ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

**NEUMANN JÁNOS
INFORMATIKAI KAR**

SZAKDOLGOZAT

OE-NIK Hallgató neve: **Ferenczi Márton**
2021 Hallgató törzskönyvi száma: T/004937/FI12904/N

Óbudai Egyetem
Neumann János Informatikai Kar
Alkalmazott Informatikai Intézet

SZAKDOLGOZAT

FELADATLAP

Hallgató neve: **Ferenczi Márton**
Törzskönyvi száma: T/004937/FI12904/N
Neptun kódja: [redacted]

A dolgozat címe:

Magánegészségügyi szolgáltatók éves adatai alapján történő következő éves ajánlatkészítő alkalmazás

Tender generator application based on healthcare providers annual data statistics

Intézményi konzulens: Dr. Fleiner Rita
Külső konzulens: Nagy András

Beadási határidő: 2022. május 15.

A záróvizsga tárgyai: Számítógép architektúrák
Big Data és Üzleti Intelligencia specializáció

A feladat

Egy olyan applikáció elkészítése, amely egészségügyi adatok alapján képes személyreszabott ajánlatot készíteni egy leendő ügyfél számára. Figyelembe kell vennie az illető korát, alapbetegségeit, családi hajlamokat, életmódot. Ezek alapján kalkulál egy lehetséges megbetegedési listát. Maga az alkalmazás egy Angular webes felülettel, Java Spring Boot backenddel rendelkezik, amely docker konténerizációt használ. Az egyes számítási részegységek microservicekként futnak a dockeren belül.

A dolgozatnak tartalmaznia kell:

- A feladat leírását
- A megvalósítandó rendszer tervét
- Az implementáció folyamatát
- Egészségügyi adatokat
- Tesztelés terveit és eredményeit
- Továbbfejlesztési lehetőségeket

ÓBUDAI EGYETEM * Neumann János Informatikai Kar * 1. Budapest

P.H.

Dr. Fleiner Rita
intézetigazgató

A szakdolgozat elévülésének határideje: **2024. május 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

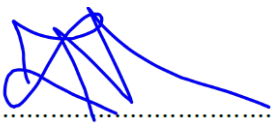
……..
külső konzulens

Fleiner Rita
……
intézményi konzulens

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2022. május 05.



.....

hallgató aláírása

KONZULTÁCIÓS NAPLÓ

Hallgató neve:	Neptun Kód:	Tagozat:
Ferenczi Márton	[redacted]	esti

Telefon: [redacted]

Levelezési cím (pl.: lakcím): [redacted]

Diplomamunka címe magyarul:
Magánegészségügyi szolgáltatók éves adatai alapján történő következő éves ajánlatkészítő alkalmazás

Diplomamunka címe angolul:
Tender generator application based on healthcare providers annual data statistics

Intézményi konzulens:	Külső konzulens:
Dr. Fleiner Rita Dominika	Nagy András

Kérjük, hogy az adatokat nyomtatott nagy betűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	**2021.09.31**	**Követelmények megfogalmazása**	
2.	**2021.10.06**	**Első vázlat revízió**	
3.	**2021.11.11**	**Tartalom ellenőrzés**	
4.	**2021.12.09**	**Végső változtatások ellenőrzése**	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. tantárgy követelményeit teljesítette, beszámolóra bocsátható.

Budapest, 2021.12.09.

.....

intézményi konzulens

KONZULTÁCIÓS NAPLÓ

Hallgató neve:	Neptun Kód:	Tagozat:
Ferenczi Márton	[redacted]	esti

Telefon: [redacted]

Levelezési cím (pl.: lakcím): [redacted]

Diplomamunka címe magyarul:
Magánegészségügyi szolgáltatók éves adatai alapján történő következő éves ajánlatkészítő alkalmazás

Diplomamunka címe angolul:
Tender generator application based on healthcare providers annual data statistics

Intézményi konzulens:	Külső konzulens:
Dr. Fleiner Rita Dominika	Nagy András

Kérjük, hogy az adatokat nyomtatott nagy betűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2022.02.28	Adatok/Táblák átnézése	Fleiner Rita
2.	2022.03.14	A szoftver működésével szemben támasztott követelmények, folyamatok	Fleiner Rita
3.	2022.04.20	Webes felület kialakításának véleményezése	Fleiner Rita
4.	2022.05.05	A szakdolgozat átnézése	Fleiner Rita

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat II. tantárgy követelményeit teljesítette, védésre bocsátható.

Budapest, 2022. május 05.

Fleiner Rita

.....

intézményi konzulens

Tartalomjegyzék

1 Bevezetés 1
2 Célok meghatározása 2
3 Irodalomkutatás 3
3.1 Heart Disease 3
3.2 BiliScreen 3
3.3 ResApp 4
4 Adatok 5
4.1 Beviteli, független adatok 5
4.1.1 Input adatok 6
4.1.2 Target adatok 6
4.2 Adatok beszerzése 7
4.2.1 Generált Adatok 7
4.2.2 Önkéntesekkel Online Kutatás 7
4.2.3 Konklúzió 7
5 Technológiák 8
5.1 PostgreSQL 8
5.2 Java 8
5.3 Spring Boot 8
5.4 Hibernate 9
5.5 Angular 9
5.6 Maven 9
5.7 Docker 9
5.8 Keycloak 10
6 Szoftvertechnológiai eszközök 11
6.1 Neurális háló 11
6.2 MicroService VS Monolit 14
6.3 Összegzés 17
7 Rendszerterv 18
7.1 Angular UI 19
7.2 Patient API (Spring Boot rest) 19
7.3 Illness API (Spring Boot rest) 19
7.4 Weight API (Spring Boot rest) 19
7.5 Health Status API (Spring Boot rest) 19
7.6 Machine Learn API (Spring Boot rest) 19
7.7 Adatbázis 19

7.8 Neural Processor 19
8 Adatbázis terv 21
8.1 Táblák 22
8.1.1 Active_sport_activities 22
8.1.2 Ancestor_death_causes 22
8.1.3 Ancestor_illnesses 22
8.1.4 Healthstatus 22
8.1.5 Illness 23
8.1.6 Known_illnesses 23
8.1.7 Patient data 24
8.1.8 Post_sport_activities 24
8.1.9 Potential_illness 25
8.1.10 Sport 25
8.2 Java Persistence API 25
9 Algoritmus terv 26
10 Projekt megvalósítása, fejlesztés 27
10.1 Docker-Compose, env, környezet 27
10.2 Backend, Servicek 30
10.2.1 Datastore Component 30
10.2.2 Calculator Component 36
10.2.3 Neural Processor Component 36
10.2.4 Medicare Common Component 36
10.2.5 User Service Component 37
10.2.6 SwaggerUI 37
10.3 Frontend 38
10.3.1 Mockup 39
10.3.2 About 41
10.3.3 Fill Form 41
10.3.4 Calculate 42
10.3.5 Keycloak Integráció 42
10.4 Neurális Háló 45
10.4.1 Tanítás 45
10.4.2 Kalkulációk 46
10.5 Tesztelés 47
10.5.1 Integrációs teszt 47
11 Összegzés 48

11.1 Továbbfejlesztési lehetőségek 48
12 Summary 49
12.1 Further development options 49
13 Irodalomjegyzék 50
14 Ábrajegyzék 53
15 Táblajegyzék 55

1 Bevezetés

A mai világban egyre fontosabbá vált a magánegészségügyi szolgáltatók jelenléte. Mivel nem mindenki engedheti meg magának, hogy ad-hoc jelleggel válasszon egy kórházat, akiknél elvégezteti a vizsgálatokat vagy kezelést, ezért sokan éves biztosítást kezdenek választani. Ezekhez a biztosításokhoz viszont elengedhetetlen egy olyan összetett önkórképelemzés, ami alapján az ember eldöntheti, hogy melyik biztosítási ajánlat a legmegfelelőbb számára. Sajnos a többség nincsen tisztában a saját egészségügyi állapotával ilyen mélységekben, ezért ennek megkönnyítésére hozom létre a kalkulátorral ellátott ajánlatkészítő rendszert, ami kérdőívszerű kérdéssor után tud egy jó közelítésű kalkulációt csinálni a lehetséges megbetegedésekre. Ezzel a rendszerrel nem csak a biztosítók munkáját lehet támogatni, de egy átfogó képet is kaphatnak a felhasználók az aktuális állapotukról.

A projekt három fő részre bontható. Rendelkezésre kell álljon egy online felület, ahol a felhasználó fel tudja vinni az adatait, és elküldeni a backend számára. Az adatok ezek után anonim módon mentésre kerülnek a később részletezendő, tanuló algoritmus számára. Ezzel párhuzamosan egy java microservice elvégzi a kalkulációt arra vonatkozóan, hogy milyen típusú lehetséges betegségekre számíthat a felhasználó rövid- és hosszútávon.

Kutatásom szerint csak olyan specifikus online szoftverek érhetők el, amivel bizonyos tünetegyüttesekből számítható ki az, hogy milyen betegsége van aktuálisan a felhasználónak. Valószínűleg nem általánosan elérhető alkalmazások között szerepel olyan, akár biztosítók, pénzügyi intézetek, klinikák részéről, amely egy ehhez hasonló koncepcióval “jósol”.

2 Célok meghatározása

Ezzel a rendszerrel szélesebb körben elérhetővé válna egy olyan szoftver és lehetőség, amivel az emberek monitorozni tudják saját életmódjukat. Így nagyobb figyelmet kaphat az öngondoskodás, időben felismerhetővé válnak a közemberek számára is a kezdődő betegségeik tünetei. A létrejövő anonim adatbázis pedig rengeteg egyéb kutatást könnyíthet meg, és szolgálhat ki adatokkal.

Későbbi célok közé tartozik a szoftver továbbfejlesztése, és erre a tervezés során nyitottnak kell hagyni az alkalmazást. Ezzel akár specifikálható bizonyos területekre és különböző ipari környezetekben is el lehet helyezni. Az erre nyitott alkalmazás a praktikussága mellett a SOLID elvek közül is többet teljesít, így a kódolás a tervezési alapelveknek is meg fog felelni. Ha a jövőben több fejlesztőnek kellene csatlakoznia a projekthez, akkor a kód jól érthető, olvasható, és jól dokumentáltsága hozzá fog járulni a gyors felzárkózáshoz és folytatáshoz.

A tanuló algoritmus, ami hivatott a betegségek kalkulációját elvégezni, egy kifejezetten kísérleti megoldás, hiszen erre legtöbbször meglévő tünetegyütteseket használnak, ami még a kontakt orvoslásban is az egyik elsődleges diagnosztikai forma. Érdekes megoldás lehet ez forradalmasítani az orvosdiagnosztikát, akár ilyen mikro szinten is, és nem logikai úton végezni a bemenetek kiértékelését.

3 Irodalomkutatás

Mint korábban említettem, biztosan vannak olyan szabad szemmel nem látható rendszerek a biztosító cégek mögött, amelyek hasonló megoldásokkal operálnak, viszont nem bennfentesként nem elérhetőek. Található azonban néhány online kalkulátor, amik hasonló koncepcióval lettek elkészítve. Ezeknek a kalkulátoroknak a többsége viszont tünetek alapján készít elemzést, és nem "jóslást" hajt végre.

Az ilyen prediktív rendszer létrehozásához elsődleges problémát a beteginformációk nagy mennyiségű beszerzése jelenti. Ezek ugyanis olyan szenzitív információk, amiket még anonim módon is igen nehezek beszerezni, illetve a már beszerzett adatokat is igen borsos áron szolgáltatják ki. Egy jól működő algoritmus betanításához pedig nagyon nagy mennyiségű adatra van szükség, így az online kitölthető kérdőíves merítések nem elégséges mennyiségű információt szolgáltatnak.

A digitális eszközök fejlődésével egyre elterjedtebbek az olyan applikációk, amelyek az okosórák, mérlegek és hasonló kiegészítőkből származó adatok alapján készítenek kalkulációkat. Ezeknek a szoftvereknek a többsége valamilyen területre specializálódik a mérhető adatok alapján. Ilyenek például a szívbetegségek, a mentális problémák, illetve az epilepsziás rohamok megjóslására szolgálnak.

3.1 Heart Disease

A Huntington Medical Research Institute nonprofit alkalmazott orvosi kutatószervezet egy projektjeként fejlesztett applikáció a kardiovaszkuláris egészség monitorozására, a Caltech egyetem mérnökei által. Az app a szívverés monitorozásával készíti a számításokat (LVEF). A felhasználó a nyaki artériához tartja a telefont, ahol a véráram mérhető az érfal tágulásából és összehúzódásából. Kísérletek bizonyították, hogy ezzel az alkalmazással hasonló eredmények érhetők el, mint egy echocardiographiai vizsgálat során mérnek. [1]

3.2 BiliScreen

A hasnyálmirigyrák az egyik legnehezebben felismerhető rákos megbetegedés. Jelenleg az egyetlen biztos diagnosztikai formája egy igen hosszadalmas és költséges vérvizsgálat. Ez viszont hamarosan megváltozhat, hiszen a Washington egyetem Ubiquitous Computing Lab-jában fejlesztettek egy alkalmazást, ami megkönnyíti a korai felismerést. Az egyik első tünetek közé tartozik a jaundice megjelenése, amit a vérben található bilirubin szint emelkedése okoz. Amikor ez már a szabad szem számára is látható mértékű sárgaságot okoz, az a rák előrehaladott stádiumát jelzi. A fejlesztés egy olyan

képfelismerő algoritmust tartalmaz, ami képes felismerni a legkisebb bilirubin szintemelkedést is, így könnyítve a korai diagnosztikát. A klinikai vizsgálatok bizonyították, hogy képes felismerni a betegséget az esetek 89.7 százalékában. [2]

3.3 ResApp

A Queenslandi egyetem professzora Udantha Abeyratne fejlesztett egy olyan szoftvert, ami segít felismerni a különböző légzőszervi megbetegedéseket, mint például az asthma, vagy a pneumonia, csupán a köhögés elemzéséből. A klasszikus orvoslásban a légzőszervi megbetegedések diagnosztikáját sztetoszkóp segítségével végzik az orvosok, ám így sok információ elveszik a köhögés magas frekvenciás tartományából, hiszen a lehallgatáshoz először a mellkas izomszövetein kell átjutnia a hangnak. Ezt kiküszöbölve a fejlesztés egy a gépi tanuláson alapuló diagnosztikával elemzi a köhögést és lélegzetvételeket. A mérések után a felvételeket egy nagy adatbázisban lévő adatokkal vetik össze, és az algoritmus pontosan képes ezek alapján felismerni a páciens légzőrendszeri állapotát. [3]

4 Adatok

Mint fentebb említettem, az ilyen szenzitív adatok beszerzése igen nehézkes folyamat, és korábbi tapasztalataim is azt mutatják, hogy cégek még anonim módon sem hajlandóak kiszolgáltatni őket. Két járható út marad így a megfelelő mennyiségű információ beszerzésére.

4.1 Beviteli, független adatok

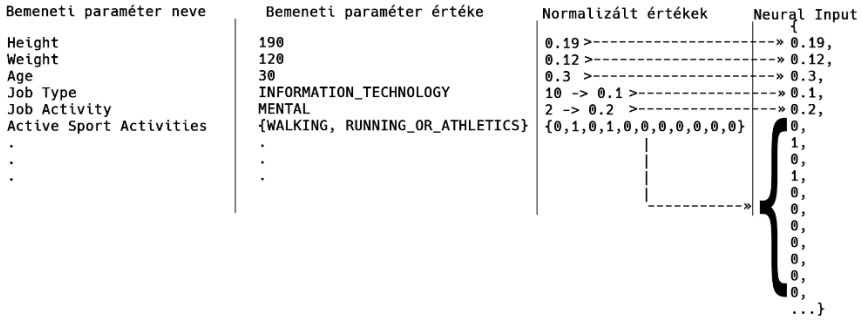
Olyan perszonális, szenzitív adatokra van szükség mind a kalkulációhoz, mind a tanításhoz amely a korábban megszokott orvosi kérdőívekhez nagyon hasonló módon érdeklődik a következő adatokról; a beteg kora, neme, mozgási szokásai, felmenők betegségei, felmenők halálának ismert okai, stb… (bővebben: *1. tábla: PatientData DTO*).
Ezek a betegadatok képzik a független bemeneti változók összességét amik alapján készül egy kalkuláció arra, hogy milyen betegségekre nézve van magasabb kockázatnak kitéve a kitöltő. Ezt egy százalékos megjelenítéses oldalon tekintheti meg a regisztrált felhasználó, ami alapján egy iránymutatást láthat az egészségi állapotáról.

A bemeneti adatok 3 módon kerülnek bevitelre:

- Első bemeneti típus amikor egy számadat kerül felvitelre a felületre. Ebben az esetben például a súlyadatoknál ami egy 10-es, vagy 100-as lépték (120kg) az adat normalizálásra kerül, így a 0-1 tartományba kerül át.
- Második módszerként enum értéket állít be a felhasználó, ami egy bemeneti cellás esetben (például nem) az enum számértékét veszi fel (female = 1, male = 2, other = 3) és ezt az értéket normalizálja 0-1 közé, amiből az eredmény female esetben 0.1 lesz.
- Az utolsó módszer amikor egy listából több enum elemet választhat ki a felhasználó (például aktív sportok), amikor egy olyan adathalmazt képzünk, amiben a sportok mellett 1-es szerepel, ha aktív és 0 amikor nem aktív.

