
UNIVERSITATEA „SAPIENTIA” DIN CLUJ-NAPOCA
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE,
TÎRGU-MUREȘ
SPECIALIZAREA CALCULATOARE

Aplicație web pentru alocarea temelor de diplomă la studenți

LUCRARE DE DIPLOMĂ

Coordonator științific:
Dr. Szabó László Zsolt

Absolvent:
Pál Andor

2023

UNIVERSITATEA „SAPIENTIA” din CLUJ-NAPOCA
Facultatea de Științe Tehnice și Umaniste din Târgu Mureș
Specializarea: **Calculatoare**

Viza facultății:



LUCRARE DE DIPLOMĂ

Coordonator științific:
ș.l. dr. ing. Szabó László Zsolt

Candidat: **Pál Andor**
Anul absolvirii: **2023**

a) Tema lucrării de licență:

Aplicație web pentru alocarea temelor de diplomă la studenți

b) Probleme principale tratate:

- Alegerea unui framework backend și frontend adecvat temei
- Realizarea unei aplicații web pentru alocarea temelor de diplomă
- Dezvoltarea aplicației bazat pe teste

c) Desene obligatorii:

- Schema bloc al aplicației
- Diagrame UML privind software-ul realizat.

d) Softuri obligatorii:

-Aplicație web pentru alocarea temelor de diplomă

e) Bibliografia recomandată:

- Y. Fain and A. Moiseev. Angular development with typescript. Manning, 2019
- Lucas da Costa: Testing JavaScript Applications, Manning 2021.
- D. D. B. V. SELJI. Full-stack web development with Spring Boot and angular: A practical guide to building your full-stack web application. PACKT PUBLISHING LIMITED, 2022.

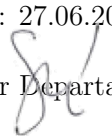
f) Termene obligatorii de consultații: săptămânal

g) Locul și durata practicii: Universitatea „SAPIENTIA” din Cluj-Napoca,
Facultatea de Științe Tehnice și Umaniste din Târgu Mureș

Primit tema la data de: 31.03.2022

Termen de predare: 27.06.2023

Semnătura Director Departament



Semnătura coordonatorului



Semnătura responsabilului
programului de studiu



Semnătura candidatului



Declarație

Subsemnata/ul Pál Andor, absolvent(ă) al/a specializării Calculatoare, promoția 2023 cunoscând prevederile Legii Educației Naționale 1/2011 și a Codului de etică și deontologie profesională a Universității Sapientia cu privire la furt intelectual declar pe propria răspundere că prezenta lucrare de licență/proiect de diplomă/disertație se bazează pe activitatea personală, cercetarea/proiectarea este efectuată de mine, informațiile și datele preluate din literatura de specialitate sunt citate în mod corespunzător.

Localitatea,

Absolvent

Data:

Semnătura

Extras

Scopul tezei este de a proiecta și dezvolta o aplicație web care să digitalizeze alegerea subiectelor pentru diplomă. Era necesară o soluție care să ajute și să faciliteze activitatea comisiei universitare și a studenților în procesul de selecție a temelor pentru diplomă.

Obiectivele noastre au inclus dezvoltarea unei aplicații web care să poată gestiona un număr mare de date și de utilizatori și care să poată fi utilizată cu ușurință de oricine. Aplicația ar trebui să îndeplinească toate cerințele universității și ar putea fi dezvoltată în continuare la cerere. Să fie sigură și să se asigure că aplicația poate fi întreținută.

Obiectivele au fost atinse în toate domeniile. A fost dezvoltată o aplicație web la care secretarele se pot conecta și pot crea perioade de lucru și invita profesori și studenți în sistem. Profesorii își pot încărca subiectele de diplomă, iar studenții pot aplica pentru acestea. Iar invitații pot vizualiza lucrările de diplomă create și făcute publice.

O gamă largă de tehnologii au fost folosite în dezvoltare. Cele mai proeminente sunt cadrele frontend și backend. Folosind Angular și Spring to boot am reușit să dezvolt o aplicație web foarte scalabilă și sigură. Folosind baza de date PostgreSQL pentru a stoca date și cadrele și aplicațiile de testare Cypress și Postman pentru mentenabilitate.

SAPIENTIA ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR
SZÁMÍTÁSTECHNIKA SZAK

Diploma témák választási folyamatát megvalósító web alkalmazás

DIPLOMADOLGOZAT

Témavezető:
Dr. Szabó László Zsolt

Végzős hallgató:
Pál Andor

2023

Kivonat

A dolgozat célja egy web alkalmazás, amely digitalizálja a diploma témák választását. Szükség volt egy megoldásra, ami segítene és megkönnyítene hallgatók munkáját a diploma témával kapcsolatos teendőknel.

Célkitűzések közé tartozott egy olyan web applikáció lefejlesztése, amely képes nagyszámú adatot és felhasználót kezelni és könnyedén használható bárki számára. Az adott applikáció feleljen meg minden egyetemi elvárásnak és kérés esetén tovább fejleszthető legyen. Legyen biztonságos és biztosítva legyen, hogy az applikáció karbantartható.

Sikerült megvalósítani a célkitűzéseket minden terén. Egy olyan web applikáció született, amelyet azoak a személyek használhatnak akik részt vesznek a diploma dolgozatok meghirdetésében és hallgatókhoz való hozzárendelésében. A tanszéki titkárok és munkafolyamatot hozhatnak létre és meghívhatják a tanárokat és a hallgatókat a rendszerbe. A tanárok feltölthetik a diploma témáikat és a hallgatók jelentkezhetnek azokra. Vendégek pedig megtekinthetik az elkészített diploma témákat.

A fejlesztéshez használtam az Angular és Spring Boot keretrendszereket, melyeknek segítségével egy skálázható és biztonságos web applikációt lett fejleszve. Az adatok tárolásához a PostgreSQL adatbázis használva és a karbantarthatóság érdekében tesztelő keretrendszereket és applikációkat, a Cypress-t és Postman-t.

Abstract

The aim of the thesis is to design and develop a web application that digitises the choice of subjects for the diploma. A solution was needed that would help and facilitate the work of the university committee and students in the diploma topic selection process.

Our objectives included the development of a web application that could handle a large number of data and users and could be easily used by anyone. The application should meet all university requirements and could be further developed on request. Be secure and ensure that the application is maintainable.

The objectives have been achieved in all areas. A web application has been developed that secretaries can log into and create periods and invite teachers and students to the system. Teachers can upload their degree topics and students can apply for them. And guests can view the diploma topics they have created.

A wide range of technologies have been used in the development. Most prominent are the frontend and backend frameworks. Using Angular and Spring to boot I was able to develop a highly scalable and secure web application. Using PostgreSQL database to store data and testing frameworks and applications Cypress and Postman for maintainability.

Tartalomjegyzék

Ábrák jegyzéke	3
Táblázatok jegyzéke	4
1. Bevezető	5
2. Elméleti megalapozás és szakirodalmi tanulmány	7
2.1. Szakirodalmi tanulmány	7
2.2. Elméleti alapok	8
2.2.1. Hatékony eszközök a webalkalmazások fejlesztéséhez	8
2.3. Ismert hasonló alkalmazások	8
2.4. Felhasznált technológiák	9
2.4.1. Angular frontend keretrendszer	9
2.4.2. Spring boot backend keretrendszer	10
2.4.3. Szervletek	13
2.4.4. Postgres és annak kezelése	14
2.4.5. Liquibase	15
2.4.6. Tesztelést elősegítő frameworkok	15
2.4.7. Verzió követő és feladat vezérlő alkalmazások	16
3. A rendszer specifikációi és architektúrája	17
3.1. Követelmény specifikációk	17
3.1.1. Funkcionális követelmények	17
3.1.2. Nem funkcionális követelmények	21
3.2. Adatbázis	22
3.3. Alkalmazás architektúra	24
4. Részletes tervezés	30
4.1. Tervezési fázisok	30
4.1.1. Frontend	30
4.1.2. Backend	42
4.2. Tesztelés	48
4.2.1. Frontend tesztelés	48
4.2.2. Backend tesztelés	50
4.2.3. Éles tesztelés	51

5. Üzembe helyezési lépések	52
5.1. Felmerült problémák és megoldásaik	52
5.1.1. Időszakok beintegrálása az applikációba	52
5.1.2. Hallgatók leosztása automatikusan	53
6. Következtetések	54
6.1. Megvalósítások	54
6.2. Továbbfejlesztési lehetőségek	54
Irodalomjegyzék	56
Függelék	57
F.1. Kód részletek	57

Ábrák jegyzéke

2.1. Különbség egy MPA (Multiple Page Application) és egy SPA (Single Page Application) között [8]	9
2.2. Angular applikáció elemi felépítése [1]	10
2.3. Mikroszolgáltatások felépítése [5]	11
2.4. Rétegelt architektúra megvalósítása Spring bootban.	12
2.5. ORM működése[4]	14
3.1. Adatbázis diagram	24
3.2. Alkalmazás architektúra [3]	25
4.1. Frontend oldali typescript osztály	37
4.2. Szerver oldali java osztály	37
4.3. Cypress tesztelés	50
4.4. Postman tesztelés	50
4.5. Diploma téma jelentkezés hallgató szemszögéből	51
4.6. Diploma téma leosztás eredményei	51

Táblázatok jegyzéke

3.1. Bejelentkezési funkció	19
3.2. Hallgatók feltöltése funkció	20
3.3. Diplomamunka létrehozása funkció	20
3.4. Véglegesíteni a leosztott eredményeket a saját szakukról funkció.	20
3.5. Diplomamunkára jelentkezni funkció.	21
3.6. Keresni az elvégzett diplomamunkák között név és kulcsszavak alapján funkció.	21

1. fejezet

Bevezető

A mostani előrehaladott korban nagy mértékben kiegészíti életünket a digitális világ. A digitális világ nélkül már el sem tudnánk képzelni életünket, mivel már annyira szerves részévé vált, hogy szinte nélkülözhetetlen, hiszen nagyon sok olyan problémát segít megoldani, amelyek eddig lehetetlennek tűntek vagy talán túlságosan bonyolultnak bizonyultak. Most már olyan problémákat vagyunk képesek megoldani a digitális technológia segítségével, amik eddig hosszadalmasak, időigényesek vagy nehezen karbantarthatóak voltak. Úgy érzem, hogy minél nagyobb teret engedünk a digitális világnak az életünkben, annál inkább képesek vagyunk más fontosabb dolgokra koncentrálni és ezáltal jobban fejlődni és gyarapodni. Ez annál igazabb, hogy ha az egyetemi rendszerbe próbálunk a digitalizálással kérdéseket megválaszolni és megoldani.

A mi példánkat felvetve, mint végzős egyetemi hallgatók, egy nagy problémával szembesültünk az utolsó évünkben, és pedig a diplomamunka kiválasztása, megtervezése és kivitelezése. Ezen a téren nagy könnyebbiséget jelentett volna számunkra, ha képesek lettünk volna a digitális világban a diplomamunka elvégzéséhez szükséges teendőket elvégezni. Véleményem szerint sokkal gördülékenyebb, egyszerűbb és sok embernek a munkáját könnyítené meg, ha a diplomamunkánkat a digitális világban tudnánk kiválasztani, ott jelentkezni és ott értesülni egy platformon belül. Ezért a projektem célja egy olyan megoldás keresése és létrehozása, ami a diplomamunkával járó szükségtelen nehézségeket átemelné egy olyan platformra, ahol sokkal egyszerűbb módon tudnánk kivitelezni az ezzel járó teendőket: mint például böngészni a meghirdetett diplomamunkákra javasolt témák között, könnyebben megértenénk, hogy egy meghirdetett diplomamunka milyen céllal is jött létre, milyen elvárásokkal van ellátva és hogy mit szeretne az adott diplomamunka megvalósítani és a legfontosabb, hogy milyen ismeretek szükségesek ennek a kivitelezéséhez.

Az projektem az előbb megemlített indokok alapján jött létre, azzal a céllal, hogy gördülékenyebbé tegye a diplomamunkával kapcsolatos teendőket. A projekt célja egy olyan webapplikáció lefejlesztése, ahol **atitkárok** képesek megadni minden fontos eseménynek a határidejét, ami a diplomamunka köré csoportosul, meghatározott dátumokkal létrehozni egy időszakot minden egyes szaknak, amik később behatárolják az applikáció működését és a szükséges iratokat feltölteni. Képesek kezelni a tanulók listáját, egyetlen kattintással képesek egy teljes szak hallgatóságát feltölteni, akik ezek után e-mailben értesülve egy kapott linken keresztül képesek lesznek belépni a platformra. Ugyanakkor lehetővé válik feltölteni a tanárokat és tanszékvezetőket a megadott szakokra.

A **tanárok** pedig könnyedén feltölthetik a diplomamunka ötleteiket, egy vagy több szaknak, kulcsszavakkal ellátva, látni azoknak a listáját, akik jelentkeztek az adott diplomamunkára. Viszont ha már egy hallgató felkereste az adott tanárt egy diplomamunka ötlettel, akkor a tanár képes a hallgatónak névszerint kiírni a megbeszélte diplomamunkát. A diplomamunka kivitelezése közben pedig beszélgethetnek a kivitelező hallgatóval a felületen, illetve letöltheti a hallgató által feltöltött kész diplomamunkát és a végén értékelheti azt.

A **hallgatók** pedig könnyű módszerrel tudnak böngészni az összes meghirdetett diploma között, kulcsszavak alapján keresni és pontos informáló leírásokat, követelményeket, bibliográfiákat és sok minden mást találni, amik segítségével könnyebben és céltudatosabban tudnak dönteni, hogy az adott diplomamunka számukra megfelelő-e. Ha egy hallgató sikeresen ki lett választva egy adott diplomamunkára, akkor lehetősége válik a platform segítségével írni a vezetőtanárnak és a platformra feltölteni a végleges diplomamunka fájljait.

A platform másik célja, hogy lehetőséget biztosítson a véglegesített és elvégzett diplomamunkák böngészésére, megtekintésére és letöltésére, ezáltal a jövő diákjainak adva egyfajta útmutatót, hogy fel tudjanak készülni ők is a saját diplomamunkájuk megírására.

2. fejezet

Elméleti megalapozás és szakirodalmi tanulmány

2.1. Szakirodalmi tanulmány

A következő könyvekben informálódhatunk részletesen azokról a technológiákról, elvekről, amelyek szükségesek az adott webapplikáció fejlesztéséhez.

[12] Minden egyes webapplikáció egyik lételeme a Hypertext Transfer Protocol (HTTP), ami konkrétan reprezentálja azt a nyelvet, más néven protokollt, aminek a segítségével az applikációk kommunikálnak. A HTTP protokoll megértése és használata gyakorlatilag minden webalapú tervezéshez nélkülözhetetlen. Az adott könyv számtalan logikusan szervezett fejezetben elmagyarázza a HTTP-t és a hasonló technológiákat, több száz példával és érdekes illusztrációval.

Az alábbi 2 könyv [23] , [11] bevezet az Angular rejtelseibe és megtanít mindenre, ami szükséges ahhoz, hogy egy Angular applikációt hozzunk létre. Mind a két könyv célratorően kódolással kezd és az alapoktól a fejlett technikák felé halad. Magukba foglalják a legjobb megoldásokat, tesztelések kezelését és megírását, az Angularban használt konstruktoros függőség beinjektálását, a legjobb teljesítményre való törekvést, 'pipe'-okat és sok más.

[22] Az adott szakirodalom segít abban, hogy elsajátítsuk és megértsük az egyoldalas web-alkalmazások értelmét és szükségességét. Egyoldalas webalkalmazás alatt leginkább azt értjük, hogy az applikációnk igazából egyetlen HyperText Markup Language (HTML) fájlból épül fel és weboldal váltások közben csak az adott HTML fájlban található komponensek cseréje történik meg. Ezáltal gyorsabb lesz az alkalmazásunk.

[18] Ez a könyv pedig nagyon hasznos volt, hogy tisztázza és letisztítsa a fogalmakat a tesztelés szempontjából. Egy általános ismertetéssel kezdődően mutatta be, hogy mi is a tesztelés és mit értünk egy applikáció tesztelése során. Mit érdemes tesztelni és mikor van értelme és mikor kifizetődő, hogy ha egy applikációhoz tesztek vannak írva. Külön kitért részletesen a három tesztelési típusra. A 'unit testing', amikor egy alkalmazásnak a legkisebb elemét teszteljük, a függvényeket. Majd a 'component testing', ez esetben több függvény vagy egy teljes komponenst vagy egy folyamatot tesztelünk az applikációban. Majd később a legvégén az End to End (E2E) testingről kapunk információt, amikor a teljes működő applikációt teszteljük a kliens szemszögéből.

2.2. Elméleti alapok

A jelenlegi korban fénykorukat élik a webalkalmazások, mivel egyre nagyobb jelentőséggel bírnak. Ezek az alkalmazások az interneten keresztül érhetőek el és megkönnyítik a felhasználók dolgát azzal, hogy akár otthonról, nagy eséllyel akármilyen problémát megoldjanak az online környezetben.

