Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Rideg Vilmos Dániel, Göndöcs Martin

KÖZÖSSÉGI ALKALMAZÁS FEJLESZTÉSE SPRING BOOT PLATFORMON

KONZULENS

Imre Gábor

BUDAPEST, 2024

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Rideg Vilmos Dániel és Göndöcs Martin**, szigorló hallgatók kijelentjük, hogy ezt a dokumentáció meg nem engedett segítség nélkül, magunk készítettük, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtuk fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettünk, egyértelműen, a forrás megadásával megjelöltünk.

Hozzájárulunk, hogy a jelen munkánk alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentjük, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2024. 05. 25.

...…….
Rideg Vilmos Dániel

...…….
Göndöcs Martin

Összefoglaló

Az internet megjelenése óta egyre több weboldal születik meg a világhálón. Ezek funkciója nagyon sokrétű, manapság a személyi jövedelemadó bevallástól kezdve a barátokkal vagy rokonokkal való kapcsolattartás is itt folyik akár csevegő alkalmazásokban, akár a közösségi felületeken. Utóbbiakból egyik legnépszerűbb még mindig a Facebook, de az Instagram és a TikTok is kezd felzárkózni.

Az Önálló laboratórium 1. tárgy keretein belül egy ilyen webes alkalmazás tervezése és megvalósítása volt a célunk. Az olvasót végigvezetjük egészen a kezdeti tervezési fázistól, azaz az adatbázis séma létrehozásától egészen az összetettebb funkciók megírásának és a frontend, illetve a backend összehangolásáig a lehető legtöbb aspektuson, ami egy ilyen szoftver keletkezése során felmerül. Szó lesz tervezési döntésekről, az alkalmazás írásakor felmerülő bonyodalmakról, azok megoldásáról.

A feladat végén érezhető lesz, hogy egy ilyen webes felület megírása nem egyszerű feladat. A kezdeti tervezői döntések nem mindig bizonyulnak helyesnek vagy teljesnek, sok újratervezés van a háttérben. Ez a dokumentum ezen folyamatok megértését is szolgálja.

1 Bevezetés

Egy webes alkalmazáshoz jellemzően 2 elem szükséges, egy frontend és egy backend adatbázissal. Az adatbázis szolgálja ki a backendet, ami a frontednek kiadott utasítások szerint lekérdezi a szükséges adatokat, majd azokat rendezett formában továbbítja a frontend felé. A frontend pedig összeilleszti a backendtől kapott adatokat, és a felhasználó elé tárja.

1.1 Spring Boot

A Spring Boot egy Java alapú keretrendszer, mellyel kisebb méretű webes alkalmazások készíthetőek. Backend környezeten használjuk. Egyszerű, minimális konfigurációt igénylő működést biztosít. Sok beépített függvénynek köszönhetően a szolgáltatások fejlesztési idejét és a hibalehetőségek számát csökkenti.

1.1.1 Annotációk

A programozási nyelvek egyik eleme. Ezek olyan extra információkat szolgáltatnak akár osztályokról, változókról vagy metódusokról, amelyeket a fordító fordítási időben hozzát tud tenni a lefordított bytecode-hoz. Ilyen például az alapértelmezett, paraméter nélküli vagy az összes paramétert tartalmazó konstruktor készítése anélkül, hogy azt a programozó explicit leírná. Alább összegyűjtöttük a legfontosabb Spring Boot annotációkat.

1.1.1.1 @OneToMany

Adatbázis táblák közötti kapcsolattípust reprezentál. Ez jellemzően kollekciókon jelenik meg annotációként. Tegyük fel például, hogy létezik egy **Felhasználó** és **Poszt** osztály. A Felhasználó osztályban létezik egy Poszt kollekció, illetve a Poszt osztályban egy Felhasználó változó. Ekkor @OneToMany-t teszünk a Poszt kollekcióra. Ekkor ennek jelentése, hogy egy Felhasználóhoz egy vagy több Poszt is rendelhető.

1.1.1.2 @ManyToOne

Egy másik kapcsolattípust reprezentál. Ez jellemzően nem kollekciókon, hanem változókon jelenik meg. Általánosságban a @OneToMany-vel használjuk párban. Vegyük az előző alfejezetet példának. @ManyToOne annotációt alkalmazunk a Poszt

osztály Felhasználó adattagján. Ekkor ennek jelentése, hogy egy Poszthoz kizárólag egy Felhasználó rendelhető.

1.1.1.3 @ManyToMany és @JoinTable

A @**ManyToMany** egy újabb kapcsolattípust reprezentál. Tegyük fel, hogy létezik egy **Felhasználó** és egy **Csoport** osztály. Mindkét osztályban létezik egy-egy kollekció a másik osztály típusával. Ekkor az annotáció jelentése – bármelyik kollekcióra is alkalmazzuk az lesz – , hogy egy Felhasználóhoz akár több Csoport is rendelhető, illetve egy Csoporthoz akár több Felhasználó is rendelhető.

