



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

Informatikai Kar

Információs Rendszerek Tanszék

UML-Conference elosztott webes alkalmazás

Témavezető:

Szalai-Gindl János Márk

adjunktus

ELTE Információs Rendszerek Tanszék

Szerző:

Szeifert Péter

Programtervező informatikus BSc.

Budapest, 2021.

1. Tartalomjegyzék

1. Tartalomjegyzék.....	2
2. Bevezetés.....	4
3. Felhasználói dokumentáció	5
3.1 Rendszerkövetelmények	5
3.2 A program telepítése.....	5
3.3 Program futtatása Windows esetén	5
3.4 Program futtatása Linux esetén.....	5
3.5 A program használata.....	6
3.5.1 Felhasznált fogalmak.....	6
3.5.2 Belépés az oldalra	6
3.5.3 A felhasználói felület	9
3.5.4 Fájlmenedzser panel	10
3.5.5 Fájlok megosztása	12
3.5.6 A szerkesztőfelület	14
4. Fejlesztői dokumentáció.....	18
4.1 Felhasznált technológiák	18
4.1.1 Kliens oldal.....	18
4.1.2 Szerver oldal	18
4.1.3 Adatbázis	19
4.1.4 Fejlesztői környezet.....	19
4.2 Funkciók összegzése felhasználói esettanulmányokkal (tervezés).....	20
4.2.1 Egy projekt felépítése.....	20
4.2.2 AuthModule funkciói.....	20
4.2.3 A főbb menüpontok	21
4.2.4 Fájlmenedzser funkciói.....	22

4.2.5	Szerkesztőfelület funkciói.....	23
4.3	Szerkezeti felépítés	24
4.3.1	Kliensoldal.....	25
4.3.2	Szerver oldal	40
4.4	Tesztelés	53
4.4.1	Unit tesztelés	54
4.4.2	Integrációs tesztelés.....	54
5.	Továbbfejlesztési lehetőségek.....	63
5.1	File Manager	63
5.2	Editor	63
5.3	Felhasználói fiókok	63
6.	Összegzés.....	65
7.	Irodalomjegyzék.....	66

2. Bevezetés

Sokféle diagram szerkesztő áll rendelkezésünkre manapság, de többrésztvevős alig akad. Nagyobb szoftverek megtervezése esetén fontos, hogy a tervezési folyamat minél dinamikusabb, gyorsabb legyen, ezért a munkám során egy elosztott webes alkalmazást készítettem, amelyben közösen, akár egyidőben is lehet UML osztálydiagramokat és csomagdiagramokat szerkeszteni.

Azért ezt a témát választottam, mert korábbi tanulmányaim és egyéb fejlesztési munkáim során mindig nehézséget okozott az, hogy többrésztvevős diagramtervező szoftvert találjak. A 4. és 5. félévemben részt vettem egy ösztöndíjas egyetemi start-up pályázaton, ahol diáktársaimmal egy Androidos alkalmazást készítettünk. A tervezés kezdeti fázisában hasznos lett volna egy olyan eszköz, amelynek segítségével közösen lehet megtervezni egy többretegű alkalmazás architektúráját. Ezért választottam az UML Conference alkalmazást szakdolgozatom témájának.

A program 3 rétegből áll. Egy [Angular](#)[1] kliensből, Java [Spring-boot](#)[2] szerverből, illetve [PostgreSQL](#)[3] adatbázisból. Az interaktív szerkesztő felülethez natív, szöveg alapú web socketeket használtam, az összes többi kommunikáció [REST API](#)[4]-n keresztül történik.

A dokumentum tartalmazza a felhasználói dokumentációt, melynek forgatásával egy új felhasználó közelebb kerül és segítséget kap a program helyes és produktív használatához.

Emellett tartalmazza a fejlesztői dokumentációt, amely a fejlesztőknek nyújt segítséget. Bemutatja a főbb komponensek egymáshoz való viszonyát és struktúráját, a legfontosabb osztályokat, szervízeket, függvényeket, teszteseteket. Ennek a fejezetnek a forgatásával az olvasó közelebb kerül a szoftver működésének megértéséhez, illetve reprodukálásához. A tesztelési tervet és azok eredményeit a fejezet végén olvashatjuk.

A továbbfejlesztéshez ötleteket találhatunk a dolgozat végén, az irodalomjegyzék előtt.

3. Felhasználói dokumentáció

Ez a program olyan embereknek készült, akik valamilyen tervezési feladatot csapatban vagy egyedül szeretnének elvégezni, de közben fontos számukra, hogy mindez dinamikusan, valós időben, jól átláthatóan és kényelmesen történjen.

3.1 Rendszerkövetelmények

Windows esetében a 3 programkomponens futtatásához legalább 1,5 GB memóriát ajánlott szabadon hagyni. A java szerver átlagosan 800-1000 mb memóriát foglal le, a kliensszerver és az adatbázisszerver memóiafelhasználása elhanyagolható. Processzor teljesítmény tekintetében bármi megfelel, amin az operációs rendszer elfut. A lefordított, futtatható állományok kevesebb, mint 150 mb tárhelyet foglalnak.

Szükségünk lesz továbbá egy böngészőre. Én chrome-ot, operát vagy firefox-ot ajánlok a használathoz. Az Internet Explorer és Edge böngészők nem támogatottak.

3.2 A program telepítése

- Telepítsük fel A [Java8-hoz tartozó JRE](#)[5]-t, adjuk hozzá a környezeti változókhoz! (fűzzük fel a classpath-ra)
- Telepítsük fel a [nodeJs 14-es verzióját](#)[6] vagy újabbat, ezt is adjuk hozzá a környezeti változókhoz!
- Telepítsük a [PostgreSQL 13](#)[7]-os verzióját!
- Indítsuk el az `init.bat` fájlt. Ez feltelepíti az Angular kliens futtatásához szükséges `http-server` segédprogramot, amely a kliensszervert hosztolja. Linux esetén a fájl tartalmát írjuk be a terminálba.

3.3 Program futtatása Windows esetén

1. Lépjünk be a futtatható_kódok mappába!
2. Indítsuk el a `win_start_application.bat` fájlt.

3.4 Program futtatása Linux esetén

1. Lépjünk be a futtatható_kódok mappába!
2. Írjuk be a terminálba a `start_postgre.bat` fájl tartalmát.

3. Futtassuk a Java szerveret a `java -jar ./back-end/target/uml-conference-0.0.1-SNAPSHOT.jar` paranccsal!
4. Egy új terminálban futtassuk http szerveret a `http-server ./front-end/dist/front-end --port=8100` paranccsal!
5. Nyissuk meg a böngészőnket a <http://localhost:8100> –as címen!

3.5 A program használata

3.5.1 Felhasznált fogalmak

diagram

- Osztálydobozok, jegyzetdobozok, csomagdobozok vonalak rendezett halmaza.

projektmappa

- Ez egy olyan különleges mappa, amely egy diagramot reprezentál a projekt struktúrájában

projekt

- projektmappák struktúrált halmaza. Egy fájltypus. Amikor a fájlmenedzserben megnyitunk egy projektet, akkor a projekt gyökérmappája nyílik meg számunkra.

fájl

- A program kontextusában fájlnek számít majdnem minden, amit a fájlmenedzserben látunk. Fájlnek számítanak a mappák, projektek és projektmappák. Nem számítanak fájlnek a virtuális osztálycímek a projektmappákban.

3.5.2 Belépés az oldalra

Nyissuk meg a böngészőt, navigáljunk el a <http://localhost:8100> oldalra! A program használatához szükségünk lesz egy felhasználói fiókra. A belépéshez kövessük a [3.5.2.1 Regisztráció](#) és a [3.5.2.2 Belépés](#) bekezdés utasításait!

3.5.2.1 Regisztráció

Ha először nyitjuk meg a programot, akkor szükségünk lesz egy felhasználói fiókra. Új fiók létrehozásához kövessük a következő lépéseket:

1. A fenti menüsávban kattintsunk a „**Register**” linkre!
2. Egy regisztrációs űrlap (1. ábra) jelenik meg számunkra, amelyet értelemszerűen töltünk ki, majd kattintsunk a „**Register**” gombra!
1. A regisztráció sikerességéről, illetve az esetleges formai hibákról az űrlap alatt kapunk visszajelzést (2. ábra). Ha sikerült regisztrálni, akkor bejelentkezhetünk a megadott adatainkkal.

1. ábra: Regisztrációs űrlap helyes kitöltése

Hibaüzenetek

Az űrlapra felvitt adatokra formai követelmények vonatkoznak, hiba esetén a program az űrlap alatt tájékoztat minket (2. ábra).

- A felhasználónév csak alfanumerikus, illetve [-._] karaktereket tartalmazhat.
- A jelszónak legalább 4 karakter hosszúnak kell lennie, illetve tartalmaznia kell legalább 1 betűt és legalább 1 számot.
- Az e-mail mezőben a az e-mail cím megfelel a *hivatalos formátumnak*[8] erre egy példát a 1. ábra-n láthatunk.

Akció leírás	Hibaüzenet (űrlap alatt)
Rossz a felhasználónév vagy jelszó kombináció	Error: Incorrect username or password
A felhasználónév formátuma nem megfelelő pl.: túl rövid, vagy speciális	username format is not valid. Make sure, it only contains alphanumeric characters and

karaktert tartalmaz.	[-.] characters
Az email formátuma nem megfelelő	Email format is not valid.
A jelszó formátuma nem megfelelő	Password format is not valid (minimum 4 characters, must contain at least one number and letter)

The screenshot shows a registration form titled "Registration" on a dark background. The form includes fields for Username, Name, Email, Password, and Repeat password. The Email field contains the text "hibás_email_format" and is highlighted with a blue border, indicating an error. Below the fields is a purple "Register" button. At the bottom of the form, the message "email format is not valid" is displayed in a light blue font.

2. ábra: Hiba a regisztráció során

3.5.2.2 Belépés

A program funkcionalitásának eléréséhez először is be kell jelentkezünk. Ha nem rendelkezünk fiókkal, akkor kövessük a [3.5.2.1-es](#) bekezdés utasításait. Ha rendelkezünk, akkor a következő utasításokat követve bejelentkezhetünk:

1. Kattintstunk a „**Login**” linkre a bejelentkezési űrlap eléréséhez.
2. Töltsük ki a bejelentkezési űrlapot értelemszerűen, majd kattintsunk a „**Login**” gombra!
3. Ha az adatok helyesek, akkor a rendszer beléptet minket, hozzáférést kapva a teljes funkcionalitáshoz.

Hibaüzenetek:

Ha helytelenek a beírt adatok, akkor az űrlap alatt kapunk tájékoztatást a hibáról.

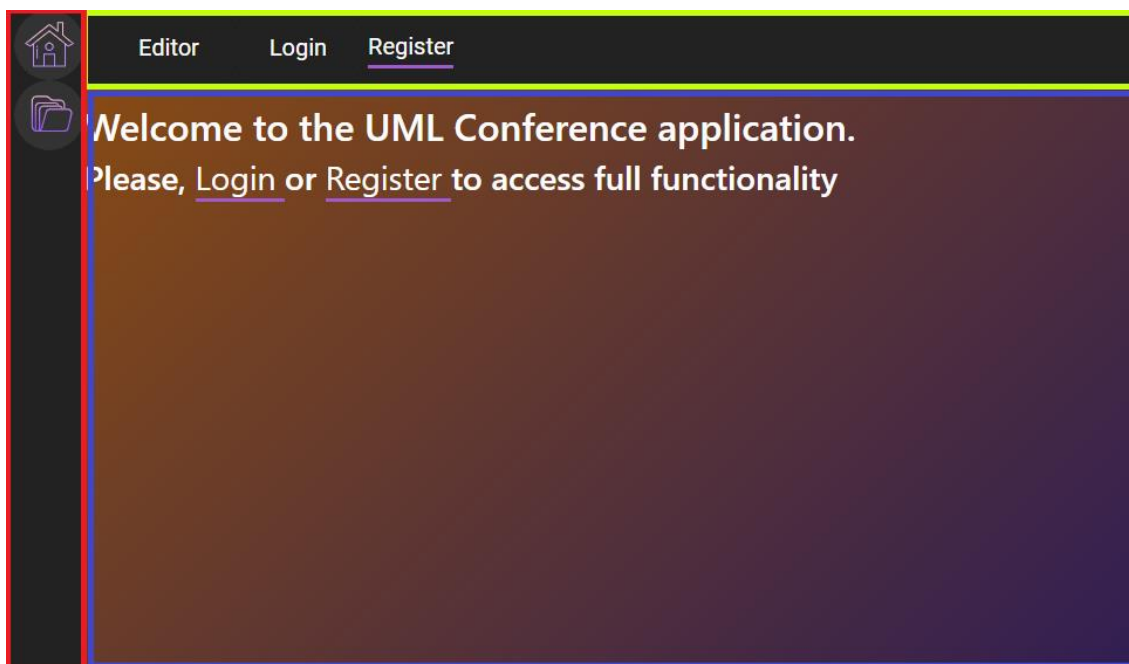
Akció leírás	Hibaüzenet(úrlap alatt)
Rossz a felhasználónév vagy jelszó kombináció	Error: Incorrect username or password

A felhasználónévre és a jelszóra itt is érvényesek a [3.5.2.1 Regisztráció](#) bekezdésben megfogalmazott szabályok, hibaüzenetek.

3.5.3 A felhasználói felület

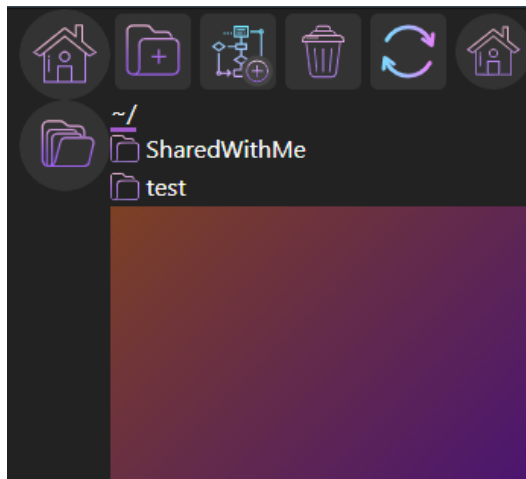
A felhasználói felület 3 részből áll, ezt a [3. ábra-n](#) láthatjuk:

- Felső menü sáv (zöld)
- oldalsó menüsáv (piros)
- Tartalmi oldal (kék).



3. ábra: felhasználói felület felépítése

Az oldalsó menüsávban a házikó gomb a „Home” oldalra visz, míg a mappa ikon a fájlmenedzser panelt ([4. ábra](#)) nyitja meg, amely a képernyő bal oldaláról úszik be. A fájlmenedzserről a [3.5.4-es](#) bekezdésben olvashatunk.



4. ábra Fájlmenedzser panel

Az Editor menüpont kiválasztása esetén a **Tartalmi oldalon** megjelenik a szerkesztőfelület. A szerkesztőfelületről a [3.5.6](#)-es bekezdésben olvashatunk.

3.5.4 Fájlmenedzser panel

Itt tudjuk böngészni a **mappáinkat, projektjeinket** és azon belül a **diagramjainkat**.

A panelt kinyitva a gyökérkönyvtárt nyitja meg az oldal. Itt egy fix mappa található `~/sharedWithMe/` ([4. ábra](#)).

- A `~/sharedWithMe/` mappában a felénk megosztott tartalmak láthatók. Ezek virtuális linkek, kattintáskor ugyan arra a fájlra navigálnak, mint amit a másik felhasználó is lát.

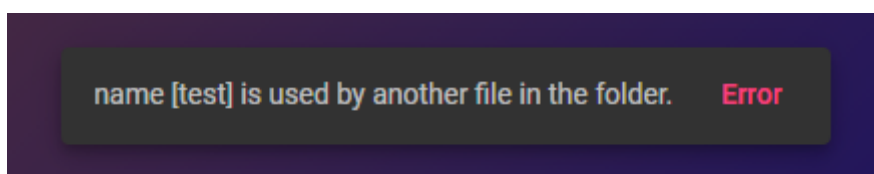
A fájlmenedzser felület 2 részből áll: Egy **eszköztárból (felül)** és nagyobb részt **böngészőből (alul)**. A fájlmenedzserrel projekteket, mappákat hozhatunk létre, törölhetünk, illetve oszthatunk meg másokkal. Megosztáskor a célszemély olvasási jogot kap a mappákhoz, illetve szerkesztői jogot a projektekhez. Másik személy projektjét vagy mappáját nem törölhetjük ki.

Minden normális fájlkezelőhöz hasonlóan a Fájlmenedzser panel is rendelkezik az alapvető funkcionalitásokkal: Kattintással elemeket jelölhetünk ki, illetve jelölést szüntethetünk meg. A kijelölt elemek kék keretet kapnak.

Felül, az eszköztárban 5 gomb van: mappa létrehozása, projekt létrehozása, törlés, Frissítés, Gyökérmappa elérése (Home).

