

# **DIPLOMAMUNKA**

Komjáthy Benjamin

Debrecen

2022

**Debreceni Egyetem**

**Informatikai Kar**

**Webalkalmazás automatizált üzembehelyezése**

Témavezető:

Dr. Vágner Anikó Szilvia

Egyetemi adjunktus

Készítette:

Komjáthy Benjamin

Gazdaságinformatikus MSc

Debrecen

2022



# 1 TARTALOMJEGYZÉK

2	BEVEZETÉS.....	5
3	ÁLTALÁNOS ÁTTEKINTÉS.....	7
3.1	DevOps.....	7
3.2	CI/CD.....	11
4	DEVELOPMENT.....	14
4.1	A kód.....	14
4.1.1	Backend.....	14
4.1.2	Frontend.....	17
5	OPERATIONS.....	20
5.1	Infrastruktúra.....	20
5.1.1	Terraform.....	22
5.1.2	Ansible.....	24
5.1.3	Docker és Konténerizáció.....	26
5.1.4	Traefik.....	27
5.1.5	Portainer.....	28
5.1.6	Monitoring.....	28
5.1.7	Backup.....	36
5.2	Az alkalmazás üzemeltetése.....	37
5.2.1	Gitlab.....	37
5.2.2	Gitlab runner.....	39
5.2.3	Gitlab CI/CD.....	39
5.2.4	Database Backup&Restore.....	41
5.2.5	Monitoring.....	42
6	PROOF OF CONCEPT.....	43
7	ÖSSZEFOGLALÁS.....	46
8	IRODALOMJEGYZÉK.....	47
9	INTERNETES FORRÁSOK.....	48
10	MELLÉKLETEK.....	49

## 2 BEVEZETÉS

Diplomamunkám témaválasztása egy webalkalmazás automatizált üzembehelyezésére esett. Bővebben kifejtve, hogy ez mit is takar: egy általános webalkalmazás példáján keresztül mutatom be, hogy üzemeltetői szempontból hogyan lesz a megírt kódból teljes automatizáció útján development és production környezet (vagy esetleg igény szerint még ezek mellett demo, és production-like környezet), azaz egy teljes DevOps folyamat modellezése.

Azért ezt a témát választottam, mert a DevOps, konténerizáció, CI/CD folyamatok, felhő-infrastruktúra mind viszonylag új keletű, és még annyira nem elterjedt módszertanok, technológiák, viszont szerintem mérhetetlenül érdekes, izgalmas fel nem fedezett területek ezek. Annak a cégnek a fő profilja is ez, ahol az MSc mellett dolgozok, így valamennyi tapasztalatom és belelátásom már van a témában. Az üzemeltetés mellett, mivel kódolni is szeretek, így szerepet játszott benne az is, hogy valamilyen saját alkalmazás példáján keresztül vezessem be a kedves olvasót ebbe a világba.

A témaválasztás azért releváns, mert már most is hatalmas igény van a cégeknek a DevOps-ra és a különböző felhő technológiákra, automatizációkra, és a jövőben is egyre inkább fontos szerepet fog játszani az életükben. Mindemellet hasznos lenne a cégek számára is, ha a fejlesztőik is jobban bele látnának, hogy üzemeltetői oldalon is milyen precíz és modern folyamatok mennek végbe azért, hogy az általuk megírt kód, gyorsan és megfelelő módon kikerüljön egy éles környezetbe, és a vezetőség is jobban belelátna, hogy miért is ad ki pénzt, miért érdemes költeni a DevOps-ra is.

Tartalmilag a diplomamunkám felépítése a következő lesz. Először általánosan bemutatom, hogy mi az a DevOps, mi az a CI/CD, majd magát az elkészült alkalmazást mutatom be. Ezután megnézzük az infrastruktúrát, hogyan fog felépülni, milyen komponensekből fog állni, mik a szerepük, hogyan jönnek létre ezek a virtuális gépek automatikusan, hogy készülnek elő arra, hogy ellássák a feladatukat. Magukat a különböző szolgáltatásokat is bemutatom, amik elengedhetetlen kellékei annak, hogy minden a megfelelően működjön, lesz itt szó git-ről, konténerizációról, monitorozásról, backup és restore-ról. Végezetül amikor minden elkészült, lesz egy POC (proof of concept) arról, hogy hogyan is nézne ez ki egy cég életében, amikor egy adott feature tegyük fel elkészült, development környezetben már megnézték, tesztelték, és hogyan kerül ez ki két kattintással production környezetre.

Szeretném még kiemelni, hogy az 5. fejezetben bemutatott infrastruktúra egy része újonnan létrehozott szervergépeket és szolgáltatásokat tartalmaz, viszont volt pár olyan elem, aminél a már meglévő, beüzemelt szerver/szolgáltatás lett felhasználva, ugyanis a cég, ahol dolgozok, név szerint

a SysCops Technologies Kft.(későbbiekben SysCops), megengedte, hogy a diplomamunka céljára felhasználjam a már meglévő erőforrásokat is és így csak erre a célra külön létrehozni ezeket nem lett volna értelme. Ezt abban a fejezetben majd bővebben taglalom.

Ugyanezzel a céggel a 2022 tavaszi Informatikai Szakmai Napok keretein belül két előadást is tartottunk, hasonló témában, mint a diplomamunkám, aki azokon részt vett, legalább egy minimális betekintést kaphatott a következőben leírtakban.

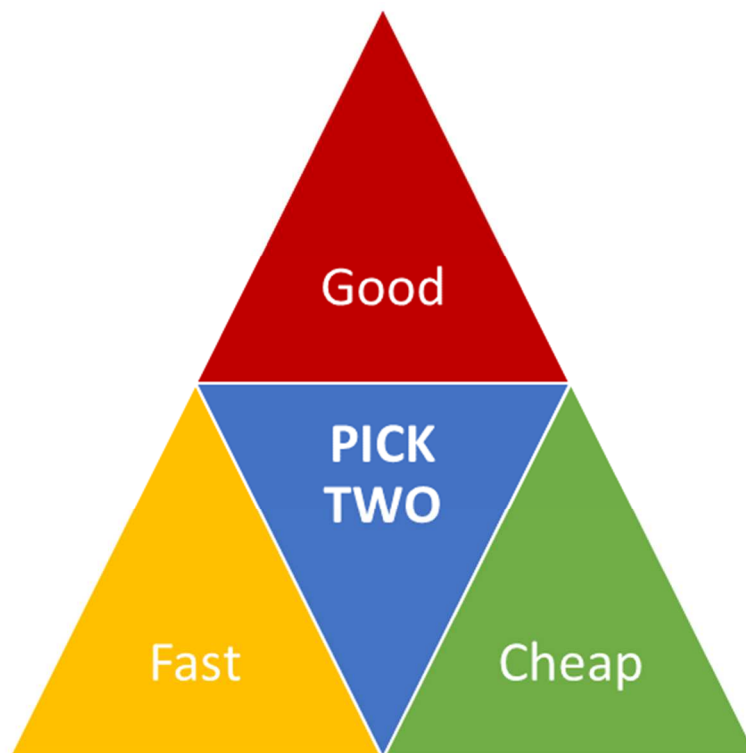
### 3 ÁLTALÁNOS ÁTTEKINTÉS

Ebben a fejezetben, ahogy a cím is mutatja, általánosan nézzük meg, hogy mi az a DevOps, és a Continuous Integration/Continuous Delivery, azaz a CI/CD.

#### 3.1 DevOps

Erről valójában oldalakat lehetne írni, és órákon át beszélgetni, hogy kinek mit is jelent a DevOps. Van, aki egy terminológiát lát benne, van, aki egy módszertant, van, akinek ez egy életérzés, vagy az önkifejezés egy sajátos módja. Néha a vállalat vezetőségének csak fölöslegesen túlázott üzemeltetést jelent, mivel nem látnak bele, és nem értik, hogy miért is lesz a DevOps mérhetetlenül fontos, mert még az a kép van bennük a régi időkből, hogy csak néhány rendszergazda bütyköli az onpremise szervereiket, állítgatja a tűzfalat, telepítget néhány dolgot, meg kávézik egész nap.

Nyugaton már ez teljesen elfogadott és szükséges velejárója a szoftverfejlesztésnek, de Magyarországra, főleg a KKV szektorra ez nem igazán jellemző, még. Nagyon sok esetben a SysCops-nál is azt a projekt menedzsmentből át vett elvet („iron triangle”) alkalmazzuk, kicsit magunkra szabva, hogy a gyors, olcsó, jó hármashból az adott ügyfél választhat kettőt.



3.1. ábra Átalakított Iron Triangle

Forrás: <https://medium.com/>

Természetesen törekszünk arra, hogy közelítsünk ahhoz, hogy mind a három megvalósuljon, csak a való élet nem azt mutatja, hogy ez lehetséges. A másik hasonló pénzübeli probléma, még az szokott lenni, hogy minden cégnek HA (High Availability) kell, abból is persze a 99.999%-os („five nines”), meg ügyelet/készenlét, meg ha valamilyen munkaidőn kívüli időben történik rendkívüli incidens, akkor be kell beavatkozni, legyünk ott, segítsünk be, viszont fizetni ezekért alapvetően nem annyira szeretnének.

A piac helyzetén való rövid elmélkedés után tekintsünk meg néhány szakirodalmat a témában.

*„Devops is a way of thinking and a way of working. It is a framework for sharing stories and developing empathy, enabling people and teams to practice their crafts in effective and lasting ways. It is part of the cultural weave that shapes how we work and why. Many people think about devops as specific tools like Chef or Docker, but tools alone are not devops. What makes tools “devops” is the manner of their use, not fundamental characteristics of the tools themselves.” (Davis, Daniels 2016:26)*

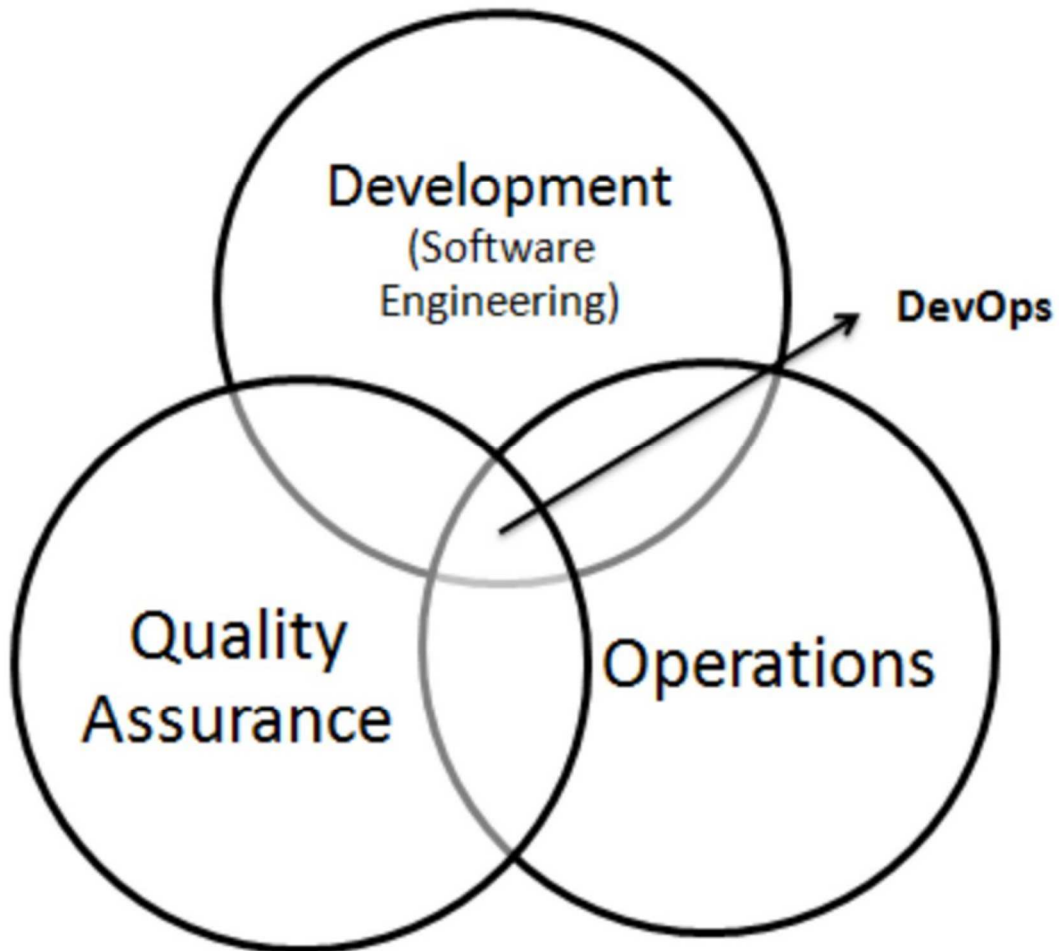
*„DevOps methods build on Agile software development and service management techniques by emphasizing close collaboration between the roles of software development and technical operations. Using high degrees of automation to free up the time of skilled professionals so that they can focus on value-adding activities, DevOps is able to shine a light on aspects such as operability, reliability, and maintainability of software products that can assist in the management of services. Cultural aspects that DevOps practitioners advocate can, and should, be extended across the value stream and all service value chain activities so that product and service teams are aligned with the same goals and use the same methods.” (ITIL Foundation 2019:40)*

Gyakran mondják, hogy a DevOps egyesíti a szoftverfejlesztési technikákat (Agile), a jó kormányzást és az értéktársítás holisztikus megközelítését (ITIL), valamint az értékteremtési módok megismerését és folyamatos javítását (Lean).

*„A new abbreviation or term called DevOps was developed with the blending of two significant trends. The first of these trends is called agile infrastructure or agile operations, which sprung from the application of the Agile and Lean approaches to projects. The second trend is a deeper understanding of the collaboration between the development and operations staff when creating or operating a service, through every stage of the process. DevOps also takes into account how essential operations have become for any business, as every industry is now customer-centric and service-oriented.” (Fleming 2019:15)*



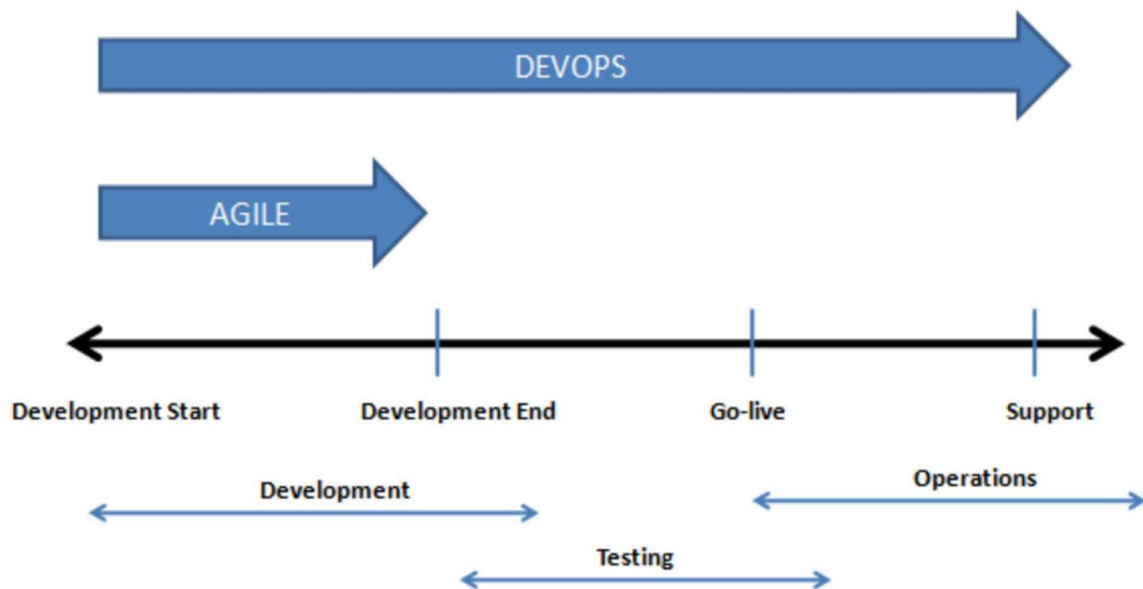
Ahogy a 3.2.-es ábra mutatja, a DevOps a fejlesztés, üzemeltetés és a minőségbiztosítás közös metszetében helyezkedik el.