4.1.1 Input adatok

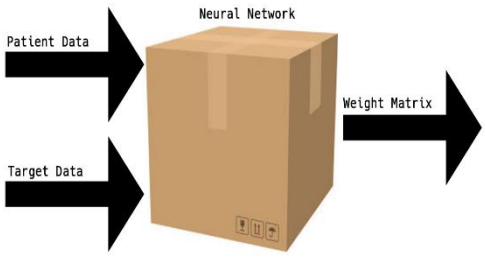
Ezeket a bemeneti adatokat a neurális háló bemeneti paramétereinél olyan módon alakítjuk át, hogy egy 100 elemű lista képződjön 0-1 közötti értékekkel. Minden adat a bemeneti betegadatokból képződik. A transzformációk a korábbiakban említett átalakításokkal és normalizálással képződnek.



1. ábra: Input adatok transzformációja

4.1.2 Target adatok

A tanításhoz céladatokat is meg kell adni, ami ebben az esetben egy olyan tömb lesz, ami 0-1 közötti értéket tartalmaz. Ezek az adatok úgy képződnek, hogy az adott betegségcsoporthoz tartozni fog egy szám 0 és 1 között, ami a százalékos esélyét határozza meg a betegségnek. Egy daganatos halálozás esetén például ez a szám az 1-es lesz a daganatos megbetegedések indexedig elemében. Tehát egy teljes target data set úgy fog kinézni, hogy az összes betegségcsoportból képződik egy list (blood, cancer, carddiovascul…) és ezekhez hozzárendelődik egy érték, halál esetén 1, egyébként pedig 0. Majd az így képződött {0, 1, 0, 0, 0, 1, 0, …} gyűjtemény lesz a target data a neurális háló tanítása során. A háló fejlődése érdekében egy „természetes tanulás” is bevezetésre kerül, amikor a bemeneti betegadatokból a Known Illnesses listát fogja meg a háló és



2. ábra: Neurális hálózat bemenetei és kimenetei

használja fel úgy, mint a korábban említett Target adatot, hiszen a szerkezetük megegyezik, így könnyű az átalakítás.

4.2 Adatok beszerzése

4.2.1 Generált Adatok

A komplex adatstruktúra és széleskörű beteginformációk miatt a legegyszerűbb megoldás az adatok generálása. Ezekkel egy alap állapotba beállítható az algoritmus, ami alapján a prediktív rendszer már működésbe léphet. A későbbi bemeneti adatok alapján pedig tovább fejlődhet az algoritmus és képes pontosabb eredményt adni.

4.2.2 Önkéntesekkel Online Kutatás

Ezzel a módszerrel ugyan viszonylag kevés adatot lehet meríteni, viszont sokkal relevánsabbak lesznek az eredmények. Egy célzottan összeállított kérdőívvel a megfelelő helyekre eljuttatva azt egy jó alap adatbázist össze lehet állítani a kiindulási alaphoz. A megfelelő pontosság eléréséhez minimum több ezer kitöltőre lenne szükség, viszont jó esetben is csak több százat lehet gyűjteni az ilyen típusú merítéssel.

4.2.3 Konklúzió

A két adatszerzési módszer egymással párhuzamosan is elvégezhető és nem kizáróak. A fejlesztés megkezdése előtt egy célzott kérdőív elindításával kezdődhet az adatgyűjtés és az eredmények fényében döntést kell hozni, hogy esetleg elégséges-e a beérkező mennyiség, ki kell egészíteni generátumokkal, vagy teljesen generált adatokkal kell elvégezni az algoritmus tanítását. Ebből következtethető, hogy ez a projekt nem képez egy végleges és jól használható megoldást, csupán alapot képez egy továbbfejlesztett, jobban tanított programnak. Az éles futtatáshoz egy cloudban elhelyezett rendszert lehetne telepíteni ami az első időszakában még csak tanuló módban futna és gyűjtené az adatokat. A megfelelő mennyiségű bemeneti információk elérését követően nyílna lehetőség az éles futtatáshoz, mikor meggyőződve a helyes futásról, már nem lennének félrevezetőek az információk a felhasználók számára.

5 Technológiák

Ebben a fejezetben azokat az eszközöket és folyamatokat fogom részletezni, amiket tervezek felhasználni a szoftver megvalósítása során.

5.1 PostgreSQL

A legszélesebb körben elterjedt és használt nyílt forráskódú adatbázis a PostgreSQL. Rengeteg lehetőséget biztosít akár a szokásos relációs adatbázis felhasználási területen, akár "felokosított" változatában egy gráf adatbázisként, vagy idősorosan is. Mivel az Ipar 4.0 területen is előszeretettel használják, így logikusnak tűnt a választás.

Adatintegritás szempontjából is tökéletes választás lehet, hiszen sok más adatforrásból képes importálni adatot, ami segíthet a későbbi nagy adatmennyiségek befogadására utóprocesszálás nélkül. (Oracle, MySQL, MongoDB, Redis, Neo4j…)

Támogatja a legtöbb adattípust, mint például a document, primitives, geometry, structure stb. [4]

5.2 Java

Mint a standard enterprise OOP fejlesztési nyelv, nem nagyon volt kérdéses az, hogy a java-át válasszam. A rengeteg meglévő lib és modul pedig még többet ad hozzá a pro oldalhoz. Széleskörű felhasználása, robosztussága, jól dokumentáltsága az egyik legelterjedtebb nyelvvé teszi. A multiplatform támogatottság, konténerizáció pedig további könnyebbségeket jelent az applikáció fejlesztéséhez, így a dockeres futtatás is könnyen, pár kattintással elindítható és kihelyezhető bármilyen szerverre és környezetbe. [5]

5.3 Spring Boot

A Spring Boot egy Java fejlesztők számára készített micro framework amivel a fejlesztők könnyen készíthetnek kevés konfigurációval alkalmazásokat. Rengeteg függőség rendelhető hozzá, amivel gyakorlatilag plug and play módon építhető igen komplex alkalmazás is, ilyen például a Spring Kafka, LDAP, Security stb.

Az alkalmazások szerkezetét jelentős módon megkönnyítik a beépített annotációk és eszközök a különféle programozási paradigmák megvalósításához. Ilyen például a leegyszerűsített dependency injection, webalkalmazás annotációk, request mappingek. Támogatja a JDBC frameworköt amivel leegyszerűsödik és biztonságossá válik az adatbázis kapcsolat és szerkesztés. [6]

5.4 Hibernate

Egy olyan ORM (object relational mapping) framework, ami megkönnyíti az objektum orientált domain model leképezését relációs adatbázis kapcsolatokra. Hatalmas előnyt jelent, hogy az adatbázis létrehozása és felépítése, kapcsolatok bekötése maga az alkalmazás indulásával létrejön, nem kell kézzel SQL kódot és scripteket írni hozzá. Ezzel és sok más eszközével lecsökkenti a fejlesztési időt, leveszi a terhet a fejlesztők válláról a perzisztálás manuális kezelésének. [7]

5.5 Angular

A modern Typescript nyelvek zászlóshajója, ami folyamatos fejlesztés alatt áll, egyik legszélesebb körben felhasznált frontend framework. Az adatvizualizációtól kezdve webshopok és sok más reszponzív weboldalon már bizonyította a robosztusságát. A kiterjesztett HTML nyelv a már jól ismert eszköztárat szélesíti, de mégis könnyen módosíthatóvá teszi az ebben fejlesztett felületeket. Hatalmas online tudásbázis és aktív support segíti a friss fejlesztők munkáját is, illetve templatekkel teszi még gyorsabbá a fejlesztést. [8]

5.6 Maven

Egy project management toolra is szükség van összefogni a service-eket és komponenseket, amire a Gradle és a Maven a legismertebb eszköz. Korábbi tapasztalataim alapján számomra a kényelmes és ismert eszköz a Maven, így ezt az eszközt választottam ennek megvalósítására. Ezzel a toollal egy egyszerű pom.xml leíróval beimportálhatóak a különböző forrásokból származó dependenciák. Erre több dependency formátum is van, elég egy parentbe importálni a később szükséges elemeket és így a gyermek elemek mind látják ezeket. A dependenciák akár BOM formában is behúzhatóak, így eszközök csoportja egy egyszerű importálással kerül be a projektünkbe. [9]

5.7 Docker

Szükség van egy környezetre, ahol a már felépült vagy folyamatban lévő szoftvert lehet tesztelni, működtetni. Erre akár egy szervert is fel lehetne húzni, de a modern megközelítést követve a DevOps eszközökhöz érdemes nyúlni, hiszen nem generál annyi felesleges környezeti beállítást, rendszerépítést, mint egy szerveres környezet konfigurálása, kitelepítése. Ezért esett a választásom a Dockerre, hiszen ezzel egy operációs rendszer szintű virtualizációs környezetet hozunk létre. A lokálisan felépített rendszerünk egy mozdulattal telepíthető ki szerveres környezetbe is, ahol módosítás nélkül működtethetjük a már production módban futó szoftverünket. A konténerizált formában futó microserviceink könnyen menedzselhetők. [10]

5.8 Keycloak

A felhasználó azonosítására több módszer is létezik, különböző szerveroldali azonosítás, adatbázisba mentés, viszont mindegyik viszonylag komplex a beállítások és integráció szempontjából. Ennek kiküszöbölésére választottam a Keycloakot, amely egy nyílt forráskódú Identity and Access Management rendszer. A szoftver egy komplett bejelentkeztető, felhasználókezelő csomagot ajánl egy igen könnyen integrálható formában. Gyakorlatilag plug and play módon felhasználható, csak egy-két hozzáférési pont és átirányítás megadására van szükség és máris működik. Nagyon jól dokumentáltságának köszönhetően gyakorlatilag pillanatok alatt alakíthatjuk a környezetünket egy oauth2-es rendszerré. [11]

6 Szoftvertechnológiai eszközök

6.1 Neurális háló

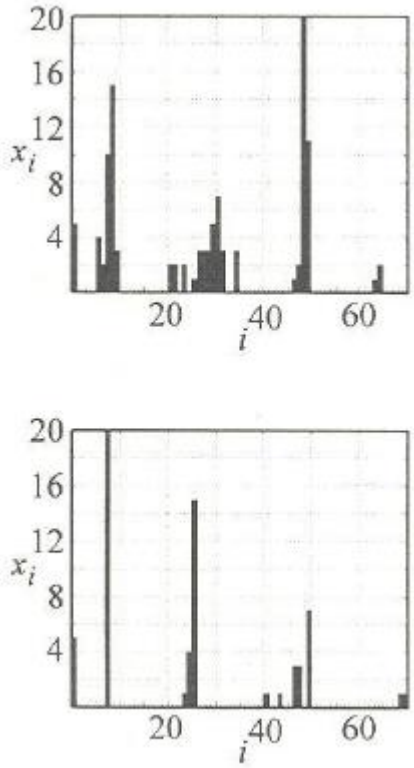
Az emberi agy leképezésével olyan programokat hozhatunk létre, amikkel könnyen felismerhetővé válnak patternek és más általános problémák az AI, machine learning és deep learning buzzwordök alatt.

Az artifical neural networkök (ANNs) node layerekből, input, hidden és output layerekből épülnek fel. Minden node, vagy artifical neuron kapcsolódik egy másikhoz és ennek van egy súlyozása és küszöbértéke. Ha bármelyik node-nak a kimenete a definiált küszöbérték alá esik, az a node aktiválódik és tovább küldi az adatot a következő rétegnek a hálóban. A neurális hálók training adatokon tudják fejleszteni pontosságukat időről időre. A feladatok átszervezése ezen algoritmusok számára akár percekre is rövidíthetik a szakértők által manuálisan akár hetekig, hónapokig történő folyamatokat. Az egyik legismertebb neurális alapokon működő eszköz a Google keresőmotorja.

Nem deduktív tudomány.

Működési elvének lényege, hogy rengeteg adat álljon rendelkezésre, hiszen adat nélkül nem tudunk mihez kezdeni, nincsenek konkrét algoritmusaink. A szabályok és képletek ismeretlenek, ha ezekkel rendelkeznénk, akkor azokat használnánk fel a probléma megoldásának megvalósításához. Ez a cél, hogy az adat alapú megismerésünket képlet alapúvá tegyük. A rengeteg empirikus tapasztalatot néhány képlet össze tudja foglalni, de ehhez fel kell ismernünk a képletet, ha van. Elképzelhető az is, hogy nem teljes a mintánk, de ezzel a módszerrel ez nem jelent problémát, ellentétben a hagyományosnak tekintett módszerekkel, mert ott amennyiben egy változó hiányzik, akkor megrekedünk. A sok összefüggő adattal viszont képesek vagyunk tanítani, súlyozást létrehozni, hiába a köztük lévő relációt nem ismerjük, megfelelő mennyiségű adatnál ez nem jelent problémát.

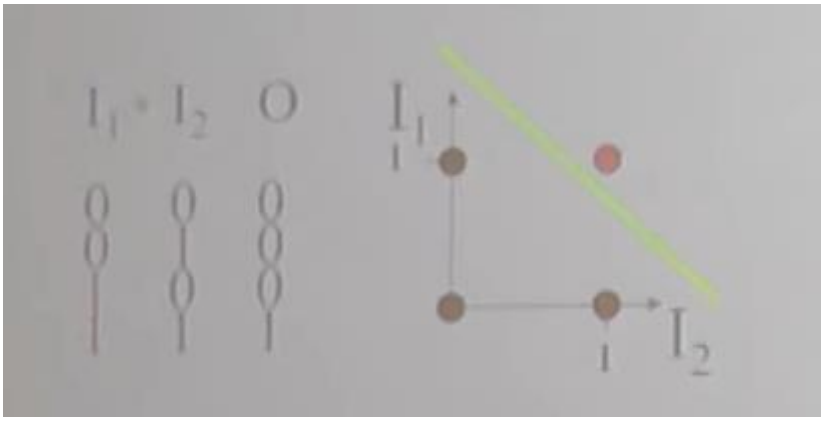
A neuronok (mesterséges) összeköttetésüket úgynevezett súlytényezőkkel képviseltetjük. Tehát, ha az egyik a másikkal össze van kötve, akkor úgy tekinthetjük, mintha egy szorzószám lenne a kettő között. Ezeket a hálózatokat nem programozzuk, hanem tanítjuk. A tárolt információk a hálózatban elosztottan szerepelnek és a súlytényezők közvetítésével jelennek meg. Ennek a következménye, hogy hibatűrő a rendszer, egy-egy súlytényező megváltoztatása tulajdonképpen nem jelent érdemi változást önmagában. Három fő tényező határozza meg: átviteli függvény, hálózati összeköttetések, tanítás.



6.1. ábra: Mérések eloszlása [12]

Alkalmazás menete:

- Összegyűjtöm a tanító adatokat
- Kitaláljuk a megfelelő paradigmát
- Hálózat jellemzőit beállítjuk
- Megállapítás, hogy jó-e, vagy nem jó (teljesítmény mérő módszer kiválasztása)
- Tanítás, tesztelés (amíg jó nem lesz)
- Validáljuk, hogy megfelelően működik



6.2. ábra: Lineárisan elválasztható minták [12]

Az első ilyen alkalmazás a Perceptron (1957) csak lineárisan elválasztható mintákra lehetett alkalmazni, tehát két nagyobb halmaz elválasztása egyértelműen működik vele. Ha már átfedés volt a dolgok között, akkor azt nem lehetett megoldani. Ez a 80-as években megváltozott, hiszen a réteges szerkezetű hálózattal, felügyelt tanulással (amikor van cél érték, ismerjük azt), ha vannak közbeeső neuronok, akkor hangolható az egész nem lineáris problémák megoldására is.

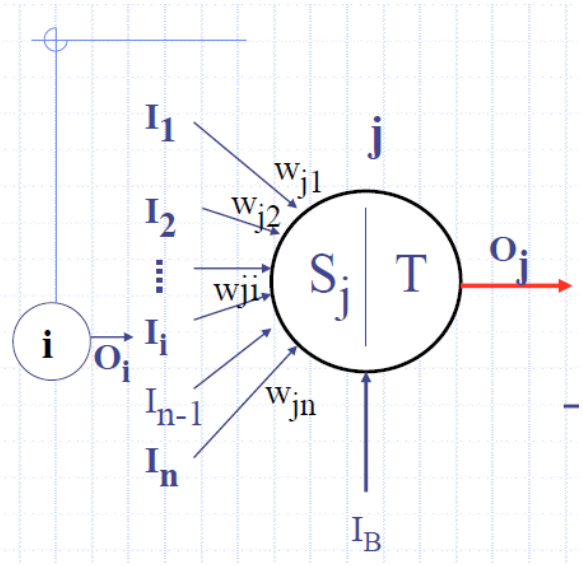
Meghatározó tényezők:
- Neuron
- Összeköttetései séma
- Tanítás

Mivel a legújabb megközelítés szerint az átviteli függvények differenciálhatóak

$$O_j = 1/(1+e^{-S_j})$$

5. ábra: Neuron függvénye [12]

ebből következik, hogy a mélyben lévő neuronokat is tudjuk tanítani. Ezt hívjuk előrecsatolt hálózatnak, ahol csak a következő szintig jutnak összeköttetések, tovább nem. Itt kettő tanítható réteg. Érdekes tény, hogy az állatvilágban gyakran előfordul az előrecsatolt, 3 rétegű idegrendszer. Ezeket a hálózatokat ábrázolhatjuk úgy, mint mátrixokat. Ebből az a következtetés vonható le, hogy a neurális hálózatok leképezése mátrximűveletekre redukálható. Ebből levonható, hogy a neuronok kimenete az egyenlő a bemeneti vektor szorozva a súlymátrixával.