3.5.4.1 Mappa vagy projekt létrehozása

A „mappa létrehozása” és a „projekt létrehozása” gombbal létrehozhatunk egy új mappát vagy projektet oda, ahol jelenleg állunk. A gomb lenyomása után a fájllista végén egy beviteli mező várja, hogy beírjuk a kívánt nevet. Nem megfelelő, vagy már használt név bevitele esetén a program kis szövegbuborékban, a képernyő alján jelzi a hibát (5. ábra).



5. ábra: Hibaüzenet példa

Hibaüzenetek:

Akció Leírása	Hibaüzenet (Alulról felugró kisablakban)
Olyan nevet adunk meg, amit egy másik fájl használ (azon a szinten, ahol állunk)	name [...] is used by another file in the folder (5. ábra).
Üres vagy „~” nevet adunk meg.	name [...] is not valid.
Másik felhasználó megosztott mappájában szeretnénk létrehozni valamit.	Error: You are not the owner of the parent folder.
A sharedWithMe mappánkba szeretnénk létrehozni valamit.	You can not create files here.

3.5.4.2 Fájl törlése

A törölni kívánt fájlt jelöljük ki bal kattintással, majd kattintsunk a kuka ikonra az eszköztárban! A gomb lenyomása után a kiválasztott elem eltűnik a listából. Ha más által megosztott tartalmat szeretnénk törölni, akkor a program hibaüzenetekkel válaszol, ugyanis más által létrehozott tartalmakat nem törölhetünk ki.

Hibaüzenetek:

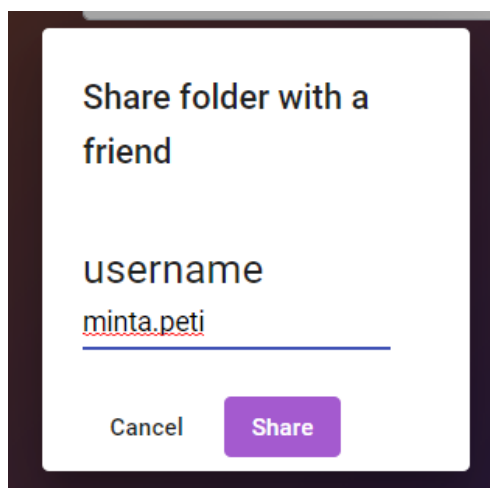
Akció Leírása	Hibaüzenet (Alulról felugró kisablakban)
Más által létrehozott tartalmat szeretnénk törölni	Error: You are not the owner of the file.
Speciális mappát szeretnénk törölni. Például gyökérmappát vagy a „sharedWithMe” mappát.	This file can not be deleted.

3.5.5 Fájlok megosztása

Egy mappa megosztásához jelöljük ki azt, majd kattintsunk a mellette található share gombra (6. ábra) ! Ha megtettük, akkor a megosztás ablak jelenik meg a képernyőn (7. ábra).

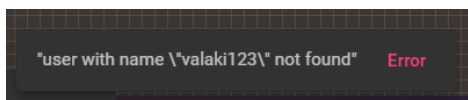


6. ábra: fájl kijelölése

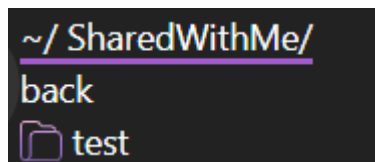


7. ábra: megosztás ablak

Megosztott elemeket nem lehet tovább osztani. Ezt csak az eredeti tulajdonos teheti meg. Csak mappákat oszthatunk meg, projekteket önmagában nem. Ha a szerver megtalálta a beírt nevet, akkor a mappa bekerül a vele megosztott mappák közé (9. ábra). Ha nem, a szerver hibaüzenetet küld kis felugró ablakban (8. ábra).

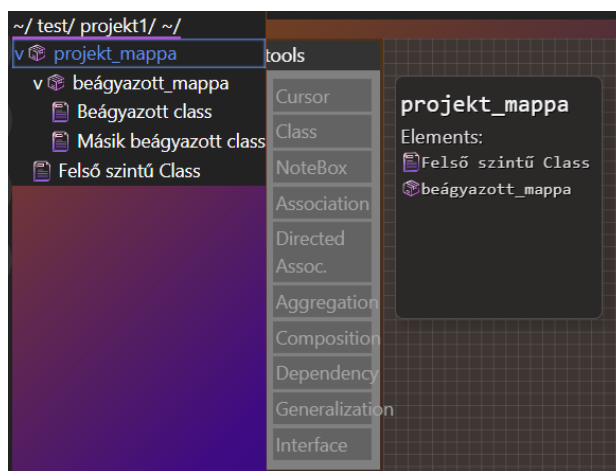


8. ábra: nem létező felhasználóval próbálunk megosztani valamit



9. ábra: a mappa megosztás sikeres, a másik felhasználó számára már látható a „test” mappa

Ha a mappa amiben állunk, egy **projektmappa**, (egy projekt gyöker mappája is ilyen) akkor az eszköztár részen már csak 4 gomb látható. A projekt létrehozása gomb eltűnt, ugyanis projekt módban erre nincs szükségünk. Továbbá, ha a fájlmenedzser panelen létrehozunk egy **projektmappát**, akkor az **automatikusan megjelenik a szerkesztőfelületen (10. ábra).**



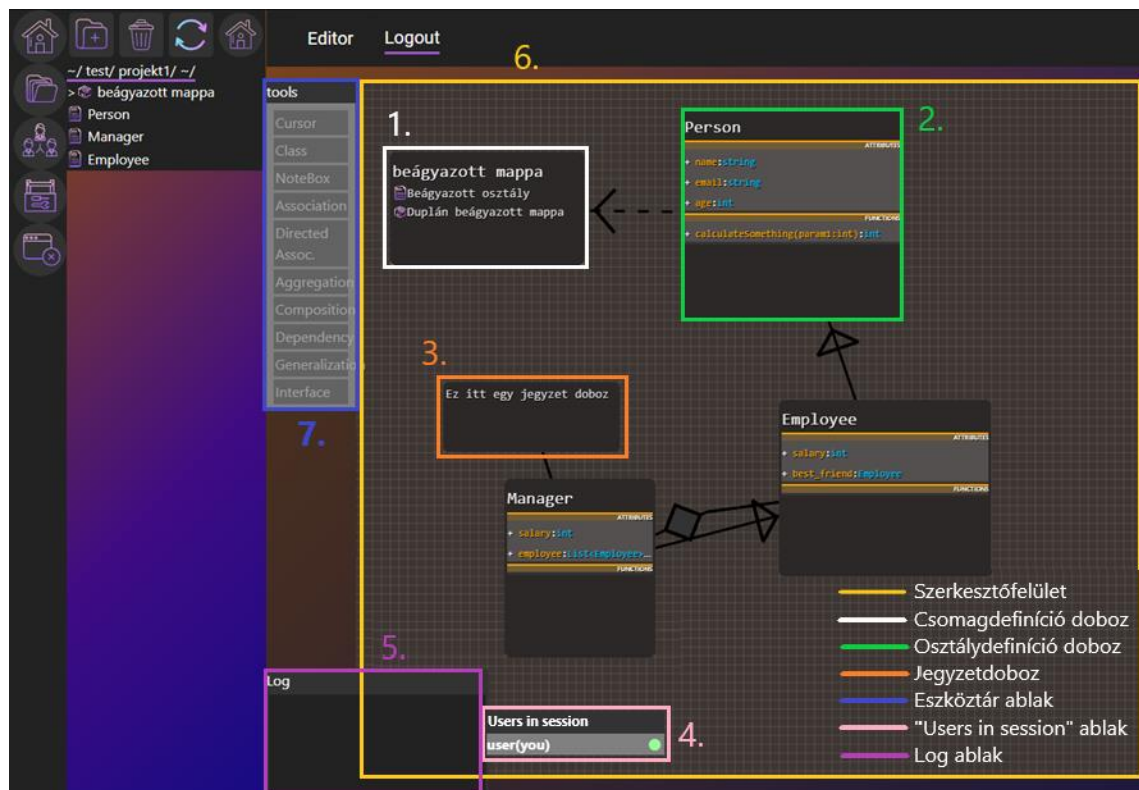
10. ábra: a szerkesztőfelület és a fájlmenedzser szoros kapcsolata

Ha egy osztálydobozt létrehozunk, (3.5.6.1.2 *Class*) akkor az a szülő mappa alá **virtuálisan bekerül**, hogy jobban átlátható legyen, mi hova tartozik (10. ábra). Továbbá a mappa ikonja mellett balra található kis nyílacszkával kinyithatók, illetve becsukhatók a mappában található elemek. Osztálydobozok szerkesztésekor a nagyszülő diagramban automatikusan frissülnek a csomagdefiníciók.

Hibaüzenetek:

Akció Leírása	Hibaüzenet (Alulról felugró kisablakban)
Más által megosztott tartalmat szeretnénk tovább osztani.	Error: You are not the owner of the file.
A megosztás ablakban megadott felhasználó nem létezik.	User not found: ...

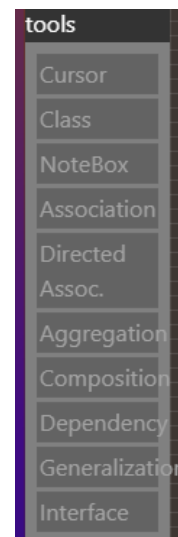
3.5.6 A szerkesztőfelület



11. ábra: szerkesztő felület felépítése

A szerkesztőfelület felépítését a 11. ábra-n láthatjuk. A felület a következő elemekből áll össze:

- **„Users in session” ablak:** ahol a munkamenetben résztvevő felhasználók neveit és a hozzájuk társított színeket látjuk.
- **„Log” ablak:** Itt jelennek meg az esetleges hibaüzenetek és a zárolási kivételek kezelései.
- **Szerkesztőfelület** Itt szerkeszthetjük a fájlmenedzserben kiválasztott diagramot.
 - **Csomagdoboz:** beágyazott struktúra ábrázolásához.
 - **Osztálydoboz:** Osztály vázlat megjelenítéséhez
 - **Jegyzetdoboz:** Megjegyzések megjelenítéséhez.
 - **Vonalak:** A kapcsolatok szemléltetéséhez.
- **Eszközök – „Tools” ablak:** A szerkesztőeszközök tárolására használatos.



12. ábra
Eszköztár ablak

Az ablakok mozgathatók, de nem átméretezhetők. A szerkesztőfelületet az egérgörgő használatával nagyíthatjuk, illetve kicsinyíthetjük.

3.5.6.1 Toolbox ablak

Itt osztálydobozok [3.5.6.1.2 Class](#), megjegyzések [3.5.6.1.3 NoteBox](#), illetve kapcsolati vonalak [3.5.6.1.5 Vonalak](#) beszúrását választhatjuk ki ([12. ábra](#)).

3.5.6.1.1 Általános tulajdonságok

Ha valaki egy *szabad*¹ dobozt, vonalat vagy attribútumot kiválaszt (megkattint), akkor az elem a zároló felhasználó színének megfelelő kiemelést kap. Ha esetleg színvak vagy szintévesztő felhasználó használná az alkalmazást, az egyértelműség kedvéért a lezárt elem mellett egy ceruza(■) színbórium is megjelenik a lezáró felhasználó oldalán, a szerkesztés engedélyezését megerősítve. A többiekénél egy lakat(🔒) jelenik meg a szerkeszthetetlenség egyértelmű jelzéséért. Vonalak esetén az ikonok nem jelennek meg.

3.5.6.1.2 Class

Ha kiválaszjuk a „Class” eszközt, akkor a következő alkalommal, amint az egeret lenyomjuk a vásznon, egy új osztálydoboz jön létre. Az egér húzásával beállíthatjuk a doboz méretét. A doboz méretét és pozícióját bármikor módosíthatjuk. Elég csak szimplán rákattintani és „drag’n drop” módszerrel már mozgathatjuk is. Ha a doboz szélére visszük az egeret, akkor a kurzor nyilacskára vált, az átméretezés irányához alkalmazkodva. A keretet megfogva állíthatunk a doboz méretén. A doboz kiválasztásának menete a [3.5.6.1.1](#)-os bekezdés szerint történik. A del gomb lenyomása esetén a doboz törlődik az összes hozzá kapcsolódó vonallal együtt.

3.5.6.1.3 NoteBox

Ezzel az eszközzel egy ugyanolyan dobozt hozhatunk létre, mint a [3.5.6.1.2 Class](#) esetében, teljesen hasonló tulajdonságokkal. Itt a legfőbb különbség a dobozon belüli

¹ Ha, nincs színes keret az adott elem körül, illetve lakat sem, akkor az elem szabad.








tartalom, ugyanis ez a doboz csak egyszerű szöveg tárolására alkalmas. Megjegyzések írásához tökéletes.

3.5.6.1.4 Csomagdefiníciós dobozok

Ezeket a dobozokat nem lehet létrehozni az eszköztár segítségével, sem kitörölni a del gombbal. Ezeket a műveleteket a fájlmenedzser végzi a háttérben. Ha új projektmappát hozunk létre, akkor az Csomagdefiníciós dobozként jelenik meg a szerkesztőfelületen. Projektmappa törlése esetén értelemszerűen eltűnik a szerkesztőfelületről a hozzátartozó doboz. Ezt a típusú dobozt ugyanúgy mozgathatjuk, átméretezhetjük, mint a [3.5.6.1.2 Class](#) vagy a [3.5.6.1.3 NoteBox](#) esetében.

3.5.6.1.5 Vonalak

3.5.6.1.5.1 Vonaltípusok

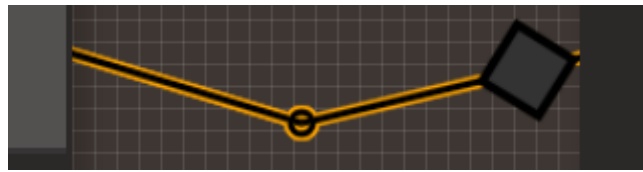
- Asszociáció 
- Irányított asszociáció 
- Aggregáció 
- Kompozíció 
- Függőség 
- Öröklődés 
- Interface 

3.5.6.1.5.2 Vonalak létrehozása

Vonalakat úgy húzhatunk, hogy az előbb felsorolt vonalak egyikét kiválasztjuk, egy dobozra rányomunk, majd elkezdjük húzni az egeret egy másik doboz felé. Az egér felengedésével a program rögzíti a vonalat, amennyiben a 2. végpont pozíciójában is volt egy doboz. Ellenkező esetben a megkezdett vonal törlődik.

Ha egy szabad (nincs rajta színes kiemelés) vonalra kattintunk, akkor az a zároló színének megfelelő kijelölést kap (a [13. ábra-n](#) a sárga színű felhasználó jelölte ki a vonalat). Ha a vonal bármelyik pontját megfogjuk, akkor ott egy töréspont keletkezik ([13. ábra](#)), amelyet akárhova elhelyezhetünk a „drag’n drop” módszerrel. A töréspontokkal a vonalunkat több kisebb szakaszra oszthatjuk, hogy jobb minőségű,

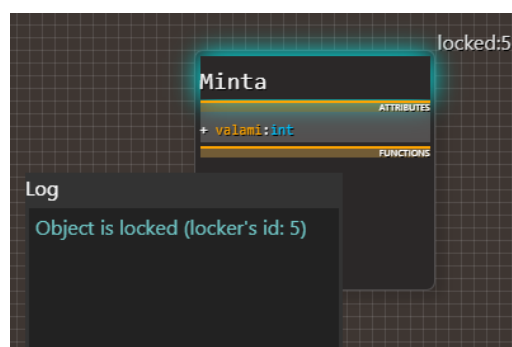
átláthatóbb ábrákat készíthessünk. Egy kiválasztott vonalat a del gomb lenyomásával törölhetünk.



13. ábra: töréspont

3.5.6.2 Log ablak

Itt az esetleges hibákról kapunk értesítéseket. Például nem szerkeszthető elemet próbálunk módosítani, vagy ha a magas válaszidő miatt nem megengedett műveletet hajtunk végre, akkor a visszaállításokról, figyelmeztetésekről itt kapunk tájékoztatást. Az egyes sorok fölé húzva az egeret az halvány kiemelést kap, illetve a tárgyban szereplő objektum is a szerkesztőfelületen (14. ábra).



14. ábra: Log ablak, működés közben

4. Fejlesztői dokumentáció

4.1 Felhasznált technológiák

4.1.1 Kliens oldal

A Kliens oldalt, amit a végfelhasználó lát, teljes mértékben AngularJS keretrendszerben készítettem. Azért az Angulart választottam, mert nagyon megtetszett a moduláris felépítése, a komponenseinek újrafelhasználhatósága, a nézetek egymásba ágyazása könnyedén és átláthatóan. A nézetmodell (.ts fájl) elkülönül a nézettől (.html) illetve a stíluslaptól (.css vagy .scss). Az Angularról mindenképp érdemes megemlíteni azt, hogy a többi mai front-end keretrendszerrel ellentétben az Angular alapértelmezett módon typescript-et használ, amelynek sok előnye van. Főleg, ha valami nagyobb, vállalati méretű applikációt szeretnénk készíteni. A typescript tulajdonságaiból kiemelnék néhányat:

1. Lehet vele statikusan típusos javascript kódot írni.
2. A típusozott „typescript” kódot a saját fordítója mezei javascript kóddá generálja.
3. Az Intellisense figyelmeztet a típusok betartására és helyes használatára. Ezzel rengeteg időt lehet megspórolni. Egyrészt a folyamatos autocomplete funkcióval gyorsabb a kódírás, másrészt típusibákra, elgépelésekre már fordítási időben figyelmeztet a fordító.