3.2. ábra Szemléltető ábra a DevOps-ról I.

Forrás: Fleming 2019:17

*„There is also widespread misunderstanding about what DevOps actually is: A job title? A team? A methodology? A skill set? The influential DevOps writer John Willis has identified four key pillars of DevOps, which he calls culture, automation, measurement, and sharing (CAMS). Organizations practicing DevOps have a culture that embraces collaboration, rejects siloing knowledge between teams, and comes up with ways of measuring how they can be constantly improving. Another way to break it down is what Brian Dawson has called the DevOps trinity: people and culture, process and practice, and tools and technology.” (Domingus, Arundel 2022 33)*



3.3. ábra Szemléltető ábra a DevOps-ról II:

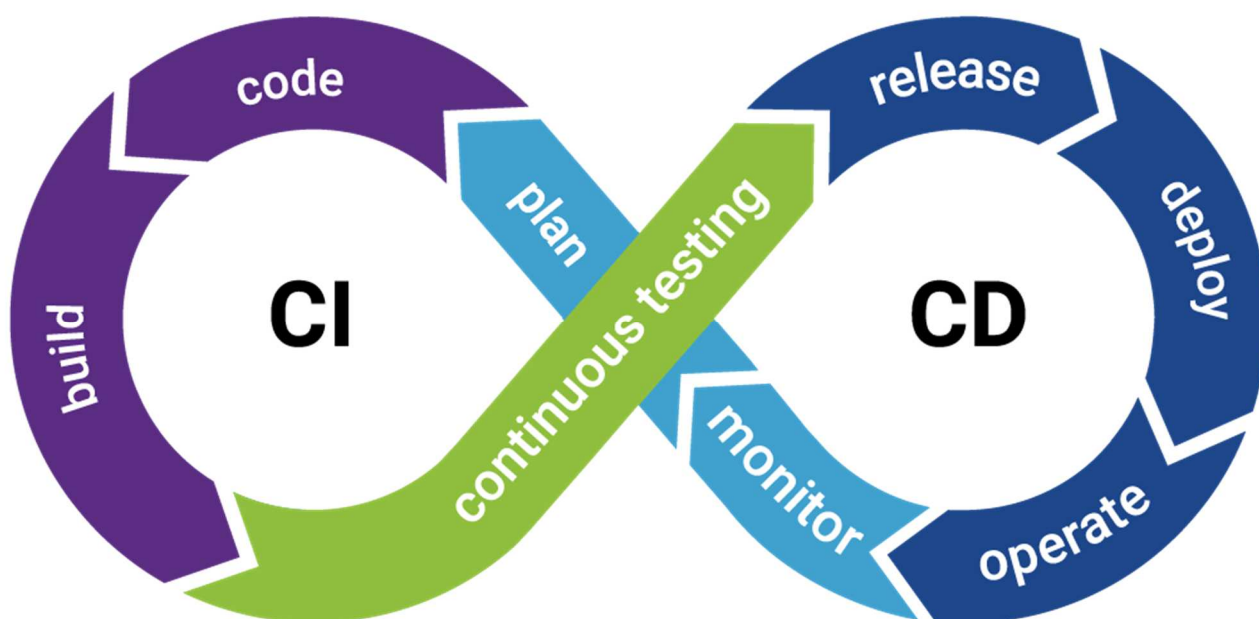
Forrás: Fleming 2019:19

Egy DevOps-osnak ismernie és értenie kell a kódot, akár bele is kell tudjon nyúlni abba (megtörtént eset). Értenie kell a teszteseteket és tudnia kell automatikusan futtatni azokat, valamint dockerizálni az alkalmazást. Ismernie kell a legújabb technológiákat az infrastruktúra felhúzására, tudnia kell automatizálni a kód deploymentet, monitorozni és „bombabiztossá” tenni az egész rendszert, illetve, ha gond van tisztában kell legyen azzal, hogy hol van baj, és hogy kell beavatkozni, szóval a teljes szoftveréletről végig kell tudnia kísérni a kódot, ahogy ez a 3.3.-as ábrán is látszik. A sok technológia közül is meg kell találni a legjobbat, legmegbízhatóbbat, legsokoldalúbbat. A cégek számára tudnia kell tanácsot adni, hogy melyik felhőszolgáltatónál a legérdekesebb felhúzni az infrastruktúrát, sőt akár abban is, hogy egyáltalán milyen infrastruktúra kell, ahhoz, hogy a szoftver, amit fejlesztenek, milyen környezetben fogja a legjobban érezni magát.

### 3.2 CI/CD

Ahogy azt már megtudhattuk, a betűszó kifejtve Continuous Integration/Continuous Delivery, vagy éppen Continuous Deployment.

Nagyon sokszor találkozunk azzal a megszólalással különböző besült demók után, hogy „de nekem lokálban lefutott”. Na ennek a kiküszöbölésére jött létre a 3.4.-es ábrán is látható végtelen ciklusunk. Alap feltétele ennek az egésznek a megfelelő verziókezelés (git), és különböző automatizált pipeline-ok futtatása.



3.4. ábra A CI/CD végtelen ciklusa

Forrás: <https://flagship.io/>

*„On the other hand, we have seen projects that spend at most a few minutes in a state where their application is not working with the latest changes. The difference is the use of continuous integration. Continuous integration requires that every time somebody commits any change, the entire application is built and a comprehensive set of automated tests is run against it. Crucially, if the build or test process fails, the development team stops whatever they are doing and fixes the problem immediately. The goal of continuous integration is that the software is in a working state all the time.” (Humble, Farley 2011:55)*

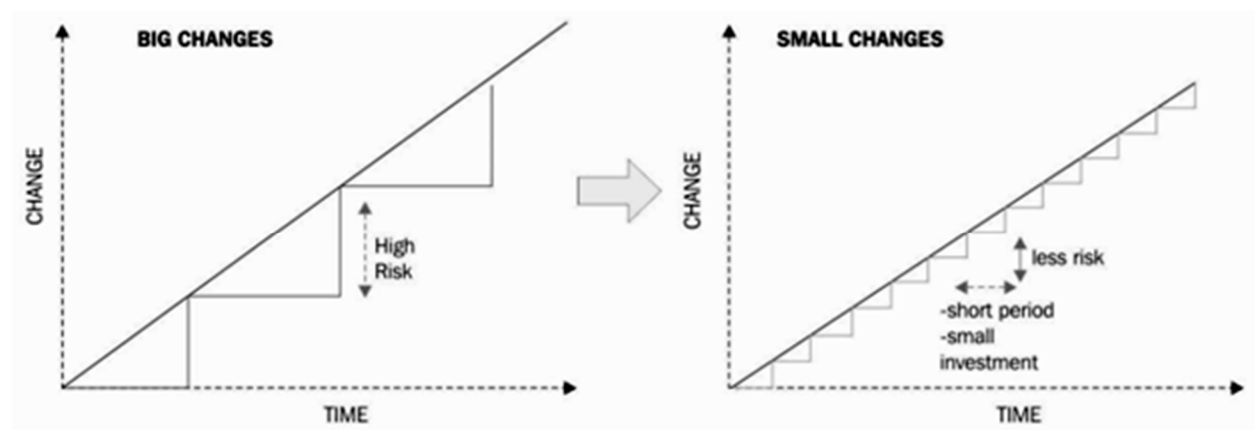
A kódot folyamatosan integráljuk, azaz naponta többször, de legalább egyszer pusholjuk a kódot a master branch-re, vagy a fejlesztésre kijelölt, például developer branch-re. És itt akkor lefutnak

mindenféle tesztesetek, és kikerül a fejlesztői környezetbe, ahol meg tudják nézni, hogy hogyan is működik, és a friss hibákat azonnal, vagy záros határidőn belül meg lehet javítani, mivel, ha még nem olyan rég írtuk azt a kódot, könnyebb javítani, mintha hónapok múlva jönne elő ugyan ez a hiba, mert nincsen a Continuous Integration beépítve a fejlesztés folyamatába.

*„Continuous deployment (CD) is the automatic deployment of successful builds to production. Like the test suite, deployment should also be managed centrally and automated. Developers should be able to deploy new versions by either pushing a button, or merging a merge request, or pushing a Git release tag” (Domingus, Arundel 2022:253)*

Nem arról szól, hogy csak valahol nyomunk egy gombot, és kikerül valami az éles környezetbe, hanem hogy hogyan tudjuk elérni, hogyha megnyomjuk azt a gombot, akkor biztosak legyünk abban, hogy ami kikerül, az tényleg megbízható, és működni is fog, vagy ha mégsem, akkor könnyen vissza tudunk állni egy korábbi verzióra. A célfüggvényünk, hogy minimalizáljuk az MTTR-t (mean time to recover)

Erősen párhuzamba vonható ez is a Lean alapelvekkel. Fontos a minőség mihamarabbi beépítése. Már az első pillanattól kezdve jó minőségű szoftvereket gyártunk, és hamar találjuk meg a hibákat, mivel minél később kerül feltárára, annál nehezebb, és időigényesebb kijavítani. Kis egységekben dolgozzunk, ne legyenek nagy feature release-ek, mert mivel nagyobb egy release, annál nagyobb a hibalehetőség is, és annál nehezebb rájönni, hogy konkrétan mi is okozza a hibát.



3.5. ábra A rizikó faktor csökkenése

Forrás: Srivastava 2021:22

Következő fontos alapelv, hogy minden, ami repetatív, azt automatizáljuk valamilyen rendelkezésre álló eszközzel, mivel ezeknek a kézzel futtatásába könnyű belefásulni, és könnyebb hibázni is. El kell fogadnunk azt is, hogy sosem vagyunk kész, mindig van hová tovább fejlődnünk és igazából az egész életünk egy folyamatos tanulás. Még itt a Continuous Delivery-nél, ha csinálunk valamit, azt is folyamatosan fejleszteni kell, jönnek be új eszközök, amik esetleg jobbak lehetnek számunkra az adott célra, ezeket kipróbálni, tesztelni kell, esetleg az egész folyamatot utána hozzá alakítani, ha jobban bevált, mint a korábbi megoldás. És azt is el kell tudni fogadnunk, hogyha valami félremegy éles környezetben, az nem a vezetőség hibája, nem az üzemeltetőké, nem a tesztelőké, nem a fejlesztőké, hanem egységesen mindenkié, mivel mindenkin végig ment a kód, és hozzájárult ahhoz, hogy az kikerüljön.

Összefoglalva, hogy milyen hasznunk is származik ebből az egészből. Csökken a kockázat, nő az egész folyamatnak és a termékünknek a minősége, csökkennek a költségek, akár a fejlesztéssel járó költségek, akár a leállás miatti kiesés költsége. És az egész csapat stresszmentesebben, jobb közérzettel tud dolgozni, ugyanis annak az esélye is csökken, hogy nem megfelelő kód kerüljön ki az éles környezetbe.

## 4 DEVELOPMENT

Ebben a fejezetben a DevOps-nak a Development részét vesszük szemügyre, azaz fejlesztői szempontból megnézzük, hogy hogyan is épül fel az általam a diplomamunkához írt, általános alkalmazás. Ehhez időben egy kicsit előre fogunk szaladni, ugyanis a dolgozat megírásának az idején már nyilvánosan elérhető maga az applikációnak a production része, szóval szemléltető célzattal, az alábbi linken már meg is lehet tekinteni <https://todo.prđ.benidiploma.hu/> , a későbbiekben pedig kifejtem, hogyan „került ki ide” az alkalmazás.

### 4.1 A kód

Maga az alkalmazás szándékosan viszonylag egyszerűre sikeredett, ugyanis nem arra akartam kihegyezni a diplomadolgozatomat. Egy szimpla kis Todo (Teendők) alkalmazás, regisztrációs felülettel, felhasználói adatbázissal, különálló frontenddel és backenddel, így legalább egy kicsit a többrétegű szoftverarchitektúra rejtelmeibe is betekintést nyertem. Továbbfejleszteni megannyi féle módon lehetne, de mivel maga az üzemeltetési rész jelenti most a fő szempontot, így ezzel a részével kevésbé foglalkoztam. A célt, amire készült teljes mértékben ellátja, jól be tudok vele mutatni mindent, amit szeretnék.

#### 4.1.1 Backend

Néhány fontos dolog előzetesen.

A backend-nek a forráskódja az alábbi linken érhető el <https://gitlab.syscops.dev/benjamin.komjathy/diplomamunka-backend>

Az éles rendszer API-ja (application programming interface) pedig az alábbi linken <https://todo.prđ.benidiploma.hu/api>

Alapvetően Pythonban(3.8) íródott, a FastAPI keretrendszer felhasználásával, adatbázisnak pedig PostgreSQL relációs adatbázist használtam, abból is a 13-as főverziót. A FastAPI a Django-hoz és a Flask-hez eléggé hasonló keretrendszerről van szó, de egy kicsit újabb, és kicsit könnyedebb súlyú, és ki akartam próbálni, ezért esett erre a választásom. Még, ami pozitívum a másik két pythonos webes keretrendszerhez képest, hogy egy interaktív API dokumentációt is legenerál nekünk magától. Ez mind fejlesztéskor, mind teszteléskor jól jöhet. Az éles rendszer API dokumentációja az alábbi linken érhető el. <https://todo.prđ.benidiploma.hu/api/docs>

Servers

/api

Authorize

## Root

GET	/	Read Root	✓
GET	/db	Read Root	✓

## Users

GET	/users/	Read Users	✓
PUT	/users/	Modify User	✓
POST	/users/	Create User	✓
DELETE	/users/	Delete User	✓
GET	/users/me	Read Users	✓
POST	/users/login	User Login	✓
POST	/users/signup	Signup	✓
GET	/users/verify	Verify	✓

### 4.1. ábra Interaktív API dokumentáció

Összefoglalja nekünk az összes endpoint-ot, hogy milyen request-eket vár, példa JSON-okat (JavaScript Object Notation) is ír nekünk, na meg ténylegesen ki lehet próbálni az adott endpointokat, és lefut a háttérben, hogy mit is csinálna amikor meghívja a frontend.

Az OpenAPI definíciós leírást, amit kapunk, legeneráltam egy Redocs nevű tool-lal is, ez megtekinthető a mellékleteknél.

Az adatbázisban két táblánk van. A felhasználókat tartalmazó tábla és a teendőket tartalmazó tábla. Ezeket különböző API hívásokkal tudjuk lekérdezni (GET), új rekordot létrehozni (POST), szerkeszteni (PUT) és törölni (DELETE). Ez a 2x4 endpoint szimpla SQL lekérdezéseket futtat a háttérben.

Ezen kívül fontosabb endpoint még a /api (ami maga a root) ahol backend visszaad egy HTTP 200-as státusz kódot, hogy a backendünk egészséges. Ezt a frontendünk is hívogatja, és a Monitoring Stackünk Blackbox Exportere is, de ezekről majd később.

A /api/db endpoint az nem túl biztonságos, igazából magamnak csináltam tesztelésre, ott a megfelelő emailcímmel belépve le lehet kérdezni az egész adatbázist egyben, felhasználókkal, todo-kkal együtt, ténylegesen éles rendszerben ilyen endpoint természetesen nem lenne.