6.4. ábra: Neuron [12]

A tanítás szabályai:

- Hebb szabály (wji(t)+alfa[tanítási tényező 0<=alfa<=1]*Oi *Oj -> megerősödik az összeköttetés)
- Delta szabály (ha a kimeneti értékemhez képest a célérték magasabb, akkor meg kell növelnem a súlytényezőt -> wji(t)+alfa*Oi*(Cj-Oj) ahol Cj-Oj=delta)

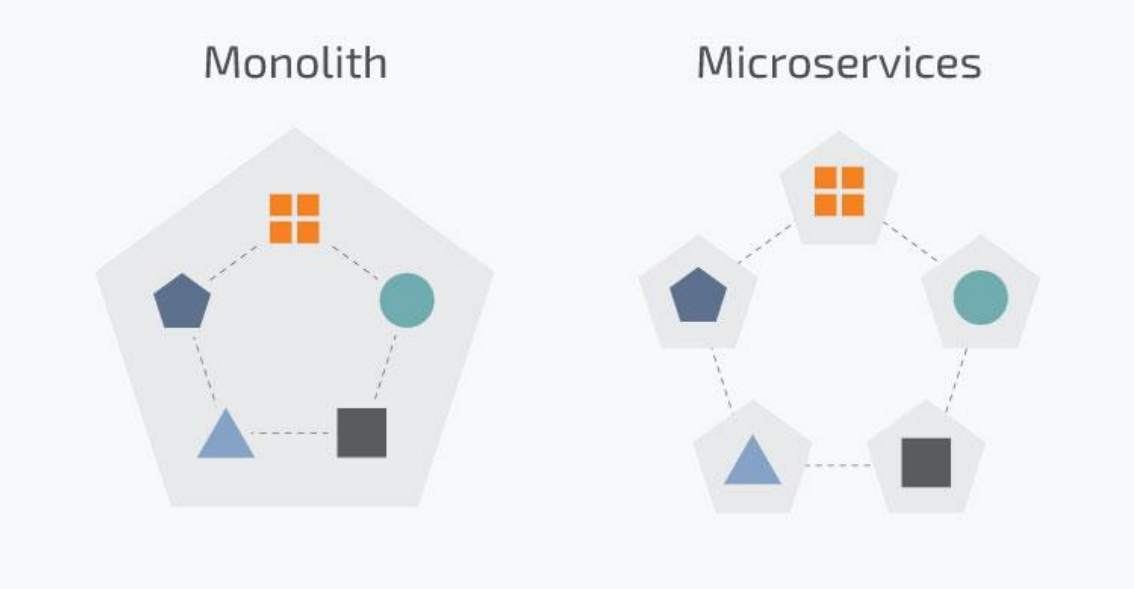
Felügyelt tanítás menete (ha ismert a cél érték)

- Kezdeti súlytényezők beállítása
- Tanítóminta bemeneti értéke alapján a hálózat kimeneti értékének kiszámítása
- A tanítóminta célértékének összehasonlítása a hálózat célértékével
- Szükség esetén a hálózat súlytényezőinek módosítása
- A tanítás folytatása mindaddig, amíg a hálózat az összes tanítómintára – egy előre rögzített hibahatárnál kisebb hibával a célértéknek megfelelő kimeneti értéket nem tud előállítani

[13] [14] [12]

6.2 MicroService VS Monolit

A microservicek egy hatalmas trenddé kezd válni manapság, hiszen elvitathatatlan pozitívumokat ad hozzá a szoftverfejlesztéshez, többek között úgy, mint a skálázhatóság, flexibilitás, agilitás stb. Több technikai vezető cég is váltott monolitikus alkalmazásról microservicekre. Ezek közé a cégek közé tartozik többek között a Google, Netflix, Amazon.
Ennek ellenére a monolitikus szoftver felépítés egy teljesen általános formája továbbra is a fejlesztésnek. Az ilyen felépítésű szoftverek fejlesztésénél viszont több probléma felmerül, hiszen a hatalmas kódbázis, új technológiák implementálása, skálázás, deployment az egész alkalmazás forráskódját érintik, nem csak egy szeletét azoknak. Felmerül a kérdés, hogy akkor a monolitikus architektúra felett eljárt az idő és mindennel át kéne-e térnünk microservice-ekbe.



6.7. ábra: Monolith vs. Microservices [15]

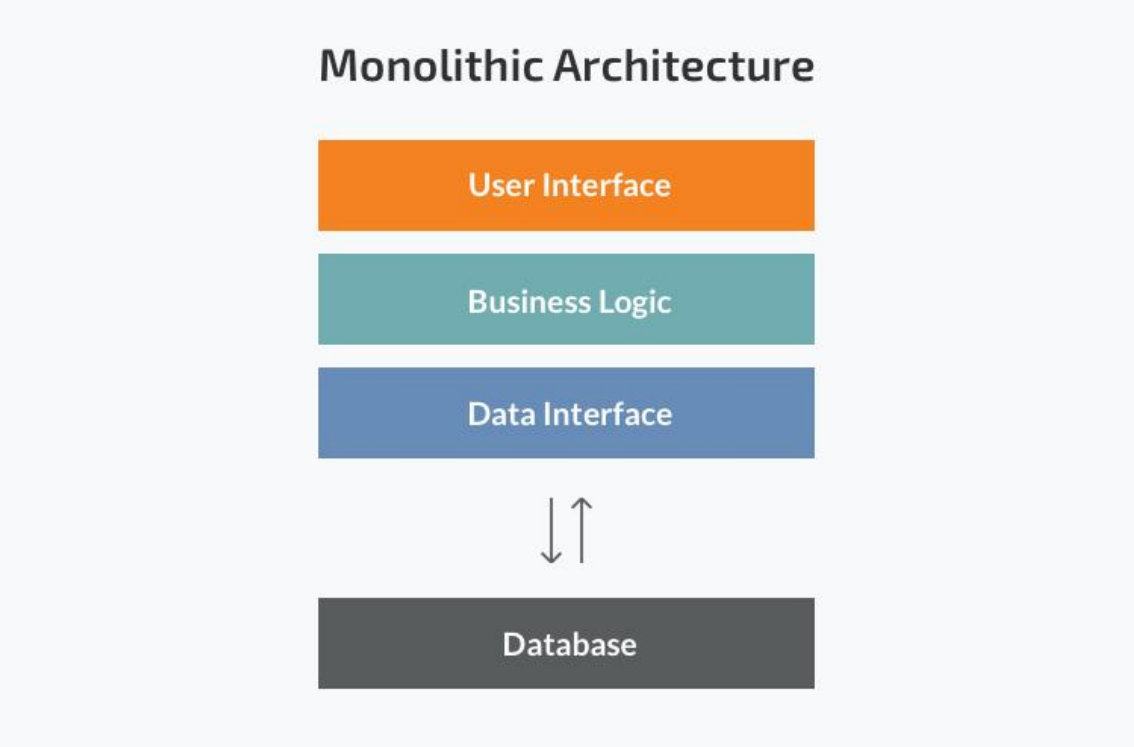
A monolit architektúra szerint felépített alkalmazások egy egységként készülnek el, nem feloszthető egységekből. Általában egy kliens-oldali felhasználói interfész, szerver-oldali applikáció és egy adatbázis alkotja. Ezek a funkcionalitások mind bele vannak építve egy alkalmazásba. Ezeknek a szoftvereknek általában egy kódbázisuk van, kevés modularitási lehetőséggel. Amikor a fejlesztők szeretnének módosítani, vagy updatelni valamit a szoftverben, akkor a teljes kódbázishoz hozzá kell férniük, így a teljes stackben végzik el a módosításokat.

Monolitikus architektúra pozitívumai

- Kevesebb helyen felvett függőségek
- Könnyebb debugolás és tesztelés
- Könnyebb deployolás

Monolitikus architektúra negatívumai

- Nehezebben olvasható kód
- Változtatások az alkalmazásban
- Skálázhatóság
- Új technológiák implementálása



6.8. ábra: Monolithic Architecture [15]

Amíg a monolitikus alkalmazások egy egységbe zárják a teljes alkalmazást, addig a microservice-ek lebontják ezeket az alkalmazásokat kisebb, független részegységekre. Minden különálló service rendelkezik saját logikai egységgel.

“Röviden összefoglalva a microservicek architekturális megközelítésének a lényege egy olyan alkalmazás fejlesztése, ami fel van bontva kisebb servicekre, amelyek mindegyike külön futtatja a saját processeit és gyakran HTTP resource API-n kommunikál.” – Martin Fowler. [16]

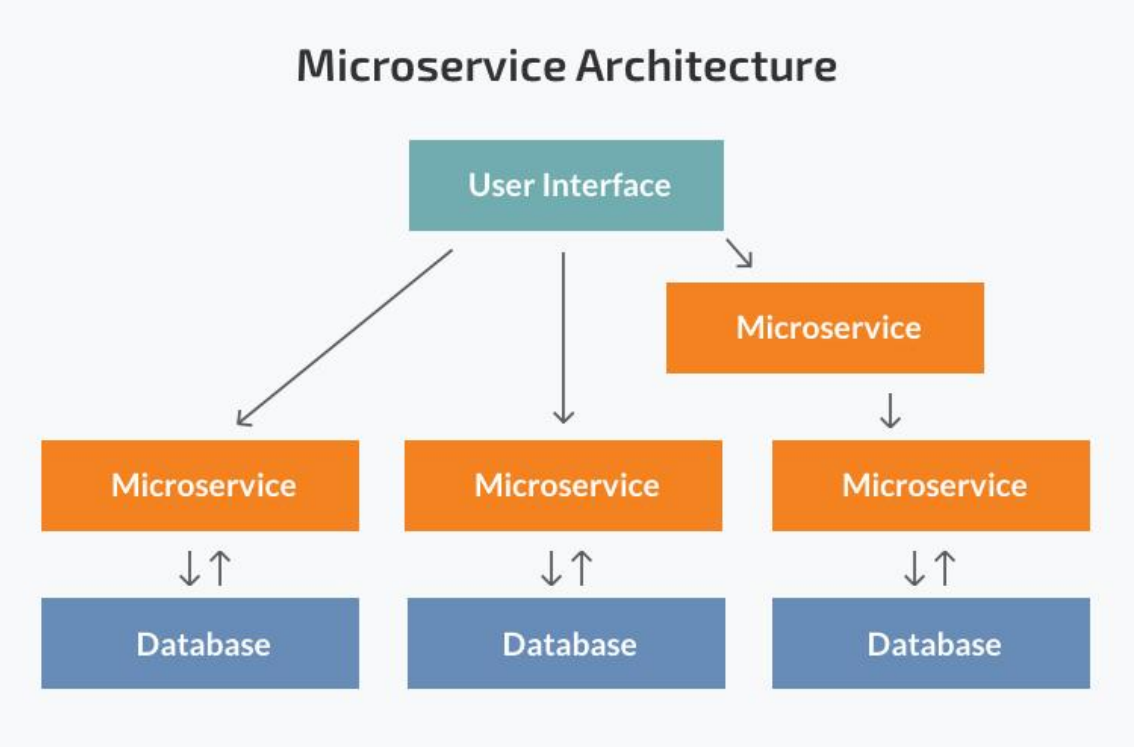
Az architektúrában a teljes funkcionalitás fel van bontva független, deployolható részmodulokra amik egymással kommunikálnak API-kon keresztül. Mindegyik service rendelkezik saját feladatkörrel, updatelhető és függetlenül skálázható.

Microservice architektúra pozitívumai

- Független komponensek
- Könnyebb érthetőség
- Skálázhatóság
- Rugalmasság a technológia kiválasztásában
- Magas szintű agilitás

Microservice architektúra negatívumai

- Komplex rendszer
- Többszörösen felvett függőségek
- Tesztelhetőség



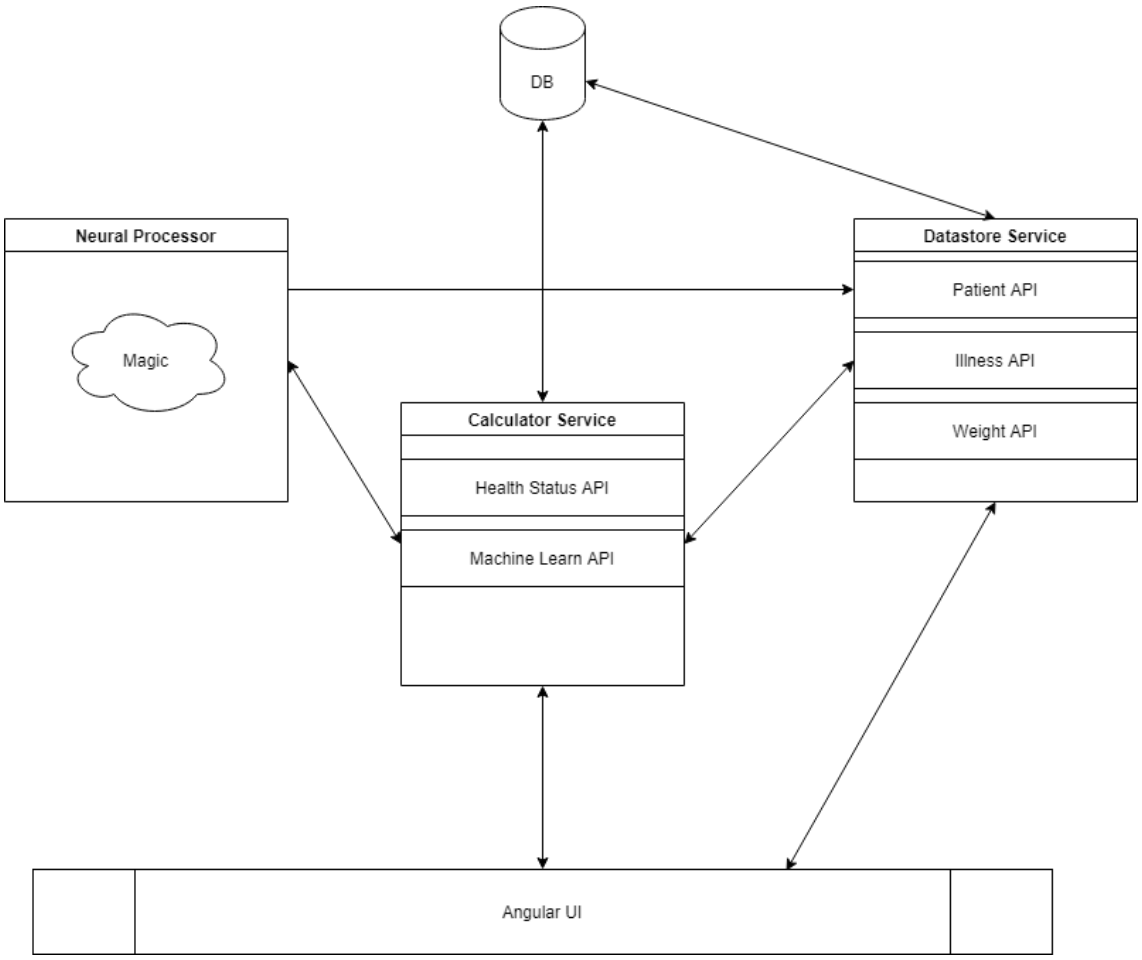
6.9. ábra: Microservice Architecture [15]

6.3 Összegzés

A microservices megközelítés nem egy univerzális megoldás minden alkalmazás fejlesztésére. Monolitikus alkalmazásoknak továbbra is helye van és lesz a piacon. Komplex, folyamatosan fejlődő alkalmazások fejlesztéséhez jó választás a microservices megközelítés, ami viszont tapasztalat nélkül könnyen fordulhat át rémálommá a bonyolultsága miatt.
[15] [17]

7 Rendszerterv

Az elkészítendő rendszer könnyen mozgathatósága érdekében egy konténerizált csomagot készítek, így bárhova egyszerűen kihelyezhető a szoftver. A komponensek könnyen kicserélhetőek egy esetleges frissítés esetén, illetve különböző volume-okon el tudok tárolni adatokat/állapotokat, így újraindulás esetén sem kell újratölteni és konfigurálni a működéseket. Alapvetően egy REST API rendszerről beszélhetünk, így a komponensek közötti kommunikáció adatai ***JSON*** formátumban közlekednek. Az adatbázis kommunikáció pedig ***JPA*** és ***Hibernate*** technológiákat használ.



