TDK-dolgozat

Antal Krisztián-Tamás

Réthi Anita-Bella

XXXVI. Országos Tudományos Diákköri Konferencia (OTDK)

Marosvásárhely, 2023. április 16-19.

medR

Anamnézis felvételt és betegvizsgálati adatok kezelését segítő szoftverrendszer

Szerzők:

Antal Krisztián-Tamás

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év

Réthi Anita-Bella

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatikai matematika szak, III. év

Témavezetők:

Fosztó Mátyás, szoftverfejlesztő,

Codespring

Máthé Orsolya, szoftverfejlesztő,

Codespring

Dr. Koncz Tamás, szoftverfejlesztő,

Codespring

Dr. Simon Károly, egyetemi adjunktus

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

CODESPRING

OUR CODE YOUR SUCCESS

Kivonat

Az egészségügyi rendszer egyik legnagyobb terhelését a nagy mennyiségű információ nem optimális kezelése adja. A medR projekt az anamnézis felvételében és a betegvizsgálati adatok kezelésében hivatott segítséget nyújtani.

A betegvizsgálat első lépéseiben olyan kritikus információk lehetnek begyűjtve, amelyek meghatározzák a betegség kezelésének a teljes szakaszát. Jelenleg a romániai betegellátó rendszerben körülményes adatrögzítési lehetőségekkel, nehézkes előhívhatósággal, és az átláthatóság hiányával lehet találkozni. Mindez értékes információk elvesztéséhez, helytelen döntések meghozatalához, és nem megfelelő, akár pénzügyi vagy fizikai károkkal járó betegellátáshoz vezethet. A medR webalkalmazás széleskörű adatrögzítésre és saját betegállományra vonatkozó statisztika generálásra ad lehetőséget, ezekben a feladatokban segítséget nyújtva már meglévő rendszerek beépítésével, illetve könnyű és több helyről történő egyidejű használatot biztosít az egészségügyi szakembereknek.

A dolgozatban a rendszer funkcionalitásainak, architektúrájának, a munkafolyamat során felhasznált eszközöknek ismertetésére, és az alkalmazás működésének rövid bemutatására kerül sor.

Tartalomjegyzék

Bevezető **1**

1. Funkcionalitások **3**

2. A medR alkalmazás felépítése **5**

2.1. Szerver 5

2.1.1. Domain adatmodell 6

2.1.2. Adathozzáférési réteg 8

2.1.3. Üzleti logika réteg 9

2.1.4. Kommunikációs réteg 9

2.1.5. Hibakezelés 10

2.1.6. Biztonság 10

2.2. Kliens 11

2.2.1. PWA és WebAPK 12

2.2.2. Komponensek 12

2.2.3. Service osztályok 12

2.2.4. Routing 13

2.2.5. Guarding 14

3. Felhasznált technológiák és eszközök **15**

3.1. Szerver oldali technológiák 15

3.2. Kliens oldali technológiák 16

3.3. Eszközök 18

4. Munkamódszerek **22**

4.1. Scrum 22

4.2. GitFlow 22

4.3. Kódellenőrzés 23

4.4. Folyamatos integráció és kitelepítés 23

5. A medR alkalmazás működése **25**

6. Következtetések és továbbfejlesztési lehetőségek **29**

Bevezető

Az egészségügyi rendszer fő feladata a segítséget kérő beteg problémájának a kezelése. Ez egy folyamat, melynek első lépéseiben az egészségügyi csapat, – azaz az orvos, egészségügyi asszisztens, ápoló, – nagy mennyiségű információt gyűjt össze, majd ezeknek az információknak az alapján kritikus döntések sorozatát hozza meg. Egyértelmű tehát, hogy a betegektől és a vizsgálatokból származó információk helyes rögzítése, rendezése és átláthatósága kritikus a kezelés folyamatában, illetve elmondható, hogy a jelen romániai egészségügyi rendszer egyik legnagyobb terhelése az említett folyamat nem optimális menedzselése [1][26]. A medR az említett adatgyűjtő folyamatból nyert információk rögzítésére, tárolására, illetve előhívására és feldolgozására biztosít megoldást.

Az út, ami a helyes diagnózis megállapításához, illetve az erre alapuló megfelelő kezelés beállításához vezet a következő: az anamnézis felvétele, melynek során a beteg adatainak széleskörű begyűjtése történik, ami magába foglalja a személyes adatokat, az egyéni és családi kórtörténetet, az egyéb társuló egészségügyi adatokat, illetve a beteg aktuális panaszait [24]. Ezt követi a beteg fizikális vizsgálata, azaz az egészségügyi csapat által végzett megtekintésen, hallgatáson, kopogtatáson alapuló vizsgálat. A begyűjtött információra alapozva megszületik egy elsődleges diagnózis, ami alapján további, paraklinikai (labor, orvosi képalkotás stb.) vizsgálatok rendelhetőek el (ld. 1. ábra). Amennyiben a vizsgálatok elrendelése hibás információkra épül, ezek felesleges anyagi terhelést jelentenek a rendszer és a beteg számára is, illetve nagyon sok esetben fizikai károknak tehetik ki a beteget.

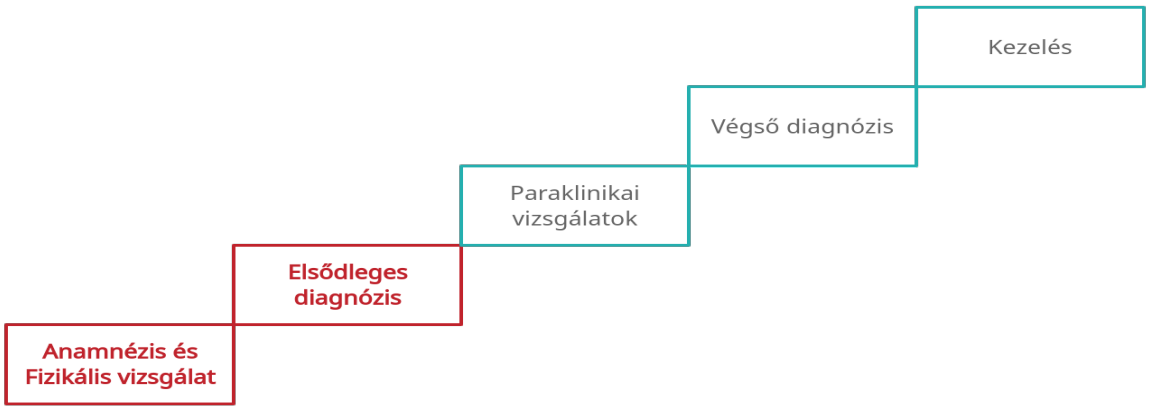
Annak ellenére, hogy már léteznek olyan szoftverrendszerek, amelyek érintik a kezelés említett szakaszait (ld. ERA Medical[1], Teamnet Dedalus[2]), ezek átfogó megoldást hívatottak nyújtani egy beteghez tartozó teljes adathalmaz kezelésére, és gyengeségük az, hogy kevés hangsúlyt fektetnek a kezelés első szakaszaira. Erre alapozva született meg egy olyan alkalmazás terve, amely az említett szakaszokra fekteti a hangsúlyt, és bármilyen platformon működik, amely képes egy modern böngészőt futtatni. Így mobileszközön is használható, például a betegágy mellett, vagy egy asztali számítógépen is a rendelőben. Az alkalmazás használatával az egészségügyi szakembernek lehetősége nyílik saját betegadatbázis felépítésére, a betegekhez tartozó információk rögzítésére, vizsgálatok hozzáadására, illetve ezeknek az adatoknak a szerkesztésére, megjelenítésére. Ugyanakkor az alkalmazás segítséget nyújt

[1] Az ERA Medical hivatalos oldala: http://www.midasoft.ro/modulul-medical/
[2] A Teamnet Group hivatalos oldala: http://www.teamnet.ro/

az adatok értelmezésében, statisztikák, kimutatások generálásával. A betegről begyűjtött információ széleskörű: begyűjthetőek a beteg személyes adatai, saját és családi kórtörténete, allergiák, felírt gyógyszerek, különleges igények, kapott oltások, anamnézis alatt rögzített panaszok és az azokhoz tartozó lokalizáció, erősség, kvalitás, előfordulás gyakorisága, időbeli megjelenése, a vizsgálat alatt felismert különböző életviteli körülmények, illetve más fizikai vizsgálat alatt nyert adatok.

A dolgozat 1. fejezetében az alkalmazás funkcionalitásai kerülnek bemutatásra, a 2. fejezet az architektúrára tér ki, míg a 3. és a 4. fejezetek a felhasznált technológiákat és munkamódszereket részletezik. Az 5. fejezetben az alkalmazás rövid bemutatására kerül sor, végül pedig a 6. fejezet a következtetéseket és a továbbfejlesztési lehetőségeket hívatott tárgyalni.

A fejlesztés 2020-ban, egy Codespringnél végzett nyári szakmai gyakorlat alatt indult, az előzőleg az egészségügyben orvosként tevékenykedő dr. Koncz Tamás kezdeményezésére, aki először fejlesztői, később tanácsadói és mentori szerepet vállalt a projektben. A projekt indítása után egy hónappal, a U-Hub Mentorprogram[3] kereteiben csatlakoztak a csapathoz Réthi Anita-Bella és Antal Krisztián-Tamás. A munkacsoportot Dr. Simon Károly egyetemi adjunktus és a Codespring két munkatársa, Máthé Orsolya és Fosztó Mátyás irányította, mentorálta. A 2020-2021-es egyetemi tanév folyamán a projekt fejlesztése a csoportos projekt tantárgy keretein belül is folytatódott, ahol ideiglenesen csatlakozott a csapathoz Király Éva, Kiss Réka, Orosz Arnold és Ördög Zoltán. Ezúton szeretnénk köszönetet nyilvánítani mentorainknak és mindenkinek, aki a projekthez hozzájárult.



1. ábra. Az út a segítséget kérő beteg érkezésétől egészen a kezelésig. A pirossal jelölt szintek képezik a projekt fő hatókörét