A felhasználó bejelentkezéshez használt `/api/users/login`, amin az utána kapott JWT-vel (JSON Web Tokens) autentikálja magát a felhasználó minden további API hívásnál. A `/api/users/me` endpointon meg az adott bejelentkezett userünk saját magát tudja lekérdezni, ez szintén a frontendünkhöz volt fontos.

Backend-en el lett még kezdve külön a regisztráció és email megerősítés két endpointja, ezek nem teljesen kidolgozottak, viszont működnek, de utána már ezzel a részével nem foglalkoztam tovább, ugyanis mint említettem, inkább az üzemeltetés részére akartam fektetni a hangsúlyt.

Természetesen sokkal több van a háttérben, mint az itt leírtak, minden endpointon szépen le van kezelve, hogy mit vár, milyen értékeket és requesteket fogad el, mit ad vissza, ha valami nem jó, legyen ez akár JSON response, vagy HTTP státusz kód, ugyan ez, hogy mi történik, ha megfelelő requestet kap stb.

Mivel az adatbázis maga is egy docker konténerben fut (erről részletesebben az 5.1.3 fejezetben lesz szó), és még csak a portja sincs kiengedve (ami Postgres-nél a default az 5432 lenne), hanem egy belső virtuális hálózaton kommunikál a backenddel, így külső forrásból feltörhetetlen. Egyedül a szervergépre bejutva lehetne feltörni, adatokat kinyerni az adatbázisból, de ezt meg az infrastruktúráról szóló részben megtudjuk, hogy miért nem olyan egyszerű ez.

Az alkalmazás szintén be lett dockerizálva, és már a lokális gépen való fejlesztés közben is úgy használtam, mivel kellett hozzá az adatbázis. Na de mit is jelent ez. A repoban láthatunk Dockerfile-okat, kettőt is, az egyiket a Production környezethez ([Dockerfile](#)), a másikat a Dev környezethez ([Dockerfile dev](#)), bár igazából a lokális fejlesztéshez is azt használtam, és a repositoryban található `docker-compose.yml` fájl. Amikor még csak fejlesztettem, akkor csak a `Dockerfile_dev` létezett. A `docker-compose`-ban láthatjuk a [build](#): utáni sorokat, ami annyit csinál, hogy a konténer indulásakor lebuildeli az adott mappában lévő `Dockerfile_dev`-et, vagy akár más Dockerfile-t is használhatunk és azt veszi alapul a konténerünkhöz, így elég egy

```
docker-compose up -d
```

parancs, és szépen elindult az alkalmazásunk, és hozzá az adatbázis, így localban is be lehet lőni fejlesztgetni, tesztelgetni. Maga az API a <http://localhost/api> endpoint-on fog figyelni. Még egy pozitívuma, hogy mind Windows-on, mind Linuxon tudtam haladni az alkalmazással így, ugyanis Docker-t lehet telepíteni mindkét OS-en.

A Production Dockerfile-ról pedig igazából annyit, hogy elindul egy Python 3.8-as base imageből, bele másolja az alkalmazásunkat tartalmazó mappát, feltelepíti a szükséges package-ket, dependency-



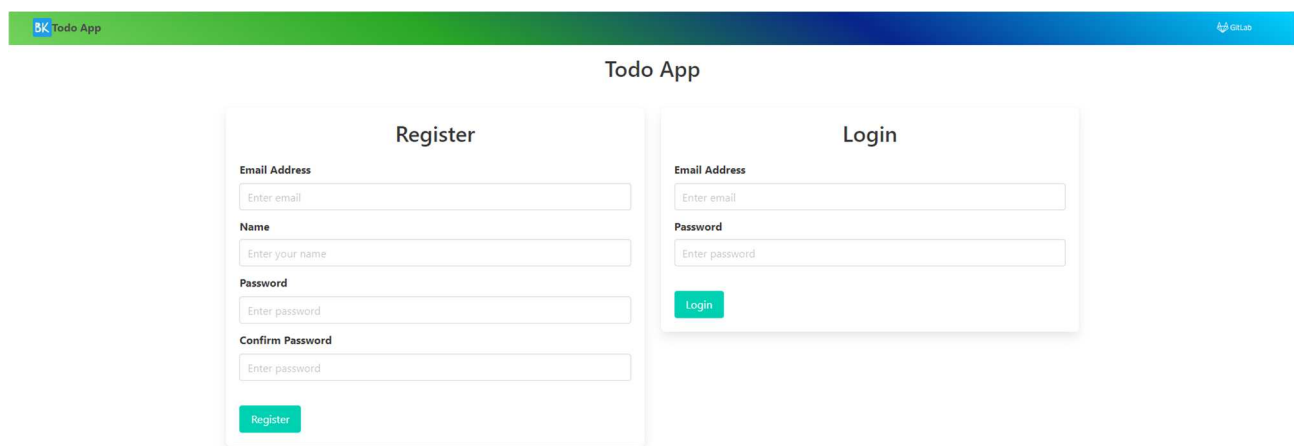
ket, és különböző init script-ek segítségével már futni is fog az alkalmazásunk a 80as porton, a konténeren belül.

#### 4.1.2 Frontend

Az igazat megvallva, az egész diplomamunkának ez volt a legnehezebb része, legalábbis számomra, és ez állt tőlem a legtávolabb, de mindenképp ki akartam próbálni, magát a webfejlesztést is, és a React keretrendszert is (ha már ilyen felkapott manapság), szóval adta magát, hogy a diplomamunkához szánt applikációhoz egy ilyen frontendet készítek. A React mellett szóló pozitívum, hogy különböző komponensekből épül fel, azt gyúrja össze a keretrendszerünk magává a vizuálisan is megtekinthető alkalmazássá, így újrafelhasználható.

A frontend-nek a forráskódja az alábbi linken érhető el <https://gitlab.syscops.dev/benjamin.komjathy/diplomamunka-frontend>

Az éles rendszer frontendje pedig az alábbi linken <https://todo.prd.benidiploma.hu/>

The image shows a web browser window with a green and blue header bar. The header bar contains the text 'BK Todo App' on the left and a small logo on the right. Below the header, the title 'Todo App' is centered. The main content area contains two side-by-side forms. The left form is titled 'Register' and has four input fields: 'Email Address' (with placeholder 'Enter email'), 'Name' (with placeholder 'Enter your name'), 'Password' (with placeholder 'Enter password'), and 'Confirm Password' (with placeholder 'Enter password'). There is a green 'Register' button at the bottom. The right form is titled 'Login' and has two input fields: 'Email Address' (with placeholder 'Enter email') and 'Password' (with placeholder 'Enter password'). There is a green 'Login' button at the bottom.

4.2. ábra Az alkalmazás, register/login felülete

Fejlesztés közben én 16 os NodeJS verziót és 8-as NPM (Node package manager) verziót használtam.

Ha ki akarjuk próbálni magát a frontendet, vagy belőni magunknak lokálban fejlesztéshez, akkor nincs más dolgunk, mint feltelepíteni a szükséges függőségeket az

```
npm install
```

paranccsal, utána az [example.env](#) alapján csinálni egy `.env` fájlt a mappában, ahol elérhető a diplomamunka backendünk. Ha lokálisan belőttük a backendet is, akkor egyszerűen úgy hagyjuk, ahogy az a példában van, de megadhatjuk neki ugyanúgy az éles rendszer backendjét is például

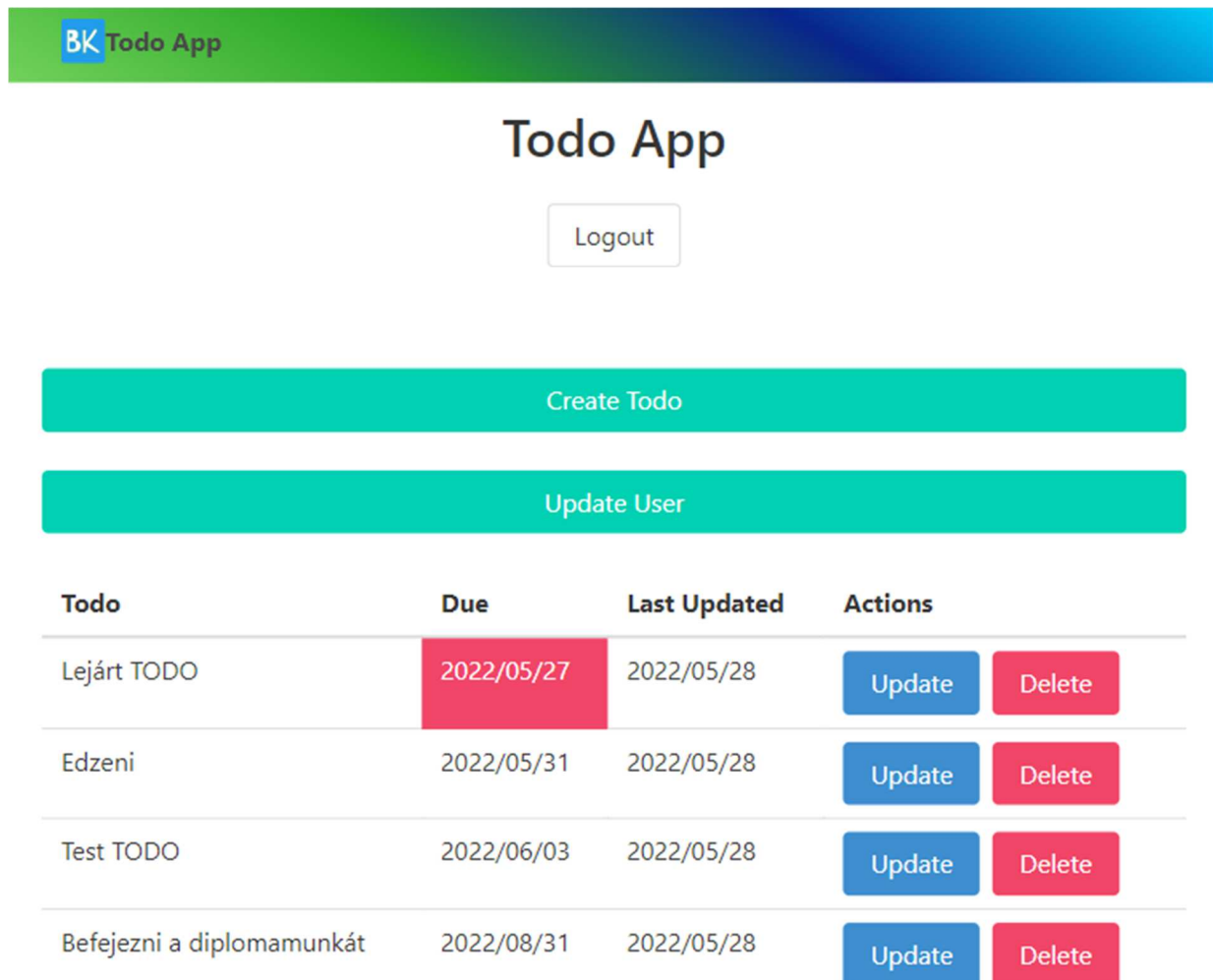
```
REACT_APP_BACKEND_URL=https://todo.prd.benidiploma.hu/api
```

Ezután egy

```
npm start
```

és már a 3000-es porton el is érhető a frontendünk, ami pedig folyamatosan frissül, ha változtatunk a kódon, ergo live debugger-ként is működik.

Hogy hogyan is működik. Alapvetően figyeli, hogy egészséges e a backend, és hogy be van e lépve a felhasználónk. Ha nincs, akkor láthatjuk a login és a regisztrációs felületet. Regisztrációkor figyeli, hogy létezik-e az adott emailcím, és hogy milyen erősségű jelszót szükséges megadnia a felhasználónak, ezt a backend megfelelő endpointjának a segítségével. Belépéskor szintén egy ahhoz szükséges endpoint-ra POST-ol, és a response alapján dönti el, hogy helyesek-e az adatok vagy sem. Ha helyesek, akkor a visszakapott JWT token bekerül a session-be, és onnantól a felhasználó eléri a saját teendőit, ha nem akkor kap megfelelő visszajelzést, hogy miért is nem tudott bejelentkezni.



4.3. ábra A felület bejelentkezés után

A felhasználó a saját teendőit szerkesztheti, vehet fel újat, és törölheti őket, ugyanez a saját adataival, módosíthatja az emailcímét, jelszavát, és törölheti is saját magát a rendszerből. Ezeket is a megfelelő API endpointok meghívásával. A teendőket sorba rendezi lejárat alapján, és pirossal jelzi a lassan lejárókat, vagy a már lejártakat, és látjuk, hogy mikor volt utoljára módosítva az adott teendő.

A [Dockerfile](#)-unk egy Node 16-os imaget vesz alapul, feltelepíti a függőségeket, mint ahogy mi is tennénk, de aztán simán `npm start` helyett

```
npm run build
```

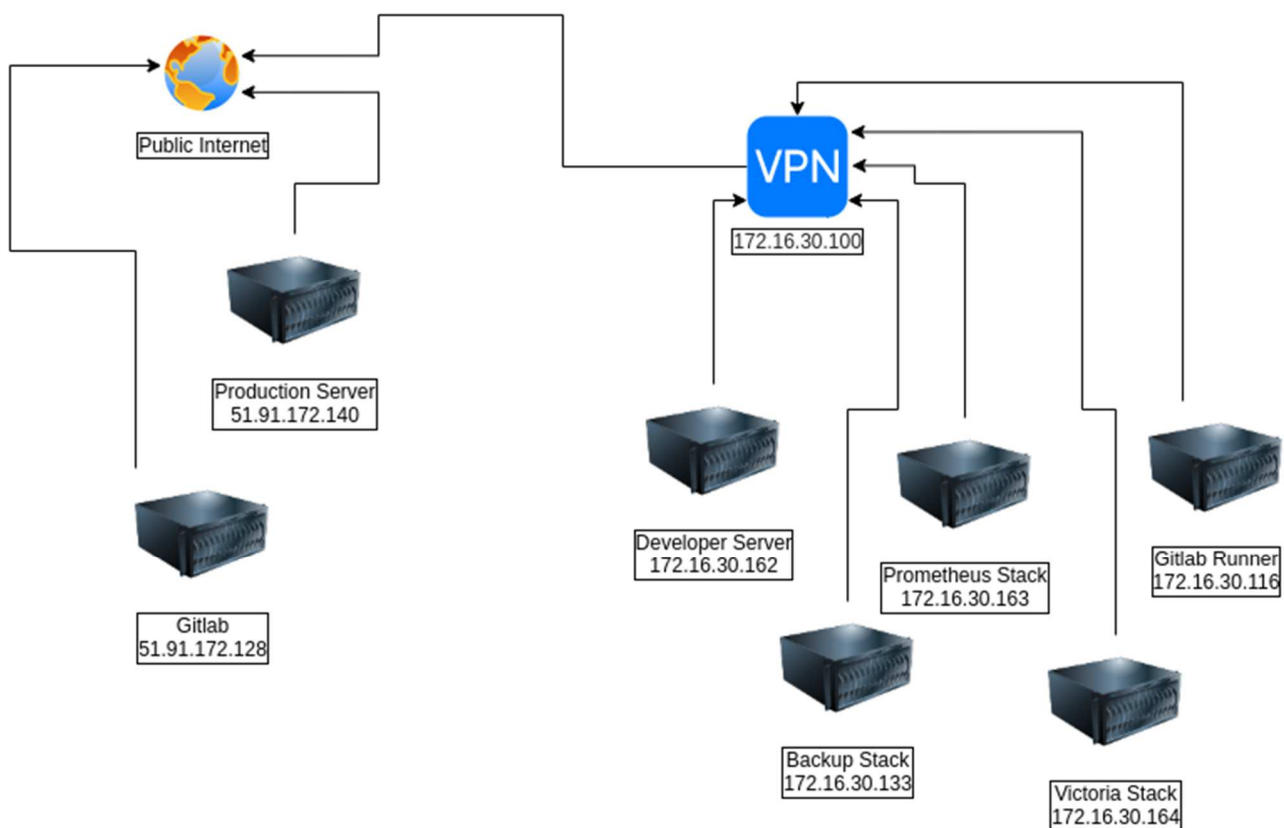
parancsot futtat, ami developer környezet helyett production-re optimalizált build-et csinál, amit ezután egy Nginx webserverral statikus weboldalként tudunk host-olni. Ez azért is jó nekünk, mert minify-olja és hasheli a kódot, így a tényleges forráskód nem kerül ki az internetre, így nem tudja csak úgy valaki ellopní, lemásolni.