A webalkalmazások olyan szoftveralkalmazások, amelyek futtatásához csupán egy böngészőre van szükségünk, nincs szükség arra, hogy mi a saját gépünkön különböző egyéb szoftvereket telepítsünk fel. Egy jól megtervezet alkalmazásnál, amelynek tervezése során figyelem volt fordítva a responzivitásra és a hatékonyság optimalizálására, nem okoz gondot futtatni akár egy tableten vagy akár egy okostelefonon is. A webalkalmazások fejlesztése során szimultán sok dologra kell figyelmet fektetni. Az alkalmazásnak a kezelőfelülete impozáns kell legyen, meg kell ragadja a felhasználó tetszését, viszont letisztult és útbaigazító kell legyen. Ügyelni kell arra, hogy minden kijelzőméreten is ugyan azt a hatást érje el. Ezek mellett gyorsnak kell lennie, mivel átlagosan egy felhasználó 7 másodpercnél tovább nem hajlandó várni egy-egy oldal betöltésére. Ezt a gyorsaságot a szerveroldali optimalizálásával érhetjük el, hogy a kérések minél hamarabb választ kapjanak és ezek mellett minden szükséges számítást el tudjanak végezni és a szükséges adatot visszaszolgáltatni a felhasználónak.[20]

2.2.1. Hatékony eszközök a webalkalmazások fejlesztéséhez

Véleményem szerint ahhoz, hogy egy hatékony és jól kinéző webalkalmazást fejlesszünk, elkerülhetetlen a célhoz legjobban találó keretrendszerek kiválasztása. Az én választásom a diplomamunkámhoz az Angular és Spring boot volt. Az Angular és a Spring boot egy nagyon erőteljes kombinációt biztosít, amit az utóbbi időben nagyon gyakran használnak webalkalmazások fejlesztésére. Ez a kombináció biztosítja azt, hogy képesek legyünk modern, megbízható és hosszútávon karbantartható szoftvereket létrehozni. Íme egy pár jelentős indok amit fel lehet hozni az Angular és Spring boot alkalmazása mellett:

1. Kifinomult felhasználói felület: Az Angular egy erőteljes frontend keretrendszer, hiszen a fejlesztők számára hatalmas nagy segítséget nyújt abban, hogy dinamikus és interaktív felhasználói felületeket hozhassanak létre. Az Angularben a komponensalapú megközelítés segít abban, hogy modularizáljuk az applikációnkat, ezáltal serkenti az újrahasználatosságot.
2. Kiterjeszthető backend: A Spring abban remekel a legjobban, hogy egy nagyon megbízható kódbázis biztosító keretrendszeré vált az utóbbi évek alatt. A fejlesztők a Spring boot használatával könnyedséggel képesek RESTful API-kat írni. [21]

2.3. Ismert hasonló alkalmazások

Ilyen jellegű applikáció fejlesztésénél előnyös lehet eddig már más egyetemeken által lefejlesztett hasonló diplomamunka böngésző applikációk megfigyelése.

Két, a projektemhez hasonló webalkalmazást találtam.

1. [2] Az alábbi weboldal a Budapesti Műszaki és Gazdaságtudományi Egyetem (BME) Diplomaterv Adatbázisához tartozik. Az oldal a BME hallgatóinak lehetőséget nyújt abban, hogy megosszák, bemutassák vagy böngésszenek más elkészített diplomák között. Az oldal egy kicsit régimódi látszatot nyújt és nehezen átlátható. Az oldalon nagyon jó elvek és gondolatok vannak megvalósítva. Ezt böngészve fogalmazódott meg bennem az a cél, hogy olyan platformot hozzak létre, amely jobb, megkapóbb és könnyebben kezelhető, áttekinthető módon lesz a jövőbeni hallgatók segítségére.
2. A második hasonló oldal [7] a Szegedi Tudományegyetem hasonló elképzelésű weboldala. Ez már jobb kezelőfelületet biztosít a látogatóknak és sokkal letisztultabb.

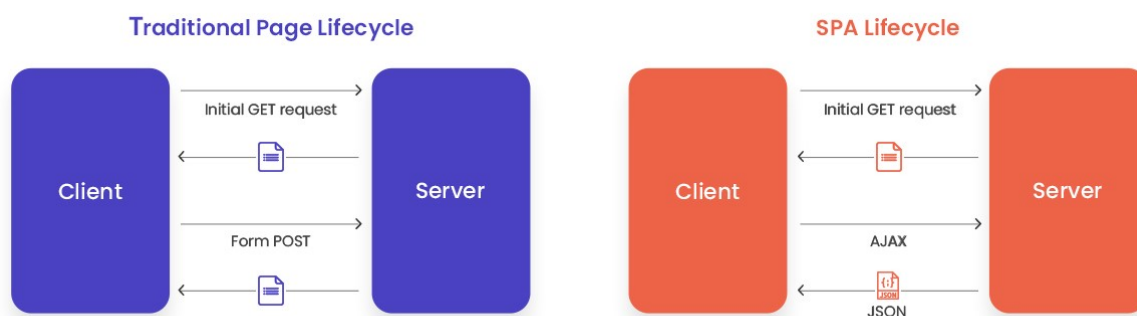
Mindkét megemlített weboldal esetében azoknak a hallgatóknak, akiknek szolgáltatva van belépési engedély az oldalra, azok képesek saját diploma ötleteket létrehozni és böngészni a meghirdetett diplomák között.

2.4. Felhasznált technológiák

2.4.1. Angular frontend keretrendszer

A már fentebb említett használt keretrendszer az Angular. Az Angular az egy modern komponens alapú keretrendszer. Fejlesztését támogatja a Google és az a céljuk hogy hatékony és dinamikus élményt biztosítson mindenféle eszközön.[10]

Az Angular felhasználja a TypeScript nyelvet. A TypeScript az egy nyílt forráskódú nyelv, ami a JavaScriptre épül. Kiterjeszti teljes egészében a JavaScriptet, plusz mellette ellátja típus ellenőrzéssel és típusdeklarációkkal. Azt jelenti, hogy fejlesztés közben meghatározhatjuk a változóink típusát és struktúráját, ez segít abban, hogy sokkal letisztultabb kódot tudjunk írni, sokkal kevesebb hibát ejtsünk és ne töltsünk túl sok időt a hibák okának a felfedésével. Az eddigi meglévő kódjainkat is nyugodtan be tudjuk építeni a TypeScript fájllokba, mivel később A TypeScript fordító a TypeScript kódot JavaScriptre fordítja le. [11].



2.1. ábra. Különbség egy MPA (Multiple Page Application) és egy SPA (Single Page Application) között [8]

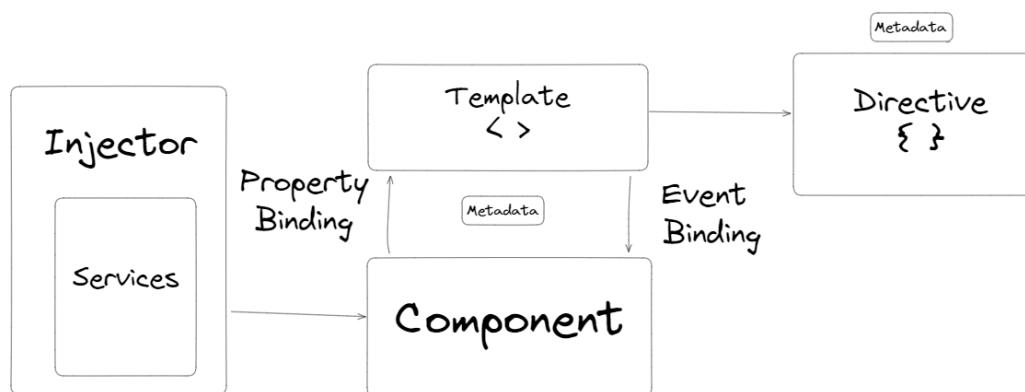
Az Angular ereje igazából az újrahasználatosságban nyilvánul meg. Mivel az Angular egy komponens alapú keretrendszer, komponenseket vagyunk képesek létrehozni és azokat

annyiszor tudjuk újrahasználni, ahányszor csak szeretnénk. Ezeknek a komponenseknek a cserélgetésével pedig képesek vagyunk könnyedén egyoldalas webalkalmazás létrehozásában.

Az egyoldalas (SPA) és többoldalas (MPA) alkalmazások közötti különbség az alkalmazás felépítésében és működésében rejlik.

Egy többoldalas alkalmazás esetében, minden egyes alkalommal, amikor a felhasználó navigál az oldalon, akkor a szerver újratölti az egész oldalt. Tehát a szerver elküldi az összes szükséges fájlt és adatot ahhoz, hogy betöltse az oldalt, legyen az JavaScript, HTML vagy CSS. Ez azt eredményezi, hogy sokkal időigényesebb az applikáció különösen akkor, ha egy bonyolult applikációról beszélünk.

Ezzel szemben egy egyoldalas applikáció pont fordítva működik. Az első betöltésnél a szerver elküldi az összes szükséges fájlt és adatot, ilyenkor egy hatalmas mennyiségű adat érkezik és ez kiépül a felhasználó böngészőjében. Ezek után minden egyes navigálás során csak a szükséges új adatokat kéri le a böngésző a szervertől, ezáltal sokkal gyorsabb és rugalmasabb lesz az applikáció. Az adatok dinamikusan frissülnek ahelyett hogy az egész oldal újratöltődne.[8]



2.2. ábra. Angular applikáció elemi felépítése [1]

Egy alap Angular applikáció felépítése a fenti 2.2 ábrán látható. Konstruktoros injektálással beinjektálódnak az adott komponensbe a ‘Service’-ek. A ‘Service’-ek felelősek ahhoz, hogy a szerveroldalról kinyerjék az információkat HTTP kéréseken keresztül. A beérkező adatokat beillesztve a komponens HTML fájljába megjelenítjük a weboldalon. Minden egyes adatváltozás során újraépül a HTML fájl és a HTML fájlban lévő eventeken keresztül vagyunk képesek manipulálni azokat. [1]

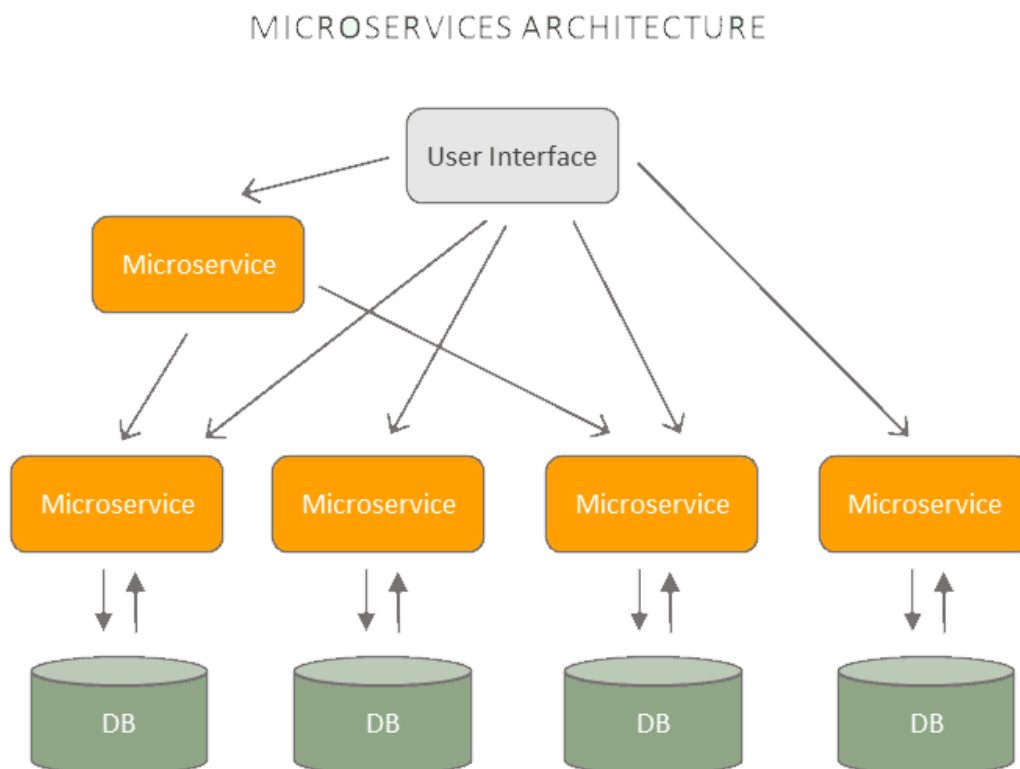
2.4.2. Spring boot backend keretrendszer

Java egy erősen objektumorientált programozási nyelv, ami azt jelenti, hogy nem vagyunk képesek kódot írni osztályokon kívül és csakis osztályokkal tudunk dolgozni. A nyelvet James Gosling fejlesztette ki az 1990-es években, ami a mai napig egy nagyon releváns nyelvnek bizonyult és a hosszú múltja miatt a legmegbízhatóbbnak minősült. [13].

Egy másik nagy előnye a nyelvnek a platformfüggetlenség. Ez a Java koncepciója, hogy egyszer kell megírni és utána bárhol lehet futtatni. Ez azt jelenti, hogy a Java alkalmazások bármilyen operációs rendszeren tudnak futni, mivel egy virtuális téren Java Virtual Machine-en (JVM) futnak. A Java futtatókörnyezet a Java Runtime Environment (JRE) vagy a Java Fejlesztői Készlet a JDK (Java Development Kit) olyan szoftvercsomagok, amelyek lehetővé teszik a Java alkalmazások futtatását az adott rendszeren. Ezek a csomagok tartalmazzák az előbb említett JVM-t és ez a JVM az adott operációs rendszerre optimalizált bináris kódra fordítja a bytecodeban tárolt Java kódot.[13].

A Spring boot az egy nyílt forráskódú Javára épített keretrendszer, amely a Springből származik. A Spring boot leegyszerűsíti a konfigurációk beállítását, automatikusan beállítja a meghatározott függőségeket, csupán el kell lássuk a main függvényünket a @SpringBootApplication annotációval és ez által sokkal gyorsabban és hamarabb el tudjuk érni az applikáció futtatását és nekiláthatunk az üzleti logika felépítésének.

A Spring bootban úgynevezett mikroszolgáltatásokat tudjunk létrehozni. A mikroszolgáltatások alatt azt értjük, hogy minden egyes üzleti funkció el van izolálva a többitől. Ezáltal a Java Spring moduláris és laza csatolású lesz, mivel minden mikroszolgáltatás rendelkezik saját API-val, üzleti logikával és adatbázis eléréssel. Ezt szemléltetem a következő ábrán 2.3.



2.3. ábra. Mikroszolgáltatások felépítése [5]

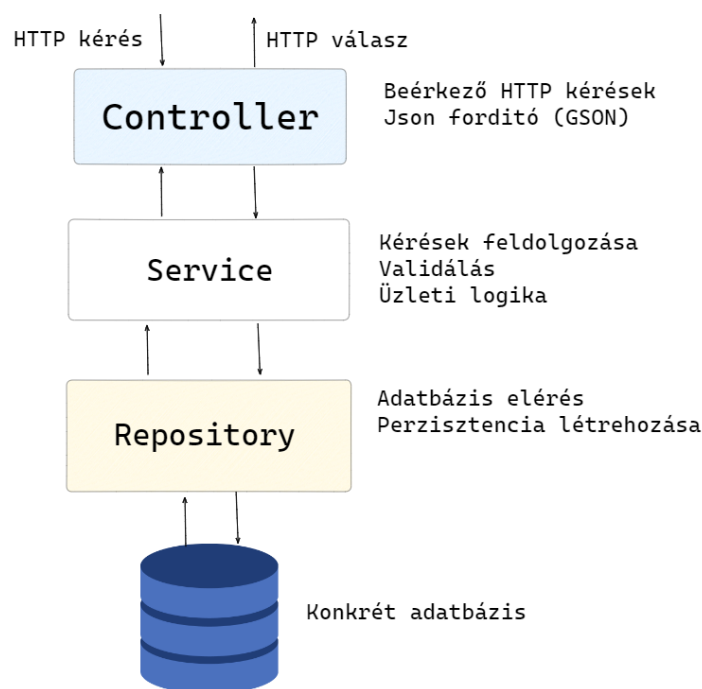
Mint ahogyan az ábrán 2.3 látszik a User Interface reprezentálja a kliensoldalt. A felhasználó által kért adatok, mind a saját külön útjukat járják be. Mindegyik mikroszolgáltatásnak külön adatbázis elérése van és ezek nem keveredhetnek, viszont a mikroszolgáltatások kommunikálhatnak egymással.

Spring boot architektúrája

Minden egyes jól meghatározott szoftver kell kövessen egy tervezési architektúrát. A Java Springben szinte gyönyörű módon lehet alkalmazni a rétegelt architektúrát.

A rétegelt architektúra egy olyan tervezési minta, amely a szoftverünket több komponensre osztja fel. Minden egyes rétegnek meghatározott szerepe van és semmi több. A rétegek elválasztása több előnnyel is jár:

1. Letisztult fájlrendszer: Ha a rétegelt architektúrát használjuk, akkor sokkal átláthatóbb lesz az alkalmazásunk, emiatt könnyebb lesz annak fejlesztése hosszú távon, akár hatalmas kódbázisok esetében is.
2. Modularitás és gyengébb függőség: A rétegzett architektúrának a másik hatalmas előnye, hogy a rétegek nem függenek szorosan egymástól, emiatt fellép egy moduláris összkép. Mivel nem olyan szoros a kötődés az elemek között, bármikor megtehetjük, hogy az adott réteget kicseréljük egy teljesen más implementációra vagy esetleg lecseréljük a teljes adatbázisunkat. Ezek a műveletek nem fognak akkora refaktorálást igényelni, mivel csak konkrétan kiemeljük a réteget és helyettesítjük egy másikkal. [15]



2.4. ábra. Rétegelt architektúra megvalósítása Spring bootban.