7.1. ábra: Rendszerterv

7.1 Angular UI

Egy egyszerű UI felület létrehozása a felhasználók számára, ahol fel tudják vinni a szükséges adatokat a kalkulációhoz és az eredményeket meg tudjuk jeleníteni. Későbbiekben akár tovább lehet fejleszteni a megjelenítést Grafana API használatával beágyazott Iframe-eken keresztül.

7.2 Patient API (Spring Boot rest)

Spring Boot alkalmazás a betegadatok tárolására, lekérésére, módosítására, törlésére. Ez lesz közvetlen rákötve a beviteli UI rétegre. Az adatok mentésénél triggerelődik a Machine Learn API-n keresztül egy tanulási folyamat.

7.3 Illness API (Spring Boot rest)

Ezen a Spring Boot API-n keresztül lehet kezelni a betegségeket. Belső használatra készül, de a későbbiekben a továbbfejlesztés lehetőségére nyitott marad az alkalmazás.

7.4 Weight API (Spring Boot rest)

A neurális service ezen az API-n keresztül tudja lekérni és menteni a súlytényezőket a kalkulációkhoz.

7.5 Health Status API (Spring Boot rest)

Átadva a felhasználó ID-ját, lekérhetőek az adatok, súlytényezők és visszaad egy kalkulációt az egészségi állapotról és riskekről.

7.6 Machine Learn API (Spring Boot rest)

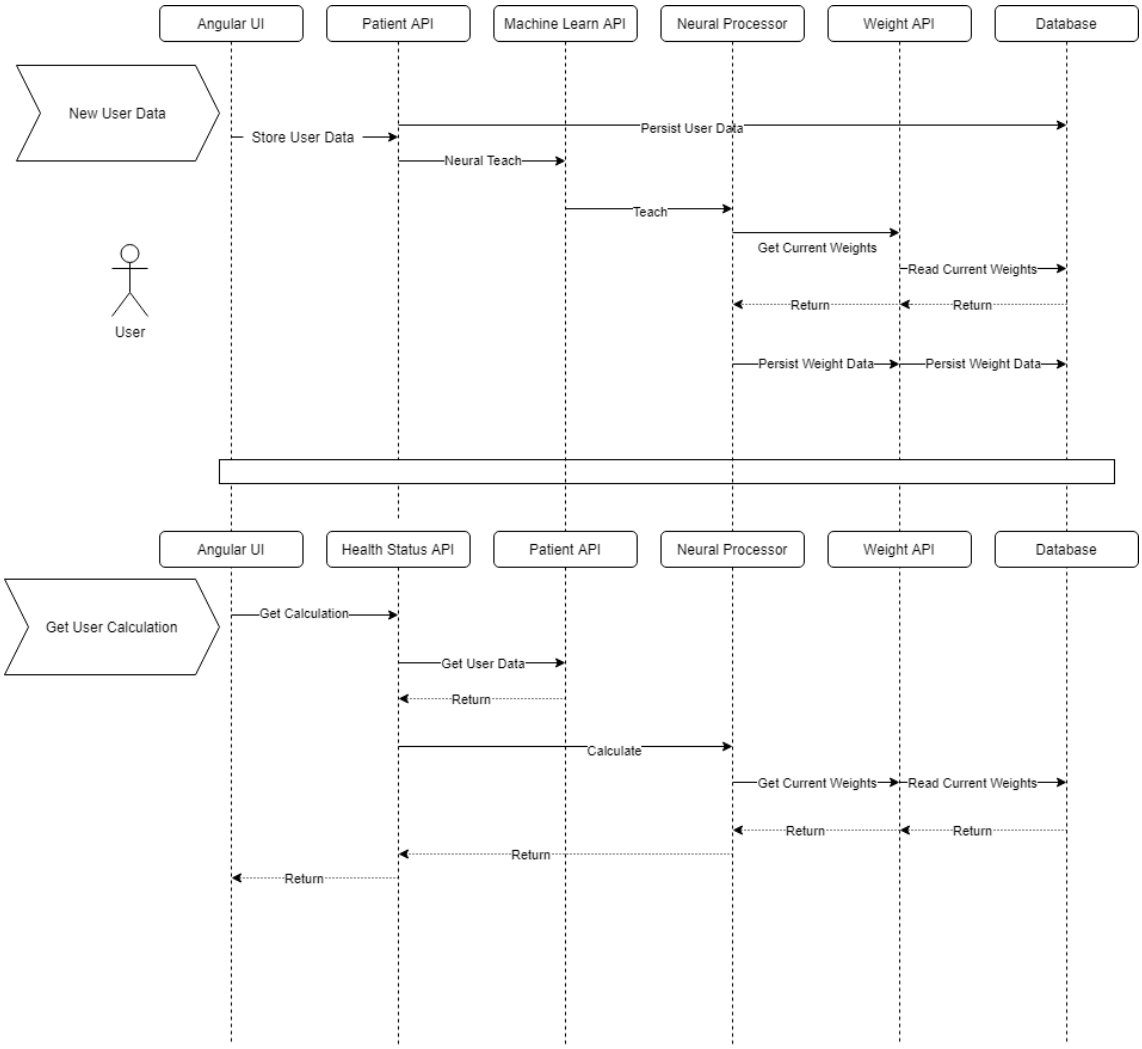
Ezen az API-n keresztül tudunk tanulási folyamatot elindítani a Neural Service-en belül. Ehhez lekéri a súlytényezőket és a megfelelő patient data-kat átalakítja, majd továbbítja a servicenek.

7.7 Adatbázis

PostgreSQL relációs adatbázis az alkalmazás adatainak tárolására.

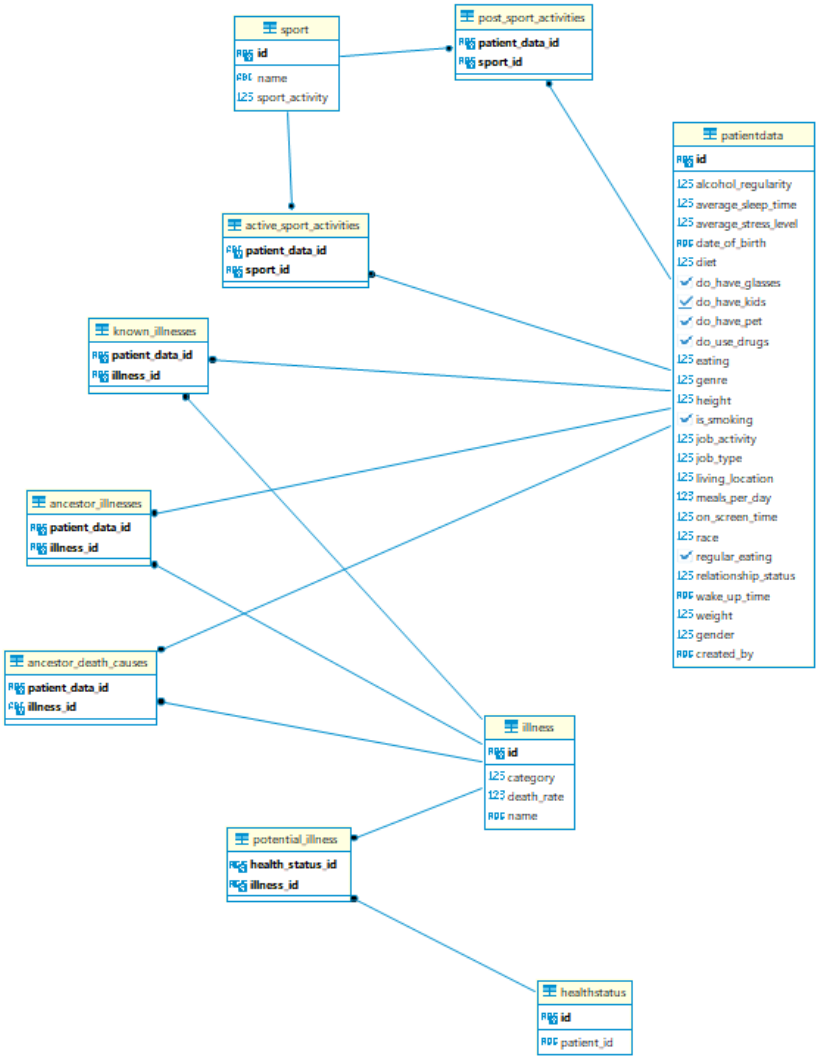
7.8 Neural Processor

Microservice amely egy Java implementációban megvalósított neurális háló. Fogad betegadatokat, súlytényezőket olvas és állít be, illetve becsléseket képes készíteni a korábbi adatok ismeretében.



7.2. ábra: Folyamat terv

8 Adatbázis terv



8.1. ábra: Táblakapcsolat terv

8.1 Táblák

8.1.1 Active_sport_activities

Column Name	#	Data type	Identity	Collation	Not Null	Default	Comment
patient_data_id	1	varchar(255)		default	[v]		
sport_id	2	varchar(255)		default	[v]		

8.2. ábra: Active sport activities

8.1.2 Ancestor_death_causes

Column Name	#	Data type	Identity	Collation	Not Null	Default	Comment
patient_data_id	1	varchar(255)		default	[v]		
illness_id	2	varchar(255)		default	[v]		

8.3. ábra: Ancestor death causes

8.1.3 Ancestor_illnesses

Column Name	#	Data type	Identity	Collation	Not Null	Default	Comment
patient_data_id	1	varchar(255)		default	[v]		
illness_id	2	varchar(255)		default	[v]		

8.4. ábra: Ancestor illnesses

8.1.4 Healthstatus

Column Name	#	Data type	Identity	Collation	Not Null	Default	Comment
id	1	varchar(255)		default	[v]		
patient_id	2	varchar(255)		default	[ ]		

8.5. ábra: Healthstatus

8.1.5 Illness

Column Name	#	Data type	Identity	Collation	Not Null	Default	Comment
id	1	varchar(255)		default	[v]		
category	2	int4			[ ]		
death_rate	3	float8			[ ]		
name	4	varchar(255)		default	[ ]		

8.6. ábra: Illness

8.1.6 Known_illnesses

Column Name	#	Data type	Identity	Collation	Not Null	Default	Comment
patient_data_id	1	varchar(255)		default	[v]		
illness_id	2	varchar(255)		default	[v]		

8.7. ábra: Known illnesses

8.1.7 Patient data

Column Name	#	Data type	Identity	Collation	Not Null	Default	Comment
id	1	varchar(255)		default	[v]		
alcohol_regularity	2	int4			[ ]		
average_sleep_time	3	int4			[ ]		
average_stress_level	4	int4			[ ]		
date_of_birth	5	varchar(255)		default	[ ]		
diet	6	int4			[ ]		
do_have_glasses	7	bool			[v]		
do_have_kids	8	bool			[v]		
do_have_pet	9	bool			[v]		
do_use_drugs	10	bool			[v]		
eating	11	int4			[ ]		
genre	12	int4			[ ]		
height	13	float8			[ ]		
is_smoking	14	bool			[v]		
job_activity	15	int4			[ ]		
job_type	16	int4			[ ]		
living_location	17	int4			[ ]		
meals_per_day	18	int4			[ ]		
on_screen_time	19	int4			[ ]		
race	20	int4			[ ]		
regular_eating	21	bool			[v]		
relationship_status	22	int4			[ ]		
wake_up_time	23	varchar(255)		default	[ ]		
weight	24	float8			[ ]		
gender	25	int4			[ ]		
created_by	26	varchar(255)		default	[ ]		

8.8. ábra: Patient data

8.1.8 Post_sport_activities

Column Name	#	Data type	Identity	Collation	Not Null	Default	Comment
patient_data_id	1	varchar(255)		default	[v]		
sport_id	2	varchar(255)		default	[v]		

8.9. ábra: Post sport activities

8.1.9 Potential_illness

Column Name	#	Data type	Identity	Collation	Not Null	Default	Comment
health_status_id	1	varchar(255)		default	[v]		
illness_id	2	varchar(255)		default	[v]		

8.10. ábra: Potential illness

8.1.10 Sport

Column Name	#	Data type	Identity	Collation	Not Null	Default	Comment
id	1	varchar(255)		default	[v]		
name	2	varchar(255)		default	[v]		
sport_activity	3	int4			[ ]		

8.11. ábra: Sport

Mivel lehetővé teszi a Java-s framework, hogy ne kelljen külön létrehoznom az adatbázist, hanem konfiguráció alapján frissíteni tudom a kódom alapján, így hagyom az implementációnak, hogy kezelje az táblák és kapcsolataik létrehozását. Ilyen módon a DB sokkal könnyebben kezelhető, és jobban reagál a módosításokra amik esetlegesen implementáció közben felmerülnek. A megszorítások és key kezelések pedig rögtön a kód szintjén, entitásokként jelennek meg, így már fordítási időben szól a ***JPA***, ha valamit nagyon elrontanánk.

8.2 Java Persistence API

A keretrendszer feladata a relációs adatok kezelése a Java Platform Standard és az Enterprise Editiont használó alkalmazásokban. A Java Persistence API a JSR 220 Expert Group, a JPA 2.0 a JSR 317 Expert Group munkája. Az elnevezésben szereplő perzisztencia utal a létrehozó folyamatot való túlélésre, amit JDBC szerializáción keresztül küldök be a ***PostgreSQL*** adatbázisba. [18] [19]

9 Algoritmus terv

A megvalósításban valamilyen módon szükséges átforgatnom a bejövő adatokat bináris formátumra, mostani meglátásom szerint így lesz a legkönnyebb az implementáció és az összefüggések megtalálása. A betegadatokat egy új formába kell elrendezni, ami betáplálható a neuronok első rétegébe és ezzel elindulhat mind a tanítás, mind a becslés folyamata. Nagyon nagy pontosságot nem várhatunk el a viszonylag kicsi adatmennyiség miatt, de ez idővel javulhat a névtelen kitöltőknek köszönhetően. Egy N bemenetű hálót kell megvalósítani, ahol ismertek a kimenetelek, illetve a már ismert betegségek. Mind a kettő lehet forrása a tanításnak.

10 Projekt megvalósítása, fejlesztés

A megvalósítás előtt szükséges volt megtervezni a data transfer objecteket, api-okat és kommunikációs sorrendeket. A skeleton projektek létrehozása után kipróbálhatóvá vált a futtathatósága a service-eknek egyelőre még valós kommunikáció nélkül. Ezek után lépésenként implementálni kellett a funkcionalitásokat. Talán a legnehezebbnek mondható az implementációs folyamatban a folyamatos kisebb refaktorálások kezelése.

10.1 Docker-Compose, env, környezet

Első lépésként össze kell állítani a környezetet amit folyamatosan bővítve lehet folytatni az alkalmazás fejlesztését. A ***Docker*** yml fileok alapján tud létrehozni infrákat amik egy szerveres környezetet jelentenek számunkra. Ezek a konfigurációs fileok úgy épülnek fel, hogy megadjuk a service-eket amiket futtatniuk kell, azokat felparaméterezzük, és beállítunk volume-okat amik perzisztálják az adatokat, így újraindításkor nem kezdjük elölről az adatbázisok feltöltését és a programok paraméterezését.

```yaml
version: "3.8"

services:
  database:
    image: postgres:10.5
    environment:
      POSTGRES_USER: ${DB_USER}
      POSTGRES_PASSWORD: ${DB_PASSWORD}
      POSTGRES_DB: ${DB_NAME}
    volumes:
      - database-data:/var/lib/postgresql/data/
    ports:
      - "54321:5432"

  keycloak:
    image: "jboss/keycloak:15.0.2"
    environment:
      KEYCLOAK_USER: "keycloak"
      KEYCLOAK_PASSWORD: ${KEYCLOAK_PASSWORD}
      DB_VENDOR: "postgres"
      DB_ADDR: "keycloak-database"
      DB_USER: "keycloak"
      DB_PASSWORD: ${KEYCLOAK_DB_PASSWORD}
      DB_DATABASE: "keycloak"
      KEYCLOAK_FRONTEND_URL: "http://localhost:8088/auth"
    ports:
      - "8088:8080"
  keycloak-database:
    image: "postgres:13.2-alpine"
    environment:
      POSTGRES_USER: "keycloak"
      POSTGRES_DB: "keycloak"
      POSTGRES_PASSWORD: ${KEYCLOAK_DB_PASSWORD}
    volumes:
      - "keycloak_data:/var/lib/postgresql/data"
    deploy:
      mode: replicated
      replicas: 1
      placement:
        constraints: [node.role == manager]

volumes:
  keycloak_data:
  database-data:
```

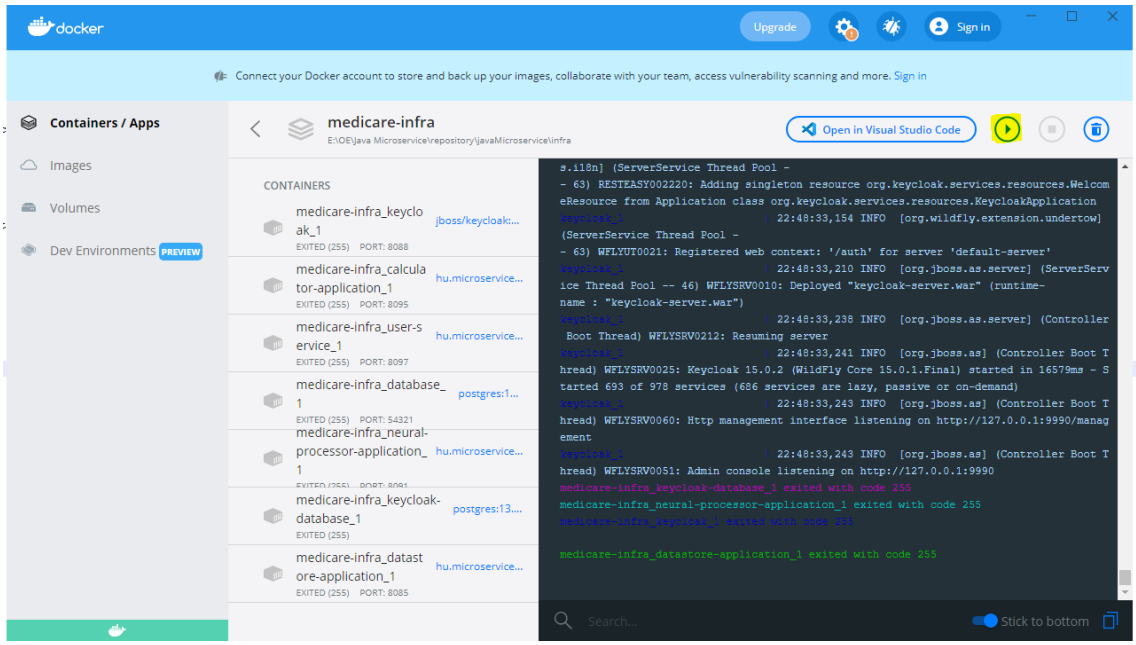
10.1. ábra: Docker-Compose, env, környezet

A $ tagekkel jelölt részek egy .env fájlból érkeznek, amiben a taggel jelölt változók felparaméterezhetők, így az esetleges módosításoknál csak egy helyen kell megváltoztatni a szükséges adatokat.