## 5 OPERATIONS

Ebben a fejezetben a DevOps-nak az Operations, azaz az üzemeltetői részét vesszük górcső alá. Bemutatom a teljes infrastruktúrát, az automatizált létrehozását, a szükséges szolgáltatásokat, ezeknek a hogyanját és miértjét, és az előző fejezetben bemutatott alkalmazás üzemeltetésének rejtjelmeibe is bevezetem a kedves olvasót.

### 5.1 Infrastruktúra

Ahogy a Bevezetésben említettem, az infrastruktúra egy része, már meglévő szervergépeket és szolgáltatásokat tartalmaz, a többi része, erre a célra lett létrehozva. A teljes infrastruktúrának a diagramja az alábbi (5.1.) ábrán látható.



5.1. ábra - Az infrastruktúra

Kicsit a háttérrel. Az infrastruktúra VPN mögötti része Budapesten a RackForestben a SysCops által elhelyezett dedikált szervergépéből (on premise) virtualizáció útján (Proxmox) létrehozott VM-ekből (virtual machine) áll. A nyilvánosan is elérhető része pedig a <https://www.soyoustart.com/> -tól bérelt szintén Proxmox virtualizációs úton létrehozott VM-ekből. Még kiegészítésképp megemlítem, hogy ezek mind Ubuntu Server 20.04-et futtatnak.

A diplomamunka elkezdése előtt a következő VM-ek már rendelkezésre álltak:

- Gitlab – 51.91.172.128
- Gitlab Runner – 172.16.30.116
- VPN – 172.16.30.100
- Backup Stack – 172.16.30.133

VM-ek, amelyek csak és kizárólag a diplomamunkához általam lettek létrehozva:

- Production Server – 51.91.172.140
- Developer Server – 172.16.30.162
- Prometheus Stack – 172.16.30.163
- Victoria Stack – 172.16.30.164

A „benti” gépek csak VPN-en keresztül érhetőek el (persze van lehetősége ezek kivezetésére is), a nyilvánosak pedig értelemszerűen publikusak. Ezekre a publikus gépekre be SSH-zni (Secure Shell Protocol) csak megfelelő kulcsokkal lehet, a jelszavas belépés biztonsági okokból tiltva lett. Ezek a „megfelelő kulcsok” vagy 2048 (vagy több) bites RSA kulcsok, vagy pedig ED25519 eliptikus görbe titkosítás alapú kulcsok.

A Gitlab VM-en egy selfhosted Gitlab 14.5.0 CE (Community Edition) fut. Ezt a céges környezetben napi szinten használjuk, forráskódok és egyéb dolgok tárolására, de erre majd az alkalmazás üzemeltetése részben bővebben kitérek.

A Gitlab Runner VM-ről és szerepéről szintén később lesz szó. Egyelőre nagyon röviden annyit, hogy ez a Gitlab Runner futtatja a Gitlab-os CI/CD pipeline-okat (mily meglepő).

A VPN VM-en egy Pritunl és HAProxy kombó fut. Így megoldható a belső hálózatra való bejutás is (persze szabályozható, hogy ki léphet be), és ha valamit mégis nyilvánosan elérhetővé szeretnénk tenni, arra is van lehetőség. Például ami ide vágó dolog, hogy a Prometheus Stack-nek a grafikus dashboardjai az alábbi linken elérhetőek <https://grafana.monitoring.benidiploma.hu/> , pedig valójában belső hálózaton fut, csak ki lett engedve HAProxy-val. Ezekről a grafikus dashboardokról szintén később, a Monitoring részben lesz szó.

A Backup Stack igazából csak egy sima virtuális gép, nagy tárhellyel, amin pedig egy NFS (Network File System) lett beállítva. Itt is állítható, hogy honnan és ki érheti el. Mivel belső hálózaton fut, így aki eléri a belső hálózatot, a jelen beállítások szerint felcsatolhatja, és használhatja külső tárhelyként.

A Production és a Developer szerver lényegében ugyan azt a célt látja el, annyi különbséggel, hogy a Production szervernek nagyobb hardver van odaadva, és hogy nyilvánosan elérhető. Ezen kívül mindkettő arra van kialakítva, hogy maga az alkalmazás itt fusson, plusz még a kiegészítő szolgáltatások, erről szintén később.

A Prometheus Stack VM-en egy Prometheus Stack fut a Victoria Stacken pedig egy VictoriaMetrics Stack, ezek a szükségességéről és hogy mik is ezek szintén később.

A továbbiakban megnézzük, hogy hogyan is jött létre a felvázolt infrastruktúra automatizáltan, és részletesebben kifejtem az adott szolgáltatásokat, ahogy ígértem.

### 5.1.1 Terraform

Ahogy a hivatalos oldalukon olvashatjuk:

*„Terraform is an open-source infrastructure as code software tool that provides a consistent CLI workflow to manage hundreds of cloud services. Terraform codifies cloud APIs into declarative configuration files.”*

*„Terraform is an IaC (infrastructure as code) tool that allows users to define a desirable infrastructure definition in a declarative language. Using terraform the infra components within the environment can be deployed and treated as a code in terraform's configuration file that you can version, share and reuse.”* (Sabharwal, Pandey, Pandey 2021:6)

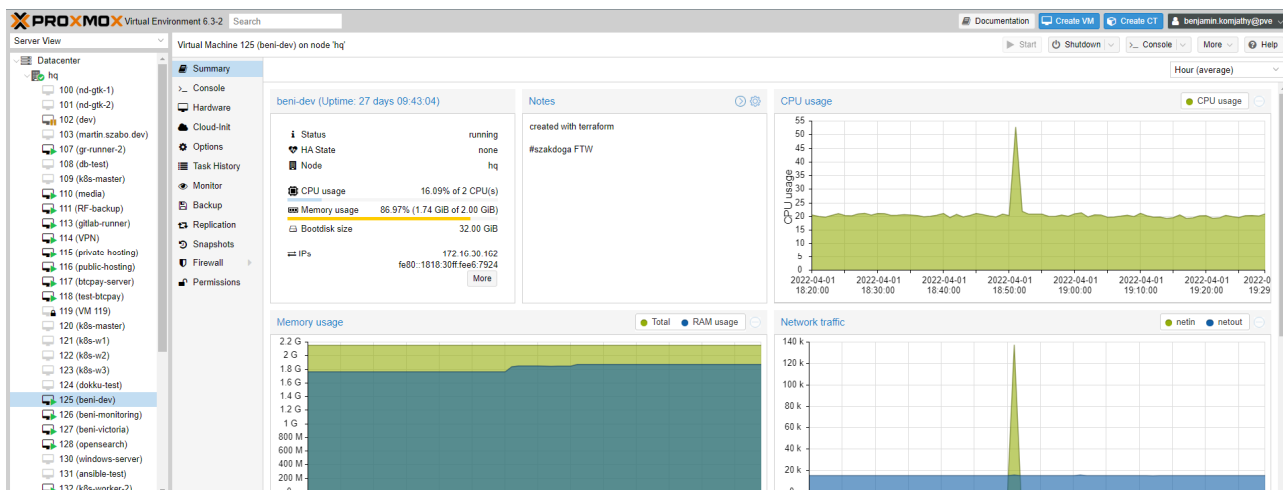
HCL (Hasicorp Configuration Language) leíró nyelven lehet vele leírni a teljes infrastruktúrát. Minden nagyobb felhőszolgáltatónak van hozzá külön provider-e. Mivel ahogy már említettem, mind két szervergép Proxmox alapon van virtualizálva, és a Proxmox API-hoz is van kész Terraform provider, így kézenfekvő volt, hogy ezzel automatizálom le a gépek létrehozásának folyamatát. Természetesen akár AWS-ben (Amazon Web Services), GCP-ben (Google Cloud Platform), vagy pedig Microsoft Azure-ban is létre hozhatóak ezek a resource-ök, annyi, hogy kicsit változik a kód, és persze ezek a felhőszolgáltatók nem ingyen állítják rendelkezésünkre a hardvereiket és szolgáltatásaikat. Természetesen kézzel is össze lehet ezeket kattintgatni, ugyanis ezeknek a felhőszolgáltatóknak van webes interface-ük is, de úgy se nem újra felhasználható, se nem automatizálható, és több hibalehetőséget is nyújt, na meg a bankszámlánknak se tesz jót, ha véletlenül valamit elfelejtünk törölni, miután már nem használjuk. A teljes Terraform kód az alábbi Gitlab repository-ban elérhető <https://gitlab.syscops.dev/benjamin.komjathy/diplomamunka-terraform>

Az itt található fájlok közül számunkra a [main.tf](#) a legfontosabb. A felépítése a következő. Az elején megmondjuk a terraformnak, hogy mit használjon backendnek. Ez lehetne a lokális gépünk is, vagy egy S3 bucket AWS-ben, vagy pedig most a jelenlegi példánkban ez a Gitlabnak a saját terraform state tárolója. Ebben a state-ben tároljuk, hogy jelenleg, hogy áll az infránk, és a terraform mindig csak az ehhez képest történt módosításokat valósítja meg, akár hozzá akarunk tenni egyéb resourcet, akár módosítani, vagy törölni egy már meglévőt. A backendünket az [example.backend.conf](#) alapján tudjuk felconfigurálni. Utána adjuk meg a terraformnak, hogy milyen providert használjon, annak a forrását és verzióját. Ahogy már említettem, mi jelen állás szerint a Proxmox provider-ét használjuk. Utána beállítjuk az adott provider-eket, hogy hol érhető el az api endpoint amihez csatlakozni akarunk, és hogy mivel tudjuk azonosítani magunkat. Ez ki lett szervezve változóba, értelemszerűen, hogy a repository-ba szenzitív adatok ne kerüljenek fel. Hogy milyen formában tudjuk megadni ezeket a változókat, az az [example.tfvars](#) -ban látható. Ha tovább megyünk, látható egy kikommentelt template a vm-ek létrehozásához, ez alapján bármennyi virtuális gépet létre tudnánk hozni, amíg bírja erőforrással a Proxmoxunk. A kód további részében a már beállított virtuális gépek láthatóak. Ha ezt le akarnánk futtatni magunknak, annyi lenne a teendő, persze feltételezve, hogy a Terraform CLI (Command Line Interface) már telepítve van, hogy létrehozzuk a szükséges backend configot és tfvars fájljainkat a megfelelő értékekkel, és lefuttatni a következő parancsokat.

```
terraform init -backend-config=backend.conf
terraform plan -var-file=env.tfvars -out=env.tfplan
terraform apply env.tfplan
```

Az init parancssal leszedi nekünk a providerhez szükséges fájlokat, és beállítja a backedet, a plan-nel kapunk egy tervezetet, hogy miket fog létrehozni, módosítani, törölni, az apply-jal pedig elfogadjuk, hogy csinálja meg a tervet.

A Proxmoxnak a webes felületén pedig így néz ki egy virtuális gép. Néhány fontosabb adatot itt is látunk, például hardvert, hogy mióta fut az adott gép, és az erőforrások jelenlegi kihasználtságát.



5.2. ábra Proxmox UI

### 5.1.2 Ansible

Az Ansible szintén egy automatizációs eszköz a DevOps-osok kezében, csak ennek a Terraformmal szemben, nem magának az infrastruktúrának a létrehozása a célja, hanem hogy a Terraform által létrehozott gépeken a lehető legautomatikusabb módon minden szükséges program, szolgáltatás stb. rendelkezésre álljon, hogy már az indulása után nem sokkal, mindent ki tudjon szolgálni, amire szánták. Jöjjön szintén egy kis „saját magukról mondták”.

*„Ansible is a radically simple IT automation system. It handles configuration management, application deployment, cloud provisioning, ad-hoc task execution, network automation, and multi-node orchestration. Ansible makes complex changes like zero-downtime rolling updates with load balancers easy.”*

*„Ansible enables you to easily deploy applications and systems consistently and repeatably using native communication protocols such as SSH and WinRM. Perhaps most importantly, Ansible is agentless and so requires nothing to be installed on the managed systems (except for Python, which, these days, is present on most systems). As a result, it enables you to build a simple yet robust automation platform for your environment.”* (Oh, Freeman, Locati 2020:32)

Alapvetően annyit csinál, hogy belép (SSH) a szintén konfigurációban megkapott gépekre, és azokon futtat különböző előre megírt parancsokat. Viszont ez azért lesz nekünk hasznos, mert teljesen újra felhasználható kódokat lehet írni benne, kiváltozózva, különböző elágazásokat, ciklusokat felhasználva, hogy az életünket a lehető legjobban megkönnyítse, hogy ne nekünk kelljen mindig ugyan azokat, vagy kicsit más parancsokat lefuttatni kitudja hány gépen, hanem csak egyszer megírjuk hozzá a template-et és azokat bármennyi alkalommal újra tudjuk futtatni.



Az én általam ehhez a projekthez írt Ansible kód az alábbi repository-ban található meg.

<https://gitlab.syscops.dev/benjamin.komjathy/diplomamunka-ansible>

Nézzük akkor végig ezt is, hogy mit tartalmaz, és miért.

Az Ansible scriptünk lelke a playbooks mappában található [main.yml](#) ebben adjuk meg, hogy milyen host-okon milyen role-okat futtasson, milyen változókkal. A host-oknak az ip-jét, hogy hova menjen az Ansible pedig az [inventory](#) fájlból kapja meg. Egy role pedig annak a leírása, hogy miket szeretnénk, hogy végrehajtsa az adott gépen. Látható, hogy egyszerre több role-t is meg tudunk adni, hogy futtasson. Hat különböző role készült el a projekthez, ezekről most egy kicsit bővebben.

#### 5.1.2.1 *Basic*

[/roles/basic/tasks/main.yml](#)

Ez a role arra szolgál, hogy tényleg a legalapvetőbb előkészületeket megtegye és az alap programokat feltelepítse a friss OS-ünkre. Beállítja a gépünk host nevét, berakja a projekthez használt kulcsnak a publikus részét az `authorized_keys` fájlba. Ez ahhoz szükséges, hogy majd a Gitlab runnerünk is gond nélkül be tudjon lépni a gépre, amikor futtatja a CICD pipeline-okat. Erről is később lesz részletesebben szó. Majd jön néhány alapvető program, zip, curl, jq, NFS client. A production gépre a Pritunl klienset is feltelepíti, és becsatlakoztatja a VPN-ünkre. Ez azért szükséges, hogy a Prod gép is fel tudja magának csatolni az NFS tárhelyet, és arra tudjon mentéseket készíteni az adatbázisról. Ezután fel is csatolja a disk-et az összes gépre, és végezetül a Prod és a Dev gépen elvégzi a szükséges előkészületeket, hogy tudja backup-olni az alkalmazásunk adatbázisát. Erről részletesebben szintén később.