4.1.2 Szerver oldal

A szerver oldalt, ahol az üzleti logika van, Java nyelvben írtam meg, Spring Boot keretrendszert használva. A Spring Boot egy nyílt forráskódú Java alapú keretrendszer, amelyet leginkább a microservice-k építésére használnak az iparban. A Spring Boot a népszerű Spring keretrendszert használja a sorok mögött, a legszembetűnőbb különbség az, hogy míg Spring XML alapú Bean (objektum) definiálást használ, addig SpringBoot-ban ezt annotációk megadásával sokkal egyszerűbben és átláthatóbban megtehetjük. Továbbá Rengeteg konfigurációt „*out of the box*” megírtak nekünk, hogy

csak a **Controller**, **Service** és **Repository** rétegeket kelljen megírunk. Természetesen minden Spring-es funkció elérhető és igény szerint konfigurálható.

Springen belül rengeteg dependenciát, „könyvtár csomagot” használtam ezek közül a legfontosabbak:

- `spring-boot-starter-data-jpa` Az adatbázis kommunikációhoz
- `spring-boot-starter-web` Az alap RestController funkcionalitáshoz
- `spring-boot-starter-test` A teszteléshez
- `spring-websocket` Az editor socketes kommunikációjához
- `jackson-databind` A Json-Java POJO könnyed konverziójához
- `spring-boot-starter-security` A biztonsági alrendszerhez.

4.1.3 Adatbázis

A perzisztencia réteg feladatait a PostgreSQL adatbáziskezelőre bízam. Itt fontos megjegyezni, hogy esetemben a spring JPA és hibernate könyvtárcsomagjai kommunikálnak az adatbázissal SQL nyelven. A táblákat és kapcsolatokat közvetlenül a Java forráskódban, [annotációk](#)[9] segítségével lehet létrehozni pl.: `@Entity`(tábla), `@OneToMany`, stb... Röviden, az adattáblák és kapcsolatok SQL-ben való megírásának terhét levették a vállamról.

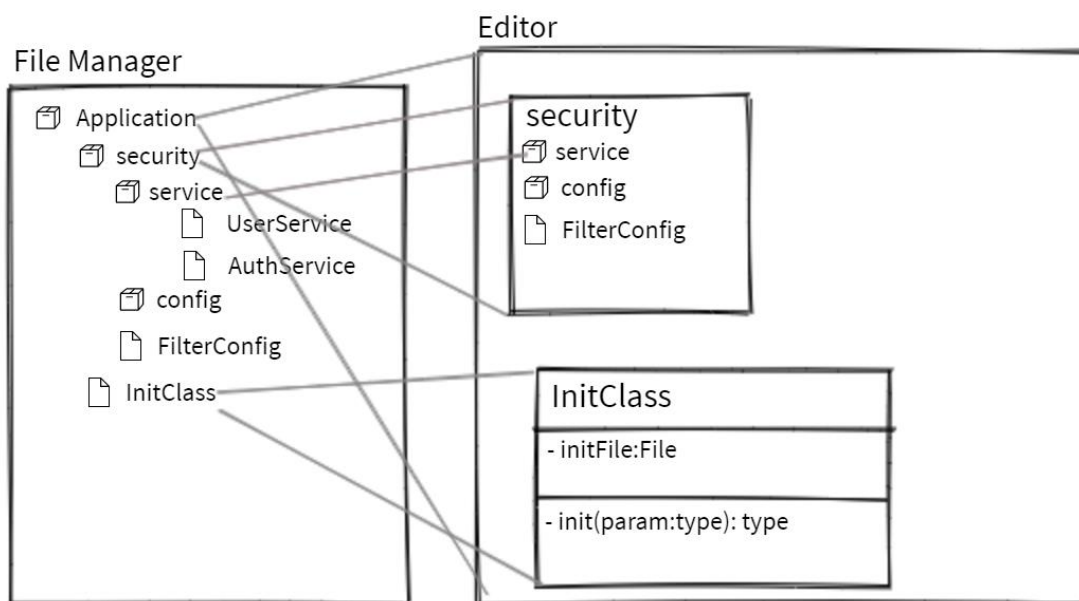
4.1.4 Fejlesztői környezet

A Kliens oldal fejlesztéséhez [Visual studio code](#)[10]-t használtam. Néhány Angular illetve html bővítmény telepítésével. A Spring-Boot szerver fejlesztéséhez [Apache NetBeans 12.0](#)[11]-t, a PostgreSQL adatbázis menedzseléséhez pedig [PgAdmin 4](#)[12]-et használtam.

4.2 Funkciók összegzése felhasználói esettanulmányokkal (tervezés)

4.2.1 Egy projekt felépítése

Egy projekt valójában diagramok strukturált halmaza. Egy projekt lényegében egy mappa, amelyben almappák (diagramok) vannak. Egy diagram osztálydefiníciók, megjegyzések, kapcsolatok és beágyazott diagramok egy csomagja, halmaza. Még a projekt kezdeti szakaszában készítettem a [15. ábraát](#) ez a fájlmenedzser és a szerkesztőfelület összefüggő működését mutatja. Ez egyébként a [10. ábra](#) vázlata.



15. ábra: egy példa projekt felépítése (vázlat)

4.2.2 AuthModule funkciói

GIVEN	WHEN	THEN
User belép az oldalra	Nincs bejelentkezve senki	„Home” oldal megjelenik
User belép az oldalra	Már be van jelentkezve valaki	A „Home” oldalra kerül a user, ahol a program köszönti őt.
User a regisztráció	Kitölti az email, username, Name és password	Ha az adatok formátumai megfelelőek, akkor a regisztráció sikeres. Most már be

linke kattint	mezőket, majd és a “Register” gombra kattint.	lehet lépni a megadott adatokkal. Nem megfelelő formátumok esetén hibaüzeneteket kapunk.
Login oldalon van a user	Beírja a login adatokat, enter	Ha helyes adatokat adtunk meg, a “Home” oldalra kerül a user, ellenkező esetben hibaüzenetet kapunk.

4.2.3 A főbb menüpontok

A “**be vagyunk lépve**” feltételt a továbbiakban “**BVL**” jelöléssel jelzem, ugyan is az elkövetkezendő funkciókhoz bejelentkezett felhasználói fiók szükséges.

GIVEN	WHEN	THEN	EXPLANATION
BVL	File browser menüpontra kattintunk	Bal oldali panelen kinyílik a fájlmenedzser	A fájlmenedzserben intézhetjük a mappáink, projektjeink, diagramjaink létrehozását, törlését, szerkesztését.
BVL, nem az EDITOR felület van megnyitva	Nagy “EDITOR” gombra kattintunk	A főképernyőn megjelenik a szerkesztő felület.	Ezen a felületen diagramjainkat szerkeszthetjük. A fent leírt funkciókat továbbra is használhatjuk a Bal oldali panelen . A fájl böngészőben kikereshetjük diagramjainkat, azokat megnyitva már szerkeszthetjük is. Ez élő socketes kapcsolattal történik, automatikus mentésekkel. Amint valaki más is elkezdti ugyanazt a diagramot szerkeszteni, automatikusan bekapcsolódik a munkamenetbe.

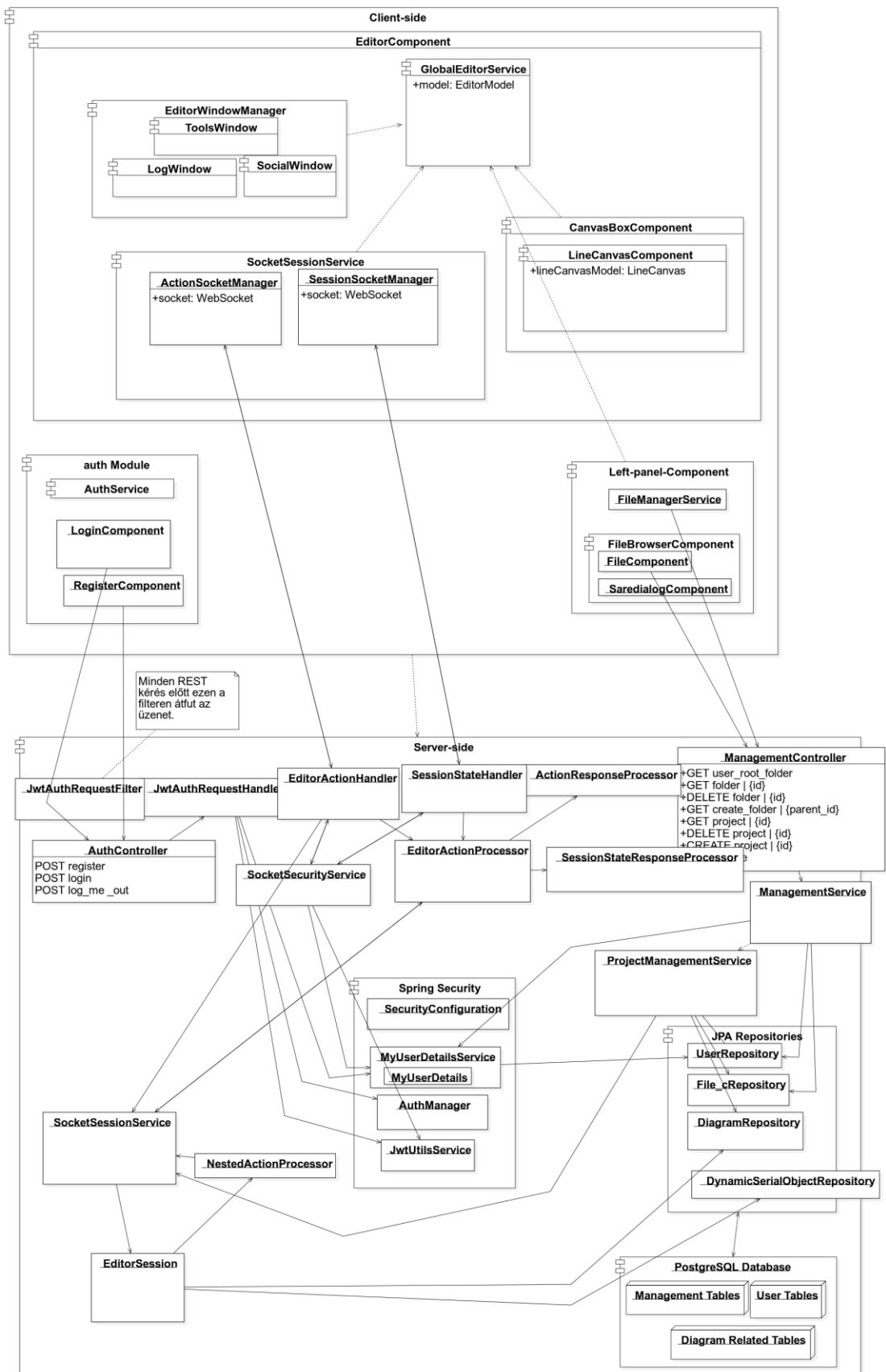
4.2.4 Fájlmenedzser funkciói

GIVEN	WHEN	THEN	EXPLANATION
File Browser panel meg van nyitva	felül, a mappa/projekt létrehozása gombra kattintunk	létrehoz egy mappát/projektet oda, ahol éppen állunk	A lista végén villogó kurzorral várja a program, hogy elnevezzük. Már foglalt vagy üres név esetén a szerver hibaüzenettel válaszol. A megosztási szabályok öröklődnek.
Kiválasztunk egy, vagy több elemet	Felül a törlés gombra kattintunk	Kitörli a kiválasztott elemet a listából	Ilyenkor fa szerűen a beágyazott mappák, projektek is törlődnek.
Kiválasztunk egy elemet	Felül az átnevezés gombra kattintunk	A választott elem neve villogni kezd, input mező keletkezik.	Átnevezés esetén, ha előtte meg volt osztva az elem másokkal, akkor náluk is átnevezésre kerül.
Kiválasztunk egy mappát	A megosztás gombra kattintunk	Megnyílik egy párbeszéd ablak, ahol egy másik nevét meg kell adni.	Ha létező felhasználó nevét adtuk meg, akkor az adatbázisban a file_share_table-ben létrejön egy kapcsolat, ami által a másik felhasználó hozzáfér a tartalomhoz.

4.2.5 Szerkesztőfelület funkciói

GIVEN	WHEN	THEN	EXPLANATION
Megosztott diagramot szerkesztünk, egyedül	Valaki más is elkezd szerkeszteni ugyanazt a diagramot	Ő is bekapcsolódik a szerkesztésbe.	Ilyenkor az elemek szerkesztését zárolásokkal látja el a program, ezt a szervert a memóriában jegyzi.
Megosztott diagramot szerkesztünk, többen	Osztályt leíró vagy komponens dobozra kattintunk , mert mozgatni, szerkeszteni vagy törölni szeretnénk.	Ha nincs zárolva , akkor új zárat kap (a benne lévő összes szövegmező is), csak szerkesztőnek engedélyezi a módosítást (pl.: kapcsolati vonalak módosítása), mozgatást, törlést . Ha zárolva van, akkor valahogyan értesül a felhasználó erről.	A zárolás alatt álló elemek nem módosíthatók, törölhetők vagy mozgathatók. Ezek az elemek a többiek számára egy lakatot kapnak, illetve egy keretet , piros színnel.
	Szövegmezőre kattintunk egy osztályban vagy komponens dobozban.	Ha nincs zárolva , akkor új zárat kap . Csak a szerkesztőnek engedélyezve a szerkesztést.	Ha szövegmezőt szerkesztünk, akkor szülőként maga a tartalmazó doboz különleges zárat kap: egy lakatot, hogy ne lehessen kitörölni, mozgatni, de a doboz többi mezője attól még szerkeszthető marad a többiek számára.

4.3 Szerkezeti felépítés



16. ábra: Az alkalmazás szerkezeti felépítése

A [16. ábra-n](#) a teljes alkalmazás magas szintű felépítését láthatjuk. A vonalak az információ áramlások és hívások irányát jelölik.

4.3.1 Kliensoldal

A kliens oldal megtervezésekor fontosnak tartottam, hogy egyértelmű, magától értetődő ikonokat, gombokat válasszak. Olyanokat amelyek nem törik meg a felhasználói szokásokat. Az ikonok nagyrészt a [Flaticons](#)[13] oldaláról töltöttem le. Az Angular komponenseket úgy terveztem meg, hogy közülük minél több újra felhasználható legyen. Ez az editor komponensben nagyobb értelmet nyer majd. Egy átlagos Angular komponens 4 fájlból tevődik össze: x.component.html, x.component.css, x.component.ts, x.component.ts

- x.component.html
 - Az x komponens nézet kódja, ez nem csak egy egyszerű html fájl. Rengeteg angular specifikus attribútumot írhatunk az egyes tag-eken belülre pl.:<div ng-If="this.value=='1' "> ezek közül néhány példa:
 - **ng-If="..."**: a "..." helyre ts kódot írhatunk. Ezt futási időben kiértékeli a javascript engine, ha igaz akkor az adott DOM elem megjelenik.
 - **ng-For="let item of list"**: egy listán végig iterálunk, az adott DOM elemet minden lista elemre létrehozza és egymás után fűzi.
- x.component.css
 - Itt definiálhatjuk azokat stílus szabályainkat, amelyek csak az adott komponens html kódjára lesznek érvényesek. Nem kötelező css formátumot használni, az angular projekt létrehozásánál scss illetve sass spreadsheet nyelveket is választhatunk én scss-t választottam. A globális szabályokat az src/app/styles.css fájlba írhatjuk bele.
- x.component.ts
 - Ez a komponens nézetmodellje. Itt definiálva van egy xComponent osztály amely @Component annotációval meghivatkozta a hozzá

tartozó html és css fájlokat. Itt fogalmazhatjuk meg a kliens oldali logikát, a felhasználói eseteket, interakciókat, gomblenyomásokat, egérmozgatásokat, kattintásokat és még sok minden mást.

- `x.component.spec.ts`
 - Itt a komponenshez tartozó teszteseteket fogalmazhatjuk meg. Az angular alapesetben Jasmine tesztkeretrendszert ad a kezünk alá, Karma tesztfutató szerverrel.

A teljes angular dokumentációt itt olvashatjuk: <https://angular.io/docs>

4.3.1.1 Auth module

Ez a modul a felhasználó autentikációjával foglalkozik. Két komponensből, illetve egy service-ből áll:

- LoginComponent
- RegisterComponent
- AuthService

4.3.1.1.1 Login Component

Ez a komponens semmi másért nem felel, csak azért, hogy a html kódban megírt form segítségével a felhasználó bejelentkezhessen. A `.ts` kódban a `sendLogin(a:AuthRequest)` függvényhívással bejelentkezési kérelmet küldünk a `/login` végpontra.