A @**JoinTable** annotáció segítségével pedig leképezhetjük ennek a több-több kapcsolatnak az adatbázisban megfelelő tábláját. Vegyük példának az előző két osztályt, és alkalmazuk az annotációt a Csoport osztály Felhasználó kollekcióján. Ekkor az annotáció *name* attribútumával megadhatjuk ennek a táblának a tetszőleges nevét. A *joinColumns* segítségével adjuk meg az egyik – jellemzően az annotációt tartalmazó – osztály elsődleges kulcsát idegenkulcsként. Illetve az *inverseJoinColumns* segítségével adjuk meg a másik osztály elsődleges kulcsát idegenkulcsként. A **Felhasználó** osztály **Csoport** kollekcióján használt @**ManyToMany** annotációt pedig a *mappedBy* attribútummal bővítjuk, mellyel megadható, hogy ezen adatok melyik másik adattaggal kerülnek „fésülésre” az adatbázisba. Ekkor pedig a Csoport osztály Felhasználó kollekciójának nevét érdemes megadni. A fenti esetre konkrét példa látható alább.

```
@ManyToMany(
        fetch = FetchType.LAZY,
        mappedBy = "participantUsers")
private List<GroupEntity> joinedGroups = List.of();
```

AppUserEntity osztály GroupEntity kollekciója

```
@ManyToMany
@JoinTable(
        name = "group_participant",
        joinColumns = @JoinColumn(name = "group_id"),
        inverseJoinColumns = @JoinColumn(name = "participant_user_id"))
private List<AppUserEntity> participantUsers;
```

GroupEntity osztály AppUserEntity kollekciója

1.1.1.4 Egyirányú kapcsolatok

Egyirányú kapcsolatok során kódban adott két entitás során csak az egyik oldalon szeretnénk tárolni a másik oldalra mutató referenciát, ezért csak ezen az oldalon vesszük fel az osztály adattagját, ami lehet változó vagy kollekció is.

Rendben, de adódik a kérdés, hogy ez adatbázis szintjén mégis hogyan működik? Hogyan lehetséges, hogy ezt a referenciát az adatbázis is érteni fogja attól független, hogy hol vettük fel a kollekciót vagy változót.

A válasz a @**JoinColumn** annotáció, amelyet használhatunk párban a @**OneToMany** vagy a @**ManyToOne** annotációval is adott változón vagy kollekción, ugyanakkor működésük eltérő. Tételezzük fel, hogy először az előbbi, majd az utóbbi annotációt alkalmazzuk a **Felhasználó** osztály **Cím** adattagján mindkét esetben. Az első esetben ekkor adatbázis szinten a Cím táblában létrejön majd egy Felhasználó idegenkulcs. Második esetben azonban pontosan fordítva, azaz a Felhasználó táblában jön létre majd egy Cím idegenkulcs. Ezért vigyázni kell, hogy hol adjuk ki a @**JoinColumn** annotációt.

1.1.1.5 Kétirányú kapcsolatok

Kétirányú kapcsolatok során kódban adott két entitás során mindkét oldalon szeretnénk egy-egy referenciát a másik oldalra. Ekkor mindkét osztályban felvesszük a másik osztályra mutató adattagot, ami lehet kollekció vagy változó is.

Rendben, de adódik újból a kérdés, hogy ez adatbázis szintjén mégis hogyan működik? Hogyan lehetséges, hogy kétirányú kapcsolat során a kollekciókra mutató referenciát képes lesz kezelni?

Korábban látható volt a @**ManyToMany** és a @**JoinTable** kombinációja, amely pontosan egy kétirányú kapcsolatot valósít meg egy új tábla létrehozásaként két osztály kollekciója között. Tehát két kollekció között ezen két annotációval elérhető egy kétirányú kapcsolat.

De, mi a helyzet egy adattag és egy kollekció esetén? A válasz pedig a @**OneToMany** annotáció ***mappedBy*** attribútumában rejlik. Tételezzük fel, hogy az annotációt alkalmazzuk a **Felhasználó** osztály **Cím** adattagján (kollekcióján). Ekkor a *mappedBy* értékének a Cím osztály Felhasználó adattagjának (változójának) a nevét kell írni.

1.1.2 Controller

A Controllerek a backend egyik alkotóelemei. Ezeknek feladatai, hogy a klienstől beérkező HTTP kéréseket kezeljék, és megfelelő választ adjanak.

1.1.3 Service

A Service a backend egyik alkotóeleme. Feladata, hogy a Controller által kapott kérésnek megfelelően elvégezzen valamilyen üzleti logikát és visszaszolgáltassa a Controllernek a művelet eredményét.

1.1.4 Repository

A Repository a backend egyik alkotóeleme. Feladat, hogy fenntartsa a backend és az adatbázis közti kommunikációt. Az egyes Service-k a Repository-n keresztül kérik le a szükséges adatbázisbeli információkat.