A ***Dockeres*** yml file futtatásához és leállításához kettő parancsunk van amit cmd-ből kiadva érhetünk el.

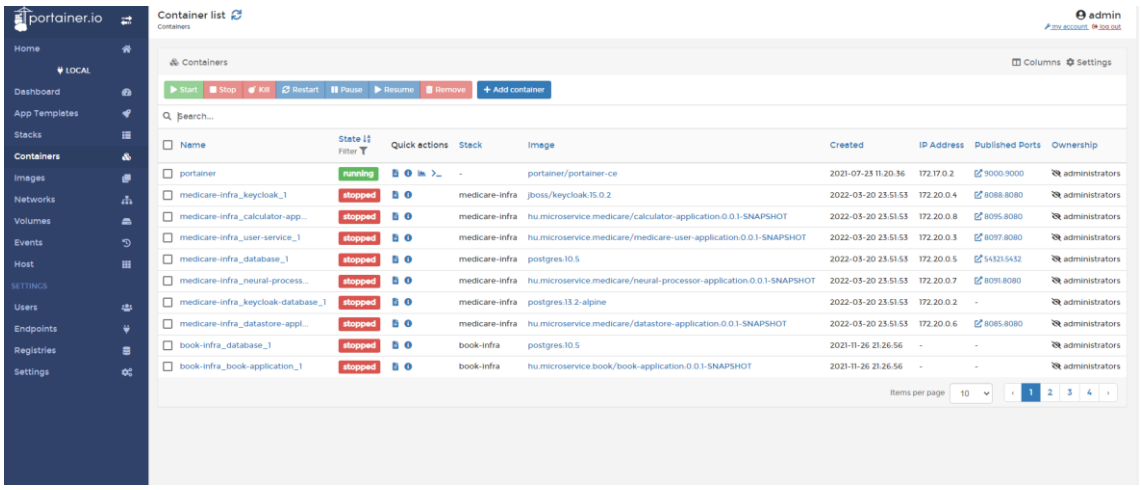
`docker-compose up -d`
`docker-compose down`

Az uppal töltjük fel a környezetbe és indítjuk a service-eket, a down-al pedig leállítjuk a futásukat. Az első deploy után viszont már a Docker felületén is láthatóvá válnak az infrák és így a play gomb megnyomásával futtatható a rendszer.



10.2. ábra: Docker Desktop nézet

Az első startup előtt érdemes a ***Docker***hez egy ***Portrainer*** nevű alkalmazást rendelni amiben könnyen figyelhetőek a futó servicek és konténerek.



10.3. ábra: Portrainerben futó servicek

Ezt követően a konfigurációs yml file-t bővítve tudunk hozzáadni komponenseket a környezetünkhöz, amiket a ***Docker*** futtatni fog és portainer felületen monitorozhatunk. Mint fentebb látszik, ezért választottam a ***Docker***es környezetet, hiszen rendkívül egyszerű, kompakt a felépítése és könnyen átmozgatható.

10.2 Backend, Servicek

Először szükséges a backend oldali API-ok jó definiálása annak érdekében, hogy a frontend elkészíthető legyen a megfelelő elérési pontokkal. Vannak olyan enumok és DTO-k is, amelyek mind a két oldalon elő kell álljanak, ezért nagyon fontos a jól dokumentáltsága ezeknek.
Amint előállt a backend és implementálásra kerültek a megfelelő servicek, még frontend nélkül lehetőségünk van ***Postman***-el tesztelni a különböző műveleteket. PUT, GET műveletek minden egyes komponens létrehozása során tesztelendőek kézzel. Ezt ***Postman***en kívül akár a ***SwaggerUI*** felületén keresztül is megtehetjük.

10.2.1 Datastore Component

PatientData DTO

Mező	Típus	Leírás
id	string	Egyedi azonosító
gender	enum	Neme
dateOfBirth	string	Születési ideje
weight	double	Súlya
height	double	Magassága
relationshipStatus	enum	Kapcsolati státusza
doHaveKids	boolean	Van-e gyermeke
jobType	enum	Munka típusa
jobActivity	enum	Munka aktivitása
livingLocation	enum	Lakhely típusa
race	enum	Rassz
createdBy	string	Felhasználó egyedi azonosítója
averageSleepTime	integer	Átlagos alvási idő
onScreenTime	integer	Számítógép előtt töltött idő
averageStressLevel	integer	Általános stressz szint %-ban
regularEating	boolean	Rendszeresen étkezik-e
mealsPerDay	integer	Napi étkezések száma
eating	enum	Ételek preferenciája
diet	enum	Diéta amit követ
alcoholRegularity	enum	Alkoholfogyasztás rendszeressége
isSmoking	boolean	Dohányzik-e
doUseDrugs	boolean	Használ-e kábítószereket
doHaveGlasses	boolean	Visel-e szemüveget
doHavePet	boolean	Rendelkezik háziállattal
activeSportActivities	enum[]	Aktív sporttevékenységek listája
postSportActivities	enum[]	Korábbi sporttevékenységek listája
knownIllnesses	enum[]	Ismert betegségek listája
ancestorIllnesses	enum[]	Felmenők ismert betegségei
ancestorDeathCauses	enum[]	Felmenők halálának okai

1. tábla: PatientData DTO

Enumok

Gender

Név	Érték
Férfi	MALE
Nő	FEMALE
Más	OTHER

2. tábla: Gender

RelationshipStatus

Név	Érték
Egyedülálló	SINGLE
Kapcsolatban	IN_A_RELATIONSHIP
Eljegyezve	ENGAGED
Házas	MARRIED
Külön él	SEPARATED
Elvált	DIVORCED
Özvegy	WIDOWED

3. tábla: Relationship Status

JobType

Név	Érték
Diák	STUDENT
Dolgozó diák	STUDENT_AND_WORKING
Mezőgazdaság	AGRICULTURE
Építőipar	ARCHITECTURE_AND_CONSTRUCTION
Művészetek	ARTS
Üzlet, pénzügy	BUSINESS_AND_FINANCE
Oktatás	EDUCATION_AND_TRAINING
Állami alkalmazott	GOVERNMENT_AND_PUBLIC_ADMINISTRATION
Egészségügy	HEALTH
Informatika	INFORMATION_TECHNOLOGY
Divatipar	FASHION
Élelmiszeripar	GOODS
Szolgáltatóipar	SERVICE_CONTRACT
Vendéglátás	HOSPITALITY
Nyugdíjas	RETIRED

4. tábla: Job Type

JobActivity

Név	Érték
Fizikai	PHYSICAL
Szellemi	MENTAL
Kereskedelem	TRADE
Sofőr	DRIVING
Nyugdíjas	RETIRED

5. tábla: Job Activity

Location

Név	Érték
Főváros	CAPITAL_CITY
Város	CITY
Kisváros	COUNTRY
Falu	VILLAGE

6. tábla: Location

Race

Név	Érték
Kaukázusi	CAUCASIAN
Ázsiai	ASIAN
Fekete	BLACK
Roma/Indiai	GYPSY
Kevert	MIXED

7. tábla: Race

Eating

Név	Érték
Otthon főzött	HOME_COOKED
Kiszállításos	ORDERING
Gyorsétel	FASTFOOD

8. tábla: Eating

Diet

Név	Érték
Nem követ diétát	NO_DIET
Paleo	PALEO
Húsevő	CARNIVORE
Vegán	VEGAN
Vegetáriánus	VEGETARIAN
Ketogén	KETOGENIC
Mediterrán	MEDITERRANEAN
Intermittent Fasting	INTERMITTENT_FASTING
Egyéb Fittness	OTHER_FITNESS

9. tábla: Diet

AlcoholRegularity

Név	Érték
Napi	DAILY
Heti többször	MORE_THAN_ONCE_A_WEEK
Hétvégente	WEEKEND
Alkalmanként	OCCASION
Soha	NEVER

10. tábla: Alcohol Regularity

SportActivity

Név	Érték
Semmi	NONE
Séta	WALKING
Edzőterem/Fittness	FITNESS_OR_GYM
Futás/Atlétika	RUNNING_OR_ATHLETICS
Úszás	SWIMMING
Biciklizés	CYCLING
Túrázás	BUSHWALKING
Csapatjátékok	FOOTBALL_OR_SOCCER
Yoga	YOGA
Küzdősport	MARTIAL_ARTS
Más	OTHER

11. tábla: Sport Activity

IllnessCategory

Név	Érték
Vér	BLOOD
Rák/Daganat	CANCER_AND_NEOPLASMS
Kardiovaszkuláris	CARDIOVASCULAR
Fül	EAR
Szem	EYE
Fertőzés	INFECTION
Autoimmun/Immunrendszeri	INFLAMMATORY_AND_IMMUNE_SYSTEM
Baleset	INJURIES_AND_ACCIDENTS
Mentális	MENTAL_HEALTH
Endokrinológia	METABOLIC_AND_ENDOCRINE
Váz/Izom-rendszer	MUSCULOSKELETAL
Neurológia	NEUROLOGICAL
Gasztro	ORAL_AND_GASTROINTESTINAL
Vese/Húgyúti	RENAL_AND_UROGENITAL
Nemzőszervi	REPRODUCTIVE_HEALTH_AND_CHILDBIRTH
Légzőszervi	RESPIRATORY
Bőr	SKIN
Stroke	STROKE
Ismeretlen	UNKNOWN

12. tábla: Illness Category

Patient API

Request URI	HTTP Method	Request payload	Response code	Response payload	Desc.
/api/DataStore/PatientData	GET	-	200	PatientData[]	Visszaadja a DataStore által kezelt összes PatientData-t
/api/DataStore/PatientData/{id}	GET	-	200	PatientData	Visszaadja az id azonosítójú PatientData-t
/api/DataStore/PatientData/{id}	GET	-	404	PatientData	Nem létezik id azonosítójú PatientData
/api/DataStore/PatientData/user	GET	-	200	PatientData	Visszaadja a user-hez tartozó PatientData-t
/api/DataStore/PatientData/user	GET	-	404	PatientData	Nem tartozik az user-hez PatientData
/api/DataStore/PatientData	PUT	PatientData	200	-	Létrehoz, vagy updatel egy PatiantData-t ami a user-hez tartozik
/api/DataStore/PatientData/{id}	DELETE	-	200	-	Az id azonosítójú PatientData törlésre kerül
/api/DataStore/PatientData/user	DELETE	-	200	-	A user-hez tartozó PatientData törlésre kerül

13. tábla: Patient API

HealthStatus DTO

Mező	Típus	Leírás
id	string	Egyedi azonosító
patientId	string	User egyedi azonosító
potentialIllnesses	PotentialIllness[]	A user-hez tartozó potenciális betegségek listája

14. tábla: Health Status DTO

PotentialIllness DTO

Mező	Típus	Leírás
id	string	Egyedi azonosító
percent	int	Százalékos esély
illness	enum	Betegség típus

15. tábla: Potential Illness DTO

HealthStatus API

Request URI	HTTP Method	Request payload	Response code	Response payload	Desc.
/api/HealthStatus/user	GET	-	200	HealthStatus	Visszaadja a user-hez tartozó HealthStatus-t
/api/HealthStatus/user	GET	-	404	HealthStatus	Nem található a user-hez tartozó HealthStatus
/api/HealthStatus	PUT	HealthStatus	200	HealthStatus	Létrehoz, vagy updateli a user-hez tartozó HealthStatust
/api/HealthStatus/{id}	DELETE	-	200	-	Törli az id azonosítójú HealthStatust

16. tábla: Health Status API

NeuralTransferObject

Mező	Típus	Leírás
weightMatrix	WeightMatrix	Súlymátrix ami szükséges a Neurális Háló működéséhez
patientData	PatientData	Egy user-hez tartozó PatientData amivel akár tanítani, akár kalkuláltatni tudunk a Neurális Hálóval

17. tábla: Neural Transfer Object

WeightMatrix DTO

Mező	Típus	Leírás
id	String	Egyedi azonosító
weights	Double[][][]	A súlymátrix adatai
eta	Double	Neurális háló szorzó száma
networkLayerSizes	Int[]	A hálózat rétegeinek száma. Ez a neuron rétegek számát jelöli
bias	Double[][]	Neurális korrekciós szám

18. tábla: Weight Matrix DTO

Weight API

Request URI	HTTP Method	Request payload	Response code	Response payload	Desc.
/api/Weight/{id}	GET	-	200	WeightMatrix	Visszaadja az id-hoz tartozó WeightMatrix-ot
/api/Weight/{id}	POST	WeightMatrix	200	WeightMatrix	Updateli az id-hoz tartozó WeightMatrix-ot

19. tábla: Weight API

10.2.2 Calculator Component

MachineLearn API

Request URI	HTTP Method	Request payload	Response code	Response payload	Desc.
/api/MachineLearn/teach	POST	PatientData	200	WeightMatrix	A PatientData alapján elkezdi tanítani a Neurális Hálót, így WeightMatrix keletkezik
/api/MachineLearn/process/{id}	PUT	-	200	HealthStatus	A user id alapján kiértékeli a HealthStatus-t

20. tábla: Machine Learn API

10.2.3 Neural Processor Component

Neural API

Request URI	HTTP Method	Request payload	Response code	Response payload	Desc.
/api/Neural/teach	POST	NeuralTransferObject	200	WeightMatrix	A NeuralTransferObject tartalma alapján tanul az algoritmus
/api/Neural/calculate	GET	NeuralTransferObject	200	HealthStatus	A NeuralTransferObject alapján kalkulál egy HealthStatust

21. tábla: Neural API

10.2.4 Medicare Common Component

Ennek a komponensnek az a célja, hogy az általános jellegű és úgynevezett univerzális felhasználású elemeket generalizáljuk. Kettő dologra értelmezett leginkább ez az absztrakció; authentikáció és proxyzás. Gyakorlatban ez úgy jelenik meg, hogy létrehozunk egy saját készítésű libeket, amiket becsomagolunk egy bom-ba, ami így egyben importálható a dependenciák közé.

Proxy Factory
A végpontokon alapuló servicek közötti proxyzást megvalósító lib. A megfelelő interceptorokat alkalmazva az alkalmazásokon belül felhívható végpontokat készít.

Security Common
Az általános ***Keycloak*** oAuth2-es authentikációt végrehajtó lib. Elég a megfelelő microservice-be importálni és máris működőképes lesz a role-ok és userek alkalmazása.

10.2.5 User Service Component

Mivel a userek egy külső alkalmazás által vannak kezelve, ezért kell egy hozzáférési pont, amin keresztül elérhetjük az adataikat. Ehhez a Keycloak-tól lekérjük az adatokat a SecurityContext alapján, mappeljük a saját objektumunkra és ezt az entitást küldjük tovább.

