszerűsített váza

Az *SMC*-k a *G3-PLC protokoll* segítségével kommunikálnak a *DC*-kkel. A *G3-PLC* megkönnyíti a nagy sebességű, nagyon megbízható, nagy hatótávolságúkommunikációt a meglévő villamosvezeték-hálózaton.

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

## 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.

# 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.

## 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.

### 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 feladatomként 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.

### 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.

### 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.

### 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 kódbázis megértését így megkönnyítve a fejlesztést és karbantartást.

### 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 teszteket.

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.

## 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.

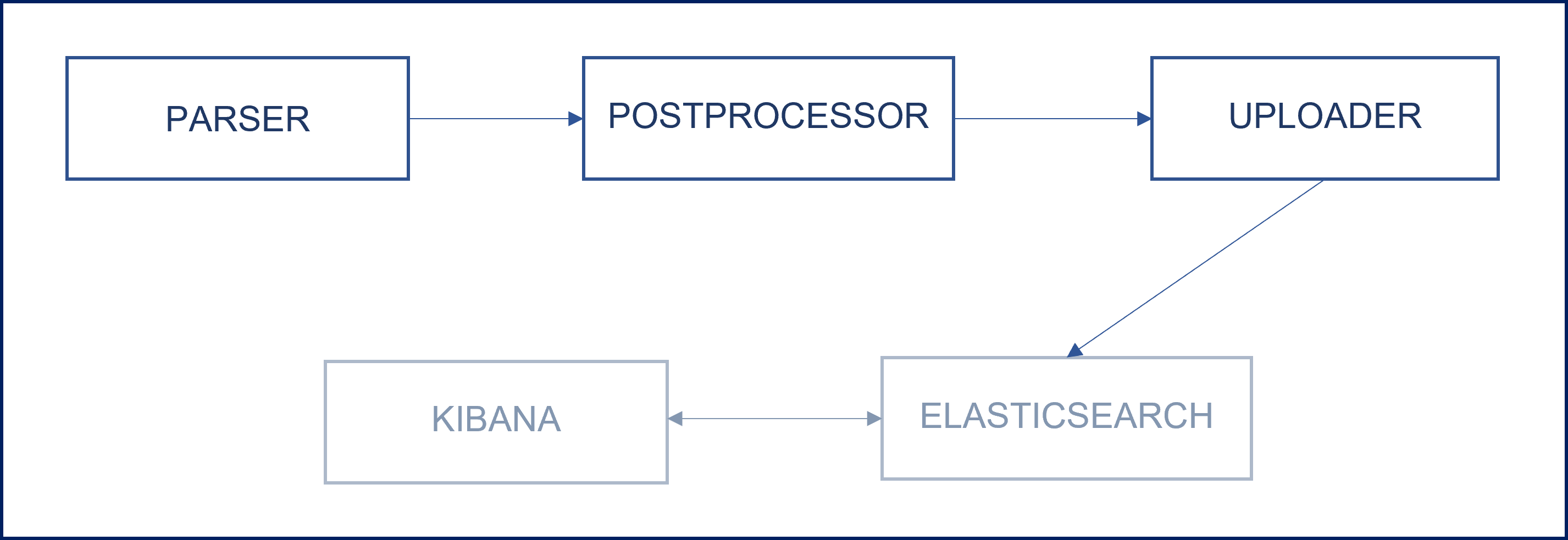
### Mikroszolgáltatások architektúra

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 csővezetéket 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 köztük 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 *Uploader* 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.

### 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.

#### 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. Szintaxisa hasonlít a *C* nyelvhez, viszont ez a programozási nyelv olyan eszközöket is tartalmaz, amelyek lehetővé teszik a biztonságos memóriakezelést, az objektumok kezelését, a szemétgyűjtést 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*). 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 *Uploader*-t is *Go* nyelven implementáltam.

#### 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. Az *Elasticsearch* „sémarugalmas” 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. 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.

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. Az *Elastic Stack* diagramkészítő eszközeként is tekinthetünk rá. Főként nagy mennyiségű adat elemzésére használják, rengeteg vizualizáció készítésére alkalmas, például vonaldiagram, oszlopdiagram, 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. A Kibana használata mellett szól továbbá a könnyű használhatóság és érthetőség.

#### Kommunikáció

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

Több tízezer felhasználóval a *RabbitMQ* az egyik legnépszerűbb nyílt forráskódú üzenetközvetítő. 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. 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.

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 túlélik. 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.
* ***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.
* ***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.