1.2 Spring Security

Az autentikációt végzi a backenden. Egy egyedi tokent generál a bejelentkezett felhasználónak, mely végigkíséri a weboldalon töltött idejét, ezzel biztosítva, hogy azt illektéktelenek nem használják.

1.3 Scrum

A Scrum egy projektmenedzselési módszertan, amely agilis megközelítést használ. Heti lebontásban, azaz sprintekbe szedve állítjuk össze a projekt feladatait, minden feladatot kisebb részegységekre bontva.

1.4 Lombok

A Lombok egy olyan könyvtár, ami az úgynevezett boilerplate kódok mennyiségének minimalizálására szolgál. A boilerplate kód az olyan kódrészleteket jelöli, amely különösebb üzleti logikával nem rendelkezik egy adott osztályban, viszont a megfelelő alapműködés biztosításához elengedhetetlen. Ilyenek például a getterek, setterek, konstruktorok stb.

1.4.1 Annotációk

A következőkben bemutatjuk a Lombok legfontosabb annotációit.

1.4.1.1 @AllArgConstructor, @RequiredArgsConstructor, @NoArgsConstructor

Mindhárom annotáció a nevéből adódóan mennyiségű és típusú paraméter alapján képes létrehozni paraméteres vagy paraméter nélküli konstruktorokat. Ezeket az osztály neve előtt alkalmazzuk.

Jelentésük rendre, az osztály összes adattagját tartalmazó konstruktor, csak a final adattagokat tartalmazó konstruktor, végül pedig paraméter nélküli konstruktorok.

1.4.1.2 @Getter, @Setter

Ezek a privát adattagok lekérdezését és beállítását szolgálják. Szintén az osztály neve előtt alkalmazzuk.

1.4.1.3 @ToString

Segítségével felülírható az osztály ToString metódusa.

Alkalmazható osztályon, ekkor például az *onlyExplicitlyIncluded* attribútumnak adott *True* érték esetén csak a @**ToString.Include** annotációval jelölt adattag kerülnek kiíratásra. Illetve például az *includedFieldNames* attribútumnak adott *False* értékkel eltávolíthatók a kiíratásból az adattagok nevei.

1.5 Postman

A Postman segítségével a backenden történő fejlesztés tesztelhető frontend nélkül is. Ezzel párhuzamosan fejleszthetővé és tesztelhető válik a frontend és backend.

1.6 Angular

Az Angular egy weblapfejlesztéshez használt TypeScript alapú keretrendszer. Frontend környezetben használjuk. Feladata az alkalmazás grafikus megjelenítése a felhasználó felé, illetve a felhasználó kéréseinek a továbbítása a frontendről a backendre.

1.6.1 Felépítése

Az Angular frontenden biztosítja a nézeteket (template-eket), a nézetek mögötti kódrészleteket (component-eket), illetve a kommunikációt a backenddel (service-eket).

1.6.1.1 Template

A Template-k, magyarul sablonok, azok a nézetek .html kiterjesztésben, amelyek a felhasználó szabad szemmel képes követni, ahol megjelenítjük a számára szükséges adatokat, műveleteket.

Az adatokkal manipulációt intézhet a frontendről a backendre. Ehhez adatkötést használunk. Az Angular kétirányú adatkötést is képes biztosítani, az egyirányú mellett. A .html fájlokban így kettős kapcsos zárójellel – {{something}} – egyirányú és egyszeri

szögletessel – [something] – egyszerű adatkötés a modell felé típusú adatkötést teszünk lehetővé. Továbbá egyszerű zárójelekkel – (event) – eseményre való figyelést valósíthatunk meg. Végül pedig a „banán egy dobozban” zárójelekkel – [()] – kétirányú adatkötést valósíthatunk meg.

1.6.1.2 Component

A Component, magyarul a komponensek valósítják meg az egyes nézetek mögötti kódot. Ezeket hívjuk code-behind-nak.

Minden a nézeten elvégzett művelet először ide érkezik be, itt tároljuk az adatok ideiglenes állapotait, illetve innen hívunk át az egyes Service-kbe.

Amikor Angularban létrehozunk egy komponenst, akkor az igazából három darab fájlt létrehozását jelenti pontosan. Egy ilyen komponens tartalmazni fogja a code-behind-ot .ts, a nézetet .html, és a stíluslapot .css kiterjesztésben.

1.6.1.3 Service

A Service, magyarul szolgáltatás tartalmazza az egyes http (hyper-text transfer protocol) metódusok deklarációit, amelyekkel kérést intézhetünk a frontendről a backendre. Válasz esetén pedig ezen **Observable** visszatérési értékkel rendelkező metódusok *subscribe()* függvényével várjuk be aszinkron az adatokat.

1.6.1.4 Lazy (module) loading

A Lazy loading, vagyis lusta betöltést Angular esetén az egyes Modulok betöltését szabályozza. Ezzel a módszerrel az egy modulok csak akkor kerülnek betöltésre, amennyiben a felhasználónak szüksége van rá.