[3] A U-Hub Mentorprogram hivatalos oldala: https://u-hub.ro/

1. Funkcionalitások

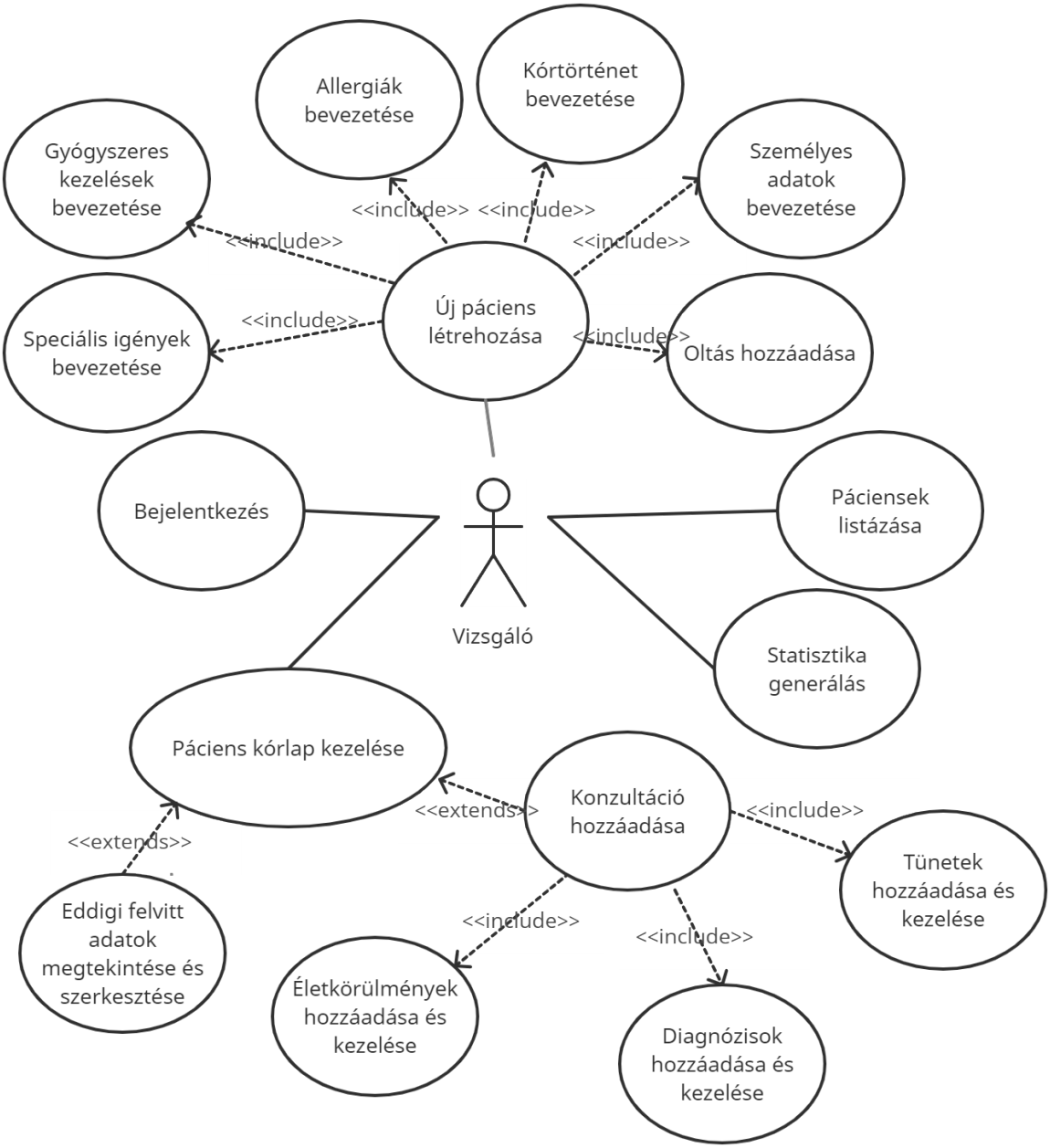
Annak ellenére, hogy az egészségügyi rendszerekben több szerepkör is létezik, a medR alkalmazás fejlesztése során ezek közül egyen volt a hangsúly. Ez a szerepkör a *vizsgáló* lenne, amely az egészségügyi dolgozókat képviseli, tehát lehet orvostanhallgató, nővér vagy orvos (ezen belül pedig rezidens, szakorvos vagy főorvos).

Nem regisztrált felhasználók számára semmilyen funkcionalitás nem érhető el a regisztráción kívül. A vizsgáló miután sikeresen regisztrált és bejelentkezett az oldalra, elérheti és használhatja az alkalmazás összes funkcionalitását (ld. 2. ábra).

Elsősorban megtekintheti a betegeinek listáját, újakat jegyezhet be, vagy törölheti őket. Új beteg bejegyzése során a vizsgáló egy kiterjedt információhalmazt vezethet be az alkalmazásba: a páciens személyes adatait, egyéni és családi kórtörténetét, eddigi ismert allergiáit és gyógyszeres kezeléseit, valamint a különböző speciális igényeit. Egy már meglévő beteg adatlapján ezek a bevezetett adatok lesznek megjelenítve, kiegészítve az eddigi oltásainak és vizsgálatainak listájával. A páciens adatlapját a vizsgáló megtekintheti és szerkesztheti. Azokat az adatokat, amelyeket a vizsgáló fontosnak talál, de egyik fent említett kategóriába sem tudja besorolni, megjegyzés formájában szintén leírhatja.

Új vizsgálat hozzáadásakor a *vizsgáló* egyrészt rögzítheti a betegnél jelentkező tüneteket, ezek lokalizációját, kiváltó- és enyhítő okait, másrészt lehetséges diagnózisokat is hozzájuk tud rendelni. Bevezethetőek a betegnek azok az életkörülményei is, amelyek befolyásolhatják a végső diagnózist (például dohányzás, alkohol, különböző drogok fogyasztása, vagy egyéb tényezők). Miután az orvos sikeresen alátámasztott egy diagnózist vagy tünetet, lezárhatja ezeket, de csak miután megadja a lezárás pontos okát. Lezárás után ez az adat többet nem lesz szerkeszthető, csak megtekinthető. Lezárása után a diagnózis azonnal bekerül a páciens kórtörténetébe.

Egy másik fontos funkcionalitás a statisztikák generálása. Minden vizsgáló a saját betegállománya alapján készíthet el különböző típusú statisztikákat, melyek könnyen áttekinthető ábrák formájában fognak megjelenni.



2. ábra. Use case diagram: A medR alkalmazás sikeresen implementált funkcionalitásai.

2. A medR alkalmazás felépítése

A medR alkalmazás kliens-szerver architektúrára épül. A *Spring* keretrendszerrel implementált szerver egy *REpresentational State Transfer* (REST) [23] *Application Programming Interface*-t (API) biztosít, amin keresztül elérhetőek a szerver saját, *PostgreSQL* adatbázisban tárolt adatai, valamint külső szolgáltatásoktól származó adatok. A kliens egy *Angular* keretrendszerrel létrehozott *Progressive Web Application* (PWA). A két alkotóelem közötti kommunikáció *REST* hívásokon keresztül valósul meg, és a *Data Transfer Object* (DTO) tervezési mintán alapszik.

A szerver jelenlegi változata egy külső szolgáltatással kommunikál: az *Amerikai Egyesült Államok Nemzeti Egészségügyi Intézet*hez (NIH) tartózó, *Orvostudományi Nemzeti Könyvtár* (NLM) által szolgáltatott API-val [28]. Az innen érkező adatok a tünetek és diagnózisok nemzetközi elnevezéseinek és *ICD-10* (International Statistical Classification of Diseases and Related Health Problems) kodifikációjaira vonatkoznak. Ez egyrészt azt biztosítja, hogy a rendszerbe felvezetett tünetek és diagnózisok megnevezései és kodifikációi megfeleljenek a nemzetközi standardoknak, másrészt kereső-kiegészítő funkcióként is működik kliens oldalon.

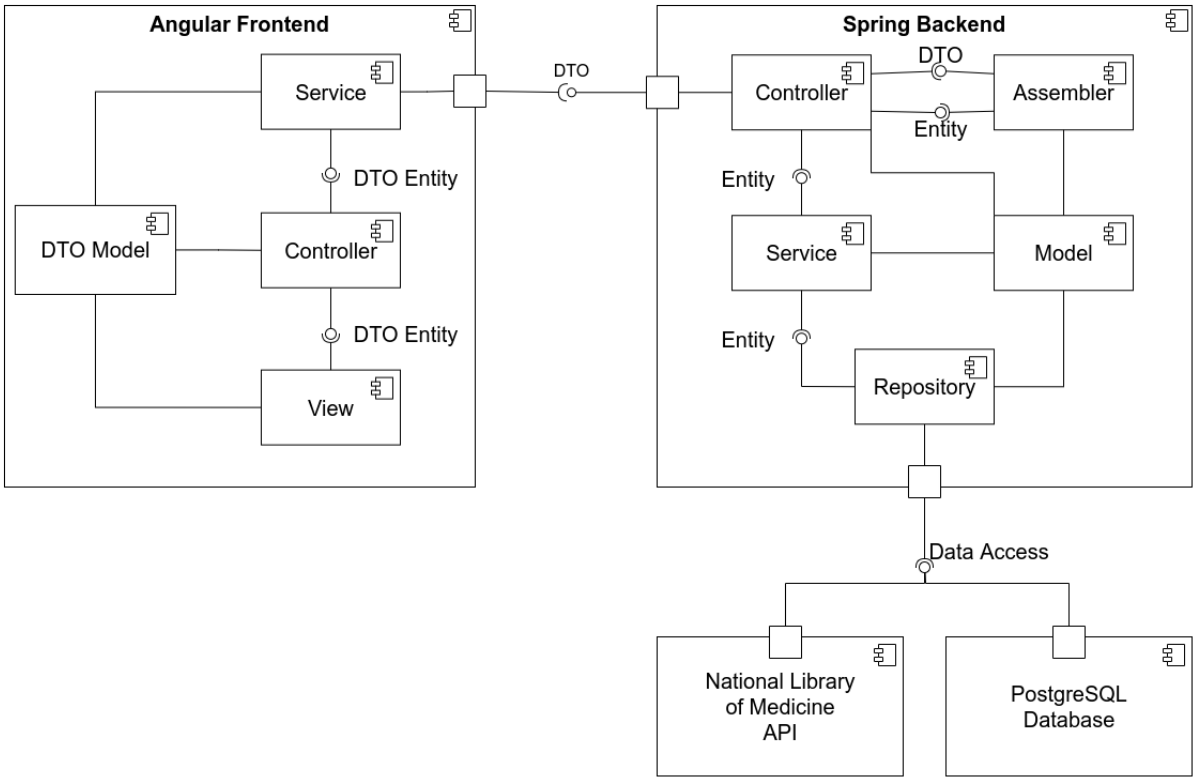
Az alkalmazás architektúráját szemlélteti a 3. ábra, az említett alkotóelemek pedig a továbbiakban lesznek részletezve.

2.1. Szerver

A szerver többrétegű architektúrára épül. A *perzisztens* adatmodell az adatbázis szerkezetét meghatározó elemek összessége, míg a *domain* adatmodell ennek a struktúrának egy sajátos formája, amely azokból a modell osztályokból tevődik össze, amelyek az alkalmazás legalapvetőbb adatait képviselik, és az üzleti logika megvalósításában játszanak szerepet. A prezisztens modell létrehozásáról a *Spring Data JPA* gondoskodik, a domain modell osztályaira helyezetett annotációk alapján.

A többrétegű architektúra könnyebb kód karbantarthatóságot biztosít, könnyít a hibák azonosításán és kijavításán, illetve, mivel a rétegek közötti kommunikáció interfészek révén valósul meg, szükség esetén egy-egy réteg könnyen módosítható, vagy akár teljesen ki is cserélhető.

A legalsó réteg a *repository*, az adathozzáférési réteg, az adatbázissal való kommunikációt valósítja meg, valamint a perzisztens és domain modellek közötti átalakítást, így rajta keresztül



3. ábra. Az alkalmazás architektúrája, komponens diagramm

érhetőek el az alapvető *CRUD* (Create, Read, Update, Delete) műveletek.