4.3.1.1.2 RegisterComponent

Ez a komponens szintén semmi másért nem felel, csak azért, hogy a html kódban megírt form segítségével a felhasználó beregisztrálhasson. A `.ts` kódban a `sendRegister(a:RegistrationRequest)` függvényhívással regisztrációs kérelmet küldünk a `/register` végpontra.

4.3.1.1.3 AuthService

Ez egy service. A service-ek olyan Typescript osztályok, amelyek más osztályokba injektálhatók dependenciaként. Ezt a `@Injectable` kulcsszó teszi lehetővé. Esetünkben az **AuthService**-nek egy lényeges függvénye van: `logout()`.

Értelemszerűen ezt meghívva kijelentkezünk. Fontos megjegyezni, hogy itt nem csak arról van szó, hogy kitöröljük a bejelentkezéskor kapott autentikációs token, hanem szerver oldalon a tokenünk feketelistára kerül. Ha időközben valaki meg is szerzi a tokenünket, nem tud vele mit kezdeni, mert kijelentkezéskor az szerver oldalon érvénytelenné válik. Feketelistára kerül, amelyet az adatbázisban tárolunk, egy trigger felel azért, hogy a token lejárat dátumát meghaladva a rekord törlődjön. A trigger kódja megtalálható a back-end/src/main/resources/sql_triggers.sql fájlban.

4.3.1.2 Bevezetés, Ősosztályok, Ősinterfészek

A következőkben megismerjük a fő kliensmodul működését, az editor modult, ezért fontosnak tartom az alap építőkövek előzetes megismerését:

4.3.1.2.1 ŐsInterfészek

- `SessionInteractiveltem`
- `SessionInteractiveContainer`
- `LogInteractive_I`

A **`SessionInteractiveltem`** interfész tartalmazza az összes olyan deklarációt, amely egy interaktív Item típusú komponens működéséhez kell.

A **`SessionInterActiveContainer`** kiterjeszti a `SessionInteractive Interface`-t néhány extra függvénydeklarációval. Ezek közül a legfontosabbak:

- `createItem(model: DynamicSerialObject, extra?: any),`
- `restoreItem(item_id: string, model: DynamicSerialObject)`
- `deleteItem(item_id: string).`

A **`LogInteractive_I`** interfész egyetlen metódust tartalmaz: `highlightMe(on: boolean, color: string)`. Ennek csupán annyi a jelentősége, hogy a megadott boolean kapcsolóval és színnel az implementációban a nézetet úgy frissítse, hogy a színnek megfelelő árnyékkal tűnjön ki.

4.3.1.2.2 Ősosztályok

- **`InteractiveltemBase`**

- **InteractiveContainerBase**

Az **InteractiveltemBase** egy olyan absztrakt Bázis osztály, amely implementálja, deklarálja az összes olyan függvényt, metódust, amely egy interaktív elemhez kell. Itt külön kiemelem az `editBegin()` és `EditEnd()` függvényeket, melyekben általában rendre `SELECT` és `UPDATE` akciók küldése történik. Továbbá konstruktorban magába injektálja a **EditorSocketControllerService**-t és a **CommonService**-t. Előbbi a **socket** alapú kommunikációért és a munkamenet állapotának tárolásáért felel, utóbbi az esetleges hibák naplózásáért. Az előbbihez hasonlókat lehet elmondani az **InteractiveContainerBase**-ről is, ugyanis kiterjeszti az **InteractiveltemBase** osztályt, de emellett implementálja az **InteractiveContainer** interface-t.

4.3.1.3 A legfontosabb service-k a fő modulban

4.3.1.3.1 GlobalEditorService

Ez talán a legfontosabb része a kliens oldalnak itt történik a **diagramok lekérése** és **tárolása**. Ha egy projektben navigálunk, projektmappák között váltogatunk minden esetben az `initFromServer(dg_id)` függvény fog lefutni. Itt eseménykezelés is történik, bárki feliratkozhat bármilyen eseményre. A diagram lekérésekor a `'diagram_fetch'` aliasra hallgató eseményt triggereli a szervíz. Erre az eseményre két komponens is fel van iratkozva:

- **LineCanvasComponent**: az esemény hatására üríti az előző munkamenet vonalait, helyükre a frisseket inicializálja be.
- **EditorSocketControllerService**: az esemény hatására létrehozza a 2 socketet és csatlakozik a szerver websocket endpointjaihoz (`/action` és `/state`)

Egy eseményfigyelő hozzáadása nagyon egyszerű. Csak meg kell hívni az `addListenerToEvent(target,fn,alias)` függvényt, ahol az `fn`-nek egy lambda függvénynek kell lennie 1 paraméterrel. Amikor a `triggerEvent(trigger_alias)` függvényt bárhonnán meghívjuk, akkor az összes lambda függvény lefut `target` paraméterrel, ahol a tárolt `alias` megegyezik a `trigger_alias`-al. A konstruktorban egy lambda függvénnyel feliratkozunk a `'canvas_size_update'`

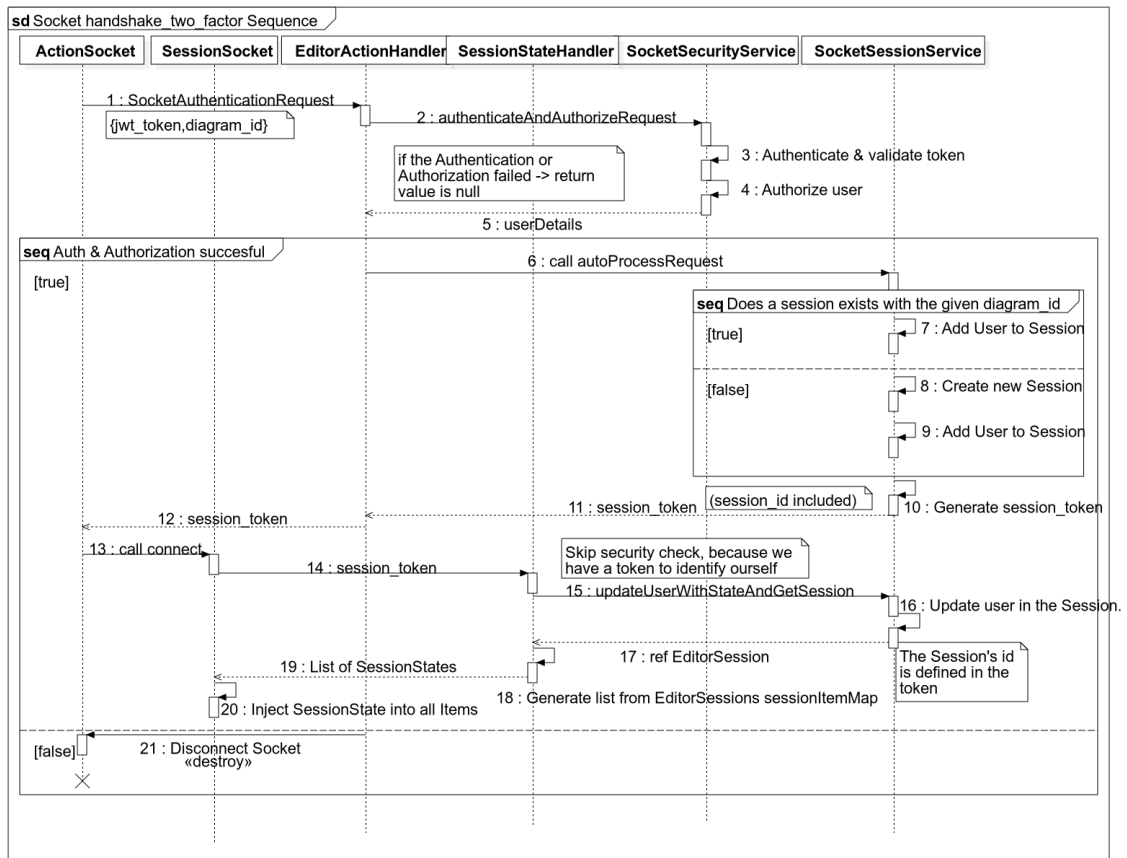
eseményre, amely tüzeléskor frissíti a **CanvasBox** és a **LineCanvas** dimenzióit, hogy az összes objektum kiférjen.

4.3.1.3.2 EditorSocketControllerService

Ez a szervíz az editor összes elemével kapcsolatban van, rajta keresztül fut a socket alapú aktív kommunikáció, itt kell felregisztrálniuk illetve leregisztrálniuk az egyes Itemeknek, konténereknek, amelyekkel interakcióba léphetünk a szerkesztőfelületen. Ezeket a `register`, `unregister`, `unregisterContainer` és `registerContainer` függvények felparaméterezett hívásával tehetjük meg. A service a nyilvántartott elemeket típusuknak megfelelően a következő listákba teszi:

- `itemViewModelMap:Pair<String, SessionInteractiveItem>[]`
- `containerViewModelMap:Pair<String, SessionInteractiveContainer>[]`

A `constructor()` szintén kiemelt fontosságú. A `'diagram_fetch'` esemény hatására a `connect()` függvény fut le, amely csatlakoztatja az **ActionSocket**-et. Ha az **ActionSocket** csatlakoztatása sikeres és van hozzáférési jogunk a kért diagramhoz, akkor kapunk egy `session_token`-t válaszként. Ennek a tokennek a felhasználásával második lépésként csatlakozhatunk a **SessionSocket**-el a szerver oldali **EditorSession**-hoz. A token hashelve tartalmazza a felhasználónevünket, illetve a munkamenet azonosítóját. Erről mellékelek egy szekvencia diagramot ([17. ábra](#)):

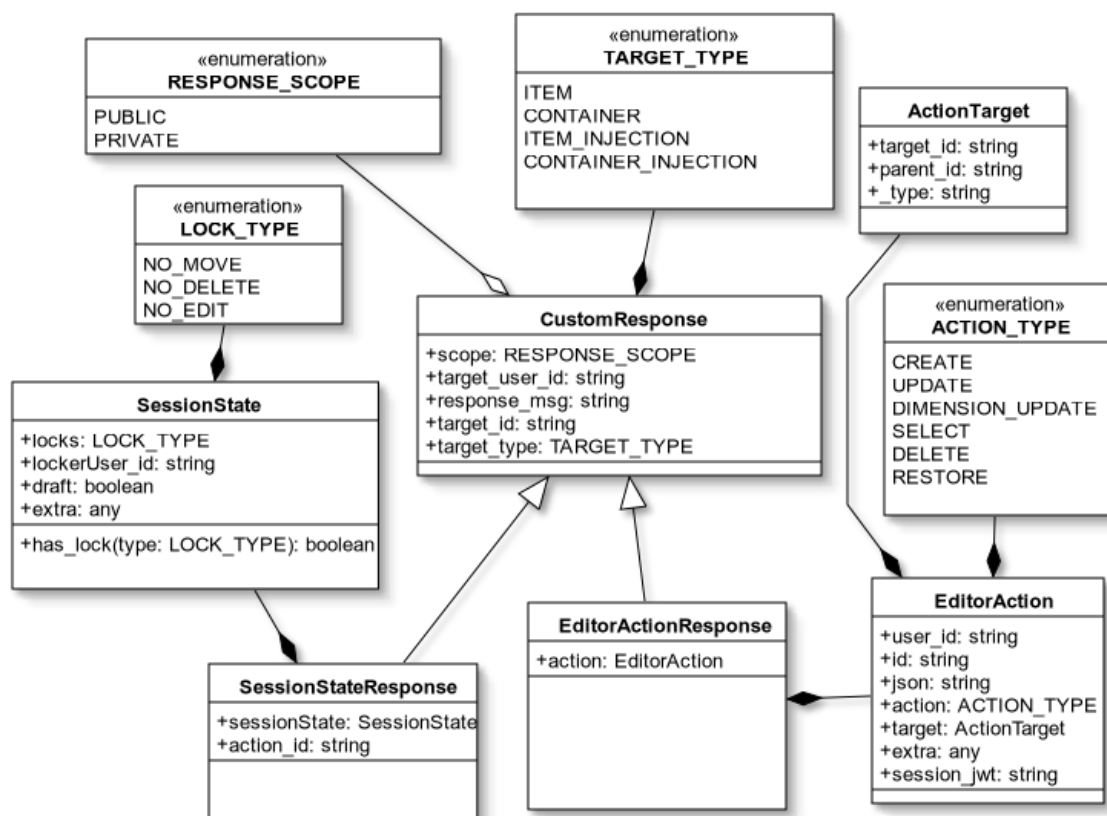


17. ábra: Kapcsolódás a socketekkel, azonosítás

A 'pre_setup' esemény hatására újra inicializálódik az összes socket státusza, törlődnek a regisztrált elemek, majd lecsatlakozik mind a 2 socket, amennyiben fenn állt aktív kapcsolat.

4.3.1.3.2.1 ActionSocket

Az interaktív szerkesztőfelület központi hídja ez az osztály. Implementálja a **SocketWrapper_I interface-t**, amelyben minden fontos függvény deklarálva van, amely a socketes interakcióhoz kellhet. A connect függvénnyel csatlakoztathatjuk a socketünket a paraméterben kapott címhez. A socket alapértelmezett függvényeit, mint például az onmessage(msg), onopen(msg) vagy onclose(msg), felüldefiniáljuk. Ha a socketes kézfogás és a http2.0-s protokollfejlesztés megtörtént, akkor az onopen()-ben elküldjük az egyedi autentikációs tokenünket, amelyet a bejelentkezéskor kaptunk és a cookie-k közé mentettünk. A folyamat többi részét fentebb már ismerttettem.



18. ábra: Response objektumok felépítése

Bármikor, amikor a szerver küld egy akciót, akkor az **onmessage(e:any)** függvény fut le. Az üzenetet a függvény egyből egy **EditorActionResponse** objektummá alakítja. A **switch**-ben négy féle **Akciótípust** különböztetünk meg:

- **ACTIONTYPE.UPDATE**
 - Valami módosítás történt, akkor az **EditorSocketControllerService**-től elkérjük a módosítandó objektum nézetmodelljét (id alapján), majd egyszerűen az `updateModel()`-t felparaméterezve meghívjuk.
- **ACTION_TYPE.RESTORE**
 - Az előbbihez hasonlóan járunk el, annyi különbséggel, hogy itt a `restoreModel()`-t hívjuk meg felparaméterezve. Ha az objektum törölve lett, amikor nem volt rá lakatunk, akkor a szülőobjektumot (container) megkérjük, hogy hozza létre újra a `response.json`-ból megalkotott komponenst.
- **ACTION_TYPE.CREATE**

- A létrehozandó objektumot a `response.json` adattagban találjuk meg, ezt javascript objektummá kell alakítani. Ha a létrehozó felhasználó kapja meg az üzenetet, akkor a régi, ideiglenes id-t ki kell cserélni a szerver oldali id-ra a már létrehozott item-ben. A régi id tárolására a `response.action.extra` objektum áll rendelkezésünkre, amelybe a szerver beletette `'old_id'` kulccsal a régi azonosítót. Ez alapján kikeresi a nézetmodell, majd frissít rajta. Ha nem mi vagyunk a tulajdonosok, akkor szimplán a szülő (`container`) objektumot kérjük ki a `response.action.target.parent_id` alapján az `EditorSocketControllerService`-től és rajta meghívjuk a `createItem()` függvényt felparaméterezve.

- **ACTION_TYPE.DELETE**

- Az objektumot id alapján elkérjük `EditorSocketControllerService`-től és a `deleteSelfFromParent()` függvényt meghívjuk rajta, aminek hatására a szülő container kitörli magából a törlendő gyermek objektumot.

4.3.1.3.2.2 SessionSocket

A [17. ábra](#) bemutatja a socketes kapcsolódás folyamatát az azonosítással együtt. Ennek a folyamatnak a második része a `SessionSocket` csatlakoztatása a szerverhez (`/state websocket` végpont). Ha ez megtörtént, akkor az `ActionSocket`hez hasonlóan itt is a `connect` függvény inicializálja a socketet illetve felülírja a socket legfontosabb handler függvényeit, amiket a javascript virtuális gép hívogat majd számunkra. Azonban itt a `this.socket.onmessage=this.oninitmessage;` paranccsal először egy ideiglenes handler függvénnyel írjuk felül az `onmessage(msg)` handler-t. Az `oninitmessage(msg)` csak az első üzenetet kezeli, amelyik ugyanis egy listát tartalmaz az összes objektum állapotáról (`state`). Az átvétel után elkéri az interaktív elemek listáját az `EditorSocketControllerService`-től, majd id alapján beinjektálja az összes elem állapotát. Ennek a műveletnek a hatására tűnnek el a „null” kifejezések a dobozok mellől. Az inicializálás után a `this.parent.socket.onmessage = this.parent.onmessage;` paranccsal átállítja a socket `onmessage(msg)` handler-jét az egyedi, általános handler-re. Az `onmessage(msg)`

4.3.1.4 CanvasBoxComponent

Ez a komponens implementálja a **SessionInteractiveContainer** interface-t, tehát ő is egy interaktív konténer, lényegében a Diagram adatmodell nézetmodelljének az egyik fele. A szerkesztői funkcionalitás nagy része ebben a komponensben van megvalósítva.