Az ábrán 2.4 látható a rétegelt architektúra megvalósítása Spring bootban. A rétegek a következő szerepeket töltik be:

1. Controller réteg: A Controller rétegnek a szerepe fogadni a kéréseket a külvilágtól, vagyis a frontendtől. Ezen a részen vannak definiálva az applikációnknak az endpointjai.

Mint a fentebbi kódrészletben is látszik, meghatározzuk, hogy a "/diploma" kulcsszóval rendelkező HTTP kérést ez a kontroller osztály fogja kezelni. Azon belül pedig "/create" kulcsszóval végződő POST HTTP kérést az alábbi metódus fogja kezelni.

Azelőtt, hogy a metódus meghívódna a kérés még átmegy egy szűrőn, ami ellenőrzi, hogy a kérést végző félnek van-e jogosultsága a kéréshez. Majd, ha GSON, amely segít abban, hogy a HTTP kérés testében tárolt JSON objektumot átalakítsa javában értelmezhető objektumokra, akkor lefut a kontroller metódus. A Controller szerepköre ennyiben le is zárul, innen átadja a munkát a 'Service' rétegnek és várja tőle a választ. [19]

2. Service réteg: A Service rétegben zajlik le az összes logikai művelet. Elsősorban validálás történik arról, hogy a kérés végrehajtásához minden adat és információ helyesen meg lett-e fogalmazva. Majd a szükséges adatokhoz kéréseket hoz létre az adatelérési réteghez. [19]
3. Adatelérési réteg: Az adatelérési réteg az a legalsóbb szint. Ez a réteg kezeli az adatoknak való lekérését, mentését és manipulációját az adatbázisban. Ez a réteg direkt eléréssel rendelkezik az adatbázisunkhoz és Structured Query Language (SQL) segítségével adatokat nyer ki belőle. Ezen a szinten történik meg a Persistencia létrehozása, ami annyit rejt, hogy az adatbázis sorok átalakíthatóak java objektumokká a könnyű használhatóságért. Ezt egy másik Framework által valósítjuk meg, a Java Persistence API-val (JPA) [19]

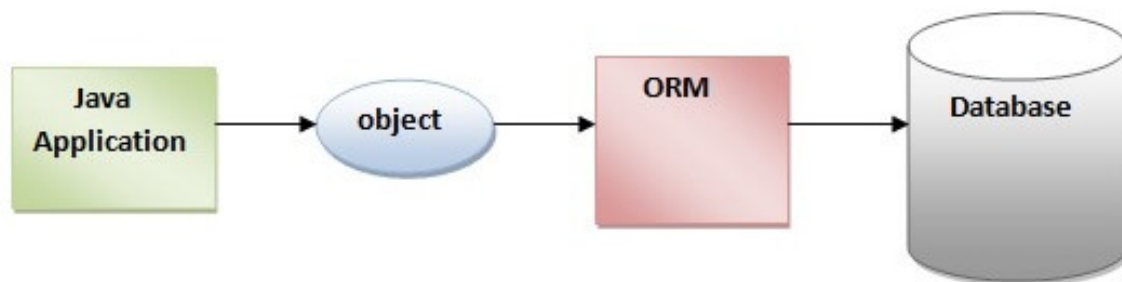
2.4.3. Szervletek

Az alkalmazásunk más alkalmazásokkal való kommunikációjához, esetünkben az Angular frontenddel szükséges, hogy meghatározzunk egy protokollt, amely megfelel az alkalmazásunk elvárásainak. A mi alkalmazás típusunkhoz a HTTP protokoll talál a legjobban. A Spring MVCben konfigurálva vannak a HTTP Szervletek és ezeket felhasználja a Spring boot. Ezáltal elkerülhetjük, hogy saját magunk írjuk meg ezeket a szervlet konfigurációkat és hosszadalmas konfigurációs feladatokkal kelljen foglalkoznunk. [19]

A HTTP protokollnak a legnagyobb hatékonysága abban rejlik, hogy állapotmentes, vagyis nem tárolódnak el semmilyen adatok a kérdőről, vagy a válaszról. Mivel protokoll alatt egy nyelvet értünk, amit, mint a két fél, vagyis mint a két applikáció beszél, pontosan meg vannak határozva a paraméterei annak, hogy a kérés hogyan kell összetevődjön és egy vessző különbség sem nem lehet bennük. [14]

A HTTP kérés három részből épül fel:

1. Kérés sora, itt meghatározzuk a HTTP kérés típusát, az erőforrást és a HTTP verziót



2.5. ábra. ORM működése[4]

2. kérés fejléce: itt meghatározunk bizonyos konfigurációkat a kérésben, mint például milyen típusú fájlokat küldjön, információk rólunk a feladóról.
3. kérés teste: Itt helyezkednek el a fájlok vagy információk, amiket szeretnénk küldeni a kéréssel együtt.

Több fajta HTTP kérés létezik és mindegyiknek meg van határozva, hogy milyen fajta kérés esetében kell használni, a leggyakrabban használtak a következők:

1. GET: A metódust akkor használjuk, amikor információt szeretnénk kérni a szervertől.
2. POST: Amikor információt szeretnénk küldeni a szerverre
3. DELETE: Amikor törölni szeretnénk a szerverről információt.
4. PUT: Amikor meglévő információt szeretnénk módosítani a szerveren.

[14]

2.4.4. Postgres és annak kezelése

Az adatbázis választásánál az volt a mérvadó, hogy minél gyorsabb legyen a mellett, hogy viszonylag nagy adathalmazokkal és sok mapping táblával kell dolgozni. Ezek mellett jól integrálódik a Spring Boot keretrendszerrel, mivel a Spring Boot kész integrációt képez a Spring Data JPA (Java Persistence API) segítségével. Jelentősen skálázható és sokkal több fajta adattípust támogat, mint más híres relációs adatbázisok. [17]

Az adatbázisban az adatok kezelését a Hibernate keretrendszer oldja meg, amely implementálja a Java Persistence API specifikációit. A Hibernate keretrendszer az egy ORM (Object-Relational Mapping), amely direkt a Java nyelvhez készült. A Hibernate a legelterjedtebb ORM amelyet elég gyakran használnak a Java Spring applikációkban. [4]

A Hibernate leegyszerűsíti az adatok létrehozását az adatbázisban, úgy hogy Java objektumokat átalakítja adatbázis sorokká és ugyan ezzel az eljárással képesek vagyunk az adatok kinyerésére is, az alábbi ábrán 2.5 illusztrálva. A másik hasznos dolog amit nyújt számunkra hogy nem kell manuálisan SQL lekérdezéseket írjunk hanem elég jól elnevezett függvény neveket megadnunk és ezeket átfordítja SQL lekérdezésekre.

Ezekre az függvény elnevezésekre létezik egy általános konvenció amit ha betartunk akkor mindig ugyanazt az eredményt fogjuk elérni mintha saját SQL lekérdezéseket írnánk. Segítségünkre lehet az úgy nevezett Jpa Designer plugin, amelyet feltelepíthetünk az IntelliJ IDEA fejlesztői környezetbe, amely egy felületen keresztül segít nekünk paraméterek alapján kigenerálni a függvényeket. Viszont, ha bonyolultabb lekérdezést szeretnénk használni akkor képesek vagyunk a @Query annotációval saját magunk megírni az adott SQL lekérdezést. [4]

2.4.5. Liquibase

Egy nyílt forráskódú eszköz, amely az adatbázis tábláink létrehozásában és verziókövetésében nyújt segítséget. A liquibase segítségével képesek vagyunk létrehozni egy konfigurációs Extensible Markup Language (XML) fájlt, amely tartalmazza az adatbázis tábla felépítését és a későbbiekben a rajta végzett változtatásokat. Ezzel egy adatbázis verzió követő rendszert kapunk, amely ellenőrzi minden applikáció indításakor,

hogy lettek-e változtatások definiálva az adatbázisban és azokat automatikusan beintegrálja.

Egy másik nagy előnye, hogy ezeket a konfigurációs fájlokat hozzá tudjuk csatolni az applikációhoz és így bármilyen rendszeren futtatva, a legfrissebb adatbázis modell épül fel. Plusz a konfigurációk mellé csatolhatunk adatbázis sorokat CSV fájl formátumban és ezeket minden indításnál automatikusan betölti az adatbázisba.

Minden egyes változtatást egy úgy nevezett „change set”-be írjuk, amelyeknek külön azonosítójuk lesz és leírásuk.

2.4.6. Tesztelést elősegítő frameworkok

Tesztelése akár minimális szinten is nélkülözhetetlen egy applikáció fejlesztése közben. Segít abban, hogy tudjuk, hogy az applikáció úgy működik-e ahogy az elvárt és segít a hibák elkerülésében.

Cypress

A Cypress az egy End-to-End (E2E) tesztelési keretrendszer, amelyet webapplikációk tesztelésére fejlesztettek ki. A Cypress rendkívül népszerű, mivel egy felhasználó barát platformot nyújt. A Cypress utasítások segítségével könnyedén tudunk formákat kitölteni, navigálni az oldalon és függvényeket írhatunk az ismétlődő utasításokra. Lépésről lépésre látjuk ahogyan lefutnak az utasítások egy adott tesztben. Minden tesztet külön elindít egy elizolált böngészőben, így valóságos teszt eredményeket kapunk.

End-to-End teszteken kívül komponens teszteket is írhatunk benne, így a Cypressen belül megtudjuk oldani az összes frontenddel kapcsolatos teszteket. [16]

Postman

A Postman egy szoftverfejlesztési eszköz, aminek segítségével a Application Programming Interface (API) tesztelését tudjuk lefedni. A Postman is mint a Cypress egy felhasználó barát platformmal rendelkezik, amin keresztül HTTP kéréseket tudunk indítani a backend felé. A backend fejlesztés közben szinte elkerülhetetlen a Postman használata,

mivel azelőtt, hogy implementálnánk a backend funkciókat képesek vagyunk letesztelni, hogy a backendünk úgy működik-e ahogy szeretnénk.

Postmanben képesek vagyunk kollekciókat létrehozni, ezekhez teszt adatokat megadva tesztelni a szerver oldalt. Ezeket a teszteket automatizálni tudjuk így a Postman segítségével megoldhatjuk a backend tesztelését. [9]

2.4.7. Verzió követő és feladat vezérlő alkalmazások

Használt technológiák közé még felsorolnám a GitHubot és a Trello kanban boardot. Ezek a technológiák nem képeztek szerves részt az applikációban és nélkülük is lehetséges egy hasonló applikáció lefejlesztése. Viszont hatalmas segítséget nyújtanak a fejlesztés során.

A Github verzió követő rendszerrel minden egyes nagyobb változtatás esetén képesek vagyunk egy mentést készíteni az egész applikáció állapotáról a GitHub szerverein. Így, ha valamilyen probléma adódik, akkor a kódbázisunk mindig egy biztonságos helyen van és kedvünk szerint lépegethetünk az applikáció verziói között. Ez a funkció akkor remekel a legjobban, ha többen dolgozunk egy adott projekten vagy a kódbázisunkat szeretnénk más emberekkel megosztani.

A kanban boardokról röviden, egy olyan feladat kezelő applikációk, amelyekben fel tudjuk tüntetni a feladatainkat egy adott projekten. Ezeket a feladatokat különböző prioritásokkal tudjuk ellátni és követni, hogy melyik feladat éppen milyen fázisban van. Ez a módja a feladat megtervezésnek. Nagyon nagy teljesítményben való növekedést jelent egy applikáció lefejlesztésében és minél bonyolultabb az adott projekt, annál nagyobb igény van egy hasonló kanban board használatára.

3. fejezet

A rendszer specifikációi és architektúrája

3.1. Követelmény specifikációk

A következő fejezetben szemléltetem az applikáció fontosabb követelményeit attól függően, hogy éppen milyen szerepkörrel rendelkezik az adott felhasználó.

3.1.1. Funkcionális követelmények

A funkcionális követelmények határolják be az applikáció működését, ezek a követelmények határozzák meg, hogy bizonyos szerepkörök milyen funkcionalitásokat érnek el és mire van jogosultságuk. Az applikációban öt fajta szerepkört határoztunk meg, ezek a következők:

1. **Titkárok.**
2. **Tanszékvezetők.**
3. **Tanárok.**
4. **Hallgatók.**
5. **Vendégek.**

A szerepkörök között nem lehet egy pontos hierarchiát felállítani, mint más hasonló alkalmazásokban, mivel minden szerepkörnek teljesen más funkciói vannak az applikációban.

Közös funkcionalitások

Mivel a szerepkörök teljesen eltérőek, meghatározhatóak bizonyos funkcionalitások, amelyek minden egyes személykört érintenek, mint például:

1. Bejelentkezés.
2. Kijelentkezés.

Titkárok

A titkárok reprezentálják az admin szerepkört az applikációban. Az ő feladataik köré tartoznak olyan adatok bevitele, amelyek a későbbiekben meghatározzák az egész applikáció működését.

Titkárok funkcionalitásai:

1. Egyetemi év létrehozása.
2. Egyetemi évek megtekintése.
3. Időszakok létrehozása szakoknak.
4. Időszakok megtekintése.
5. Időszakok törlése.
6. Időszakok frissítése.
7. Dokumentumok feltöltése.
8. Dokumentumok törlése.
9. Tanárok feltöltése.
10. Tanszékvezetők feltöltése megadott szakhoz.
11. Hallgatók feltöltése.
12. Hallgatók törlése.
13. Hallgatók módosítása

Tanárok

A tanárok funkcionalitásai:

1. Diplomamunka létrehozás.
2. Diplomamunka frissítése.
3. Diplomamunka törlése.
4. Saját diplomamunkák megtekintése.
5. Diplomamunkára jelentkezett hallgatók megtekintése.
6. Üzenet írása a kivitelező hallgatónak.
7. Látni a hallgató által írt üzeneteket.
8. Letölteni a hallgató által feltöltött dokumentumot.
9. Osztályozni a diplomamunkát.

Tanszékvezetők

A tanszékvezetőknek ugyanazok a funkcionalitások állnak rendelkezésükre, mint a tanároknak, viszont kiegészülnek pár plusz funkcióval:

1. Látni a diplomára leosztott hallgatók és diplomamunkák listáját a saját szakukról.
2. Engedélyezni a leosztott eredményeket a saját szakukról.
3. Elutasítani a leosztott eredményeket a saját szakukról.
4. Véglegesíteni a leosztott eredményeket a saját szakukról.

Hallgatók

A hallgatók funkcionalitásai:

1. Megtekinteni a meghirdetett diplomamunkákat.
2. Jelentkezni diplomamunkákra.
3. Lejelentkezni diplomamunkákról.
4. Változtatni a jelentkezési prioritást.
5. Üzenet írása a vezető tanárnak.
6. Keresni az meghirdetett diplomamunkák között név és kulcsszavak alapján.
7. Fájlokat feltölteni.

Vendégek

A vendégek funkcionalitásai:

1. Megtekinteni az elvégzett diplomamunkákat.
2. Keresni az elvégzett diplomamunkák között név és kulcsszavak alapján.

Alább néhány fontosabb és bonyolultabb funkcionalitás tekinthető meg.

3.1. táblázat. Bejelentkezési funkció

Leírás	Ha már a felhasználó létezik az adatbázisban, de tud jelentkezni.
Kiváltás	A felhasználó belép az oldalra.
Funkcionális követelmény	A felhasználó ki kell töltsse a saját adataival a bejelentkezési űrlapot. A belépés akkor sikeres, ha az email címe megtalálható az adatbázisban és hozzá a megfelelő jelszót adta meg. Ha ezek helyesek át lesz irányítva a következő oldalra, ellenkező esetben egy hibaüzenet ugrik fel.

Megszorítás	Abban az esetben képes egy felhasználó bejelentkezni, ha már előzőleg meghívást kapott az applikációba a titkároktól. Meghívás során kap minden felhasználó egy e-mailt benne egy linkkel. A linkre kattintva képesek jelszót beállítani.
-------------	---

3.2. táblázat. Hallgatók feltöltése funkció

Leírás	Egy adott szakhoz a titkár feltölthet hallgatókat.
Kiváltás	A titkár létrehoz egy időszakot egy szaknak. Ez után a időszakok listájánál a létrehozott szaknál rákattint a plusz ember ikonra.
Funkcionális követelmény	A titkár választhat két feltöltési módszer között: <ol style="list-style-type: none"> 1. Egy űrlapot kitöltve megadja a hallgató családnévét, keresztnévét, e-mailjét és médiáját. 2. Feltölt egy CSV fájlt, ami tartalmazza egy vagy több hallgató adatait.
Megszorítás	

3.3. táblázat. Diplomamunka létrehozása funkció

Leírás	Tanár vagy tanszékvezető képes diplomamunkát létrehozni.
Kiváltás	Belép a tanár vagy tanszékvezető. Rákattint az Diplomamunka létrehozására.
Funkcionális követelmény	A tanár vagy tanszékvezető ki kell töltsse az űrlapot. Meg kell adja, hogy egyedül vagy más tanárral indítják a diplomamunkát. Kiválasztja, hogy melyik szakoknak akarja kiírni a diplomamunkát. Megadja az szükséges információkat, mint cím, leírás, kulcsszavak, bibliográfia és egyéb követelmények. Majd feltölti melléje a pdf absztraktot. Ezek után rákattint a diplomamunka létrehozására.
Megszorítás	Amíg a titkárok nem hozzák létre az időszakot a szakoknak, addig az ókat nem tudja azoknak kiírni.