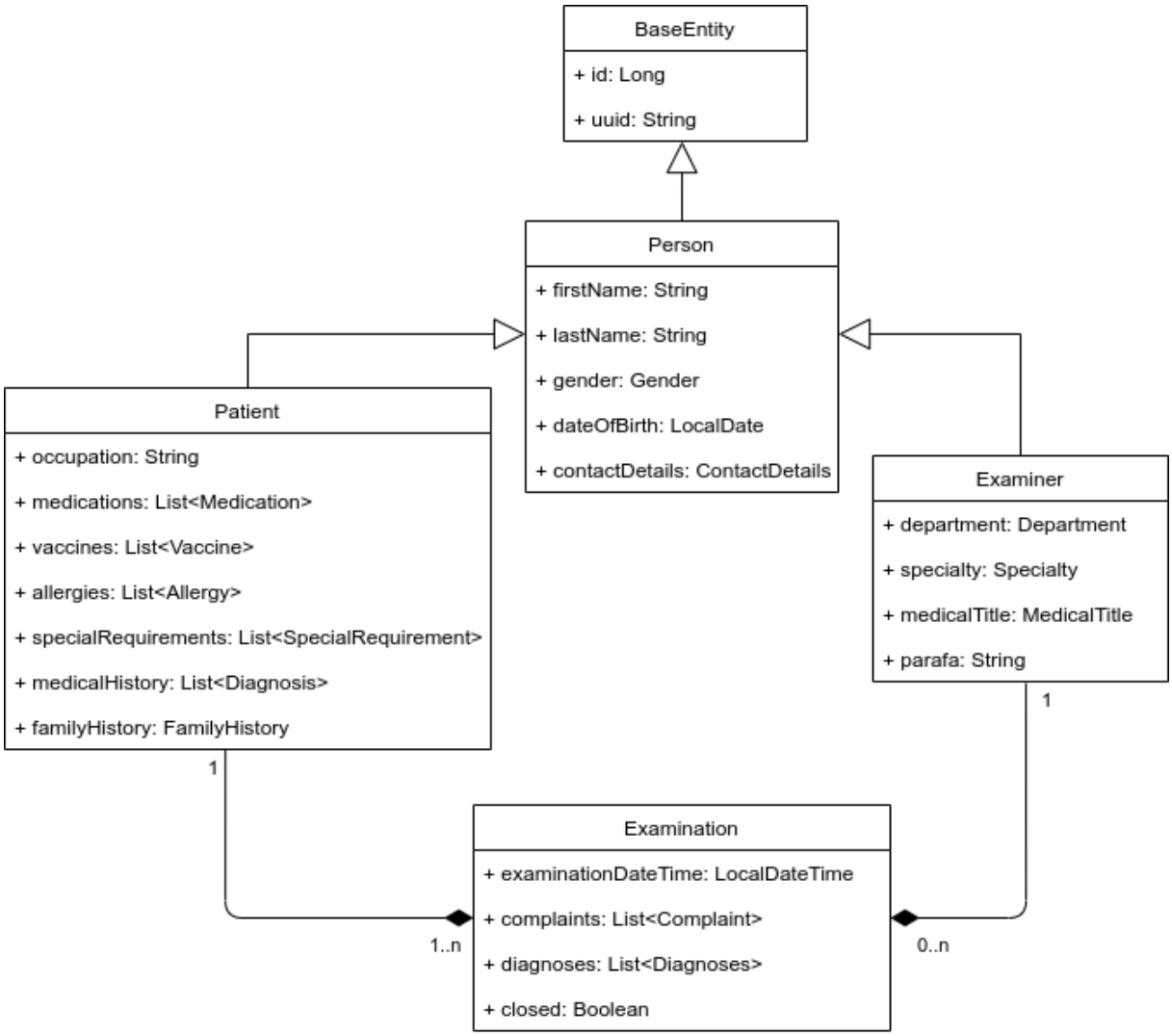
A *service* rétegben található az üzleti logika, illetve ez a réteg összeköttetést képez a *repository* és a felsőbb rétegek között.

A szerver legfelső rétege a kommunikációért felelős *controller* réteg. Ez a réteg felel a REST kérések fogadásáért és a válaszok generálásáért. A válaszok DTO-kból állnak, melyek a modellek specifikus, különböző adatokat mellőző, vagy akár bizonyos adatokat a modellekhez hozzáadó reprezentációi. A DTO-k létrehozását *assemblerek* biztosítják, melyek a *service* rétegtől érkező adatokat alakítják, rendszerezik DTO-kba.

2.1.1. Domain adatmodell

A *domain* adatmodellen belül két fogalmi kör van megkülönböztetve: a személyekkel kapcsolatos információt hordozó modellek és a betegségekkel kapcsolatos modellek. A személyekkel kapcsolatos információk két alkategóriáját alkotják a betegek, valamint a vizsgáló köré csoportosuló információk (ld. 4. ábra).

Egy személy esetében az általános adatok közé bevezetődnek a személyi adatok, lakcím és elérhetőségek. Az első alkategória részeként, a betegekhez tartoznak egészségügyben releváns



4. ábra. A páciensek és vizsgálók köré csoportosuló információk és az ezek közötti kapcsolatok

szociális adatok, mint a családi állapot vagy foglalkozás, illetve konkrétan egészségügyi adatok, mint a személyes és családi kórtörténet, allergiák, gyógyszerelések, oltások vagy különleges igények. A vizsgálók esetében rögzítve lesz az egészségügyi rendszerben betöltött funkció (orvos, nővér, ápoló, diák) és az egészségügyi egység (rendelő, kórház) amihez tartozik az illető szakember, kórházak esetében megemlítve a kórházi osztályt is, ahol a vizsgáló dolgozik.

A betegséggel kapcsolatos körben a fő entitás a vizsgálat. Egy vizsgálat panaszok halmazából, életviteli körülményekből és a vizsgálathoz tartozó diagnózisokból áll. Egy panaszhoz tartozik az adott tünet megnevezése, a tünet ICD10 kódja és a panasz karakterisztikái, azaz a lokalizációja, súlyossága, időzítése, frekvenciája, illetve enyhítő és súlyosbító faktorok. Az életviteli körülmények olyan elemek, melyek a páciens életét, egészségét befolyásolhatják, vagy összefüggésben lehetnek a konkrét panaszokkal. Ide tartoznak például a páciens dohány-, kábítószer- vagy alkoholfogyasztással kapcsolatos szokásai.

```
public interface PatientRepository extends JpaRepository<Patient, Long> {
    @Query("select distinct e.patient from Examination e "
        + "where e.examiner = :examiner")
    List<Patient> findByExaminer(@Param("examiner") Examiner examiner);
}
```

1. kódrészlet. Patient repository metódus @*Query*-vel

2.1.2. Adathozzáférési réteg

A szerver és az adatbázis közötti kommunikáció egy *Object Relational Mapping* (ORM) keretrendszer segítségével van megvalósítva, pontosabban a *Java Persistence API* (JPA) *Hibernate* implementációjával. Az erre építő *Spring Data JPA*-nak köszönhetően, a réteg osztályainak implementációja nem tartozik a fejlesztő feladatai közé, csupán olyan *interface*-eket kell létrehozni, melyek a *JpaRepository* leszármazottjai és rendelkeznek a @*Repository* annotációval. A *Spring Boot* autokonfigurációs mechanizmusának köszönhetően, ezeket az *interface*-eket megtalálja a rendszer, a *Spring Data JPA* pedig generálja a megfelelő implementációkat.

A *JpaRepository* hozza magával az alapvető *CRUD* (Create, Read, Update, Delete) műveleteket, de ezen kívül több módszer is biztosítva van egyedi lekérdezések létrehozására. Az első ilyen módszer a lekérdezések automatikus generálása az interfészben deklarált *metódusok* nevei alapján. Amennyiben egy metódus neve túl bonyolult lenne, vagy a lekérdezést komplexitása miatt nem lehet ilyen módon definiálni, lehetőség nyílik *Java Persistence Query Language*-ben (JPQL) megírt lekérdezések definiálására, a @*Query* annotáció használatával a megfelelő metódus fölött, (ld. 1. kódrészlet).

Továbbá a *Spring Data JPA* lapozást és rendezést biztosít a lekérdezések szintjén a *Pageable* és a *Sort* interfészek használatával. Ennek segítségével egyrészt biztosítani lehet azt, hogy bizonyos lekérdezések nem terhelik túl a rendszert, másrészt megoldható az adatok rendezése bizonyos releváns kritériumok alapján (pl. a pácienseket születési dátum szerint).

Ugyancsak az ORM keretrendszer felelős az adatbázis alapstruktúrájának generálásáért. A domain modell osztályaira helyezett annotációkból nyert információk alapján az ORM keretrendszer képes értelmezni a modelleket, leképezve az osztályokat táblákra, az adattagokat pedig oszlopokra, vagy kapcsolódási táblákra, ha modellek közötti kapcsolatokról van szó.

2.1.3. Üzleti logika réteg

Az üzleti logika réteg felelős az adathozzáférési rétegtől érkező adatok feldolgozásáért és rendszerezéséért, a kommunikációs rétegnek küldött válaszok létrehozása érdekében. A rétegek közötti összeköttetés interfészek segítségével valósul meg, így ezen a rétegen belül is interfészekről beszélünk, illetve az ezek mögötti projekt-specifikus implementációkról, melyek *Dependency Injection* mechanizmus segítségével férnek hozzá az adathozzáférési réteg szolgáltatásaihoz. A *@Service* annotáció jelzi a Spring keretrendszernek, hogy injektálható osztályról van szó, így a rendszer indulásakor példányosítja ezeket, és a példányokat szolgáltatja azoknak az osztályoknak, amelyekben jelezve van ezeknek a szükségessége.

Azon kívül, hogy összeköttetést képez a *repository* és *controller* rétegek között, a *service* réteg több feladatot is végez. Ez a réteg felelős többek között a betegek diagnózisaiból és tüneteiből összeállított statisztikák generálásáért, a *National Library of Medicine* API-tól érkező adatok feldolgozásáért, és a lapozáshoz, illetve rendezéshez szükséges *Pageable* és *Sort* objektumok létrehozásáért bizonyos kéréseknél. A rendszer biztonságával kapcsolatos műveletek lebonyolítása is itt történik, amiről bővebben a 2.1.6 alfejezetben van szó.

2.1.4. Kommunikációs réteg

A klienssel való kommunikáció a *controller* rétegen keresztül valósul meg, figyelembe tartva a REST konvenciót. Külön vezérlő osztály van hozzárendelve minden entitáshoz, amelynek elérhetőnek kell lennie a kliens számára. A vezérlők a *@RestController* annotációval vannak jelölve, illetve a *@RequestMapping* annotációval, amelynek értéke a vezérlő útvonalát határozza meg.

Az entitás-specifikus vezérlőkön kívül, autentikációs, valamint statisztikák és *National Library of Medicine*-től származó adatok szolgáltatására is vannak kontrollerek. Minden vezérlő osztály egy bizonyos adathalmazt tesz elérhetővé, kérés típusról (GET, PUT, POST, DELETE) és pontos útvonalról információt tartalmazó annotációval ellátott metódusok révén. A metódusok a *service* osztályoktól kapott adatokat *assemblerek* segítségével DTO-kká alakítják, majd *ResponseEntity* formájában elküldik válaszként, szükség szerint kiegészítve ezeket extra adatokkal, HTTP fejlécekkel.

```
@ExceptionHandler(ApiException.class)
public ResponseEntity<Object> handleApiException (ApiException ex,
WebRequest request) {
    return handleExceptionInternal(ex,
        Collections.singletonMap("causes", getCauses(ex)),
        new HttpHeaders(), ex.getStatus(), request);
}

private List<String> getCauses(Throwable throwable) {
    List<String> causes = new LinkedList<>();
    Throwable localThrowable = throwable;

    while (localThrowable != null) {
        causes.add(localThrowable.getMessage());
        localThrowable = localThrowable.getCause();
    }
    return causes;
}
```

2. kódrészlet. ApiException-t kezelő metódus. A dobott kivétel okai el lesznek küldve a kliensnek, a megfelelő státuszkóddal.

2.1.5. Hibakezelés

Az alkalmazás hibakezelését egy *ApiExceptionHandler* nevű osztály végzi. Ez a *ResponseEntityExceptionHandler* absztrakt osztályból származik, valamint a *@ControllerAdvice* annotációval rendelkezik. Az osztály minden metódusa *@ExceptionHandler* annotációval van megjelölve, és paraméterként valamilyen kivétel típusát fogja megkapni.

A többrétegű architektúrának köszönhetően a dobott kivételek egy lánchoz hasonló utat fognak bejárni. Mindegyik réteg külön-külön le fogja kezelni a rétegen belüli kivételeket, majd egy sajátos típusú kivételt fog továbbdobni a soron következő rétegnek, végül pedig az *ApiExceptionHandler* fogja ezt a kivételt megkapni. A kapott adatok alapján egy *ResponseEntity* objektum jön létre, tartalmazva többek között a kivétel okát, helyét és a hiba státuszkódját. Ezt az objektumot fogja a szerver visszaküldeni a kliensnek. A 2. kódrészleten egy ilyen metódus van szemléltetve.

