**Eötvös Loránd Tudományegyetem**

**Informatikai Kar**

**UML-Conference  
 webes alkalmazás**

**Témavezető:**

Szalai-Gindl János Márk

adjunktus

ELTE Információs Rendszerek Tanszék

**Szerző**:

Szeifert Péter

Programtervező informatikus BSc.

Budapest, 2021

# Tartalomjegyzék

[1. Tartalomjegyzék 2](#_Toc72255556)

[2. Bevezetés 4](#_Toc72255557)

[3. Felhasználói dokumentáció 4](#_Toc72255558)

[3.1 Rendszerkövetelmények 4](#_Toc72255559)

[3.2 A program telepítése 5](#_Toc72255560)

[3.3 A program használata 5](#_Toc72255561)

[3.3.1 Belépés az oldalra 5](#_Toc72255562)

[3.3.2 A felhasználói felület 6](#_Toc72255563)

[3.3.3 A File Browser 7](#_Toc72255564)

[3.3.4 A megosztás ablak 8](#_Toc72255565)

[3.3.5 A szerkesztő felületről és egy projekt felépítéséről 9](#_Toc72255566)

[3.3.6 Az Editor felépítése 11](#_Toc72255567)

[3.3.7 Editor funkciói részletesen 12](#_Toc72255568)

[4. Fejlesztői dokumentáció 14](#_Toc72255569)

[4.1 Felhasznált technológiák 14](#_Toc72255570)

[4.1.1 Kliens oldal 14](#_Toc72255571)

[4.1.2 Szerver oldal 14](#_Toc72255572)

[4.1.3 Adatbázis 15](#_Toc72255573)

[4.1.4 Fejlesztői környezet 15](#_Toc72255574)

[4.2 Szerkezeti felépítés 16](#_Toc72255575)

[4.2.1 Client-side 17](#_Toc72255576)

[4.2.2 Szerver oldal 32](#_Toc72255577)

[4.3 Tesztelés 43](#_Toc72255578)

[4.3.1 Unit tesztelés 43](#_Toc72255579)

[4.3.2 Integrációs tesztelés 44](#_Toc72255580)

[5. Szójegyzék 49](#_Toc72255581)

[6. Továbbfejlesztési lehetőségek 50](#_Toc72255582)

[6.1 File Manager 50](#_Toc72255583)

[6.2 Editor 50](#_Toc72255584)

[6.3 Felhasználói fiókok 50](#_Toc72255585)

[7. Irodalomjegyzék 52](#_Toc72255586)

# 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.

A program 3 rétegből áll. Egy angular kliensből, Java spring-boot szerver-ből, illetve PostgreSQL[[1]](#endnote-1) adatbázisból. Az interaktív szerkesztő felülethez natív, szöveg alapú web socketeket használtam, az összes többi kommunikáció REST API-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, a forráslista előtt.

# Felhasználói dokumentáció

## Rendszerkövetelmények

Windows esetében a 3 program komponens futtatásához legalább 1,5 GB memóriát ajánlott szabadon hagyni. A java szerver átlagosan 800-1000 mb memóriát foglal le, a kliensszerver és az adatbázisszerver memóriafelhasználása elhanyagolható. Processzor teljesítmény tekintetében bármi megfelel, amin az operációs rendszer elfut. A teljes program kevesebb, mint 1GB tárhelyet foglal.

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

## A program telepítése

* Telepítsük fel A Java8-hoz tartozó JRE[[2]](#endnote-2)-t, adjuk hozzá a környezeti változókhoz(fűzzük fel a classpath-ra)
* Telepítsük fel az npm-et, ezt is adjuk hozzá a környezeti változókhoz.
* Telepítsük a PostgreSQL 13-as verzióját.
* indítsuk el az init.bat fájlt. Ez elindítja a postgresql adatbázist.

Indítsuk el a start.bat fájlt.

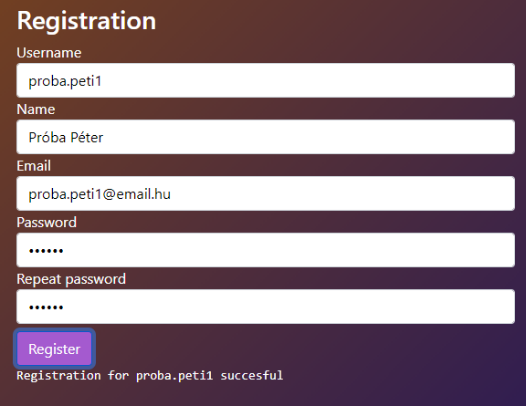
## A program használata

Nyissuk meg a böngészőt, navigáljunk el a <http://localhost:8100> oldalra!

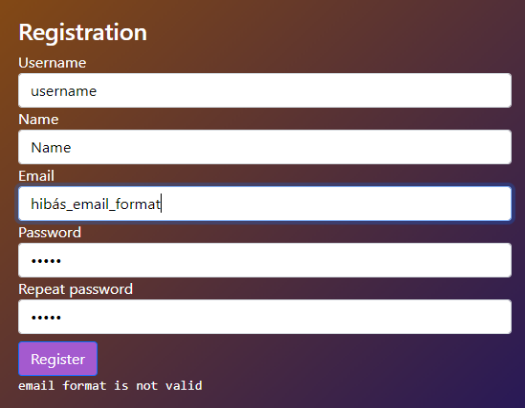
### Belépés az oldalra

|  |  |  |
| --- | --- | --- |
| 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 linkre kattint | Kitölti az email, username, Name és password mezőket, majd és a “Register” gombra kattint. | Ha az adatok formátumai megfelelőek, akkor a regisztráció sikeres. Most már be 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. |

### A felhasználói felület



1. ábra: Sikeres regisztáció



2. ábra Hiba a regisztáció során

A felső menü sávban a következő opciók vannak: Files, Home, Editor. Belépés után a „Home” felületet látjuk, ahol a program köszönt minket.

A **“be vagyunk lépve”** feltételt a továbbiakban **“BVL”** jelöléssel jeleztem.

|  |  |  |  |
| --- | --- | --- | --- |
| GIVEN | WHEN | THEN | EXPLANATION |
| BVL | File browser menüpontra kattintunk | **Bal oldali panelen** kinyílik a file browser | [A file browser](#_A_File_Browser) |
| 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 elkezdi ugyanazt a diagramot szerkeszteni, auomatikusan bekapcsolódik a munkamenetbe. |

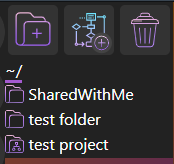
### 

### A File Browser

Itt tudjuk **böngészni** a **projektjeinket**.

A panelt kinyitva a gyökérkönyvtárt nyitja meg az oldal. Itt egy fix mappa található *~/sharedWithMe/*.

* 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 file browser felület 2 részből áll, egy eszköztárból (felül) és nagyobb részt böngészőből (alul). A menedzserrel projekteket, mappákat hozhatunk létre. Ezeket törölhetjük, illetve megoszthatjuk másokkal. Megosztáskor a célszemély olvasási jogot kap a mappákhoz, illetve szerkesztői jogot a projektekhez. Másik személy projektjét vagy mappáját nem törölhetjük ki.

*3. ábra: file browser panel*

Minden normális fájlkezelőhöz hasonlóan a *File Browser* panel is rendelkezik az alapvető funkcionalitásokkal. Egy kattintással egy elemet jelölhetünk ki. Ctrl+ bal klikk-el több elemet is ki lehet jelölni, illetve jelölést megszüntetni. A kijelölt elemek kék keretet kapnak.