3.4. táblázat. Véglegesíteni a leosztott eredményeket a saját szakukról funkció.

Leírás	A Tanszékvezető képes elutasítani vagy engedélyezni minden egyes diplomamunka leosztás eredményt, majd a döntése szerint véglegesíteni a diplomamunka jelentkezéseket.
Kiváltás	Tanszékvezető bejelentkezik az oldalra, majd átnavigál a Diplomamunka jelentkezések menüpontra.

Funkcionális követelmény	A tanszékvezető lát egy listát, ami tartalmazza diplomamunka címét, a vezető tanárt vagy tanárokat és a jelentkezett hallgatót. Egy kapcsoló gombbal tudja engedélyezni vagy elutasítani a diplomamunka elindulását. Ezt a műveletet tudja személyenként meghozni vagy egyszerre mindenkinek. Ha meghozta a döntést, akkor a véglegesítés gombra kattintva, felugrik egy figyelmeztetés, hogy ezt a döntést nem lehet későbbiekben megváltoztatni. Ezt elfogadva a diplomamunkák le lesznek osztva a diákoknak médiájuk figyelembevételével az általuk kiválasztott diplomamunkákra.
Megszorítás	A tanszékvezető meg kell várja míg a rendszer által le lesznek osztva média szerint a hallgatók a kiválasztott diplomamunkákra. Ha lejárt az első vagy második leosztási fázis dátuma mindegyik szaknál, akkor a rendszer leosztja a hallgatókat, ekkora megjelenik egy lista ezen az oldalon.

3.5. táblázat. Diplomamunkára jelentkezni funkció.

Leírás	A hallgatók képesek jelentkezni azokra a diplomákra, amelyek a szakokra lettek meghirdetve.
Kiváltás	A hallgató bejelentkezik. A diplomamunka listában rákattint valamilyen diplomamunkára átnavigálva a diplomamunka belsőoldalára.
Funkcionális követelmény	A hallgató képes informálódni a diplomamunkáról és a diplomamunka belső oldalán lévő „jelentkezek” kapcsolóval képes jelentkezni az adott diplomamunkára.
Megszorítás	Csak akkor tud jelentkezni egy diplomamunkára, ha eddig kevesebb mint 3 diplomára jelentkezett.

3.6. táblázat. Keresni az elvégzett diplomamunkák között név és kulcsszavak alapján funkció.

Leírás	Vendég képes böngészni az elkészített diplomamunkák között.
Kiváltás	Vendég belép az oldalra és rákattint a belépés vendégként gombra.
Funkcionális követelmény	A vendég az oldalon levő kereső mezőbe beírva „kulcsszavakat”, képes keresni a kész diplomamunkák között.
Megszorítás	Legyenek elvégzett diplomamunkák.

3.1.2. Nem funkcionális követelmények

Frontend

A frontend nem funkcionális követelményei közé tartozik, hogy szükséges telepítenünk a NodeJS. Angularhoz a szükséges modulokat nem tároljuk csakis lokálisan, mivel a `node_modules` mappa hatalmas méretű. A modulok letöltéséhez el kell navigálnunk a terminálon keresztül az applikáció könyvtárába és futtassuk az alábbi utasítást.

```
npm install
```

Majd, ha sikeresen fel lettek telepítve a modulok, akkor a következő paranccsal el tudjuk indítani az alkalmazást.

```
ng serve
```

Ha az alkalmazás elindul, akkor a böngészőben az alábbi URL-en fogjuk elérni: <http://localhost:4200/>

Backend

A backend futtatásához ajánlott IntelliJ IDEA fejlesztői környezetet használnunk. Az alkalmazás indítása előtt, szükséges telepítenünk a függőségeket. Ehhez a fejlesztői környezet jobb oldalán kiválasztjuk a Maven kezelőfelületét, és rákattintunk az install gombra. Ezek után a futtatás gombbal elindíthatjuk az alkalmazást, ami, ha sikeresen elindult akkor az alábbi URL-en fogjuk elérni: <http://localhost:8080/>

Postgres

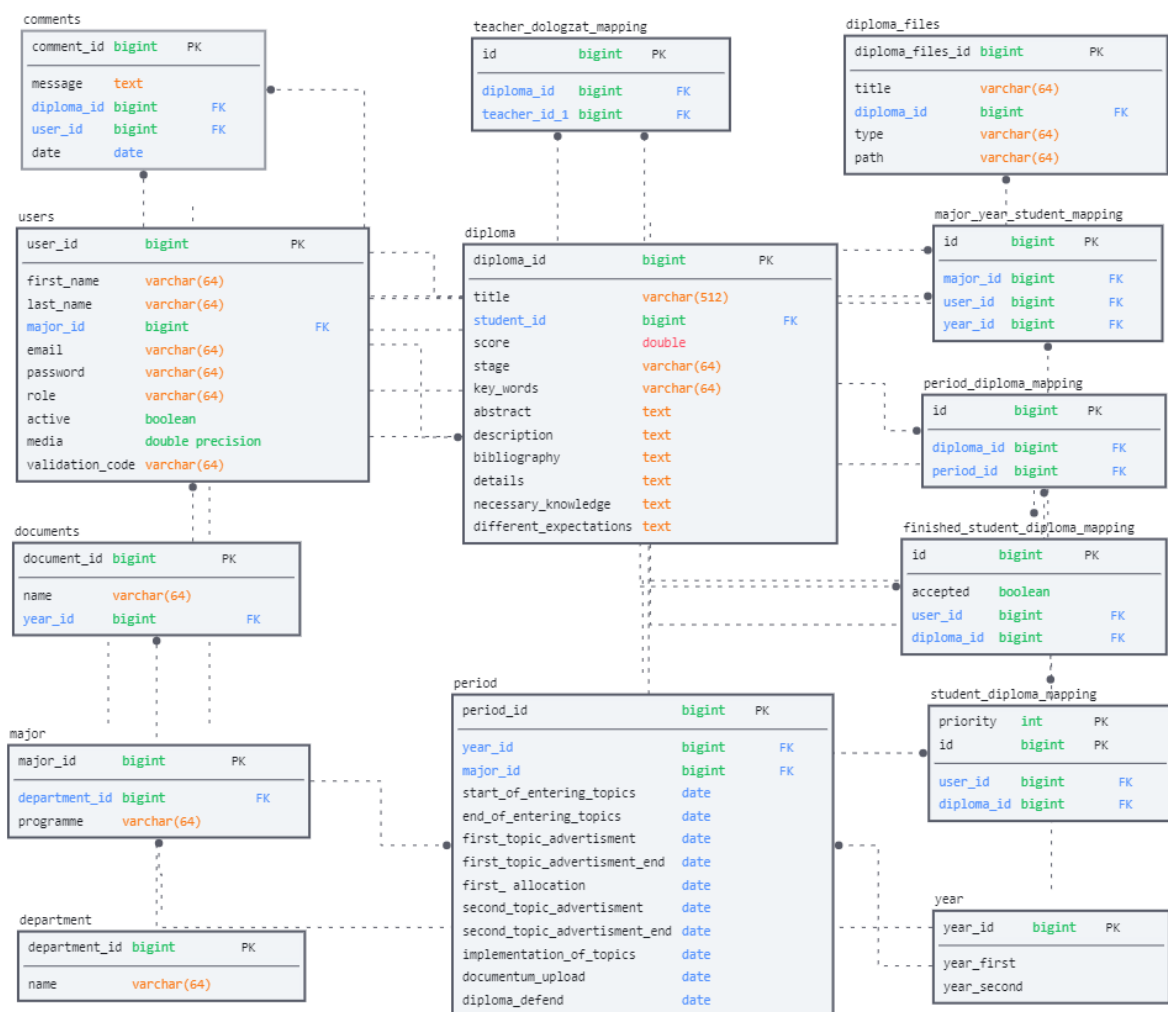
Az applikáció elindításához még szükséges telepítenünk a postgres-t és a hozzá járó pgAdmin-t kezelő felületet. A pgAdminban szükséges létrehozunk egy adatbázist a következő névvel: DW.

3.2. Adatbázis

Minden alkalmazásnak az alapját egy jól megtervezett adatbázis jelenti. Az adatbázisunkban nagyon sok egymástól függő adatot kell tároljunk. Az alábbi táblák vannak jelen az adatbázisunkban és itt megtekinthető az adatbázis diagram [3.1](#).

1. **users** : T: Tartalmazza a felhasználókat és azok adatait: családnév, keresztnév, email, jelszó, média, aktív státusz. A role oszlop határozza meg, hogy milyen típusú felhasználó. A **major_id**: egy **foreign key** amely kapcsolatot hoz létre a major táblával, így mindegyik felhasználónak meg tudjuk határozni, hogy melyik szakhoz tartozik. A **validation_code** tárolunk egy véletlen generált karakterláncot, amelyet kiküldünk a felhasználónak emailben invitáció során és ennek felhasználásával képes majd jelszót beállítani.
2. **diploma**: Tartalmazza a diplomamunka általános adatait, mint cím , bibliográfia, követelmények stb. A **stage**, hogy éppen milyen állapotban van, ez egy enum. Ez három fajta lehet: **PLAN**, **UNDER_IMPLEMENTATION**, **FINISHED**. A **student_id** **foreign key** pedig kapcsolatot hoz létre a **users** táblával, így minden diplomamunka egy hallgatóhoz lehet kiírva.
3. **diploma_files**: Tartalmazza a diplomához kötődő fájlok adatait, mint név, típus és elérési út. A **diploma_id** **foreign key** meghatározza, hogy melyik diplomához tartozik az adott fájl.

4. **department** : Tartalmazza a tanszékokat.
5. **major**:Tartalmazza a szakokat és a **department_id** foreign key megadja, hogy melyik tanszékhoz tartozik a szak.
6. **period**: Tartalmazza az időszakokat. Egy időszak tíz dátumból épül fel, amelyek meghatározzák, hogy egy diplomamunka létrehozásától a meghirdetésén keresztül a véglegesítéséig milyen határidőket kell betartani.
7. **year**: Tartalmazza az egyetemi éveket.
8. **document**: Tartalmazza a dokumentumokat az időszakokhoz. A **year_id** pedig egy évhez azonosítja.
9. **student_diploma_mapping**: : Tartalmazza a megfeleltetéseket a diplomamunkák és a hallgatók között. Mivel akármennyi hallgató képes egy adott diplomamunkára jelentkezni és több diplomamunkára képes egy hallgató jelentkezni ezért szükség volt ManyToMany relációra. Tartalmazza a **user_id**-t és **diploma_id**-t és egy prioritás oszlopot, amiben tárolva van, hogy a hallgató a három választása közül milyen prioritásban szeretné ezt a diplomamunkát megkapni.
10. **diploma_period_mapping**: Tartalmazza a megfeleltetéseket a diplomamunkák és a időszakok között. A **diploma_id** és **period_id** alapján tudjuk hogy az adott diploma melyik időszakhoz tartozik.
11. **finished_student_diploma_mapping**: Tartalmazza a leosztott diplomamunkák és hallgatók kapcsolatát. Mivel a leosztást engedélyeznie kell a tanszékvezetőnek, el kell mentünk az eredményt egy másik mapping táblába. A **user_id** és **diploma_id** meghatározza, hogy a hallgató melyik diplomamunkát kapta meg és az **accepted** oszlop hogy indulhat-e.
12. **major_year_student_mapping**: Tartalmazza az adott szak, hallgató és év kapcsolatát. Mivel egy hallgató egyetlen szakhoz tartozhat és egyetlen évhez, megkönnyíti a hallgatók kezelését, ha egy mapping táblába soroljuk őket az által, hogy a táblában megtalálható a **user_id**, **diploma_id** és **major_id**;
13. **teacher_diploma_mapping**: Tartalmazza a tanár és diplomamunka kapcsolatokat. Ha azt akarjuk, hogy egyszerre több vezető tanárt csatoljunk egy diplomamunkához, akkor szükséges erre a célra is létrehozzunk egy **ManyToMany** relációt. A tábla tartalmazza a **user_id**-t és **diploma_id**-t.
14. **comment**: Tartalmazza a felhasználók üzeneteit a diplomamunkához. Ahhoz hogy, tudjuk azonosítani, hogy egy diplomamunkánál milyen üzenetek lettek írva szükséges elmentenünk az üzenetet, a dátumot, amikor az üzenet íródott és a feladot **user_id** és hogy melyik diplomamunkánál lett elküldve **diploma_id**.



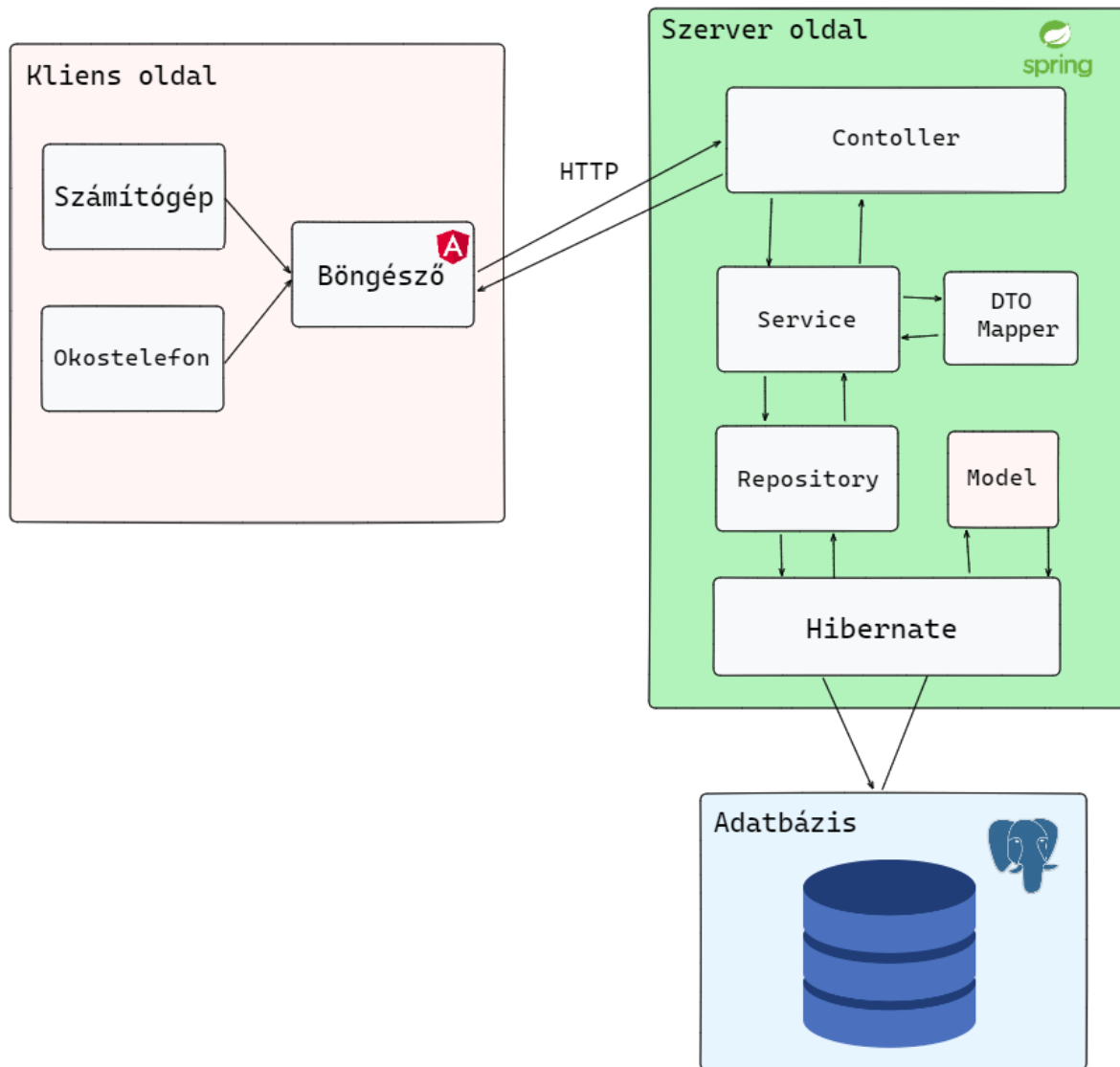
3.1. ábra. Adatbázis diagram

3.3. Alkalmazás architektúra

Az alkalmazás két fő részből épül fel:

1. Kliens oldal: böngésző.
2. Szerver oldal: alkalmazás.

A fentebb megemlített két fő rész további kisebb komponensekből épül fel, amely az alábbi ábrán 3.2 látható.



3.2. ábra. Alkalmazás architektúra [3]

A kliensoldalon akár egy számítógépet használva vagy okostelefont, meg tudjuk nyitni az applikáció frontendjét, ha a megfelelő URL-re navigálunk. Az Angular egy komponensekre épülő frontend keretrendszer. Minden komponens három fájlból épül fel.