2.1.6. Biztonság

Mivel a projekt egyik fő célja a nagy mennyiségű érzékeny információ tárolása (páciensek személyes adatai), fejlesztés közben nagy hangsúly került a biztonságra. Az implementáció a

Spring Security keretrendszer segítségével történt, amely főleg autentikációval és autorizációval foglalkozik. A *Spring Security* egy nagyon rugalmas keretrendszer, a medR esetében a *WebSecurityConfigurerAdapter* absztrakt osztály van kibővítve annak érdekében, hogy a megfelelő konfigurációt lehessen kialakítani.

Az első funkcionalitás, amelynek szempontjából elengedhetetlen a biztonság, az a regisztráció. Ekkor a felhasználó által beírt jelszó azonnal titkosításra kerül, és az adatbázisba is ilyen formában mentődik el. A jelszavak titkosítására egy *BCryptPasswordEncoder* példány van felhasználva, ami a *PasswordConfig* osztályban van beállítva.

Minden felhasználó bejelentkezésekor a *JwtTokenProvider* egy *JSON Web Token*-t [9] generál, amelyben megtalálható egy a felhasználó azonosításához szükséges adat (a medR esetében ez az email cím), az időpont amikor létrejött és a lejárási idő. A token egy süti (*cookie*) formájában van visszaküldve a kliens oldalnak a válasz fejlécében, valamint ez alapján lesz később azonosítva az adott felhasználó, amikor majd hozzá szeretne férni valamilyen útvonalhoz bejelentkezés után. A token lejárta után a felhasználónak újra be kell jelentkeznie.

Az */auth/*** mintára illeszkedő útvonalak elérhetőek bárki számára, itt a három alap funkcionalitás található: regisztrálás, bejelentkezés és kijelentkezés. Ha más formájú útvonalra érkezik kérés, akkor ezek egy *JwtRequestFilter*-hez lesznek átirányítva, ami azt ellenőrzi, hogy a kérést intéző felhasználó be van-e jelentkezve. Ennek érdekében első lépésben azt fogja ellenőrizni, hogy a kérés fejlécében létezik-e olyan süti, ami egy *JWT*-t tartalmaz. Ha talál ilyent, leellenőrzi, hogy érvényes-e még, és csak abban az esetben fogja engedélyezni az útvonal elérését, ha nem talál semmilyen hibát. Ellenkező esetben törli a sütit, és egy üzenet lesz visszaküldve a kliensnek, ami a 403-as státuszkódot fogja tartalmazni.

2.2. Kliens

A kliens oldal egy *Angular* keretrendszerben fejlesztett *Single Page Application*, aminek az egyik legnagyobb előnye az, hogy a teljes oldal csak egyszer töltődik be, később pedig csak az a része fog frissülni ahol változik a tartalom. Az oldal első megnyitásakor az összes szükséges HTML, CSS és JavaScript állomány betöltődik, majd ezt követően csak a megjelenítésre váró adatok fognak érkezni a szervertől. Emiatt az alkalmazás gyorsabb, valamint könnyebben navigálható, mint a többoldalas webalkalmazások.

2.2.1. PWA és WebAPK

A medR projekt fejlesztése során a csapat nagy hangsúlyt fektetett arra, hogy az alkalmazást bármilyen készülékről (asztali gép, táblagép, okostelefon) kényelmesen lehessen használni. Ezért a kliens oldal egy *progresszív webalkalmazás* (PWA), aminek előnye, hogy a natív alkalmazásokhoz hasonlóan működik, anélkül, hogy szükség lenne minden különböző platformra, operációs rendszerre egy-egy specifikus verziót létrehozni. Ez nem csak a fejlesztési folyamatot egyszerűsíti le, hanem a teljes fejlesztés idejét is lerövidíti.

A progresszív webalkalmazások fontos jellemzője, hogy amikor böngészőből vannak megnyitva, telepíteni lehet ezeket az adott eszközre. Ezt követően, egy natív alkalmazáshoz hasonlóan, meg fog jelenni a főoldalon az applikáció ikonja. Ezt a feltelepített verzióját az alkalmazásnak akár internetkapcsolat nélkül is lehet használni, mivel a már elmentett adatok így is elérhetőek, böngészhetőek.

A *WebAPK* egyfajta továbbfejlesztése a PWA-nak, ami abból az ötletből indult, hogy ezek az alkalmazások valamilyen módon legyenek becsomagolva egy APK-ba (*Android Application Package*). Így ezek az alkalmazások még az eddiginél is jobban fognak illeszkedni az Android operációs rendszerre, gyakorlatilag ugyanúgy viselkednek, mint a natív alkalmazások. [3]

2.2.2. Komponensek

Egy Angular alkalmazás úgynevezett építőkockái a komponensek, az Angular komponensekből felépülő faszerkezet pedig maga az alkalmazás. Minden Angular komponens egy *TypeScript* osztály, amelyet a *@Component* dekorátor jelez.

A projekt elkészítése közben nagy hangsúlyt fektetett a csapat arra, hogy a gyakran használt, akár kisebb elemeket is külön válassza, és saját komponensekbe foglalja. Ennek oka nem csak a kódismétlés elkerülése volt, hanem így többször felhasználhatóvá vált az adott logikájú komponens, ezzel meggyorsítva a fejlesztés folyamatát. Ennek a meggondolásnak az alapján jöttek létre a medR központi komponensei: a gombok, táblázat, páciens kártyák, adatlap kártyák, különböző típusú menük stb.

2.2.3. Service osztályok

A szerver oldali API-val történő kommunikáció elkülönítésének érdekében *service* osztályok voltak létrehozva a kliens oldalon. A *service*-ek olyan *TypeScript* osztályok, amelyeket az *@Injectable()* dekorátorral jelölnek és a DI (*Dependency Injection*) tervezési

```
@Injectable({ providedIn: SharedModule })
export class ExaminerService {
  constructor(private http: HttpClient) { }

  getExaminer(): Observable<Examiner>{
    return this.http.get<Examiner>('api/examiners');
  }

  updateExaminer(examiner: Examiner): Observable<Examiner>{
    return this.http.put<Examiner>(`api/examiners/${examiner.id}`, examiner);
  }
}
```

3. kódrészlet. Vizsgálókkal kapcsolatos adatok lekérésére és küldésére alkalmas *ExaminerService* osztály

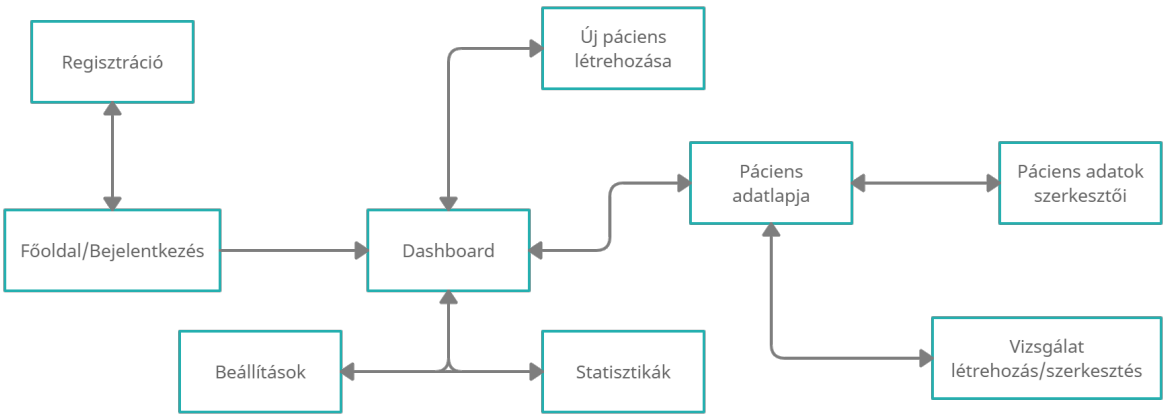
minta mechanizmusát alkalmazzák. Ezeket a serviceket fel lehet használni bármely komponensben, az osztály megadásával az adott komponens konstruktorában. A felhasználható példányt a keretrendszer fogja szolgáltatni.

A medR projektben ezek az osztályok főként a szerver oldali DTO-k lekérésére és küldésére voltak használva, amelyekhez hasonló, vagy akár identikus *TypeScript*-ben írt modellek voltak létrehozva a kliens oldalon. Erre egy példa lenne a 3. kódrészleten szemléltetett *ExaminerService*, ami vizsgáló adatokat küld és fogad.

2.2.4. Routing

Mivel az Angular alkalmazás komponensek összessége, és minden komponenshez tartozik egy saját nézet, szükség van valamilyen módszerre ahhoz, hogy a különböző nézetek közötti navigációt meg lehessen valósítani, egyeseket megjelenítve, míg másokat eltüntetve az oldalról. A kliens, ahogy az alfejezet legelején is megemlítésre került, egy *Single Page Application*, tehát nem a szerver fog új megjelenítésre váró oldalakat küldeni, hanem ezek már az oldal megnyitásakor betöltődtek, ezért egy specifikus *routing* mechanizmus szükséges.

Ezt a nézetek közötti navigációt az *Angular Router* segítségével lehet megvalósítani. Minden komponenshez egy útvonalat lehet rendelni az *AppRoutingModule*-ban, ezek az útvonalak pedig a böngészőbeli URL-t interpretálják. Az 5. ábrán a medR navigációs útvonalai vannak szemléltetve, hogy melyik nézetről pontosan hova lehet eljutni.



5. ábra. A medR navigációja.

2.2.5. Guarding

Az Angular *Route Guard* egy olyan logika vagy funkcionalitás, amely a kiválasztott útvonal betöltése előtt vagy az útvonal elhagyásakor van végrehajtva. Ezek a *CanActivate*, *CanLoad*, *CanDeactivate* metódusokat implementálhatják, annak függvényében, hogy pontosan milyen célból és mikor kerülnek használatra.

A medR esetében arra voltak ezek használva, hogy a kliens oldalon a nem felhatalmazott személyek ne tudjanak hozzáférni bizonyos nézetekhez, ehhez pedig a *CanActivate* és *CanLoad* metódusokat kellett kibővíteni. Az útvonal őröket az *AppRoutingModule*ban lehet felhasználni, és bármelyik útvonalhoz hozzá lehet ezeket rendelni. Például, ha egy nem bejelentkezett felhasználó meg próbálja nyitni a páciensek listáját, a *Route Guard* vissza fogja irányítani a főoldalra, egy megfelelő hibaüzenettel.

3. Felhasznált technológiák és eszközök

A projekt megalkotása során felhasznált technológiák és eszközök sokszínűek, kiválasztásuk nagy hangsúlyt kapott a tervezési fázisok során. A szerver fejlesztése *Java* nyelvben történt a *Spring* keretrendszer segítségével, a web kliens pedig a *HTML* (HyperText Markup Language), *Sass* (Syntactically Awesome Style Sheets) és *TypeScript* nyelvhármasban volt megírva, használva az *Angular* keretrendszert.

A következő alfejezetekben röviden bemutatásra kerülnek az említett, illetve további felhasznált technológiák és eszközök.

3.1. Szerver oldali technológiák

PostgreSQL

A *PostgreSQL* egy nyílt forráskódú objektum-relációs adatbázis rendszer. A *PostgreSQL* használata mellett álló fő érvek a rendszer architektúrája, a több, mint harminc év alatt bizonyított megbízhatósága és a kiterjeszthetősége, lehetőséget adva a felhasználónak saját adattípusok, indextípusok és bővítmények létrehozására. [13]

Objektum-relációs adatbázis mivolta miatt megfelelő választásnak bizonyul az erősen struktúrált egészségügyi adatok tárolására. Ezeknek az adatoknak a tárolása táblákban történik, ahol az oszlopok az entitások attribútumait, a sorok pedig egy-egy entitáshoz tartozó attribútumok konkrét értékeit képviselik.