#### 5.1.2.2 *Docker*

[/roles/docker/tasks/main.yml](#)

Ez a role semmi extrát nem csinál, csak feltelepíti a Dockert és a Docker-Compose-t. Nem szeretem magam ismételni, de erről is később részletesen.

#### 5.1.2.3 *Exporter*

[/roles/exporter/tasks/main.yml](#)

Ő felmásolja nekünk a CAdvisorhoz és a NodeExporterhez szükséges `docker-compose.yml` fájlokat, és elindítja ezeket a konténereket. Ennek a rendszermonitorozáshoz lesz majd köze, szintén később.

#### 5.1.2.4 Monitoring

</roles/monitoring/tasks/main.yml>

A teljes Prometheus Stackünkhöz szükséges fájlokat felmásolja, és elindítja.

#### 5.1.2.5 Traefik

</roles/traefik/tasks/main.yml>

A Traefik-hez szükséges fájlokat másolja fel és indítja el a konténert.

#### 5.1.2.6 Victoria

</roles/victoria/tasks/main.yml>

A Victoria Stackhez a fájlok, és a konténer elindítása.

Az egésznek a lefuttatáshoz mindössze az Ansible CLI-ra, az inventory kitöltésére, és a playbooks/main.yml fájlban a szükséges változók megadására, és az

```
ansible-playbook playbooks/main.yml
```

parancs lefuttatására van szükség, utána hátra is dőlhetünk, a gép dolgozik, az alkotó pihen.

### 5.1.3 Docker és Konténerizáció

*„Docker is a tool that promises to easily encapsulate the process of creating a distributable artifact for any application, deploying it at scale into any environment, and streamlining the workflow and responsiveness of agile software organizations.” (Kane, Matthias 2018:17)*

Nagyon leegyszerűsítve, olyan virtuális környezet, ami a legalapvetőbb feltételeket megteremti, hogy egy applikáció el tudjon futni benne. Mindemellett platform független, ugyan azt az alkalmazást a fejlesztők fejleszthetik Windows-on, Linuxon, Mac-en egyaránt. Dockerfile-okat írunk hozzá, amit már az alkalmazásról szóló fejezetekben láthattunk korábban, ezzel írunk le egy docker image-t. Adhatunk meg benne már meglévő docker imaget, mint szülőt, hogy azt vegyük alapul, futtathatunk különböző parancsokat, adhatunk meg környezeti változókat és argumentumokat, és adhatunk meg parancsokat, hogy mi történjen amikor az adott imageből elindul egy konténer. Ebben az image-ben fog szerepelni minden, ami ahhoz szükséges, hogy a dockerizált alkalmazásunk fusson. Bármennyiszer reprodukálható, újrafelhasználható. Az image-ek a nevük mellett még egy tag-et is kapnak, ami egy image egy adott kiadásának azonosítására szolgál. A konténer pedig egy példányosított docker image, egy adott konténer csak egyszer létezhet, viszont könnyen létrehozható bármennyi konténer egy adott image-ből. Még egy CLI toolt is kapunk a dockerhez, ezzel nagyon

sok mindent tudunk managelni, pl imaget buildelni, konténert indítani belőle, annak megnézni a logjait, bele menni(exec), ha esetleg hibát keresünk, vagy egyéb hasonló okokból stb. A docker-compose pedig ezt tool-set-et egészíti ki. Ennek a leíró nyelve yml szintaktikában a konténereknek a leírására szolgál, hogy minél egyszerűbben tudjuk őket beconfigurálni, majd pedig különböző parancsokkal őket futtatni, logokat nézegetni, megnézni az állapotukat stb. Mind a diplomamunkához készült alkalmazás, mint már korábban láthattuk, és mind a továbbiakban felsorolt szolgáltatások docker-compose.yml-ban leírt docker konténerek. Az alapok megértéséhez ennyi elég.

#### 5.1.4 Traefik

*„Traefik is an open-source Edge Router that makes publishing your services a fun and easy experience. It receives requests on behalf of your system and finds out which components are responsible for handling them.”*

A traefik is egy konténerben futó szolgáltatás lesz, reverse proxy feladatokat lát el és emellett certificate-ek menedzselésére szolgál. A lényege, hogy a 80-as és 443-as portot (http, https) megkapja, ezeken figyel és a különböző címkék alapján, (amit szintén a docker-compose.yml-ban kapnak meg az adott konténerek) irányítja a forgalmat az adott konténer megfelelő portjára, valamint még megfelelő certificate-et is generál, hogy biztonságos https kapcsolatunk legyen Ezeket a certificate-eket három havonta megújítja, nehogy lejárjon, ha pedig valami probléma lenne vele, még emailt is küld. Van egy dashboard-ja is, ami például az alábbi linken megtekinthető. <https://traefik.prd.benidiploma.hu/>. Biztonsági okokból van rajta basic-auth, hogy azért bárki ne tudja csak úgy megnézni. Az admin:96Deep15 kombóval be tudunk lépni, és szét tudunk nézni.

Lássunk egy példát a gyakorlatban. A Production géphez a DNS fel van véve úgy, hogy az adott gépre mutasson. Valaki, egy felhasználó, beírja a böngészőbe az alkalmazás frontendjének a címét. <https://todo.prd.benidiploma.hu/> . Ezt először a traefik fogja megkapni, ugyanis ő figyel a már korábban említett port-okon. A frontend docker-compose.yml-jában pedig a következő címkék láthatóak

```
labels:
- traefik.enable=true
- traefik.http.routers.todo_frontend.rule=Host(`todo.${TAG}.benidiploma.hu`)
- traefik.http.routers.todo_frontend.tls=true
- traefik.http.routers.todo_frontend.tls.certresolver=dnsResolver
- traefik.http.routers.todo_frontend.entrypoints=websecure
```

```
- traefik.http.routers.todo_frontend.tls.options=default
- traefik.http.routers.todo_frontend.middlewares=https-redirect@file,security-headers@file

- traefik.http.routers.todo_frontend-insecure.rule=Host(`todo.${TAG}.benidiploma.hu`)
- traefik.http.routers.todo_frontend-insecure.middlewares=https-redirect@file,security-headers@file
- traefik.http.routers.todo_frontend-insecure.entrypoints=web
- traefik.http.services.todo_frontend.loadbalancer.server.port=80
- traefik.http.routers.todo_frontend.service=todo_frontend
- traefik.http.routers.todo_frontend-insecure.service=todo_frontend
```

Így a traefik tudni fogja, hogy mivel erre a domainre jött a request, neki ebben a konténerben a 80-as portra kell irányítania a kérést, és így annál a felhasználónál, aki rákattintott a linkre, már be is tölt az oldal, és át is irányítja https-re, hogy mindenképp biztonságos legyen a kapcsolat. Nagyon hasznos és nagyon egyszerűen használható szolgáltatás, szinte elengedhetetlen.

#### 5.1.5 Portainer

Ez a szolgáltatás az adott gépen futó docker-es dolgok menedzselésének a megkönnyítésére szolgál, ugyanis nem kell be SSH-znünk az adott gépre, és ott különböző docker, docker-compose parancsokat futtatnunk, hanem szolgáltat nekünk egy webes felületet minderre. A Production gépnek a portainer-e az alábbi linken érhető el. <https://portainer.pr.d.benidiploma.hu/>, ide is be tudunk lépni a már korábban említett admin:96Deep15 felhasználónév-jelszó párossal. Nagyon egyszerűen belőhető szolgáltatás, de ez is nagyon hasznos. Szintén konténerben fut.

#### 5.1.6 Monitoring

A Monitoring Stack-ből csak egy van (bár az két gépre szétszedve, ennek az okáról kicsit később), az is ahogy már említettem, VPN mögött, mivel alapvetően ezek az adatok úgyis csak az adott cég számára fontosak, akik üzemeltetik az infrastruktúrát. Minden fontosabb dolog, legyen az hardver, weboldal, backup-job, amit monitorozni akarhatunk, ide van bekötve. Több konténer együttes munkája, hogy ez az egész működik és méghozzá jól. Ahhoz, hogy vizuálisan is jobban élvezhető legyen ez a rész, így a különböző szolgáltatásokat, legalábbis amik a Prometheus Stack-hez tartoznak, kiengedem traefikkal, és mindegyiknek a linkjét odarakom az adott szolgáltatáshoz, hogy hol elérhető. Ez éles környezetben nem szokott így lenni, csak az adott gép ipjén és portján szoktuk megnézni, ha valamilyen esemény történt. Egyedül a Grafanát szoktuk publikusan elérhetővé tenni,

annak a linkjét már amúgy is láthattuk korábban. Most lássuk a részeit, hogy hogy is épül fel ez az egész.

#### 5.1.6.1 Prometheus

Az egész Monitoring Stack-nek a lelke. Ő felel azért, hogy az összes mindenféle metrikákat, amit a többi szolgáltatás termel, összegyűjtse egy helyre.

*„Prometheus is an open-source systems monitoring and alerting toolkit. Collects and stores its metrics as time series data, i.e. metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels.”*

*„Prometheus does one thing and it does it well. It has a simple yet powerful data model and a query language that lets you analyse how your applications and infrastructure are performing. It does not try to solve problems outside of the metrics space, leaving those to other more appropriate tools.”*  
(Brazil 2018:11)

Konfigurálni lehet, hogy milyen szolgáltatásokhoz menjen el, milyen eseményeknél triggereljen alert-et. A mi jelen esetünkben be vannak kötve a dev és prod os gépek, az alkalmazás oldala, és a különböző backup job-ok, ezeket figyeli, gyűjti a metrikákat, és triggereli az Alertmanager-t, ha baj van.

Itt érhető el. <https://prometheus.monitoring.benidiploma.hu/>

Itt tudunk különböző query-eket futtatni az összegyűjtött metrikákon, megnézni az alerteket, hogy egyáltalán miket figyel, és hogy váltódott e ki valami.



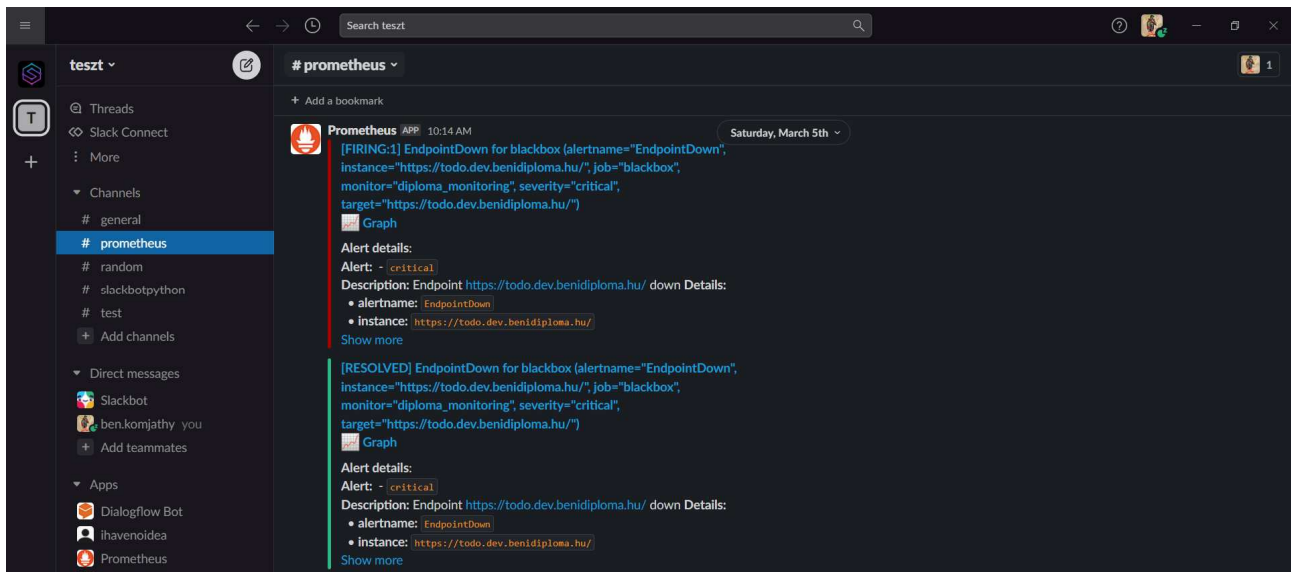
5.3. ábra Példa Prometheus query

#### 5.1.6.2 AlertManager

Ez a szolgáltatás felelős azért, hogy ténylegesen meg is kapjuk az értesítést, ha valamilyen esemény kiváltódott azok közül, amiket figyel a Prometheus. Szabadon konfigurálható, meg lehet adni, hogy hova, és milyen formában menjenek az értesítések. Ez lehet email, Slack, PagerDuty, és még sok más.

Ez a stack például Slack-re van bekötve, és annak megfelelő template van odaadva a szolgáltatásnak.

Egy példa, hogy hogy is fog kinézni egy ilyen alert.



5.4. ábra - Egy Slack értesítés az AlertManagertől

Azt láthatjuk, hogy nem volt elérhető az alkalmazásunknak a Dev környezetben lévő frontendje. Arról is szólt, hogy megoldódott a probléma, azaz újra elérhető.

Az alábbi linken érhető el. <https://alertmanager.monitoring.benidiploma.hu/>

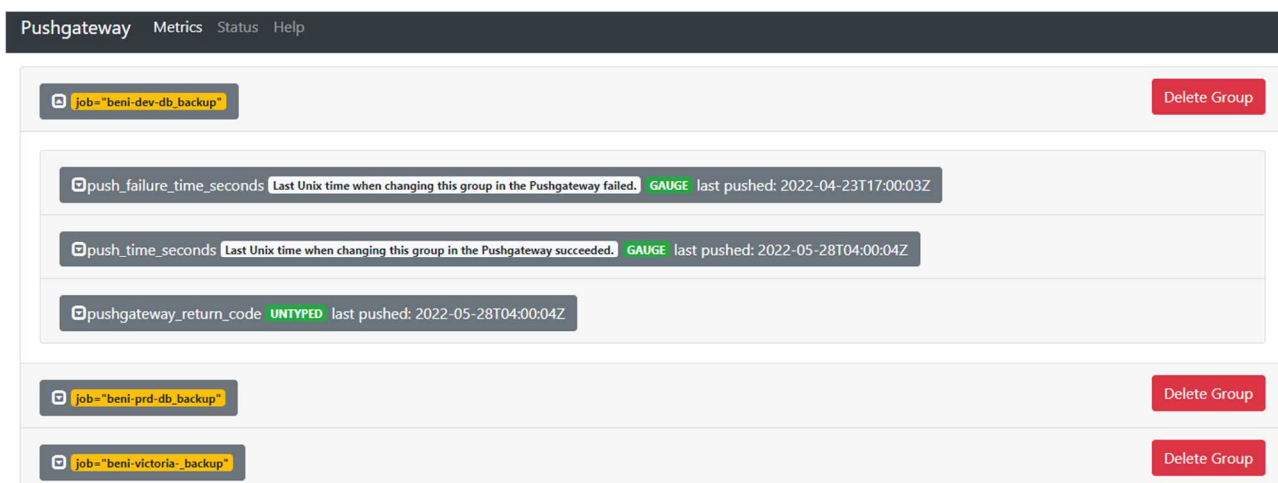
Itt a triggerelődött alerteket látni, meg esetleg lehet némítani őket, ha akarjuk, mert valami épp mégse lenne annyira releváns.

#### 5.1.6.3 PushGateway

Ez a szolgáltatás felel a különböző batch-job-ok monitorozásáért. A diplomamunkában én a különböző Crontab-ban időzített backup-ok figyelésére használtam, hogy minden rendben lefutott e. Ezt is lehet többféle módon használni, számomra csak ahhoz kellett, hogy triggereljen egy alertet a Prometheusban, hogyha valamelyik backup lefutása bármivel mással tért volna vissza, mint a nullás hibakód, azaz, hogy nincs hiba.