1. **TypeScript** fájl, ez tartalmazza egy komponens logikai egységét.
2. **HTML** fájl, ez tartalmazza a direktívákat és a HTML kódot.
3. **CSS** fájl, ebben tudunk komponens szintű design-t létrehozni.

A TypeScript egyetlen osztályból épül fel, amelyik összeköti a fentebb felsorolt három fájlt. Ebben a fájlban a konstruktoron keresztül beinjektálhatunk Serviceket, amelyek felelősek a szerver oldallal való kommunikációért. A Servicek metódusait meghívhatjuk eventeken keresztül, amik a HTML fájlban esetleg gombokra vannak kötve, vagy használhatjuk az Angularban leimplementált **hookokat**, mint például az `ngOnInit`. Az

ngOnInit az egy lifecycle hook, amit akkor érhetünk el ha az osztályunk implementálja az **OnInit** interface- t, ez mindig lefut amikor éppen betöltjük a komponenst.

A Servicek HTTP protokolon keresztül kommunikálnak a szerver oldallal. Minden kérés a Controller osztályba érkezik meg, az előtt pedig egy filteren mennek végig ami ellenőrzi, hogy van-e jogunk a kéréshez vagy be vagyunk jelentkezve. Ha minden rendbe van akkor az adott kérést kiszolgáló metódus meghívódik, ami meghívja a Service réteg metódusát. A Service használja a Repository réteget a kért számítások elvégzéséhez vagy adatok manipulációjához. A Repository a Hibernetet felhasználva kéréstől függően a java objektumokat átalakítja adatbázis sorokká vagy az adatbázis sorokat java objektumokká.

Abban az esetben, ha valamilyen objektumot kell visszaküldjünk a kliens oldalnak, az által megy a DTO mapperen, ami új objektumot hoz létre a meglévőből, ami így csak a szükséges adatokat tartalmazza. Ez által nem léphet fel a lehetőség például, hogy a felhasználónak visszaküldjük a jelszavát is egy kérésnél.

Szerver API

A szerver oldalon a következő endpointok találhatók meg amiken keresztül tudunk kommunikálni a szerverrel.

1. UserController, "/user":

- (a) **GET /health-check**, szerver pingelése.
- (b) **GET /id**, id alapján felhasználó lekérés.
- (c) **GET /get-all**, összes felhasználó lekérése.
- (d) **GET /get-all-active**, összes aktív felhasználó lekérése.
- (e) **GET /get-all-teachers**, összes tanár lekérése.
- (f) **GET /get-all-students-for-period/periodID**, időszak szerint az összes hallgató lekérése.
- (g) **GET /check-preconditions/enums**, határidő elteltének lekérdezése.
- (h) **POST /login**, bejelentkezés.
- (i) **POST /upload-students/periodID**, egyszerre több diák feltöltése.
- (j) **POST /create-student**, hallgató létrehozása.
- (k) **POST /create-teacher**, tanár létrehozása.
- (l) **POST /change-password**, jelszó beállítása.
- (m) **PATCH /update-student**, hallgató frissítése.
- (n) **PATCH /update-teacher**, tanár frissítése.
- (o) **DELETE /delete/id**, felhasználó törlése.
- (p) **DELETE /delete-teacher/id**, tanár törlése.

2. YearController, "/year":

- (a) **GET /get-current**, aktuális év lekérése.
- (b) **GET /id**, id alapján év lekérése.

(c) **POST** /create, év létrehozása.

3. **PeriodController**, "/period":

- (a) **GET** /id, időszak id alapján lekérése.
- (b) **GET** /get-by-major-id/id, időszak szak id alapján lekérése.
- (c) **GET** /get-all-period-by-year, jelenlegi évből az összes időszak lekérése
- (d) **GET** /get-current-period-for-major/majorID, jelenlegi évhez tartozó időszak lekérése szak id szerint.
- (e) **GET** /get-all, összes időszak lekérése.
- (f) **POST** /create, év létrehozása.
- (g) **POST** /update, időszak frissítése
- (h) **DELETE** /delete/id, időszak törlése.

4. **MajorController**, "/major":

- (a) **GET** /get-all-without-period/yearID, időszakonkénti összes szak lekérése egyetemi év alapján.
- (b) **GET** /get-all, összes szak lekérése

5. **DocumentController**, "/document":

- (a) **GET** /get-all, minden egyetemi évnek a dokumentumai lekérése.
- (b) **DELETE** /delete/id, törlés id alapján.
- (c) **POST** /uploadFile/yearID, dokumentum feltöltése egyetemi évhez.
- (d) **POST** /uploadMultipleFiles/yearID, több dokumentum feltöltése egyetemi évhez.
- (e) **GET** /downloadFile/documentID, dokumentum letöltése.

6. **DiplomaFileController**, "/diploma-file":

- (a) **GET** /diplomaID/download-diploma-file/userID, diplomamunka fájl letöltése diplomamunka és felhasználó alapján.
- (b) **GET** /diplomaID/get-diploma-file/userID, diploma fájl lekérése diplomamunka és felhasználó alapján.
- (c) **POST** /diplomaID/upload/userID, diplomamunka fájl feltöltése diplomamunka és felhasználó alapján.
- (d) **DELETE** /delete/id, diploma fájl törlése diplomamunka és felhasználó alapján.

7. **DiplomaController**, "/diploma":

- (a) **GET** /get-all-finished, összes elvégzett diplomamunka lekérése.
- (b) **GET** /get-finished/id, elvégzett diplomamunka lekérése.

- (c) **GET** /get-all, összes diplomamunka lekérése.
 - (d) **GET** /get-my-diplomas, saját diplomamunkám lekérése.
 - (e) **GET** /get-all-visible-diplomas-for-given-major, összes diplomamunka lekérése a felhasználó szakjának.
 - (f) **GET** /get-by-id-for-student/diplomaID, diploma lekérése hallgatónak.
 - (g) **GET** /get-all-applied-diplomas-for-approving/id, összes diplomamunka jelentkezés lekérése tanszék id alapján.
 - (h) **GET** /get-all-applied-diplomas-for-student, összes diplomamunka jelentkezést a jelenlegi felhasználónak.
 - (i) **GET** /get-current-diploma, saját diplomamunka lekérése jelenlegi felhasználó szerint.
 - (j) **POST** /create, diplomamunka létrehozása.
 - (k) **POST** /diplomaID/enable-student-diploma/studentID, hallgató és diplomamunka leosztás engedélyezése diplomamunka és felhasználó alapján.
 - (l) **POST** /enable-all-student-diploma/allaccepted, összes jelentkezés engedélyezése vagy elutasítása.
 - (m) **POST** /finalize-applies, diplomamunka jelentkezések véglegesítése.
 - (n) **POST** /diplomaID/assign-to-diploma/userID, diplomamunkára jelentkezés diplomamunka és felhasználó alapján.
 - (o) **POST** /change-applied-priority/userID, jelentkezési prioritás változtatása felhasználó szerint.
 - (p) **POST** /id, diplomamunka id alapján lekérése.
 - (q) **POST** /set-score, diplomamunka pontozása.
 - (r) **PATCH** /update, diplomamunka frissítése.
 - (s) **PATCH** /sort-students-for-diploma, hallgatók leosztása diplomamunkákra.
 - (t) **DELETE** /delete/id, törlés id alapján.
8. **DepartmentController**, "/department":
- (a) **GET** /get-all, összes tanszék lekérése.
9. **CommentController**, "/comment":
- (a) **GET** /get-by-diploma/diplomaID, összes üzenet lekérése diplomamunka alapján.
 - (b) **POST** /create, üzenet létrehozása.
 - (c) **DELETE** /delete/id, üzenet törlése.

A fentebb megemlített endpointok használatához kevés kivétellel az összeshez egy azonosítót kell használnunk, hanem a szerver egy 403 hibakóddal tér vissza. Ha 403 hibakódot kapunk akkor a frontend automatikus törli az azonosítónkat és kijelentkeztet. Ezzel eltudjuk védeni, hogy minden egyes felhasználó csak a saját megengedett műveleteit tudja végrehajtani. Az azonosító az egy JSON Web Token (JWT), amely három részből épül fel.

1. **Fejléc**, ez tartalmazza a token típusát és a használt kriptográfiai algoritmust, általában SHA256.
2. **Tartalom**, ez egy JSON formátumból épül fel és bármilyen adatot tárolhatunk benne, a mi esetünkben a felhasználó adatait.
3. **Aláírás**, fentebbi kettőből épül fel és egy secret kulcsból amit csak a szerver ismer, emiatt mindig releváns.

[6] Az applikációban található pár endpoint amit meglehet hívni JWT nélkül, a login és a jelszócsere.

4. fejezet

Részletes tervezés

4.1. Tervezési fázisok

Az applikáció tervezését három fő elemre lehet építeni:

1. **Frontend.**
2. **Backend.**

4.1.1. Frontend

Azért érdekesebb a frontenddel kezdeni az újabb funkciók beépítését, mert egy pontosabb képet kapunk arról, hogy mire is lesz a későbbiekben szükségünk. Amíg nem tudjuk biztosan, hogy egy bizonyos komponens milyen adatokat fog felhasználni, nem tudjuk, hogy ahhoz a backendben milyen műveletekre van szükségünk, azok milyen objektumokat kell visszatérítsenek és milyen Controlleren keresztül.

Frontend projekt felépítés

A frontend a következőképpen épül fel. A projekt építő köveit a komponensek építik fel. Egy Angular komponens négy fájlból épül fel:

HTML

Egy HTML fájl, ami tartalmazza a HTML kódot, itt definiáljuk azt, hogy az adott komponensünk hogyan fogja elrendezni az oldalon az adatokat.

CSS

Ebben a fájlban helyezzük el a CSS osztályokat, amivel képesek vagyunk designolni az adott komponenset. Ezt a CSS fájlt csak is az adott komponensben lévő HTML fájl látja, így minden komponensnek külön-külön használhatjuk ugyanazokat az osztályneveket és nem lesz ütközés.

Spec

A spec fájlban tudunk teszteket írni az adott komponensre. Ezeket a spec fájlokat elindíthatjuk az applikáción kívül. Itt írhatunk komponens teszteket vagy akár unit teszteket is.

TypeScript

Ez a legfontosabb eleme egy komponensnek. Ez fogja össze a fentebb leírt komponenseket, azzal, hogy a fájlban található osztályt ellátjuk a **@Component**. annotációval

```
@Component({
  selector: 'app-diplomas',
  templateUrl: './diplomas.component.html',
  styleUrls: ['./diplomas.component.scss'],
})
```

A **selector**-al pedig tudunk adni egy nevet a komponensünknek, amivel ez által tudunk referálni más komponenseken belül és itt remekel az Angular. Ez akkor előnyös, ha egy olyan komponenst hozunk létre, amelyet többször szeretnénk használni, mint például egy preloader. Amikor létrehozunk egy ilyen osztályt adhatunk neki bemeneti értékeket vagy output értékeket és ezeken keresztül tud kommunikálni a meghívó és a meghívott komponens.

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-preloader',
  templateUrl: './preloader.component.html',
  styleUrls: ['./preloader.component.scss'],
})
export class PreloaderComponent {
  constructor() {}
  @Input() loaderMessage: string = '';
  @Input() transparentBg: boolean = false;

  ngOnInit(): void {}
}
```

A fenti kódban definiáltunk egy Preloader komponenst és az app-preloader nevet kapta, illetve definiáltunk két bemeneti változót egy loaderMessage és egy transparentBg. Így most már bárhol meghívható és minden meghívásnál különböző értékeket adhatunk meg a komponensnek ezeken a változókon keresztül.

```
@Component({
  <app-preloader *ngIf="isLoading"></app-preloader>
  <app-preloader
    *ngIf="isLoadingCustom"
    loaderMessage="Hallgatk feltltse , krem vrjon"></app-preloader>
})
```

A fenti kódban láthatjuk a preloader használatát egy másik kódban. Az `*ngIf` direktívával tudjuk szabályozni, hogy milyen feltétel esetén rajzolódjon ki az oldal. Az első meghíváskor abban a komponensben, amiben meghítuk, annak a TypeScript fájljában van definiálva egy **`isLoading`** változó, ami a komponens inicializálásakor `False` értéket vesz fel és ahogy minden HTTP kérés visszatért a szerverről `True` értéket vesz fel és eltűnik az oldalról.

A második esetben pedig ugyancsak egy direktívával van szabályozva, hogy mikor legyen látható, viszont ebben az esetben az egyik input mezőjét felhasználva egy üzenetet adunk át. Ezt az értéket később felhasználva a preloader HTML fájljában tudjuk változtatni, hogy éppen milyen szöveget írjon ki a preloader

Az `*ngIf`-en kívül számtalan hasznos direktívát tudunk használni a DOM manipulációjára. Egy másik gyakran használt az `*ngFor`, ami egy `for` ciklust hoz létre a HTML-en belül.

```
<ng-container *ngFor="let student of students">
  <div class="card my-2">
    <div class="card-header p-4 py-2">
      <div
        class="row align-items-center justify-content-between">
        <div class="col-auto me-5">
          <h4 class="m-0">
            {{ student.firstName }} {{ student.lastName }}
          </h4>
        </div>
        <div
          class="col-3 d-flex justify-content-center align-items-center">
          <button
            type="button"
            class="btn btn-icon btn-outline-warning me-2"
            (click)="selectStudent(student.id)">
            <i class="ri-edit-box-line"></i>
          </button>
          <button
            type="button"
            class="btn btn-icon btn-outline-danger"
            (click)="deleteStudent(student.id)">
            <i class="ri-delete-bin-line"></i>
          </button>
        </div>
      </div>
    </div>
  </div>
</ng-container>
```

A fenti kódrészletben látható az `*ngFor` használata. A `students` a TypeScript fájlban egy listát jelöl és így minden lista elemre kigenerálunk egy HTML elemet és minden elem az adott iteráció elemének az értékeit veszi fel.

Modulok

A modulok azok a komponenseken felül állnak. Több komponenst fog össze egy-egy modul. Ezeket a modulokat pedig a szerint határozzuk meg, hogy az adott komponensek,

amik egy modulhoz tartoznak, milyen szerepkört töltenek be. Egy modul két részből épül fel. Tartalmaz egy **routes** tömböt amiben fel vannak sorolva, hogy egy-egy komponenst milyen URL-n keresztül érhetünk el, az alábbi kódban illusztrálva

```
const routes: Routes = [
  {
    path: 'periods',
    component: PeriodsComponent,
  },
  {
    path: 'create-period/:id',
    component: CreatePeriodComponent,
  },
  {
    path: 'period/:id',
    component: PeriodComponent,
  },
]
```

Illetve tartalmazza azokat a komponenseket, amiket egy másik Modul foglal magába és a mi modulunkat akarjuk használni, ezek fel vannak sorolva az importsban. Tartalmaz egy másik felsorolást arról, hogy az adott modulból milyen komponenseket exportálunk, hogy más modulok is használhassák. A legvégén pedig azt, hogy milyen komponenseket deklaráltunk a modulon belül.

```
@NgModule({
  declarations: [
    PeriodsComponent,
    CreatePeriodComponent,
    PeriodComponent,
    AddStudentsComponent,
    AddTeacherComponent,
  ],
  imports: [
    CommonModule,
    RouterModule.forChild(routes),
    BsDatepickerModule,
    FormsModule,
    ReactiveFormsModule,
    NgbModule,
    SharedModule,
  ],
})
```

Az applikációban lévő modulok a következők.

1. **Account.**
2. **Admin.**
3. **Teacher.**
4. **Student.**
5. **Guest.**

Account

Az account modul fogja össze azokat a komponenseket amelyek az applikáció kívülről való eléréséhez szolgálnak.

1. **Change Password.**

2. **Login.**

A login komponensen keresztül vagyunk képesek bejelentkezni az applikációba. A change passwordot pedig meghívó emailen keresztül érhetjük el.

Attól függően, hogy a felhasználónknak milyen role-ja van más URL-re fog irányítani minket az applikáció és elmenti a felhasználónkat a cookie-ba és a szervertől kapott JWT-t, amit később felhasználunk minden egyes HTTP kérésnél.

Admin

Az admin modul a következő komponenseket tartalmazza.

1. **add-student.**

2. **add-teacher.**

3. **create-period**

4. **period.**

5. **periods.**

Amikor átirányít minket az applikáció bejelentkezés után, akkor elsőnek a periods komponens töltődik be. Itt láthatjuk az összes létrehozott időszakot és itt tudunk újakat létrehozni, illetve dokumentumokat feltölteni. A komponens inicializálásakor egyszerre több HTTP kérést küldünk a szervernek, amíg ezek nem töltődnek be, addig egy preloader-t látunk. A kérések megérkezése után eltüntetjük a preloader-t és betöltjük a teljes HTML-t a megérkezett adatokkal.

A létrehozott időszakokat egy másik komponensen keresztül lehet megtekinteni vagy frissíteni. Az időszakok listájában valamelyikre rákattintva, berakja annak az id-ját az URL-be és átnavigál a period/id-ra. A komponens inicializálásakor kiveszük az id-t az URL-ből és lekérjük a szervertől az adott id-val rendelkező időszakot. Egy időszakot törölhetünk vagy hallgatókat tölthetünk fel hozzá.

Hallgatók feltöltése esetén átnavigál az időszak id-jával együtt az add-students komponensre, ahol fel lehet tölteni a hallgatókat egyenként, vagy egy CSV fájlon keresztül. Ha a CSV fájlon keresztül szeretnénk feltölteni akkor a következő konvenciót kell betartsuk: családnév, keresztnév, email, média. Abban az esetben, ha nem talál a konvenció vagy valamelyik email már használatban van az adatbázisban, akkor a szerver hibaüzenet küld vissza.

A create-periods komponensen belül pedig új időszakot tudunk létrehozni. Ekkor kérést küldünk a szervernek, hogy adja vissza azokat a szakokat, amelyek nem rendelkeznek időszakkal. Minden dátumot kitöltve létrehozhatjuk az időszakot. A frontend oldalon minden input fieldre validálás van létrehozva. Ebben az esetben validálva van, hogy a

dátumok azok egymás követő dátumok kell legyenek. Az alábbi kódban látható ennek megvalósítása.