Module-nak (modulnak) nevezzünk logikailag összetartozó egy vagy több komponenst (nézetet, code-behind-t, stíluslapot tartalmazó) és az ezekhez szükséges routing-t tartalmazó nagyobb egységet.

1.6.2 @Input

Az @Input annotáció segítségével DI (dependency injection) mintájára inicializálhatjuk egy-egy komponens elemét.

1.6.3 Routing

Az Angular és a weboldalkezelés egyik fontos alapillére a routing, vagyis az egyes nézetek közötti út megtervezése. Alkalmazásunkban minden modul rendelkezik külön beállítható routing-gal a lazy moduling miatt, melyet az app-routing.module.ts fájl fog össze.

1.7 Jira és a GitHub

A Jira egy munkafolyamat nyomonkövetési rendszer. A feladatokat csoportosítani lehet, időhatárokat hozhatunk benne létre és figyelhetjük, hogy ki hogyan halad az adott munkával.

A GitHub a Git weboldalas megoldása, ahol a projekteken lehet verziókövetést végezni. Itt a fő (többnyire master vagy main) ágon (branch) van a közös, elfogadott verziója a programnak, a mellékágakon pedig az új feature-ök vagy hibajavításokat tartalmazó kód van.

A továbbiakban a fejlesztői döntéseket és megoldásokat részletezzük. Ez két fejezetre bomlik, mindegyikben az adott feladatot elvégző személy fogja leírni a döntéseinek okát, lényegét.

Az első alfejezetben az adatbázis létrehozásáról és konfigurálásáról lehet bővebben olvasni, majd kitérünk az entitások felvételére, azok sémákba rendezésére, a Spring Security, azaz az autentikáció működéséről és felépítéséről, az input validációra, azaz hogyan lehet ellenőrizni, hogy a felhasználó helyes adatokat adott-e meg, illetve betekintést adunk a frontend feladatokkal kapcsolatos kisebb-nagyobb tervezési döntésekről, egyes megoldásokról.

A második alfejezetben az Angular projekt élesítéséről lehet olvasni, milyen modulokból és elemekből épül fel, ezután kitérünk még a *lazy modulingra,* hogyan működik, mi a felépítése, szót ejtünk a routingról, a Jira projekt létrehozásáról, bekötéséről a Githubba, illetve betekintést adunk a backend feladatokkal kapcsolatos fejlesztési döntésekről.

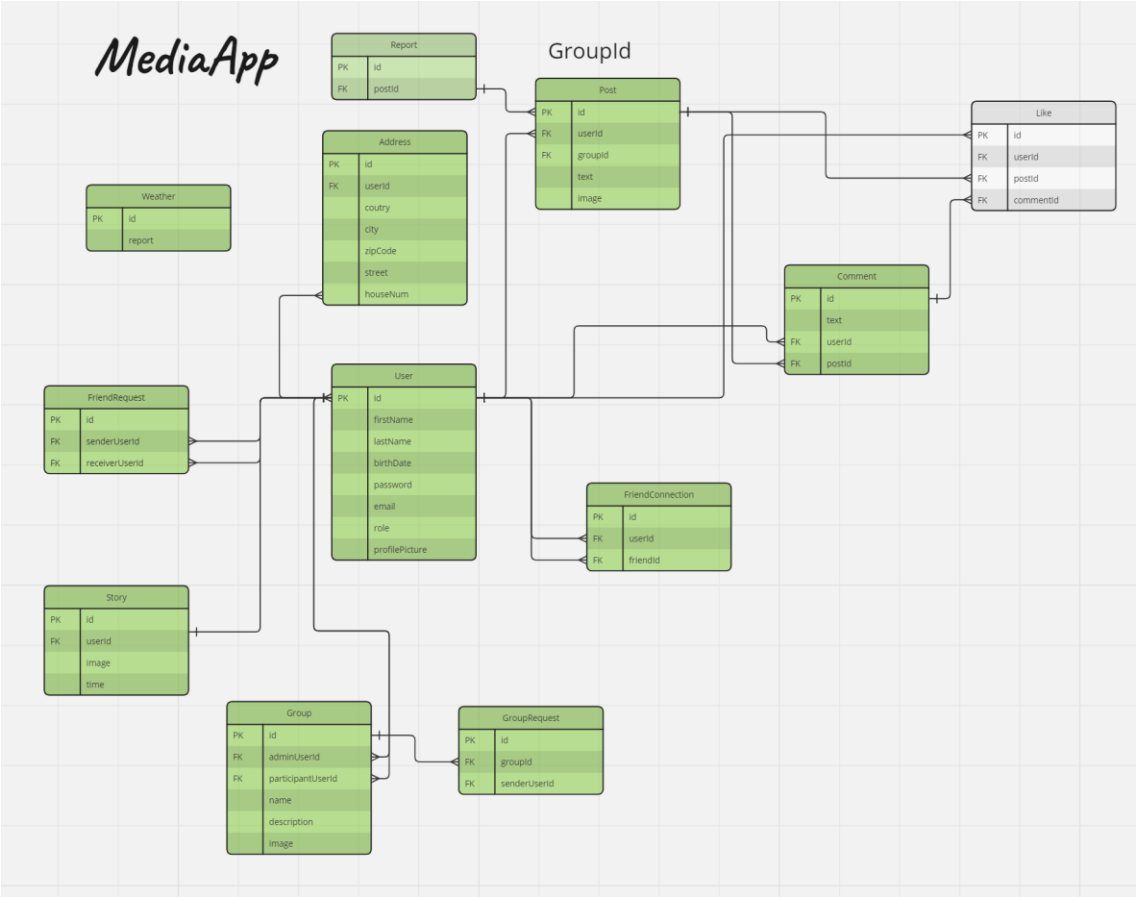
2 Önálló munka bemutatása (Vilmos)