Itt érhető el. <https://push.monitoring.benidiploma.hu/>

Látható, hogy milyen backup-ok, és milyen hibakóddal futottak le, és mikor.



## 5.5. ábra PushGateway UI

### 5.1.6.4 NodeExporter

A NodeExporter a rendszermetrikákért felel, mármint ténylegesen a hardver elemekért (CPU, RAM, disk stb.). Összegyűjt mindent az adott gépről, a Prometheus pedig begyűjti ezeket, és szintén itt is tudjuk figyelni, hogyha például túl nagy a CPU terhelés, vagy fogy a tárhely a gépen akkor szintén kiváltódik egy alert és kapunk róla értesítést.

A Prometheus Stack NodeExportere itt érhető el. <https://nodeexporter.monitoring.benidiploma.hu/>

Itt láthatjuk a metrikákat, bár így nem túl izgalmasak, de ezeket lehet szépen vizualizálni, erről a Grafanas részben olvashatunk.

### 5.1.6.5 Cadvisor

A Cadvisor kicsit hasonló, mint a NodeExporter, csak ő a rendszermetrikák helyett a konténerek metrikáit gyűjti össze. Egy konténer, ami a konténereket figyeli. Figyelhetjük vele, ha valamilyen konténer leállt, vagy szintén, ha csak túl sok erőforrást vesz el a hardvertől stb.

Itt érhető el a Prometheus Stack VM Cadvisor-ja. <https://cadvisor.monitoring.benidiploma.hu/>

Sok sok metrikát láthatunk, ő pedig már valamennyit vizualizál is rajta nekünk.

### 5.1.6.6 BlackboxExporter

A BlackboxExporter különböző endpoint-ok monitorozására való. Tudjuk figyelni vele, hogy egyáltalán elérhető-e, hogy milyen http kód-ot ad vissza, tudjuk szabályozni, hogy mi számít jónak, mit fogadjon el, a certificate-eket is tudjuk nézni, hogy van e egyáltalán és meddig érvényes még.

Itt érhető el. <https://blackbox.monitoring.benidiploma.hu/>



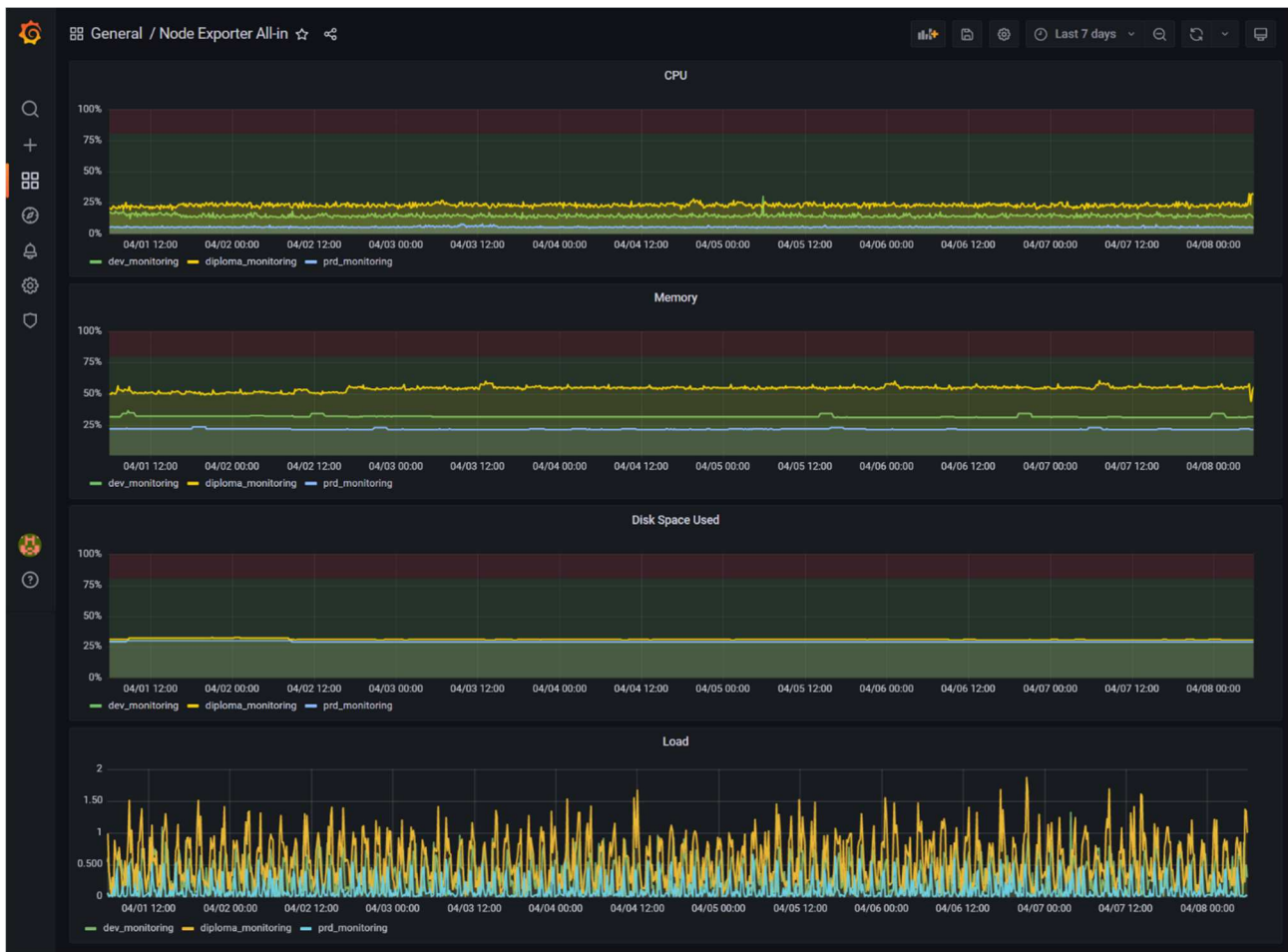
Az alkalmazás prod és dev környezetét is figyeli, a frontendet és a backendet egyaránt.

#### 5.1.6.7 Grafana

A Grafana, mint már többször is szóba került, a Prometheus által összegyűjtött rendszermetrikák vizualizására szolgál. Végtelen mennyiségű dashboard-ot készíthetünk, vagy tölthetünk le és használhatunk fel a netről. Ezt lehet akár JSON formátumban megadni neki, vagy a webes felületen összekattintgatni. Prometheus query-eket használ ő is, ahogy már láhattuk korábban, csak szép grafikonokat csinál belőlük.

A diplomamunkához felhúzott Grafana itt érhető el. <https://grafana.monitoring.benidiploma.hu/>

Már említettem, hogy általában csak ezt szokták nyilvánosan elérhetővé tenni, ezért ezen alapról van is autentikáció, ide is az admin:96Deep15 kombóval tudunk belépni. Ha rákattintunk a kis nagyító ikonra, akkor láthatóak, hogy milyen dashboardokat raktam be. Vannak dashboardok a NodeExporterhez, Cadvisorhoz, BlackboxExporterhez is. Talán a kedvenc, és egyben a legfontosabb dashboard a következő ábrán látható. Ezen a bekötött gépek rendszermetrikái láthatóak.



5.6. ábra Node Exporter All-in Dashboard

#### 5.1.6.8 VictoriaMetrics

A VictoriaMetrics lesz az a szolgáltatása a teljes Monitoring Stacknek ami külső gépen fut. Erre két okból is szükség van. Az egyik az, hogy a Prometheus nem jó sem hosszútávú adattárolásra, sem pedig nincs kiforrott megoldása a backup-olásra. A másik, hogy még ha esetleg az adataink el is vesznek a Prometheus-os VM-en, legyen egy másik hely is arra, ahol vissza tudjuk nézni a monitoring metrikáinkat, hogy éppen mi is történt, hátha kapunk valamilyen információt arról, hogy miért eshetett ki az egyik esetleg másik gép. Ezekre nyújt megoldást a Victoria Metrics. A Prometheusnak van remote write konfigurálási lehetősége, és be van neki rakva, hogy az éppen aktuális metrikákat, amiket összegyűjtött, tovább küldje. Emellett hosszabb távon is meg tudjuk őrizni a metrikákat a Victoriában, és backup megoldása is van, és még az adatokat külön időpont specifikusan vissza is tudjuk tölteni, hogyha tegyük fel arra lennénk kíváncsiak, hogy milyen adatokat kaptunk tegyük fel 3 héttel ezelőtt (ami már a Prometheus-ban nem lenne látható), vagy még régebben. Ezt két ábrával fogom szemléltetni, a már korábban említett Grafana dashboard-ok segítségével. A két ábra nagyjából

ugyan akkor készült. Az első ábra a Prometheus-os gép Grafanája lesz, a másik pedig a Victoriás gépe. A szemléltetés kedvéért mindkettőt beállítottam 28 napra, hogy ezen az intervallumon szemléltessem az adatokat.



5.7. ábra – Prometheus Stack Grafana

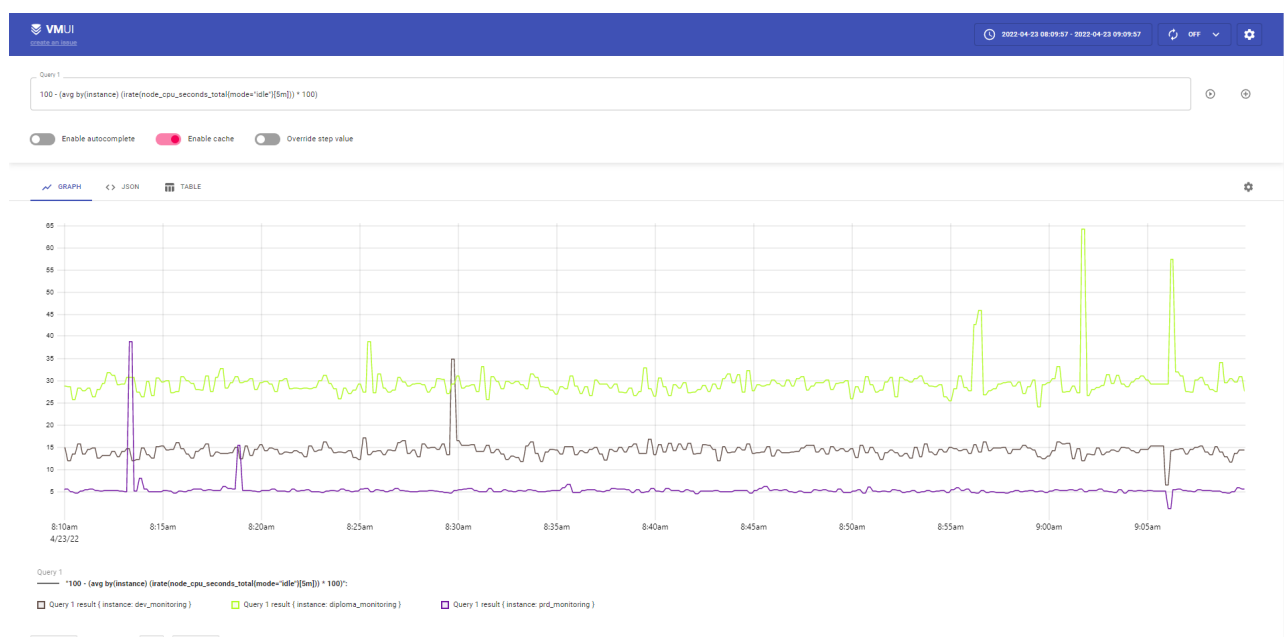


5.8. ábra – Victoria Stack Grafana

Ahogy látható, a második ábrán a teljes intervallumról meg vannak a rendszermetrikáink, míg az első ábrán csak 200 órával ezelőttig.

Sőt, még ha esetleg a Victoria VM-ünk is kiesne, mivel innen még tovább is volt backup-unk, egy harmadik helyre, jelen esetben ez a Backup Stackünk, így, ha a korábban már említett Terraform és Ansible automatizációknak hála, nagyon gyorsan be tudjuk hozni újra ezt a gépet, és visszatölteni bele az adatokat, így nullához konvergáló adatvesztés mellett rá tudunk nézni, hogy mi is történt.

Ennek a webes felületét nem tudom megmutatni, mert ugye ez a HAProxy-n keresztül nem lett kivezetve, hogy nyilvánosan elérhető legyen. Itt is tudunk például ugyan olyan query-ket futtatni, mint a Prometheus-nál.



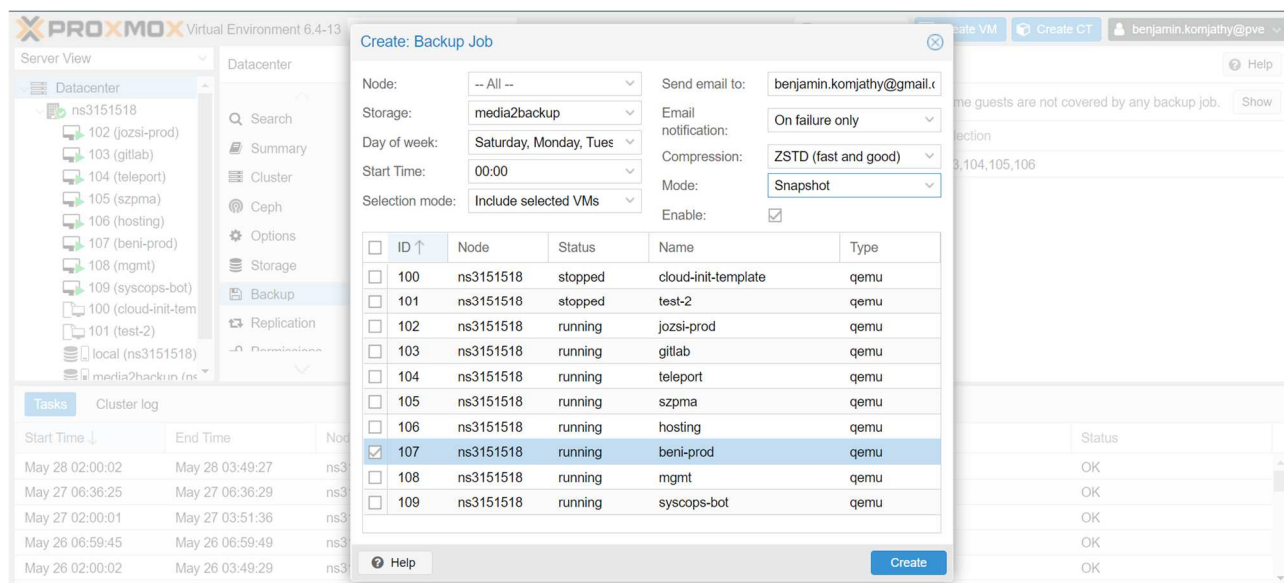
5.9. ábra – Példa query a Victorián

### 5.1.7 Backup

Infrastruktúra részről, mint látható, nem nagyon van mit backup-olni. A Terraform és Ansible kód ugyebár Gitlabon megtalálható (ez persze backup-olva van, csak céges oldalról, így arra nem térnék ki külön), meg a lokális gépen is. Ezekkel, mint láttuk könnyen megoldható új gépek behozása és beállítása, hogy ellássák a céljukat. Egyedül a VictoriaMetrics adatait érdemes menteni, erre van egy megfelelő backup script az adott gépen, ami minden nap egyszer lefut, kiexportálja az éppen aktuális adatokat, és továbbítja a Backup VM-re. Ez a Cronjob pedig be van kötve a már említett PushGateway-be, és szól, ha esetleg nem sikerült volna lefutnia.