```
validetDates() {
  let keys = Object.keys(this.periodDto);
  for (let i = 0; i < keys.length; i++) {
    if (keys[i + 1] != null) {
      if (this.periodDto[keys[i]] > this.periodDto[keys[i + 1]]) {
        this.createPeriodForm.get(keys[i]).setValue('');
        this.createPeriodForm.get(keys[i + 1]).setValue('');
        this.toastrService.toastrError('Hibs dtumok!');
        return false;
      }
    }
  }
  return true;
}
```

Abban az esetben, ha nem valamilyen egyedi validálásról van szó, akkor használhatjuk az Angulárban beépített formokhoz tartozó validálásokat. Az Angular Formok segítségével dinamikus formokat lehet létrehozni, megtudjuk adni, hogy mi legyen a kezdő értéke a fieldnek és hogy milyen validálással legyen ellátva. Például az alábbi példában automatikusan validálva van, hogy mindegyik fieldet ki kell tölteni.

```
initForm() {
  this.createPeriodForm = this._formBuilder.group({
    major: [{}, Validators.required],
    startOfEnteringTopics: ['', Validators.required],
    endOfEnteringTopics: ['', Validators.required],
    firstTopicAdvertisement: ['', Validators.required],
    firstTopicAdvertisementEnd: ['', Validators.required],
    firstAllocation: ['', Validators.required],
    secondTopicAdvertisement: ['', Validators.required],
    secondTopicAdvertisementEnd: ['', Validators.required],
    secondAllocation: ['', Validators.required],
    implementationOfTopics: ['', Validators.required],
    documentumUpload: ['', Validators.required],
    diplomaDefend: ['', Validators.required],
  });
  this.submitted = false;
  this.loading = false;
}
```

Elküldés előtt le lehet kérni a form állapotát és megtudni, hogy minden validálási feltétel teljesült-e. Az alábbi kódon látható egy példa az ellenőrzésre és HTTP kérés elküldésére és annak válaszára való felíratkozásra.

```

    create() {
      this.submitted = true;

      if (this.createPeriodForm.invalid) {
        return;
      }

      if (!this.validateDates()) {
        return;
      }

      this.periodDto.major = this.majors.filter(
        (x) => x.majorId == this.f.major.value
      )[0];

      if (this.periodDto.major == null) {
        this.toastrService.toastrError('Vlasz ki egy szakot!');
        return;
      }

      this.periodDto.year = this.yearDto;
      this.periodService
        .create(this.periodDto)
        .pipe(first())
        .subscribe({
          next: (periodDto) => {
            this.router.navigate(['admin/periods']);
            this.toastrService.toastrSuccess(
              'Idszak sikeresen létrehozva!'
            );
          },
          error: (e) => {
            this.toastrService.toastrError(
              'Hiba lépett fel az idszak létrehozása közben!'
            );
          },
        });
    }
  }

```

A periodService az egy Angular Service, ezekben a servicekben találhatók meg a HTTP kérések. Mindegyik service egy adott szerver oldalon található Controllerhez tartozó endpointoknak a HTTP kéréseit tartalmazza. Ezek a serviceket minden komponens konstruktorán keresztül injektálódnak be a komponensbe.

Egy serviceben található függvény így épül fel.

```

export class PeriodService {
  constructor(private http: HttpClient) {}

  create(periodDto: PeriodDto): Observable<PeriodDto> {
    return this.http.post<PeriodDto>({
      url: `${environment.apiUrl}/period/create`,
      body: periodDto
    });
  }
}

```

A HttpClient Service-t felhasználva képesek vagyunk HTTP kéréseket létrehozni. Megadjuk, hogy milyen típusú HTTP metódust akarunk használni, az milyen típusú objektumot fog visszaküldeni, megadjuk az URL-t és csatoljuk a bodyba a küldendő objektumot. Erre a függvényre fel tudunk iratkozni és attól függően, hogy milyen státusz kóddal tér vissza a kérés, a feliratkozásnak annak az ágába fog belemenni. Ha 200 kóddal tér vissza a kérés akkor a **subscribe**-nak a **next**-ágába megy bele, ellenkező esetben pedig az error-ágba. A **next**-ágból ki lehet nyerni a válasz bodyjában található objektumot és az error-ágban a hibaüzenetet.

Ahhoz, hogy ki tudjuk nyerni helyesen az információt, a szerveren definiált modellekhez egyenlő modelleket kell létrehozzunk frontenden is, így a GSON át tudja mappelni a válaszból az értékeket egy JavaScript objektumba.

```

export class UserDto {
  id: number;
  firstName: string;
  lastName: string;
  role: string;
  active: boolean;
  email: string;
  majorDto: MajorDto;
  infoName?: string;
  media?: number;
  status?: UserStatus;
  token?: string;
  department?: DepartmentDto;
}

```

```

@Data
public class UserResponseDto {
  private Long id;
  private String email;
  private String firstName;
  private String lastName;
  private String role;
  private Boolean active;
  private Double media;
  private MajorDto majorDto;
  private UserStatus status;
  private DepartmentDto department;
}

```

4.1. ábra. Frontend oldali typescript osztály

4.2. ábra. Szerver oldali java osztály

Az add-teacher komponensen belül tudunk új tanárt vagy tanszékvezetőt hozzáadni. Képesek vagyunk ugyan azon az oldalon belül editálni is őket vagy törölni.

Teacher

A teacher modulban vannak a Tanárok és Tanszékvezetők komponensei, ezek az alábbiak:

1. create-diploma.
2. diploma.

3. diplomas

4. diploma-applies.

5. update-diploma.

Ha tanár vagy tanszékvezető role-al léptünk be, akkor elsőnek a diplomas komponenst fogad minket. Itt találhatóak az általunk létrehozott diplomamunkák. Különböző színekkel van jelölve, hogy az adott diplomamunka milyen állapotban van. Amihez még nincsen hallgató kiválasztva, az szürke, amelyik éppen kivitelezés alatt van, az sárga és zölddel, amelyik már kész van. Ahhoz, hogy kódból képesek legyünk változtatni a HTML elemeken a CSS-t, ahhoz egy másik direktívát kell használnunk, az **ngClass**-t. Itt megadhatjuk, hogy bizonyos feltételek esetén milyen CSS osztályt rakjon a HTML elementre.

```
<div
  class="card-header"
  [ngClass]="{
    'bg-light':
      getDiplomaStage(diploma.stage) === 'PLAN',
    'bg-warning':
      getDiplomaStage(diploma.stage) ===
      'UNDER_IMPLEMENTATION',
    'bg-primary':
      getDiplomaStage(diploma.stage) === 'FINISHED'
  }">
  <h6 class="card-title mb-0">
    {{ diploma.title }}
  </h6>
</div>
```

A diplomamunka nevére rákattintva átnavigál minket a diploma belső oldalára, itt az `*ngIf` direktíva segítségével, attól függően, hogy milyen stádiumban van a diplomamunka más fajta belső oldalt jelenítünk meg. Ha még nincs kivitelező hallgató a diplomamunkára, akkor a főoldalon kívül, ahol az általános információk vannak, meg tudjuk tekinteni a jelentkezett hallgatókat. Ha már kivitelezési stádiumban van, akkor megnyílik a beszélgetés fül a hallgatóval, illetve a fül, ahol le tudjuk tölteni a hallgató által feltöltött fájlt.

Ezen az oldalon keresztül érhetjük az `update-diploma` komponenst, ahol frissíthetjük a diplomamunka adatait.

A diplomas komponensen még található egy diploma létrehozása gomb, amin keresztül létre tudunk hozni egy új diplomamunkát. A létrehozásnál szükséges, hogy csatoljuk az absztraktot, ehhez változtatnunk kell a MIME-t a HTTP küldésnél. Azzal, hogy a headerbe a Content-Type-ot átállítjuk `multipart/form-data`-ra jelezzük a backendnek, hogy a kérés bodyjában fájlok lesznek.

```

create(formData: FormData): Observable<DiplomaDto> {
  const headers = new HttpHeaders();
  headers.set('Content-Type', 'multipart/form-data');

  return this.http.post<DiplomaDto>(
    `${environment.apiUrl}/diploma/create`,
    formData,
    { headers }
  );
}

```

Student

A student modulban található az összes hallgatóhoz tartozó komponensek:

1. **diploma.**
2. **diplomas**

A hallgató belépése után a diplomas komponens töltődik be. Ezen az oldalon láthatja az összes meghírdetett diplomamunkát, és hogy eddig melyekre jelentkezett. Egy drop-down megoldással képes változtatni a prioritást a jelentkezett diplomamunkák között. Egy diplomamunkának a belső oldalára lépve, láthatja az összes információt a diplomamunkáról, illetve egy kapcsoló gombon keresztül képes jelentkezni rá.

Abban az esetben, ha a hallgató már ki lett választva egy diplomamunkára, akkor a frontend a diplomas komponens helyett automatikusan az annak kiválasztott diplomamunka belsőoldalára irányítja át. Itt, mint a tanár esetében ugyanazok a fülek nyílnak meg a hallgató számára is. A hallgató képes lesz üzenetet írni a tanárnak és feltölteni a diplomamunka fájlt.

Guest

A guest modul a külső személyeknek lett készítve, ezek a felhasználók nem rendelkeznek teljeskörű felhasználóval a platformhoz, csak az elkészített diplomamunkákat képesek böngészni. Az alábbi komponensekben olyan HTTP kérések vannak, amikhez nem kell JWT, így bárki megtudja nézni őket.

A guest modul komponensei:

1. **diploma.**
2. **diplomas**

A diplomas komponensben látja az összes kész diplomamunkát és tud keresni bennük, és valamelyikre rákattintva beölti annak a belső oldalát, ahol letöltheti a diplomamunka fájlt.

A fentebb megemlített modulokat összeköti a legfelsőbb modul, az app module. Az **app-routing.module**-ban vannak definiálva a az alap routingok.

```

const routes: Routes = [
  {
    path: 'admin',
    canActivate: [AuthAdminGuard],
    component: LayoutComponent,
    loadChildren: () =>
      import('./admin/admin.module').then((m) => m.AdminModule),
  },
  {
    path: 'teacher',
    canActivate: [AuthTeacherGouard],
    component: LayoutComponent,
    loadChildren: () =>
      import('./teacher/teacher.module').then((m) => m.TeacherModule),
  },
  {
    path: 'student',
    component: LayoutComponent,
    loadChildren: () =>
      import('./student/student.module').then((m) => m.StudentModule),
  },
  {
    path: 'guest',
    component: LayoutComponent,
    loadChildren: () =>
      import('./guest/guest.module').then((m) => m.GuestModule),
  },
  {
    path: 'access',
    loadChildren: () =>
      import('./account/account.module').then((m) => m.AccountModule),
  },
  {
    path: '',
    redirectTo: '/access/login',
    pathMatch: 'full',
  }
]

```

A fenti kódban láthatjuk az alap routingok definiálását. Attól függően, hogy éppen az URL mivel kezdődik, azt a modult fogja betölteni s majd azon belül a modul megoldja a komponensek betöltését. Itt a felső szinten használva van egy filterezés a **canActivate** által, ami ellenőrzi, hogy az adott URLre csak azok tudjanak rálépni, akiknek a roleja megengedi. Így le van védve, hogy például egy hallgató ne tudjon rálépni egy tanár komponenseire. Az alábbi kódban látható az admin filternek megvalósítása

```

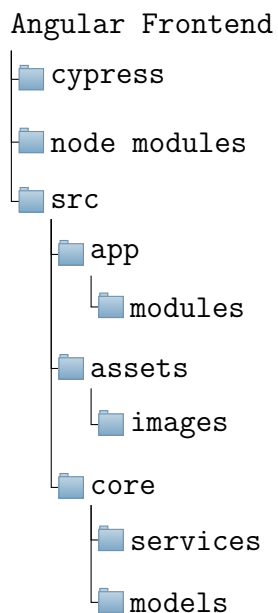
@Injectables()
export class AuthAdminGuard implements CanActivate {
  constructor(
    private cookieService: CookieService,
    private router: Router,
    private authService: AuthenticationService
  ) {}

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): boolean | Promise<boolean> {
    let currentUser = JSON.parse(
      this.cookieService.getCookie('currentUser')
    );
    if (currentUser === null || currentUser === '') {
      this.authService.logout();
    }
    if (currentUser.role !== 'admin') {
      this.authService.logout();
    }
    return true;
  }
}

```

Abban az esetben, ha nincs a felhasználó bejelentkezve vagy a felhasználónak a roleja nem Admin, akkor a rendszer kijelentkezeti és átdobja a bejelentkezési oldalra.

Frontend könyvtár struktúra



4.1.2. Backend

Ha tiszta, hogy a Frontenden milyen adatokra van szükség, elkezdhetjük a backenden is felépíteni az adott funkcionalitást. Fenti 3.2 ábrán a szerv oldalt tekintve sokkal több különálló komponenst figyelhetünk meg. A Controller komponens fogja fel a HTTP kéréseket és társítja hozzá a metódusokat. Viszont az előtt, hogy a controller osztályba beérkeznének a kérések, egy Security filteresen mennek keresztül. A filterezésnél figyelembe lesz véve, hogy elsősorban a domain, amiről a kérés érkezik, az engedélyezve van-e. Ha a domain talál, akkor az adott endpoint, amit el akar érni, az létezik-e, ha igen akkor szükséges-e JWT hozzá. Ha nem szükséges JWT, akkor a Controller osztályban meghívódik a kéréshez tartozó metódus és elkezdődik a kérés kiértékelése. Ha kell hozzá JWT, akkor ellenőrzi, hogy a JWT valódi-e és ha igen, akkor még aktív-e.

Controller

Az filterezés után belépünk az első rétegbe a Controllerbe. Minden Controller osztály egy bizonyos szerepkörrel, egy modelhez tartozó endpointokat tartalmaz. A szerepkörök nem keveredhetnek, így minden Controller meghatározza, hogy melyik modelhez tartozó számításokat képes elvégezni. Egy Controller osztály implementációja az alábbi kódban látható.

```
@Controller
@RequestMapping("/diploma")
@CrossOrigin
@AllArgsConstructor
public class DiplomaController {
    private final DiplomaService diplomaService;

    @PostMapping(value="/create")
    public ResponseEntity<?> create(@RequestPart(value = "file") MultipartFile file,
        @RequestPart(value = "DiplomaDto") DiplomaDto diplomaDto ){
        try {
            return ResponseEntity.ok().body(diplomaService.create(file,diplomaDto));
        }catch (Exception e){
            return ResponseEntity.badRequest().body(e.getMessage());
        }
    }

    @GetMapping("/{id}")
    public ResponseEntity<?> create(@PathVariable Long id){
        try {
            return ResponseEntity.ok().body(diplomaService.getByID(id));
        }catch (Exception e){
            return ResponseEntity.badRequest().body(new
                ErrorResponseDto(e.getMessage()));
        }
    }
    ...
}
```

Minden Controller osztály a következő annotációkkal vannak ellátva.

1. Controller.

2. RequestMapping

3. CrossOrigin

4. AllArgsConstructor

1. Meghatározza, hogy ez egy controller osztály.
2. Ez határozza meg a Controller osztály URL-jét.
3. Engedélyezi a Cross Origin kéréseket.
4. Kigenerál egy konstruktort minden adattagnak, amin keresztül később be lesznek injektálva a függőségek.

Minden egyes metódust ellátunk egy annotációval, ami meghatározza, hogy mi az endpoint neve és milyen típusú HTTP metódust vár el. A metódusnak a paraméter listájában más annotációkkal képesek vagyunk a kérés bodyjából vagy paraméteréből adatokat kinyerni. A `@PathVariable` Long id-el ki lehet nyerni az URLben található id-t, a `@RequestBody` DiplomaDto diplomaDto egy teljes objektumot a bodyból vagy több elszeparált objektumot `@RequestPart(value = "DiplomaDto") DiplomaDto diplomaDto` annotációval.

Service

Miután sikerült kinyerni az adatot a HTTP kérésből, meghívódik a Controllerhez tartozó Service réteg egyik metódusa. A Service réteg az mindig egy interfaceből épül fel, az interfaceben tudjuk deklarálni a metódusainkat, amit később egy implementációért felelős osztály implementál.

A Service réteg felel az összes logikai művelet elvégzéséhez és ez köti össze a legtöbb komponenst. A többi komponenst felhasználva elégíti ki a kéréseket. A metódus mielőtt feldolgozná a kérést, validálja az érkező adatokat és ellenőrzi, hogy azok az adatok amikkel dolgozni szeretnénk léteznek-e az adatbázisban. Az alábbi kódban látható ennek a megvalósítása.