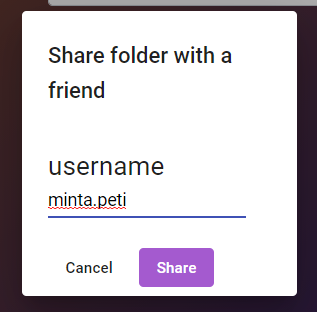
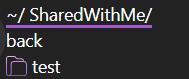
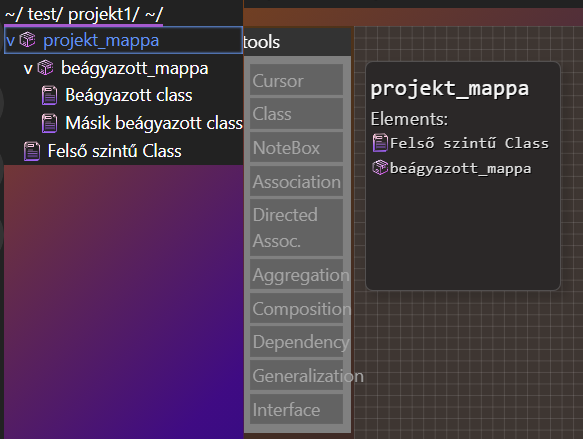
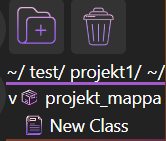
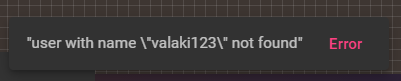
Felül, az eszköztárban 3 gomb van: mappa létrehozása, projekt létrehozása, törlés.

|  |  |  |  |
| --- | --- | --- | --- |
| 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. |
| Kiválasztunk egy, vagy több elemet | Felül a **törlés** gomra kattintunk | Kitörli a kiválasztott elemet a listából | Ilyenkor fa szerűen a beágyazott mappák, projektek is törlődnek. |
| Kiválasztunk egy elemet | Felül az **átnevezés** gomra 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** gomra kattintunk | Megnyílik egy párbeszéd ablak, ahol a barátunk nevét meg kell adni. | [Megosztás ablak](#_(4)_A_megosztás) |

### 

### A 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 nevet, akkor a mappa bekerül az ő vele megoszott mappák közé. Ha nem, a szerver hibaüzenetet küld kis popup ablakban.

Ha a mappa, amiben állunk egy **projektmappa** (a projekt gyökér mappája is ilyen), akkor az eszköztár részen már csak 2 gomb látható. A **projektmappa létrehozása** gomb, illetve a **törlés**. Ezzel a két gombbal csak a projektmappák manipulálhatók. Továbbá, ha a manager ablakban létrehozunk egy **projektmappát**, akkor **az automatikusan megjelenik a szerkesztőfelületen**. Ha egy osztály definíciót létrehozunk, (Toolbox->Class) akkor az a szülő mappa alá **virtuálisan** **bekerül** a menedzserbe, hogy jobban átlátható legyen, mi hova tartozik. Továbbá a mappa ikonja mellett balra található kis nyilacskával kinyithatók, illetve becsukhatók a mappában található elemek. Osztálydefiníciók szerkesztésekor a nagyszülő diagramban automatikusan frissülnek a csomag definíciók (akár élő kapcsolat esetén is).

8. ábra: A Menedzser és az Szerkesztőfelület szoros kapcsolata

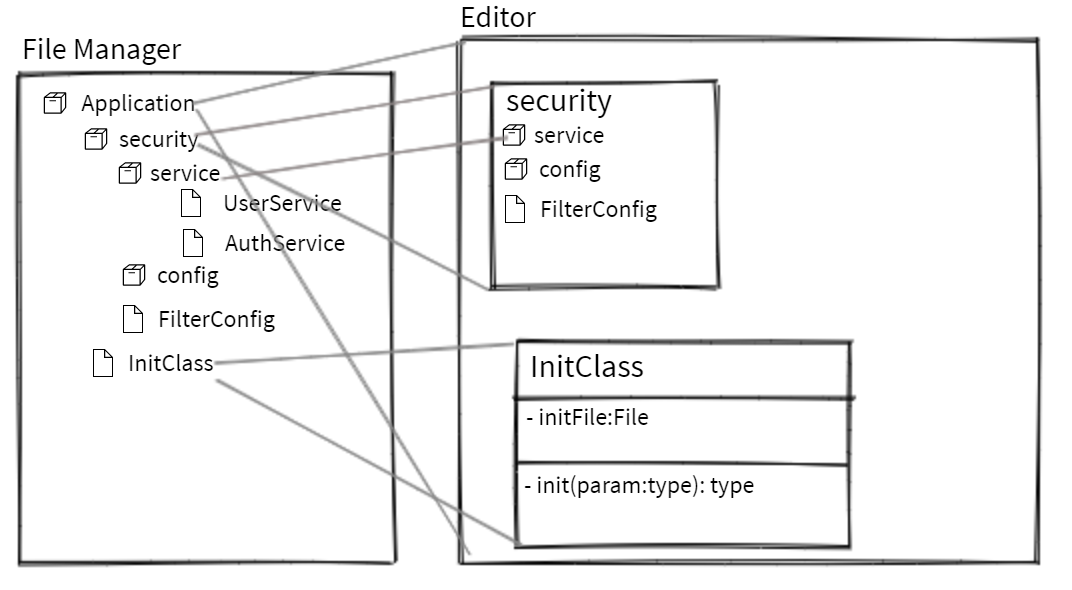
4. ábra: nem létező felhasználóval akarunk megosztani valamit

7. ábra: Project menedzsment

6. ábra: a mappa megosztás sikeres (minta.peti user)

5. ábra: megosztás ablak

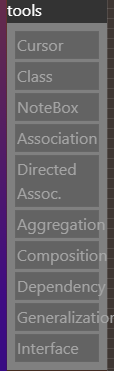
### A szerkesztő felületről és egy projekt felépítéséről

Egy projekt valójában diagrammok struktúrált halmaza. Egy projekt lényegében egy mappa, amelyben almappák (diagramok) vannak. Egy diagram osztálydefiníciók, megjegyzések, kapcsolatok  **és beágyazott diagramok** egy csomagja, halmaza. Egy projekt struktúrája pl. a következőképpen nézhet ki: 

9. ábra: egy példa projekt felépítése (vázlat)

### Az Editor felépítése

Az Editor felépítése áll egy diagram szerkesztő felületből, illetve 2 mozgatható ablakból. A szerkesztői eszközök a ToolBox ablakban kaptak helyet.



10. ábra Eszköztár ablak

#### Toolbox ablak

Itt Osztályleíró dobozok(Class), megjegyzések(note) illetve kapcsolati vonalak beszúrását választhatjuk ki.

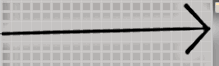
##### Class

Ha kiválaszjuk ezt az eszközt (rákattintunk) akkor a következő alkalommal, amint az egeret lenyomjuk a vásznon, egy új osztálydoboz jön létre. Méretét úgy állíthatjuk be, hogy az egér lenyomása után még lent tartjuk. A doboz méretét és pozícióját bármikor módosíthatjuk. Elég csak szimplán rákattintani és drag’n drop módszerrel 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 kiválaszott doboz kék keretet kap és editing feliratot. A del gomb lenyomása esetén a doboz törlődik az összes hozzá kapcsolódó vonallal együtt.

##### NoteBox

Ezzel az eszközzel egy ugyanolyan dobozt hozhatunk létre, mint a Class esetében, teljesen hasonló tulajdonságokkal. Itt a legfőbb különbség a dobozon belüli tartalom, ugyanis ez a doboz csak egyszerű szöveg tárolására alkalmas. Megjegyzések írásához tökéletes.

##### Vonalak

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

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

Ha egy behúzott vonalra kattintunk, akkor az narancssárga kijelölést kap. Ez azt jelenti, hogy lezártuk a vonal szerkesztését a többiek számára. Ilyenkor a többiek oldalán piros színnel emeli ki a lezárt vonalat a program. Ha a vonal bármelyik pontját megfogjuk, akkor ott egy töréspont keletkezik, amelyet akárhova elhelyezhetünk. A töréspontokkal a vonalunkat több kisseb szakaszra oszthatjuk, hogy jobb minőségű, átláthatóbb ábrákat készíthessünk. Egy kiválasztott vonalat a del gomb lenyomásával törölhetünk.

#### Log ablak

Itt az esetleges hibákról kapunk értesítéseket. Pélául nem szerkeszthető elemet próbálunk módosítani, vagy ha a magas ping 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.