Spring Boot

A *Spring* egy nyílt forráskódú, moduláris Java keretrendszer. Megvalósítja az *Inversion of Control* elvet és a *Dependency Injection* tervezési mintát, menedzselve a rendszer objektumainak életciklusát és az ezek közötti függőségi kapcsolatokat. Modularitása abból adódik, hogy a fejlesztők szükségtől függően adhatják hozzá egy projekthez a keretrendszer önálló részeit [16], ezzel elkerülve a felesleges függőségekkel együtt járó teljesítmény- és tárhelybeli hátrányokat.

A keretrendszer támogatja az annotációk használatát bizonyos konfigurációk beállítására, viszont sok funkcionalitást nehézkesen, kódból vagy *Extensible Markup Language* (XML) állományok használatával lehet csak beállítani [31]. Ennek a problémának a megoldására jött létre a *Spring Boot* projekt, aminek célja egy olyan alapkonfiguráció és függőséglista

biztosítása, amely a *Spring Boot* fejlesztői szerint elegendő kellene legyen ahhoz, hogy levegye a konfigurálás nehézségének zömét a keretrendszert használó fejlesztők válláról [30]. A Spring Boot projekt továbbá *starter*-nek nevezett függőségcsomagokat tart karban, melyek teljes feladatkörök megoldására hívatottak. Az egyik ilyen a medR projektben használt *Spring Boot Starter Web*, amely hozza magával a *Tomcat* webkonténert és *RESTful Spring MVC* applikációk létrehozására alkalmas.

Spring Data JPA

A *Spring Data JPA* a *Spring Data* projektcsalád része, és célja a JPA alapú adatelérés implementációjának megkönnyítése [30]. A modul egy absztrakciós szintet biztosít a *Hibernate* ORM keretrendszer fölött, annotációkkal vagy metódusnévből következtetett implementáció-generálással könnyítve a *repository* osztályok létrehozását. A modul használatát és a medR projektben felvett szerepét a 2.1.2. fejezet részletezi.

Spring Security

A rendszerben tárolt információkhoz való hozzáférés a *Spring Security* modul segítségével van menedzselve. A páciensek adataihoz csak regisztrált, azonosított felhasználók férhetnek hozzá, nem regisztrált felhasználóknak csak a regisztráció és a belépés elérhető.

A modul segítségével konfigurálhatóak az egy-egy útvonalra érkező kérésekhez szükséges jogosultságok, illetve további ellenőrzések is beékelhetőek a meglévők mellé, ilyen a medR-ben használt *JSON Web Token* (JWT) alapú igazoltatás is. A jelszavak titkosítása is megvalósítható, a *Spring Security* több jelszótitkosító implementációt hoz magával [27], melyek közül a medR szervere a *BCryptPasswordEncoder*-t használja. A *Spring Security* használatáról és működéséről a projekten belül a 2.1.6. fejezet számol be.

3.2. Kliens oldali technológiák

Angular

Az *Angular 2+* [6], egy a Google által fejlesztett komponens alapú keretrendszer, aminek használatával könnyen skálázható *SPA*-k (*Single Page Application*) fejlesztését lehet megvalósítani. A keretrendszer a *Model-View-Controller* és *Dependency Injection* tervezési mintákat valósítja meg azáltal, hogy a *business* réteget *TypeScript* nyelvben írt *service*-ekbe

és modellekbe csomagolja. Könyvtárak gyűjteményét biztosítja a fejlesztők számára, amelyek nagyban megkönnyítik az alkalmazások létrehozását. A medR projektben használt Angular könyvtárakról bővebben a 2.2 fejezetben volt szó.

Minden Angular komponens három különböző típusú állományból tevődik össze. Az első egy *HTML* típusú *template*, amit különféle Angular direktívákkal lehet dekorálni. Ezek az állományok a nézetet képviselik. A *Sass* szintén a megjelenítésért és annak módjáért felelős, a *TypeScript* állományok pedig a komponens viselkedését határozzák meg a futtatás alatt fellépő események hatására. A következő bekezdésekben ez a két programozási nyelv kerül röviden bemutatásra.

A *TypeScript* egy olyan JavaScriptre épülő programozási nyelv, amelyet azért terveztek, hogy megkönnyítse a JavaScript kód írását. A *JavaScript*-tel ellentétben a *TypeScript* erősen típusos, biztosít statikus típusellenőrzést, valamint támogatja a különféle dekorátorok használatát. Mivel a *TypeScript* kódot a böngészők nem tudják értelmezni, futtatás előtt ez teljesen le lesz fordítva JavaScriptre [17].

A *Sass* (*Syntactically Awesome Style Sheets*) egy stílusleíró nyelv, a *CSS* kibővítése. Az *Sass*-ben van lehetőség változók deklarálására, már elkészült stíluslapok importálásra, valamint egymásba ágyazott stílusosztályok használatára.

RxJS

Az *RxJS* (*Reactive Extensions for JavaScript*) egy olyan könyvtár, amely segítségével reaktív programozást lehet megvalósítani az *Angular* keretrendszerben [5]. Tehát a könyvtár az adatfolyamokkal kapcsolatos műveletek, valamint aszinkron események kezelését valósítja meg. Fő eleme az *Observable* típus és a hozzá kapcsolódó operátorok, függvények, amelyek megkönnyítik ennek használatát. A könyvtár az *Observer* és *Iterator* tervezési minták kombinálásával lett létrehozva [14].

A medR esetében az *RxJS* a szerver oldallal való kommunikációt, valamint az onnan érkező adatok feldolgozását könnyítette meg, mivel mindez aszinkron módon történik. Ugyancsak hasznosnak bizonyult az alkalmazás használata közben kiváltódó események kezelésében.

Bootstrap

A *Bootstrap* [19] a világ egyik legnépszerűbb CSS keretrendszere, ami a webes felületek megtervezését teszi egyszerűbbé. A medR fejlesztése során ennek az *Angular* keretrendszerrel

kompatibilis változata került használatra, az *ng-bootstrap* [18]. A keretrendszer olyan előre megtervezett és letesztelt komponensek használatát teszi lehetővé, amelyek gyakran előfordulnak a webes felhasználói felületek fejlesztésekor, mint például a gombok, navigációs elemek, táblázatok stb.

3.3. Eszközök

Git

A *Git* [2] az egyik legnépszerűbb nyílt forráskódú osztott verziókövető rendszer. A Git a fejlesztők dolgát hivatott könnyebbé tenni, mivel a távoli tárolókon megtalálható minden kis változtatás ami a projekten történt, és a lokális tárolók időnkénti frissítésével az időközben történt változtatások ezekben is megjelennek. Így minden fejlesztő egy saját biztonsági másolattal rendelkezik a projektről, ezért ha a köponti tároló meg is hibásodik, az adatvesztés kockázata csökken. Egy másik ebből adódó előny, hogy jelentősen egyszerűbbé teszi a változtatások számontartását és az azonos időben történő fejlesztést. A medR esetében három tároló van használva, egy a szerver (medr-backend), egy a kliens oldal (medr-frontend) és egy a *kubernetes* specifikus konfigurációs állományok (medr-k8s) számára.

GitLab

A *GitLab* [20] egy olyan online git tároló, amelynek segítségével nem csak a kódban történt változtatásokat lehet számon tartani, hanem egy DevOps platformot is biztosít. Ez azt jelenti, hogy a fejlesztés minden lépését monitorizálni lehet: a tervezési fázistól és a kód menedzselésétől kezdve egész a kitelepítésig. Könnyebbé teszi a kollaborációt is a különböző szerepkörök által, valamint a felhasználók könnyen át tudják nézni egymás kódját, véleményezni tudják a változtatásokat.

A *GitLab*-os pipeline rendszer segítségével lett megvalósítva a projekt kitelepítése, folyamatos integrációja. Erről bővebben a 4.4 fejezetben lesz szó.

Docker

A *Docker* egy alkalmazások létrehozását, kitelepítését és futtatását konténerek segítségével könnyítő eszköz. A konténerek lehetőséget adnak a fejlesztőknek, hogy alkalmazásokat csomagoljanak össze az összes szükséges függőségekkel együtt, majd ezeket egy virtuális

környezetben futtassák gazda géptől függetlenül. A docker és a konténerek abban különböznek a hagyományos virtuális gépektől, hogy nincs szükségük egy teljes virtuális operációs rendszerre, mivel kihasználják a gazda gép *kernel*-ét, tehát csak azokat a részeket kell becsomagolják, amelyek nincsenek már meg, ezzel tárhely- és teljesítménybeli előnyöket érve el [29].

Kubernetes

A *Kubernetes* egy hordozható, kiterjeszthető és nyílt-forráskódú konténer orkesztrációs platform, amely az alkalmazások kitelepítésére, skálázására és menedzselésére alkalmas [10]. A *Kubernetes* legalapvetőbb építőkockája a *Pod* nevű objektum, ami egy vagy több konténer futtatására alkalmas (több konténer futtatása egy *pod*-on belül akkor indokolt, ha ezek szerves kapcsolatban állnak egymással, pl. egy adatbázis és a hozzá tartozó *sidecar*). Applikációk kitelepítésére alkalmas a *Deployment*, aminek segítségével létrehozható több ugyanolyan *replikának* nevezett *pod*. Az állapottal rendelkező alkalmazások kitelepítésére, mint például az adatbázisok, a *StatefulSet* használható. Egy *statefulset* működése részleteiben különbözik, de nagyvonalakban megegyezik egy *deployment* működésével. Ugyanúgy replikák jönnek létre, azzal a különbséggel, hogy ezek létrehozásának sorrendje kötött a replikák közötti függőségi kapcsolatok miatt. Mindezeknek az elemeknek az egymással vagy a külvilággal való összeköttetését *Service*k, illetve *Ingress*ek révén lehet megvalósítani.

Ezeknek az építőkockáknak a *Yet Another Markup Language* (YAML) típusú leíróállományaiból áll össze a projekt medr-k8s tárolója. Itt megtalálhatóak a backend és a frontend kitelepítését elvégző állományok, a *PostgreSQL* klasztert létrehozó parancs, valamint a kitelepítési folyamat pontos leírása.

A *Kubernetes* nagy előnye, hogy a rendszer menedzseli a létrehozott objektumokat. Ha egy *pod* hibába ütközik és leáll, amint lehetséges automatikusan újra lesz indítva. *Statefulset*-ek és *deployment*-ek esetében a replikák száma és a félretett rendszer-erőforrások mennyisége utólagosan változtatható, ezzel lehetőséget adva az applikáció könnyű, vagy akár automatikus skálázására szükség esetén. A *service*-ek észreveszik, hogy több *pod* létezik és az új *pod*-ok felé is irányítanak kéréseket. Továbbá a replikákban futó konténer könnyen kicserélhető: a régi *pod*-ok fokozatosan lesznek lecserélve az újak által, így a végfelhasználó gyakorlatilag észre sem veszi a változtatás pillanatát.

```
{
    "scripts": {
        "lint-fix": "npx eslint -c .eslintrc.js --ext .ts . --fix"
    }
}
```

4. kódrészlet. ESLint-et futtató npm script

Build és függőségkezelő eszközök

A *build* és függőségkezelés feladata a backenden a *gradle* eszközre hárul. A *gradle* egy nyílt forráskódú automatizálási feladatok elvégzésére szánt eszköz. A *build.gradle* állományban megadható a projektkonfiguráció, amely a projekt metainformációin kívül tartalmazza a szükséges függőségeket, beállításokat arra vonatkozóan, hogy a függőségeket honnan kell letölteni, illetve ún. *task* definíciókat, melyekben a projekt életciklusához tartozó feladatokat lehet megoldani, *Groovy* vagy *Kotlin DSL* kódrészletek révén [8]. Továbbá megadhatóak *pluginok*, melyek segítségével a *gradle* alapfunkcionalitásai kiterjeszthetőek, ezzel sokszor megspórólva a *task*-ok saját kézzel való megírását.