```

@Override
public List<FinishedSDMappingDto> getAllDiplomaApplies(Long id) {
    if(id == null){
        throw new IllegalArgumentException("Id cannot be null!");
    }
    DepartmentEntity departmentEntity=
        departmentRepository.findById(id).orElseThrow(()->{
            throw new EntityNotFoundException("Entity not found!");
        });
    List<FinishedSDMappingDto> finishedSDMappingDtos = new ArrayList<>();

    finishedDSMRepository.findAll().forEach(x->{
        if(diplomaPeriodMappingRepository.
            existsByPeriod_Major_DepartmentEntityAndDiploma_DiplomaId(departmentEntity,
x.getDiploma().getDiplomaId())){
            finishedSDMappingDtos.add(finishedSDMappingMapper.toDto(x));
        }
    });

    return finishedSDMappingDtos;
}

```

Minden esetben amikor validálási hiba van, vagy bármilyen futásidejű hiba lép fel, egészen a controller osztályig tovább dobjuk a hibákat. Így a controller függvények catch ágában ezekben az esetekben visszaküldünk egy 400 error kódot a frontendre.

Repository

A Repository réteg felelős az adatbázis elérésért. Minden táblának létrehozunk egy interface-t, amely kiterjeszti a JPA interface-t. A JPA Interface egyik legismertebb implementációját a Hibernetet felhasználva objektumokat tudunk menteni az adatbázisba vagy adatbázis sorokat átalakítani java objektumokká. Az alábbi kódban látszik egy Repository interface implementálása.

```

@Repository
public interface DiplomaRepository extends JpaRepository<DiplomaEntity, Long> {
    List<DiplomaEntity> findByVisibility(Integer visibility);
    List<DiplomaEntity> findByStudentNotNull();
    List<DiplomaEntity> findByStudentNull();
    DiplomaEntity findByStudent_Id(Long id);
    boolean existsByStudent_IdAndDiplomaId(Long id, Long diplomaId);
    List<DiplomaEntity> findByStage(DiplomaStages stage);
}

```

Ahhoz, hogy a Repository tudja, hogy milyen objektumokra kell mappelje az adatbázis sorokat, létre kell hozunk egy model-t. Minden model az egy Java POJO, ami annyit takar hogy egy olyan objektum aminek csak konstruktorra, setterje és getterjei vannak. Ezeket a **Java POJO**-kat el kell lássuk egy **@Entity** annotációval, ami jelzi hogy ez egy adatbázis sorhoz mappelhető osztály, illetve egy **@Table** annotációval, ami megadja hogy pontosan melyik táblát is szeretnénk erre az osztályra rámapelni. Az osztály minden attribútumát el kell lássuk egy **@Column** annotációval, amivel megtudjuk határozni, hogy az adott attribútum melyik adatbázis oszlophoz feleltethető meg.

Ezenkívül sok más információval elláthatjuk az attribútumokat hasonló annotációk segítségével. Megadhatjuk, hogy melyik az **ID** -ja az osztálynak, az **@ID** annotációval és annak milyen típusú generálása van, az applikáció esetében auto increment.

Ugyanitt annotációk segítségével megadhatjuk, melyik attributum reprezentál egy **foreign key**-t. Ehhez meg kell adjuk a kapcsolat típusát ami lehet:

1. **@ManyToMany**.
2. **@OneToMany**.
3. **@ManyToOne**.

Ilyenkor, amikor lekérünk egy sort az adatbázisból, amelyik átmappelődik egy Java POJOba, ha annak volt valamilyen foreign key kapcsolata egy másik táblába, akkor abban az attributumban benne lesz a másik tábla teljes Java POJO leképezése. Az alábbi kód bemutat egy **@Entity** osztályt.

```
@Entity
@Table(name = "major")
@Data
public class MajorEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "major_id")
    private Long majorId;

    @Column(name = "programme")
    private String programme;

    @JoinColumn(name = "fk_department_id")
    @OneToOne(cascade = CascadeType.DETACH)
    private DepartmentEntity departmentEntity;

    @Column(name = "diploma_type")
    private String diplomaType;
}
```

A **@Data** annotáció segít abban, hogy ne kelljen minden egyes adattagnak megírni a settert, gettert vagy konstruktort vagy toStringet, hanem a háttérben kigenerálja ezeket automatikusan, így nem kell annyi ismétlődő kódot írjunk.

DTO Mapperek

A Dto mapperek akkor jönnek képbe, amikor sikeresen elvégeztük a számításokat és már csak vissza kell küldjünk az információt a frontendre. Viszont nem minden esetben úgy kell visszatéríteni az értékeket az adatbázisból, mint ahogyan azok tárolva vannak. Lehet, hogy az objektból kevesebb adatot kell visszaküldjünk vagy ki kell egészítjünk azt.

Ezt orvosolva létrehozunk be és kimeneti DTOkat. Ezek is simma Java POJO osztályok, viszont más adattagokat tartalmaznak, mint egy teljes leképezése egy adatbázis sornak. Mint, például a felhasználó esetében, mikor bejelentkezik csak egy email-t és jelszót várunk el és bejelentkezés után vissza akarjuk küldeni a saját user objektumát, de ott pedig nem szeretnénk, ha például benne lenne a jelszava.

Ezért létrehozunk mapper osztályokat, amik az Entitykből vagy Entitykbe alakítják a ki- és bemeneti DTO objektumokat.

Hallgatók leosztása

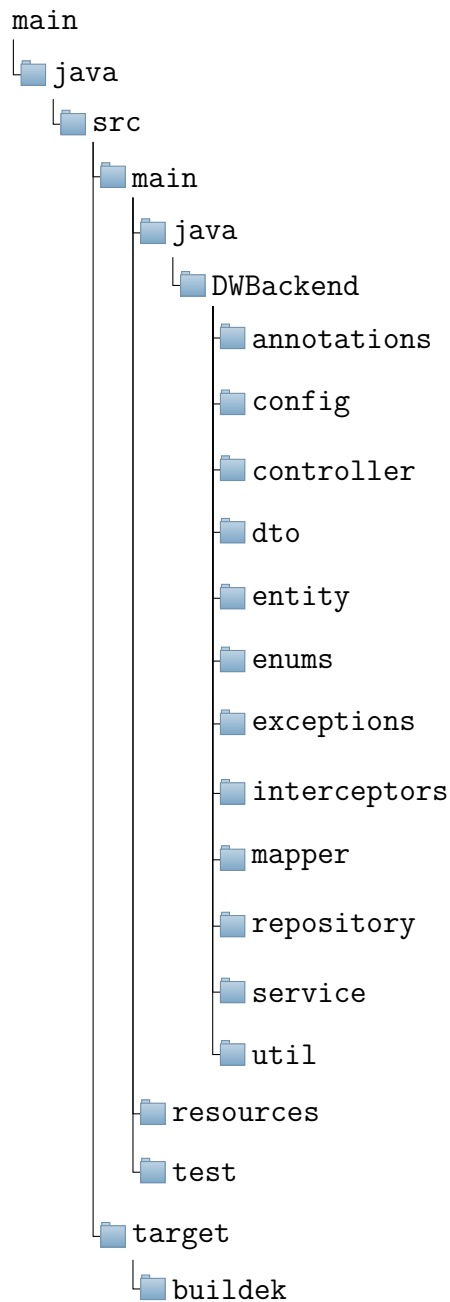
A hallgatók leosztását az alábbi service függvény oldja me.

```
@Override
public void sortStudentsForDiploma(){
    List<UserEntity> students = userRepository.getAllUsersFromDiplomaMapping();

    students.sort(Comparator.comparing(UserEntity::getMedia).reversed());

    for(UserEntity student : students){
        if(student.getStatus().equals(UserStatus.SEARCHING)){
            List<StudentDiplomaMappingEntity> studentDiplomaMappingEntities =
                studentDiplomaMappingRepository
                    .findByStudent_IdOrderByPriorityAsc(student.getId());
            for(StudentDiplomaMappingEntity mapping : studentDiplomaMappingEntities){
                if(!finishedDSMRepository
                    .existsByDiploma_DiplomaId(mapping.getDiploma().getDiplomaId())){
                    FinishedStudentDiplomaMappingEntity fsdme = new
                        FinishedStudentDiplomaMappingEntity();
                    fsdme.setStudent(student);
                    fsdme.setDiploma(mapping.getDiploma());
                    fsdme.setAccepted(false);
                    finishedDSMRepository.save(fsdme);
                    break;
                }
            }
        }
    }
}
```

Backend könyvtár struktúra



Resources

A resources mappán található még pár konfigurációs fájl és ide mentődnek a feltöltött fájlok. Az **application.properties** fájlban található pár konfigurációs beállítás, mint például az adatbázis csatlakozáshoz szükséges adatok.

Illetve itt találhatóak a Liquibase konfigurációs fájlok. Ezekbe írhatjuk az adatbázis változtatásokat, amik mindig napra készen tartják az adatbázist. Az alábbi kódban látható egy tábla leírása liquibaseben. .

Illetve itt találhatóak a liquibase konfigurációs fájlok. Ezekbe írhatjuk az adatbázis változtatásokat, amik mindig napra készen tartják

az adatbázis. Az alábbi kódban látható egy tábla leírása liquibaseben.

```
<databaseChangeLog
  xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.4.xsd">

  <changeSet id="2023_01_14_created_department" author="apal">
    <createTable tableName="department">
      <column name="department_id" type="bigint" autoIncrement="true">
        <constraints primaryKey="true"/>
      </column>
      <column name="faculty" type="varchar(64)"/>
      <column name="name" type="varchar(64)"/>
    </createTable>
  </changeSet>
</databaseChangeLog>
```

4.2. Tesztelés

Az applikácónak a tesztelése két részre van felbontva.

1. **Frontend tesztelés.**
2. **Backend tesztelés.**

4.2.1. Frontend tesztelés

A frontend tesztelést a Cypress keretrendszer oldotta meg. A frontend könyvtárában Cypress teszt fájlokat lehet elmenteni, amik a Cypress környezetében listázva lesznek és bármikor lefutathatjuk.

A tesztelés során nem vagyunk képesek garantálni, hogy az alkalmazunk teljesen hibamentes, de nagyobb funkcionalitások tesztelése esetén nagyjából el lehet mondani, hogy az applikáció jól működik.

Egy Cypress előnye, hogy ugyan olyan TypeScript kódot írhatunk, mint az applikáció fejlesztése során, viszont plusz függvényeket biztosít, melyek segítségével futás közben kérhetjük le az elemeket a DOMból és asserteket írhatunk. Az assertek olyan különleges if-ek, amikkel feltételeket tudunk felállítani, hogy mikor és milyen külső behatásra milyen válaszokat kell produkáljon az alkalmazás. Ha a tesztelésről beszélünk, van pár konvenció, amit, ha lehet be kell tartani. A legfontosabb ezek közül, hogy a tesztek nem szabad befojásolják a többi teszt eredményét és izoláltan kell fussanak. Az alábbi F.1.1 kódrészletben látható, az Admin felületen lévő időszakok létrehozásának, manipulációjának és törlésének a tesztje.

A **describe** függvényben tudunk definiálni egy tesztet. Az első paraméterének megadjuk a tesztnek a nevét, és a második paraméter egy függvényt vár el, itt megadhatunk arrow functiont is. Ebben a függvényben soroljuk fel a tesztnek a lépéseit, ezeket **it** függvények írásával tehetjük meg. Ezek az **it** függvények is két paraméterük, egy név és egy függvény, amiben megtalálható a teszt implementációja.

Ezek az **it** függvények sorról sorra fognak lefutni, olyan sorrendben, mint ahogyan deklarálva vannak. Annak érdekében, hogy minden teszt minél jobban izolált legyen, minden egyes teszt előtt bejelentkezzünk az applikációba és a teszt lefutása után kijelentkezünk. Ezeket a **beforeEach** és **afterEach** függvényekkel tehetjük meg. A **beforeEach** függvény minden **it** függvény előtt lefut és az **afterEach** pedig minden **it** függvény után lefut.

A **cy** globális objektumon keresztül elérhetjük az összes cypress által létrehozott függvényeket vagy újakat hozhatunk létre. Az alábbi kódban látható a custom login létrehozása.

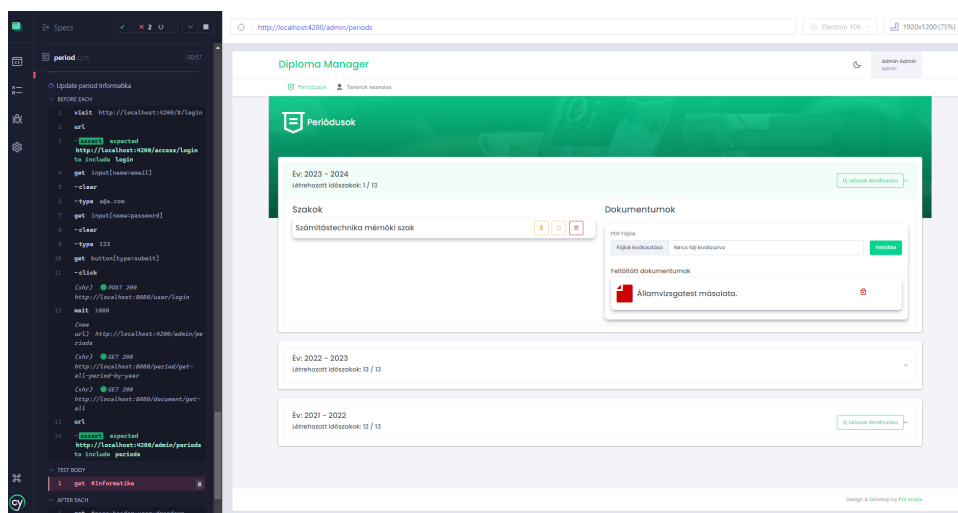
```
Cypress.Commands.add('login', (email: string, password: string) => {
  cy.visit('http://localhost:4200/#/login');
  cy.url().should('include', 'login');
  let emailInput = cy.get('input[name=email]');
  emailInput.clear();
  emailInput.type(email);
  let passwordInput = cy.get('input[name=password]');
  passwordInput.clear();
  passwordInput.type(password);
  cy.get('button[type=submit]').click();
});
```

A leggyakrabban használt command az a **cy.get()**, ezzel a felépült DOMból képesek vagyunk elemeket lekérni id, name vagy osztály szerint. Majd a lekért objektumokba írni tudunk vagy eventeket előidézni.

Minden tesztünk kell tartalmazzon legalább egy vagy több **assertion**. Az **assertion** olyan feltételek, amik eldöntik, hogy a tesztünk sikeresen futott le vagy nem. Ilyen az alábbi kódban, a **cy.should()** command, ezt használva tudjuk feltételezni, hogy, ha sikerült bejelentkezzünk, akkor az applikáció át kell navigáljon az admin belső oldalára és ekkor az URLben meg kell jelenjen a **periods**.