2.1 Adatbázis létrehozása

Adatbázis megoldásként a MySQL adatbáziskezelőjét választottuk. Ez egy elég népszerű adatbáziskezelő szoftver, tehát sok forrás a rendelkezésünkre állt mind setupot, mind használatot illetően. Ennek használatához az IntelliJ IDEA Ultimate verziója nyújt kézenfekvő, egyszerű kapcsolódási felületet, így az adatbázis URL-jét könnyen be tudtam állítani a programban. Ahhoz, hogy a backend teljesen használni tudja, még egy *application.yml* fájl létrehozása is szükséges. Ebben meg kell adni az adatbázishoz kapcsolódás alap adatait, például URL, felhasználónév és jelszó, valamint egy drivert, amin keresztül végre tudja hajtani a backend által küldött kéréseket. Itt lehet még megadni atz is, hogy a Spring alkalmazás indításakor az adatbázist, megtartsa, frissítse az entitások alapján, vagy mindig újat hozzon létre.

2.2 Entitások

Következő lépésként az entitások tervezése és létrehozása volt a feladatunk.



1. ábra: ER diagramm

A fenti képen lehet látni az ER diagramunkat. Itt látható, hogy egy *User* entitás van az applikáció középpontjában, ez a bejelentkezett felhasználót reprezentálja. A *Post* entitás reprezentálja a felhasználó által tárolt posztokat, és a felhasználóban eltároljuk őket. A Postokon belül a *Comment* entitásokat tárolunk, amik a poszt alá írt kommenteket jelölik. A *Like* entitások pedig a poszt és kommentek kedveléseit repreneztálják. A *FriendRequest*-ben tároljuk el két felhasználó csatlakozási szándékát, ami elfogadás esetén egy *FriendConnection* típusú entitás lesz. A *Group* entitás reprezentálja a csoportokat, amikbe a felhasználók csatlakozhatnak. A *Report* entitás a jelentett kommenteket jelöli meg. A *Story* entitásban a felhasználók által feltöltött időkorlátosan elérhető képeket reprezentáljuk. A *Weather* entitásban egy API hívással lekért időjárás információkat tudjuk eltárolni az adatbázisban.

A legnagyobb nehézség ebben, ami adódott, hogy sok esetben az „A B-hez tartozik” típusú kérdésekre választ kapunk az adatbázis kapcsolatok révén, azonban a „B melyik A-khoz tartozik” kérdésekre nem ilyen egyszerű válaszolni. Itt két választási lehet őségünk volt: Vagy külön lekérdezéseket kell írni, amiben összeillesztjük a megfelelő

elemeket a párjaikkal vagy kétirányú kapcsolatokat alkalmazunk. Ez utóbbi sok esetben gondot okozhat, a Lombok könyvtár által használt ToString() és entitások DTO-ba mapelésnél végtelen ciklust okozhat a körkörös referencia. Ezért minimalizáltuk az ilyen utóbbi eseteket, és saját mappelési metódusokat vezettünk be.

2.3 Spring Security

A Spring Security célja, hogy autentikálja az éppen belépő felhasználót. Ennek több lépcsőfoka van. Elsőként ellenőrzi, hogy a megadott felhasználónév-jelszó kombináció megfelelő-e. Helytelen név vagy jelszó esetén egy hibát dob, amit a weboldalon felhasználunk egy hibaablak megjelenítéséért. Amennyiben helyesek a megadott adatok, úgy elkezdődik a token generálása. Itt először egy titkos kulcsot kell definiálni, amit aztán az aláíró kulcs generálásához használ a keretrendszer. A token létrehozása közben megadhatók extra adatok *extraClaims* néven, de ezeket nem használtam, így nem is térek ki rá részletesebben. A folyamatban szükségesek a fentebb említett extra claimek, felhasználónévre, a token készítésének időpontjára. Fontos adat még az is, hogy ez a token meddig legyen használható, ezt is itt kell megadni. A token aláírásának folyamata is itt történik a titkos kulcsból létrehozott kulccsal. Ezzel végezve ezt a tokent visszaadjuk a felhasználónak, hogy ezzel azonosíthassa magát.