Az *Angular Command Line Interface* (Angular CLI) felelős a frontend projekt *build* folyamatának kezeléséért. Az Angular CLI további funkcionalitásokkal is bír, rajta keresztül a keretrendszer teljes vezérlése megvalósítható: új projekt létrehozása, komponensek vagy *service*-ek gyártása, *build* folyamat kezelése, tesztek futtatása stb. [4]

A kliens függőségeit egy külön eszköz, a *Node Package Manager* (NPM) kezeli. A *Command Line Interface*-en kívül, az npm-hez egy szoftver regiszter is tartozik, ahonnan a projekt függőségei kerülnek letöltésre [11]. A függőségek egy *package.json* nevezetű leíróállományban vannak felsorakoztatva. Az eszköz lehetőséget ad új parancsok létrehozására is, melyek *script*-eket futtatnak, így rövidítéseket lehet generálni olyan folyamatokra, amelyek a projekt kontextusában sokszor végrehajtásra kerülnek (ld. 4. kódrészlet).

Statikus kódellenőrzők

A kód minőségének biztosítása és bizonyos konvenciók betartása érdekében, a kód folyamatos ellenőrzés alatt állt. Ilyen ellenőrzésre alkalmas eszközök a statikus kódellenőrzők, melyek a karbantarthatóságot, olvashatóságot ellenőrzik, vagy akár funkcionalitást érintő hibák észlelésére és gyors javítására is alkalmasak. Az alkalmazás fejlesztése folyamán több ilyen eszköz került használatra.

A backend kód minőségét a *CheckStyle* [32], *SpotBugs* [15] és *PMD* [12] eszközhármas biztosítja, felelve sorban stilisztikai, funkcionalitásokat érintő és karbantarthatóságot érintő szabályok ellenőrzéséért és betartásáért. Mindhárom ellenőrző be van építve a *build* folyamatba, a gradle *check task*-ja által. Konfigurációjukat illetően, XML állományokba írhatóak a használni kívánt szabályok, és, hogy ezeknek a megszegése milyen mértékben jelentsen figyelmeztetést vagy hibát.

A frontend kódminőségének ellenőrzését főként az *ESLint* biztosítja, amely eredetileg *JavaScript* forrásállományok ellenőrzésére volt létrehozva [33], viszont az Angular keretrendszer által használt *TypeScript* állományok ellenőrzésére is alkalmas. Nagy előnye, hogy bizonyos, kisebb szabályszegéseket akár automatikusan is meg tud oldani, ezzel a fejlesztők dolgát könnyítve. Továbbá, egy egységes kódstílus biztosítása érdekében a *Prettier* kódformatáló eszköz van alkalmazva. Az *ESLint*-el ellentétben a *Prettier* kevésbé konfigurálható, ez viszont úgy is tekinthető, hogy egy jó, olvasható kódstílust biztosít nehézkes konfigurációs állományok írása és inkompabilitások megoldásának szükségessége nélkül [25]. Az *ESLint* stílusellenőrző funkcióit kikapcsolva, a két eszköz beállítható úgy, hogy tökéletesen működjenek egymással, a hibák nagy részét automatikusan javítsák, és a maradék hibáról pedig részletes jelentést adjanak.

4. Munkamódszerek

Egy szoftverrendszer minősége, valamint az illető rendszeren dolgozó csapat hatékonysága nagyban függ az alkalmazott módszerektől és folyamatoktól. Létfontosságú volt tehát, hogy ezek egy korszerű és bevállt munkamenetet alkossanak. A következő alfejezetek ennek a munkamenetnek az egyes mozzanatait részletezik.

4.1. Scrum

A munka során a csapat a *Scrum* agilis szoftverfejlesztési stratégiát követte, amely a csapatmunka minél hatékonyabbá tételére és a folyamatos fejlődésre fekteti a hangsúlyt [21]. Mivel a *Scrum* egy iteratív és inkrementális módszer, ezért a fejlesztés két-három hetes iterációkra volt felosztva, amelyek magukba foglaltak egy teljes fejlesztési ciklust. Ezeket az iterációs szakaszokat a *Scrum* módszer esetén *sprint*-nek nevezik. Minden *sprint* egy tervezéssel (*Sprint Planning*) kezdődik: ekkor a *product backlog*-ban levő követelmények közül lesz kiválasztva annyi, amennyit a csapat feltehetőleg az iteráció alatt teljesíteni tud. A kiválasztott követelmények a *sprint backlog*-ba kerülnek.

A Codespringnél végzett nyári gyakorlat folyamán minden *sprint* ideje alatt a csapat napi megbeszéléseket tartott. Ezeket *Daily Scrum Meeting*-nek nevezik és olyan rövid, legfeljebb tizenöt perces találkozók, ahol minden csapattag röviden beszámol arról, hogy az utolsó gyűlés óta mivel foglalkozott, a következő gyűlésig mik a tervei, valamint jelzi ha valamilyen probléma vagy elakadás felmerült. Amikor a projekt fejlesztése tovább folytatódott a csoportos projekt tantárgy keretein belül, ezeket a napi találkozókat leváltották a hetente egyszer megtartott gyűlések.

Minden fejlesztési szakasz egy áttekintéssel ér véget, amikor bemutatásra kerülnek azok a feladatok amelyek sikeresen elkészültek (demo). Egy másik iteráció végi tevékenység a *Sprint Retrospective*, ami alatt a csapattagok kifejthetik a véleményük a *sprint*-ről, valamint továbbfejlesztési javaslatokat tehetnek.

4.2. GitFlow

A medR projekt fejlesztése során a git használatával kapcsolatos választott munkafolyamat a *GitFlow* volt [22]. A módszer alapján egy git tárolónak két központi ága van, a *master* vagy *main*, és a *develop*. Erre a két fő ágra a fejlesztők nem szoktak *push*-olni. A *master*

ágon mindig az utolsó kitelepítésre kész verzió van, míg a *develop* ág kevésbé stabil és ez áll a fejlesztés központjában. Innen fognak leágazódni a *feature* ágak, amelyeken a különböző funkcionalitások fejlesztése vagy akár javítása történik. Amikor a fejlesztő befejezi a munkáját az adott ágon, egy *merge request*-et hoz létre, annak érdekében hogy a változtatások átkerülhessenek a *develop* ágra.

4.3. Kódellenőrzés

A 3.3. fejezet *Statikus kódellenőrzők* című alfejezetében leírt automatikus kódellenőrzés mellett, a minőségi kód biztosítása érdekében, kézi kódellenőrzésre is szükség volt. Az ellenőrzés nagyvonalakban *GitLab* (3.3. fejezet *GitLab* alfejezete) *merge request*-ek révén történt, ahol egy *branch* változtatásaihoz a csapattagok visszajelzéseket írhattak egyszerű, vagy specifikus kódrészlethez fűzött kommentek segítségével. A problémák javítása, vagy a megoldások tisztázása után az ellenőrző csapattagok jóváhagyásukat adhatták a *branch* változtatásainak visszacsatolására a cél *branch*-be. A kommentek továbbá a tervek finomhangolását, újragondolását, illetve a csapattagok szakmai fejlődését is elősegítették.

4.4. Folyamatos integráció és kitelepítés

A *Continuous Integration* (CI) lehetővé teszi az ellenőrző lépések futtatását minden változtatás (push) esetén, ezzel elkerülve a hibákkal rendelkező *branch*-ek visszacsatolását. A projekt folyamán a GitLab-ba beépített *GitLab CI/CD* folyamatos integráció és kitelepítés eszköz került használatra [7]. Ennek segítségével különböző *pipeline*-nak nevezett tevékenységsorozatok határozhatóak meg. Az 5. kódrészlet első fele (test) a *backend* projekt ellenőrző folyamatát írja le: minden *push* eseménykor futtatásra kerül a teljes ellenőrző csomag a *gradle check* parancs segítségével, majd a gradle konténer adatai, amiben az ellenőrzés fut elmentésre kerülnek, a későbbi futtatások felgyorsításának érdekében. Egy *merge request* visszacsatolása csak akkor valósítható meg, ha a *pipeline* minden része sikeresen lefutott.

A *folyamatos kitelepítés* (CD) lépését a szerver esetében az 5. kódrészlet második fele (build-and-push) írja le. Amikor egy *branch* visszacsatolásra kerül a *develop* ágba, egy új *docker image* jön létre a legújabb változtatásokkal. A *docker login* parancs és a GitLab által nyújtott változók segítségével hozzáférést kapunk a projekt privát konténer regiszteréhez, a *GitLab Container Registry*-hez, ahová az új *docker image* feltöltésre kerül. Az új image létrejötte után,

```
#...
test:
  stage: test
  image: gradle:6.8-jdk11
  before_script:
    - export GRADLE_USER_HOME=`pwd`/.gradle
  script:
    - gradle check --no-daemon
  cache:
    key: $CI_COMMIT_REF_SLUG
    paths:
      - .gradle/wrapper
      - .gradle/caches

build-and-push:
  stage: build
  image: docker:18.09
  services:
    - docker:18.09-dind
  variables:
    IMAGE_TAG: $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker build -t $IMAGE_TAG .
    - docker push $IMAGE_TAG
  tags:
    - docker-image
  only:
    - develop
```

5. kódrészlet. Részlet a backend CI konfigurációjából

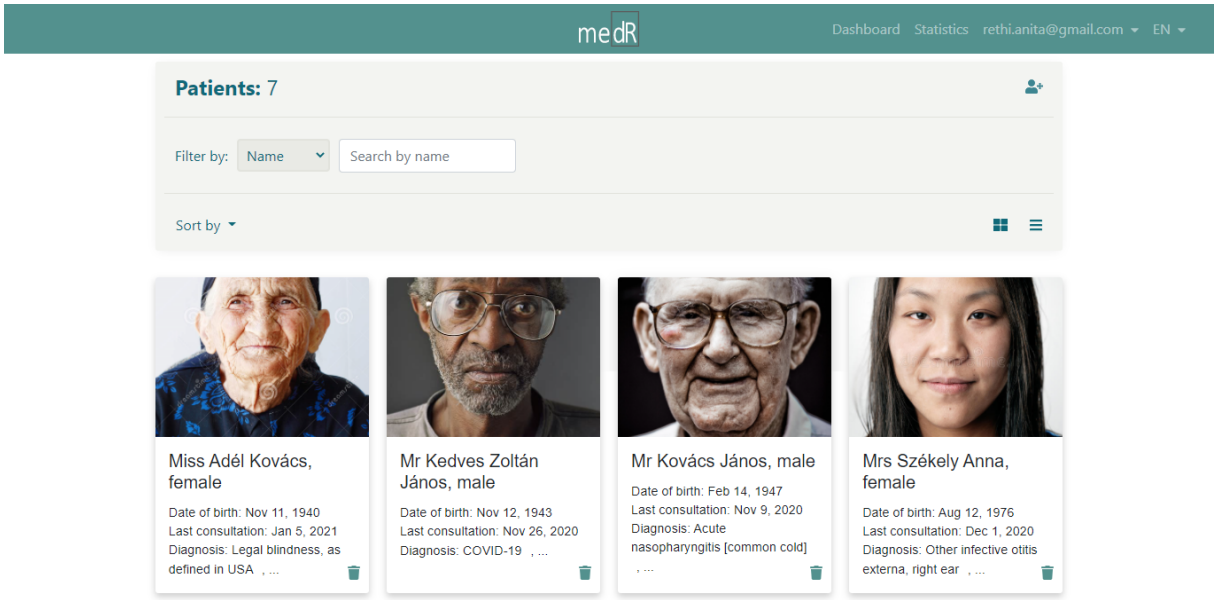
a *kubectl* CLI segítségével automatikusan kitelepíthető az alkalmazás új változata a kubernetes klaszterre.