11. ábra: Log ablak, működés közben

### Editor funkciói részletesen

|  |  |  |  |
| --- | --- | --- | --- |
| GIVEN | WHEN | THEN | EXPLANATION |
| Megosztott diagramot szerkesztünk, egyedül | Valaki más is elkezdi szerkeszteni ugyanazt a diagramot | Ő is **bekapcsolódik** a szerkesztésbe. | Ilyenkor az elemek **szerkesztését** **zárolásokkal** látja el a program, ezt a szerver 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 lock-ot** kap: egy lakattal ellátott keretet, hogy ne lehessen kitörölni, illetve mozgatni, de a doboz **többi szövegmezője** attól még **szerkeszthető** marad a többiek számára. |

# Fejlesztői dokumentáció

## Felhasznált technológiák

### 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 rengeteg újrafelhasználhatóság, a nézetek egymásba ágyazása könnyedén és átláthatóan. A nézetmodell (.ts fájl) elkülönül a nézettől (.html) illetve a stíluslaptól (.css vagy .scss). Az angularról annyit elmondanék, hogy a többi mai front-end keretrendszerrel ellentétben az Angular alapértelmezett módon typescript-et használ, amelynek sok előnye van. Főleg, ha valami nagyobb, enterprise méretű applikációt szeretnénk készíteni. A typescipt tulajdonságaiból kiemelnék néhányat.

1. Lehet vele statikusan típusos javascript kódot írni.
2. A típusozott már „typescript” kódot a saját fordítója mezei javascript kóddá generálja.
3. 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.

### Szerver oldal

A szerver oldalt, ahol az üzleti logika tartózkodik, Java nyelvben írtam meg, Spring Boot keretrendszert használva. A Spring Boot egy nyílt forráskódú Java alapú keretrendszer, amelyet leginkább a microservce-k építésére használnak az iparban. A Spring Boot a népszerű Spring keretrendszert használja a motorháztető alatt, 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 megírnunk. 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.

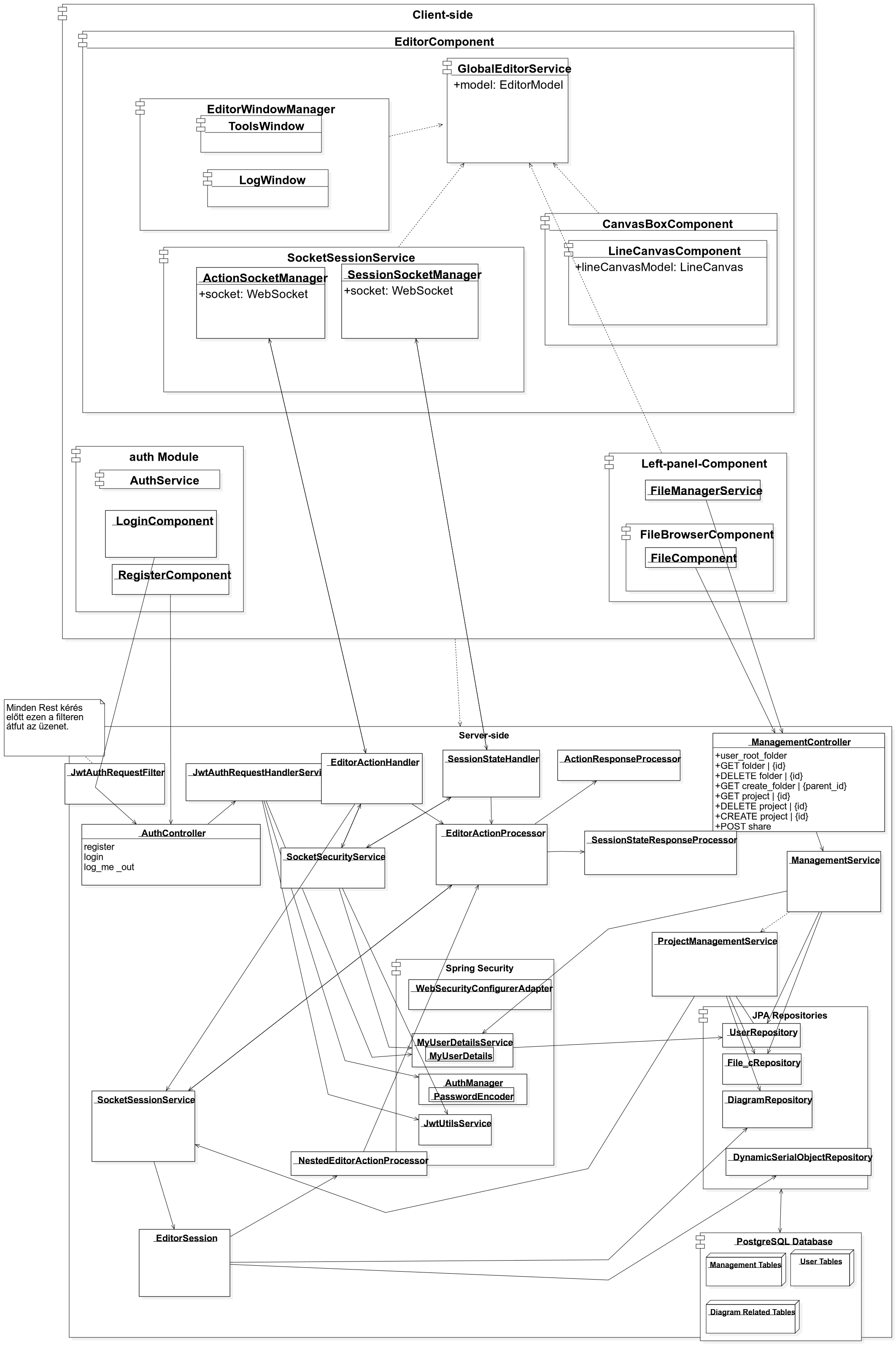
### Adatbázis

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

### Fejlesztői környezet

A Kliens oldal fejlesztéséhez Visual studio code[[3]](#endnote-3)-t használtam. Néhány angular illetve html bővítmény telepítésével. A Spring-Boot szerver fejlesztéséhez Apache NetBeans 12.0[[4]](#endnote-4)-t, a PostgreSQL adatbázis menedzseléséhez pedig PgAdmin 4-[[5]](#endnote-5)et használtam.

## Szerkezeti felépítés



NestedActionProcessor hiba van ->SocketSessionService-hez kéne kötni, SharedialogComponent, EditorSocketControllerService rosszul van írva.

### Client-side

A kliens oldal megtervezésekor fontosnak tartottam, hogy egyértelmű, magától érthetődő ikonokat, gombokat válasszak. Olyanokat amelyek nem törik meg a felhasználói szokásokat. Az ikonok nagyrészét a Flaticon[[6]](#endnote-6) oldalráról 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 mezei html fájl. Rengeteg angular splecifikus attribute-ot írhatunk az egyes <tag>-ekbe 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 meghivatkozza 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 default esetben Jasmine teszt keretrendszert ad a kezünk alá, Karma tesztfuttató szerverrel.

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

#### Auth module

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

* LoginComponent
* RegisterComponent
* AuthService

##### 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) fügvényhívással bejelentkezési kérelmet küldünk a /login végpontra.

##### RegisterComponent

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

##### AuthService

Ez egy service. A servicek olyan Typescript osztályok, amelyek más osztályokba injektálhatók dependency-ként. 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 authetication tokent, 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 lejárati dátumát meghaladva a rekord törlődjön. A trigger kódja megtalálható a back-end/src/main/resources/sql\_triggers.sql fájlban.

#### Bevezetés, Ősosztályok, Ősinterface-k

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

##### ŐsInterface-k

* SessionInteractiveItem
* SessionInteractiveContainer
* LogInteractive\_I

A **SessionInteractiveItem** interface 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** interface 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.