Amikor egy weboldalra belép egy felhasználó, a backend eldönti, hogy szükséges-e autentikáció az adott oldalra. Jellemzően az olyan oldalakon, amiken még nem lehet autentikálva egy leendő felhasználó (ilyen például a regisztráció és a belépés oldalai) nem kell tokeneket kérni a felhasználótól. Azonban egy létező felhasználóhoz kötött rendszeren belül már fontos, hogy a lekért adatokat egy létező személyhez köthessük. Ilyenkor a belépés után megkapott tokent minden backend kérés mellé headerben, azaz fejlécben társítani kell. Ilyenkor a backend ellenőrzi, hogy a token nem járt-e le és a kért token valós-e. Amennyiben helyes, úgy a felhasználó megtekintheti az adott oldalt, mivel felhatalmazott rá és van létező felhasználó, akihez léteznek hozzárendelve adatok az adatbázisban.

2.4 Input validáció

Fontos dolog felkészíteni a szervert a hiányos vagy rossz adatokra, azonban ezeket érdemes minél hamarabb megfogni, még a frontenden. Ezt input validációval lehet a legkézenfekvőbben megoldani, aminek segítségével még hasznos tanácsokkal is

elláthatjuk a belépni vagy regisztrálni vágyó felhasználót. Az Angular keretrendszernek köszönhetően az input tageken belül lehetőségünk van komplexebb kód használatára is, azaz függvényhívásokat is intézetünk. Ez a HTML kód jobb átláthatóságát is nagyban segíti, ráadásul esetünkben a TypeScript tudását is kihasználhatjuk. Én az email címnek egy reguláris kifejezést adtam meg ellenőrzendő feltételnek. Amennyiben a bemenet illeszkedik a megadott reguláris kifejezésre, akkor elfogadjuk a felhasználó által beírta adatot, ellenben nem, és megkérjük, hogy érvényes adatokat adjon meg.

2.5 További frontendes feladatok, bonyodalmak

Alapvető megközelítés, hogy a frontenden a felelősségek szétváljanak. A http kéréseket egy külön fájlban kezeljük, minden oldalnak legyen meg a saját modulja, komponense.

Sok gondot okozott a címek megfelelő kezelése a szerkesztésük közben. Eleinte úgy szerettem volna megoldani a szerkesztést, hogy minden címet egyszerre szerkeszthetővé teszek. Ez azonban az Angular ngModel mechanikája, azaz a kettős adatkötés miatt nehézségekbe ütközött. A kettős adatkötés azt jelenti, hogy a UI-on megváltozó elemek az adatszerkezetre is hatással vannak és fordítva, ha az adatszerkezet megváltozik, akkor a UI elemek is frissülnek az új információval. Egy ciklussal íratom ki az adott felhasználó címeit, ami mindig egy input mezőbe teszek. Ez azonban a fentebb említett kettős adatkötés miatt valószínűleg ugyanarra a bementre kötötte le az adott változót, így amikor betöltötte az oldalt a böngésző, mindegyik cím helyén az utolsóé szerepelt a formban. Ezt úgy küszöböltem ki, hogy először is kikötöttem, hogy egyszerre csak egy cím szerkeszthető. Mindegyik címnek készítettem egy szöveges reprezentációját, amit a „szerkesztetlen” állapotban használok. Szerkesztés közben az egyik címet teszem csak szerkeszthetővé, és az egy cím változót tudom már szerkeszteni.

3 Önálló munka bemutatása (Martin)

Az alábbiakban bemutatom a legfontosabb fejlesztői döntéseket.

3.1 Angular projekt

A teljes frontendet tartalmazó különálló Angular alkalmazás elkészítéséhez Node.js, Visual Studio Code környezetet és a legfrissebb Angular keretrendszer verziót töltöttük le.

Az Angular saját parancssoros utasításokkal rendelkezik. Így például ***ng new angular-media-app*** paranccsal generálható parancssorból egy teljesen új webes alkalmazás. Ezt pedig szintén parancssorból a ***ng serve*** paranccsal indítható el. Ezt követően egy böngészőből (alapértelmezetten) a *localhost:4200* címen és porton érthető el az alkalmazás.

Ezt követően a backend-nek megfelelő modell osztályokat generáltunk le és egészítettünk ki a megfelelő adattagokkal. Elkészítettük az backend-re történő hívásokat tartalmazó szolgáltatásokat. Végül minden egyes nagyobb felületnek külön-külön modult generáltunk egyedi routing-gal, amelyet be is kötöttünk az alkalmazás főmoduljának a routing-jába. Amennyiben egy modulhoz szerettünk volna még komponenst hozzáadni, azokat szintén parancssorból generáltuk le. Az alábbi táblázat egy jó összefoglaló a kiadott parancsokra és a legenerált fájlokra.

(Kacsacsőrök elhagyásával kell megadni adott elem nevét.)