A Proxmoxnak is van külön egy olyan feature-e, hogy bizonyos időközönként csináljon snapshotot adott VM-ekből, és ezt érdemes is használni éles rendszereknél, így a Produciton gépen be is

állítottam, hogy ez lefusson minden nap. Emailben tud szólni, ha valami félrecsúszna. Ezekből a snapshotokból azért egy fokkal könnyebb visszaállni, mint teljesen új gépet behozni.



5.10. ábra – Proxmox Backup Feature

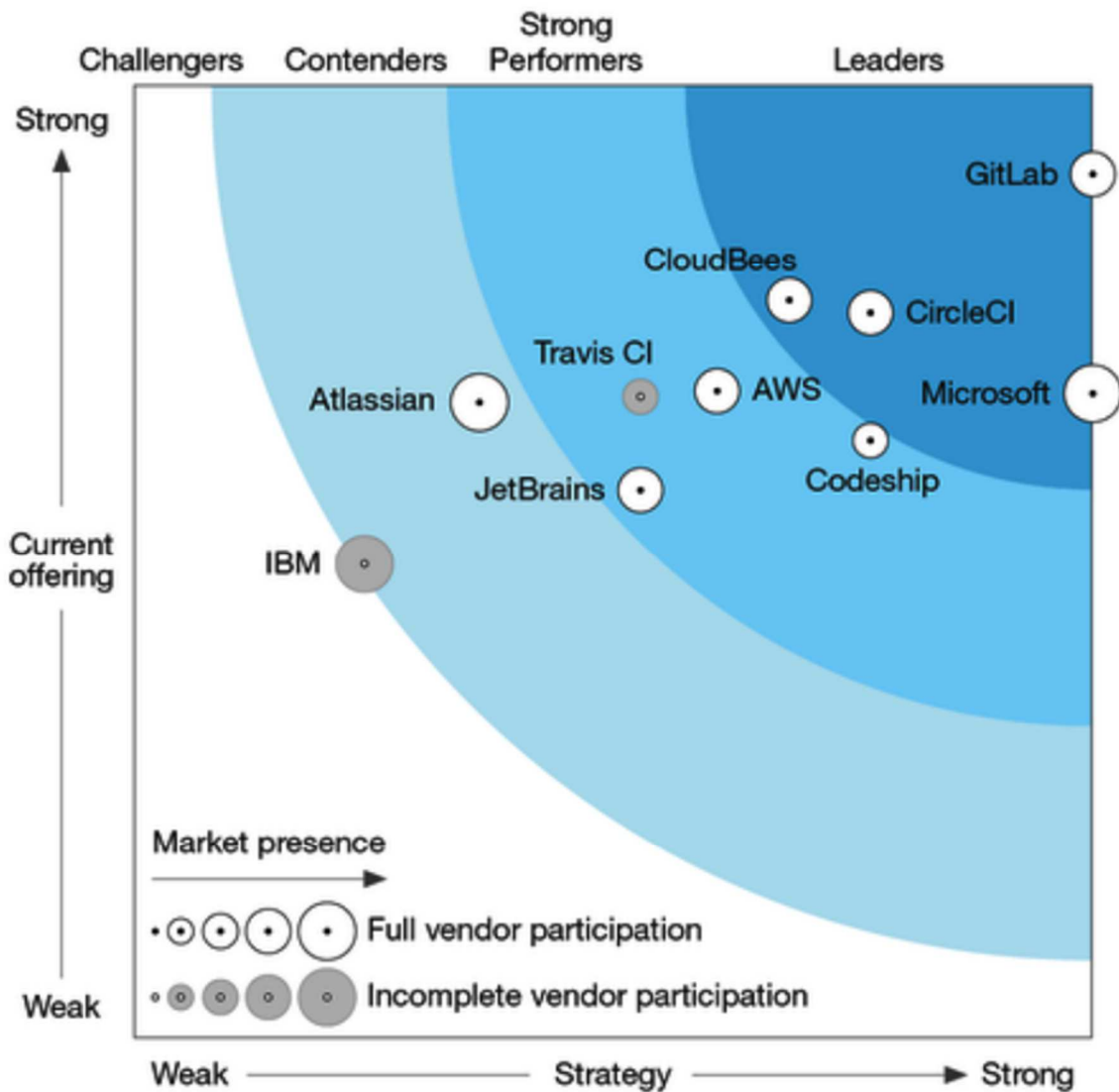
## 5.2 Az alkalmazás üzemeltetése

### 5.2.1 Gitlab

*„GitLab is The DevOps platform that empowers organizations to maximize the overall return on software development by delivering software faster and efficiently, while strengthening security and compliance. With GitLab, every team in your organization can collaboratively plan, build, secure, and deploy software to drive business outcomes faster with complete transparency, consistency and traceability.”*

Ha nagyon röviden be akarom mutatni, a Gitlab egy git alapú verziókezelő, hasonló, mint a Github, viszont sokkal jellemzőbb céges környezetben. Ez több okból kifolyólag is így van. A legfontosabb talán az, hogy van selfhosted verziója, így olyan onpremise, vagy cloud gépen futtatjuk magunknak, ahol csak szeretnénk, így nem kell attól félni, hogy egy jelszó kikerülésével az egész kódrendszerünk nyilvánossá válik, nem kell attól se félni, hogy egyszerűen csak valamilyen hiba történik és eltűnik, mivel a backup-ról is tudunk saját magunknak gondoskodni. Persze Github-ról sem fog csak úgy eltűnni, de ott ténylegesen nem tudjuk ezt befolyásolni, ott valaki másra, jelen esetben a Microsoft-ra vagyunk utalva. Mivel selfhosted, így a user managementen kívül még a hálózati korlátozásokkal, és tűzfalakkal is tudjuk menedzselni, hogy ki férhet hozzá a kódrendszerünkhöz, például a már korábban említett VPN-es megoldás. Néhány cégnél még az is lehet előírás, hogy meg van szabva, hogy

fizikailag/földrajzilag hol kell tárolni kódbázist, ez még egy érv a Gitlab mellett. A CI/CD-ről is volt már szó, erre is van megoldása mind két szolgáltatónak, viszont itt is az a tapasztalat, hogy a Gitlab CI/CD-hez képest a Github Actions még gyerekcipőben jár, nem lehet olyan jó és customizálható és újrafelhasználható pipeline-okat írni, mint a Gitlab-on.



5.11. ábra CI Tool-ok összehasonlítása

Forrás: The Forrester Wave: Continuous Integration Tools, Q3 2017

Az alkalmazásunk szempontjából azért van erre szükség, hogy tudjunk commitolni, pusholni a kódot, mindezt verziókezelve, tudjunk brancheket kezelni, mergelni őket, na meg persze futtatni a pipeline-okat. Ezekről a következőkben lesz szó.

### 5.2.2 Gitlab runner

Ez egy plusz szolgáltatás a Gitlabhoz, ami a különböző pipeline-ok futtatására hivatott. Lehet ugyan azon a gépen is, amelyiken a Gitlab fut, de általában ez külön gépre szokott kerülni, az erőforrások hatékonyabb elosztása érdekében. Alapvetően úgy működik, hogy ő megy ez a Gitlabhoz, lekérdezi, hogy van-e épp aktuális job, amit le kéne futtatni. Van lehetőség shell executor-os runnert futtatni, vagy docker alapút. A különbség annyi, hogy a shell-esnél ténylegesen az adott gépen futnak le a különböző parancsok, míg a docker executor behoz erre a célra is egy adott konténert, itt mi adhatunk meg image-t és tag-et is, hogy milyen docker image-n és melyik verzióján szeretnénk futtatni az adott job-ot. A második sokkal hatékonyabb, ugyanis a gépet se szemeteli tele, és ahogy lefut a job, törlődik a konténer is, így nem kell még az adott job végén „feltakarítanunk” magunk után. Emellett mint már megtudtuk a docker imagekről korábban, itt már lehetnek különböző előre telepített függőségeink is, amik gyorsítják a munkafolyamatot. Különböző tag-eket is rendelhetünk az adott job-okhoz, hogy melyik runner-en szeretnénk futtatni, ha esetleg több lenne, vagy többféle is, csatlakoztatva a Gitlab-unokhoz. Az alapbeállítás, az az, hogy egy runner egyszerre csak egy job-ot képes futtatni, ez általában shell executor-os runnereknél így is van hagyva, viszont docker-esnél ezt feljebb lehet venni, hogy konkurensen többet is tudjon futtatni egyszerre, persze amíg bírja erőforrással. Nagy cégeknél és nagyon elosztott rendszereknél arra is van lehetőség, hogy például AWS-en teljesen autoscaling Gitlab Runner Stacket behozzunk, így ahogy egyre több és egyre erőforrásigényesebb pipeline-okat kell futtatunk, magától behoz egyre több runnert, ugyanígy, ha már kipörögtek, akkor magától lecsökkenti a számukat.

### 5.2.3 Gitlab CI/CD

Na de lássuk, mik is ezek a pipeline-ok, amiket a Gitlab Runerek futtatnak nekünk. Az alap felállítás, az az, hogy a repository -kban található [.gitlab-ci.yml](#) automatikusan triggerelődik, amikor új commit érkezik. A teljes munkafolyamat a pipeline, ezek vannak különböző stage-ekre bontva, és az elemi egységek lesznek a job-ok. Itt szintén yml leíró nyelven lehet megadni, hogy milyen munkaegységek fussanak le ilyenkor. Szerteágazó feltételrendszert lehet kialakítani, például, hogy egy adott job, csak egy adott branch-re való commitoláskor fusson le, vagy csak ha egy adott mappán belül történt változás, vagy hogy teljesen automatikusan fusson le, vagy manuálisan triggerelni kelljen. Van még lehetőség különböző függőségeket is felállítani, hogy milyen sorrendben fussanak le, vagy hogy egy job csak akkor futthat le, ha egy másik hiba nélkül lefutott.

Itt is van lehetőség változók használatára, így itt is tudunk teljesen újra felhasználható blokkokat írni. Változók lehetnek leírva magában a yml-ban, vagy lehetnek különböző CI/CD változóink is, ez a Gitlabon a Settings->CI/CD->Variables alatt érhető el.

**Variables** Collapse

Variables store information, like passwords and secret keys, that you can use in job scripts. [Learn more.](#)

Variables can be:

- Protected:** Only exposed to protected branches or tags.
- Masked:** Hidden in job logs. Must match masking requirements. [Learn more.](#)

Environment variables are configured by your administrator to be **protected** by default.

Type	↑ Key	Value	Protected	Masked	Environments	
Variable	DEV_HOST	*****	×	×	All (default)	
Variable	ENV_DEV	*****	×	×	All (default)	
Variable	ENV_PRD	*****	×	×	All (default)	
Variable	PRD_HOST	*****	×	×	All (default)	
Variable	SSH_KEY	*****	×	×	All (default)	

Add variable Reveal values

### 5.12. ábra Gitlab CI/CD Variables példa

Itt általában a szenzitívebb adatokat szokás tárolni, amiknek nem szabad belekerülnie még a repository-ba sem (lásd 5.12. ábra, SSH\_KEY), vagy csak szimplán olyan változókat is, amiket, amikor változtatunk, nem akarunk emiatt fölösleges commit-okat a repo-ba. Ezeket csak, akik Project Memberként hozzá vannak adva, megfelelő jogosultsággal látják, tudják szerkeszteni, még nyilvános projekteknél is.

Mind a Frontendnél, mind a Backendnél nagyon hasonló a pipeline. Van két stage-ünk, a build és a deploy. Tag-nek be van rakva az imagebuild, ami azért szükséges, mint ahogy már a Gitlab Runner részben láttuk, hogy egy specifikus runnerhez, jelen esetben a docker executor-os runnerhez menjen, ami a Gitlab Runner VM-en fut. Külön van szedve, hogy milyen jobok futnak le dev és prd (production) branch-ekre való commit-kor. Itt igazából annyi lesz a különbség, hogy dev branchen teljesen automatikusan végig fut a pipeline, ellenben prod-on manuálisan kell triggerelni, hogy tényleg ki menjen-e az új release.

A Build Stage-ben annyi történik, hogy belép a konténer registry-be, ami jelen esetben a Gitlab sajátja, lebuildeli a Dockerfile-unk alapján az új image-et, méghozzá két példányban, az egyik kap egy rövidített commit hash tag-et, a másik meg dev-et vagy prd-t a branch-nek megfelelően. Erre



azért van szükség, hogy így könnyebb visszaállni egy régebbi tag-re, ha az új release-ben valami mégse lenne jó. Erre van külön Gitlab policy beállítva, hogy meddig őrizze meg ezeket a tag-eket, mivelhogy a tárhely sem végtelen azért. Jelen esetben az 5 legfrissebb image-t őrzi meg ami dev, vagy prod tag-et kapott, és töröl minden 30 napnál régebbit. Majd fel is pusholja ezt a lebuildelt két fel-tag-elt image-t a registrybe.

A Deploy Stage-ben pedig be SSH-zik az adott gépre, felmásolja a `.deploy` mappában található docker-compose.yml-t, és a CI/CD változóból kinyert `.env` fájlt, majd belép a konténer registry-be, le pull-olja az új image-t, ami az előző Stage-ben lebuildelt dev, vagy prd tag-elt image lesz, és elindítja a konténert. Így nagyon minimális vagy nulla down-time után már újra fut is az alkalmazásunk, már egy frissebb verziója.

#### 5.2.4 Database Backup&Restore

Mind a dev, mind a prod környezetben CronTab-ban ütemezve futnak az adatbázis backup-olására szolgáló bash scriptek. Ennek a gyakorisága könnyen állítható, jelen esetben napi 2x fut le, reggel négykor, és délután ötkor. Ténylegesen éles környezetben ez persze jóval gyakoribb szokott lenni.

```
# Cronab
0 4,17 * * * sudo bash /home/ubuntu/backup.sh >> /home/ubuntu/cronlog/daily_`date
+%\Y%\m%\d%\H%\M%\S`.log 2>&1 ; echo "pushgateway_return_code $?" | curl -silent --
data-binary @- https://push.monitoring.benidiploma.hu/metrics/job/{{hostname}}-
db_backup > /dev/null
```

```
# backup.sh
#!/bin/bash

date=`date +"%Y-%m-%d_%H-%M-%S"`
DIR=/mnt/nfs_clientshare/{{hostname}}

mkdir -p $DIR/backup
mkdir -p $DIR/backup/logs
mkdir -p $DIR/backup/sqls

logfold=$DIR/backup/logs
sqlfold=$DIR/backup/sqls
logfile=backup_${date}.log
```

```

echo $date >> $logfold/$logfile
sqlfile=dump_$(date +%Y-%m-%d_%H-%M-%S).sql
docker exec -t diplomamunka-backend_db_1 pg_dumpall -c -U todo_backend >
$sqlfold/$sqlfile

find $logfold -type f -mtime +7 -name '*.log' -execdir rm -rf -- '{}' \; >>
$logfold/$logfile

find $sqlfold -type f -mtime +7 -name '*.sql' -execdir rm -rf -- '{}' \; >>
$logfold/$logfile

```

A script (a fentebb látható *backup.sh*) annyit csinál, hogy a felcsatolt NFS-en megcsinálja magának a géphez tartozó mappákat, mind az sql fájlok tárolására, mind a log-ok tárolására (ha nem lenne) ki dump-olja az egész PostgreSQL adatbázist, megfelelő időbélyegzővel, majd törli a 7 napnál régebbi fájlokat, hogy ne foglalják fölöslegesen a helyet. Éles környezetben az a bevált szokás, hogy pár napig az adott gépen lokálisan is eltároljuk, és legalább 30 napig pedig valamilyen külső helyen. Ha valamilyen hibakóddal térne vissza a Cronjob, akkor pedig a AlertManager -en keresztül küld értesítést Slacken. A logokból pedig kiderül, hogy mi is volt az, ami félrecsúszott.