Keycloak API

Request URI	HTTP Method	Request payload	Response code	Response payload	Desc.
/auth/admin/realms/{realms}/users/{id}	GET	-	200	KeycloakUser	Visszaadja a Keycloak user adatait

22. tábla: Keycloak API

KeycloakUser

Mező	Típus	Leírás
id	string	Egyedi azonosító
Email	string	A felhasználó email címe

23. tábla: Keycloak User

User API

Request URI	HTTP Method	Request payload	Response code	Response payload	Desc.
/api/Users/Current	GET	-	200	UserMeta	Visszaadja az aktuális user meta adatait

24. tábla: User API

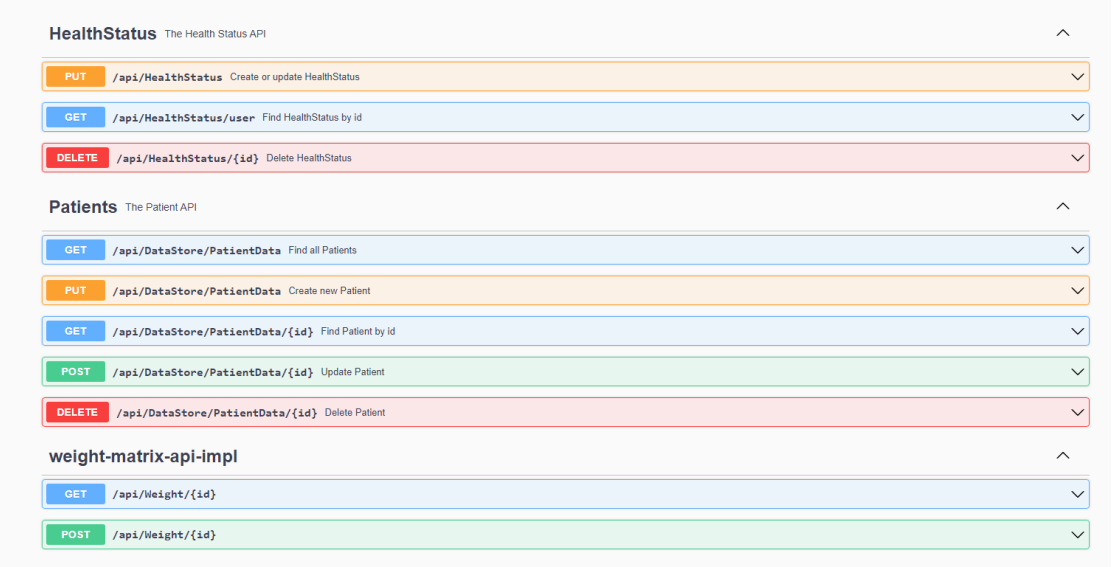
UserMeta

Mező	Típus	Leírás
id	string	Egyedi azonosító
email	string	A felhasználó email címe

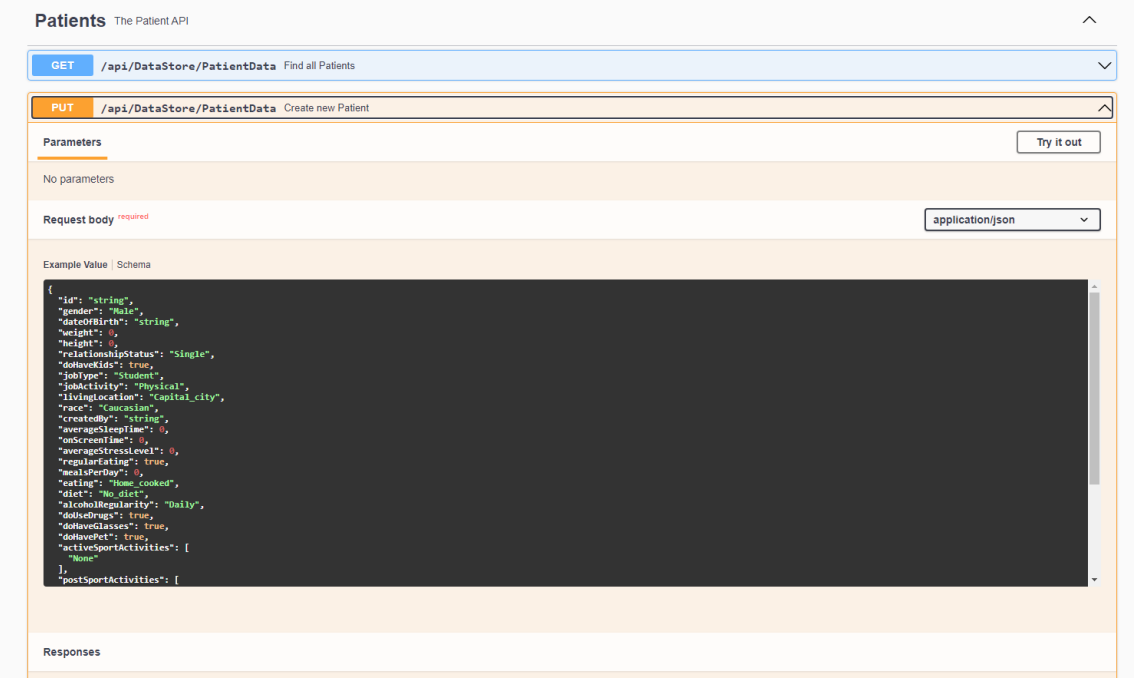
25. tábla: User Meta

10.2.6 SwaggerUI

Ezzel az eszközzel lehet ellenőrizni a futó servicek elérési pontjait és tesztelni egy-egy requestet is.



10.4. ábra: Swagger UI



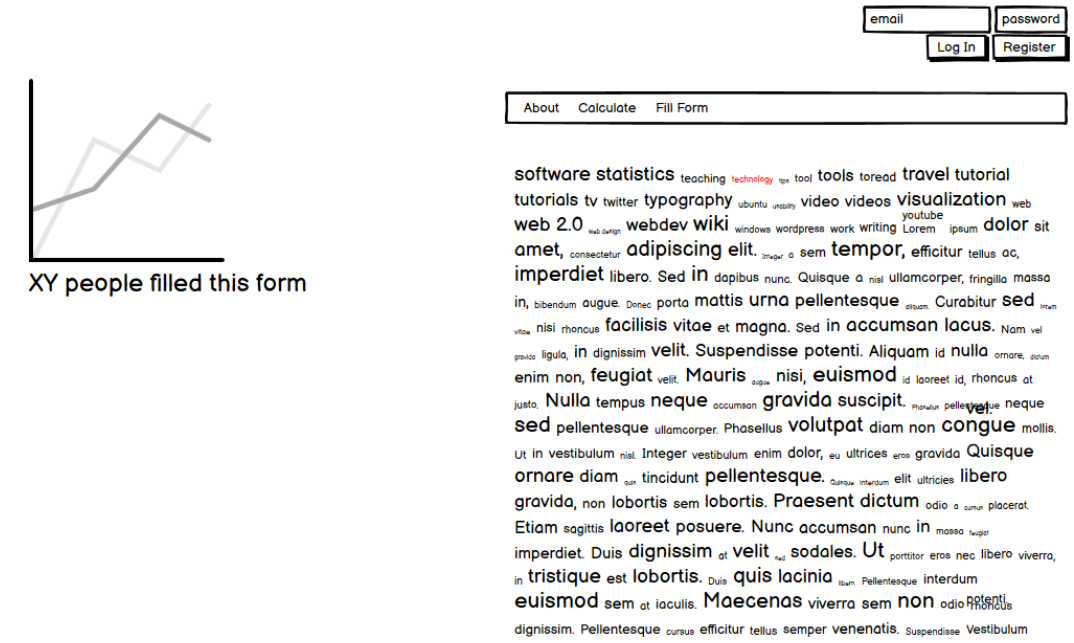
10.5. ábra: Swagger UI request

10.3 Frontend

A frontend megvalósításának első lépésében érdemes mockupokat készíteni, hogy mit is szeretnénk pontosan látni a felületen. Erre én a ***Balsamiq Wireframes*** eszközt használtam. Nem kell teljesen tökéletesnek lennie, de nagyjából képet tudunk kapni arról, hogy miket és hogyan akarunk megjeleníteni a felhasználói felületen.

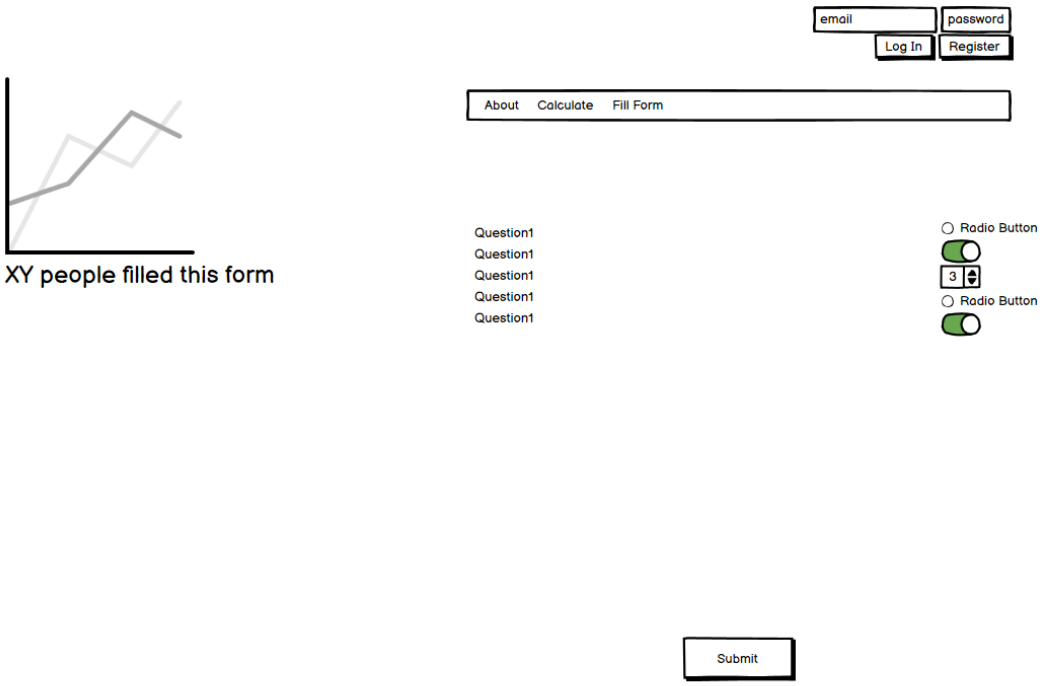
Következő lépésnek az Angularos componensek gyártásába kell kezdeni, modulokat kialakítani és modelleket képezni. Amikor a komponensek már elkészültek, mockolt fake adatokkal kell kipróbálni a működést.
Végül a végpontokat összekötve el kell indítani az applikációt és tesztelni a kommunikációt.
(Ezen a ponton még érdemes finomítani a UI felület megjelenítéseit.)

10.3.1 Mockup



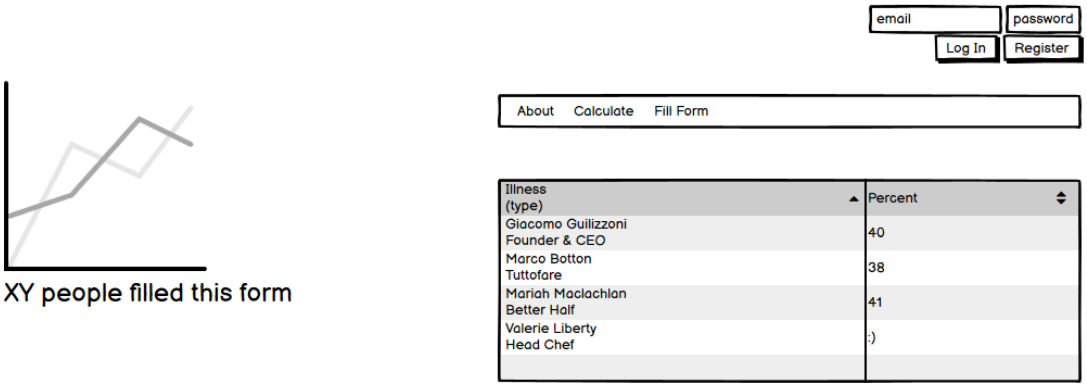
10.6. ábra: About mockup

Fill Form



10.7. ábra: Fill Form mockup

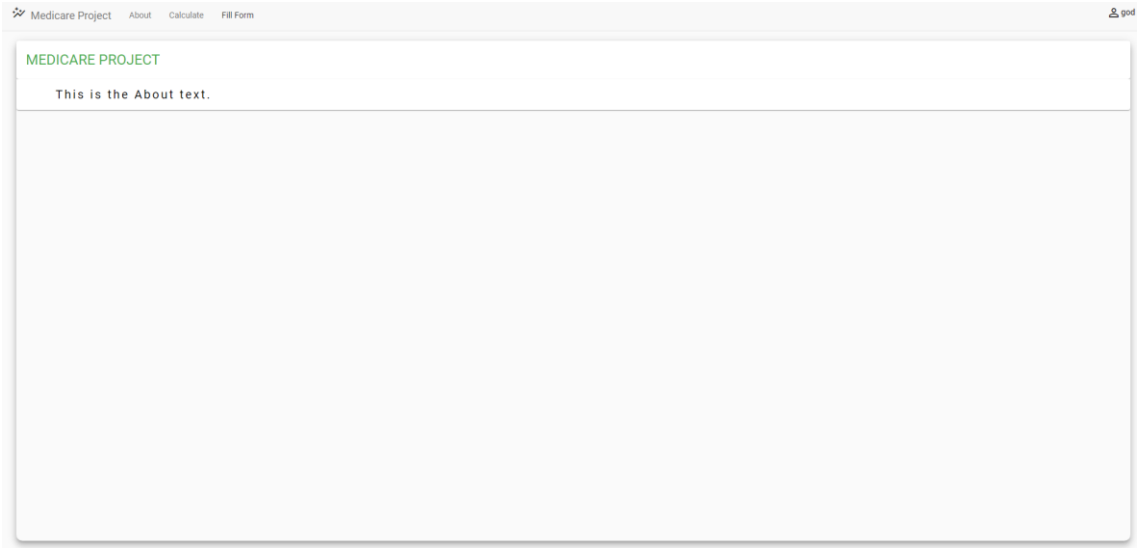
Calculate



10.8. ábra: Calculate mockup

10.3.2 About

Ebben a komponensben kerül elhelyezésre a projekt leírása. Nem rendelkezik üzleti értékkel, csupán egy tájékoztató jellegű oldal, ahol a felhasználó el tudja helyezni a magában, hogy miről szól a termék. Ehhez tartozik még egy bal oldalról beugró bar, amin pedig a kitöltöttségi statisztikákat lehet látni. Ez egy érdekes adat azt figyelembe véve, hogy milyen pontosságú becsléseket tud adni jelenleg a rendszer.

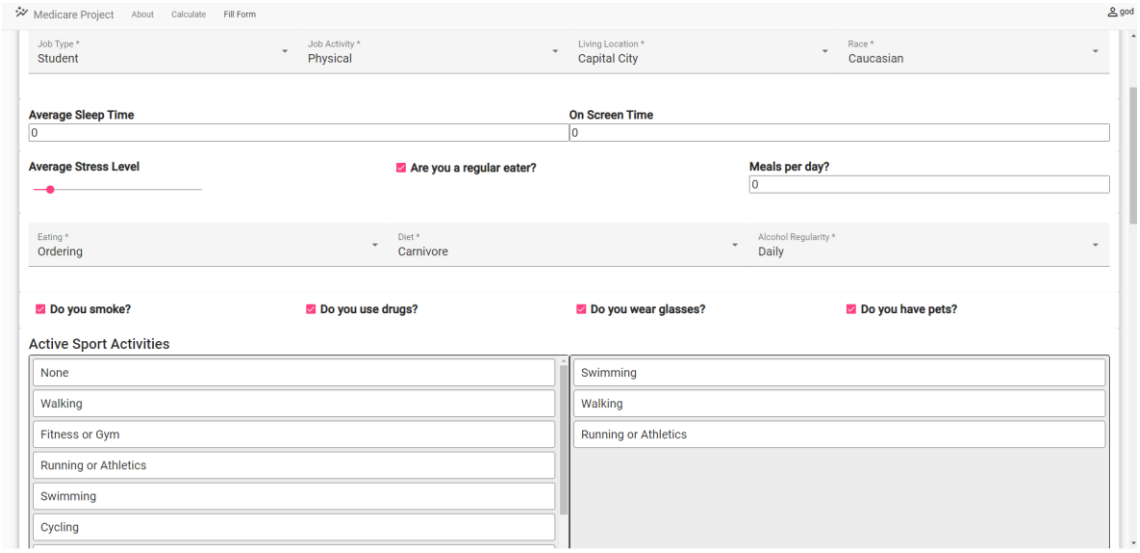


10.9. ábra: About page

A megvalósított felületen a tartalom feltöltése az adatbázisból érkezik, amit a backend indulásával egy script szúr be liquibase segítségével.