##### Ősosztályok

* **InteractiveItemBase**
* **InteractiveContainerBase**

Az **InteractiveItemBase** egy olyan absztrakt Bázis osztály, amely implementálja, deklarálja az összes olyan függvényt, methódust, amely egy interaktív elemhez kell. Itt külön kiemelném 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 a logolásért. Az előbbihez hasonlókat lehet elmondani az **InteractiveContainerBase**-ről is, ugyanis kiterjeszti az **InteractiveItemBase** osztályt, de emellett implementálja az **InteractiveContainer** interface-t.

#### A legfontosabb service-k a fő modulban

##### 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 event kezelés 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:

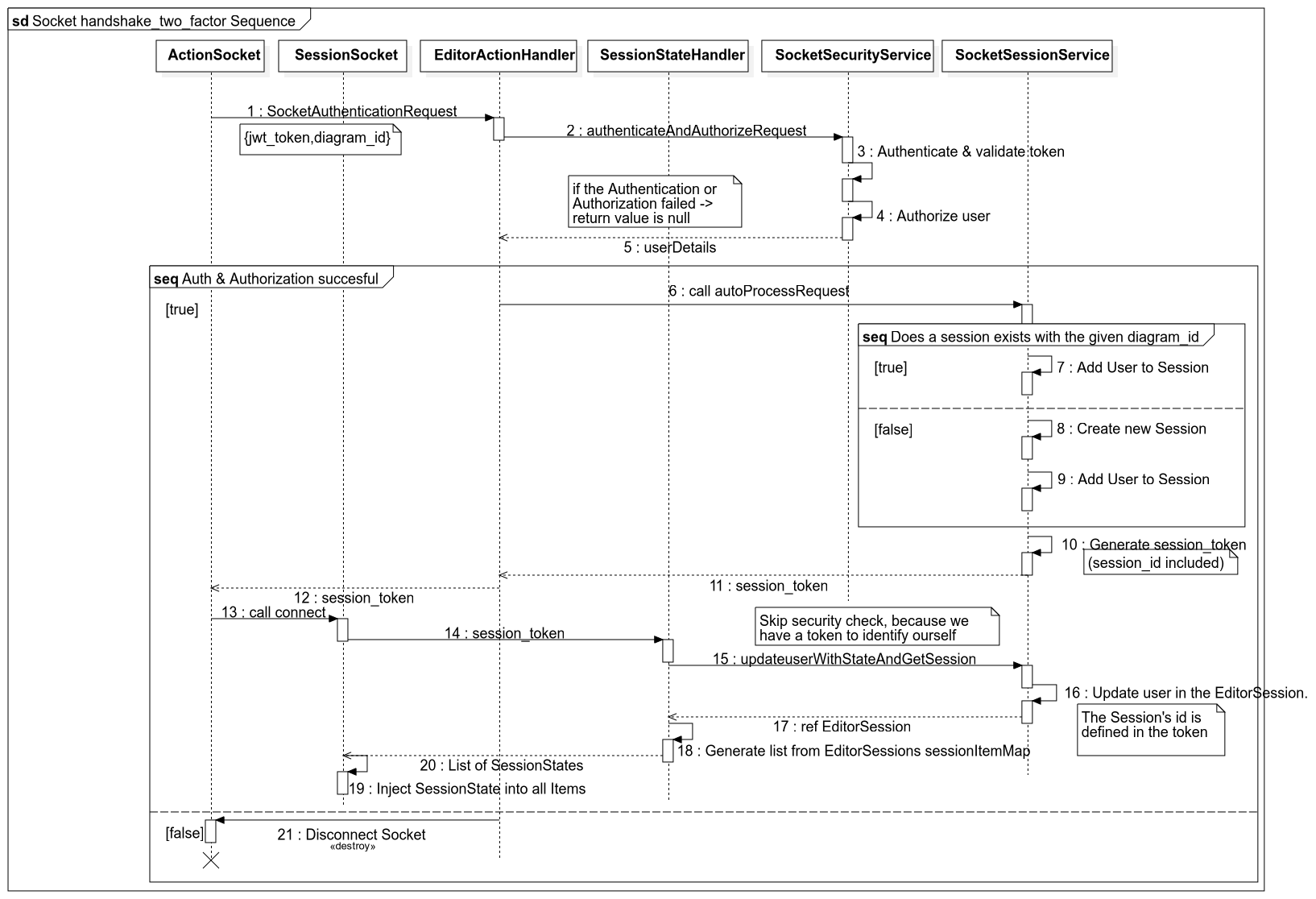
* **LineCanvasComponent**: az esemény hatására üríti az előző munkamenet vonalait, helyükre a frisseket inicializálja be.
* **EditorSocketControllerService**: az esemény hatására létrehozza a 2 socketet és csatlakozik a szerver websocket endpointjaihoz (/action és /state)

Egy event listener 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 bárhonnan 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.

##### 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 interaktálhatunk 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>[]

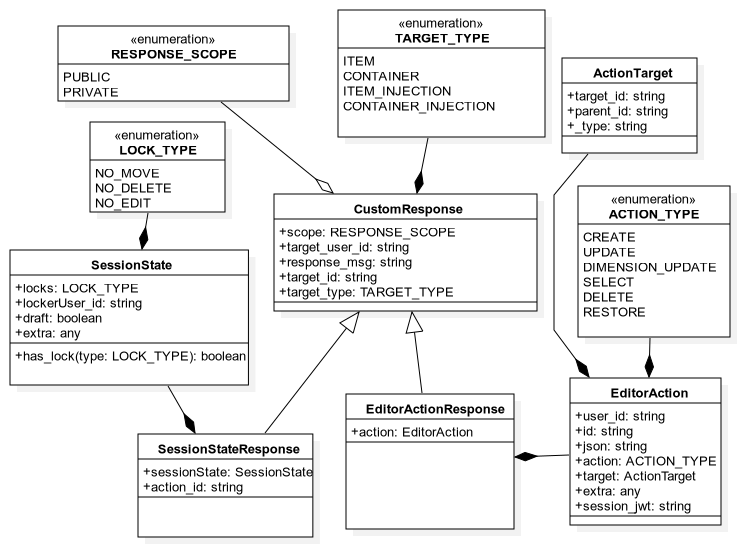
A constructor() szintén kiemelt fontosságú. A ’diagram\_fetch’ esemény hatására a connect() függvény fut le, amely csatlakoztatja az **ActionSocket**-et. Ha az **ActionSocket** csatlakoztatása sikeres és van hozzáférési jogunk a kért diagramhoz, akkor kapunk egy session\_token-t válaszként. Ennek a tokennek 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:

12. 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.

###### 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 authentikációs tokenünket, amelyet a bejelentkezéskor kaptunk és a cookie-k közé mentettünk. A folyamat többi részét fentebb már ismertettem.