Ha esetleg valami komolyabb hiba miatt az egész backendet újra kéne építeni, akkor ezekből a lebackup-olt sql fájlokból bármikor vissza tudjuk állítani az elvesztett adatokat.

```

cat dump_2022-05-31_17-00-01.sql | docker exec -i diplomamunka-backend_db_1 psql -U
todo_backend

```

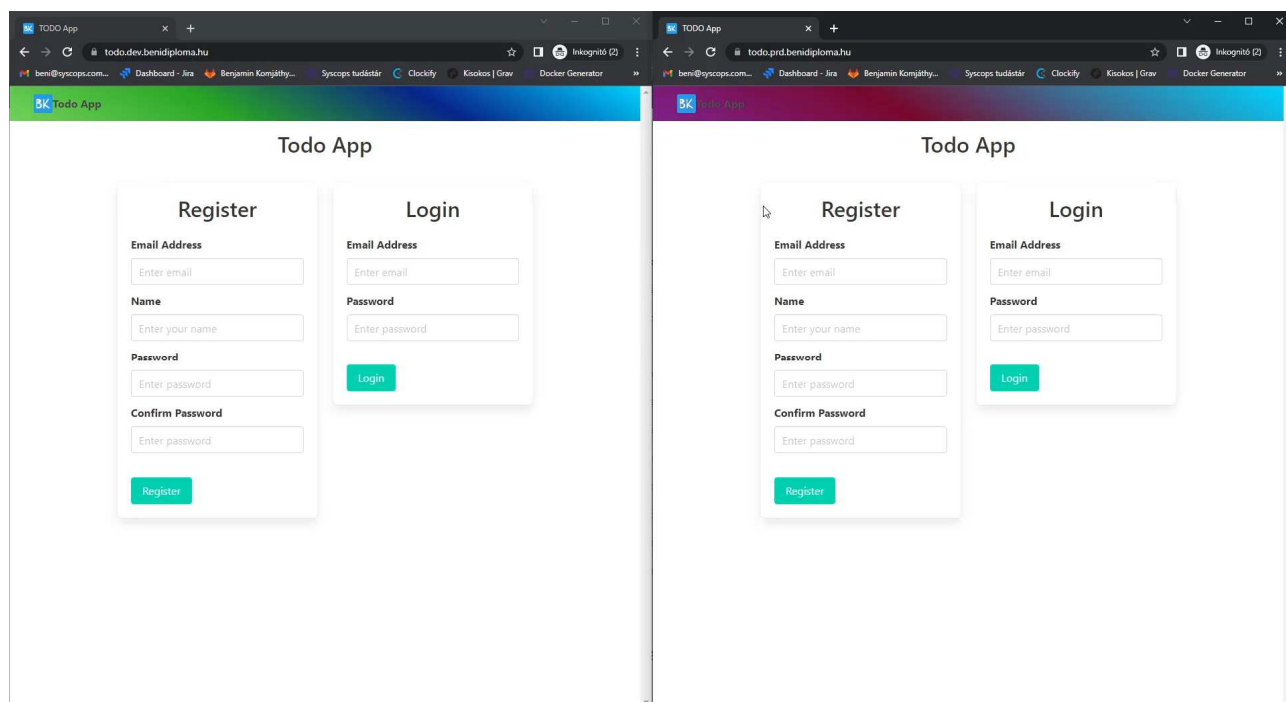
### 5.2.5 Monitoring

Az infrastruktúrához tartozó Monitoring részben ez ki lett fejtve bővebben, ami ténylegesen az alkalmazáshoz tartozik, az magának az alkalmazást futtató gépek monitorozásra (NodeExporter), a batch job-ok monitorozása (PushGateway) és maguknak az alkalmazás endpointjainak a monitorozása, hogy elérhetőek-e (BlackboxExporter), na meg persze az AlertManager, hogy kapjunk is értesítést róla, ha gond van.

## 6 PROOF OF CONCEPT

Ebben a fejezetben pedig megnézzük, hogy hogyan is nézne ez ki egy cég életében, amikor egy adott feature tegyük fel elkészült, development környezetben már megnézték, tesztelték, és hogyan kerül ez ki két kattintással production környezetre.

Ez a „feature” most csak hogy stilizáljunk, és vizuálisan is értelmezhető legyen, annyi lesz, hogy az alkalmazásunk frontendjének a Header-jét átszínezzük. Az alábbi ábrán láthatjuk a kiinduló állapotot, hogy a fejlesztői környezetre már kikerült az új feature, de élesre még nem.



6.1. ábra POC, kiinduló állapot

Amikor a projekt manager, vagy a vezetőség úgy dönt, hogy eljutott odáig a fejlesztő csapat, hogy kész vannak, és mehet a release, akkor a teendő, hogy csinálnak egy Merge Requestet, ami a dev branch-et mergeli a prd branch-be, ott még utoljára megnéznék mindent, hogy ténylegesen mi is változik kódban, és ha mindent úgy találnak, akkor megnyomják a Merge gombot (6.2. ábra).

**Open** Created 1 minute ago by Benjamin Komjáthy Maintainer

Edit Mark as draft ▾

## Dev

Overview 0 Commits 3 Pipelines 1 Changes 1

**Request to merge dev** **into prd**

The source branch is 2 commits behind the target branch

Open in Web IDE Check out branch

Pipeline #1650 passed for b74971db on dev 4 minutes ago

**Approved by you**

Squash commits

> Adds 1 commit and 1 merge commit to prd. [Modify commit messages](#)

6.2. ábra POC, Első kattintás

### Merge branch 'dev' into 'prd'

Dev

See merge request !5

2 jobs for prd in 1 minute and 9 seconds (queued for 3 seconds)

latest

af641d1d

No related merge requests found.

Pipeline Needs Jobs 2 Tests 0

Build

prd\_build

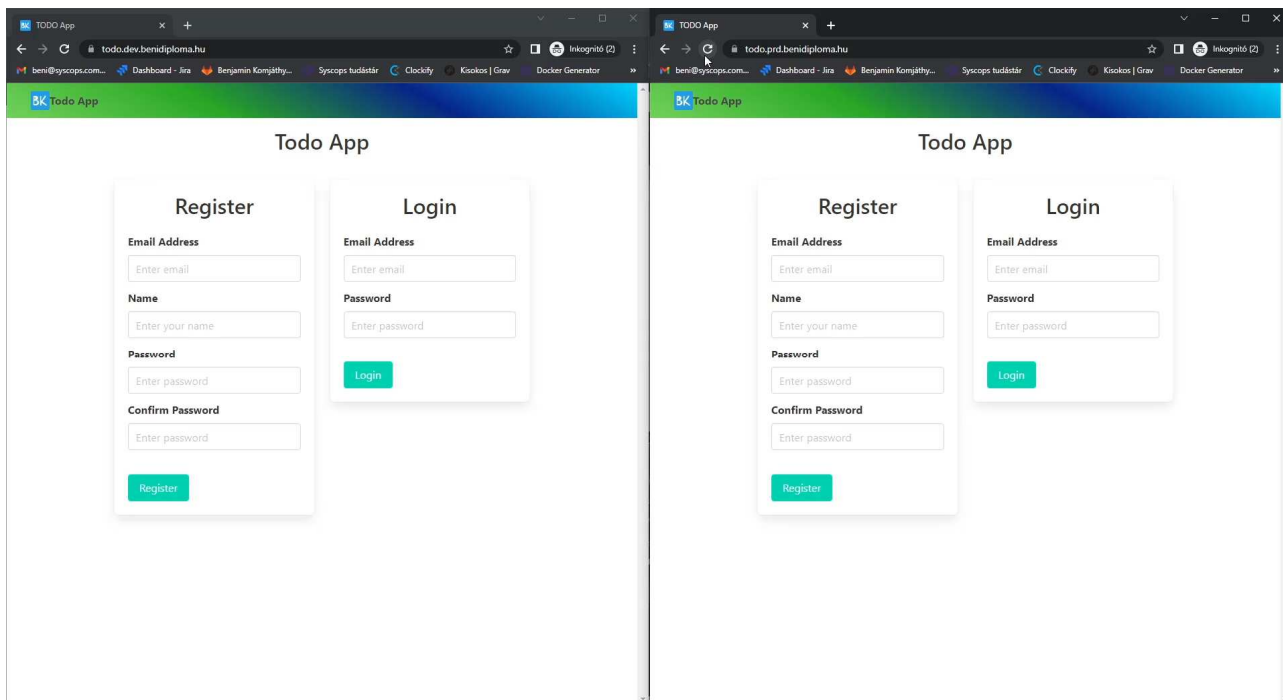
Deploy

prd\_deploy

6.3. ábra POC, Második kattintás

Ekkor triggerelődik a pipeline, mert ugyebár változott a kód a prd branch-en is. Lefut a build, elkészül az új image, és felkerül a registry-be.

Ahogy már korábban megtudtuk, itt nem megy ki teljesen automatikusan az új verzió, van még egy utolsó failsafe, hogy manuálisan még rá is kell nyomni, hogy lefusson a deploy job. (6.3. ábra)



6.4. ábra POC, vég állapot

Ezután, ha ráfrissítünk az éles környezetre, már ott is láthatjuk, hogy kikerült az új verzió (6.4. ábra). Erről a folyamatról egy videót is csináltam, ami megtekinthető az alábbi linken.

<https://youtu.be/TMCrPHP-fy8>

Ha esetleg mégis vissza akarunk állni az előző verzióra, akkor a leggyorsabban úgy tudunk, hogyha belépünk a production gépre, és átírjuk az adott docker-compose-ban a tag-et az előző commit hash-ére, mivel azt is megtudtuk, hogy egy build folyamatban 2 image is készül. Például:

```
image: registry.gitlab.syscops.dev/benjamin.komjathy/diplomamunka-frontend:98e2ca75
```

Ezután már csak újra el kell indítani a konténerünket, és már a régi verzióval fog futni. Másik lehetőség, hogy revert-eljük a Merge Requestet, és akkor újra lebuildelődik a korábbi verzió, és ki tudjuk rakni, mindezt minimális down time mellett.

## 7 ÖSSZEFOGLALÁS

A témám választásával az volt a célom, hogy egy általános webalkalmazás példáján keresztül mutassam be, hogy üzemeltetői szempontból hogyan is lesz a megírt kódból teljes automatizáció útján development és production környezet, és hogy bevezessem az olvasót a DevOps, konténerizáció, CI/CD folyamatok, felhő-infrastruktúra mérhetetlenül érdekes és még fel nem fedezett világába.

Először általánosan bemutattam, hogy mi is az a DevOps, mi is az a CI/CD, és az ide vágó szakirodalmakat. Megtudtuk, hogy miért is érdemes a cégeknek erre pénzt áldozniuk. Ezután magát az elkészült alkalmazásomat mutattam be, kitérve külön a backendre, és a frontendre. Az is bemutatásra került, hogy ha valaki szeretné, hogyan tudná elindítani magának lokálisan. Ezután megnéztük az infrastruktúrát, hogyan épült fel, milyen komponensekből áll, mik a szerepük, hogyan is jöttek létre ezek a virtuális gépek automatikusan (Terraform), hogy készítettem elő arra őket, hogy ellássák a feladatukat (Ansible). Magukat a különböző szolgáltatásokat is bemutattam. Betekintést nyertünk a konténerizáció, és a rendszermonitorozás rejtelseibe. Kitértem külön az alkalmazás üzemeltetésére is, hogy hogyan tároljuk a kódot, hogyan futtatunk Gitlab specifikusan CI/CD pipeline-okat, és hogyan backupoljuk a felhasználók adatait. Végezetül Proof of Concept formájában bemutattam, hogy tényleg működik ez az egész. Megnéztük, hogy hogyan release-eljük az alkalmazásunk új verzióját Production-re két kattintással.

Én úgy gondolom, hogy minden a Bevezetésben kitűzött célt elértem, sikerült mindent bemutatni, amit szerettem volna. Remélem, hogy a leendő fejlesztők, és cégvezetők, akik ezt olvasták, mostmár teljesen máshogy gondolkodnak az üzemeltetői oldalról, és néhány olvasó hátha kedvet kapott ilyen irányban elhelyezkedni, ha már egyáltalán megtudta, hogy van ilyen lehetőség is az IT-n belül.

Külön köszönetet szeretnék mondani Nagy Dánielnek a SysCops CEO-jának, hogy dolgozatom készítése közben használhattam a céges infrastruktúrát saját célokra, így egyszerűbbé és költséghatékonyabbá téve a munkát. Továbbá szeretnék még köszönetet mondani Dr. Vágner Anikó Szilviának a Debreceni Egyetem egyetemi adjunktusának, azért, mert partner volt abban, hogy egyedi diplomamunka témát választhassak, vállalta a témavezetői szerepet és végig támogatott a közös munka során.

## 8 IRODALOMJEGYZÉK

1. Brian Brazil (2018): Prometheus: Up & Running, O'Reilly Media, Sebastopol
2. Daniel Oh, James Freeman, Fabio Alessandro Locati (2020): Practical Ansible 2, Packt, Birmingham
3. Gene Kim, Jez Humble, Patrick Debois, John Willis, and Nicole Forsgren (2021): The DevOps Handbook, IT Revolution Press, Portland
4. ITIL Foundation 4 Edition (2019), AXELOS, London
5. Jennifer Davis and Katherine Daniels (2016): Effective DevOps, O'Reilly Media, Sebastopol
6. Jez Humble and David Farley (2011): Continuous Delivery, Addison-Wesley, Boston
7. Justin Domingus and John Arundel (2022): Cloud Native DevOps with Kubernetes, O'Reilly Media, Sebastopol
8. Navin Sabharwal, Sarvesh Pandey, Piyush Pandey (2021): Infrastructure-as-Code Automation Using Terraform, Packer, Vault, Nomad and Consul: Hands-on Deployment, Configuration, and Best Practices, Apress Media, New York
9. Sachin Srivastava (2021): DevOps with AWS
10. Sean P. Kane and Karl Matthias (2018): Docker: Up & Running, O'Reilly Media, Sebastopol
11. Stephen Fleming (2019): The DevOps Engineer's Career Guide

## 9 INTERNETES FORRÁSOK

1. <https://www.terraform.io/>
2. <https://www.ansible.com/>
3. <https://prometheus.io/docs/introduction/overview/>
4. <https://fastapi.tiangolo.com/>
5. <https://doc.traefik.io/traefik/>
6. <https://reactjs.org/>
7. <https://docs.gitlab.com/>



## 10 MELLÉKLETEK

A diplomamunkában a <https://gitlab.syscops.dev/benjamin.komjathy> alatt megjelent linkeket feltöltöttem a hivatalos <https://gitlab.com> -ra is, ha valaki valamilyen későbbi időpontban olvassa, a forráskódokat akkor is meg tudja nézni.

- <https://gitlab.com/benjamin.komjathy/diplomamunka-backend>
- <https://gitlab.com/benjamin.komjathy/diplomamunka-frontend>
- <https://gitlab.com/benjamin.komjathy/diplomamunka-terraform>
- <https://gitlab.com/benjamin.komjathy/diplomamunka-ansible>

A Redocs tool-lal legenerált API dokumentáció elérhető itt:  
[https://drive.google.com/file/d/1OxDL6PIT\\_rblokLpA5hbZakZ0a-zTwbp/view?usp=sharing](https://drive.google.com/file/d/1OxDL6PIT_rblokLpA5hbZakZ0a-zTwbp/view?usp=sharing)