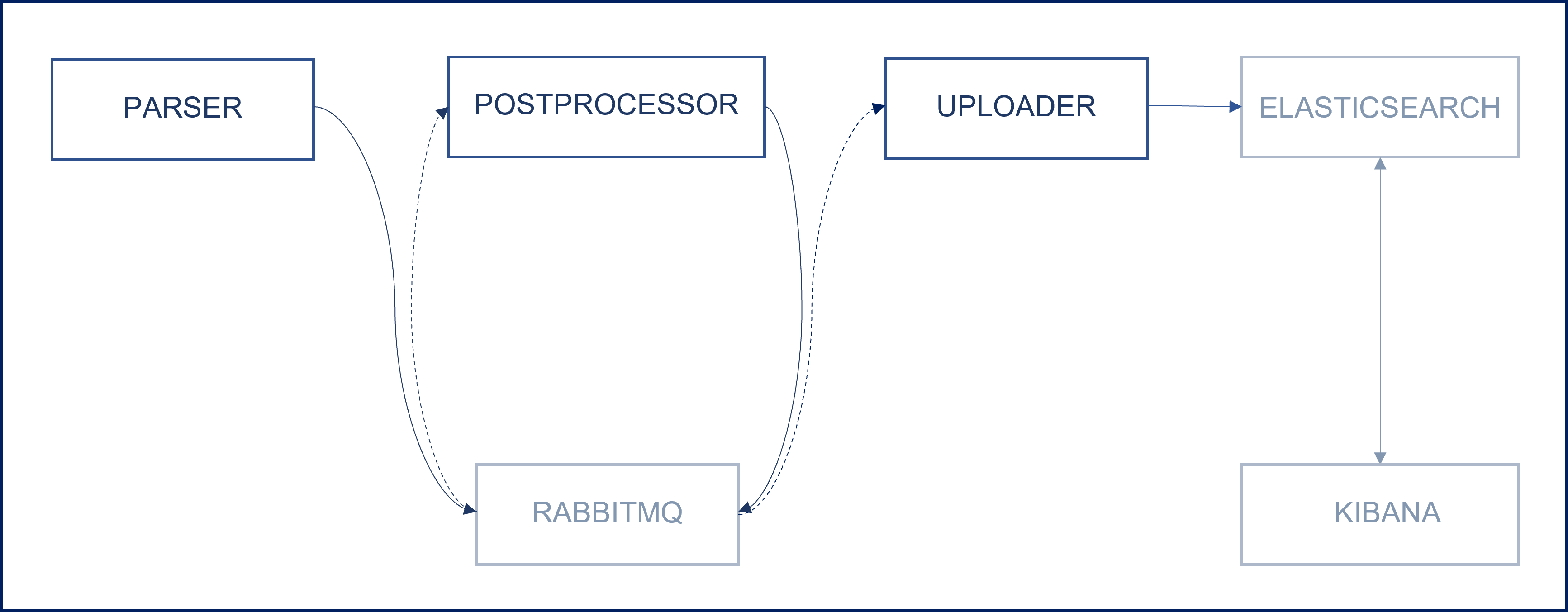
A napló feldolgozó rendszerben minden esetben ***direct*** típusú exchange-t használtam, mivel az én esetemben a cél, hogy minden üzenetet pontosan egy szolgáltatás kapjon meg, viszont, ha a jövőben esetleg be kell vezetni különböző jellegű feldolgozó szolgáltatásokat is, akkor a *topic* exchange-nek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

#### Docker

Minden szolgáltatás *Docker* konténerben fut, amelyek összehangolására *Docker Compose*-t használtam. Ezzel a megoldással az egész feldolgozó rendszert egyetlen *docker-compose.yml* fájllal 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.

#### 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

# 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.

## 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.

### 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:

* ***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. 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 gyökér 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 tartalmazott konfigurációnak megfelelően lesz beállítva és telepítve. Ezután a lokális *Visual Studio Code* kapcsolódik a *Visual Studio Code* szerverhez, amely a létrejött fejlesztői konténerben fut.

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. Az így létrejött mappaszerkezet látható a következő ábrán.

A képen szöveg látható

Automatikusan generált leírás

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 lenyomás után kiválasztjuk a *„Remote-Containers: Open Folder in Container…”* opciót, majd kiválasztjuk az egyik szolgáltatás mappáját, például a *Parser* szolgáltatásét. 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 konténereken 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 többiről a konténerizált környezet és *Docker Compose* gondoskodik.

### Verziókezelés

A verziókezelés, vagy *version control*, a szoftverkód változásainak nyomon követésének és kezelésének gyakorlata. A szoftverfejlesztés magában foglalja a programkód 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.