```
beforeEach('Login', () => {
  cy.login('a@a.com', '123').then(() => {
    cy.wait(1000);
    cy.url().should('include', 'periods');
  });
});
```

Az alábbi [4.3](#) képen látható a tesztelés lefutása.



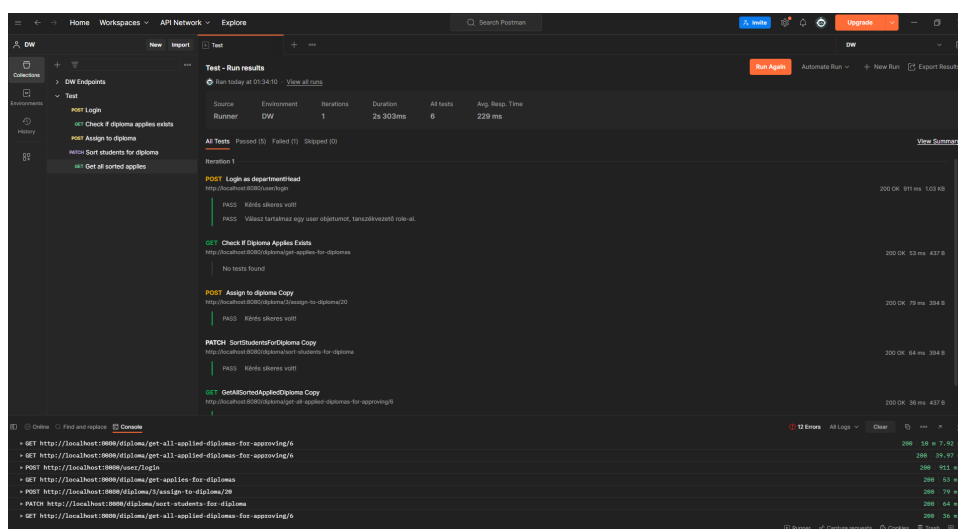
4.3. ábra. Cypress tesztelés

4.2.2. Backend tesztelés

A backend mivel mikroszerviszekre épül ezért API tesztelésekkel igazolni tudjuk a backend helyes működését. Erre a feladatra tökéletes a Postman. A postman az egy külső alkalmazás, amin keresztül HTTP kéréseket küldhetünk a backendnek. A postmanben kollektciókat tudunk létrehozni a HTTP kéréseinkről. Ezeknek a HTTP kéréshez teszteteket tudunk írni, ami jelzi minden futtatáskor, hogy az adott kérés olyan adatokkal tért-e vissza, amiket elvártunk.

Ezeket a kéréseket láncba tudjuk kötni, így akár egy teljes Controller-t vagy funkcióanalitást letudunk tesztelni.

Az alábbi képen 4.4 látható a Postman tesztelés.

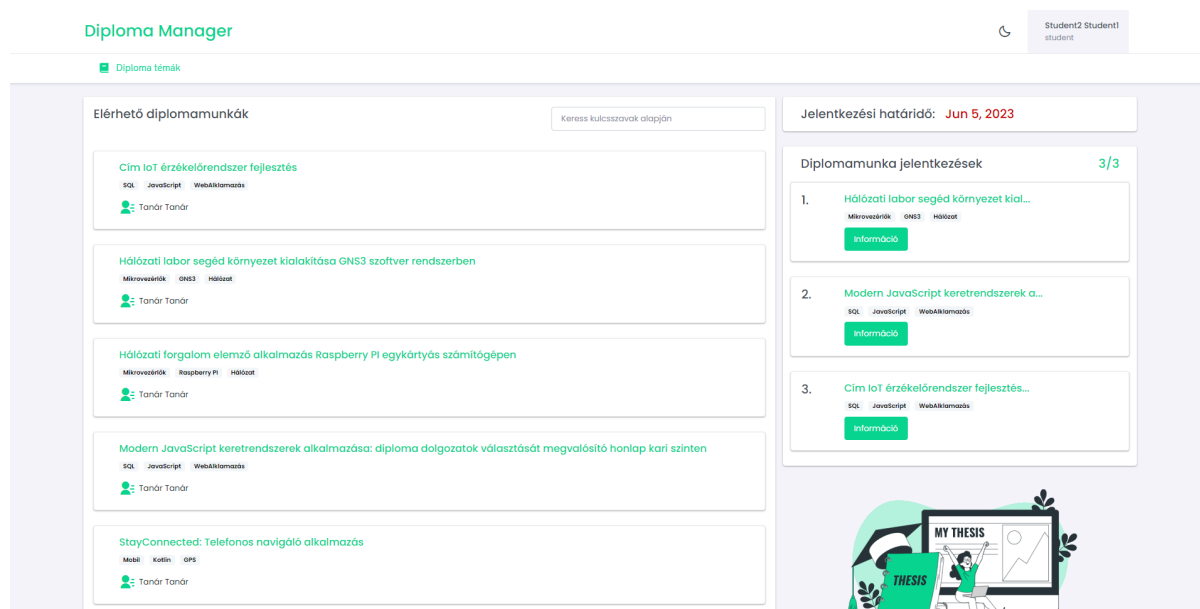


4.4. ábra. Postman tesztelés

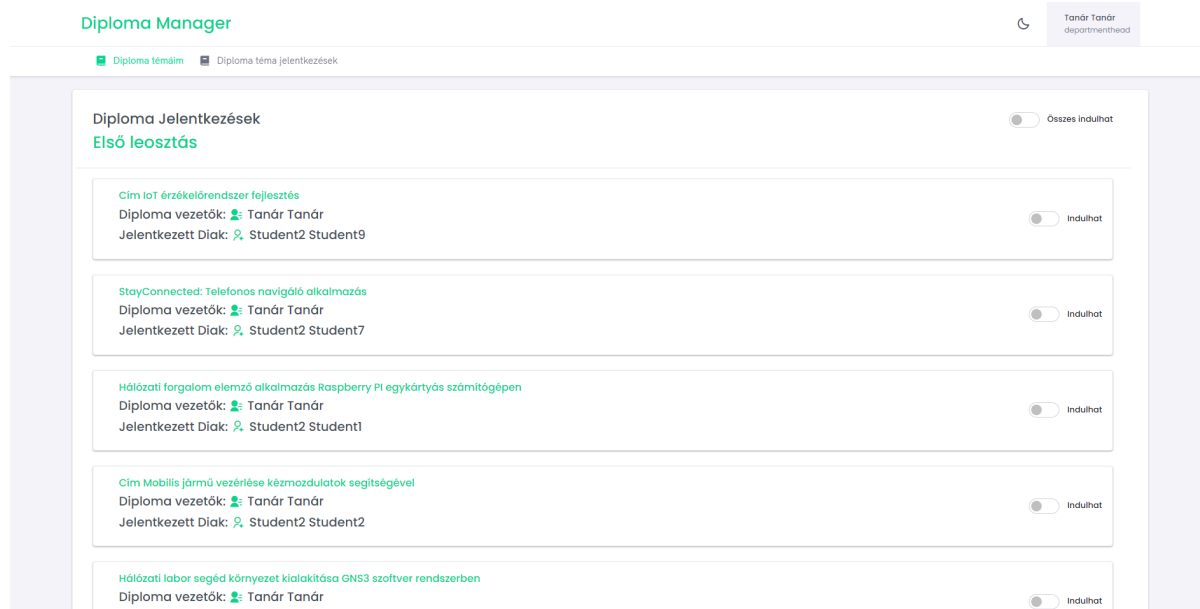
4.2.3. Éles tesztelés

Az applikáció valós adatok alapján is tesztelve volt. Az applikációba fel volt töltve több mint 10 diák, illetve 3 tanár és azoknak a tanároknak a meghirdetett diplomamunkái. Majd le volt tesztelve, hogy a diák látja-e ezeket, illetve tud-e rájuk jelentkezni.

Az alábbi képeken 4.5,4.6 látható a valós adatokkal való tesztelés.



4.5. ábra. Diploma téma jelentkezés hallgató szemszögéből



4.6. ábra. Diploma téma leosztás eredményei

5. fejezet

Üzembe helyezési lépések

5.1. Felmerült problémák és megoldásaik

5.1.1. Időszakok beintegrálása az applikációba

Az alkalmazásban a legnagyobb probléma, ami felderült, az az időszakokra épülő logika kiépítése. Mivel idővel egyre több időszak lesz a rendszerben, minden időszakhoz egy szak tartozik és egy szakhoz számtalan hallgató tartozik egy általános megoldásra volt szükség. A szerint, hogy éppen milyen dátumot írunk, az adott felhasználó más és más dolgokat képes végrehajtani az oldalon attól függően, hogy a szak, amibe tartozik, milyen határidőkkel van ellátva. Ezek a limitációk egyaránt kitérnek a tanár és tanszékvezetőkre is nem csak a hallgatókra.

Ezt a problémát csak backenden lehetett megoldani, mert így biztosítva van az, hogy minden egyes felhasználó ugyan az dátum szerint lesz manipulálva. Ezért a backend létre lett hozva egy utility függvény, amely bizonyos enumok megadásával, visszaadja, hogy az adott műveletet engedélyezett-e. Ezek az enumok a következők:

1. `START_OF_ENTERING_TOPICS.`
2. `END_OF_ENTERING_TOPICS.`
3. `FIRST_TOPIC_ADVERTISEMENTS.`
4. `FIRST_TOPIC_ADVERTISEMENT_END.`
5. `FIRST_ALLOCATION.`
6. `SECOND_TOPIC_ADVERTISEMENT.`
7. `SECOND_TOPIC_ADVERTISEMENT_END.`
8. `SECOND_ALLOCATION.`
9. `IMPLEMENTATION_OF_TOPICS.`
10. `DOCUMENT_UPLOAD.`
11. `DIPLOMA_DEFEND.`

```

@GetMapping("/check-preconditions/{enums}")
public ResponseEntity<?> checkPreconditions( @PathVariable PeriodEnums enums){
    try {
        return ResponseEntity.ok().body(utility.requestAccepted(enums));
    }catch (Exception e){
        return ResponseEntity.badRequest().body(e.getMessage());
    }
}

```

A fent lévő endpointra, HTTP kérést küldve, és a kérésbe paraméterként megadva valamelyik enumot, az visszatérít egy boolean értéket, aszerint, hogy a enum által értelmezett határidő lejárt-e már az összes szaknak.

Ezt az endpointot felhasználva tudjuk a felhasználót értesíteni a határidőkkel kapcsolatos eseményekről.

5.1.2. Hallgatók leosztása automatikusan

Annak érdekében, hogy a diákok leosztásáért ne kelljen egy plusz szerepkört beiktatni és hogy ezek a megfelelő időben legyenek kérvényezve, kellett egy automatikus megoldás, ami a meghatározott időpontban leosztja a hallgatókat.

Ebben segítségemre volt a Spring framework **@Scheduled** annotációja. Az annotáció elindítja az adott függvényt egy mellék szálon. Ennek az annotációnak megadhatunk egy cron értéket. A cron érték az egy rövidített kodolás, amivel ismétlődő időpontot adhatunk meg. Az alábbi cron érték **0 8 * * * *** az jelenti, hogy a folyamat ismétlődjön meg minden nap reggel nyolc órakor.

```

@Scheduled(cron = "0 8 * * *")
public void sortStudents(){
    System.out.println("Request for sorting students!");
    if(utility.requestAccepted(PeriodEnums.FIRST_ALLOCATION)){
        diplomaService.sortStudentsForDiploma();
    }
    if(utility.requestAccepted(PeriodEnums.SECOND_ALLOCATION)){
        diplomaService.sortStudentsForDiploma();
    }
}

```

A fenti kód minden reggel nyolc órakor ellenőrzi, hogy lejárt-e a leosztási határidő és ha igen, akkor leosztja a hallgatókat a médiájuk szerint a diplomamunkákra.

6. fejezet

Következtetések

6.1. Megvalósítások

Az applikáció fejlesztése során sikerült egy olyan adatbázist felépíteni, ami nem tartalmaz redundáns csatolásokat és értékeket, de nyitott a bővítésre. Ezt a horizontális skálázást elősegíti a Lquibase keretrendszer használata, mivel segítségével könnyen tudunk új táblákat definiálni vagy a meglévőket könnyedén módosítani.

Az adatbázishoz sikerült egy biztonságos és megbízható backendet létrehozni, ez köszönhető a Javának és a Spring keretrendszernek. És mivel az egész applikáció mikroszerviszekre van építve, nyitott a horizontális skálázásra, bármilyen plusz funkció beépítése végett nem szükséges semmilyen refaktorizálást csinálnunk az applikáció többi részében. Ebben az előnyben közrejátszik a rétegelt architektúra, ami azt eredményezi, hogy bármilyen nagyra is nő az alkalmazás, funkcionális és kódbázis szinten is mindig letisztult és könnyen kezelhető marad.

Frontenden, pedig sikerült egy reszponzív, gyors és imponáns kezelőfelületet megvalósítani. Sikerült egy működő login rendszert és regisztrációs rendszer megalkotni email küldéssel. Egy biztonságos oldalt, ahol az információk el vannak rejtve más felhasználók elől.

6.2. Továbbfejlesztési lehetőségek

Továbbfejlesztési lehetőségnek be lehetne implementálni értesítéseket minden üzenetre hallgatónak és a tanárok számára egyaránt. Lehetséges lenne a PDF megtekintőt beimplementálni, hogy a dokumentációkat el lehessen olvasni az oldalon. Sokat dobna az alkalmazáson, ha minden felhasználónak email-es értesítést küldene a rendszer bizonyos határidők lejárása előtt.

A felhasznált technológiák miatt a fejlesztési lehetőség határtalan.

Irodalomjegyzék

- [1] Angular (web framework). [https://en.wikipedia.org/wiki/Angular_\(web_framework\)](https://en.wikipedia.org/wiki/Angular_(web_framework)).
- [2] Budapesti műszaki és gazdaságtudományi egyetem (bme) diplomaterv adatbázis. <https://diplomaterv.vik.bme.hu/hu/>.
- [3] Használt ikonok. <https://www.pngwing.com/>.
- [4] Hibernate tutorial. <https://www.javatpoint.com/hibernate-tutorial>.
- [5] Introduction to microservices: What are microservices? use cases and examples. <https://www.datarobot.com/blog/introduction-to-microservices/>.
- [6] Json web token. <https://jwt.io/>.
- [7] Szegedi tudományegyetem. <https://www.inf.u-szeged.hu/bir2/Thesis/Browse>.
- [8] What is a single page application? meaning, pitfalls benefits. <https://www.excellentwebworld.com/what-is-a-single-page-application/>.
- [9] W. Dave. *API Testing and Development with Postman: A practical guide to creating, testing, and managing APIs for automated software testing*. Packt Pub, 2021.
- [10] F. B. S. Emma Thorén. Usage of angular from developer's perspective. Master's thesis, Faculty of Computing at Blekinge Institute of Technology, 2017.
- [11] Y. Fain and A. Moiseev. *Angular development with typescript*. Manning, 2019.
- [12] D. Gourley and B. Totty. *HTTP the definitive guide; Understanding web internals*. O'Reilly, 2002.
- [13] B. B. Kathy Sierra. *Head First Java, 2nd Edition*. O'Reilly Media, 2005.
- [14] A. Margit. *Java alapu webtechnologiak*. Scientia, 2009.
- [15] R. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C. Martin Series)*. Pearson, 2017.
- [16] W. MWAURA. *End-to-End Web Testing with Cypress: Explore techniques for automated frontend web testing with Cypress and JavaScript*. Packt Pub, 2021.
- [17] O. R. O. *PostgreSQL: Up and running*. O'Reilly, 2015.

- [18] J. Palmer, C. Cohn, M. Giambalvo, and C. Nishina. *Testing angular applications*. Manning Publications, 2018.
- [19] T. Point. Spring boot tutorial. https://www.tutorialspoint.com/spring_boot/spring_boot_tutorial.pdf, 2018.
- [20] S. Purewal. *Learning Web App Development: Build Quickly with Proven JavaScript Techniques*. O’Reilly Media, 2014.
- [21] D. D. B. V. SEIJI. *Full-stack web development with Spring Boot and angular: A practical guide to building your full-stack web application*. PACKT PUBLISHING LIMITED, 2022.
- [22] J. Wilken. *Single Page Web Applications: Javascript end-to-end*. Manning Publication, 2014.
- [23] J. Wilken. *Angular in action*. Manning Publication, 2018.

Függelék

F.1. Kód részletek

F.1.1. kódrészlet. Cypress teszt

```
import '../support/commands';
import { slowCypressDown } from 'cypress-slow-down';

slowCypressDown(150);

describe('Creating new period', () => {
  beforeEach('Login', () => {
    cy.login('a@a.com', '123').then(() => {
      cy.wait(1000);
      cy.url().should('include', 'periods');
    });
  });

  afterEach('Logout', () => {
    cy.logout().then(() => {
      cy.wait(1000);
      cy.url().should('include', 'login');
    });
  });

  it('Deleting Period for Informatika', () => {
    const selectedCard = cy.get('#Informatika');
    if (selectedCard === null) {
      return;
    }
    selectedCard.find('.btn-outline-danger').click();
    cy.wait(500);
    const deleteButton = cy.get('button').contains('Yes, delete it!');
    deleteButton.click();
    cy.get('.toast-success').should('be.visible');
  });

  it('Create Period for Informatika', () => {
    const currentDate = new Date().toLocaleDateString();
    const ngPanle = cy.get('[aria-expanded="true"]');
```



```

ngPanle
  .get('button')
  .contains('j idszak ltrehozsa ')
  .scrollIntoView()
  .click();
cy.url().should('include', 'create-period');
cy.get('[formcontrolName="major"]').select('Informatika');
cy.get('[formcontrolName="startOfEnteringTopics"]').type(
  currentDate.toString()
);

cy.get('[formcontrolName="endOfEnteringTopics"]').type(
  currentDate.toString()
);

cy.get('[formcontrolName="firstTopicAdvertisement"]').type(
  currentDate.toString()
);

cy.get('[formcontrolName="firstTopicAdvertisementEnd"]').type(
  currentDate.toString()
);

cy.get('[formcontrolName="firstAllocation"]').type(
  currentDate.toString()
);

cy.get('[formcontrolName="secondTopicAdvertisement"]').type(
  currentDate.toString()
);

cy.get('[formcontrolName="secondTopicAdvertisementEnd"]').type(
  currentDate.toString()
);

cy.get('[formcontrolName="secondAllocation"]').type(
  currentDate.toString()
);

cy.get('[formcontrolName="implementationOfTopics"]').type(
  currentDate.toString()
);

cy.get('[formcontrolName="documentumUpload"]').type(
  currentDate.toString()
);

cy.get('[formcontrolName="diplomaDefend"]').type(
  currentDate.toString()
);

cy.get('button').contains('Idszak ltrehozsa ').click();

```

```

    cy.wait(1500);

    cy.url().should('include', 'periods');

    cy.get('.toast-success').should('be.visible');
  });

  it('Update period Informatika', () => {
    let currentDate = new Date();
    currentDate.setMonth(currentDate.getMonth() + 1);
    const selectedCard = cy.get('#Informatika');
    selectedCard.find('.btn-outline-warning').click();
    cy.url().should('include', 'period');
    const inputField = cy.get('[formcontrolName="diplomaDefend"]');
    inputField.clear();
    inputField.type(currentDate.toLocaleDateString());
    cy.get('button').contains('Idszak frisstse!').click();
    cy.url().should('include', 'periods');
    cy.get('.toast-success').should('be.visible');
  });
});
}

```

UNIVERSITATEA SAPIENTIA DIN CLUJ-NAPOCA
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE,
TÎRGU-MUREȘ
SPECIALIZAREA CALCULATOARE

Vizat decan:
Conf. dr. ing. Domokos József

Vizat director departament:
Ș.l. dr. ing. Szabó László Zsolt