Parancs	***Generált elem***
ng generate class <Name>	Osztály (.ts fájl)
ng generate component <Name>	Komponens (.html, .css, .ts)
ng generate module <Name> --route route/path -- module app.module	Modul egyedi route/path routinggal
ng generate service <Name>	Szolgáltatás

Parancsok táblázata

3.2 AuthGuard

Az AuthGuard (Authentication Guarding) feladata, hogy a felhasználó csak olyan útvonalakat érjen el, olyan nézeteket tekinthessen meg, amelyre sikeres bejelentkezés után elvándorolhat, illetve megtekinthet.

Vagyis feladata az AuthGuard-nak, hogy kiszűrje, hogy adott felhasználó bejelentkezett-e, vagy sem. Amennyiben igen a válasz az előzőr kérdésünkre, akkor bizonyos nézeteket elérhet, ellenkező esetben bárhova is szeretne elmenni, visszadobjuk a bejelentkeztető felületre.

AuthGuard, vagy másnéven Angular Őr az ***ng generate guard <Name>*** parancs segítségével valósítható meg. Az alkalmazásunkban a bejelentkezésért felelős szolgáltatás (AuthService) segítségével lekérdezzük, hogy az aktuális felhasználó bejelentkezett-e – vagyis ellenőrizzük, hogy a backend-től kapott Spring Security tokenje rendelkezésére áll-e – a(z) *isLoggedIn()* függvénnyel egy elágazás keretében. Amennyiben ez igaz, akkor logikai igazat térítünk vissza. Ellenkező esetben egy olyan nézetre dobjuk vissza, ahol a bejelentkezést megteheti.

Magát az AuthGuard-t pedig a routing során a **canMatch** mező értékeként adjuk át szögletes zárójelek között. Ennek jelentése, hogy csak azon útvonalak – melyeken alkalmaztuk ezt a *canMatch* mezőt – lesznek csak elérhető, ahol a mező logikai igaz értékkel tér vissza.

3.3 Lazy moduling

Alapértelmezetten az Angular **eagerly loading**-t használ. Ennek során ahogy betöltödik az Angular alkalmazásunk minden egyes modul is betöltésre kerül attól függetlenül, hogy éppen szükségesek vagy sem. Nagy alkalmazások esetén ezért célszerű **lazy loading**-t használni, mely során adott modul csak akkor kerül betöltésre, mielőtt annak egy nézetére a felhasználó ellátogatna. (Például alkalmazásunkban a bejelentkezés vagy regisztráció elvégzése előtt nincs szükségünk mondjuk a felhasználó posztjainak a megjelenítését végző modulra.)

Így minden összetartozó komponens egy-egy modulba került a fejlesztés során. Így például a bejelentkezés és regisztráció külön-külön komponensek egy authentikációt megvalósító modulban, így ezek egyszerre kerülnek betöltésre. Minden más funkció,

mint például posztok listázása, felhasználó adatainak megjelenítése, csoportok listázása külön-külön modul az alkalmazásban.

3.3.1 Megvalósítása

Ehhez szükséges elem a korábban is említett modult készítő Angular parancssori parancs, mellyel minden modulnak elkészíthető a saját, releváns routing-ja. Ezen routing-okat pedig majd az **app-routing.module.ts** fájlban kell összefogni. Itt minden modulnak kezdeti elérési útvonalát meg kell adni a *path* mezőben. Majd a modul egyedi routing-jában elérhető útvonalakat a *loadChildren* mezőben kell megadni.

3.4 Paraméterezett routing

Az alapértelmezett routing mellett adódhat olyan routing is, ahol az elérési útvonal valamilyen paramétert tartalmaz, melyet adott komponensnek fel kell dolgoznia vagy éppen eltárolnia. Ezt hívjuk paraméterezett routing-nak.

Ilyen például a felhasználók saját oldalainak a megjelenítése, ahol az elérési útvonalban szerepel az adott felhasználó backend-ről érkezett azonosítója. Itt az elérési útvonal a következőként néz ki, **user-page/:userId**, ahol a kettősponttal elválasztott érték az úgynevezett paraméter, amely értékében mindig változhat a program élettartama alatt. Amennyiben például a *userId* a bejelentkezett felhasználó azonosítója, akkor az megtekintheti, törölheti és módosíthatja a UserPage komponensen keresztül posztjait. Ellenkező esetben csak a másik felhasználó posztjainak a megtekintése érhető el szintén ugyanezen komponensen. Vagyis, a paraméterezett routing egy komponensnek, akár többféle megjelenítést, működést is adhatunk. Ekkor nincs szükség új komponens írására, a komponensek újrafelhasználhatóvá válnak.

Az elérési útvonalban szereplő *userId* paraméternek az értéket a **user-page.component.ts** fájlban fogadjuk. Ehhez a **@Input** annotációt használjuk. Mivel az elérési útvonalban szövegként kezeli, és szövegként jelenik meg az érték, így az annotáció *transform* mezőjének segítségével átalakíthatjuk egy számmá, ahogy az a backend-n is szerepel.