A 3.3. fejezet *Kubernetes* alfejezetében említett kitelepítési folyamat a következőket hozza létre: egy *statefulset* az adatbázis számára, ami létrehoz legalább egy *master (read-write)* adatbázist és legalább egy *slave (read-only)* adatbázist. Továbbá létrejön egy-egy *deployment* a kliens és szerver számára. Ezek legalább két-két replikát hoznak létre, alapként használva a megfelelő tárolóból (medr-frontend vagy medr-backend) letöltött *docker image*-eket. Végül egy *ingress* jön létre, ami elérhetővé teszi az API-t és a klienst a leíró állományban meghatározott *URL*-ek (Uniform Resource Locator) révén.

5. A medR alkalmazás működése

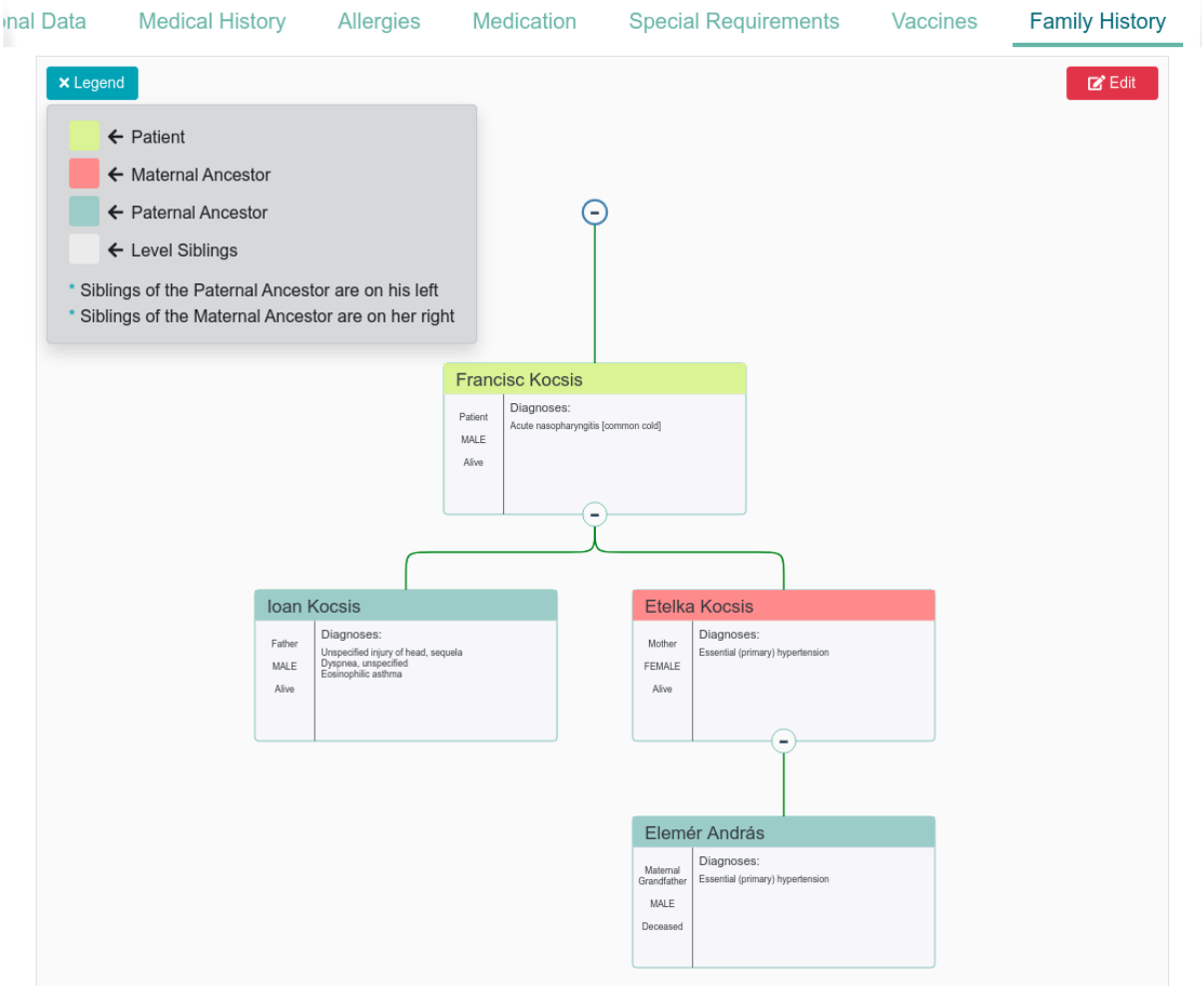
Az alkalmazás felhasználásához szükséges első lépés a regisztráció. A jelenlegi prototípusban ez egy szabadon elérhető folyamat, ahol a regisztrálni kívánó vizsgálónak meg kell adnia a személyes adatait, elérhetőségeit, egészségügyi adatait, mint a titulus, szerepkör, szakág, majd a munkahelyének az adatait, végül pedig egy e-mail címet és egy jelszót. Az elkért adatok mennyisége a vizsgáló személyének igazolásából, valamint az adatok jövőbeli funkcionalitásokban betöltött szerepéből adódik. Amennyiben a regisztráció sikeres volt, a felhasználó visszakerül a főoldalra, ahol bejelentkezhet.

Sikeres bejelentkezés után a *Dashboard* (ld. 6. ábra) jelenik meg, amely a felhasználó eddig bevezetett betegeinek listáját tartalmazza két típusú nézetben (rács és lista). A pácienseket lehet rendezni név és életkor szerinti növekvő és csökkenő sorrendbe, illetve a legutolsó konzultáció dátuma alapján. Annak érdekében, hogy a vizsgáló könnyebben tudjon megtalálni egy beteget, lehetősége van név és diagnózisok alapján keresni. Szintén a *Dashboard*ról elérhetőek a páciens törlése (a páciens kártyák jobb alsó sarkában található ikonra kattintva) és az új páciens létrehozása funkciók.



6. ábra. Dashboard: Az alkalmazás főoldala, ahol a vizsgáló betegei láthatók és menedzselhetők

Új páciens adatlap létrehozása során a vizsgáló hat tab között navigálhat, ezek közül mindegyik más típusú adatok fogadására alkalmas. Alapértelmezetten a *Personal Data* fülben lévő formot kell először kitölteni, és csak azután lehet a többi tabot feltölteni adatokkal, ha minden beírt információ helyesnek bizonyul. Ezen a ponton a páciens elmentődik az

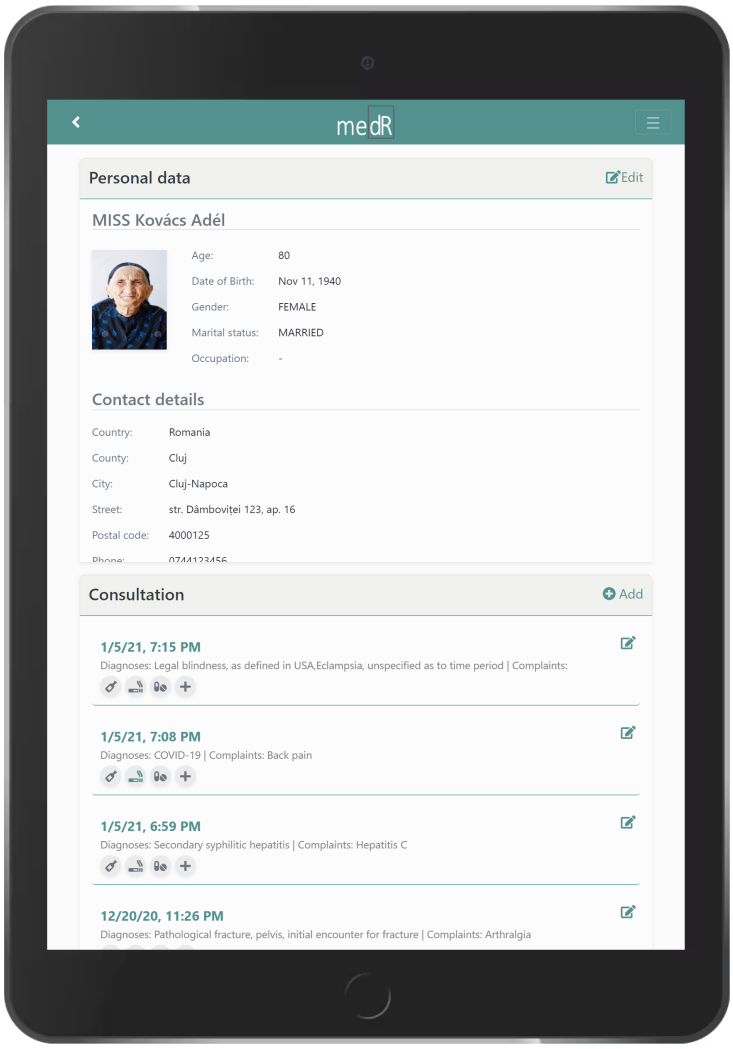


7. ábra. *Family History* fül új páciens létrehozásánál

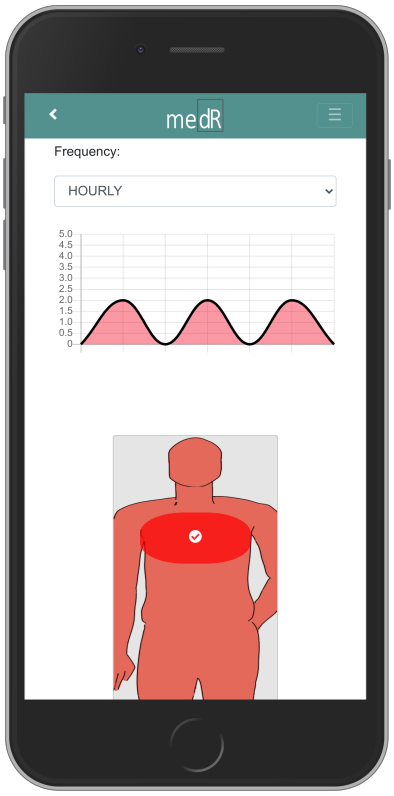
adatbázisba, és a vizsgáló választhat, hogy később tér vissza a többi adat bevezetésére, vagy folytatja. Ez a folyamat sok adatbevitellel jár, mivel a beteg személyes adatain kívül begyűjtésre kerül a személyes és családi kórtörténete, allergiái, gyógyszeres kezelései, eddig kapott oltásai, valamint a különleges szükségletei, ha vannak. A 7. ábrán a tab rendszer van szemléltetve, valamint az utolsó fül, ahova a beteg családi kórtörténetét lehet bevezetni.

A beteglap kitöltése után, visszalépve a dashboardra ki lehet választani egy pácienst és át lehet navigálni a 8. ábrán látható kórlapjára. Ezen az oldalon áttekinthető a páciensh tartozó összes adat, fontossági sorrendbe rendezett kategóriákra osztva. Ugyancsak erről az oldalról el lehet érni egy-egy kategória szerkesztőjét, ahol a páciens létrehozása közben látott eszközökkel lehetőség nyílik új adatok bevitelére, vagy a rendszerbe már bevitt információk szerkesztésére.