10.3.3 Fill Form

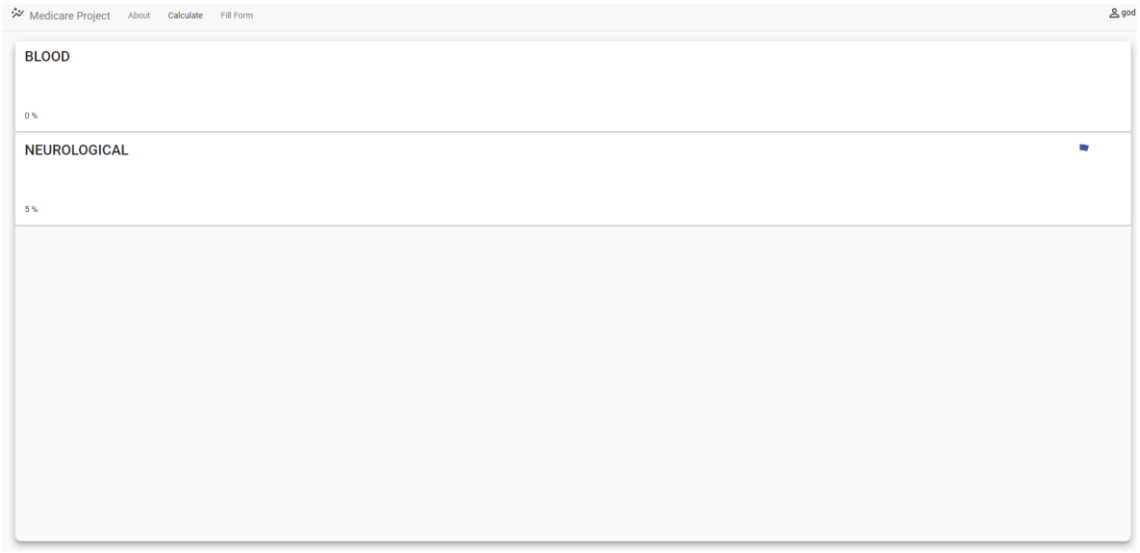
Itt található az a form ami alapján a kalkulációkat el tudjuk végezni. A felhasználó ezt bármikor tudja frissíteni, korábbi adatait felülírni. Submit esetén érdemes valamilyen visszajelzést adni számára, ha a mentés sikeres volt. A kérdéseket, gombokat, datepickereket érdemes jól elkülöníthetően, átláthatóan elhelyezni, hogy a felhasználó számára ergonomikus legyen.
A form kitöltése után a neurális háló irányába elindul egy tanítás, amivel az alap adatok és az ismert betegségek kapcsolatba vonódnak. Ez a funkció azt garantálja, hogy hasonló életmódot élő emberekről legyen egy statisztika amivel a háló fejlődik és minél pontosabbá válik.



10.10. ábra: Fill Form page

10.3.4 Calculate

Ebben a modulban jelenik meg a felhasználó számára elkészült számítások listája. A betegség típusa mellett megjelenő %-os esély jellemzi az aktuális egészségügyi helyzetét. A modulra való kattintással egy GET request indul el a backend betegadatokat kiszolgáló komponense felé. Amennyiben létezik ilyen adat, akkor kérünk egy új kalkulációt a jelenlegi tudása alapján a neurális hálónak. Ezeket az adatokat elmentjük, majd kiszolgáljuk a felhasználó irányába is.

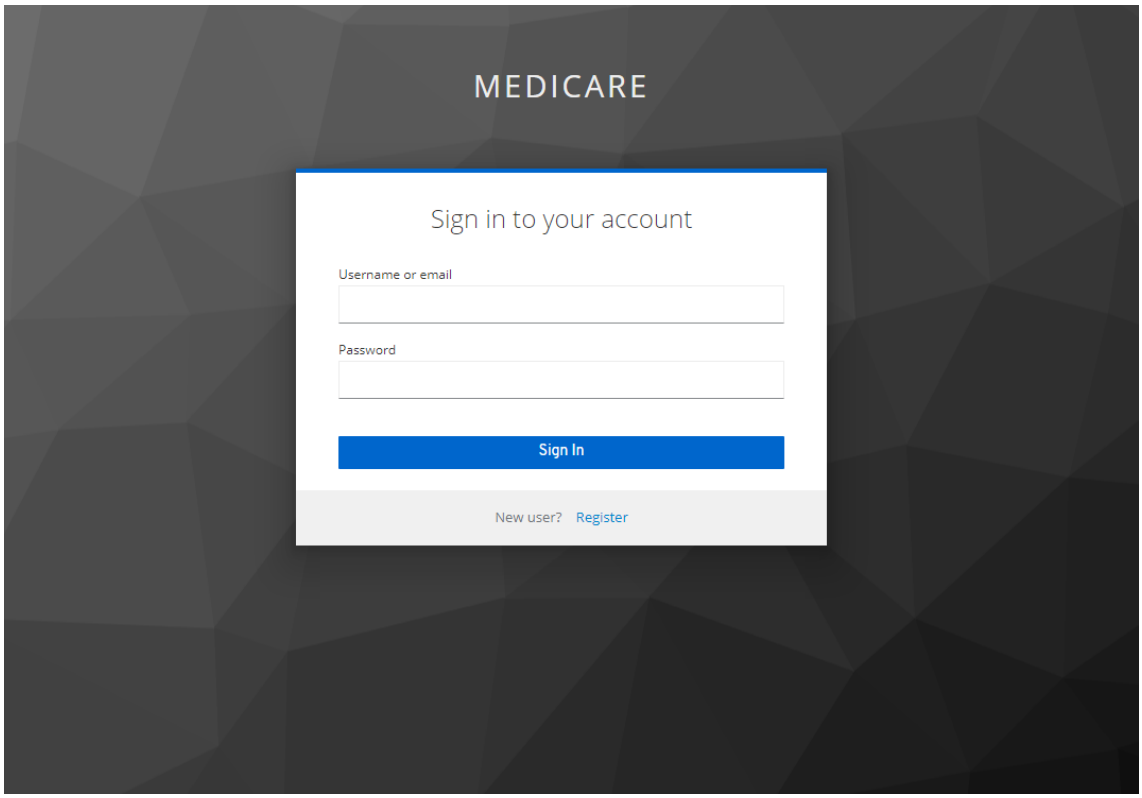


10.11. ábra: Calculate page

10.3.5 Keycloak Integráció

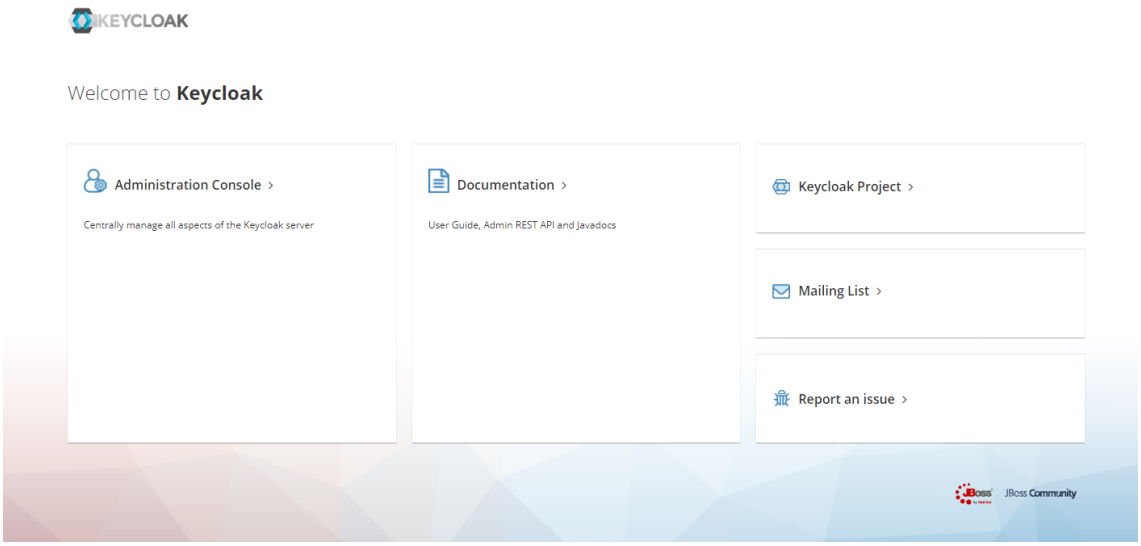
A felhasználói felületre valamilyen módon be kell tudni jelentkezni, hogy a megfelelő embernek jelenjenek meg a számítások és beküldhetőek legyenek az adatok. Ehhez az

alkalmazás indulásakor szükséges átirányítást végezni a ***Keycloak*** bejelentkező felületére. Ezen a felületen nem csak bejelentkezni, de regisztrálni is lehet.



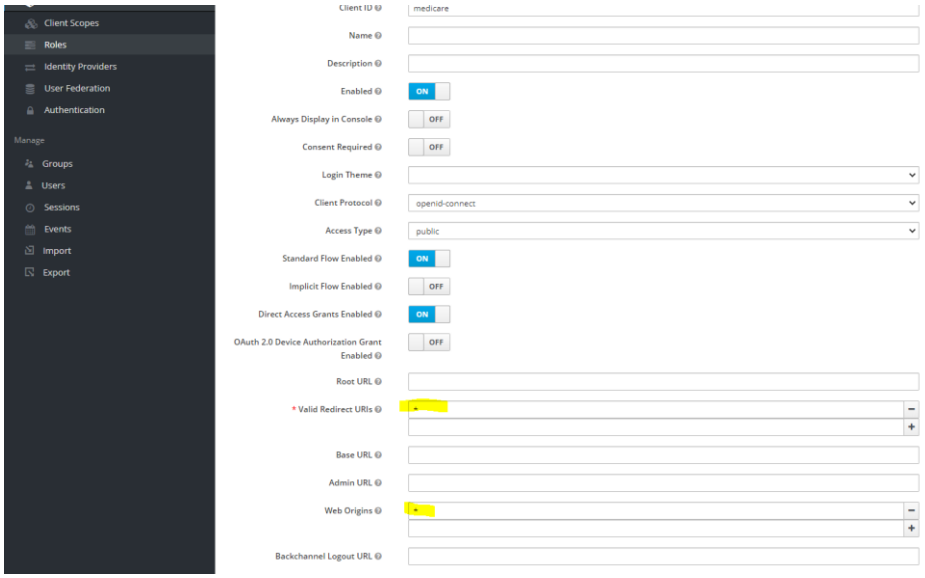
10.12. ábra: Keycloak login page

A későbbiekben lehetőség van ennek a login felületnek a customizálására is, így hasonló designt kaphat, mint az alkalmazásunk többi eleme.
Bejelentkezéshez, felhasználói regisztrációhoz szükség van a ***Keycloak*** konfigurálásához, amit az admin felületen keresztül végezhetünk el. A korábban .env fileban megadott bejelentkezési adatokkal kerülhetünk erre a felületre.



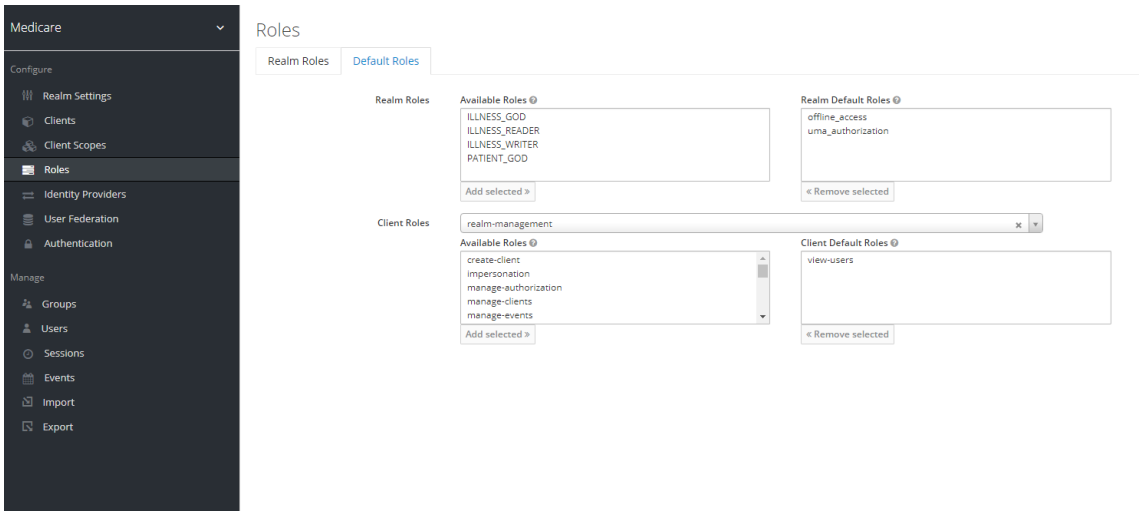
10.13. ábra: Keycloak admin console

Itt létre kell hoznunk a ’Medicare’ realmet, ami a mi fogalmi körünket fogja jelenteni, ezen belül fogjuk elkészíteni a felhasználókat, client-eket és roleokat. Ennek a létrehozása után szükséges egy új client ’medicare’ létrehozása és konfigurálása. A legtöbb beállítás megfelelő számunkra, de a UI redirect miatt szükségünk van a Valid Redirect URIs és a Web Origins esetében a következő kettő beállításra:



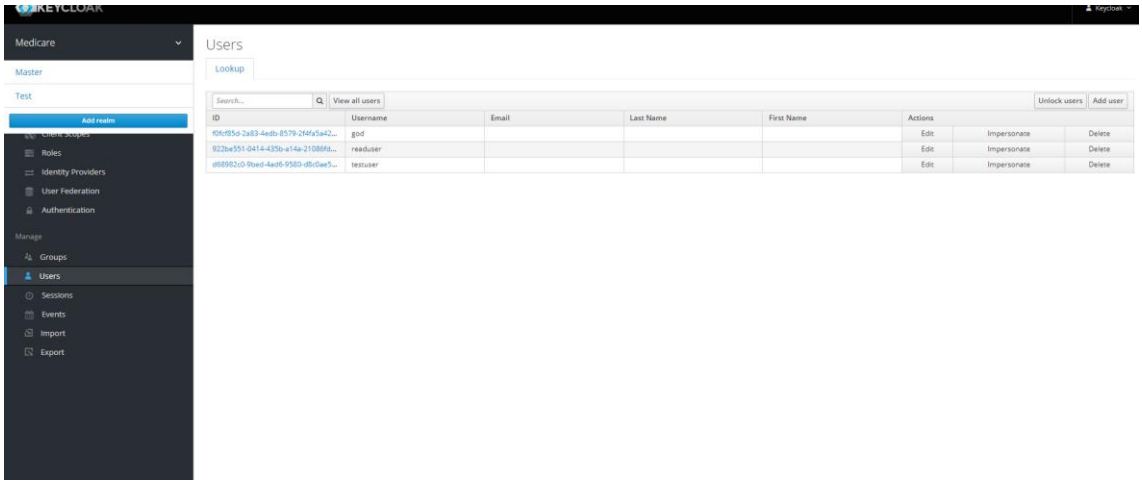
10.14. ábra: Keycloak beállítások

Következő lépésben szükségünk van kiengedni a user meta adatokat is, különben a User Service nem fog megfelelően működni, nem lesz hozzáférése az adatokhoz. Ehhez a Roles menüpont alatt fel kell vennünk egy Default Role-t ami minden felhasználóhoz hozzárendeli ezt az általános lehetőséget. Ez a realm-management alatt található, view-users néven.



10.15. ábra: Keycloak user roles

Ez után teszteléshez létrehozunk néhány felhasználót.



10.16. ábra: Keycloak users

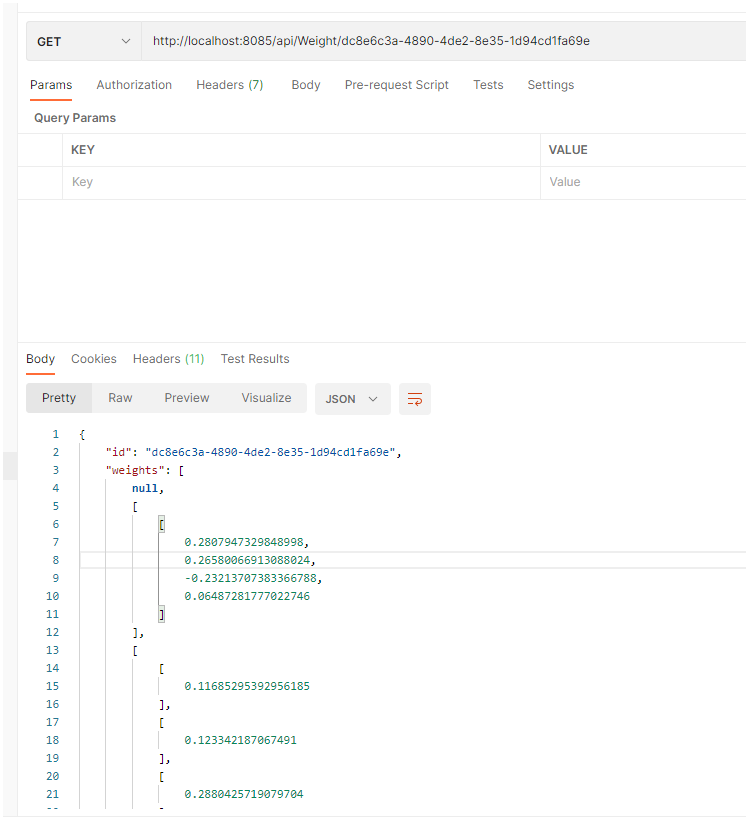
10.4 Neurális Háló

Ennek a komponensnek a működése tekinthető úgy, mint egy feketedoboz. A háló és a neuronok rétegei előre konfiguráltak (itt még mindenképp javítható lenne, hogy a compose yml fileokból töltse be ezt az adatot), a tanítás és kalkuláció elérhető kívülről, és nagyjából ennyit látunk felülről a rendszerről. A hálózat folyamatosan frissíti tudását a súlymátrixba és a bias-ba. Ettől a ponttól kezdve elengedjük a kezét és hagyjuk, hogy fejlődjön és pontosodjon. Természetesen ehhez rengeteg adatra van szükségünk.