3.5 További backend-i feladatok

Nemcsak frontenden, de backend-n is fejlesztettem. Ugyanúgy, ahogy kollégám, szintén készítettem Controller, Service, Repository osztályokat a megfelelő logikákkal.

Továbbá megvalósítottam az adatbázis entitások és a frontenden megjelenítendő adatok közötti konvertálást.

3.5.1 Konvertálás

Konvertálás elsődleges célja, hogy – alkalmazásunk esetében – egyirányú adatkonverziót biztosítson az adatbázis entitások és a frontenden megjelenítendő adatok között. Jelen esetben a **Service** által lekérdezett adatbázisbeli entitásokat a **Controller** osztályokban egy soros utasítással át lehessen konvertálni a frontend-n megjelenítendő adatok típusára.

Az adatbázisbeli entitás osztályokat **Entity**, míg a frontend-n megjelenítendő adatosztályokat a **DTO** kulcsszavakkal láttuk el. Az Entity→DTO konvertálás megvalósítására készítettem az **IGenericConverter<E, D>** interfészt. Ennek egy függvénye a *convertFromEntityToDTO()* metódus, mely visszatérési értékében **D**, vagyis DTO típust, míg **E** típusú paraméterében Entity típust vár.

Ezen függvényt kell minden speciális Entity→DTO konvertálás során megvalósítani egy külön osztályban. Például, ha a GroupEntity→GroupDTO konverziót szeretnénk elérhetővé tenni egyetlen metóduson keresztül, akkor létrehozunk egy GroupConverter osztályt, amely implementálja az IGenericConverter interfészt.

4 Önálló munka értékelése, eredmények

Az egység teszt írásnál alapvető rendezési elve, hogy minden osztályhoz egy tesztosztály tartozik, azokon belül a tesztmetódusok pedig egy-egy metódus viselkedését ellenőrzik. Külső hívásokat nem engedélyezzük ebben az esetben, ilyenkor ezeket a Mockito nevű könyvtárral lehet “kimockolni”. Ez azt jelenti, hogy gyakorlatilag elhisszük, hogy működésük helyes, mivel azok is le vannak tesztelve.

Léteznek átfogóbb use-case-t fedő tesztek is. Ebből található meg néhány az applikációban. Itt egy adott funkció tesztelése a cél, minden alkalmazásbeli komponenst megmozgat. Itt is, mint a fenit unit teszteknél 3 részre bontjuk a teszteket (a megnevezés változhat, de működési elvükben ugyanaz): *arrange,* ahol a használt változókat inicializáljuk, az *act*, ahol a tesztelni kívánt metódust meghívjuk és az *assert,* ahol a viselkedés verifikációja zajlik.

5 Irodalomjegyzék

[1] „AllArgConstructor,” [Online]. Available: https://stevenmwesigwa.com/tutorials/project-lombok/8/how-to-use-allargsconstructor-annotation-with-project-lombok-in-java-applicat.

[2] „@OneToMany,” [Online]. Available: https://vladmihalcea.com/the-best-way-to-map-a-onetomany-association-with-jpa-and-hibernate/.

[3] Hibernate, „@OneToMany,” [Online]. Available: https://www.baeldung.com/hibernate-one-to-many.

[4] Medium, „Full-Stack Web App with Angular and Spring Boot,” [Online]. Available: https://medium.com/@attia.imeed/building-a-full-stack-web-application-with-angular-15-and-spring-boot-3-from-scratch-2023-df12c1e01233.

[5] Hibernate, „@ToString,” [Online]. Available: https://www.baeldung.com/lombok-tostring.

[6] Lombok, „Constructor annotations,” [Online]. Available: https://projectlombok.org/features/constructor.

[7] Medium, „The dangers of Lombok annotations,” [Online]. Available: https://medium.com/@miguelangelperezdiaz444/the-hidden-dangers-of-lombok-annotations-in-your-java-code-what-you-need-to-know-8acdce2d6b89.

[8] Hibernate, „JPA/Hibernate Associations,” [Online]. Available: https://www.baeldung.com/jpa-hibernate-associations?fbclid=IwAR30d_HKQuVrH4N3hPt_itG2GX98COvu-cag4WFGGvjD3VO4hOGxNpgDE1U.

[9] Hibernate, „Mapping LOB Data,” [Online]. Available: https://www.baeldung.com/hibernate-lob?fbclid=IwAR0mBEeDU5tVIMn1w-lUDUDIRNE6MLLNHltCrAO8EnnaPArmFzFOY_6HtHA.

[10] Hibernate, „Building Web App with Spring boot and Angular," [Online]. Available: https://www.baeldung.com/spring-boot-angular-web.

[11] Angular, „Angular tutorial," [Online]. Available: https://angular.io/guide/router-tutorial-toh.

[12] S. Overflow, „Spring Security: bearer token (Postman error)," [Online]. Available: https://stackoverflow.com/questions/49802163/authorization-bearer-token-angular-5.

[13] FreeCodeCamp, „@Input," [Online]. Available: https://www.freecodecamp.org/news/use-input-for-angular-route-parameters/.