Néhány fontosabb függvény:

- `setup()`: Feliratkozik egy lambda függvénnyel a 'canvas_size_update' eseményre, amely bármilyen `DiagramObject` pozíciófrissítése esetén tüzel. A lambda frissíti a `Clip` dimenzióit, amely a szerkesztőfelület keretét adja.
- `zoom(e)`: A görgetési eseményeket kezeli le, módosítja a `clientModel` nagy részét.
- `onMouseMove(e)`, `onMouseUp(e)`, `onMouseDown(e)`: a szerkesztői logika és az ezekhez tartozó interakciók logikáit tartalmazzák.
- `cursorModel(e)`, `drawClassMode(e)`, `drawNoteMode(e)`: ezek a függvények a toolbox ablakban felsorolt műveletek implementációit tartalmazzák. A `drawClassMode(e)` és `drawNoteMode(e)` a sorok között elküldik a megrajzolt **DiagramObject**-et socketen keresztül a szervernek.
- `corrigateTargetClassPosition()`, `corrigateTargetClassDimension()`: az éppen megfogott doboz pozícióját illetve méretét a tervezőrácshoz igazítja.
- `repositionCanvas()`: amikor a vásznat mozgatjuk (egér lenyomása + mozgatás), akkor módosítja a canvas pozícióját.
- `corrigateCanvasPosition()`: A szerkesztővásznat visszaigazítja a keretbe (`Clip`), ha esetleg kihúznánk belőle.
- A **CanvasBoxComponent** a html template fájljában felsorolja a **DiagramObject**-ből származtatott dobozokat.

4.3.1.4.1 DiagramObjectComponent

Minden doboztípusnak külön komponensdefiniációja van. Az ősosztály megjelenítése a **DiagramObjectComponent**-ben történik, amely kiterjeszti a **SessionInteractivItemBase** bázisosztályt. Itt definiálva van minden olyan funkció, amelyre egy doboznak szüksége lehet. Érdeemes megnézni a html template sablont, különös tekintettel a 16-19 sorokra:

```
16 <div class="window-content">
17   <ng-container *ngTemplateOutlet="contentTemplate"></ng-container>
18 </div>
```

20. ábra: Template injekció az Őskomponens nézetébe

```
1 > <ng-template #contentc>...
23 </ng-template>
24 <app-diagram-object
25   [contentTemplate]="contentc"
26   [model]="this.model"
27   [general]="this.general"
28 >
29 </app-diagram-object>
30
```

21. ábra: Template injekció a másik, gyermek oldaláról nézve

A 20. ábra-n és a 21. ábra-n a **DiagramObjectComponent** és a **SimpleClassComponent** nézetének egymásba ágyazását láthatjuk. A gyermek komponens template kódjában meghívtam az ősosztály selectorját (21. ábra, 24-29-es sor), illetve beinjektáltam a gyermek komponens speciális dependenciáit ([model], [general]), illetve a gyermekosztály belső nézetét, amelyet #contentc azonosítóval láttam el. Ez több szempontból is jó nekünk. Egyrészt az ősosztály nézete és template kódja gondoskodik arról, hogy mozgatható, átméretezhető, kiválasztható legyen illetve socketes **UPDATE** akció esetén dimenziói frissüljenek. A 19. ábra-n ismertetett **DiagramObject** osztályból származtatott modellek mindegyikéről elmondhatók az előbb ismertetett tulajdonságok. Mindegyik komponensnek már csak a saját, specifikus funkcióit kell megvalósítani a fájljaikban.

4.3.1.4.2 SimpleClassComponent

SimpleClassComponent-ben a doboz tartalma egy címből (**AttributeComponent**), és csoportok felsorolásából (**AttributeGroupComponent**) áll. Az attribútumok megjelenítése a csoportkomponensekre vannak bízva. Az **AttributeComponent** univerzálisnak számít, ugyanis minden **Element_c** osztályból származtatott modellt ez a komponens jelenít meg.

4.3.1.4.3 NoteBoxComponent

A **NoteBoxComponent** nagyon egyszerűen csak egy szöveget jelenít meg a doboz belsejében, amely kattintásra szerkeszthető. A html kódban `<pre>...</pre>` tag-ek közé tettem a szöveget, hogy az pontosan úgy jelenjen meg, ahogy a felhasználó azt megírta.

4.3.1.4.4 PackageObjectComponent

Ez a komponens különleges abból a szempontból, hogy ez csupán meglévő információkat összesít más diagramokból, illetve a FileManager-ből. Úgy kell elképzelni mint egy nézetet. A cím nem szerkeszthető, mert egy meglévő projektmappát reprezentál. A cím alatt lévő sorok sem szerkeszthetők, mert azok a reprezentált projektmappa almappáinak neveit, illetve az abban definiált osztályokat tartalmazzák.

4.3.1.4.4.1 LineCanvasComponent

A szerkesztői funkciók maradék részei ebben a komponensben kaptak helyet. Implementálja a **SessionInteractiveContainer** interface-t, tehát ez is egy konténer. Miket tárolunk? Vonalakat. Ez a komponens a Vonalak megjelenítéséről gondoskodik. A megjelenítéshez [html5Canvas](#)[14]-t használok 2 dimenziós kontextusban, és a beépített `beginPath()`, `moveTo(x,y)`, `stroke()`, stb... funkciókra támaszkodva rajzolom a vonalakat. A komponens az előbb ismertetett **CanvasBoxComponent** alárendelt részét képezi, ezért a legtöbb felhasználói eseménykezelést (pl.: kattintás, egérmozgatás, billentyű leütés, stb...) a szülő komponens delegálja és továbbítja. Az angular speciális `@ViewChild('valami_id')` annotációját használva hivatkozhatunk bármely DOM elemre a html sablonban, ha azt `#valami_id` jelzéssel elláttuk. Jelen esetben a `#canvas`-ra hivatkozunk.

4.3.1.4.4.2 Néhány fontosabb függvény:

Az `init()` függvényben a `html5 Canvas`-tól elkérjük a `2d-s Context`-et. Itt különösen vigyáznom kellett arra, hogy a `context`-et csak akkor próbáljam elkérni, amikor a `cavas` már teljes mértékben inicializálódott. Erre is van az `angular`nak megoldása, mégpedig az `AfterViewInit` interface, amely csak az `ngAfterViewInit()` függvényt deklarálja. Ez a függvény akkor fog lefutni, miután a teljes `html` sablon legenerálódott. Ez nekünk pont kapóra jön, úgyhogy itt hívjuk meg az `init()`-et. Az `init` ezen felül még a **ResourceLoaderService** segítségével betölti a vonalvégződések `svg` képeit.

A `drawBegin(e,type)`, `drawMove(e)`, `drawEnd(e)` függvények új vonalak konkrét megrajzolásáért felelősek. A `drawEnd(e)` teszi fel az `i-re` a pontot, ugyanis itt jön létre a vonal saját nézetmodellje (`this.createLineWithControllerLocally (this.lineInstance)`) és kerül elküldésre (`this.sendLineCreated(this.lineInstance)`).

4.3.1.4.4.3 MathHelper

A matematikai számolásokért, ütközésérzékelésért, vektor metszéspont és egyebek számolásáért a **MathHelper** segédosztály felel.

4.3.1.4.4.4 LineController

Ezt az osztályt úgy kell elképzelni, mint egy `angular` komponens, csak nincs hozzá `html` `template`, sem `css` fájl. Implementálja az `InteractiveltemBase` interface-t ami felruházza őt minden olyan tulajdonsággal, hogy egy `session`-ban interaktívan lehessen használni. A vonalak a megjelenítésükért maguk felelnek. Ennek a kötelességnek a `drawLine()` függvény tesz eleget: Amennyiben nincsenek töréspontok (`BreakPoint`) a vonal 2 végpontja között, akkor a vásznon egy egyenes vonalat húz. Ha vannak töréspontok, akkor végig iterál a skálázott töréspontokon, (`breaks_scaled`) (mert van `zoom` funkciónk), és azok összekötésével alakul ki a `spline`.

4.3.1.5 EditorRootComponent

Ez az osztály lényegében a globális felhasználói eseményeket kapja el, illetve azok egy részét le is kezeli, pl.: törlés esemény, ha a `del` billentyűt lenyomjuk.

4.3.1.6 LeftPanelComponent

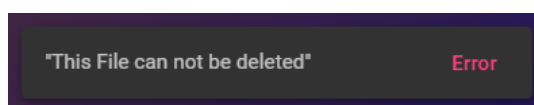
Ez a komponens tartalmazza a baloldali panelben elhelyezkedő fájlmenedzsert. A ts fájl tartalmazza a kattintás, illetve billentyűesemények kezelését és a szerver REST végpontjait hívogató függvények egy részét, ezek a következők:

- createProjectToActual(name: string)
- createFolderToActual(name: string)
- createProjectFolderToActual(name: string)
- deleteFile(id, _type)
- getFile(id, _type)
- getRootFolder()

Egy tipikus REST kérés a következőképpen néz ki:

```
this.http
    .get<FileResponse>(environment.api_url_http +... {
        headers: { 'Authorization': 'Bearer ' + getCookie("jwt_token")
    }
    })
    .pipe(catchError(this.handleError<FileResponse>(this, '...', ...
)))
    .subscribe((r) => { ... }
```

A fenti példában látszik, hogy **http GET** kérést indítunk az alapértelmezett **api_url**-re, az azonosításhoz a bejelentkezéskor kapott token mellékelni kell a kéréshez, különben **403-as Forbidden** http választ kapunk. A további azonosítás a token segítségével szerver oldalon történik, amely hashelve tartalmazza többek között a felhasználónevet is. A komponens esetében az összes http kérés során fellépő hibát a `handleError<T>(view: LeftPanelComponentComponent, operation = 'operation', result?: T)` függvény kezeli le, illetve egy **snackBar** segítségével közvetíti a **hibaüzeneteket** a felhasználó felé. Erre egy példa:



22. ábra: Példa a `snackBar` használatára

4.3.1.7 FileManagerService

Ez a szervíz néhány **http REST** kérést intéző függvényt tartalmaz, amelyet közösen használ a **FileComponent** és a **LeftPanelComponent**. Itt nem történik hibakezelés, a függvények a nyers válaszokat adják vissza.

4.3.1.8 FileComponent

Ez a komponens a fájlok megjelenítésére szolgál. A kapott fájl típusa és az **ICON** enumeráció értékétől függően dönti el a megjelenítés formáját.

4.3.1.8.1 A fájlok megjelenítése

A típusok a következők lehetnek:

- FolderDto
- folder
- project
- ProjectFolderDto
- projectFileDto
- ProjectFile
- ProjectFolder

Az osztálytípust a generált JSON szövegbe futási időben a `_type` mezőbe írja a [Jackson](#)[16] könyvtár csomag. Ezt az összes fájl típus őszinterfészében definiáltam a `@JsonTypeInfo` és `@JsonSubTypes` annotációk segítségével. Ha az objektumokat a **Jackson** saját **ObjectMapper**jével konvertáljuk JSON formátumba, akkor az annotációk figyelembevételével dolgozik.

A megjelenítés mellett némi eseménykezelés is helyet kapott a komponensben, mivel előfordulhat, hogy a **FileComponent**-ek beágyazott módon helyezkednek el egymásban, ezért kiemelkedően fontos, hogy tudjuk melyik gyermek komponens indította a hívást és mit szeretne.

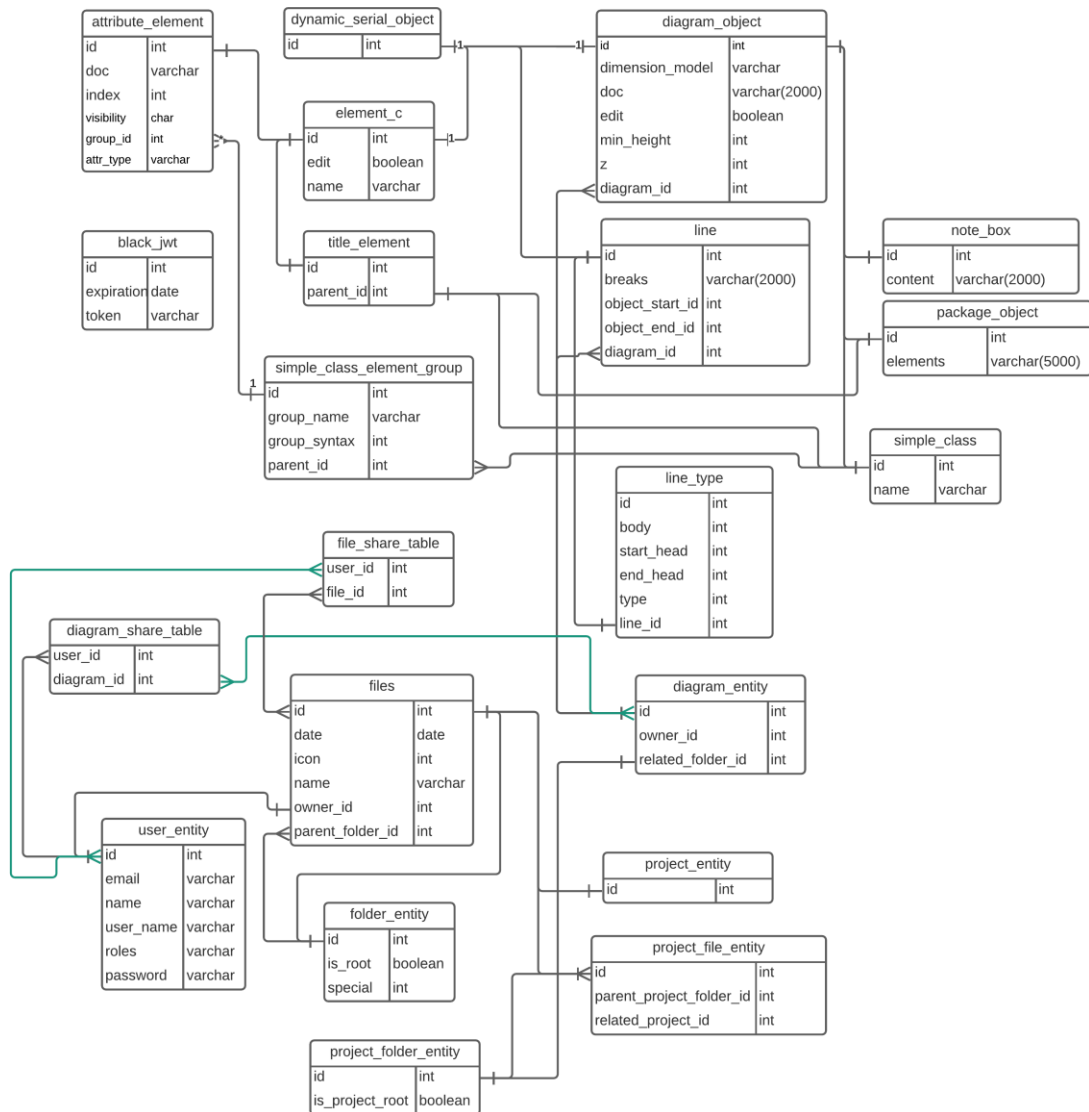
4.3.1.8.2 ShareDialogComponent

Ez a kis komponens a FileComponent mellett kapott helyet, mivel másol nem használtam fel. A nézetmodelljét szintén a file.component.ts fájlban lehet megtalálni, a nézet külön html-template-t kapott: ./dialog-share.html.

4.3.2 Szerver oldal

4.3.2.1 Adatmodellek

A [19. ábra-n](#) már bemutattam az adatmodellek nagy részét. A back-end rétegen az adatmodellek kis mértékben eltérnek az ott ismertettektől. Minimális eltérések a perzisztencia réteg igényei miatt keletkeztek, ez a legtöbb esetben azt jelenti, hogy a gyermek objektum mindig hordoz egy referenciát a szülő objektumra. Ez azért szükséges, hogy a JPA könyvtárcsomag el tudja készíteni az adattáblákat az annotációkkal kiegészített Java osztályokból (Entitásokból). Az alábbi, [23. ábra-n](#) a JPA által létrehozott adattáblákat és azok kapcsolatait láthatjuk.