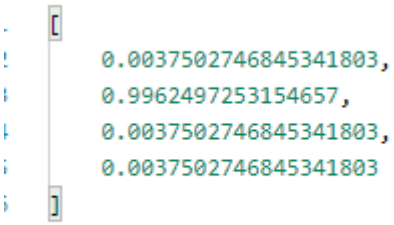
10.4.1 Tanítás

A tanítás megkezdéséhez szükségünk van egy NeuralTransferObject-re, amely tartalmazza a számításhoz elengedhetetlen adatokat, mint a súlymátrix, betegadat, céladat. Ezekkel a bejövő információkkal elkezdődhet a neurális háló műveletvégzése,

amelyik 10 000 alkalommal fut le, hogy mérhető léptékű súlymátrix módosulás történjen. Majd válaszban visszaadjuk ezt a frissített adatot.
Ezt tesztelésre kipróbáltam jóval kisebb adatmennyiségre, hogy jól működik-e a folyamat. Itt az volt a lényeg, hogy egy négy bemenetes neurális hálóval tudok-e közelítőleg hasonló eredményeket elérni. Generáltattam hozzá egy súlymátrixot, biast és ezt lekérve Postman használatával megnéztem az eredményeket.
Majd egy próba metódussal elkezdtem tanítani a hálót arra, hogy a { 0.1, 0.5, 0.2, 0.9 } bemenetekre a { 0, 1, 0, 0 } eredményeket adja vissza. Ezt a tanítást is 10 000 alkalommal futtattam le, hogy jó közelítést kapjunk. Ennek eredménye elég jó becslést adott:



10.17. ábra: Postman hívás és response



10.18. ábra: Teszt adat eredmények

10.4.2 Kalkulációk

A kisebb hálózati méretű tesztek eredménye azt mutatta, hogy a hálózatom jó működésre képes, ezért a részeredményeket értékelve egy finomítás lett szükséges. Ebben a lépésben normalizáltam az adataimat amikkel elkezdtem tanítani az algoritmust. Ez azért volt szükséges, mert olyan nagy szélsőértékek, mint akár egy 100%-os stressz szint, vagy egy 14-re konvertált enum nem eredményezett jól követhető logikai összefüggést, így az összes értéket 0-1 közötti intervallumra helyeztem. A későbbi fejlesztésekhez szükséges lehet olyan tesztek írása, amivel a trivialitásokra kezdjük megtanítani a hálónkat, mint a dohányzás megemeli a daganatos megbetegedések kockázatát.

10.5 Tesztelés

10.5.1 Integrációs teszt

A felhasználói felületről történő adatok beolvasása hibátlanul bekerül az adatbázisba a megfelelő formátumban. A servicek kommunikációja is akadály nélkül működik, így jól látható, hogy alapvető műveleti sorrendek és scenariok végrehajtódnak. Egy teljes integrációs teszt lépései a következők:

Act	Expected	Result
Felhasználói regisztráció	Létrejön a felhasználó	passed
Belépés a felületre	Megjelenik az about oldalon a szöveg	passed
Calculate oldalra navigálás	Calculate oldal üres	passed
Fill Form oldalra navigálás	Fill Form kitölthető	passed
Submit gomb megnyomása kitöltés nélkül	Fill Form nem küldhető be amíg minden szükséges adat nem került kitöltésre	passed
Adatokkal való feltöltés	Minden mező működik és kitölthető	passed
Submit gomb megnyomása	Amennyiben az összes szükséges adat kitöltésre került, elmenthetőek az adatok	passed
Calculate oldalra navigálás	Megjelennek a betegség kategóriák nevei mellett a százalékos valószínűségük	passed
Jobb felső sarokban kilépésre katt	Kilogol a felhasználó	passed
Belépés	About oldal megjelenik	passed
Calculate oldalra navigálás	A korábban látott betegség kategóriák nevei mellett a százalékos valószínűség látható	passed
Fill Form oldalra navigálás	A korábbi kitöltés eredményeivel fel vannak töltve a mezők	passed

26. tábla: Integrációs teszt lépések

A későbbiekben érdemes ezeket a teszteket automatizálni Selenium és TestNG segítségével, így az ilyen általános jellegű folyamatok tesztelése futhat smoke jelleggel. Az így levédett folyamatok beépítésével a CICD pipelineba a további fejlesztések gyors visszajelzést fognak kapni arról, ha regresszió kerül be a fejlesztés során.

11 Összegzés

Az elkészült szoftver már használható, de még elég sok fejlesztési lehetőség rejlik benne amivel pontosabb és gyorsabb lesz mind a számítás, mind a tanulási folyamata. A korábban tervezett eszközkészlethez újat nem volt szükséges bevonni a fejlesztésbe, az előre meghatározott toolok elegendőek voltak a szoftver felépítéséhez. Tanulságos, hogy egy ilyen típusú webalkalmazás felépítéséhez ténylegesen szükséges nagy adatmennyiség, amin tanítható és tesztelhető, különben redukált datasetekkel lehet csak bizonyítani a folyamatok helyes működését. A dockerizált megoldás valóban könnyedén szállíthatóvá tette a szoftvert, bármikor kellett másik gépen elindítani és tesztelni, hibátlanul volt deployolható az alkalmazás. A felhasználói felület fejlesztése, hogy még ergonomikusabb legyen egy nagyobb feladat, kifejezetten erre specializálódott szoftvermérnökök biztosabban és stabilabban el tudják készíteni. Helyenként sok REST hívás közlekedik és bizonyos felületek betöltése is érezhetően enyhén lassú. Ez valószínűleg a nem tökéletesen megfelelő Angular megjelenítő, vagy a frontend adattárolása miatt alakulhatott így. A neurális háló típusa megfelelően lett kiválasztva és a normalizálással elég jó működést eredményez nagyobb adatmennyiségen. Tanulságos, hogy a tanulást érdemes lehet más felbontásokban, kisebb részhalmazokra bontva figyelembe venni, így nem egy nagy egészet próbál megtanulni a rendszer, hanem bizonyos betegségtípusok könnyebben felismerhetővé válnak.

11.1 Továbbfejlesztési lehetőségek

A felhasználói felületen érdemes lenne különböző statisztikai mutatókat megjeleníteni a betegségek kapcsán. Akár kattintható részletek menü a különböző betegségcsoportok kapcsán egy elég hasznos funkció lehetne. A kiértékelést követően a felhasználók egy rosszabb eredmény esetén pedig ajánlást kaphatnának a megfelelő életmódbeli váltásokra, vagy elérhetőséget segítségkérésre. Az oldalon két fajta felületi csoportot lehetne létrehozni; az egyik csak regisztrált felhasználók részére elérhető menüpontokat adna, a másik pedig kívülről is elérhető, érdekes információkat tartalmazna, valamint az about felületet. Külön jogosultság felvételével akár orvosok, szolgáltatók is regisztrálhatnának, és anonim módon hozzáférhetővé, szűrhetővé válnának adatok. Egy szerződési felületre is szükség lenne, ahol a felhasználók elfogadják a felhasználási feltételeket mielőtt beküldenék a kitöltött form-ot. A backend oldalán optimalizálni kellene a kalkulációkat. Ehhez szét kellene választani betegségcsoportokra a kiértékeléseket és így kevesebb adattal, gyorsabb tanulás lenne elérhető. A komplikált rest hívások egyszerűsíthetőek, így bizonyos komponensek összevonhatóvá válnának. Jelenleg egy service fér csak hozzá az adatbázishoz, így az adatoknak keresztül kell haladnia adott esetekben 3 modulon keresztül is mire mentésre kerülnek, viszont ez teljesen felesleges, a persistence rétegben bárhol megtehetnénk ezt. A frontend hívásiláncait is át kellene alakítani ennek következtében.

12 Summary

The developed software is usable but there are many further opportunities left open for faster calculation and learning process as well. No new tool was required for the development beside the previously mentioned ones. A predicitive web application like this requires a tons of data because without that many information we are restricted to a small dataset which is barely sufficient for proving the expected results. Lessons learned. The Docker containerized application proved as a portable and robust solution, as many times I had to develop or demonstrate the software on different devices there were no problem to deploy and run without an error. The frontend development shall be commited for specialized software engineers to be more efficient and ergonomic. This kind of software development expects way more experience than just the know-how of a microservice architecture, it is some kind of an art anyway. There are places where I used too many requests and it takes effects on the display rate. This issue probably was caused by the non-professional usage of the Angular language or the frontend data storing process or both. With big data sets the neural network could be more precise and this is a result of the correct network type which is a feed forward one and the data normalization. In the future I suggest to reduce the processing for smaller subgroups like one illness type, because if we try to see the whole picture at once the details starts to absorb.

12.1 Further development options

The user interface shall have a display for statistics and graphs for the illnes types. There could be a details menu for the illness groups as well. The user could have got a suggestion list for changing the way of living or contact for help after a bad analization result. Adding two more display groups could be also nice; one for the registered users only and an open for all with interesting details and a summary of the about. With an additional user role the doctors and medical workers could register an account to have an interface where the anonim data is available and filterable. Also we shall mention an online engagement for the users about using their private data after submitting the form. The backend side shall be optimalized in terms of the calculations. Separating the illness groups by the input data could improve the accuracy and speed of learning. The overcomplicated rest requests are a bad approach for a microservice application. Most of the data flow is unnecessary because only one module has the responsibility for data storage and this could be moved to every moduls persistance layer. This affects the chaincalls of the frontend as well.

13 Irodalomjegyzék

[1] M. Dery, „5 medical apps that are changing how we diagnose illnesses,” 2017.

Utolsó megtekintés: 2021-10-20

[2] „BiliScreen: Smartphone-Based Scleral Jaundice Monitoring for Liver and Pancreatic Disorders”.

[3] „Instantly diagnose and manage respiratory disease using only a smartphone.”.

[4] „What is PostgreSQL?”.

[5] S. Bhatt, „7 Reasons Java is Perfect for Enterprise Software,” 2021.

[6] „Spring Boot”.

[7] Mohanakrishnan, „15 Reasons Why You Need To Choose Hibernate Over JDBC”.

[8] S. Saini, „8 Proven Reasons You Need Angular for Your Next Development Project,” 2021.

[9] „Maven in 5 Minutes”.

[10] S. Yegulalp, „Why you should use Docker and containers,” 2018.

[11] A. Koserwal, „Keycloak: Core concepts of open source identity and access management,” 2019.

[12] D. K. László, „Neurális hálózatok alapfogalmai I-II,” 2014-10-02.

Utolsó megtekintés: 2021-11-25

[13] S. Sonawange, „Neurális Hálózatok megértése és implementálása Java nyelven,” 2020-06-21.

Utolsó megtekintés: 2021-11-25

[14] „Neurális hálók IBM oktatóanyag,” 2020-08-17.

Utolsó megtekintés: 2021-11-25

[15] A. Vasylenko, „Monolitikus és Microservice alkalmazások,” 2018-10-03.

Utolsó megtekintés: 2021-10-20

[16] M. Fowler, „Microservices,” 2014.

[17] R. Gnatyk, „Microservices vs Monolith: which architecture is the best choice for your business?,” 2018.

[18] K. Erik, „Szerializáció,” 2013.

[19] A. Comsian, „Spring Data JPA Custom queryk és annotációk,” 2019-10-06.

[20] „Hibernate története”.

Utolsó megtekintés: 2021-10-20

[21] „Java és történelme”.

Utolsó megtekintés: 2021-9-10

[22] „Java programozási nyelv”.

Utolsó megtekintés: 2021-9-10

[23] „Spring Boot dokumentáció”.

Utolsó megtekintés: 2021-10-01

[24] „Angular dokumentáció”.

Utolsó megtekintés: 2021-11-12

[25] „Selenium dokumentáció”.

Utolsó megtekintés: 2021-10-20

[26] C. Beust, „TestNG dokumentáció,” 2019-08-20.

Utolsó megtekintés: 2021-10-20

[27] „Maven dokumentáció”.

Utolsó megtekintés: 2021-10-20

[28] V. Mihalcea, „Adatbázis kapcsolatok,” 2019-01-22.

Utolsó megtekintés: 2021-10-20

[29] V. Mihalcea, „JPA és Hibernate ManyToMany kapcsoalt,” 2020-11-19.

Utolsó megtekintés: 2021-9-10

[30] J. Atwood, „Ajánlott olvasmányok fejlesztőknek,” 2004-03.

Utolsó megtekintés: 2021-10-20

[31] R. Fadatare, „Pagination és rendezés Spring Data JPA használatával,” 2020-06-04.

Utolsó megtekintés: 2021-9-10

[32] „Docker dokumentáció”.

Utolsó megtekintés: 2021-12-05

[33] R. Fernando, „Docker konténerek mavenes deploymentje,” 2019-04-13.

Utolsó megtekintés: 2021-12-05

[34] J. Watmore, „Angular 7: Reactive Formok és validáció,” 2019-11-07.

Utolsó megtekintés: 2021-11-12

[35] H. Schmitz, „CRUD applikáció építése Angular és Node használatával,” 2019-11-29.

Utolsó megtekintés: 2021-11-12

[36] „Angular Bootstrap contact form”.

Utolsó megtekintés: 2021-11-12

[37] „Angular CLI dokumentáció”.

Utolsó megtekintés: 2021-11-12

14 Ábrajegyzék

1. ábra: Input adatok transzformációja 6
2. ábra: Neurális hálózat bemenetei és kimenetei 6
6.1. ábra: Mérések eloszlása 12
6.2. ábra: Lineárisan elválasztható minták 12
6.3. ábra: Neuron 13
6. ábra: Neuron függvénye 13
6.5. ábra: Monolith vs. Microservices 15
6.6. ábra: Monolithic Architecture 16
6.7. ábra: Microservice Architecture 17
7.1. ábra: Rendszerterv 18
7.2. ábra: Folyamat terv 20
8.1. ábra: Táblakapcsolat terv 21
8.2. ábra: Active sport activities 22
8.3. ábra: Ancestor death causes 22
8.4. ábra: Ancestor illnesses 22
8.5. ábra: Healthstatus 22
8.6. ábra: Illness 23
8.7. ábra: Known illnesses 23
8.8. ábra: Patient data 24
8.9. ábra: Post sport activities 24
8.10. ábra: Potential illness 25
8.11. ábra: Sport 25
10.1. ábra: Docker-Compose, env, környezet 28
10.2. ábra: Docker Desktop nézet 29
10.3. ábra: Portrainerben futó servicek 29
10.4. ábra: Swagger UI 38
10.5. ábra: Swagger UI request 38
10.6. ábra: About mockup 39
10.7. ábra: Fill Form mockup 40
10.8. ábra: Calculate mockup 40
10.9. ábra: About page 41
10.10. ábra: Fill Form page 42
10.11. ábra: Calculate page 42
10.12. ábra: Keycloak login page 43
10.13. ábra: Keycloak admin console 44
10.14. ábra: Keycloak beállítások 44
10.15. ábra: Keycloak user roles 45
10.16. ábra: Keycloak users 45
10.17. ábra: Postman hívás és response 46

10.18. ábra: Teszt adat eredmények.. 46

15 Táblajegyzék

1. tábla: PatientData DTO 35
2. tábla: Gender 36
3. tábla: Relationship Status 36
4. tábla: Job Type 36
5. tábla: Job Activity 36
6. tábla: Location 37
7. tábla: Race 37
8. tábla: Eating 37
9. tábla: Diet 37
10. tábla: Alcohol Regularity 37
11. tábla: Sport Activity 38
12. tábla: Illness Category 38
13. tábla: Patient API 39
14. tábla: Health Status DTO 39
15. tábla: Potential Illness DTO 39
16. tábla: Health Status API 40
17. tábla: Neural Transfer Object 40
18. tábla: Weight Matrix DTO 40
19. tábla: Weight API 40
20. tábla: Machine Learn API 41
21. tábla: Neural API 41
22. tábla: Keycloak API 42
23. tábla: Keycloak User 42
24. tábla: User API 42
25. tábla: User Meta 42
26. tábla: Integrációs teszt lépések 52