13. ábra: Response objektumok felépítése

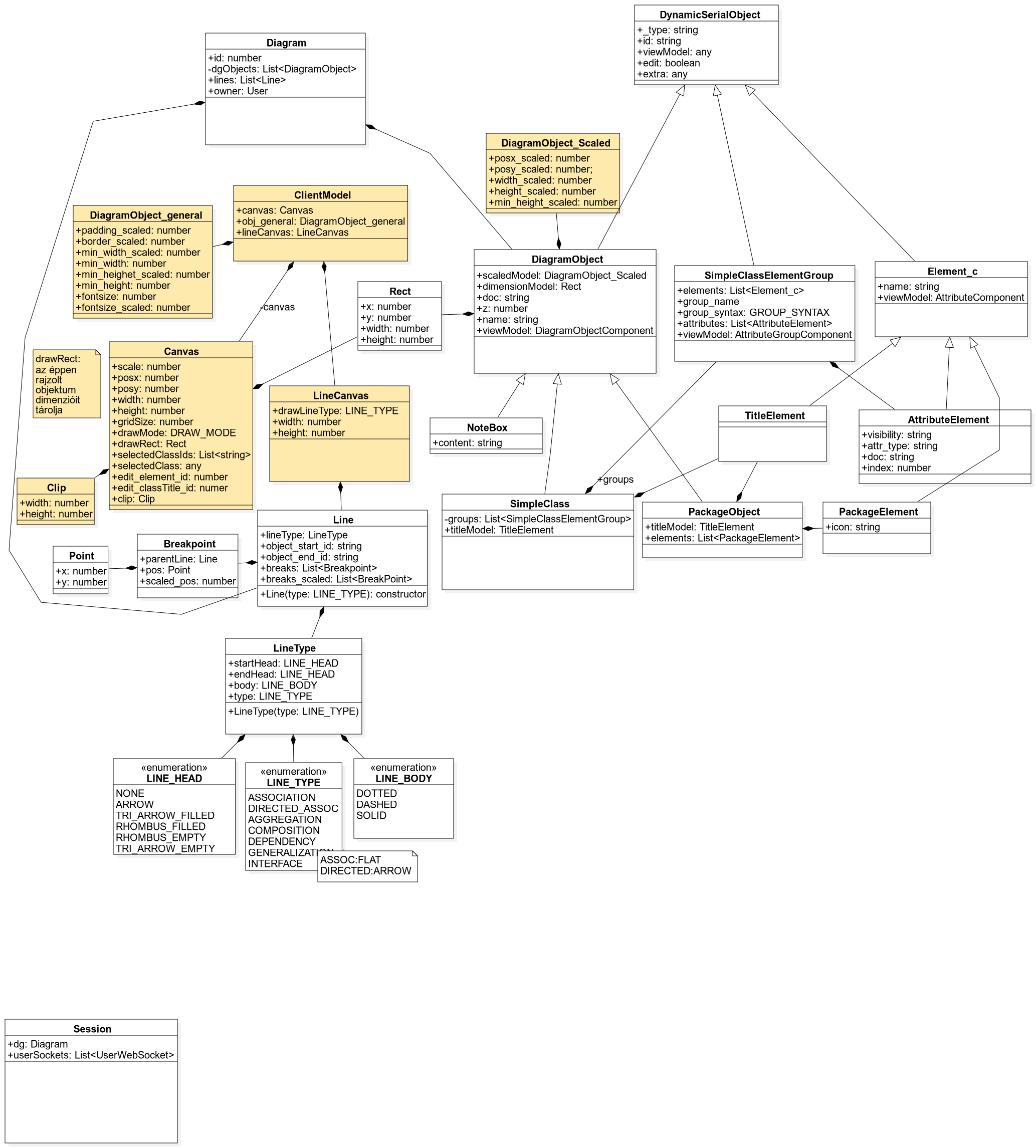
Bármikor, amikor a szerver küld egy akciót, akkor az **onmessage(e:any)** függvény fut le. Az üzenetet a függvény egyből egy **EditorActionResponse** objektummá alakítja. A **switch**-ben négy féle **Akciótípust** különböztetünk meg:

* **ACTIONTYPE.UPDATE**
  + Valami módosítás történt,akkor az **EditorSocketControllerService**-től elkérjük a módosítandó objektum nézetmondelljét (id alapján), majd nemes egyszerűséggel 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 user kapja meg az üzenetet, akkor a régi, ideiglenes id-t ki kell cserélni a szerver oldali id-ra a már létrehozozott 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 nézetmodellt 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.

###### SessionSocket

A 12-es 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 ActionSockethez 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 onmessae(msg) handler-jét az egyedi, általános handler-re. Az onmessage(msg)handler a státusz változásokat kezeli. Státuszváltozásnak számít az, amikor valaki létrehoz egy Objektumot (ezt nevezzük *„draft”* állapotnak), vagy amikor valaki kiválaszt valamit (zárol).

A lenti ábrán az alkalmazás adatmodelljei láthatók. A csak kliens oldalon létező modelleket **sárga** színnel emeltem ki.



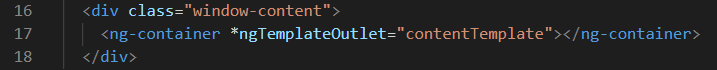
14. ábra: Az alkalmazás adatmodelljei (Objektum orientált megözelítésben)

#### CanvasBoxComponent

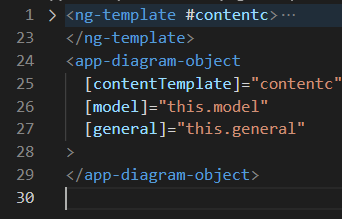
Ez a komponens implementálja a **SessionInteractiveContainer** interface-t, tehát ő is egy Interaktív konténer, lényegében a Diagram adatmodell nézetmodelljének az egyik fele. A szerkesztői funkcionalitás nagy része ebben a komponensben van leimplementálva. Néhány fonotsabb függvény:

* 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.

##### DiagramObjectComponent

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


15. ábra: Template injekció az Őskomponens nézetébe



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

A fenti két ábrán a **DiagramObjectComponent** és a **SimpleClassComponent** nézetének egymásba ágyazását láthatjuk. A gyermek komponens template kódjában meghívtam az ősosztály selectorját *(16. 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 esetetén dimenziói frissüljenek. A 14. ábrán ismertetett **DiagramObject** osztályból származtatott modellek mindegyikéről elmondhatók az előbb ismertetett tulajdonságok. Mindegyik komponensnek már csak a saját, specifikus fukcionalitásait kell megvalósítani a fájljaikban.

##### 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.

##### 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.

##### 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.

###### LineCanvasComponent

A szerkesztői funkciók maradék részei ebben a komponensben kaptak helyet. Implementálja a **SessionInteractioveContainer** interface-t, tehát ez is egy konténer. Miket tárolunk? Vonalakat. Ez a komponens a Vonalak megjelenítéséről gondoskodik. A megjelenítéshez **html5 Canvas**[[7]](#endnote-7)-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.

###### 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 angularnak 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)).

###### MathHelper

A matematikai számolásokért, collision detektálásért, vektor metszéspont és egyebek számolásáért a MathHelper segédosztály felel.

###### LineController

Ezt az osztályt úgy kell elképzelni, mint egy angular komponenst, csak nincs hozzá html template, sem css fájl. Implementálja az InteractiveItemBase 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.

#### EditorRootComponent

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

#### 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 következő képpen néz ki:

this.http

      .get<FileResponse>(environment.api\_url\_http +... {

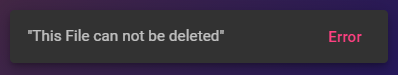
        headers: { 'Authorization': 'Bearer ' + getCookie("jwt\_token") }

      })

      .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-t 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:



17. ábra: Példa a snackBar használatára

#### 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.

#### FileComponent

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

##### 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** könyvtárcsomag. Ezt az összes fájltípus ősinterfészében definiáltam a @JsonTypeInfo és @JsonSubTypes annotációk segítségével. Ha az objektumokat a **Jackson** saját **ObjectMapper**jével konvertáljuk JSON formátumba, akkor az annotációk figyelembevételével dolgozik. Egy folder például így néz ki:

{

...

icon: "FOLDER",

id: 3,

name: "SharedWithMe",

owner: {id: 1, userName: "user", email: "example@email.hu",   
 name:"Peter"},

\_type: "folder"

...

}

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.