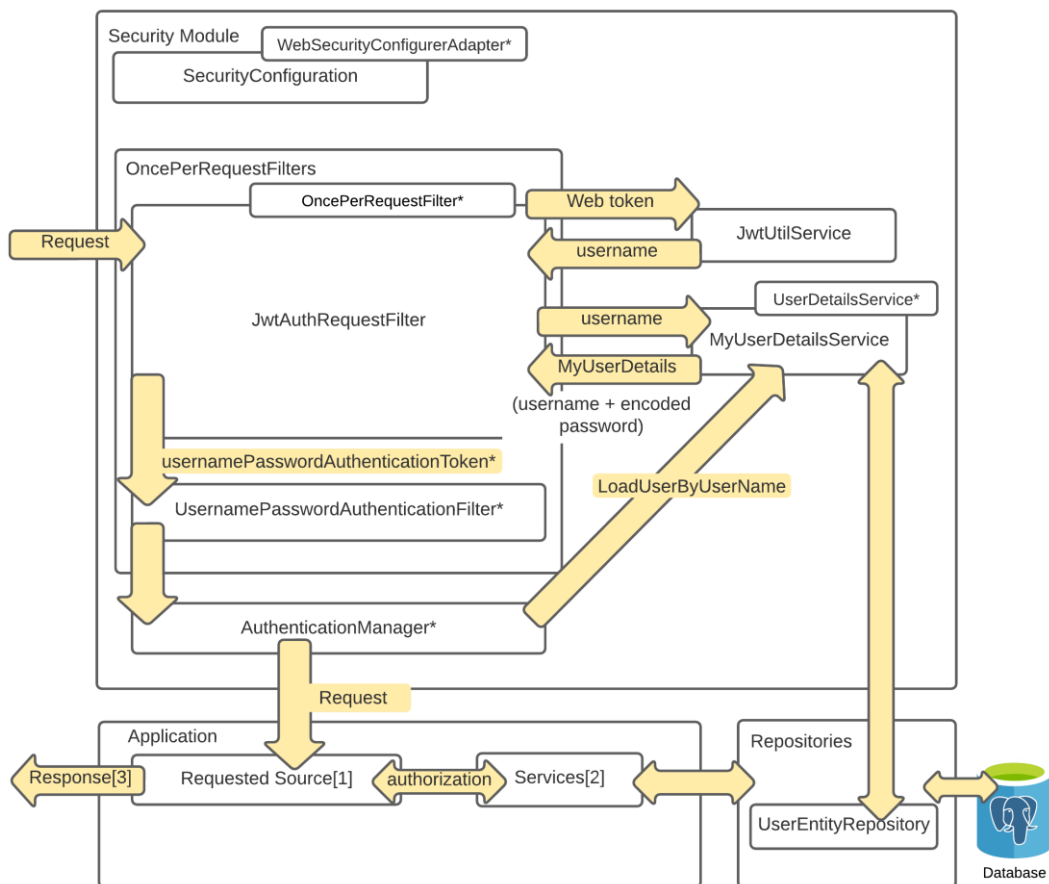
23. ábra: adatbázis táblák diagramja

4.3.2.2 Az alkalmazás rétegei.

A szerver több modulból tevődik össze. Ezek az **editor**, **management** és **security**. Minden modulról elmondható, hogy állnak egy **controller** csomagból, ahol a modulhoz tartozó végpontok lekezelése történik. Egy **model** csomagból, ahol az adatmodellek definíciói találhatóak. Egy **service** csomagból, ahol az üzleti logika található. Illetve egy **repository** csomagból, ahol az egyes adatmodellekhez tartozó adatbázis műveletek vannak definiálva. Az **editor** és **management** modulok sok helyen egymásra hivatkoznak, kicserélésük nem ajánlott, mert inkonzisztens állapotot idézhet elő.

4.3.2.3 Security

Először a **Security** modullal kezdem, mert kiemelten fontosnak tartom a felhasználói adatok és tartalmak védését. a **Security** réteg a `com.szefi.uml_conference.security` csomagban található. Itt rengeteg alcsomagot találhatunk. A **config** csomagban a **SecurityConfiguration** osztály, azaz security réteg konfigurációja található. Itt a `configure` metódusokban beállítjuk a végpontok hozzáférési szabályzatát. A **publikus** végpontok: `/login`, `/`, `/state`, `/action`. Csak felhasználók számára elérhető végpontok: `/get/**`, `/log_me_out`, `/management/**`, `/project_management/**`. Az autorizálást a **JwtAuthRequestFilter** végzi. A Security réteg működésének működéséről készítettem egy „high-level” ábrát. (24. ábra)



24. ábra: Security réteg működése

A 24. ábra egy átlagos HTTP REST kérés útját mutatja be az alkalmazásban. A „*”-al jelölt Osztályok, modulok a Spring keretrendszer *Security csomagjának*[15] részei. Az ábrán A „Request” nyíl jelzi a HTTP kérés beérkezésének helyét. A kérést legelőször a

[4.3.2.3.1 JwtAuthRequestFilter](#) osztály nézi meg. A filter áthív a [4.3.2.3.2 JwtUtilService](#)-be, hogy az autentikációs tokenből kisedje a felhasználónevet. Ha ez megtörtént, akkor a [4.3.2.3.3 MyUserDetailsService](#)-től elkérjük a felhasználóhoz tartozó MyUserDetails objektumot, amely már többek között a titkosított jelszót is tartalmazza. A szabványos felhasználónév és jelszó ellenőrzést a Spring keretrendszer teszi meg az **UsernamePasswordAuthenticationFilter** és **AuthenticationManager** osztályokkal, itt a felhasználó jogainak ellenőrzése megtörténik a saját, általam implementált MyUserDetailsService service segítségével. Ezt a SecurityConfiguration osztályban állítottam be. A felhasználó elbírálásra kerül a (Spring tekintetében szabványos) adatai alapján, hogy van-e joga a kért végpontot elérni. Ha hozzáférhet a kért tartalomhoz, akkor továbbengedi a kérést, ha nincs akkor **403-as Forbidden** választ továbbít a következő filternek. Ha van még filter a láncban, akkor azokon is végigmegy a kérés. Ha minden filteren átment, akkor végül a kért végpontra jut. Ne feledjük, ez csak az alapvető autentikáció. Az egyedi autorizáció a végpontokból hívott szervizekben történik, ezt a [24. ábra](#) alsó részén láthatjuk. A hívások sorrendjét szögletes zárójelek között tüntettem fel.

[4.3.2.3.1 JwtAuthRequestFilter](#)

Ez a filter minden egyes rest hívást megállít, és belenéz a header állományba. Ha abban található egy „Authorization” kulcs „Bearer xxx” értékkel, akkor a következő lépés az, hogy az xxx tokenből kicsomagoljuk a felhasználónevet a JwtUtilService segítségével, majd a **UserDetailsService** az exportált username alapján kikeresi a felhasználót az adatbázisból. Ha van joga az adott forrást elérni (jelenleg csak USER szerepkör van), akkor a kérést tovább engedi a filter. Ellenkező esetben **403-as Forbidden** hibával válaszol a kérésre.

[4.3.2.3.2 JwtUtilService](#)

Ez egy segéd szervíz a security rétegben. Feladata, hogy minden, tokenekkel kapcsolatos műveletet elvégezzen.

Az **extractUsername(String)** és **extractExpiration(String)** függvények a paraméterben megadott tokenbe csomagolt **felhasználónevet**, illetve **lejárat dátumot** bontják ki és adják vissza számunkra.

A **createToken(Map<String, Object>, String)** függvény a legfontosabb az egész osztályban. Ennek a meghívásával állíthatunk elő hiteles webtokeneket. A claims paraméter egy String kulcsokkal ellátott Object Map. A subject annak felhasználónak a felhasználóneve, akinek készítjük a tokenet. A claims Map-ba bármilyen Serializable Objektumot betehetünk és azt az algoritmus beleírja a tokenbe. A felsorolt 2 paraméter mellett a tokenbe belekerül a lejárat dátum, illetve a végére egy **aláírás HS256-os algoritmussal**, amelynek titkosító kulcsa csak szimplán be van égetve a SECRET_KEY privát változóba az egyszerűség kedvéért.

A **blackListToken(String)** függvény a paraméterben megadott tokent feketelistára helyezi az adatbázisban, azaz többé már nem használható azonosításra. A feketelistázott tokenek a lejárat dátumot követően egy **adatbázistrigger** segítségével **automatikusan törlődnek**.

4.3.2.3.3 MyUserDetailsService

Ennek a service-nek a dolga nagyon egyszerű. Az adatbázissal ő kommunikál, ha bármilyen felhasználói fiókokkal kapcsolatos műveletet szeretnénk elvégezni. Itt lehet a regisztrációt véglegesíteni, illetve a bejelentkezéskor a felhasználó adatait lekérni.

4.3.2.3.4 JwtAuthRequestHandlerService

Ez a szervíz már a kontroller réteg alatti legfelső autorizációs osztály.

A **jwtAuth** függvény a bejelentkezési kérelmet dolgozza fel. Ellenőrzi a beírt adatok formai helyességét, majd az **AuthenticationManager** segítségével azonosítja a kérdésben megfogalmazott felhasználót. A háttérben az **AuthenticationManager** többek között a **JwtAuthRequestFilter** osztálynak a **doFilter(...)** metódusát is meghívja, amely a tényleges autorizációt végzi. Ha az azonosítás sikeres, akkor gyártunk a felhasználónak egy tokenet amellyel 8 órán keresztül azonosíthatja magát. Ezt a tokenet egy az egyben visszaküldjük a kliensnek.

A **register(...)** függvény a regisztrációs kéréseket dolgozza fel. Némi formai validáció után a **UserDetailsService** közreműködésével végrehajtja a regisztrációt.

4.3.2.3.5 AuthController

A **/login**, **/register** és **/log_me_out** végpontokra érkező kéréseket itt kezeljük le.

4.3.2.4 Management és ProjectManagement

Ez az összetett csomagmodul a mappák, projektek menedzseléséért és ezek megosztásáért felel. A funkcionalitás két kisebb egységre bomlik: **ProjectManagement** és **Management**. Egy összetett csomagmodulban kaptak helyet, mert működésük szorosan összefonódik.

4.3.2.4.1 Management

4.3.2.4.1.1 ManagementController

A **ManagementController** fogad minden olyan kérést, amely a közönséges mappák létrehozására, törlésére és azok megosztására vonatkozik. Minden végpont autorizált illetve autentikált. Az autentikációt már a **ManagementService** végzi a **MyUserDetailsService** közreműködésével.

4.3.2.4.1.2 ManagementService

Ez a service a ManagementController-t szolgálja ki. Az elején kiemelem, hogy minden publikus, kontrollerből hívott függvény végez token validációt, illetve jogosultság ellenőrzést a kért tartalomhoz.

A **getUserRootFolder** függvény az Authorization header-ben található token alapján megkeresi a felhasználót, majd annak gyökérmappáját és egy FileResponse objektumba csomagolja azt.

A **createFolder** függvény a megadott parent_id alapján megkeresi a szülő mappát. Ha a kérést indító felhasználó a tulajdonosa a mappának, akkor a szülő .files listájához hozzáfűz egy új mappát a name paraméternek megfelelő elnevezéssel. Az új fájl örökli a szülő megosztási szabályait. A módosítások azonnal mentésre kerülnek az adatbázisban.

A **shareFile függvény** működése összetett. Egy ShareFileRequest típusú paraméterben megkapunk minden információt, ami a megosztáshoz kell. Az auth_jwt a megosztó felhasználó tokenje, a file_id a megosztandó fájl azonosítója, a target_UserName a másik felhasználó felhasználóneve. Miután megtaláltuk a fájlt, a saját felhasználónk entitását, illetve a másik felhasználó fiókjának entitását, annyi a dolgunk, hogy a fájl saját usersIamSaredWith listájába beszúrjuk a másik felhasználó entitását, illetve a másik felhasználó fiókjának sharedFilesWithMe listájába beszúrjuk a fájl entitását. Majd, ezt a lépést rekurzívan folytatjuk (updateShareRecursively függvény), amennyiben mappát osztunk meg vagy a rekurzió során mappába ütközünk. Projektek esetében nem kell törődni a beágyazott projektmappákkal, mert azok a projekt entitásának hozzáférési jogaira hivatkoznak.

4.3.2.4.2 ProjectManagement

Ez a modul minden, projektekkel kapcsolatos interakció kezeléséért felel.

4.3.2.4.2.1 ProjectManagementController

Ennek a kontrollernek a működése nagyon hasonló a ManagementController-hez, annyi különbséggel, hogy itt a kontroller mögötti logikát a ProjectManagementService adja.

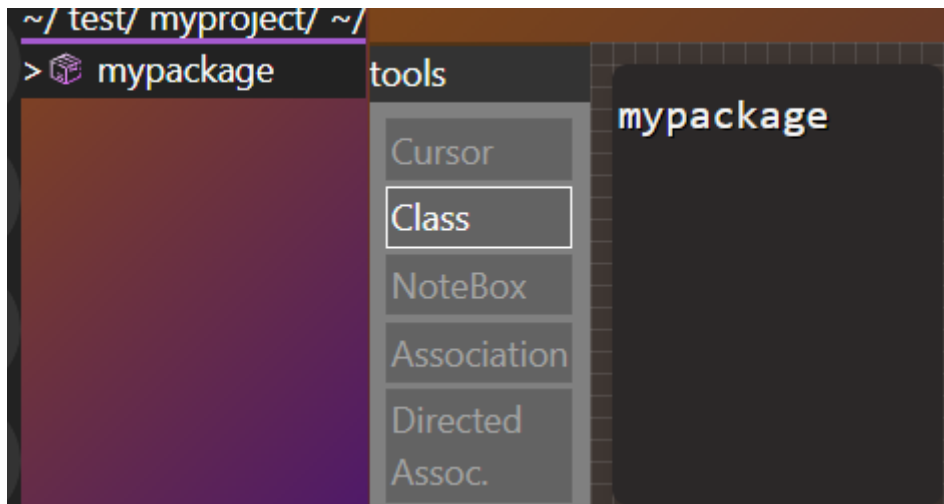
4.3.2.4.2.2 ProjectManagementService

Ez a szervíz a következő feladatokat képes ellátni: Projekt létrehozása, projektmappa létrehozása, Projektmappa lekérése, projektmappa törlése. A függvények közül hármat emelnék ki.

A **createProject függvény** projektek létrehozását végzi. A ProjectEntity konstruktor hívásánál automatikusan létrejön egy gyökér-projektmappa, ehhez automatikusan létrejön egy DiagramEntity is.

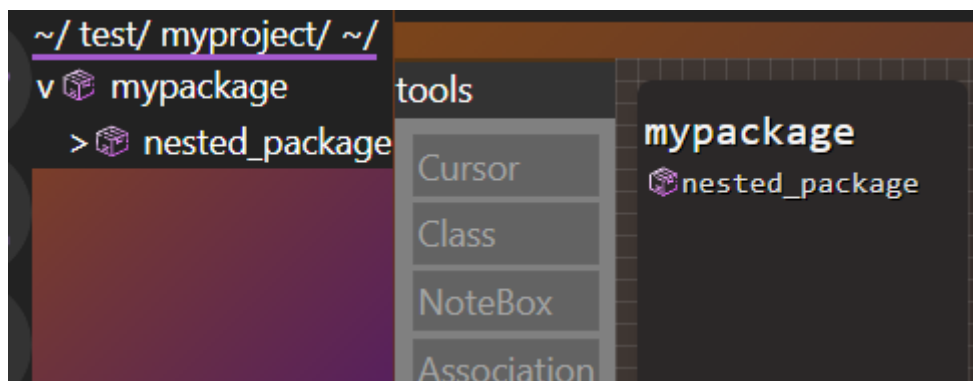
A createProjectFolder függvény a legösszetettebb az osztályban. A szokásos azonosítások után egy ProjectFolderEntity-t hozunk létre a paraméterben megadott névvel, és hozzáadjuk a szülő mappához, amelyet a parent_id alapján kerestünk ki. Az érdekesség ott van, hogy egy DiagramEntity-t is létrehozunk a

mappához és összekötjük őket. Majd az **injectPackageObjectToParentDiagram** függvény segítségével a szülő mappa Diagramjába tesszük PackageObject formájában.



25. ábra: A szülő diagramjában létrejön az új mappa(mypackage) egy PackageObject doboz formájában

A függvény végét szintén kiemelem, ugyanis ha nem a projekt gyökérben vagyunk, akkor a szülő mappa linkelt Diagramjában lévő **PackageObject**-et megkeressük és annak az elemei közé besúrjuk a friss projektmappa nevét, egy kis csomag ikonnal (25. ábra).



26. ábra: A nagyszülő mappa(~) diagramjában lévő szülő mappát (mypackage) reprezentáló PackageObject elemei közé bekerül az újonnan létrehozott mappa (nested_package)

Ha van aktív session a nagyszülő diagramján, akkor a kapcsolódott kliensek részére automatikusan elküldésre kerülnek a módosítások, hogy az állapot a lehető legkonzisztensebb legyen.