### 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 átállással, 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.

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

A képen szöveg látható

Automatikusan generált leírás

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

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

A képen szöveg látható

Automatikusan generált leírás

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.

## 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.

### 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.

#### 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*** 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 figyelnünk, 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.

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 az a *parser* szolgáltatás esetében csak a *models* package került, amely az adatmodellek definícióit tartalmazza, mivel ezekre épít a *postprocessor* szolgáltatás.   
  A szolgáltatásban minden más kód *internal*.
* ***tests***: Itt találhatók a szolgáltatáshoz tartozó teszt fájlok, illetve egyéb teszteléshez használt segéd funkciók és erőforrás fájlok. Erről részletesebben a 4.3. Tesztelés című fejezetben fogok írni.

#### 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 bárhonnan 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. Használatához széles körben érhetők el kliens könyvtárak például .*NET*-hez, *Python*-hoz és *Go*-hoz is, amely a *github.com/azure/azure-storage-blob-go* címen érhető el.

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. 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 változókban vannak definiálva.

#### 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 logika 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 szolgáltatás használatával.

Ezeken 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.

#### A feldolgozott napló sorok és az adatmodellek

Ebben az alfejezetben bemutatom, hogyan épülnek fel az adatmodellek, amelyek 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 mező 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

}

Ez 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 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

DCMessage \*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* típus, 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]](#footnote-1)* konfigurációs adatokat tartalmazzák.
* *SmcAddress*: Az ilyen típusú bejegyzések az adatbázisból érkező SMCcím 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

}

#### 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. 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 *sendDataToPostprocessor* 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 változókban vannak definiálva. A közzétételhez perzisztens szállítási módot használ, annak érdekében, hogy a küldött üzenetek túléljék az esetleges RabbitMQ szerver újraindításokat.

func (producer \*AmqpProducer) sendDataToPostprocessor(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.

### 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. Az 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.

#### 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 találhatók 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.

#### 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.ConsumtionValue)

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 változókban tárolom.

#### 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, még egy *ConnectionAttempt* típusú sorból a hozzá tartozó *URL* olvasható ki.
* ***smcUIDsByURL***: Ez egy *map* adatstruktúra, ahol a kulcsok az *SMC*-k *URL*-ek, az értékek pedig az *URL*-ekhez tartozó *SMC* egyedi azonosítók. A *ConnectionAttempt* típusú sorokból 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]](#footnote-2)* adatokat tartalmazza egy adott időintervallumban. Ezekből kiolvasható egy *pod* 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 élőállítanak 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 *InfoEntryProcessor* 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 *ConsumtionValue* 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 *InfoEntryProcessor*, *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.

#### 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, naplófájlonként egy-egy *goroutine* indításával olvassa fel, így a bejegyzések konkurens módon továbbítódnak a *RabbitMQ* szerver felé. 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.

### ElasticUploader mikroszolgáltatás

#### A szolgáltatás felépítése

#### A feldolgozott adatok fogyasztása

#### A feltöltendő adatok pufferelt feltöltése

#### Időzített index létrehozás

## Adattárolás és vizualizáció

kb. 6 oldal

### Adattárolás

### Vizualizáció

## Tesztelés

### Integrációs tesztek

### Unit tesztek

### CI automatizált teszt futtatás

# Konklúzió és továbbfejlesztési lehetőségek

kb. 2-3 oldal

# Köszönetnyilvánítás

maximum 1 oldal, de ezt inkább nem is számítom bele a dolgozat hosszába

Irodalomjegyzék

1. Levendovszky, J., Jereb, L., Elek, Zs., Vesztergombi, Gy.: Adaptive statistical algorithms in network reliability analysis, Performance Evaluation - Elsevier, Vol. 48, 2002, pp. 225-236
2. National Istruments: LabVIEW grafikus fejlesztői környezet leírása, <http://www.ni.com/> (2010. nov.)
3. Fowler, M.: UML Distilled, 3rd edition, ISBN 0-321-19368-7, Addison-Wesley, 2004
4. Wikipedia: Evaluation strategy, <http://en.wikipedia.org/wiki/Evaluation_strategy> (revision 18:11, 31 July 2012)

Függelék

1. Egy pod a következtetésem szerint egy elektromos mérőt jelent, viszont ezzel kapcsolatban nem állt rendelkezésemre dokumentáció vagy magyarázat, mindössze a naplófájlokban lévő adatokból következtettem erre. [↑](#footnote-ref-1)
2. Egy index a *DC* által az *SMC*-ktől periodikusan gyűjtött adatokat jeleni. [↑](#footnote-ref-2)