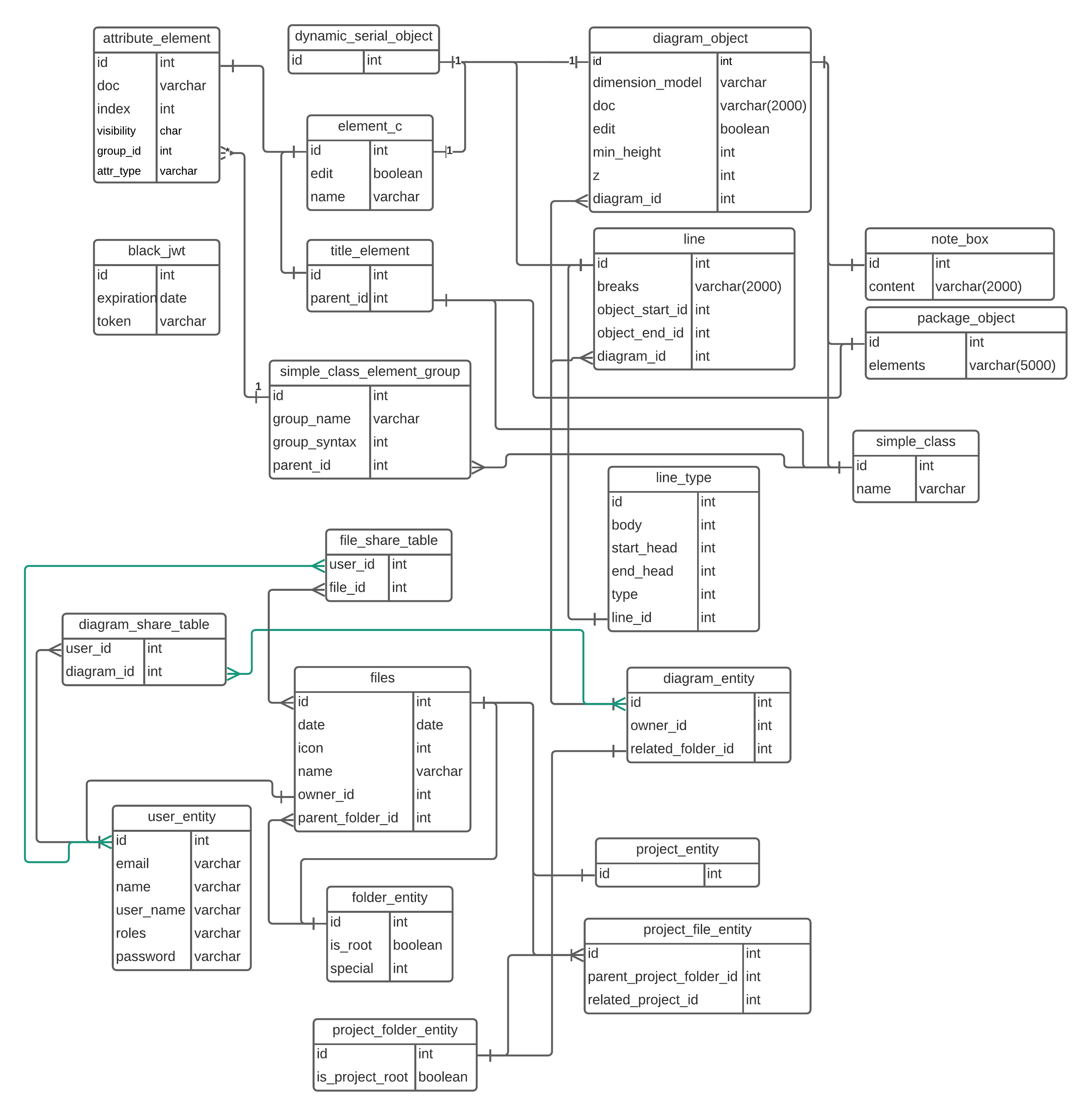
##### ShareDialogComponent

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.

### Szerver oldal

#### Adatmodellek

A 14-es ábrán már bemutattam az adatmodellek nagy részét. A back-end rétegen az adatmodellek kis mértékben eltérnek az ott ismertetettektől. A Minimális eltérések a perzisztancia 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, 18. ábrán az alkalmazás legfelső komponensei láthatók.



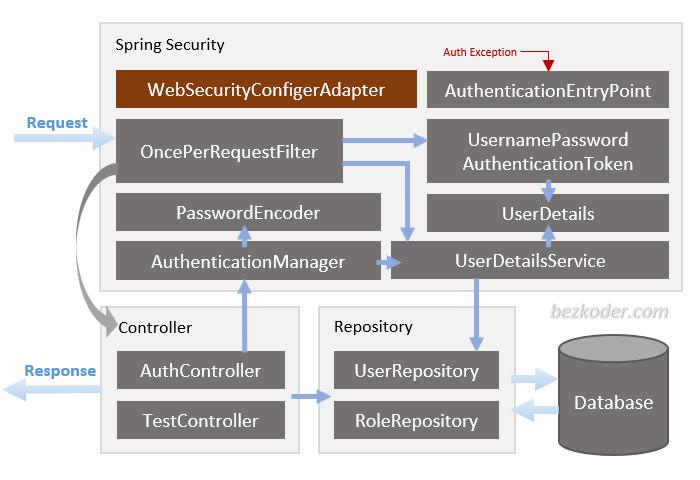
19. ábra: adatbázis táblák diagramja

#### Az alkalmazás rétegei.

A szerver több komponensből/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 találhatók. Egy **service** csomagból, ahol az üzleti logika található. Illetve egy **repository** csomagból, ahol az egyes adatmodellekhez tartozó adatbázis műveletek vannak definiálva.

#### Security

Először a **Security** modullal kezdeném, 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 authorizálást a **JwtAuthRequestFilter** végzi. A Spring Security műkődésének jobb megértéséhez beszúrok egy képet:[[8]](#endnote-8)



Itt a Request nyíl jelzi a http kérés beérkezésének helyét. A OncePerRequestFilter esetünkben a JwAuthRequestFilter.//magyarázat

##### JwtAuthRequestFilter

A filter minden egyes rest hívást megfog, és belenéz a header állományba. Ha abban található egy „Authorization” kulcs „Bearer xxx” értékkel, akkor helyben vagyunk. Először is az xxx token-ből kicsomagoljuk a felhasználónevet a JwtUtilService segítségével, majd a **UserDetailsService** az exportált username alapján kikeresi a user-t az adatbázisból. Ha van joga az adott forrást elérni (jelenleg csak USER role-van), akkor a kérést tovább engedi a filter. Ellenkező esetben **403-as Forbidden** hibával válaszol a kérésre.

##### JwtUtilService

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

###### Itt kiemelnék néhány függvényt:

Az **extractUsername(String)** és **extractExpiration(String)** függvények a paraméterben megadott token-be csomagolt **felhasználónevet,** illetve **lejárati 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 token-t. A claims Map-ba bármilyen Serializable Objektumot betehetünk és azt az algoritmus beleírja a token-be. A felsorolt 2 paraméter mellett a tokenbe belekerül a lejárati 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 token-t feketelistára helyezi az adatbázisban, azaz többé már nem használható azonosításra. A feketelistázott tokenek a lejárati dátumot követően egy **adatbázistrigger** segítségével **automatikusan** **törlődnek**.

##### 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.

##### JwtAuthRequestHandlerService

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

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

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

##### AuthController

A **/login, /register** és **/log\_me\_out** végpontokra érkező kéréseket itt kezeljük le.

#### 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. A **hibakezelés** mennyiségét külön **kiemelném** ebben a két modulban.

##### Management

###### ManagementController

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

###### ManagementService

Ez a service a ManagementController-t szolgálja ki. Az elején kiemelném, hogy minden publikus, kontrollerből hívott függvény végez token validációt illetve jogosultság ellenőrzést illetve, ha [dml művelet](#_Szójegyzék)ről van szó.

A fontosabb függvények:

A **getUserRootFolder függvény** az Authorization header-ben található token alapján megkeresi a user-t, 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 megörökli 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 barátunk felhasználóneve. Miután megtaláltuk a fájlt, a saját felhasználónk entitását, illetve a barátunk fiókjának entitását, annyi a dolgunk, hogy a fájl saját usersIamSaredWith listájába beszúrjuk a barátunk entitását, illetve a barátunk 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.

##### ProjectManagement

Ez a modul minden, projektekkel kapcsolatos interakció lekezeléséért felel.

###### ProjectManagementController

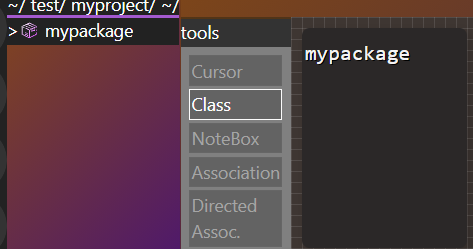
Erről a controllerről nem is mondanék sokat, működése nagyon hasonló a ManagementController-hez, annyi különbséggel, hogy itt a kontroller mögötti logikát a ProjectManagementService adja.