4.3.2.5 Editor

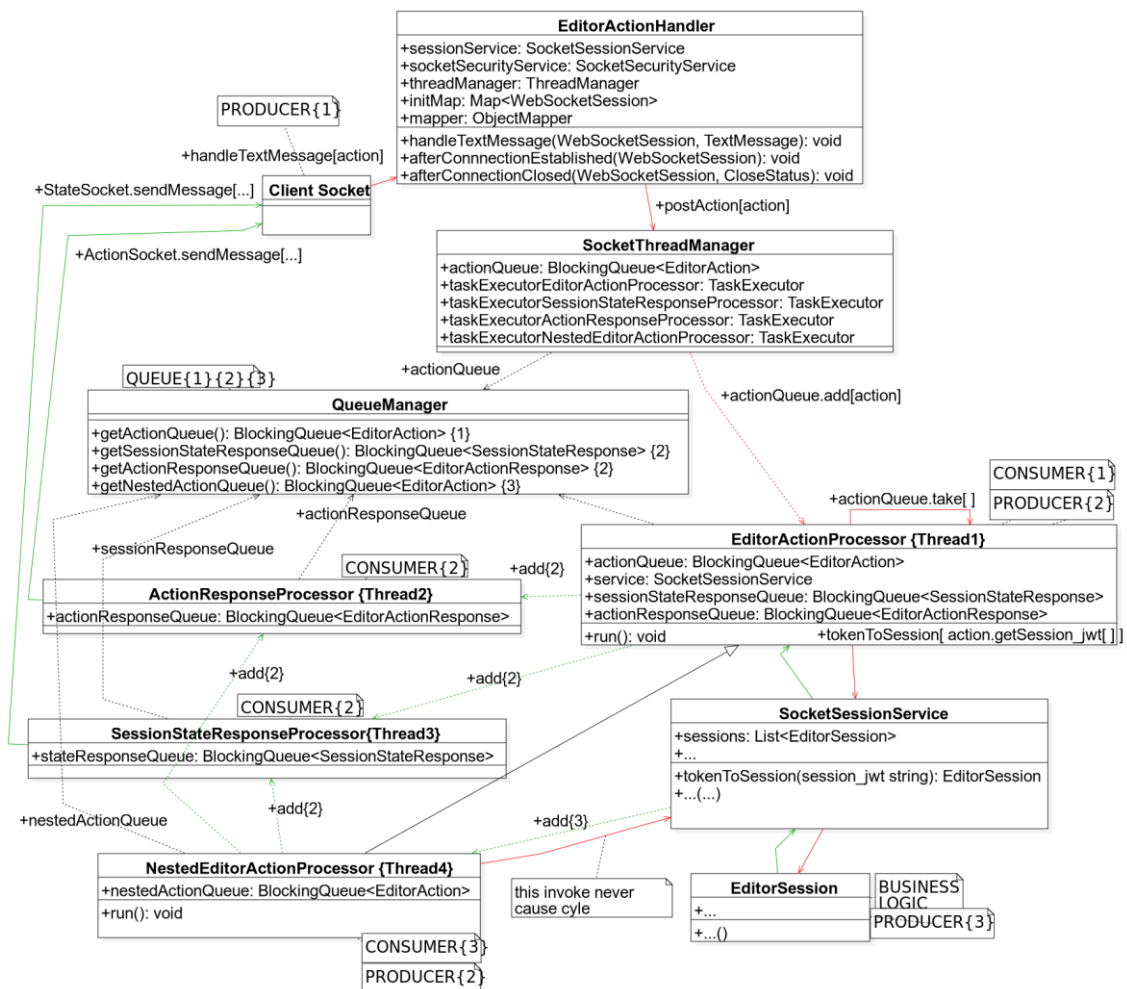
Az alkalmazás legfontosabb modulja, a legtöbb munkaidőt ennek a résznek a tökéletesítése tette ki. Itt helyezkedik el a szerkesztői munkamenetek logikája és a socket-es kapcsolódás.

4.3.2.5.1 WebSocketConfig

Itt a socketek konfigurációja történik. Ezt a `registerWebSocketHandlers` függvény teszi meg. Felhívnom a figyelmet arra, hogy az osztály `@Configuration` annotációval van ellátva, ez azzal a képességgel ruházza fel őt, hogy képes objektumokat létrehozni az `ApplicationContext`-ben. Ezek a függvények `@Bean` annotációval vannak ellátva. A `@Scope` alapján a spring intelligens módon eldönti, hogy kell-e új objektumot gyártani a függvényhívással. Ha scope alapján nem kell, akkor az `ApplicationContext`-ből kikeresi a már létező objektumot. Az alapértelmezett scope a „singleton”, amely azt jelenti, hogy egyetlen példányt hoz létre a spring az alkalmazás indulásakor. A `getNewTaskExecutor()` függvény „prototype” hatókörű, ez azt jelenti, hogy minden függvényhívásnál új objektumot készít, ad vissza és ír be az `ApplicationContext`-be a spring keretrendszer. A hatáskörökről bővebben a [spring dokumentációjának ide tartozó részében](https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html)² olvashat.

² <https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html>

4.3.2.5.2 Az Editor réteg működése



27. ábra: Az alkalmazás tripla producer-consumer architektúrájának diagramja.

- Minden Fogyasztót(**Consumer**), termelőt(**Producer**), illetve szállítóközeget (**Queue**) {sorszámokkal} indexeltem, hogy jól kivehető legyen, ki melyik csoportba tartozik.
- Az ábrán fekete, zöld, illetve piros vonalakat láthatunk. Ezeknek különleges jelentésük van:
 - A piros vonalak az üzleti logika felé irányuló üzeneteket jelentik.
 - A zöld vonalak az üzleti logikától kifelé menő üzeneteket, hívásokat jelölik.
- Az ábrán a piros és zöld szaggatott vonalak a szállítóközegen keresztüli hívásokat jelölik.

Először a Kliens oldali socket elküldi az akciót a szervernek. Ezt az EditorActionHandler handleTextMessage függvénye fogadja. Az akciót json formátumból Java objektummá alakítja a [jackson könyvtárcsomag](#)[16] ObjectMapper-je, majd a ThreadManager-nek adja át a postAction(action) hívással. A ThreadManager aztán beteszi az action-t az ActionQueue{1}-ba, eközben az 1-es szálon az EditorActionProcessor {C1,P2} folyamatosan figyeli az ActionQueue{Q1}-t. Az EditorActionProcessor {C1,P2} kiolvassa az action-t az ActionQueue{Q1}-ből a take() utasítással. Az akció típusa szerint egy switch ágra fut a vezérlés. Itt az akció session_jwt adattagjának segítségével megállapítjuk, hogy melyik EditorSession-nál kell folytatni a hívási láncot. A megfelelő session megállapításában közreműködik a jwtUtilService, illetve a SocketSessionService. Miután az üzleti logika kifejtette hatását, az EditorActionProcessor{C1,P2} válasz objektumokat készít, azokat felteszi a sessionStateResponseQueue{Q2}-ra vagy az actionResponseQueue{Q2}-ra. Az előbb említett 2 blokkoló sor tartalmát folyamatosan figyeli a külön szálon futó ActionResponseProcessor{C2} és SessionStateResponseProcessor{C2}. Ez a két osztály az EditorActionResponse vagy a SessionStateResponse szabályai alapján továbbítja a választ a megfelelő socket végpontok felé.

Vannak különleges esetek, amikor kliens oldali hívás nélkül kell a szerver oldalról akciót küldeni a kliensek felé. Az ilyesfajta speciális akciókhoz egy külön sort; NestedEditorActionQueue{Q3} illetve egy külön feldolgozó osztályt; NestedEditorActionProcessor{P2,C3} (továbbiakban Processor) hoztam létre. Erre egy remek példa az az eset, amikor egy dobozt kitörlünk, de ahhoz vonalak vannak csatlakoztatva. Ebben az esetben minden, a dobozhoz kapcsolódó vonalhoz készítünk szerver oldalon egy DELETE EditorAction-t és betesszük azt a NestedEditorActionQueue{Q3}-ba. Processor{P2,C3} kiolvassa a akciót, és mint egy sima vonal törlés esetén, elvégzi az üzleti logika a törlést, majd a Processor{P2,C3} beteszi a válaszbjektumot valamelyik {Q2}-es szállítóba. A többi már csak történelem.

4.3.2.5.2.1 SocketThreadManager és QueueManager

Ezek az osztályok a producer-consumer architektúra fontos részét képezik. A **QueueManager** egy konfigurációs osztály, azaz képes arra, hogy az `ApplicationContext`-ből objektumokat vegyen ki. Itt gettereket találunk nevesített `@Bean` annotációkkal. Mivel nem találunk `@Scope` annotációt, ezért alapértelmezett módon induláskor jönnek létre a közös várakozási sorok(`Queue`).

4.3.2.5.2.2 EditorActionHandler és SessionStateHandler

Mindkét osztály kiterjeszti a `TextWebSocketHandler`-t.

A legfontosabb felülírandó függvények:

- `afterConnectionEstablished(WebSocketSession)`
- `handleTextMessage(WebSocketSession, TextMessage)`
- `afterConnectionClosed(WebSocketSession, CloseStatus)`

A socket működéséről annyit emelnék ki, hogy a működése nagyon egyszerű; az információt bináris vagy szöveges formában lehet küldeni. Én csak az utóbbit használtam a program elkészítése során.

4.3.2.5.2.2.1 EditorActionHandler

Ez az osztály fogadja a `/action` végpontra érkező socket kapcsolódásokat. Itt történik az első fázisú azonosítás, amikor eldöntjük a `SocketSecurityService` segítségével, hogy az illető hozzáférhet-e a megadott diagramhoz. Ha igen, akkor továbbítjuk a kérést, a socketet, illetve a felhasználó adatait a **SocketSessionService**-nek. Az `autoProcessRequest` metódus feldolgozza az előbb leírt paraméterek alapján a kérést és beilleszti a felhasználót sockettel együtt a diagramhoz tartozó `EditorSession`-ba. Ha nincs ilyen munkamenet, akkor létrehoz egy frisset. Visszatérési értékként egy `session_jwt` tokenet ad vissza (benne van a session azonosítója is), amelyet a kliens megkap, mint kulcsot a `/socket` végponton figyelő `SessionStateHandler`-hez.

4.3.2.5.2.2.2 SessionStateHandler

Ehhez a sockethez a fent leírtak alapján egy `session_jwt` kulcs-al tudunk kapcsolódni. Ha a kapcsolódás sikeres, akkor a megfelelő `EditorSession`-tól elkéri a

handler a `sessionItemMap`-ot. Ebben a map-ben van az összes objektum állapota, amellyel kölcsönhatásba léphetünk a szerkesztőfelületen. Ezen a map-en végigmegyünk, listába fűzzük és válaszként elküldjük a kliensnek. Amikor a kliens megkapja a listát, azt kiolvassa, megvárja még minden elemet létrehoz az Angular. Ha készen vannak az előbbi lépések, onnantól számít üzemkésznek a munkamenet.

Lecsatlakozás esetén a **SessionStateHandler** lesz az, aki a natív socket alapján megkeresi a felhasználót reprezentáló `UserWebSocketWrapper`-t és `EditorSession`-t, majd felfűz egy csak szerver oldalon küldhető `S_USER_DISCONNECT` akciót, az `ActionQueue{Q1}`-re `EditorActionProcessor` kiolvassa, értesíti a releváns `EditorSession`-t a kilépésről, majd a feloldott objektumok állapotait(state) továbbítja a kliensek felé a már ismertetett producer-consumer architektúrának megfelelően.

4.4 Tesztelés

A fejlesztés ideje alatt legfőképp manuális teszteléssel ellenőriztem a program helyes működését. Ennek részeként a szerver végpontok teszteléséhez a [Postman](#)[17] nevű programot használtam. Amikor már a kliensoldal megbízhatóan működött, annak segítségével, a felhasználó szemszögéből végeztem a tesztelést. Az adatbázis változásait az [H2 adatbázis](#)[18] konzoljából követtem. Ha refaktorálás történt valahol, akkor arra különös tekintettel voltam.

A szerver oldalon a tesztesetekhez egy külön konfigurációs fájlt használok (`src/test/resources/application-test.properties`). Az alapértelmezett, `application.properties` konfigurációs fájljal ellentétben, ebben van megfogalmazva, hogy a spring boot egy H2-es adatbázist inicializáljon. Ez egy virtuális, memóriában tárolt adatbázis, amely az alapvető műveletekkel rendelkezik. *„Egy gyors prototípus megépítéséhez tökéletes”*[19]. Ez azért nagyszerű, mert teljesen izolált adatbázisban futtathatjuk a teszteseteinket, így a PostgreSQL szerveren tárolt meglévő adataink véletlenül sem fognak megsérülni.

4.4.1 Unit tesztelés

A programnak kevés olyan része van, amely izoláltan végez el feladatot. Ez annak a következménye, hogy a programban rengeteg az azonnali mentés, frissítés, törlés, valamint a manager és az editor modul szorosan összedolgoznak sok esetben. Ezért az adatok a szerver és az adatbázis között állandóan konzisztensek. Ez megnehezíti az egységtesztelést, mivel rengeteg függőséget kell mockolni. A szerkesztőfelület fő logikája az **EditorSession** osztályban van. Itt a zárolási mechanizmust alaposan le tudtam tesztelni unit tesztek segítségével. Néhány példa erejéig a [Mockito](#)[20] keretrendszert használtam a mockoláshoz. A probléma a bonyolultságának szemléltetésére egy jó példa a `deleteSimpleClass_Test()` tesztmetódus, amelyben 35 sornyi előkészület után a `SimpleClass` típusú doboz törlését tesztelem. Az előkészület azért ilyen hosszadalmas, mert az adatmodell hivatkozásait az valódi futás esetén több modul együttesen építi fel.

4.4.2 Integrációs tesztelés

4.4.2.1 Szerveroldal

A bonyolultabb, sok adatbázis műveletet igénylő metódusok tesztelését, illetve a management modul helyes működését inkább az integrációs tesztmódszerrel ellenőriztem. A `ManagementController` és `ProjectManagementController` teszteléséhez a [MockMvc](#)[21] tesztkönyvtárt használtam. A `MockMvc` segítségével könnyedén küldhetünk REST kéréseket tetszőleges végpontokra. A tesztelést nagyban megnehezítette az, hogy az adatbázissal kommunikáló Repository-k tranzakciókezelése egy kicsit eltér a normális manuális teszteléshez viszonyítva. A beszúrások, törlések nem azonnal történnek meg, ezért amikor egy tesztmetóduson belül több olyan végpontot is hívunk, amely dml műveletet végez, akkor az elvégzett módosításokat közvetlen Repository injekcióval, ugyanabban a tesztmetódusban nem lehet biztonságosan ellenőrizni.

A szerver oldal tesztadatokkal történő izolált teszteléséhez létrehoztam egy `dev` nevű profilt: Ha szervert „dev” (fejlesztői) módban szeretnénk indítani tiszta, memóriában futó adatbázissal, illetve két tesztfelhasználóval, akkor a

src/main/resources/application.properties fájlban állítsuk a `spring.profiles.active` változó értékét `dev`-re! Ez a `dev` profilt fogja aktiválni a következő szerver indításnál, ezáltal az `application-dev.properties` konfigurációs fájl szabályainak megfelelően. Figyeljünk arra, ha a `target` mappából futtatjuk az alkalmazást, akkor a `classes` mappában találjuk meg a konfigurációs fájlokat. A kliens oldal socketes funktionalitásának felület-teszteléséhez előfeltétel, hogy a szerver `dev` módban fusson.

4.4.2.2 Kliensoldal

A kliens oldal tesztelését a [Selenium](#)[22] keretrendszer segítségével automatizáltam. A Selenium egy nyílt forráskódú tesztelőkönyvtár, amely a böngésző tesztelő illesztőprogramjának segítségével lép kölcsönhatásba a weboldallal. Szinte bármilyen információ elkérhető a html oldalról. A tesztprogram elkészítése során sok hasznos, új technológiával ismerkedtem meg, amelyekre biztos, hogy szükségem lesz később a szakmában, ezek közül néhány példa: ActionBuilder, XPATH, WebDriver, WebElement, TestSuite.

Ha teljes **integrációs felület tesztet** szeretnénk csinálni, akkor a szervert `dev` profillal indítsuk el, a [4.4.2.1 Szerveroldal](#) bekezdés utasításai alapján. Futtassuk le a `Selenium-ClientTest` tesztjeit! Ez a Java alkalmazás az összes integrációs tesztet lefuttatja a felhasználói felületen; kivétel nélkül, a teljes funktionalitást ellenőrizve.

4.4.2.3 Tesztesetek

**manual* -al jelölt teszteseteket csak manuálisan lehet végrehajtani, ezekhez az esetekhez nem készítettem automata tesztprogramot, mert az eredmények ellenőrzése technikai határokba ütközik.

4.4.2.3.1 Regisztráció




- Leírás: Egy új felhasználói fiókot hozunk létre
- Előfeltétel: Meg van nyitva a böngészőben a felhasználói felület, nincs bejelentkezve senki.
- Tesztlépések:

- Kattintsunk a **Register** linkre a fejlécen
- Töltsük ki az űrlapot helyes adatokkal.
- Kattintsunk a Register gombra.
- Várt eredmény: a képernyő alján a program visszajelzést ad számunkra a regisztráció sikerességéről; „**Registration for test was succesful**”

4.4.2.3.2 Bejelentkezés

- Leírás: Meglévő fiók adataival belépünk az alkalmazásba
- Előfeltétel: Meg van nyitva a böngészőben a felhasználói felület, nincs bejelentkezve senki.
- Tesztlépések:
 - Kattintsunk a Login linkre a fejlécen!
 - Töltsük ki a Username és Password mezőket!
 - Kattintsunk a Login gombra!
- Várt eredmény: A képernyő alján a program visszajelzést ad számunkra a regisztráció sikerességéről; „**Login successful**”



4.4.2.3.3 Új mappa létrehozása

- Leírás: egy új mappát fogunk létrehozni a felhasználó gyökérkönyvtárában (~/)
- Előfeltétel: Be vagyunk jelentkezve.
- Tesztlépések:
 - Kattintsunk a bal oldali menüsávban a mappa ikonra! 
 - kattintsunk az eszköztárban a mappa létrehozása ikonra! 
 - A mappa lista végén megjelenő beviteli mezőbe írunk be azt, hogy „teszt”!
 - Kattintsunk valahová, vagy nyomjunk Enter-t!
- Várt eredmény: Új sor jelenik meg a fájl lista végén mappa ikonnal.  **teszt**

4.4.2.3.4 Új projekt létrehozása

- Leírás: egy új projektet fogunk létrehozni a felhasználó gyökérkönyvtárában (~/)
- Előfeltétel: Be vagyunk jelentkezve.