A páciens létrehozása során használt kategóriákat leszámítva, az adatlapján megtekinthetőek és szerkeszthetőek az eddig bevezetett vizsgálatai, valamint új vizsgálatot is lehet létrehozni. Amikor a vizsgáló új konzultációt hoz létre, az első lépés az időpont megadása. Ezt követi a



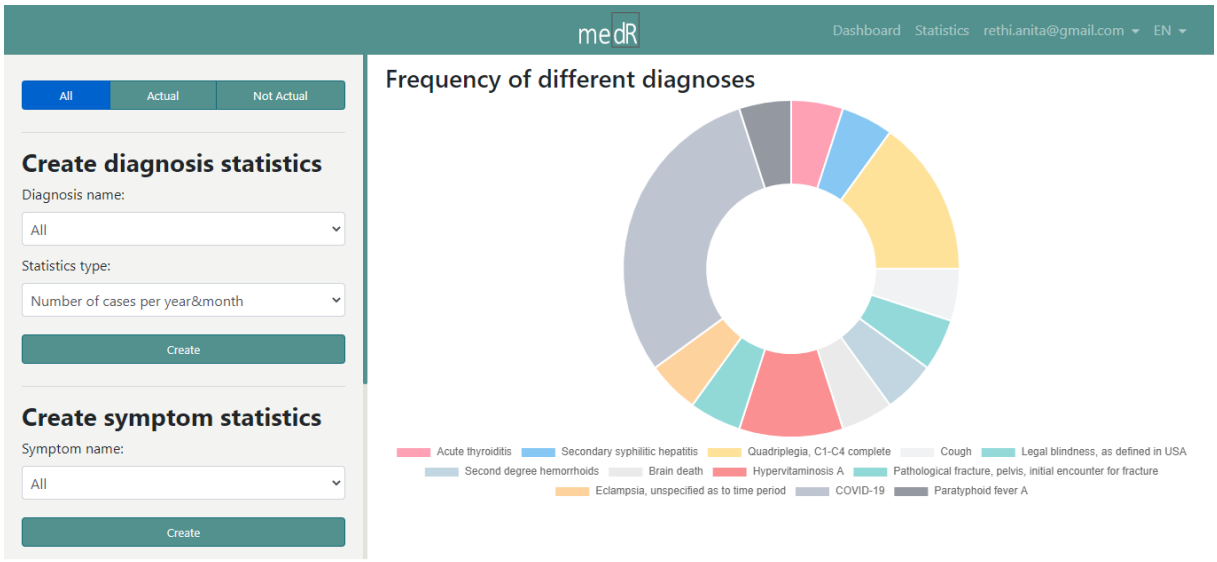
8. ábra. Páciens kórlapja táblagépen, a pácienshez tartozó adatok egy részével



9. ábra. Tünet frekvenciája és lokalizációja mobil eszközön

beteg panaszainak bevezetése a rendszerbe, amely során megadhatja a tünet nevét, súlyosságát, valamint a tünet rövid leírását a beteg elmondása alapján. Szintén be lehet vezetni a tünet első megjelenésének idejét, hogy mennyit tartott, ha pedig visszatérő, akkor a frekvenciáját. Enyhítő és súlyosbító tényezőket is hozzá lehet rendelni. A legutolsó lépés a tünet lokalizációjának meghatározása, az érintett testrészt a bábun lehet kiválasztani (ld. 9. ábra). Ezt követően a *Save* gombot lenyomva elmenthető a bevezetett tünet.

Ehhez hasonlóan rögzítésre kerül az összes lehetséges diagnózis is, valamint a beteg olyan életkörülményei, amelyek valamilyen befolyással lehetnek a végső diagnózis megállapítására. Egy másik fontos funkcionalitás a diagnózisok, tünetek lezárása. Amikor a vizsgáló sikeresen alátámasztott egy diagnózist, vagy nem szeretne többet szerkeszteni egy tünetet, kiválaszthatja a *Save&Lock* funkciót. Diagnózisok lezárása esetén a vizsgálónak meg kell adnia a lezárás pontos okát is, majd ezt követően a diagnózis meg fog jelenni a páciens kórtörténetében.



10. ábra. Statisztika oldal

Egy másik bárhonnan elérhető funkcionalitás a *Statistics* oldal (ld. 10. ábra), amely a bejelentkezett felhasználó betegállománya alapján képes különböző statisztikákat generálni. A vizsgáló elsősorban meghatározhatja, hogy az aktuális, vagyis a kezelés alatt levő páciensek adatai alapján, vagy a már kigyógyult betegek alapján készüljenek el a statisztikák. Ezután három opció közül kell egyet kiválasztania (diagnózisok, tünetek vagy összehasonlító statisztika) és a legördülő listákból a számára megfelelő opciókat kijelölni, majd végül a *Create* gombra kattintva egy könnyen áttekinthető ábra formájában meg fog jelenni a statisztika.

6. Következtetések és továbbfejlesztési lehetőségek

A dolgozatban bemutatott változata a medR projeknek egy olyan felületet nyújt, aminek segítségével az egészségügyi szakemberek saját betegadatbázist hozhatnak létre a dolgozatban említett adatok begyűjtésével, könnyen kezelhetik azt, valamint összesítő vagy összehasonlító statisztikákat generálhatnak a saját betegeikhez tartozó információk alapján. Mindhárom szakaszt nagyban elősegíti az átlátható, logikusan tagolt felhasználói felület, a külső szolgáltatások segítségével létrehozott diagnózis- és tünetkereső funkció, illetve a vizsgálatkezelőhöz hasonló interaktív komponensek.

A projekt tervezése és fejlesztése közben több továbbfejlesztési lehetőség is megfogalmazódott, melyeknek megvalósítása a későbbiekben lesz elvégezve. Az alkalmazás életciklusában egy fontos mérföldkő lenne a kollaborációs funkcionalitások megvalósítása. Ennek érdekében export és import funkciók, valamint oktatást és kutatást elősegítő modulok létrehozása szükséges. Ugyancsak ide tartozik az orvosi jelentések generálása, melynek megvalósítása érdekében szükséges az alkalmazás által begyűjtött adatok bővítése és pontosabb rendszerezése. Annak ellenére, hogy széleskörű adatgyűjtésre kerül sor, ez még mindig csak kis részben fedi le egy valós vizsgálat alatt gyűjtött adatokat. A bővítés által a statisztika generáló funkcionalitás is továbbfejleszthető, akár egy tünetek közötti korrelációk igazolására vagy cáfolására alkalmas modullal.

Tervbe került egy gyógyszer adatokat szolgáltató API bekötése is. Ennek segítségével hasznos kereső, vagy gyógyszerek mellékhatásai és tünetek közötti kapcsolatokat felismerő funkcionalitásokat lehetne biztosítani. A *European Medicines Agency*-nek (EMA) jelenleg nincs egy szabadon elérhető API-ja, tehát egyelőre ez kereskedelmi API-k használatát, vagy egy saját implementáció létrehozását igényelné.

Továbbá egy újabb, pácienseknek szánt felület létrehozása is része kell legyen a fejlesztésnek, ahol a betegek láthatják és kezelhetik a róluk gyűjtött információkat. Az orvosi munkát elősegítő alkalmazásokat szabályozó elrendeléseknek való megfelelés érdekében nem csak elérhetővé kell tenni az adatokat a páciensek számára, de ezeket az elvárásoknak megfelelően enkriptálni kell, ezzel védve minden felhasználó személyes adatait.

Hivatkozások

[1] Mida Soft Business. *Provocările sistemului medical românesc*. 2019. ápr. 12. URL: https://www.midasoft.ro/provocarile-sistemului-medical-romanesc/ (elérés dátuma: 2021. máj. 3.)

[2] Scott Chacon és Ben Straub. *Pro Git*. Apress, 2014.

[3] Filipe Santos Correa. *Progressive Web Apps vs. WebAPKs*. URL: https://www.inovex.de/blog/progressive-web-apps-vs-webapks/ (elérés dátuma: 2021. ápr. 9.)

[4] Angular dokumentáció. *CLI Overview and Command Reference*. URL: https://angular.io/cli (elérés dátuma: 2021. ápr. 12.)

[5] Angular dokumentáció. *The RxJS library*. URL: https://angular.io/guide/rx-library (elérés dátuma: 2021. ápr. 10.)

[6] Angular dokumentáció. *What is Angular?* URL: https://angular.io/guide/what-is-angular (elérés dátuma: 2021. ápr. 9.)

[7] GitLab dokumentáció. *GitLab CI/CD*. URL: https://docs.gitlab.com/ee/ci/README.html (elérés dátuma: 2021. ápr. 14.)

[8] Gradle dokumentáció. *Gradle User Manual*. URL: https://docs.gradle.org/current/userguide/userguide.html (elérés dátuma: 2021. ápr. 12.)

[9] JWT dokumentáció. *Introduction to JSON Web Tokens*. URL: https://jwt.io/introduction (elérés dátuma: 2021. máj. 4.)

[10] Kubernetes dokumentáció. *What is Kubernetes?* URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (elérés dátuma: 2021. máj. 4.)

[11] Npm dokumentáció. *About npm*. URL: https://docs.npmjs.com/about-npm (elérés dátuma: 2021. ápr. 12.)

[12] PMD dokumentáció. *PMD 6.33.0*. URL: https://pmd.github.io/pmd-6.33.0/ (elérés dátuma: 2021. ápr. 11.)

[13] PostgreSQL dokumentáció. *PostgreSQL: About*. URL: https://www.postgresql.org/about/ (elérés dátuma: 2021. ápr. 8.)

[14] RxJS dokumentáció. *Introduction*. URL: https://rxjs.dev/guide/overview (elérés dátuma: 2021. ápr. 13.)

[15] SpotBugs dokumentáció. *SpotBugs manual*. URL: https://spotbugs.readthedocs.io/en/stable/ (elérés dátuma: 2021. ápr. 11.)

[16] Spring dokumentáció. *Spring Framework Overview*. URL: https://docs.spring.io/spring-framework/docs/current/reference/html/overview.html (elérés dátuma: 2021. ápr. 8.)

[17] TypeScript dokumentáció. *TypeScript for the New Programmer*. URL: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html (elérés dátuma: 2021. ápr. 16.)

[18] ng-bootstrap dokumentácó. *Introduction*. URL: https://ng-bootstrap.github.io/#/getting-started (elérés dátuma: 2021. ápr. 12.)

[19] Bootstrap dokumentácó. *Introduction*. URL: https://getbootstrap.com/docs/4.3/getting-started/introduction// (elérés dátuma: 2021. ápr. 12.)

[20] GitLab dokumentácó. *What is GitLab*. URL: https://about.gitlab.com/what-is-gitlab/ (elérés dátuma: 2021. ápr. 14.)

[21] Claire Drumond. *What is Scrum?* URL: https://www.atlassian.com/agile/scrum (elérés dátuma: 2021. ápr. 14.)

[22] Bharat Dwarkani. *Gitflow*. URL: https://blog.axosoft.com/gitflow/ (elérés dátuma: 2021. ápr. 14.)

[23] Roy T Fielding. *Architectural styles and the design of network-based software architectures*. 7. kötet. University of California, Irvine Irvine, 2000.

[24] Sir Robert Hutchison, Donald Hunter és RR Bomford. *Clinical methods*. Cassell, 1963.

[25] James Long. *A Prettier JavaScript Formatter*. 2017. jan. 10. URL: https://archive.jlongster.com/A-Prettier-Formatter (elérés dátuma: 2021. ápr. 11.)

[26] Oana Miron. „O radiografie a sistemului de sănătate din România”. *Revista Polis* 8.1 (2019). URL: https://revistapolis.ro/o-radiografie-a-sistemului-de-sanatate-din-romania/ (elérés dátuma: 2021. máj. 3.)

[27] Peter Mularien. *Spring Security 3*. Packt Publishing Birmingham,, England, 2010.

[28] *National Library of Medicine Clinical Table Search Service weboldala*. URL: https://clinicaltables.nlm.nih.gov/ (elérés dátuma: 2021. ápr. 1.)

[29] opensource.com. *What is Docker?* URL: https://opensource.com/resources/what-docker (elérés dátuma: 2021. ápr. 11.)

[30] *Spring Projects oldal*. URL: https://spring.io/projects/ (elérés dátuma: 2021. ápr. 8.)

[31] Craig Walls. *Spring Boot in action*. Manning Publications, 2016.

[32] CheckStyle weboldal. *Documentation*. URL: https://checkstyle.sourceforge.io/ (elérés dátuma: 2021. ápr. 11.)

[33] ESLint weboldal. *About*. URL: https://eslint.org/docs/about/ (elérés dátuma: 2021. ápr. 11.)