###### ProjectManagementService

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

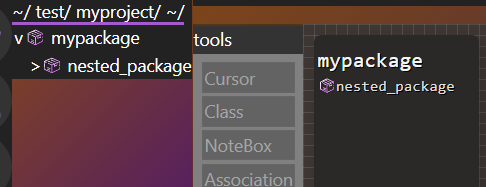
A **createProject függvény** projektek létrehozását végzi. A ProjectEntity konstruktor hívásánál automatikusan létrejön egy gyökér-projektmappa, ehez automatikusan létrejön egy DiagramEntity is.

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.



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

A függvény végét szintén kiemelném, ugyanis ha nem a projekt gyökérben vagyunk, akkor a szülő mappa linkelt Diagramjában lévő PackageObject-et megkeressük és annak az elemei közé beszúrjuk a friss projektmappa nevét, egy kis csomag ikonnal.



21. ábra: A nagyszülő mappa(~) diagrammjában lévő szülő mappát(mypaxkage) 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.

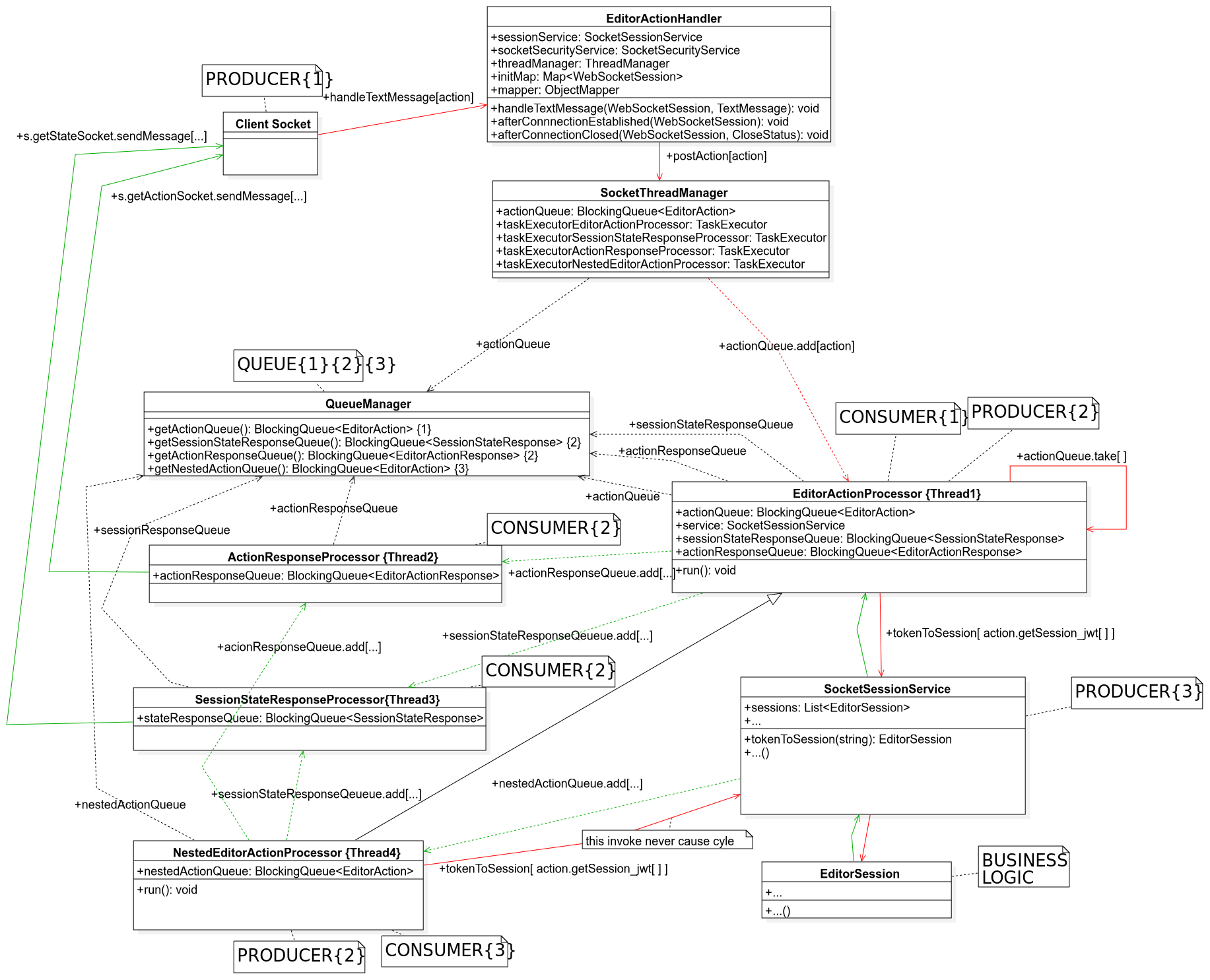
#### Editor

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.

##### WebSocketConfig

Itt a socketek konfigurációja történik a registerWebSocketHandlers függvényben. Felhívnám 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[[9]](#endnote-9) olvashat.

##### Az Editor réteg működése



22. 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[[10]](#endnote-10) ObjectMapper-je, majd a ThreadManager-nek passzolja 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}-ból 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 szálakon futó ActionResponseProcessor{C2} és SessionStateResponseProcessor{C2}. Ez a két osztály az EditorActionResponse vagy a SessionStateResponse szabályai alapján továbbítják a választ a megfelelő socket végponok 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 válaszobjektumot valamelyik {Q2}-es szállítóba. A többi már csak történelem.

###### SocketThreadManager és QueueManager

Ezek az osztályok a producer-consumer arhitektúra megeremtéséért felelnek. 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).

###### EditorActionHandler és SessionStateHandler

Mindkét osztály kiterjeszti a TextWebSocketHandler-t.

**A legfontosabb felülírandó függvények:**

* afterConnectionEstablished(WebSocketSession)
* handleTextMessage(WebSocketSession,TextMessage)
* afterConnectionClosed(WebSocketSession,CloseStatus)

Az elnevezések magukért beszélnek. 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.

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 token-t ad vissza (benne van a session azonosítója is), amelyet a kliens megkap, mint kulcsot a /socket végponton figyelő SessionStateHandler-hez.

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 interaktálhatunk a szerkesztőfelületen. Ezen a Map-en végigmegyünk, listába fűzzük és válaszként elküldjük a kliensnek. Amikor a kliens megkapja a listát és kiolvasta, 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 felfüz 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.

## Tesztelés

A fejlesztés ideje alatt legfőképp manuális teszeléssel ellenőriztem a program helyes működését. Ennek részeként a szerver végpontok teszteléséhez a Postman[[11]](#endnote-11) nevű programot előszeretettel használtam. Amikor már a kliensoldal megbízhatóan működött, annak segítségével, a felhasználó szemszögéből végeztem a tesztelést. Ha refaktorálás történt valahol, akkor arra különös tekintettel voltam.

A szerver oldalon tesztesetekhez egy külön konfigurációs fájlt használok (src/main/resources/test.properties). Az alapértelmezett, application.properties konfigurációs fájllal ellentétben, ebben van megfogalmazva, hogy a spring boot egy M2-es adatbázist inicializáljon. Ez egy Virtuális, memóriában tárolt adatbázis, amely az alapvető adatbázis műveletekkel rendelkezik. *„Egy gyors prototípus megépítéséhez tökéletes”.* Ez azért nagyszerű, mert teljesen izolált adatbázisban futtathatjuk a teszteseteinket, így a PostgreSQL szerveren tárolt meglévő adataink véletlenül sem fognak megsérülni.