- Tesztlépések:

- Kattintsunk a bal oldali menüsávban a mappa ikonra! 
- Kattintsunk az eszköztárban a projekt létrehozása ikonra! 
- A mappa lista végén megjelenő beviteli mezőbe írunk be azt, hogy „tesztprojekt”!
- Kattintsunk valahová, vagy nyomjunk Enter-t!

- Várt eredmény: Új sor jelenik meg a fájl lista végén projekt ikonnal. 

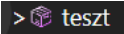
4.4.2.3.5 Projekt vagy mappa törlése

- Leírás: Egy mappát vagy projektet fogunk kitörölni
- Előfeltétel: Legyen egy sima mappa a gyökérkönyvtárban
- Tesztlépések:

- Kattintsunk a bal oldali menüsávban a mappa ikonra! 
- Válasszunk ki egy mappát vagy projektet kattintással! Fontos, hogy ne a SharedWithMe legyen az.
- Kattintsunk a kuka ikonra! 


- Várt eredmény: A lista frissül, a törölt elem már nem jelenik meg a listában.

4.4.2.3.6 Projektmappa létrehozása


- Leírás: Egy projekten belül projektmappát fogunk létrehozni.
- Előfeltétel: A fájlmenedzserben egy projektbe vagyunk lépve, illetve az **Editor** funkció meg van nyitva.
- A Tesztlépések megegyeznek az **Új mappa létrehozása** tesztelésével.
- Várt eredmény:
 - Létrejön egy projektmappa. 
 - Létrejön a szerkesztőfelületen egy doboz, **teszt** címmel.

4.4.2.3.7 Projektmappa törlése

- Leírás: Egy projekten belül projektmappát fogunk kitörölni.

- Előfeltétel: A fájlmenedzserben egy projektbe vagyunk lépve, illetve az **Editor** funkció meg van nyitva.
- A Tesztlépések:
 - Kövessük az [Új mappa létrehozása](#) tesztet lépéseit!
 - Kattintsunk az újonnan létrehozott mappára a fájlmenedzserben!
 - Majd, a kuka ikonra az eszköztárban!
- Várt eredmény:
 - Létrejön egy projektmappa. , amely a törlés hatására eltűnik
 - Létrejön a szerkesztőfelületen egy doboz, **teszt** címmel, majd a törlés hatására a doboz eltűnik.

4.4.2.3.8 Beágyazott projektmappa létrehozása

- Leírás: Egy projekten belül egy beágyazott projektmappát fogunk létrehozni.
- Előfeltétel: A fájlmenedzserben egy projektbe vagyunk lépve, a projekten belül van egy „teszt” projektmappánk, illetve az **Editor** funkció meg van nyitva.
- Tesztlépések:
 - Dupla kattintással lépünk be a **teszt** mappába!
 - Kövessük az [Új mappa létrehozása](#) tesztet lépéseit!
 - Kattintsunk a **back** linkre a fájlmenedzser felületen!
 - Kattintsunk a kis  gombra a „teszt” mappa mellett!
- Várt eredmény:
 - A fájlmenedzser felületen egyszerre látható a **teszt**, illetve a beágyazott mappa némi baloldali behúzással.
 - A szerkesztőfelületen a „teszt” dobozon belül, listaelemként megjelenik a beágyazott mappa neve.

4.4.2.3.9 Osztály létrehozása duplán beágyazott projektmappában

- Leírás: Egy projekten belül egy duplán beágyazott projektmappát fogunk létrehozni, a középső szinten egy osztálydobozzal.
- Előfeltétel: A fájlmenedzserben egy projektbe vagyunk lépve, a projekten belül van egy „baseFolder” projektmappánk, illetve az **Editor** funkció meg van nyitva.
- Tesztlépések:

- Dupla kattintással lépünk be a „baseFolder” projektmapába!
- Kövessük az *Új mappa létrehozása* tesztet lépéseit, a név legyen „nestedFolder”!
- Hozzunk létre egy osztálydobozt!
 1. Kattintsunk a Class gombra az eszköztárban!
 2. Rajzoljuk meg a dobozt
 3. Állítsuk át a nevét „TestClass”-ra!
 1. Kattintsunk a doboz címére duplán!
 2. Írjuk be az új nevet!
 3. Kattintsunk a beviteli mezőn kívülre, vagy nyomjunk Enter-t!
- Kattintsunk a **back** linkre a fájlmenedzser felületen!
- Kattintsunk a kis  gombra a „baseFolder” mappa mellett!
- Várt eredmény:
 - A fájlmenedzser felületen egyszerre látható a „baseFolder”, illetve a beágyazott „nestedFolder” és „TestClass” némi baloldali behúzással, a „baseFolder” alatt.
 - A szerkesztőfelületen a „baseFolder” dobozban megjelenik beágyazott módon a „nestedFolder” és a „TestClass”.
 - Érdekesség: a felső szinten az osztály megjelenik.

4.4.2.3.10 Osztály törlése duplán beágyazott projektmapában

- Leírás: Egy projekten belül egy duplán beágyazott projektmapát fogunk létrehozni, a középső szinten egy osztálydobozzal, amelyet azonnal ki is törölünk.
- Előfeltétel: A fájlmenedzserben egy projektbe vagyunk lépve, a projekten belül van egy „baseFolder” projektmapánk, illetve az **Editor** funkció meg van nyitva.
- Tesztlépések:
 - Kövessük az *Osztály létrehozása duplán beágyazott projektmapában* tesztet lépéseit!
 - Kattintsunk duplán a basefolder mappára!
 - Jelöljük ki (kattintással) „TestClass” osztályt!
 - Nyomjuk meg a del gombot!

- Várt eredmény:
 - A „baseFolder” mappa alatt már csak a „nestedFolder” projektmappa található, „TestClass”-t nem látunk.
 - A szerkesztőfelületen a „baseFolder” dobozon belül már csak a „nestedFolder” deklarációja látható.



4.4.2.3.11 Vonal létrehozása **manual*

- Leírás: Egy projekten belül létrehozunk két dobozt, közéjük vonalat húzunk, majd néhány töréspontot készítünk bele.
- Előfeltétel: A fájlmenedzserben egy projektbe vagyunk lépve, a projekten belül van két bármilyen dobozunk, lehet az jegyzetdoboz, osztálydoboz vagy csomagdoboz.
- Tesztlépések:
 - Válasszuk ki bármelyik vonal típust az eszköztárból!
 - Húzzuk meg a vonalat a két doboz közé!
 - Jelöljük ki a behúzott vonalat! (színes kiemelést kap)
 - Fogjuk meg bármelyik pontját a vonalnak!
 - Kicsit mozgassunk az egeren, majd engedjük fel az egérgombot!
- Várható eredmény: Létrejön egy vonal a két doboz között egy törésponttal.

4.4.2.3.12 Vonal törlése **manual*

- Leírás: Egy projekten belül létrehozunk két dobozt, közéjük vonalat húzunk, majd a vonalat kitöröljük.
- Előfeltétel: A fájlmenedzserben egy projektbe vagyunk lépve, a projekten belül van két bármilyen dobozunk, lehet az jegyzetdoboz, osztálydoboz vagy csomagdoboz.
- Tesztlépések:
 - Válasszuk ki bármelyik vonal típust az eszköztárból!
 - Húzzuk meg a vonalat a két doboz közé!
 - Jelöljük ki a behúzott vonalat! (színes kiemelést kap)
 - Nyomjuk meg a del gombot!
- Várható eredmény: A vonal eltűnik.

4.4.2.3.13 Attribútum vagy metódus hozzáadása osztálydobozhoz

- Leírás: Egy projekten belül egy osztálydobozba beszúrunk egy új attribútumot.
- Előfeltétel: A fájlmenedzserben egy projektbe vagyunk lépve, a projekten belül van egy osztálydobozunk.
- Tesztlépések:
 - Válasszuk ki az osztálydobozt!
 - Vigyük az egeret az „attributes” szekció fölé! 
 - Kattintsunk a „+” gombra!
 - Adjunk meg egy nevet a beviteli mezőn keresztül, például „name:string”!
 - Nyomjunk egy Enter-t vagy kattintsunk valahova!
- Várható eredmény: Az attribútum rögzítésre kerül: Az edit szócska eltűnik és színes szintaxissal megjelenik a beírt attribútum. 

4.4.2.3.14 Jegyzetdoboz létrehozása

- Leírás: Egy projekten belül egy egyszerű jegyzetdobozt fogunk létrehozni.
- Előfeltétel: A fájlmenedzserben egy projektbe vagyunk lépve, illetve az **Editor** funkció meg van nyitva.
- Tesztlépések:
 - Hozzunk létre egy jegyzetdobozt!
 1. Kattintsunk a „NoteBox” gombra az eszköztárban!
 2. Rajzoljuk meg a dobozt
 3. Állítsuk át a doboz tartalmát „Test123\$@.”-ra!
 1. Kattintsunk a doboz címére duplán!
 2. Írjuk be az új tartalmat!
 3. Kattintsunk a beviteli mezőn kívülre, vagy nyomjunk Enter-t!
- Várt eredmény:
 - A szerkesztőfelületen megjelenik a jegyzetdoboz a beírt tartalommal.

4.4.2.3.15 Olyan doboz törlése, amelyhez vonal tartozik **manual*

- Leírás: Egy projekten belül létrehozunk két dobozt, közéjük vonalat húzunk, majd az egyik dobozt kitöröljük.
- Előfeltétel: A fájlmenedzserben egy projektbe vagyunk lépve, a projekten belül van két dobozunk, fontos, hogy jegyzetdoboz vagy osztálydoboz legyen mindkettő.
- Tesztlépések:
 - Válasszuk ki bármelyik vonaltípust az eszköztárból!
 - Húzzuk meg a vonalat a két doboz közé!
 - Jelöljük ki az egyik dobozt!
 - Nyomjuk meg a del gombot!
- Várható eredmény: A doboz és a hozzá tartozó vonal is eltűnik.

4.4.2.4 Manuális tesztelés

A többrésztvevős szerkesztés teszteléséhez az első lépés az, hogy megnyitunk két különböző böngészőt, vagy egy böngészőt és egy új ablakot privát módban. Ez azért kell, hogy a kliens oldalon használt sütik ne keveredjenek össze. Belépünk két különböző fiókkal, megnyitjuk az oldalsó panelen ugyanazt a megosztott mappát és már lehet is tesztelni! A várható eredmény az, hogy a diagram szerkesztése közben szinkronizált módon, teljesen ugyan az történik mindkét ablakban.

5. Továbbfejlesztési lehetőségek

5.1 File Manager

Ehez a modulhoz a következő új funkciókat lehetne megvalósítani:

- Ha egy osztály header-re nyomunk, akkor a canvas a kijelölt osztályt kiválasztja a szerkesztő felület és ráfókuszál (oda ugrik a canvas).
- Refresh gomb, amely a panel tartalmát frissíti, illetve auto refresh minden dml művelet után, amit a szerkesztő felületen végzünk: delete, create, update.
- Átnevezés funkció: a meglévő mappák és projektek átnevezése.
- Áthelyezés funkció: a kiválasztott fájlok kivágása és beillesztése máshová.
- Megosztás megszüntetése gomb
- Egy-egy fájl részleteinek megtekintése: Ki hozta létre, mikor, kikkel lett megosztva. (utóbbi csak akkor látszik, ha a tulajdonos nézi)

5.2 Editor

Szerkesztőfelület:

A szerkesztőfelülettel kapcsolatban a következő fejlesztéseket lehetne megvalósítani:

- Ha, egy vonalan már sok töréspont van, akkor 2 pontot egymásra húzva törölhetjük az egyiket, másszóval össze olvadnak.
- Dokumentáció írási lehetőség bármihez: osztályhoz, csomag definícióhoz, attribútumhoz, egy külön ablakban. Ehhez külön akciótípust definiálnék, pl.: UPDATE_DOC.

Java kód generálás:

A felhasználónak lehetősége nyílik az elkészített projektet a megadott definíciók alapján java csomagokká és forrásfájlokká kigenerálni. Ezt zip formátumban letöltheti a felhasználó.

5.3 Felhasználói fiókok

- Elfelejtett jelszó funkció

- 2 lépéses e-mailben megerősítő linkre kattintásos regisztráció
- Felhasználói adatok módosításának lehetősége belépés után.

6. Összegzés

Úgy érzem, a bevezetőben megfogalmazott probléma hipotézist sikerült megoldani a dolgozatban ismertetett három rétegű alkalmazással. Az alkalmazást olyan embereknek ajánlom, akiknek fontos az, hogy bármilyen projekt megtervezését, felvázolását a kollégáival, társaival együtt interaktívan és hatékonyan tudja szerkeszteni. Az UML Conference egy böngészőben futtatható UML diagram tervező alkalmazás. A szerkesztési funkcionalitás hatékonyságának maximalizálásához élő, socketes kapcsolattal kötöttem össze az összes szerkesztésben résztvevő felhasználót. Diagramjainkat projektmappákba, azokat projektekbe, majd a projektjeinket mappákba rendezhetjük, amelyeket megoszthatunk másokkal.

Rengeteg olyan technológiát, keretrendszert, könyvtárcsomagot használtam, amelyek nem képezik az egyetemi tananyag részét, néhány példa: Angular, typescript, Spring-Boot, JPA, MockMvc, Mockito, Selenium, XPATH, WebDriver.

A felhasználói felülethez Angular keretrendszert használtam. A typescript nyelv statikusan típusozhatósága azonnal megtetszett, amikor a szakmai gyakorlaton megannyi előnyét megismertem.

A webalkalmazás elkészítéséhez a Spring-boot keretrendszert választottam, mert ez volt az első microservice támogatottsággal rendelkező keretrendszer, amit megismertem még 2018-ban. A megismeréséhez a [SanFranciscobol Jöttem youtube csatorna](#)[23] a kezdetekben nagy ambíciót adott, amit ezúton is köszönök neki. A szerveroldal implementációjának legnehezebb része a szerkesztőfelület logikájának megtervezése volt. A socketes kapcsolat állandó üzenetküldéseinek, válaszainak sorbarendezeésére a termelő-fogyasztó tervminta többszörös egymásba ágyazása adta meg a végleges vázat 4 állandó feldolgozó szállal és 4 külön üzenetcsatornával.

Az adatbázisszerver típusa számomra nem volt lényeges, mert tudtam, hogy a Spring JPA könyvtárcsomagja gondoskodni fog az adatbázis kommunikáció zökkenőmentes lebonyolításáról. A szakmai gyakorlati helyen az ingyenes PostgreSQL-ről viszonylag sok szó esett, a hatékonyságát több helyen is dicsérve. Platform független is, ezért ezt választottam.

7. Irodalomjegyzék

- [1] <https://angular.io/docs>
- [2] <https://spring.io/projects/spring-boot>
- [3] <https://www.postgresql.org>
- [4] <https://www.redhat.com/en/topics/api/what-is-a-rest-api>
- [5] <https://www.oracle.com/java/technologies/javase-jre8-downloads.html>
- [6] <https://nodejs.org/en/download/>
- [7] <https://www.postgresql.org/download/>
- [8] https://help.xmatters.com/ondemand/trial/valid_email_format.htm
- [9] <https://www.baeldung.com/spring-data-annotations> ,
<https://www.baeldung.com/jpa-entities>
- [10] <https://code.visualstudio.com>
- [11] <https://netbeans.apache.org/download/nb120/>
- [12] <https://www.pgadmin.org>
- [13] <https://www.flaticon.com>
- [14] https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API
- [15] <https://spring.io/projects/spring-security>
- [16] <https://github.com/FasterXML/jackson-docs>
- [17] <https://www.postman.com>
- [18] <https://www.h2database.com>
- [19] https://www.youtube.com/watch?v=NbBf8LU6UB0&list=PLyriihBWoulywcSbZijjeS_IHH19uJZG5q&index=22 [3:00] (Utolsó elérés dátuma: 2021.05.27)
- [20] <https://site.mockito.org>
- [21] <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.test.web.servlet.MockMvc.html>
- [22] <https://www.selenium.dev>
- [23] https://www.youtube.com/channel/UCK_DAAIso6GNsOKyL2funLw