### Unit tesztelés

A programnak kevés olyan része van, amely izoláltan végez el feladatot. Ez annak a követezménye, hogy a programban a rengeteg az azonnali mentés, frissítés, törlés, ezért adatok a szerver és az adatbázis között állandóan konzisztensek. Ez megnehezíti az egységtesztelést, mivel rengeteg függőséget kell mockolni. A szerkesztő felület **fő logikája** az **EditorSession** osztályban van. Itt a zárolási mechanizmust alaposan le tudtam tesztelni unit tesztek segítségével. Néhány példa erejéig a Mockito[[12]](#endnote-12) keretrendszert használtam a mockoláshoz. Ennek a bonyolultságának szemléltetésére egy jó példa a deleteSimpleClass\_Test() tesztmetódus, amelyben 35 sornyi előkészület után SimpleClass típusú doboz törlését tesztelem. Az előkészület azért ilyen hosszadalmas, mert a létrehozás során ezeket a hivatkozásokat több modul együttesen állítja össze.

### Integrációs tesztelés

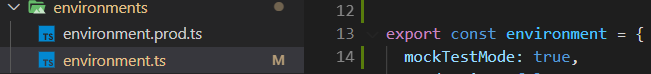
#### Szerver oldal

A bonyolultabb, sok adatbázis műveletet igénylő metódusok tesztelését, illetve a menedzsment modul helyes működését inkább az integrációs tesztmódszerrel ellenőriztem. A ManagementController és ProjectManagementController teszteléséhez a MockMvc tesztkönyvtárt használtam. A MockMvc segítségével könnyedén küldhetünk rest kéréseket tetszőleges végpontokra. A tesztelést nagyban megnehezítette az, hogy az adatbázissal kommunkáló 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 művelet](#_Szójegyzék_1)et végez, akkor az elvégzett módosításokat közvetlenül, ugyanabban a tesztmetódusban nem lehet biztonságosan ellenőrizni.

A szerver oldal tesztadatokkal történő manuális teszteléséhez létrehoztam egy **dev** nevű profilt; Ha szervert „dev” (fejlesztői) módban szeretnénk indítani tiszta, memóriában futó adatbázissal, illetve 2 teszt felhasználóval, akkor az 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, ezúttal 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 Socket-es funkcionalitásának Seleniumos teszteléséhez előfeltétel, hogy a szerver dev módban fusson.

#### Kliens oldal

Az alább felsorolt teszteseteket a Selenium[[13]](#endnote-13) keretrendszer segítségével automatizáltam. A Selenium egy nyílt forráskódú tesztelőkönyvtár, amely a böngésző tesztelő illesztőprogramjának segítségével interaktál a weboldallal. Szinte bármilyen információ elkérhető a html oldalról. A Kliens mock üzemmódban is tesztelhető, ennek eléréséhez állítsuk az environment.ts fájlban az environment konstans mockTestMode értékét true-ra:



Ha ezt megtettük, akkor az src/app/testing mappában található HttpInterceptor osztályok működésbe lépnek, leárnyékolva a kimenő kéréseket és beégetett (mockolt) válaszokat küldenek vissza. Csak a valódi REST végpontok vannak lemockolva, a socketes végpontok nem. Nyissuk meg a Selenium-ClientTest projektet a kedvenc fejlesztői környezetünkben (én a NetBeans 12.0-t használtam), majd futtassuk le a UnitTestSuite tesztsorozatot. Ez a fájl meg fog hívni minden olyan tesztmetódust, amely a Csak a kliens oldal működését teszteli.

Ha Teljes integrációs tesztet szeretnénk csinálni:

1. A szervert **dev** profillal indítsuk el, az [előző fejezet](#_Szerver_oldal) utasításai alapján
2. A kliens oldalon environment.ts konfigurációs fájlban, a mockTestMode értékét állítsuk vissza false-ra! Ellenkező esetben hiba fog fellépni!

Ha a fenti utasításokat elvégeztük, akkor futtassuk le a FullIntegrationTestSuite tesztsorozatot! Ez a tesztsorozat az összes tesztet lefuttatja, kivétel nélkül, a teljes funkcionalitást ellenőrizve.

#### Tesztesetek

A \* -al jelölt eredmények csak integrációs tesztelés esetén teljesülnek.

##### 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”***

##### 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”***

##### 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! 
  + kattinsunk az eszköztárban a mappa létrehozása ikonra! 
  + A mappa lista végén megjelenő beviteli mezőbe írjunk be azt, hogy „teszt”!
  + Kattintsunk valahová, vagy nyomjunk Enter-t!
* Várt eredmény: Új sor jelenik meg a fájl lista végén mappa ikonnal. 

##### 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! 
  + kattinsunk az eszköztárban a projekt létrehozása ikonra! 
  + A mappa lista végén megjelenő beviteli mezőbe írjunk be azt, hogy „tesztprojekt”!
  + Kattintsunk valahová, vagy nyomjunk Enter-t!
* Várt eredmény: Új sor jelenik meg a fájl lista végén projekt ikonnal. 

##### 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.

##### Projektmappa létrehozása

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

##### Beágyazott projektmappa létrehozása

* Leírás: Egy projekten belül egy beágyazott projektmappát fogunk létrehozni.
* Előfeltétel: A fájl menedzserben egy projektbe vagyunk lépve, a projekten belül van egy „teszt” projektmappánk, illetve az **Editor** funkció meg van nyitva.
* Tesztlépések:
  + Dupla kattintással lépjünk be a teszt mappába
  + Kövessük az **Új mappa létrehozása**  teszteset lépéseit!
  + Kattintsunk a **back** linkre a menedzser felületen
  + kattintsunk a kis  gombra a „teszt” mappa mellett
* Várt eredmény:
  + A menedzser felületen egyszerre látható a teszt, illetve a beágyazott mappa némi baloldali behúzással.
  + A szerkesztőfelületen a teszt dobozban megjelenik a beágyazott mappa neve. \*

##### Osztály létrehozása beágyazott projektmappában

érdekesség: a felső szinten az osztály megjelenik

##### Attribútum vagy metódus hozzáadása osztálydefiníció dobozhoz

##### Osztály nevének módosítása

##### Jegyzetdoboz létrehozása

##### Két doboz összekötése

##### Doboz törlése

##### Vonal törlése

##### Olyan doboz törlése, amelyhez vonal tartozik

#### Manuális tesztelés

Ha manuálisan szeretnénk tesztelni, akkor 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 szerkesztői felületet.

# Szójegyzék

**dml művelet:** data modifying művelet. Ilyen műveletnek számít a létrehozás(create), törlés(delete) és a módosítás(update)

# Továbbfejlesztési lehetőségek

## File Manager

* Ha egy 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, lletve 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 meszűntetése gomb
* Egy-egy fájl részleteinek megtekintése: Ki hozta létre, mikor, kikkel let megosztva, ami csak akkor látszik, ha a tulajdonos nézi.

## Editor

**Session Ablak:**

* Megjeleníti az aktív résztvevőket, mindenkihez egy színt rendelve. Ha valaki egy dobozt, vonalat vagy attribútumot kiválaszt, akkor az a saját színének megfelelő keretet kap.

**Szerkesztő felület:**

* 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.

## Felhasználói fiókok

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

# Irodalomjegyzék

Angular dokumentáció <https://angular.io/docs>

1. <https://www.postgresql.org> [↑](#endnote-ref-1)
2. <https://www.oracle.com/java/technologies/javase-jre8-downloads.html> [↑](#endnote-ref-2)
3. <https://code.visualstudio.com> [↑](#endnote-ref-3)
4. <https://netbeans.apache.org/download/nb120/> [↑](#endnote-ref-4)
5. <https://www.pgadmin.org> [↑](#endnote-ref-5)
6. <https://www.flaticon.com> [↑](#endnote-ref-6)
7. <https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API> [↑](#endnote-ref-7)
8. <https://www.designmycodes.com/examples/spring-boot-jwt-authentication.html>   
    utolsó elérés dátuma: 2020.05.13 [↑](#endnote-ref-8)
9. <https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html> [↑](#endnote-ref-9)
10. <https://github.com/FasterXML/jackson-docs> [↑](#endnote-ref-10)
11. <https://www.postman.com> [↑](#endnote-ref-11)
12. <https://site.mockito.org> [↑](#endnote-ref-12)
13. <https://www.selenium.dev> [↑](#endnote-ref-13)